

© 2026 Gerasimos Gerogiannis

Coupling, Scaling, and Optimizing Accelerator-Centric Systems

by

Gerasimos Gerogiannis

Dissertation

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois Urbana-Champaign, 2026

Urbana, Illinois

Doctoral Committee:

Professor Josep Torrellas, Chair and Director of Research

Professor Deming Chen

Associate Professor Jian Huang

Assistant Professor Charith Mendis

Associate Professor Tushar Krishna, Georgia Institute of Technology

Dr. Stijn Eyerman, Intel Corporation

Abstract

With the end of Dennard scaling and the slowing of Moore’s Law, the performance of general-purpose processors has plateaued. At the same time, the demand for computing today is higher than ever. In response to this new reality, computing systems are transitioning toward highly heterogeneous accelerator-intensive platforms. As system designers prepare for this new era, the focus is largely on designing individual accelerator architectures. Unfortunately, much of the surrounding computing stack continues to rely on legacy processor-centric assumptions. This mismatch introduces new challenges when attempting to control, scale, and optimize accelerators in modern heterogeneous systems. To help unleash the potential of accelerators, this thesis aims to provide answers to the following broad, yet fundamental, question: *As computation increasingly moves to accelerators, how should the architecture of computing systems change?*

This thesis is conceptually organized into three parts. Each part revisits a different legacy processor-centric principle and replaces it with an accelerator-centric alternative, with the goal of making accelerators easier to control, scale, or optimize.

Accelerators and their CPU hosts often need to interleave their execution, but current decoupled accelerator interfaces can introduce significant control and data movement overheads. To eliminate these overheads, the first part of this thesis revisits the view of accelerators as decoupled devices and proposes unified *tightly-coupled* host-accelerator architectures. It first introduces *SPADE*, a tightly-coupled accelerator for sparse computations that is integrated into the same die as its CPU host, sharing the cache subsystem and address translation mechanisms. It then extends this idea to the important Large Language Model workloads, with *DECA*, a tightly-coupled accelerator for decompressing model weights during inference. The first part concludes with *ATX*, which generalizes tight coupling beyond individual accelerator designs by introducing the

Near-Core Accelerator abstraction and a set of architectural extensions that allow accelerators to operate close to the core’s architectural state.

When special-purpose accelerators are distributed across many nodes, their communication frequently continues to rely on processor-centric algorithms and networking hardware. With accelerated computation, bottlenecks induced by general-purpose communication infrastructures become more exposed, and can prevent accelerator-based systems from scaling. To make accelerators easier to scale, the second part of this thesis introduces techniques for *specializing communication* software and hardware. On the software side, it proposes *Two-Face*, a software algorithm for scheduling communication in distributed sparse computations by combining collective and one-sided accesses. On the hardware side, it presents *NetSparse*, a collection of networking hardware extensions catered to sparse communication that deliver an order of magnitude better performance for sparse accelerator clusters.

Finally, as accelerator-based systems become increasingly heterogeneous, manual, human-based performance engineering becomes increasingly unsustainable. To make accelerators easier to optimize, the third and final part of this thesis revisits this legacy optimization approach by proposing *automated and reusable* techniques for improving the performance of different heterogeneous architectures. First, it introduces the *Roof-Surface* performance model, a 3D extension of the popular Roofline model. The *Roof-Surface* model can be used to reason about and optimize the performance of ensembles with many heterogeneous accelerators at design time. Then, it develops *HotTiles*, a static optimization method for work partitioning in heterogeneous sparse accelerator architectures. The thesis concludes with *Bandit*, a lightweight and reusable hardware agent for runtime performance optimization of general microarchitectural problems.

*To my parents, for their love, support, and sacrifices,
and to my brothers, whom I love deeply.*

Acknowledgements

If I were to start my PhD journey all over again, the one thing I would most confidently leave unchanged is the people I got to share it with.

First and foremost, I would like to thank my advisor, Professor Josep Torrellas. Josep needs no introduction or praise. Yet, what has always amazed me most about him is that, despite his undeniably successful career, he remains extremely modest. Above all, Josep is motivated by his passion for working with his students. I will never forget that he stayed awake until 5 a.m. to help me improve my first ISCA submission before the deadline. Among his many talents, I especially want to highlight his unique ability to communicate research cleanly, clearly, and effectively. Josep, thank you for taking the time to look at my PhD application, even though it did not list you as a potential advisor and did not include “computer architecture” as a keyword. Thank you for everything you taught me and for the opportunity to work with you. This opportunity changed my life.

Besides my advisor, I would like to sincerely thank the rest of my PhD committee: Professor Deming Chen, Professor Jian Huang, Professor Tushar Krishna, Professor Charith Mendis, and Dr. Stijn Eyerman. The guidance, support, and perspectives of Professor Deming Chen and Professor Jian Huang were especially valuable for the completion of this thesis. I would also like to thank both of them for their wonderful courses I was fortunate to attend. It was in those courses that the ideas behind HotTiles and Bandit first began to take shape. I have always admired Professor Tushar Krishna’s work on deep learning accelerators, which served as one of the main inspirations for this thesis. I am very grateful for his support, encouragement, and guidance.

Professor Charith Mendis has been one of my most important collaborators since my very early days as a PhD student. I have always found Charith’s research to be exciting and refreshing, and I was very fortunate to work closely with him and his students Damitha and Chamika. I hope that we will continue tackling exciting problems together in the future.

Finally, I would like to thank Dr. Stijn Eyerman and, through him, also express my gratitude to Intel. As I jokingly say, I did a "mini PhD" with Intel during my internships, and I am very grateful that the company gave me the freedom to pursue research directions I found exciting. Both Stijn and Dr. Wim Heirman acted as second advisors for me during this mini PhD and I was extremely fortunate to learn from them over the last three years. Besides Stijn and Wim, I learned a lot from Dr. Evangelos Georganas and Dr. Sriram Ananthakrishnan. Finally, I would like to thank my first manager, Dr. Ibrahim Hur. His trust and continuous support mean a lot to me.

Beyond my committee, I would also like to thank Professor Ariful Azad and Professor Muhammad Shahbaz. I was fortunate to work closely with both Ariful and Shahbaz, as well as with their students Isuru and Annus. Ariful's expertise in high-performance computing and distributed algorithms, together with Shahbaz's expertise in networking and in-network computing, was crucial for the completion of the second part of my thesis.

The i-acoma group is filled with individuals who are not only talented researchers, but, most importantly, wonderful human beings. I would like to acknowledge my academic brothers and sisters with whom I was blessed to overlap, collaborate, and, in many cases, become friends: Apo, Serif, Andy, Neil, Nam, Antonis, Jovan, Dingyuan, Yao, Chase, Nikoleta, Jakob, Burak, Chloe, Abe, Shashwat, Dimitris Merkouriadis, Filippos, Victor, Kostas, Raul, Rishabh, and Pedro.

Serif was my mentor during my first year, and I would like to thank him for helping me kick off my research. Antonis, "The Leader," and I first arrived in Champaign together—on the same flight—five years ago. Since then, he has been a great friend and a constant pillar of support, and we have made so many wonderful memories together. I cannot find the words to express how grateful I am to Jovan for his support during my academic job search, for his friendship, and for standing by me through so many corners of this journey. Chase has extraordinary technical depth and talent, and he was my closest collaborator during my PhD. We spent countless hours brainstorming ideas, implementing systems, preparing

papers, and rebutting reviewers, and I could not have asked for a better teammate. Finally, Dimitris Merkouriadis and I became inseparable during the last two years, and I am sure we will stay friends for many years to come. I would also like to thank Dimitris Skarlatos, Raghav, and Ulya. Although we did not overlap, I was fortunate to meet them and receive great advice. Being an i-acomer surrounded by so many amazing individuals was a blessing.

Preparing to leave Urbana-Champaign brings back the exact feeling I had when leaving Patras, my hometown in Greece: the sense that I am leaving home once again.

Urbana-Champaign became my second home because of two communities I was fortunate to be part of.

Since my first day in Urbana-Champaign, I felt a sense of belonging largely due to the community of the UIUC Greek graduate students. Although I am surely forgetting some people (and without repeating the Greek i-acomers), I would like to thank Zoïs, Thymios, Thodoris, Olnancy, Anna Karanika, Tzortzis, Anna Papadopoulou, Argyris Michaloliakos, Eva, Simoni, Lila, Costas, Polyvios, and Efthymis. Special mention goes to my closest friends Dimitris Mandravelis, Marios, George, and Panagiotis. Our adventures in the Midwest were some of the best memories from my PhD. To the newer members of the community, I hope you will do your best to keep it alive.

The second group of people I would like to acknowledge is the Urbana-Champaign Latin Social Dancing community, together with the Dancing Illini and the Urbana Dance Company. Through social dancing, I was able to find joy even during the most difficult times. Christian, Elaine, Salil, and Spencer, thank you for teaching me how to dance. Although dancing brought many more wonderful individuals into my life than I am able to list here, I was very fortunate to share dances and memories with Mohul, Pratham, Kim, Sameeksha, Jose, Sandry, Yiran, Vincent, Lucia, Sofiia, and Sylvia. Hannah, I was very lucky to meet you. Thank you for bringing light to some of my darkest moments.

I would also like to thank the other friends I made in Urbana-Champaign: Siddharth, Nikunj, Rama, and Liset. And, in case I have forgotten anyone—most likely

unintentionally—thank you for the moments, the laughs, and the lessons.

My closest friends back in Greece made the 5,300-mile distance that separates us feel much smaller. Aimilie, Mitso, Foti, Lio, Ntino, Tony, Petro, and Giorgio, thank you for being by my side for over 15 years. Andreas has acted as an older brother for me, and his advice shaped many of my important life decisions. Andrea, from the bottom of my heart, thank you for everything. Through Andreas, I would also like to acknowledge all of my teachers and mentors who influenced me over the years.

The most important people in my life are my brothers, Argyris and Dimitris. The greatest blessing of my PhD in the American Midwest was that I got to share those years with Argyris as my roommate. At the same time, Dimitris was doing his PhD in London, and visiting him remains one of my sweetest memories from these years. With Argyris and Dimitris by my side, I feel fearless. I deeply hope that life will bring the three of us geographically close once again. I would also like to thank my grandma Nana and my uncle Xaris for everything they have done for me and my brothers. Finally, who I am today, and whatever I have managed to achieve, I owe to the love and support of my parents, Reggina and Giannis. The sacrifices they have made, and continue to make, for me and my brothers are beyond words. The least I can do is to dedicate this thesis to them.

Contents

Chapter 1	Thesis Overview	13
1.1	Introduction	13
1.2	Problems	14
1.3	Contributions	16
Chapter 2	General Background on Sparse Computations	24
2.1	Sparse Matrices	24
2.2	Sparse Computations	26
2.3	Distributed Sparse Computations	29
Chapter 3	Tightly-Coupled Accelerator Architecture for Sparse Computations . .	33
3.1	Introduction	33
3.2	Motivation	36
3.3	Overview of the SPADE System	38
3.4	Key Architecture Components	47
3.5	Methodology	52
3.6	Evaluation	56
3.7	Related work	66
3.8	Conclusion	69
3.9	Chapter Supplement: A Tiled Sparse Matrix	69
Chapter 4	Tightly-Coupled Accelerator Architecture and Performance Model for Compressed Large Language Models	71
4.1	Introduction	71
4.2	Background	74
4.3	Motivation	78
4.4	The Roof-Surface Model	81
4.5	DECA Overview and TEPL Instructions	86
4.6	DECA Microarchitecture	92
4.7	Methodology	97
4.8	Evaluation	100
4.9	Discussion	108
4.10	Related work	110
4.11	Conclusion	112

4.12	Chapter Supplement: Effect of Batch Size	112
Chapter 5	Near-Core Accelerator Abstraction and Extensions	115
5.1	Introduction	115
5.2	The case for Near-Core Accelerators	118
5.3	ATX Overview and Instructions	122
5.4	The Unified Transfer Engine (UTE)	128
5.5	Methodology	141
5.6	Evaluation	144
5.7	Limitations and Discussion	151
5.8	Related Work	152
5.9	Conclusion	153
Chapter 6	Software Acceleration of Distributed Sparse Computations	154
6.1	Introduction	154
6.2	Motivation	156
6.3	Overview of Two-Face	159
6.4	The Two-Face Algorithm	165
6.5	Methodology	172
6.6	Evaluation	177
6.7	Related Work	184
6.8	Conclusion	186
Chapter 7	Hardware Acceleration of Distributed Sparse Computations	187
7.1	Introduction	187
7.2	Background	189
7.3	Motivation	192
7.4	NetSparse Architecture	195
7.5	NIC-centric NetSparse Mechanisms	198
7.6	Network-centric NetSparse Mechanisms	203
7.7	Other Considerations	211
7.8	Methodology	213
7.9	Evaluation	217
7.10	Related work	229
7.11	Conclusion	229
Chapter 8	Heuristic Optimization of Hardware-Accelerated Sparse Computations	231

8.1	Introduction	231
8.2	Background	233
8.3	Motivation	235
8.4	Intra Matrix Heterogeneity (IMH) Aware Performance Modeling	242
8.5	Intra Matrix Heterogeneity (IMH) Aware Partitioning	248
8.6	Architectures and Framework	253
8.7	Methodology	256
8.8	Evaluation	259
8.9	Related Work	270
8.10	Further Directions	271
8.11	Conclusion	272
Chapter 9	Hardware Agent for General Data-Driven Runtime Optimization	273
9.1	Introduction	273
9.2	Background	275
9.3	Motivation	279
9.4	Multi-Armed Bandit Algorithms for Microarchitecture	285
9.5	Micro-Armed Bandit Design	290
9.6	Evaluation Methodology	294
9.7	Evaluation	299
9.8	Related Work	309
9.9	Further Directions	311
9.10	Conclusion	312
9.11	Chapter Supplement: Guidelines to Use Bandit for Other Problems	312
Chapter 10	Future Work	317
Chapter 11	Conclusions	320
References		322
Appendix:	Other Works	389

Chapter 1 Thesis Overview

1.1 Introduction

For much of computing history, advances in process technology were the primary drivers of improvements in the performance and energy efficiency of computing systems. Dennard scaling [1] and Moore’s Law [2] collectively suggested that with time, transistors were becoming smaller, faster, and more energy efficient. Transistor scaling, in turn, enabled the continued development of increasingly powerful general-purpose processors.

Today, computing is undergoing a paradigm shift. Dennard scaling has ended and Moore’s Law has severely slowed down. This slowdown comes at a time when the demand for computation is higher than ever. A recent report from Lawrence Berkeley National Laboratory [3] projects that datacenters will account for up to 12% of the total electricity consumption in the United States. Back in 2018, the corresponding contribution was only about 2%. The cost of computing is following an exponential trajectory, and today we need drastically more efficient computing systems.

In response to this new reality, computing is increasingly shifting away from general-purpose processors and toward a highly heterogeneous ecosystem centered around hardware accelerators [4], [5]. Accelerators are special-purpose computing architectures optimized for a particular kernel, application, or application domain. By specializing their datapaths, control paths, and memory, accelerators offer higher performance and energy efficiency than general-purpose processors for the particular domains they target. Almost every major company in the computing industry has introduced custom commercial accelerator architectures. Notable examples for machine learning include NVIDIA Tensor Core GPUs [6], [7], [8], Google TPUs [9], [10], Amazon Trainium [11], Meta’s Training and Inference Accelerator (MTIA) [12], Graphcore’s Intelligence Processing Unit (IPU) [13], Cerebras’ Wafer-Scale Engine (WSE) [14], and SambaNova’s Reconfigurable Dataflow Unit (RDU) [15]. Other examples for cloud and datacenter workloads include Intel DSA, IAA,

QAT, and DLB [16], [17], [18].

As computing systems are transitioning to this new era, the focus has largely been on designing individual accelerator architectures. This thesis argues that, despite the significant advances in individual accelerators, much of the surrounding computing stack continues to rely on legacy processor-centric assumptions. This mismatch introduces novel and interconnected challenges when attempting to control, scale, and optimize accelerators in modern heterogeneous systems.

This thesis aims to address these challenges by rethinking different aspects of the design of modern computing systems from an accelerator-centric perspective.

1.2 Problems

This section describes the challenges associated with controlling, scaling, and optimizing accelerators and why they are particularly relevant in modern accelerator-intensive systems.

1.2.1 Control and Data Movement Overheads

Accelerators are very effective when computations map well to their specialized hardware. In practice, however, many applications require some computations not supported by specialized hardware, or demand more memory than available in specialized architectures. In such scenarios, programs running on GPUs or accelerators need to fall back to CPU hosts for assistance [19], [20], [21], [22], [23], [24]. An important example application is the emerging agentic AI [25], [26]. Unfortunately, interleaving CPU and accelerator phases often carries substantial control and data movement overheads that decrease the end-to-end efficiency [24], [27]. One would like an architecture that combines domain-specific levels of performance with the programmability and flexibility of CPUs without high coordination or communication costs.

1.2.2 Scaling Bottlenecks

Accelerators today are deployed at unprecedented scales. Industry reports describe machine-learning oriented clusters and fleets ranging from thousands to over one hundred thousand interconnected accelerators [10], [28], [29]. At the same time, planned investments in AI infrastructure suggest that this scale will keep increasing [30], [31], [32], [33]. In this emerging landscape, efficient inter-accelerator communication mechanisms are critical.

However, when special-purpose accelerators are distributed across many nodes, their communication frequently continues to rely on processor-centric algorithms and networking hardware. As we show later, with accelerated computation, bottlenecks induced by general-purpose communication infrastructures become more exposed, and can prevent accelerator-based systems from scaling. Addressing these bottlenecks requires rethinking both communication software and networking hardware around the demands of accelerator-intensive systems.

1.2.3 Performance Optimization Complexity

Even today, optimizing the performance of a specific software kernel or hardware component often requires substantial effort from large teams of engineers. This manual optimization model is rapidly becoming unsustainable. Beyond their increasing scale, emerging accelerator-based computing systems are also characterized by increasing architectural diversity. For example, the Epoch AI Machine Learning Hardware dataset tracks more than 170 commercially deployed AI accelerators [34]. As new heterogeneous accelerator architectures emerge at unprecedented rates, they increasingly outpace the available performance engineering capacity. It is evident that keeping up with the optimization complexity of modern heterogeneous architectures requires rethinking legacy manual optimization models.

1.3 Contributions

To help unlock the full potential of accelerator-based computing, this thesis rethinks different aspects of the design of modern computing systems from an accelerator-centric perspective. The thesis is conceptually organized into three parts. Each part revisits a different legacy processor-centric principle and replaces it with an accelerator-centric alternative with the goal of making accelerators easier to control, scale, or optimize.

First, the thesis shows how the data movement and control overheads caused by the traditional decoupled view of accelerators can be reduced through tightly-coupled host-accelerator architectures. Second, it extends specialization beyond computation to distributed communication software and hardware in large-scale multi-accelerator systems. Third, it revisits the manual, human-driven optimization process by introducing automated and reusable techniques for improving the performance of different heterogeneous architectures.

This section further details the specific contributions made by this thesis.

1.3.1 Processor-Accelerator Coupling for Reducing Coordination Overheads

The first part of this thesis revisits the view of accelerators as decoupled devices by proposing tightly-coupled host-accelerator architectures that reduce control and data movement overheads. We start by presenting two accelerator architectures that are tightly coupled on the same die with their CPU hosts and share the cache subsystem and address translation mechanisms: *SPADE*, which targets sparse computations, and *DECA*, which targets the decompression of Large Language Model (LLM) weights during inference. The first part concludes with *ATX*, which generalizes tight coupling beyond *SPADE* and *DECA* by introducing the Near-Core Accelerator abstraction and a set of architectural extensions that allow accelerators to operate close to the core’s architectural state.

- **SPADE [35] (Chapter 3):** The widespread use of Sparse Matrix Dense Matrix Multiplication (SpMM) and Sampled Dense Matrix Dense Matrix Multiplication

(SDDMM) kernels makes them candidates for hardware acceleration. However, accelerator design for these kernels faces two main challenges: (1) the overhead of moving data between CPU and accelerator (often including an address space conversion from the CPU’s virtual addresses) and (2) marginal flexibility to leverage the fact that different sparse input matrices benefit from different variations of the SpMM and SDDMM algorithms.

To address these challenges, we propose *SPADE*, a new SpMM and SDDMM hardware accelerator. SPADE avoids data transfers by tightly coupling accelerator processing elements (PEs) with the cores of a multicore processor, allowing the accelerator to reuse the CPU memory system and its virtual addresses. SPADE attains flexibility and programmability by supporting a *tile-based* ISA—high level enough to eliminate the overhead of fetching and decoding fine-grained instructions. To prove the SPADE concept, we have taped-out a simplified SPADE chip. Further, simulations of a SPADE system with 224–1792 PEs show its high performance. A 224-PE SPADE system is on average $2.3\times$, $1.3\times$, and $2.5\times$ faster than a 56-core CPU, a server-class GPU, and an SpMM accelerator, respectively, without accounting for the host-accelerator data transfer overhead. If such overhead is taken into account, the 224-PE SPADE system is on average $43.4\times$ and $52.4\times$ faster than the GPU and the accelerator, respectively. Further, SPADE has a small area and power footprint.

Impact: SPADE was recognized with an IEEE Micro Top Picks Honorable Mention for contributing a novel approach to integrate accelerators with the cache hierarchy of CPU cores and demonstrating it for mixed sparse-dense algebra. Further, it attracted industry attention from IBM.

- **DECA [36] (Chapter 4):** To alleviate the memory bandwidth bottleneck in Large Language Model (LLM) inference workloads, weight matrices are stored in memory in quantized and sparsified formats. Hence, before tiles of these matrices can be processed by in-core generalized matrix multiplication (GeMM) hardware engines, they need to be dequantized and de-sparsified. This is currently performed in software with vector

operations. Unfortunately, this approach delivers only modest performance.

To address software inefficiencies in such environments, we propose *DECA*, a new ML-model decompression accelerator that is tightly-coupled with its CPU host. DECA offloads tile de-sparsification and dequantization from the CPU, producing ready-to-use tiles for in-core GeMM engines. Further, expanding on the SPADE integration scheme, we introduce an ISA extension that enables out-of-order invocation of DECA. With this extension, accelerator and core computations can interleave and overlap with high performance. Our evaluation shows that, in a simulated 56-core Xeon 4 server with HBM, DECA accelerates the execution of compressed GeMMs by up to 4x over the use of optimized Intel software kernels. Further, DECA reduces the next-token generation time of Llama-2-70B and OPT-66B by $1.6\times$ – $1.9\times$.

Impact: DECA was developed in collaboration with Intel, and led to the filing of one US patent with the company.

- **ATX [37] (Chapter 5):** Integrating accelerators like SPADE and DECA in CPU multicores combines the benefits of accelerated computation with the flexibility and programmability of CPUs. CPU-integrated accelerators can be classified into *In-Core Accelerators (ICAs)*, which typically reside inside the core’s pipeline, and *Out-of-Core Accelerators (OCAs)*, which are typically attached at the core’s cache subsystem. Both designs have shortcomings: ICAs can be bottlenecked by the core’s general-purpose memory access interface, while OCAs’ core-accelerator interface can limit execution overlap and expose communication overheads.

In Chapter 5, we introduce *Near-Core Accelerators (NCAs)*, a new accelerator abstraction that aims to combine the advantages of ICAs and OCAs, and address their shortcomings. Specifically, NCAs have their own read interface to the memory system, eliminating ICAs’ bottleneck. At the same time, NCAs can be invoked speculatively and out-of-order, enabling both core-accelerator execution overlap and low-overhead core-accelerator communication. We also propose the *Accelerator Task Extensions (ATX)*,

a set of instructions and hardware extensions to support NCAs. With ATX instructions, CPU cores can speculatively invoke a diverse range of NCAs. ATX includes the *Unified Transfer Engine (UTE)*, a programmable hardware module that efficiently supplies data to NCAs and virtualizes the interface between the CPU core and the NCAs. The UTE interfaces the CPU core with multiple NCAs and the cache subsystem, fetching and prefetching accelerator data, and scheduling tasks to potentially multiple NCAs transparently to the CPU core. We evaluate ATX NCAs with a variety of important kernels from machine learning and scientific computing. We show that ATX NCAs accelerate these kernels by 1.3–18 $\times$ over various CPU-integrated accelerator alternatives. ***Impact:*** ATX was developed in collaboration with Intel, attracted significant attention within the company, and led to the filing of three US patents.

1.3.2 Communication Specialization for Scaling Accelerators

To solve scaling bottlenecks in distributed accelerator-based machines, the second part of this thesis extends specialization beyond computation, by proposing specialized communication hardware and software. On the software side, it contributes *Two-Face*, a communication algorithm that combines coarse-grained collectives and fine-grained one-sided accesses to accelerate distributed sparse communication. On the hardware side, it introduces *NetSparse*, which extends modern networking devices with hardware extensions catered to the domain of sparsity.

- **Two-Face [38] (Chapter 6):** Sparse Matrix Dense Matrix Multiplication (SpMM) is commonly used in applications ranging from scientific computing to graph neural networks. Typically, when SpMM is executed in a distributed platform, communication costs dominate. Such costs depend on how communication is scheduled. If it is scheduled in a sparsity-unaware manner, such as with collectives, execution is often inefficient due to unnecessary data transfers. On the other hand, if communication is scheduled in a fine-grained sparsity-aware manner, communicating only the necessary data, execution can

also be inefficient due to high software overhead.

We observe that individual sparse matrices often contain regions that are denser and regions that are sparser. Based on this domain-specific insight, we develop a new specialized algorithm that uses a preprocessing model to perform collective communication for the denser regions, and fine-grained, one-sided communication for the sparser regions. We call the algorithm *Two-Face*. We show that *Two-Face* attains an average speedup of $2.11\times$ over prior work when evaluated on a 4096-core supercomputer. Additionally, *Two-Face* scales well with the machine size.

• **NetSparse [39] (Chapter 7):** Many hardware accelerators have been proposed to accelerate sparse computations. When these accelerators are placed in the nodes of a large cluster, distributed sparse applications become heavily communication-bound. Unfortunately, despite the useful insights provided by *Two-Face*, software-only communication optimizations remain insufficient for scaling accelerator-based sparse computations.

In Chapter 7, we introduce novel hardware mechanisms to optimize network communication in distributed sparse computations. Our proposal, called *NetSparse*, consists of four mechanisms. Communication is offloaded to new processing units in the NIC that support efficient *remote indexed gather* operations, minimizing host-NIC communication. Moreover, these units have the ability to identify and eliminate redundant requests, which minimizes traffic. Further, new hardware modules in the NICs and switches concatenate multiple requests with the same destination node in a single packet, saving traffic and header overheads. Finally, switches are augmented with a hardware cache that stores fetched data from remote racks, making it available to all the nodes in the local rack on demand. Our evaluation on a simulated 128-node cluster with a SPADE sparse accelerator in every node running sparse workloads reveals that NetSparse improves performance substantially. When the cluster uses traditional software-based communication, the workloads run only $3\times$ faster than on a single-node system; when it is

augmented with the NetSparse hardware, the workloads run $38\times$ faster than on the single-node system—attaining more than half of the performance of an ideal system that has no communication overheads.

1.3.3 Automated and Reusable Techniques for Performance Optimization

The final part of this thesis contributes automated techniques that can be reused for the performance optimization of different heterogeneous architectures. First, it introduces a 3D extension of the Roofline model, called the *Roof-Surface*, that can be used to reason about the performance bottlenecks in heterogeneous accelerator ensembles at design time. It then proposes *HotTiles*, a static work-partitioning method for optimizing the performance of heterogeneous sparse accelerators. Finally, it contributes *Bandit*, a lightweight and reusable hardware agent for runtime performance optimization of general microarchitectural problems.

- **Roof-Surface [36] (Chapter 4):** The Roofline model is integral to modern computer architecture. It captures the optimal performance of a system limited by either its peak computational throughput or its memory bandwidth. However, in accelerator-intensive systems, an application may involve multiple heterogeneous compute domains that interact with one another. In the compressed LLM inference setting described in Chapter 4, the overall GeMM performance depends on the memory bandwidth and on the interaction between vector engines and matrix engines—two heterogeneous compute domains with separate instructions and functional units. Unfortunately, the traditional 2D Roofline model cannot accurately capture optimal performance in settings where multiple compute domains cooperatively participate in the computation.

To fill this gap, we propose the *Roof-Surface*: a novel 3D performance model that captures how memory resources, vector units, and hardware matrix engines interact to deliver compressed GeMM performance. We show how this model can guide accelerator design both qualitatively and quantitatively, by using it to drive the design of our DECA

accelerator. We then explain how the Roof-Surface can be generalized beyond vector and matrix engines to model systems with any number of abstract cooperative compute domains and memory resources. Given the broad influence of the traditional Roofline model, we believe that the Roof-Surface can help guide the design and optimization of future accelerator-centric systems.

- **HotTiles [40] (Chapter 8):** As discussed earlier in this section, in many sparse matrices, the pattern of nonzeros is nonuniform: nonzeros form dense and sparse regions, rather than being uniformly distributed across the whole matrix. We refer to this property as *Intra-Matrix Heterogeneity (IMH)*. Currently, SpMM accelerator designs do not leverage this heterogeneity. They employ the same processing elements (PEs) for all the regions of a sparse matrix, resulting in suboptimal acceleration.

To address this limitation, we utilize *heterogeneous* SpMM accelerator architectures, which include different types of PEs to exploit IMH. We develop an analytical modeling framework to predict the performance of different types of accelerator PEs taking into account IMH. Furthermore, we present a heuristic for partitioning sparse matrices among heterogeneous PEs. We call our matrix modeling and partitioning method *HotTiles*. To evaluate *HotTiles*, we simulate three different heterogeneous architectures. Each one consists of two types of workers (i.e., PEs): one suited for compute-bound denser regions (*Hot Worker*) and one for memory-bound sparser regions (*Cold Worker*). Our results show *HotTiles* can substantially optimize the performance of heterogeneous sparse accelerators. Depending on the architecture, heterogeneous execution with *HotTiles* outperforms homogeneous execution using only hot or only cold workers by $9.2\text{-}16.8\times$ and $1.4\text{-}3.7\times$, respectively. In addition, *HotTiles* outperforms the best worker type used on a per-matrix basis by $1.3\text{-}2.5\times$. Finally, *HotTiles* outperforms an IMH-unaware heterogeneous execution strategy by $1.4\text{-}2.2\times$.

- **Bandit [41], [42] (Chapter 9):** Online Reinforcement Learning (RL) has been adopted as an effective mechanism in various decision-making problems in

microarchitecture. Its high adaptability and the ability to learn at runtime are attractive characteristics in microarchitecture settings. However, although hardware RL agents are effective, they suffer from two main problems. First, they have high complexity and storage overhead. This complexity stems from decomposing the environment into a large number of states and then, for each of these states, bookkeeping many action values. Second, many RL agents are engineered for a specific application and are not reusable.

In Chapter 9, we tackle both of these shortcomings by designing an RL agent that is both lightweight and reusable across different microarchitecture decision-making problems. We find that, in some of these problems, only a small fraction of the action space is useful in a given time window. We refer to this property as *temporal homogeneity in the action space*. Motivated by this property, we design a hardware RL agent based on Multi-Armed Bandit algorithms, the simplest form of RL. We call our agent *Micro-Armed Bandit* (or *Bandit* for short).

We showcase our agent in two diverse use cases: data prefetching and instruction fetch in simultaneous multithreaded (SMT) processors. For prefetching, our agent attains similar performance as the state-of-the-art RL prefetcher Pythia—with the dramatically lower storage requirement of only 100 bytes. For SMT instruction fetch, our agent outperforms the Hill Climbing method by 2.2% (geometric mean). In the context of this thesis, Bandit contributes to the increasingly important goal of reducing manual optimization effort in heterogeneous architectures by showing that lightweight, reusable online-learning agents can automate runtime microarchitectural decision-making across diverse use cases.

Impact: Bandit was recognized with an IEEE Micro Top Pick, for making reinforcement learning modules practical for online microarchitecture tuning. In addition, it spurred various follow-up research works, and received significant attention from Intel and AMD.

Chapter 2 General Background on Sparse Computations

In this section, we provide background on sparse computations, which serve as a central workload across several chapters of this thesis: Chapter 3 presents a tightly-coupled accelerator design for sparse computations. Chapter 6 and Chapter 7 investigate specialized software and hardware techniques for scaling sparse computations across the nodes of large clusters. Finally, Chapter 8 discusses how heterogeneous sparse accelerators can be optimized.¹ As sparse computations have received significantly less attention than their dense counterparts, they provide a useful workload domain for illustrating several of the accelerator-centric design principles explored in this thesis.

2.1 Sparse Matrices

A matrix is sparse when it contains many zero elements (*zeros*), or, equivalently, few non-zero elements (*nonzeros*). Although there is no formal threshold for how few those nonzeros should be, in general, their number should be sufficiently small to justify the use of compressed representations to reduce storage, computation, and/or data movement.

Figure 1(a) shows an example of a matrix with many zeros in a dense representation. In this representation all elements of the matrix are stored explicitly in memory, regardless of whether they are zeros or nonzeros. Figures 1(b), 1(c), and 1(d) show the same matrix stored in compressed representations with different popular compression formats. The compressed representations of the matrix avoid storing zeros. Instead, only nonzeros are stored in a *vals* array. The coordinates of those nonzeros in the original matrix can be recovered with the help of additional metadata. Different sparse matrix compression formats differ in the type of metadata used.

For example, the popular Coordinate (COO) format (Figure 1(b)) utilizes two metadata arrays for directly storing the coordinates of each nonzero in the original matrix.

¹Sparsity is also discussed in Chapter 4, and sparse workloads are also used alongside other applications as benchmarks in Chapter 5 and Chapter 9.

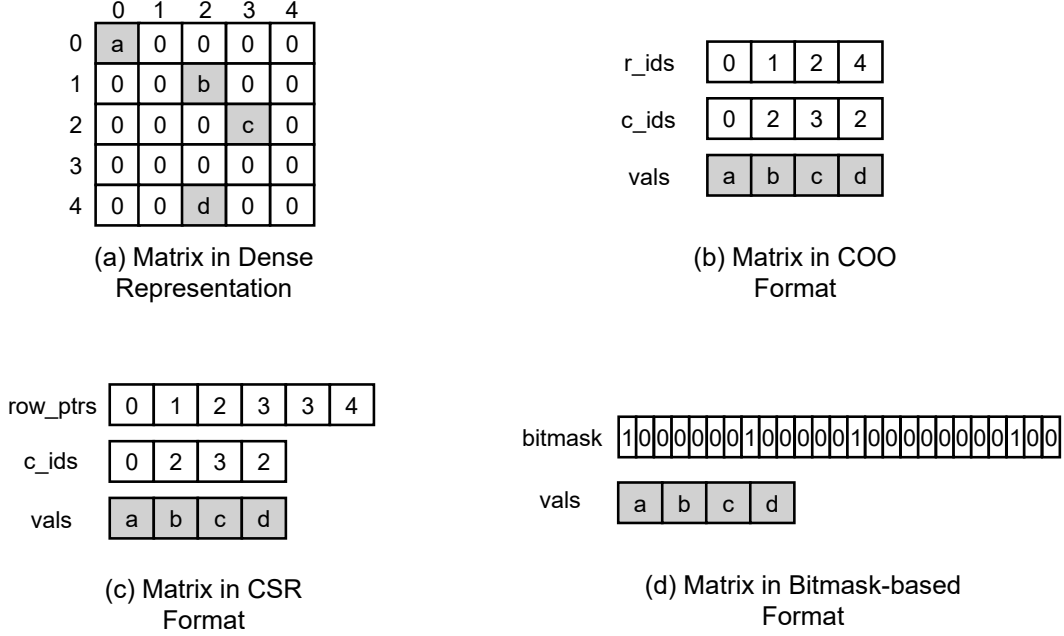


Figure 1: An example matrix in dense representation and in different sparse compression formats.

The `r_ids` array stores the row indices, while the `c_ids` array stores the column indices.

Another popular format is the Compressed Sparse Row (CSR) format shown in Figure 1(c). Similarly to the COO format, CSR also uses the `c_ids` array. However, the `r_ids` array is replaced by the `row_ptrs` array. Instead of storing row indices for each nonzero, the `row_ptrs` array stores pointers to the beginning and end of each row in the `vals` and `c_ids` arrays. Specifically, row i begins at index $row_ptrs[i]$ and ends right before index $row_ptrs[i + 1]$. Thus, the number of nonzeros in row i is given by $row_ptrs[i + 1] - row_ptrs[i]$. For example, in Figure 1(c), the `row_ptrs` array is [0, 1, 2, 3, 3, 4]: The first row (row 0) starts at index 0 of the `vals` array and ends before index 1; the second row starts at index 1 and ends before index 2; the third row starts at index 2 and ends before index 3; the fourth row starts at index 3 and ends before index 3 (thus, it is empty); and the fifth row starts at index 3 and ends before index 4. The final entry of `row_ptrs` is 4, indicating that the matrix contains four nonzeros in total. We refer to the total number of nonzeros in a sparse matrix as NNZ .

Finally, Figure 1(d) shows the same matrix in a bitmask-based compression format. This format has a single metadata array called *bitmask* which has a single bit for every element in the matrix. The bit is 1 if the corresponding element is a nonzero, and 0 if it is a zero. The bitmask achieves compression because zeros, which would otherwise be stored using a full data word (e.g., 32 bits), are instead represented by a single bit.

Overall, different compression formats may work best depending on the density level of the matrix, the structure of nonzeros in the sparse matrix (*nonzero pattern*), and the underlying operation or kernel [43]. Typically, bitmask-based representations can achieve higher storage efficiency for moderately sparse matrices, where storing one bit per matrix element can be cheaper than storing explicit index metadata for every nonzero. Such matrices can arise, for example, from pruning machine-learning models [44]. On the other hand, CSR- or COO-like representations are typically preferred for highly sparse matrices, such as those stemming from graph analytics problems [45], [46], because their metadata scales with the number of nonzeros rather than the total number of matrix elements.

Throughout this thesis, any matrix referred to as sparse is stored in a compressed format.

2.2 Sparse Computations

Sparse computations are computations involving at least one sparse data structure [47], [48], [49]. In this thesis, we mainly focus on two fundamental sparse kernels: Sparse Matrix Dense Matrix Multiplication (SpMM) and Sampled Dense Matrix Dense Matrix Multiplication (SDDMM). These kernels are widely used in domains such as machine learning [44], [50], [51], [52], parameter estimation [53], image segmentation [54], atmospheric modeling [55], [56], aerodynamic design [57], matrix factorization [58], and linear algebra solvers [59], [60], [61], [62]. SpMM and SDDMM are also key primitives for many popular Graph Neural Networks (GNNs) [52], [63], [64], [65], [66], [67], [68] and an integral part of graph learning frameworks such as the Deep Graph Library (DGL) [69]. In

GNNs, the message-passing kernels that aggregate information for a vertex from its neighbors can be implemented with SpMMs. Further, the message-passing kernels that aggregate information for an edge from its incident vertices can be implemented with SDDMMs [69], [70].

Figure 2 shows the operation of SpMM (top chart) and of SDDMM (bottom chart). SpMM takes an input sparse matrix A and an input dense matrix B , and produces an output dense matrix C . Both dense matrices have K columns. We refer to K as the dense matrix row size. Given a nonzero in A (e.g., a), its column index (`c_id`) is used to index a row of B and its row index (`r_id`) is used to index a row of C . Figure 2 illustrates this² for the nonzero with value a , which has `c_id`=6, and `r_id`=1. This nonzero triggers an access for the row indexed by 6 (`c_id` of a) in B and the row indexed by 1 (`r_id` of a) in C . In SpMM, for every nonzero, the corresponding row of B is multiplied by the nonzero value, and the result is accumulated on the corresponding row of C . Thus, each nonzero triggers a SIMD multiply-accumulate operation over rows with K elements. Sparse Matrix Dense Vector Multiplication (SpMV) is a special case of SpMM when $K=1$.

SDDMM takes an input sparse matrix A and two input dense matrices (B and the transposed of C denoted as C^T), and produces an output sparse matrix D that has the same nonzero structure as A . Given a nonzero in A , its `r_id` is used to index a row of B , and its `c_id` is used to index a row of C^T . In SDDMM, for every nonzero, an inner product is performed between the corresponding dense rows, and the result is multiplied with the nonzero value, and stored in the corresponding position in D .

To emphasize that the nonzeros drive accesses to rows of the dense matrices in SpMM and SDDMM, and to simplify visualization, throughout the rest of this thesis we illustrate these primitives as shown in Figure 3. In this representation, the dense matrix B in SpMM and the dense matrix C^T in SDDMM are shown transposed relative to Figure 2.

²Throughout this thesis, unless stated otherwise, we assume a row-based Gustavson-style dataflow for SpMM and SDDMM, where the computation proceeds by traversing sparse rows and using each nonzero to access the corresponding dense rows [71].

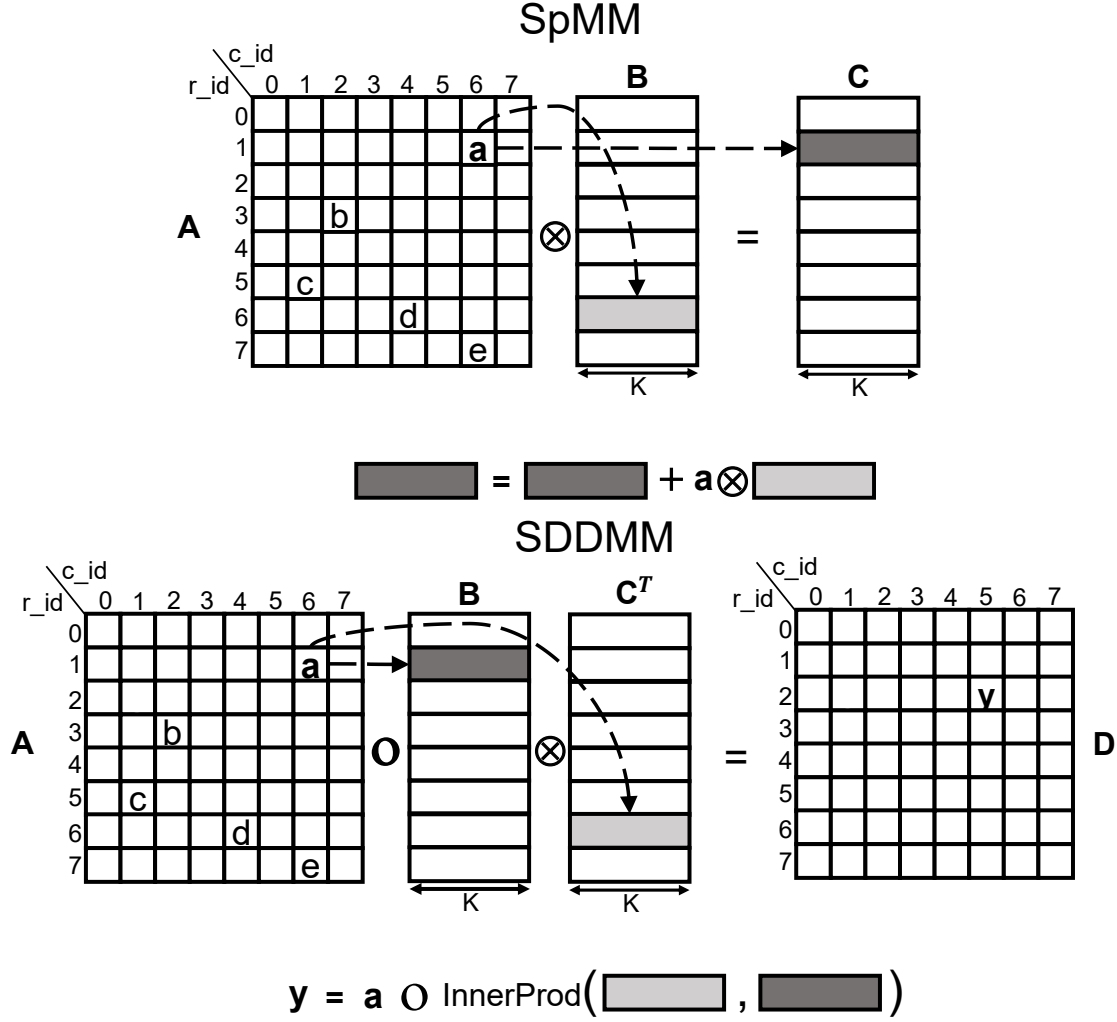


Figure 2: SpMM and SDDMM operation.

In both kernels, one of the dense matrices is accessed using the nonzero `r_id` and the other dense matrix using the nonzero `c_id`. Hence, reuse opportunities for the dense matrices depend on the structure of the input sparse matrix A : nonzeros with the same `r_id` indicate reuse for C in SpMM and B in SDDMM, while nonzeros with the same `c_id` indicate reuse for B in SpMM and C^T in SDDMM. The nonzero values are used only for a single SIMD operation and are not reused. It is evident that the structure of the input sparse matrix correlates with higher reuse of the rows of the dense matrices. For example, a denser subregion of A contains more nonzeros within the same row or column, increasing the reuse of the dense rows indexed by those nonzeros.

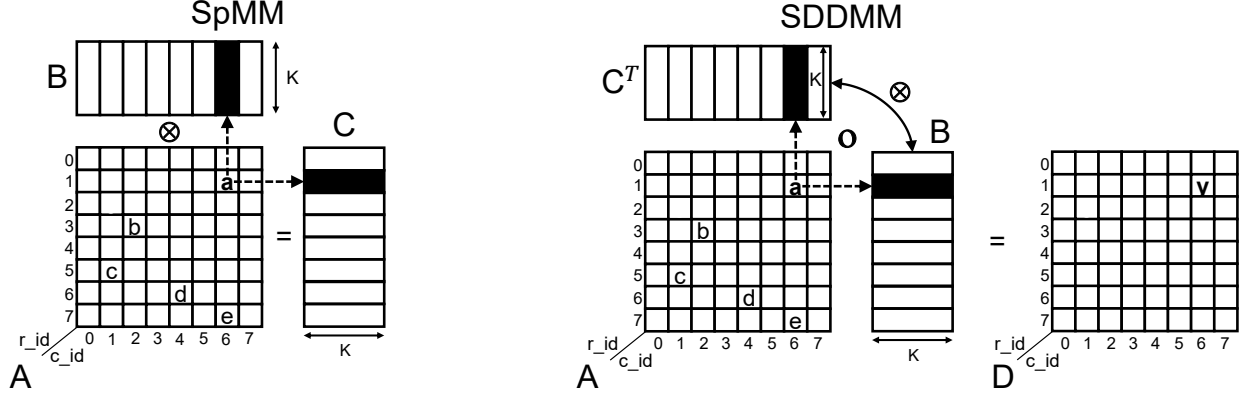


Figure 3: Simplified illustration of SpMM and SDDMM operation used throughout this thesis.

Many applications execute SpMM and SDDMM repeatedly [72], potentially interleaving the two kernels [70] and using the output of one iteration as the input to the next iteration. Across iterations, the structure of the sparse matrices may change (e.g., in graph neural network applications with sampling [64]).

2.3 Distributed Sparse Computations

Sparse computations frequently involve sparse datasets, such as social networks or computational genomics graphs, that can have terabyte-scale memory footprints [73] and high compute demands, requiring execution on large-scale cluster platforms. To that end, this section provides background on the communication patterns that arise when sparse computations are distributed across system nodes. These patterns are the optimization target of Chapter 6 and Chapter 7.

2.3.1 1D Partitioning

When executing SpMM on a distributed system, the sparse and dense matrices should be partitioned across nodes. Throughout this thesis, we use 1D partitioning, which in prior work [74] was shown to display good performance for many sparse input matrices. Figure 4 illustrates 1D partitioning for a system consisting of 4 nodes ($N_0 \dots N_3$). Elements with the

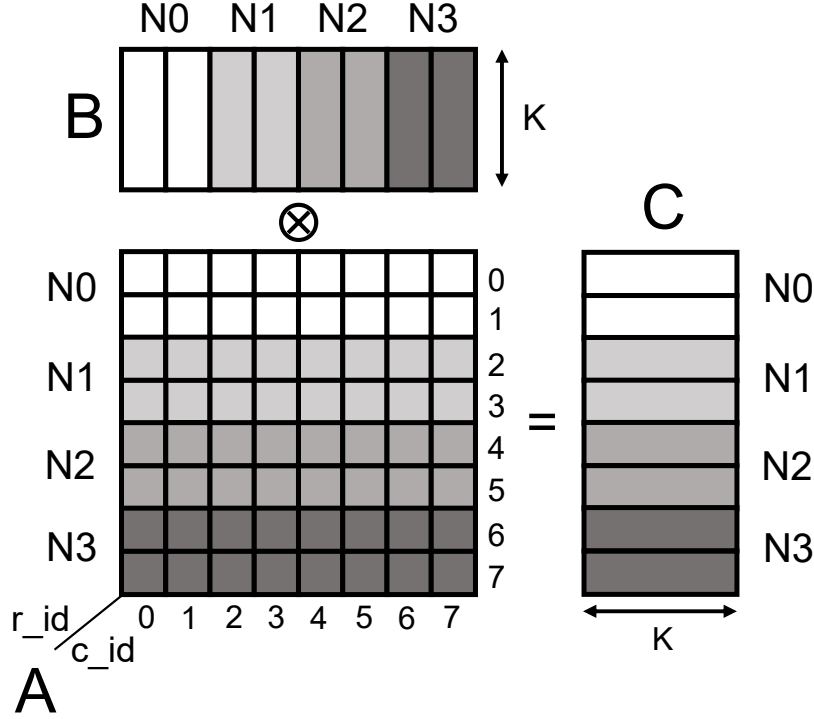


Figure 4: SpMM 1D partitioning.

same color belong to the same node. Each node is responsible for the nonzeros in a set of consecutive rows of A . It additionally hosts a set of consecutive B and C rows as shown in Figure 4. Since the dense output matrix C is indexed by the nonzeros' r_ids , the read and write accesses to C are always local. The accesses to the dense input matrix B are often remote, except for nonzeros with c_ids that index to the local portion of B (e.g., nonzeros with c_id equal to 0 or 1 in Node 1). Overall, with this partitioning scheme, remote data accesses and transfers occur only for B .

SDDMM (right part of Figure 3) exhibits similar communication patterns under 1D partitioning: accesses to B are local because B is indexed by the nonzero r_id , while accesses to C^T may be remote because C^T is indexed by the nonzero c_id . The output sparse matrix D is also local, since it has the same structure and partitioning as A .

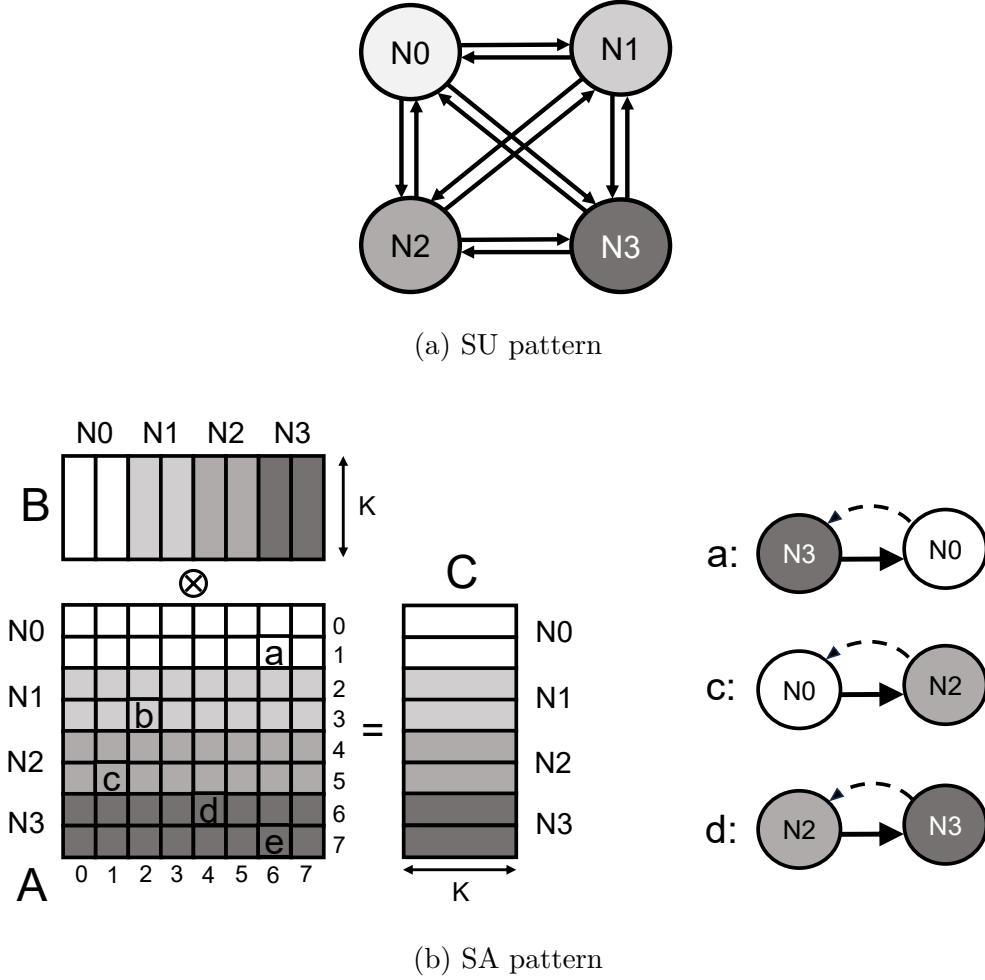


Figure 5: Examples of *SU* and *SA* patterns.

2.3.2 Sparsity-Unaware (SU) and Sparsity-Aware (SA) Transfers

Distributed sparse computations typically follow one of two communication approaches: sparsity-unaware (*SU*) or sparsity-aware (*SA*) transfers. In the *SU* approach [74], [75], data transfers are scheduled as if the computation was dense. The matrix-specific nonzero pattern is ignored, and all the nodes receive all of the dense input rows regardless of whether all rows are actually needed by all nodes. As a result, the *SU* approach leads to redundant transfers.

Figure 5a illustrates an *SU* pattern. Each node sends its local rows of B to all of the other nodes. However, the nonzero structure of the input sparse matrix A in Figure 5b

shows that this communication is conservative: although N0 receives all six remote dense input rows, it actually needs only one input row from N3.

This pattern can be implemented either with collectives [75] such as AllGather [76], [77], which fully replicates the dense input, or with shifting algorithms [74] that perform the transfers iteratively in a cyclic manner. A shifting algorithm consists of a series of computation and communication steps. Despite the redundant transfers, the SU approach can typically achieve high utilization of the network’s bandwidth, since it avoids the software overheads of generating many small network messages.

Figure 5b illustrates an example of the *SA* approach. In the SA approach, the nonzero structure of A affects the inter-node data transfer patterns. Each node traverses its local partition of the sparse matrix A and issues fine-grained read requests for rows of B . For example, Node N0 is responsible for the nonzero a , but the corresponding B row that this nonzero requires is hosted in N3. Hence, as shown in the figure, N0 issues a request to N3 for the row (dashed arrow) and N3 sends the row (solid arrow). Similar requests are issued for nonzeros c and d , but not for b and e , since the required B rows of the latter are already local.

The *SA* transfers are fine-grained – only a single row is transferred instead of all the rows of a node as in the *SU* approach. In addition, requests are typically initiated by the receiver and are asynchronous (i.e., the set of all nodes does not need to synchronize for the request to be completed). This communication pattern can be implemented with Remote Direct Memory Access (RDMA) [78]. Although SA eliminates a lot of the redundant transfers associated with SU, it can suffer significant software overhead to generate many fine-grained network messages, as we discuss later in Chapter 6 and Chapter 7.

Chapter 3 Tightly-Coupled Accelerator Architecture for Sparse Computations

3.1 Introduction

Sparse Matrix Dense Matrix Multiplication (SpMM) and Sampled Dense Matrix Dense Matrix Multiplication (SDDMM) are two fundamental sparse linear algebra kernels (Section 2.2). The SpMM and SDDMM kernels have very similar computational and memory access behavior. They have a unique mix of sparse and dense operands that sets them apart. Indeed, like in other sparse computations, the nonzero structure of the input sparse matrix determines the kernel characteristics. At the same time, the dense matrices in SpMM and SDDMM increase the data reuse and arithmetic intensity in the kernel. However, the data reuse behavior heavily depends on the input sparse matrix structure. As a result, for some matrices, SpMM and SDDMM can exhibit high locality, benefiting from fast local memories (e.g., caches), while for other matrices, they can show highly-irregular accesses and benefit little from fast local memories. In the latter case, main memory bandwidth becomes the bottleneck. In both cases, a non-negligible fraction of the requests is typically served by the lower levels of the memory hierarchy, making latency tolerance a major concern.

In addition, like dense kernels, SpMM and SDDMM can use SIMD execution units effectively but, unlike dense kernels, they are not easily amenable to spatial processing element (PE) arrays due to their hard-to-predict sparse matrix-driven memory access patterns. Overall, SpMM and SDDMM belong to a special category, which has its unique challenges.

Given the importance of SpMM and SDDMM, there are proposals for hardware accelerators of one or both kernels (e.g., [79], [80], [81]). The Tensaurus accelerator [80], which supports SpMM and other kernels, is a specialized array of PEs with access to HBM memory. It uses a custom compression format to stream sparse data. It provides some hardware configurability that allows it to support different matrix operations. Sextans [81]

is an FPGA-based accelerator designed for SpMM. It has an HBM module from which dense and sparse data is streamed to on-chip scratchpads. Extensor [79] proposes a technique to eliminate redundant computation in sparse algebra. While very useful in computations involving more than one sparse matrix, it is less beneficial for SpMM and SDDMM. More details are presented in Section 3.7.

While these proposals advance the state of the art, the approach they use faces two main challenges. First, moving data from the CPU to the accelerator and back entails substantial overhead. Such overhead results from the spatial separation between CPU and accelerator, and their connection through slow links. Even if the separation could be overcome, non-GPU accelerators still require an address space conversion (e.g., with a DMA engine), as their PEs do not support the CPU’s virtual addresses. GPUs support a mechanism for virtual address sharing with CPUs, although it has limited performance [82], [83]. Since future applications are likely to benefit from fine-grain interleaving between CPU and accelerator phases [84], [85], [86], these overheads need to be eliminated.

Second, these designs have only very marginal flexibility and programmability. For SpMM and SDDMM, they do not support different execution strategies based on the nonzero structure of the sparse input matrix. It is well known [87], [88] that such structure determines the flavors of SpMM and SDDMM algorithms that deliver the highest performance. To be future-proof, SpMM and SDDMM accelerators must be programmable.

In this chapter, we propose a novel hardware accelerator for SpMM and SDDMM that explicitly focuses on minimizing the overhead of data moves between CPU and accelerator, and on providing programmability without compromising performance. The accelerator is called *SPADE*.³

SPADE is an on-chip accelerator integrated on the same die as its host multicore CPU. SPADE minimizes data transfer overheads by tightly-coupling accelerator processing elements (PEs) with the cores of the multicore, as if the accelerator PEs were advanced

³SPADE is named after the mix of SPArse and DEnse operands in SpMM and SDDMM.

functional units. Each core is associated with one or more SPADE PEs, which share the core’s secondary TLB, L2 cache, and last level cache (LLC). While such a design is intrusive, it allows the accelerator to reuse the CPU’s memory subsystem and virtual addresses, forgoing any data move or address space conversion. Moreover, the design is scalable: larger multicores with higher memory bandwidth become larger, higher-bandwidth SPADE accelerators.

SPADE attains flexibility and programmability by supporting a special ISA. The ISA does not use fine-grained instructions, which induce instruction fetch and decode overheads; instead, it uses *high-level tile-based* instructions. A core passes these high-level instructions to the PEs, which break them into micro-operations. The resulting programmability allows the accelerator to adapt to the diverse sparsity patterns found in matrices originating from different application domains.

SPADE is designed for high performance. As indicated above, PEs do not have the overhead of fetching and decoding fine-grained instructions. Moreover, the PE pipeline is conceived for latency tolerance. Further, the cache subsystem flexibly allows cache bypassing to reduce cache pollution.

To prove the SPADE concept, we have prototyped and taped-out a simplified four-PE SPADE chip using TSMC 65nm technology. In addition, we have simulated a SPADE system with 224-1792 PEs, and shown that SPADE attains high speedups and is scalable. A 224-PE SPADE system is on average 2.3x, 1.3x, and 2.5x faster than a 56-core Intel Ice Lake CPU, a server-class NVIDIA V100 GPU, and a scaled-up, idealized version of the state-of-the-art Sextans SpMM accelerator [81], respectively, without accounting for the host-accelerator data transfer overhead. If such overhead is taken into account, the 224-PE SPADE system is on average 43.4x and 52.4x faster than the GPU and the accelerator, respectively. Furthermore, SPADE has a small area and power footprint.

Overall, this chapter makes the following contributions:

- The SPADE approach to hardware acceleration of SpMM and SDDMM, which attains

high performance by eliminating data transfer overheads through tightly-coupled host integration and provides flexibility through a tile-based ISA.

- The SPADE architecture, including a pipeline designed for latency tolerance and a flexible memory subsystem.
- Demonstration of SPADE with a simulation-based evaluation of its performance.

3.2 Motivation

At this point, we refer the reader to Section 2.2 for the necessary background on the SpMM and SDDMM operations. Note that in this chapter, we also use the term *rMatrix* to refer to matrix C in SpMM and matrix B in SDDMM as both of those matrices are accessed using the nonzero `r_ids`. Similarly, we use the term *cMatrix* to refer to matrix B in SpMM and matrix C^T in SDDMM, as both of those matrices are accessed using the nonzero `c_ids`.

Next, we motivate the two main design decisions behind the system presented in this chapter: tightly-coupled host-accelerator integration and architectural flexibility for diverse data-reuse behaviors.

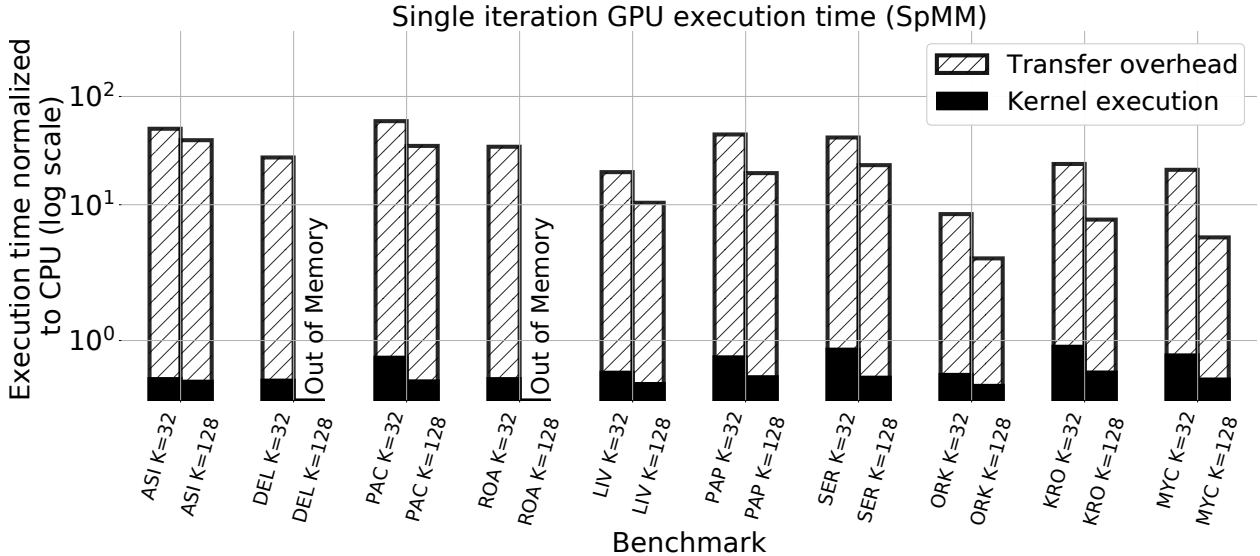


Figure 6: GPU execution times of a single SpMM iteration (including the transfer overhead) normalized to CPU execution times. Times are broken down into transfer overhead and kernel execution. The y scale is logarithmic.

3.2.1 Data Transfer Cost

In most existing accelerator designs, the host memory is separate from the accelerator memory (e.g., [89], [90]), and explicit data transfer to and from the accelerator memory is needed to perform the computation and get the results. In sparse computations, the cost of this data transfer is harder to amortize, since the data reuse in the accelerator is lower than in dense computations. Non-iterative applications and applications whose data does not fit in the accelerator memory [91] are especially sensitive to this overhead.

To quantify the overhead of these data transfers, we measured the execution time of a single SpMM iteration on two machines: (1) a dual-socket Intel Ice Lake CPU server and (2) an NVIDIA V100 GPU connected to its own host via PCIe. The GPU execution time includes the host-to-device and device-to-host data transfer time, and any address mapping or translation overhead. In the GPU experiment, we measure time using CUDA events. Unfortunately, these measurements do not allow us to decouple the address mapping overhead from the data transfer overhead. Hence, we report the value of the combined overhead. We execute 10 different matrices from the popular SuiteSparse [73] matrix collection (which we describe in Table 2 of Section 3.5) and use two different dense matrix row sizes (K equal to 32 and 128).

Figure 6 shows the execution time of the GPU system normalized to the execution time of the CPU system. We show bars for each matrix and K value. The bars are broken down into transfer overhead and kernel execution. We see that, if we just consider the kernel execution, the GPU is always faster than the CPU. However, if we consider both kernel execution and transfer overhead, the GPU is always much slower than the CPU. This is because the transfer overhead accounts, on average, for 97% of the total time.

The above analysis motivates tightly-coupled host-accelerator integration as a way to mitigate data-transfer costs in sparse computations.

3.2.2 Versatile Architecture for Data Reuse

Matrix tiling is a well-studied method to enhance data reuse in linear algebra kernels. Tiles limit the working set size so that it fits in fast local memory. In dense operations, deriving a good tiling strategy is relatively straightforward. However, in sparse operations, this is not the case, as the effect of the tile size is strongly affected by the sparsity pattern of the matrices [87]. In fact, at high sparsity levels, tiling may be useless due to limited reuse opportunities.

Sparse matrices may display reuse opportunities localized inside tiles (*Local Reuse*) or spread across different tiles (*Distant Reuse*). For an accelerator to be able to support a wide range of matrices, it needs to provide architectural knobs to exploit different reuse behaviors. Ideally, it needs to support: (1) tiles of arbitrary size, (2) techniques to control the concurrent working set of multiple threads in fast shared memories by deliberately ordering tile execution across threads, and (3) the bypass of different levels of the memory hierarchy when there are no reuse opportunities. Note that scratchpad-based solutions enforce strict limitations on tile sizes and are not as versatile as caches.

Accelerators should support these capabilities, and allow programmers to select the tiling parameters, order tile execution across threads to control the concurrent working set, and enforce caching selectively. With this support, accelerators can adapt the computation to the sparsity patterns of the input matrix.

3.3 Overview of the SPADE System

SPADE is built on two principles: attain high performance by eliminating any data move between CPU and accelerator, and attain flexibility without hurting performance by using a high-level tile-based ISA for the accelerator.

As hinted in Section 3.2.1, the overhead of moving data from the CPU to the accelerator and back can be substantial. It results from the spatial separation between CPU and accelerator, which often implies transfers through the slow PCIe bus. It is also

caused by the fact that non-GPU accelerators do not typically support the CPU’s virtual addresses, and an address space conversion is required (e.g., with a DMA engine). GPUs support a mechanism for virtual address sharing with CPUs, although it is inefficient [82], [83]. Since future applications are likely to benefit from fine-grain interleaving between CPU and accelerator phases [84], [85], [86], this overhead needs to be eliminated.

SPADE avoids this overhead by tightly-coupling accelerator processing elements (PEs) with the cores of a multicore. Each core is associated with one or more SPADE PEs, which share the core’s secondary TLB (STLB), L2 cache, and lower cache hierarchy. While such a design is intrusive, it allows the accelerator to reuse the CPU’s memory subsystem and virtual addresses, and forgo any data move or address space conversion.

The second principle driving SPADE’s design is programmability without lowering performance. Non-programmable accelerators risk obsolescence as the algorithms they accelerate evolve. For example, graph structures have been evolving, as social networks create power-law graphs. Such graphs, represented as sparse matrices, change the SpMM and SDDMM algorithm flavors that perform best. Hence, for programmability, SPADE has an ISA. However, using fine-grained instructions carries the overheads of instruction fetching and decoding. To avoid these overheads, SPADE uses a *high-level tile-based* ISA. A CPU core passes these instructions to the PEs, which then break the instructions into micro-operations. The result is programmability and high performance.

3.3.1 Tight Integration with the Host

SPADE is composed of many PEs that execute tiles of SpMM and SDDMM operations, as required by the instructions supplied by a simple general-purpose core called Control Processing Element (CPE). The PEs are integrated in the cores of a multicore to be able to: (1) directly access the cores’ memory system and (2) use the cores’ virtual addresses and STLBs.

Figure 7 shows the multicore architecture. Each CPU core is associated with a few

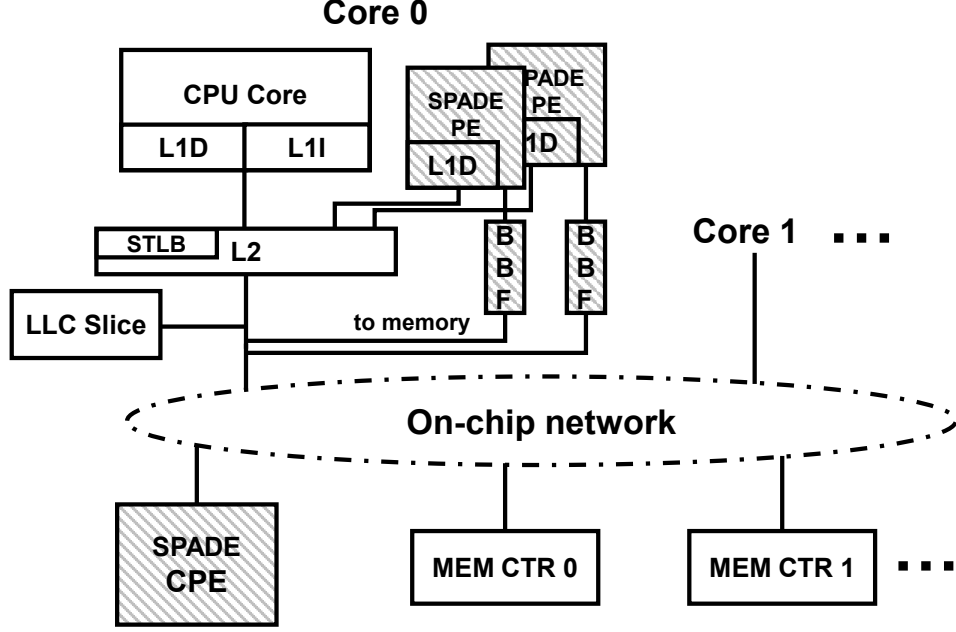


Figure 7: Integration of SPADE processing elements (PEs) and the Control Processing Element (CPE) in a CPU multicore.

(1-4) SPADE PEs which, for energy efficiency, are clocked at a fraction of the core’s frequency (e.g., $1/4$). The PEs share the core’s STLB (like the DMA engines in [92]), the core’s L2 cache, and the lower cache hierarchy. Each PE has an L1 data cache and, because many of the sparse data structures do not use caches efficiently, a Bypass Buffer (BBF) that optionally allows PE accesses to bypass the cache hierarchy. No instruction cache is needed, as explained later. Overall, this coupling of CPU core and SPADE PE(s) makes the system scalable: larger multicores with higher memory bandwidth will result in larger, higher-bandwidth accelerators.

Programs interleave CPU-mode execution sections with SPADE-mode execution sections. In the former, only CPU cores are active; in the latter, only the SPADE PEs and the CPE are. Before a SPADE-mode section starts, the pages of the matrix data structures are pinned in physical memory, so that SPADE PEs do not suffer page faults. SPADE PEs can suffer TLB misses.

Because SPADE PEs have their own L1 cache and can access data through the BBF, special cache operations are needed in the transitions between the two modes. Specifically,

before transitioning from SPADE to CPU mode, the SPADE PEs' L1 caches are written back to the L2 caches and invalidated, and the BBFs are written back to main memory and invalidated.

Similarly, before transitioning from CPU to SPADE mode, two actions are done. First, the CPU cores' L1 caches are written back to the L2 caches and invalidated. Second, any data currently in the cache hierarchy that, in the upcoming SPADE-mode section, could be accessed by the SPADE PEs through the BBFs needs to be written back to memory and invalidated from the cache hierarchy. This is to ensure that the accesses through the BBFs will get the correct data versions and that no stale data will be left in the caches after the next SPADE-mode section. If the programmer or compiler cannot identify the aforementioned data, all cached data may need to be written back and invalidated.

The CPE gets the PEs to start, stop, or execute any instruction. To do so, the CPE uses a mechanism similar to the Intel Architecture MWAIT instruction [93], which enables a processor to wait for a store operation to an address. Specifically, the CPE has a few memory-mapped registers for each PE called *Input* registers. When the CPE writes to one of the Input registers of a given PE, that PE is informed in hardware, unless notifications are masked. With this mechanism, the CPE can trigger operations on a PE.

Putting all this together, programs execute as follows. When the CPU cores run, the PEs are paused. When the program wants to switch from CPU mode to SPADE mode, the cores perform the selective cache writebacks and invalidations described above, notify the CPE, and pause. The CPE writes an initialization instruction in the Input registers of all the PEs, and then the tile-based coarse-grained instructions that each PE needs to execute. Every time that the CPE writes to an Input register, the corresponding PE is informed. That PE reads the Input register that contains the instruction and executes the instruction. As soon as a PE has read an Input register, the CPE is informed, and the CPE can then overwrite the Input register with a new instruction. After the SpMM or SDDMM operation completes, the CPE commands the PEs to write back and invalidate

the L1s and BBFs, and terminate execution. The program then resumes in CPU mode.

3.3.2 Tile ISA and Programming

The instructions that the CPE sends to PEs to execute are SpMM or SDDMM operations on a tile. Before these instructions are generated, a compiler or a programmer analyzes the sparse input matrix and decides on a set of good configuration parameters. These parameters include the size of the tiles of the sparse input matrix (Row Panel Size and Column Panel Size, as shown in Figure 8(a)), the order of tile execution, and whether PE accesses to the different data structures should bypass the caches. The row and column panel sizes chosen determine the layout of the matrix in memory, as shown in the Chapter Supplement 3.9. Both in the supplementary example and in our evaluation, we use the COO format.

At run time, a program running on the CPE takes the layout of a tiled matrix and generates instructions for the PEs. Specifically, assume that we want to perform an SpMM or SDDMM operation as shown in Figures 8(a) and 8(b). The figures show all the parameters and, in shaded pattern, the tiles.

The CPE first creates the *Initialization* instruction, shown in Figure 8(c), which is passed to all PEs to initialize them. The instruction includes, as arguments: the primitive type (SpMM or SDDMM); base addresses of the rMatrix and cMatrix; base addresses of the r_ids, c_ids, and vals arrays of the tiled input sparse matrix (shown in Chapter Supplement 3.9) which, together with the *sparse_in_start_offset* metadata (discussed in Chapter Supplement 3.9), generate the addresses of the beginning of each tile data; for SDDMM only, the base address of the vals array of the tiled output sparse matrix; the cache hierarchy bypass strategy for the row and column matrices (the two choices for each of them are bypass or no bypass); the size of the indices and vals (which vary depending on the size of the matrix and the type of the data); and the size of the dense matrix row K (shown in Figures 8(a) and 8(b)). On reception of this instruction, the PEs store it in

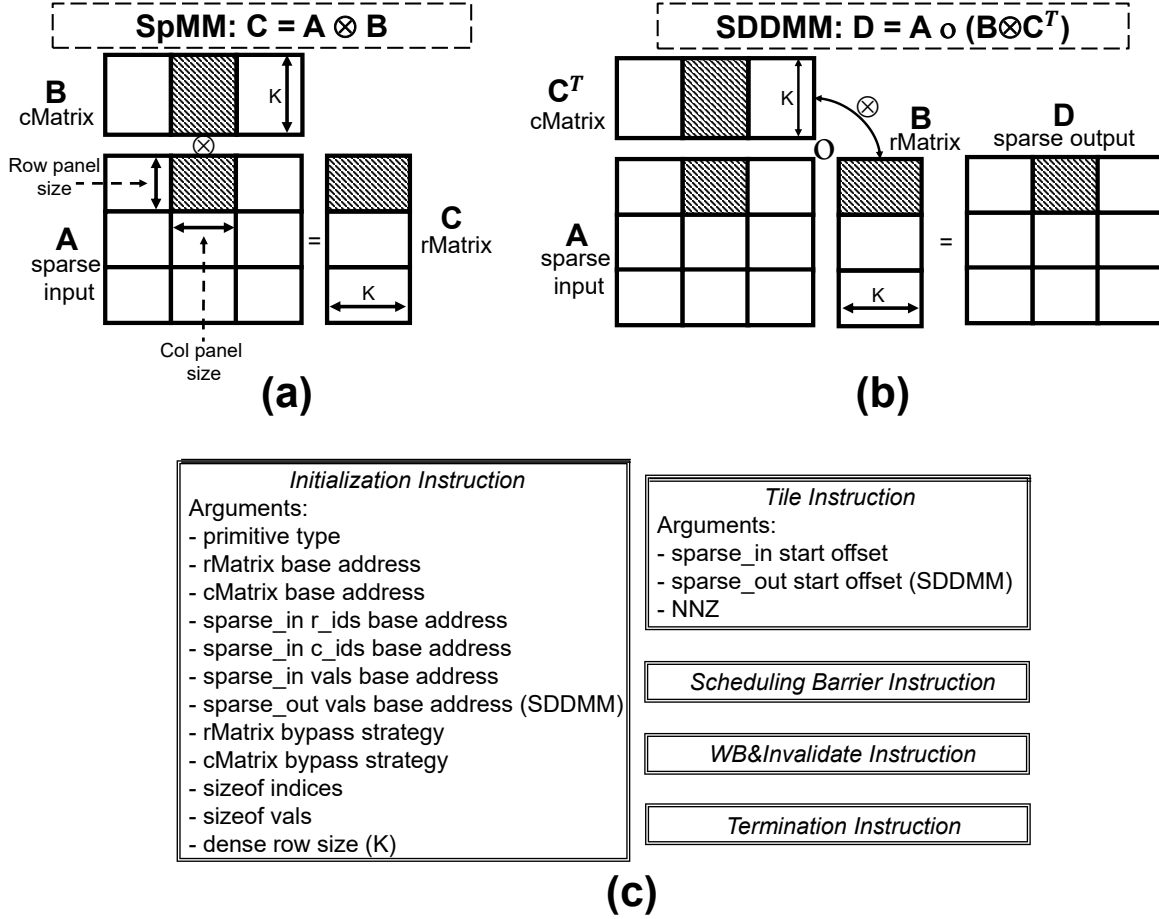


Figure 8: Matrix operations and instructions supported by SPADE.

special registers and reconfigure some of their hardware.

After that, the CPE creates *Tile* instructions that it sends to individual PEs. A tile instruction, shown in Figure 8(c), specifies the tile to work on. Its arguments are data from the tiling metadata shown in Chapter Supplement 3.9: the offset of the first nonzero of the tile in the r_ids, c_ids, and vals arrays of the input sparse matrix; for SDDMM only, the offset of the first nonzero of the tile in the vals array of the output sparse matrix; and the number of nonzeros in the tile. The PE that receives this instruction processes the tile. There are no upper/lower bound constraints on the tile size.

The figure shows three more instructions, *Scheduling Barrier*, *WB&Invalidate*, and *Termination*. They are discussed next.

3.3.3 Tile Scheduling

The CPE freely assigns tiles to the PEs. However, in SpMM, the assignment must respect one constraint: all the tiles in a *row panel* should be assigned to the same PE. A row panel is the set of tiles that span the same set of rows of the matrix, as shown shaded in Figure 9(a). The reason is that the SpMM operation on two tiles of the same row panel updates the same rows of the rMatrix. To avoid data races, only one PE processes such tiles. As an example, Figure 9(a) shows the order of assignment of tiles to two PEs. PE 0 (in white circles) gets the tiles in Row Panel 0 and then those of Row Panel 2; PE 1 (in black circles) gets the tiles in Row Panel 1 and 3. This assignment restriction does not apply to SDDMM.

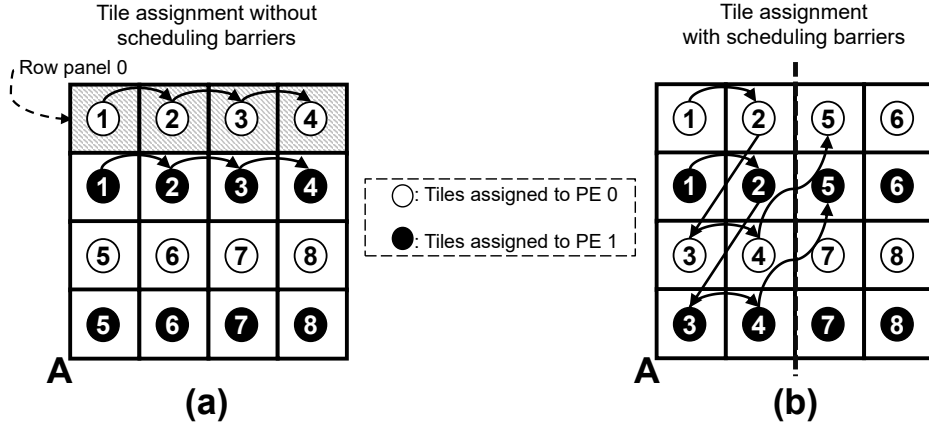


Figure 9: Tile scheduling without (a) and with (b) barriers.

A schedule like Figure 9(a) maximizes inter-tile rMatrix reuse, but hinders inter-tile cMatrix reuse. Specifically, consider the tiles assigned to PE 0 only (i.e., those with white circles), and assume that Tiles labeled 2 and 6 in Figure 9(a) have nonzeros in the same columns. As PE 0 executes Tile 6, it reuses cMatrix data brought into caches during its execution of Tile 2. However, this *Distant Reuse*, as described in Section 3.2.2, can be missed if the execution of Tiles 3, 4, and 5 evicts the cMatrix data.

When an analysis of the input matrix determines that this reuse of cMatrix data is important, we want to force a schedule like in Figure 9(b). In the figure, PE 0 and PE 1

first execute tiles in column panels 0 and 1, and once they are both done, they move to the next two column panels. Both PEs help each other reuse cMatrix data. This behavior is attained with the SPADE *Scheduling Barrier* instruction. The CPE passes Tiles 1, 2, 3, and 4 in Figure 9(b) to PE 0 and, after PE 0 has read the instruction for Tile 4, the CPE sends it the barrier instruction. Similarly, after PE 1 has read the instruction for its own Tile 4, the CPE sends it a barrier. The CPE will not send any new tile instruction to any PE until *both PEs* have read the barrier instruction—signifying that both PEs have completed their last assigned tile.

A similar procedure is used to terminate a SPADE-mode section. The CPE sends the *WB&Invalidate* instruction to all PEs. When a PE reads it, the PE writes back and invalidates its L1 cache and BBF, while the CPE sends the *Termination* instruction to the PE. When the PE has completed the writebacks and invalidations, it reads the termination instruction and pauses. When the CPE notices that all PEs have read the termination instruction, it stops SPADE-mode execution and CPU-mode execution can restart.

Finally, to ensure correctness in the presence of BBFs, there are two data layout requirements. First, the dense matrix row size (K) must be a multiple of the cache line size and, therefore, dense rows start at cache line boundaries. Second, in SDDMM, the first nonzero value of each tile in the output sparse matrix must be at the beginning of a cache line. To satisfy these conditions, padding may be inserted in the cMatrix, rMatrix, and the output sparse matrix of SDDMM.

3.3.4 Processing Element Pipeline

The PE pipeline is designed for latency tolerance. It is composed of three logical stages: sparse front-end, vOp Generator, and dense back-end (Figure 10). We describe each stage next.

Sparse front-end. This stage receives the tile instruction from the CPE and performs two functions: issue requests for the sparse matrix data and generate the addresses of the

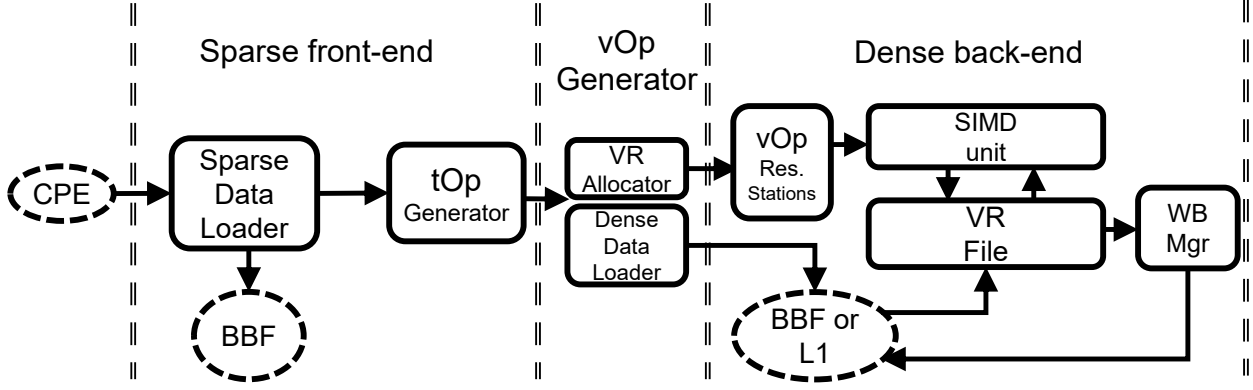


Figure 10: SPADE processing element pipeline overview.

dense matrix data. As discussed in Section 3.3.2, the instruction has all the information needed to load the sparse data. Since the sparse data is not reused, the loads are issued by a Sparse Data Loader to the BBF and bypass all caches. As the data is received, the hardware uses the information in the r_ids and c_ids arrays to generate the addresses of the dense cMatrix and rMatrix rows needed. Specifically, for each value in the $vals$ array, it generates a *Tuple Operation* (tOp) with: {begin address of rMatrix row, begin address of cMatrix row, value from the vals array}. For SDDMM, there is a fourth field: address of the entry in the val array of the output matrix that will receive the result.

vOp Generator. The PE operates as a simple out-of-order vector engine and has a Vector Register File (VRF). Each Vector Register (VR) stores a full cache line. However, a tOp refers to dense matrix rows, which may extend over multiple cache lines. Hence, the vOp Generator stage breaks down each tOp into potentially multiple vector-long operations called *Vector Operations* ($vOps$). The two operands of a vOp are cache-line sized and thus fit in VRs.

Next, for each vOp, the vOp Generator allocates two VRs for the two operands, and the Dense Data Loader requests the operands from memory. These dense accesses are directed to either the L1 or the BBF, depending on the cache bypassing strategy chosen. A vOp includes: {VR ID for the first operand, VR ID for the second operand, value from the vals array}. In addition, for SDDMM, the vOp also includes a destination VR ID (for the

scalar output data), and the offset of the output data in the destination VR. Eventually, in SDDMM, this destination VR will contain scalar outputs from multiple vector operations that generate the data in the destination memory line. For SpMM, the implicit destination VR is the VR of the rMatrix input. In all cases, the VRs allocated are tagged with the memory line addresses requested. Note that, before allocating a register, the hardware checks the VRF tags to see if the requested data is already in a register from a previous operation. If so, the data is not requested from memory.

Dense back-end. This stage pushes the newly-generated vOps to the vOp Reservation Stations. Once the two operands of a vOp are ready, the operation is dispatched to a pipelined SIMD unit, which executes multiply-accumulate and multiply-reduce operations. The result is stored in the VRF. Since there are no explicit store instructions, a Write-back Manager unit periodically writes back the contents of dirty VRs to the memory subsystem.

This pipeline is designed for latency tolerance, maximizing the overlap of memory accesses with each other and with computation. First, the sparse front-end coalesces and overlaps read requests to sparse data. Second, the pipeline overlaps sparse data read requests issued in the front-end with requests issued in the subsequent two stages (dense data read and write requests and sparse data write requests). Finally, the back-end uses out-of-order execution, overlapping read and write requests with each other and with computation.

3.4 Key Architecture Components

In this section, we provide more architectural details.

3.4.1 Detailed Pipeline Analysis

To understand the pipeline better, we explain it with an SpMM example. Figure 11 illustrates, with numbers in circles, the correspondence between the pipeline structures, the pipeline functionality and the memory accesses. It also shows the processing of the first tile

of an input matrix in COO format.

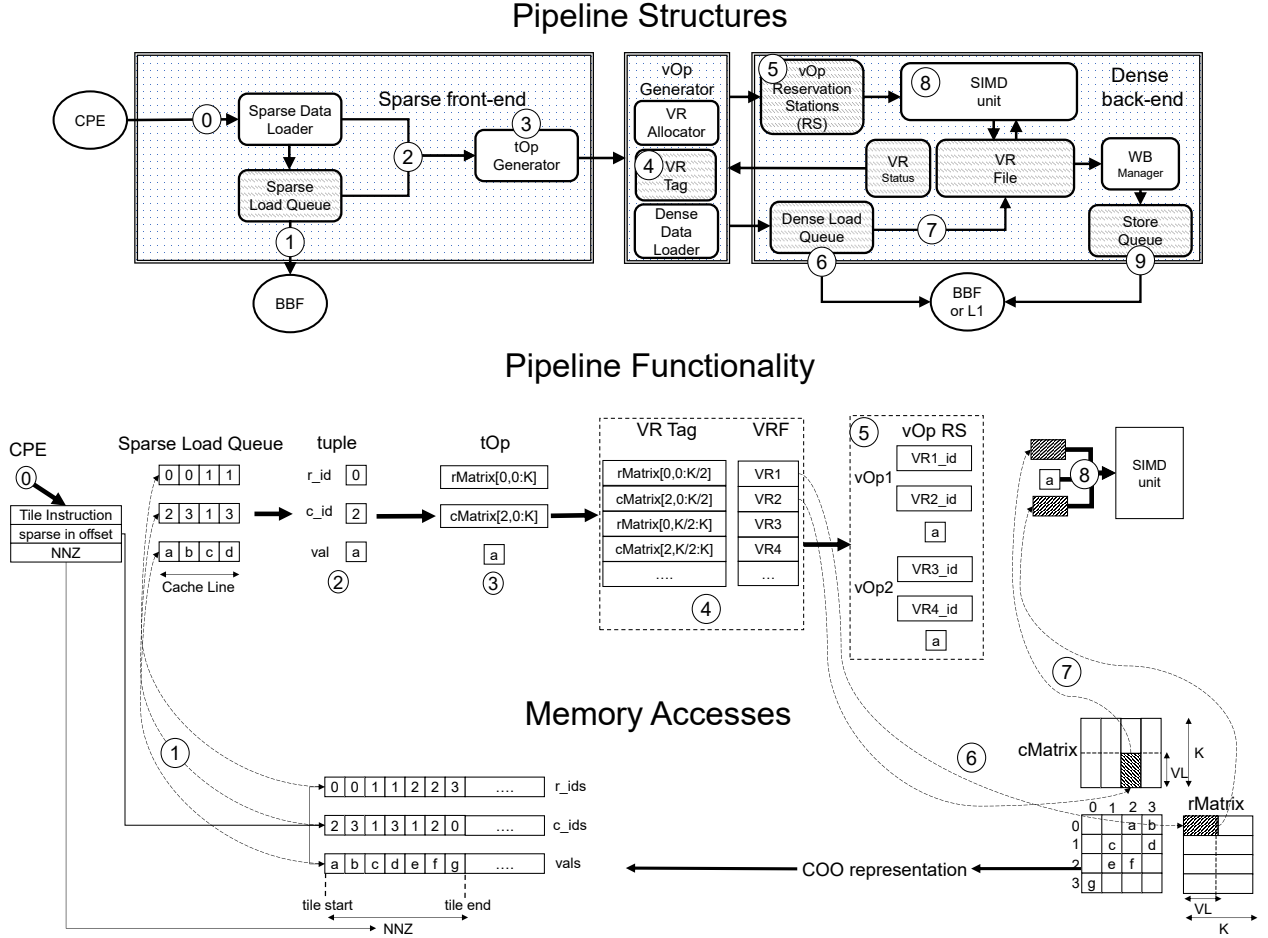


Figure 11: Pipeline structures, pipeline functionality, and memory access behavior.

Tile instruction ①. Figure 11 shows the three arrays (r_ids , c_ids and $vals$) of the sparse matrix. The tile instruction has the offset of the tile (*sparse_in_offset*) relative to the beginning of the arrays (given in an earlier initialization instruction) and the number of nonzeros (*NNZ*) of the tile. In our example, these values are 0 and 7. This information is enough to address all the tile nonzeros.

Loading nonzeros ①. The Sparse Data Loader generates cache-line sized requests for data from the 3 one-dimensional arrays. We assume that a cache line contains 4 elements. The figure shows the response for the first request. For as long as there are free entries in the Sparse Load Queue and the total tile *NNZ* has not been reached, a new request is

issued every cycle. In our example, the next request would bring the next four elements of the arrays. However, because NNZ is 7, only three of the elements of the second request would be used. No more requests will be issued.

Determining dense addresses and generating tOps ②-③. Elements from the Sparse Load Queue are popped and tuples of the form (r_id, c_id, val) are isolated. In the figure, we show the first tuple, namely $(0, 2, a)$. The r_id and c_id are used to index an rMatrix row and a cMatrix row, respectively. The base addresses of the dense matrices (which have been given in an earlier initialization instruction) are incremented by the r_id and c_id displacements. In our example, r_id is multiplied with the size of a dense row (K) and added to the rMatrix base to create the address range $rMatrix[0,0:K]$; the same is done with c_id to create $cMatrix[2,0:K]$. The tOp Generator module takes these two address ranges plus the nonzero value (a) and creates a Tuple Operation (tOp) (Figure 11).

Allocating vector registers and generating vOps ④. The core operation of SPADE is a SIMD operation between a sparse value and dense rMatrix/cMatrix rows. In order for the SIMD operations to take place, the rMatrix and cMatrix operands must be fetched from memory and stored in VRs. The pipeline is equipped with a Vector Register File (VRF) with a Vector Length (VL) equal to the system cache line size. However, a dense row may extend over multiple cache lines. For this reason, the tOp is broken down into finer-grained Vector Operations (vOps) whose operands are cache-line sized. In our example, we assume that the dense row size K extends over 2 cache lines (i.e., $K=2*VL$). Hence, 2 vOps are generated from each tOp; one for the first half of the dense rows and one for the second half of the dense rows.

The vOp Generator module has a VR Tag structure, which records the addresses of the cache-line-sized data stored in each VR. In our example, the vOp Generator module picks four empty registers and tags them with the four addresses of the two vOps. As shown in the figure, $rMatrix[0,0:K/2]$, $cMatrix[2,0:K/2]$, $rMatrix[0,K/2:K]$, and $cMatrix[2,K/2:K]$ are assigned to VR1, VR2, VR3, and VR4.

A desired vOp operand may already be in the register file from a previous operation. Consequently, before filling an entry in the VR Tag, the vOp Generator checks whether the address is already in the VR Tag. If so, the vOp Generator picks the corresponding VR for the vOp. The VR Tag structure is organized as a content-addressable memory (CAM).

There is also an auxiliary VR Status RAM that has additional status bits for each VR. They indicate if the entry is dirty or unused. They can be updated on a VR access and allocation/deallocation.

Pushing vOps into the vOp reservation stations ⑤. Once the VRs of vOps have been allocated, the vOps are pushed to slots in the vOp Reservation Stations for out-of-order execution. Figure 11 shows the two slots used by vOp1 and vOp2. A vOp carries additional information beyond the VR ids. It carries the input scalar value (a in our example) and, not shown in the figure, information that records: (i) inter-vOp dependencies and (ii) offsets of sub-cache line operands. These offsets are translated to masks to prevent overwriting unwanted cache-line contents.

Loading dense values ⑥. The Dense Load Queue issues requests for the operands of the vOps. These are cache-line requests for rMatrix and cMatrix data. If the VRs corresponding to these addresses already contain the data from a previous operation, no request is issued to memory.

Dense data arrival and execution ⑦-⑧. The vOps execute out of order. A vOp waits in a vOp reservation station if an operand has not yet arrived from memory or has a data dependence with an operand of an earlier vOp. Once the vOp receives all its operands and its dependencies are resolved, the vOp is dispatched for execution in the pipelined SIMD.

Data dependence tracking between vOps is simple for SpMM and SDDMM. Consider SpMM first. A vOp can only write to an rMatrix cache line and, furthermore, it must read the line first. Moreover, when the vOp Generator module accesses the VR Tag for a vOp, it marks that the selected VR will be both read and then written. As a result, the only data dependence that can exist between two different vOps is RAW—i.e., the second vOp

will read the value that the first vOp will write. The dependence is detected and recorded when the vOp Generator accesses the VR Tag for the second vOp.

With this design, the pipeline may process vOps out of program order. This is not a problem because the accumulation operation of SpMM is associative. The same argument is true for SDDMM, except that the output is a single scalar rather than a vector.

Writing data back to memory ⑨. SPADE does not have explicit store instructions. vOps simply update the VRF. Hence, dirty VRs must be written back to memory in the background to free-up VRs to allocate new operands. Two extreme approaches are: (1) to write back a VR as soon as it gets dirty and (2) to write back VRs only when the VRF is all filled with dirty VRs. Unfortunately, the first approach causes an excessive number of stores to the memory subsystem, and the second one places writebacks in the critical path. Therefore, SPADE has a Write-back Manager unit in the pipeline that uses an intermediate approach: it initiates writebacks when the fraction of VRs that are dirty exceeds a certain threshold, and stops when the fraction falls below a second threshold.

3.4.2 Cache Bypassing

The ability of SPADE to provide a flexible cache hierarchy, with the capability to bypass the caches is important. The reason is that different input matrices can use the cache hierarchy very differently. To understand the choices, consider the different data structures in an SpMM and SDDMM operation.

For the input sparse matrix data, since each tuple is read once, used for a single tOp, and never reused again, it is best to bypass all caches and avoid polluting them.

When it comes to the rMatrix data, recall that individual rMatrix rows are reused only by a single PE to avoid data races. Since L2 caches and lower-level caches are shared by multiple PEs, one needs to be careful to cache rMatrix data, lest it pollute the caches for other PEs. Consequently, we identify three cases, depending on the pattern of nonzeros in the sparse input matrix and the size and order of tiles processed by a PE, as determined by

barriers.

First, if the VRs have high reuse in the VRF, then caching the rMatrix does not offer significant benefits and, therefore, it is best to bypass the caches. Second, if the size and schedule of tiles is such that the distance between reusing rMatrix lines is very large, then VR reuse in the VRF will be low, and it is best to cache the lines, so they can still be in the caches when needed. The third case is like the second one except that the working set of reused rMatrix lines is small at any time, while the overall footprint of rMatrix lines is large. Caching all the rMatrix lines would pollute the caches for small gains. Hence, we augment the BBF with a small *Victim Cache*. The few rMatrix lines in the working set bypass the caches and use the victim cache in the BBF.

For the cMatrix data, caching is the best choice. The reason is that, inside a tile, data is processed in row order. As a result, the VRs in the VRF rarely have good reuse. Finally, for the output sparse matrix of SDDMM, cache bypass is most beneficial. The reasons are that bypassing eliminates cache pollution and, in addition, the output sparse matrix has high reuse in the VRF, as the vOps that access a given output cache line are bunched up.

3.5 Methodology

A. Architectures: Table 1 lists the architecture parameters of the simulated SPADE accelerator and its host CPU multicore. The latter is modeled after a dual-socket Intel Ice Lake server with 28 cores per socket. Its architectural details are taken from the literature [94], [95]. The SPADE PEs cycle at only 0.8GHz and, for hardware simplicity, only issue one tOp and vOp every cycle. A PE has 64 physical vector registers. As soon as 25% of them are dirty, the writeback manager starts writing them back until only 15% are dirty. Four PEs share the L2 of a CPU core. Neither L2 nor the LLC are partitioned. We evaluate SPADE with cycle-level simulations using SST [96]. To simulate DRAM, we integrate DRAMsim3 [97] in SST.

As baseline machines to compare to SPADE, we use a CPU, a GPU, and the

Table 1: Microarchitecture of SPADE and its host CPU multicore system, modeled after a 2-socket Ice Lake with 56 cores total.

	Ice Lake	SPADE
Frequency	2.6/3.5GHz (base/turbo)	0.8GHz
Issue width	5 uOp / cycle	1 vOp / cycle
Load queue size	128 entries	32 (dense) + 6 (sparse)
ROB size	352 entries	N/A
Store queue size	72 entries	8 entries
Physical vector registers	224	64 (WB thres: 0.25, 0.15)
SIMD FP units	3	1 (single precision)
Integer units	yes	no (hardwired address calculation)
Frontend-Backend interface	Allocation queue; 140 entries	tOp queue; 16 entries
OoO execution support	160 scheduler entries	32 vOp res. station entries
Phys. register allocation	Register alias table	VR tag CAM
Speculative execution	yes: shadow RAT etc.	no
Frontend complexity	High: L1I, branch predictor, etc.	Low: simple address calculation
L1D size	Core: 48KB; Total: 2.625 MB	PE: 32KB; Total: 7.2MB
L1I size	Core: 32KB; Total: 1.75 MB	-
BBF	-	PE: 32 entries; 64B/entry; Total: 0.45MB
Victim cache	-	PE: 16KB; 2-way; 64B/entry; Total: 3.6MB
L2 size	Core: 1.25 MB; Total: 70 MB	1.25 MB per 4 PEs; Total: 70 MB
LLC size	Core: 1.5 MB; Total: 84 MB	1.5 MB per 4 PEs; Total: 84 MB
# of cores or PEs in 2 sockets	56	224
Total DRAM Bandwidth	max theoretical: 410GB/s	max theoretical: 410GB/s

state-of-the-art Sextans SpMM accelerator [81], which is an FPGA-based accelerator connected to its host through PCIe. The CPU baseline platform is the dual-socket Intel Ice Lake server described above. The GPU baseline platform is a server-class NVIDIA V100 connected to its host through PCIe, with 16GB of global memory and 900GB/s achievable global memory bandwidth. We simulate Sextans using SST and DRAMsim3. For fairness, we scale-up Sextans by using 16 PE groups (PEGs), each with 16 PEs clocked at 0.8 GHz. Each PE is equipped with 16 Processing Units (PUs). We scale-up the capacity of the on-chip scratchpads to 170MB (which is more than the total capacity of the SPADE cache subsystem). We additionally: (1) do not take into account AXI-related FPGA limitations, (2) disregard intra-PEG load imbalance, and (3) idealize the Sextans computation engine by only accounting for the time needed for memory accesses. Finally, we compress the sparse data so that each {row_id, col_id, val} tuple consumes 8B, and implement the address interleaving used by the authors. Although SPADE could also benefit from such

optimizations, we do not employ them for SPADE.

B. Benchmarks: We use 10 large graphs from SuiteSparse [73] coming from different domains (Table 2). We present our evaluation for two different dense row sizes ($K=32$ and $K=128$). We group matrices based on whether they benefit from tiling or scheduling barriers. If they always benefit from them, we say the matrices have a high *Restructuring Utility* (RU); if they sometimes benefit from them (e.g., in only one of SpMM or SDDMM, or only for $K=128$), we say they have a medium RU; for the remaining matrices, we say that they have a low RU. Table 2 shows the RU of the matrices.

Table 2: Benchmark graphs evaluated.

Graph (Short Name)	Domain	Nodes (Mill.)	Edges (Mill.)	Density (Order)	Restructuring Utility (RU)
asia_osm (ASI)	Road graph	11.9	25.4	10^{-7}	Low
com-LiveJournal (LIV)	Social network	4.0	69.4	10^{-6}	Medium
com-Orkut (ORK)	Social network	3.1	234.4	10^{-5}	High
coPapersCiteseer (PAP)	Citation graph	0.4	32.1	10^{-4}	Medium
delaunay_n24 (DEL)	Geometry problem	16.8	100.7	10^{-7}	Low
kron_g500-logn20 (KRO)	Synthetic graph	1.0	89.2	10^{-5}	High
mycielskian17 (MYC)	Mathematics (fractals)	0.1	100.2	10^{-2}	High
packing-500x100x100-b050 (PAC)	Numerical simulations	2.1	35.0	10^{-5}	Low
road_usa (ROA)	Highway graph	23.9	57.7	10^{-7}	Low
Serena (SER)	Environmental science	1.4	64.1	10^{-5}	Medium

C. Software kernels for the baseline machines: For the Ice Lake server, we use the Inspector-Executor (IE) kernel from Intel’s MKL [98] library for SpMM. We believe that MKL IE already employs optimizations such as tiled execution. Since MKL does not include an SDDMM kernel, we use the SDDMM implementation from TACO [99] (similar to prior work [79]). TACO does not include an input-aware kernel. The kernels were compiled using the 2022.2.1 version of Intel oneAPI DPC++/C++ compiler. For the GPU, we use the SpMM kernel from NVIDIA’s cuSPARSE library [100], and the SDDMM kernel from dgSPARSE [101] using CUDA 11.6. We profiled both the transfer time and the kernel execution time.

In our baselines, we use the CSR format for high performance. In SPADE, we use the COO format. This is conservative, since SPADE could also benefit from CSR’s reduced

memory footprint.

D. Chip prototype: We prototyped in a chip a simplified SPADE design called *miniSPADE* and taped it out using TSMC 65nm technology. Each miniSPADE die (Figure 12) consists of 4 in-order SPADE PEs. Each PE is equipped with a bypass buffer (BBF) and an L1 cache. The die also includes a shared L2 cache and a memory buffer.

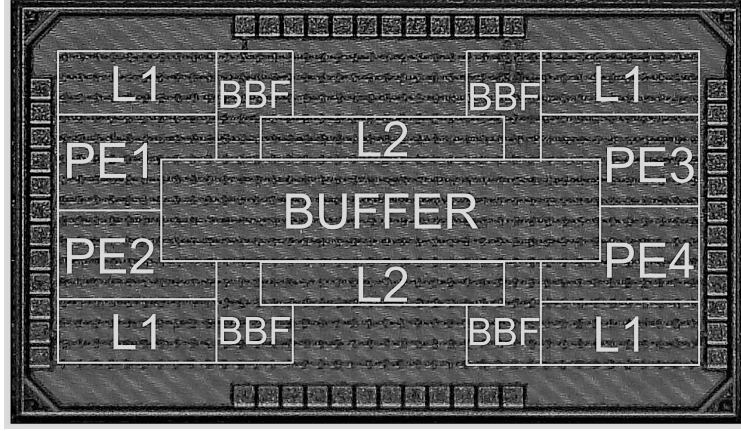


Figure 12: The miniSPADE die.

At 65nm, the die size is 1.75mm x 1.00 mm, dominated mainly by SRAM for the caches and memory buffer. At 200MHz, the power consumption of the die is 30mW. Due to the simplified PE design and the large feature size, miniSPADE is not a fully representative depiction of the actual envisioned system. However, it serves as a proof-of-concept for various SPADE features (e.g., front-end, tOps, vOps, and cache bypassing).

E. Area and power estimation: To estimate the area and power of SPADE, we model the L1D, BBF, victim cache, and all the pipeline memory structures (CAMs, RAMs, and registers) using CACTI [102] targeting 32nm. For the single-precision FP SIMD unit, we use the numbers from [103]. Our synthesis of miniSPADE suggests that additional logic (e.g., multiplexers and finite state machines) account for less than 5% of the miniSPADE pipeline area. Thus, for SPADE, we conservatively assume that the additional logic accounts for 20% of its pipeline area and power. Since Ice Lake uses 10nm technology, we scale both the power and area of SPADE down to 10nm by using the scaling factors

from [104]. Finally, we use CACTI to estimate the L2 and LLC power consumption and DRAMsim3 for the DRAM.

3.6 Evaluation

A. Performance comparison to CPU and GPU: We consider three SPADE variants: *SPADE Base*, *SPADE Opt*, and *SPADE2 Base*. The SPADE Base variant does not use any flexibility knobs: sparse matrix tiles always have a row panel size equal to 256 rows and a column panel size equal to the total number of columns in the matrix; the rMatrix and cMatrix data is accessed without bypassing caches; and there are no scheduling barriers.

To configure SPADE Opt, we test: (1) three different tile row panel sizes and three column panel sizes; (2) accesses to rMatrix that bypass or do not bypass the cache hierarchy; and (3) for the medium-sized column panels that we test, not inserting or inserting scheduling barriers to limit the LLC working set. For the small and large column panels, we do not use scheduling barriers. The parameters tested are shown in Table 3. For the MYC sparse matrix, which has a very small number of rows, we also test a row panel size of 16 to mitigate load imbalance. Although SPADE supports a large space of different parameter settings, we limit our analysis to these. Then, we set SPADE Opt to be, for each individual matrix, the version with the best-performing parameter settings that we tried. SPADE Opt is the proposed system variant in this thesis.

Table 3: Parameters examined to set SPADE Opt.

Dense Row Sizes	Row Panel (RP) Sizes	Column Panel (CP) Sizes	rMatrix Caching Strategies	Scheduling Barriers
K=32	64, 256, 1024	8192, 524288, all_columns	No bypass, bypass	No, yes for CP=524288
K=128	64, 256, 1024	2048, 131072, all_columns	No bypass, bypass	No, yes for CP=131072

SPADE2 Base is a scaled-up version of SPADE Base with double PE count, DRAM bandwidth, LLC size, and link latencies.

Figure 13 shows the speed-up of the server-class GPU (*assuming zero host-device data transfer overhead*), SPADE Base, SPADE Opt, and SPADE2 Base over the baseline CPU

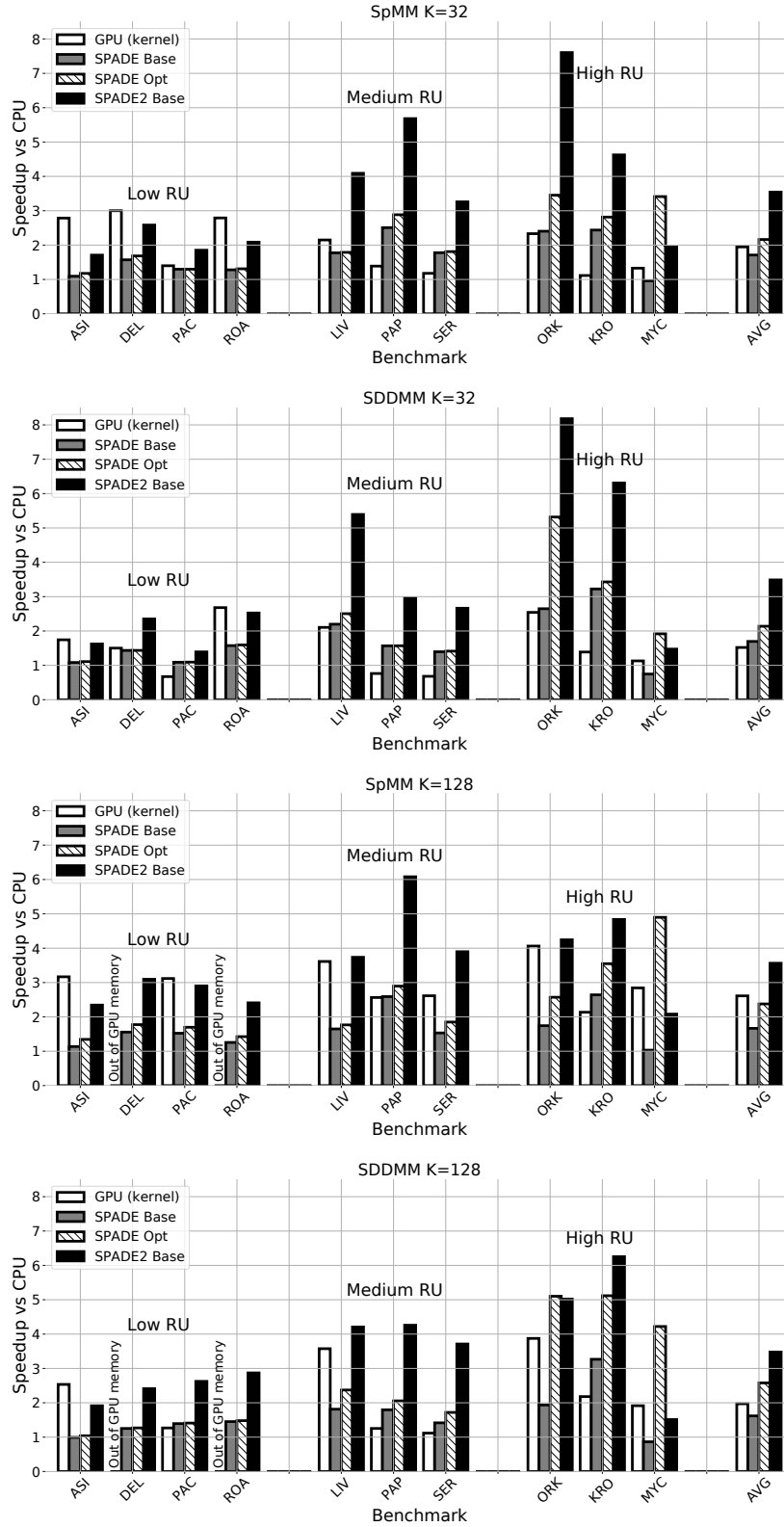


Figure 13: Speedup of the SPADE variants and the GPU (ignoring data transfers) over the CPU. Matrices are grouped into low, medium, and high Restructuring Utility (RU).

server. For matrices that do not fit in the GPU memory (i.e., DEL and ROA for $K=128$) we assume a GPU speedup over the CPU of 1.

As shown in Figure 13, the speedups of SPADE Base, and the effectiveness of SPADE’s flexibility as demonstrated by SPADE Opt, vary for each matrix category. First, for *low RU* matrices, the SPADE Base speedups are lower. This is because matrices in this category show limited reuse opportunities and, therefore, the SPADE Base speedup mainly stems from better utilization of the available memory bandwidth. In addition, we see that the benefit of SPADE Opt’s flexibility is smaller. The reason is that SPADE Base with very large tiles is already a good fit for this category. Furthermore, without accounting for the host-device data transfer, the GPU is faster than SPADE Opt, due to having an achievable memory bandwidth (900GB/s) that is much higher than the maximum observed bandwidth of SPADE Opt (304GB/s). Although SPADE2 Base only has a bandwidth of 600GB/s, it is competitive with the GPU baseline.

For *high* and *medium RU* matrices, both the SPADE Base speedup and the flexibility benefits of SPADE Opt are higher. A sizable portion of the requests are served from caches with low-latency and without introducing DRAM bandwidth pressure. The SPADE PEs can utilize this pattern, by being able to satisfy a larger number of requests from the caches while long-latency main memory ones are pending, hence hiding the latency of the latter. By using the flexibility knobs, SPADE Opt outperforms the GPU for most matrices in these categories. The speedups of SPADE2 Base are higher.

Overall, on average across all the environments shown in Figure 13, SPADE Base, SPADE Opt, and SPADE2 Base are 1.67x, 2.32x, and 3.52x faster, respectively, than the CPU. The corresponding speedups over the GPU are 1.03x, 1.34x, and 2.00x. The performance gap between SPADE Opt and SPADE Base reveals that flexibility is a key feature for an SpMM/SDDMM accelerator. Further, the performance of SPADE2 Base suggests that our system is scalable. Our evaluation reveals that SPADE is a superior architecture in comparison to GPUs *even if host-to-device data transfers and address*

remapping overheads are ignored. If these overheads are taken into account (Figure 6), SPADE Opt is 43.4x faster than the GPU.

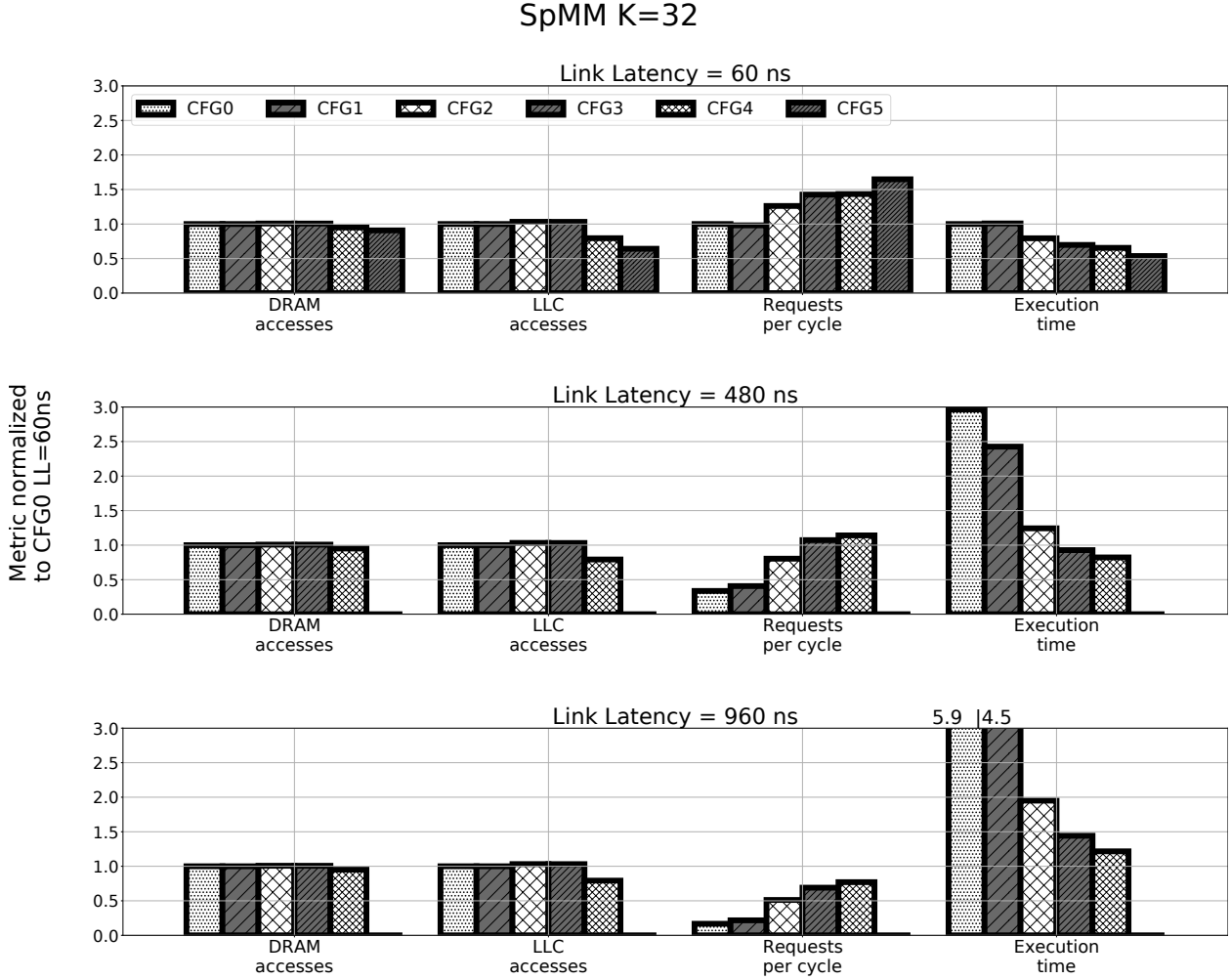


Figure 14: Impact of progressively adding system features to SPADE system configurations.

B. Pipeline analysis: To provide further insight into the effect of some of the system optimizations, we perform an optimization analysis for SpMM with K=32. We evaluate 5 different SPADE system configurations by progressively adding system features. Table 4 presents the configurations.

Since SPADE is built for high latency tolerance, we evaluate each of the SPADE configurations for three different link latencies. Specifically, we consider the average round-trip latency of an access from the PE to a memory controller, without including the

Table 4: Evaluated SPADE configurations.

	SPADE features included
CFG0:	Tile instructions, 3-entry sparse load queue, overlap of sparse and dense requests, 16 vOps RS entries, 56 SPADE PEs at 3.2GHz
CFG1:	CFG0 with 32 vOps RS entries
CFG2:	CFG1 with 224 SPADE PEs at 0.8GHz
CFG3:	CFG2 with 6-entry sparse load queue
CFG4:	CFG3 + sparse data bypasses the cache hierarchy ($\equiv$ SPADE Base)
CFG5:	CFG4 + flexible execution ($\equiv$ SPADE Opt)

latencies to access the memory or any of the caches on the way. We call this time the link latency (LL). We set this LL to 60ns (default for our SPADE architecture), 480ns, and 960ns. For CFG5, we only evaluate LL=60ns to reduce the number of experiments required. For each configuration and LL value, Figure 14 shows the number of DRAM accesses, the number of LLC accesses, the average number of requests that the pipelines collectively issue per cycle, and the execution time. The figure shows the geometric mean of the metrics across all the matrices, normalized to CFG0 for 60ns LL.

Compare the different configurations for a given LL. A high number of requests per cycle indicates active pipelines. It can mean either enhanced latency tolerance or decreased latency due to better utilization of the caches. If an increase in requests per cycle is not accompanied by a decrease in DRAM and/or LLC accesses, it is an indicator of increased latency tolerance.

From the figure, we see that the optimizations progressively applied in CFG1, CFG2, and CFG3 result in an increased number of requests per cycle being generated *without the LLC or DRAM accesses decreasing*. Hence, they increase latency tolerance. CFG4 and CFG5 increase the number of requests per cycle while decreasing the LLC and DRAM accesses, and thus decrease the average request latency. We also see that the reduction in execution time induced by the progressive optimizations increases as LL increases.

C. Evaluation of flexibility knobs: We now show how the different flexibility knobs of SPADE benefit different sparse matrices. Recall that the knobs are: different tile sizes, cache bypassing or not, and scheduling barriers or not. First, we perform a sensitivity

analysis of how the tile row panel (RP) and column panel (CP) sizes affect the execution time. We disable bypassing and insert no barrier. Figure 15 shows the results for three matrices. It shows the execution times normalized to the execution time of the worst-performing setting. Thus, lower values and darker colors are better. We observe that KRO, which belongs in the High RU category, benefits from tiles with a small CP and a large RP since, in this way, cMatrix reuse opportunities can be maximized. On the other hand, DEL, which has low RU, benefits from tiles with a CP that extends over all of the columns. Finally, for MYC, which has a small number of rows, the selection of small RPs mitigates load imbalance.

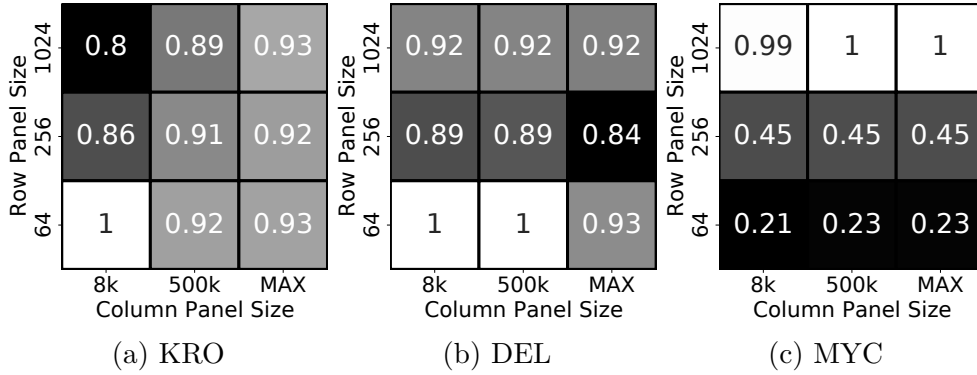


Figure 15: Execution time of SpMM with K=32 for different tile column/row panel sizes, normalized to the worst parameter setting. Lower numbers and darker colors are better.

Table 5 examines the impact of applying scheduling barriers. It shows the percentage change in execution time of the setting with medium RP and CP sizes and no cache bypassing when we apply barriers. Positive numbers are slowdowns. We observe that the effect of barriers is matrix-dependent. It can either increase the execution time (by up to 80.5% in ASI SpMM128 – i.e., SpMM with K=128) or reduce it (by up to 57.1% in ORK SpMM with K=128).

Table 6 examines the impact of applying cache bypassing for the rMatrix. For each benchmark, we pick the setting of tile RP and CP sizes and scheduling barriers that delivered the best performance. On top of this setting, we apply cache bypassing. The table shows the percentage change in execution time when we apply cache bypassing.

Table 5: Percentage change in execution time by applying scheduling barriers. Positive numbers are slowdowns.

	ASI	DEL	PAC	ROA	LIV	PAP	SER	ORK	KRO	MYC
	Low RU	Low RU	Low RU	Low RU	Med. RU	Med. RU	Med. RU	High RU	High RU	High RU
SpMM32	22.8	41.1	4.3	31.7	24.2	0	1	-29.6	-12.6	0
SDDMM32	9	6.5	0.6	5	-15	0	0.6	-48.3	-1.2	0
SpMM128	80.5	76.6	12.8	56.2	34	47.7	9.4	-57.1	-37	0
SDDMM128	19.6	37.8	8.2	22.9	-16.9	13.2	-3.8	-53.7	-36.9	0

Positive numbers are slowdowns. We observe that, for most of the benchmarks, rMatrix cache bypassing is beneficial. It can reduce the execution time by up to 32.9% in ORK SpMM with K=128. However, it can also slow down the execution. For example, in KRO SpMM with K=32, cache bypassing leads to a 169.2% increase in execution time. Recall from Figure 15 that KRO benefits from a large row panel. Unfortunately, this row panel size overflows the victim cache in the BBF, leading to many main memory spills.

Table 6: Percentage change in execution time by bypassing the caches for the rMatrix. Positive numbers are slowdowns.

	ASI	DEL	PAC	ROA	LIV	PAP	SER	ORK	KRO	MYC
	Low RU	Low RU	Low RU	Low RU	Med. RU	Med. RU	Med. RU	High RU	High RU	High RU
SpMM32	-7	-6.7	4.2	-2.3	-0.8	13.9	-1.8	-17.2	169.2	0.7
SDDMM32	-3.2	0.1	-0.3	-1.2	-6.8	0.2	-0.1	-7	0.2	0.1
SpMM128	-9.9	-10.4	-9	-12.2	-0.7	-7.1	-14.4	-32.9	-6.3	-1.9
SDDMM128	-4.5	-0.3	-1.1	-1.9	-13.2	-6.3	-12.8	-18.7	-8.4	-2.7

Overall, the input-dependent impact of tiling, scheduling barriers, and cache bypassing justifies our emphasis on flexibility.

D. Overhead of mode transitions: When execution repeatedly interleaves CPU and SPADE mode sections, it will repeatedly suffer mode transition overheads. We have measured such overheads for our benchmarks. The transition from SPADE to CPU mode involves writing back the PEs’ L1 caches to the L2s, writing back the BBFs and victim caches to memory, and invalidating L1s, BBFs, and victim caches. For our benchmarks, this overhead is on average 0.2% of the SPADE mode duration.

The overhead of the transition from CPU to SPADE mode is generally application

dependent. We assume an execution that repeatedly interleaves CPU sections and SPADE PEs executing SpMM, or CPU sections and SPADE PEs executing SDDMM. We additionally assume that the CPU updates a dense matrix that will be an input for SPADE (i.e., cMatrix for SpMM and rMatrix or cMatrix for SDDMM); this is a reasonable assumption for social network GNN applications. For SpMM, all that is needed is to write back and invalidate the CPU’s L1 caches. There is no action to take for the two structures that SPADE PEs will access through the BBFs (rMatrix and input sparse matrix): the rMatrix is not accessed by the CPU and the input sparse matrix is only read. For SDDMM, the transition needs to write back and invalidate the CPU’s L1 caches and write back and invalidate the rMatrix from the caches. This is because, of the three structures that SPADE PEs access through the BBFs (the rMatrix and the input and output sparse matrices), only the rMatrix is a potential concern: the sparse output matrix is not accessed by the CPU and the input sparse matrix is only read. For our benchmarks, these transition overheads are negligible for SpMM and on average 3.4% of the SPADE mode duration for SDDMM. Overall, transition overheads are small.

In all of our experiments, when computing the execution time of an application on SPADE, we have included two overheads: the start-up overhead due to caches starting empty rather than warmed-up and the termination overhead, which is the above overhead of transitioning from SPADE to CPU mode. For our benchmarks, the start-up overhead is on average 0.9% of the SPADE mode duration.

E. Scalability: We perform a strong scaling analysis of SPADE. Figure 16 shows the speedup of different SPADE system sizes over the baseline 224-PE SPADE (as described in Table 1), which has the default link latency of 60ns. SPADE2 Base, SPADE4 Base, and SPADE8 Base are scaled-up versions with 2x, 4x, and 8x the PE count, DRAM bandwidth, LLC size, and link latency. For reference, the figure also shows bars with linear scaling.

SPADE scales well in most benchmarks. The exceptions are MYC and KRO, which have few sparse matrix rows; load imbalance hinders their strong scaling. Superlinear

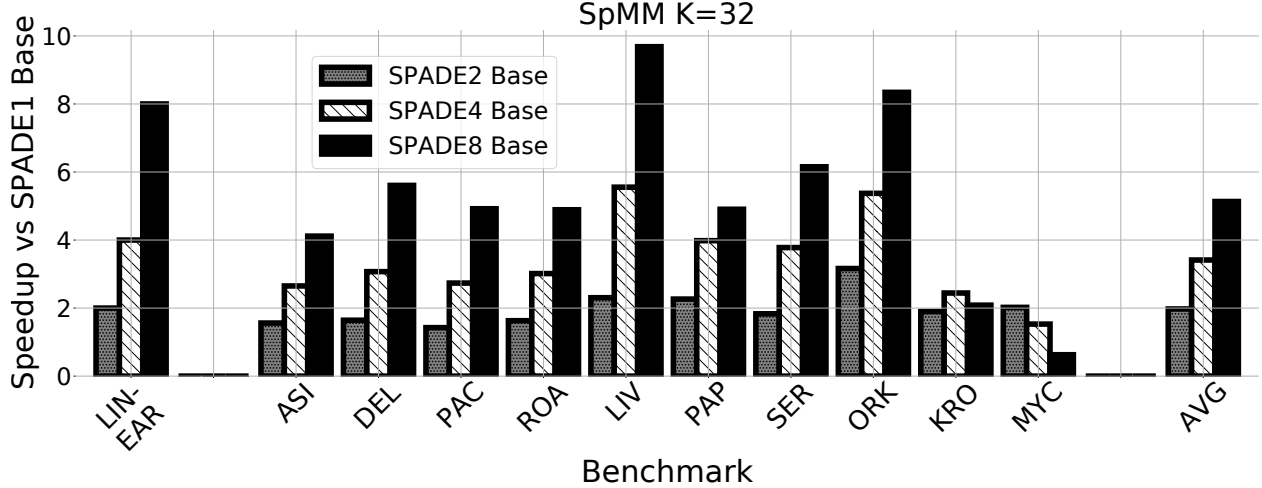


Figure 16: Scaling analysis of SPADE.

speed-up is observed in some cases due to the increase in LLC size. We repeated this experiment for $K=128$ and for SDDMM, and observed similar results.

F. Performance comparison to an ideal Sextans accelerator: Sextans [81] is a high-performance FPGA-based accelerator for SpMM. Like other accelerators, it does not address the cost of host-device data transfers. It adopts a one-size-fits-all execution model based on data streaming and sequentially-batched phases, which has some similarities with applying scheduling barriers in SPADE. Its execution model suffers from multiple reads to sparse data when the dense row size K increases. It can also lead to multiple accesses to the dense input for large matrices (e.g., ROA, DEL and ASI) when the dense output does not fit on the Sextans scratchpad.

We simulate Sextans and compare it to SPADE. As discussed in Section 3.5.A, we idealize many parts of the Sextans architecture in our simulation and disregard the host data transfer cost. For these reasons we call it *ideal* Sextans. Note that by removing the FPGA-related limitations, our Sextans version achieves a maximum memory bandwidth utilization of 50%, which is significantly higher than the 15% reported in [81].

Figure 17 shows the bandwidth utilization, DRAM accesses, and speedup of SPADE Opt, all normalized to Sextans (for SpMM $K=32$). SPADE Opt issues many concurrent

requests for both sparse and dense data, across different matrix regions. This results in a 40% higher average bandwidth utilization. In addition, by supporting different parameter settings on a per-matrix basis, SPADE eliminates redundant memory accesses, issuing 32% fewer memory accesses (up to 73% for ROA). Moreover, thanks to SPADE Opt’s decoupled and flexible design, SPADE Opt is much faster than ideal Sextans. Its average speedup is a significant 2.4x (and a maximum of 5.1x). Ideal Sextans performs marginally better only for ORK and LIV (10% and 3%). Recall from Table 5 that these matrices benefit from barriers, which resemble the Sextans execution strategy.

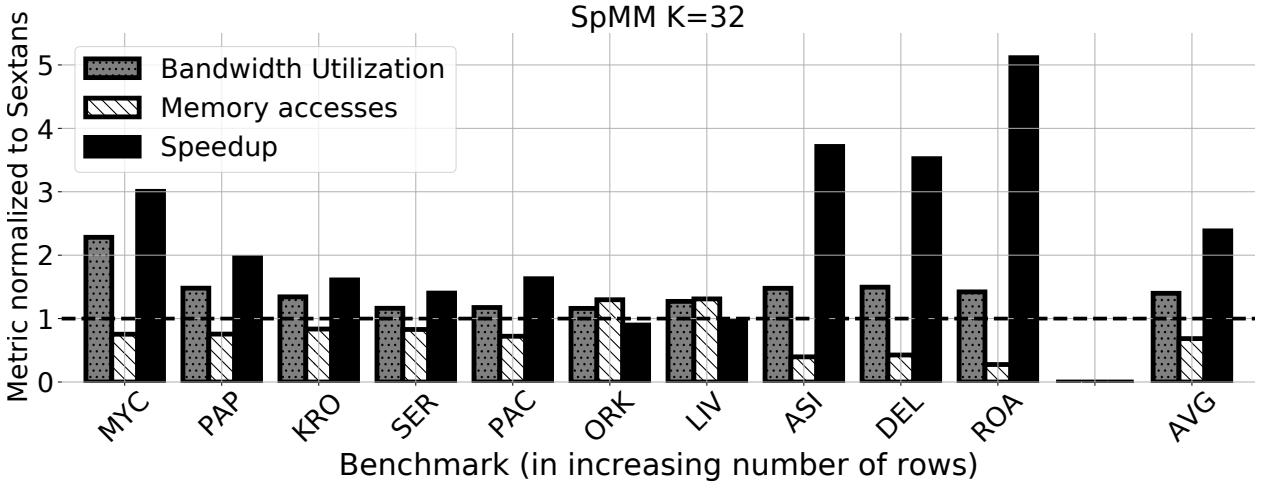


Figure 17: Bandwidth utilization, DRAM accesses, and speedup of SPADE Opt, all normalized to ideal Sextans.

For $K=128$, it can be shown that SPADE Opt’s average speedup is 2.6x. Sextans does not support SDDMM. Finally, including the PCIe data transfer overhead for Sextans, the average SPADE Opt speedup is 52.4x (for a single SpMM iteration).

G. Area and power evaluation: Augmenting a dual-socket Ice Lake [95] with the SPADE PEs, L1 caches, BBFs, and victim caches adds a very small area and power overhead. This is because, compared to the 5-issue speculative Xeon core, a SPADE PE has a simple, single-issue pipeline (Table 1). Further, the PE L1 and victim cache have a single port and low associativity. Using the tools of Section 3.5.E, we estimate that the 224 SPADE PEs with their L1s, BBFs, and victim caches consume 20.3W and take $24.64mm^2$

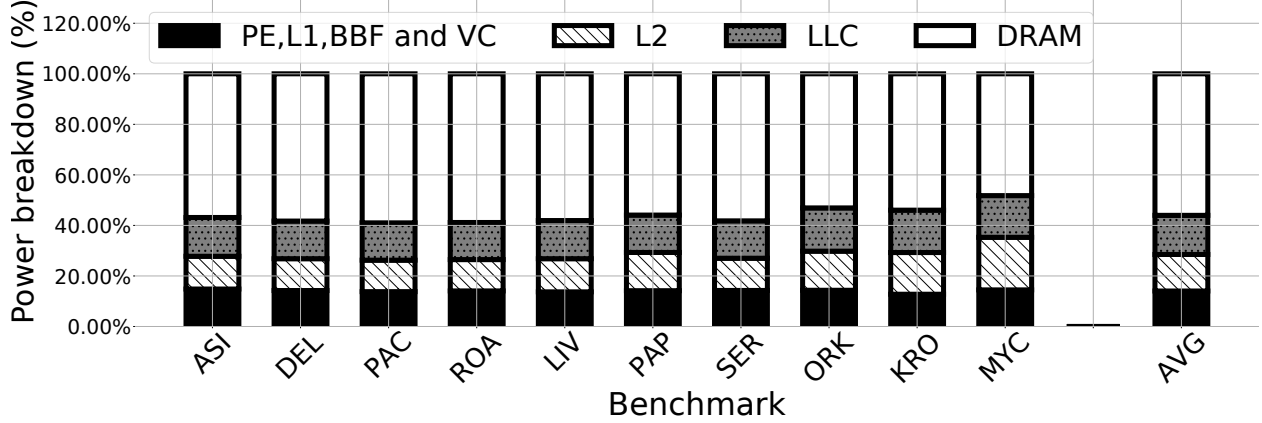


Figure 18: Breakdown of the power consumed in SPADE-mode execution for SpMM and K=32.

at 10nm. Compared to a dual-socket Ice Lake server that has a TDP of 470W and a combined die size of 1000 mm^2 [105], the SPADE power is 4.3% of the host TDP, and its area is 2.5% of the host area.

In SPADE-mode execution, the server disables the Xeon cores and L1 caches, and the SPADE PEs use the server’s memory subsystem. In this environment, Figure 18 breaks down the power consumed into the fraction spent by the 224 SPADE PEs, L1s, BBFs, and victim caches; by the L2 caches; by the LLC; and by the DRAM. We conservatively assume that the PE pipelines operate with maximum dynamic power. We see that the SPADE PEs, L1s, BBFs, and victim caches consume on average only 14% of the total power. The power dissipated in the lower levels of the cache hierarchy is also low because the sparse matrix and (sometimes) the rMatrix bypass the caches. Finally, DRAM power accounts for more than 50% of the total power.

3.7 Related work

A. SpMM/SDDMM accelerators: Several accelerators have been proposed for either both or one of SpMM and SDDMM [79], [80], [81]. Extensor [79] proposes a technique to eliminate redundant computation in sparse algebra. This technique only yields significant benefits in kernels where more than one matrix is sparse (e.g., Sparse Matrix Sparse Matrix

multiplication). For SpMM and SDDMM, Extensor’s speedups are lower and they mainly stem from placing specialized units closer to memory. Tensaurus [80] relies on a custom compression format to ensure streaming accesses of sparse data, a specialized PE array, and HBM. Sextans [81] was described in Section 3.6.F. Recently, the scalability of SpMM was investigated in PIUMA, which is an architecture designed by Intel for graph analytics at scale [91], [106]. Thanks to the huge capacity of the PIUMA memory pool, the number of data moves from host CPU to PIUMA is minimized.

These prior works do not provide a solution for fine-grained interleaving between host and accelerator phases. For example, Tensaurus and Sextans assume that the matrices already reside in the accelerator’s HBM and do not account for the low bandwidth of host-device data transfers. In addition, none of these works offers knobs for flexible execution catered to the unique characteristics of SpMM and SDDMM .

B. GNN and sparse Deep Neural Network (DNN) accelerators: There are several accelerator and software optimization proposals for accelerating GNNs [70], [92], [107], [108], [109], [110], [111], [112], [113], [114], [115], [116], [117], [118]. They often focus on overlapping the memory intensive SpMM with compute-intensive dense matrix multiplications. However, these approaches are not applicable to accelerate SpMM and SDDMM as stand-alone primitives. Accelerators [119], [120], [121] and CPU extensions such as SAVE [122] and VEGETA [123] have been proposed to increase the utilization of the compute units in sparse DNN applications. Although these approaches are effective for the low sparsity levels of DNNs (10%-90%), for the high sparsity levels of real world graphs (Table 2), the memory subsystem becomes the bottleneck. In order to accommodate such sparsity levels, SPADE interacts with the memory subsystem in an efficient and flexible manner.

C. Eliminating host-accelerator transfer overhead: In some architectures such as the Qualcomm Snapdragon [124], the GPU and CPU are integrated in the same SoC and share the datapath and memory. Although this approach may be viable for mobile devices

or desktops, integrating a server-class CPU (e.g., a $500mm^2$ Ice Lake) with a server-class GPU (e.g., a $815mm^2$ V100 [125]) in the same die is currently impractical. Instead, combining the small area footprint of SPADE with the reuse of the area-hungry lower levels of the CPU caches is a more attractive alternative. Recently, NVIDIA introduced Grace/Hopper [126], where GPUs are able to access the host CPU main memory through the high-bandwidth NVLink interconnect. Although such high-bandwidth interconnects can alleviate much of the CPU-GPU data transfer penalty [127], it is still unknown whether this new approach will completely eliminate the inefficiencies of prior CPU-GPU virtual address space sharing methods [82], [83]. With SPADE, we show a way to eliminate both overheads that does not require precious high-performance network resources. Finally, our evaluation reveals that even in the ideal case when both data transfer and address remapping overheads are eliminated in GPUs and accelerators, SPADE still outperforms both GPUs and accelerators (Figures 13 and 17). Recent trends towards built-in HBM in high-end CPU servers [128] are only going to increase the performance benefits of SPADE.

D. Flexible accelerators: Motivated by the importance of flexibility in graph and sparse linear algebra algorithms, researchers have developed accelerators that can be reconfigured to support different algorithm variants or dataflows [129], [130], [131], [132], compression formats [133], data-dependent execution patterns [134], or that use heterogeneity [135] as a flexibility enabler. Transmuter [136] is a CGRA-based architecture that reconfigures its memory type and dataflow to accommodate kernels of varying arithmetic intensity. SPADE supports flexibility through a tile-based ISA, providing flexibility knobs tailored to the unique characteristics of SpMM/SDDMM.

E. Flexible execution strategies in CPU/GPU systems: Input-aware techniques such as enhancing locality through reordering [87], [137], [138], [139], improving load-balancing [101] through row partitioning, or selecting the appropriate execution strategy based on machine learning [88], [140] improve SpMM and SDDMM performance in CPUs and GPUs. These techniques are orthogonal to SPADE.

3.8 Conclusion

This chapter proposed *SPADE*, a hardware accelerator for SpMM and SDDMM. SPADE avoids data transfer overheads by tightly-coupling accelerator PEs with the cores of a multicore processor, allowing the accelerator to reuse the cores' memory system and virtual addresses. SPADE attains flexibility and programmability by supporting a *tile-based* ISA. A simulated 224-PE SPADE system is on average 2.3x, 1.3x, and 2.5x faster than a 56-core CPU, a server-class GPU, and a state-of-the-art SpMM accelerator, respectively, without accounting for the host-accelerator data transfer overhead. If such overhead is taken into account, the 224-PE SPADE system is on average 43.4x and 52.4x faster than the GPU and the accelerator, respectively. Further, SPADE has a small area and power footprint.

Besides SpMM and SDDMM, SPADE can already support Sparse Matrix Vector Multiplication (SpMV) and Sampled Dense Vector-Dense Vector Multiplication (SDDVV). We believe that, with lightweight modifications, it can support a richer set of primitives (e.g., TTV, TTM, and MTTKRP [99]).

3.9 Chapter Supplement: A Tiled Sparse Matrix

This section shows the tiled sparse matrix representation that we use. Figure 19(a) shows a 4x4 sparse matrix, represented with the three conventional arrays in COO format.

Figure 19(b) shows the representation of the matrix when tiled with a row panel size and a column panel size equal to two. The entries of the `r_ids`, `c_ids`, and `vals` arrays have been reordered, consolidating the per-tile entries together. In addition, we have the tiling metadata: offsets of the beginning of each tile (*sparse_in_start_offset*); number of nonzeros in each tile (*tile_NNZ*); for SDDMM only, the offsets of the beginning of each tile in the output sparse matrix (*sparse_out_start_offset*, necessary because output tiles have to be aligned to cache lines); and which row panels each tile belongs to (*tile_row_panel_id*), needed so that all the tiles in a panel row are executed by the same PE to avoid data races in SpMM.

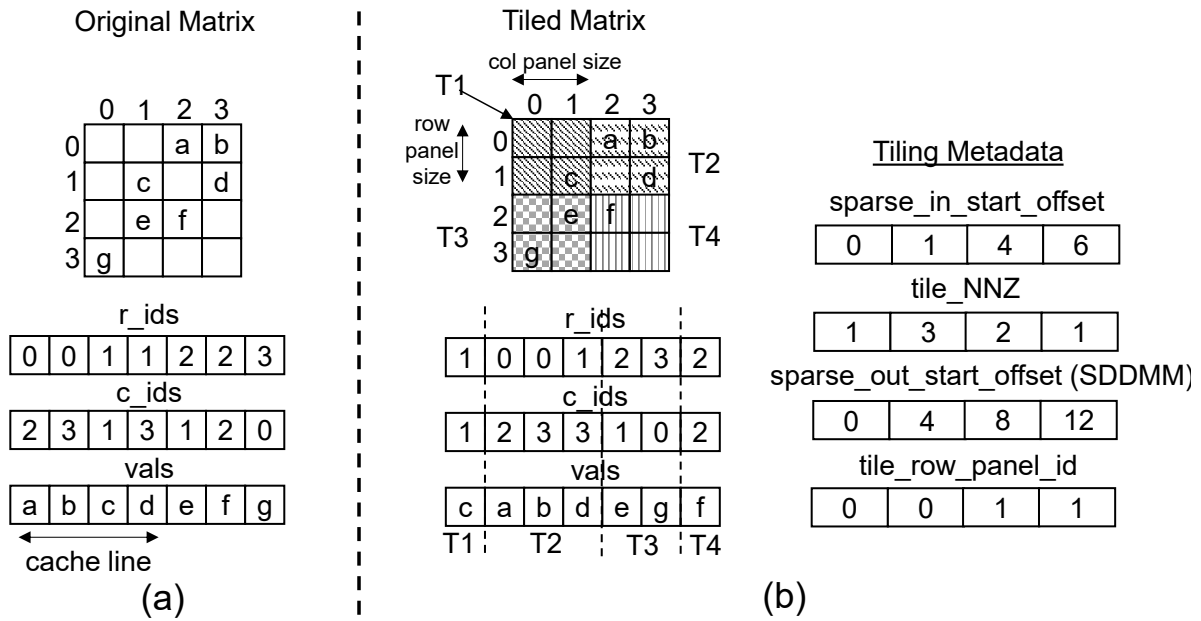


Figure 19: Representation of a tiled sparse matrix.

Chapter 4 Tightly-Coupled Accelerator Architecture and Performance Model for Compressed Large Language Models

4.1 Introduction

Large Language Models (LLMs) are an important Machine Learning (ML) workload, excelling at tasks such as chatbots, translation, text summarization, and content creation [141], [142], [143], [144]. LLMs use transformers [145] and mainly consist of multi-head attention and fully connected (FC) layers. The largest models contain trillions of parameters (weights) in the FC layers [146], [147]. During inference, the batch sizes are typically small and weights have low reuse—stressing the memory bandwidth of modern platforms [148].

GPUs are regarded as the standard platform for LLM inference because of their high compute and memory bandwidth. However, recent advances introduced by Intel Xeon 4 servers (codenamed Sapphire Rapids (SPR)) [149], make CPUs an attractive option for LLM inference. Such processors are equipped with an in-core generalized matrix multiplication (GeMM) engine called TMUL [150]. The TMUL serves the same purpose as the GPU Tensor Cores [151]. It is programmed with the AMX ISA extensions [150] to perform GeMMs on matrix tiles. The result is an order of magnitude increase in GeMM computational throughput compared to relying solely on vector SIMD units. Further, SPR servers can be equipped with High Bandwidth Memory (HBM), increasing the available memory bandwidth by 3-4 $\times$ over their DDR-based counterparts.

In SPR CPUs, we observe that, similar to GPUs [148], LLM inference is memory-bandwidth bound. The large GeMMs in the FC layers account for more than 90% of the next token generation time for Llama-2-70B [152]. Such GeMMs have low arithmetic intensity and require loading a large number of weights from main memory. To a large extent, accelerating LLM inference on CPUs means speeding-up these large GeMMs.

Compressing deep neural network (DNN) models [153], [154] with techniques such as

low-bit weight quantization [155] and sparsification/pruning [156], [157], [158] can improve GeMM performance. The amount of data that needs to be loaded from memory is reduced, leading to significant speedups in memory-bound kernels. Sadly, like systolic arrays [10] and Tensor Cores [157], the TMUL cannot handle arbitrary quantization schemes or sparse patterns. It expects well-formed dense input tiles (i.e., zero values must be included), either in BF16 [159] or INT8 format.

To benefit from both model compression and TMUL GeMM throughput, Intel has recently introduced specialized kernels in the libxsmm framework [160]. Libxsmm uses a sequence of vector instructions (AVX) to read compressed tiles from memory, de-sparsify and/or dequantize them, and feed them to the TMUL AMX unit. This cooperative processing mode involves two different computational domains (vector and matrix), each with its own instructions (AVX and AMX), and functional units (SIMD units and TMUL).

We profiled the performance of the libxsmm kernels for different quantized and sparsified workloads. Our analysis shows that, although these kernels are very effective for moderately compressed GeMM data and with the relatively low-bandwidth DDR memory, their performance degrades with HBM. This degradation cannot be explained using a traditional two-dimensional (2D) roofline performance model [161] that only considers the memory bandwidth and the TMUL throughput as bounding factors.

To understand how to effectively accelerate kernels involving three different resources (memory, matrix, and vector resources), we introduce a novel three-dimensional (3D) performance model that we call *Roof-Surface*. This model constructs a roof-surface separating achievable from non-achievable performance. It offers greater performance explainability than its traditional 2D roofline counterpart, accurately attributing the libxsmm performance limitation to the AVX vector decompression sequence. The Roof-Surface model suggests that overcoming the decompression inefficiencies would require a prohibitive scaling of CPU core resources.

To address this problem, this chapter proposes *DECA*, a new ML model decompression

accelerator tightly coupled with its CPU host. DECA offloads tile de-sparsification and dequantization from the CPU, producing ready-to-use tiles for the TMUL. DECA can be programmed to handle quantized number formats with any number of bits between 1 and 8, supports any level of unstructured sparsity, and supports group quantization [155]. The DECA microarchitecture performs decompression by utilizing a pipeline with advanced vector operations. Importantly, we use the Roof-Surface model to: (1) make decisions about the vector pipeline microarchitecture and (2) perform design space exploration and derive a well-balanced DECA design.

We also observe that if CPU cores use regular memory-mapped load/store instructions to communicate with DECA, the communication latency gets exposed and hurts performance. Consequently, we introduce a new ISA extension to CPU cores that hides the CPU-DECA communication latency by invoking DECA out-of-order. We call this extension *Tile External Preprocess and Load* (TEPL).

Our evaluation for two different low-bit quantization formats (BF8 and MXFP4) and different unstructured sparsity levels shows that DECA is very effective. In a simulated 56-core SPR with HBM, DECA accelerates the execution of compressed GeMMs by up to 4x over the optimized Intel libxsmm software kernels. In addition, by speeding-up the FC layers, DECA reduces the next-token generation time of Llama-2-70B and OPT-66B [162] by $1.6\times$ — $1.9\times$ over the software-only solution, and by $2.5\times$ — $3.3\times$ over the uncompressed baseline model.

This chapter’s contributions are:

- The *Roof-Surface* 3-D performance model that models the interaction between vector units, matrix units, and memory. Its use extends beyond CPUs and LLM decompression.
- The *DECA* tightly-coupled accelerator, designed to accelerate the de-sparsification and dequantization of compressed ML models.
- The *Tile External Preprocess & Load (TEPL)* extension that enables out-of-order invocation of *DECA*.

- A simulation-based evaluation of the performance of *DECA* for compressed GeMMs and LLM inference.

4.2 Background

4.2.1 LLM Inference

Large Language Models (LLMs) consist of different layers, such as Embedding layers, Fully-Connected (FC) layers, and Attention layers [145]. LLM inference has two phases [163]. The first one encodes the input tokens and generates the first token (prompt phase). The second one generates the next output tokens (generation phase). In this chapter, we focus on executing the low arithmetic-intensity generation phase efficiently since, for many practical use cases, it dominates the end-to-end LLM inference time [148].

GPUs are regarded as the standard platform for LLM inference [163], [164] because of their high compute and memory bandwidth. However, there has been increasing research and industrial interest in making CPUs better at machine learning (ML), by either incorporating extensions or small accelerators on the CPU die [17], [92], [122], [123], [165], [166]. In particular, recent advances such as HBM and in-core GeMM engines [149] make CPUs attractive for LLM inference. For this reason, in this chapter, we focus on LLM inference on modern CPU servers. We mainly target batch sizes from 1 to 16, which are typical in local interactive inference. Larger batch sizes in CPUs increase the inference latency without proportional gains in throughput (Chapter Supplement 4.12.1).

4.2.2 Model Compression

For low arithmetic-intensity LLM FC layers, compressing the weight matrices reduces data movement and, therefore, can directly improve performance in both GPUs and CPUs.

There are two main ways to compress an ML model [44], [158], [167]:

- **Quantization** involves storing weights in a lower-bit format, e.g., FP8 or FP4 instead of FP16. Multiple quantization schemes exist [168], [169], [170], [171]. Some of them

additionally split weights in groups and introduce a per-group scaling factor (i.e., *group quantization*) to achieve higher accuracy. In this chapter, we evaluate two types of weight quantization: BF8 (8-bit brain floating point) and MXFP4 [172]. The latter uses a 4-bit floating point and group quantization with a shared scaling factor for every 32 weights. MXFP4 has been shown to not degrade LLM accuracy [172].

- **Sparsification** consists of eliminating (i.e., *pruning*) weights that do not contribute much to the model’s accuracy [156], [173], [174]. *Unstructured sparsity* does not impose restrictions on which weights can be removed, while *structured sparsity* requires the sparsity to follow a certain pattern. The former achieves higher accuracy for the same sparsity level [175], [176]. In this chapter, we assume unstructured sparsity, and use a bitmask-based sparse format to avoid storing zeros (Section 2.1). To reconstruct the position of the nonzeros in the original weight matrix, the bitmask has as many bits as the number of elements in the matrix, where the ‘1’ bits indicate the location of the nonzeros. The nonzero values are stored consecutively in an array (nonzero array). Recently proposed LLM weight pruning methods such as SparseGPT [176] have achieved unstructured sparsity levels of up to 60-70% without significant loss in accuracy. For traditional ML models like ResNet50 [177], sparsity of up to 95% is easy to achieve [178].

Models may be both sparse and quantized [167]. Let us define the density factor d as the fraction of weights that are nonzeros. Then, starting from a dense BF16 model, a Q bit quantized model with density factor of d reduces the model size by a factor of $16/(Q \times d + 1)$, where the ‘1’ comes from the bitmask bit. We refer to this factor as the *compression factor*—as we assume that the footprint of activations is negligible. Later, to decompress a sparse and quantized model, one must read the compressed weights, the bitmasks, and the scaling factors (if group quantization is used). These three data structures are stored as separate arrays in memory.

The compression process is executed offline (e.g., after training). In this chapter, we assume an already compressed model that we want to use online for inference.

4.2.3 Matrix Extensions

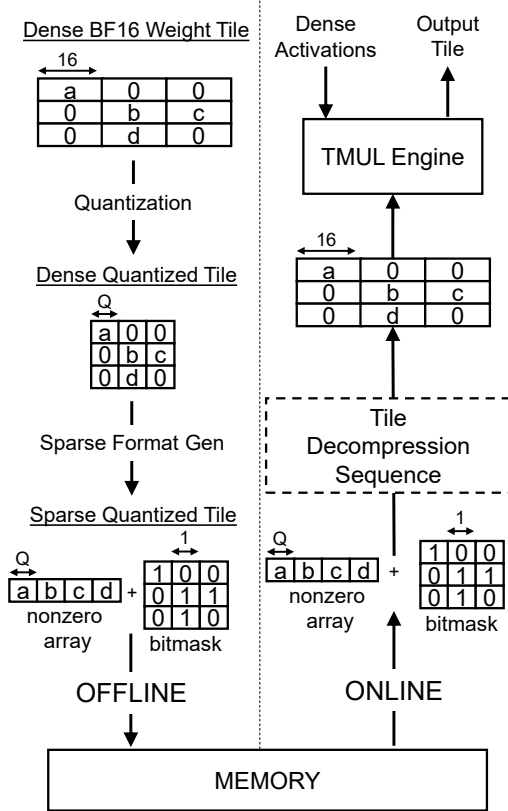
There are several hardware extensions to improve the efficiency of matrix multiplication on CPUs [179], [180], [181], [182]. In this work, we use Intel’s Advanced Matrix Extensions (AMX) [179]. AMX extends a core’s registers with 8 matrix registers, called *Tile Registers*. Each one can hold up to 16 rows, with 64 bytes of data per row that can be interpreted as 32 2-byte elements (BF16) or 64 1-byte elements (INT8). Each tile register can contain up to 1 KB of data.

Each core has the tile registers and a matrix multiplication (TMUL) unit. To load/store data to/from the tile registers, AMX includes tload/tstore instructions. The TMUL multiplies tiles in the tile registers. For example, assume that an LLM generation phase uses a batch size of $N \leq 16$ and BF16. A tile register can contain a weight tile W with $M = 16$ rows and $K = 32$ columns. Another tile register can contain an activation tile A with N rows and $K = 32$ columns. The TMUL can then perform the operation $A \times W^T$, to produce an $N \times M$ output tile. A TMUL operation can be launched every 16 cycles, regardless of the N value. It performs a total of $N \times K \times M = N \times 32 \times 16 = 512N$ fused multiply-adds (FMAs)—or equivalently $32N$ FMAs per cycle. For $N > 16$, the TMUL throughput saturates at 512 FMAs per cycle, since the activation tile can hold no more than 16 rows. Any mention of FLOPs in this chapter refers to FMAs.

4.2.4 GeMM Decompression

The TMUL does not handle sparse data and, similar to other GeMM engines [10], [151], can only handle data in very specific data formats (i.e., BF16 and INT8). If a GeMM contains compressed weights, decompression is needed to produce tiles that conform to the TMUL requirements. Unlike compression, decompression is performed online. Thus, it can impact performance. Figure 20a shows the process of offline compression of weights and their online decompression.

To partially hide the decompression overhead and attain high performance in



(a) Weight offline compression and online decompression.

```

1 .....
2 //Decompress Ti+1
3 for(r=0 to 16):
4 {
5 //Decompress
6 //row r of Ti+1
7 //using AVX VectorOps
8 ...
9 }
10
11 //GeMM Ti
12 //using AMX MatrixOps
13 TLoad Ti
14 TComp Tout, Ti
15
16 //Decompress Ti+2
17 //GeMM Ti+1
18 .....

```

(b) Libxsmm compressed GeMM kernel pseudocode.

Figure 20: Weight compression/decompression workflow and corresponding compressed GeMM kernel pseudocode.

compressed GeMM kernels, Intel recently introduced a software solution integrated in the Libxsmm framework [160]. As shown in Figure 20b, the solution involves a decompression sequence handled using AVX vector operations, and a GeMM operation executed using AMX matrix operations. Libxsmm adopts a smart method to overlap the execution of the two: software allocates a double software buffer, and tries to keep it in the L1 cache. The output of the AVX decompression sequence for tile T_{i+1} is written in one of the two software buffers. At the same time, AMX instructions load data from the other software buffer that contains T_i , which has been previously decompressed by the AVX sequence. The execution of AMX and AVX instructions overlap in the pipeline, while dependencies are naturally honored.

The decompression sequence uses vector operations such as permutes for the

decompression, and masked vector expands to insert zeros in the appropriate positions of the nonzero array. Although we omit the specifics, the first takeaway is that decompression is done using AVX, utilizing a different "domain" (i.e., separate instructions and functional units) than AMX. The second takeaway is that the AVX dynamic instructions vastly outnumber the AMX ones, since AMX uses tile-sized operands (1KB), while AVX operates on cache-line sized ones (64B, one tile row). Overall, dozens of dynamic AVX instructions are needed per AMX instruction. As a result, the decompression loop increases the total dynamic instructions of the kernel by more than an order of magnitude.

4.3 Motivation

4.3.1 GeMMs in FC Layers Dominate Inference

Table 7 shows the fraction of the "next-token" time spent in the GeMMs of the different Fully Connected (FC) layers of Llama-2-70B [152] on an SPR server with either DDR5 or HBM. We show results for an uncompressed model with BF16 weights, with different numbers of input tokens and batch sizes (N). The rest of the time is spent on kernels such as attention, for which weight compression does not apply. We see that the time spent in such GeMMs is over 95% for DDR5 and 85–90% for HBM. Hence, accelerating these GeMMs can greatly improve the next-token time.

Table 7: Contribution of FC layer GeMMs to next-token time.

Memory	DDR (260GB/s)		HBM (850GB/s)	
Input Tokens	32	128	32	128
Batch size (N): 1	97.4%	97.5%	89.8%	89.5%
4	97.3%	97.1%	89.4%	88.9%
16	96.6%	95.5%	88.3%	85.9%

4.3.2 GeMMs in FC Layers are Bandwidth Bound

We show the roofline models for one of the large GeMMs of the FC layers in Llama-2-70B for an SPR with either DDR5 (Figure 21a) or HBM (Figure 21b), and $N=16$. We use the TMUL FLOPS limit (Section 4.2.3) for the maximum achievable GeMM FLOPS in the compute-bound area. Further, when calculating the Arithmetic Intensity (AI) in FLOPs per memory byte, we only consider the footprint of the weight matrices. We neglect the footprint of the activation matrices because it is much smaller than the one of the weight matrices for small values of N . In both graphs, the leftmost circle labeled as ‘BF16’ is our baseline uncompressed execution. We see that this execution is memory-bandwidth bound in both cases due to a low AI. This motivates model compression, to reduce the amount of data that needs to be read from memory.

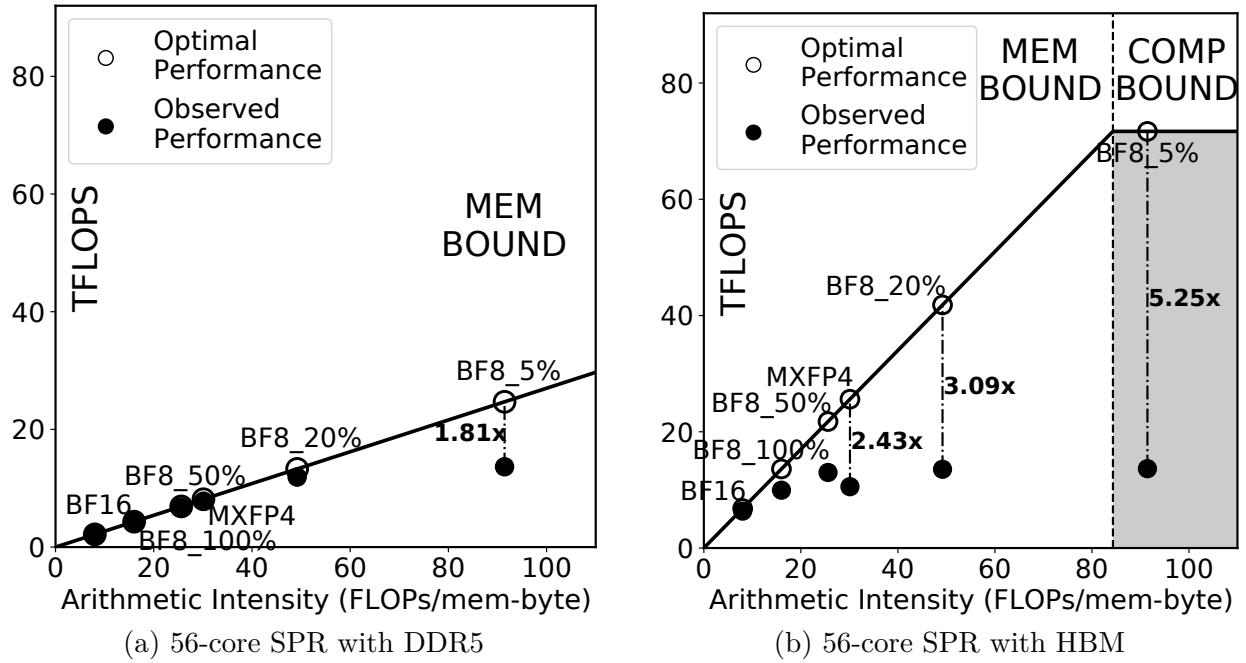


Figure 21: Traditional rooflines for a GeMM with $N = 16$.

4.3.3 Compressed GeMMs can be Inefficient

The other data points in Figure 21 represent compressed models with 4-bit quantization (MXFP4) or with 8-bit quantization (BF8) with density levels (i.e., the fraction of nonzeros) ranging from 5% to 100%. For example, BF8_20% is for BF8 and density 20%. As the compression factor increases, the amount of data fetched from memory decreases, and the AI goes up. As a result, the circles move to the right. For each design point, we show two circles: one at the *Observed* performance, and one on the roofline for the same AI. We call the latter *Optimal* performance.

We see that, as we increase the compression factor, the Observed and Optimal points increasingly diverge. In the DDR5 graph, the divergence appears at BF8_5%. However, in the HBM graph, all the compressed models are below their Optimal performance; at BF8_5%, the ratio between Optimal and Observed performance is 5.25x. This means that performance is limited by some inefficiency that is not captured by the roofline model. By manual profiling, we find that the root cause is the overhead of the AVX decompress instruction sequence. Effectively, the AVX SIMD processing units of the cores are unable to keep up with the memory bandwidth and/or the throughput of TMUL.

Given the importance of LLM workloads, some form of hardware support to minimize the decompression overhead could be justified. However, one has to be cautious when making changes in the resource-constrained CPU setting. The roofline model does not tell us the required vector throughput improvement needed for the kernels to shift from being bounded by vector processing to being bounded by memory or matrix computation. There is a danger of constructing hardware solutions that are either underprovisioned or overprovisioned. To avoid this danger, we need a new model that can guide the system design to efficiently eliminate the decompression overhead in compressed GeMMs.

4.4 The Roof-Surface Model

To guide the system design for kernels that involve matrix, vector, and memory operations, we develop a new performance model that captures their interaction. This model is called *Roof-Surface* and has a three-dimensional (3D) visualization.

4.4.1 The 3D Roof-Surface Performance Model

When multiple interacting factors can affect performance, the slowest factor ends up determining the performance. In our case, the factors are memory, vector, and matrix operations. To understand which one is the slowest, we should first express how fast (1) memory can provide compressed tiles (MEM), (2) vector hardware can process compressed tiles (VEC), and (3) matrix hardware can process decompressed tiles (MTX).

Memory can provide compressed tiles at a rate of $MBW/Bytes_{tile}$ tiles per second, where MBW is the memory bandwidth in bytes per second and $Bytes_{tile}$ is the number of bytes in a compressed tile. Since a compressed weight tile will be used for a single TMUL matrix operation, we refer to $1/Bytes_{tile}$ as the matrix-to-Memory arithmetic intensity or AI_{XM} . It expresses how many matrix operations can be executed per byte loaded from memory, and it is very similar to the traditional arithmetic intensity used in the rooflines of Figure 21. The main difference is that its units are matrix operations per byte and not FLOPs per byte. In our setting, compression schemes with higher compression factors (Section 4.2.2) have a higher AI_{XM} . Overall, the MEM rate in compressed tiles per second is $MBW * AI_{XM}$.

The Vector Hardware decompresses tiles at a rate of VOS/VO_{tile} , where VOS is the number of vector operations per second that can be executed by the architecture, and VO_{tile} is the number of vector operations needed per tile. VOS is the vector throughput and is an architecture-dependent parameter. For example, for our SPR system, it is given by the product of the processor frequency (f), the number of cores (c), and the number of

SIMD units per core. VO_{tile} is a kernel-dependent parameter. Since only the weight matrix in a GeMM needs to be decompressed, VO_{tile} effectively expresses how many vector operations are needed per matrix operation. We refer to $1/VO_{tile}$ as the matrix-to-Vector arithmetic intensity or AI_{XV} , since it expresses how many matrix operations can be executed per vector operation. Overall, the VEC rate in tiles per second is $VOS * AI_{XV}$.

The Matrix Hardware can perform MOS matrix operations per second. MOS depends on the architecture and not on the kernel. For example, in SPR systems, it is given by $f * c/16$, since each core has a TMUL that can perform a tile multiplication every 16 cycles. Overall, the MTX rate in tiles per second is simply MOS .

The Final Performance is determined by the lowest tile processing rate among the three rates considered. Specifically, the number of tiles per second (TPS) that the architecture can process is:

$$TPS = \min\{MBW * AI_{XM}, VOS * AI_{XV}, MOS\} \quad (1)$$

We can easily get the rate of FLOPs per second ($FLOPS$) by recalling from Section 4.2.3 that a TMUL tile operation corresponds to $512*N$ FMAs. Thus:

$$FLOPS = 512 * N * \min\{MBW * AI_{XM}, VOS * AI_{XV}, MOS\} \quad (2)$$

We call this equation the *Roof-Surface* equation. Any of the three terms inside the *min* clause can be the one limiting performance. For a given architecture (i.e., fixed MBW , VOS , and MOS), there are *two kernel-dependent variables* inside the *min* clause: AI_{XM} and AI_{XV} . These are the kernel’s “signature”—if two kernels have the same signature, they have the same projected performance. In contrast, in the roofline model, the kernel signature is just one variable: the traditional FLOP-to-memory AI. Now, the illustration of the performance model can no longer be done in the two dimensions of Figure 21: FLOPS as a function of FLOP-to-memory AI. Instead, we need three dimensions: FLOPS (z

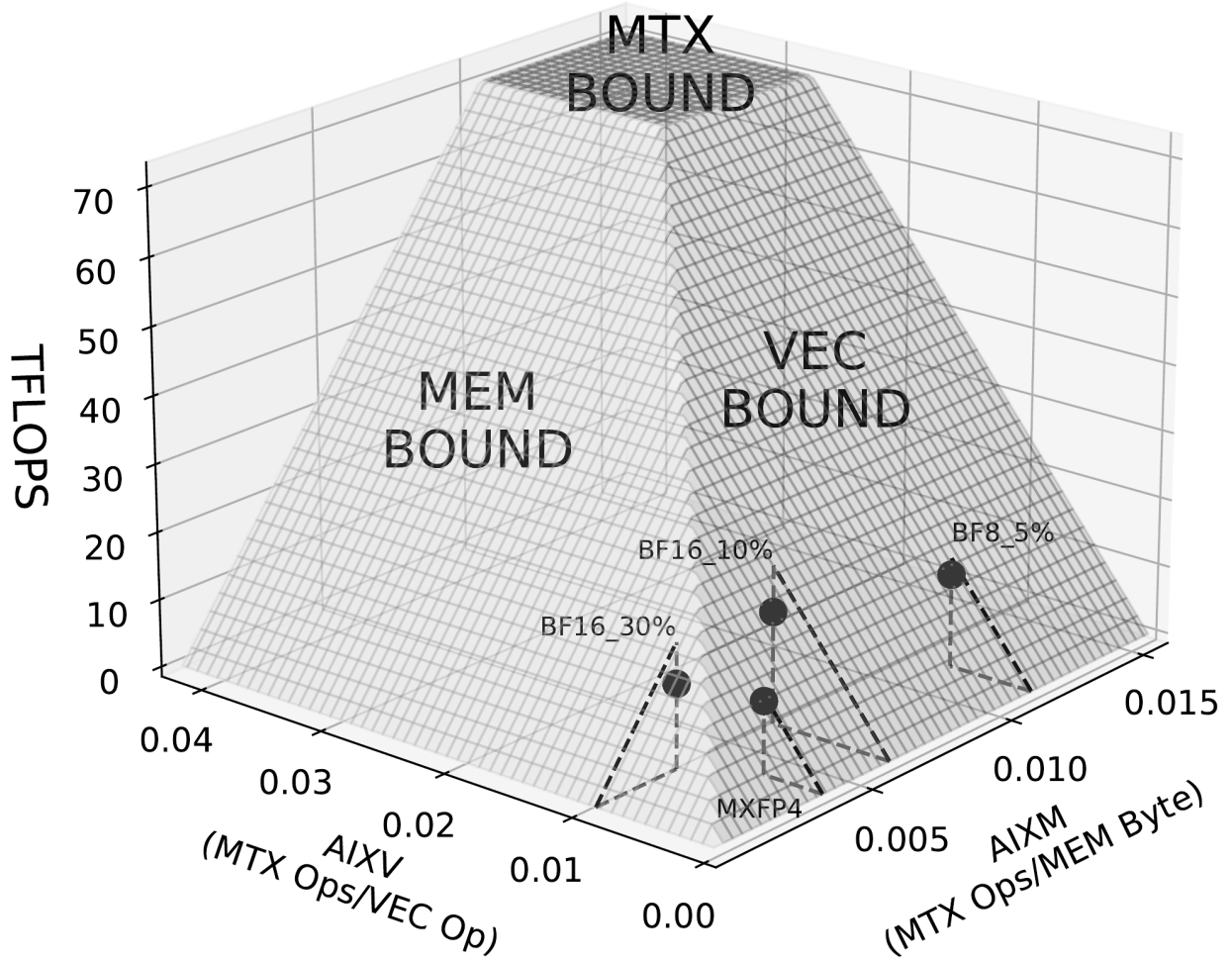


Figure 22: The 3D Roof-Surface model (N=16).

dimension) as function of the AI_{XM} (x dimension) and AI_{XV} (y dimension).

Figure 22 shows the result of plotting Equation 2 (for N=4,HBM) in three dimensions to form the *Roof-Surface* plot. A Roof-Surface plot has three regions, depicted in different colors. In each of the regions, a different term of the Roof-Surface Equation is the smallest one, and thus bounds performance. The operation points below the MTX-bound subsurface are bound by the MTX factor, the ones below the MEM-bound subsurface are bound by the MEM factor, and the ones below the VEC-bound subsurface are bound by the VEC factor. Kernel performance is depicted by points in the 3D space. The achievable performance is bounded by the overall surface, rather than by a line like in the roofline model. For this reason, we call the model Roof-Surface. Points above the overall surface

are not achievable.

Figure 22 shows four points that correspond to the observed performance of different compression schemes. We see that the MXFP4 and BF8_5% points are very near to the top of the corresponding tangent triangles (i.e., almost exactly on the VEC-bound subsurface). This visually reveals that they are bounded by vector operations. The other two points, namely BF16_30% and BF16_10%, are primarily bounded by memory bandwidth and vector operations, respectively. However, since these points are slightly below the roof surface, for these kernels, non-plotted factors such as memory latency or cache latency are also leaving some performance on the table.

Table 8: Performance limits predicted by the roofline (R-L) and Roof-Surface (R-S) models, and the real measured performance for kernels with different compressions in TFLOPS.

	MXFP4	BF8	BF8_50%	BF8_30%	BF8_20%	BF8_10%
R-L	25.2	13.3	21.2	31.2	40.8	59.2
R-S	11.5	13.3	16.1	16.1	16.1	16.1
Real	10.6	10.0	13.0	13.5	13.6	13.5

	BF8_5%	BF16_50%	BF16_30%	BF16_20%	BF16_10%	BF16_5%
R-L	70	11.8	18.4	25.2	40.8	59.2
R-S	16.1	11.8	18.4	23.0	23.0	23.0
Real	13.7	9.2	12.3	15.4	16.3	16.75

To gain further insight, Table 8 shows, for kernels with different compression schemes, the performance limits predicted by the roofline model (R-L) and the Roof-Surface model (R-S), and the real measured performance. We see that, for almost all kernels, the Roof-Surface produces relatively accurate performance bounds, while the roofline is often way off. For kernels BF8, BF16_50%, and BF16_30%, the performance bounds predicted by R-L and R-S are the same. The reason is that these kernels are classified as MEM-bound by both models.

4.4.2 The 2D Bounding Region Diagram

We now introduce a 2D representation of the Roof-Surface plot that is easier to visualize. We call it the Bounding Region Diagram (BORD). BORD is the projection of the roof

surface on the xy plane. A BORD does not depict FLOPS information, but identifies which one of the plotted factors bounds the performance of a given kernel.

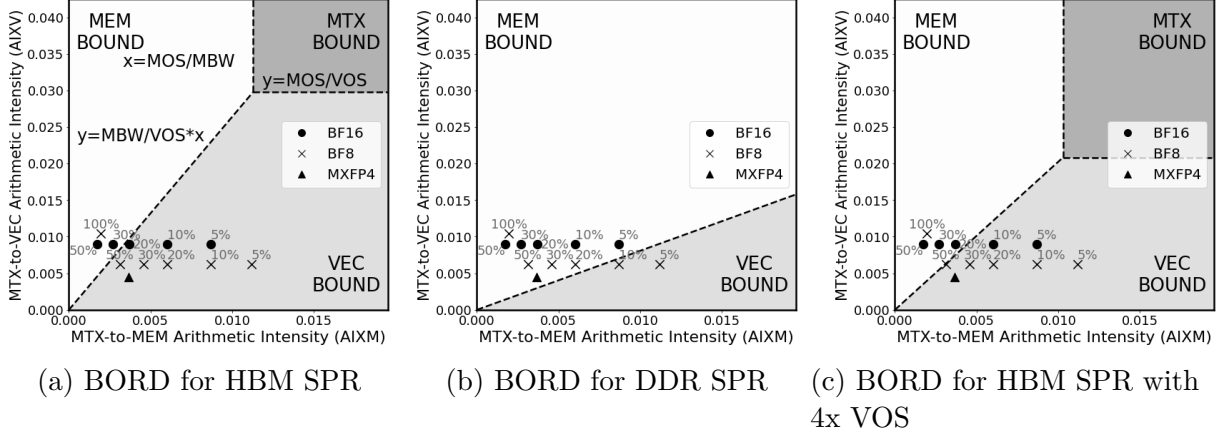


Figure 23: Bounding Region Diagrams (BORD). The batch size is $N=16$.

Figures 23a, 23b, and 23c show the BORD for some variants of the SPR architecture. In the general case, a BORD divides the plane in three regions, bound by memory, vector, or memory operations. As shown in Figure 23a, the equations of the lines separating the three regions are: $y = (MBW/VOS) * x$, $x = MOS/MBW$, and $y = MOS/VOS$.

The BORD in Figure 23a corresponds to SPR with HBM. The figure shows the positions of different compressed GeMM kernels that use BF16 or BF8 with different density levels, or MXFP4. Some of the BF8 and MXFP4 points were shown in Figure 21b. We observe that the vast majority of kernels are VEC-bound. To reach the performance of the roofline in Figure 21b, these points must be pushed away from the VEC-bound region.

Figure 23b shows the BORD for SPR with DDR, which has a smaller MBW value. Now, the area of the MEM-bound region increases. The MTX-bound region is no longer visible for the AI_{XM} and AI_{XV} value ranges plotted in the BORD. The BORD also shows that all of our kernels except BF8_10% and BF8_5% are in the MEM-bound area. This explains why most of the kernels in Figure 21a reach the roofline.

Finally, Figure 23c shows the BORD when we take the HBM SPR variant and increase the vector throughput (VOS) by 4x, in an attempt to eliminate the vector bottleneck.

When compared to Figure 23a, we see that the area of the VEC-bound region decreases and the MEM-bound region covers more kernels. However, even a 4x VOS increase is not enough to make all kernels not VEC-bound.

We find that, in the HBM SPR variant of Figure 23a, over 95% of the dynamic instructions of the cores typically perform tile decompression, and that cores are already using 40–80% of their commit slots. Hence, increasing the VOS by 4x would require not only a 4x increase in the number of SIMD AVX units, but also a prohibitive increase in the core’s superscalar width [183]. Alternatively, increasing the SIMD AVX vector width by at least 4x is also undesirable, as it requires significant pipeline changes (e.g., redesigning wider versions of all the vector instructions, adding new register files, etc.). Further, feeding the core with vectors so large would require, at a minimum, increasing the number of ports in the L1 cache. This would in turn hurt the L1 access latency and the core’s cycle time, affecting the core’s performance for general-purpose workloads. We further evaluate the limitations of traditional CPU scaling in Section 4.8.

Chapter Supplement 4.12.2 shows the effect of the batch size on the Roof-Surface, while Section 4.9 discusses how the Roof-Surface generalizes to other use-cases or architectures.

4.5 DECA Overview and TEPL Instructions

The previous analysis reveals that, to hide the decompression overheads with a conventional solution, one would need a very expensive scaling of the resources of a general-purpose core. This motivates us to propose *DECA*, a *tightly-coupled decompression accelerator for ML models*. DECA offloads vector processing for decompression from the cores. In this section, we describe DECA’s integration and then introduce a new mechanism and ISA extensions for efficiently overlapping the operation of CPU cores and DECA accelerators.

4.5.1 DECA Placement & System Integration

We envision each CPU core to have an associated DECA accelerator, as shown in Figure 24. At a high level, DECA has a processing element (PE), control registers, and tile output (*TOut*) registers. It has a memory-mapped interface that allows the core to write commands and read data. The core uses privileged stores to the control registers to configure the PE to perform decompression of tiles with a given quantization scheme and with or without sparsity. These stores also fill some look-up tables (LUTs) that DECA employs for efficient dequantization (Section 4.6).

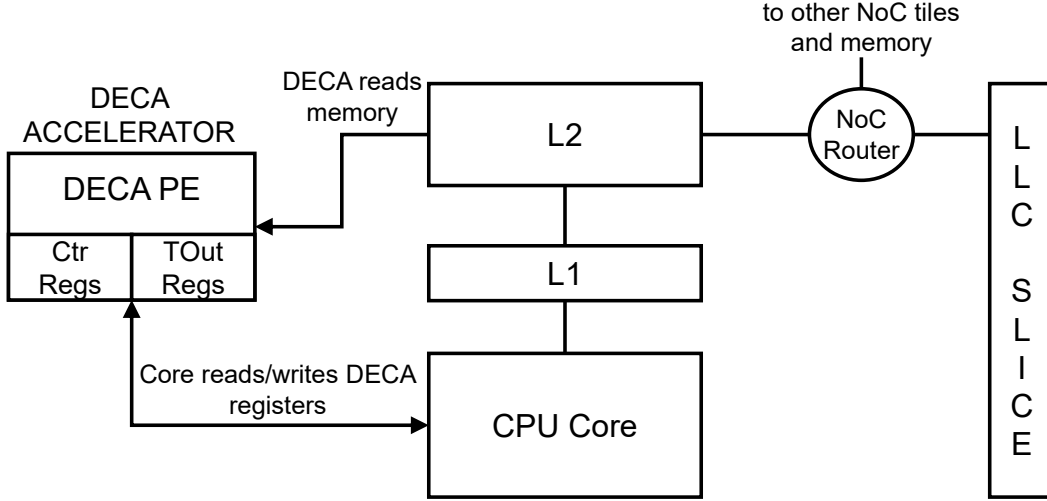


Figure 24: DECA placement next to a core.

The DECA PE reads a compressed tile from memory, processes it, and then writes the decompressed tile to the TOut registers. Then, the CPU core reads the TOut registers and uses the data to execute the GeMM using AMX instructions. The PE accesses memory through the L2, issuing both regular loads (but never stores) and prefetch requests, generated by a prefetcher integrated in the PE. DECA shares the L2 TLB with the core like prior work [92], [165] and, therefore, uses the virtual space of the CPU core.

CPU writes to the L1 propagate to DECA with the mechanism used for a 2-core cluster with a shared L2 [184]. CPU writes to L1 change the cache line's state at the inclusive L2, so if DECA tries to read a line written by the CPU, the updated copy will be moved from

the CPU’s L1, to the L2, and eventually to DECA.

DECA can potentially be used by multiple processes. One way to handle this would be to save and restore the DECA state on context switches. Alternatively, we propose that DECA retains its state across context switches and only when a new process attempts to use DECA, a trap to the OS saves the state and reconfigures DECA. Further, in the case of SMT, the OS ensures that only one thread per core has DECA access, and traps otherwise. Linear algebra kernels are typically able to saturate a core with a single hyperthread, and thus Intel advises against using SMT in combination with the TMUL [185].

4.5.2 DECA-Core Cooperative Tile Processing

To execute compressed GeMMs with high performance, we introduce a mechanism that overlaps vector operations in DECA with AMX operations in a CPU core using hardware double buffering. The design is shown in Figure 25. DECA has two Loader modules and two TOut registers. A Loader reads a compressed tile from the memory system, which includes three data structures: the data, a bitmask, and scaling factors. A Loader can also issue prefetches to load a tile in advance.

The journey of a tile involves DECA loading it into a Loader (D1 in Figure 25), decompressing it in the DECA vector pipeline (D2), and storing it in a TOut register (D3). Then, the core reads it (C1), uses it to perform the AMX operation (C2), and prompts a Loader (C3) to initiate the fetching of the next tile by passing the starting address and the length of the three data structures of the tile. As shown in the figure, the double buffers enable overlapping of the operations on two tiles. While the core is reading and processing Tile $i-1$, DECA reads, processes, and writes out Tile i . After the core finishes $i-1$, it triggers the fetching of Tile $i+1$.

We explore two options for a CPU core to communicate with DECA. The first one uses regular stores to the memory-mapped DECA interface; the second uses ISA extensions that we describe in Section 4.5.3. Using the first approach, Figure 26a shows the pseudocode of

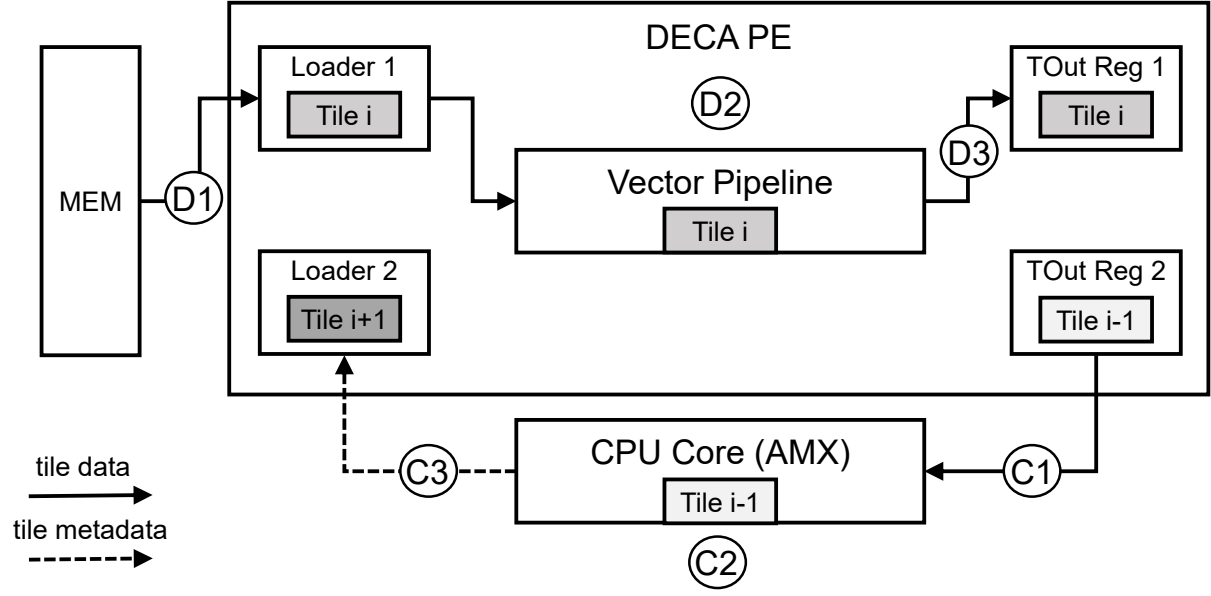


Figure 25: DECA-CPU core cooperative tile processing.

the core as it processes tiles as shown in Figure 25. The key instructions are those in Lines 4-6. The core uses *TLoad* (an AMX instruction) to load tile T_{i-1} from a DECA TOut register into a tile register $TReg_1$ (Line 4). It then uses this tile in a *TComp* instruction (an AMX instruction that performs a GeMM), saving the output in a tile register $TReg_2$ (Line 5). Finally, it writes the metadata for tile T_{i+1} (shown as M_{i+1}) to a memory-mapped register in DECA's Loader2 using a plain store. The write prompts Loader2 to initiate the fetch of tile T_{i+1} . In parallel with Lines 4-6, DECA is decompressing T_i .

```

1 .....
2  $DECA_{ldr1} \leftarrow ST\ M_i$ 
3 Fence
4  $TReg_1 \leftarrow TLoad\ T_{i-1}$ 
5  $TReg_2 \leftarrow TComp\ TReg_1$ 
6  $DECA_{ldr2} \leftarrow ST\ M_{i+1}$ 
7 Fence
8  $TReg_1 \leftarrow TLoad\ T_i$ 
9  $TReg_2 \leftarrow TComp\ TReg_1$ 
10 .....
```

(a) CPU core pseudocode for store-based DECA invocation.

```

1 .....
2  $TReg_1 \leftarrow TEPL\ M_{i-1}$ 
3  $TReg_2 \leftarrow TComp\ TReg_1$ 
4  $TReg_1 \leftarrow TEPL\ M_i$ 
5  $TReg_2 \leftarrow TComp\ TReg_1$ 
6  $TReg_1 \leftarrow TEPL\ M_{i+1}$ 
7  $TReg_2 \leftarrow TComp\ TReg_1$ 
8 .....
```

(b) CPU pseudocode for TEPL-based DECA invocation. The architectural tile registers $TReg_1$ and $TReg_2$ are renamed to different physical tile registers in each iteration.

Figure 26: CPU-side pseudocode for store-based and TEPL-based DECA invocation.

Figure 26a also shows a piece of the previous iteration (Line 2) and of the subsequent iteration (Lines 8-9). To prevent incorrect memory operation reordering, we add a memory fence per iteration. Specifically, the load of tile T_i (Line 8) should not execute before the metadata for T_i is written to the control register in DECA’s Loader1 (Line 2), which resets TOut Register 1 and initiates the tile fetch from memory. Since these two instructions do not depend on each other, we place a fence in Line 3. There is a fence in each iteration.

Unfortunately, this approach is likely to deliver limited performance for two reasons. First, each iteration has a fence that prevents cross-iteration overlap. Second, within an iteration, no instruction overlaps: the instructions in Lines 4 and 5 have a true dependence, and the store in Line 6 can only perform the update when it is at the head of the reorder buffer (ROB). The execution of all instructions is serialized, as if the core was in-order. As a result, in every iteration, the latency of the communication between core and DECA (both the load and the store) is fully exposed.

4.5.3 ISA Support for Out-of-Order Invocation

To reinstate out-of-order execution and hide the core–DECA communication, we propose a different approach that relies on an extension to the CPU AMX ISA. We call the extension *Tile External Preprocess and Load* (TEPL). The main idea is to eliminate the per-iteration fence in Figure 26a by combining, in hardware, the instructions in Lines 2 and 8 into a single instruction. This instruction updates the control register of a loader with metadata, triggering a tile fetch, and only returns to the core when DECA has decompressed the tile and stored it in a core tile register (e.g., TReg₁).

A TEPL instruction takes as arguments a core source register with the metadata for a tile, and a core destination tile register. During the execution of the instruction, the metadata in the core source register is transferred to DECA to initiate decompression and the instruction completes when the decompressed tile is received by the core destination tile register. In this design, the maximum number of TEPL instructions that can execute

concurrently is equal to the total number of DECA Loaders (i.e., two). A structural hazard prevents more TEPLs from executing, since each DECA loader is able to handle only one tile at a time.

With this design, the code in Figure 26a is rewritten as Figure 26b. Fences are removed and an iteration has only two instructions (Lines 4, 5). There are no register dependencies between the iterations because $TReg_1$ and $TReg_2$ are renamed. However, a structural hazard causes the TEPL in Line 6 to stall until one of the two previous TEPLs completes.

A context switch can only occur in between two instructions. Hence, the DECA state that needs to be saved and restored when a new process attempts to use DECA is only the DECA control registers and LUTs, and not any tile data.

To support these instructions, the core has a *TEPL Queue*, and two *TEPL execution ports*, each leading to a DECA loader. A TEPL instruction is deposited into both the ROB and the TEPL Queue. When the TEPL source register is available and there is a free TEPL execution port, the TEPL is issued to DECA.

To attain high performance, TEPLs are issued to DECA as soon as possible—they do not wait until they reach the ROB head. Hence, like a load instruction, they execute speculatively and out-of-order. Invoking DECA speculatively is always safe, as DECA does not update memory state. If the core needs to flush the pipeline (e.g., on a branch misprediction or exception) while a TEPL instruction is outstanding, the core sends a squash signal to DECA. At that point, DECA aborts any tile operation in progress, no matter the state it is in. The core may safely reissue the same TEPL. For a TEPL to commit, it needs to both (1) finish its execution on DECA and write the output to the CPU core’s physical tile register and (2) be at the head of the CPU core’s ROB.

If DECA reads a cache line and then a core updates the line, DECA does not receive an invalidation from the L2. This is not a concern because DECA uses read-only weights, which are not updated by CPU cores during LLM inference. In the uncommon case that CPU cores update data that can potentially be accessed by a TEPL, we place a fence

before the TEPL, to ensure that the TEPL reads the updated data.

Overall, this design hides the communication between the core and DECA. The core executes without fences and overlaps the operation on multiple tiles.

4.6 DECA Microarchitecture

We now describe the microarchitecture that enables DECA to sustain high decompression performance and, at the same time, support a rich set of compression schemes. For simplicity, in the rest of the chapter, we assume that DECA’s output tile is in BF16 format. DECA can be trivially configured to produce INT8 output tiles.

4.6.1 DECA Microarchitecture

Figure 27 displays the DECA PE microarchitecture. To understand it, we describe its multiple components.

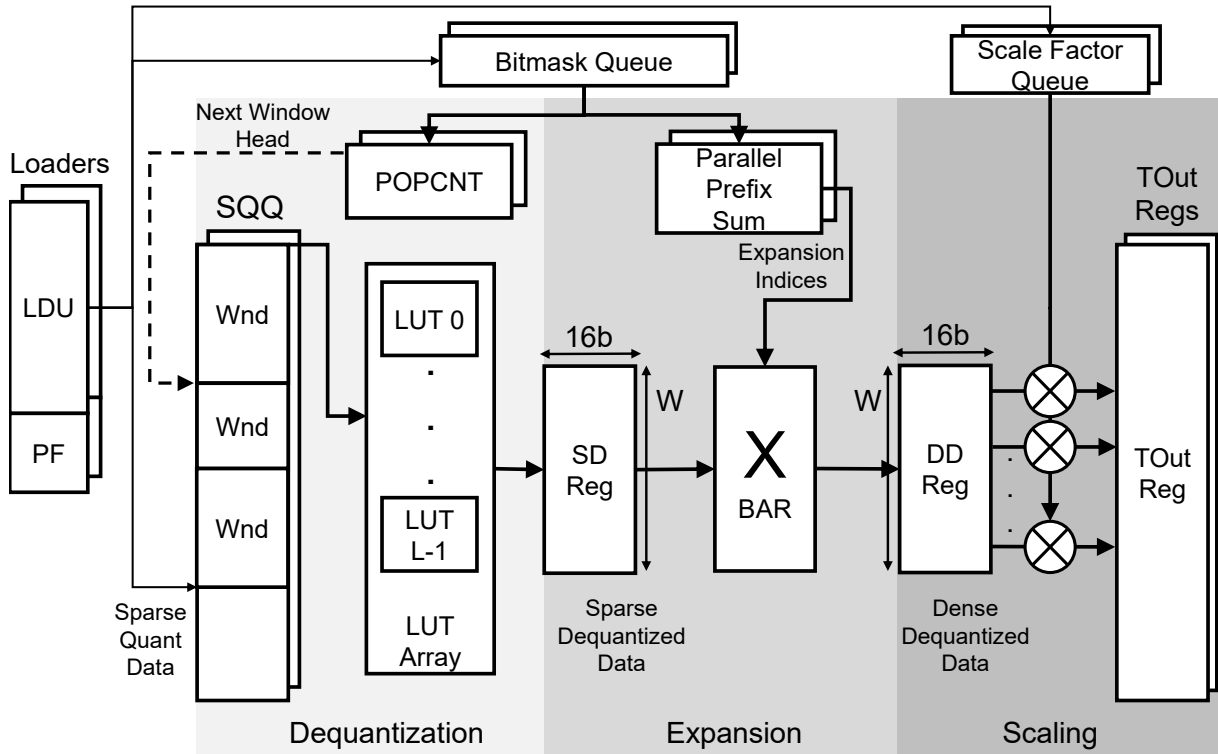


Figure 27: DECA PE microarchitecture.

1. Accessing Memory. DECA has two Loaders, each composed of a *Load Unit* (LDU) and a prefetcher. The LDU accesses memory to read a tile’s compressed weights, bitmask, and scaling factors. The base memory addresses and lengths of these structures are part of the metadata provided by the CPU on DECA invocation. When a requested cache line arrives from memory, depending on which of the three types of data it contains, it is placed in the *Sparse Quantized Queue* (SQQ), *Bitmask Queue*, or *Scale Factor Queue*.

2. Prefetcher. The prefetcher (PF) observes the address bases and lengths used for a tile, and predicts the ones for future tiles. The PF then generates prefetch requests that will bring this data to the L2 cache. Since DECA operates on virtual addresses, it uses the L2 TLB for address translation for prefetches, similar to normal DECA loads. The DECA PE does not have any prefetch buffers, since the prefetched data stays at the L2 and only moves to DECA through demand loads. Instead, the DECA PE reuses the L2 prefetch buffers/MSHRs. The existing L2 prefetcher only issues prefetches on behalf of the CPU core. The PF aggressiveness (i.e., how many tiles ahead are prefetched) is configurable.

3. Pipeline Stages. The pipeline is split into three stages, responsible for dequantization, expansion (i.e., de-sparsification) and scaling. To enable pipelining, each stage has its own output register, namely the SD, DD, and TOut registers in the figure. The Dequantization stage reads values from the SQQ, dequantizes them using an array of L Lookup Tables (*LUT Array*), and writes dequantized BF16 values to the *Sparse Dequantized* (SD) register. These values are potentially sparse—stored contiguously with zero values skipped.

The Expansion stage de-sparsifies data by inserting zeros in the positions indicated by the bitmask. This operation is done using a crossbar (*XBAR*) that is controlled using expansion indices. The latter are generated from the bitmask using the *Parallel Prefix Sum* circuitry. The result is written to the *Dense Dequantized* (DD) register, which has dense (i.e., with explicit zeros) dequantized data.

Finally, if group quantization is used, the Scaling stage applies appropriate scaling to the BF16 values by multiplying them with the scaling factors. It then writes the final

values to the $TOut$ register.

4. Duplicated Modules. A DECA PE contains two Loaders and two $TOut$ registers to enable the overlapping of DECA and CPU operation. Hence, as shown in Figure 27, the PE replicates LDU, PF, the input queues (SQQ, Bitmask queue, and Scale Factor queue), and $TOut$. One Loader can be loading data while the pipeline is processing data that was provided by the other Loader. The POPCNT and Parallel Prefix Sum circuitry process the bitmask by mainly performing additions of 1-bit data; they are also duplicated so their latency can be hidden. The rest of the pipeline is not duplicated and is used by one Loader- $TOut$ pair at a time.

5. Vector Operations (vOps). It takes multiple cycles to generate a decompressed tile, which always contains 512 BF16 elements. This is because the pipeline generates output *chunks* of W elements (e.g., 32 in our design) at a time, each using one DECA Vector Operation (vOp). In the absence of pipeline bubbles, a new chunk is generated every cycle. A vOp reads data from the SQQ, executes in the pipeline stages, and finally writes the chunk of W elements to a $TOut$. vOps exploit pipelining: when a vOp enters the Expansion stage, the next vOp can enter the Dequantization stage. The vOps are processed in order, and can enter the pipeline as long as (1) their input has arrived from memory and (2) the first pipeline stage is free.

Without sparsity, a vOp reads W elements from the SQQ. With sparsity, less than W elements are needed, since the SQQ does not contain zero values. We refer to the elements that a given vOp needs to read from the SQQ as the vOp’s window (Wnd). To determine the size of a Wnd , the POPCNT circuitry counts the number of “1s” in the bitmask, and determines the end of the current Wnd and the start of the next Wnd . The latter is the next SQQ position from which data will be read into the pipeline.

6. LUT Array Organization. The DECA dequantization stage supports quantized numbers of 8 bits or less, which can represent a maximum of 256 different values. For this reason, each of the L LUTs in the LUT array stores 256 BF16 values. Dequantizing an

8-bit value corresponds to a lookup using the 8-bit value as the LUT address to read a BF16 dequantized value. DECA contains L LUTs to allow for parallel dequantization of multiple values.

In addition, each LUT is internally divided into 4 smaller sub-LUTs, each one with a read port and 64 entries. If the quantized data bitwidth is 6 bits or less, the 4 "small" LUTs can be used in parallel to enable 4 reads from one 256-entry "big" LUT. For less than 6-bit quantization, some of the LUT entries are redundant and are not used at runtime.

7. Bubbles and the Roof-Surface. We set the number of "big" LUTs to $L < W$ to limit DECA's area. If the Wnd of a vOp is larger than L elements, the vOp occupies the Dequantization stage for more than one cycle. This injects one or more *bubbles* in the pipeline, which reduce the vOp throughput. For example, the Wnd of a dense 8-bit quantization scheme is W and, therefore, a vOp will always require W/L cycles for dequantization. Although setting $L < W$ limits the DECA throughput for dense quantization schemes, this is not a major concern. The reason is that dense schemes like BF8_100% and MXFP4 require less vector throughput (i.e., VOS) in order to escape the vector (VEC) region. This can be seen in the BORDs of Figure 23.

On the other hand, sparser schemes require a higher VOS to escape the VEC-bound region. Luckily, this is naturally achieved by the DECA pipeline: the probability that the Wnd of a vOp is larger than L decreases with sparsity. Thus, fewer bubbles are introduced for sparse schemes, naturally achieving higher throughput than their dense counterparts for the same L . The same behavior is achieved for lower bitwidth schemes because they can perform more than L reads in parallel from the LUT array.

8. Generality and Performance. DECA supports quantization formats of 8 bits and lower, group quantization, and unstructured sparsity, which cover most current and likely future model compression schemes. DECA's design is flexible, since by changing the values in its LUT array and/or using different scale factors, it enables the support for a rich set of formats without redesigning the hardware. Additionally, individual stages can be skipped if

they are unneeded (e.g., quantization without sparsity). In terms of performance, the main benefit of DECA is that it replaces multiple vector (AVX) instructions by a single vOp that performs the whole decompression—dequantization, expansion, and scaling—for W elements. The decreased vOp count *increases the AI_{XV}* (Section 4.4), moving the points away from the VEC region.

4.6.2 Quantitative Microarchitecture Design

We now show how DECA’s W and L relate to the Roof-Surface model quantitatively. Consider Equation 2. We should express how the parameters in the equation depend on W and L . Of all the parameters, only AI_{XV} depends on W and L . Indeed, VOS is fixed to $c * 1 * f$, since each of the c CPU cores has one DECA PE that completes at most one vOp per cycle and operates at the core frequency f . On the other hand, the AI_{XV} of different kernels depends on DECA’s W and L parameters. To calculate AI_{XV} , we need to add-up the number of vOps that are needed per tile plus the number of bubbles that are generated per tile.⁴

The number of vOps per tile is $\#vOps = 512/W$, since each tile has 512 elements and a single vOp produces W elements. We express the number of bubbles per tile as $\#bbl = \#vOps * bpv$, where bpv is the number of bubbles per vOp. Since bubbles can only be generated due to insufficient resources in the Dequantization stage, we use L_q to denote the maximum number of elements that DECA can dequantize in a cycle. L_q is equal to L for 8-bit quantization schemes, $2 * L$ for 7-bit, and $4 * L$ for 6-bit and below. Without sparsity, $bpv = \text{ceil}(W/L_q) - 1$. With sparsity, the bubble generation is not deterministic, as it depends on the number of nonzeros in a compressed tile. For a matrix of density d , if we assume that nonzeros are uniformly distributed, then the number of nonzeros in W consecutive matrix elements is a binomial distribution with parameters W , d . We compute

⁴Note that to calculate the AI_{XV} for decompression with a CPU core instead of with DECA, we also have to take into account CPU core bubbles (e.g., due to dependencies between AVX operations and operation latencies).

the expected number of bubbles as:

$$\begin{aligned} bpv &= \sum_{k=0}^{\frac{W}{L_q}-1} k \cdot P(kL_q < NNZ \leq (k+1)L_q) \\ &= \sum_{k=0}^{\frac{W}{L_q}-1} k \cdot [F((k+1)L_q; W, d) - F(kL_q; W, d)] \end{aligned}$$

where $F(i; W, d)$ is the binomial cumulative distribution function. Finally, the AI_{XV} is given by $1/[\#vOps * (1 + bpv)]$.

Intuitively, given a DECA architecture with a given W and L , a certain kernel introduces an average number of bubbles per decompressed tile equal to $\#vOps * bpv$. As one tunes the architecture by modifying W or L , the number of bubbles may increase or decrease, pushing the kernel in the Roof-Surface plot down or up, respectively, and moving it closer to or farther away from the VEC-bound region, respectively. Consequently, we can use the Roof-Surface model to perform an analytical Design Space Exploration (DSE). We can plot the BORDs of different (W, L) pairs and pick the design that pushes all the kernels out of the VEC-bound area at the minimum DECA hardware cost. In Section 4.8.2, we perform such a DSE.

4.7 Methodology

1. Simulation and System Parameters. To evaluate our work, we simulate a 56-core server with SPR-like parameters using an in-house simulator based on Sniper [186]. We evaluate a DDR5-based and an HBM-based memory with about 260GB/s and 850GB/s achievable memory bandwidth, respectively. We extend the simulator so each core has: (1) a DECA PE, and (2) an 8-entry TEPL Queue and ports to support TEPLs in the core pipeline. Both cores and DECA PEs run at 2.5GHz. Our baseline DECA PE is dimensioned with $W=32$ and $L=8$, but we also evaluate other options in Section 4.8.2.

2. Software and DECA Control Code Generation. As our software baseline, we use

the Intel libxsmm kernels that overlap weight decompression and GeMM execution (Section 4.2.4). To invoke DECA, we modify the libxsmm JIT compiler by replacing the AVX decompression sequence with TEPL instructions.

To evaluate the effectiveness of DECA for compressed GeMMs in isolation, we implement a large cascade of Fully Connected (FC) layers (without other types of layers) and use Parlooper [187] for loop parallelization. The weight matrices in these layers have $\approx 250\text{M}$ parameters, similar to the large FC layers of Llama-2-70B. Libxsmm and Parlooper are integrated in the Intel Tensor Processing Primitives (TPP) Framework [188], which supports end-to-end Llama-2 and OPT inference on CPUs. Hence, we use TPP as is for software-only LLM inference, and by invoking the TEPL-augmented libxsmm kernels for inference with DECA. We test batch sizes (N) of 1–64.

3. Compression Schemes. We evaluate the BF16, BF8, and MXFP4 compression schemes, which we refer to as Q16, Q8, and Q4, respectively. We limit the compression schemes to these because these are the ones supported by libxsmm. We also evaluate unstructured sparsity with weight density ranging from 50% to 5% for Q16 (only sparsity) and Q8 (quantization plus sparsity). The Q4 sparse kernels are not currently supported by our libxsmm-Parlooper-TPP stack, so we cannot directly evaluate them. For end-to-end Llama-2-70B and OPT-66B inference, the uncompressed dense Q16 baseline, Q16 with 50% density (Q16_50%), and dense Q8 do not fit in the 64GB of HBM. Hence, we simulate a larger HBM capacity for those schemes.

Table 9: Key DECA structure types, sizes, and port counts.

Structure	Type	Total Size	Entry Size	# Rd Ports	# Wr Ports	# per PE
SQQ	SRAM	2KB	64B	1	1	2
TOut Reg	SRAM	2KB	64B	1	1	2
Bitmask Queue	Regs	64B	4B	1	1	2
Scale Factor Queue	Regs	64B	2B	1	1	2
LDU	CAM	256B	8B	1 (+1 search)	1	2
Big LUT	SRAM	512B	2B	4	1	L=8
SD/DD Reg	Regs	64B	64B	1	1	1

4. Area and Power. We estimate the area and power of our DECA design with $W=32$ and $L=8$. Table 9 shows the key DECA structures. The key for the LDU CAM enters through the search port. The design also includes 64 32-bit parallel prefix adders for Parallel Prefix Sum / POPCNT, 32 BF16 multipliers for scaling, and a triangular crossbar with 32 I/O ports, where each port is 2B wide.

To estimate the area of the SRAM/CAM structures (e.g., LDU and SQQ) and registers, we use CACTI [102]. For the crossbar, BF16 multipliers, and adders, we use numbers from [189], [190], and [191], respectively. We then use [104] to scale down the numbers to 7nm. Overall, we estimate the total area for 56 DECA PEs to be $\approx 2.51 \text{ mm}^2$. Of this area, the LDUs, SQQs, Bitmask queues, Scale Factor queues, and TOut registers consume about 55%; the LUT array consumes 22%; and the rest consumes 23%. Given that the total die area of a 56-core SPR is around 1600 mm^2 [192], the DECA area overhead is less than 0.2%.

To estimate the power of the DECA PEs, we use CACTI and information from the literature. We estimate the combined dynamic power (at maximum activity) and static power to be 3.1W, or about 1% of the SPR TDP of 350W. Queues and registers contribute with about 50% of DECA’s power, the LUTs with 21%, while the rest with 29%. When idle, the DECA PEs consume less than 0.1% of the TDP.

We use CACTI to validate that all the memory structures meet timing, and [189], [190], [191] to validate that the rest of the components also do.

5. Systems Evaluated. In our evaluation, we compare several SPR systems with TMUL hardware that execute GeMMs and LLM workloads. *Baseline* is a system that loads uncompressed BF16 weights from memory. *Software-only* is a state-of-the-art system that loads compressed weights from memory and decompresses them using AVX instructions; it uses the libxsmm library. *DECA* is our system, which loads compressed weights from memory and decompresses them using DECA. Sometimes, we also use *Optimal*, an ideal system that delivers the performance predicted by the roofline model and, therefore,

assumes that all VEC overheads are hidden.

4.8 Evaluation

4.8.1 DECA for Compressed GeMMs

Figures 28 and 29 show, for different compression schemes, the speedups of the Software-only, DECA, and Optimal systems over the Baseline, running the GeMMs in our FC layer cascade for $N=1$. The compression schemes appear in increasing compression factor.

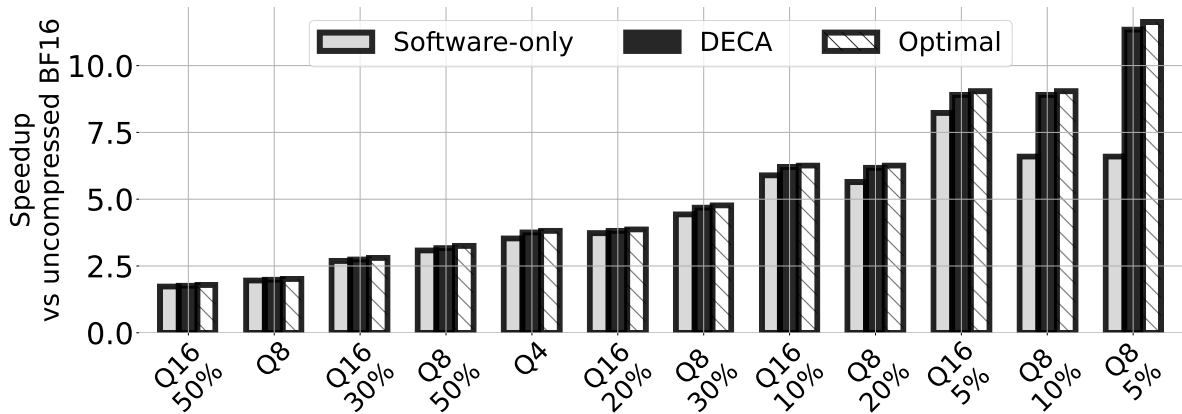


Figure 28: Compressed GeMM speedups for DDR and $N=1$.

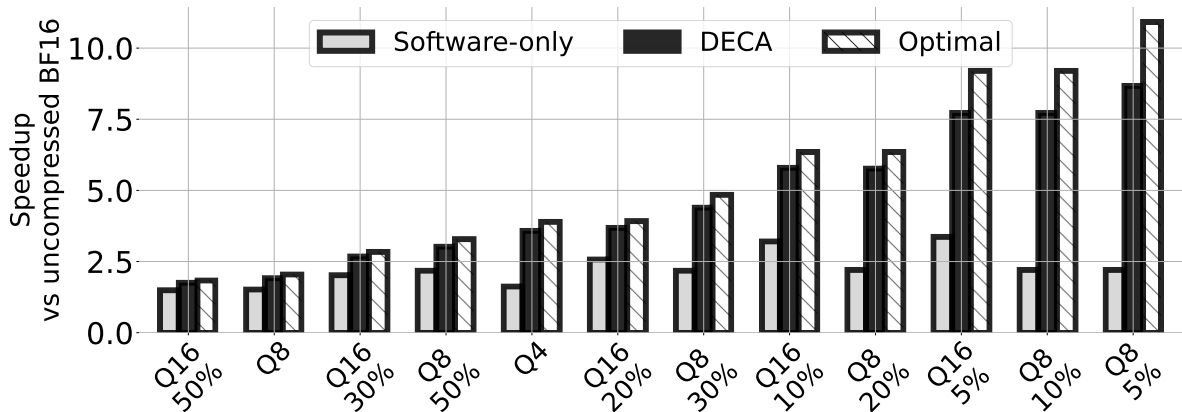


Figure 29: Compressed GeMM speedups for HBM and $N=1$.

For the DDR setting (Figure 28), DECA offers speedups over the Software-only baseline only for high compression factors. This is expected since, according to the BORD

in Figure 23b, only high compression factors are VEC-bound. The speedup of DECA over Software-only reaches 1.7 for Q8_5%. For the HBM setting (Figure 29), DECA offers speedups over Software-only for almost all the compression schemes. This is because, as shown by the BORD in Figure 23a, almost all schemes are VEC-bound. The speedups reach $4.0\times$ for Q8_5%. In addition, in both DDR and HBM, the performance of DECA is close to the Optimal one, revealing that DECA successfully hides the VEC overheads. We repeated this analysis for batch sizes of up to $N=16$ and observed similar results.

DECA-augmented cores are much more capable at vector processing than conventional cores. Figure 30 compares the performance of the DECA and Software-only systems for different numbers of cores. The figure shows data averaged across all the compression schemes for the DDR setting with $N=4$. We see that 16 DECA-augmented cores achieve higher performance than 56 conventional cores. With DECA, the extra cores can run other workloads that consume little memory bandwidth, or be power-gated.

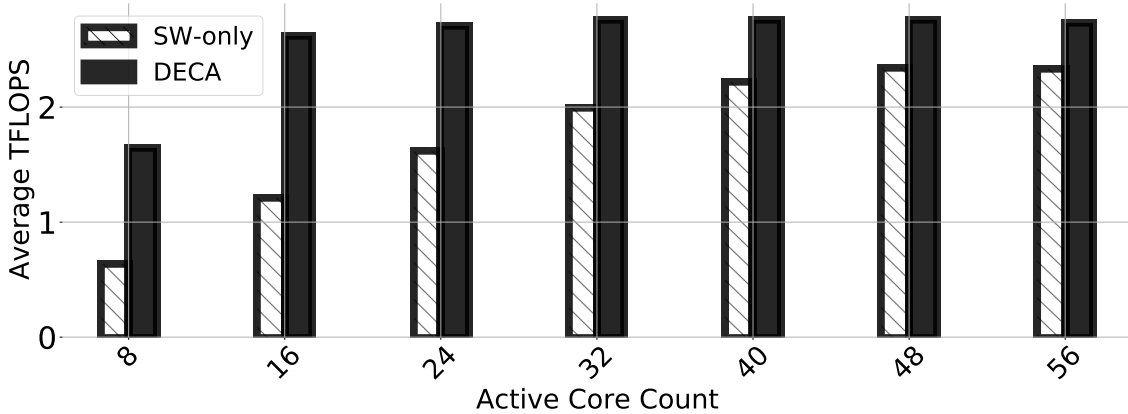


Figure 30: TFLOPS across all compressions for DDR and $N=4$.

To provide further insights into the performance of the Software-only (SW-only) and DECA systems, Table 10 displays the percent utilization of the memory bandwidth, of the TMUL, and of either the CPU’s AVX units or the DECA PE pipelines for different density factors of the weight data. Since performance is proportional to the utilization of the TMUL, the table shows that the DECA system has much higher performance than Software-only.

Table 10: Component utilization for Q8, N=1, and HBM.

	SW-only			DECA		
Density Factor	MEM	TMUL	AVX	MEM	TMUL	DECA
100%	74%	14%	50%	93%	18%	75%
50%	66%	20%	88%	92%	28%	71%
20%	35%	20%	89%	91%	53%	63%
5%	19%	20%	89%	73%	79%	87%

Since the operations of the three components overlap, the one with the highest utilization ends up being the bottleneck. In the Software-only system, for almost all of the densities, the bottleneck is the AVX vector units. This observation validates the Roof-Surface prediction. With DECA, the memory bandwidth is much better utilized, leading to direct performance improvements. As sparsity increases, the kernels take less time to execute, as shown by the higher TMUL utilization. Hence, one would expect a large increase in the DECA utilization. In reality, this is not seen. The reason is that, as explained in Section 4.6, as sparsity increases, DECA naturally suffers fewer pipeline bubbles and, therefore, increases its throughput.

Finally, we compare DECA with the alternative of scaling a CPU core’s vector resources as a method to alleviate the decompression overhead. We model cores with: (1) $4\times$ more vector AVX units (*More AVX Units*) or (2) $4\times$ wider AVX units (*Wider AVX Units*). These AVX2048 units are optimistically modeled by removing the dynamic instructions from 3 out of 4 iterations of the decompression loop. Since we do not modify the system cache line, each AVX2048 memory operation is executed as 4 cache-line sized operations. For these non-DECA systems, we do not scale the superscalar width of the core or the number of L1 ports since, as explained in Section 4.4, such changes are prohibitive. Figure 31 compares the speedups of DECA, *More AVX Units*, and *Wider AVX Units* over Baseline for different compression schemes. We see that the performance of conventional vector scaling methods is far below DECA’s.

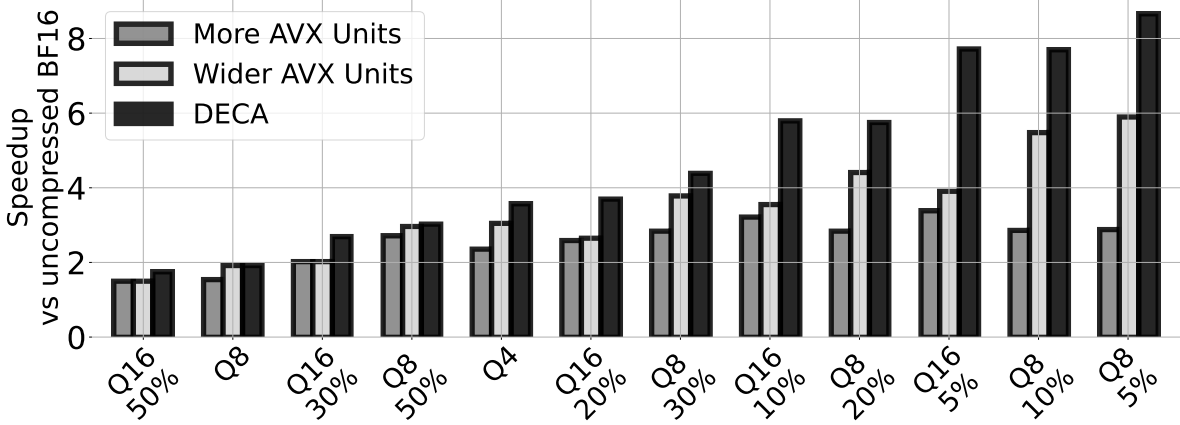


Figure 31: DECA vs traditional vector scaling for HBM & N=1.

4.8.2 Design Space Exploration with Roof-Surface

The DECA W and L parameters determine how fast DECA can decompress, but values that are too large may increase area without real benefit. Here, we use the *Roof-Surface* to examine the performance for different $\{W, L\}$ pairs. To dimension DECA, we want to pick the smallest $\{W, L\}$ pair for which the predicted performance saturates (i.e., all the kernels are predicted *not* to be VEC-bound anymore). Figure 32 shows the BORDs for the HBM SPR system without DECA (a) and with DECA (b) with three different $\{W, L\}$ sizes: $\{W=8, L=4\}$ (which we find is underprovisioned), $\{W=32, L=8\}$ (which we find is the best one), and $\{W=64, L=64\}$ (which we find is overprovisioned).

We observe that, compared to a CPU without DECA, DECA has fewer vector operations per second (VOS), since its VEC-bound region is larger. However, DECA decreases the number of vector operations needed per matrix operation (i.e., it increases AI_{XV}) as discussed in Section 4.6. The underprovisioned DECA with $\{W=8, L=4\}$ is unable to push the kernels out of the VEC-bound region. The overprovisioned one with $\{W=64, L=64\}$ pushes them out, but more than needed. The best design ($\{W=32, L=8\}$) pushes them just enough so that they are out of the VEC-bound region.

We simulate the performance of these designs to validate the model’s accuracy. We find that the DECA-best system is $2\times$ faster than the DECA-underprovisioned one. The

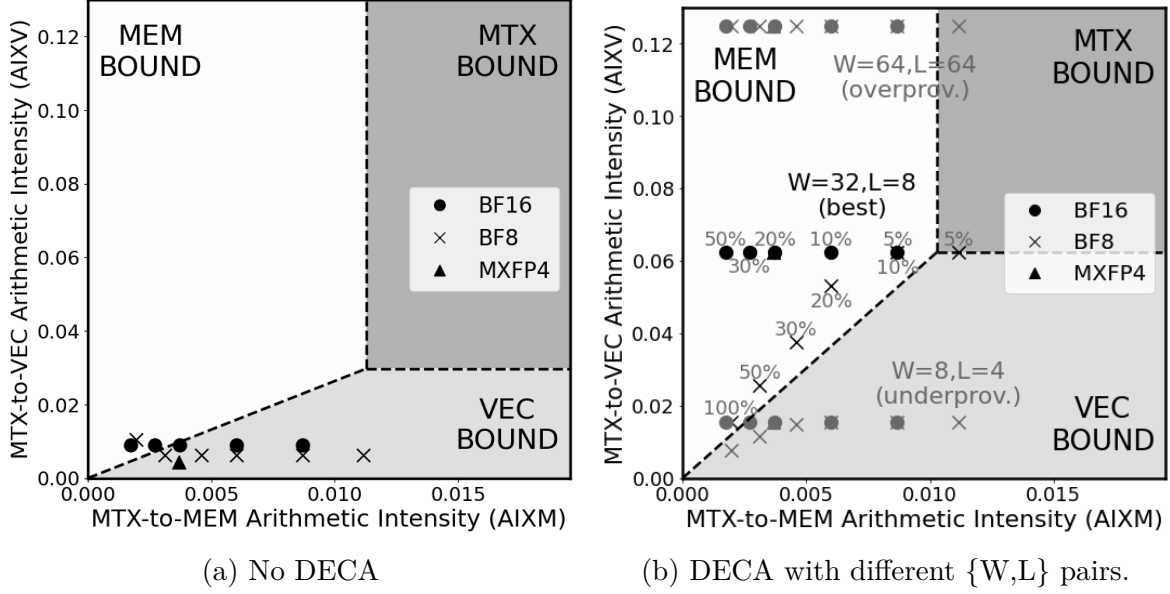


Figure 32: HBM BORDs with different-sized DECAs.

DECA-overprovisioned system is less than 3% faster than the DECA-best one. At the same time, DECA-best is much cheaper than DECA-overprovisioned: it has $8\times$ fewer LUTs and half the W . Overall, the Roof-Surface model accurately captures the dynamics of the matrix-vector-memory interaction and can guide microarchitectural decisions.

4.8.3 Analysis of DECA Integration and TEPLs

We now evaluate different decisions we made regarding DECA’s integration with a core. We start with a base configuration where DECA reads compressed tiles from the LLC (bypassing L2), writes decompressed tiles into the L2 for the core to read, and is invoked using loads, stores, and fences. Then, we progressively enhance it to: (1) allow the accelerator to read compressed tiles from the L2 and use the L2 prefetcher (*+Reads L2*), (2) use its own prefetcher instead of the L2 prefetcher (*+DECA prefetcher*), (3) write to the TOut Regs instead of to the L2 (*+TOut Regs*), and (4) use TEPL instructions as in Figure 26b instead of loads, stores, and fences (*+TEPL (DECA)*).

Figure 33 shows the speedup of each configuration over the base configuration for Q8 with different densities. We see that *+Reads L2* improves performance for all densities.

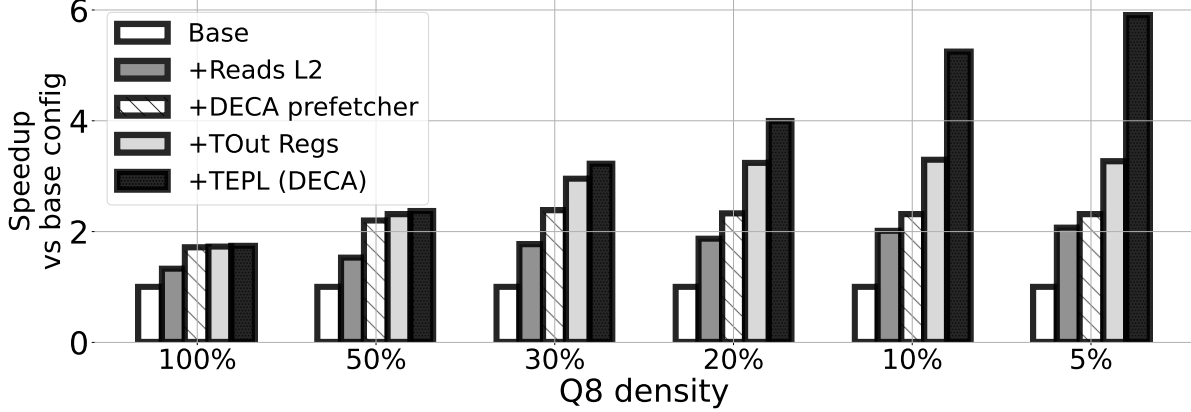


Figure 33: Speedups due to different DECA integration features for HBM and N=4.

The benefit comes from the L2 hardware prefetcher already available in the system, which fetches future tiles, hiding the memory/LLC access latencies. *+DECA prefetcher* further improves performance by using the DECA prefetcher rather than the default L2 one.

+TOut Regs and *+TEPL (DECA)* reduce or hide the DECA-core communication latency, and are needed to execute instructions speculatively and out-of-order. Specifically, *+TOut Regs* enables the core to directly fetch data from DECA, instead of taking the longer path through the L2. Further, *+TEPL (DECA)* overlaps communication with computation, hiding the former.

The effectiveness of *+TOut Regs* and *+TEPL (DECA)* increases as the density decreases. This is because DECA takes less time to process a lower density tile, while the overhead of communication with the core remains constant. Thus, for lower densities, the communication cost gets more exposed. TEPLs are very effective for low-density models: for 5% density, they double the performance.

In Table 11, we perform a sensitivity study of the DECA prefetch degree (i.e., how many tiles ahead to prefetch) for a GeMM compressed with MXFP4. We also compare to no weight prefetching, and to prefetching weights using the default L2 prefetcher. In all configurations, the L2 prefetcher also prefetches non-weight data (e.g., activations) triggered by CPU loads (*Other PFs*). When the DECA prefetcher is active, the L2 one does not prefetch weights.

Table 11: Prefetching sensitivity analysis (MXFP4,N=4,HBM).

Prefetching Scheme	L2 Miss (Million)	Weight PFs (M)	Other PFs (M)	Perf. (TFLOPS)
No Weight Prefetching	90	0	6	2.9
Default L2 Prefetcher	12	78	6	3.5
DECA Pref. Degree 1	12	79	6	3.4
DECA Pref. Degree 5	1	89	6	6.0
DECA Pref. Degree 15	2	90	5	5.4

We see that the DECA prefetcher with degree of 5 (which is the one our evaluation uses) offers great gains over both the default L2 prefetcher, and the DECA prefetcher with degree 1. On the other hand, too deep prefetches (e.g., degree 15) start to hurt performance. Also, we found that one of the key reasons for the performance gains enabled by the DECA prefetcher is that it generates prefetches with virtual addresses, which work seamlessly across physical page boundaries—in contrast to the physically-addressed prefetches issued by the default L2 prefetcher that don’t cross pages. Finally, the number of weight prefetches does not increase drastically with the degree. The reason is that the DECA prefetcher does not issue duplicate prefetches; it filters out tiles for which prefetches have already been issued.

4.8.4 DECA for LLM Inference

Lastly, we show the performance benefit of DECA for LLM next token generation (i.e., including the non-GeMM stages). Table 12 shows the next token latencies of the Llama-2-70B and OPT-66B models on SPR with HBM, for 128 input tokens, 128 output tokens, batch sizes of 1, 16, and 64, and different compression schemes. We compare Software-only (*SW-only*) with DECA. In the table, the Q16 entries do not have a DECA entry because DECA is not used with uncompressed data. Also, as explained in Section 4.7, we simulate the Q16 model in *SW-only* assuming a larger HBM size that can fit it. Indeed, an additional benefit of DECA is that it enables efficient execution of large models that only fit in the HBM compressed.

Table 12: Llama-2-70B & OPT-66B next-token latency (ms).

Model	System	N=1			N=16			N=64		
		Q16	Q4	Q8 30%	Q16	Q4	Q8 30%	Q16	Q4	Q8 30%
Llama	SW-only	192.3	124.6	98.2	211.2	139.1	116.6	452.5	441.6	423.4
Llama	DECA	-	68.3	59.6	-	82.7	75.7	-	277.1	259.8
OPT	SW-only	178.5	117.0	91.3	203.9	132.3	111.7	470.0	436.4	407.3
OPT	DECA	-	60.8	53.9	-	81.8	75.5	-	236.5	230.6

We see that DECA substantially reduces the next-token latency compared to *SW-only* for both models, and for all schemes and batch sizes. Using $N=64$ increases the next-token latency to high values, and is generally less meaningful for inference with CPUs (Chapter Supplement 4.12.1). Since $N=64$ is more compute-bound, compressed models for this batch size are less effective in comparison with $N=1$ and $N=16$. In fact, without DECA, MXFP4 is almost ineffective. Overall, across models, batch sizes, and compression schemes, DECA reduces the next-token time by $1.6\times-1.9\times$ over *SW-only*. If we only account for the more meaningful batch sizes (i.e., $N=1$ and 16), this translates into a $2.5\times-3.3\times$ reduction over the uncompressed base model (or a $1.6\times-3.3\times$ reduction if we include $N=64$). We observed similar results for shorter and longer token sequences.

Table 13: Comparing CPU, CPU plus DECA, and GPU platforms for W4A16 compression, Llama-2-70B, and $N=1$.

Metric	SPR HBM	SPR HBM+DECA	RTX 6000	A100	H100
BW (GB/s)	850	850	960	1935	2000
Max compute (TFLOPS)	72	72	364	312	989
Next-token latency (ms)	125	68	58	40	38

We consider how close the performance of a CPU with DECA gets in comparison with GPUs. Table 13 considers three GPUs (RTX 6000 ADA, A100 80GB PCIe, and H100 80GB PCIe), and lists their bandwidth, maximum compute capability, and next-token latency from the repository in [193]. The results are for 4-bit quantized weights and 16-bit activations (W4A16), Llama-2-70B, and $N=1$. We compare these numbers to results from simulations of our CPU system with and without DECA. We see that, with DECA, the SPR HBM system attains a next-token latency that is very close to RTX6000 ADA’s. In addition, it has only a 1.8x higher next-token latency than an H100 GPU, despite the

latter having 2.4x higher bandwidth and 14x higher maximum computational throughput.

4.9 Discussion

In this section, we briefly discuss two additional topics.

1. Utility of a DECA-inspired engine for other architectures. DECA is not limited to Intel CPU architectures. It can also be used in other x86 processors such as AMD’s, as well as in processors with different ISAs, such as those of ARM and RISC-V. In both the ARM and RISC-V memory models, fences are also needed to enforce $st \rightarrow ld$ ordering when using store-based DECA invocation. Fences are needed to ensure that the accelerator is first invoked before trying to read from it. Hence, TEPL-like instructions would also be useful, and we expect the design to remain largely the same.

Regarding GPUs, similar to the TMUL, GPU Tensor Cores support only limited quantization formats and do not support unstructured sparsity. Table 13 reveals that the performance of kernels with 4-bit quantized weights largely scales with the bandwidth of the GPU. This suggests that, for these kernels, GPUs are largely bottlenecked by memory and not by decompression—thanks to the high vector FLOPS of GPUs compared to CPUs.

Currently, there are multiple software systems to decompress weights for GPUs. For example, AWQ [168] includes a dequantization kernel, and Flash-LLM [157] de-sparsifies data with unstructured sparsity and feeds it to the Tensor Cores. Although effective, Flash-LLM ends-up substantially increasing the utilization of the L1 and shared memory of the SMs, preventing full Tensor Core and HBM utilization. A DECA-inspired decompression engine could reduce the L1 and shared memory pressure, and thus potentially prove useful for GPUs for kernels involving sparsity or combination of sparsity and quantization. NVIDIA recently introduced the TMA accelerator [194] for supplying data from memory to Tensor Cores. Augmenting TMA with DECA-inspired decompression capabilities is an interesting future direction.

2. Roof-Surface Generality. The Roof-Surface is applicable beyond Intel chips, CPUs,

and decompression. It applies in any scenario involving chains of matrix, vector, and memory operations. For example, beyond Intel, it can be used to model the performance of a kernel involving Scalable Vector Extensions (SVE)[195] and Scalable Matrix Extensions (SME) [180] in an ARM architecture. It can be used for non-CPU architectures that involve separate matrix/vector units such as the AWS Trainium [11], [196], the TPU [9], GPUs with their Tensor Cores and SIMT Cores, and the Tandem processor [19].

In addition, it is straightforward to extend the Roof-Surface for the general case of n pipelined abstract cooperative compute domains and memory. A general n -dimensional analytical model of the interaction of the domains can be derived as follows. First, one of the domains is chosen as the *target* (T). The final performance equation will be a limit for the operations of that domain. In this chapter, we chose the MTX as the T domain. Then, we should express the AI of the T domain with respect to (1) memory (AI_{TM}), and (2) every other domain D_i (AI_{TD_i}). AI_{TM} is $\frac{1}{Bytes_{Top}}$, where $Bytes_{Top}$ is how many bytes need to be loaded from memory to execute one operation of the T domain. The AI_{TD_i} is given by $\frac{\#Tops}{\#D_iops}$, where $\#Tops$ and $\#D_iops$ are the total number of operations in the target kernel from domains T and D_i , respectively. Then, the performance model equation in predicted $Tops/second$ or $T\hat{OS}$ is given by:

$$T\hat{OS} = \min\{MBW * AI_{TM}, D_1OS * AI_{TD_1}, ..., D_nOS * AI_{TD_n}\} \quad (3)$$

where D_iOS is the rate at which the architecture can execute operations from domain D_i . For more than two domains plus memory, visualization requires more than 3 dimensions, but the model can still be used to produce performance limits or identify limiting factors analytically. If the resulting model is 3-D, then the visualization techniques in this chapter (Roof-Surface and BORDs) apply.

Table 14: Comparison of DECA with related tightly-coupled accelerators.

Accelerator	Supports Different Quant.	Supports Struct. Sparsity	Supports Unstruct. Sparsity	GeMM Execution Units	High GeMM Tput	Fine-grained Interleaving with the Core	Core HW Changes Required
TMUL [150]	Limited	X	X	Matrix	✓	✓	N/A
TensorCore [6]	Limited	2:4	X	Matrix	✓	✓	N/A
RASA [197]	X	X	X	Matrix	✓	✓	Many
VEGETA [123]	X	2:4,1:4	X	Matrix	✓	✓	Many
SAVE [122]	X	✓	✓	Vector	X	✓	Many
SPADE (Chapter 3)	X	✓	✓	Vector	X	X	None
DECA	✓	✓	✓	Matrix	✓	✓	Few, reusable

4.10 Related work

1. Decoupled Accelerators. A variety of stand-alone decoupled accelerators that target sparsity in ML and scientific applications have been proposed [79], [80], [120], [198], [199], [200], [201], [202]. Other decoupled accelerators rely on quantization [203], [204], [205]. Recently, accelerators for attention are also becoming popular [206], [207], [208], [209], [210]. Decoupled accelerators come with large area and power budgets [123], and suffer from data movement overheads (Chapter 3).

2. CPU-integrated Accelerators. Due to the previous reasons, CPU-integrated tightly-coupled accelerators that reside in or near the CPU cores have been proposed [17], [92], [122], [123], [197]. DECA falls in this line of work. Table 14 shows a taxonomy of the tightly-coupled accelerators that are most related to DECA. The table classifies the accelerators based on whether they support (1) different quantization schemes, (2) sparsity (and what type), (3) high GeMM throughput, and (4) fine-grained interleaving with the core; whether they use matrix or vector units to execute GeMMs; and whether they require core hardware changes.

3. Tightly-coupled accelerators using matrix operations. Traditional matrix units such as TMUL [150] and RASA [197] cannot deal with compressed tiles. To avoid the need for tile decompression, some accelerator designs inside the core pipelines [6], [123], [211] including VEGETA [123] augment matrix units with support for specific structured sparsity patterns. Such an approach increases hardware complexity in the core (e.g., a

larger matrix unit, more architectural registers, or changes in register renaming). Further, although this approach can increase the matrix throughput (*MOS*) by skipping some computations with zeros, our *Roof-Surface* analysis of Section 4.4 reveals that such an increase is unneeded for our kernels: most of them become memory-bound after leaving the vector-bound region. Note that VEGETA can be paired with software that conservatively transforms unstructured to structured sparsity, as the hardware does not support unstructured sparsity natively.

Other designs augment the matrix units with native support for more efficient lower bit quantization formats [6], [205]. However, such designs require extra hardware in the matrix unit for each one of the supported formats. Further, the hardware needs to be redesigned if a new, previously unseen, quantization format emerges. In contrast, DECA can support a rich set of quantization formats without requiring extra hardware for each one: it can change the values in its LUT array and/or use different scale factors.

4. Tightly-coupled accelerators using vector operations. SPADE (Chapter 3) and SAVE [122] are accelerators for sparse applications designed to be integrated with CPUs. Instead of relying on matrix units, they use vector units to execute the actual GeMM. While this approach might work for highly sparse matrices, using the high-throughput matrix units is necessary for the moderately sparse matrices found in ML models [212].

Overall, as shown in Table 14, DECA offers a unique combination of characteristics. It supports a rich set of quantization schemes combined with structured or unstructured sparsity. It enables high GeMM throughput by cooperating with the TMUL matrix units. Further, by using the TEPL instructions for speculative invocation, DECA enables fine-grained interleaving with the core. At the same time, it introduces only a few changes to a core’s pipeline, which could potentially be reused for other accelerators.

5. Performance Models. Gables [213] is a related performance model for multiple compute domains and memory with a 2-D visualization. Gables models heterogeneous compute engines accessing memory independently and *in parallel*. In contrast, Roof-Surface

models a *serial* (pipelined) combination of multiple engines, where one engine accesses memory and produces data for a second engine. As a result, Gables only uses Arithmetic Intensity (AI) with respect to memory for each engine, while the Roof-Surface defines AI from one engine to the other (e.g., AI_{XV}). Overall, the two models target different settings and are orthogonal. Combining them into a single model is an interesting future direction.

4.11 Conclusion

To improve LLM inference in advanced CPU platforms with in-core GeMM engines and HBM, this chapter made three contributions: the *Roof-Surface* performance model, the *DECA* tightly-coupled accelerator for ML-model decompression, and the TEPL ISA extension for out-of-order accelerator invocation to hide communication latency. Our evaluation showed that DECA is very effective. In a simulated 56-core Xeon 4 server with HBM, DECA accelerated the execution of compressed GeMMs by up to 4x over the use of optimized Intel software kernels. Further, DECA reduced the next-token generation time of Llama-2-70B and OPT-66B by $1.6\times$ — $1.9\times$.

4.12 Chapter Supplement: Effect of Batch Size

4.12.1 Batch Sizes in LLM Inference with CPUs

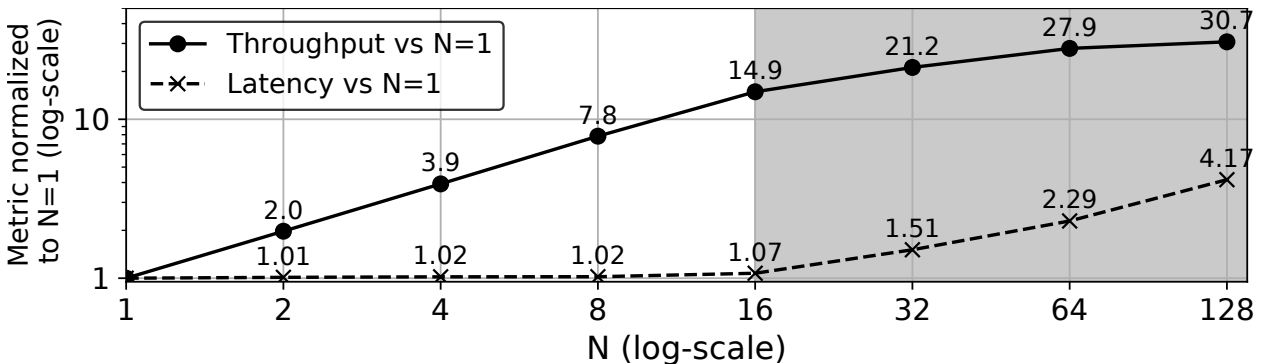


Figure 34: Llama-2-70B next-token time (latency) and tokens-per-second (throughput) scaling for different batch sizes N .

This section examines what batch sizes are meaningful for LLM inference using modern CPU servers like SPR equipped with AMX extensions. Figure 34 shows how the next-token time (latency) and tokens-per-second (throughput) of the (uncompressed) Llama-2-70B model scale with the batch size N . We see that as N scales from 1 to 16, the throughput increases linearly, while the latency remains constant. For $N > 16$, which is the TMUL throughput saturation point (Section 4.2.3), the next-token time increases without linear gains in throughput. Although hyperscalers commonly use batch sizes of 64 in LLM serving with (multiple) GPUs [214], a single CPU has utility for smaller batch sizes (e.g., for low-latency single-user scenarios). For this reason, in this work we mainly focus on sizes from 1 to 16.

4.12.2 Effect of Batch Size on the Roof-Surface

This section examines how the Roof-Surface and BORDs change as the batch size N changes from 1 to 16. Note from Equation 2 that N appears outside of the min-clause. Further, the AI_{XM} is not affected: A weight tile is only used in one matrix operation, since batch sizes from 1 to 16 can all be handled using a single matrix operation (Section 4.2.3). Although such operation has different FLOPS depending on the batch size, AI_{XM} is not affected. Finally, AI_{XV} also remains unchanged. Thus, as the batch size changes from 1 to 16, the min-clause does not change and the BORDs look exactly the same. For example, if a kernel is VEC-bound for $N=16$, it is also so for $N=1$ and vice-versa. The 3D roof-surface itself just scales in height as N increases.

As the batch size increases beyond 16, things change. Take $N=64$. First, the TMUL throughput saturates, and thus the height of the 3D roof-surface remains equal to the one for $N=16$. Second, each weight tile can now be reused in 4 matrix operations, increasing the AI_{XM} by 4. The AI_{XV} can also increase by 4 by decompressing only once per weight tile, and re-using the result for the other 3 matrix operations. The resulting BORD is shown in Figure 35. Note that the axes in the figure display a larger range of arithmetic

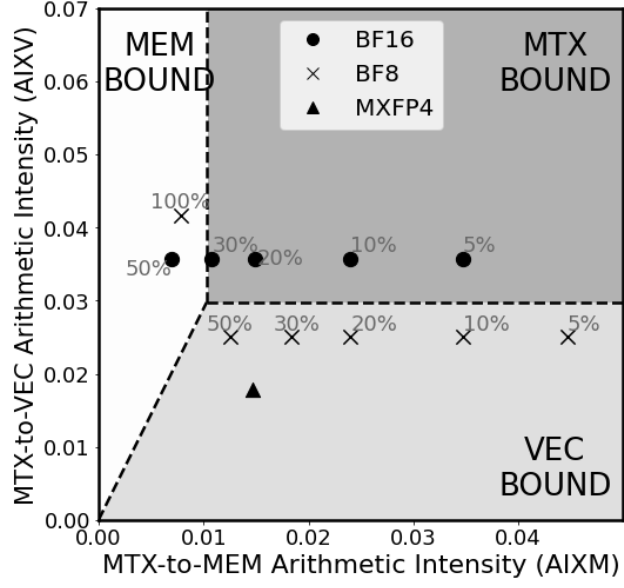


Figure 35: HBM BORD N=64.

intensities than in Figure 23a. We see that points are shifted up and right. However, many kernels still remain VEC-bound, even for this batch size that is less meaningful for low-latency CPU inference.

Chapter 5 Near-Core Accelerator Abstraction and Extensions

5.1 Introduction

For over a decade, GPUs and ASIC accelerators [9], [10], [11], [12], [196] have emerged as popular platforms that use specialized hardware designed for certain tasks. Such architectures deliver substantial improvements in domains such as machine learning [215] and scientific computing [216].

In practice, however, many applications require some computations not supported by specialized hardware, or demand more memory than available in specialized architectures. In such scenarios, programs running on GPUs or accelerators need to fall back to CPU hosts to perform such computations [19], [20], [21], [22], [23], [24]. Unfortunately, as discussed in Chapter 3, interleaving CPU and accelerator phases often carries substantial overheads (e.g., data movement) that decrease the end-to-end efficiency [24], [27]. In some cases, this overhead can be mitigated with substantial systems engineering effort and expertise [23], [24], [217]. However, such resources are typically unavailable to the average user. Hence, one would like an architecture that combines domain-specific levels of performance with the programmability and flexibility of CPUs without high coordination or communication costs.

For this reason, both industry and academia have shown interest in integrating accelerators in CPU multicores [16], [18], [92], [122], [123], [149], [211], [218], [219], [220]. Accelerators have been integrated with CPUs in two different ways. The first one integrates accelerators as advanced functional units inside the core pipeline. One example is the Intel TMUL [150] for matrix multiplication. The second approach places the accelerators outside of the core pipelines. Notable examples include the Intel DSA and IAA for data-center taxes [16], [18]. Such accelerators are decoupled from the CPU cores and are typically attached to the cache subsystem.

These two approaches differ beyond their spatial placement of the accelerators: the

interaction of the accelerators with the architectural state of the CPU cores is different. The first approach follows what we define as the *In-Core Accelerator (ICA)* abstraction. ICAs read input data from core architectural registers and write output data to core architectural registers. Further, ICA invocations are stateless: they do not leave any internal state in the accelerator after execution. An advantage of ICAs is that they can seamlessly interleave their operations with those of the CPU core at fine granularity. A shortcoming is that, as ICAs rely on the core’s general-purpose memory access hardware, they are limited by the modest memory-level parallelism that CPU cores can sustain [221].

The second integration approach follows what we call the *Out-of-Core Accelerator (OCA)* abstraction. OCAs can read their inputs from the memory system and write their outputs to the memory system. The execution of a task may leave state in the accelerator after the task completes. An advantage of OCAs is that they can access the memory system with high performance by utilizing specialized memory access hardware [92], [165]. A shortcoming is that OCAs cannot be invoked out-of-order or speculatively by CPU cores, as there is no way of undoing the effects of the accelerator execution in case of misspeculation. As we show later, this limitation exposes control and communication overheads when cores and OCAs interleave their computations at fine granularities.

To address the shortcomings of ICAs and OCAs and combine their advantages, this chapter proposes the *Near-Core Accelerator (NCA)* paradigm. Like OCAs, NCAs can directly read data from the memory system, potentially utilizing specialized memory access hardware for high performance. Like ICAs, NCAs never write to memory, but instead always write their outputs to core registers, and their invocations are stateless. This allows NCAs to be invoked speculatively and out-of-order, enabling both core-accelerator execution overlap and high-performance core-accelerator communication.

We then introduce the *Accelerator Task Extensions (ATX)* to support the interaction between a CPU core, multiple NCAs, and the cache subsystem. ATX consists of a set of instructions and hardware extensions for NCA control and communication. From the core’s

viewpoint, an ATX instruction appears as a long-latency load that eventually returns data to core registers. In the background, the instruction starts a task at an NCA that issues a set of loads, performs the required computation, and returns its outputs to core registers. ATX instructions can execute speculatively and out-of-order because NCAs never write to the memory system and are stateless.

ATX includes the *Unified Transfer Engine (UTE)*, a programmable hardware module that interfaces the CPU core with multiple NCAs, and the memory system. One of the UTE functions is to fetch and prefetch data from the memory system and move it to the NCAs with high performance. The UTE is programmed to handle diverse memory access patterns using a stream-based abstraction [222]. The second UTE function is to act as a *virtualization layer*, to schedule, overlap, and pipeline memory accesses and computation across multiple NCAs transparently to the CPU core.

We evaluate ATX NCAs through simulation of a Sapphire Rapids-like (SPR) server with 64 cores, each augmented with an NCA. The server has HBM [149], [223] with scaled-up memory bandwidth to match state-of-the-art decoupled accelerator platforms [7], [224]. We evaluate important kernels from machine learning and scientific computing that interleave CPU core and accelerator execution in a fine-grained manner. Our evaluation reveals that ATX NCAs are very effective: they deliver speedups of 2.7–4.0 $\times$, 1.3–2.3 $\times$, 1.4–3.9 $\times$, and 2.6–18 $\times$ over execution using CPU cores only, ICAs, L2-attached OCAs, and LLC-attached OCAs, respectively.

Overall, this chapter’s contributions are:

- A classification of CPU-integrated accelerators based on their interaction with the architectural state of the CPU cores.
- The Near-Core Accelerator (NCA) paradigm.
- The ATX instructions and hardware extensions for interfacing a CPU core, NCAs, and the memory system.
- A simulation-based evaluation of a CPU server with ATX NCAs for various important

kernels.

5.2 The case for Near-Core Accelerators

The integration of accelerators in CPU multicores combines domain-specific levels of performance with the programmability and flexibility of CPUs [16], [18], [92], [122], [123], [149], [211], [218], [219], [220]. In this section, we analyze two different approaches that have been used to integrate accelerators with CPUs: In-Core Accelerators (ICAs) and Out-of-Core Accelerators (OCAs). Based on the analysis, we then introduce a new abstraction for a class of accelerators that we term Near-Core Accelerators (NCAs). NCAs attempt to combine the advantages of ICAs and OCAs for applications that require fine-grained core-accelerator interleaving and high-performance data provisioning from memory.

5.2.1 In-Core Accelerators (ICAs)

One approach to integrate accelerators in CPUs is to place them in the core pipeline as advanced functional units. This is the approach followed by commercial offerings such as the Intel TMUL [150] for matrix multiplication controlled with the AMX instruction set [150], [225], or the vector units employed by most major CPU vendors [195], [226], [227], [228]. This approach is also followed by some research proposals [122], [123], [211].

These accelerators have a way of interacting with the core’s architectural state that we refer to as the *In-Core Accelerator* (ICA) abstraction. An ICA reads its inputs from core architectural registers and writes its outputs back to core architectural registers—e.g., tile registers for the TMUL. Further, ICA invocations are stateless: they do not leave any internal state in the accelerator after execution.

ICAs are attractive because they can seamlessly interleave their operations with those of the CPU core at fine granularity. Indeed, they can be invoked by the CPU core with out-of-order instructions, and execute while the core executes other instructions. This is

possible because the effects of a wrong ICA invocation can be reversed in case of a misspeculation: the wrong data the accelerator has produced is still in uncommitted register state, which can be rolled back using existing core mechanisms. Further, as the invocation is stateless, no “wrong state” is left in the accelerator that could affect future accelerator invocations. This is why Intel AVX and AMX instructions can execute speculatively. With this support, ICAs can deliver high performance in workloads that require fine-grained interleaving between accelerator and core tasks.

The main shortcoming of ICAs is that they access the memory system through the core’s general-purpose memory access hardware—e.g., the load-store queue (LSQ). This limits the amount of exploitable memory-level parallelism (MLP), and in turn the performance. Although high-performance CPU cores rely on advanced techniques to exploit MLP such as advanced prefetchers, the increased demands of the accelerators render such techniques insufficient. With insufficient MLP, the accelerated kernels can quickly become memory bound.

To assess the problem, we simulate two scenarios using the simulation infrastructure of Section 5.5. One is the 64 cores of an Intel Sapphire Rapids (SPR) server executing a Sampled Dense Matrix - Dense Matrix Multiplication (SDDMM) kernel [70], [99]. The other is the same server where cores are augmented with *perfect* ICAs that execute the computation operations of the kernel at zero cost. We call these scenarios *Core* and *+ICA*, respectively. In both cases, the architecture includes industry-grade prefetchers, validated against the actual prefetchers in SPR servers (Section 5.5).

Figure 36 shows the execution time of these two scenarios for five sparse matrices described in Section 5.5. For each matrix, the bars are normalized to *Core* and broken down into memory- and core-bound execution cycles. The former are the stall cycles due to memory system accesses. We see that, without the accelerators, the average execution is composed of $\approx 65\%$ of core-bound cycles and $\approx 35\%$ of memory-bound cycles. This justifies the need for an accelerator. With *+ICA*, computation gets accelerated and the core-bound

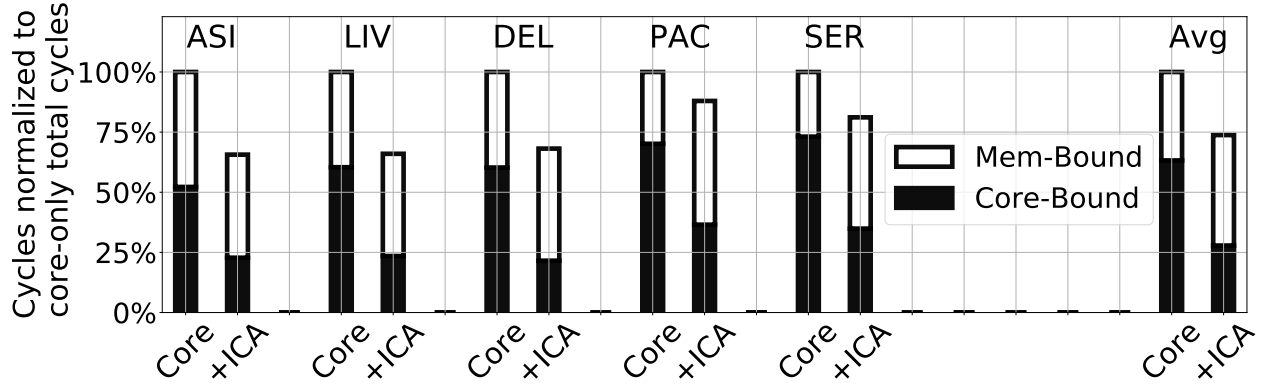


Figure 36: Breakdown of SDDMM cycles with/without an ICA.

cycles decrease by more than 55% on average. Unfortunately, the total cycles decrease by only 25% on average. The reason is that the memory-bound cycles remain or even increase. More than 62% of the total *+ICA* cycles are memory-bound. This shows that, even with the help of state-of-the-art prefetchers, the core is unable to provide data at the rate required by the ICA.

5.2.2 Out-of-Core Accelerators (OCAs)

The second approach to integrate accelerators in CPUs is to place them outside of the core pipeline [16], [18], [92], [218], [229], [230], [231]. Examples of commercial offerings following this approach are the Intel Data Streaming Accelerator (DSA) and In-memory Analytics Accelerator (IAA) [16], [18], the Intel NPU [229], and the GPU in the AMD APU [230], [231]. These accelerators are placed on the chip far away from the cores (sometimes in the corner of the die [16], [192]), and can only access the LLC and main memory. Other accelerator proposals [218] can also access private cache levels.

Such accelerators interact with the system’s state by following what we call the *Out-of-Core Accelerator* (OCA) abstraction. OCAs can read their inputs from the memory system and write their outputs to the memory system. The execution of a task may leave state in the accelerator after the task completes [232].

An advantage of OCAs is that they can access the memory system with high

performance, as they can utilize specialized out-of-core memory access hardware [92], [165]. A limitation of OCAs is that they cannot be invoked out-of-order or speculatively by CPU cores, as there is no way of undoing the effects of their execution in case of misspeculation—e.g., if an exception or branch misprediction happens. In many cases, the communication between a core and an OCA is performed with polling [18], [233], where the accelerator is treated as an MMIO device and is invoked using non-speculative CPU stores. The CPU first checks the accelerator status by reading accelerator registers and, when the accelerator is ready, starts a new task by writing to other accelerator registers. Fences are required between accelerator invocations (which are memory stores) and subsequent polling on the accelerator (which are memory loads), to prevent store→load reordering.

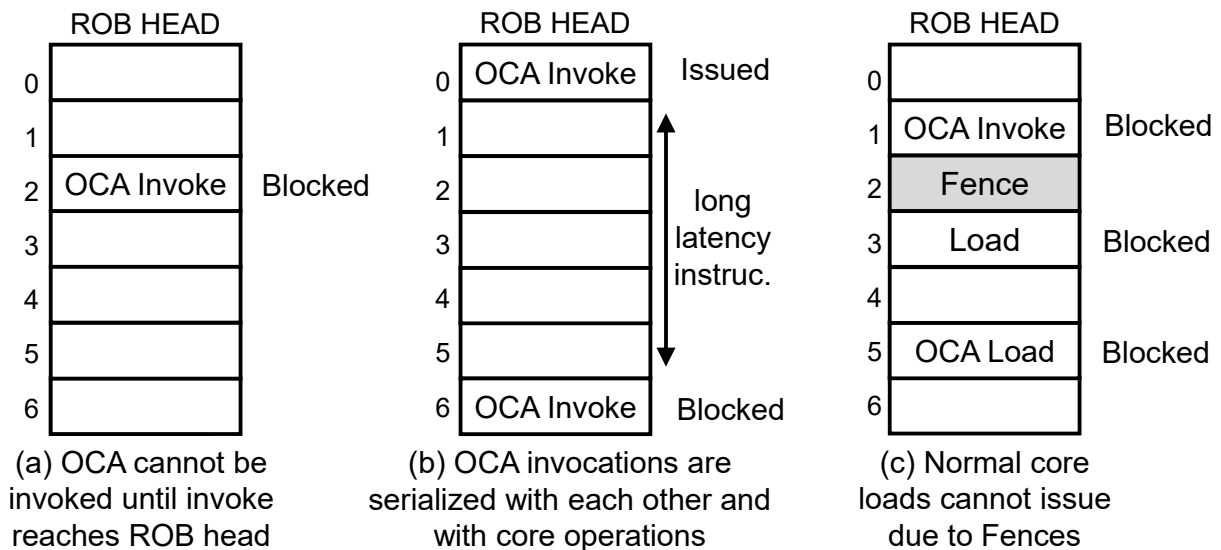


Figure 37: Examples of core ROB state when invoking an OCA.

To understand the inefficiencies of the current approach for CPU cores to communicate with OCAs, consider Figure 37, which shows three examples of the contents of a core’s reorder buffer (ROB). In the figure, the ROB head is at the top. In Figure 37(a), we show that an OCA invocation instruction (which is a regular memory store instruction) is blocked from being issued until it reaches the head of the ROB—and hence is not speculative. Figure 37(b) shows that the invocations of two OCAs cannot overlap, since

only one invocation can be at the ROB head at a time. Note that this does not mean that OCA tasks cannot overlap, but rather that the operations of starting tasks cannot overlap. Finally, Figure 37(c) shows that, for correctness, a fence is needed between an OCA invocation (i.e., a write) and a subsequent load to check the status of that OCA. Unfortunately, the fence also blocks normal (non-OCA) loads (e.g., the one at ROB index 3).

These communication inefficiencies do not matter for long tasks. However, they are detrimental to performance for small tasks that are finely interleaved with core execution. In an ideal scenario, OCA invocations would behave like normal instructions: they would execute speculatively and out-of-order when their inputs are available, and be squashed and re-executed on branch mispredictions or exceptions.

5.2.3 The Case for Near-Core Accelerators (NCAs)

Neither ICAs nor OCAs simultaneously support fine-grained CPU-accelerator execution interleaving and high-performance data access to the memory system. To fill this gap, we propose the *Near-Core Accelerator (NCA)*, a new abstraction for CPU-integrated accelerators that can be viewed as an intermediate point between ICAs and OCAs. Like OCAs, NCAs can directly read data from the memory system, potentially utilizing specialized memory access hardware for high performance. Like ICAs, NCAs never write to the memory system, but instead write their outputs to core registers, and their invocations are stateless. This allows NCAs to be invoked speculatively and out-of-order, enabling both high-performance core-accelerator communication and fine-grained core-accelerator execution interleaving.

5.3 ATX Overview and Instructions

To support NCAs, we propose a general framework that we call *Accelerator Task Extensions* (ATX). ATX consists of a set of instructions and hardware extensions to

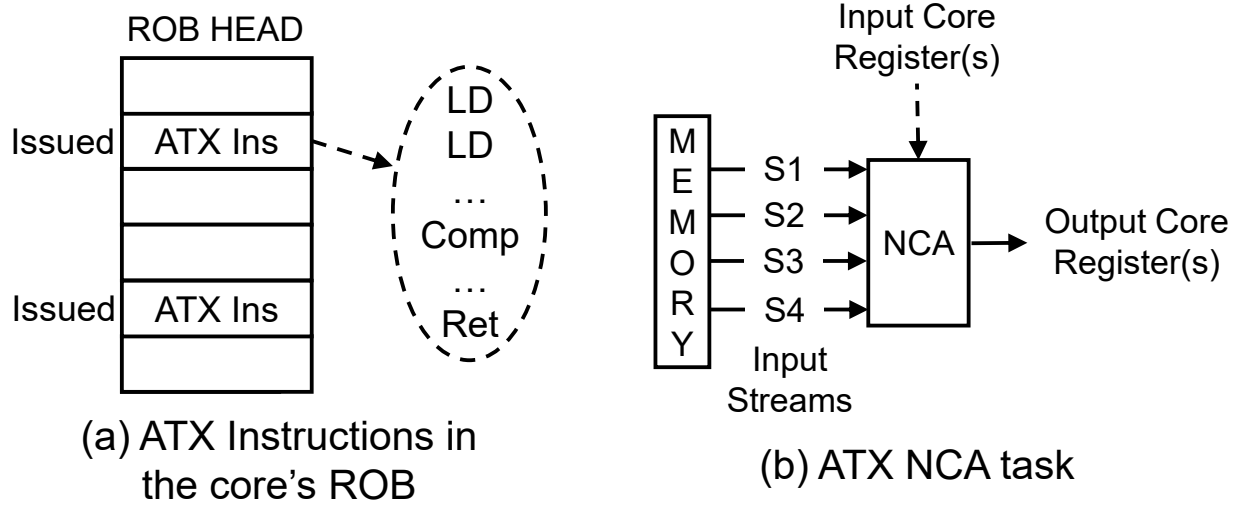


Figure 38: ATX instructions and an ATX NCA task.

support the interaction between a CPU core, multiple NCAs, and the cache subsystem. ATX unlocks the full potential of NCAs by supporting speculative and out-of-order NCA invocation, and accelerated data provision from the memory system. In this section, we first describe the high-level interaction between cores, NCAs, and the memory system. We then detail the ATX instructions, which cores use to invoke and control NCAs.

5.3.1 High-Level Overview

A single ATX instruction executed by a core invokes an NCA *task* and returns the task's output in core registers. Figure 38(a) shows ATX instructions (called *ATX Ins*). As an NCA task executes, it reads data from the memory system without core involvement, performs computation, and then returns a result. As represented in Figure 38(b), the reads from memory use configurable and potentially inter-dependent memory streams [222]. An ATX instruction includes input core register operands that carry metadata encoding all the necessary information to configure those streams. We discuss how this is done in Section 5.4. An ATX instruction also defines output core register operands to which the NCA will write output data when the task completes. An ATX instruction retires from the ROB when three conditions occur: the task has completed, the task output has been

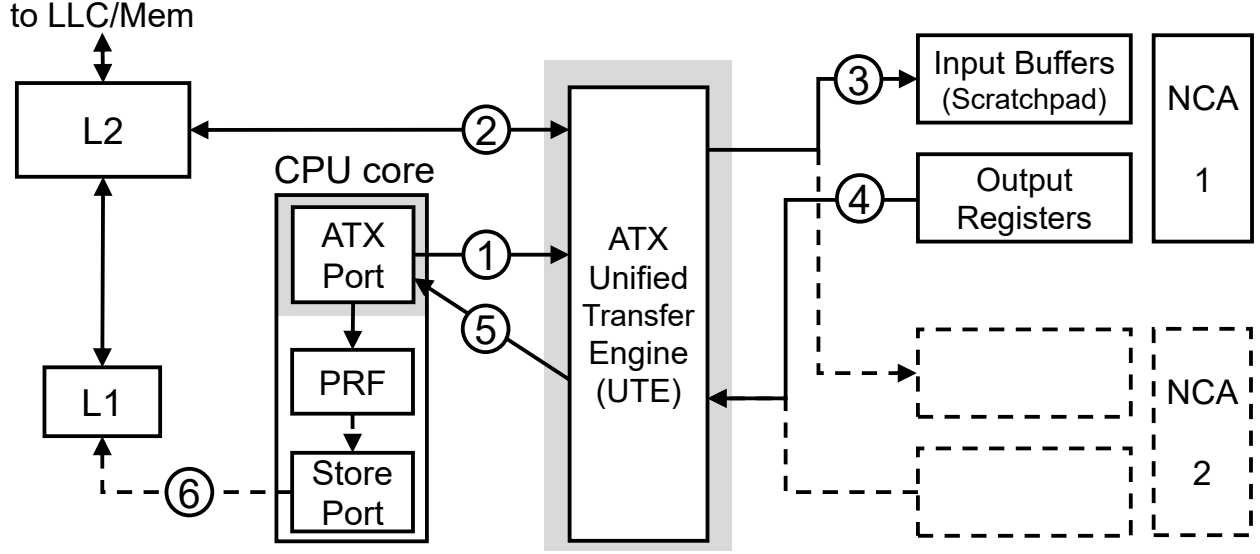


Figure 39: Interaction between a core, multiple NCAs, and the memory system.

written to core registers, and the instruction is at the head of the ROB. An NCA never writes to memory.

From a core’s viewpoint, an ATX instruction behaves like a load instruction that loads data from memory into registers. The computations performed by the task are invisible to the core. The core treats ATX instructions like normal loads; they can be issued speculatively and out-of-order, as soon as the instructions that produce their register inputs have finished. If an ATX instruction is squashed due to a wrong speculation, it is interrupted in the NCA without affecting architectural state.

Figure 39 shows a high-level overview of the interaction between a CPU core, potentially several NCAs that it controls, and the memory system. The figure includes the ATX Unified Transfer Engine (UTE), a hardware module that interfaces the three subcomponents. There is one UTE per core. The UTE schedules the tasks invoked by the CPU cores on the appropriate NCAs. Further, it reads task inputs from the memory system on behalf of the NCAs, forwards the fetched data to the NCA input buffers (i.e., scratchpads), and routes task outputs from NCAs back to the CPU. We describe the UTE in detail in Section 5.4.

While the NCA abstraction does not specify the location of the accelerator in the cache

hierarchy of a core, the ATX design that we propose places the UTE and NCAs next to the L2 cache. This is a “goldilocks zone” that enables fast core-accelerator communication, while limiting intrusion into the core pipeline. Further, such proximity enables the UTE to share the L2 TLB with the core and utilize the L2 for buffering and reusing data—minimizing the need for large scratchpads.

The interaction between core, NCAs, and memory system is as follows. At step ①, the UTE receives an ATX instruction from the core’s ATX Port for an NCA task. The UTE generates the addresses of the input data of the task, utilizing metadata in the ATX instruction, and generates read requests directed to the core’s L2 at step ②. We describe the exact mechanism of address generation later. Requests use the existing memory access flow when they encounter an L2 cache miss.

The UTE operates on virtual addresses and uses the core’s L2 TLB and MMU for memory address translation. Also, when the UTE reads the L2, it always gets the latest value of the data. If the requested line is dirty in the core’s L1, existing coherence hardware mechanisms will provide the latest version of the line from L1. The UTE never writes to the L2 or memory.

At step ③, the input data received from the L2 is written to the input buffers of the appropriate NCA. Once all the input data is collected, the UTE signals the NCA to start execution. When the NCA execution completes, the UTE is notified and, at step ④, the UTE reads the output data. At that point, the NCA is freed and can accept a new task. Then, at step ⑤, the NCA output is returned to the core through the ATX port and written to a core register (specified in the ATX instruction) in the core’s physical register file (PRF). The ATX instruction then completes. The core may use the output data for other computations or write it to the memory system (step ⑥).

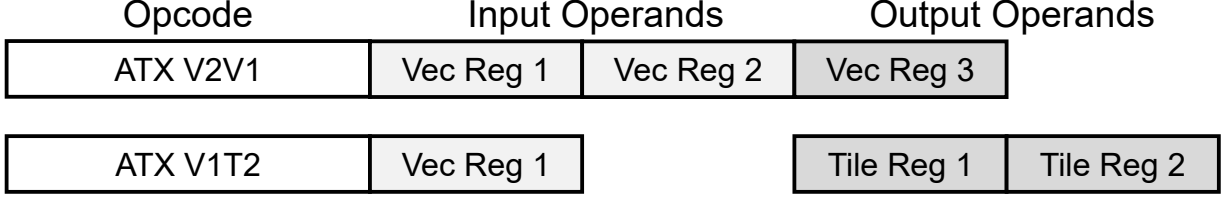


Figure 40: ATX instruction format for two example instructions.

5.3.2 ATX Instructions

The ATX instructions and the core pipeline extensions needed to support them are NCA-agnostic and reusable across different NCAs. The ATX instruction format consists of (1) an opcode that determines the number and type of input/output operands, (2) input architectural register operands, and (3) output architectural register operands. Figure 40 shows two example instructions. The opcode of the first instruction (*ATX V2V1*) means that the instruction has two input vector register operands and one output vector register operand. The opcode of the second instruction (*ATX V1T2*) means that the instruction has one input vector register operand and two output tile register operands (using AMX terminology [150]). We use ATX instructions with a different number and type of input and output register operands to accommodate the diversity in NCA tasks. The input operands carry the type of the task and information needed by the UTE to generate the input memory addresses (Section 5.4). They may additionally include control data that is transferred directly to the NCA. When the instruction completes, the NCA output is written to the output operand registers.

To support the ATX instructions, the core pipeline is extended with an ATX scheduler as shown in Figure 41. It consists of: (1) the ATX Queue, which contains information about the ATX instructions in the ROB, (2) ATX Reservation Stations that participate in the core’s wake-up logic, and (3) scheduling logic that issues ATX instructions to the ATX Port once their (renamed) input registers are ready. After the instructions are issued to the ATX Port, they are sent to an internal queue in the UTE. If this queue is full, a structural hazard prevents further ATX instructions from being issued. When an ATX instruction

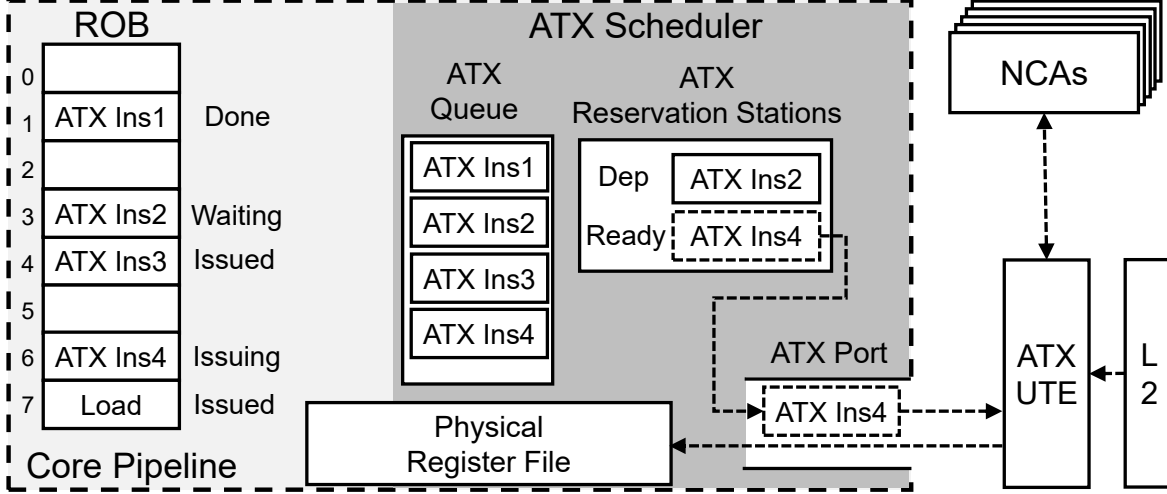


Figure 41: Pipeline extensions to support ATX instructions.

completes, the UTE returns the output to the core, which is then written to the physical register file. Note that, although in our work we have used a single port for the UTE, multiple ATX ports are possible. ATX instructions are committed in-order with the other instructions.

Figure 41 shows the status of 4 ATX instructions and a regular load in a core’s ROB. In contrast to Figure 37, ATX instructions do not have to reach the ROB head to be issued to the NCA. ATX instructions can be issued out-of-order, as long as their dependencies are resolved. Further, the need for fences is eliminated and regular loads can be issued freely. The figure shows that ATX Ins1 has completed but not retired, ATX Ins2 is in a reservation station because it depends on an input that is not yet available, ATX Ins3 has already been issued to the UTE, and ATX Ins4 is currently being issued from a reservation station to the UTE.

5.3.3 Memory Consistency Implications of ATX Instructions

ATX instructions do not affect the possible orderings between regular (non-ATX) loads and stores in a TSO-consistent system. However, the loads in a task started by an ATX instruction (i.e., *ATX Task loads*), have more relaxed memory consistency semantics. Such

loads are not tracked by the core’s load queue and, therefore, do not get squashed and replayed if a cache line accessed by the task gets evicted from the local caches or receives an invalidation. As a result, ATX Task loads appear weakly-ordered with respect to other ATX Task loads and with respect to normal core loads. This behavior is exposed as part of the ATX memory model, such that programmers and compiler developers can add appropriate synchronization code if stronger ordering is needed.

A second issue with ATX instructions relates to store-to-load forwarding. An ATX Task load cannot read data from stores currently residing in the core’s store queue or store buffer. The reason is that the load is issued by the UTE, and the UTE does not have access to the core’s store queue or store buffer. The load always reads data directly from the L2 cache. To ensure that the ATX Task loads do not read stale data, the programmer or compiler must add a fence between a store and a subsequent ATX instruction that could start a task that consumes data produced by the store. This situation should be avoided, as it cancels out the benefit of speculative execution of ATX instructions. Hence, during the execution of a kernel, any data that the core generates for an NCA should preferably be communicated through input registers rather than through memory. Communication through registers is handled correctly by the existing dependency checking mechanism in the core without needing fences.

5.4 The Unified Transfer Engine (UTE)

This section details the ATX Unified Transfer Engine (UTE).

5.4.1 Main Idea

The UTE is an out-of-core module that interfaces the NCAs, memory system, and the core (Figure 39). It serves two main goals: core-NCA interface virtualization and accelerated data provision for NCAs.

1. Core-NCA Interface Virtualization. The UTE acts as an interface between the

core and the different NCAs to avoid making changes to the core’s pipeline or adding core ports each time a new NCA is added. The core sends ATX instructions to different NCAs indirectly, by issuing them to a single UTE port. When the core invokes an NCA task, it does not need to track the status of the NCA that will execute it. In addition, the core does not need to know how many different NCA instances can execute the task. Each different task type has an identifier that we call *Virtual Accelerator (VAcc) Id*. The first input operand of an ATX instruction includes, among other metadata, the VAcc Id of the task type that the instruction invokes. The UTE has a mapping between VAcc Ids and physical NCAs, each given by a *Physical Accelerator (PAcc) Id*. The UTE is responsible for assigning and scheduling tasks to the appropriate physical NCAs. The UTE also routes output data from NCAs to the core’s ATX port(s). The data is then written to the core’s register file.

2. Accelerated Data Provision for NCAs. In Section 5.2 and Figure 36, we quantified the inefficiency of using a core’s general-purpose memory access hardware to provide data to accelerators. To eliminate this inefficiency, the UTE acts as an out-of-core interface between the NCAs and the memory system. The UTE fetches and prefetches data for the NCAs, and then writes it to NCA input buffers (i.e., scratchpads). To be able to handle diverse NCA memory access patterns, the UTE is programmed using a *stream-based* abstraction, largely inspired by [222]. In our work, *streaming* is the process of reading memory elements of a specified size from a start to an end virtual address, possibly with a stride. A task started by an ATX instruction needs at least one stream for each of the input data structures participating in the NCA computation. For indirect memory accesses, more than one inter-dependent streams may be needed to realize the necessary fetch pattern of a data structure.

3. UTE Architecture. Figure 42 shows the architecture of the UTE. Before execution begins, the programmer uses regular writes to UTE configuration registers to configure the UTE with: (1) the mapping from VAcc Ids to PAcc Ids (i.e., which physical NCA instances

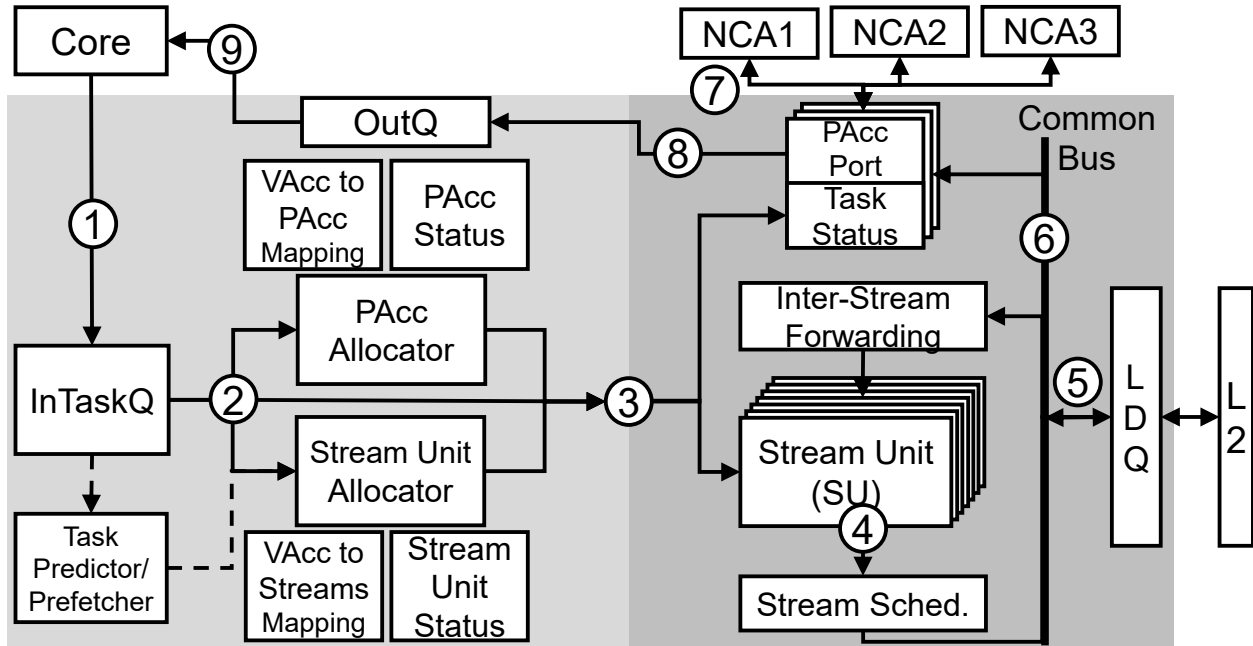


Figure 42: Unified Transfer Engine (UTE) architecture.

can support which task types), and (2) the stream information for each of the VAcc Ids. The latter includes the number of streams that the task type reads, their dependencies (if any), and other metadata for their configuration. The UTE configuration information is stored in two content-addressable memory (CAM) structures: the *VAcc to PAcc Mapping* and the *VAcc to Streams Mapping* (Figure 42). Since the size of these structures is limited, the programmer can remove entries when a task type is no longer needed—again using regular writes to UTE configuration registers. If the core tries to configure more task types than what the CAMs can hold, the UTE signals the core and an exception occurs.

The UTE CAM contents are part of the process state. To reduce context-switch overhead, this state is lazily saved and restored by the OS only when a new process attempts to use the UTE (and not at every context switch), via a trap mechanism.

In the following, we give details on the UTE microarchitecture, including the frontend, the backend, the stream units, and the task prefetcher. We also show an example of how the UTE is programmed.

5.4.2 UTE Frontend Microarchitecture

The frontend is shown in light shade in Figure 42. To understand its operation, consider the journey of a task. First, the ATX instruction that starts the task is placed in the InTaskQ at step ①. It waits there until it is allocated a physical accelerator instance (PAcc), and a set of *Stream Units* at step ②. To allocate a PAcc, the *PAcc Allocator* checks the *VAcc to PAcc Mapping* module to see which PAccs are capable of executing the task. To allocate Stream Units, the *Stream Unit Allocator* checks the *VAcc to Streams Mapping* module to determine how many Stream Units are needed for the task. The Stream Units are responsible for generating the memory addresses to load data from, and the accelerator’s scratchpad addresses to write the loaded data to. The VAcc to Streams Mapping module can optionally contain parent-child dependencies between streams. Once the appropriate PAcc and Stream Units are identified as free, the task is dispatched to the backend at step ③.

Tasks do not need to be dispatched to the backend in order. If the task at the head of the InTaskQ cannot find the necessary resources, the next task in the InTaskQ is processed. If the architecture does not include a PAcc that can handle the task (or if the VAcc to PAcc mapping has not been configured), the UTE signals the core and the ATX instruction causes an exception handled by the OS. We describe the Task Predictor/Prefetcher of the UTE frontend in Section 5.4.6.

5.4.3 UTE Backend Microarchitecture

The backend is shown in dark shade in Figure 42. It contains a number of Stream Units for address generation, and a *Load Queue (LDQ)* to load data from memory. Since different Stream Units can be concurrently active, a *Stream Scheduler* (step ④) selects which Stream Unit will issue a request to the memory subsystem (step ⑤) at each cycle. We use a simple age-based scheduling policy: the oldest streams issue first, while ties are resolved in a round-robin fashion.

The backend also contains one port per NCA (*PAcc Port*), which is used to write and read data to/from the NCA. Each port has a *Task Status* flag to track the status of the task. When data arrives from memory, it is sent to a *Common Bus* (step ⑥) that connects to the PAcc ports, and is forwarded to the appropriate NCA (step ⑦). Further, data that arrives from memory for a parent stream is also forwarded through the Common Bus to the Stream Units that implement its children streams. The Task Status is updated every time the complete data for a stream arrives from memory and is written to the NCA. When the data for all the streams of a task has arrived, the NCA is notified to start processing. Once processing is done, the NCA’s output is moved to the *OutQ* (step ⑧), and the task completes, freeing-up the PAcc Port and Stream Units, and updating the PAcc Status and Stream Unit Status modules. The output is written to registers of the CPU core at step ⑨.

The core signals the UTE every time that an ATX instruction is squashed because of a branch misprediction or an exception. Then, the UTE interrupts the execution of the corresponding NCA, invalidates any data already loaded into the NCA’s input buffers, frees-up the corresponding PAcc Port and Stream Units, and updates the PAcc Status and Stream Unit Status modules. Since NCAs are stateless, no state is preserved across NCA invocations, and the NCA is ready to accept a new task.

5.4.4 Streams and UTE Stream Unit Microarchitecture

The UTE supports a rich set of NCA access patterns using the Stream Units. A Stream Unit generates: (1) memory addresses to load from, and (2) NCA scratchpad (i.e. input buffer) addresses to write to. We now describe only memory address generation, as scratchpad address generation is similar. It is the programmer’s responsibility to size a task so that: (1) the data loaded from Stream Units does not overflow the NCA’s scratchpad, and (2) the output data of the NCA fits in the output register operands of the ATX instruction.

A stream follows the fetch pattern of Figure 43. In the inner loop, data is loaded from


```

1  while(1)
2      setup beg,end; // Start of a stream repetition
3      for(addr=beg; addr < end; addr+=size*stride)
4          load *addr; // Stream iteration
5      if (no_parent_stream or parent_stream_done) terminate;

```

Figure 43: Memory access pattern of a stream.

memory starting at address *beg* and ending at address *end*, with a specified element *size* and *stride*. A stream may have a parent stream. In this case, each iteration of the parent stream causes a new repetition in each of its children streams. A repetition is the full execution of the inner loop in the figure. Each stream repetition may have different *bounds* (i.e., *beg* and *end* addresses). Each stream occupies one Stream Unit in the UTE.

To illustrate the operation of a Stream Unit, consider the example of a task that takes a set of rows of a sparse matrix stored in CSR format. For each row in the set, the task computes the sum of its elements, and then stores the sum in a per-row buffer. The pseudocode of the task is shown in Figure 44. The task uses two streams. Stream *S1* loads the row pointers stored in array *row_ptrs*, and stream *S2* loads the element values stored in array *vals*. The number of rows is *r_end* - *r_start*. For each row, the task reads the pointers to the boundary elements in the row (*edge_start* and *edge_end*), and then reads the values of the elements in the row and accumulates them into a per-row buffer in the NCA. After the task is done, the NCA output goes to the OutQ, and the UTE moves the OutQ contents to CPU core register(s).

In the example, before a repetition of the inner loop can execute and fetch a row of data from *S2*, *S1* must have brought the two elements that mark the bounds of the row. This example shows that fetch patterns such as indirection and pointer chasing can be implemented by making the bounds parameters of a stream (*S2*) depend on the values returned by another stream (*S1*), forming a parent-child dependency tree.

Figure 45 shows the information needed to generate memory addresses for the task. Specifically, Figure 45(a) shows the values contained in the input operands of the ATX instruction that are needed for address generation by the Stream Units. We call these

```

1  for (r = r_start; r < r_end; r++)
2      load edge_start = row_ptrs[r]; // stream S1
3      load edge_end = row_ptrs[r+1]; // stream S1
4      for (e = edge_start; e < edge_end; e++)
5          load val = vals[e]; // stream S2
6          NCA accumulates val into a per-row buffer
7      UTE moves OutQ contents to CPU core register(s)

```

Figure 44: Pseudocode of an example task.

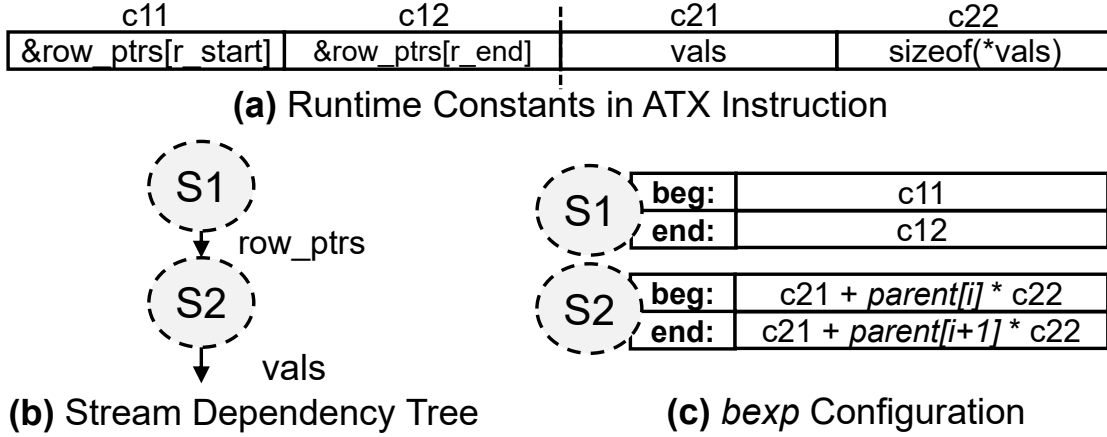


Figure 45: Information needed for task address generation.

values *runtime constants*. The constants for $S1$ are $c11$ and $c12$: pointers to the beginning and end of the *row_ptrs* array. The ones for $S2$ are $c21$ and $c22$: the address of the *vals* array and the size of an element in *vals*. Figure 45(b) shows the parent-child dependency tree for the streams in our example. $S2$ uses the row pointers fetched by $S1$ as begin/end indices in the *val* array, so $S2$ is a child of $S1$. Every iteration of the parent stream $S1$ starts a repetition of the child stream $S2$, each with different bounds.

To calculate the bounds of each repetition of a stream at runtime, the UTE contains bound expressions (*bexps*) that are set at task configuration time, and propagated to the Stream Units at task execution time. Figure 45(c) shows the *bexps* for our example. $S1$ has bounds directly given by the ATX instruction runtime constants $c11$ and $c12$ (Figure 45(a)). $S2$ has bounds that use runtime constants $c21$ and $c22$, but that also depend on the data fetched by its parent $S1$ (indicated by *parent*[]) and the stream repetition index i . In general, *bexps* take as inputs: (1) runtime constants from the ATX

instructions, (2) data loaded by parent streams, and (3) the index of the stream repetition. Although we omit details due to space, a *bexp* has the form $Op1(I_1, Op2(I_2, I_3))$, where I_i are inputs, and *Op1* and *Op2* are either additions, multiplications, comparisons, or shifts. This format is expressive enough to support diverse task data access patterns.

The microarchitecture of a Stream Unit is shown in Figure 46. It has three modules: the *Repetition Initializer*, the *Mem Address Generator*, and the *NCA (Scratchpad) Address Generator*. The Mem and NCA Address Generators have simple increment arithmetic that calculates the next iteration address to read from or to write to, respectively, within a given stream repetition (inner loop of Figure 43). Once these addresses are calculated, they are pushed into the Access Queue, waiting to be selected for issuing by the Stream Scheduler. The Access Queue additionally coalesces accesses from consecutive iterations that target the same cache line.

The Repetition Initializer calculates the bounds (*beg* and *end*) for each new repetition of a stream, using a *Bounds ALU* that computes *bexps*. Streams without parent start their single repetition when the task is issued, while child streams start a new repetition when their parent produces the needed value(s). *bexps* are configured before the kernel begins, and are propagated from the *VAcc-to-Streams Mapping* (Figure 42) to a Stream Unit when the latter is allocated to a task.

The *Parent Data Queue (PDQ)* holds data fetched by the stream’s parent to be used in computing *bexps*. The size of this queue determines how far a parent stream can run ahead with respect to its children, similar to [222].

5.4.5 Programming with ATX

Figure 47 shows the code executed by a CPU core to configure the UTE and generate ATX instructions, where each ATX instruction triggers one instance of the example task of Figure 44 executed on the NCA. During the *Task Configuration Time*, before the tasks are invoked, the core uses regular reads and writes to UTE configuration registers to configure

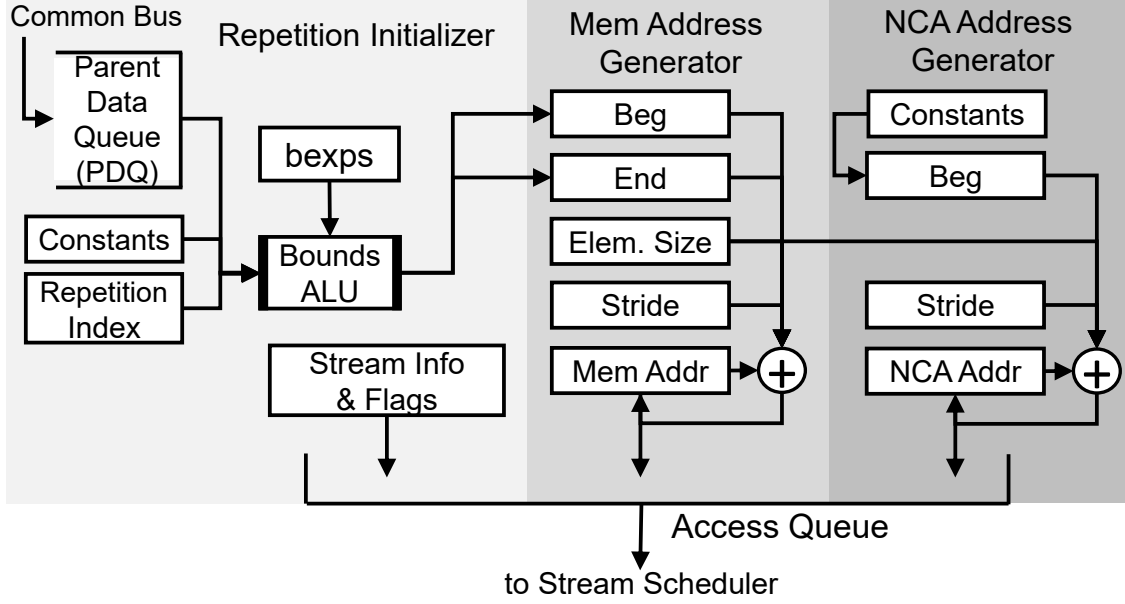


Figure 46: Stream Unit microarchitecture.

the UTE. The code first checks whether the UTE is attached to at least one instance of the accelerator type needed to execute the task type (Line 3). This is done by comparing the type identifier of the needed NCA against the contents of an architecture-dependent UTE read-only hardware table. This table stores the type identifiers of the physical NCAs attached to the UTE. If no physical instance of the needed type is found, the code falls back to the default non-ATX CPU execution (Lines 4–6). In this way, the binary remains compatible with other CPUs with the same ISA but different NCAs.

The programmer then defines an identifier for the specific task type (VAcc Id). Different task types in the same execution context must have different VAcc Ids. The UTE is then configured to map all tasks with this VAcc Id to the appropriate NCA type (Line 10). The UTE hardware fills the VAcc to PAcc Mapping module (Figure 42), to mark all physical accelerator instances (PAcc Ids) of this NCA type as capable of supporting tasks with this VAcc Id. Different VAcc Ids may be concurrently active at the UTE. Then, the number of streams for this VAcc Id is communicated to the UTE (Line 11).

The next step is to configure each stream of the task. This process involves configuring: (1) the size of the stream elements, (2) the parent id for the stream (-1 in the case of a root

```

1  *** Task Configuration Time ***
2  // NCA_Type is identifier of needed NCA type
3  success = UTE_check(NCA_Type);
4  if(!success){
5      // NCA type unavailable, fallback to non-ATX code
6  }
7
8  // General configuration
9  VAccId = 1; // Define VAccId for tasks of this type
10 UTE_cfg_VAcc_to_Type(VAccId,NCA_Type);
11 UTE_cfg_num_streams(VAccId,2);
12
13 // Stream S1 configuration
14 stream_id = 1
15 parent_id = -1 // This is a root stream
16 // 8 bytes per S1 element
17 UTE_cfg_stream_size(VAccId,stream_id,8);
18 UTE_cfg_parent(VAccId,stream_id,parent_id);
19 UTE_cfg_bexp_beg(VAccId,stream_id,S1_BEG_ENCODE);
20 UTE_cfg_bexp_end(VAccId,stream_id,S1_END_ENCODE);
21
22 // Stream S2 configuration
23 stream_id = 2
24 parent_id = 1 // S1 is the parent of S2
25 // 4 bytes per S2 element
26 UTE_cfg_stream_size(VAccId,stream_id,4);
27 ...
28
29 *** Task Execution Time ***
30 c21 = vals; // Starting address of vals[] array
31 c22 = sizeof(*vals);
32 rows_per_task = 16;
33 out_vregister = 0; // Clear output vector reg
34
35 for (r = 0; r < num_rows; r+= rows_per_task)
36     c11 = &row_ptrs[r];
37     c12 = &row_ptrs[r+rows_per_task];
38     in_vregister = {VAccId,c11,c12,c21,c22,0...};
39     // ATX instruction that triggers an NCA task
40     ATXV1V1{in_vregister,out_vregister};

```

Figure 47: CPU-core pseudocode to configure the UTE and generate ATX instructions. Each ATX instruction triggers one instance of the example task of Figure 44 executed on the NCA.

stream), and (3) the *bexps* that are used to calculate the beginning and end bounds of a stream repetition. Although we omit the details for simplicity, the *bexp* is encoded with 2 bytes that include the *Op1* and *Op2* operators, and specifiers for the *I1*, *I2*, *I3* operands (Section 5.4.4). Optionally, the programmer can also configure: (4) a non-unit stride, and/or (5) flags. The stream configuration information is stored in the VAcc to Streams Mapping module (Figure 42).

At this point, the programmer can use ATX instructions to trigger many different tasks of this type. This is shown in the *Task Execution Time* code (Lines 32–43). The code has a loop (Line 35) that, in each iteration, uses an ATX instruction (ATXV1V1 in Line 40) to trigger a task that performs the operation of Figure 44 on 16 consecutive sparse matrix rows. We choose 16 rows per task so that the NCA output fits in a 512-bit vector register—although we could have used one or two 1KB tile registers to support larger outputs. The code loads from memory the task-specific runtime constants *c21*, *c22*, *c11*, and *c12*. It then concatenates the VAcc Id and these four constants, and stores them in a vector register padded with zeros (Line 38). This vector register is passed as the input operand of the ATX instruction (Line 40). After the ATX instruction completes, the core can use the output vector register (*out_vregister*) for other computations or write its contents to memory using regular store instructions. Note that, in actual code, the ATX input and output data would be defined as a struct variable, which the compiler then translates into input and output registers. The loop in Line 35 can be parallelized across multiple cores and their NCAs using, for example, OpenMP constructs.

The overhead of task configuration is negligible, as task configuration is only needed once per task type, and is then reused for potentially thousands of tasks per core during the actual task execution time.

5.4.6 UTE Task Prefetching

The ATX design enables NCAs to exploit a high degree of memory-level parallelism (MLP). This results from encoding all the memory accesses of a task in a single ATX instruction, and from supporting fast and efficient memory address and access generation in the UTE. However, even with this support, there are scenarios where even more MLP is desirable to fully saturate the memory resources of the architecture.

A natural way to do so is to increase the number of tasks that are concurrently processed by the UTE backend. However, there can be two obstacles to attain this. First, there may not be enough NCA scratchpads to store all the data that tasks fetch from memory. Second, the CPU cores may not be able to initiate new tasks fast enough. To overcome these limitations, we introduce two forms of *task prefetching*—each one targeting a different obstacle.

1. Assisted Task Prefetching. This mechanism targets the scenario where the CPU core can produce new tasks fast enough, but new tasks cannot be dispatched to the UTE backend due to insufficient NCA scratchpads to accept the data fetched from memory. In this case, new tasks will wait in InTaskQ (Figure 42) until an NCA is freed up. With Assisted Prefetching, tasks that wait in InTaskQ are allowed to be dispatched to the backend in *prefetch mode*. In this mode, no NCA is assigned to the task. Instead, the hardware simply prefetches data from leaf streams (e.g., *S2* in Figure 45(b)) to the L2. Data from non-leaf streams (e.g., *S1* in Figure 45(b)) is fetched into the Stream Units of the children streams in the UTE, in order to generate addresses. However, such data is not propagated to a PAcc port, as would happen in non-prefetch mode. We call these task prefetches *assisted*, since the CPU core assists the UTE with precise information about future tasks. Later, when an NCA is freed up, the task will be re-issued to the backend in non-prefetch mode, and likely find most of its data already in the L2.

2. Predicted Task Prefetching. This mechanism targets the scenario where the CPU core cannot produce new tasks fast enough. In this case, InTaskQ remains mostly empty,

so the UTE has to predict the tasks that will come next. Luckily, predicting tasks is easier than predicting individual memory accesses. In conventional data prefetching, a hardware prefetcher predicts which memory accesses will come next, given the past memory accesses. In Predicted Task Prefetching, the *Task Predictor/Prefetcher* (Figure 42) predicts which tasks will come next, given the past tasks. Recall that different tasks of the same type share the stream dependency and *bexps* configuration (Figure 45). They only differ in the values of their runtime constants (Figure 45(a)). Thus, we can reformulate the goal of the Task Predictor as to *predict which runtime constants will come next, given the ones observed in the past*.

While we can reuse the ideas of many existing prefetch algorithms, a simple stride algorithm proved effective. Our predictor observes the runtime constants in two consecutive tasks of the same type and extracts their strides. Then, to produce the constants for the next predicted task, it increments the constants of the current task by the extracted strides. In this way, a new predicted task is produced for each real task.

To hide latency effectively, we do not predict the next task, but the next N_{th} task. Hence, we increment the constants of the current task by N times the extracted strides. We call N the *Prefetch Distance*. High distances correspond to deeper prefetches. To set the prefetch distance for a task, we use a simple runtime heuristic that monitors the average task input sizes, and uses small distances for large sizes and vice-versa. In particular, we use a distance of 1 for tasks with average input size larger than 32KB, 2 for 16KB, etc. More sophisticated techniques to adjust the prefetch distance at runtime such as the one presented later in Chapter 9 are possible, but we leave them as future work. After the constants for the predicted task have been determined, the task is dispatched to the UTE backend in *prefetch mode*, as described above.

5.5 Methodology

1. Simulation and System Architecture. We evaluate ATX NCAs with an internal silicon-validated multicore CPU simulator that extends Sniper [186]. We further extend our simulator to model NCAs, ICAs, OCAs, ATX instructions, and the UTE. Our baseline architecture is a 64-core CPU with the system parameters of Sapphire Rapids (SPR) with HBM (i.e., Xeon-Max) [149], [221], [223]. We scale the HBM memory bandwidth of the architecture to 4TB/s to match state-of-the-art decoupled accelerator platforms [7], [224]. To eliminate MSHR-related MLP limitations [221], we scale the number of L2 MSHRs to 128. The architecture includes industry-grade L1 and L2 spatial prefetchers that have been silicon-validated against the real SPR prefetchers. Table 15 shows the main CPU, UTE, and accelerator parameters.

Table 15: System parameters.

	CPU
General	1 socket; 64 cores; 2.5GHz; AVX/AMX support;
Caches	16-entry ATX Queue
Mem	48KB L1; 2MB L2; 1.875MB LLC per core
	DDR 1TB, 270GB/s; HBM 64GB, 4TB/s
	UTE
General	Per-core UTE: 2 Input Buffers (i.e., scratchpads) per NCA; 32KB/buffer; 32 Stream Units; Common Bus with 128B(data) + 32B(addr); 128-entry LDQ; 1KB PDQ per Stream Unit; only Predicted Task prefetching enabled
	Accelerators Modeled
SpMM/ SDDMM	SPADE (Chapter 3)-like arithmetic unit
GeMM	GeMM unit operating on 8x8 μ tiles
Decompr.	DECA (Chapter 4)-like; W=64, L=16

The L2 cache is shared by the core and the UTE, and an arbiter selects between requests from the core’s L1 and the UTE, similar to L2 caches shared by multiple cores [234]. An ATX execution port is added to the core pipeline, which competes with the other

Table 16: Benchmark sparse matrices.

Short name	asi	liv	del	pac	ser
Rows (mil)	12	4	17	2	1
NNZ (mil)	25	69	101	35	64

execution ports (INT, FP, LD, VEC, etc.) for physical register file (PRF) write access. Although we did not increase the number of PRF write ports, we found that ATX reduces PRF pressure compared to core-only execution, as an ATX instruction replaces multiple loads.

As shown in Table 15, we model three ATX NCA accelerators: for SpMM/SDDMM operations, GeMM operations, and ML weight decompression. The arithmetic units in the SpMM/SDDMM and decompression accelerators are modeled after SPADE (Chapter 3) and DECA (Chapter 4), respectively. For GeMM, we model double-precision GeMM units operating on 8×8 μ tiles. The maximum combined throughput of all 64 SpMM/SDDMM accelerators in our modeled platform is 5.1 double-precision TFLOPs. For the 64 GeMM accelerators, the maximum total throughput is 20.5 double-precision TFLOPs.

We implement double buffering using two 32KB Input Buffers (i.e., scratchpads) per NCA, each of which is attached to a different UTE PAcc port. The scratchpads determine the maximum task input size. For the output operands of the ATX instructions, we use up to two tile registers of 1KB each. This limits the maximum task output size to 2KB.

2. Comparison with Alternative Accelerator Organizations. We compare ATX NCAs against different organizations of the same accelerators: (1) *perfect* ICAs that take zero time to perform computations, (2) L2-attached OCAs (*L2 OCA*) controlled with RoCC-like instructions [218], [235], which, unlike ATX, do not execute speculatively, and (3) LLC-attached OCAs (*LLC OCA*) controlled with memory loads and stores (Section 5.2). The two OCA configurations include optimized out-of-core memory access interfaces that feature the stream support of the UTE.

3. Evaluated Kernels. We use four kernels from scientific computing and machine

learning (ML):

- The Sparse Matrix - Dense Matrix Multiplication (SpMM) and Sampled Dense Matrix - Dense Matrix Multiplication (SDDMM) sparse kernels (Section 2.2). In these kernels, the CPU traverses the sparse matrices, observes the sparsity patterns, creates tasks on the fly, and invokes the accelerator. The programmer or library needs to ensure that the generated tasks do not overflow the accelerator scratchpads. These kernels use fine core-accelerator interleaving. As benchmark sparse matrices, we use `asia_osm`, `com-LiveJournal`, `delaunay_n24`, `packing-500x100x100`, and `Serena` from SuiteSparse [73], based on the selections made in Chapter 3. Table 16 characterizes the matrices. SpMM and SDDMM are executed with double precision and with 2MB virtual pages to reduce the impact of TLB misses and page walks.
- The popular GeMM kernel, to demonstrate ATX generality.
- Decompression of a sparsified and quantized ML model where, alongside the accelerator, the CPU executes computations using AMX (Chapter 4). This kernel also requires fine core-accelerator interleaving.

4. Software. For our baseline CPU-only execution, we use the vendor-optimized MKL library [236] for SpMM and GeMM. MKL does not support SDDMM and, thus, we use an optimized SDDMM kernel from TACO [99]. For the ML model decompression, we use the `decompress+execute` kernel from `libxsmm` [160]. All kernels are vectorized with AVX512, while the decompression also uses AMX. For ATX, we create hand-crafted kernels that break down the computation into tasks and use ATX instructions to invoke the NCAs. For parallelization across cores, UTEs, and NCAs, we use OpenMP, to demonstrate that our framework is fully compatible with popular CPU parallel programming models.

5. Overhead of ATX Extensions. The main overhead of the ATX extensions is the UTE. The UTE has modest storage needs. In the UTE, the VAcc-to-PAcc and VAcc-to-Streams Mapping tables contain the only architectural process state. In our configuration, these tables require 0.5KB and 4KB, respectively, which is more than

enough to concurrently configure the three NCAs used in our evaluation. For comparison, Intel cores with AMX [150] need 8KB of architectural state for AMX tile registers alone. The remaining UTE storage is non-architectural state in structures such as the InTaskQ, OutQ, LDQ, Stream Units, and other queues. We calculate that the total storage overhead for one UTE is less than 128KB. Using CACTI [102] and [104], we estimate that 64 UTEs account for less than 1% of the SPR die area [192]. Using the same tools, we estimate that their combined static and dynamic power (at maximum activity) is 4.37% of the SPR TDP of 350W.

5.6 Evaluation

In this section, we compare ATX NCAs with other accelerator organizations and evaluate different aspects of ATX NCAs.

1. Comparison with ICAs. Figure 48 shows the speedups of executing with ICA or with our ATX NCA, over core-only execution. For SpMM/SDDMM, we run our five sparse matrices; for GeMM, we evaluate five configurations which, using BLAS [237] terminology, use $N=128$ and $M=K=1k, 2k, 4k, 8k, 16k$. ICA relies on the core’s memory access infrastructure, and thus suffers from reduced MLP exploitation capabilities. This limitation does not exist in ATX NCA. As a result, ATX NCA delivers average speedups of 2.3x, 2.0x, and 1.3x over ICA for SpMM, SDDMM, and GeMM. The speedup for GeMM is slightly lower, since this kernel has more regular access patterns, which are easier to handle using the existing core’s memory access infrastructure. Compared to core-only execution, ATX NCA delivers average speedups of 2.8x, 2.7x, and 2.7x. We see similar trends across a wide variety of sparse matrices and GeMM sizes.

2. Comparison with L2 OCA and Ablation Study. We now compare ATX NCAs with L2 OCAs controlled with RoCC-like instructions. Although L2 OCAs share the same spatial placement as ATX NCAs, ATX NCAs have two advantages over them: (1) ATX instructions can be invoked out-of-order and speculatively, and (2) the ATX framework

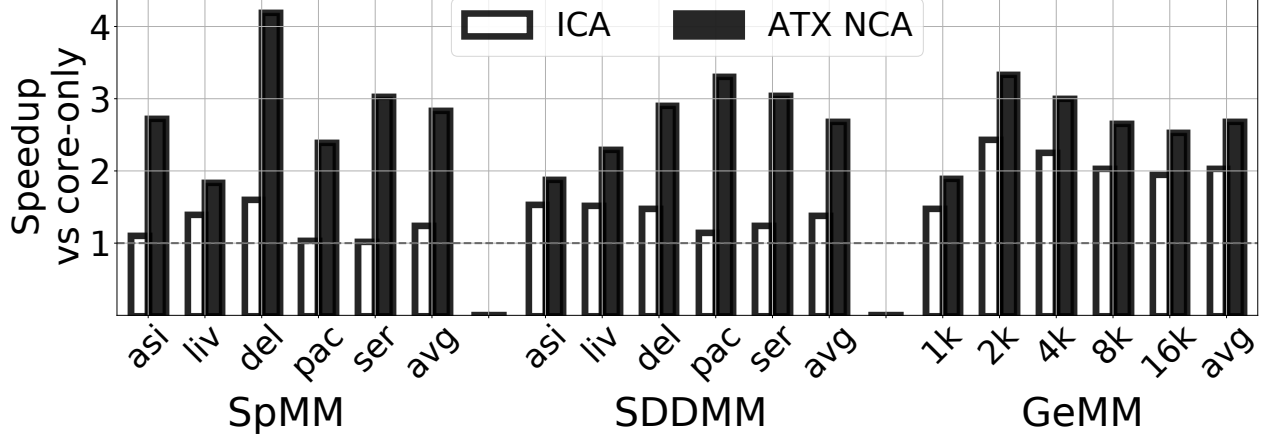


Figure 48: Speedups of ICA and of our ATX NCA, over core-only execution for different kernels.

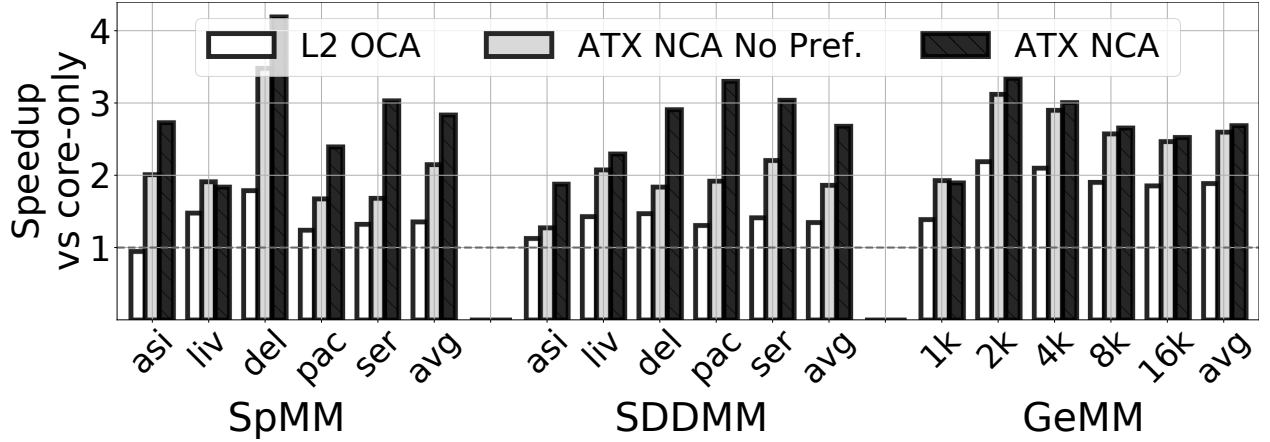


Figure 49: Comparison with RoCC L2 OCA and ablation study.

supports task prefetching that can further accelerate data provision from memory. To isolate each effect, Figure 49 shows the speedups of executing with L2 OCA, which is controlled with RoCC-like instructions and uses the SPR L2 prefetcher (*L2 OCA*), with ATX NCA with task prefetching disabled (*ATX NCA No Pref.*), and with our full proposed ATX NCA (*ATX NCA*), over core-only execution. We see that *ATX NCA No Pref.* delivers average speedups of 1.6x, 1.4x, and 1.3x over L2 OCA for SpMM, SDDMM, and GeMM, respectively. With prefetching, our full ATX NCA design attains average speedups over L2 OCA of 2.1x, 2.0x, and 1.4x.

3. Comparison with LLC OCA and Effect of Task Input Size. We compare ATX

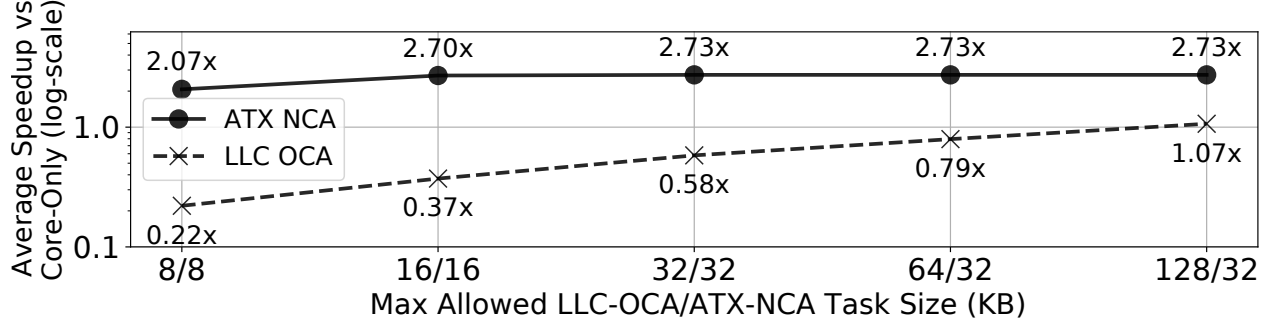


Figure 50: Speedup of ATX NCA and LLC OCA over core-only with various task sizes running SpMM.

NCA with the most common form of OCAs, namely OCAs attached at the LLC. We evaluate different sizes of the Input Buffers (i.e., scratchpads), which limit the maximum task input size. Each LLC OCA or ATX NCA has two scratchpads for double buffering. We vary the size of each scratchpad from 8KB to 128KB in LLC OCA and, to keep the near-core SRAM small, from 8KB to 32KB in ATX NCA. Hence, for the same kernel, an ATX NCA with smaller input buffers executes more, smaller tasks than an LLC OCA with larger input buffers. For the LLC OCA, the maximum output size is equal to the input size, while the ATX NCA uses 1-2 output tile registers of 1KB each.

Figure 50 shows the speedups of ATX NCA and of LLC OCA, over core-only for various task sizes running SpMM. We do not show the other kernels for space reasons, but they have similar trends. LLC OCAs incur high core-accelerator communication latency. As OCA invocations do not execute out-of-order, this extra latency has a noticeable performance impact. The figure shows that ATX NCA is 9.4x faster than LLC OCA for an 8KB maximum input task size, and 2.6x for a 128KB maximum input task size. The NCA significantly outperforms the OCA for small tasks finely-interleaved with core execution. The OCA’s performance will eventually match NCA’s for very large tasks, but at the cost of reduced interleaving with the core and more on-chip scratchpad memory.

4. Roofline Performance. Figure 51 presents a roofline analysis for the SDDMM kernel with: (a) core-only execution, and (b) computations offloaded to different ATX NCA variants. Performance is shown in Giga Vector Operations per second, where not all vector

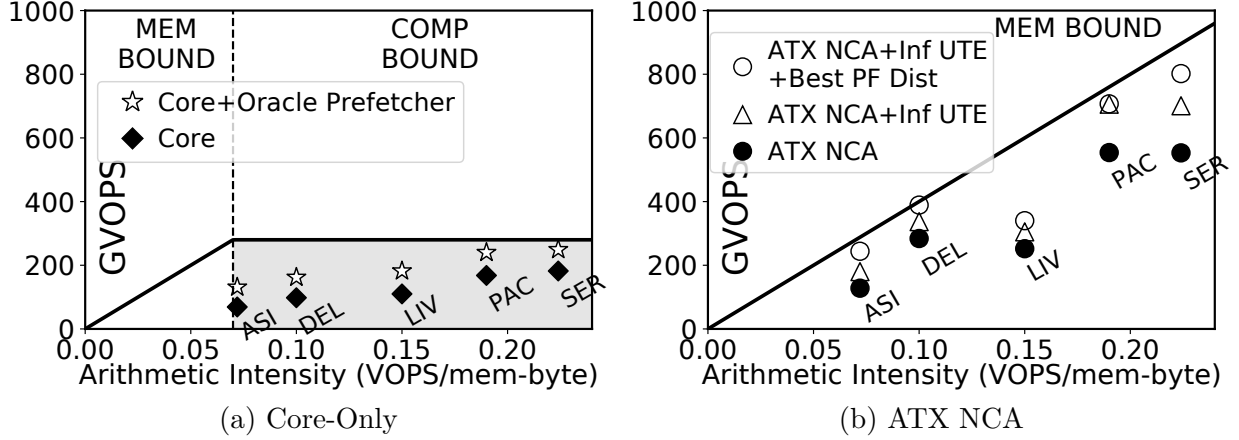


Figure 51: SDDMM rooflines.

operations produce FLOPs. Figure 51(a) shows that, without an accelerator, all performance points are in the compute-bound region. Figure 51(a) also shows a scenario where we replace the SPR prefetchers with oracle ones (*Core + Oracle Prefetcher*) that eliminate all stalls due to the memory subsystem. This unrealistic scenario serves as a performance limit of prior prefetching works for provisioning data to CPU cores (e.g., [238], [239]). Despite this addition, performance does not even reach the core’s roofline in the compute-bound region due to dependencies, frontend stalls, and other types of core-bound stalls in the CPU core pipelines.

However, when variations of ATX NCAs are added in Figure 51(b), the roofline ridge point moves far to the right, and all the kernels are now in the memory-bound region. We see that our proposed ATX NCA offers significant improvements. However, there is still some performance left on the table. There are two main reasons for this. First, UTE resources are finite. Second, the heuristic that we use to determine the predicted task prefetch distance (Section 5.4.6) is suboptimal. To gain further insight, in Figure 51(b), we also include performance points for: (1) a UTE provisioned with very high resources such as number of Stream Units and PDQ size (*ATX NCA+Inf UTE*), and (2) additionally, an oracle that sets the optimal task prefetch distance for each matrix statically (*ATX NCA+Inf UTE+Best PF Dist*). We see that *Inf UTE+Best PF Dist* reaches the roofline

for all matrices except for LIV.

The reason for the gap in LIV is the task prediction mechanism itself. To predict future tasks, we used a simple algorithm inspired by stride prefetchers. Naturally, such an algorithm (even with the best distance) is not always optimal.

5. Performance, Area, and Power for Different UTE Design Points. To test which UTE design parameters affect performance the most, we perform a design space exploration. We vary: (1) the number of Stream Units, which limit how many real and prefetch tasks can be handled simultaneously in the UTE backend, (2) the LDQ entries (Figure 42), which limit the number of outstanding memory requests, (3) the Common Bus (CB) data width (Figure 42), which limits how many fetch/prefetch requests from the Stream Units the Scheduler can accept per cycle, and (4) the PDQ size (Figure 46), which limits how much a parent stream can run ahead of its children. We start with the Inf UTE, for which we have assumed $\{\text{\#Stream Units, LDQ Entries, CB Width, PDQ Size}\} = \{64, 512, 512\text{B}, 4\text{KB}\}$, and progressively reduce each resource while keeping the rest to their "Inf" values to isolate the effect of each resource.

Table 17 shows the performance of SpMM for the different design points normalized to Inf UTE. Our default UTE parameter values are $\{32, 128, 128\text{B}, 1\text{KB}\}$, which can be shown to achieve 80% of the performance of the Inf UTE. We see that the PDQ Size and the CB Data Width are the two most critical resources bounding performance.

In Table 18, we additionally compare different UTE configurations with respect to their area and power impact. These configurations are a Small UTE with parameter values $\{8, 32, 64\text{B}, 256\text{B}\}$, our Default UTE with $\{32, 128, 128\text{B}, 1\text{KB}\}$, and the Inf UTE with $\{64, 512, 512\text{B}, 4\text{KB}\}$. The area overhead is shown as a percentage of the SPR die size (1600mm^2). The power overhead combines static and dynamic power, and is shown as a percentage of the SPR TDP (350W). Dynamic power is estimated at maximum feasible activity. All overheads are for a total of 64 UTEs. We see that our Default UTE is a good design point. It has 3x less area and 2.1x less power overhead than the Inf UTE, while the

Table 17: Average SpMM performance for different UTE design points as a percentage of the performance of Inf UTE.

#Stream Units	Performance
4	56%
8	86%
16	99%
32	99%
64	100%

LDQ Entries	Performance
32	70%
64	90%
128	98%
256	99%
512	100%

CB Data Width	Performance
64B	73%
128B	88%
256B	98%
512B	100%

PDQ Size (Bytes/Stream Unit)	Performance
128	16%
256	36%
512	62%
1k	85%
2k	96%
4k	100%

Table 18: Area and power overheads of different UTEs as a percentage of SPR die size and TDP.

	Small UTE {8,32,64B,256B}	Default UTE {32,128,128B,1KB}	Inf UTE {64,512,512B,4KB}
Area	0.53%	0.90%	2.71%
Power	3.64%	4.37%	9.17%

Inf UTE has only 1.25x higher performance. Further, the Default UTE has 1.7x the area and 1.2x the power overhead of the Small UTE, while it improves performance by 2.5x.

6. Task Prefetching Analysis. To evaluate task prefetching, we show results for only SpMM and SDDMM, since prefetching helps GeMM relatively little (Figure 49). Figure 52 shows the speedups of ATX NCA with Assisted Prefetching, Predicted Prefetching, or both, over ATX NCA without prefetching. We see that Predicted Prefetching is the most effective. Only the *ser* matrix benefits from Assisted Prefetching. In most cases for these kernels, the CPU cannot produce new tasks fast enough for assisted prefetches to be timely generated. Combining the two schemes does not lead to significant improvements over only Predicted Prefetching, which is the default mechanism in our evaluation.

Figure 53 shows the impact of the distance of the predicted task prefetches. The figure shows the speedups obtained by applying Predicted Prefetching with distances 1, 2, 4, and

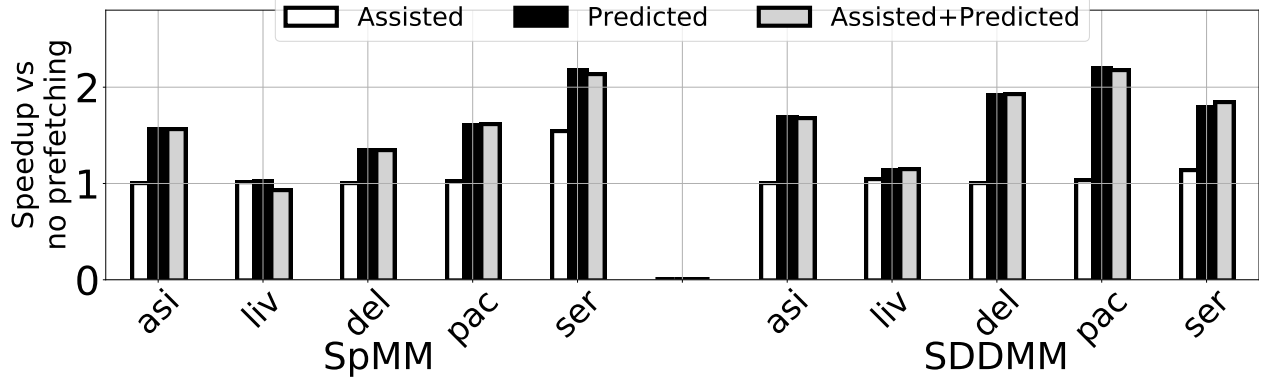


Figure 52: Sensitivity to the task prefetch mechanism.

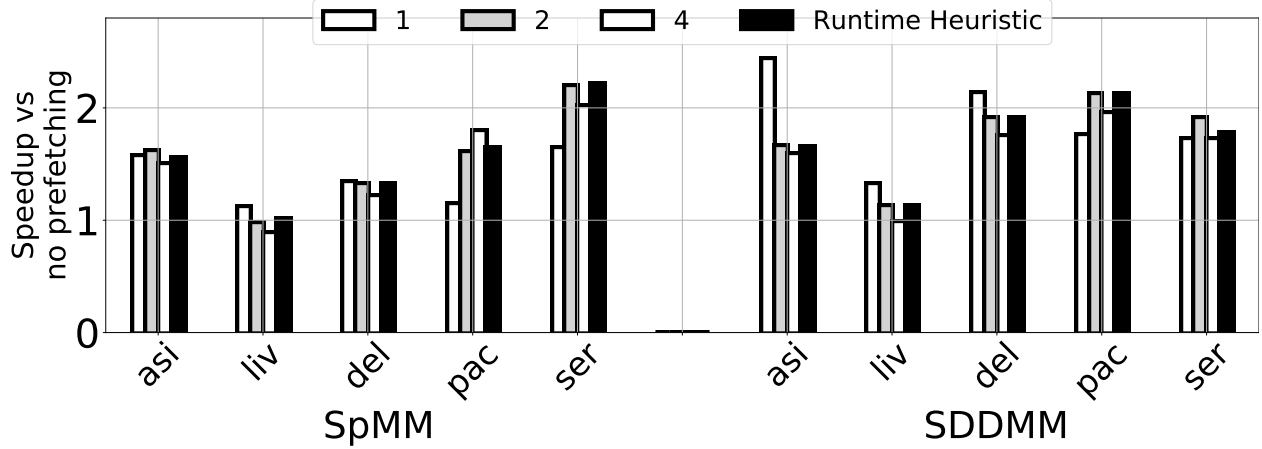


Figure 53: Sensitivity to the task prefetch distance.

with our runtime heuristic, over ATX NCA without prefetching. The best distance varies across kernels and matrices. Also, our simple heuristic of Section 5.4.6 is not always the best. We believe that more sophisticated heuristics for adjusting the distance at runtime, potentially inspired by conventional hardware prefetching, may prove effective.

7. Decompression Use Case. Up to now, we have considered cases where the computation is primarily done by the accelerator, and the core is primarily used for input-inspection, task sizing, or control. We now discuss a case where both core and accelerator perform computation and interleave in a fine-grained manner: the accelerator (i.e., DECA) reads tiles of an ML model from memory in a compressed form (quantized/sparsified) and decompresses them; the core takes the resulting tiles and executes GeMMs with AMX instructions. Compared to the previous kernels, the input

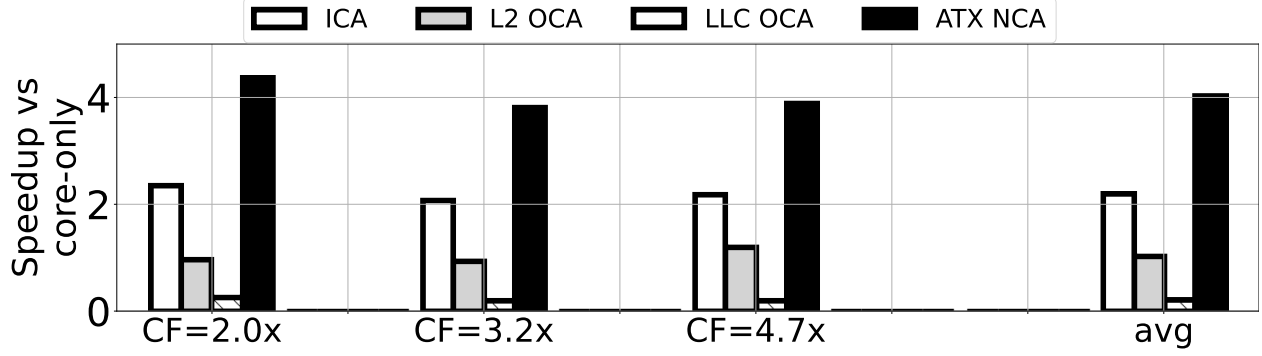


Figure 54: Comparing accelerator schemes for decompression.

task sizes here are notably small (512B–2KB). Figure 54 shows the speedups of the different accelerator integration schemes over the core-only execution for different model compression factors (CF). We see that ATX NCA delivers average speedups of 4.0x, 1.8x, 3.9x, and 18x over core-only, ICA, L2 OCA, and LLC OCA, respectively. The large speedups over the OCAs for this use case are due to the small task sizes, which require fine core-accelerator interleaving.

5.7 Limitations and Discussion

NCAAs are very effective for fine-grained tasks that are both computationally and memory demanding, and that require tight CPU core control and communication. We demonstrated the NCA effectiveness in popular kernels of important workloads. However, NCAs and ATX are not intended to replace ICAs or OCAs for all accelerated code. In this section, we discuss the limitations of NCAs when compared to them.

OCAs have some advantages over NCAs. In contrast to OCAs, NCAs cannot write to the memory system, a design decision that (1) limits the output size of the task to register sizes, and (2) does not improve the performance of memory stores over general-purpose cores. The first limitation can be largely alleviated by using the large register sizes provided by current processors (e.g., 1 KB tile registers) and by the fact that, often, tasks can be scaled to the desired size (e.g., with matrix tiling). The second limitation is not crucial for many workloads because stores are generally less critical than loads for

performance [240], [241]. Still, for coarse-grained write-intensive operations such as memory scatters [242] or memory copies, OCAs may be the better option. Another advantage of OCAs is that they can be shared between cores, whereas NCAs are private to each core.

In some applications, ICAs may be the best option. NCAs excel at fine-grained tasks whose input data is stored in memory. However, ICAs may be better for tasks where most input data can be stored in registers and the L1 cache.

Overall, NCAs are an additional design point between ICAs and OCAs, addressing a gap in terms of core-accelerator communication overhead and memory access efficiency. A full analysis and partitioning of tasks between ICAs, NCAs, and OCAs is an open problem and our future work.

5.8 Related Work

1. CPU-integrated accelerators. A wide variety of CPU-integrated accelerators have been proposed (e.g., [17], [18], [92], [122], [123], [165], [211], [219]). In this work, we classify them based on how they interact with the system’s state. Examples of commercial accelerators that follow the OCA abstraction include the Intel DSA, QAT, IAA, and DLB that target data-center taxes [16], [18], the NPU for AI acceleration in client architectures [229], [243], and the GPU in the AMD APU [230], [231]. Commercial accelerators that follow the ICA abstraction include the Intel TMUL [150] and the Arm SME accelerator [244]. In this work, we introduce a third abstraction for CPU-integrated accelerators: NCAs. Note that the term Near-Core Accelerator has been previously used to refer to the specific spatial placement of the accelerator, namely L2-attached [36], [245]. In this work, we use the term to refer to the abstraction used by the accelerators to interact with the architectural state of the system, which is not tied to their spatial placement.

2. Controlling CPU-integrated accelerators. ICAs are typically controlled through accelerator-specific ISA extensions [123], [150], [180], [246], such as Intel AMX and Arm SME. OCAs are typically controlled through polling with regular load and store

instructions [18], [233]. An alternative to polling is interrupts [247], which often have high latency even with optimizations [233]. Works such as RoCC [218], [220], [235], [248] and NUCD [249] introduce instructions for controlling accelerators outside the core pipelines. However, as such works implicitly assume an OCA abstraction for the controlled accelerators (e.g., they can write to memory), RoCC or NUCD instructions cannot be executed speculatively. DECA (Chapter 4) includes instructions for speculative accelerator invocation. However, such instructions are DECA-specific and do not directly generalize to other accelerators. In this chapter, inspired by DECA’s implicit execution semantics, we defined the NCA abstraction and developed the ATX framework and instructions that support a diverse range of NCAs.

3. Accelerating CPU data accesses. A variety of techniques can increase MLP in CPUs, ranging from prefetching [239], [250], [251], [252], [253], [254] to decoupled-access units [165], [166], [255]. SSP [222] first introduced the idea of encoding memory accesses with streams, while works such as the TMU [165] have adopted a similar abstraction. The Stream Units in our UTE are inspired by [222]. We showed that even augmenting CPU cores with an oracle prefetcher cannot match the performance of ATX NCAs. More advanced UTE task prefetching, potentially inspired by conventional hardware prefetching [250], [253], is a promising direction.

5.9 Conclusion

This chapter classified CPU-integrated accelerators based on their interaction with the architectural state of CPU cores. It then proposed a new class of accelerators called Near-Core Accelerators (NCAs). To support NCAs, it introduced ATX, a set of instructions and hardware extensions so that NCAs are invoked speculatively and out-of-order, and exploit a high-performance interface with the memory system. ATX NCAs accelerate key kernels by 1.3–18 $\times$ over various alternatives.

Chapter 6 Software Acceleration of Distributed Sparse Computations

Author contribution statement: The work described in this chapter was co-led by the thesis author and Charles Block, who made equal and substantial contributions. The thesis author motivated the work, conceptualized the core idea and high-level algorithm, and implemented an early shared-memory prototype. Charles Block developed the preprocessing model, implemented the distributed-memory algorithm, and conducted the evaluation experiments.

6.1 Introduction

The ever-increasing computing and memory demands of sparse matrix computations introduce the need for efficient *distributed* SpMM. However, designing efficient distributed SpMM algorithms is challenging [74], [75], [256]. Due to the low arithmetic intensity of this kernel, the communication cost, rather than the computation cost, typically dominates the execution time. Such cost depends on how communication is scheduled (Section 2.3).

Communication can be scheduled in a sparsity-unaware manner, such as with collective communications. For example, assume that each node originally hosts a block of the dense input matrix. This block may be sent to all the other nodes by using collectives or RDMA accesses to fully replicate it [75] or by using shifting algorithms [74] similar to those that would be used for dense computation. With this strategy, execution is often inefficient due to redundant data transfers, since parts of the dense input matrix may not be needed by some nodes.

On the other hand, communication can be scheduled in a fine-grained sparsity-aware manner [257], communicating only the truly necessary data and computing asynchronously. Specifically, when a node processes a sparse element but does not own the necessary dense row for that computation, it gets that row with a fine-grained one-sided request. With this strategy, execution can also be inefficient due to high software overheads and the need for

more network round-trips [74].

In this work, we observe that individual sparse matrices often contain regions that are relatively denser and regions that are relatively sparser. Based on this observation, we develop a model that partitions computation and communication into sparsity-unaware and sparsity-aware portions. Relatively denser regions of the sparse matrix are broken down into *Synchronous Stripes* and transfer the corresponding parts of the dense input matrix with *Sparsity-Unaware (SU)* transfers. Relatively sparser regions are broken down into *Asynchronous Stripes* and transfer the corresponding parts of the dense input matrix with *Sparsity-Aware (SA)* transfers.

Leveraging the partition, we develop a new algorithm that performs collective communication for the synchronous stripes, and fine-grained, one-sided communication and asynchronous computation for the asynchronous stripes. We call the algorithm *Two-Face*. The synchronous and asynchronous parts of the sparse matrix are processed in parallel, and the model aims at equalizing the runtimes of the two parts.

We evaluate *Two-Face* on a CPU-based supercomputer using large matrices and compare it to state-of-the-art baselines. For a system with 32 nodes, 128 cores per node, and dense matrices with 128 columns, *Two-Face* attains an average speedup of 2.11x against dense shifting [74], a high-performing baseline. In addition, *Two-Face* is a scalable algorithm: its average speedup over dense-shifting increases to 2.21x for 64 nodes. Finally, the overhead introduced by the necessary matrix preprocessing step is small enough to make *Two-Face* suitable for applications that use the same sparse matrix only a few dozen times.

Overall, this chapter’s contributions are:

- The *Two-Face* software algorithm for distributed SpMM, which is based on a mix of collective and one-sided communication.
- A low-overhead model and method to partition sparse matrices into regions corresponding to the two access types.

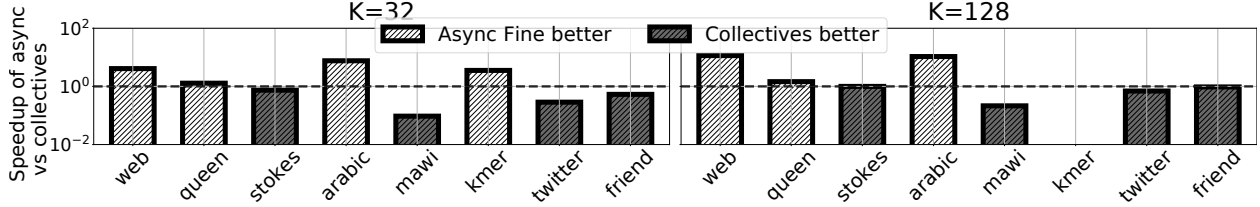


Figure 55: Speedup of *Async Fine* over a full-replication AllGather collective implementation for $K = 32$ (left) and $K = 128$ (right). There is no data for *kmer* with $K = 128$ due to the large memory consumption of the full-replication collective algorithm.

- An evaluation of *Two-Face* on a supercomputer and a comparison with the state-of-the-art.

6.2 Motivation

The reader is referred to Section 2.3 for the necessary background on the access patterns in SpMM, the 1D partitioning method for distributing the SpMM data structures in a multi-node system, and the differences in the communication patterns of the *SU* and *SA* approaches.

We now show that, for some matrices, the *SA* pattern is more suitable, while for others the *SU* is better. Then, we provide an example to motivate that a combination of the two approaches for the same matrix can yield the best results.

6.2.1 Choice of SA & SU SpMM is Input Dependent

Both the *SU* and *SA* approaches have pitfalls: *SU* can lead to unnecessary data transfers, while *SA* transfers can have high software overhead and more round-trips between nodes. To compare their performance, we profile the execution of distributed SpMM using *Async Fine* (a *SA* approach that uses RDMA for fine-grained transfers) and AllGather collectives (a *SU* approach). We use a distributed machine with 32 nodes and 128 CPU cores per node, running the two algorithms for 8 large sparse matrices from SuiteSparse [73], and for two different values of K (32 and 128). Section 6.5 gives more details about our methodology. Figure 55 displays our findings. The figure shows the speedup of *Async Fine*

over the AllGather implementation (*Collectives*) for $K = 32$ (left) and $K = 128$ (right). We do not include results for the kmer matrix with $K = 128$ because the single-node memory demand of *Collectives* exceeds the single-node capacity in our system.

We see that, for half of the matrices, *Async Fine* outperforms *Collectives*, while the opposite is the case for the other half. The speedup of *Async Fine* over *Collectives* reaches 11.5x, while the speedup of *Collectives* over *Async Fine* reaches 10.5x. Clearly, whether *SU* or *SA* works best depends on the sparsity pattern of the input matrix.

6.2.2 Combining SU & SA Transfers for a Single SpMM

Typically, the nonzeros in a sparse matrix are not evenly distributed. Consequently, combining the *SA* and *SU* communication flavors for a single SpMM could be beneficial. An example of this idea is shown in Figure 56. We split the nonzeros into three categories: (1) *local-input nonzeros* are those for which the dense input rows needed are already local to the node with the nonzeros and, therefore, no remote transfers are needed; (2) *sync nonzeros* are those for which the dense input rows needed are better off transferred through *SU* transfers; and (3) *async nonzeros* are those for which it is more beneficial to transfer the rows through *SA* transfers.

To understand which nonzeros are *sync* and which are *async*, consider the example. Columns 4 and 5 of the sparse matrix are quite dense. This means that the corresponding dense input rows of B (shaded in the figure) are useful to many nodes. Specifically, $B[4,0:K-1]$ is needed by N1, N2, and N3, while $B[5,0:K-1]$ is needed by all the nodes. Hence, it is likely beneficial to transfer the whole group of rows hosted by N2 to all the other nodes through a collective broadcast operation. Hence, we classify the nonzeros at (0,5), (2,4), (3,5), (6,4), and (6,5) as *sync*. On the other hand, $B[0,0:K-1]$ is not needed by any of the remote nodes and $B[1,0:K-1]$ is only needed by the N2 remote node—N0 also accesses both rows, but they are already local and no transfers are needed. Transferring the whole group of dense rows that N0 hosts through a collective broadcast would lead to

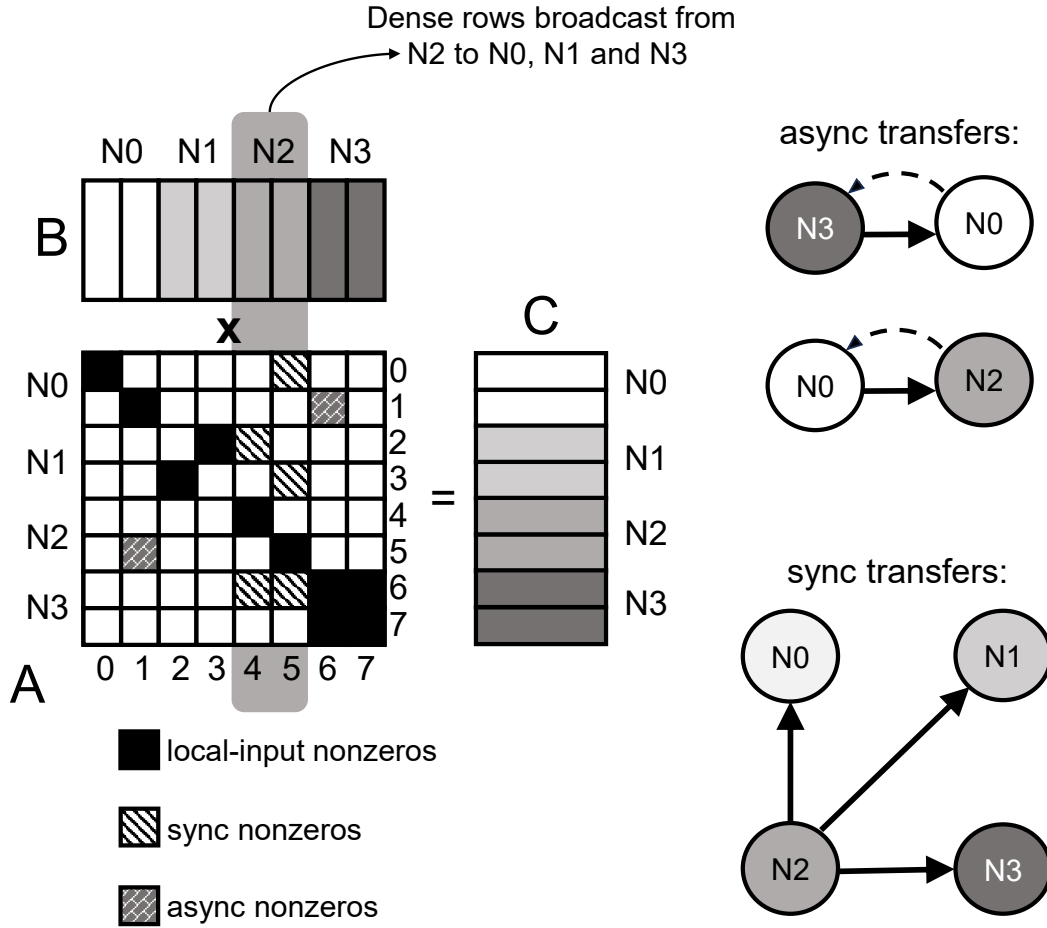


Figure 56: Example of how combining the two communication flavors can be beneficial.

many unnecessary data transfers. Thus, the best strategy is likely for N2 to issue a one-sided request to N0 to get $B[1,0:K-1]$, without synchronizing with the rest of the nodes. Therefore, the nonzero at (5,1) is classified as *async*. A similar situation occurs with the nonzero at (1,6), which is an *async* nonzero. Note that a collective broadcast operation does not necessarily need to include all nodes as destinations. For example, if the nonzero at (0, 5) was absent, then N2 could still issue a multicast transfer directed only to N1 and N3.

6.3 Overview of Two-Face

In this section, we translate the intuition about combining the *SU* and *SA* approaches into an algorithm called *Two-Face*. *Two-Face* combines sparsity-unaware and sparsity-aware transfers for efficient distributed SpMM. In this section, we describe how sparse matrices are analyzed and partitioned into two types of regions.

6.3.1 Sparse Matrix Partitioning

Before the SpMM execution begins, the sparse matrix is preprocessed in order to determine which of the data transfers will use coarse-grained multicast operations, and which will use fine-grained one-sided communication. Then, during runtime, both types of transfers and their corresponding computations will proceed in parallel.

Two-Face adopts 1D partitioning (Section 2.3.1). Thus, each node, which contains one MPI rank, is responsible for the nonzeros in a group of consecutive rows of sparse matrix A . In addition, the node hosts the corresponding group of consecutive rows of the dense output matrix C , and a group of consecutive rows of the dense input matrix B . As discussed earlier, the accesses to C and A are always local, while the accesses to B can either be local or require a remote data transfer, depending on the c_ids of the processed nonzeros. The matrices are partitioned in the following way:

Megatile. As shown in Figure 57, we logically divide the matrix A into *megatiles* (MT). Given the matrix A with N rows and M columns, and given p nodes in the distributed system, a megatile is formed with N/p consecutive rows and M/p consecutive columns. Node i stores the i^{th} row of megatiles. We logically divide the matrices B and C based on the width and height of a megatile, respectively. Figure 57 shows the breakdown of the matrices for $p = 4$. The chunks of the B and C matrices are distributed across the nodes as shown in the figure, where node i is labeled Ni .

Sparse stripe. Each megatile is further divided into *sparse stripes*, which determine the

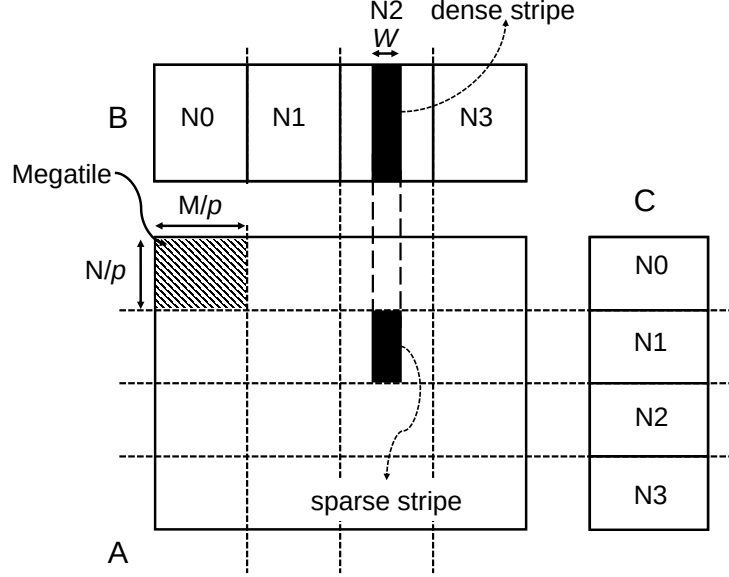


Figure 57: *Two-Face* megatiles and sparse and dense stripes.

communication patterns. A sparse stripe has the same number of rows as a megatile and a fixed number of columns W . We choose to divide megatiles into sparse stripes to allow for partitioning the sparse matrix into two types of regions at a fine granularity. Sparse stripes are classified as *local-input* if their corresponding B rows are owned by the local node; otherwise, they are *remote-input*. A remote-input sparse stripe can either trigger a coarse-grained collective transfer or a fine-grained asynchronous one. At a high level, remote-input stripes requiring many rows of the dense input matrix B will be marked as *synchronous* as, during execution, they will benefit from the coarse-grained collectives. On the contrary, remote-input stripes requiring few rows of B will be marked as *asynchronous* as, during execution, they will benefit from fine-grained asynchronous transfers. During execution, multiple nodes containing synchronous stripes that require the same data from B will participate in the same collective multicast operation to receive that data. It is possible that a “multicast” transfers data to only a single destination node.

Dense stripe. All the sparse stripes that have the same range of c_ids in A access the same group of rows in matrix B . We call this group of dense rows a *dense stripe*. A synchronous sparse stripe will trigger the coarse-grained transfer of a dense stripe using a

collective operation with, potentially, additional destination nodes; an asynchronous sparse stripe will trigger the fine-grained one-sided transfer of individual rows (or groups of adjacent rows) within the dense stripe. These asynchronous transfers will only transfer the rows of the dense stripe that are needed for the computation. If a node does not need any of a dense stripe's rows, that dense stripe will not be communicated to it at all.

In the next sections, we use *stripe* to refer to sparse stripes. Any mention of dense stripes is explicit.

Stripes are classified as asynchronous or synchronous during a preprocessing step using a model that tries to minimize the expected execution time. We present the model in Section 6.3.2. During the actual execution after the preprocessing step, the local threads in an MPI rank operate in parallel and are split into two groups: (1) synchronous threads, which handle the data transfers for the synchronous stripes as well as the computation for synchronous and local-input stripes, and (2) asynchronous threads, which handle the data transfers and computation for the asynchronous stripes. All synchronous communication is completed before any synchronous computation begins. On the other hand, asynchronous communication and computation overlap: a thread may compute on one asynchronous stripe while another thread transfers data for a second asynchronous stripe.

To optimize computation and communication efficiency, the nonzeros in sparse stripes are ordered in row-major order in synchronous stripes and column-major order in asynchronous stripes. In synchronous stripes, the nonzeros are stored in row-major order because this benefits computation. Specifically, a thread can process the nonzeros of a whole row of the stripe and buffer the results in a thread-local buffer before updating C . Then, the thread uses a single synchronization operation to accumulate the contents of the thread-local buffer into the corresponding C row. In asynchronous stripes, the nonzeros are stored in column-major order to benefit communication. Specifically, a column-major format allows a thread to quickly traverse the nonzeros and determine the unique c_ids of the nonzeros in a stripe, which in turn identify the rows of matrix B that need to be

transferred. This comes at the cost of computational inefficiency since this format makes buffering the output of a thread’s computations hard. As a result, the thread must typically use one synchronization operation for each nonzero to accumulate results onto C .

6.3.2 Preprocessing Model

During execution, *Two-Face* will process the asynchronous stripes in parallel with the synchronous and local-input stripes. Consequently, the optimal choice to partition the sparse matrix into synchronous and asynchronous stripes is one that equalizes the execution times of asynchronous stripes and synchronous/local-input stripes. To this end, we create a model of execution based on the following ideas.

For the synchronous stripes, the model assumes that the computation time will be negligible compared to the synchronous communication time. The reason is that the row-major format of the nonzeros lends itself to efficient execution: the output of the nonzeros in a row of the stripe is reused through a thread-local buffer and accumulated into the corresponding C row with a single synchronization operation. In addition, we take advantage of this parallelizability by assigning more parallel threads to the computation of synchronous/local-input stripes than for the asynchronous stripes. For the local-input stripes, since they do not need communication, the model neglects both communication and computation time.

In contrast, the computation time for asynchronous stripes may be significant because the column-major format of the nonzeros lends itself to inefficient execution: thread-local buffers are not used and we need to perform a synchronization operation for every nonzero. In addition, because synchronization may be a bottleneck, we assign fewer threads to the computation of asynchronous stripes than to the others. Therefore, for the asynchronous stripes, the model considers both computation and communication.

We model the cost of synchronous communication ($Comm_S$), asynchronous communication ($Comm_A$), and asynchronous computation ($Comp_A$) for a particular node

as:

$$Comm_S = S_S(\beta_S KW + \alpha_S)$$

$$Comm_A = \beta_A KL_A + \alpha_A S_A$$

$$Comp_A = \gamma_A KN_A + \kappa_A S_A$$

Consider $Comm_S$ first. S_S is the number of synchronous stripes processed by the node, W is the stripe width, and K is the number of columns in the dense matrices. β_S is the cost of synchronous transfer per element of B (i.e., it is inversely proportional to the bandwidth), and α_S is other per-stripe overheads of synchronous transfers.

Next, for $Comm_A$, S_A is the number of asynchronous stripes processed by the node, and L_A is the total number of rows of the dense matrix B transferred for these stripes via fine-grained accesses. β_A and α_A represent the same costs as β_S and α_S but for asynchronous accesses.

Finally, for $Comp_A$, N_A is the total number of nonzeros in the asynchronous stripes processed by the node. γ_A is the computational cost per operation, and κ_A is the additional per-stripe software overhead of asynchronous computation.

The coefficients β_S , α_S , β_A , α_A , γ_A , and κ_A are determined via a linear regression [258] calibration step (details in Section 6.5.2). These parameters are dependent on the system configuration. For example, a system with a large bisection bandwidth should have small β terms. The α terms may be reduced by reducing the round-trip communication latency, including the latency incurred in software libraries/drivers and in the network.

In the optimal case, $Comm_S = Comm_A + Comp_A$, so that there is a perfect overlap of the asynchronous and synchronous components. Defining $S_T = S_S + S_A$ to be the total number of non-local-input stripes processed by a particular node and rearranging this equation gives the following:

$$\begin{aligned}
S_S(\beta_S KW + \alpha_S) &= \beta_A KL_A + \alpha_A S_A + \gamma_A KN_A + \kappa_A S_A \\
\implies (S_T - S_A)(\beta_S KW + \alpha_S) &= K(\beta_A L_A + \gamma_A N_A) + S_A(\alpha_A + \kappa_A) \\
\implies S_T(\beta_S KW + \alpha_S) &= K(\beta_A L_A + \gamma_A N_A) \\
&\quad + S_A(\alpha_A + \kappa_A + \beta_S KW + \alpha_S)
\end{aligned} \tag{4}$$

We now classify the stripes requiring communication as either synchronous or asynchronous. Initially, we assume that all stripes are synchronous, which makes the right-hand side of Equation 4 equal to 0. Then, we take each stripe i and consider classifying it as asynchronous, instead. In this case, the stripe's contribution (call it z_i) to the right-hand side of Equation 4 would be given by:

$$\begin{aligned}
z_i &= v_i + u, \\
v_i &= K(\beta_A l_i + \gamma_A n_i), \\
u &= \alpha_A + \kappa_A + \beta_S KW + \alpha_S,
\end{aligned}$$

where stripe i requires l_i dense rows from matrix B and contains n_i nonzeros. Note that u depends only on the stripe width (W) and other constants, and is therefore constant for all the stripes in the matrix.

To identify the most beneficial stripes to classify as asynchronous, we look for stripes with low values of z_i . This is because a low z_i implies that, if the stripe is classified as asynchronous, it requires few dense rows to be transferred from a remote node and contains few nonzeros. Therefore, its communication and computation costs are relatively low. On the other hand, if the stripe is classified as synchronous, it has a constant communication cost.

Consequently, we sort all of this node's stripes by their z_i in ascending order. Then, we take one stripe at a time, in order, and classify it as asynchronous, until we have taken the first r stripes, where r is the greatest number that satisfies

$$S_T(\beta_S KW + \alpha_S) \geq \sum_{i=0}^{r-1} z_i.$$

The rest of the stripes are classified as synchronous. With this method, the two sides of Equation 4 are approximately equal, which is a necessary condition to attain optimal execution time. Indeed, following this method, the total runtimes of the synchronous and asynchronous stripes in *Two-Face* should be nearly equal, assuming the simplified cost model that we use does hold. Additionally, sorting the stripes by their z_i maximizes the number of stripes classified as asynchronous and minimizes the number of synchronous stripes. Because the cost of communication for a synchronous stripe is constant for a given K and W , this strategy minimizes $Comm_S$ and, therefore, the total cost of the operation.

There are other possible methods of classifying stripes. One such method is to analyze columns of stripes in the sparse matrix and classify a stripe as synchronous when its corresponding dense stripe is needed by many nodes and, therefore, is likely to benefit from optimized multicast operations. We leave the investigation of such methods for future work.

6.4 The Two-Face Algorithm

In this section, we provide greater details about *Two-Face*. We discuss the sparse matrix representation, the *Two-Face* algorithm, its tuning and portability, and its applicability to GNN training.

6.4.1 Sparse Matrix Representation

Two-Face represents the sparse matrix A in a modified COO format. The nonzeros in asynchronous stripes are extracted from A and are stored in an *Asynchronous* sparse matrix; the nonzeros in synchronous/local-input stripes are extracted into a *Synchronous/Local-Input* sparse matrix. Because we use a compressed sparse matrix representation, this format does not significantly increase overall memory use.

Figure 58 shows: (a) an example of an input sparse matrix A , (b) its corresponding

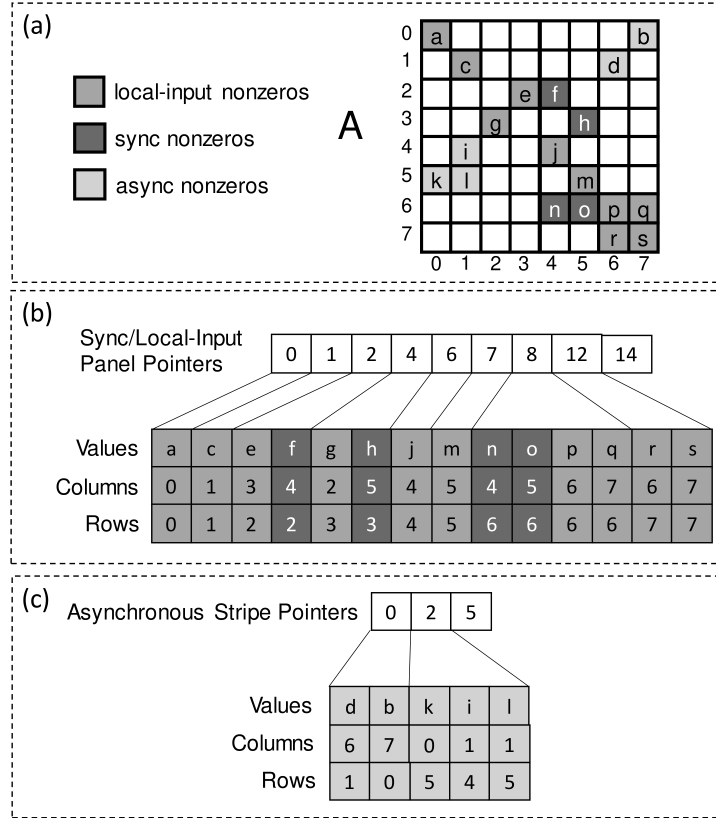


Figure 58: Sparse matrix representation in *Two-Face*: (a) an input sparse matrix A , showing an example of nonzeros classified into the local-input, sync, and async categories; (b) the corresponding synchronous/local-input sparse matrix; and (c) the corresponding asynchronous sparse matrix. This figure assumes a 4-node system, a stripe width of 2, and a row panel height of 1.

synchronous/local-input sparse matrix, and (c) its corresponding asynchronous sparse matrix. The figure assumes that there are four nodes and one sparse stripe for each 2x2 megatile. Assume that, after running the preprocessing step, the stripes have been classified as local-input, synchronous, and asynchronous such that the nonzeros in A end up in the categories shown in Figure 58a.

The corresponding synchronous / local-input sparse matrix (Figure 58b) organizes the synchronous/local-input nonzeros in a row-major order structure. The elements in this structure are divided into row panels—e.g., nonzeros e and f in Figure 58 are in one row panel. Row panels are the units of work assigned to threads *computing* on

synchronous/local-input nonzeros. In Figure 58b, these panels are one row tall, and an array of *Synchronous/Local-input Panel Pointers* points to the beginning of each panel.

The corresponding asynchronous sparse matrix (Figure 58c) organizes the asynchronous nonzeros within stripes in a column-major order structure. The order of the stripes themselves is row-major, to simplify the distribution of the asynchronous sparse matrix across nodes at runtime. An array of *Asynchronous Stripe Pointers* points to the beginning of each asynchronous stripe.

At runtime, each node will only store those portions of the synchronous/local-input and asynchronous sparse matrices that are relevant to its computation. In addition, for each dense stripe of B , the preprocessing step generates metadata containing a list of nodes that are destinations of the collective transfer of that stripe. At runtime, this metadata is replicated across all nodes.

6.4.2 *Two-Face* Algorithm Description

The algorithm consists of three parts: top-level algorithm, processing synchronous row panels, and processing asynchronous stripes. We describe each part in turn.

6.4.2.1 Top-Level Algorithm

Algorithm 1 shows the top-level operation of the *Two-Face* algorithm. All nodes of the distributed system execute Algorithm 1 in parallel. First, the node initializes a flag and two atomic queues for work-sharing (Lines 2-3). These queues provide indices of asynchronous stripes (*async_q*) and indices of row panels (*sync_q*). In the example of Figure 58, the *sync_q* of N2 is $\{4, 5\}$, which are the indices of the two pointers in the Sync/Local-Input Panel Pointers array used by N2 (pointing to rows containing nonzeros j and m). The *async_q* of N2 is $\{1\}$, which is the index of the pointer in the Asynchronous Stripe Pointers array that points to the asynchronous stripe assigned to N2.

Then, the code starting at Line 4 is executed by all the threads of the node in parallel.

Algorithm 1 Top-Level Two-Face Pseudo-code.

```
1: procedure DISTSPMM( $A, B, C$ )
2:    $sync\_transfer\_done \leftarrow \text{False}$ 
3:    $async\_q, sync\_q \leftarrow \text{InitQueues}(A)$ 
4:   DoParallel
5:     if  $tid = 0$  then ▷ Sync Transfers
6:       TransferDenseStripes( $A, B$ )
7:        $sync\_transfer\_done \leftarrow \text{True}$ 
8:     end if
9:     if  $tid \in AsyncThreads$  then ▷ Async Processing
10:      while  $async\_q.nonempty()$  do
11:         $n \leftarrow async\_q.pop()$ 
12:        ProcessAsyncStripe( $A, B, C, n$ )
13:      end while
14:    end if
15:    WaitForFlag( $sync\_transfer\_done$ )
16:    while  $sync\_q.nonempty()$  do ▷ Sync Compute
17:       $n \leftarrow sync\_q.pop()$ 
18:      ProcessSyncRowPanel( $A, B, C, n$ )
19:    end while
20:  EndParallel
21: end procedure
```

Specifically, Thread 0 initiates the transmission/reception of the dense stripes needed for synchronous operations (Lines 5-8). Data transmission is implemented as non-blocking, but data reception is blocking. These transfers are done via a series of calls to `MPI_Bcast`. The destination nodes are determined via metadata produced by the preprocessing step, and these transfers occur at the stripe granularity.

In parallel, all the threads assigned to asynchronous stripes begin processing those stripes (Lines 9-14). Once all needed dense input data from collectives has been received (Line 15), all threads (including the asynchronous ones after they have processed the asynchronous stripes) process the synchronous row panels (Lines 16-19).

6.4.2.2 Processing Synchronous Row Panels

Algorithm 2 describes the processing of a row panel. The operation starts by initializing a thread-local *Accumulation Buffer* to zero (acc in Line 2) and reading the row panel ($panel$

in Line 3). Then, the algorithm iterates through all of the nonzeros in the row panel, accumulating each result onto acc (Line 10). When we either complete a row of nonzeros (Line 7) or complete the whole row panel (Line 13), we add acc to the corresponding row of C . Atomics are required in this operation because some threads operating on asynchronous stripes may also be writing to the same rows of C .

Algorithm 2 Two-Face Sync Compute Pseudo-code

```

1: procedure PROCESSSYNCROWPANEL( $A, B, C, n$ )
2:    $acc \leftarrow \{0, \dots, 0\}$  ▷ Output row buffer
3:    $panel \leftarrow A.panel\_ptrs[n]$ 
4:    $prev\_row \leftarrow panel[0].row$  ▷ Initialize to first row
5:   for  $nz \in panel$  do
6:     if  $nz.row \neq prev\_row$  then
7:       AtomicAdd( $C[prev\_row], acc$ )
8:        $acc \leftarrow \{0, \dots, 0\}$ 
9:     end if
10:     $acc \leftarrow acc + nz.val * B[nz.col]$ 
11:     $prev\_row \leftarrow nz.row$ 
12:  end for
13:  AtomicAdd( $C[prev\_row], acc$ )
14: end procedure

```

6.4.2.3 Processing Asynchronous Stripes

Algorithm 3 shows the algorithm to process an asynchronous stripe. A thread reads the asynchronous stripe ($stripe$ in Line 2) and iterates over the nonzeros in the stripe to identify the unique c_ids of the nonzeros (Line 3). These determine the indices of the dense rows from B that are required. The asynchronous thread then initiates the remote access of the dense rows by calling `GetRemoteRows` (Line 4). This procedure uses `MPI_Rget` and a custom MPI datatype defined with `MPI_Type_indexed` to select only the rows of interest for the transfer.

Once the dense rows arrive, they are stored in $drows$ (Line 4), and multiple threads begin computing on them. Each thread processes a subset of the nonzeros in the sparse stripe. Each nonzero is multiplied with the corresponding row of $drows$ and accumulated

Algorithm 3 Two-Face Async Pseudo-code

```
1: procedure PROCESSASYNCSTRIPE( $A, B, C, n$ )
2:    $stripe \leftarrow A.async\_stripe\_ptrs[n]$ 
3:    $drow\_ids \leftarrow stripe.UniqueColIDs()$ 
4:    $drows \leftarrow GetRemoteRows(drow\_ids)$ 
5:   for  $nz \in stripe$  do in parallel
6:     AtomicAdd( $C[nz.row], nz.val * drows[nz.col]$ )
7:   end for
8: end procedure
```

into C (Line 6). Atomics are required for correct accumulation into C , just as in the synchronous stripe case. However, since asynchronous nonzeros are stored in column-major order, we cannot easily use thread-local buffers to reduce the number of atomics.

To reduce transfer overheads, inside the `GetRemoteRows` routine, we coalesce the transfer of nearby rows of B . For example, if a sparse stripe requires B rows $\{2, 3, 6, 8\}$, we transfer three groups of rows, with $(offset, size)$ pairs equal to $\{(2, 2), (6, 1), (8, 1)\}$. This optimization reduces software overheads. For small K , we also coalesce rows separated by unused rows, potentially reducing the software overhead further, but transferring some useless data. Using the example from before, we might transfer groups of rows $\{(2, 2), (6, 3)\}$, retrieving one unnecessary row (row 7).

6.4.3 Tuning Knobs and Portability

Two-Face has several parameters that may need to be calibrated for each individual system to achieve maximal performance. Among these are the coefficients used in the preprocessing cost model $(\beta_S, \alpha_S, \beta_A, \alpha_A, \kappa_A, \gamma_A)$. As mentioned before, in our evaluation, we determine the values of these coefficients via linear regression on a small number of workloads. These parameters only need to be calibrated once for a system, possibly at installation time.

In addition to the preprocessing cost model coefficients, the runtime algorithm is parameterized by the number of threads assigned to sync/async stripe processing, the aggressiveness of row coalescing in async stripe transfers, the height of the row panels used

for computation in the sync stripes, and the width of the stripes. The optimal choice for these parameters may vary between systems and workloads, but we show in Sections 6.5.2 and 6.6 that choosing reasonable, static values can provide good performance. In practice, these parameters could be determined at installation time similarly to the preprocessing coefficients.

Thus, although *Two-Face* relies on knowledge of system characteristics to make decisions about how to schedule the work, porting to a new system just requires a one-time profiling step during installation.

6.4.4 Applicability to GNN Training

While SpMM is used in a variety of domains, one of the most important ones is GNN training. GNN training is often done in relatively small-scale systems, where the amount of memory is a limitation. To alleviate this problem, GNN training algorithms have recently resorted to the use of sampling [64] and mini-batching. While these techniques reduce the memory footprint requirements, they may introduce some inaccuracy [64], [259], and may introduce runtime overhead which can sometimes increase the end-to-end execution time [92]. In *Two-Face*, like in most of the prior work in distributed GNN training [259], [260], we are less concerned about the lack of memory because we can use the full aggregate memory of a very large cluster [260]. As a result, we do not consider sampling or mini-batching and, instead, support full-graph GNN training.

It is interesting, however, to consider the application of *Two-Face* to an environment with sampling or mini-batching. In principle, in its current form, *Two-Face* is incompatible with sampling or mini-batching. This is because, in sampling or mini-batching algorithms, different iterations of the SpMM computation use a different reduced (or sampled) matrix. As a result, *Two-Face* would have to re-run the preprocessing step every time the reduced matrix changes.

Future work may involve adapting *Two-Face* to apply to GNNs with sampling. One

possible approach may involve making preprocessing decisions offline once, based on the expected stripes’ densities, given knowledge of the sampling to be done at runtime. Then, stripes that are expected to be dense enough even after sampling would still be classified as synchronous, and the other stripes would be classified as asynchronous. At runtime, the graph would still be stored as shown in Figure 58, but with the addition of masks to filter nonzeros eliminated by the sampling at each iteration.

In current full-graph GNN training [52], [63], the preprocessing cost can be easily amortized. We will quantify the exact cost of the preprocessing step in Section 6.6.3. In GNN training, the same sparse matrix is used for hundreds or even thousands of SpMM iterations. Additionally, in many GNN applications, the same graph is used for both training and inference. This is the assumption in most GNNs for semi-supervised node classification applications [52], [63]. Since the sparse matrix does not change, the preprocessing done in training can be reused for inference. For these reasons, the overhead of *Two-Face* preprocessing in full-graph GNN training is negligible.

6.5 Methodology

Here, we describe our evaluation methodology. We begin with details about the hardware configuration and software libraries used to evaluate *Two-Face*, as well as the sparse matrices used as benchmarks. Then, we discuss how we determined the values of various parameters. Finally, we describe the other algorithms which we use as baselines to compare to *Two-Face*.

6.5.1 Overview

We evaluate *Two-Face* and multiple baseline algorithms using large matrices on Delta [261], a supercomputer at the National Center for Supercomputing Applications (NCSA). We use up to 64 CPU nodes, with a default of 32 CPU nodes. Each Delta node is a dual-socket system with two 64-core AMD EPYC 7763 processor chips running at 2.45

GHz and a total of 256 GiB of DRAM. The nodes are connected through a Cray Slingshot interconnect [262].

We build on the code published by Bharadwaj et al. [74], adapting it as necessary to support our algorithms and larger matrices. We use hybrid OpenMP / MPI programming, with one MPI rank and 128 OpenMP threads per node. We use OpenMP 4.5 [263] and Open MPI 4.1.2 [264], [265] with UCX 1.11.2 [266]. All of the baseline algorithms use the Intel Math Kernel Library [236] (version 2022.0.2) for local SpMM computations. These baselines also rely on CombBLAS [267] for I/O. Our implementation of *Two-Face* handles I/O by way of custom data loaders for our preprocessed sparse matrix format. All algorithms used in these experiments make use of Eigen [268] for handling dense matrices locally.

We use eight large sparse matrices from SuiteSparse [73], described in Table 19. These matrices are derived from a variety of domains, including internet traffic, social networks, web crawls, and scientific applications.

Table 19: Matrices used in the evaluation. All matrices are among the largest in SuiteSparse [73] and are square. Stripe widths are chosen to scale with the number of columns.

Matrix Name Long	Matrix Name Short	# Rows (Mill)	# Nonzeros (Mill)	Stripe Width
mawi_201512020030	mawi	68.86	143.41	128K
Queen_4147	queen	4.15	316.55	8K
stokes	stokes	11.45	349.32	32K
kmer_V1r	kmer	214.01	465.41	512K
arabic-2005	arabic	22.74	640.00	64K
twitter7	twitter	41.65	1,468.37	128K
GAP-web	web	50.64	1,930.29	128K
com-Friendster	friendster	65.61	3,612.13	128K

Our evaluation in Section 6.6 supports the claim that distributed SpMM is typically a communication-bound workload. Thus, we expect that extending *Two-Face* to other computing hardware would provide similar results. For example, using GPUs in the nodes may accelerate the local computation, but communication will remain a bottleneck. Thus, we expect that *Two-Face* will still see speedups if used with GPUs. Here, we evaluate a CPU implementation.

6.5.2 Two-Face Parameterization

Two-Face is a parameterizable algorithm. To determine its parameters for our system, we analyzed several combinations of parameters on a small set of workloads.

To determine the appropriate width of stripes, we analyzed the performance of SpMM using the queen, arabic, and twitter matrices with various choices for W . There was increasing overhead in both the preprocessing and runtime steps as the number of stripes grew, suggesting that the stripe width should not be made too small, relative to the size of the matrix. We decided to scale the stripe width proportionally to the dimensions of the matrices, rounding to the nearest power of two. Table 19 shows the stripe widths we chose.

All run-time parameters other than the stripe width are held constant across matrices. Table 20 shows these parameters. Each node runs 128 OpenMP threads. Since a large number of one-sided transfers results in high resource contention, we limit the number of threads communicating asynchronous data to 2 per node. We allow each of these threads to fork up to four ways (for a total of 8 threads) when computing on the asynchronous stripes. We dedicate the remaining 120 threads in the node to computation on the synchronous and local-input stripes. We define the maximum row coalescing distance for asynchronous transfers to be proportional to $\frac{1}{K}$, since the cost of transferring unnecessary dense rows grows with K .

Table 20: Constant runtime parameters used in *Two-Face*.

Parameter Name	Value
Async Communication Threads per Node	2
Async Computation Threads per Node	8
Sync/Local-Input Computation Threads per Node	120
Max Async Coalescing Distance	$(127/K) + 1$
Row Panel Height of Sync/Local-Input Sparse Matrix	32 rows

To determine the values of the preprocessing parameters used in stripe classification (Section 6.3.2), we employ linear regression [258]. We collect data by processing the twitter matrix [269] using $K = 32$, $p = 32$, and nine different combinations of stripe widths and asynchronous/synchronous stripe classifications. The number of samples is kept small to

ensure that it is reasonable to calibrate these coefficients when installing *Two-Face* on a new system. The derived coefficient values, which we use when preprocessing all matrices in our evaluation (unless otherwise specified), are shown in Table 21.

Table 21: Coefficient values used in the preprocessing of matrices. The β parameters relate to the system bandwidth, the α parameters relate to other communication overheads, and the γ & κ terms relate to computational throughput and other overheads.

Coefficient	Experimental Value
β_S	1.95×10^{-10}
α_S	1.36×10^{-6}
β_A	3.61×10^{-9}
α_A	1.02×10^{-5}
γ_A	2.07×10^{-8}
κ_A	8.72×10^{-9}

These coefficients provide some insight into the performance difference between one-sided asynchronous and collective synchronous communication. For example, they suggest that asynchronous transfers are more expensive per transferred element of B than synchronous transfers by a factor of $\beta_A/\beta_S \approx 18.5$.

In Section 6.6.4 of our evaluation, we evaluate the impact of different values of these coefficients.

6.5.3 Algorithms Evaluated

In our evaluation, we compare *Two-Face* to other algorithms shown in Table 22. All the algorithms use 1D partitioning. We divide the dense input matrix B into as many equally-sized portions as the number of nodes p , and call each portion a “block”. The B matrix is distributed across all nodes, where each node stores a single block.

Table 22: SpMM algorithms being compared.

Algorithm Name	MPI Transfer Operations
Dense Shifting [74]	MPI_Allgather, MPI_Sendrecv
Allgather	MPI_Allgather
Async Coarse-Grained	MPI_Get
Two-Face	MPI_Rget, MPI_Ibcast
Async Fine-Grained	MPI_Rget

Dense Shifting (DS) is a synchronous SpMM algorithm that has been investigated by Bharadwaj et al. [74] and found to be highly competitive compared to other state-of-the-art implementations. We use it as our main baseline. DS begins by using `MPI_Allgather` to replicate a certain number of blocks in each node, as determined by a replication factor c . It then continues by shifting the replicated blocks cyclically via `MPI_Sendrecv` after each computation step. For instance, with $c = 4$, this algorithm replicates each block such that each node holds four blocks at a time. It then performs p/c computation and shifting steps to complete the SpMM operation. In our experiments, we evaluate this algorithm for $c = 2$, $c = 4$, and $c = 8$, and refer to these settings as DS2, DS4, and DS8, respectively.

The next two algorithms replicate all or nearly all of the matrix B before beginning the computation. In *Allgather*, each node uses `MPI_Allgather` to broadcast its block of B to all others and receive theirs in turn. In *Asynchronous Coarse-Grained*, each node uses `MPI_Get` to obtain the blocks that it needs for its computation. In both cases, substantial memory has to be allocated, creating issues as the problem size scales.

Two-Face is the algorithm we propose. We use the parameters as described before. However, if the preprocessing algorithm determines that the chosen sync/async classification of stripes would result in too much memory consumption in one or more nodes during SpMM execution, it will classify additional stripes as async until the expected memory consumption in those nodes is feasible.

Asynchronous Fine-Grained is implemented in the same way as *Two-Face*, except that all stripes are asynchronous. This algorithm is used as an extreme example to illustrate the tradeoffs made by a balanced *Two-Face* implementation. This baseline was used in Section 6.2.

All algorithms are evaluated by averaging out the time of 5 consecutive SpMM operations. By default, our experiments use $p = 32$ and $K = 128$. Some experiments use $K = 32$ or $K = 256$, and others use $p = 1, 2, 4, 8, 16, 32$, or 64 .

6.6 Evaluation

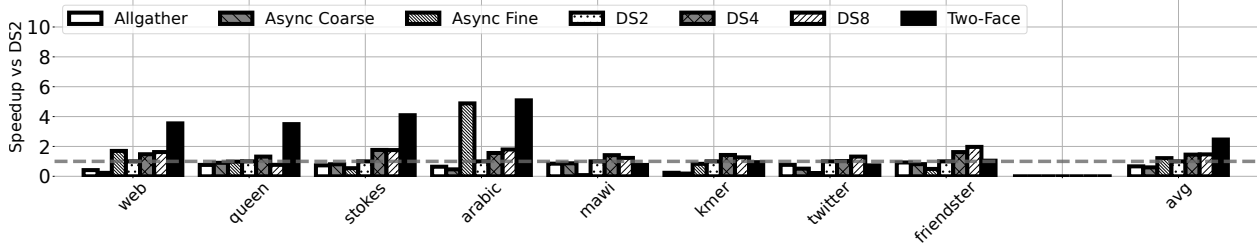


Figure 59: Speedups of various SpMM algorithms over DS2 for $K = 32$.

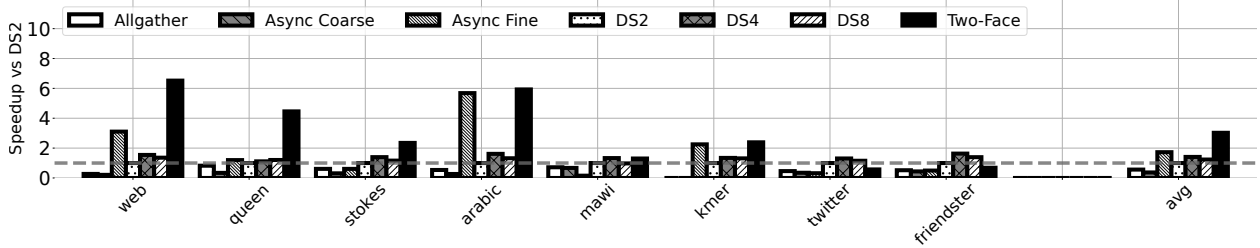


Figure 60: Speedups of various SpMM algorithms over DS2 for $K = 128$.

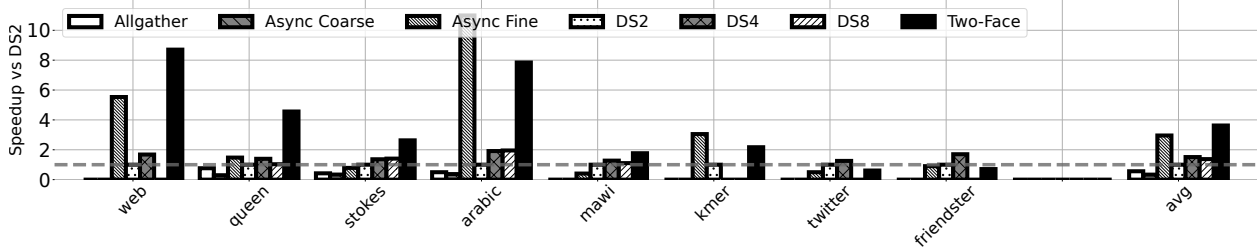


Figure 61: Speedups of various SpMM algorithms over DS2 for $K = 512$.

In this section, we evaluate *Two-Face*. First, we compare the performance of *Two-Face* to the various baselines and discuss any bottlenecks observed. Next, we discuss the scaling behavior of *Two-Face* as we vary the number of nodes in the system. Finally, we analyze the preprocessing cost of *Two-Face* and the sensitivity of *Two-Face* to the choice of the preprocessing parameters.

6.6.1 Comparing *Two-Face* to Various Baselines

Figures 59, 60, and 61 show the speedups of *Two-Face* and the other SpMM algorithms over DS2 for $K = 32$, $K = 128$, and $K = 512$, respectively. We normalize to DS2 because,

Table 23: Absolute execution times of DS2 and *Two-Face* for the experiments in Figures 59, 60, and 61. The numbers are the average of five SpMM operations.

		web	queen	stokes	arabic	mawi	kmer	twitter	friendster
K=32	DS2 (seconds)	1.97	0.28	0.96	1.32	6.46	6.33	4.17	8.79
	Two-Face (seconds)	0.56	0.08	0.23	0.26	8.50	6.70	5.71	8.41
K=128	DS2 (seconds)	7.198	0.86	2.22	3.85	19.78	35.77	11.57	20.61
	Two-Face (seconds)	1.10	0.19	0.94	0.65	15.18	14.98	20.24	30.08
K=512	DS2 (seconds)	38.86	2.89	9.34	21.46	97.95	136.21	52.77	83.02
	Two-Face (seconds)	4.46	0.634	3.552	2.74	55.40	62.77	86.62	117.31

unlike DS4 or DS8, DS2 does not run out of memory for any matrices or value of K in our evaluation. From the figures, we see that, on average, across matrices and K values, *Two-Face* is the fastest algorithm, and delivers substantial speedups.

As K increases, the advantage of *Two-Face* over the dense shifting algorithms becomes more prominent. This is because the cost of transferring unnecessary rows in the dense shifting algorithms increases with K , providing a greater advantage to the fine-grained one-sided accesses of *Two-Face*. At $K = 32$, *Two-Face*'s average speedup over the dense shifting algorithm with the best choice of replication factor for each individual matrix is 1.53x. At $K = 128$, the same speedup is 2.11x, and at $K = 512$, is it 2.35x. The average speedup across all values of K shown here is 1.99x.

The *Async Fine* and dense shifting algorithms are on average faster than the *Async Coarse* and *Allgather* algorithms. Dense shifting is sometimes unable to run with higher replication factors due to memory constraints. For example, for $K = 512$, DS8 fails to run for half of the matrices, and DS4 fails in one matrix. As a reference, Table 23 provides the absolute execution times of *Two-Face* and DS2 in these figures.

The figures also show that the speedups (or slowdowns) are highly dependent on the matrix. For example, *Two-Face* is not the fastest algorithm for twitter and friendster and, for $K = 32$, additionally for mawi and kmer. To understand this behavior, Figure 62 breaks down the total execution time of DS4 and *Two-Face* for each matrix for $K = 128$. For *Two-Face*, we break down the execution time into *Sync Comp*, *Sync Comm*, *Async Comp*, and *Async Comm*. We stack the Sync components in the left bar and the Async

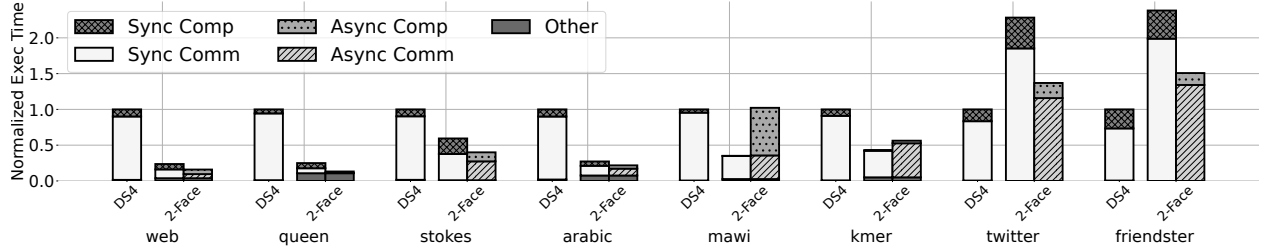


Figure 62: Breakdown of the total execution times of DS4 and *Two-Face* for $K = 128$. *Two-Face*'s time is divided into synchronous and asynchronous components (left and right bars, respectively), which operate in parallel. These are further broken down into computation (*Comp*) and communication (*Comm*). DS4 only has a Sync component. The *Other* category mainly consists of the initial setup of data structures for MPI. Execution times are normalized to DS4.

components in the right bar, and show both bars side-to-side, since the execution time is equal to the highest of the two bars. *Two-Face* also has some *Other* overheads, which mainly consist of initializing necessary MPI structures before the main communication/computation begins. For DS4, only *Sync Comp* and *Sync Comm* are relevant. For each matrix, the bars are normalized to DS4.

We see that the dominant contributor to DS4's execution time is its communication. *Two-Face* is able to attain significant speedups over DS4 by reducing the amount of communication through fine-grained accesses. In five of the matrices, we can see that the sum of the communication time spent by *Two-Face* in *Sync Comm* and *Async Comm* is significantly less than the amount of time spent by DS4 in its communication.

Two exceptions are twitter and friendster. In these matrices, *Two-Face*'s *Sync Comp* and *Sync Comm* have both increased over DS4, despite the fact that less data is being transferred. We note that *Two-Face*'s synchronous broadcast operations are significantly slower than the cyclic shifting operations in DS4 when a large portion of the input dense matrix is required by many nodes. When *Two-Face* operates on a matrix like friendster, each node participates in many more MPI calls than it does if dense shifting is used, due to the finer granularity of the transfers.

An interesting case is mawi, where *Two-Face* is unable to reduce the execution time

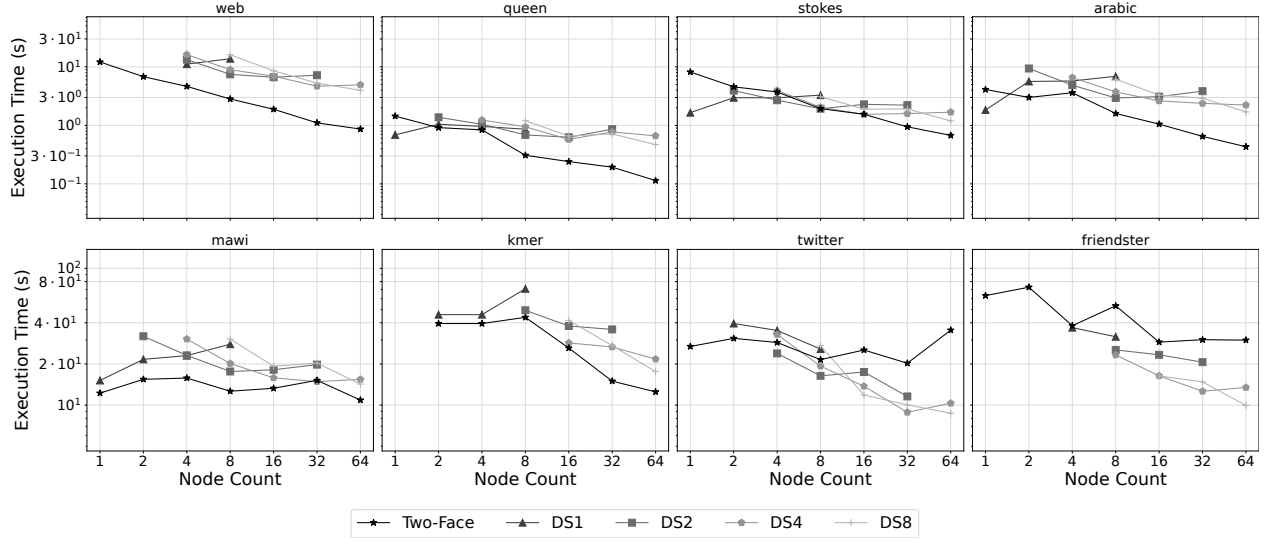


Figure 63: Execution time of *Two-Face* and the dense shifting algorithm with different replication factors (DS1, DS2, DS4, and DS8) as the number of nodes changes. The K value is 128. Some data points for the dense shifting algorithm are missing because they need too much memory or take too long to execute. Both axes in the plots use a logarithmic scale.

over DS4 because of the cost of asynchronous computation. The mawi sparse matrix has regions that have a relatively high density of nonzeros. Computing on such asynchronous stripes is likely expensive due to the heavy use of atomics, as the nonzeros are organized in column-major order. During this work, we conducted initial tests into storing the nonzeros in row-major order instead. However, this change did not result in faster execution, as the cost of identifying which columns contained nonzeros (and, therefore, which dense rows were required) became drastically higher.

6.6.2 *Two-Face* Strong Scaling

Figure 63 shows the execution times of *Two-Face* and the dense shifting algorithm with different replication factors (DS1, DS2, DS4, and DS8) as we scale the number of nodes from 1 to 64. There is a plot for each matrix and both axes are in logarithmic scale. Some data points are missing, since some workloads either exceed the memory capacity of one or more nodes (at small node counts or high replication factors) or take too long to run.

The figure shows that, in most of the matrices, *Two-Face* scales well with the number

of nodes and, in fact, as well or better than the dense shifting algorithm. The exceptions are mawi, twitter, and, to a lesser extent, friendster. With mawi, none of the algorithms scale particularly well due to the high load imbalance across nodes induced by the matrix. With twitter and friendster, we saw in Figure 62 that *Two-Face* is impacted by inefficient synchronous communication. This is the reason for the worse scaling performance.

To understand the behavior of twitter and friendster better, we profile the collectives in the 64-node runs. We measure the number of recipients of each multicast operation. On average, this number is 35.7 for twitter and 43.5 for friendster. In contrast, the matrix with the next largest average recipient count is kmer, with an average of only 5.7. It appears that the large collectives needed for *Two-Face* in twitter and friendster are responsible for the inefficient execution and limited scaling. This effect does not appear at low node counts, where the execution is primarily bottlenecked by local computation, but it dominates at high node counts. Future work should investigate methods to reduce the size of collectives in the algorithm or the design of more regular data movement patterns for the synchronous stripes.

Overall, the performance of *Two-Face* improves as we scale from 1 to 64 nodes by 7.47x on average, with a best-case speedup of 12.12x for queen and a worst-case of 0.76x for twitter. Moreover, compared to the dense shifting algorithm with the optimal replication factor, *Two-Face* sees an average speedup ranging from 1.25x at 4 nodes to 2.21x at 64 nodes.

6.6.3 *Two-Face* Preprocessing Cost

Two-Face requires a preprocessing step that involves, mainly: (1) running our model to classify the stripes into synchronous and asynchronous, and (2) creating the asynchronous and the synchronous/local-input sparse matrices. In this section, we give an idea of the execution time of the preprocessing step. Note that we have not fully optimized it; in particular, we have not parallelized it across multiple nodes. Therefore, the numbers

reported are a pessimistic bound.

Table 24: The overhead of preprocessing in *Two-Face*, normalized to the cost of a single SpMM operation.

Matrix	$t_{norm_I/O}$	t_{norm}
web	428.74	102.00
queen	302.55	23.60
stokes	116.70	11.18
arabic	180.35	36.57
mawi	2.58	1.50
kmer	6.16	3.25
twitter	17.89	7.29
friendster	19.81	8.79
Average	134.35	24.27

Table 24 shows the overhead of the preprocessing step for 32 nodes and $K=128$ for each matrix. Column 2 shows $t_{norm_I/O}$, which is the time of the preprocessing step normalized to the time of one SpMM operation. On average, $t_{norm_I/O}$ is 134.35. However, the preprocessing step is dominated by I/O time, as the original sparse matrix is read from the file system in a textual Matrix Market format [270] and the final asynchronous and synchronous/local-input sparse matrices are written to the file system in a bespoke binary format.

Since in many realistic environments, this I/O will not be present, Column 3 shows the more relevant t_{norm} , which is the preprocessing overhead without I/O normalized to the time of one SpMM operation. In this case, the numbers have reduced substantially. We see that t_{norm} ranges from 1.50 to 102.00, with an average of 24.27. For $K = 512$ (not shown in the table), the average value of t_{norm} is 6.15.

From these numbers, we see that the cost of the preprocessing step can be easily amortized. For the matrices where *Two-Face* demonstrates a speedup over dense shifting when $K = 128$, an average of only 15 SpMM operations need to be performed by *Two-Face* to already see a speedup when including preprocessing time. For $K = 512$, this decreases to only 3 SpMM operations, on average. In contexts such as GNN training, with hundreds of epochs, we can expect to perform many more SpMM operations with the same matrices

than these numbers. In addition, the preprocessing step from training may be reusable during inference.

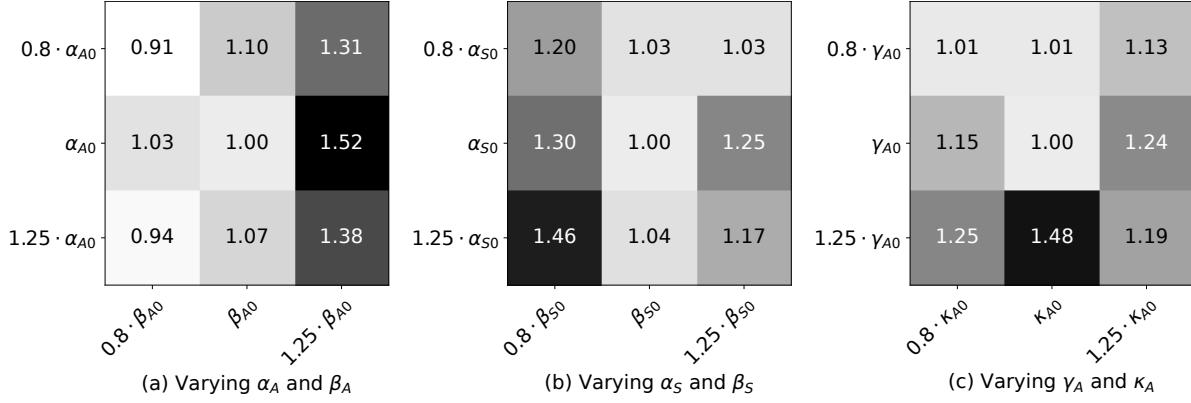


Figure 64: Sensitivity of *Two-Face*'s execution time to the values of the parameters of the execution model used during the preprocessing step. The default values of these parameters, as set in Section 6.5.2 and used in all the earlier experiments, are represented with the 0 subscript (e.g., α_{A0}).

6.6.4 Sensitivity to Parameter Values of the Preprocessing Model

The model of execution that we use during preprocessing (Section 6.3.2) uses parameters α_A , β_A , α_S , β_S , γ_A , and κ_A . In Section 6.5.2, we used linear regression to set their default values. We used such values in all the experiments so far.

In this section, we change the values of these parameters, repeat the experiments, and measure the changes in *Two-Face*'s execution time. We perform three sets of changes. In the first one, we vary α_A and β_A , keeping the other parameters unchanged. Specifically, if α_{A0} and β_{A0} are the default values of α_A and β_A , we consider all combinations of $\{0.8 \cdot \alpha_{A0}, \alpha_{A0}, 1.25 \cdot \alpha_{A0}\} \times \{0.8 \cdot \beta_{A0}, \beta_{A0}, 1.25 \cdot \beta_{A0}\}$. In the second set of changes, we vary α_S and β_S in the same way, keeping the other parameters unchanged. Finally, we vary γ_A and κ_A , again keeping the others unchanged.

Figure 64 shows the outcome of the three sets of changes for the average of three representative matrices: web (*Two-Face*'s best case), twitter (*Two-Face*'s worst case), and stokes (*Two-Face*'s median case). For example, Figure 64a corresponds to the experiments

varying α_A and β_A . The number in each box is *Two-Face*'s execution time with the new parameters relative to *Two-Face*'s execution time with the default parameters. For example, if we use $0.8 \cdot \alpha_{A0}$ and $1.25 \cdot \beta_{A0}$, the *Two-Face*'s execution time becomes 1.31x higher than when using the default values.

Overall, the figure shows that using the default parameters obtained using linear regression is a good choice. Changes to the parameter values typically end up increasing *Two-Face*'s execution time. The execution time decreases in only two cases, and the decrease is small.

6.7 Related Work

Distributed SpMM: Existing work on distributed SpMM is rather limited, but there are recent works exploring the topic. Bharadwaj et al. [74] investigate distributed SpMM, SDDMM, and methods of fusing the two for machine learning applications. The implementations of dense shifting in our evaluation originate from their work. Additionally, Bharadwaj et al. [74] present a sparse shifting implementation. In our work, we did not evaluate their approach, since it partitions the dense input and output matrices in a way that requires additional all-to-all communication for GNNs or other applications that interleave SpMM with a row-wise operator. Bharadwaj et al. [74] compare their implementations to the SpMM provided by PETSc [256]. Selvitopi et al. [75] investigate multiple algorithms for SpMM, including algorithms that use bulk-synchronous collective communication and algorithms that use one-sided asynchronous RDMA communication. They do not, however, investigate combining these communication primitives in a single algorithm.

GNN Training: Prior work has addressed the issue of large graphs in GNN training via sampling techniques [64]. However, the benefits of sampling can come at a cost to accuracy, leading prior work to investigate full-batch distributed GNN training [259], [260]. However, in Tripathy et al. [260], dense matrices used in SpMM operations are only transferred in a

coarse-grained sparsity-unaware fashion. Conversely, Jia et al. [259], using Lux [271], assume a GNN runtime that operates via pushing/pulling node embeddings in a fine-grained manner. This is distinct from *Two-Face*, which uses a combination of coarse and fine-grained transfers to leverage the benefits of both approaches.

Non-Distributed Sparse Kernels: SpMM optimization has been the topic of several investigations. Works such as WACO [272], WISE [140], and DDB [88] attempt to optimize sparse computations by using machine learning techniques to predict the performance of various configurations. Many CPU and GPU tiling techniques and implementations have been published [66], [87], [100], [273]. Other sparse kernels have also been subject to several investigations aiming to tame irregular access patterns [274], [275], [276], [277]. These optimizations for non-distributed kernels may be applicable to the distributed case, but they tend to assume a shared memory system, and they are largely orthogonal to our work.

Other Distributed Sparse Kernels: Other distributed sparse kernels have recently received attention. CombBLAS [267] is a library for distributed sparse kernels such as SpGEMM and SpMV. CombBLAS provides a number of GPU SpMM implementations using different partitioning and communication patterns. All of them use sparsity-unaware collectives. In contrast, *Two-Face* uses a hybrid approach. Hussain et al. [278] investigate communication-avoiding algorithms for SpGEMM. DGCL [279] is a library for distributed GNN training that partitions graphs and processes GNN computations at the level of nodes in the graphs, without explicitly expressing the computation with SpMM operations.

Domain-specific Architectures and Network Support: Several architectural designs that offer hardware support for SpMM computation have been recently proposed [79], [80], [166], [280]. One example is SPADE, presented in Chapter 3. We believe that algorithms such as *Two-Face* can be useful in orchestrating the communication in scaled-up multi-node versions of these accelerators or for other large-scale graph analytics architectures [91], [106], [281]. In addition, we believe that such algorithms can also be beneficial for inter-cube or inter-chip communication in PIM-based architectures for graph

analytics [282], [283], [284], [285]

Finally, scheduling algorithms for collectives such as Themis [286] have been proposed to maximize the bandwidth utilization of multidimensional, heterogeneous networks. These works could inspire network hardware support for *Two-Face*, but one would also require innovations to support the asynchronous communication operations.

6.8 Conclusion

Sparse matrices often contain regions that are denser and regions that are sparser. Based on the observation, this chapter presented *Two-Face*, a software algorithm for distributed SpMM that, leveraging a preprocessing model, performs collective communications for the denser regions, and fine-grained one-sided communications for the sparser regions.

Two-Face attains an average speedup of 2.11x over dense shifting when evaluated on a 4096-core supercomputer. Additionally, *Two-Face* scales well with the machine size.

Two-Face suggests that distributed sparse algorithms should be input-matrix aware, in that different sections of a sparse input matrix prefer using different communication methods. The algorithms should also be communication-oriented, since minimizing communication is a first-class concern.

With simple modifications, the *Two-Face* algorithm should also be applicable to sparse kernels such as Sampled Dense-Dense Matrix Multiplication (SDDMM), which exhibits very similar patterns to SpMM. Likewise, with proper parameter tuning, *Two-Face* may also be applicable to accelerate SpMV, which is a special case of SpMM.

Chapter 7 Hardware Acceleration of Distributed Sparse Computations

7.1 Introduction

Over the last few years, there has been significant interest in the development of hardware accelerators for sparse computations [79], [80], [81], [91], [131], [132], [202], [287]. Although these architectures are effective, they mostly target a single node. When these accelerators are placed on each node of a cluster, computation gets sped-up, and cluster-level programs can become heavily bound by network communication. One could optimize network communication with hardware support, but the research in this area has focused on supporting distributed *dense* kernels, such as those in dense machine learning [288]. Network-centric hardware optimizations for distributed *sparse* computations remain underexplored.

Sparse communication algorithms typically follow one of two approaches (Section 2.3). In the *Sparsity-Unaware* (SU) approach, communication is scheduled as if the kernel was dense. Data is transferred in coarse granularities using all-to-all transfers (e.g., collectives [75]), disregarding whether every data item is actually needed by the requester. As this approach minimizes the software overheads of generating many small network messages (Chapter 6), it leads to high utilization of the network’s bandwidth. However, by ignoring the sparsity pattern of the data, it induces redundant data transfers.

The second alternative is the *Sparsity-Aware* (SA) approach [289], [290], [291]. Here, the indices of the nonzero values of the sparse matrix (or edges in a graph) are scanned and trigger the transfer of small data items (e.g., with RDMA [292]). In this chapter, we refer to these data items as *properties* since, in graph problems, they represent properties of the vertices connected to each edge (e.g., embeddings [52]). Similarly, we refer to the fine-grained remote requests as *Property Requests* (PRs). Although SA results in fewer redundant transfers, it induces significant software overheads for PR generation. These overheads lead to network bandwidth underutilization.

In this work, we first quantify the inefficiencies of software-only implementations of the two communication approaches. On average, the SU approach generates 720 redundant property transfers for every useful one. Although the SA approach leads to fewer redundant transfers, it can drop the line utilization to below 1% in a large cluster. In addition, due to the small sizes of the SA payloads (4B-1KB), the network traffic is dominated by packet headers, reducing the goodput [293]. Overall, both approaches are inefficient, leaving orders of magnitude of performance on the table.

To drastically improve performance, this chapter presents *NetSparse*, a set of hardware techniques to accelerate sparse communication. NetSparse extends SmartNICs (SNICs) [294], [295], [296], [297], [298], [299], [300] and programmable switches [301], [302], [303], [304], [305] with hardware extensions catered to the sparse domain. NetSparse enhances the SA approach by (1) increasing the network bandwidth utilization, (2) reducing the traffic, and (3) increasing the goodput by reducing the header overheads of network packets.

NetSparse introduces four hardware mechanisms. First, it extends RDMA to support one-sided *Remote Indexed Gather* (RIG) operations. Using coarse-grained RIG commands, the host offloads PR generation for batches of nonzeros to the NIC, reducing the overhead of frequent host-NIC communication through the PCIe bus. PRs are generated and served by special hardware *RIG Units* in the NIC without host involvement. The second mechanism enables RIG Units to *filter out* redundant PRs at runtime. With both techniques, PRs are generated at higher rates, the network bandwidth is better utilized, and useless traffic is minimized.

The third mechanism is a low-level network protocol that concatenates different PRs with the same destination node in a single packet. This is implemented with *(De)Concatenator* hardware modules in NICs and Switches. The goal is to increase the goodput by sharing the header overhead over multiple PRs. Finally, the fourth mechanism augments Top-of-Rack (ToR) switches with a hardware cache that stores properties fetched

from remote racks. Subsequent PRs from the local rack can quickly obtain the property from the cache. In contrast to P4-based [306], [307] switch caching mechanisms [308], the NetSparse cache is updated in the data plane to accommodate short-lived distributed sparse kernels.

We evaluate NetSparse with simulations of a 128-node cluster with a SPADE sparse accelerator (Chapter 3) in each node while running distributed sparse computations on matrices from SuiteSparse [73]. Our results show that NetSparse substantially improves performance over ideal versions of SU and SA software approaches (where SA is augmented with the Conveyors framework [309]). For example, with SA software, the workloads run only 3x faster than on a single-node system; with NetSparse, the workloads run 38x faster than on the single-node system—attaining more than half of the performance of an ideal system that has no communication overheads.

This chapter makes the following contributions:

- A characterization of software solutions for communication in distributed sparse kernels, exposing major inefficiencies.
- The four hardware techniques of NetSparse to accelerate the communication of sparse workloads in a distributed machine.
- A simulation-based evaluation of NetSparse in a 128-node cluster.

7.2 Background

At this point, we refer the reader back to Section 2.3 for a refresher on distributed sparse communication, 1D partitioning, and the characteristics of the SU and SA transfers.

As we later verify quantitatively, the SU approach suffers from a fundamental algorithmic inefficiency: it generates excessive network traffic. In contrast, the software overheads of fine-grained SA transfers could potentially be mitigated with system-level support. In Chapter 6, we attempted to optimize sparse communication using an algorithm that combines SA software and SU software. Inevitably, this algorithm inherits many of

the limitations of its imperfect components. In this chapter, we follow a different path: rather than combining existing software mechanisms, we attempt to accelerate the SA approach with new hardware support in the network. To that end, this section provides additional background on SA transfers and on the reasons behind their overheads.

7.2.1 Sparsity-Aware Property Requests

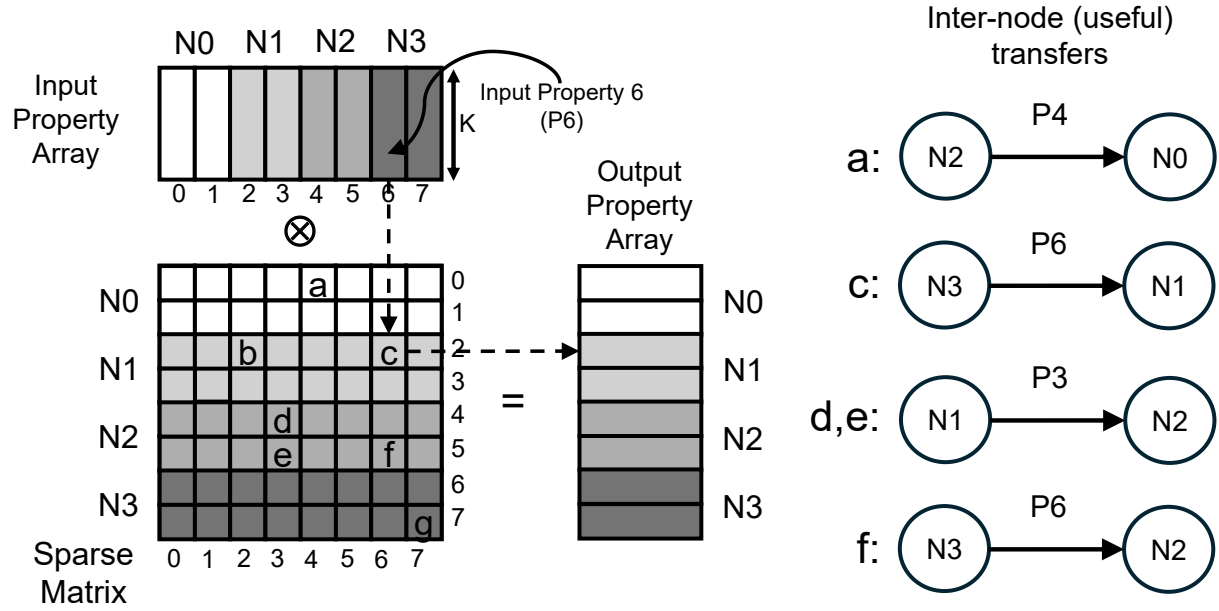


Figure 65: Communication in a distributed sparse kernel.

Figure 65 shows the data structures of an 1D-partitioned distributed sparse kernel. The sparse kernel corresponds to the multiplication of a sparse matrix with an input array to produce an output array. The matrix may represent the edges of a graph, while the input and output arrays may represent features or properties of the vertices of the graph. The input and/or output arrays can be dense vectors (e.g., in SpMV), or dense matrices that are typically tall and skinny (e.g., in SpMM)⁵. We use K to denote the number of elements

⁵Using the SpMM notation of Chapter 2, the input property array in Figure 65 corresponds to dense matrix B , while the output property array corresponds to dense matrix C . For SDDMM, there are two input property arrays: one is local to the node, and the other may be fetched remotely over the network. Although SpMM and SDDMM perform different local computations, their inter-node communication patterns are the same for a given sparse matrix and K value. The hardware support in this chapter applies to both kernels.

in a property.

In an SA approach, the coordinates of the nonzeros in the sparse matrix dictate the accesses to the input and output property arrays. Consider nonzero c in Figure 65. Its row index (rid) is 2, which leads to a write to the output property at index 2. Its column index (cid) is 6, which leads to a read of the input property at index 6. Note that both c and the output property that it accesses reside in the same node (i.e., $N1$). With 1D partitioning, writes to the output property array are always local. However, the read for input property 6 is remote. The right part of Figure 65 shows all the necessary inter-node input property transfers for the specific nonzero pattern in the example sparse matrix. Notice that nonzeros d and e can be covered using a single property transfer from $N1$ to $N2$. However, if requests are not deduplicated, a node may issue multiple requests for the same property when multiple local nonzeros have the same cid . No remote transfers are needed for nonzeros b and g , as the properties they access are already local.

Since writes are always local, from now on, we refer to input properties simply as properties. Also, we use *Property Requests* (PRs) to refer to inter-node reads and read responses for these properties.

7.2.2 Lifetime of an SA Property Request

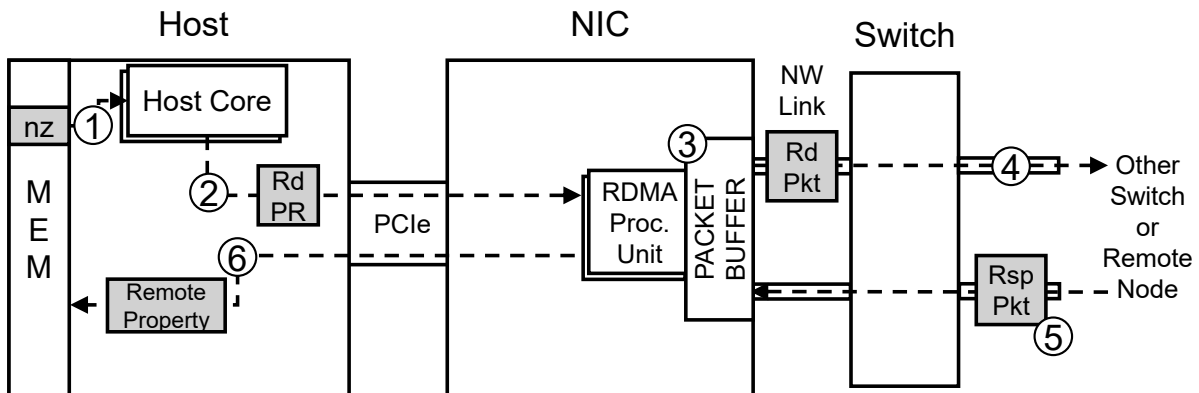


Figure 66: The lifetime of a sparsity-aware property request.

Figure 66 shows the lifetime of an SA PR that is converted into an RDMA read request

and carried over the network as a packet. First, a core reads one of the nonzeros assigned to its node ①, to determine what remote property needs to be fetched. Recall that the nonzero’s cid represents the index of the remote property that needs to be fetched. Hence, in the rest of the chapter, we refer to cid as *idx* (for index). After the idx is read from memory, the destination node is determined and a read PR is generated ②. This read PR will be directed through RDMA to the remote node that hosts the target property. Specifically, the read PR is forwarded to an RDMA processing unit in the NIC and a network packet is generated ③. The packet travels through the network and eventually reaches its destination ④. The destination node generates a read response containing the necessary property data, which travels back to the original node ⑤ and is written to its memory ⑥.

This process introduces significant inefficiencies. The host cores and the NIC communicate via MMIO writes for every nonzero of the sparse matrix, leading to the serialization of PR generation for different idxs. Furthermore, given that typical K values (Section 7.2.1) range from 1 to 256 (i.e., 4B–1KB payloads, assuming single precision), this process creates many fine-grained small-payload network messages. The resulting software overheads limit the rate at which new PRs are generated and limit network bandwidth utilization.

7.3 Motivation

In this section, we characterize the communication patterns exhibited by the SpMM, SDDMM, and SpMV kernels for five large sparse matrices from the popular SuiteSparse matrix collection [73] (details in Section 7.8) running on the NCSA Delta supercomputer [261].

- SU algorithms issue three orders of magnitude more redundant transfers than useful ones. The first row of Table 25 displays the ratio of useful to redundant property transfers for the SU approach in a 128-node system. On average, 720 redundant transfers are

generated for every useful one. Hence, an optimal system that avoids redundant transfers could potentially offer up to 720x average performance gains. This is a fundamental inefficiency of SU.

Table 25: Ratio of useful to redundant property transfers in a 128-node system for the SU and SA algorithms.

Matrix	arabic	europa	queen	stokes	uk
SU	1 : 1947	1 : 582	1 : 74	1 : 32	1 : 966
SA	1 : 27	1 : 0.02	1 : 25	1 : 3.6	1 : 4.5

- SA can also issue redundant transfers. The second row of Table 25 shows that, with the exception of the europa matrix, the SA approach can also lead to redundant transfers. This can happen if nonzeros in the same node share the same idx—e.g., the nonzeros d and e in Figure 65 both have idx=3. Note that, although some SA algorithms such as [291] eliminate these redundant transfers, they typically require costly preprocessing that needs to be repeated every time the sparse matrix changes.

- The software overhead of SA leads to severe underutilization of the network bandwidth. As explained in Section 7.2, SA transfers have software overheads that limit network bandwidth utilization. To quantify this effect, we use the source code of the Two-Face distributed SpMM algorithm (Chapter 6), configure it to SA-only mode, and evaluate it for a property size of K=32 (i.e., 128B). We measure the data transfer rates for a 2-node setup in the Delta supercomputer, which is interconnected with Cray Slingshot [310]. The measurements are shown in Table 26. On average, the achieved transfer rate is a little less than 0.5Gbps. We then compute the Line Utilization, which is the ratio of the transfer rate to the maximum line bandwidth (i.e. line rate). From the table, we see that the average line utilization is 0.24%, which leaves a lot of performance on the table.

Table 26: SA transfer rate and related metrics for a 2-node setup in the Delta machine.

Matrix	arabic	europa	queen	uk
Transfer Rate	0.5 Gbps	0.2 Gbps	0.7 Gbps	0.5 Gbps
Line Util.	0.26%	0.09%	0.36%	0.25%
Goodput	0.11%	0.04%	0.16%	0.11%

- The header overhead of small-payload packets impacts goodput. SA suffers from a high packet header overhead. Table 27 displays the contribution of the packet headers in the total SA network traffic for different K values. For low K, this contribution is as high as 97.6%. The last row of Table 26 shows that, for K=32, less than half of the achieved throughput is actually goodput.

Table 27: Contribution of packet headers to the total SA network traffic for properties with different K.

K	1	2	4	8	16	32	64	128	256
%	97.6	95.2	90.9	83.3	71.4	55.6	38.5	23.8	13.5

- PRs generated close in time are often directed to the same node. We find that PRs generated close in time by a node often request properties hosted by the same destination node. We refer to this property as *temporal remote destination locality*. As shown in Table 28, for queen, on average, all 64 consecutive PRs from a source node are directed to the same destination node (out of the 127 possible destination nodes). The reason is that queen has its nonzeros largely concentrated around the diagonal. PRs in other matrices also show temporal remote destination locality—even in europe, 64 consecutive PRs have only 7.4 different destinations out of the 127 possible ones. Later, we show how we exploit this property to concatenate network messages and improve goodput.

Table 28: Average number of unique remote destination nodes in 64 consecutive PRs from a node in a 128-node system.

Matrix	arabic	europe	queen	stokes	uk
Nodes	2.51	7.43	1.00	1.85	5.61

- Remote properties fetched by a node in a rack can be reused by other nodes in the same rack. Most of today’s high performance computing and datacenter networks are hierarchical [311], [312], [313]. For example, datacenter clusters are organized in racks, with nodes in the same rack connected to the same ToR switch. Similarly, the Delta supercomputer is interconnected with a Dragonfly topology [311], where nodes are organized in small groups and intra-group communication is faster. Hence, we investigate

the potential for sharing properties across nodes of the same rack/group, assuming groups of size 16. Excluding redundant transfers, we find that, on average, 85% of the PRs are for properties useful to more than one node in the same group. We later show how we employ in-network caching to utilize this sharing potential.

- Putting it all together. These results reveal that distributed sparse kernels are missing out on substantial performance potential. Fortunately, they also hint at how to attain some of this performance. To avoid the excessive redundant transfers of SU approaches, one must (1) exploit the sparsity patterns and adopt an SA approach. At the same time, one should: (2) eliminate any remaining redundant transfers of SA approaches at no preprocessing cost, (3) increase the SA line utilization by eliminating the software overheads associated with PR generation, and (4) decrease network header overheads. Throughout the process, one should leverage (5) temporal remote destination locality and (6) intra-group property sharing.

7.4 NetSparse Architecture

Based on the previous findings, we introduce four hardware techniques to accelerate distributed sparse communication. We call the resulting architecture *NetSparse*. NetSparse extends SNICs and programmable switches with hardware extensions catered to the sparse domain. In this section, we give a high-level overview of these mechanisms; we describe the mechanisms in more detail in Sections 7.5 and 7.6. Our design uses the vanilla SA approach of Figure 66 as a starting point.

1. Hardware Support for One-Sided Remote Indexed Gathers. The SA communication pattern involves reading a list that indexes properties needed by the nonzeros and that are hosted in various remote nodes. This operation as a whole is a large *Remote Indexed Gather (RIG)*. This operation is not supported natively by RDMA today (e.g., as defined by InfiniBand verbs [314]). In a vanilla SA execution, the host cores break-down the RIG into fine-grained per-nonzero RDMA reads, and issue each one of

them to the NIC. As a result, the host and the NIC need to communicate and synchronize through the PCIe bus very frequently. This leads to a high software cost and high latencies that cannot be hidden, limiting the effective rate at which new PRs can be generated.

To increase this rate, we propose extending RDMA with native support for RIG operations. The host cores issue RIG operations that are handled by novel specialized processing units (*RIG Units*) in an SNIC. Each RIG command contains the information needed for the SNIC to fetch the properties for a coarse-grained batch of nonzeros, without requiring involvement from the cores of either the host or remote nodes. RIG operations are one-sided. A RIG completes and the host core is informed when: (1) all the PRs in the batch have been generated, and (2) all the remote properties have been fetched and written back to the host memory (through DMA). By coarsening the granularity of NIC-host communication and eliminating general-purpose overheads with the specialized RIG Units, the PR generation rate increases, leading to a substantial *increase in the network bandwidth utilization*.

2. Property Request Filtering and Coalescing. Our second technique *eliminates redundant SA PRs at runtime*. Recall that nonzeros with the same idx generate PRs for the same property. Since the input property array is not updated during an iteration of the sparse kernel, all the requests to a given property after the first one are redundant and waste network resources. To eliminate as many of them as possible, we use a per-node bitvector with as many bits as different idxs the sparse matrix has. We call this bitvector *Idx Filter* and allocate it in the DRAM memory available in modern SNICs [294]. A bit in the Idx Filter is set by the RIG Units after a remote property arrives and is written to the host’s memory. Also, before the RIG Units forward a PR for packetization, they check if the bit for the specific idx is set in the Idx Filter. If it is, the PR is redundant and is dropped. The PR is also dropped if another PR with the same idx is currently outstanding, but the requested property has not yet arrived. This is achieved with a request coalescing mechanism supported by the RIG Units.

3. Property Request Concatenation. This technique is grounded on the temporal remote destination locality (Section 7.3). When multiple fine-grained PRs generated close in time are directed to the same node, we concatenate them in a single network packet. For this, we introduce a low-level NetSparse network protocol implemented on top of RDMA. Our protocol has two layers: the *Concatenation* layer and the *PR* layer. The headers of the concatenation and higher-level networking layers can be shared across multiple PRs. Hence, the effective per-PR header overhead decreases, *increasing goodput*. PR concatenation is realized using a mechanism based on delay queues (Section 7.6.1). We augment SNICs and switches with special *Concatenator* units that implement this mechanism in hardware.

4. In-Switch Property Caching. To *share properties* fetched from remote racks within nodes of the same rack, we augment ToR switches with a hardware cache. We call it *Property Cache*. In contrast with P4-based switch caching mechanisms [308], the Property Cache is updated in the data plane, to accommodate the short-lived distributed sparse kernels. The Property Cache is placed in a new layer of *middle pipes*, that complement the existing ingress/egress pipes in Tofino-like switches (Section 7.6.2).

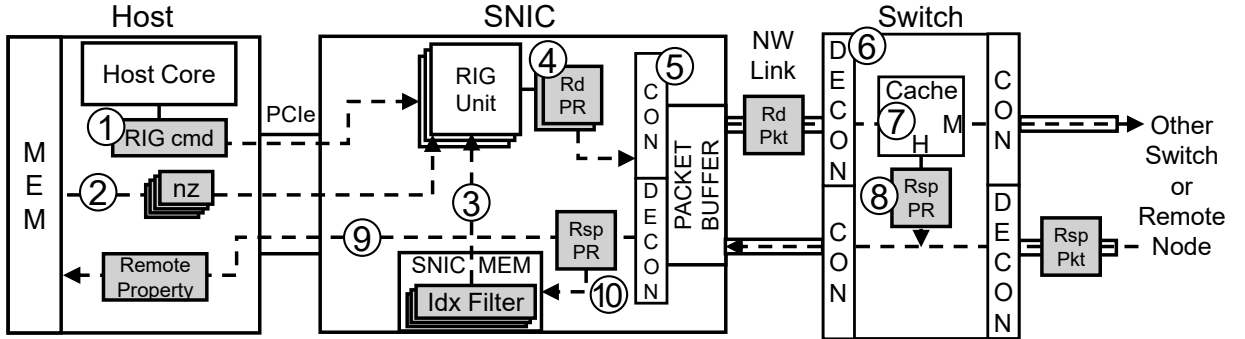


Figure 67: The lifetime of a Sparsity Aware (SA) Property Request (PR) with NetSparse.

Figure 67 shows how the lifetime of a PR changes with NetSparse in comparison with Figure 66. At step ①, one of the host cores sends a RIG command to one of the RIG Units of the SNIC. The RIG Unit then reads the appropriate idxs for a batch of nonzeros from the host’s memory with DMA (step ②). At ③, for every idx, the RIG Unit checks the corresponding bit in the Idx Filter. If the bit is set, it means that the property for this

idx has already been fetched by this or another RIG Unit. In this case, a PR is not generated and the idx is dropped. An idx is also dropped if a PR with the same idx is currently outstanding. This is determined through a coalescing mechanism before step ③ (not shown). For the non-dropped idxs, read PRs are generated ④. At step ⑤, multiple PRs may be concatenated in a single packet, which is forwarded to the network. When the packet reaches the ToR switch, it is de-concatenated ⑥ and the Property Cache is looked-up using the idx of each PR ⑦. For read PRs that hit, response PRs containing the requested properties are generated and forwarded back to the node ⑧. The generated response PRs can be concatenated with others to form response packets. When a response packet arrives back at the SNIC, it is deconcatenated, and the properties it carries are written to the host’s memory ⑨. Finally, the Idx Filter is updated ⑩, marking the corresponding properties as already fetched.

On Property Cache misses, read PRs are concatenated into packets and forwarded further into the network. When a read PR reaches its destination, a remote RIG Unit loads the requested property from its host’s memory and creates a response PR for the original node.

A RIG command completes when all the PRs for the non-dropped idxs of the batch are generated and all their responses are received at the requesting node. Parallelism increases when multiple commands are concurrently active in different RIG Units of an SNIC.

7.5 NIC-centric NetSparse Mechanisms

This section describes the NIC-centric NetSparse mechanisms: how the RIG operation is offloaded to the NIC and how redundant PRs are eliminated. We also describe the architecture of the RIG Units.

We assume an RDMA-capable SNIC with an NoC interconnecting the different SNIC components, including DRAM memory, packet buffer, existing RDMA Processing Units, and other accelerators or lightweight cores. This architecture, shown in Figure 68, is based

on an AMD Pensando SNIC [294], [295]. However, our mechanisms are not tied to this type of SNIC. The RIG Units are attached to the SNIC's NoC and are connected directly to the packetization and buffering logic through hardware queues.

7.5.1 Offloading the RIG Operation to the NIC

Although SA communication can be expressed as RIGs, RDMA today lacks native support for this coarse-grained operation. In a naive SA execution, the host issues fine-grained RDMA reads with different destinations. For each of these reads, the host and the NIC need to communicate. This is an expensive process that leads to high software costs and limits the PR generation rate.

To eliminate these costs, we propose extending RDMA hardware with the new RIG Units, capable of performing RIG operations. A RIG Unit can be configured to operate as a client thread or as a server thread. Client threads receive RIG commands from the host's cores, generate the necessary read PRs for a coarse-grained batch of idxs, and inform the host when the operation is complete. Server threads respond to requests from the network by fetching properties from their host's memory, and generating response PRs.

Figure 68 shows a NetSparse SNIC, with our hardware extensions highlighted. The RIG Units have memory-mapped control registers to communicate with the host. A client thread begins its execution when the host writes a RIG command to a register. The command contains the host address that the client thread should read the nonzero idxs from, the host address to write the gathered remote properties, the number of idxs, and the size of a property. After the thread receives the command, it issues a local DMA transfer that copies the idxs from the host memory to a buffer in the RIG Unit. It then generates PRs and forwards them to a hardware queue (Tx Q) for packetization. When a response PR arrives from the network through an Rx Q, the property it carries is written to the host's memory at the appropriate location based on its idx. When all the properties have been received, the host is notified. The intermediate concatenator/deconcatenator modules

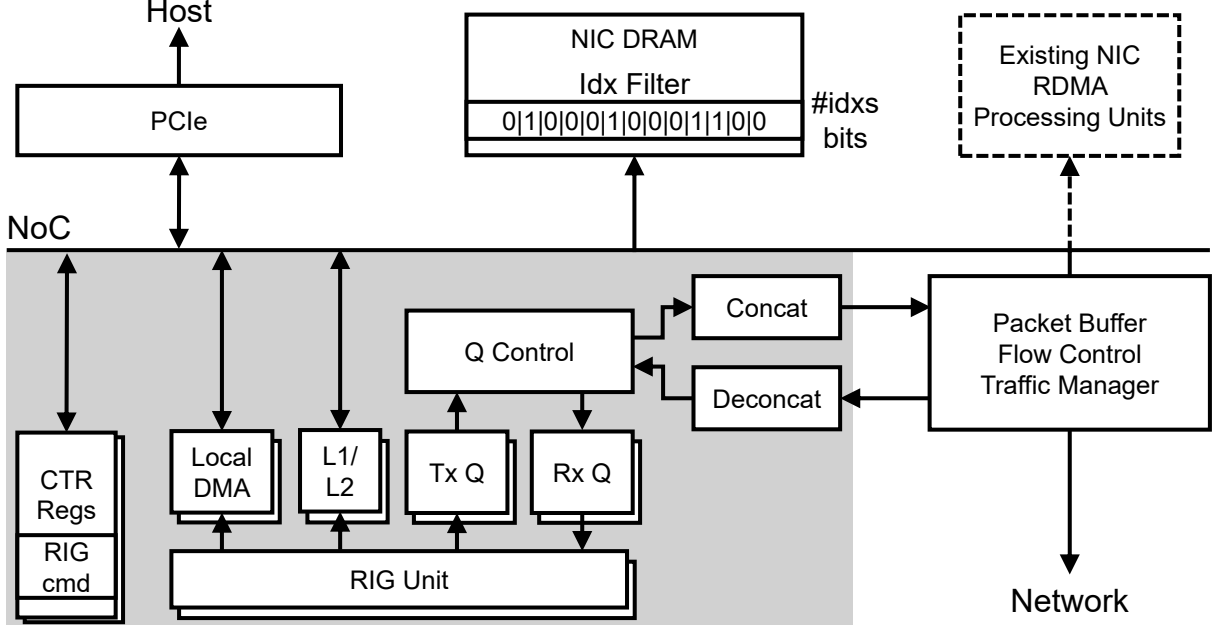


Figure 68: NetSparse SNIC, with our hardware extensions highlighted. Concat/Deconcat are discussed in Section 7.6.

are analyzed in Section 7.6.

The nonzeros processed by a node are grouped into batches, and multiple RIG commands are active concurrently in different RIG Units. This improves parallelism and enhances the overlapping of computation and communication. However, there is a tradeoff in the RIG batch size. A very small batch size can expose the communication overhead between the SNIC and the host core and can limit the effectiveness of SNIC offloading. In fact, for a batch size of 1, the RIG command is very similar to a vanilla RDMA read. A very large batch size limits parallelism and can cause intra-node load imbalance between the different client threads. We explore this tradeoff in our evaluation.

7.5.2 Redundant PR Elimination

An Idx Filter bitvector is a software structure used to eliminate redundant PRs due to idx repetitions. It has one bit for every column in the sparse matrix and is allocated in the SNIC’s DRAM memory (Figure 68). Given that modern SNICs can easily support 16GB of DRAM memory [315], they have enough memory capacity to fit Idx Filters for sparse

matrices with 100 billion columns—well over the largest SuiteSparse matrices. A bit is set in the filter when the property for the corresponding idx has been received and written to the host memory. The same filter is shared by all client RIG Units, which access the SNIC DRAM through a small coherent L1/L2 cache subsystem (and additionally through a shared LLC attached to the SNIC NoC, if there is one available, as in Pensando [294], [295]).

A RIG Unit also contains a private hardware *Pending PR Table*, which is a CAM that tracks the currently outstanding PRs issued by this particular RIG Unit. The table is updated in hardware when the RIG Unit issues a PR and when a PR response is received.

With these two structures, NetSparse eliminates redundant PRs. Specifically, after a client thread running on a RIG Unit reads an idx, it checks the Pending PR Table and the Idx Filter, to see if there is an outstanding or an already completed PR, respectively, for the same idx. If any condition is true, the PR is not issued—an event we call *coalescing* and *filtering*, respectively, for the two cases. Note that only PRs originating from the same RIG Unit can be coalesced. We made this design decision because implementing coalescing across RIG Units in a node requires synchronization support.

7.5.3 Architecture of a RIG Unit

Figure 69 shows the architecture of a RIG Unit in client mode. The RIG Unit contains: (i) the Logic Engine ①, which includes logic to check for idx coalescing and filtering, and to perform PR generation, and (ii) its memory structures ②, namely, the SRAM Idx and Rx Property Buffers, and the Pending PR Table. The figure also shows the RIG Unit’s auxiliary structures: a DMA engine that accesses the host’s memory ③, L1 and L2 caches to access the SNIC DRAM ④, and the Tx and Rx Queues to communicate with the network interface ⑤.

The DMA engine accesses the host’s memory to fetch batches of idxs into the Idx Buffer. For each idx, the Logic Engine checks the Pending PR Table and the Idx Filter

(through its LSQ and L1/L2 cache hierarchy) for redundant PR elimination. If the current idx is not eliminated, the Destination Solver calculates the destination node of the PR, and the local host's memory address to store the remote property when it arrives. Then, the PR Generator registers an entry in the Pending PR Table, creates a read PR, and pushes it to the Tx Queue. When the corresponding response PR is received at the Rx Queue, the fetched property is stored in the Rx Property Buffer. Then, the property is written to the host's memory through DMA, and the Idx Filter and Pending PR Table are updated appropriately.

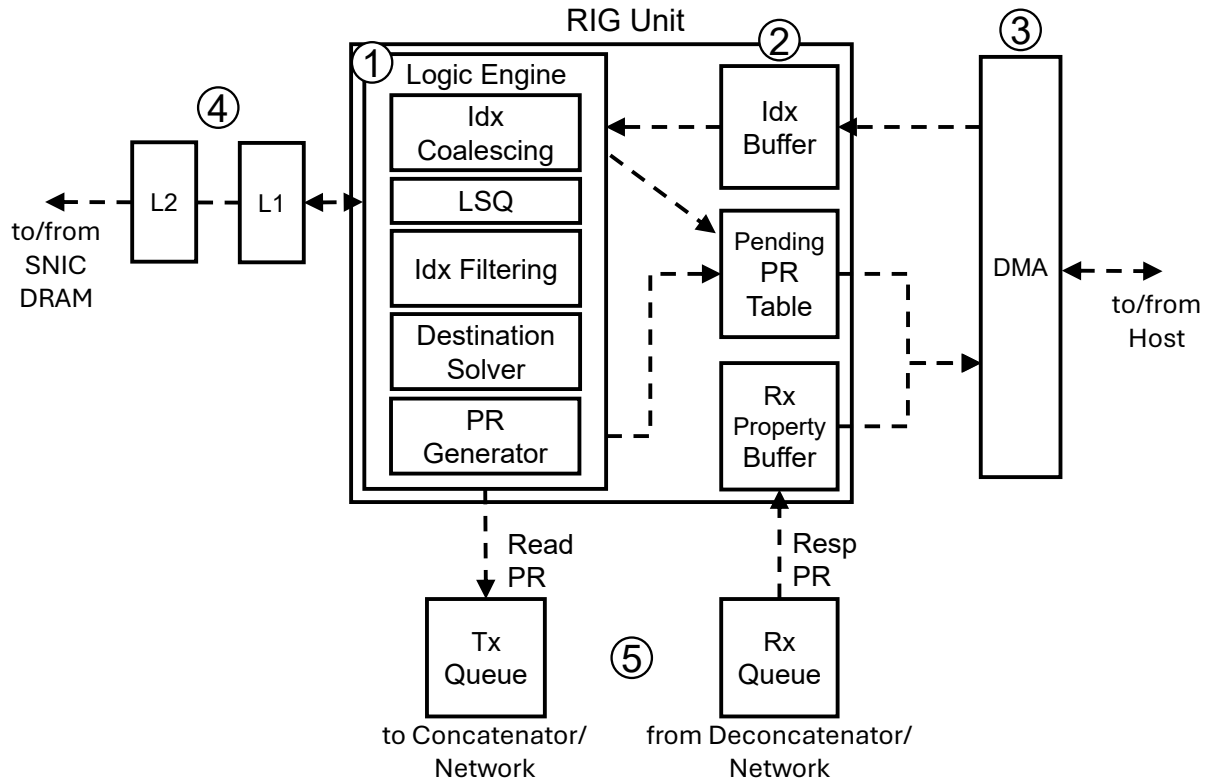


Figure 69: RIG Unit in client mode and auxiliary structures.

A RIG Unit does not stall while processing an idx. It processes multiple idxs in a pipelined manner. It can read a new idx from the Idx Buffer every cycle, and overlap the Idx Filter accesses, the DMA operations, and the communication with the network interface across multiple PRs. This overlapping hides latencies that prevent current software solutions from generating PRs at high rates. A RIG Unit only stalls if it has

issued the maximum number of network requests that its Pending PR Table can track, or if its LSQ or Tx Queue are full.

The same RIG Unit can be reconfigured to implement a server thread as follows. First, the Rx Queue receives read PRs and the PR Generator generates response PRs that are pushed to the Tx Queue. Further, the Destination Solver calculates the local host's memory address where the requested properties reside. Finally, the Rx Property Buffer receives properties from the host and deposits them in the Tx Queue. The Idx Buffer and the Idx Filtering/Coalescing logic are unused. Incoming Read PRs are assigned to Server RIG Units dynamically through the Q Control module (Figure 68).

7.5.4 Host-Side Software Changes for RIG

In our design, RIG offload is exposed as a seamless extension to the existing RDMA-Verbs API [314] rather than as a separate library or a new system call. We introduce a new opcode *IBV_WR_RIG* alongside standard opcodes such as *IBV_WR_RDMA_READ* in the *ibv_send_wr* union. This requires only small changes to the *libibverbs* driver to recognize the new opcode and program the control registers of the RIG Units in the SNIC. Along with the opcode, the host passes all the other fields necessary for a RIG command (Section 7.5.1).

7.6 Network-centric NetSparse Mechanisms

We now detail the network-centric NetSparse mechanisms: Property Request (PR) Concatenation and In-Switch Property Caching.

7.6.1 Property Request Concatenation

7.6.1.1 Protocol

With NetSparse, multiple PRs of the same type (read or response PRs) that share the same destination can be concatenated into a single network packet. This is achieved by

using our proposed application-level NetSparse network protocol (Figure 70). The protocol is structured into two layers: a Concatenation and a PR layer. We use RDMA as the base protocol before the NetSparse header; however, other alternatives are possible.

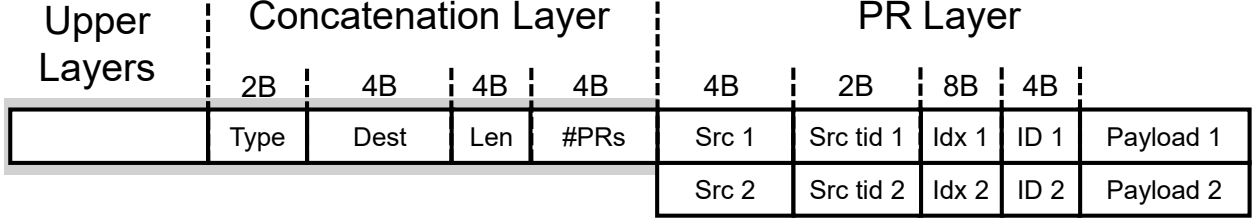


Figure 70: NetSparse 2-layer network protocol.

As shown in Figure 70, the concatenation layer header includes fields for the PR type (i.e., read or response), destination, property length (which is the same for all PRs in a given kernel), and the number of concatenated PRs. The PR layer header specifies, for each individual PR, the source node, source RIG Unit ID (i.e., tid), property idx, and request ID. A single NetSparse packet can encapsulate multiple PRs, along with their respective PR layer headers and payloads, as long as the packet remains within the maximum transmission unit (MTU) size limit. Importantly, NetSparse packets have a single destination and are not multicast. Hence, they do not alter the network’s routing.

By concatenating PRs, the header overhead of the concatenation and upper layers is shared across multiple PRs. This results in a large reduction in header overhead. To see why, assume a header of 50 bytes for the upper layers. Without concatenation, each PR packet requires a header of $50+10+18 = 78$ bytes. Thus N PRs in separate packets have a header of $78N$ bytes. With concatenation, a packet with N PRs has a header of $50+14+18N = 64+18N$ bytes. For even small values of N , this is a significant reduction in overhead, which improves goodput—particularly for small property sizes. In addition, the resulting reduction in the number of network packets reduces network contention and improves performance.

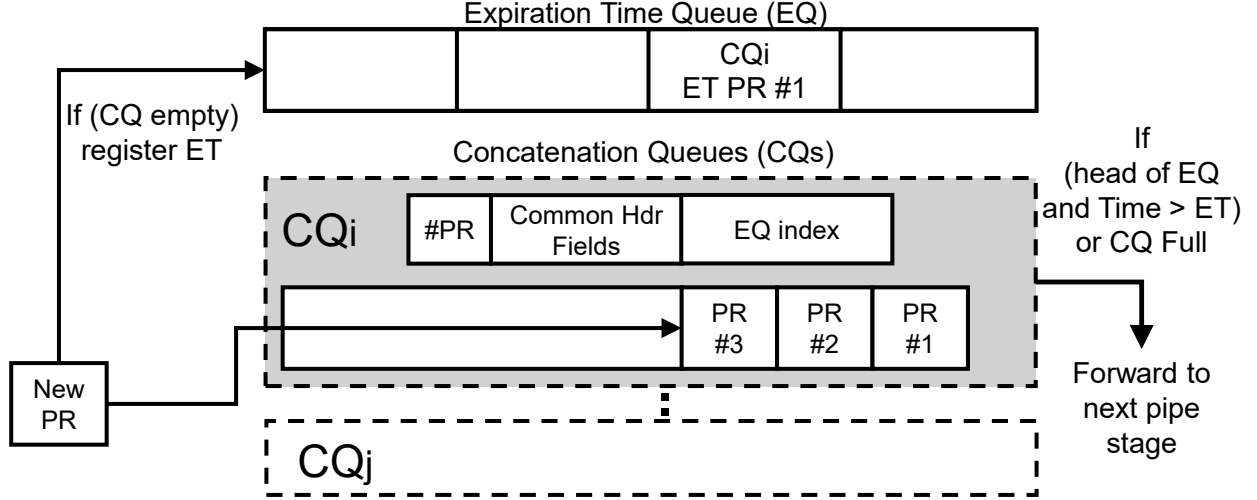


Figure 71: PR concatenation mechanism.

7.6.1.2 Mechanism

The concatenated PRs can come from the same or different threads, and from the same or different nodes. Concatenation happens transparently, without requiring the threads to synchronize. It occurs in the data plane, in both the ToR switches and the SNICs. We call these points *Concatenation Points*.

The concatenation mechanism is shown in Figure 71. A Concatenation Point has multiple *Concatenation Queues* (CQs), which are delay queues that collect PRs of the same type and destination. There is one CQ for reads and one CQ for responses per possible destination (in the worst case, for a total of $2(N-1)$ CQs, where N is the number of nodes). Each CQ is MTU sized (i.e., 1500B). When a PR arrives, it is pushed into the appropriate CQ, based on its type and destination. In a CQ, PRs wait to be concatenated. When a CQ gets full, the PRs it contains are concatenated and forwarded for packetization. However, the PRs in a CQ can also be forwarded for packetization before the CQ gets full. We describe this case next.

Each non-empty CQ is associated with an Expiration Time (ET), which is the time when its first PR entered the CQ plus a *DelayCycles* parameter. *DelayCycles* is the maximum allowed time that any PR can spend waiting to concatenate with others. If the

ET of a CQ is exceeded, the PRs it contains are concatenated and forwarded for packetization.

To efficiently identify CQs close to expiration, we employ a circular queue called *Expiration Time Queue* (EQ). When a PR enters an empty CQ, the index of the CQ plus the computed ET are inserted at the tail of the EQ. The EQ is shown in Figure 71, where each entry has a pair $\{CQ_i, ET\ PR\ \#1\}$. At every cycle, the hardware checks the entry at the head of the EQ for expiration—all the other CQs are guaranteed to expire later.

Beyond a set of PRs, each CQ holds some metadata. As shown in Figure 71, the metadata is the number of PRs currently in the CQ, the common header fields of the PRs in the CQ, and the position in the EQ that holds the entry for the CQ (i.e., the EQ index). The latter is needed in case the CQ gets full before its ET expires. In this case, the corresponding EQ entry needs to be cleared.

PR concatenation delays individual PRs for some cycles. This is tolerable within some limits. What we care about is when the whole kernel completes—not when individual PRs do. Typically, this delay is negligible compared to network round trip times.

PR deconcatenation breaks down a concatenated package into its component PRs. It also happens in the SNICs and ToR Switches. Its implementation is straightforward.

7.6.1.3 Implementation

Since (de)concatenation happens in the data plane, one approach would be to implement it using the P4 pipelines included in Tofino switches and Pensando SNICs. Unfortunately, the register arrays of the P4 pipelines cannot handle data types larger than 32 bits. As a result, the number of register accesses that would be needed per cycle for concatenation to work would quickly exhaust the number of register accesses that the hardware can support per stage and cycle. For this reason, we extend the switch/SNIC architecture with new hardware *Concatenator* and *Deconcatenator* modules. Their integration in the SNIC is shown in Figure 68. To integrate them in switches, we can follow an approach similar to

Taurus [316] and implement them in separate chiplets. Networking chips such as Broadcom’s Tomahawk [317] have already started moving towards a multi-chiplet direction.

P4 supports *externs* [318] to extend the language and provide operations not natively supported. To use (De)Concatenators, one approach is to introduce new dedicated *control-block* [316] P4 types in P4. Within these new control blocks, (De)Concatenator units can be invoked using the *Concatenate* and *Deconcatenate* constructs, similar to how hashing units and stateful memory (meters, counters, and registers) are made available in the Tofino chip [301], [302].

7.6.2 In-Switch Property Caching

To share properties fetched from remote racks within the nodes of the same rack, we augment ToR switches with a cache. Although prior work such as NetCache [308] has implemented caching in Tofino switches by utilizing the P4 register arrays and Match-Action Tables (MATs), such an approach would not work for sparse kernels. NetCache targets very long-running Key-Value Stores (KVS) and, due to P4 pipeline limitations, NetCache is periodically updated with values through the control plane. Instead, an iteration of a distributed sparse-kernel is much shorter-lived than KVS. The control plane-based cache update rate is not sufficient. Hence, we include a new hardware cache in the switch called *Property Cache*, intended to be used in sparse supercomputers. Similar to the Concatenators, this is a new hardware module that could be invoked, for example, in P4 as a control-block using Lookup and Insert constructs. Before the kernel begins, the *Property Cache* is configured by the control plane and any data it contains is marked as invalid.

7.6.2.1 Switch Architecture

To support our technique, we modify the architecture of a switch to handle arriving packets as follows. An arriving read packet is deconcatenated and each of its PRs checks

the Property Cache to see if it contains the desired property. If a PR finds its property (i.e., it hits in the cache), the PR becomes a response PR and its destination node is changed to be the node that issued the PR. Otherwise, no action is taken. Then, all the PRs (irrespective of whether they hit or missed in the cache) go through a concatenation step and are then issued to the correct output port.

An arriving response packet is deconcatenated and each of its PRs checks the Property Cache to see if it contains the corresponding property. If a PR finds the property, no action is taken. Otherwise, the PR's property is saved in the cache for future use. Then, all the PRs go through a concatenation step and are issued to the correct switch output port.

NetSparse supports this algorithm with high performance by adding a second crossbar to the network switch. The design is shown in Figure 72. The figure shows a switch with 16 input ports (leftmost part of the figure) grouped into four *ingress* pipes and 16 output ports (rightmost part of the figure) grouped into four *egress* pipes. Rather than having a single 16x16 crossbar, we add four more pipes in the middle and one additional 16x16 crossbar between the middle and egress pipes. Each middle pipe has a deconcatenator, a Property Cache, and a concatenator.

To prevent the access latency of the Property Cache from blocking the pipe, the cache itself is pipelined. Further, a shift queue (not shown) is used to host PRs while they check the cache. At every cycle, a new PR can enter this queue, without applying backpressure.

Figure 72 shows two example packet flows. First, a PR packet arrives at Port 12 in ingress Pipe 3 and must be routed to Port 8 in egress Pipe 2. NetSparse first routes it to the middle pipe at the same offset as the ingress pipe (i.e., Pipe 3) ①, where it will update the Property Cache. In the middle pipe, the packet is first deconcatenated and then each of its PRs deposits its property in the Property Cache (unless it is already there). Then, all the PRs go through a concatenation step, and are then routed to their requested output port (Port 8 in egress Pipe 2) ②.

The second example is a read packet that arrives at Port 4 in ingress Pipe 1 and must

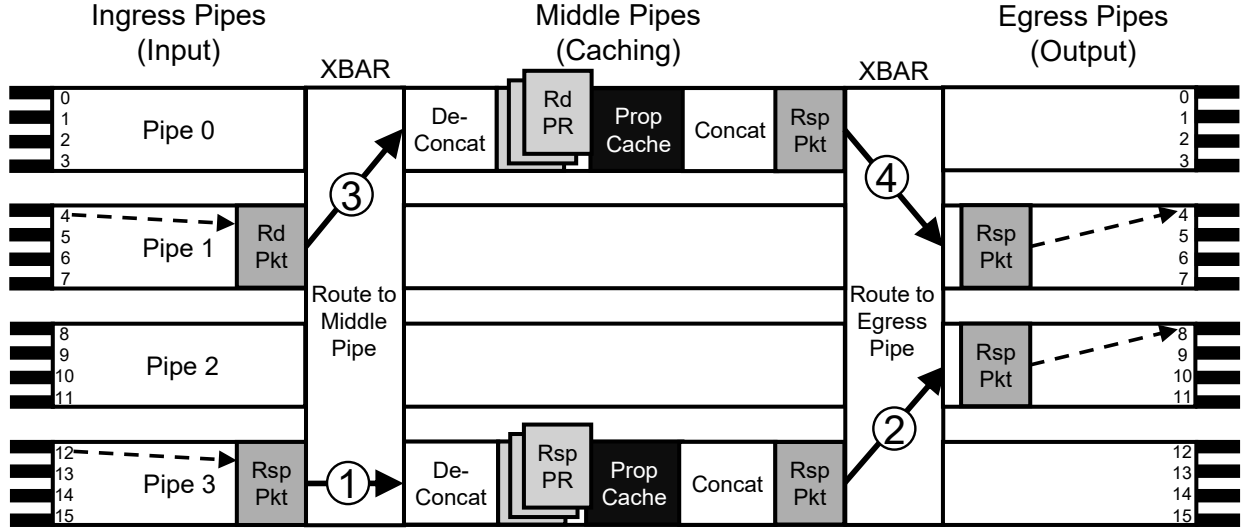


Figure 72: NetSparse switch with two example packet flows.

be routed to Port 0 in egress Pipe 0. NetSparse first routes it to middle pipe at the same offset as its egress pipe (in the example, Pipe 0) ③. This is because middle Pipe 0 has the cache where prior PRs that can source the response would have saved the property. Indeed, for a pair of {response, read} PRs to communicate a property in a deterministic routing network, the egress port of the read must match the ingress port of the response.

In the example, as the read packet reaches middle Pipe 0, it is deconcatenated and each of its PRs accesses the Property Cache. Assume that only one PR hits (shown in the figure). That read PR gets the property from the cache, becomes a response PR and changes its destination to be its source node. Consequently, NetSparse routes it to the output port that matches its input port: Port 4 in egress Pipe 1 ④. Before reaching the egress pipe, however, the PR goes through a concatenation step. The read PRs that missed in the Property Cache remain unchanged, go through a concatenation step, and are routed to their initial output port (Port 0 in egress Pipe 0).

An alternative design that we considered was to place the Property Cache in the Egress Pipes. This approach does not need the middle pipes or the second crossbar. However, it disrupts the packet flow for both read and response PRs.

First, for read PRs, if a read PR finds its property in the Property Cache and needs to

change its destination node, it must be recirculated to an input port of the switch. It is then routed through the crossbar a second time to reach the new destination. Second, for response PRs, a response PR must be mirrored: it must be routed to both its destination egress pipe and to the egress pipe with the same index as its ingress pipe (since this is the egress pipe that contains the correct Property Cache to update).

7.6.2.2 Property Cache Architecture

The Property Cache is a set-associative hardware cache that is accessed with the property idx bits (i.e., the cid bits of the sparse matrix) and returns the property associated with the target idx. In practice, different kernels may have different property sizes. Hence, we envision the Property Cache to support different property sizes and be configurable. Specifically, before a kernel executes, the control plane configures the Property Cache for the single property size to use.

To support this configurability, the Property Cache is implemented in *Segments*. To see how it works, assume that it needs to support property sizes of {16B, 32B, 64B, ... 512B}. In this case, the Property Cache is designed with 32 segments, each able to provide 16B of property data. If the cache is configured for 16B properties, a cache access will only activate one segment, which will provide the 16B. If the cache is configured for 32B properties, a cache access will activate two adjacent segments, each of which will provide 16B. If the cache is configured for 512B properties, a cache access will activate all 32 segments, which together will provide 512B.

Figure 73 shows the Property Cache described. Because it has 32 segments, each with a certain number of sets and ways, the property idx bits include a 5-bit *Segment bits* field, in between some Set bits and the Tag bits. These Segment bits will identify the segment(s) with the desired property value. The Segment bits are routed to a *Segment Selector* that is configured with a *Mode* and generates a bitmask of 32 *Enable* bits to enable the desired segment(s). For example, if Mode is 16B, the bitmask has only 1 Enable bit set; if Mode is

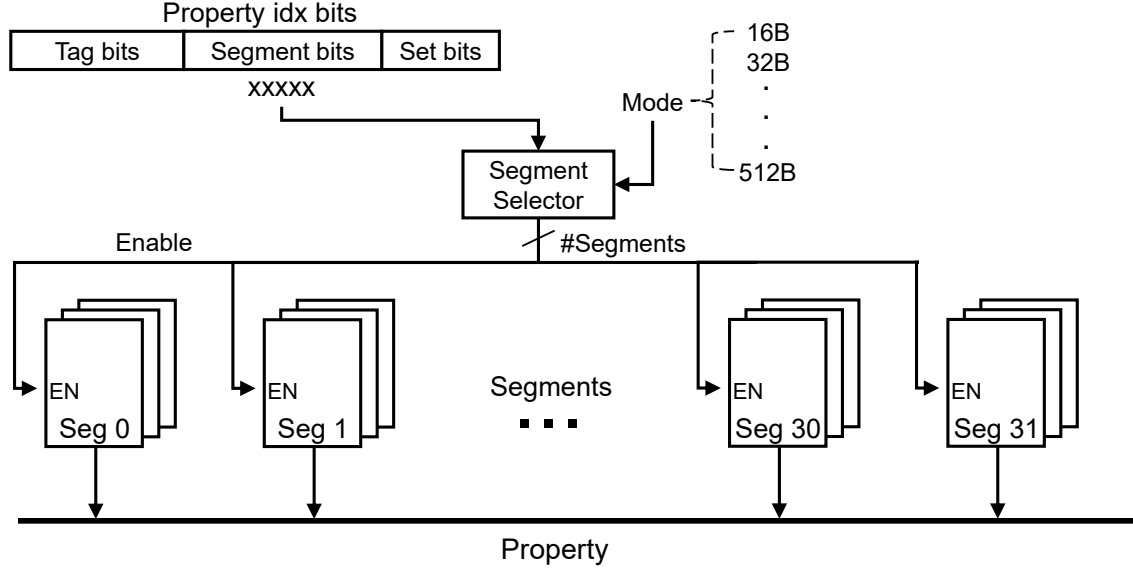


Figure 73: Segmented design of the Property Cache.

32B, the bitmask has only 2 adjacent Enable bits set; if Mode is 512B, all the Enable bits are set. The Set bits and Tag bits are routed to all the segments, and prompt the segments that are enabled to dump the property bits onto an output network.

As an example, assume that the Property Cache is configured in 32B Mode. This means that the Segment Selector will ignore the LSB of the Segment bits. Assume that the Segment bits are 1110X. In this case, we are looking for the pair of segments that are one before last. Hence, the Segment Selector will set the Enable bits to 0...01100. Overall, with this Property Cache design, the whole cache capacity can be utilized regardless of the property size.

If the property size of the kernel is larger than the largest one (S_{max}) supported by the Property Cache, the input property array can be tiled into chunks, so that the maximum property size in each chunk does not exceed S_{max} .

7.7 Other Considerations

1. Network Packet Loss. NetSparse is designed for a lossless RDMA-based networking environment (i.e., InfiniBand-like), which applies backpressure when network queues get

full. This limits packet losses to hardware failures mostly. Detection of packet losses is accomplished by setting up a watchdog timer when a RIG Unit initiates a RIG operation and resetting it when the operation terminates. If the watchdog timer times-out, the RIG operation is considered failed. Then, the host is informed, and the whole buffer in the host’s memory to which the RIG Unit has possibly written partial results is discarded.

2. Scalability of the Concatenation Mechanism. Since our concatenation mechanism allocates one Concatenation Queue (CQ) for each possible destination node, the total SRAM requirement scales with the total number of nodes in the network. We show in our evaluation that this does not cause issues for 128 nodes. However, it could cause excessive memory overhead and low queue utilization under large-scale deployments. To solve this issue, a possible approach is to *virtualize* the CQs. With this approach, each concatenation point includes a fixed number (i.e., independent of the number of nodes) of smaller sub-MTU sized (e.g., 128B) ”physical” CQs. These physical CQs are dynamically assigned on demand, whenever a PR for a new destination node appears or a physical CQ for a desired destination node is found to be full. Multiple physical CQs with PRs with the same destination are linked into a ”virtual” CQ. If the total occupancy of a virtual CQ reaches the MTU, the contents of its physical CQs are concatenated together into a single packet that is issued into the network, and the corresponding physical CQs are freed.

3. Deployment. NetSparse is backward-compatible with state-of-the-art existing RDMA-capable HPC infrastructure. NetSparse headers appear as payloads of the RDMA layer, and thus they are opaque to the existing network routing infrastructure. However, to be able to generate and decode NetSparse packets, at a minimum, the end-host SNICs need to include RIG Units that can handle RIG commands. Both the source and destination nodes should be RIG-capable. In fact, we believe that RIG operations can be useful for other throughput-oriented kernels with fine-grained irregular accesses. NetSparse switches can be deployed incrementally.

7.8 Methodology

7.8.1 Software Baselines

We compare NetSparse to an SU and an SA software-only baseline. We make idealistic assumptions for both of them. For the SU baseline (*SUOpt*), the communication time is assumed to be equal to only the time needed for a single node to receive all of the data bytes needed from the network at 100% line bandwidth utilization and without any header overheads. There is no network or SNIC latency and all nodes perfectly overlap their message reception. This is the optimal performance limit of the SU approach.

For the SA baseline (*SAOpt*), we assume an environment where some of the NetSparse mechanisms are implemented in software. Specifically, we augment the SA algorithm with the Conveyors framework [309] and implement in software the three NetSparse mechanisms that have a reasonable design in software: communication batching, message concatenation, and filtering.

To mimic the first two mechanisms, we rely on Conveyors. With Conveyors, each core aggregates batches of PR idxs with the same destination into a software buffer, and issues them to the network as a single message. Since RDMA does not support one-sided Remote Indexed Gathers, Conveyors use two-sided communication (i.e., send and receive messages) that require synchronization between cores in different nodes. Note also that sharing the headers across PRs for message concatenation is only possible if they originate from the same node. Finally, to mimic filtering, we optimistically assume that the software perfectly pre-filters redundant PRs offline without any preprocessing or runtime cost. The other NetSparse techniques (concatenation of messages originating from different nodes and in-network caching) are not feasible with software. Further, we also assume that there is no network or SNIC latency. Overall, for *SAOpt*, we only account for the software overheads of PR generation, book-keeping, synchronization, and buffering.

To isolate the software overheads of *SAOpt*, we measured the rate of data transfers

under an ideal scenario: all communication happens between cores of the same node and is perfectly balanced. Specifically, we evaluated such a setting on a high-performance NCSA Delta node with 64 out-of-order cores. By dividing the total traffic by the execution time, we computed the rate of data transfers. Then, we divided this rate by the line rate, ignoring headers, to calculate the goodput as a percentage of the line rate. The results are shown in Figure 74 for two different property sizes K (Section 7.2) and different numbers of cores in the node. We observe that the (optimistic) goodput scales almost linearly with the number of cores. We see that the maximum achievable goodput—even with 64 high-performance cores, perfectly balanced communication load, and no network overheads—is far from the optimal 100%. From this experiment, we measure the total software overhead per PR. We then use this value to calibrate our simulations of SAOpt in the evaluation section.

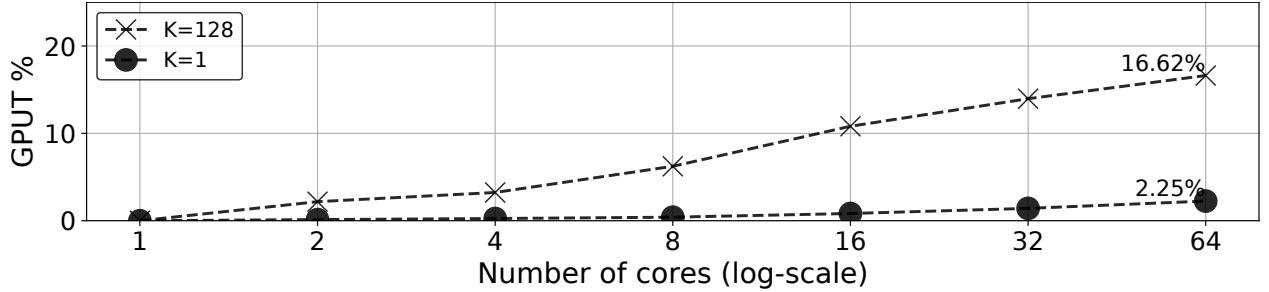


Figure 74: Ideal SAOpt goodput vs number of cores in a node.

In our evaluation, in SAOpt, we use all 64 CPU cores in a node to support the cross-node communication; in NetSparse, we use a single CPU core per node to control the RIG Units of the SNIC. We do this to highlight the capabilities of NetSparse when most CPU cores are not available to speed-up communication.

7.8.2 Simulated System and Benchmarks

To evaluate NetSparse, we use the SST Simulator [96], with the Merlin [319] component library for the network and switches, and DRAMSim3 [97] for DRAMs. We modify the rdmaNIC component of the simulator to model a NetSparse-augmented Pensando-like SNIC. Further, we augment the Merlin switches to model a Tofino-like architecture with

NetSparse extensions. Our communication experiments use communication patterns that are representative of the SpMM, SDDMM and SpMV kernels. For experiments including computation, we run SpMM with SPADE accelerators (Chapter 3) integrated with each host, again simulated with SST.

Table 29: System Parameters.

Cluster	
Topology	128 nodes; 8 racks; 16 nodes/rack; Leaf-Spine
Node	
Node	1 socket; 64 cores; 2.2GHz; DDR 256GB, 270GB/s
In-Node Accel.	128 SPADE PEs; 1GHz; HBM 64GB, 800GB/s
PCIe	Gen6; 128GB/s; 200ns one-way latency
SNIC	
General	AMD Pensando-like; 2.2GHz
RIG Units	DDR Memory 16GB, 64GB/s; 16MB LLC 32 RIG Units; 16 32KB L1s; 16 128KB L2s; 64/256-entry LSQ/Pending PR Table; 4KB Property Buffer/Idx Buffer
Network Interface	2MB RxBuffer/TxBuffer; 400Gbps; 1500B MTU
Network	
Bandwidth	400Gbps per link
Zero-load Latency	Switch: 300ns; Network link: 450ns one-way; Intra/inter-rack RTT: 2.4us/5.4us
Switches	Intel Tofino-like; 32 ports x 400Gbps; 8 pipes, 2GHz; 96MB Packet buffer; NetSparse extensions only in ToR switches
NetSparse Header Concatenation	Upper/Concat/PR-layers: 50B/12B/18B 512KB SRAM per switch pipe and per SNIC; NIC/Switch concat. delay cycles: 500/125; Deconcatenation has no delay cycles;
Property Cache	32MB total per switch; 16/512B min/max cache line; 32 segments; 16-way; 16 cycles lat; LRU

We simulate a 128-node machine with 400Gbps links. Table 29 shows the system parameters. The system has a Leaf-Spine network topology, with racks of 16 nodes (Figure 75). We assume an InfiniBand-like environment which applies backpressure when network queues get full. Packets are routed deterministically. In the SNIC, each pair of

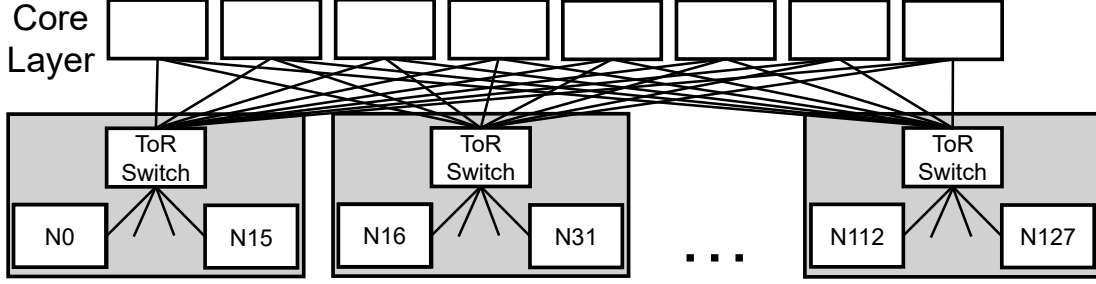


Figure 75: Simulated system topology.

RIG Units shares a single L1 and L2 cache (typically, a client and a server unit).

We use 5 large sparse matrices from the SuiteSparse [73] collection as benchmarks (Table 30). These matrices come from different domains such as web-crawls, road networks, and scientific applications. We evaluate them for properties with $K=\{1,16,128\}$ (i.e., 4B, 64B, and 512B). We use a RIG batch size of 32k nonzeros for arabic, queen, and stokes, and 8k for europe and uk. We also analyze the performance sensitivity to other values.

Table 30: Benchmark sparse matrices.

Matrix	arabic-2005	europe_osm	queen_4147	stokes	uk-2002
Rows (M)	23	51	4	11	19
NNZ (M)	640	108	317	350	298

7.8.3 NetSparse Hardware Overheads

NetSparse requires new hardware extensions to the SNICs and switches. To evaluate their area and power impact, we implemented the main logic structures (i.e., RIG pipelines and the Concatenators) in RTL using System Verilog. We synthesized our designs targeting a 45nm process with the Synopsys Design Compiler [320] and the open-source FreePDK45 cell library [321]. We verified that our designs easily meet timing for a frequency target of 1.5GHz at 45nm. Thus, a frequency of 2.2GHz for more modern 7-10nm process technologies is very reasonable [104]. For the caches, SRAMs, and CAMs in the NetSparse extensions, we used CACTI [102], and also validated that they meet timing. We scale down from 45nm to 10nm using the process scaling equations from [104]. We present our area

and power evaluation in Section 7.9.5. Dynamic power is reported under maximum activity.

7.9 Evaluation

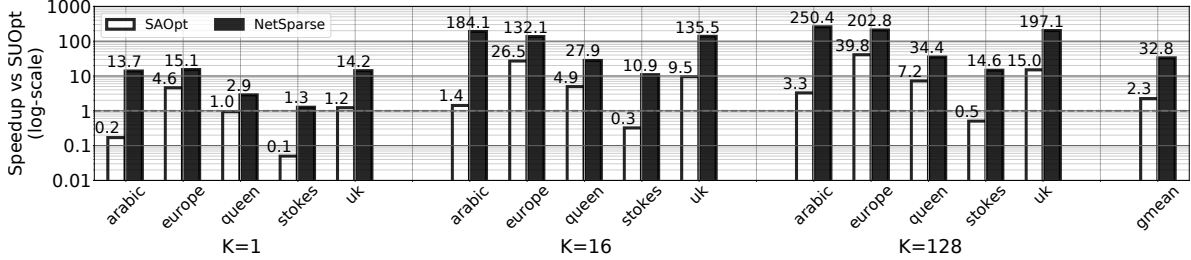


Figure 76: NetSparse and SAOpt communication speedups over SUOpt for the 128-node system.

7.9.1 Performance and Scalability Analysis

In this section, we analyze the performance and scalability of *NetSparse*. We start by comparing the time taken by the inter-node communication in NetSparse, SAOpt, and SUOpt assuming that computation has no cost. Figure 76 compares the communication speedup of NetSparse and SAOpt over SUOpt for different property sizes. SUOpt is favored by small properties ($K=1$), since the network traffic caused by redundant transfers is lower. For this reason, we see that the speedups of SAOpt and NetSparse increase for larger properties. We observe that NetSparse outperforms both baselines for all matrices and all property sizes. In contrast, SAOpt performs worse than SUOpt for stokes (all K values) and arabic (for $K=1$). The performance gain of NetSparse over SUOpt correlates directly with the number of redundant transfers SUOpt issues for each matrix (Table 25). On average (i.e., gmean), NetSparse outperforms SUOpt by $33\times$ and SAOpt by $15\times$.

Table 31 shows key statistics and reveals the reasons behind the communication speedups of NetSparse. We focus on the node that takes the longest to finish (*Tail Node*), since it determines the execution time. We see that *NetSparse* mechanisms work. First, PRs get filtered and coalesced, as revealed by the filtering and coalescing (F+C) rates.

Second, PRs get concatenated, as shown by the high number of PRs per packet. Third, we have hits in the Property Caches. For europe, the filter and cache hit rates are low because the matrix is very sparse; it is rare to find PRs with the same idx.

Table 31: Performance statistics for the tail node in NetSparse, and comparison to SUOpt and SAOpt (K=16).

Matrix	F+C Rate	Avg #PR per Pkt	PCache Hit Rate	Gput	Line Util.	-Trfc vs SU	Gput SA	-#PR vs SA
arabic	97%	5.7	26%	35%	65%	283x	1%	3.8x
europe	8%	4.5	5%	37%	70%	188x	10%	1.3x
queen	95%	19.6	50%	40%	66%	42x	11%	1.1x
stokes	90%	12.1	6%	38%	64%	17x	8%	4.4x
uk	61%	17	30%	30%	50%	271x	9%	2.6x

The main reason for the NetSparse speedup over SUOpt is the significant decrease in network traffic (Column 6 of Table 31). For example, arabic has 283x less traffic in NetSparse than in SUOpt. When compared to SAOpt, NetSparse eliminates software overheads and minimizes CPU involvement and synchronization. As a result, it raises the goodput to healthy levels, as can be seen by comparing Column 4 to Column 7. In addition, NetSparse also decreases the number of PRs relative to SAOpt due to more effective filtering (Column 8). For example, arabic has 3.8x fewer PRs in NetSparse than in SAOpt. This is because the Conveyors framework included in SAOpt assigns threads in the same node to different ranks and cross-rank filtering is not possible. SAOpt shows lower goodput for arabic in comparison with other matrices. This is due to intra-node load imbalance in the tail node, which gets magnified with SAOpt.

Next, we evaluate how the substantial communication speedups translate to end-to-end performance improvements. For this, we execute SpMM on the 128-node system, using the SPADE accelerators for computation. For communication, we use SUOpt, SAOpt, or NetSparse. SPADE does not support SpMM with K=1, so we exclude this property size from this experiment.

Figure 77 illustrates how the system strong-scales from 1 to 128 nodes with different

communication approaches. In other words, it shows the end-to-end speedup of the 128-node system over the execution on a single node. We also include, as a performance limit, the ideal speedup of a hypothetical system without any communication. Such speedup is shown with dashed lines.

Unsurprisingly, in the presence of hardware-accelerated computation, software solutions for communication are not enough. Due to the huge communication inefficiencies of SUOpt, the 128-node system is, on average, slower (i.e., its speedup is $0.7\times$) than the single-node system—which does not require communication. With SAOpt, the 128-node system is only $3\times$ faster than the single-node one. With NetSparse, hardware-accelerated computation is paired with hardware-accelerated communication, enabling end-to-end scalability. The 128-node system is $38\times$ faster than the single-node one. This is more than half of the speedup of a hypothetical ideal system, which reaches a speedup of 72.

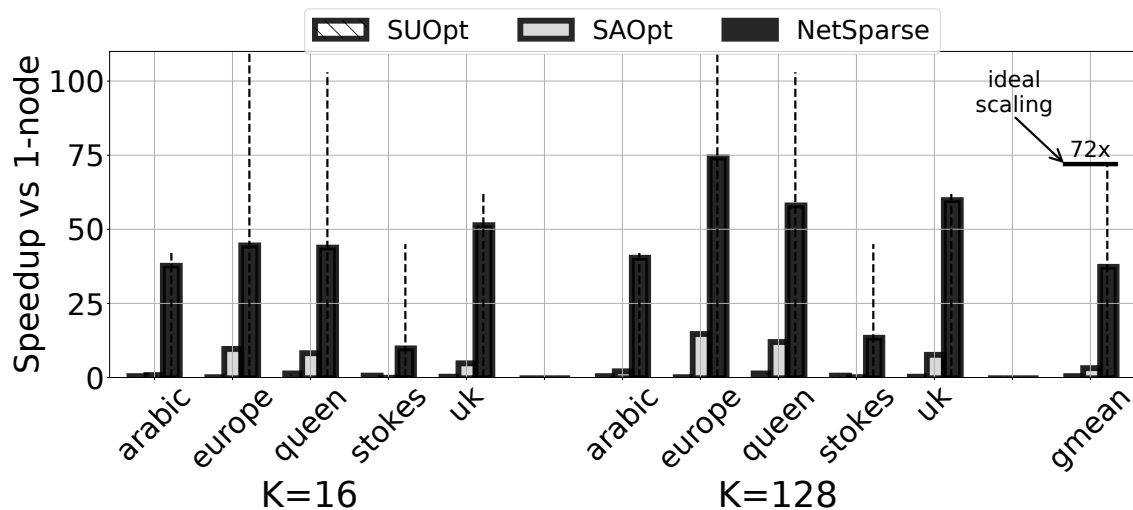


Figure 77: End-to-end speedup of a 128-node system over a single-node system for SpMM.

Figure 78 compares the communication to computation ratios for SAOpt and NetSparse. Note that, in practice, communication and computation (partially) overlap. SAOpt is dominated by non-hardware accelerated communication. With NetSparse, communication is accelerated and becomes comparable or faster than accelerated computation for arabic, queen, and uk. For europe and stokes, communication has still

non-negligible potential for improvement. An extra improvement would bring the end-to-end performance of those matrices closer to the ideal scaling, displayed in Figure 77. We discuss how this remaining potential could be exploited later.

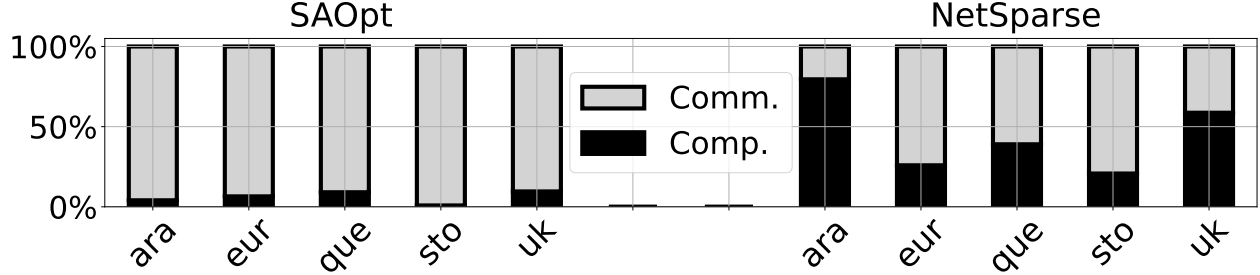


Figure 78: Communication/computation breakdown.

Table 32: Ablation analysis vs SUOpt.

arabic (denser)											
		K=1			K=16			K=128			
Optim.	Spd	-Trfc	Gput		Spd	-Trfc	Gput		Spd	-Trfc	Gput
RIG	0.2	3.1x	0.4%		1.8	5.3x	22.9%		3.6	8.5x	39.4%
Filter	3.4	17.4x	0.8%		34.2	63.2x	22.2%		78.7	109.0x	56.7%
Coalesce	8.4	34.6x	0.9%		88.0	118.5x	23.8%		184.8	279.9x	54.0%
Conc _{NIC}	12.6	80.8x	1.1%		129.1	191.2x	34.0%		184.2	310.7x	52.6%
Switch	13.7	115.5x	0.9%		184.1	283.4x	35.2%		250.4	398.0x	55.1%

europe (sparser)											
		K=1			K=16			K=128			
Optim.	Spd	-Trfc	Gput		Spd	-Trfc	Gput		Spd	-Trfc	Gput
RIG	7.4	8.8x	2.3%		82.8	99.1x	26.2%		176.0	248.0x	55.7%
Filter	7.5	8.9x	2.4%		84.8	100.8x	26.4%		175.5	252.2x	54.7%
Coalesce	8.1	9.6x	2.3%		91.3	108.7x	26.4%		190.3	272.1x	54.9%
Conc _{NIC}	14.1	19.1x	4.1%		122.1	159.6x	35.2%		197.8	292.9x	57.1%
Switch	15.1	24.8x	4.0%		132.1	188.4x	36.5%		202.8	301.5x	58.4%

7.9.2 Ablation Analysis

In Table 32, we perform an ablation study. We progressively apply each *NetSparse* mechanism, and show the resulting communication speedup (Spd), tail node traffic reduction (-Trfc), and tail node goodput for different property sizes. Speedups and traffic reductions are calculated with respect to SUOpt. We use arabic, which is denser, and

europa, which is sparser, to assess the impact for different sparsity patterns. We now discuss the effect of each optimization.

First, we use the SNIC RIG Units. Although not shown in the table, they improve the line utilization and, as a result, the goodput compared with SAOpt. Since RIG operations are sparsity-aware, they naturally lead to traffic reductions vs SUOpt (e.g., Columns 5 and 8). Although they are the most impactful optimization for europa, arabic still has a lot to gain after this optimization.

Next, by additionally applying *Filtering and Coalescing*, redundant PRs are eliminated and traffic is further reduced, especially for arabic. For europa, filtering and coalescing are less effective, since as explained, it is rarer to find PRs with the same idx for this matrix. With NetSparse, filtering and coalescing do not introduce software overheads and, thus, do not hurt the goodput.

We then apply *Concatenation at the NIC*. Concatenation decreases both network traffic and increases goodput by decreasing the per-PR header overhead. Naturally, concatenation is more effective when K is small (K=1,16). It benefits both matrices.

Finally, we include the NetSparse switch, with the Property Cache, concatenators and deconcatenators. The NetSparse switch decreases traffic for two reasons: cross-node concatenation is now possible, and some PRs are now served by the Property Cache. For the sparser europa, concatenation is the main reason: notice that there is comparatively less traffic reduction for K=128, which benefits less from concatenation. For arabic, which has a higher cache hit rate (Table 31), both reasons contribute to the traffic reduction.

7.9.3 Sensitivity Analysis

Next, we perform a sensitivity analysis on how different values of the NetSparse parameters affect performance for N=128 and K=16.

Figure 79 shows the speedup of the execution with different RIG batch sizes (i.e., the number of nonzeros in a RIG command), over the one with a batch size of 16k. The values

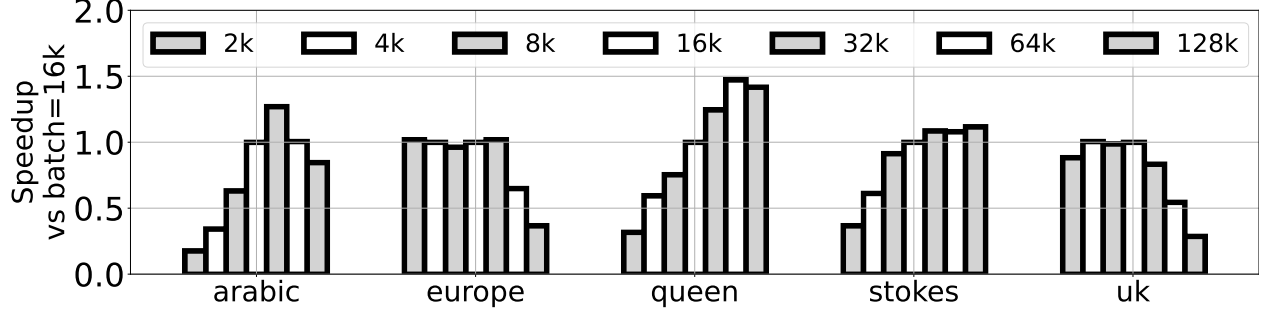


Figure 79: Sensitivity to RIG batch size.

that we have used by default are 32k for arabic, queen, and stokes, and 8k for europe and uk. As explained in Section 7.5, a small batch size can introduce software overhead, while a large one can introduce SNIC thread load imbalance. This explains why, for many matrices, the best batch size is not at the extremes. We also see that the optimal choice is input-sensitive: it is not the same for all the matrices, and sometimes is different than the one we used.

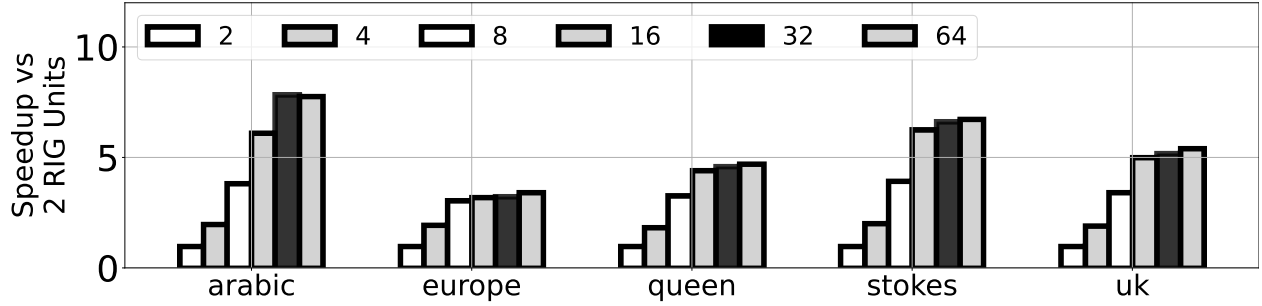


Figure 80: Sensitivity to number of RIG Units.

Figure 80 shows how performance scales with varying numbers of RIG units, using a 2-unit configuration (1 server + 1 client) as the baseline. Speedups grow until 32 units (our default design point), after which no matrix achieves significant additional gains.

In Figure 81, we show the effect of the concatenation delay cycles. We sweep the maximum cycles a PR can be delayed in CQs from 0 to 50000, and present the speedups over no concatenation. The figure refers to the SNIC delay cycles, but the switch delay cycles are also scaled. Our default design has 500 cycles. Initially, as the cycles increase, performance increases due to more PRs getting concatenated. However, when the delay

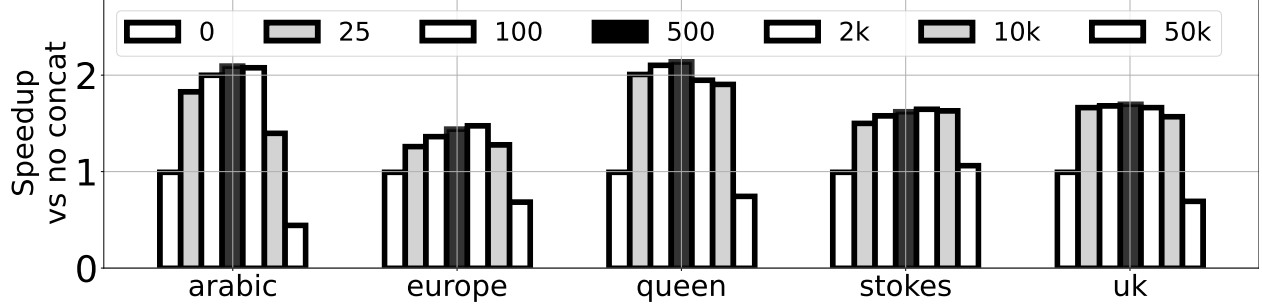


Figure 81: Sensitivity to concatenation delay cycles.

gets too large, performance gets worse than with no concatenation. Notice how concatenation effectiveness correlates with the temporal remote destination locality property (Table 28). Queen has the highest destination locality and gets the most benefit. In contrast, europe gets the least benefit.

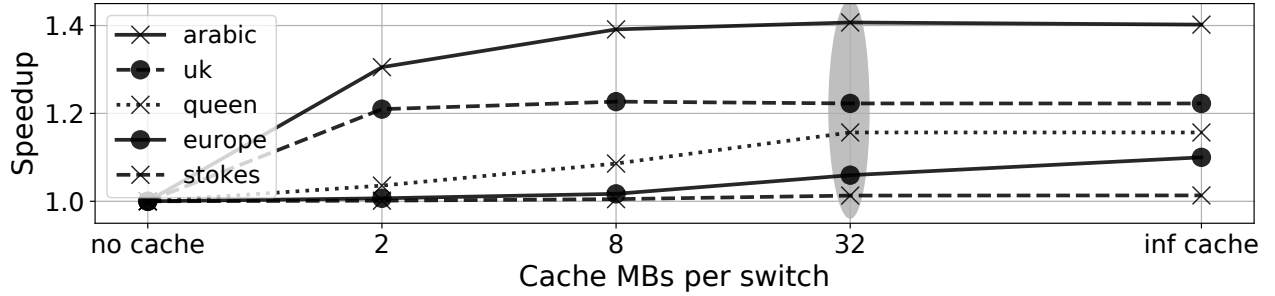


Figure 82: Speedup vs no-cache for different cache sizes.

Finally, we consider the impact of the Property Cache. Figure 82 shows the speedups vs no-cache as the cache size increases from 0 to infinite. The size we use by default is 32MB per switch. We see that caching can provide up to 40% improvement for arabic, while it does not improve stokes, regardless of the cache size. Also, we see that the cache size we chose is sufficient for most matrices, despite the fact that they are among the largest in SuiteSparse.

7.9.4 Remaining Performance Potential Analysis

Our results in Figures 77 and 78 suggest that matrices like stokes and europe could benefit from further communication improvements. We started the chapter by revealing a large

performance potential in Section 7.3 and then we introduced NetSparse to exploit some of this potential. Thus, two natural questions are: (1) how close did we get to the optimal communication performance, and (2) how could any remaining performance potential be exploited.

Consider the europe matrix. While the optimal traffic reduction vs SU is 582x (Table 25), we achieved 188x in Table 31. Further, in Table 25, although the line utilization is 70%, the goodput is only 37%. In summary, although we accelerated communication for europe by 132x (Figure 76 for $K=16$), there is still room for further gains. We now discuss the inefficiencies that prevent *NetSparse* from achieving the full potential.

A first reason is that concatenation in NetSparse is imperfect. In the best case, NetSparse reduces the effective header bytes to 18B per PR—a non-negligible overhead for fine-grained properties. A second reason is that the statically-selected values for the NetSparse parameters (e.g., the RIG batch size) are often nonoptimal. Future work could investigate techniques to dynamically adjust NetSparse parameters such as the RIG batch size.

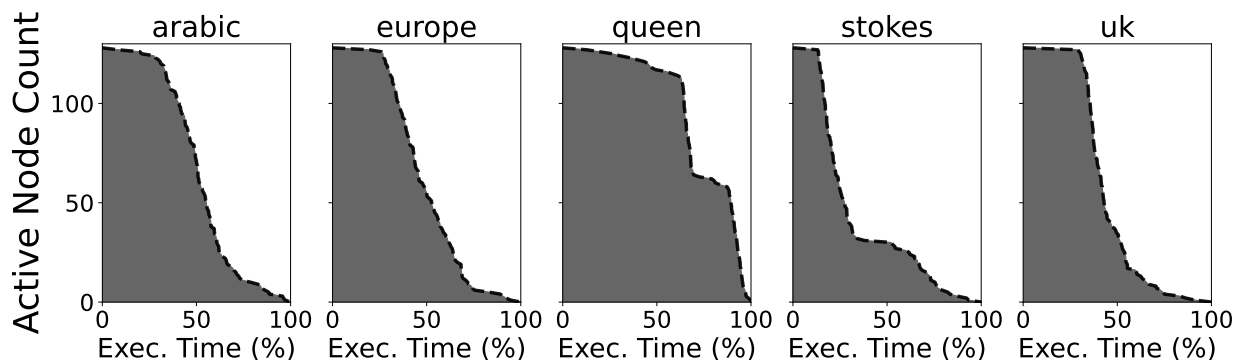


Figure 83: Inter-node load imbalance assuming no computation costs.

Another key reason is inter-node communication imbalance, which leads to suboptimal traffic reduction. Table 25 displayed the potential for *average traffic* reduction. However the NetSparse tail node, which determines performance, typically receives more than the average traffic. This limits the potential for execution time reduction. The problem is

shown in Figure 83, which illustrates the number of active nodes as the execution progresses, assuming no computation costs. It reveals that, almost all matrices (except queen) have significant imbalance in communication. This imbalance is not a consequence of the NetSparse hardware, but of the way the sparse matrix is partitioned across nodes. To maximize the performance with fast sparse networking hardware, future work should rethink partitioning methods and algorithms.

7.9.5 Hardware Overhead Analysis

This section discusses the area and power overheads of NetSparse in SNICs and switches.

1. SNIC overheads. The main overheads are the RIG Units, their L1 and L2 caches, and the Con/De-concatenation blocks (Table 29). For a 10nm process, Figure 84 shows our estimated static power, (maximum) dynamic power, and area for each structure in an SNIC. We see that the combined (worst-case) power is 2.1W and the area is 1.43mm². Given that a Pensando Elba chip with a 200Gbps network link consumes up to 50W [322], the power overhead of the NetSparse extensions at maximum activity is 4% and, when idle, 1%. Regarding the area overhead, using the Elba die floorplan [323] and the relative area that the 16 ARM Cortex A-72 block [324] consumes, we extrapolate a conservative die size of 300 mm² for the SNIC. Hence, the NetSparse area overhead is only 0.5%. The total storage overhead of all the NetSparse extensions is only about 3.5MB.

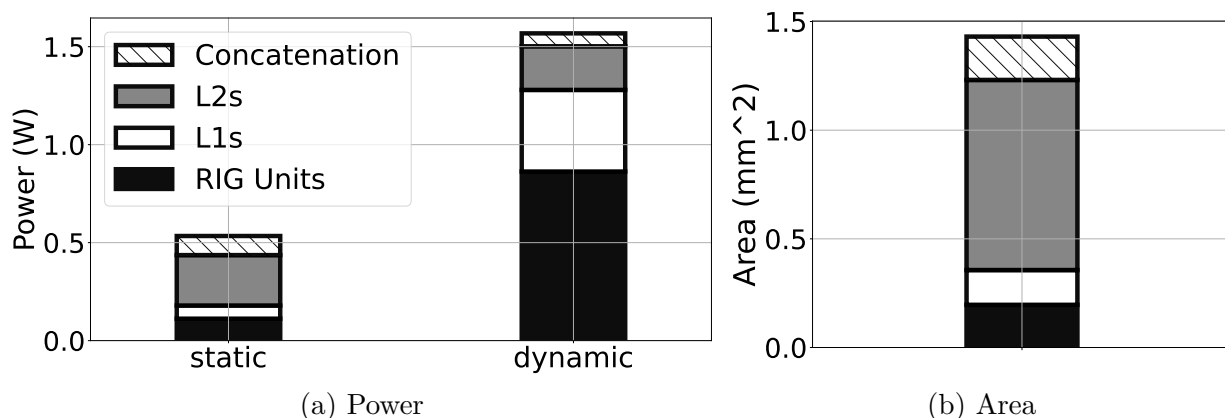


Figure 84: Power and area breakdown of the SNIC extensions.

The figure also shows that the L2s are the dominant source of area and static power, while the RIG Units account for most of the dynamic power. Further, Table 33 shows the contribution of each of the RIG Unit structures (Figure 69) to its area. The most area-hungry structure is the Pending PR Table, which is a CAM structure that book-keeps PRs pending in the network.

Table 33: Contribution of each structure to RIG Unit area.

Structure	Idx Buffer	Pend. PR Table	Prop. Buffer	LSQ	Rest
Area	12%	53%	12%	10%	13%

2. Switch overheads. The extensions in the ToR switches are the Property Caches, the Con/De-concatenation blocks, and the second crossbar. We estimate the area of the caches and concatenators to be 21.3 mm^2 and 1.5 mm^2 , respectively, and their combined power about 10W. Given a power consumption of 270W for a Tofino2 switch [325], the caches and concatenators add a 4% overhead. For area, given that modern high-performance networking switches are large ASICs that can reach 700 mm^2 [326], [327], [328], we estimate the area overhead for caching and concatenation to be about 3.2%.

Estimating the impact of the second crossbar is harder. Intel reports that the packet buffer, QoS circuitry, and ingress-to-egress routing logic account for a combined area of 25% in a Tofino2 switch [325]. However, a large fraction of this area is likely consumed by the SRAMs of the packet buffer. In our design, we have used input-queued crossbars [329] with virtual output queues [330] and split the packet buffer between the two crossbars without duplicating it. Further, the literature [189], [328] reports that a stand-alone (i.e., without cross-point buffering [331]) 32×32 switch crossbar like the one we use can be realized in less than 5 mm^2 (i.e. 1% of the switch area). Nevertheless, due to limitations in publicly available data, we can only place the area overhead of the extra crossbar and inter-pipe routing at a range of 1–15%.

We believe this switch overhead is justifiable for inter-chip networking of sparse accelerators in high-value sparse applications. Dense accelerator clusters like the TPU

supercomputer [10] already have their own custom networking.

7.9.6 Performance Analysis for Other Settings

We now evaluate NetSparse for other system settings.

1. Using NetSparse when Computation is Performed by a CPU rather than an Accelerator. In this section, we investigate the effect of replacing the SPADE accelerators with CPU cores for computation. For this, we calibrated our simulator using SpMM runs on two Intel Sapphire Rapids CPU servers [149]: a 48-core with DDR, and a 56-core that also includes HBM [332] (with similar bandwidth to the one modeled for the SPADE PEs). Measurements were taken using the optimized MKL Inspector-Executor routine [236].

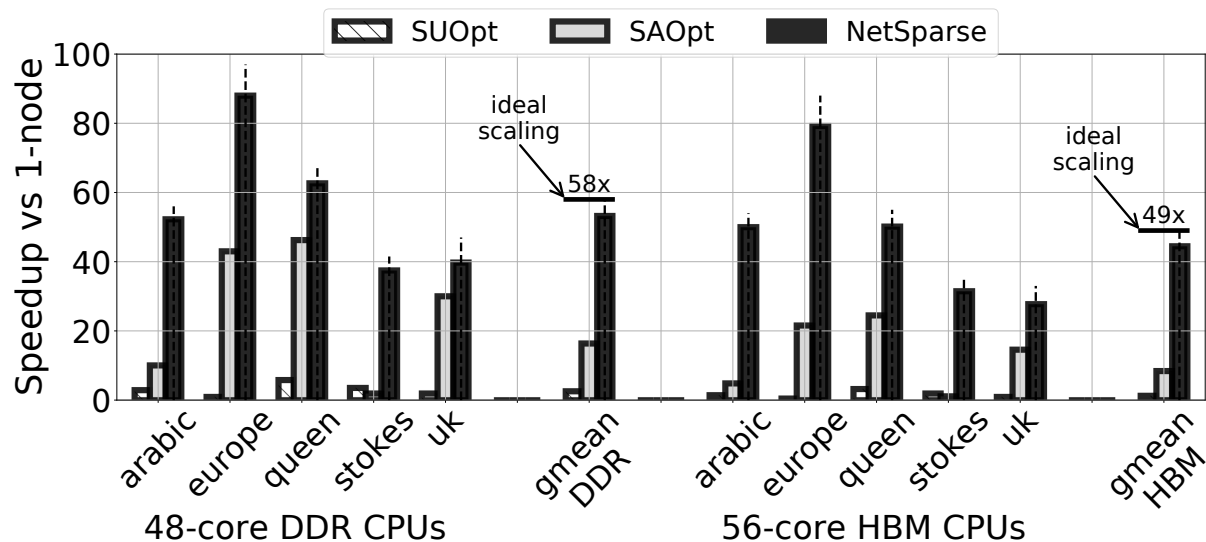


Figure 85: End-to-end speedup of a 128-node system over a single-node system for SpMM using CPUs for computation.

Figure 85 shows the end-to-end speedup of the 128-node system over the execution on a single node for SpMM (for $K=128$) using CPUs for computation. The figure is organized as Figure 77, with bars for SUOpt, SAOpt, and NetSparse, and dashed lines for a hypothetical system without any communication.

We see that NetSparse significantly outperforms both SUOpt and SAOpt, and approaches ideal scaling. The scalability of SAOpt for the DDR setting is still

disappointing, although less than in Figure 77. This is expected, as communication inefficiencies are less exposed when computation is not accelerated as much. Since SpMM is very sensitive to the memory bandwidth, SPR+HBM accelerates local computation more than SPR+DDR. Hence, communication inefficiencies become more pronounced, and the scalability of SUOpt and SAOpt drops. On average, across $K=128$ and $K=16$ (not shown), the 128-node DDR system becomes 2.6x, 13x, and 53x faster than the single-node one with SUOpt, SAOpt, and NetSparse, respectively. The same numbers for the 128-node HBM system are 1.4x, 7x, and 42x. Overall, even with CPU computation, NetSparse delivers much higher speedups than SUOpt and SAOpt.

2. Using Other Network Topologies. We simulate machines with two other network topologies. One is HyperX [333] with 32 switches organized in a 3D 4x4x2 topology, with 4 hosts per switch, and 4 cross-switch links in every dimension (i.e., HyperX width of 4). The other is Dragonfly [311] with 32 switches divided into 4 groups, 4 hosts per switch, 4 inter-group links, and minimal routing. These 2 network configurations have similar bisection bandwidth as our original Leaf-Spine one but have a higher diameter. We evaluate all networks with the same per-link latency.

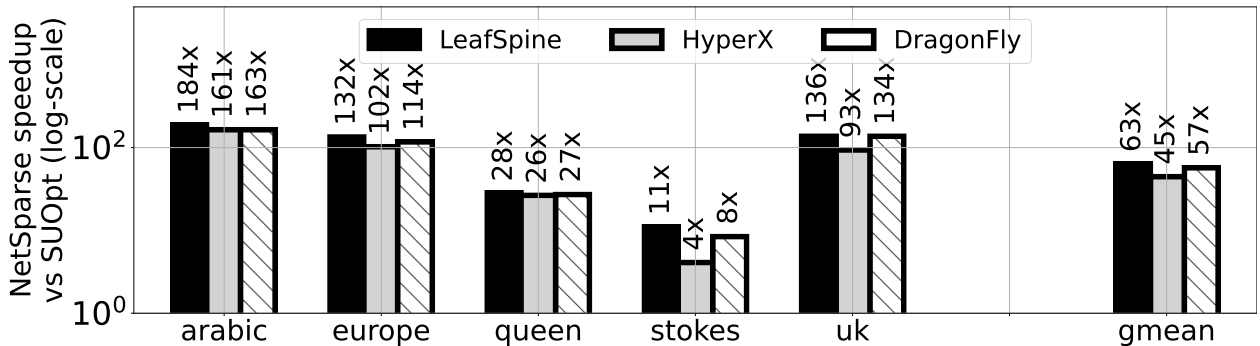


Figure 86: Communication speedup of NetSparse over SUOpt for different network topologies.

Figure 86 shows the communication speedup of NetSparse over SUOpt for the three different network topologies, for 128 nodes and $K=16$. We see that the performance of NetSparse remains high for all three networks. One notable exception is stokes with

HyperX, where the speedup drops by more than 2x compared to the Leaf-Spine network, for which we originally designed NetSparse. This performance degradation is due to the higher number of hops in the HyperX network.

7.10 Related work

In-Network Computing: Modern smart networking hardware such as SNICs [294], [296], [297], [298], [299], [300] and programmable switches [301], [302], [307], [334] have enabled a wide variety of in-network computing applications. Such applications range from dense machine learning [288], [335], [336], [337] and Key-Value Stores (KVS) [308], [338], to coherence, persistence, and consistency [339], [340], [341], [342], [343]. In-network caching has been proposed before for KVS [308], [338]. However, sparse computations differ significantly from both KVS and dense machine learning. Flare [344] introduces a switch design that supports collective reduction operations of sparse data, which is useful for accumulating sparse gradients during ML training. To the best of our knowledge, *NetSparse* is the first work to introduce in-network hardware extensions for general distributed sparse kernels.

Hardware and Software for Sparsity: The importance of sparsity in computing has motivated both hardware and software research. On the hardware-front, accelerators [79], [80], [81], [131], [132], [202], [287], [345] aim at optimizing sparse computations on a single node. At the same time, a variety of SA [289], [290], [291], SU [74], [75], [267], [346] and hybrid (Chapter 6) software efforts aim at scaling sparse computations across multiple nodes. *NetSparse* proposes hardware mechanisms that target the inefficiencies of software-based sparse communication.

7.11 Conclusion

This chapter introduced novel hardware mechanisms to improve communication in distributed sparse computations. Our proposal, called *NetSparse*, consists of four

mechanisms: remote indexed gather in the SNIC, filtering of redundant requests, concatenation of multiple requests into a single packet, and hardware caches in switches to store fetched data from remote racks. By reducing the network traffic, and improving the bandwidth utilization and goodput, *NetSparse* enables sparse accelerators to scale when distributed across the nodes of a large cluster.

Chapter 8 Heuristic Optimization of Hardware-Accelerated Sparse Computations

8.1 Introduction

Given the widespread use of Sparse Matrix Dense Matrix Multiplication (SpMM), a variety of hardware accelerators for SpMM have been proposed [79], [80], [81], [91], [119], [202]. Optimizing SpMM on such accelerators is a challenging task, as the density and pattern of the nonzeros (i.e., the sparsity pattern) in the sparse matrix determine the characteristics of the kernel. For example, SpMM is memory-bound when the sparse matrix has high sparsity, whereas it is compute-bound in low sparsity scenarios.

In many sparse matrices, nonzeros form dense and sparse regions, rather than being uniformly distributed across the whole matrix. For example, consider power-law graphs [347]. In such graphs, most of the edges are associated with a small number of nodes. Consequently, the nonzeros in the corresponding sparse adjacency matrix are not uniformly distributed across the whole sparse matrix. We refer to this property as *Intra-Matrix Heterogeneity (IMH)*.

Currently, accelerator designs do not leverage IMH. Instead, they are built out of homogeneous processing elements (PEs) with identical compute capacity, latency tolerance, and interface to the memory subsystem. Some designs provide flexibility to leverage different sparsity patterns across *different* matrices but are not designed to exploit IMH. For example, the SPADE architecture (Chapter 3) offers three different flexibility knobs that can be configured on a per-matrix basis. AESPA [135] introduces the concept of heterogeneous subaccelerators that target different levels of sparsity and compression formats. However, AESPA does not map different regions of a given matrix to the subaccelerators that would suit them the most, and thus fails to exploit IMH.

In this work, we argue that combining heterogeneous PEs in a single accelerator and carefully mapping different regions of a matrix to the most appropriate PE type to leverage

IMH can substantially optimize performance. Hence, we investigate the potential of heterogeneous accelerator architectures to leverage IMH. The denser regions of a sparse matrix are associated with higher arithmetic intensity and, therefore, are more suitable for PEs with higher computing capability. On the other hand, the sparser regions are dominated by memory accesses and, therefore, are more suitable for PEs with higher latency tolerance capabilities. Based on these ideas, we utilize heterogeneous accelerators that incorporate two types of workers (i.e., PEs): *Hot Workers*, which are more appropriate for compute-bound regions (i.e., *hot tiles*), and *Cold Workers*, which are more appropriate for memory-bound regions (i.e., *cold tiles*).

Our approach partitions the sparse matrix between the two types of workers to maximize performance. However, this is not a straightforward task. First, we need a model to estimate the performance of each worker for a given region. Then, we need a method to map each region to the appropriate worker type. Our evaluation shows that using holistic performance models is insufficient, since such models do not account for IMH. In addition, for a fixed collection of hot and cold workers and a set of regions, the problem of optimally scheduling regions to workers requires an exhaustive search over an exponentially large number of possible options [348], [349].

To address these problems, we develop (1) an analytical model that quickly predicts the performance of a worker for a given region, and (2) a heuristic-based technique to partition the matrix among the different worker types to maximize overall performance. First, for each sparse matrix region, we analytically estimate the execution time and number of main memory accesses for each worker type. Then, we apply a lightweight algorithm to partition the matrix among the different worker types in polynomial time. We collectively refer to our IMH-aware modeling and partitioning method as *HotTiles*.

We evaluate *HotTiles* through simulation for three distinct heterogeneous accelerator architectures: (1) a combination of SPADE (Chapter 3) and Sextans-like [81] PEs integrated into the same die and sharing memory; (2) a combination of SPADE PEs with

off-die Sextans-like PEs with enhanced computational capabilities; and (3) Intel’s PIUMA architecture for graph analytics [106]. Our results show that the combination of heterogeneous PEs and our IMH-aware modeling and partitioning technique is very effective. Depending on the architecture, heterogeneous execution with *HotTiles* outperforms homogeneous execution using only hot or only cold workers by $9.2\text{-}16.8\times$ and $1.4\text{-}3.7\times$, respectively. In addition, *HotTiles* outperforms the best worker type used on a per-matrix basis by $1.3\text{-}2.5\times$. Finally, *HotTiles* outperforms an IMH-unaware heterogeneous execution strategy by $1.4\text{-}2.2\times$.

Overall, this chapter’s contributions are:

- The proposal of using heterogeneous SpMM accelerators to leverage intra-matrix heterogeneity in sparse matrices.
- An analytical modeling framework to predict the performance of different accelerator PEs that takes into account intra-matrix heterogeneity.
- A heuristic that optimizes the performance of heterogeneous SpMM accelerators by partitioning sparse matrices among different worker types.
- A simulation-based evaluation for three different heterogeneous SpMM accelerator architectures.

8.2 Background

The reader is referred to Section 2.2 for a refresher on SpMM computation, memory access patterns, and data-reuse behaviors. Since this chapter emphasizes the different forms of data reuse present in the dense input matrix B and dense output matrix C , we refer to them as the Dense Input (Din) and Dense Output ($Dout$), respectively. We now provide additional background on a generalized form of SpMM and on the architectural differences among SpMM accelerator processing elements.

8.2.1 Generalized SpMM (gSpMM)

A generalized version of SpMM (gSpMM) over algebraic semirings is presented by Davis et al. [350]. gSpMM has the same memory access pattern as SpMM but can have different arithmetic intensity. Similar to SpMM, dense rows indexed by the `r_id` and `c_id` are accessed for every nonzero. However, the operations involved differ. The multiplication operation is substituted by the generalized multiplicative monoid $\otimes$, while the addition operation is substituted by the generalized additive monoid $\oplus$. Depending on the computational cost of these monoids compared to their vanilla versions, the arithmetic intensity of the kernel can increase or decrease. The generalized version of SpMM is an important primitive for GNN applications [116].

8.2.2 Processing Elements Used in SpMM Accelerators

In this section, we give a high-level description of the processing elements (PEs) that we use as building elements for our heterogeneous accelerator architectures.

SPADE: The SPADE accelerator, presented in Chapter 3, has lightweight PEs (Figure 87(a)) that are out-of-order (OoO) non-speculative vector engines. The PEs operate on tiles of the sparse matrix. The SPADE architecture includes three levels of caches and a Bypass Buffer (BBF) that bypasses the cache subsystem. The BBF is used to access the sparse matrix A and, depending on A , potentially the dense input Din (i.e., B in Section 2.2) and/or dense output $Dout$ (i.e., C in Section 2.2) matrices. The out-of-order PE design overlaps many outstanding memory accesses with computation and allows for latency tolerance and scalability.

Sextans: Sextans [81] is an accelerator for SpMM that relies on streaming accesses for both the dense and sparse structures. It employs large scratchpads in order to utilize the reuse of dense rows (Figure 87(b)). Similar to SPADE PEs, Sextans executes computations in a SIMD manner and offers high computational throughput. Since Sextans relies on a

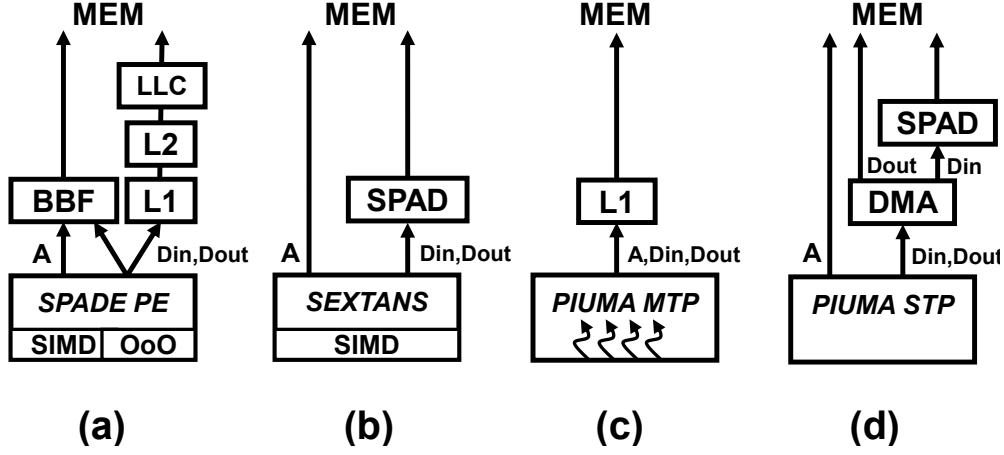


Figure 87: PEs used in our SpMM accelerators.

scratchpad, full dense tiles should be streamed in and out, regardless of whether all their contents are necessary in a given sparse matrix tile. As explained later in Section 8.3.1, PEs employing scratchpads are more effective for relatively denser sparse matrix regions and less effective for sparser ones.

PIUMA: PIUMA [106] is an architecture designed by Intel to address graph analytics challenges at scale. The PIUMA architecture includes two types of pipelines: Multi-Threaded Pipelines (MTPs) and Single-Threaded Pipelines (STPs). The MTPs are designed to tolerate high memory access latencies through fine-grained round-robin multithreading, whereas the STPs are simple in-order cores designed for single-threaded tasks (Figures 87(c) and (d)). Both STPs and MTPs implement the same custom RISC ISA. In addition to the pipelines, the PIUMA architecture includes scratchpads, DMA engines capable of executing arithmetic operations near memory, collective engines, and atomic engines. In this work, we aim to accelerate SpMM by using the various heterogeneous features of the PIUMA architecture.

8.3 Motivation

In this section, we first motivate the need for heterogeneous PEs when processing heterogeneous sparse matrix regions. We then discuss the pitfalls of Intra-Matrix

Heterogeneity (IMH)-unaware performance modeling and partitioning in SpMM accelerator architectures with heterogeneous PEs.

8.3.1 Heterogeneous PEs for Heterogeneous Matrix Regions

We start with a motivating example of how different worker (i.e., PE) types are more appropriate for different sparse matrix regions. We refer to the workers that are better suited for compute-bound, denser regions as *Hot Workers*, while we refer to the workers that are better suited for highly memory-intensive, sparser regions as *Cold Workers*. We also refer to the tiles that are assigned to the corresponding worker type as *Hot* and *Cold*. The main tasks of an SpMM accelerator worker are reading the appropriate elements of the sparse input, dense input, and dense output; executing the SIMD multiply-accumulate (MAC) operation; and writing back the final dense output elements. Naturally, workers of different types execute and overlap these operations in different ways.

As an example, Figure 88 focuses on the task of reading the dense input (Din). Recall that the nonzero `c_ids` index the rows of Din that need to be accessed. The figure includes two workers: Worker 1 (cold), which does not store Din in any fast local memory (FLM), and Worker 2 (hot), which uses a scratchpad. We consider two different 3x3 tiles (T1 and T2) of a sparse matrix. T1 is highly sparse and contains only a single nonzero, while T2 is denser and contains five nonzeros.

Figure 88(a) shows the cold worker processing T1. Since the cold worker does not use any FLM and accesses Din on demand, it only issues a memory access for a single Din row during T1 execution. Figure 88(b) shows the hot worker processing T1. The hot worker is forced to transfer from memory to its scratchpad all three Din rows that might be accessed during T1 execution. This is because, unlike caches, scratchpads do not include a hardware mechanism for detecting and handling misses. Unfortunately, since only the first Din row is eventually useful, two of the main memory accesses are unnecessary.

Figures 88(c) and 88(d) illustrate the main memory accesses issued by the cold and hot

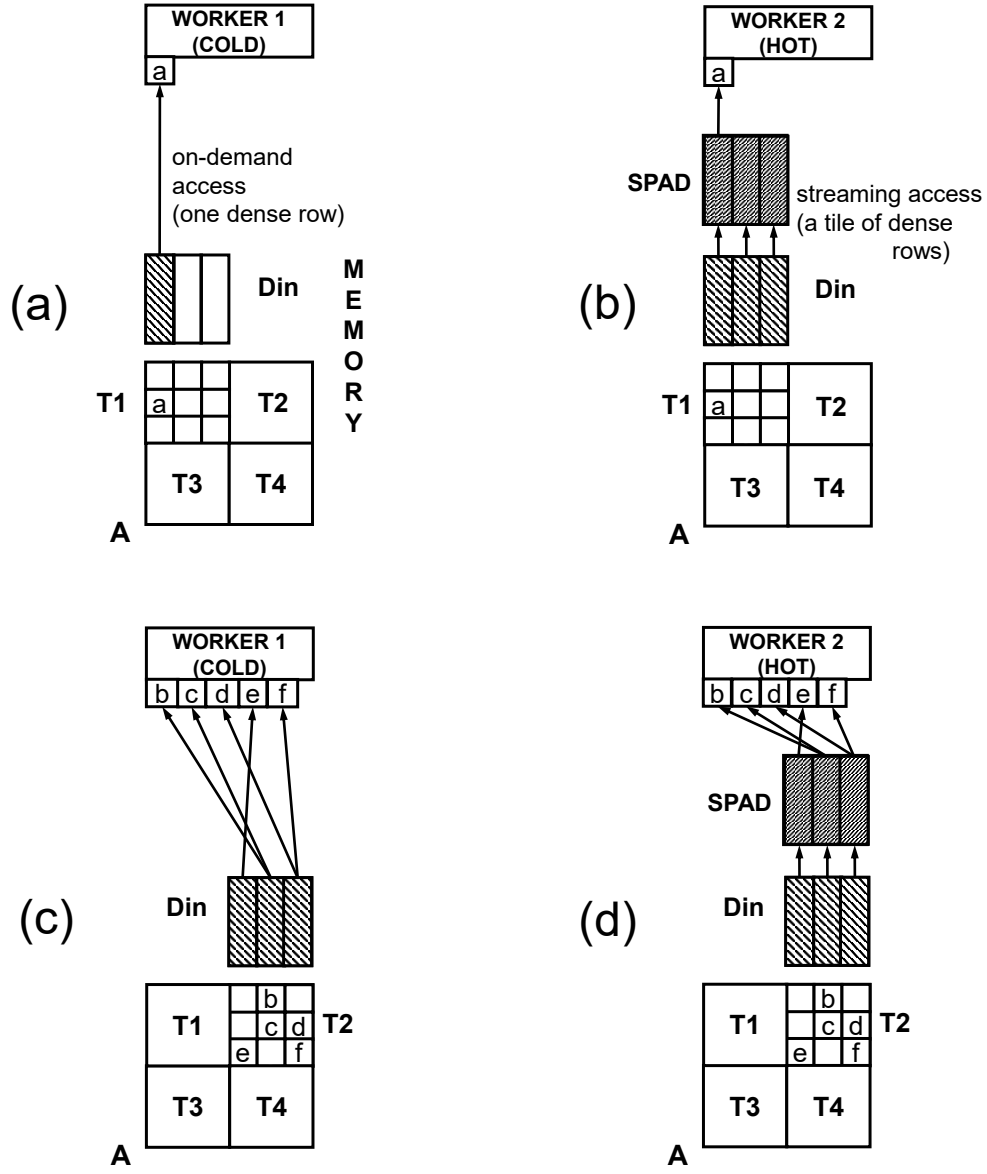


Figure 88: Number of memory accesses to Din for two different sparse matrix tiles and two different worker types.

workers, respectively, for the denser tile T2. The cold worker requests one row for every nonzero of the tile. Therefore, it requests a total of five rows from main memory. On the other hand, since the hot worker can reuse data through its scratchpad, it only requests three rows from main memory. Therefore, for T1, the cold worker accesses *Din* more efficiently than the hot worker while, for T2, the hot worker is better. Thus, we classify T1 as a Cold tile and T2 as a Hot tile.

In practice, accurately predicting which worker type is more suitable for each tile requires taking into account more factors than the number of memory accesses due to the presence of scratchpads. As a second example, consider two workers with scratchpads that have the following differences: (1) the cold worker can overlap memory requests, effectively hiding some of the memory access latency, and (2) the hot worker has higher computational capability. Assuming the same tiles as the ones in Figure 88, there is no difference in the amount of *Din* memory accesses—both workers access memory three times. For T1, both workers perform three *Din* memory accesses and one SIMD MAC operation (for nonzero a). For T2, both workers perform three *Din* memory accesses and five SIMD MAC operations. Thus, one can expect that, for T1, the cold worker will do better than the hot one, while for T2, the hot worker will likely do better than the cold one (depending on the relative cost of accessing memory and performing a SIMD MAC).

In fact, determining the appropriate worker for each tile is a much more complex task, since one should also consider the *Dout* reads and writes, the sparse input reads, and the overlapping of the different operations in each PE. In Section 8.4, we describe our IMH-aware performance models.

8.3.2 Pitfalls of IMH-unaware Modeling and Partitioning

In this section, we discuss a method to model and partition sparse matrices without considering IMH and discuss its pitfalls. We refer to the heterogeneous execution that stems from this modeling and partitioning as *IUnaware*.

We start by estimating the performance when processing the whole matrix by a single worker. We use the Roofline model [161], where the execution time is estimated as the maximum of the computation time and the memory access time. The computation time is the number of GFLOPs needed to process the whole matrix divided by the computational throughput of the worker. The number of GFLOPs needed is determined by the number of nonzeros in the matrix and is not affected by the nonzero distribution. The memory access time is the number of memory bytes accessed divided by the memory bandwidth. To estimate the number of memory bytes accessed, we assume a uniform distribution of nonzeros in the matrix, similar to AESPA [135]. With this model, we compute the predicted execution time with a single hot worker (th) and the predicted execution time with a single cold worker (tc).

We can now partition the processing of the matrix among hot and cold workers as follows. Assume we have N_{hw} hot workers and N_{cw} cold workers. If we use only the hot workers operating in parallel, the total execution time of processing the matrix can be approximated as $Ex_{hw} = th/N_{hw}$; if we use only cold workers, it can be approximated as $Ex_{cw} = tc/N_{cw}$. We would like to use all hot and cold workers working in parallel and assign a fraction of the tiles to each type of worker so that the total execution time is minimized. To find the fraction of tiles to be assigned to each type of worker, we use the technique proposed by Huang et al. [351]. Specifically, the fraction of tiles to be assigned to hot workers is:

$$frac_tile_{hot} = \frac{Ex_{cw}}{Ex_{cw} + Ex_{hw}} \quad (5)$$

Then, we distribute tiles randomly to the two worker types while ensuring that the total fraction of tiles assigned to hot workers satisfies Equation 5. This partitioning scheme, which we call IUnaware, resembles the one used by AESPA [135].

In Figure 89, we compare the heterogeneous execution using the IUnaware method to homogeneous executions, in which the whole matrix is assigned to either hot workers or to

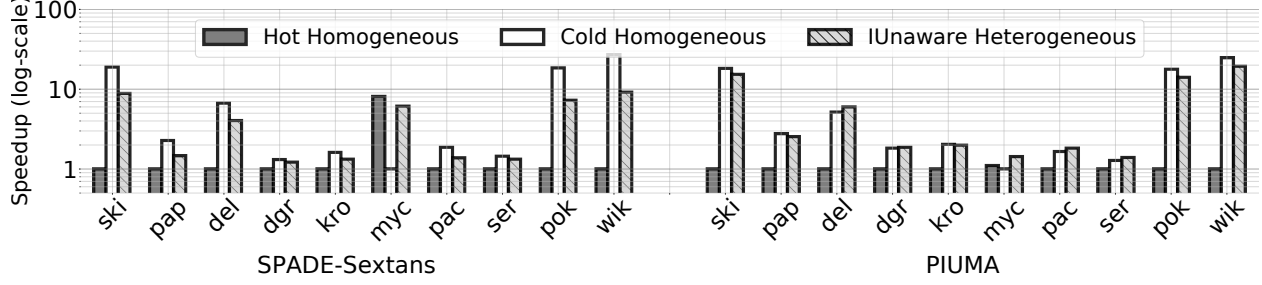


Figure 89: Comparing the performance of IUnaware and homogeneous execution. Speedup is calculated with respect to the worse homogeneous execution.

cold workers. We use ten matrices from SuiteSparse [73] (Table 38) with a tile size of 8192x8192, and two different heterogeneous architectures that will be described in Section 8.6. One of the architectures consists of SPADE (cold) and Sextans (hot) workers; the other consists of PIUMA MTPs (cold) and STPs (hot). For SPADE-Sextans, we use 16 cold workers and a single high-performance hot worker ($N_{cw}=16$, $N_{hw}=1$), while for PIUMA, $N_{cw}=4$ and $N_{hw}=2$. For each matrix, we present the speedups over the worst-performing homogeneous execution (hot or cold). Note that, for a given matrix, the three bars correspond to different numbers of workers used.

As shown in Figure 89, IUnaware delivers speedups over the worst-performing homogeneous execution for all the matrices and architectures. However, compared to the best-performing homogeneous execution, the results are unimpressive. Specifically, in PIUMA, IUnaware performs about the same as the best-performing homogeneous execution. Further, in SPADE-Sextans, IUnaware performs significantly worse than the best-performing homogeneous execution. The reason is that, in SPADE-Sextans, the cold workers already have enough compute power, and execution is mostly limited by memory accesses. Adding hot workers in an IMH-unaware manner as in IUnaware increases the pressure on the memory bandwidth without visibly improving the compute time. The result is an increase in execution time. Overall, our results reveal that heterogeneous execution with IMH-unaware modeling and partitioning is undesirable.

To improve the performance of heterogeneous execution, we propose assigning

individual tiles to the most appropriate worker type, using IMH awareness, as presented in Sections 8.4 and 8.5. We call our approach *HotTiles*. To see if HotTiles would generate an assignment of tiles to workers that is significantly different than in IUnaware, we generated Figure 90. The figure shows, for SPADE-Sextans (SP-SE), the tiles assigned to hot workers in black and the tiles assigned to cold workers in white, in IUnaware and in HotTiles. The data corresponds to the coPapersCiteseer (*pap*) matrix (Table 38) with an 8192x8192 tile size.

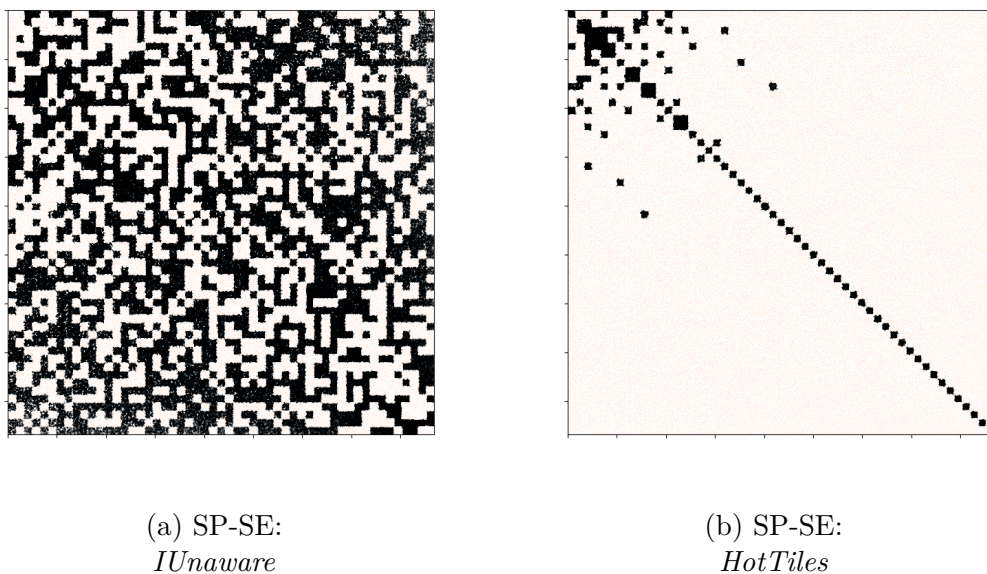


Figure 90: Assignment of tiles to hot workers (in black) and to cold workers (in white) in IUnaware and HotTiles for the *pap* matrix.

The figure shows that IUnaware and HotTiles assign the tiles very differently. IUnaware assigns tiles in a random manner, with the only constraint that the fraction of hot tiles satisfies Equation 5. HotTiles, instead, exhibits an obvious pattern: tiles classified as hot do cluster around the diagonal and in the upper left corner. Upon inspection, we find that the *pap* graph, which represents a paper citation network, forms denser sub-communities around the diagonal. This observation agrees with the findings in [139]. HotTiles realizes that these sub-communities are associated with higher arithmetic intensity and assigns them to hot workers. As a result, the percentage of nonzeros assigned to hot workers

changes from 52% in IUnaware to 72% in HotTiles.

8.4 Intra Matrix Heterogeneity (IMH) Aware Performance Modeling

The goal of an IMH-aware performance modeling methodology is to extract quantitative metrics that will enable us to determine a suitable worker type for each sparse matrix tile. The sparse matrix should be initially partitioned into tiles of a specific size. If one or both of the worker types use scratchpads to store *Din* and/or *Dout*, then the tile width and/or height are set to the largest value that does not overflow any of the worker’s scratchpads. For example, in Figure 88, the tile width is set to 3 elements, since any larger value would overflow the hot worker’s scratchpad. If none of the worker types uses scratchpads for *Din*, then the tile width is free and can be set to any value; if none of the worker types uses scratchpads for *Dout*, the tile height can be set to any value. For a free dimension, the IMH-aware modeling and partitioning methodology can be iteratively applied to find the value that is predicted to deliver the maximum performance. For the rest of this chapter, *tile_width* and *tile_height* will refer to the tiles of the sparse matrix.

After the tile height and width have been defined, we assume that each tile could be executed by a hot or by a cold worker. Then, we predict the tile’s execution time and the number of main memory accesses for each worker type. For the execution time, we assume that each worker is operating independently, without any other worker of the same or different type being active. Thus, when we compute the execution time, we ignore memory bandwidth contention. We account for the computation time, the memory access time (ignoring bandwidth contention), and their overlapping. Specifically, we take the maximum of the two times for workers that overlap memory accesses with computation, and the sum of the two times for workers that do not. We estimate the number of main memory accesses separately in order to account for potential memory bandwidth contention when multiple workers are operating in parallel.

8.4.1 Estimating the Number of Main Memory Accesses

When predicting the execution time, we must also estimate the number of main memory accesses. Consider a scenario where a worker type is estimated to be much faster for a given tile, but at the cost of significantly more main memory accesses. This may introduce significant pressure on the memory bandwidth, which is a shared resource of the heterogeneous architecture. For accurate prediction of the execution time, this effect should be taken into account.

The number of main memory accesses depends on the worker's fast local memory (FLM), the ordering of nonzeros in the sparse matrix (i.e., row or column ordered), the sparse matrix compression format (e.g., CSR or COO), and the matrix traversal order (i.e., the order in which the elements of the sparse matrices are accessed). Note that it is not necessary that all workers traverse the sparse matrix in a tiled manner. Depending on the traversal order, a worker may not need to finish processing all the nonzeros of a tile before processing nonzeros from another tile. Figure 91 illustrates untiled (Chart a) and tiled (Chart b) traversals of a sparse matrix with row-ordered nonzeros. The arrows represent the order in which a worker processes the nonzeros of its assigned tiles. The matrix traversal order directly impacts the number of accesses to main memory and affects the reuse behavior of D_{in} and D_{out} .

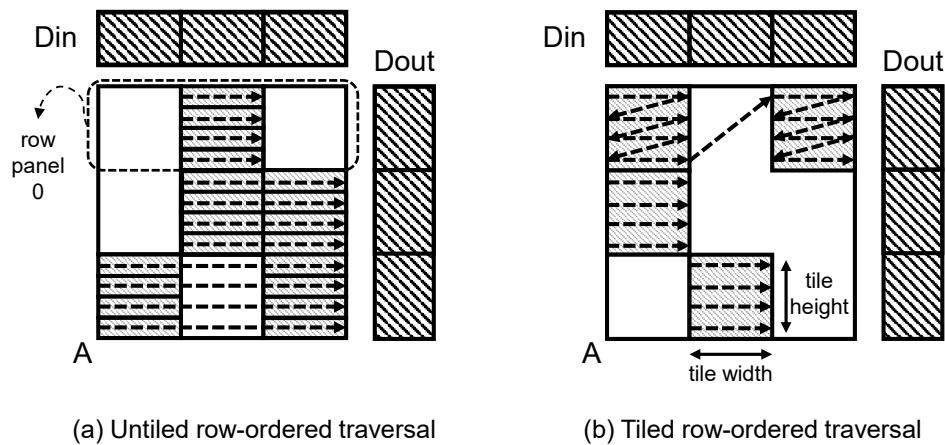


Figure 91: Untiled and tiled row-ordered sparse matrix traversal.

We consider four main types of data reuse:

Inter-tile: This reuse occurs when the dense rows needed for a tile have already been brought in to fast local memory by a previous tile. For example, consider a worker that stores *Dout* in its scratchpad. In this case, the *Dout* rows requested from memory by the first tile of a row panel (Figure 91) are reused by the rest of the tiles of the row panel.

Intra-tile (stream): This reuse occurs when a worker streams a full tile of dense rows into its scratchpad before the nonzeros of the sparse tile are accessed. The reuse of *Din* by the hot worker of Figure 88(d) falls into this category. For *Din*, a full dense tile includes *tile_width* rows, while for *Dout*, it includes *tile_height* rows.

Intra-tile (demand): This reuse occurs through registers or caches. For example, consider a sparse matrix where the nonzeros are row-ordered (Figure 91). When processing a tile, all the nonzeros with the same *r_id* access the same *Dout* row. Hence, when a nonzero brings a *Dout* row to registers or caches, the subsequent nonzeros with the same *r_id* can reuse it. As a result, the total number of *Dout* rows that will be accessed from memory is equal to the number of unique *r_ids* of the nonzeros in the tile (*tile_uniq_rids*).

None: This case happens when each nonzero ends up bringing from memory a dense row of the corresponding dense matrix. As an example, consider two nonzeros of *A* in Figure 91(a) with the same *c_id* and in consecutive rows. If, between processing the first and the second of these two nonzeros, there is a large number of nonzeros to process and the worker does not have a large enough FLM, *Din* rows will not be reused from the FLM.

The upper part of Table 34 shows, for the different reuse types, the number of dense rows from *Din* and *Dout* accessed from main memory during the processing of a tile of the sparse matrix.

The bottom part of Table 34 shows, for two sparse formats, the number of sparse input data items accessed from main memory during the processing of a tile of the sparse matrix. By data item, we mean, for example, the *r_id*, *c_id*, or *val* of a nonzero element in a COO-like format. For COO-like formats, $3 * \text{tile_nnz}$ data items are accessed per tile,

Table 34: Dense rows (upper subtable) and sparse input data items (bottom subtable) accessed from main memory during the processing of a tile under different reuse types and sparse formats. Tile refers to sparse matrix tiles.

Reuse Type	Dense Input Rows Accessed From Memory	Dense Output Rows Accessed From Memory
Inter-tile	0	0
Intra-tile (stream)	tile_width	tile_height
Intra-tile (demand)	tile_uniq_cids	tile_uniq_rids
None	tile_nnz	tile_nnz

Sparse Format	Sparse Input Data Items Accessed From Memory
COO-like	tile_nnz * 3
CSR-like	tile_height + tile_nnz * 2

since each nonzero is represented with an `r_id`, `c_id`, and `val`. For CSR-like formats, the `r_ids` array is substituted by an array holding the begin offsets of each row in the `c_ids` and `vals` arrays. Since each tile has $tile_height$ rows, a total of $2 * tile_nnz + tile_height$ data items are accessed per tile.

We refer to the total number of bytes accessed from main memory for tile i as bh_i or bc_i , depending on whether the tile is processed by a hot or a cold worker, respectively. Naturally, these bytes are the sum of the bytes from Din , $Dout$, and A .

8.4.2 Estimating Execution Time

To estimate the execution time, we model the time needed for each worker to perform all its five *tasks*: read the sparse input; read the dense input; read the dense output; execute the SIMD multiply-accumulate operation; and write back the dense output. For each nonzero, a SIMD multiply and accumulate operation is performed on rows with K elements (Section 2.2). Thus, the FLOPs associated with each sparse matrix tile are $2 * K * tile_nnz$. By dividing these FLOPs by the computational throughput of a given worker, we can get an estimate of the time needed by that worker to perform the computation. To estimate the time needed for the memory accesses, we multiply the number of bytes read/written from main memory in each task by a parameter that we call

visible latency per byte (vis_lat). This parameter captures the latency hiding that a worker type is capable of. Quantifying this parameter analytically is challenging. Hence, we determine it in a data-driven manner by measuring the runtime of homogeneous executions. More details about how vis_lat is obtained are given in Section 8.6.2.

To estimate the final tile execution time, we sum-up the estimated times of the five tasks, while accounting for any overlap of the tasks. For example, for a worker that overlaps all the tasks, the execution time is given by the longest task, while for a worker that does not overlap any task, the execution time is given by the sum of the times of all the tasks. Workers may overlap only some of these tasks.

In summary, the main novelty of our IMH-aware modeling approach over IUnaware is that we model the execution time of each individual tile, while IUnaware only models the execution time of the whole matrix. Modeling at tile granularity enables HotTiles to capture the unique sparsity pattern of the input matrix. This type of modeling requires taking into account the different inter- and intra-tile reuse types of Table 34. An additional novelty of HotTiles is the data-driven modeling of memory access latency hiding through the vis_lat parameter.

8.4.3 Modeling Limitations

In this subsection, we discuss two limitations of our modeling methodology. The first one results from the fact that, without knowing the final assignment of tiles to worker types (which is decided in Section 8.5), it is impossible to determine the reuse types of Din and $Dout$ in certain tiles. To understand this limitation, note that our assignment of tiles to workers in Section 8.5 takes all the tiles in a row panel and assigns all the tiles deemed cold to a *single* cold worker, and all the tiles deemed hot to a *single* hot worker. Moreover, the hot worker can reuse state across its tiles without caring about the potentially interleaved cold tiles in the same row panel assigned to the cold worker (and vice-versa).

Keeping this in mind, we give two examples of this limitation. First, consider a worker

that streams *Dout* to its scratchpad and implements a tiled row-ordered traversal (Figure 91(b)). The reuse type for the first tile assigned to it in a row panel is *Intra-tile (stream)*, while the reuse type for the remaining tiles assigned to it from the row panel is *Inter-tile*. However, without knowing the final assignment of tiles to worker types, it is impossible to determine whether a given tile is the first one of its type in the row panel or not. In our algorithm, we assume maximum reuse and, therefore, assume that a tile is never the first one of its type in the row panel.

In a second example, consider a worker that uses an untiled row-ordered sparse matrix *traversal* such as in Figure 91(a). In this case, the tile where a nonzero with a given *r_id* appears for the first time in the row has *Intra-tile (demand)* reuse for *Dout* for this *r_id*. All other tiles where a nonzero with this *r_id* appears have *Inter-tile* reuse. Again, without knowing the final assignment of tiles to worker types, it is impossible to determine whether a tile is the first one of its type with a nonzero with this *r_id* or not. As before, we assume maximum reuse and, therefore, assume that the tile is never the first tile of its type with a nonzero with this *r_id*.

In the tile assignment of Section 8.5, we initially use this maximum reuse assumption. After that, when we know the final assignment of tiles to worker types, we readjust the reuse types if needed when predicting the final execution time. It is evident that, since the tile assignment is done based on an imprecise model, the quality of the derived solution is slightly degraded.

The second limitation of our approach is that it disregards any reuse through caches. Caches could help exploit two types of reuse in Table 34: *Intra-tile (demand)* and *Inter-tile*. However, since caching effects are hard to model analytically, we make this pessimistic assumption.

Despite our assumptions of maximum reuse and no reuse through caches, our evaluation (Section 8.8) suggests that our modeling approach has low prediction error in the majority of cases.

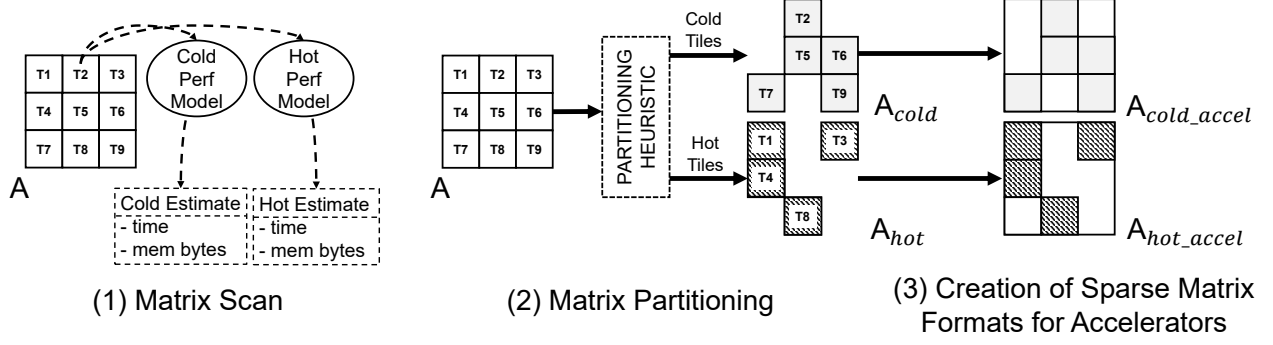


Figure 92: *HotTiles* preprocessing steps.

8.5 Intra Matrix Heterogeneity (IMH) Aware Partitioning

Figure 92 illustrates the preprocessing steps of our IMH-aware modeling and partitioning method, which we call *HotTiles*. First, the matrix tiles are scanned and fed to the cold and hot performance models of Section 8.4 to extract, for each tile and worker type, the estimated execution time and the estimated number of bytes accessed from main memory. Then, a partitioning heuristic is used to split the sparse matrix into cold and hot tiles. Finally, the cold and hot sections of the initial sparse matrix are stored in the appropriate sparse compression format as required by each heterogeneous accelerator. In this section, we focus on the partitioning heuristic.

8.5.1 Optimal Partitioning

Let us call th_i and tc_i the estimated times that a hot and a cold worker, respectively, take to execute tile i . Similarly, bh_i and bc_i are the estimated number of bytes read/written from memory if a hot and a cold worker, respectively, execute tile i . Then, the total execution time of all the hot workers processing the hot tiles in parallel (th_{total}) and of all the cold workers processing the cold tiles in parallel (tc_{total}) are:

$$th_{total} = \sum_{i \in hot} \frac{th_i}{N_{hw}} \quad tc_{total} = \sum_{i \in cold} \frac{tc_i}{N_{cw}} \quad (6)$$

and the total number of bytes read/written from memory by all the workers (b_{total}), all the hot workers (bh_{total}), and all the cold workers (bc_{total}) are:

$$b_{total} = bh_{total} + bc_{total} = \sum_{i \in hot} bh_i + \sum_{i \in cold} bc_i \quad (7)$$

Assuming that hot and cold workers are operating in parallel, and that BW is the total memory bandwidth of the heterogeneous architecture, the optimal partitioning is the solution to the following optimization problem:

$$minimize\{max\{max\{th_{total}, tc_{total}\}, \frac{b_{total}}{BW}\}\} \quad (8)$$

This optimization problem is not trivial. For example, it cannot be trivially solved by assigning each tile to the worker type that is estimated to be faster. This is because of two factors: (1) the workers operate in parallel and (2) the bandwidth saturation impacts the total runtime.

In addition, in some architectures, there is no mechanism to avoid data races when heterogeneous workers are writing to the same output memory locations. In this case, each worker type must update a private output buffer and the buffers are merged at the end of the execution. This introduces an additional t_{merge} cost term in the final execution time. This term can be estimated by considering the memory footprint of the output buffers and the system memory bandwidth. In this case, the optimization problem can be expressed as:

$$minimize\{max\{max\{th_{total}, tc_{total}\}, \frac{b_{total}}{BW}\} + t_{merge}\} \quad (9)$$

We assume a buffer accumulation design such that t_{merge} does not depend on the data that has been written in each buffer. Hence, t_{merge} has the same value for all the possible partitionings. With this assumption, the optimal solution for equations 8 and 9 is the same. In addition, for some sparse matrices, the t_{merge} cost might be too high to justify

parallel operation of the heterogeneous workers. Instead, in such cases it might be faster that the workers execute serially, using the same output buffer. In this case, the optimization problem involves minimizing the *minimum* of

$$\max\{\max\{th_{total}, tc_{total}\}, \frac{b_{total}}{BW}\} + t_{merge} \quad (10)$$

and the predicted runtime for the serial operation:

$$\max\{th_{total}, \frac{bh_{total}}{BW}\} + \max\{tc_{total}, \frac{bc_{total}}{BW}\} \quad (11)$$

Thus, the full optimization problem becomes:

$$\begin{aligned} & \text{minimize}\{\min\{\max\{\max\{th_{total}, tc_{total}\}, \frac{b_{total}}{BW}\} \\ & + t_{merge}, \max\{th_{total}, \frac{bh_{total}}{BW}\} + \max\{tc_{total}, \frac{bc_{total}}{BW}\}\}\} \end{aligned} \quad (12)$$

8.5.2 Proposed Heuristic Partitioning

In order to find the optimal solution, an exhaustive search over all the possible combinations of hot and cold tiles would be required. This corresponds to $2^{N_{tiles}}$ combinations, making the complexity of such an approach prohibitive. To solve the problem approximately, we decompose it into four simpler subproblems, each of which has $N \log N$ complexity, as explained later. Each of the subproblems produces a different partitioning. We then compare the predicted runtime of each of the four partitioning decisions and keep the one with the lowest predicted runtime. The four subproblems produce different heuristic-based partitioning decisions (*HotTiles heuristics*). They aim at either minimizing the execution time assuming that the system has sufficient memory bandwidth (*MinTime* heuristics) or at minimizing the bytes read/written from main memory (*MinByte* heuristics). In addition, they either assume that the heterogeneous

workers are operating in parallel (*Parallel* heuristics) or serially (*Serial* heuristics).

Each heuristic is expected to work best for different system configurations. For example, when one worker type is already able to saturate the memory bandwidth (which is a shared resource of the architecture), the *Serial* heuristics are expected to perform better. This is because bandwidth contention will limit the performance of *Parallel* heuristics, and the benefit of parallel execution will not outweigh the cost of merging the partial output buffers. In addition, in bandwidth-constrained configurations, the *MinByte* heuristics are expected to perform better than the *MinTime* heuristics since they reduce main memory accesses. Importantly, the effectiveness of each heuristic also depends on the structure of the input sparse matrix. For the same heterogeneous architecture, different heuristics may work best for different sparse matrices. We summarize the four heuristics in Table 35.

Table 35: *HotTiles* heuristics

Heuristic	Minimizes	Worker execution	Effective when memory bandwidth pressure is
MinTime Parallel	time	parallel	low
MinTime Serial	time	serial	medium
MinByte Parallel	bytes	parallel	medium
MinByte Serial	bytes	serial	high

Heuristic	Optimization Subproblem Objective	Tile Ordering	Cutoff Index Placement	Final Predicted Runtime
MinTime Parallel	$\text{minimize}\left\{\max\left\{\sum_{i \in \text{hot}} \frac{th_i}{N_{hw}}, \sum_{i \in \text{cold}} \frac{tc_i}{N_{cw}}\right\}\right\}$	increasing hot - cold execution time difference	$\xrightarrow{\text{Hot tiles}} \downarrow \text{Cold Tiles}$ 	$\max\left\{\max\{th_{total}, tc_{total}\}, \frac{b_{total}}{BW}\right\} + t_{merge}$
MinTime Serial	$\text{minimize}\left\{\sum_{i \in \text{hot}} \frac{th_i}{N_{hw}} + \sum_{i \in \text{cold}} \frac{tc_i}{N_{cw}}\right\}$	increasing hot - cold execution time difference	$\downarrow$ 	$\max\left\{th_{total}, \frac{bh_{total}}{BW}\right\} + \max\left\{tc_{total}, \frac{bc_{total}}{BW}\right\}$
MinByte Parallel	$\text{minimize}\{b_{total}\}$	increasing hot - cold mem bytes difference	$\xrightarrow{\text{Hot tiles}} \downarrow \text{Cold Tiles}$ 	$\max\left\{\max\{th_{total}, tc_{total}\}, \frac{b_{total}}{BW}\right\} + t_{merge}$
MinByte Serial	$\text{minimize}\{b_{total}\}$	increasing hot - cold mem bytes difference	$\xrightarrow{\text{Hot tiles}} \downarrow \text{Cold Tiles}$ 	$\max\left\{th_{total}, \frac{bh_{total}}{BW}\right\} + \max\left\{tc_{total}, \frac{bc_{total}}{BW}\right\}$

Figure 93: Optimization subproblems.

We now discuss how we derive the partitioning decisions by solving the four optimization subproblems (Figure 93). All of the optimization subproblems can be easily solved by sorting the tiles and performing a linear pass over the sorted arrays. For the

MinTime heuristics, we create an array that is sorted in increasing difference $th_i - tc_i$. Hence, tiles estimated to be executed faster by hot workers are placed first in the array. Then, we initialize a pointer to the beginning of the array, which represents the cutoff point between hot and cold tiles. We call this pointer *cutoff index* (Figure 93). We start moving the cutoff index to the right. Every time we move the cutoff index, the partitioning assignment changes and we calculate the new value of the subproblem optimization objective. If the value has decreased, we continue moving the cutoff index. Otherwise, we roll back to the previous position and the algorithm has converged, producing an approximate partitioning solution. Note that the optimization objective is different for the *MinTime Parallel* and *MinTime Serial* heuristics.

To estimate the final execution time of the resulting partitioning we use the formulas in the final column of Figure 93. Of course, the formulas are different for *MinTime Parallel* and *MinTime Serial*. Note that although we do not take into account the system bandwidth effect or the merging cost while deciding on the partitioning, we take these factors into account when determining the predicted runtime.

The procedure for the *MinByte* heuristics is similar. The only differences are: (1) the tiles are initially sorted in increasing difference of $bh_i - bc_i$ and (2) the optimization subproblem objective is different. Again, the final predicted runtime is different between *MinByte Parallel* and *MinByte Serial* heuristics.

Finally, we compare the predicted runtime of all four heuristics and keep the partitioning from the heuristic that produces the lowest runtime. Note that some architectures support special writes that avoid data races when heterogeneous workers perform read-modify-write operations to the same memory location. Then, there is no need for the output buffers, and t_{merge} is zero. In such cases, it can be shown that, under our model, there is no benefit in serial operation and, therefore, we only consider the *MinTime Parallel* and *MinByte Parallel* heuristics. Overall, the complexity of the proposed heuristics is $N \log N$: the array sorting is known to have $N \log N$ complexity and the cutoff

index placement can be completed in linear time.

8.6 Architectures and Framework

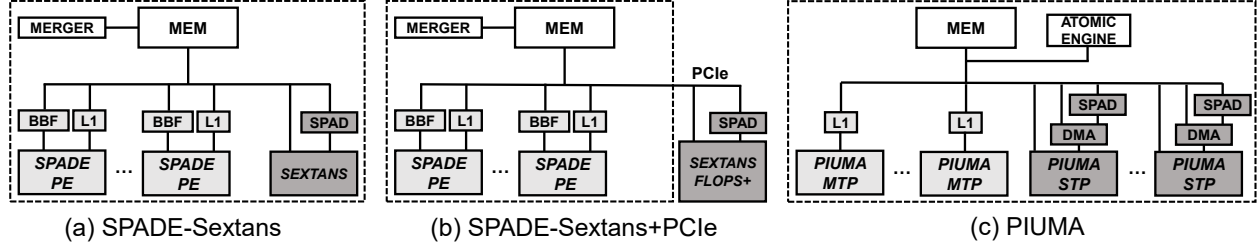


Figure 94: Heterogeneous architectures evaluated.

8.6.1 Heterogeneous Architectures

In this section, we analyze the three heterogeneous architectures used in our evaluation. They include PEs from the SPADE and Sextans SpMM accelerators, and from PIUMA, which can support SpMM and other kernels. In total, we use four different PE pipelines, with diverse characteristics (e.g., out-of-order, in-order, SIMD, and scalar). Table 36 describes the PEs, which we call workers. The different workers access sparse and dense structures like in Figure 87.

We evaluate both on-chip and off-chip PEs, as well as different mechanisms to avoid data races. We selected three architectures to emphasize the generality of our approach across multiple settings. Figure 94 illustrates our three heterogeneous architectures. We describe them next.

Table 36: Heterogeneous workers.

Worker	Worker Type	Sparse Format	<i>Din</i> Reuse	<i>Dout</i> Reuse
SPADE PE	Cold	COO-like	None	Inter-tile
Sextans	Hot	COO-like	Intra-tile (stream)	Inter-tile
PIUMA MTP	Cold	CSR-like	None	Inter-tile
PIUMA STP	Hot	CSR-like	Intra-tile (stream)	Intra-tile (demand)

(a) **SPADE-Sextans:** SPADE (cold) and Sextans (hot) PEs are integrated in the same die and share the same memory controllers (Figure 94(a)). For simplicity, we do not equip

the SPADE PEs with a 3-level cache subsystem as in Chapter 3; we only include a private L1 and a BBF. The SPADE PEs access the sparse input and dense output through the BBF, while they access the dense input through the L1. The SPADE PEs traverse the sparse matrix using an untiled row-ordered traversal (Figure 91(a)), while Sextans uses a tiled row-ordered traversal (Figure 91(b)). The SPADE PEs and the Sextans PEs can either operate in parallel or serially. To avoid data races when the SPADE and Sextans PEs are operating in parallel, we use two output buffers, one for each worker type. A Merger module is responsible for merging the output buffers at the end of execution. Note that in the serial operation mode, the Merger module is not used. To avoid data races between SPADE PEs, we closely follow the assignment and padding requirements discussed in Chapter 3. For example, cold tiles in the same row panel are always assigned to the same SPADE PE.

(b) SPADE-Sextans+PCIe: SPADE was proposed as an accelerator that is integrated in the same die as the CPU host. On the other hand, Sextans was proposed as a PCIe-attached accelerator. For our second architecture, we use on-chip SPADE PEs and an off-chip Sextans that accesses main memory through PCIe. We additionally assume that Sextans has enhanced computational throughput compared to the one in SPADE-Sextans.

(c) PIUMA: This architecture consists of MTPs, STPs, DMA engines, and an Atomic engine, all sharing the same memory subsystem. The cold workers are the MTPs, which are latency tolerant and access memory on-demand. The hot workers are the STPs, which we equip with scratchpads and with DMA engines to increase their ability to exploit memory-level parallelism. The STPs access the sparse input on-demand and issue DMA descriptors for *Din* and *Dout*. *Din* is stored in the scratchpad. The Atomic engine enables MTPs and STPs to perform read-modify-write operations on the same memory location without suffering data races. As a result, the MTPs and STPs always operate in parallel, and we only employ the *MinTime Parallel* and *MinByte Parallel* heuristics (Figure 93).

8.6.2 Framework Implementation Details

HotTiles is implemented as a framework that is fully integrated into the software pipeline that generates the sparse matrix format for each accelerator. The HotTiles software is executed on the host of the heterogeneous architecture. It reads a sparse matrix from disk in MatrixMarket [270] file format. It then analyzes it and partitions it into hot and cold tiles transparently to the users. Finally, it generates the sparse matrix formats. These formats, in combination with *Din*, can be directly accessed by the heterogeneous workers to execute the SpMM kernel. Alternatively, they can be stored for later use—e.g., they can be generated and used during GNN training and then saved and reused during GNN inference.

For the model, the users need to set the following traits of the heterogeneous architecture: the maximum computational throughput of each worker type (in GFLOP/s); the number of workers of each type; the size of the workers’ scratchpads; the shared main memory bandwidth (in GB/s); the *Din* and *Dout* reuse types and the sparse matrix formats (Table 34); and the way that each worker overlaps the SpMM tasks (Section 8.4.2).

The *visible latency per byte* (*vis_lat*) for each worker type is hard to determine analytically. For this reason, a small number of profiling runs are executed using a set of small test matrices. In each run, only one worker type is used. After these runs, *vis_lat* is automatically determined using a simple search method. The search objective is to minimize the error between the execution times predicted by our framework and the real execution times of the profiling runs. Note that this tuning process only needs to be done once when the *HotTiles* framework is first installed on a particular machine. The derived *vis_lat* values can be reused for SpMM runs with different sparse matrices and thus do not contribute to the preprocessing overhead. In future work, we aim at employing sophisticated search or machine learning techniques (e.g., similar to [352]) to automatically set the values of *vis_lat* and all the other heterogeneous architecture traits.

8.7 Methodology

8.7.1 Architectures and simulation

We evaluate our architectures through simulation. For SPADE-Sextans, we use SST [96] and DRAMSim3 [97]. For PIUMA, we use an in-house simulator based on Sniper [186], [353]. We preprocess the sparse matrices to generate the sparse matrix formats on a dual-socket 48-core Intel Xeon Platinum 8260M CPU [354].

For the SPADE-Sextans architecture, we test different system scales. We keep the PE frequency and the memory bandwidth constant, but we change: (1) the number of SPADE PEs and (2) the computational throughput and the scratchpad size of Sextans. The memory bandwidth is kept constant to evaluate our approach under different worker to memory bandwidth ratios.

Table 37 displays the parameters for the different system scales. We use the system scale 4 as the baseline scale. In all scales, the cache line size is 64 bytes, the PE frequency is 0.8 GHz, and the main memory bandwidth is 205 GB/s. Note that this memory bandwidth is the maximum theoretical value that the memory controllers can provide. The maximum observed bandwidth for the heterogeneous architecture is 161 GB/s at system scale 8. For the SPADE PE out-of-order pipeline, we use the parameters described in Chapter 3. Although SPADE supports different execution strategies, for simplicity, we set the SPADE PEs to execute an untiled sparse matrix traversal (Figure 91(a)), with each PE operating on a chunk of 64 continuous sparse matrix rows at a time. SPADE PEs use an untiled COO format, while Sextans operates on a tiled COO format. Sparse and dense matrix values are stored using single-precision floating point.

For the SPADE-Sextans+PCIe architecture, we assume that Sextans is connected to the on-chip memory bus through PCIe, which has a maximum bandwidth of 32GB/s. We use this architecture to evaluate gSpMM versions with higher arithmetic intensity. As we increase the kernel arithmetic intensity, more SIMD operations are required per nonzero.

Table 37: Architectural parameters for different SPADE-Sextans system scales. System scale 4 is the base one.

System Scale	SPADE Num PEs	SPADE SIMD MACs/Cycle	SPADE L1 Size (kB)	Sextans Num PEs	Sextans SIMD MACs/Cycle	Sextans SPAD Size (MB)
1	4	1	32	1	5	0.5
2	8	1	32	1	10	1
4	16	1	32	1	20	2
8	32	1	32	1	40	4

As a result, the SPADE PEs require more cycles to execute all the arithmetic operations per nonzero. However, we assume that the off-chip Sextans increases its computational power beyond the numbers in Table 37. Specifically, for system scale 4, Sextans can now process 20 nonzeros per cycle, regardless of the arithmetic intensity of the kernel. All the other system parameters are kept the same.

For the PIUMA architecture, we use 4 MTPs and 2 STPs. The STPs use a tiled CSR-like format, while the MTPs use untiled CSR. We choose to store the sparse and dense matrix values using double-precision floating point to show that the method generalizes to various data sizes and types.

8.7.2 Benchmark Matrices

We use ten square sparse matrices from SuiteSparse [73] as benchmarks. They have 100K to 4.2M rows and 22M to 100M nonzeros (Table 38). Most of the matrices (pap, del, kro, myc, pac, ser) are the same or scaled-down versions of the ones used in Chapter 3. Since the high fidelity of the PIUMA simulator leads to significantly increased simulation times, we replaced 4 of the matrices used in Chapter 3 with smaller ones to decrease the PIUMA simulation time. The 4 new matrices (ski, dgr, pok, wik) are selected so that they represent different application domains than the matrices borrowed from Chapter 3. We set the number of columns of the dense matrices (K) to 32.

Table 38: Benchmark sparse matrices used.

Benchmark	Short	Domain	Rows (Mill)	NNZ (Mill)	Density
as-Skitter	ski	Internet topology	1.7	22	$8 * 10^{-6}$
coPapersCiteseer	pap	Citation network	0.4	32	$2 * 10^{-4}$
delaunay_n22	del	Geometry problem	4.2	25	$1 * 10^{-6}$
dgreen	dgr	VLSI	1.2	27	$2 * 10^{-5}$
kron_g500-logn19	kro	Synthetic graph	0.5	44	$2 * 10^{-4}$
mycielskian17	myc	Math.	0.1	100	$1 * 10^{-2}$
packing-500x100x100-b050	pac	Numerical simulation	2.1	35	$8 * 10^{-6}$
Serena	ser	Environ. science	1.4	64	$3 * 10^{-5}$
soc-Pokec	pok	Social network	1.6	31	$1 * 10^{-5}$
wiki-topcats	wik	Web graph	1.8	29	$9 * 10^{-6}$

8.7.3 Heterogeneity Area, Power, and Control Overhead

Since PIUMA includes heterogeneous PEs by design, HotTiles can be supported without additional hardware overhead. This is not the case for the SPADE-Sextans architecture, where HotTiles adds some overheads associated with combining and controlling different PEs. Specifically, HotTiles needs some control logic to orchestrate the two subaccelerators. Since when the two subaccelerators are operating in parallel they are writing to private output buffers, all that is needed from the new control logic is: (1) to signal the beginning of operation of each PE, (2) to monitor the termination of operation of each PE, and (3) to signal the beginning of operation of the Merger module. This control functionality can be accomplished in software by utilizing the SPADE Control Processing Element (CPE), which is a general-purpose core that can write to memory-mapped registers (Chapter 3).

The only additional hardware module that needs to be integrated into the heterogeneous SPADE-Sextans architecture is the Merger module. It includes a SIMD ADD module and some registers. We estimate its area and power using CACTI [102] for the memory structures and the numbers from [103] for the SIMD arithmetic. We scale area and power to 10 nm using the scaling factors from [104]. We compare them to the area and power of a SPADE PE, including PE pipeline, L1 cache, and BBF. Our results suggest that

the Merger module has very small overheads. It accounts for less than 20% of the area and power of a single SPADE PE.

8.8 Evaluation

The evaluation is organized in three parts. Subsection 8.8.1 compares the performance of heterogeneous execution with *HotTiles* against other execution environments. Then, subsection 8.8.2 investigates the effectiveness of using *HotTiles* for architecture exploration. Finally, subsection 8.8.3 discusses the preprocessing cost.

8.8.1 Performance Evaluation

We compare the performance of heterogeneous execution with *HotTiles* against: (1) a homogeneous execution using only the cold workers of each architecture (*ColdOnly*); (2) a homogeneous execution using only the hot workers (*HotOnly*); and (3) a heterogeneous execution that partitions tiles based on the *IUnaware* method. Note that *IUnaware* is very similar to the partitioning technique used in AESPA [135]. We additionally compare against the *BestHomogeneous* baseline, which manually selects the best homogeneous strategy between *HotOnly* and *ColdOnly* on a per-matrix basis.

Figure 95 presents the results for SPADE-Sextans (with system scale 4), while Figure 96 presents the results for PIUMA. The figures show the speedup over the *worst-performing* homogeneous execution (hot or cold) on a per-matrix basis. We observe that *HotTiles* is very effective. It outperforms practically all the baselines. For SPADE-Sextans, *HotTiles* provides average speedups of 8.7x, 1.9x, and 2.0x over *HotOnly*, *ColdOnly*, and *IUnaware*, respectively. Although not shown, it also provides average speedups of 1.25x over *BestHomogeneous*. For PIUMA, the speedups are similar: 9.2x, 1.4x, and 1.4x over *HotOnly*, *ColdOnly*, and *IUnaware*, and 1.4x over *BestHomogeneous*. Note that, typically, *HotOnly* is the slower homogeneous execution due to the low density of the benchmark sparse matrices. The exception is the myc matrix, which is the densest

matrix in our evaluation. For this matrix, hot workers are significantly better than cold ones for SPADE-Sextans, but only slightly better for PIUMA. This is because the hot to cold worker computational throughput ratio in PIUMA is smaller than in SPADE-Sextans.

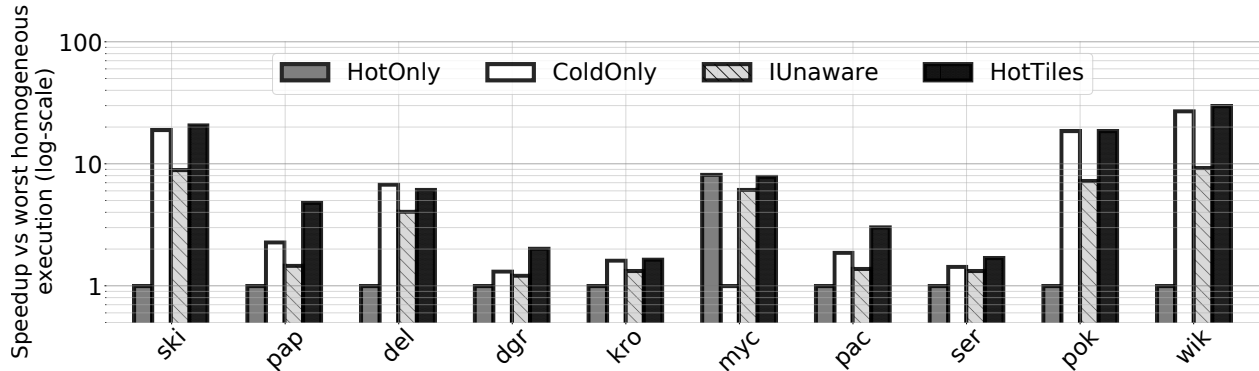


Figure 95: Comparison of homogeneous and heterogeneous execution for SPADE-Sextans.

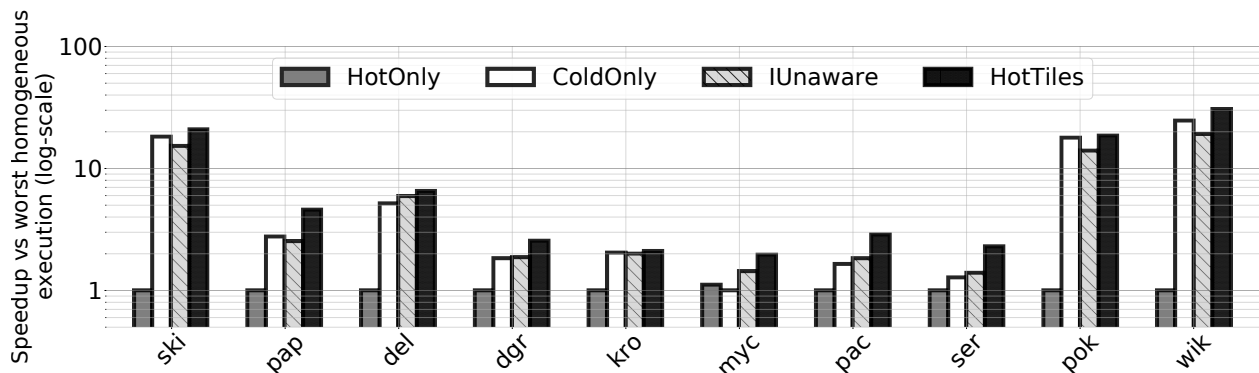


Figure 96: Comparison of homogeneous and heterogeneous execution for PIUMA.

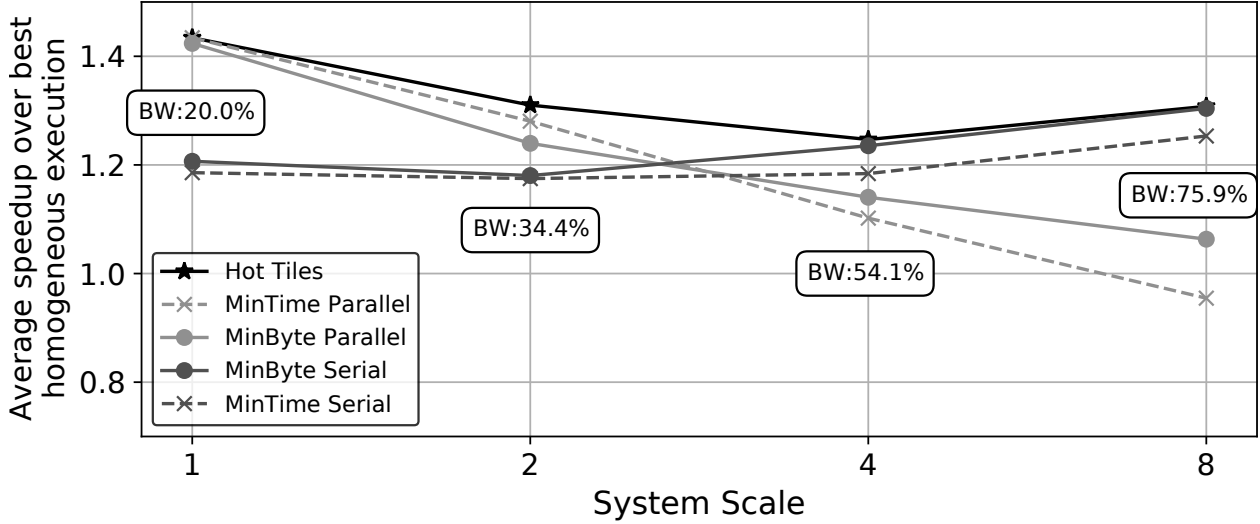
Table 39 displays the absolute simulated runtimes for the SPADE-Sextans architecture. Since the microarchitectural details of PIUMA are proprietary, we do not release the raw execution times in this use-case.

Next, we focus on the four *HotTiles* partitioning heuristics. Recall that during the partitioning step, they generate four different partitioning variants. Then, *HotTiles* selects the one that is predicted to take less time. To provide further insight, we tested the four heuristics for different scales of the SPADE-Sextans system (Table 37). Figure 97 compares the *actual* average performance of *HotTiles* to the average performance of the partitioning

Table 39: Runtime in ms for SPADE-Sextans.

Matrix	HotOnly	ColdOnly	BestHom	IUnaware	HotTiles
ski	369.4	19.8	19.8	42.3	18.3
pap	42.1	18.6	18.6	29.0	8.9
del	138.0	20.8	20.8	34.3	22.8
dgr	46.8	35.6	35.6	38.5	23.3
kro	61.3	37.9	37.9	46.3	37.2
myc	13.5	108.6	13.5	17.9	14.1
pac	37.7	20.2	20.2	27.4	12.6
ser	38.8	27.0	27.0	29.3	23.1
pok	539.0	29.7	29.7	75.3	29.7
wik	642.2	24.3	24.3	70.5	22.1

suggested by each individual heuristic. We present the average speedup with respect to *BestHomogeneous*. For each scale, we also display the system bandwidth utilization averaged across both homogeneous executions (*HotOnly* and *ColdOnly*).

Figure 97: Average performance of *HotTiles* and the different heuristics for different SPADE-Sextans system scales.

We observe that, for all scales, *HotTiles* outperforms the best of its heuristics. This is because, for a given scale, *HotTiles* chooses different heuristics for different matrices. We see that, for larger scales, where one worker type is typically sufficient to saturate most of the system bandwidth, the *Serial* heuristics perform better than the *Parallel* ones, since they avoid the merging cost. In addition, focusing on the *Parallel* heuristics, we see that,

in smaller scales, due to smaller bandwidth pressure, *MinTime Parallel* performs better, while for larger scales, *MinByte Parallel* performs better. Overall, the four heuristics act in a complementary manner, allowing *HotTiles* to produce a high-quality partitioning under different system scenarios. On average across all system scales, *HotTiles* provides speedups of 16.8x, 2.0x, 2.2x, and 1.3x over *HotOnly*, *ColdOnly*, *IUnaware*, and *BestHomogeneous*, respectively.

To provide further insight, Table 40 displays different utilization statistics for two different scales of the SPADE-Sextans architecture. All the statistics are geometric values across the 10 matrices. The table displays: the main memory bandwidth utilization; the number of cache lines accessed from memory normalized to the number of nonzeros of each matrix; and the utilization of the computational units of the cold and hot workers (measured in GFLOP/s) for the time period when each worker type is not idle.

Table 40: Architecture utilization statistics for SPADE-Sextans (geometric mean).

System Scale = 1	HotOnly	ColdOnly	IUnaware	HotTiles
Bandwidth Util. (GB/s)	27.96	49.68	49.04	67.41
Cache Lines Acc. from Mem./Nonzero	6.78	1.59	2.27	1.47
SPADE GFLOP/s	0.00	48.72	46.49	43.52
Sextans GFLOP/s	6.44	0.00	4.94	51.14

System Scale = 4	HotOnly	ColdOnly	IUnaware	HotTiles
Bandwidth Util. (GB/s)	82.61	132.28	127.03	124.68
Cache Lines Acc. from Mem./Nonzero	3.13	1.60	1.99	1.02
SPADE GFLOP/s	0.00	129.58	102.50	85.63
Sextans GFLOP/s	41.18	0.00	25.47	228.37

Consider first system scale 1. With *HotTiles*, the bandwidth utilization is increased. This is partly because, with *HotTiles*, both hot and cold workers are actively accessing the memory subsystem in parallel. Note that, although this is also the case with *IUnaware*, this baseline fails to increase the bandwidth utilization due to its unsophisticated IMH-unaware matrix partitioning. In addition, with *HotTiles*, redundant main memory accesses are reduced by an effective mapping of tiles to worker types (Figure 88). Also, in

HotTiles, the utilization of the SIMD units of the SPADE workers slightly drops, since they are mainly assigned the sparser, less arithmetic intense cold tiles. However, this drop is compensated by an 8x increase in the utilization of the Sextans workers.

Most of the *HotTiles* results for system scale 4 are similar. However, (1) the geomean bandwidth utilization of *HotTiles* is slightly lower than in the *ColdOnly* and *IUnaware* baselines and (2) the decrease in redundant main memory accesses is more significant. This is because, as shown in Figure 97, for this system scale, the Serial and the MinByte heuristics are preferred. Given the already high bandwidth pressure caused by the larger number of workers at this scale, *HotTiles* smartly trades-off a marginally lower bandwidth utilization for a significant decrease in the memory accesses—reducing overall execution time.

Next, we compare the performance of heterogeneous execution with *HotTiles* against homogeneous architectures that have double the number of hot or cold workers (Figure 98). We use the SPADE-Sextans system scale 4 as our heterogeneous architecture (*HotTiles₄*). For the homogeneous architectures, we use only the hot or only the cold workers from system scale 8 (*HotOnly8* and *ColdOnly8*). *HotTiles₄* provides an average speedup of 2.9x and 1.6x against *HotOnly8* and *ColdOnly8*, respectively. This reveals that a heterogeneous architecture with both worker types is more effective than a homogeneous architecture with twice the number of workers of one type.

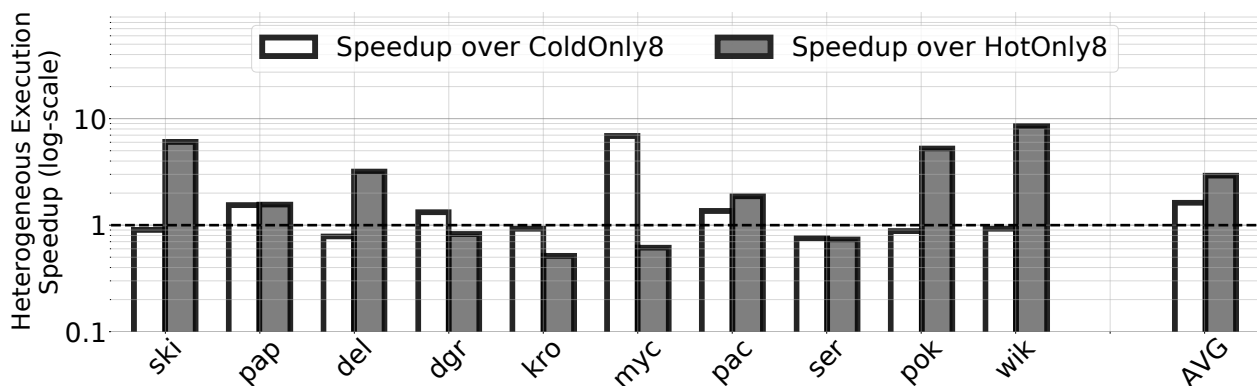


Figure 98: Speedup of *HotTiles* with system scale 4 over homogeneous execution with system scale 8.

As discussed in Section 8.7, for the SPADE-Sextans+PCIe architecture, we test gSpMM variants with higher arithmetic intensity (AI). As AI increases, more SIMD operations are required per nonzero. As the AI increases, a SPADE PE requires more cycles to execute all the arithmetic operations for one nonzero. However, as mentioned in Section 8.7, we assume an enhanced Sextans that increases its computational power proportionally to the number of arithmetic operations per nonzero to keep up with the increased AI. Since we consider a system scale of 4, the enhanced Sextans can now process 20 nonzeros per cycle irrespective of the AI.

Figure 99 displays the speedup of *HotTiles* over *HotOnly* and *ColdOnly* as the number of SIMD operations per nonzero (and thus the AI) increases. We also show the percentage of nonzeros that are assigned to the hot workers. We observe that, at low AIs, most of the nonzeros are assigned to the cold workers, since they provide enough computational throughput to accommodate the low arithmetic intensity. At these AIs, the speedup of *HotTiles* against *HotOnly* execution is high, since the low-bandwidth PCIe bus makes data transfer a significant bottleneck, while the speedup against *ColdOnly* is low, since most of the nonzeros are assigned to the cold workers anyway. As the AI increases, the situation is reversed, as computational time dominates over data transfers. In all cases, *HotTiles* offers clear benefits. On average across all AIs, *HotTiles* provides average speedups of 11.9x, 3.7x, and 2.5x over *HotOnly*, *ColdOnly*, and *BestHomogeneous*, respectively.

Finally, since most of the matrices of Table 38 favor the cold workers, we evaluated *HotTiles* on an additional set of matrices from SuiteSparse. We selected five denser matrices with a similar number of nonzeros and different application domains than the original ten. They are shown in Table 41. Figure 100 illustrates the effectiveness of *HotTiles* in this new matrix set for SPADE-Sextans with system scale 1 and system scale 4. From the figure, we see that *HotTiles* is again faster than the other architectures. The average speedups of *HotTiles* across both system scales are 1.5x, 3.8x, and 1.4x over *HotOnly*, *ColdOnly*, and *IUnaware*, respectively. It can be shown that the speedup is 1.5x

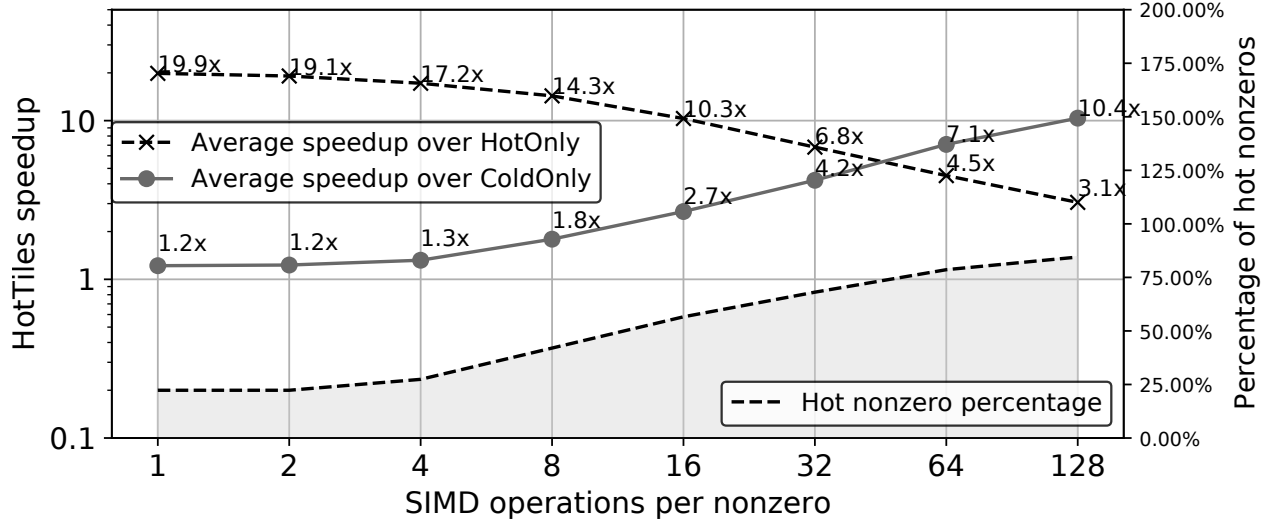


Figure 99: Performance of *HotTiles* for different gSpMM arithmetic intensities for the SPADE-Sextans+PCIe architecture.

over *BestHomogeneous*.

Table 41: Additional set of higher-density sparse matrices.

Benchmark	Short	Domain	Rows (Mill)	NNZ (Mill)	Density
gearbox	gea	Aerospace engineering	0.15	9	$4 * 10^{-4}$
mouse_gene	mou	Molecular biology	0.05	29	$1 * 10^{-2}$
nd24k	nd2	2D/3D problem	0.07	29	$1 * 10^{-2}$
RM07R	rm0	Comput. dynamics	0.38	37	$3 * 10^{-4}$
Si41Ge41H72	si4	Quantum chemistry	0.19	15	$4 * 10^{-4}$

8.8.2 Architecture Exploration with *HotTiles*

In this subsection, we use the performance prediction capabilities of *HotTiles* to explore different heterogeneous architecture alternatives. We focus mostly on the SPADE-Sextans architecture. We examine heterogeneous architectures that have more workers of one type at the expense of fewer workers of the other type. Specifically, we start with our scale 4 SPADE-Sextans architecture in Table 37, which we call 4-4. Then we consider "skewed" SPADE-Sextans architectures, such as 0-8, which includes no cold workers and hot workers with the same parameters as a scale 8 SPADE-Sextans architecture. We focus on skewed architectures where the sum of cold and hot scales equals to 8, namely 0-8, 1-7, 2-6, 3-5,

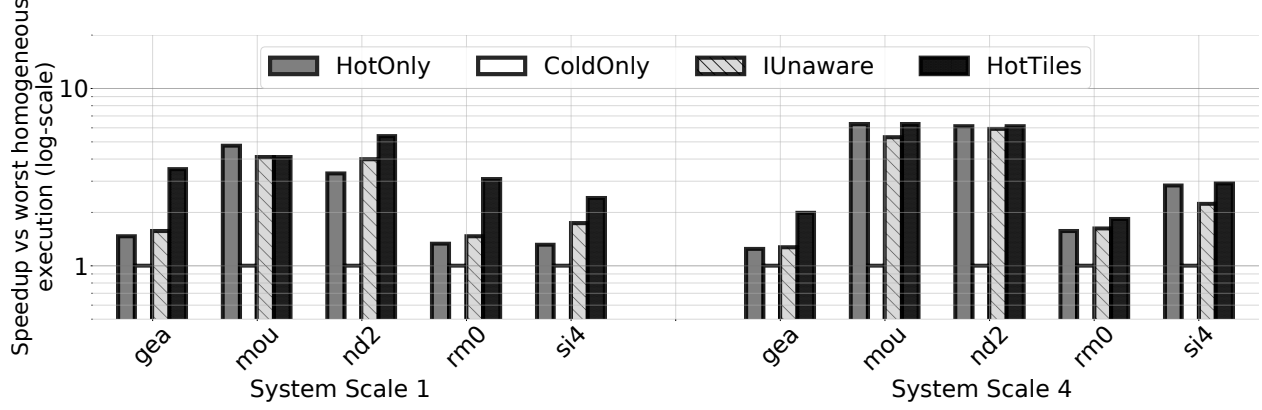


Figure 100: Comparison of homogeneous and heterogeneous execution of SPADE-Sextans for higher-density sparse matrices.

4-4, 5-3, 6-2, 7-1, and 8-0. We call these architectures *iso-scale*.

We consider two different scenarios. In the first one, we assume that the architecture is not reconfigurable, and want to find which of the 9 iso-scale architectures delivers the highest average performance for our benchmarks. In the second scenario, we assume that the architecture is reconfigurable, and want to find which of the 9 possible configurations delivers the highest performance for each individual benchmark. The first scenario corresponds to an ASIC accelerator, while the second one corresponds, e.g., to an FPGA-based one. In both cases, we use the execution time predictions made by *HotTiles*. For simplicity, we ignore any differences in area or power among the 9 iso-scale architectures. A future, more detailed analysis can compare performance per unit area or unit power.

We focus first on the first scenario. Figure 101 shows, for each iso-scale architecture, the performance predicted by *HotTiles* and the actual performance. The performance is given as speedup over the base SPADE-Sextans architecture (i.e., 4-4) and is the average across all the benchmarks of Table 38. We observe that, although the predicted performance is slightly higher than the actual one for the architectures that are dominated by hot workers, the trends of predicted and actual performance are the same. In the 7-1 and 8-0 architectures, which are dominated by cold workers, the predicted performance is

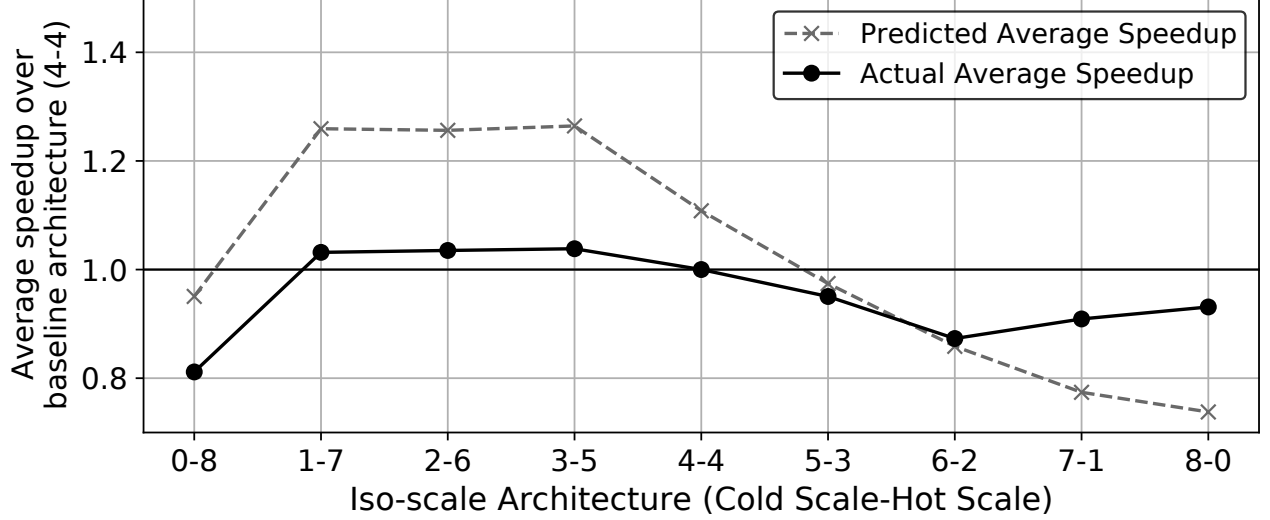


Figure 101: Predicted and actual average performance of different iso-scale heterogeneous architectures.

slightly lower than the actual one. This is because, as explained before, *HotTiles* ignores reuse in caches. Overall, the architecture predicted to perform the best (3-5) is also the one that performs the best.

We now focus on the second scenario, where the heterogeneous architecture is reconfigurable on a per-matrix basis. Table 42 displays, for each matrix, the predicted best architecture, its speedup with respect to 4-4, the actual best architecture, its speedup with respect to 4-4, and whether the prediction is correct. We see that *HotTiles* selects the best architecture out of the nine iso-scale alternatives in 50% of the matrices. *HotTiles* predictions have some bias towards using more hot workers than needed. Again, this is due to ignoring caching effects and therefore under-predicting the true capabilities of cold workers. Despite this limitation, we observe that reconfiguring the architecture based on the predictions delivers a 1.23x speedup over the baseline 4-4 architecture. This is an encouraging result, considering that the oracle speedup is 1.33x.

Finally, Figure 102 shows the error in the execution time predicted by *HotTiles* compared to the actual execution time for SPADE-Sextans and PIUMA. The figure also includes the error in the predictions for the homogeneous executions (*HotOnly* and *ColdOnly*), although we do not directly utilize these predictions. For both SPADE-Sextans

Table 42: Predicted and actual best architecture per matrix.

Matrix	Pred. Best Arch.	Speedup of Pred. Best	Actual Best Arch.	Speedup of Actual Best	Correct Pred.?
ski	3-5	0.95	8-0	1.11	N
pap	1-7	0.78	4-4	1.00	N
del	3-5	1.01	8-0	1.28	N
dgr	1-7	1.59	1-7	1.59	Y
kro	0-8	1.93	0-8	1.93	Y
myc	0-8	1.63	0-8	1.63	Y
pac	2-6	1.16	2-6	1.16	Y
ser	0-8	1.36	0-8	1.36	Y
pok	3-5	0.89	8-0	1.14	N
wik	3-5	0.96	8-0	1.08	N
AVG		1.23		1.33	50%

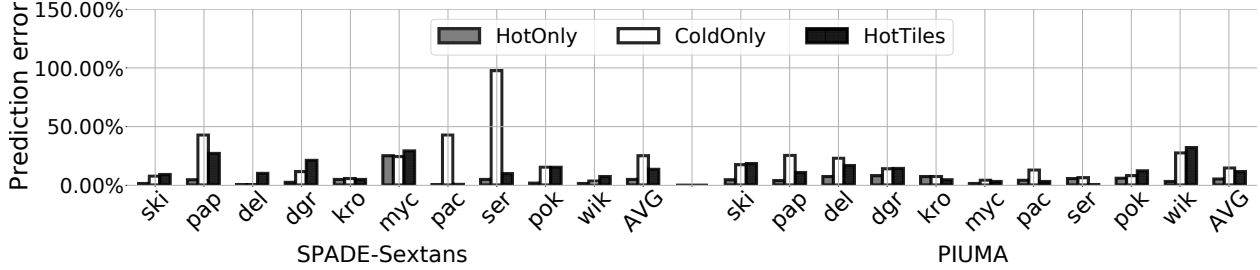


Figure 102: Error in the predicted execution time of different homogeneous and heterogeneous executions.

and PIUMA, the average error between the predicted and actual execution time is relatively low, at 4.8%, 19.6%, and 12.4% for *HotOnly*, *ColdOnly*, and *HotTiles*, respectively. The highest error occurs for certain matrices (pap, myc, pac, ser) in the *ColdOnly* execution. Upon further investigation, we found that these matrices display significant *Din* reuse through caches. Since our model ignores caching effects, the predicted *ColdOnly* execution time is higher than the actual one. This prediction error is more significant in SPADE-Sextans because the caches of the SPADE PEs are larger than the caches of the PIUMA MTPs. We believe that extending our modeling methodology to account for caching effects can further enhance the effectiveness of *HotTiles* predictions.

8.8.3 Preprocessing Cost

HotTiles requires the data preprocessing steps shown in Figure 92: matrix scan, matrix partitioning, and creation of the sparse matrix formats for the two worker types. In reality, code generation for homogeneous accelerators such as SPADE or Sextans also needs the third preprocessing step, except that it is for a single worker type. For example, the input matrix may be stored in matrix market file format [270], and should be converted to the specific format that the accelerator operates on. Hence, when we quantify *HotTiles*' preprocessing overhead, in the third step, we only consider the cost of generating the matrix format for one additional worker type.

The *HotTiles* preprocessing overhead is a one-time cost only. For example, it can be incurred once during GNN training and not affect GNN inference later on, or it can be incurred once and amortized across many SpMM iterations.

Figure 103 shows the normalized execution time of the preprocessing operations on a Xeon host for the PIUMA architecture. We break it down into matrix format creation for a homogeneous accelerator (which is not *HotTiles* specific), and the rest of the preprocessing steps listed above, which we call *Hot Tiles Overhead*. On average, *Hot Tiles Overhead* is 73% of the total preprocessing overhead. This means that *HotTiles* has about four times the preprocessing overhead of a homogeneous accelerator. This is a small overhead, which is amortized over many SpMM iterations. We obtain similar numbers for SPADE-Sextans and SPADE-Sextans+PCIe. Finally, note that, in practice, this overhead is dwarfed by the overhead of reading the matrix from disk. If we include reading the matrix from disk in the preprocessing overhead, it can be shown that *Hot Tiles Overhead* increases the preprocessing overhead by only 6% on average.

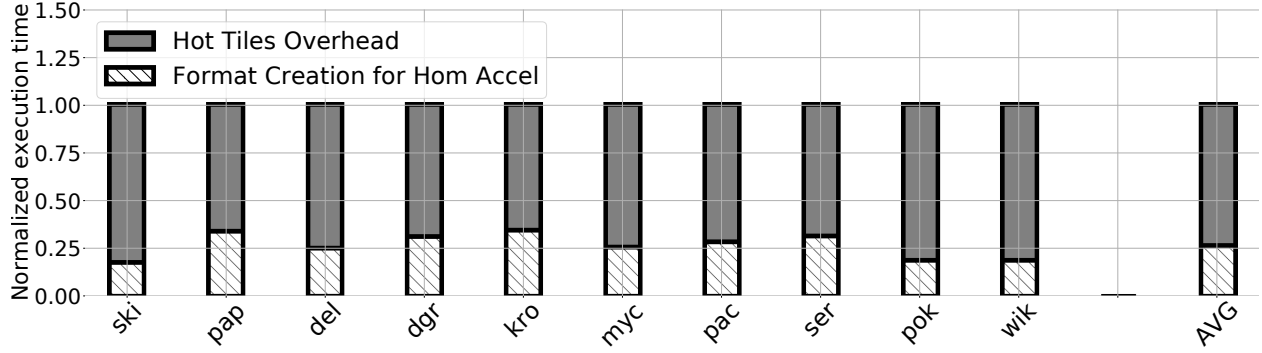


Figure 103: Normalized execution time of the preprocessing operations on a Xeon host for the PIUMA architecture.

8.9 Related Work

A. SpMM accelerators: In recent years, many domain-specific systems and accelerators that support SpMM have been proposed [79], [80], [81], [91], [119], [166], [202], [280], [285]. Commonly, they employ homogeneous processing elements, missing opportunities to exploit IMH. For example, Adiletta et al. [91] use the PIUMA MTPs and DMA engines to accelerate SpMM in the context of Graph Convolutional Networks. However, they use PIUMA in a homogeneous mode by disabling the STPs. AESPA [135] is an architecture that consists of heterogeneous subaccelerators that target different levels of sparsity and compression formats. However, that work assumes uniform random sparsity. Therefore, it does not map matrix regions to the subaccelerators that suit them the most and thus fails to exploit IMH. Our *IUnaware* baseline is inspired by that work.

B. Sparse accelerator modeling: Some works present abstractions and models for sparse tensor algebra [275], [355], [356]. In this work, we use a simple analytical model that can accurately predict the performance of different heterogeneous PEs at a tile level. Importantly, our performance estimation method increases the preprocessing time by only a small amount. Extending or incorporating alternative performance models is a valuable research direction.

C. Work partitioning and scheduling for heterogeneous systems: Some works

investigate work partitioning and scheduling for heterogeneous systems [349], [357], [358], [359], [360], [361], [362], [363], [364]. These works mainly focus on partitioning dense kernels among different workers or mapping different functions to different workers. In contrast, our work focuses on partitioning sparse kernels for heterogeneous accelerators—a topic that has barely been explored. We show that the partitioning problem in such scenarios is non-trivial and requires taking into account both IMH and the low-level microarchitecture details of the heterogeneous workers.

D. SpMM for general-purpose systems and tiling: Prior art incorporates tiling and other optimizations in order to optimize SpMM on CPUs and/or GPUs [87], [88], [92], [365], [366], [367], [368]. In particular, ASpT [87] partitions the matrix into denser and sparser regions, which are then however assigned to the same homogeneous entities (e.g., CPU cores). Some of these works incorporate reordering [139], [369], [370] to transform the sparse matrix into an equivalent more "favorable" form. As shown by Arai et al. [139], the reordered sparse matrix can have more well-formed dense and sparse regions, leading to more efficient execution. Reordering could also increase the effectiveness of *HotTiles*.

NVIDIA’s cuSPARSE [100] supports Block-SpMM [371], an SpMM variant that targets sparse matrices that are partitioned into high-density and completely empty tiles (block-level sparsity). Optimizations for block-level sparsity are also proposed by Demmel et al. [372]. On the contrary, our method does not require the sparse tiles to be completely empty. In fact, we completely eliminate empty tiles during preprocessing.

8.10 Further Directions

We now discuss two additional directions for extending this work. The first one is the application of *HotTiles* to other heterogeneous architectures and sparse kernels. An interesting additional architecture is a heterogeneous system consisting of CPUs and on-chip accelerators such as the Intel Data Streaming Accelerator (DSA) [18], [149]. In addition, *HotTiles* is applicable to SpMV and SDDMM [99], which exhibit access patterns

similar to SpMM. Further, we believe that, with some modifications, our methodology can be extended to more sparse kernels such as SpGEMM [373].

A second potential direction for extending HotTiles is to further explore the partitioning space and enhance the modeling capabilities. We believe that making the HotTiles performance model account for reuse through caches and removing the maximum reuse assumption (Section 8.4.3) can further enhance the model’s accuracy. In addition, accounting for reuse through shared levels of fast local memory such as lower level caches can make *HotTiles* applicable to more heterogeneous architectures. Finally, smart tile sizing in cases where one or both of the sparse tile dimensions are free, and heterogeneity-aware sparse matrix reordering approaches can augment the *HotTiles* partitioning space, leading to further performance improvements.

8.11 Conclusion

In many sparse matrices, nonzeros form dense and sparse regions—a property that we call intra-matrix heterogeneity (IMH). To leverage IMH, we utilize heterogeneous SpMM accelerator architectures that include different types of PEs. To optimize the performance of such architectures, we develop a modeling framework that predicts the performance of different PE types, and a heuristic that partitions sparse matrices among heterogeneous PEs. We call our modeling and partitioning method *HotTiles*. To evaluate *HotTiles*, we simulate three heterogeneous architectures with compute-intensive PEs (*Hot Workers*) and memory-latency tolerant PEs (*Cold Workers*). Heterogeneous execution with *HotTiles* outperforms homogeneous execution using only hot or only cold workers by 9.2-16.8 $\times$ and 1.4-3.7 $\times$, respectively. In addition, *HotTiles* outperforms an IMH-unaware heterogeneous execution strategy by 1.4-2.2 $\times$.

Chapter 9 Hardware Agent for General Data-Driven Runtime Optimization

9.1 Introduction

Over recent years, Machine Learning (ML) has received widespread attention in the architecture community due to its ability to efficiently model complex patterns, optimize system operation, and adapt to dynamic workloads. ML has found a myriad of applications in microarchitecture, ranging from predictors [374], [375], [376], prefetchers [377], [378], and cache replacement policies [379], [380] to resource management and control [381], [382]. Online Reinforcement Learning (RL) [383], [384] is a subclass of ML algorithms that is particularly well-suited for microarchitecture decision-making problems. In online RL, agents interact with their environment without having to pre-train with an offline dataset. This enables better adaptability and generalization to previously unseen environment configurations.

Hardware agents that employ RL [385], [386], [387], [388], [389], [390], [391], [392] typically decompose the environment into a complex set of states, where the RL agent tries to discover the best actions that maximize its reward. Such actions cause transitions between states. Although this approach is effective, it has high complexity, as it introduces the need to track action values and transition probabilities for many different states. In the resource-constrained environment of processor microarchitecture, this results in significant storage overhead. In addition, most of these RL agents are engineered for a specific problem and are not reusable. Designing, validating, and integrating a new RL hardware agent in a processor every time a new potential use case emerges is very costly. This challenge is likely to become even more pronounced in modern accelerator-intensive heterogeneous systems. As use cases, architectures, and optimization objectives become increasingly heterogeneous, designing problem-specific decision-making agents becomes less attractive.

To address these problems, this chapter introduces a lightweight and reusable hardware RL agent. We find that, in some microarchitecture decision-making problems, only a small

fraction of the action space is useful in a given time window. We refer to this property as *temporal homogeneity in the action space*. We show that, when this property is present, the problem can be roughly modeled using a single RL state.

Based on this insight, we explore the effectiveness of *Multi-Armed Bandit* (MAB) algorithms [393], [394], which are the RL flavor with the lowest complexity. Due to their simplicity, they are especially suitable for decision-making in highly area-constrained settings like a processor’s pipeline. Specifically, we design an RL agent that implements the *Discounted Upper Confidence Bound* (DUCB) algorithm [395] in hardware. We call our agent *Micro-Armed Bandit* or *Bandit*.

We showcase Bandit for two different decision-making problems that have sufficient temporal homogeneity in their action space to be tackled with MABs. Firstly, we use Bandit to orchestrate simple and widely adopted non-RL L2 data prefetchers—i.e., next-line, stride, and stream prefetchers. Secondly, we use it to control the instruction fetch policy in simultaneous multithreaded (SMT) processors [396]. In particular, in the second use case, we introduce a novel design space for SMT instruction fetching that customizes both the thread *fetch priority* [397] and the thread *fetch gating* [398] policies.

Our simulation-based evaluation suggests that Bandit is effective. In the data prefetching use case, for single-core experiments, Bandit outperforms non-RL prefetchers Bingo [399] and MLOP [400] by 2.6% and 2.3% (geometric mean), respectively, and attains similar performance as the state-of-the-art RL prefetcher Pythia [387]—with the dramatically lower storage requirement of only 100 bytes. In addition, Bandit attains good performance in four-core experiments and outperforms Pythia by 2.5% in bandwidth-constrained scenarios. In the SMT instruction fetch use case, Bandit outperforms the Hill Climbing method [398] by 2.2% (geometric mean).

Overall, this chapter’s contributions are:

- The concept of temporal homogeneity in the action space, which enables low overhead RL for microarchitecture decision-making.

- The adoption of Multi-Armed Bandits as an efficient and reusable learning mechanism for such problems.
- The Micro-Armed Bandit, an RL agent that implements the Discounted Upper Confidence Bound algorithm in hardware and a simulation-based evaluation of its performance for data prefetching and SMT instruction fetching.
- A new design space for SMT instruction fetching that customizes both the thread fetch priority and thread fetch gating policies.

9.2 Background

9.2.1 Online Reinforcement Learning

RL [383], [384] is a subclass of ML algorithms that targets action selection problems. Commonly, an RL agent interacts with its environment by trying different actions and receiving feedback in the form of a reward. The goal is to learn the optimal balance between *exploration* (trying new actions) and *exploitation* (selecting the best-known action) to maximize the accumulated reward in the long term. Exploration is necessary for an RL algorithm because it helps the agent better understand the dynamics of its environment. The duration of an agent’s interaction with its environment is called an RL *episode*.

In online RL, agents interact with their environment without having to pre-train with an offline dataset. As a result, online RL has several attractive characteristics [387], [401] for microarchitecture decision-making problems compared to other methods such as Supervised Machine Learning [352], [381], [382], [402], [403], [404], [405] and Formal Control [406], [407], [408], [409], [410]. In comparison with traditional ML, online RL eliminates the need for offline data collection. It provides better adaptability and generalizes to environment configurations not encountered before. As an example, consider an agent that is trained offline using the SPEC benchmarks [411], [412]. Ensuring high performance in unseen workloads stemming from graph analytics or cloud applications is non-trivial. Even if the offline dataset is large enough to include a diverse set of

benchmarks, there are no guarantees that the agent will adapt successfully to unforeseen system conditions such as power fluctuations. When compared to Formal Control, online RL does not depend on an offline static model of the system and does not require the actions to be linearly correlated with their outcomes [401]. For these reasons, online RL is an attractive solution for action selection problems in microarchitecture.

9.2.2 RL Problem Formulations

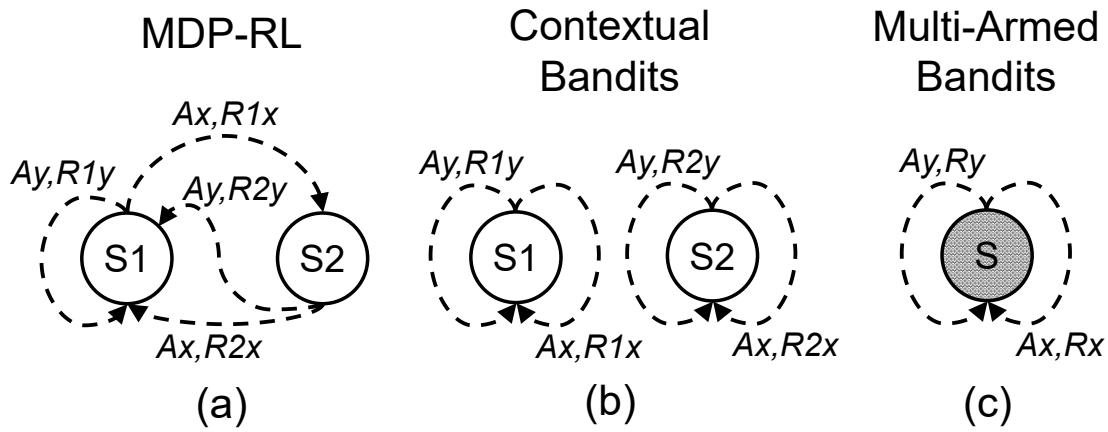


Figure 104: Examples of states and actions in different RL problem formulations. S represents states, A represents actions, and R represents rewards. The arrows illustrate state transitions.

There are multiple RL problem formulations, which differ in how they model the environment. They have different levels of complexity. The level of complexity of a formulation is directly correlated with the implementation complexity of the RL agent in hardware. In this section, we describe three alternative formulations. In addition, we identify a common property of microarchitectural problems that enables the use of the RL flavor with the lowest complexity.

(1) Markov Decision Process-based Reinforcement Learning (MDP-RL):

Modeling the environment as a Markov Decision Process (MDP) [413] is the most general and highest-complexity problem formulation. The environment is decomposed into many states and, in each state, different actions yield different rewards. The actions cause

non-deterministic transitions between states. Before deciding on an action, the agent considers its current state as well as the states it might transition to after taking that action. Figure 104(a) shows a simple MDP environment with 2 states ($S1$ and $S2$). In each state, two actions are possible (Ax and Ay). After the agent selects an action, it receives a reward (R_{ij}), which depends on both the current state and the selected action.

The high complexity of MDP-RL stems from the requirement of bookkeeping action values and transition probabilities for each distinct state of the environment. Popular algorithms in this category include Q-Learning [414] and SARSA [383], [415]. Numerous works propose hardware agents that model the environment as an MDP. For example, Ipek et al. [386] propose a SARSA-based memory controller that uses the number and characteristics of reads and writes in the transaction queue to encode the state. Cohmeleon [389] proposes a Q-Learning-based agent to orchestrate accelerator coherence in heterogeneous SoCs. The Pythia L2 prefetcher [387] is based on SARSA and can be customized to use different data flow and control flow features to encode the state.

(2) Contextual Bandits: Modeling a decision-making problem with Contextual Bandits [416], [417] abstracts away the state transition probabilities. State transitions can still occur but they are not influenced by the agent's actions. Figure 104(b) shows a Contextual Bandit environment. The agent can take actions. However, any transition between states is not a consequence of selecting an action. Instead, the system *randomly* transitions between states due to other effects. The agent observes the context associated with each state and tries to determine the best action for a given context to maximize its accumulated reward. Contextual Bandits are a lower complexity problem formulation than MDP-RL but still require the bookkeeping of action values for each state. The most notable application of Contextual Bandits in microarchitecture is the Context Prefetcher [390], which utilizes both hardware and compiler-injected software context to decide on the appropriate prefetching actions.

(3) Multi-Armed Bandits (MABs): Multi-Armed Bandits (MABs) [393], [394] collapse

the environment into a single state as displayed in Figure 104(c). The goal is to identify the single best action that yields the highest reward while minimizing time spent trying sub-optimal actions. In MAB terminology, the available actions are referred to as *arms*. MABs have significantly lower complexity than the two previous methods because there is a single state and, therefore, only a single value needs to be tracked per action. Although MABs' low complexity makes them attractive for resource-constrained settings, their application to microarchitecture has not been investigated.

MABs are less powerful than the other two RL problem formulations. However, they can still be applied to the microarchitecture domain. Consider a scenario where the same action is repeatedly optimal for a large enough time period inside an RL episode. In this case, the action is *temporally* optimal regardless of the environment's state and the whole state space can collapse into a single state. We refer to this property as *temporal homogeneity in the action space*.

Figure 105 provides examples of (fully) temporal homogeneous and temporal heterogeneous action spaces. In the figure, the whole action space consists of two actions: Ax and Ay . Figure 105(a) shows an environment with a fully temporal homogeneous action space. It shows the sequence of optimal actions over time for two different RL episodes (e.g., the execution of 2 different benchmarks). We see that, in Episode 1, action Ax is always optimal, while in Episode 2, Ay is always optimal. On the other hand, Figure 105(b) shows an environment with a temporal heterogeneous action space, where the optimal action changes rapidly with time during the same episode. Note that temporal homogeneity does not require the same action to be optimal during the whole episode. Instead, it imposes optimality requirements for a time period long enough so that the MAB agent can detect the appropriate action. As we will see in Section 9.3, high temporal homogeneity is present in some microarchitectural problems. This property enables the effective use of the simple MAB RL flavor.

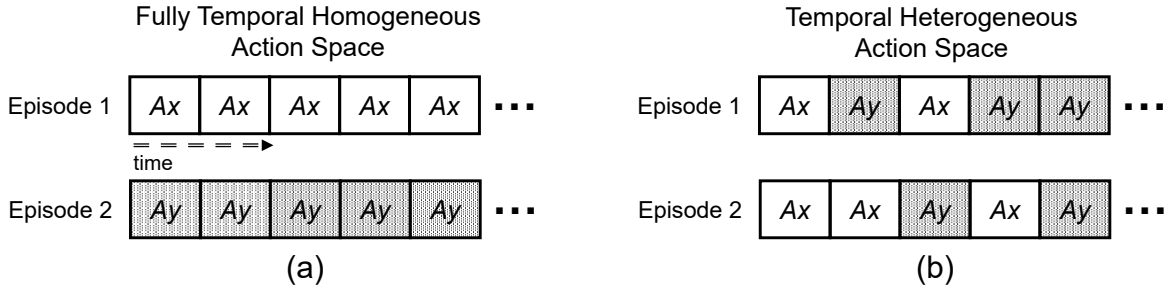


Figure 105: Examples of temporal homogeneous and heterogeneous action spaces.

9.3 Motivation

This section examines the viability of using MAB algorithms to attack two different microarchitectural action selection problems: (i) data prefetching and (ii) instruction fetching in simultaneous multithreaded (SMT) processors.

9.3.1 Temporal Homogeneity in Prefetching

In data prefetching, a hardware agent reacts to a request for a line at address X by also prefetching a set of cache lines that are likely to be accessed in the near future. Spatial prefetching algorithms typically specify the lines to be prefetched through degree (d) and offset (o) parameters. The addresses of the prefetched lines are $X + o$, $X + 2 \times o$, ..., $X + d \times o$. In this case, prefetching can be formulated as an RL decision-making problem with the action space corresponding to the selection of the prefetch degree and offset. It is well known that different degrees and offsets work best for different scenarios [253], [418], [419], [420], [421]. In this section, we examine the temporal homogeneity of the prefetching action space.

Characterizing the temporal homogeneity of the prefetching action space requires identifying the optimal prefetching action at every point in the program. Unfortunately, this is a non-trivial task. Consequently, we use as a proxy the actions taken by Pythia [387], which is a spatial MDP-RL prefetcher that shows high performance. Pythia

supports 16 different offsets and four different degrees, for a total of 64 actions. We profile the frequency of the two most popular Pythia actions in SPEC traces for a duration of 1 billion instructions. Section 9.6 presents more details about our methodology. Figure 106 displays our findings. On average, the most selected action in each application accounts for 60% of the total selections, while the second most selected accounts for 15% of the total action selections of Pythia. In other words, for a duration of 1 billion instructions, 3% of the action space accounts for 75% of the total action selections. Note that the most selected action is different in each application.

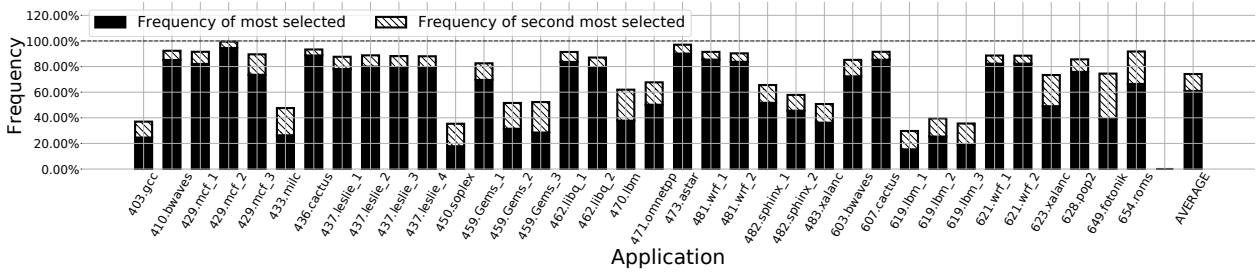


Figure 106: Frequency of the top 2 most selected actions by Pythia in SPEC applications during a trace of 1B instructions.

The data suggests that the prefetching action space has high temporal homogeneity but not a perfect one. Also recall that MABs cannot distinguish between environment states (e.g., between cache line addresses or between PCs). Hence, using a MAB agent to directly select a single best degree and offset for all the cache lines would often not work. Instead, we leverage the following observation: conventional and widely-adopted non-RL prefetchers such as the stride and stream prefetchers can already distinguish between environment states to some extent. For example, the PC-based stride prefetcher can concurrently support different offsets for different PCs. Thus, instead of using the MAB agent to directly determine the prefetch degrees and offsets, we use it as a *coordinator* of various conventional lightweight prefetchers—in a manner similar to [418]. Section 9.5 provides more details about how we employ our agent towards this goal, and Section 9.7 presents its effectiveness.

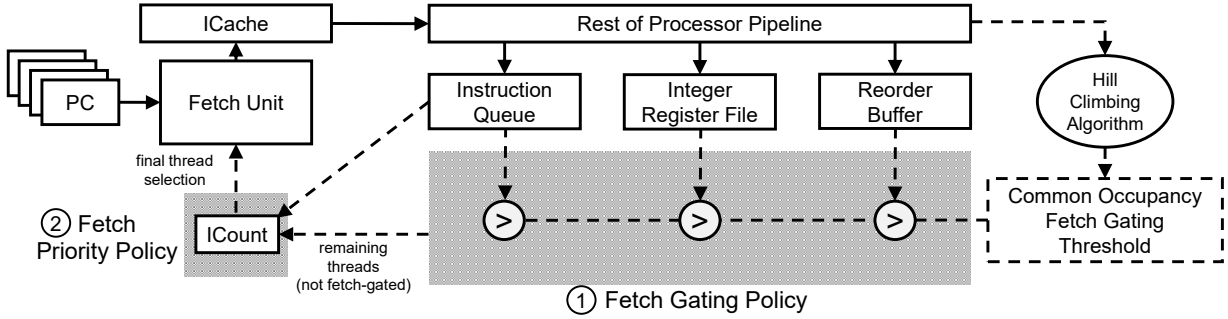


Figure 107: Pipeline structures and fetch priority & gating policy in [398]. Threads that exceed the occupancy threshold in either the Instruction Queue, Register File, or ROB are fetch-gated. The remaining threads are prioritized using ICount.

9.3.2 Policies for Fetch Priority and Gating of SMT Threads

An important factor that affects performance in SMT processors [396] is the policy that determines which thread to fetch instructions from. In [397], different *fetch priority* policies were investigated. They give priority to threads based on different microarchitectural metrics: Branch Count (BrC) prefers to fetch from threads that have fewer branch instructions in the Reorder Buffer (ROB); ICount (IC) favors threads with fewer occupied entries in the Instruction Queue (IQ); and LSQCount (LSQC) prioritizes threads with fewer entries in the Load-Store Queue (LSQ). The Round Robin (RR) policy alternates between threads in a round-robin manner without considering any microarchitectural metric.

Choi and Yeung [398] introduce an adaptive mechanism for the *fetch gating* of threads. The mechanism dynamically sets a per thread occupancy threshold for hardware structures using a Hill Climbing algorithm. If the occupancy of a thread in the IQ, the Integer Register File (IRF), or the ROB exceeds this threshold (which is the same for all the structures) the thread is fetch-gated. The best threshold is determined by trial epochs, dynamically increasing the allowed entries for one thread by δ units at the expense of the other threads. An interesting property of the optimal thresholds is that they are mostly *temporally stable* as shown in [398]. This is consistent with what we observed for prefetching in Section 9.3.1. Finally, Choi and Yeung determine the priority of the non

fetch-gated threads using ICount.

In our work, we call the combination of the thread fetch gating policy and the thread priority policy the fetch *Priority & Gating (PG)* policy. Figure 107 illustrates the pipeline structures and the fetch PG policy used by Choi and Yeung. In the rest of this chapter, we refer to the fetch PG policy used by Choi and Yeung as the *Choi* policy.

Because the fetch gating policy uses thresholds, it does not prevent threads with high ILP from utilizing a large number of entries in the shared hardware structures. A thread with high ILP can be allowed to use most of the hardware structure entries, while one with low ILP may be restricted to fewer entries—if this entry distribution maximizes performance.

The fetch gating and the fetch priority policies do not accomplish the same goal; instead, they complement one another. Consider the times when none of the threads is fetch-gated. One still needs to select which thread to fetch from. In addition, in contrast to fetch gating, not all the fetch priority policies are based on measuring the stress that different threads place on hardware structures. Specifically, the Round Robin (RR) and Branch Count (BrC) policies take different approaches.

9.3.3 Extending the Choi Algorithm

Although the Choi policy is very effective, it has limited adaptability. First, it always uses ICount for the non fetch-gated threads. Second, its fetch gating policy is fixed: it fetch-gates a thread if its occupancy of the IQ, IRF, or ROB exceeds a given threshold. As a result, the fetch gating policy is unaware of the occupancy of other pipeline structures such as the LSQ. In addition, in some workloads, threads may use resources asymmetrically, and it may be best to disregard the occupancy of certain hardware structures. For example, assume a 2-threaded scenario where we limit the occupancy of all three structures to 50%. However, one thread wants to use more entries from the ROB and does not need to use as many entries from the IQ, while the other thread has the opposite

behavior. Under the Choi policy, both threads are conservatively fetch-gated, while the pipeline could support more instructions. In this case, one could exclude monitoring the occupancy of the ROB or the IQ (or both) from the fetch gating policy and attain higher performance. As we show next, determining which structures to monitor is an interesting use case for RL-based decision-making mechanisms.

To mitigate the aforementioned limitations, we consider many fetch PG policies beyond the Choi one. Specifically, we consider the four fetch priority policies of Section 9.3.2 (BrC, IC, LSQC, and RR), and fetch gating policies based on the occupancies of zero or more of the following hardware structures: IQ, LSQ, ROB, and IRF. We represent these fetch PG policies as $X_b_3b_2b_1b_0$, where X is the fetch priority policy and can be {BrC, IC, LSQC, RR}, and $b_3b_2b_1b_0$ are bits that denote whether the fetch gating policy monitors the occupancy of the IQ, LSQ, ROB, or IRF, respectively.

Table 43 shows some examples of these fetch PG policies. For example, IC_0000 uses ICount as the fetch priority policy and does not consider the occupancy of any structure when fetch gating. Thus, it does not perform fetch gating; it is the original ICount policy of Tullsen et al. [397]. IC_1110 uses ICount and fetch-gates a thread whose utilization of the IQ, LSQ, or ROB is above the threshold. IC_1011 corresponds to the Choi policy.

Table 43: Examples of different fetch PG policies.

Policy Mnemonic	Fetch Priority	IQ*	LSQ*	ROB*	IRF*
IC_0000	ICount	no	no	no	no
BrC_1000	Branch Count	yes	no	no	no
IC_1110	ICount	yes	yes	yes	no
IC_1111	ICount	yes	yes	yes	yes
LSQC_1111	LSQ Count	yes	yes	yes	yes
RR_1111	Round Robin	yes	yes	yes	yes

*Columns IQ, LSQ, ROB, and IRF indicate whether fetch is gated when the occupancy of the corresponding structure exceeds a threshold.

In this way, we have a total of 64 ($4 * 2^4$) different fetch PG policies. We tested all of them for a set of 43 2-threaded workloads from SPEC17 [412] (refer to Section 9.6 for our methodology). For each of the workloads, Figure 108 shows the IPC change relative to the

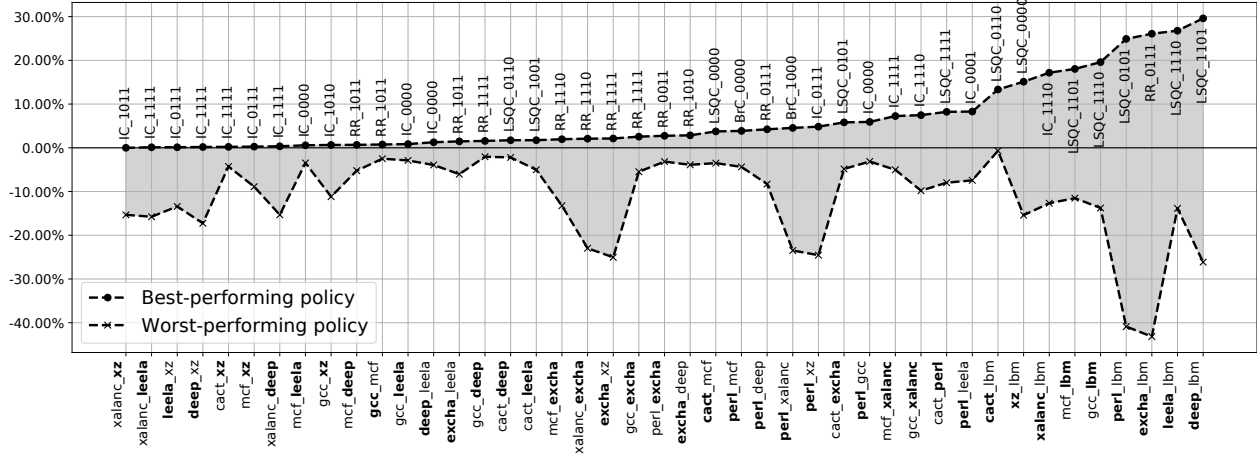


Figure 108: IPC change relative to the Choi policy of the best- and the worst-performing fetch PG policies among the ones we consider for different multithreaded SPEC17 mixes. The policy that performs the best for each mix is added as a label. For the best policy, the thread that committed the most instructions in the mix appears in bold.

Choi policy of the best- and the worst-performing fetch PG policies among the ones we consider. The policy that performs the best for each mix is added as a label. For the best policy, the thread that committed the most instructions in the mix appears in bold.

From the figure, we see that different fetch PG policies work best in different application mixes. In addition, there is significant IPC variation between the best- and the worst-performing policies in an application mix. Selecting a bad policy can decrease performance by more than 40% compared to IC_1011.

Application mixes consisting of lbm attain large speed-ups. In this case, policies that use LSQC as the fetch priority policy or consider LSQ occupancy when fetch gating (i.e., x1xx) offer substantial speedups (13–30%) over IC_1011. Note that these policies still allow the thread that benefits the most from memory-level parallelism to get more LSQ entries, if this distribution of entries maximizes the average performance. As discussed in [422], in a fully dynamically shared pipeline (with no LSQ-aware fetch gating mechanism) lbm frequently tends to aggressively consume all of the SQ entries, leaving the other thread with close to none SQ entries. In such cases, an LSQ-aware gating or priority mechanism prevents the exhaustion of the SQ entries by a single thread. Such policies

result in a higher average IPC than the LSQ-unaware IC_1011.

Overall, the temporal stability of the threshold of the Hill Climbing algorithm in combination with the heterogeneity in the optimal fetch PG policy across different applications is fertile ground for MAB-based decision-making mechanisms.

9.4 Multi-Armed Bandit Algorithms for Microarchitecture

In this section, we focus on different Multi-Armed Bandit (MAB) algorithms and discuss their implications for decision-making problems in microarchitecture.

Table 44: MAB algorithm variables.

Variable	Definition
arm :	Action available to the MAB agent
$bandit\ step$:	Time duration that the agent is idle waiting to observe the reward from its previous arm selection
r_i :	Average reward that previous selections of arm i have yielded
n_i :	Number of times that arm i has been selected in the past
n_{total} :	Total number of times that all arms have been selected in the past
r_{step} :	Reward received at the end of a bandit step

9.4.1 General Template for MAB Algorithms

The variables used in a MAB algorithm are shown in Table 44. An *arm* refers to a specific action available to the MAB agent, while the *bandit step* is defined as the time duration for which the agent is idle waiting to observe the outcome of its previous arm selection. For every arm i , two variables are needed: the average reward r_i that previous selections of this arm have yielded; and the number of times n_i that this arm has been selected in the past. Finally, n_{total} refers to the total number of selections of all arms and r_{step} is the reward received at the end of a bandit step.

Algorithm 4 provides a general template for MAB algorithms. For compactness, we use r_{arm} and n_{arm} instead of $r_{i=arm}$ and $n_{i=arm}$, respectively, in our notation. The algorithm takes the number of arms available M . It begins with an initial round robin phase, during

Algorithm 4 General template for MAB algorithms

```
1: Inputs:  $M$  arms
2: Variables:  $r_i$  : average reward of arm  $i$ ,
3:  $n_i$  : number of times that arm  $i$  has been selected
4:
5: Initial Round Robin Phase
6:  $n_{total} \leftarrow 0$ 
7: for  $t = 1$  to  $M$  do
8:    $arm \leftarrow t$ 
9:    $n_{arm} \leftarrow 1$ 
10:   $n_{total} \leftarrow n_{total} + 1$ 
11:  // receive reward at the end of the bandit step
12:   $r_{arm} \leftarrow r_{step}$ 
13: end for
14:
15: Main Loop
16: for  $t = M + 1$  to  $\infty$  do
17:    $arm \leftarrow nextArm()$ 
18:    $updSels(arm)$ 
19:   // receive reward at the end of the bandit step
20:    $r_{arm} \leftarrow updRew(r_{step})$ 
21: end for
22:
```

which all arms are tried once. r_{arm} is set to the r_{step} received during that arm's step, and n_{arm} is set to 1. Then, the main loop of the algorithm begins, which lasts for as long as the agent keeps interacting with the environment. The main loop consists of three basic functions, which depend on the specific MAB algorithm used. Those are $nextArm()$, which selects the next arm to be tried; $updSels(arm)$, which updates the number of selections n_{arm} for the currently selected arm and potentially other arms; and $updRew(r_{step})$, which updates the reward r_{arm} for the currently selected arm after the bandit step is over and the r_{step} has been collected.

9.4.2 Three MAB Algorithms

The implementations of these three functions define how a MAB algorithm handles the exploration-exploitation tradeoff. We now discuss the implementations of these functions

Table 45: Multi-armed bandit algorithm variants.

	ϵ -Greedy	Upper Confidence Bound	Disc. Upper Confidence Bound
<i>nextArm</i>	$\begin{cases} \text{arg max}\{r_i\} & \text{with prob. } 1 - \epsilon \\ \text{rand. arm} & \text{with prob. } \epsilon \end{cases}$	$\begin{cases} \text{arg max}\{r_i + c\sqrt{\frac{\ln(n_{\text{total}})}{n_i}}\} \end{cases}$	$\begin{cases} \text{arg max}\{r_i + c\sqrt{\frac{\ln(n_{\text{total}})}{n_i}}\} \end{cases}$
<i>updSels</i>	$\begin{aligned} n_{\text{arm}} &\leftarrow n_{\text{arm}} + 1 \\ n_{\text{total}} &\leftarrow n_{\text{total}} + 1 \end{aligned}$	$\begin{aligned} n_{\text{arm}} &\leftarrow n_{\text{arm}} + 1 \\ n_{\text{total}} &\leftarrow n_{\text{total}} + 1 \end{aligned}$	$\begin{aligned} n_i &\leftarrow \gamma \times n_i, \forall i \in [1, M] \\ n_{\text{arm}} &\leftarrow n_{\text{arm}} + 1 \\ n_{\text{total}} &\leftarrow \gamma \times n_{\text{total}} + 1 \end{aligned}$
<i>updRew</i>	$\frac{r_{\text{arm}} * (n_{\text{arm}} - 1) + r_{\text{step}}}{n_{\text{arm}}}$	$\frac{r_{\text{arm}} * (n_{\text{arm}} - 1) + r_{\text{step}}}{n_{\text{arm}}}$	$\frac{r_{\text{arm}} * (n_{\text{arm}} - 1) + r_{\text{step}}}{n_{\text{arm}}}$

for three MAB algorithms: ϵ -Greedy, Upper Confidence Bound (UCB), and Discounted Upper Confidence Bound (DUCB). Their concise implementation is shown in Table 45.

(a) ϵ -Greedy: ϵ -Greedy [423] is the simplest of the algorithms discussed. As shown in Table 45, the *nextArm* function selects with probability $1 - \epsilon$ the arm that has the highest average reward so far ($\text{arg max}\{r_i\}$), and with probability ϵ a random arm. ϵ is an algorithm hyperparameter that is directly related to the degree of exploration: large values of ϵ mean that the agent is eager to explore more and exploit less. The *updSels* and *updRew* functions are also simple. In *updSels*, n_{arm} and n_{total} are incremented. In *updRew*, the r_{step} is added to the running average. ϵ -Greedy is used in prior MDP-RL works [387], [388], [390] as an exploration technique (not as a standalone algorithm).

(b) Upper Confidence Bound (UCB): Two shortcomings of ϵ -Greedy is that the exploration picks random arms and is non-decaying. The problem with randomized exploration is that very bad arms and nearly-optimal ones have similar probabilities to be explored. Consider an example from the microarchitecture domain with two different actions. In the past, one action has resulted in low IPCs and the other in high IPCs. ϵ -Greedy will keep on exploring the two with equal probability. In addition, non-decaying exploration means that the agent will keep on exploring at the same rate as time passes, even if it has acquired enough certainty about the quality of each arm.

The Upper Confidence Bound (UCB) algorithm [424] tries to solve both problems. While it uses the same *updSels* and *updRew* functions as ϵ -Greedy, it uses a more

sophisticated exploration method in the *nextArm* function, as shown in Table 45. The next arm to be selected is the one with the highest *arm potential*. The potential is the sum of the arm’s average reward so far r_i plus a term added as a "bonus" given by $c\sqrt{\frac{\ln(n_{\text{total}})}{n_i}}$. This term is the exploration factor. In this expression, c is an algorithm hyperparameter, which we call *exploration constant*. It controls the degree of exploration. If an arm has not been selected much in the past, n_i is small compared to $\ln(n_{\text{total}})$, and that arm’s exploration factor is large. That arm will be favored for selection. The exception is if the arm’s r_i is unacceptably low, in which case preference will be given to other arms. As time passes and different arms are tried, exploration decays since $\frac{\ln(n)}{n}$ tends to 0 as n increases.

(c) Discounted Upper Confidence Bound (DUCB): The Discounted Upper Confidence Bound (DUCB) [395] is an extension to the UCB algorithm that is suitable for non-stationary (i.e., highly varying) environments. These environments are common in microarchitectural problems since the behavior of benchmarks changes with time. In these cases, we want to forget the past behavior that we observed.

As shown in Table 45, DUCB shares the *nextArm* and *updRew* functions with UCB, but employs a different *updSels* function. In *updSels*, all selections n_i are discounted by a γ constant, which is a hyperparameter that is less than 1. γ acts as a forgetting factor. For the currently selected arm, n_{arm} is first discounted by γ and then incremented by 1. As time progresses, since the n_i for rarely selected arms is discounted, the exploration factor for such arms grows, and the arms are possibly retried. This enables the capture of varying behaviors of an arm. Since DUCB is more dynamic than UCB, it is less likely to get trapped in suboptimal arms than UCB.

9.4.3 Modifications for Microarchitecture Environments

During our evaluation, we found two algorithm modifications that result in higher performance in microarchitecture environments. The first one is related to the fact that, in all of our use cases, we use IPC as the MAB reward, which can vary drastically across

different benchmarks. At the same time, we use a common exploration constant c for all benchmarks. Recall that the potential of an arm in UCB and DUCB is given by $r_i + c\sqrt{\frac{\ln(n_{\text{total}})}{n_i}}$. This leads to the following unwanted effect: for low-IPC benchmarks, r_i is small and the effect of the exploration factor $c\sqrt{\frac{\ln(n_{\text{total}})}{n_i}}$ is relatively more prominent than in high-IPC benchmarks. As a result, MAB algorithms end up exploring a lot more in low-IPC benchmarks.

To mitigate this unwanted effect, we modify the MAB algorithm as follows. First, after the initial round robin phase of Algorithm 4 is completed, we calculate the average initial reward across all arms (r_{avg}). Then, we use r_{avg} to normalize: (1) the r_{arm} calculated during the initial round robin phase for each arm and (2) every new r_{step} collected during the main loop of the algorithm. This ensures that there is much less variation in exploration across benchmarks.

The second modification is related to our observation that when many cores are running MAB algorithms concurrently, inter-core interference phenomena can affect the bandit exploration. For example, assume that a core selects a very aggressive arm that causes DRAM bandwidth starvation for other cores. Then, those cores may mistakenly attribute the drops they are observing in their IPCs to the arm that they are currently exploring. This can lead to cores getting trapped in suboptimal actions.

To address this problem, a core p independently restarts the initial round robin phase of the algorithm with a small probability $rr_restart_prob$ during the main loop execution, but without resetting the already-collected r_i and n_i values. With this design, p has a chance to re-evaluate all the arms again in a more stable environment, as it can be assumed that all the other cores have reached a steady state and little exploration is in progress. While this is not a perfect solution, it addresses the problem to some extent.

9.5 Micro-Armed Bandit Design

We design a lightweight microarchitectural agent that implements MAB algorithms in hardware. We call the agent *Micro-Armed Bandit* or *Bandit* for short. In this section, we describe its functionality and microarchitecture. We focus on the DUCB algorithm since our evaluation (Section 9.7) reveals that it is the highest-performing one. We discuss how Bandit can be used for prefetching and SMT fetch PG policy selection. While our work focuses on these two use cases, we anticipate that Bandit will find applications in other microarchitectural decision-making problems with sufficient temporal homogeneity in their action space.

9.5.1 Microarchitecture and Functionality

Bandit has a simple microarchitecture. It consists of two tables, an arithmetic unit, and some control logic. The two tables are the *nTable* and the *rTable*, and each has as many entries as the number of arms. The *nTable* contains, for each arm i , the number of times that i has been selected so far (n_i); the *rTable* contains, for each arm i , the current value of its reward (r_i). The arithmetic unit executes the arithmetic operations in the *nextArm*, *updSels*, and *updRew* functions of Table 45 in hardware. The control logic triggers the selection of the next arm to apply, communicates the selected arm to the controlled microarchitecture units (e.g., data prefetcher or instruction fetch unit), and waits until the bandit step is over to read the values of the appropriate hardware performance counters into the Bandit microarchitecture. The latter will update its state so that it is ready for the next arm selection. This process repeats continuously.

Figure 109 shows the steps in more detail. Figure 109(a) shows the implementation of the *nextArm* function. The hardware sequentially reads the *nTable* and *rTable* for all the arms, calculates the corresponding potentials, and selects the arm with the highest potential as the new arm. Then, in Figure 109(b), the Bandit control logic communicates the arm selection to the controlled entity—in the figure, the L2 data prefetcher and the

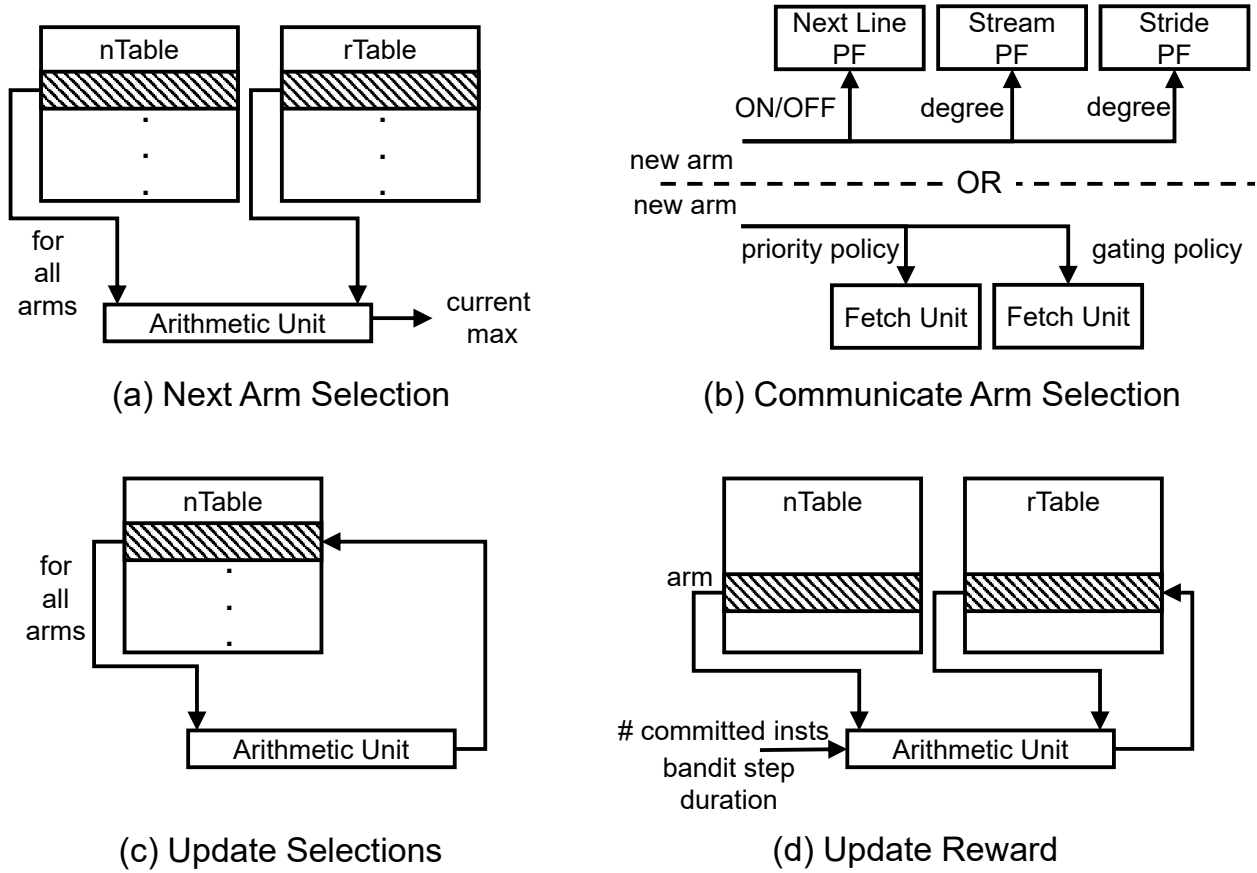


Figure 109: Micro-Armed Bandit microarchitecture.

instruction fetch unit of an SMT core. In the background while the arm selection is communicated, the Bandit updates the nTable (Figure 109(c)) according to the logic of function *updSels* shown in Table 45.

Once the bandit step is over, the Bandit arithmetic unit receives the appropriate hardware performance counters, computes the step’s reward (r_{step}), and accumulates it into the rTable entry of the corresponding arm (Figure 109(d)). The figure assumes that the reward is the core’s average IPC. Hence, the arithmetic unit receives the total number of committed instructions by the CPU core, subtracts from it the number of committed instructions at the end of the previous bandit step, and divides the result by the step duration in cycles (calculated similarly through hardware counters). The result is the step’s IPC or r_{step} . We will see in Section 9.5.4 that some of the operations in

Figure 109(a) and Figure 109(d) can be performed before the bandit step is over.

9.5.2 Prefetching Use Case

We use Bandit to control the degree and type of a set of lightweight and widely adopted L2 prefetchers. This set includes a next-line (NL) prefetcher, a stream prefetcher, and a PC-based stride prefetcher. A bandit arm encodes whether the NL prefetcher is on or off, the degree of the stream prefetcher, and the degree of the stride prefetcher (Figure 109(b)). A degree of zero means that the corresponding prefetcher is off. We assume that, like in the POWER7 [425], the prefetcher degrees can be controlled through programmable registers. Hence, Bandit communicates its selected arm to the prefetchers by writing to the programmable registers. We define the duration of a bandit step as a certain number of L2 demand accesses.

9.5.3 SMT Instruction Fetch Use Case

We use Bandit to control the fetch PG policy of an SMT processor as introduced in Section 9.3. The processor additionally runs the Hill Climbing algorithm [398] to determine the best occupancy threshold to trigger thread fetch gating. Bandit runs on top of the Hill Climbing algorithm and does not control the occupancy threshold. A bandit arm encodes which fetch priority policy will be used and which pipeline structures will be considered for fetch gating (Figure 109(b)). The bandit step duration is defined as a certain number of Hill Climbing epochs, where a different threshold is tested in each epoch [398]. After this specific number of epochs, Bandit is invoked to select the next arm. The Hill Climbing algorithm requires the execution of some epochs before it can pick a good enough threshold for a given arm. For this reason, we use a larger bandit step during the initial round robin phase of the MAB algorithm (Algorithm 4). By holding the selected arm stable for a longer period of time, the Hill Climbing algorithm is given more time to converge, and the r_{step} received in this phase of the algorithm is more representative of the true capabilities

of the arm. We refer to this initial bandit step as the *bandit step-RR*. After the initial round robin phase is over, we decrease the bandit step to a smaller number of epochs during the main loop of the algorithm. Section 9.6.3 describes how we select appropriate values for the bandit steps. Finally, every time the arm changes, the Hill Climbing threshold of the old arm is saved, and the one for the new arm is restored.

9.5.4 Storage Overhead and Latency

Bandit has a tiny storage overhead, which scales linearly with the number of arms. Assuming that r is stored using a single-precision floating-point data type and n is stored using an unsigned integer data type, the storage overhead per arm is 8B. For the maximum number of arms in our evaluation, which is 11 (Section 9.6), the total storage overhead is less than 100B. This low number is a direct result of not decomposing the agent’s environment into individual states but instead treating it as a single MAB state as explained in Section 9.2.2 and Figure 104. In comparison, Pythia, which is an MDP-RL agent, requires 24KB just to store the state-action values.

The main operation that contributes to Bandit’s latency is the selection of the next arm (Figure 109(a)). In a naive design, when the current step’s reward (r_{step}) is received, the agent calculates the average reward of the current arm (r_{arm}). Then, it sequentially calculates the potential for all arms and picks the arm with the highest potential. Assume that n_{total} is available and that $\ln(n_{total})$ is calculated once and reused for all arms, since n_{total} is common. In this case, computing the potential of an arm involves two reads (one from the nTable and one from the rTable), a division, a square root, a multiplication, and an addition (Table 45). By conservatively assuming a single non-pipelined arithmetic unit and a latency of 20 cycles for each of a square root and a division [93], [426], we estimate the total latency for picking the highest potential arm among 11 arms to be less than 500 cycles.

In practice, an advanced design takes much less time. Specifically, while the bandit step

is in progress, the agent can compute in the background the potential of all arms except for the one being tested, and identify the best among them. It can also compute part of the r_{arm} from *updRew* in Table 45. When r_{step} is finally available, the critical path involves finishing the computation of the tested arm reward (r_{arm}), calculating the potential of the tested arm, comparing it to the potential of the best arm identified so far, and picking the best of the two. This operation takes about 50 cycles.

To be conservative, in our evaluation, we assume a critical path of 500 cycles, as if the potentials of all the 11 arms are computed in the critical path. Still, this number is negligible compared to the total duration of a bandit step in our evaluation, which corresponds to 1,000 L2 demand accesses for prefetching and 128k cycles for SMT fetch priority and gating, as described in Section 9.6.

9.6 Evaluation Methodology

9.6.1 Evaluation Environment

We evaluate Bandit and other prior microarchitecture proposals through simulation for both the prefetching and the SMT instruction fetch use cases. For the prefetching use case, we use the trace-driven Champsim [427] simulator, which is the standard choice for such studies [428], [429], [430], [431]. We build on the framework released by Pythia, which has integrated multiple prefetchers with Champsim. For fairness, we use the exact same parameters as the ones used in the Pythia paper for the simulated cores, memory subsystem, and prior prefetchers. The core resembles an Intel Skylake [432] and the parameters are given in Table 46. The prefetcher is associated with the L2: it is trained on L1 cache misses and fills prefetched lines into L2 and LLC. We additionally simulate a configuration with alternative cache sizes (i.e., L2=1MB and LLC/core=1.5MB). We run both single-core and 4-core experiments.

For the SMT use case, since Champsim does not offer a very detailed pipeline model and does not support SMT, we use Gem5 v20 [433]. In more detail, we build on top of the

Table 46: CPU parameters in prefetching experiments.

Fetch Width:	6	LQ/SQ Size:	72/56 entries
Decode Width:	6	IRF/FRF Size:	256/256 regs
Issue Width:	8	Frequency:	4 GHz
Commit Width:	4	L1:	32KB 8-way
IQ Size:	128 entries	L2:	256KB 8-way
ROB Size:	256 entries	LLC/core:	2MB 16-way

framework released by SecSMT [422], [434], which has added full support for SMT in the out-of-order core model of Gem5. Similar to SecSMT, we assume that all the pipeline structures (including ROB, IRF, and LSQ) are dynamically shared between threads. We use the exact same parameters as the SecSMT paper. The parameters are shown in Table 47. The architecture again resembles an Intel Skylake core.

Table 47: CPU parameters in SMT experiments.

Fetch Width:	16B	LQ/SQ Size:	72/56 entries
Decode Width:	5 uops	IRF/FRF Size:	180/164 regs
Issue Width:	8 uops	Frequency:	3.3 GHz
Commit Width:	8 uops	L1:	32KB 8-way
IQ Size:	97 entries	L2:	4MB 16-way
ROB Size:	224 entries	LLC:	No L3

In our simulation, we model a conservative latency of 500 cycles at the end of each Bandit step, until the next arm is selected. During those cycles, the prefetcher and the SMT scheduler do not stall but continue operating with the previously selected arm. This 500-cycle latency is negligible compared to the duration of a bandit step which, as shown in Table 48, is 1,000 L2 demand accesses for the prefetching experiments, and 128k cycles for the SMT instruction fetch experiments.

9.6.2 Applications

To evaluate Bandit for prefetching, we use a large collection of traces spanning different application suites, including SPEC06 [411], SPEC17 [412], PARSEC [435], the Ligra sparse graph workloads [436], and the CloudSuite cloud workloads [437]. For SPEC06 and

SPEC17 we use the traces from the 3rd Data Prefetching Championship (DPC-3) [438]. For CloudSuite, we use traces provided by the 2nd Cache Replacement Championship (CRC-2) [439]. For Ligra and PARSEC, we use the traces released by Pythia.

For single-core experiments, we simulate 1 billion (B) instructions to capture as many program dynamics as possible.

When a trace is less than 1B instructions, we create two new traces. One is created by concatenating the original trace multiple times until 1B instructions are reached; another is created by extending the original trace with smaller traces from different phases of the same program until 1B instructions are reached, to simulate highly-dynamic scenarios. We evaluate both traces and report the average result. Note that, if we stop execution when short traces are consumed, some of the prefetchers may not have time to reach steady-state behavior, potentially leading to inaccurate performance results.

For 4-core experiments, to reduce the simulation time, we simulate until each core has completed 250M instructions. Similar to Pythia, we simulate two types of scenarios for 4-core experiments. First, we assign the same application to every core (forming homogeneous mixes). Second, we assign different applications to different cores (forming heterogeneous mixes).

Since Gem5 is execution-driven, for our SMT evaluation, we capture simpoints [440] from 22 SPEC17 applications using the reference input set. From those applications, we create 226 2-threaded combinations and simulate until each thread has completed 150M instructions. We also simulate an environment where each thread has to complete 250M instructions.

9.6.3 Bandit Tuning and Hyperparameters

To tune the different algorithm hyperparameters (e.g., c , γ , bandit step duration, and Hill Climbing δ in SMT), we use a small subset of our applications, which we will refer to as *the tune set*. For prefetching, we use 46 SPEC traces as the tune set, while for SMT, we use 43

2-threaded mixes stemming from 10 applications. We do not include non-SPEC traces in the prefetching tune set since we want to test the Bandit’s adaptability to completely unseen application suites. This is conservative since Pythia uses more traces spanning different application suites when tuning its hyperparameters. By trying different parameter values in the tune set, we identify the best combinations and use them for our evaluation. Table 48 displays those parameter values. Note that, similar to the original Hill Climbing paper, we define δ (Section 9.3.2) in terms of IQ entries. In the prefetching use case, we tune the hyperparameters using the cache sizes of Table 46. We do not retune Bandit (or any other prefetcher) for the experiment with the alternative cache sizes.

Table 48: Parameters for the SMT thread fetch PG policies and data prefetching policies.

SMT Thread Fetch Priority and Gating		Data Prefetching	
Bandit Algorithm:	DUCB	Bandit Algorithm:	DUCB
γ:	0.975	γ:	0.999
c:	0.01	c:	0.04
# Arms:	6	# Arms:	11
Bandit Step-RR:	32 HillClimb Epochs	Bandit Step:	1000 L2 acc
Bandit Step:	2 HillClimb Epochs	# Trackers in Streamer :	64
Hill Climbing Epoch:	64k cycles	# Trackers in Stride:	64
Hill Climbing δ:	2	rr_restart_prob (4 cores):	0.001

We use 11 bandit arms for prefetching. Table 49 displays the semantics of these arms. For SMT, we prune the number of arms from 64 to the 6 that are displayed in Table 43. The combination of those 6 arms achieved performance very close to the best static performance of all 64 possible fetch PG policies in the tune set.

Table 49: Arms used for prefetching.

Arm id	0	1	2	3	4	5	6	7	8	9	10
NL On/Off	Off	Off	On	Off	Off	Off	Off	Off	On	Off	Off
Stride Degree	0	0	0	0	2	4	0	8	0	0	15
Streamer Deg.	4	0	0	2	2	4	6	6	8	15	15

9.6.4 Comparison to Prior Proposals

For prefetching, we use the simple IP-Stride prefetcher [419] as our baseline prefetcher. Furthermore, we compare our design against the state-of-the-art Pythia, MLOP [400], and Bingo [399] L2 prefetchers. We additionally combine Bandit at L2 and a simple IP-stride at L1 and compare it against the state-of-the-art multi-level IPCP prefetcher [441]. For SMT, we compare our design against ICount (IC_0000) and the Choi policy (IC_1011) (Section 9.3). For the tune set, we additionally evaluate Bandit against the best static arm selection, by exhaustively keeping individual arms stable for the full experiment duration and selecting the best-performing arm on a per-application basis. We do this experiment to evaluate how close Bandit gets to an oracle that statically selects the best arm.

Although our evaluation reveals that DUCB is the best algorithm choice for Bandit, for the tune set, we additionally evaluate the other 2 MAB algorithms and also compare against simple heuristics that do not use MAB algorithms for the exploration. For simplicity, in the 4-core prefetching experiments and the SMT experiments, we evaluate performance based on the sum of the achieved IPCs by all threads. However, Bandit can easily optimize other metrics, such as the average weighted IPC [442] or harmonic mean of weighted IPC [443] by simply changing the Bandit reward and training Hill Climbing using the appropriate metric [398].

9.6.5 Area and Power

To estimate Bandit’s area and power, we use CACTI [102] for the nTable and rTable, and the numbers from [444] for the arithmetic floating-point unit. We use the area and power scaling factors from [104] to scale down to 10nm. At 10nm, the total area and power of a single Bandit agent are 0.00044 mm^2 and 0.11 mW , respectively. To estimate the relative overheads in a real CPU design, we focus on a server-class 40-core Intel Icelake [95] with a TDP of 270W and a total die area of 628 mm^2 [105]. Assuming that each core of the processor is equipped with one Bandit agent, the relative area and power overheads of all

Bandits are less than 0.003%.

9.7 Evaluation

9.7.1 Performance Using the Tune Set

First, we present the results of our evaluation using the tune set. The experiments evaluate all bandit algorithms of Section 9.4, the best static arm selection algorithm, and two additional heuristic exploration methods. We call these two methods the *Single* and the *Periodic* exploration heuristics. The former stops exploring after the initial round robin phase of Algorithm 4, and keeps the arm that performed the best during that initial phase; the latter alternates between periodic phases of round robin exploration of all arms and exploitation of the best arm. We augment *Periodic* with a moving average buffer similar to the one in the POWER7 adaptive prefetcher [418]. This section aims to provide insights regarding the performance of bandit algorithms. We tune the parameters of all the algorithms to achieve good performance.

We start with the prefetching use case. Table 50 shows the min, max, and geometric mean IPC across all tune traces for Pythia, Single, Periodic, ϵ -Greedy, UCB, and DUCB *as a percentage* of the IPC of the best static arm. We observe that DUCB is the best algorithm in terms of geometric mean and min performance, while Pythia is the best for max performance. By holistically modeling the environment through a single state, bandit algorithms are able to explore effectively. Despite their simplicity, UCB and DUCB show better geometric mean performance than Pythia in the tune set.

Table 50: Min, max, and geometric mean IPC of several heuristic and bandit algorithms as a percentage of the IPC of the best static arm for the prefetching use case.

	Pythia	Single	Periodic	ϵ-Greedy	UCB	DUCB
min	88.7	72.8	80.3	89.8	88.6	95.0
max	102.5	100.0	99.8	99.9	100.0	101.6
gmean	98.4	96.5	94.1	97.3	98.8	99.1

When it comes to the highest max performance, we observe that Pythia outperforms

the best static arm by 2.5%. This is because not all applications exhibit sufficient temporal homogeneity in their action space. However, as explained, temporal homogeneity is the common case. Interestingly, the maximum performance of DUCB is also above the best static arm. As we will see later, this is due to coarse-grained dynamic phases inside an application, during which different arms are optimal. DUCB, which is more appropriate than UCB for such scenarios, is able to detect and adapt to phase changes. Naturally, Single displays the lowest min performance since the one-time exploration can lead to very bad choices. The performance of Periodic and ϵ -Greedy is low due to their randomized and non-decaying exploration.

Similar results are presented in Table 51 for the SMT instruction fetch use case. In this table, we also add the Choi policy for reference. The geometric mean performance of DUCB normalized to the best static arm is slightly lower than the one in prefetching (Table 50). We also see that, once again, the max performance of DUCB exceeds the best static arm. We found out that the reason for this is not because the applications have significant phase changes. Instead, it is likely because of the operation of the Hill Climbing algorithm. In the best static arm experiments, the Hill Climbing algorithm can get trapped in local maxima. With DUCB, alternating between different fetch PG policies during exploration injects noise in the IPC that helps Hill Climbing escape those maxima.

Table 51: Min, max, and geometric mean IPC of several heuristic and bandit algorithms as a percentage of the IPC of the best static arm for the SMT thread fetch use case.

	Choi	Single	Periodic	ϵ -Greedy	UCB	DUCB
min	77.2	77.8	88.4	92.0	90.9	92.2
max	101.0	101.1	100.4	100.5	101.1	101.4
gmean	94.5	96.8	97.2	97.8	98.4	98.6

Figure 110 visualizes the exploration that different algorithms perform for different applications. Each plot shows the index of the arm explored as a function of time. The plots in the same row correspond to a specific algorithm (Best Static, Single, UCB, and DUCB); the plots in the same column correspond to a specific application (cactus for

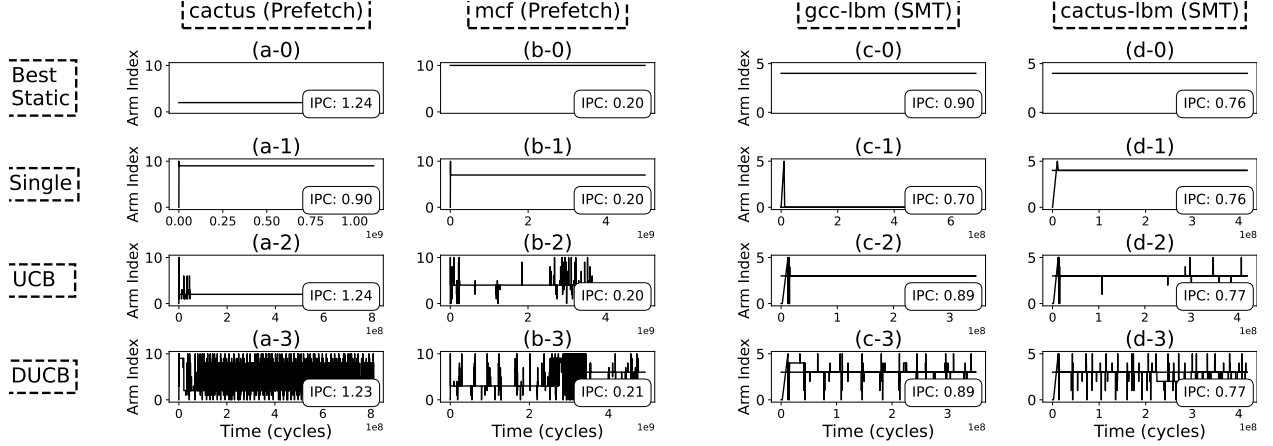


Figure 110: Exploration performed by different algorithms (rows of plots) for different applications (columns of plots).

prefetching, mcf for prefetching, gcc-lbm for SMT, and cactus-lbm for SMT). Each plot also shows the IPC of the algorithm/application combination. We do not include ϵ -Greedy or Periodic in this figure because their plots are very noisy due to their non-decaying exploration. The leftmost two applications run under the prefetching use case, while the rightmost two run under the SMT thread fetching use case.

The plots show that Best Static does not perform arm exploration and that Single explores only during the initial round robin phase. Both UCB and DUCB explore, but DUCB explores more. Different algorithms choose different arm indices for the same application. For example, Single may choose a different arm than Best Static. This results in a dramatic IPC drop in cactus and gcc-lbm. On the other hand, although UCB and DUCB may frequently not select the best arm, they select an alternative arm that does equivalently well. For example, in gcc-lbm, both UCB and DUCB select arm three, which has a very similar IPC as the optimal arm four.

The mcf plot is an example of how DUCB is able to adapt to phase changes. At around three billion cycles, a phase change occurs. DUCB realizes this and starts increasing its exploration rate. Finally, it settles with a different arm than the one used in the first three billion cycles. The resulting 0.21 IPC is higher than the 0.20 IPC of the Best Static algorithm. On the other hand, UCB is unable to adapt to the phase change and keeps the

same arm used in the first three billion cycles. Also note that, for cactus-lbm, the IPCs of UCB and DUCB are higher than the Best Static algorithm and no phase change occurs. This is due to the injection of exploration noise, which helps the Hill Climbing threshold escape local maxima.

Since DUCB outperforms the other bandit algorithms in the tune set, in the rest of this section, we focus on this bandit algorithm.

9.7.2 Performance of Prefetching

9.7.2.1 Single-core Evaluation

We compare the performance of Bandit to that of the baseline Stride prefetcher, the non-RL prefetchers MLOP and Bingo, and the MDP-RL prefetcher Pythia. Figure 111 compares the geometric mean IPC of these designs across all the traces for a single core. All IPC numbers are normalized to a scenario with no L2 prefetcher.

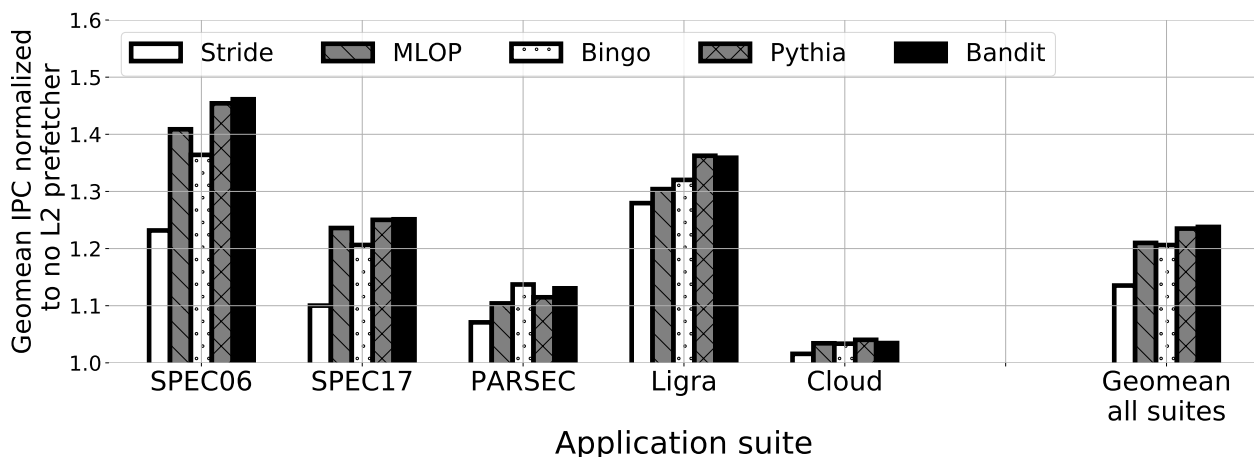


Figure 111: Single-core performance for different state-of-the-art L2 prefetchers.

We observe that Bandit achieves the best or close to the best performance in all the application suites. When the geometric mean of all the suites is considered, Bandit outperforms Stride by 9%, Bingo by 2.6%, MLOP by 2.3%, and Pythia by 0.2%. Remarkably, Bandit matches the performance of Pythia, even though it introduces less than 100 bytes of storage overhead on a conventional processor and can be reused across

different use cases. In contrast, the storage overheads of Pythia, MLOP, and Bingo are 25.5KB, 8KB, and 46KB, respectively. Even if we include in the Bandit’s storage overhead calculation the storage of the NL, stream, and stride prefetchers (which are already fundamental parts of modern processors), Bandit’s total storage overhead for prefetching is less than 2KB.

There is a combination of reasons why the simple bandit algorithm matches the performance of the more complex Pythia algorithm. First, prefetching is characterized by substantial temporal homogeneity (Section 9.3.1). Secondly, Bandit uses the sophisticated DUCB exploration, while Pythia uses an ϵ -Greedy action selection mechanism to explore actions for different states. Finally, Bandit uses the IPC as the reward, while Pythia assigns rewards based on prefetch timeliness and accuracy. Therefore, Bandit is designed to maximize performance directly, while Pythia is designed to maximize performance indirectly through prefetch timeliness and accuracy.

Figure 112 shows single-core prefetches and LLC misses normalized to LLC misses without prefetching (similar to [387]) for NoPrefetch, Stride, Bingo, MLOP, Pythia, *BanditIdeal*, and Bandit. *BanditIdeal* does not account for the 500 cycles needed to update the bandit structures at the end of each bandit step. The figure shows the prefetches that are timely, the LLC misses that are uncovered (or that there is a late prefetch that is unable to cover the whole miss latency), and the prefetches that are wrong (i.e., the overpredictions).

The figure suggests that Bandit is a conservative prefetcher. On average, it reduces the absolute number of wrong prefetches by 66% and 58% compared to Bingo and MLOP, respectively, while not issuing many more wrong prefetches than Pythia. The fraction of LLC misses covered with timely prefetches is 49% for Stride, 69% for Bingo, 63% for MLOP, 72% for Pythia, and 67% for Bandit. This number is slightly lower in Bandit than in Pythia due to cases of medium temporal homogeneity. However, recall that Bandit is trained to maximize the end result (i.e., IPC) and not the prefetch coverage as in Pythia.

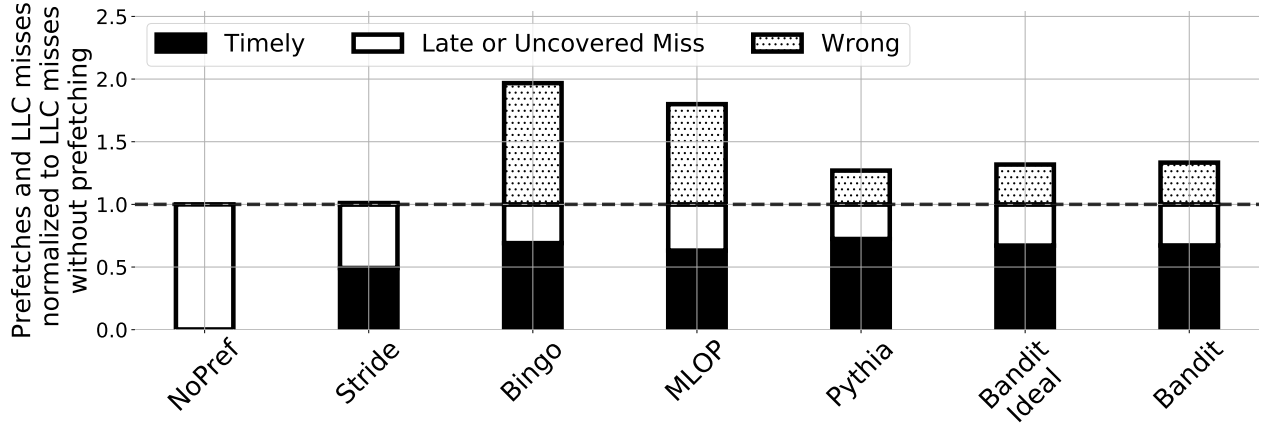


Figure 112: Single-core LLC misses and prefetches classified into timely, late, and wrong.

For this reason, Bandit is able to detect the configurations that cover the misses that are most critical for performance. As a result, although covering slightly fewer misses, Bandit is able to match Pythia’s IPC (Figure 111). Finally, the number of timely prefetches is very similar in Bandit and in *BanditIdeal*. This result suggests that the Bandit latency does not significantly impact the timeliness of prefetches.

As explained by the Pythia authors [387], Pythia uses information provided to it about the memory bandwidth usage in the system. This memory bandwidth usage awareness allows Pythia to substantially outperform MLOP and Bingo in bandwidth-constrained system configurations. In Figure 113, we compare the performance of Bandit and Pythia for different available DRAM bandwidth values in megatransfers per second (MTPS). Our baseline configuration has an available DRAM bandwidth of 2400 MTPS, and the figure performs a sweep using 16 times smaller, 4 times smaller, and 4 times larger available DRAM bandwidth.

From the figure, we see that Bandit performs as well as Pythia in all configurations. Interestingly, Bandit outperforms Pythia by 2.5% in the most bandwidth-constrained configuration (i.e., for 150 MTPS). This is because the Bandit agent judges actions based on the end result (i.e., final IPC). Aggressive prefetching actions in bandwidth-constrained system configurations do not yield good IPC results. Bandit learns this and outperforms

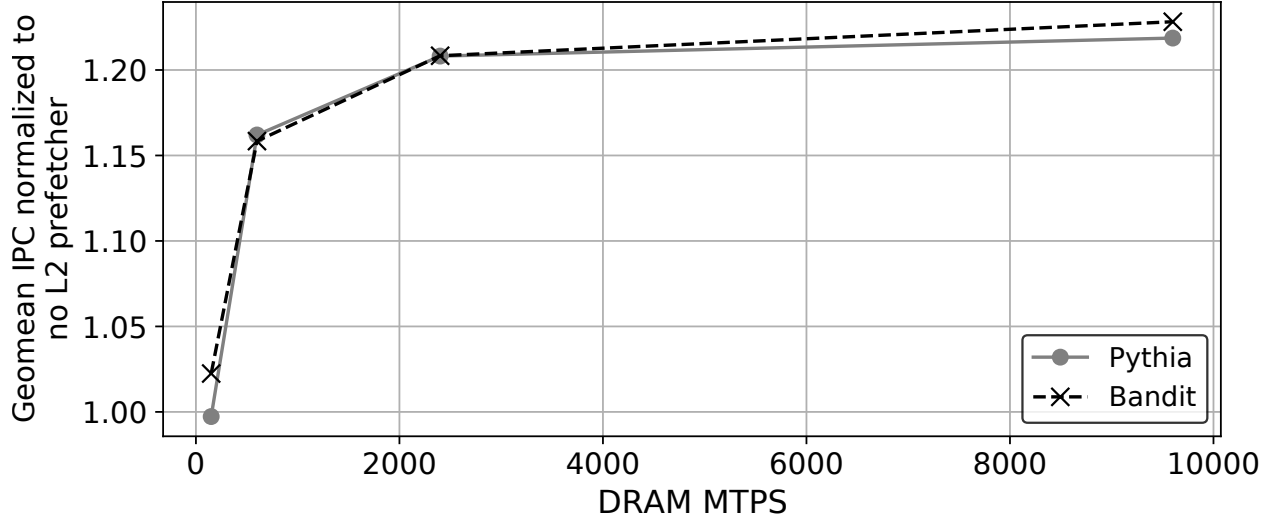


Figure 113: Performance of Pythia and Bandit for different available DRAM bandwidth values in megatransfers per second (MTPS).

Pythia without needing to use any bandwidth-related information in its reward function.

9.7.2.2 Comparison for Other Configurations

In this section, we consider some architectural variations. Note that we do not retune any of the prefetchers for the new architectures. Figure 114 repeats the experiment in Figure 111 in a system with a different cache hierarchy. We change the L2 size to 1MB and the LLC size to 1.5MB, which are similar to the sizes used in the Intel Skylake. The rest of the system parameters are kept the same. We observe that Bandit still attains better performance than the other prefetchers. When the geomean of all suites is considered, Bandit outperforms Stride by 9%, Bingo by 1.5%, MLOP by 4.9%, and Pythia by 0.2%.

We now consider hierarchies with both L1 and L2 prefetchers. They include the multi-level IPCP prefetcher and different L2 prefetchers (Stride, Pythia, and Bandit) augmented with a simple stride prefetcher at the L1 cache (*Stride_Stride*, *Stride_Pythia*, and *Stride_Bandit*). Figure 115 shows the geometric mean IPC of each of these configurations relative to a scenario with no L1 or L2 prefetcher. The figure shows that the geomean IPC increase over no-prefetching is 16% for *Stride_Stride*, 24.5% for IPCP, 24.8%

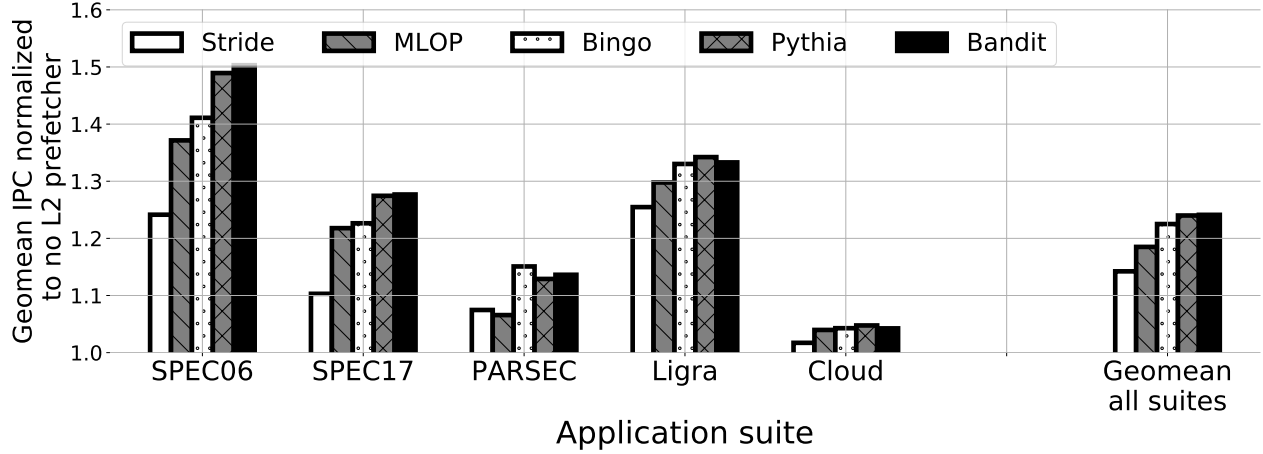


Figure 114: Single-core performance for a different cache hierarchy.

for Stride_Pythia, and 24.5% for Stride_Bandit. This reveals that Bandit at L2 in combination with a simple stride at L1 is an excellent option. Some extensions that could further increase the performance of Bandit could be: (1) use a second Bandit to control lightweight prefetchers at L1 or (2) use a single Bandit to control both the L1 and L2 prefetchers in conjunction. We plan to evaluate such extensions in future work.

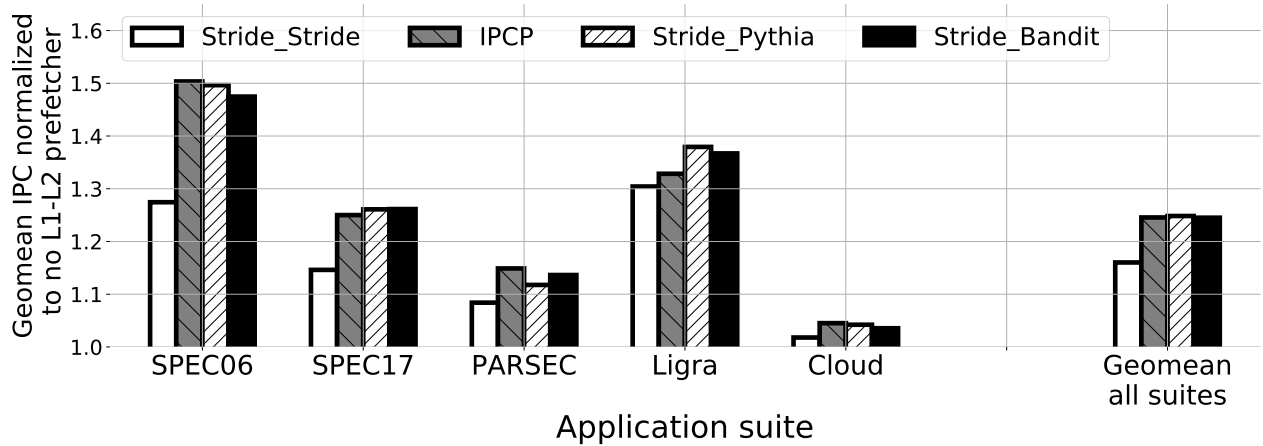


Figure 115: Single-core performance for different multi-level prefetcher combinations.

9.7.2.3 Multicore Evaluation

In this section, we evaluate a four-core architecture. Figure 116 shows the four-core performance for different state-of-the-art L2 prefetchers. The four cores of the architecture

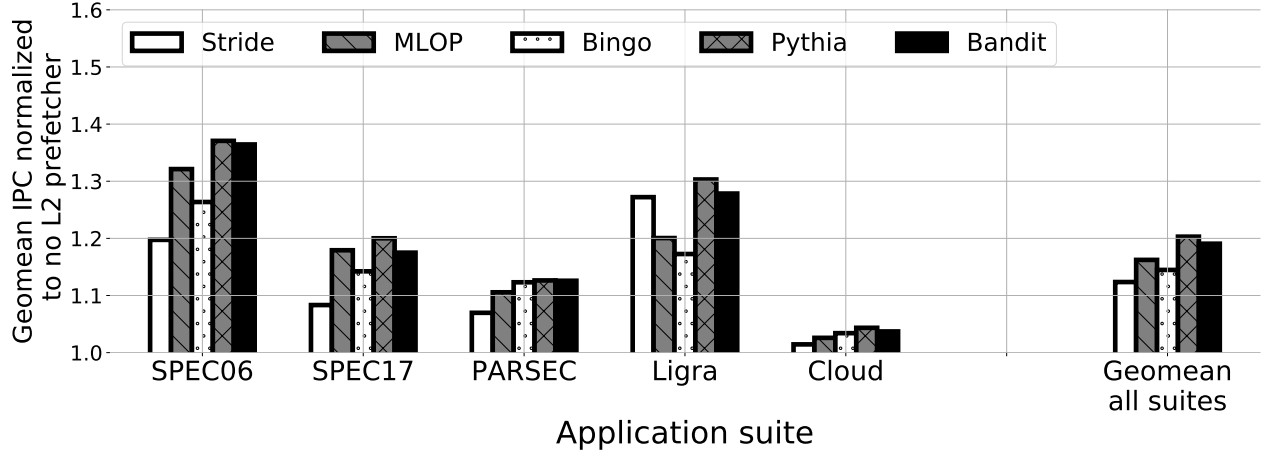


Figure 116: Four-core performance for different state-of-the-art L2 prefetchers.

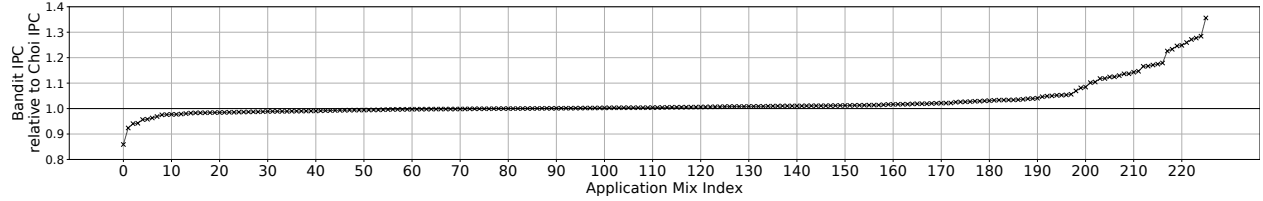


Figure 117: Performance of Bandit relative to Choi for 226 2-thread mixes from 22 SPEC17 applications.

all run the same single-threaded application. Looking at the geometric mean of all suites, Bandit outperforms Stride by 6%, MLOP by 2.4%, and Bingo by 4.0%. However, it performs slightly worse (by 1.0%) than Pythia. The reason is that, in the 4-core scenario, the reward is more noisy than in the single-core scenario. Since four bandits are operating and exploring in parallel, their decisions interfere. If a bandit chooses an aggressive arm that causes DRAM bandwidth starvation for the other cores, the bandits of these other cores may mistakenly attribute the drop in their IPCs to the arm they are currently exploring. The result could be to get trapped in suboptimal arms. It can be shown that, for workloads that combine different applications, the result is similar.

Overall, Bandit’s performance for 4-core mixes is good, especially considering its tiny storage footprint. In future work, we plan to investigate alternative rewards that are not very sensitive to inter-core interference and advanced techniques to orchestrate the exploration of multiple bandits [401].

9.7.3 Performance of SMT Thread Fetching

We now consider the SMT fetch policy use case. We compare the performance when Bandit is used to control the SMT thread fetch PG policy and when the Choi policy is used. In the evaluation, we use the 226 2-thread mixes from 22 SPEC17 applications discussed in Section 9.6.2. Figure 117 shows the IPC achieved with Bandit relative to the IPC attained with Choi. In the x-axis, the application mixes are sorted from lower to higher ratio of Bandit IPC to Choi IPC.

We observe that Bandit offers clear benefits over Choi. It outperforms Choi by more than 4% in 36 application mixes, and by 36% in one mix. On the other hand, Choi outperforms Bandit by more than 4% in only 6 application mixes. These are cases in which the Bandit exploration is not optimal and settles for policies worse than Choi (i.e., IC_1011). Overall, including all the mixes, Bandit offers a 2.2% geometric mean speedup over Choi and, although not shown in the figure, a 7% geometric mean speedup over plain ICount (i.e., IC_0000). Finally, we repeated these experiments extending the simulation duration to 250M instructions from each thread and observed similar results.

To understand the performance difference between Bandit and Choi, we focus on the activity of the rename stage of the pipeline of the core. The rename stage can be stalled, idle, or running (i.e., doing useful work). Rename is stalled due to full conditions in various shared pipeline structures—mostly the ROB, IQ, LQ, SQ, and RF. Rename is idle when the fetch and decode stages have not pushed any instructions to it, possibly due to fetch gating.

Figure 118 shows, for Choi and Bandit, the average fraction of cycles that the rename stage is in different states. From left to right, the bars show cycles stalled due to ROB being full, IQ being full, LQ being full, SQ being full, and RF being full. The next three bars are the average fraction of cycles when rename is stalled due to any structure being full, when rename is idle, and when rename is running.

The figure shows that Bandit reduces the fraction of cycles when rename is stalled and when rename is idle. As a result, it increases the fraction of running cycles. In particular,

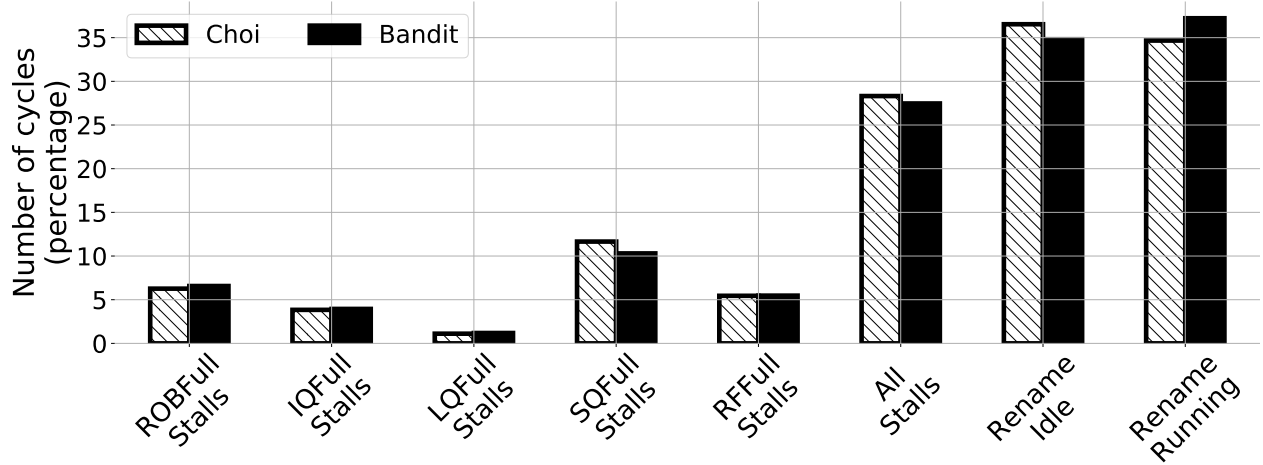


Figure 118: Percentage of cycles when rename is stalled, idle, or running for Bandit and Choi.

Bandit decreases rename stalls because it includes arms that are aware of SQ occupancy. As a result, it decreases the stalls due to SQ full conditions. Importantly, Bandit decreases rename idle cycles by being able to detect the criticality of each structure in cases of asymmetric utilization (Section 9.3.3). In this way, it eliminates a substantial number of conservative fetch gating occurrences. As a result, on average, the fraction of cycles when rename is running is 2.6% higher with Bandit than with Choi.

9.8 Related Work

1. Prefetching: Throughout the years, numerous prefetchers have been proposed, including adaptive [253], [418], [421] and composite [420], [445] prefetchers. IPCP [441] groups instruction pointers in classes based on their access patterns and uses a lightweight prefetcher per class. Jimenez et al. [418] propose heuristic software techniques to control the aggressiveness of the IBM POWER7 stream and stride prefetchers. Software techniques are not as agile as hardware agents and cannot detect short-lived program mini-phases. Our *Periodic* heuristic is inspired by this work.

The Best Offset Prefetcher (BOP) [446] tries to identify a single best offset that is used for all cache lines, by scanning all the possible options and learning the optimal one per

epoch. In addition, it always prefetches with a degree of 1. Although such an approach would work in cases of perfect temporal homogeneity, it fails to adapt in scenarios with high (but not perfect) homogeneity, in which a few different degrees and offsets are optimal (Figure 106). On the other hand, Bandit controls an adaptable prefetcher mix and thus can concurrently sustain a few different degrees and offsets in the same program phase. Hence, it can adapt in cases of high but imperfect temporal homogeneity. An additional difference is that, instead of scanning all the possible options as in BOP, Bandit uses the sophisticated DUCB exploration algorithm.

In the RL domain, the most notable examples are Pythia [387] and the Context Prefetcher [390], which were discussed in Section 9.2. Pythia was shown to outperform the Context and adaptive POWER7 prefetchers [387]. In our evaluation, we extensively compared Bandit’s performance to Pythia’s and other state-of-the-art prefetchers [399], [400], [441]. Recently, ML and RL-inspired prefetch managers have been proposed [391], [447] to coordinate multi-level prefetchers. We believe that Bandit can also be useful to manage multi-level prefetchers and we plan to test this use case in future work.

2. SMT resource distribution: Apart from the Hill Climbing algorithm [398], there are other methods for distributing shared resources in an SMT pipeline. For example, ARPA [448] assigns more resources to threads based on their usage efficiency. Using Bandit to augment such alternative methods, in a manner similar to how we used it for Hill Climbing is an interesting research direction. Some recent works employ neural networks for SMT processor resource distribution [449], [450]. However, the high complexity and overhead of such ML models are hard to justify for resource-constrained environments such as a processor pipeline.

3. Hardware RL agents for other use cases: Apart from the use cases that we focused on in this work, hardware RL agents have been proposed for a variety of other applications. For example, Ipek et al. [386] propose a SARSA-based memory controller, while Zheng and Louri [392] propose a NoC design with per-router Q-Learning agents.

Sibyl [388] uses RL for data placement in hybrid storage systems. Cohmeleon [389] employs Q-Learning to find the best cache coherence mode for different accelerators in heterogeneous SoCs. While effective, all of these approaches suffer from the increased complexity and overhead introduced by decomposing the environment into multiple states, as explained in Section 9.2. SOSA [401] is a cross-layer system for hierarchical SoC management. It employs a software supervisor that coordinates low-level hardware agents. Ideas from SOSA could be useful to coordinate multiple Bandits operating in parallel.

9.9 Further Directions

We outline two possible future research directions. The first is to investigate Bandit’s applicability to additional use cases and architectures, including accelerators. Some potential use cases are discussed in the last part of Section 9.8. In future work, we plan to investigate whether the simplified approach followed by Bandit is sufficient for some of these use cases.

The second direction is to study Bandit extensions with slightly higher storage needs than the design proposed here, which can increase performance further. There are many ways in which potential Bandit extensions could utilize additional storage budget. An obvious way is to increase the action space. For example, for the prefetching use case, Bandit could also control the target cache level and/or prefetch throttling mechanisms. An alternative extension is to use a single Bandit to control multiple ensembles. One example is to control both the L1 and L2 prefetcher configurations or to jointly select the prefetcher configuration and the cache replacement policy. Note that the action space size in this case is given by the product of the action space of each ensemble (e.g., for 10 L1 and 10 L2 prefetcher configurations, there are a total of 100 Bandit actions).

During our tuning experiments, we observed that different values of the DUCB hyperparameters (e.g., γ and c in Table 48) worked best for different applications. This observation motivates an alternative way to utilize any additional storage budget: multiple

low-level Bandits with different hyperparameter values could be concurrently active, and a high-level Bandit would select the best one. Finally, although Bandit is unable to discriminate environment states (Figure 104), it could be augmented with a classifier module that classifies memory access patterns [441], [451] online. Then, a separate Bandit could be used to pick the appropriate actions for each pattern type.

9.10 Conclusion

Our analysis of some microarchitecture decision-making problems reveals that, in a given time window, only a small fraction of the action space is useful. We exploit this observation by modeling the decision-making as a Reinforcement Learning (RL) environment using a single state, drastically reducing complexity and storage requirements. We then design a hardware RL agent implementation based on Multi-Armed Bandit algorithms that is lightweight and reusable across different microarchitecture problems. We call our RL agent *Micro-Armed Bandit* or Bandit.

We showcase Bandit in two use cases: data prefetching and instruction fetch in SMT processors. For prefetching, our agent outperforms non-RL prefetchers Bingo and MLOP by 2.6% and 2.3% (geometric mean), respectively, and attains similar performance as the state-of-the-art RL prefetcher Pythia—with the dramatically lower storage requirement of only 100 bytes. For SMT instruction fetch, our agent outperforms the Hill Climbing method by 2.2% (geometric mean).

9.11 Chapter Supplement: Guidelines to Use Bandit for Other Problems

Although our evaluation demonstrated Bandit’s effectiveness on two CPU decision-making problems, we believe that Bandit can find applications in a broader range of problems and heterogeneous architectures, including accelerators. To that end, in this chapter supplement, we provide guidelines to use Bandit for more general microarchitecture problems. We discuss: (1) a method for determining Bandit’s applicability for a specific

problem, (2) practical ways to maximize Bandit’s performance, and (3) an example of a problem ill-suited for Bandit.

9.11.1 Prerequisites

Initially, the microarchitecture decision-making problem should be expressed as an RL problem, and its *action space* (e.g., the possible combinations of prefetching degree and offsets for prefetching) should be defined. After that, a representative set of traces or simpoints from phases of different benchmarks should be collected. This set will be used to (1) characterize the applicability of Bandit and (2) tune Bandit to maximize its performance. We refer to this set of traces as the *Tune Set*. For the use-cases described earlier in this chapter, we used about 10% of our traces/simpoints as the tune set.

Finally, we need a state-of-the-art RL or non-RL decision-making agent for the problem of interest to assist in assessing the effectiveness of a Bandit-based solution. We refer to this agent as the *Baseline Agent*. For our use-cases, we used the Pythia prefetcher [387] and the fixed SMT policy proposed by Choi and Yeung [398] as our baseline agents.

9.11.2 Determining if Bandit is Applicable to a Microarchitecture Decision-making Problem

After the *action space*, the *tune set*, and the *baseline agent* have been defined, it is time to determine whether Bandit can be a good choice for the problem of interest. As explained, for Bandit to be a good choice, the problem should be characterized by *temporal homogeneity in the action space* and it should offer *adaptation opportunities*.

Characterizing Temporal Homogeneity Characterizing the temporal homogeneity in the action space of the problem of interest is not trivial, since it would require a perfect agent that selects the optimal action at every timestep. One could observe the actions of the baseline agent and determine if they are temporal homogeneous. If they are, this is an

indication that a Bandit agent could potentially be used as a low-cost alternative to the baseline agent. Otherwise, one cannot yet declare that the problem lacks temporal homogeneity. This is because the baseline agent is not perfect.

In cases of high but imperfect temporal homogeneity, one can also consider redefining the action space. For example, a possible approach is to use Bandit as a coordinator of lower-level agents or predictors, in the same manner that we used it as a coordinator of lightweight prefetchers.

Characterizing Adaptation Opportunities In order for Bandit to offer benefits for a given problem, different arms should be optimal for different benchmarks or for different phases of the same benchmark. In this case, we say that the problem offers *adaptation opportunities*.

To determine whether the problem of interest offers adaptation opportunities, one should execute, for each trace (or simpoint), as many runs as the number of actions (i.e., arms) available to Bandit. In each run, a given arm is applied for the whole duration of the run (i.e., *constant arm run*). After all *constant arm runs* are completed, one should identify the arm that performed the best on average across all traces of the tune set (i.e., *best constant arm*). In addition, one can compute the resulting performance if the best arm for each trace was selected by an oracle before the run (i.e., *per-trace arm oracle*).

If the performance of the *per-trace arm oracle* is not higher than the one of the *best constant arm*, and assuming that no phase changes occur during the execution of a trace, one can conclude that Bandit is not a good fit for the problem of interest.⁶ In addition, if the *per-trace arm oracle* does not outperform the baseline agent, then Bandit will likely not offer performance improvements over the baseline agent.

⁶Note that this could change by redefining the action space.

9.11.3 Maximizing Bandit Performance

After deciding to employ Bandit for a microarchitecture decision-making problem, in order to get the most out of it, one needs to choose: the appropriate MAB algorithm, an effectively-pruned action space, and appropriate values for the algorithm hyperparameters (e.g., the bandit step).

Algorithm Selection In our experience, DUCB performed better than ϵ -Greedy and UCB in all our use cases. Therefore, we recommend it as a starting point. Note that MAB algorithms are an active research topic and we could not exhaustively evaluate all possible options. We leave for future work evaluating Bandit with alternative MAB algorithms.

Arm Pruning An optimization that we employed to increase Bandit’s performance is *arm pruning*. Simply put, we removed arms that were not expected to contribute much to the performance at runtime. To achieve this, we iteratively removed (pruned) one arm and re-computed the performance of the per-trace arm oracle. If the pruning of the arm resulted in performance degradation below a threshold, we discarded that arm. Otherwise, we re-inserted the arm in the arm pool. Decreasing the number of arms to be tried at runtime decreases Bandit’s exploration cost and can boost performance. However, one limitation of arm pruning is that it is performed based on the contents of the tune set. Thus, there is the danger of removing arms that could potentially be very useful in circumstances not captured in the tune set.

Hyperparameter Tuning MAB algorithms involve hyperparameters that need to be set to appropriate values in order to maximize performance. They include the bandit step and a few algorithm-specific parameters (e.g., the exploration probability ϵ in ϵ -Greedy [383]). We tuned the hyperparameters using a simple multi-dimensional grid search, performing tune set runs for each point of the grid (which encodes a combination of hyperparameter values). An indicator of the quality of a specific hyperparameter value set

is whether Bandit’s average performance gets close, or even exceeds, the one of the per-trace arm oracle.

9.11.4 An Ill-suited Problem for Bandit

To help the reader better understand the limitations of Bandit, we now give an example of a microarchitecture problem that is ill-suited for Bandit. Imagine a scenario where Bandit is used to determine, for a given cache set, the way that should be replaced in the event of an eviction. The action space has as many actions as the number of ways of the set, and each arm encodes the id of the way to be replaced. It is easy to show that this problem is not characterized by temporal homogeneity. If we assumed temporal homogeneity, we would repeatedly select the same way of the set for replacement during a phase of execution. This would lead to a LIFO replacement policy, which is known to deliver sub-par performance. It is well-known that determining the optimal way to evict requires contextual information that conveys future knowledge (what cache line will be used further in the future). Thus, using Bandit for this problem is suboptimal.

Chapter 10 Future Work

Substantial research remains before truly efficient next-generation accelerator-centric systems can be realized. Holistically redesigning the computing stack around accelerators is an interdisciplinary effort that requires cross-stack collaboration across computer architecture, applied machine learning, high-performance computing algorithms, systems and compilers, networking, and circuits. This section outlines several cross-stack research directions inspired by this thesis, focusing on the scaling, optimization, programming, and integration of future accelerator-centric systems.

Scaling. In this thesis, we presented NetSparse, which specializes networking hardware to enable sparse accelerators to scale when distributed across the nodes of large clusters. We believe that NetSparse-inspired specialized in-network support is a promising research direction. For sparsity alone, there is a lot more to be done. One direction we are currently exploring is in-network support for sparse collectives. Unlike traditional dense collective communication, sparse collectives could avoid redundant transfers by automatically pruning communication in the network. To support this functionality, we envision extending programmable switches with hardware support for selective multicast.

Beyond sparsity, we believe that specialized networking hardware may prove useful for more applications such as Agentic AI [25]. Further, with the emergence of Compute Express Link (CXL) [452], CXL devices offer an additional potential specialization target beyond NICs and Switches.

In addition to delivering high performance, specialized networking hardware must also be usable by multiple tenants in large-scale shared clusters. To that end, future work could extend specialized networking hardware with virtualization support while jointly considering performance, isolation, and security.

Programming. Heterogeneous systems are complex to program. For example, to use Near-Core Accelerators with ATX (Chapter 5), a programmer has to configure the UTE, generate task metadata, and invoke the NCA in a high-level language. Automating ATX programming is an interesting future step. In particular, a compiler could be used to automatically generate source-level ATX programs from source-level non-ATX programs. Such a compiler would detect acceleratable code segments, identify streams and dependencies, configure the UTE, partition loops into tasks, and invoke accelerators using ATX instructions. Recently, Large Language Models (LLMs) have shown promise for source-to-source transformations of CPU code [453]. They may also prove useful to perform non-ATX to ATX source code transformations.

Near-Core Accelerators may not be the optimal choice for all acceleratable code. Hence, another future direction is to build compilers that, starting from CPU source code possibly written in a domain-specific language, automatically partition it into Near-Core, In-Core, Out-of-Core, and decoupled accelerator segments for maximum performance.

Optimization. The third part of this thesis primarily investigated performance modeling and optimization techniques for systems confined within a single node. Future work could extend the scope to model and tune the end-to-end performance of heterogeneous specialization-rich supercomputers. To achieve this, future frameworks should co-optimize and co-design the system as a whole and not as a collection of components. A long-term vision is the development of Agentic AI-based toolchains that, given a target application, can largely automate the design of high-performing end-to-end systems, including accelerators, networking devices and topologies, memory hierarchies, compilers and runtimes.

Integration. In the first part of this thesis, we investigated processor–accelerator coupling at the architectural level. An interesting future direction is the physical realization of such tightly-coupled heterogeneous architectures. Advances in heterogeneous

integration are creating new opportunities in this space. Notably, 3D stacking [454] enables vertically stacking heterogeneous layers, potentially built with different device technologies — such as logic, memory, analog, and photonic components — into a single tightly integrated system. However, it remains unclear how to connect heterogeneity at the device and integration level with heterogeneity at the computer architecture level. Future work could explore the design of efficient 3D heterogeneous systems by physically partitioning the logic structures of CPU cores and different types of accelerators, such as Near-Core and In-Core accelerators, across heterogeneous 3D layers.

Chapter 11 Conclusions

Today, computing is undergoing a paradigm shift, with computation increasingly moving away from general-purpose processors and toward highly heterogeneous, accelerator-intensive platforms. To improve accelerator efficiency in this emerging landscape, this thesis introduced new hardware and software support that makes accelerators easier to control, scale, and optimize.

The first part of this thesis proposed *tightly-coupled* host-accelerator architectures that reduce control and data movement overheads. It first developed two accelerator architectures that are tightly coupled with their CPU hosts and share the cache subsystem and address translation mechanisms: *SPADE*, which targets sparse computations, and *DECA*, which targets the decompression of Large Language Model weights during inference. It then generalized the host-accelerator coupling ideas of *SPADE* and *DECA* beyond individual accelerator designs through the Near-Core Accelerator abstraction and the Accelerator Task Extensions (*ATX*).

The second part of this thesis introduced techniques that specialize communication hardware and software to enable scalable multi-accelerator systems. On the software side, it described *Two-Face*, a communication algorithm that combines coarse-grained collectives and fine-grained one-sided accesses to accelerate distributed sparse communication. On the hardware side, it proposed *NetSparse*, which extends modern Network Interface Cards and Programmable Switches with hardware support specialized for sparsity. By accelerating communication, *NetSparse* was shown to improve the performance of distributed sparse accelerator systems by an order of magnitude.

The final part of this thesis contributed automated and reusable techniques for optimizing modern heterogeneous architectures. First, it developed the 3D *Roof-Surface* performance model for reasoning about the performance of heterogeneous accelerator ensembles at design time. It then proposed *HotTiles*, a static work-partitioning method for

optimizing the performance of heterogeneous sparse accelerators. Finally, it contributed *Bandit*, a hardware agent that optimizes performance by modeling microarchitectural decision-making as a lightweight reinforcement-learning problem.

Overall, the proposed techniques make accelerators easier to control, scale, and optimize, supporting the design of more efficient next-generation accelerator-centric systems.

References

- [1] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc, “Design of ion-implanted MOSFET’s with very small physical dimensions,” *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974. DOI: 10.1109/JSSC.1974.1050511
- [2] G. E. Moore, “Cramming more components onto integrated circuits,” *Electronics*, vol. 38, no. 8, pp. 114–117, 1965.
- [3] A. Shehabi, S. J. Smith, A. Hubbard, A. Newkirk, N. Lei, M. A. B. Siddik, B. Holecek, J. G. Koomey, E. R. Masanet, and D. A. Sartor, “2024 United States Data Center Energy Usage Report,” Lawrence Berkeley National Laboratory, Berkeley, California, Tech. Rep. LBNL-2001637, 2024. DOI: 10.71468/P1WC7Q
- [4] J. L. Hennessy and D. A. Patterson, “A new golden age for computer architecture,” *Commun. ACM*, vol. 62, no. 2, pp. 48–60, Jan. 2019, ISSN: 0001-0782. DOI: 10.1145/3282307 [Online]. Available: <https://doi.org/10.1145/3282307>
- [5] C. E. Leiserson, N. C. Thompson, J. S. Emer, B. C. Kuszmaul, B. W. Lampson, D. Sanchez, and T. B. Schardl, “There’s plenty of room at the Top: What will drive computer performance after Moore’s law?” *Science*, vol. 368, no. 6495, eaam9744, 2020. DOI: 10.1126/science.aam9744 [Online]. Available: <https://www.science.org/doi/abs/10.1126/science.aam9744>
- [6] NVIDIA, *NVIDIA Blackwell Architecture Technical Brief*, NVIDIA, 2024. Accessed: 2024. [Online]. Available: <https://resources.nvidia.com/en-us-blackwell-architecture>
- [7] J. Choquette, “NVIDIA Hopper H100 GPU: Scaling performance,” *IEEE Micro*, vol. 43, no. 3, pp. 9–17, 2023.

- [8] NVIDIA, *NVIDIA Tesla V100 GPU Accelerator datasheet*, NVIDIA, 2018. [Online]. Available:
<https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf>
- [9] T. Norrie, N. Patil, D. H. Yoon, G. Kurian, S. Li, J. Laudon, C. Young, N. Jouppi, and D. Patterson, “The Design Process for Google’s Training Chips: TPUv2 and TPUv3,” *IEEE Micro*, vol. 41, no. 2, pp. 56–63, 2021. DOI: 10.1109/MM.2021.3058217
- [10] N. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai, N. Patil, S. Subramanian, A. Swing, B. Towles, C. Young, X. Zhou, Z. Zhou, and D. A. Patterson, “TPU v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings,” in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–14.
- [11] N. Bshara, “AWS Trainium: The Journey for Designing and Optimization Full Stack ML Hardware,” in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS ’24, La Jolla, CA, USA: Association for Computing Machinery, 2024, p. 4. DOI: 10.1145/3620666.3655592 [Online]. Available: <https://doi.org/10.1145/3620666.3655592>
- [12] J. Coburn, C. Tang, S. A. Asal, N. Agrawal, R. Chinta, H. Dixit, B. Dodds, S. Dwarakapuram, A. Firoozshahian, C. Gao, K. Gondkar, T. Graf, J. Hu, J. Huang, S. Hughes, A. Hutchin, B. Jakka, G. J. Chen, I. Kalyanaraman, A. Kamath, P. Kansal, E. Kazi, R. Levenstein, M. Maddury, A. Mastro, S. Medaiyese, P. Modi, J. Montgomery, S. Nadathur, A. Nagpal, A. Narasimha, M. Naumov, E. Ozer, J. Park, P. Ramani, H. Reddy, D. Reiss, D. Roy, S. Sekar, A. Sharma, P. Shetty, A. Sukumaran-Rajam, E. Tal, M. Tsai, S. Varshini,

- R. Wareing, O. Wu, X. Xie, J. Yang, H. Yu, T. Zargar, Z. Zeng, F. Zhang, A. Matthews, X. Jiao, J. Zhang, E. Menage, T. E. Stokke, and M. Sourouri, “Meta’s Second Generation AI Chip: Model-Chip Co-Design and Productionization Experiences,” in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, 2025, pp. 1689–1702.
- [13] Z. Jia, B. Tillman, M. Maggioni, and D. P. Scarpazza, *Dissecting the Graphcore IPU Architecture via Microbenchmarking*, 2019. arXiv: 1912.03413 [cs.DC]. [Online]. Available: <https://arxiv.org/abs/1912.03413>
- [14] S. Lie, “Cerebras Architecture Deep Dive: First Look Inside the Hardware/Software Co-Design for Deep Learning,” *IEEE Micro*, vol. 43, no. 3, pp. 18–30, 2023. DOI: 10.1109/MM.2023.3256384
- [15] M. Emani, V. Vishwanath, C. Adams, M. E. Papka, R. Stevens, L. Florescu, S. Jairath, W. Liu, T. Nama, and A. Sujeeth, “Accelerating Scientific Applications With SambaNova Reconfigurable Dataflow Architecture,” *Computing in Science & Engineering*, vol. 23, no. 2, pp. 114–119, 2021. DOI: 10.1109/MCSE.2021.3057203
- [16] Y. Yuan, R. Wang, N. Ranganathan, N. Rao, S. Kumar, P. Lantz, V. Sanjeevan, J. Cabrera, A. Kwatra, R. Sankaran, I. Jeong, and N. S. Kim, “Intel Accelerators Ecosystem: An SoC-Oriented Perspective : Industry Product,” in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2024, pp. 848–862.
- [17] N. Nassif, A. O. Munch, C. L. Molnar, G. Pasdast, S. V. Lyer, Z. Yang, O. Mendoza, M. Huddart, S. Venkataraman, S. Kandula, R. Marom, A. M. Kern, B. Bowhill, D. R. Mulvihill, S. Nimmagadda, V. Kalidindi, J. Krause, M. M. Haq, R. Sharma, and K. Duda, “Sapphire Rapids: The next-generation Intel Xeon scalable processor,” in *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, IEEE, vol. 65, 2022, pp. 44–46.

- [18] R. Kuper, I. Jeong, Y. Yuan, J. Hu, R. Wang, N. Ranganathan, and N. S. Kim, “A Quantitative Analysis and Guideline of Data Streaming Accelerator in Intel 4th Gen Xeon Scalable Processors,” *arXiv preprint arXiv:2305.02480*, 2023.
- [19] S. Ghodrati, S. Kinzer, H. Xu, R. Mahapatra, Y. Kim, B. H. Ahn, D. K. Wang, L. Karthikeyan, A. Yazdanbakhsh, J. Park, N. S. Kim, and H. Esmailzadeh, “Tandem processor: Grappling with emerging operators in neural networks,” in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 1165–1182.
- [20] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, “GPUfs: Integrating a file system with GPUs,” in *Proceedings of the eighteenth international conference on Architectural Support for Programming Languages and Operating Systems*, 2013, pp. 485–498.
- [21] J. Vesely, A. Basu, A. Bhattacharjee, G. H. Loh, M. Oskin, and S. K. Reinhardt, “Generic system calls for GPUs,” in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2018, pp. 843–856.
- [22] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhaharov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein, J. Montgomery, B. Maher, S. Nadathur, J. Olesen, J. Park, A. Rakhov, M. Smelyanskiy, and M. Wang, “Glow: Graph lowering compiler techniques for neural networks,” *arXiv preprint arXiv:1805.00907*, 2018.
- [23] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, “vDNN: Virtualized Deep Neural Networks for scalable, memory-efficient Neural Network design,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2016, pp. 1–13.
- [24] C.-C. Huang, G. Jin, and J. Li, “SwapAdvisor: Pushing deep learning beyond the GPU memory limit via smart swapping,” in *Proceedings of the Twenty-Fifth*

- International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1341–1355.
- [25] A. Plaat, M. van Duijn, N. Van Stein, M. Preuss, P. van der Putten, and K. J. Batenburg, “Agentic Large Language Models, a Survey,” *Journal of Artificial Intelligence Research*, vol. 84, 2025.
 - [26] R. Raj, S. Kundu, I. Vohra, H. Wang, and T. Krishna, *Towards Understanding, Analyzing, and Optimizing Agentic AI Execution: A CPU-Centric Perspective*, 2026. arXiv: 2511.00739 [cs.AI]. [Online]. Available: <https://arxiv.org/abs/2511.00739>
 - [27] Y. Fujii, T. Azumi, N. Nishio, S. Kato, and M. Edahiro, “Data transfer matters for GPU computing,” in *2013 International Conference on Parallel and Distributed Systems*, IEEE, 2013, pp. 275–282.
 - [28] A. Wongpanich, T. Oguntebi, J. B. Paredes, Y. E. Wang, P. M. Phothilimthana, R. Mitra, Z. Zhou, N. Kumar, and V. J. Reddi, “Machine Learning Fleet Efficiency: Analyzing and Optimizing Large-Scale Google TPU Systems with ML Productivity Goodput,” *arXiv preprint arXiv:2502.06982*, 2025.
 - [29] Meta Engineering, *Meta’s Infrastructure Evolution and the Advent of AI*, <https://engineering.fb.com/2025/09/29/data-infrastructure/metas-infrastructure-evolution-and-the-advent-of-ai/>, 2025.
 - [30] OpenAI, *OpenAI, Oracle, and SoftBank Expand Stargate with Five New AI Data Center Sites*, <https://openai.com/index/five-new-stargate-sites/>, 2025.
 - [31] OpenAI and NVIDIA, *OpenAI and NVIDIA Announce Strategic Partnership to Deploy 10 Gigawatts of NVIDIA Systems*, <https://openai.com/index/openai-nvidia-systems-partnership/>, 2025.

- [32] Morgan Stanley Research, *Bridging a \$1.5tr Data Center Financing Gap*,
https://www.morganstanley.com/content/dam/msdotcom/en/assets/pdfs/Research_Bridging-Data-Center-Gap.pdf, 2025.
- [33] McKinsey & Company, *The \$7 Trillion Data Center Build-Out: How Industrials Can Capture Their Share*, <https://www.mckinsey.com/industries/technology-media-and-telecommunications/our-insights/the-7-trillion-dollar-data-center-build-out-how-industrials-can-capture-their-share>, 2026.
- [34] Epoch AI, *Machine Learning Hardware*,
<https://epoch.ai/data/machine-learning-hardware>, Accessed: 2026-06-10, 2026.
- [35] G. Gerogiannis, S. Yesil, D. Lenadora, D. Cao, C. Mendis, and J. Torrellas, “SPADE: A Flexible and Scalable Accelerator for SpMM and SDDMM,” in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA ’23, Orlando, FL, USA: Association for Computing Machinery, 2023. DOI: 10.1145/3579371.3589054 [Online]. Available: <https://doi.org/10.1145/3579371.3589054>
- [36] G. Gerogiannis, S. Eyerman, E. Georganas, W. Heirman, and J. Torrellas, “DECA: A Near-Core LLM Decompression Accelerator Grounded on a 3D Roofline Model,” in *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’25, Association for Computing Machinery, 2025, pp. 184–200, ISBN: 9798400715730. DOI: 10.1145/3725843.3756073 [Online]. Available: <https://doi.org/10.1145/3725843.3756073>
- [37] G. Gerogiannis, S. Eyerman, J. Torrellas, and W. Heirman, “ATX: Accelerator Task Extensions,” in *Proceedings of the 53rd Annual International Symposium on Computer Architecture*, ser. ISCA ’26, Raleigh, NC, USA: Association for Computing Machinery, 2026.

- [38] C. Block, G. Gerogiannis, C. Mendis, A. Azad, and J. Torrellas, “Two-Face: Combining Collective and One-Sided Communication for Efficient Distributed SpMM,” in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS ’24, La Jolla, CA, USA: Association for Computing Machinery, 2024, pp. 1200–1217, ISBN: 9798400703850. DOI: 10.1145/3620665.3640427 [Online]. Available: <https://doi.org/10.1145/3620665.3640427>
- [39] G. Gerogiannis, D. Merkouriadis, C. Block, A. Zulfiqar, F. Tofalos, M. Shahbaz, and J. Torrellas, “NetSparse: In-Network Acceleration of Distributed Sparse Kernels,” in *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’25, Association for Computing Machinery, 2025, pp. 958–974, ISBN: 9798400715730. DOI: 10.1145/3725843.3756076 [Online]. Available: <https://doi.org/10.1145/3725843.3756076>
- [40] G. Gerogiannis, S. Aananthakrishnan, J. Torrellas, and I. Hur, “HotTiles: Accelerating SpMM with Heterogeneous Accelerator Architectures,” in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2024, pp. 1012–1028. DOI: 10.1109/HPCA57654.2024.00081
- [41] G. Gerogiannis and J. Torrellas, “Micro-Armed Bandit: Lightweight & Reusable Reinforcement Learning for Microarchitecture Decision-Making,” in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’23, Toronto, ON, Canada: Association for Computing Machinery, 2023, pp. 698–713, ISBN: 9798400703294. DOI: 10.1145/3613424.3623780 [Online]. Available: <https://doi.org/10.1145/3613424.3623780>
- [42] G. Gerogiannis and J. Torrellas, “Practical Online Reinforcement Learning for Microprocessors With Micro-Armed Bandit,” *IEEE Micro*, vol. 44, no. 4, pp. 80–87, 2024. DOI: 10.1109/MM.2024.3408719

- [43] N. Bell and M. Garland, “Implementing sparse matrix-vector multiplication on throughput-oriented processors,” in *Proceedings of the conference on high performance computing networking, storage and analysis*, 2009, pp. 1–11.
- [44] S. Han, H. Mao, and W. J. Dally, “Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding,” *arXiv preprint arXiv:1510.00149*, 2015.
- [45] A. Buluç, J. Gilbert, and V. B. Shah, “Implementing sparse matrices for graph algorithms,” *Graph Algorithms in the Language of Linear Algebra*, pp. 287–313, 2011.
- [46] L. Page, S. Brin, R. Motwani, and T. Winograd, “The PageRank citation ranking: Bringing order to the web.,” Stanford infolab, Tech. Rep., 1999.
- [47] A. Buluç, “The Ubiquitous Sparse Matrix-Matrix Products,” *arXiv preprint arXiv:2508.04077*, 2025.
- [48] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, C. Yang, J. D. Owens, M. Zalewski, T. Mattson, and J. Moreira, “Mathematical foundations of the GraphBLAS,” in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, 2016, pp. 1–9.
- [49] A. Buluç, T. Mattson, S. McMillan, J. Moreira, and C. Yang, “Design of the GraphBLAS API for C,” in *2017 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, IEEE, 2017, pp. 643–652.
- [50] J. Canny, H. Zhao, B. Jaros, Y. Chen, and J. Mao, “Machine learning at the limit,” in *2015 IEEE International Conference on Big Data (Big Data)*, IEEE, 2015, pp. 233–242.

- [51] I. Nisa, A. Sukumaran-Rajam, S. E. Kurt, C. Hong, and P. Sadayappan, “Sampled Dense Matrix Multiplication for High-Performance Machine Learning,” in *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, IEEE, 2018, pp. 32–41.
- [52] T. N. Kipf and M. Welling, “Semi-Supervised Classification with Graph Convolutional Networks,” *arXiv preprint arXiv:1609.02907*, 2016.
- [53] H. Zhao, B. Jiang, J. F. Canny, and B. Jaros, “SAME but Different: Fast and High Quality Gibbs Parameter Estimation,” in *Proceedings of the 21st ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2015, pp. 1495–1502.
- [54] N. Sundaram and K. Keutzer, “Long term video segmentation through pixel level spectral clustering on GPUs,” in *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, IEEE, 2011, pp. 475–482.
- [55] M. K. Ng and Z. Zhu, “Sparse matrix computation for air quality forecast data assimilation,” *Numerical Algorithms*, vol. 80, no. 3, pp. 687–707, 2019.
- [56] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, *Templates for the solution of algebraic eigenvalue problems: a practical guide*. SIAM, 2000.
- [57] X. Pinel and M. Montagnac, “Block Krylov methods to solve adjoint problems in aerodynamic design optimization,” *AIAA journal*, vol. 51, no. 9, pp. 2183–2191, 2013.
- [58] R. Kannan, G. Ballard, and H. Park, “A high-performance parallel algorithm for nonnegative matrix factorization,” *ACM SIGPLAN Notices*, vol. 51, no. 8, pp. 1–11, 2016.
- [59] A. V. Knyazev, “Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method,” *SIAM journal on scientific computing*, vol. 23, no. 2, pp. 517–541, 2001.

- [60] H. Anzt, S. Tomov, and J. J. Dongarra, “Accelerating the LOBPCG method on GPUs using a blocked sparse matrix vector product,” in *SpringSim (HPS)*, 2015, pp. 75–82.
- [61] A. E. Sarıyüce, E. Saule, K. Kaya, and Ü. V. Çatalyürek, “Regularizing graph centrality computations,” *Journal of Parallel and Distributed Computing*, vol. 76, pp. 106–119, 2015.
- [62] H. M. Aktulga, A. Buluç, S. Williams, and C. Yang, “Optimizing Sparse Matrix-Multiple Vectors Multiplication for Nuclear Configuration Interaction Calculations,” in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 1213–1222. DOI: 10.1109/IPDPS.2014.125
- [63] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, “Graph attention networks,” *arXiv preprint arXiv:1710.10903*, 2017.
- [64] W. L. Hamilton, R. Ying, and J. Leskovec, “Inductive representation learning on large graphs,” in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017, pp. 1025–1035.
- [65] E. Chien, J. Peng, P. Li, and O. Milenkovic, “Adaptive Universal Generalized PageRank Graph Neural Network,” *arXiv preprint arXiv:2006.07988*, 2020.
- [66] G. Huang, G. Dai, Y. Wang, and H. Yang, “GE-SpMM: General-Purpose Sparse Matrix-Matrix Multiplication on GPUs for Graph Neural Networks,” in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–12. DOI: 10.1109/SC41405.2020.00076
- [67] Y. Hu, Z. Ye, M. Wang, J. Yu, D. Zheng, M. Li, Z. Zhang, Z. Zhang, and Y. Wang, “FeatGraph: A Flexible and Efficient Backend for Graph Neural Network Systems,” in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–13. DOI: 10.1109/SC41405.2020.00075

- [68] R. Fan, W. Wang, and X. Chu, “Fast Sparse GPU Kernels for Accelerated Training of Graph Neural Networks,” in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023, pp. 501–511. DOI: 10.1109/IPDPS54959.2023.00057
- [69] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang, “Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks,” *arXiv preprint arXiv:1909.01315*, 2019.
- [70] M. K. Rahman, M. H. Sujon, and A. Azad, “FusedMM: A unified SDDMM-SpMM kernel for graph embedding and graph neural networks,” in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2021, pp. 256–266.
- [71] F. G. Gustavson, “Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition,” *ACM Trans. Math. Softw.*, vol. 4, no. 3, pp. 250–269, 1978, ISSN: 0098-3500.
- [72] Y. Saad, *Iterative methods for sparse linear systems*. SIAM, 2003.
- [73] T. A. Davis and Y. Hu, “The University of Florida sparse matrix collection,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.
- [74] V. Bharadwaj, A. Buluç, and J. Demmel, “Distributed-memory sparse kernels for machine learning,” in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2022, pp. 47–58.
- [75] O. Selvitopi, B. Brock, I. Nisa, A. Tripathy, K. Yelick, and A. Buluç, “Distributed-memory parallel algorithms for sparse times tall-skinny-dense matrix multiplication,” in *Proceedings of the ACM International Conference on Supercomputing*, 2021, pp. 431–442.

- [76] E. Chan, R. Van De Geijn, W. Gropp, and R. Thakur, “Collective communication on architectures that support simultaneous communication over multiple links,” in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2006, pp. 2–11.
- [77] R. Thakur, R. Rabenseifner, and W. Gropp, “Optimization of collective communication operations in MPICH,” *The International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.
- [78] A. Kalia, M. Kaminsky, and D. G. Andersen, “Design guidelines for high performance rdma systems,” in *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC ’16, Denver, CO, USA: USENIX Association, 2016, pp. 437–450, ISBN: 9781931971300.
- [79] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, “ExTensor: An Accelerator for Sparse Tensor Algebra,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’52, Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 319–333. DOI: 10.1145/3352460.3358275 [Online]. Available: <https://doi.org/10.1145/3352460.3358275>
- [80] N. Srivastava, H. Jin, S. Smith, H. Rong, D. Albonesi, and Z. Zhang, “Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations,” in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2020, pp. 689–702.
- [81] L. Song, Y. Chi, A. Sohrabizadeh, Y.-k. Choi, J. Lau, and J. Cong, “Sextans: A Streaming Accelerator for General-Purpose Sparse-Matrix Dense-Matrix Multiplication,” in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2022, pp. 65–77.

- [82] T. Allen and R. Ge, “In-depth analyses of unified virtual memory system for GPU accelerated computing,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–15.
- [83] J. Veselý, A. Basu, M. Oskin, G. H. Loh, and A. Bhattacharjee, “Observations and opportunities in architecting shared virtual memory for heterogeneous systems,” in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, 2016, pp. 161–171.
- [84] Y.-C. Lin, B. Zhang, and V. Prasanna, “HP-GNN: Generating high throughput GNN training implementation on CPU-FPGA heterogeneous platform,” in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2022, pp. 123–133.
- [85] D. Zheng, X. Song, C. Yang, D. LaSalle, and G. Karypis, “Distributed hybrid CPU and GPU training for Graph Neural Networks on billion-scale heterogeneous graphs,” in *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2022, pp. 4582–4591.
- [86] B. Zhang, S. R. Kuppannagari, R. Kannan, and V. Prasanna, “Efficient neighbor-sampling-based GNN training on CPU-FPGA heterogeneous platform,” in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, IEEE, 2021, pp. 1–7.
- [87] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan, “Adaptive Sparse Tiling for Sparse Matrix Multiplication,” in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP ’19, Washington, District of Columbia: Association for Computing Machinery, 2019, pp. 300–314. DOI: 10.1145/3293883.3295712
- [88] S. Yesil, J. E. Moreira, and J. Torrellas, “Dense Dynamic Blocks: Optimizing SpMM for Processors with Vector and Matrix Units Using Machine Learning Techniques,”

- in *Proceedings of the 36th ACM International Conference on Supercomputing*, ser. ICS '22, Virtual Event: Association for Computing Machinery, 2022. DOI: 10.1145/3524059.3532369 [Online]. Available: <https://doi.org/10.1145/3524059.3532369>
- [89] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, “Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks,” *IEEE journal of solid-state circuits*, vol. 52, no. 1, pp. 127–138, 2016.
- [90] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, “Optimizing FPGA-Based Accelerator Design for Deep Convolutional Neural Networks,” in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15, Monterey, California, USA: Association for Computing Machinery, 2015, pp. 161–170. DOI: 10.1145/2684746.2689060 [Online]. Available: <https://doi.org/10.1145/2684746.2689060>
- [91] M. Adiletta, J. J. Tithi, E.-I. Farsarakis, G. Gerogiannis, R. Adolf, R. Benke, S. Kashyap, S. Hsia, K. Lakhotia, F. Petrini, G.-Y. Wei, and D. Brooks, “Characterizing the Scalability of Graph Convolutional Networks on Intel® PIUMA,” in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023.
- [92] Z. Gong, H. Ji, Y. Yao, C. W. Fletcher, C. J. Hughes, and J. Torrellas, “Graphite: Optimizing Graph Neural Networks on CPUs through Cooperative Software-Hardware Techniques,” in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22, New York, New York: Association for Computing Machinery, 2022, pp. 916–931. DOI: 10.1145/3470496.3527403 [Online]. Available: <https://doi.org/10.1145/3470496.3527403>

- [93] Intel, *Intel 64 and IA-32 Architectures Software Developer’s Manual*, Apr. 2022.
[Online]. Available: <https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf>
- [94] WikiChip, *Sunny Cove - Microarchitectures - Intel*, WikiChip, 2023. [Online]. Available:
https://en.wikichip.org/wiki/intel/microarchitectures/sunny_cove
- [95] Intel, *Intel Xeon Gold 6348 Processor 42M Cache 2.60 GHz Product Specifications*, Intel, 2021. [Online]. Available:
<https://www.intel.com/content/www/us/en/products/sku/212456/intel-xeon-gold-6348-processor-42m-cache-2-60-ghz/specifications.html>
- [96] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis, and B. Jacob, “The Structural Simulation Toolkit,” *SIGMETRICS Perform. Eval. Rev.*, vol. 38, no. 4, pp. 37–42, Mar. 2011. DOI: 10.1145/1964218.1964225 [Online]. Available:
<https://doi.org/10.1145/1964218.1964225>
- [97] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, “DRAMsim3: a cycle-accurate, thermal-capable DRAM simulator,” *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 106–109, 2020.
- [98] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, “Intel math kernel library,” in *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 2014, pp. 167–188.
- [99] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, “The tensor algebra compiler,” *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–29, 2017.

- [100] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, “cuSPARSE library,” in *GPU Technology Conference*, 2010.
- [101] G. Huang, G. Dai, Y. Wang, Y. Ding, and Y. Xie, *Efficient Sparse Matrix Kernels Based on Adaptive Workload-Balancing and Parallel-Reduction*, 2021. arXiv: 2106.16064 [cs.DC].
- [102] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, “CACTI 7: New tools for interconnect exploration in innovative off-chip memories,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, pp. 1–25, 2017.
- [103] S. Galal and M. Horowitz, “Energy-efficient floating-point unit design,” *IEEE Transactions on computers*, vol. 60, no. 7, pp. 913–922, 2010.
- [104] A. Stillmaker and B. Baas, “Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm,” *Integration*, vol. 58, pp. 74–81, 2017.
- [105] D. Schor, *Intel launches 3rd Gen Ice Lake Xeon Scalable*, WikiChip Fuse, May 2021. [Online]. Available: <https://fuse.wikichip.org/news/4734/intel-launches-3rd-gen-ice-lake-xeon-scalable/>
- [106] S. Aananthakrishnan, S. Abedin, V. Cavé, F. Checconi, K. D. Bois, S. Eyerman, J. B. Fryman, W. Heirman, J. Howard, I. Hur, S. Jain, M. M. Landowski, K. Ma, J. A. Nelson, R. Pawlowski, F. Petrini, S. Szkoda, S. Tayal, J. J. Tithi, and Y. Vandriessche, “The Intel Programmable and Integrated Unified Memory Architecture Graph Analytics Processor,” *IEEE Micro*, vol. 43, no. 5, pp. 78–87, 2023. DOI: 10.1109/MM.2023.3295848
- [107] S. Liang, Y. Wang, C. Liu, L. He, L. Huawei, D. Xu, and X. Li, “EnGN: A High-Throughput and Energy-Efficient Accelerator for Large Graph Neural Networks,” *IEEE Transactions on Computers*, 2020.

- [108] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, “HyGCN: A GCN accelerator with hybrid architecture,” in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2020, pp. 15–29.
- [109] T. Geng, A. Li, R. Shi, C. Wu, T. Wang, Y. Li, P. Haghi, A. Tumeo, S. Che, S. Reinhardt, and M. C. Herbordt, “AWB-GCN: A Graph Convolutional Network Accelerator with Runtime Workload Rebalancing,” in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 922–936. DOI: 10.1109/MICRO50266.2020.00079
- [110] A. Auten, M. Tomei, and R. Kumar, “Hardware Acceleration of Graph Neural Networks,” in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, IEEE, 2020, pp. 1–6.
- [111] K. Kinningham, P. Levis, and C. Ré, “GRIP: A Graph Neural Network Accelerator Architecture,” *IEEE Transactions on Computers*, 2022.
- [112] R. Garg, E. Qin, F. Muñoz-Martínez, R. Guirado, A. Jain, S. Abadal, J. L. Abellán, M. E. Acacio, E. Alarcón, S. Rajamanickam, and T. Krishna, “Understanding the Design-Space of Sparse/Dense Multiphase GNN Dataflows on Spatial Accelerators,” in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2022, pp. 571–582. DOI: 10.1109/IPDPS53621.2022.00062
- [113] Y. Wang, B. Feng, G. Li, S. Li, L. Deng, Y. Xie, and Y. Ding, “GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs,” in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, USENIX Association, Jul. 2021, pp. 515–531. [Online]. Available: <https://www.usenix.org/conference/osdi21/presentation/wang-yuke>
- [114] K. Huang, J. Zhai, Z. Zheng, Y. Yi, and X. Shen, “Understanding and Bridging the Gaps in Current GNN Performance Optimizations,” in *Proceedings of the 26th ACM*

- SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '21, Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 119–132. DOI: 10.1145/3437801.3441585 [Online]. Available: <https://doi.org/10.1145/3437801.3441585>
- [115] L. Ma, Z. Yang, Y. Miao, J. Xue, M. Wu, L. Zhou, and Y. Dai, “NeuGraph: Parallel deep neural network computation on large graphs,” in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 443–458.
 - [116] M. Y. Wang, “Deep Graph Library: towards efficient and scalable deep learning on graphs,” *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019. [Online]. Available: <https://par.nsf.gov/biblio/10311680>
 - [117] Y. Wang, B. Feng, and Y. Ding, “TC-GNN: Accelerating Sparse Graph Neural Network Computation Via Dense Tensor Core on GPUs,” *arXiv preprint arXiv:2112.02052*, 2021.
 - [118] J. Li, A. Louri, A. Karanth, and R. Bunesu, “GCNAX: A Flexible and Energy-efficient Accelerator for Graph Convolutional Neural Networks,” in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, 2021, pp. 775–788.
 - [119] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, “Sigma: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training,” in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2020, pp. 58–70.
 - [120] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, “SCNN: An accelerator for compressed-sparse convolutional neural networks,” *ACM SIGARCH computer architecture news*, vol. 45, no. 2, 2017.

- [121] X. He, S. Pal, A. Amarnath, S. Feng, D.-H. Park, A. Rovinski, H. Ye, Y. Chen, R. Dreslinski, and T. Mudge, “Sparse-TPU: Adapting systolic arrays for sparse matrices,” in *Proceedings of the 34th ACM international conference on supercomputing*, 2020, pp. 1–12.
- [122] Z. Gong, H. Ji, C. W. Fletcher, C. J. Hughes, S. Baghsorkhi, and J. Torrellas, “SAVE: Sparsity-Aware Vector Engine for Accelerating DNN Training and Inference on CPUs,” in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2020, pp. 796–810.
- [123] G. Jeong, S. Damani, A. R. Bambhaniya, E. Qin, C. J. Hughes, S. Subramoney, H. Kim, and T. Krishna, “VEGETA: Vertically-Integrated Extensions for Sparse/Dense GEMM Tile Acceleration on CPUs,” in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, 2023, pp. 259–272.
- [124] A. Cabrera, S. Hitefield, J. Kim, S. Lee, N. R. Miniskar, and J. S. Vetter, “Toward performance portable programming for heterogeneous systems on a chip: A case study with Qualcomm Snapdragon SoC,” in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, 2021, pp. 1–7.
- [125] NVIDIA, *NVIDIA Tesla V100 whitepaper*, NVIDIA, 2017. [Online]. Available: <https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>
- [126] NVIDIA, *NVIDIA Grace Hopper superchip architecture whitepaper*, NVIDIA, 2022. [Online]. Available: <https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper>
- [127] A. C. Elster and T. A. Haugdahl, “NVIDIA Hopper GPU and Grace CPU highlights,” *Computing in Science & Engineering*, vol. 24, no. 2, pp. 95–100, 2022.

- [128] R. Smith, *Intel showcases Sapphire Rapids Plus HBM Xeon Performance at ISC 2022*, RSS, May 2022. [Online]. Available: <https://www.anandtech.com/show/17422/intel-showcases-sapphire-rapids-plus-hbm-xeon-performance-isc-2022>
- [129] V. Dadu, S. Liu, and T. Nowatzki, “PolyGraph: Exposing the Value of Flexibility for Graph Processing Accelerators,” in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2021, pp. 595–608.
- [130] S. Feng, J. Sun, S. Pal, X. He, K. Kaszyk, D.-h. Park, M. Morton, T. Mudge, M. Cole, M. O’Boyle, C. Chakrabarti, and R. Dreslinski, “CoSPARSE: A Software and Hardware Reconfigurable SpMV Framework for Graph Analytics,” in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 949–954. DOI: 10.1109/DAC18074.2021.9586114
- [131] F. Muñoz-Martínez, R. Garg, M. Pellauer, J. L. Abellán, M. E. Acacio, and T. Krishna, “Flexagon: A Multi-Dataflow Sparse-Sparse Matrix Multiplication Accelerator for Efficient DNN Processing,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023, Vancouver, BC, Canada: Association for Computing Machinery, 2023, pp. 252–265. DOI: 10.1145/3582016.3582069 [Online]. Available: <https://doi.org/10.1145/3582016.3582069>
- [132] Z. Li, J. Li, T. Chen, D. Niu, H. Zheng, Y. Xie, and M. Gao, “Spada: Accelerating Sparse Matrix Multiplication with Adaptive Dataflow,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023, Vancouver, BC, Canada: Association for Computing Machinery, 2023, pp. 747–761. DOI:

- 10.1145/3575693.3575706 [Online]. Available:
<https://doi.org/10.1145/3575693.3575706>
- [133] E. Qin, G. Jeong, W. Won, S.-C. Kao, H. Kwon, S. Srinivasan, D. Das, G. E. Moon, S. Rajamanickam, and T. Krishna, “Extending sparse tensor accelerators to support multiple compression formats,” in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2021, pp. 1014–1024.
 - [134] B. Asgari, R. Hadidi, T. Krishna, H. Kim, and S. Yalamanchili, “ALRESCHA: A lightweight reconfigurable sparse-computation accelerator,” in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2020, pp. 249–260.
 - [135] E. Qin, R. Garg, A. Bambhaniya, M. Pellauer, A. Parashar, S. Rajamanickam, C. Hao, and T. Krishna, “Enabling Flexibility for Sparse Tensor Acceleration via Heterogeneity,” *arXiv preprint arXiv:2201.08916*, 2022.
 - [136] S. Pal, S. Feng, D.-h. Park, S. Kim, A. Amarnath, C.-S. Yang, X. He, J. Beaumont, K. May, Y. Xiong, K. Kaszyk, J. M. Morton, J. Sun, M. O’Boyle, M. Cole, C. Chakrabarti, D. Blaauw, H.-S. Kim, T. Mudge, and R. Dreslinski, “Transmuter: Bridging the Efficiency Gap Using Memory and Dataflow Reconfiguration,” in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT ’20, Virtual Event, GA, USA: Association for Computing Machinery, 2020, pp. 175–190. DOI: 10.1145/3410463.3414627 [Online]. Available: <https://doi.org/10.1145/3410463.3414627>
 - [137] A. Mehrabi, D. Lee, N. Chatterjee, D. J. Sorin, B. C. Lee, and M. O’Connor, “Learning sparse matrix row permutations for efficient SpMM on GPU architectures,” in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, 2021, pp. 48–58.

- [138] V. Balaji and B. Lucia, “When is graph reordering an optimization? Studying the effect of lightweight graph reordering across applications and input graphs,” in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, IEEE, 2018, pp. 203–214.
- [139] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura, “Rabbit Order: Just-in-time parallel reordering for fast graph analysis,” in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2016, pp. 22–31.
- [140] S. Yesil, A. Heidarshenas, A. Morrison, and J. Torrellas, “WISE: Predicting the Performance of Sparse Matrix Vector Multiplication with Machine Learning,” in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP ’23, Montreal, QC, Canada: Association for Computing Machinery, 2023, pp. 329–341. DOI: 10.1145/3572848.3577506 [Online]. Available: <https://doi.org/10.1145/3572848.3577506>
- [141] P. Zhou, L. Wang, Z. Liu, Y. Hao, P. Hui, S. Tarkoma, and J. Kangasharju, “A survey on generative AI and LLM for video generation, understanding, and streaming,” *arXiv preprint arXiv:2404.16038*, 2024.
- [142] H. Zhang, X. Liu, and J. Zhang, “Summit: Iterative text summarization via ChatGPT,” *arXiv preprint arXiv:2305.14835*, 2023.
- [143] D. Kalla, N. Smith, F. Samaah, and S. Kuraku, “Study and analysis of Chat GPT and its impact on different fields of study,” *International journal of innovative science and research technology*, vol. 8, no. 3, 2023.
- [144] B. Yao, M. Jiang, D. Yang, and J. Hu, “Empowering LLM-based machine translation with cultural awareness,” *arXiv preprint arXiv:2305.14328*, 2023.

- [145] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.
- [146] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, R. Avila, I. Babuschkin, S. Balaji, V. Balcom, P. Baltescu, H.-i. Bao, M. Bavarian, J. Belgum, I. Bello, J. Berdine, G. Bernadett-Shapiro, C. Berner, L. Bogdonoff, O. Boiko, M. Boyd, A.-L. Brakman, G. Brockman, T. Brooks, M. Brundage, K. Button, T. Cai, R. Campbell, A. Cann, B. Carey, C. Carlson, R. Carmichael, B. Chan, C. Chang, F. Chantzis, D. Chen, S. Chen, R. Chen, J. Chen, M. Chen, B. Chess, C. Cho, C. Chu, H. W. Chung, D. Cummings, J. Currier, Y. Dai, C. Decareaux, T. Degry, N. Deutsch, D. Deville, A. Dhar, D. Dohan, S. Dowling, S. Dunning, A. Ecoffet, A. Eleti, T. Eloundou, D. Farhi, L. Fedus, N. Felix, S. P. Fishman, J. Forte, I.-a. Fulford, L. Gao, E. Georges, C. Gibson, V. Goel, T. Gogineni, G. Goh, R. Gontijo-Lopes, J. Gordon, M. Grafstein, S. Gray, R. Greene, J. Gross, S. S. Gu, Y. Guo, C. Hallacy, J. Han, J. Harris, Y. He, M. Heaton, J. Heidecke, C. Hesse, A. Hickey, W. Hickey, P. Hoeschele, B. Houghton, K. Hsu, S. Hu, X. Hu, J. Huizinga, S. Jain, S. Jain, et al., “GPT-4 Technical Report,” *arXiv preprint arXiv:2303.08774*, 2023.
- [147] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, Y. Du, C. Yang, Y. Chen, Z. Chen, J. Jiang, R. Ren, Y. Li, X. Tang, Z. Liu, P. Liu, J.-Y. Nie, and J.-R. Wen, “A survey of large language models,” *arXiv preprint arXiv:2303.18223*, 2023.
- [148] Z. Yuan, Y. Shang, Y. Zhou, Z. Dong, Z. Zhou, C. Xue, B. Wu, Z. Li, Q. Gu, Y. J. Lee, Y. Yan, B. Chen, G. Sun, and K. Keutzer, “LLM Inference Unveiled: Survey and Roofline Model Insights,” *arXiv preprint arXiv:2402.16363*, 2024.

- [149] A. Biswas and S. Kottapalli, “Next-Gen Intel Xeon CPU-Sapphire Rapids,” in *Hot Chips*, vol. 33, 2021.
- [150] Intel, *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 2024.
- [151] S. Markidis, S. W. Der Chien, E. Laure, I. B. Peng, and J. S. Vetter, “NVIDIA Tensor Core programmability, performance & precision,” in *2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, IEEE, 2018, pp. 522–531.
- [152] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, “Llama 2: Open foundation and fine-tuned chat models,” *arXiv preprint arXiv:2307.09288*, 2023.
- [153] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang, “Pruning and quantization for deep neural network acceleration: A survey,” *Neurocomputing*, vol. 461, pp. 370–403, 2021.
- [154] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, “Model compression and hardware acceleration for neural networks: A comprehensive survey,” *Proceedings of the IEEE*, vol. 108, no. 4, pp. 485–532, 2020.

- [155] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, “A survey of quantization methods for efficient neural network inference,” in *Low-Power Computer Vision*. Chapman and Hall/CRC, 2022, pp. 291–326.
- [156] T. Hoefer, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste, “Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks,” *Journal of Machine Learning Research*, vol. 22, no. 241, pp. 1–124, 2021.
- [157] H. Xia, Z. Zheng, Y. Li, D. Zhuang, Z. Zhou, X. Qiu, Y. Li, W. Lin, and S. L. Song, “Flash-LLM: Enabling Cost-Effective and Highly-Efficient Large Generative Model Inference with Unstructured Sparsity,” *Proceedings of the VLDB Endowment*, vol. 17, no. 2, pp. 211–224, 2023.
- [158] X. Zhu, J. Li, Y. Liu, C. Ma, and W. Wang, “A survey on model compression for large language models,” *arXiv preprint arXiv:2308.07633*, 2023.
- [159] D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen, J. Yang, J. Park, A. Heinecke, E. Georganas, S. Srinivasan, A. Kundu, M. Smelyanskiy, B. Kaul, and P. Dubey, “A study of BFLOAT16 for deep learning training,” *arXiv preprint arXiv:1905.12322*, 2019.
- [160] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst, “LIBXSMM: accelerating small matrix multiplications by runtime code generation,” in *SC’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2016, pp. 981–991.
- [161] S. Williams, A. Waterman, and D. Patterson, “Roofline: an insightful visual performance model for multicore architectures,” *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.
- [162] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, T. Mihaylov, M. Ott, S. Shleifer, K. Shuster, D. Simig,

- P. S. Koura, A. Sridhar, T. Wang, and L. Zettlemoyer, “OPT: Open pre-trained transformer language models,” *arXiv preprint arXiv:2205.01068*, 2022.
- [163] P. Patel, E. Choukse, C. Zhang, Í. Goiri, A. Shah, S. Maleki, and R. Bianchini, “Splitwise: Efficient generative LLM inference using phase splitting,” *arXiv preprint arXiv:2311.18677*, 2023.
- [164] Q. Su, C. Giannoula, and G. Pekhimenko, “The synergy of speculative decoding and batching in serving large language models,” *arXiv preprint arXiv:2310.18813*, 2023.
- [165] M. Siracusa, V. Soria-Pardos, F. Sgherzi, J. Randall, D. J. Joseph, M. Moretó Planas, and A. Armejach, “A Tensor Marshaling Unit for Sparse Tensor Algebra on General-Purpose Processors,” in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 1332–1346.
- [166] M. Orenes-Vera, A. Manocha, J. Balkind, F. Gao, J. L. Aragón, D. Wentzlaff, and M. Martonosi, “Tiny but mighty: designing and realizing scalable latency tolerance for manycore SoCs,” in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 817–830.
- [167] S. B. Harma, A. Chakraborty, E. Kostenok, D. Mishin, D. Ha, B. Falsafi, M. Jaggi, M. Liu, Y. Oh, S. Subramanian, and A. Yazdanbakhsh, “Effective Interplay between Sparsity and Quantization: From Theory to Practice,” *arXiv preprint arXiv:2405.20935*, 2024.
- [168] J. Lin, J. Tang, H. Tang, S. Yang, W.-M. Chen, W.-C. Wang, G. Xiao, X. Dang, C. Gan, and S. Han, “AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration,” *arXiv preprint arXiv:2306.00978*, 2023.
- [169] Y. Zhao, C.-Y. Lin, K. Zhu, Z. Ye, L. Chen, S. Zheng, L. Ceze, A. Krishnamurthy, T. Chen, and B. Kasikci, “Atom: Low-bit quantization for efficient and accurate LLM serving,” *Proceedings of Machine Learning and Systems*, vol. 6, pp. 196–209, 2024.

- [170] J. Kim, J. H. Lee, S. Kim, J. Park, K. M. Yoo, S. J. Kwon, and D. Lee, “Memory-Efficient Fine-Tuning of Compressed Large Language Models via sub-4-bit Integer Quantization,” *Advances in Neural Information Processing Systems*, vol. 36, 2024.
- [171] X. Wei, Y. Zhang, Y. Li, X. Zhang, R. Gong, J. Guo, and X. Liu, “Outlier suppression+: Accurate quantization of large language models by equivalent and optimal shifting and scaling,” *arXiv preprint arXiv:2304.09145*, 2023.
- [172] R. Bitar Darvish, G. Nitin, S. Tom, M. Ankit, H. Kyung-Nam, Z. Ritchie, H. Mathew, K. Jasmine, C. Eric, Y. Yuan, S. Michael, W. Ralph, B. Ian, S. Nigel, M. Jelena, B. John, D. Pradeep, C. Marius, H. Alexander, R. Andres, L. Martin, D. Summer, N. Maxim, M. Paulius, S. Michael, and V. Colin, *OCP Microscaling Formats (MX) Specification*, 2023. [Online]. Available: <https://www.opencompute.org/documents/ocp-microscaling-formats-mx-v1-0-spec-final-pdf>
- [173] Y. LeCun, J. Denker, and S. Solla, “Optimal brain damage,” *Advances in neural information processing systems*, vol. 2, 1989.
- [174] D. Blalock, J. J. Gonzalez Ortiz, J. Frankle, and J. Guttag, “What is the state of neural network pruning?” *Proceedings of machine learning and systems*, vol. 2, pp. 129–146, 2020.
- [175] L. Liu, S. Zhang, Z. Kuang, A. Zhou, J.-H. Xue, X. Wang, Y. Chen, W. Yang, Q. Liao, and W. Zhang, “Group fisher pruning for practical network compression,” in *International Conference on Machine Learning*, PMLR, 2021, pp. 7021–7032.
- [176] E. Frantar and D. Alistarh, “SparseGPT: Massive Language Models Can Be Accurately Pruned in One-shot,” in *International Conference on Machine Learning*, PMLR, 2023, pp. 10 323–10 337.

- [177] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
- [178] A. Peste, E. Iofinova, A. Vladu, and D. Alistarh, “AC/DC: Alternating Compressed/DeCompressed Training of Deep Neural Networks,” *Advances in neural information processing systems*, vol. 34, pp. 8557–8570, 2021.
- [179] Intel, *Accelerate Artificial Intelligence (AI) Workloads with Intel Advanced Matrix Extensions (Intel AMX)*, 2022. [Online]. Available: <https://www.intel.com/content/dam/www/central-libraries/us/en/documents/2022-12/accelerate-ai-with-amx-sb.pdf>
- [180] F. Wilkinson and S. McIntosh-Smith, “An Initial Evaluation of Arm’s Scalable Matrix Extension,” in *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, IEEE, 2022, pp. 135–140.
- [181] J. P. de Carvalho, J. E. Moreira, and J. N. Amaral, “Compiling for the IBM matrix engine for enterprise workloads,” *IEEE Micro*, vol. 42, no. 5, pp. 34–40, 2022.
- [182] P. Bhat, J. Moreira, and S. K. Sadasivam, “Matrix-multiply Assist Best Practices Guide,” IBM, Tech. Rep., 2021. [Online]. Available: <https://www.redbooks.ibm.com>, Tech. Rep., 2021.
- [183] S. Palacharla, N. P. Jouppi, and J. E. Smith, “Complexity-effective superscalar processors,” in *Proceedings of the 24th annual international symposium on Computer architecture*, 1997, pp. 206–218.
- [184] P. Alcorn. “Intel Details Sierra Forest and Granite Rapids Architectures, Xeon Roadmap.” Accessed: 2025-09-04. [Online]. Available: <https://www.tomshardware.com/news/intel-details-sierra-forest-and-granite-rapids-architecture-xeon-roadmap>

- [185] Intel, *Intel[®] 64 and IA-32 Architectures Optimization Reference Manual*, Chapter 20: Intel AMX, Section 20.17.2 – Intel[®] Hyper-Threading Technology. Available at <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>, Intel, 2023.
- [186] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, “An Evaluation of High-Level Mechanistic Core Models,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 3, 28:1–28:25, Aug. 2014.
- [187] E. Georganas, D. Kalamkar, K. Voronin, A. Kundu, A. Noack, H. Pabst, A. Breuer, and A. Heinecke, “Harnessing Deep Learning and HPC Kernels via High-Level Loop and Tensor Abstractions on CPU Architectures,” *arXiv preprint arXiv:2304.12576*, 2023.
- [188] E. Georganas, D. Kalamkar, S. Avancha, M. Adelman, C. Anderson, A. Breuer, J. Bruestle, N. Chaudhary, A. Kundu, D. Kutnick, F. Laub, V. Md, S. Misra, R. Mohanty, H. Pabst, B. Ziv, and A. Heinecke, “Tensor processing primitives: A programming abstraction for efficiency and portability in deep learning workloads,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.
- [189] C. Cakir, R. Ho, J. Lexau, and K. Mai, “Modeling and design of high-radix on-chip crossbar switches,” in *Proceedings of the 9th International Symposium on Networks-on-Chip*, 2015, pp. 1–8.
- [190] H. Zhang, D. Chen, and S.-B. Ko, “New flexible multiple-precision multiply-accumulate unit for deep neural network training and inference,” *IEEE Transactions on Computers*, vol. 69, no. 1, pp. 26–38, 2019.
- [191] R. Roy, J. Raiman, N. Kant, I. Elkin, R. Kirby, M. Siu, S. Oberman, S. Godil, and B. Catanzaro, “PrefixRL: Optimization of Parallel Prefix Circuits Using Deep

- Reinforcement Learning,” in *Proceedings of the 58th Annual ACM/IEEE Design Automation Conference*, IEEE Press, 2022, pp. 853–858, ISBN: 9781665432740.
- [Online]. Available: <https://doi.org/10.1109/DAC18074.2021.9586094>
- [192] Wikipedia, *Sapphire Rapids Die Configurations*, Wikipedia, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Sapphire_Rapids#Die_configurations
- [193] X. Dai, *GPU-Benchmarks-on-LLM-Inference*, 2025. [Online]. Available: <https://github.com/XiongjieDai/GPU-Benchmarks-on-LLM-Inference>
- [194] W. Luo, R. Fan, Z. Li, D. Du, Q. Wang, and X. Chu, “Benchmarking and dissecting the NVIDIA Hopper GPU architecture,” *arXiv preprint arXiv:2402.13499*, 2024.
- [195] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker, “The ARM Scalable Vector Extension,” *IEEE micro*, vol. 37, no. 2, pp. 26–39, 2017.
- [196] H. Fan, H. Zhou, G. Huang, P. Raman, X. Fu, G. Gupta, D. Ram, Y. Wang, and J. Huan, “HLAT: High-quality Large Language Model Pre-trained on AWS Trainium,” *arXiv preprint arXiv:2404.10630*, 2024.
- [197] G. Jeong, E. Qin, A. Samajdar, C. J. Hughes, S. Subramoney, H. Kim, and T. Krishna, “RASA: Efficient Register-Aware Systolic Array Matrix Engine for CPU,” in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, IEEE, 2021, pp. 253–258.
- [198] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, “Cambricon-X: An accelerator for sparse neural networks,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2016, pp. 1–12.

- [199] L. Lu, J. Xie, R. Huang, J. Zhang, W. Lin, and Y. Liang, “An efficient hardware accelerator for sparse convolutional neural networks on FPGAs,” in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2019, pp. 17–25.
- [200] X. Chen, Y. Chen, F. Cheng, H. Tan, B. He, and W.-F. Wong, “ReGraph: Scaling graph processing on HBM-enabled FPGAs with heterogeneous pipelines,” in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2022, pp. 1342–1358.
- [201] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. Vijaykumar, “SparTen: A sparse tensor accelerator for convolutional neural networks,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 151–165.
- [202] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, “EIE: Efficient inference engine on compressed deep neural network,” *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016.
- [203] Z. Zhu, F. Li, G. Li, Z. Liu, Z. Mo, Q. Hu, X. Liang, and J. Cheng, “MEGA: A Memory-Efficient GNN Accelerator Exploiting Degree-Aware Mixed-Precision Quantization,” in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, 2024, pp. 124–138.
- [204] S. Ryu, H. Kim, W. Yi, E. Kim, Y. Kim, T. Kim, and J.-J. Kim, “Bitblade: Energy-efficient variable bit-precision hardware accelerator for quantized neural networks,” *IEEE Journal of Solid-State Circuits*, vol. 57, no. 6, pp. 1924–1935, 2022.
- [205] J. Jang, Y. Kim, J. Lee, and J.-J. Kim, “FIGNA: Integer Unit-Based Accelerator Design for FP-INT GEMM Preserving Numerical Accuracy,” in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, 2024, pp. 760–773.

- [206] H. Wang, Z. Zhang, and S. Han, “Spatten: Efficient sparse attention architecture with cascade token and head pruning,” in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, 2021, pp. 97–110.
- [207] C. Kachris, “A Survey on Hardware Accelerators for Large Language Models,” *arXiv preprint arXiv:2401.09890*, 2024.
- [208] L. Lu, Y. Jin, H. Bi, Z. Luo, P. Li, T. Wang, and Y. Liang, “Sanger: A Co-Design Framework for Enabling Sparse Attention using Reconfigurable Architecture,” in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 977–991.
- [209] T. J. Ham, Y. Lee, S. H. Seo, S. Kim, H. Choi, S. J. Jung, and J. W. Lee, “ELSA: Hardware-software co-design for efficient, lightweight self-attention mechanism in neural networks,” in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2021, pp. 692–705.
- [210] T. J. Ham, S. J. Jung, S. Kim, Y. H. Oh, Y. Park, Y. Song, J.-H. Park, S. Lee, K. Park, J. W. Lee, and D.-K. Jeong, “A[^] 3: Accelerating attention mechanisms in neural networks with approximation,” in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2020, pp. 328–341.
- [211] C. Peltekis, V. Titopoulos, C. Nicopoulos, and G. Dimitrakopoulos, “DeMM: A Decoupled Matrix Multiplication Engine Supporting Relaxed Structured Sparsity,” *IEEE Computer Architecture Letters*, 2024.
- [212] Y. Yang, J. S. Emer, and D. Sanchez, “Trapezoid: A Versatile Accelerator for Dense and Sparse Matrix Multiplications,” in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2024, pp. 931–945.
- [213] M. Hill and V. J. Reddi, “Gables: A roofline model for mobile SOCs,” in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2019, pp. 317–330.

- [214] J. Stojkovic, E. Choukse, C. Zhang, I. Goiri, and J. Torrellas, “Towards Greener LLMs: Bringing Energy-efficiency to the Forefront of LLM Inference,” *arXiv preprint arXiv:2403.20306*, 2024.
- [215] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, “Survey and benchmarking of machine learning accelerators,” in *2019 IEEE high performance extreme computing conference (HPEC)*, IEEE, 2019, pp. 1–9.
- [216] R. Weber, A. Gothandaraman, R. J. Hinde, and G. D. Peterson, “Comparing hardware accelerators in scientific applications: A case study,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 58–68, 2010.
- [217] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, “PTask: Operating system abstractions to manage GPUs as compute devices,” in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011, pp. 233–248.
- [218] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, and J. Koenig, “The Rocket Chip Generator,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, 2016. [Online]. Available: <https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.pdf>
- [219] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, A. Ou, C. Schmidt, S. Steffl, J. Wright, I. Stoica, J. Ragan-Kelley, K. Asanovic, B. Nikolic, and Y. S. Shao, “Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration,” in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, IEEE, 2021, pp. 769–774.
- [220] Y. Lee, A. Ou, C. Schmidt, S. Karandikar, H. Mao, and K. Asanovic, “The Hwacha Microarchitecture Manual, Version 3.8.1,” EECS Department, University of

- California, Berkeley, Tech. Rep. UCB/EECS-2015-263, 2015. [Online]. Available: <https://people.eecs.berkeley.edu/~krste/papers/EECS-2015-263.pdf>
- [221] J. D. McCalpin, “Bandwidth Limits in the Intel Xeon Max (Sapphire Rapids with HBM) Processors,” in *High Performance Computing*, A. Bienz, M. Weiland, M. Baboulin, and C. Kruse, Eds., Cham: Springer Nature Switzerland, 2023, pp. 403–413, ISBN: 978-3-031-40843-4.
- [222] Z. Wang and T. Nowatzki, “Stream-based memory access specialization for general purpose processors,” in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 736–749.
- [223] I. Z. Reguly, “Comparative evaluation of bandwidth-bound applications on the Intel Xeon CPU Max series,” in *Proceedings of the SC’23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, 2023, pp. 1236–1244.
- [224] NVIDIA, *NVIDIA H200 Tensor Core GPU*, 2023. [Online]. Available: <https://nvdam.widen.net/s/nb5zzzsjdf/hpc-datasheet-sc23-h200-datasheet-3002446>
- [225] H. Kim, G. Ye, N. Wang, A. Yazdanbakhsh, and N. S. Kim, “Exploiting Intel Advanced Matrix Extensions (AMX) for Large Language Model Inference,” *IEEE Computer Architecture Letters*, vol. 23, no. 1, pp. 117–120, 2024.
- [226] Intel, *Intel 64 and IA-32 Architectures Software Developer’s Manual*, Intel, 2026.
- [227] Arm, *Arm Neon*, <https://developer.arm.com/Architectures/Neon>.
- [228] *AMD64 Architecture Programmer’s Manual Volume 4: 128-Bit and 256-Bit Media Instructions*, 26568, Advanced Micro Devices, 2021.
- [229] A. Fei and M. S. Abdelfattah, “NITRO: LLM Inference on Intel Laptop NPUs,” *arXiv preprint arXiv:2412.11053*, 2024.

- [230] M. Daga, A. M. Aji, and W.-C. Feng, “On the efficacy of a fused CPU+GPU processor (or APU) for parallel computing,” in *2011 Symposium on Application Accelerators in High-Performance Computing*, IEEE, 2011, pp. 141–149.
- [231] F. Kružel and K. Banaś, “AMD APU systems as a platform for scientific computing,” *Computer Methods in Materials Science*, vol. 15, no. 2, pp. 362–369, 2015.
- [232] Intel, *Intel QuickAssist Technology API Programmer’s Guide*, 330684-012, Intel, 2022.
- [233] B. Aydogmus, L. Guo, D. Zuberi, T. Garfinkel, D. Tullsen, A. Ousterhout, and K. Taram, “Extended User Interrupts (xUI): Fast and Flexible Notification without Polling,” in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2025, pp. 373–389.
- [234] D. Soltis and S. Robinson, “Clearwater Forest: The Next Generation Intel® Xeon® Processor with Efficiency Cores,” in *2025 IEEE Hot Chips 37 Symposium (HCS)*, IEEE Computer Society, 2025, pp. 1–15.
- [235] D. Pala, “Design and programming of a coprocessor for a RISC-V architecture: Guidelines for embedding computing cores as RISC-V coprocessors,” M.S. thesis, Politecnico di Torino, 2017. [Online]. Available: <https://webthesis.biblio.polito.it/6589/1/tesi.pdf>
- [236] Intel. “Intel® oneAPI Math Kernel Library,” Intel, Accessed: 2023. [Online]. Available: <https://intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html>
- [237] L. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. Whaley, “An updated set of Basic Linear Algebra Subprograms (BLAS),” *ACM*

Transactions on Mathematical Software, vol. 28, Jun. 2002. DOI:
10.1145/567806.567807

- [238] A. Naithani, J. Roelandts, S. Ainsworth, T. M. Jones, and L. Eeckhout, “Decoupled Vector Runahead,” in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 17–31.
- [239] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, “IMP: Indirect memory prefetcher,” in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 178–190.
- [240] S. Khan, A. R. Alameldeen, C. Wilkerson, O. Mutlu, and D. A. Jimenez, “Improving cache performance using read-write partitioning,” in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 452–463.
- [241] S. Subramaniam, A. Bracy, H. Wang, and G. H. Loh, “Criticality-based optimizations for efficient load processing,” in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, 2009, pp. 419–430.
- [242] P. Lavin, J. Young, R. Vuduc, J. Riedy, A. Vose, and D. Ernst, “Evaluating gather and scatter performance on CPUs and GPUs,” in *Proceedings of the International Symposium on Memory Systems*, 2020, pp. 209–222.
- [243] R. Jayanth, N. Gupta, and V. Prasanna, “Benchmarking Edge AI Platforms for High-Performance ML Inference,” in *2024 IEEE High Performance Extreme Computing Conference (HPEC)*, IEEE, 2024, pp. 1–7.
- [244] F. Wilkinson, J. P. Jones, R. Muneeb, and S. N. McIntosh-Smith, “Leveraging Arm’s Scalable Matrix Extension in Accelerating Matrix Multiplications Kernels,” in *Workshop on Modeling & Simulation of Systems and Applications*, 2023.

- [245] S. Karandikar, A. N. Udipi, J. Choi, J. Whangbo, J. Zhao, S. Kanev, E. Lim, J. Alakuijala, V. Madduri, Y. S. Shao, B. Nikolic, K. Asanovic, and P. Ranganathan, “CDPU: Co-designing Compression and Decompression Processing Units for Hyperscale Systems,” in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23, Orlando, FL, USA: Association for Computing Machinery, 2023, ISBN: 9798400700958.
- [246] M. Weidmann, *Introducing the Scalable Matrix Extension for the Armv9-A Architecture*, 2021. [Online]. Available: <https://community.arm.com/arm-community-blogs/b/architectures-andprocessors-blog/posts/scalable-matrix-extension-armv9-a-architecture>
- [247] Y. Kone, L. Duval, R. Lachaize, P. Felber, D. Hagimont, and A. Tchana, “Understanding Intel User Interrupts,” *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 9, no. 2, pp. 1–32, 2025.
- [248] K. Kovacs, “A Hardware Implementation of the Snappy Compression Algorithm,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2019-85, 2019, Master’s thesis. [Online]. Available: <https://people.eecs.berkeley.edu/~krste/papers/EECS-2019-85.pdf>
- [249] M. Asri, C. Dunham, R. Rusitoru, A. Gerstlauer, and J. Beard, “The Non-Uniform Compute Device (NUCD) architecture for lightweight accelerator offload,” in *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, IEEE, 2020, pp. 38–45.
- [250] C. Block, G. Gerogiannis, and J. Torrellas, “Micro-MAMA: Multi-Agent Reinforcement Learning for Multicore Prefetching,” in *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*, 2025, pp. 884–898.
- [251] S. Ainsworth and T. M. Jones, “An event-triggered programmable prefetcher for irregular workloads,” *ACM Sigplan Notices*, vol. 53, no. 2, pp. 578–592, 2018.

- [252] N. Talati, K. May, A. Behroozi, Y. Yang, K. Kaszyk, C. Vasiladiotis, T. Verma, L. Li, B. Nguyen, J. Sun, J. M. Morton, A. Ahmadi, T. Austin, M. O’Boyle, S. Mahlke, T. Mudge, and R. Dreslinski, “Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design,” in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, 2021, pp. 654–667.
- [253] W. Heirman, K. D. Bois, Y. Vandriessche, S. Eyerman, and I. Hur, “Near-Side Prefetch Throttling: Adaptive Prefetching for High-Performance Many-Core Processors,” in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT ’18, Limassol, Cyprus: Association for Computing Machinery, 2018.
- [254] A. V. Jamet, G. Vavouliotis, D. A. Jiménez, L. Alvarez, and M. Casas, “A Two Level Neural Approach Combining Off-Chip Prediction with Adaptive Prefetch Filtering,” in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, 2024, pp. 528–542.
- [255] Y. Yang, J. S. Emer, and D. Sanchez, “SpZip: Architectural support for effective data compression in irregular applications,” in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2021, pp. 1069–1082.
- [256] S. Balay, S. Abhyankar, M. F. Adams, S. Benson, J. Brown, P. Brune, K. Buschelman, E. M. Constantinescu, L. Dalcin, A. Dener, V. Eijkhout, J. Faibussowitsch, W. D. Gropp, V. Hapla, T. Isaac, P. Jolivet, D. Karpeev, D. Kaushik, M. G. Knepley, F. Kong, S. Kruger, D. A. May, L. C. McInnes, R. T. Mills, L. Mitchell, T. Munson, J. E. Roman, K. Rupp, P. Sanan, J. Sarich, B. F. Smith, S. Zampini, H. Zhang, H. Zhang, and J. Zhang, *PETSc Web page*, <https://petsc.org/>, 2023. [Online]. Available: <https://petsc.org/>

- [257] S. Acer, O. Selvitopi, and C. Aykanat, “Improving performance of sparse matrix dense matrix multiplication on large-scale parallel systems,” *Parallel Computing*, vol. 59, pp. 71–96, 2016.
- [258] D. C. Montgomery, E. A. Peck, and G. G. Vining, *Introduction to linear regression analysis*. John Wiley & Sons, 2021.
- [259] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken, “Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc,” *Proceedings of Machine Learning and Systems*, vol. 2, pp. 187–198, 2020.
- [260] A. Tripathy, K. Yelick, and A. Buluç, “Reducing Communication in Graph Neural Network Training,” in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–14. DOI: 10.1109/SC41405.2020.00074
- [261] NCSA, *Delta User Documentation*, NCSA, 2024. Accessed: 2024. [Online]. Available: <https://docs.ncsa.illinois.edu/systems/delta/en/latest/>
- [262] H. P. Enterprise. “HPE Slingshot interconnect,” Hewlett Packard Enterprise, Accessed: Jan. 2024. [Online]. Available: <https://www.hpe.com/us/en/compute/hpc/slingshot-interconnect.html>
- [263] OpenMP Architecture Review Board, *OpenMP Application Program Interface Version 4.5*, Nov. 2015. [Online]. Available: <https://openmp.org/wp-content/uploads/openmp-4.5.pdf>
- [264] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, “Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation,” in *Proceedings, 11th European PVM/MPI Users’ Group Meeting*, Sep. 2004, pp. 97–104.

- [265] Message Passing Interface Forum, *MPI: A message-passing interface standard version 4.0*, <https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf>, Jun. 2021. [Online]. Available:
<https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf>
- [266] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, Y. Shahar, S. Potluri, D. Rossetti, D. Becker, D. Poole, C. Lamb, S. Kumar, C. Stunkel, G. Bosilca, and A. Bouteiller, “UCX: an open source framework for HPC network APIs and beyond,” in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, IEEE, 2015, pp. 40–43.
- [267] A. Azad, O. Selvitopi, M. T. Hussain, J. R. Gilbert, and A. Buluç, “Combinatorial BLAS 2.0: Scaling combinatorial algorithms on distributed-memory systems,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 989–1001, 2021.
- [268] “Eigen v3.4,” Accessed: Jan. 2024. [Online]. Available:
<https://eigen.tuxfamily.org>
- [269] H. Kwak, C. Lee, H. Park, and S. Moon, “What is Twitter, a social network or a news media?” In *WWW ’10: Proc. the 19th Intl. Conf. on World Wide Web*, Raleigh, North Carolina, USA: ACM, 2010, pp. 591–600. DOI:
<http://doi.acm.org/10.1145/1772690.1772751>
- [270] R. F. Boisvert, R. F. Boisvert, and K. A. Remington, *The matrix market exchange formats: Initial design*. Citeseer, 1996, vol. 5935.
- [271] Z. Jia, Y. Kwon, G. Shipman, P. McCormick, M. Erez, and A. Aiken, “A Distributed Multi-GPU System for Fast Graph Processing,” *Proceedings of the VLDB Endowment*, vol. 11, no. 3, pp. 297–310, Nov. 2017. DOI:
[10.14778/3157794.3157799](https://doi.org/10.14778/3157794.3157799) [Online]. Available:
<https://doi.org/10.14778/3157794.3157799>

- [272] J. Won, C. Mendis, J. S. Emer, and S. Amarasinghe, “WACO: Learning Workload-Aware Co-Optimization of the Format and Schedule of a Sparse Tensor Program,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023, Vancouver, BC, Canada: Association for Computing Machinery, 2023, pp. 920–934. DOI: 10.1145/3575693.3575742 [Online]. Available: <https://doi.org/10.1145/3575693.3575742>
- [273] S. Li, K. Osawa, and T. Hoefer, “Efficient Quantized Sparse Matrix Operations on Tensor Cores,” in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2022, pp. 1–15. DOI: 10.1109/SC41404.2022.00042
- [274] Z. Gu, J. Moreira, D. Edelsohn, and A. Azad, “Bandwidth Optimized Parallel Algorithms for Sparse Matrix-Matrix Multiplication Using Propagation Blocking,” in *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA ’20, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 293–303. DOI: 10.1145/3350755.3400216 [Online]. Available: <https://doi.org/10.1145/3350755.3400216>
- [275] O. Hsu, M. Strange, R. Sharma, J. Won, K. Olukotun, J. S. Emer, M. A. Horowitz, and F. Kjølstad, “The Sparse Abstract Machine,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023, Vancouver, BC, Canada: Association for Computing Machinery, 2023, pp. 710–726. DOI: 10.1145/3582016.3582051 [Online]. Available: <https://doi.org/10.1145/3582016.3582051>
- [276] H. D. Tran, M. Fernando, K. Saurabh, B. Ganapathysubramanian, R. M. Kirby, and H. Sundar, “A scalable adaptive-matrix SPMV for heterogeneous

- architectures,” in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2022, pp. 13–24. DOI: 10.1109/IPDPS53621.2022.00011
- [277] T. O. Odemuyiwa, H. Asghari-Moghaddam, M. Pellauer, K. Hegde, P.-A. Tsai, N. C. Crago, A. Jaleel, J. D. Owens, E. Solomonik, J. S. Emer, and C. W. Fletcher, “Accelerating Sparse Data Orchestration via Dynamic Reflexive Tiling,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023, Vancouver, BC, Canada: Association for Computing Machinery, 2023, pp. 18–32. DOI: 10.1145/3582016.3582064 [Online]. Available: <https://doi.org/10.1145/3582016.3582064>
- [278] M. T. Hussain, O. Selvitopi, A. Buluç, and A. Azad, “Communication-Avoiding and Memory-Constrained Sparse Matrix-Matrix Multiplication at Extreme Scale,” in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 90–100. DOI: 10.1109/IPDPS49936.2021.00018
- [279] Z. Cai, X. Yan, Y. Wu, K. Ma, J. Cheng, and F. Yu, “DGCL: An Efficient Communication Library for Distributed GNN Training,” in *Proceedings of the Sixteenth European Conference on Computer Systems*, ser. EuroSys ’21, Online Event, United Kingdom: Association for Computing Machinery, 2021, pp. 130–144. DOI: 10.1145/3447786.3456233 [Online]. Available: <https://doi.org/10.1145/3447786.3456233>
- [280] K. Kanellopoulos, N. Vijaykumar, C. Giannoula, R. Azizi, S. Koppula, N. M. Ghiasi, T. Shahroodi, J. G. Luna, and O. Mutlu, “SMASH: Co-Designing Software Compression and Hardware-Accelerated Indexing for Efficient Sparse Matrix Operations,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’52, Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 600–614. DOI: 10.1145/3352460.3358286

- [281] M. Orenes-Vera, E. Tureci, D. Wentzlaff, and M. Martonosi, “Dalorex: A data-local program execution and architecture for memory-bound applications,” in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, 2023, pp. 718–730.
- [282] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, “A scalable processing-in-memory accelerator for parallel graph processing,” in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 105–117.
- [283] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, “GraphP: Reducing communication for PIM-based graph processing with efficient data partition,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2018, pp. 544–557.
- [284] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, “GraphQ: Scalable PIM-based graph processing,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 712–725.
- [285] C. Giannoula, I. Fernandez, J. G. Luna, N. Koziris, G. Goumas, and O. Mutlu, “SparseP: Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Architectures,” *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 6, no. 1, Feb. 2022. DOI: 10.1145/3508041
- [286] S. Rashidi, W. Won, S. Srinivasan, S. Sridharan, and T. Krishna, “Themis: A Network Bandwidth-Aware Collective Scheduling Policy for Distributed Training of DL Models,” in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA ’22, New York, New York: Association for Computing Machinery, 2022, pp. 581–596. DOI: 10.1145/3470496.3527382 [Online]. Available: <https://doi.org/10.1145/3470496.3527382>
- [287] X. Lu, B. Long, X. Chen, Y. Han, and X.-H. Sun, “ACES: Accelerating Sparse Matrix Multiplication with Adaptive Execution Flow and Concurrency-Aware

- Cache Optimizations,” in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2024, pp. 71–85.
- [288] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtárik, “Scaling Distributed Machine Learning With In-Network Aggregation,” in *USENIX NSDI*, 2021.
- [289] U. Mukhopadhyay, A. Tripathy, O. Selvitopi, K. Yelick, and A. Buluc, “Sparsity-aware communication for distributed graph neural network training,” in *Proceedings of the 53rd International Conference on Parallel Processing*, 2024, pp. 117–126.
- [290] Y. Hong and A. Buluç, “A Sparsity-Aware Distributed-Memory Algorithm for Sparse-Sparse Matrix Multiplication,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC ’24, Atlanta, GA, USA: IEEE Press, 2024. DOI: 10.1109/SC41406.2024.00053 [Online]. Available: <https://doi.org/10.1109/SC41406.2024.00053>
- [291] I. Ranawaka, M. T. Hussain, C. Block, G. Gerogiannis, J. Torrellas, and A. Azad, “Distributed-Memory Parallel Algorithms for Sparse Matrix and Sparse Tall-and-Skinny Matrix Multiplication,” in *2024 SC24: International Conference for High Performance Computing, Networking, Storage and Analysis SC*, IEEE Computer Society, Nov. 2024, pp. 730–746. DOI: 10.1109/SC41406.2024.00052 [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/SC41406.2024.00052>
- [292] NVIDIA, *RoCE v2 Considerations*, NVIDIA, 2024. Accessed: 2024. [Online]. Available: <https://enterprise-support.nvidia.com/s/article/roce-v2-considerations>

- [293] Wikipedia, *Goodput*, Wikipedia, 2024. Accessed: 2024. [Online]. Available: <https://en.wikipedia.org/wiki/Goodput>
- [294] AMD, *Pensando*, website, 2024. Accessed: Oct. 18, 2024. [Online]. Available: <https://www.amd.com/en/accelerators/pensando>
- [295] AMD, *Performance of a Multi-Stage SDN Pipeline on Arm® vs AMD Pensando™ Programmable Silicon*, White Paper, 2024. Accessed: Oct. 18, 2024. [Online]. Available: <https://www.amd.com/content/dam/amd/en/documents/pensando-technical-docs/white-papers/pensando-comparison-of-dpu-hardware-strategies.pdf>
- [296] NVIDIA, *CONNECTX-6 DX*, website, 2024. Accessed: Oct. 18, 2024. [Online]. Available: <https://www.nvidia.com/en-us/networking/ethernet/connectx-6-dx/>
- [297] NVIDIA, *NVIDIA Bluefield Data Processing Units*, website, 2024. Accessed: Oct. 18, 2024. [Online]. Available: <https://www.nvidia.com/en-us/networking/products/data-processing-unit/>
- [298] Intel, *Intel Ethernet Controller 700 Series - Open vSwitch Hardware Acceleration Application Note*, website, 2024. Accessed: Oct. 18, 2024. [Online]. Available: <https://builders.intel.com/docs/networkbuilders/intel-ethernet-controller-700-series-open-vswitch-hardware-acceleration-application-note.pdf>
- [299] Xilinx, *Alveo SN1000 SmartNICs*, website, 2024. Accessed: Oct. 18, 2024. [Online]. Available: <https://www.xilinx.com/content/dam/xilinx/publications/product-briefs/sn1000-product-brief.pdf>
- [300] Xilinx, *Alveo U250 Data Center Accelerator Card*, website, 2024. Accessed: Oct. 18, 2024. [Online]. Available: <https://www.xilinx.com/products/boards-and-kits/alveo/u250.html>

- [301] Intel, *Tofino: P4-programmable Ethernet switch ASIC that delivers better performance at lower power*, website, 2024. Accessed: Oct. 18, 2024. [Online]. Available: <https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html>
- [302] Intel, *Tofino2: Second-generation P4-programmable Ethernet Switch ASIC that Continues to Deliver Programmability without Compromise*, website, 2024. Accessed: Oct. 18, 2024. [Online]. Available: <https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html>
- [303] BROADCOM, *BCM88690*, website, 2024. Accessed: Oct. 18, 2024. [Online]. Available: <https://www.broadcom.com/products/ethernet-connectivity/switching/stratadnx/bcm88690>
- [304] BROADCOM, *Trident4 / BCM56880 Series*, website, 2024. Accessed: Oct. 18, 2024. [Online]. Available: <https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series>
- [305] BROADCOM, *Trident 5 / BCM78800 Series*, website, 2024. Accessed: Oct. 18, 2024. [Online]. Available: <https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm78800>
- [306] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, “P4: Programming Protocol-Independent Packet Processors,” in *ACM SIGCOMM CCR*, 2014.
- [307] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, “Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN,” *ACM SIGCOMM CCR*, 2013.

- [308] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, “NetCache: Balancing Key-Value Stores with Fast In-Network Caching,” in *ACM SOSR*, 2017.
- [309] F. M. Maley and J. G. DeVinney, “Conveyors for Streaming Many-To-Many Communication,” in *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, 2019, pp. 1–8. DOI: 10.1109/IA349570.2019.00007
- [310] K. Shafie Khorassani, C. C. Chen, B. Ramesh, A. Shafi, H. Subramoni, and D. Panda, “High Performance MPI over the Slingshot Interconnect: Early Experiences,” in *Practice and Experience in Advanced Research Computing*. 2022, pp. 1–7.
- [311] J. Kim, W. J. Dally, S. Scott, and D. Abts, “Technology-Driven, Highly-Scalable Dragonfly Topology,” *ACM SIGARCH Computer Architecture News*, vol. 36, no. 3, pp. 77–88, 2008.
- [312] K. Lakhotia, M. Besta, L. Monroe, K. Isham, P. Iff, T. Hoefler, and F. Petrini, “PolarFly: a cost-effective and flexible low-diameter topology,” in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2022, pp. 1–15.
- [313] K. Lakhotia, L. Monroe, K. Isham, M. Besta, N. Blach, T. Hoefler, and F. Petrini, “PolarStar: Expanding the Horizon of Diameter-3 Networks,” in *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*, 2024, pp. 345–357.
- [314] P. MacArthur, Q. Liu, R. D. Russell, F. Mizero, M. Veeraraghavan, and J. M. Dennis, “An Integrated Tutorial on InfiniBand, Verbs, and MPI,” *IEEE Communications Surveys & Tutorials*, vol. 19, no. 4, pp. 2894–2926, 2017.

- [315] NVIDIA, *NVIDIA Bluefield-3 Datasheet*, Datasheet, 2024. Accessed: Oct. 18, 2024. [Online]. Available:
<https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf/>
- [316] T. Swamy, A. Rucker, M. Shahbaz, I. Gaur, and K. Olukotun, “Taurus: a data plane architecture for per-packet ML,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 1099–1114.
- [317] BROADCOM, *Broadcom Ships Tomahawk 5, Industry’s Highest Bandwidth Switch Chip to Accelerate AI/ML Workloads*, website, 2024. Accessed: Oct. 18, 2024. [Online]. Available:
<https://investors.broadcom.com/news-releases/news-release-details/broadcom-ships-tomahawk-5-industrys-highest-bandwidth-switch>
- [318] *P416 Language Specification*, website, The P4 Language Consortium, 2024. Accessed: Nov. 20, 2024. [Online]. Available:
<https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html>
- [319] K. S. Hemmert, “Merlin Element Library Deep Dive.,” Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2018.
- [320] Synopsys, *Design Compiler User Guide*, Version L-2016.03, Synopsys, 2016.
- [321] J. E. Stine, I. D. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P. D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, and R. Jenkal, “FreePDK: An Open-Source Variation-Aware Design Kit,” in *2007 IEEE International Conference on Microelectronic Systems Education (MSE’07)*, San Diego, CA, Jun. 2007, pp. 173–174. DOI: 10.1109/MSE.2007.44
- [322] M. Galles and F. Matus, “Pensando distributed services architecture,” *IEEE Micro*, vol. 41, no. 2, pp. 43–49, 2021.

- [323] ServeTheHome, *Pensando Distributed Services Architecture – SmartNIC*, 2020.
- [324] Cepulis, Darren, *ARM: Power-efficient Compute for HPC*, Accessed: 2025-06-20. Originally presented at the OCP & HPC on ARM workshop, 2015.
- [325] A. Agrawal and C. Kim, “Intel Tofino2 - A 12.9 Tbps P4-Programmable Ethernet Switch,” in *2020 IEEE Hot Chips 32 Symposium (HCS)*, IEEE Computer Society, 2020, pp. 1–32.
- [326] V. Shrivastav, “Stateful multi-pipelined programmable switches,” in *Proceedings of the ACM SIGCOMM 2022 Conference*, 2022, pp. 663–676.
- [327] S. Chen, S. Pal, and R. Kumar, “Waferscale network switches,” in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2024, pp. 215–229.
- [328] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall, “dRMT: Disaggregated Programmable Switching,” in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017, pp. 1–14.
- [329] M. Karol, M. Hluchyj, and S. Morgan, “Input versus output queueing on a space-division packet switch,” *IEEE Transactions on communications*, vol. 35, no. 12, pp. 1347–1356, 2003.
- [330] N. McKeown, “The iSLIP scheduling algorithm for input-queued switches,” *IEEE/ACM Transactions on Networking*, vol. 7, no. 2, pp. 188–201, 1999. DOI: 10.1109/90.769767
- [331] R. Rojas-Cessa, E. Oki, Z. Jing, and H. J. Chao, “CIXB-1: Combined input-one-cell-crosspoint buffered switch,” in *2001 IEEE Workshop on High Performance Switching and Routing (IEEE Cat. No. 01TH8552)*, IEEE, 2001, pp. 324–329.

- [332] H. Ibeid, V. Narayana, J. Kim, A. Nguyen, V. Morozov, and Y. Luo, “Performance Analysis of HPC Applications on the Aurora Supercomputer: Exploring the Impact of HBM-Enabled Intel Xeon Max CPUs,” in *ISC High Performance 2025 Research Paper Proceedings (40th International Conference)*, Prometeus GmbH, 2025, pp. 1–11.
- [333] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber, “HyperX: topology, routing, and packaging of efficient large-scale networks,” in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009, pp. 1–11.
- [334] M. Yang, A. Baban, V. Kugel, J. Libby, S. Mackie, S. S. R. Kananda, C.-H. Wu, and M. Ghobadi, “Using Trio: Juniper Networks’ Programmable Chipset for Emerging In-Network Applications,” in *ACM SIGCOMM*, 2022.
- [335] NVIDIA, *AllReduce*, website, 2024. Accessed: Nov. 20, 2024. [Online]. Available: <https://docs.nvidia.com/doca/archive/doca-v1.3/allreduce/index.html>
- [336] Y. Li, I.-J. Liu, Y. Yuan, D. Chen, A. Schwing, and J. Huang, “Accelerating distributed reinforcement learning with in-switch computing,” in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 279–291.
- [337] K. Lakhotia, K. Isham, L. Monroe, M. Besta, T. Hoefer, and F. Petrini, “In-network Allreduce with Multiple Spanning Trees on PolarFly,” in *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures*, 2023, pp. 165–176.
- [338] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya, “IncBricks: Toward In-Network Computation with an In-Network Cache,” in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 795–809.

- [339] J. Li, J. Nelson, E. Michael, X. Jin, and D. R. Ports, “Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories,” in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 387–406.
- [340] A. Kokolis, A. Psistakis, B. Reidys, J. Huang, and J. Torrellas, “HADES: Hardware-Assisted Distributed Transactions in the Age of Fast Networks and SmartNICs,” in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2024, pp. 785–800.
- [341] A. Psistakis, F. Chaix, and J. Torrellas, “MINOS: Distributed Consistency and Persistency Protocol Implementation & Offloading to SmartNICs,” in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, 2024, pp. 1–17.
- [342] A. Kokolis, A. Psistakis, B. Reidys, J. Huang, and J. Torrellas, “Distributed Data Persistency,” in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 71–85.
- [343] K. Seemakhupt, S. Liu, Y. Senevirathne, M. Shahbaz, and S. Khan, “PMNet: In-Network Data Persistence,” in *ACM/IEEE ISCA*, IEEE, 2021.
- [344] D. De Sensi, S. Di Girolamo, S. Ashkboos, S. Li, and T. Hoefer, “Flare: Flexible In-Network Allreduce,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–16.
- [345] H. Jin, Z. Yue, Z. Zhao, Y. Du, C. Deng, N. Srivastava, and Z. Zhang, “Vesper: A Versatile Sparse Linear Algebra Accelerator With Configurable Compute Patterns,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2024. DOI: 10.1109/TCAD.2024.3496882

- [346] A. Azad, A. Buluç, and J. Gilbert, “Parallel triangle counting and enumeration using matrix algebra,” in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, IEEE, 2015, pp. 804–811.
- [347] A. Abou-Rjeili and G. Karypis, “Multilevel algorithms for partitioning power-law graphs,” in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, IEEE, 2006, 10–pp.
- [348] J. D. Ullman, “NP-complete scheduling problems,” *Journal of Computer and System sciences*, vol. 10, no. 3, pp. 384–393, 1975.
- [349] N. R. Miniskar, F. Liu, A. R. Young, D. Chakraborty, and J. S. Vetter, “A Hierarchical Task Scheduler for Heterogeneous Computing,” in *International Conference on High Performance Computing*, Springer, 2021, pp. 57–76.
- [350] T. A. Davis, “Algorithm 1000: SuiteSparse: GraphBLAS: Graph algorithms in the language of sparse linear algebra,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 45, no. 4, pp. 1–25, 2019.
- [351] S. Huang, L.-W. Chang, I. El Hajj, S. Garcia de Gonzalo, J. Gómez-Luna, S. R. Chalamalasetti, M. El-Hadedy, D. Milojicic, O. Mutlu, D. Chen, and W.-m. Hwu, “Analysis and Modeling of Collaborative Execution Strategies for Heterogeneous CPU-FPGA Architectures,” in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE ’19, Mumbai, India: Association for Computing Machinery, 2019, pp. 79–90.
- [352] S. Kaufman, P. Phothilimthana, Y. Zhou, C. Mendis, S. Roy, A. Sabne, and M. Burrows, “A Learned Performance Model for Tensor Processing Units,” in *Proceedings of Machine Learning and Systems*, vol. 3, 2021, pp. 387–400.
- [353] T. E. Carlson, W. Heirman, and L. Eeckhout, “Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulation,” in *Proceedings of 2011 International Conference for High Performance Computing*,

- Networking, Storage and Analysis*, ser. SC '11, Seattle, Washington: Association for Computing Machinery, 2011. DOI: 10.1145/2063384.2063454
- [354] Intel, *Intel Xeon Platinum 8260 Processor 35.75MB Cache 2.40 GHz Product Specifications*, Intel, 2019. [Online]. Available: <https://ark.intel.com/content/www/us/en/ark/products/192474/intel-xeon-platinum-8260-processor-35-75m-cache-2-40-ghz.html>
- [355] N. Nayak, T. O. Odemuyiwa, S. Ugare, C. Fletcher, M. Pellauer, and J. Emer, “Teaal: A declarative framework for modeling sparse tensor accelerators,” in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23, Toronto, ON, Canada: Association for Computing Machinery, 2023, pp. 1255–1270.
- [356] Y. N. Wu, P.-A. Tsai, A. Parashar, V. Sze, and J. S. Emer, “Sparseloop: An analytical approach to sparse tensor accelerator modeling,” in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2022, pp. 1377–1395.
- [357] D. Grewe and M. F. O’Boyle, “A static task partitioning approach for heterogeneous systems using OpenCL,” in *20th International Conference on Compiler Construction*, Springer, 2011, pp. 286–305.
- [358] K.-C. Hsu and H.-W. Tseng, “Simultaneous and heterogenous multithreading,” in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23, Toronto, ON, Canada: Association for Computing Machinery, 2023, pp. 137–152.
- [359] S. Zheng, Y. Liang, S. Wang, R. Chen, and K. Sheng, “FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating*

- Systems*, ser. ASPLOS '20, Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 859–873. DOI: 10.1145/3373376.3378508
- [360] L. Zhou, M. H. Samavatian, A. Bacha, S. Majumdar, and R. Teodorescu, “Adaptive Parallel Execution of Deep Neural Networks on Heterogeneous Edge Devices,” in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, ser. SEC '19, Arlington, Virginia: Association for Computing Machinery, 2019, pp. 195–208. DOI: 10.1145/3318216.3363312
 - [361] Z. Jia, M. Zaharia, and A. Aiken, “Beyond Data and Model Parallelism for Deep Neural Networks,” *Proceedings of Machine Learning and Systems*, vol. 1, pp. 1–13, 2019.
 - [362] H. Kwon, L. Lai, M. Pellauer, T. Krishna, Y.-H. Chen, and V. Chandra, “Heterogeneous Dataflow Accelerators for Multi-DNN Workloads,” in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 71–83. DOI: 10.1109/HPCA51647.2021.00016
 - [363] S. Wang, Y. Liang, and W. Zhang, “Poly: Efficient heterogeneous system and application management for interactive applications,” in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2019, pp. 199–210.
 - [364] T. K. Bandara, D. Wijerathne, T. Mitra, and L.-S. Peh, “REVAMP: A Systematic Framework for Heterogeneous CGRA Realization,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22, Lausanne, Switzerland: Association for Computing Machinery, 2022, pp. 918–932. DOI: 10.1145/3503222.3507772
 - [365] T. Gale, M. Zaharia, C. Young, and E. Elsen, “Sparse GPU Kernels for Deep Learning,” in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–14.

- [366] S. E. Kurt, A. Sukumaran-Rajam, F. Rastello, and P. Sadayyapan, “Efficient Tiled Sparse Matrix Multiplication through Matrix Signatures,” in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–14. DOI: 10.1109/SC41405.2020.00091
- [367] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, “Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks,” in *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '09, Calgary, AB, Canada: Association for Computing Machinery, 2009, pp. 233–244. DOI: 10.1145/1583991.1584053
- [368] D. Lenadora, V. Sathia, G. Gerogiannis, S. Yesil, J. Torrellas, and C. Mendis, “GRANII: Selection and Ordering of Primitives in GRaph Neural Networks using Input Inspection,” in *2026 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2026, pp. 14–27.
- [369] V. Balaji, N. C. Crago, A. Jaleel, and S. W. Keckler, “Community-based Matrix Reordering for Sparse Linear Algebra Optimization,” in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, 2023, pp. 214–223.
- [370] P. Boldi, M. Rosa, M. Santini, and S. Vigna, “Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks,” in *Proceedings of the 20th international conference on World Wide Web*, 2011, pp. 587–596.
- [371] NVIDIA, *Accelerating Matrix Multiplication with Block Sparse Format and NVIDIA Tensor Cores*, NVIDIA, 2021. [Online]. Available: <https://developer.nvidia.com/blog/accelerating-matrix-multiplication-with-block-sparse-format-and-nvidia-tensor-cores>

- [372] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. C. Whaley, and K. Yelick, “Self-adapting linear algebra algorithms and software,” *Proceedings of the IEEE*, vol. 93, no. 2, pp. 293–312, 2005.
- [373] A. Azad, G. Ballard, A. Buluc, J. Demmel, L. Grigori, O. Schwartz, S. Toledo, and S. Williams, “Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication,” *SIAM Journal on Scientific Computing*, vol. 38, no. 6, pp. C624–C651, 2016.
- [374] R. Bera, K. Kanellopoulos, S. Balachandran, D. Novo, A. Olgun, M. Sadrosadat, and O. Mutlu, “Hermes: Accelerating Long-Latency Load Requests via Perceptron-Based Off-Chip Load Prediction,” in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 1–18.
- [375] D. Jimenez and C. Lin, “Dynamic Branch Prediction with Perceptrons,” in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, 2001, pp. 197–206.
- [376] S. J. Tarsa, C.-K. Lin, G. Keskin, G. Chinya, and H. Wang, “Improving Branch Prediction by Modeling Global History with Convolutional Neural Networks,” *arXiv preprint arXiv:1906.09889*, 2019.
- [377] Z. Shi, A. Jain, K. Swersky, M. Hashemi, P. Ranganathan, and C. Lin, “A Hierarchical Neural Model of Data Prefetching,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’21, Virtual, USA: Association for Computing Machinery, 2021, pp. 861–873.
- [378] M. Hashemi, K. Swersky, J. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, “Learning Memory Access Patterns,” in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 80, PMLR, 2018, pp. 1919–1928.

- [379] S. Sethumurugan, J. Yin, and J. Sartori, “Designing a Cost-Effective Cache Replacement Policy using Machine Learning,” in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 291–303.
- [380] Z. Shi, X. Huang, A. Jain, and C. Lin, “Applying Deep Learning to the Cache Replacement Problem,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’52, Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 413–425.
- [381] R. Bitirgen, E. İpek, and J. F. Martinez, “Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach,” in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 41, IEEE Computer Society, 2008, pp. 318–329.
- [382] C. Dubach, T. M. Jones, E. V. Bonilla, and M. F. P. O’Boyle, “A Predictive Model for Dynamic Microarchitectural Adaptivity Control,” in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’43, IEEE Computer Society, 2010, pp. 485–496.
- [383] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT press, 2018.
- [384] D. Bertsekas, *Reinforcement Learning and Optimal Control*. Athena Scientific, 2019.
- [385] A. Zouzias, K. Kalaitzidis, and B. Grot, “Branch Prediction as a Reinforcement Learning Problem: Why, How and Case Studies,” *arXiv preprint arXiv:2106.13429*, 2021.
- [386] E. İpek, O. Mutlu, J. F. Martínez, and R. Caruana, “Self-Optimizing Memory Controllers: A Reinforcement Learning Approach,” in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA ’08, IEEE Computer Society, 2008, pp. 39–50.

- [387] R. Bera, K. Kanellopoulos, A. Nori, T. Shahroodi, S. Subramoney, and O. Mutlu, “Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning,” in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’21, Virtual Event, Greece: Association for Computing Machinery, 2021, pp. 1121–1137.
- [388] G. Singh, R. Nadig, J. Park, R. Bera, N. Hajinazar, D. Novo, J. Gómez-Luna, S. Stuijk, H. Corporaal, and O. Mutlu, “Sibyl: Adaptive and Extensible Data Placement in Hybrid Storage Systems Using Online Reinforcement Learning,” in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA ’22, New York, New York: Association for Computing Machinery, 2022, pp. 320–336.
- [389] J. Zuckerman, D. Giri, J. Kwon, P. Mantovani, and L. P. Carloni, “Cohmeleon: Learning-Based Orchestration of Accelerator Coherence in Heterogeneous SoCs,” in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’21, Virtual Event, Greece: Association for Computing Machinery, 2021, pp. 350–365.
- [390] L. Peled, S. Mannor, U. Weiser, and Y. Etsion, “Semantic Locality and Context-Based Prefetching Using Reinforcement Learning,” in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA ’15, Portland, Oregon: Association for Computing Machinery, 2015, pp. 285–297.
- [391] M. Jalili and M. Erez, “Managing Prefetchers with Deep Reinforcement Learning,” *IEEE Computer Architecture Letters*, vol. 21, no. 2, pp. 105–108, 2022.
- [392] H. Zheng and A. Louri, “An Energy-Efficient Network-on-Chip Design Using Reinforcement Learning,” in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC ’19, Las Vegas, NV, USA: Association for Computing Machinery, 2019.

- [393] W. R. Thompson, “On the Likelihood that One Unknown Probability Exceeds Another in View of the Evidence of Two Samples,” *Biometrika*, vol. 25, no. 3-4, pp. 285–294, 1933.
- [394] H. Robbins, “Some Aspects of the Sequential Design of Experiments,” *Bulletin of the American Mathematical Society*, vol. 58, no. 5, pp. 527–535, 1952.
- [395] A. Garivier and E. Moulines, “On Upper-Confidence Bound Policies for Non-Stationary Bandit Problems,” *arXiv arXiv:0805.3415*, 2008.
- [396] D. M. Tullsen, S. J. Eggers, and H. M. Levy, “Simultaneous Multithreading: Maximizing on-Chip Parallelism,” in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ser. ISCA ’95, S. Margherita Ligure, Italy: Association for Computing Machinery, 1995, pp. 392–403.
- [397] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, “Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor,” in *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ser. ISCA ’96, Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 191–202.
- [398] S. Choi and D. Yeung, “Learning-Based SMT Processor Resource Distribution via Hill-Climbing,” in *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ser. ISCA ’06, IEEE Computer Society, 2006, pp. 239–251.
- [399] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, “Bingo Spatial Data Prefetcher,” in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 399–411.
- [400] M. Shakerinava, M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, “Multi-Lookahead Offset Prefetching,” *The Third Data Prefetching Championship*, 2019.

- [401] B. Donyanavard, T. Mück, A. M. Rahmani, N. Dutt, A. Sadighi, F. Maurer, and A. Herkersdorf, “SOSA: Self-Optimizing Learning with Self-Adaptive Control for Hierarchical System-on-Chip Management,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’52, Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 685–698.
- [402] E. İpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz, “Efficiently Exploring Architectural Design Spaces via Predictive Modeling,” in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII, San Jose, California, USA: Association for Computing Machinery, 2006, pp. 195–206.
- [403] P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil, “A Predictive Performance Model for Superscalar Processors,” in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 39, IEEE Computer Society, 2006, pp. 161–170.
- [404] B. C. Lee and D. M. Brooks, “Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction,” in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII, San Jose, California, USA: Association for Computing Machinery, 2006, pp. 185–194.
- [405] C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin, “Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks,” in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 97, PMLR, 2019, pp. 4505–4515.
- [406] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, “Dynamic Knobs for Responsive Power-Aware Computing,” in *Proceedings of the*

- Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI, Newport Beach, California, USA: Association for Computing Machinery, 2011, pp. 199–212.
- [407] K. Ma, X. Li, M. Chen, and X. Wang, “Scalable Power Control for Many-Core Architectures Running Multi-Threaded Applications,” in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA ’11, San Jose, California, USA: Association for Computing Machinery, 2011, pp. 449–460.
 - [408] R. P. Pothukuchi, A. Ansari, P. Voulgaris, and J. Torrellas, “Using Multiple Input, Multiple Output Formal Control to Maximize Resource Efficiency in Architectures,” in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA ’16, Seoul, Republic of Korea: IEEE Press, 2016, pp. 658–670.
 - [409] R. P. Pothukuchi, S. Y. Pothukuchi, P. Voulgaris, and J. Torrellas, “Yukta: Multilayer Resource Controllers to Maximize Efficiency,” in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA ’18, Los Angeles, California: IEEE Press, 2018, pp. 505–518.
 - [410] R. P. Pothukuchi, J. L. Greathouse, K. Rao, C. Erb, L. Piga, P. G. Voulgaris, and J. Torrellas, “Tangram: Integrated Control of Heterogeneous Computers,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’52, Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 384–398.
 - [411] SPEC2006, *spec06*, 2006. [Online]. Available: <https://www.spec.org/cpu2006/>
 - [412] SPEC2017, *spec17*, 2017. [Online]. Available: <https://www.spec.org/cpu2017/>
 - [413] R. Bellman, “A Markovian Decision Process,” *Journal of mathematics and mechanics*, pp. 679–684, 1957.
 - [414] C. J. C. H. Watkins and P. Dayan, “Q-learning,” *Machine learning*, vol. 8, pp. 279–292, 1992.

- [415] G. Rummery and M. Niranjan, “On-Line Q-Learning Using Connectionist Systems,” *Technical Report CUED/F-INFENG/TR 166*, 1994.
- [416] S. L. Scott, “A Modern Bayesian Look at the Multi-Armed Bandit,” *Appl. Stoch. Model. Bus. Ind.*, vol. 26, no. 6, pp. 639–658, 2010.
- [417] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire, “The Nonstochastic Multiarmed Bandit Problem,” *SIAM journal on computing*, vol. 32, no. 1, pp. 48–77, 2002.
- [418] V. Jiménez, R. Gioiosa, F. J. Cazorla, A. Buyuktosunoglu, P. Bose, and F. P. O’Connell, “Making Data Prefetch Smarter: Adaptive Prefetching on POWER7,” in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT ’12, Minneapolis, Minnesota, USA: Association for Computing Machinery, 2012, pp. 137–146.
- [419] J. W. C. Fu, J. H. Patel, and B. L. Janssens, “Stride Directed Prefetching in Scalar Processors,” in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, ser. MICRO 25, Portland, Oregon, USA: IEEE Computer Society Press, 1992, pp. 102–110.
- [420] S. Kondguli and M. Huang, “Division of Labor: A More Effective Approach to Prefetching,” in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA ’18, Los Angeles, California: IEEE Press, 2018, pp. 83–95.
- [421] I. Hur and C. Lin, “Memory Prefetching Using Adaptive Stream Detection,” in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 39, IEEE Computer Society, 2006, pp. 397–408.
- [422] M. Taram, X. Ren, A. Venkat, and D. Tullsen, “SecSMT: Securing SMT Processors against Contention-Based Covert Channels,” in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3165–3182.

- [423] P. Auer, N. Cesa-Bianchi, and P. Fischer, “Finite-time Analysis of the Multiarmed Bandit Problem,” *Machine learning*, vol. 47, pp. 235–256, 2002.
- [424] P. Auer, “Using Confidence Bounds for Exploitation-Exploration Trade-Offs,” *J. Mach. Learn. Res.*, vol. 3, pp. 397–422, 2003.
- [425] R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd, “POWER7: IBM’s Next-Generation Server Processor,” *IEEE Micro*, vol. 30, no. 2, pp. 7–15, 2010.
- [426] A. Fog, “Instruction Tables: Lists of Instruction Latencies, Throughputs and Micro-Operation Breakdowns for Intel, AMD and VIA CPUs,” *Copenhagen University College of Engineering*, vol. 93, p. 110, 2011.
- [427] N. Gober, G. Chacon, L. Wang, P. V. Gratz, D. A. Jimenez, E. Teran, S. Pugsley, and J. Kim, “The Championship Simulator: Architectural Simulation for Education and Competition,” *arXiv preprint arXiv:2210.14324*, 2022.
- [428] G. Vavouliotis, L. Alvarez, V. Karakostas, K. Nikas, N. Koziris, D. A. Jiménez, and M. Casas, “Exploiting Page Table Locality for Agile TLB Prefetching,” in *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ser. ISCA ’21, Virtual Event, Spain: IEEE Press, 2021, pp. 85–98.
- [429] G. Vavouliotis, G. Chacon, L. Alvarez, P. V. Gratz, D. A. Jiménez, and M. Casas, “Page Size Aware Cache Prefetching,” in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2022, pp. 956–974.
- [430] E. Bhatia, G. Chacon, S. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, “Perceptron-Based Prefetch Filtering,” in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA ’19, Phoenix, Arizona: Association for Computing Machinery, 2019, pp. 1–13.
- [431] S. Jiang, Q. Yang, and Y. Ci, “Merging Similar Patterns for Hardware Prefetching,” in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2022, pp. 1012–1026.

- [432] WikiChip, *Skylake (client)*, 2016. [Online]. Available:
[https://en.wikichip.org/wiki/intel/microarchitectures/skylake_\(client\)](https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client))
- [433] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The gem5 Simulator,” *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.
- [434] M. Taram, *SecSMT Artifact*, 2022. [Online]. Available:
https://github.com/mktrm/SecSMT_Artifact
- [435] PARSEC, *parsec*, 2010. [Online]. Available: <http://parsec.cs.princeton.edu>
- [436] J. Shun and G. E. Blelloch, “Ligra: A Lightweight Graph Processing Framework for Shared Memory,” in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP ’13, Shenzhen, China: Association for Computing Machinery, 2013, pp. 135–146.
- [437] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafae, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, “Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware,” *SIGPLAN Not.*, vol. 47, no. 4, pp. 37–48, 2012.
- [438] DPC-3, *3rd Data Prefetching Championship*, 2019. [Online]. Available:
<https://dpc3.compas.cs.stonybrook.edu>
- [439] CRC-2, *2nd Cache Replacement Championship*, 2017. [Online]. Available:
<https://crc2.ece.tamu.edu>
- [440] G. Hamerly, E. Perelman, J. Lau, and B. Calder, “Simpoint 3.0: Faster and More Flexible Program Phase Analysis,” *Journal of Instruction Level Parallelism*, vol. 7, no. 4, pp. 1–28, 2005.

- [441] S. Pakalapati and B. Panda, “Bouquet of Instruction Pointers: Instruction Pointer Classifier-based Spatial Hardware Prefetching,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 118–131.
- [442] A. Snavey, D. M. Tullsen, and G. Voelker, “Symbiotic Job Scheduling with Priorities for a Simultaneous Multithreading Processor,” in *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS ’02, Marina Del Rey, California: Association for Computing Machinery, 2002, pp. 66–76.
- [443] K. Luo, J. Gummaraju, and M. Franklin, “Balancing Throughput and Fairness in SMT Processors,” in *2001 IEEE International Symposium on Performance Analysis of Systems and Software. ISPASS.*, 2001, pp. 164–171.
- [444] S. Salehi and R. F. DeMara, “Energy and Area Analysis of a Floating-Point Unit in 15nm CMOS Process Technology,” in *SoutheastCon 2015*, 2015, pp. 1–5.
- [445] G. Vavouliotis, L. Alvarez, B. Grot, D. Jiménez, and M. Casas, “Morrigan: A Composite Instruction TLB Prefetcher,” in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’21, Virtual Event, Greece: Association for Computing Machinery, 2021, pp. 1138–1153.
- [446] P. Michaud, “Best-Offset Hardware Prefetching,” in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 469–480.
- [447] F. Eris, M. Louis, K. Eris, J. Abellán, and A. Joshi, “Puppeteer: A Random Forest Based Manager for Hardware Prefetchers Across the Memory Hierarchy,” *ACM Trans. Archit. Code Optim.*, vol. 20, no. 1, 2022.
- [448] H. Wang, I. Koren, and C. M. Krishna, “An Adaptive Resource Partitioning Algorithm for SMT Processors,” in *Proceedings of the 17th International Conference*

- on *Parallel Architectures and Compilation Techniques*, ser. PACT '08, Toronto, Ontario, Canada: Association for Computing Machinery, 2008, pp. 230–239.
- [449] S. Carroll and W.-M. Lin, “Applied On-Chip Machine Learning for Dynamic Resource Control in Multithreaded Processors,” *Parallel Processing Letters*, vol. 29, no. 03, p. 1950013, 2019.
- [450] H. Zhan, V. S. Sheng, and W.-M. Lin, “Reinforcement Learning-Based Register Renaming Policy for Simultaneous Multithreading CPUs,” *Expert Syst. Appl.*, vol. 186, no. C, 2022.
- [451] G. Ayers, H. Litz, C. Kozyrakis, and P. Ranganathan, “Classifying Memory Access Patterns for Prefetching,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20, Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 513–526.
- [452] D. Das Sharma, R. Blankenship, and D. Berger, “An introduction to the compute express link (cxl) interconnect,” *ACM Comput. Surv.*, vol. 56, no. 11, 2024, ISSN: 0360-0300. DOI: 10.1145/3669900 [Online]. Available: <https://doi.org/10.1145/3669900>
- [453] H. Lin, M. Maas, M. Roquemoire, A. Hasanzadeh, F. Lewis, Y. Simonson, T.-W. Yang, A. Yazdanbakhsh, D. Altinbüken, F. Papa, M. N. Edmonds, A. Patil, D. Schwarz, S. Chandra, C. Kennelly, M. Hashemi, and P. Ranganathan, *ECO: An LLM-Driven Efficient Code Optimizer for Warehouse Scale Computers*, 2025. arXiv: 2503.15669. [Online]. Available: <https://arxiv.org/abs/2503.15669>
- [454] R. Patti, “Three-dimensional integrated circuits and the future of system-on-chip designs,” *Proceedings of the IEEE*, vol. 94, no. 6, pp. 1214–1224, 2006. DOI: 10.1109/JPROC.2006.873612

- [455] H. Nam, G. Gerogiannis, and J. Torrellas, “MeshSlice: Efficient 2D Tensor Parallelism for Distributed DNN Training,” in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ser. ISCA ’25, Association for Computing Machinery, 2025, pp. 821–834, ISBN: 9798400712616. DOI: 10.1145/3695053.3731077 [Online]. Available: <https://doi.org/10.1145/3695053.3731077>
- [456] C. Sudusinghe, G. Gerogiannis, D. Lenadora, C. Block, J. Torrellas, and C. Mendis, “COGNATE: acceleration of sparse tensor programs on emerging hardware using transfer learning,” in *Proceedings of the 42nd International Conference on Machine Learning*, ser. ICML’25, Vancouver, Canada: JMLR.org, 2025.

Appendix: Other Works

In parallel to this thesis, I contributed to other research projects related to the scaling of accelerated computations and the optimization of heterogeneous systems. In this section, I briefly describe those other projects.⁷

Scaling. The Intel Programmable Integrated Unified Memory Architecture (PIUMA) is an accelerator architecture developed by Intel to address the challenges of large-scale graph processing. PIUMA features massive multithreading, efficient remote atomics, and extreme memory capacity and bandwidth in a distributed global address space. In [91], we performed a detailed characterization of how Graph Convolutional Networks (GCNs) scale on such an architecture. This characterization provides insights into how PIUMA’s memory-centric design can overcome scaling limitations of traditional CPU- and GPU-based systems.

In addition to the Two-Face distributed SpMM algorithm presented in Chapter 6, I contributed to two software algorithms that target other distributed computations. In [291], we presented a distributed-memory algorithm for TS-SpGEMM, a special variant of Sparse Matrix Sparse Matrix Multiplication (SpGEMM) in which one matrix is square and the other is tall and skinny. The proposed algorithm achieves high performance and scalability through a customized 1D partitioning method, sparsity-aware tiling, and careful balancing between local and remote computations.

Finally, MeshSlice [455] targets distributed dense general matrix multiplication (GeMM) on the Google TPU supercomputer. In contrast to 1D partitioning, 2D partitioning can reduce the communication volume for dense computations. However, overlapping 2D collective communication with computation over the TPU torus network is

⁷The works described in this section were led by Matthew Adiletta [91], Isuru Ranawaka [291], Hyoungho Nam [455], Chamika Sudusinghe [456], Damitha Lenadora [368], and Charles Block [250], who made the most significant contributions to their respective projects. I am grateful to have had the opportunity to contribute to these efforts and to work closely with each of them.

challenging. To effectively hide communication latency, MeshSlice slices collective communications into multiple partial collectives that can be overlapped with computation.

Optimization. The SPADE sparse accelerator introduced in Chapter 3 provides multiple configuration knobs exposed to the software through its tile ISA. However, choosing the optimal values for these knobs is challenging. With COGNATE [456], we introduced a machine learning framework that automatically builds performance models for emerging hardware accelerators like SPADE. COGNATE relies on Transfer Learning to take cost models trained using data from widely-available CPUs and fine-tune them for emerging accelerators. This is especially useful to tune software at the early design stages of an accelerator, when the actual chip is not yet taped-out, and performance evaluation relies on time-consuming simulation. We demonstrated that the performance models produced by COGNATE can be used to statically derive near-optimal software configurations for SPADE.

GRANII [368] automatically derives optimized kernel selections and orderings for Graph Neural Networks (GNNs). GNN computations can be expressed using different compositions and orderings of sparse and dense matrix primitives (e.g., SpMM, SDDMM, and GeMM), and the best choice depends on the input, target hardware, and underlying GNN model. GRANII utilizes lightweight input-aware cost models to select the best primitive composition at runtime. We demonstrated that GRANII improves GNN performance across multiple CPU and GPU platforms.

Finally, we found that when multiple uncoordinated Bandit hardware agents (Chapter 9) compete for shared resources in multicore systems, they can often converge to sub-optimal policies due to conflicting incentives. In the follow-up Micro-MAMA [250] work, we introduced a hardware meta-agent driven by reinforcement learning that ensures that multiple interacting Bandit agents make globally optimized decisions.