2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) | 979-8-3503-9739-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/ISPASS57527.2023.00025

Characterizing the Scalability of Graph Convolutional Networks on Intel® PIUMA

Matthew Joseph Adiletta[1,2], Jesmin Jahan Tithi[2], Emmanouil-Ioannis Farsarakis[2], Gerasimos Gerogiannis[2,3]
Robert Adolf[1,2], Robert Benke[2], Sidharth Kashyap[2], Samuel Hsia[1], Kartik Lakhotia[2], Fabrizio Petrini[2]
Gu-Yeon Wei[1], David Brooks[1]

[1]Harvard University [2]Intel Corporation [3]University of Illinois at Urbana-Champaign

***Abstract*—Large-scale Graph Convolutional Network (GCN) inference on traditional CPU/GPU systems is challenging due to a large memory footprint, sparse computational patterns, and irregular memory accesses with poor locality. Intel's Programmable Integrated Unified Memory Architecture (PIUMA) is designed to address these challenges for graph analytics. In this paper, a detailed characterization of GCNs is presented using the Open-Graph Benchmark (OGB) datasets to determine the viability of PIUMA as a potential solution to GCN scalability.**

First, the extent of sparse matrix dense matrix multiplication (SpMM) as a performance driver for GCN on CPU and GPU is explored, offering a methodology for predicting GCN behavior as a function of dataset characteristics. Second, an SpMM kernel optimized for PIUMA is described and investigated for sensitivity to system parameters including memory bandwidth, latency, and thread count. SpMM scalability on PIUMA is demonstrated, while the scalability limitations of a Xeon-optimized SpMM implementation are discussed. Finally, GCN performance is compared on PIUMA versus a Xeon CPU system and Ampere GPU system, showing impressive results on PIUMA for large-scale datasets.

***Index Terms*—Graph Convolution, SpMM, Memory Bandwidth Scaling, Latency Sensitivity, PIUMA, GCN**

I. Introduction

Graph Neural Networks (GNNs) are emerging as a powerful tool for graph analysis and learning. The graphical structure native to particular datasets can be leveraged by GNNs to improve task accuracy. GNNs have demonstrated success across a wide range of application domains including recommendation, health and drug discovery, and quantum chemistry [1]–[3]

Graph Convolutional Networks (GCN) [4] are a class of GNNs that generalize convolutions over graph-structured data. The fundamental GCN algorithm can be broken down into two phases – (a) aggregation and (b) update. The computation during the aggregation phase largely depends on the neighborhood structure (which influences sparsity) of the graph, while the computation during the update involves dense matrix multiplication. The key kernel of the GCN algorithm described in [4] is SpMM, introducing sparse computation, unlike conventional Convolutional Neural Networks (CNNs) which are mostly dense. CPU and GPU systems are well suited for dense computation; however, they do not scale well for sparse computation due to memory bandwidth and memory capacity constraints. This motivates the need for fundamentally different computer hardware and optimization techniques for large-scale GCN inference.

Intel's Programmable Integrated Unified Memory Architecture (PIUMA) system was recently proposed for scalable graph analytics using massive multi-threading, efficient remote atomics, extreme capacity and memory bandwidth in a distributed global address space [5]. Irregular memory access patterns, low arithmetic intensity kernels, and large memory footprints are key areas addressed by PIUMA, making it an attractive platform for GCN inference at scale.

In this work, GCN was characterized on the PIUMA system, demonstrating how PIUMA overcomes the performance bottlenecks of conventional shared-memory systems to enable GCNs at scale. The OGB datasets, one of the largest graph datasets using non-synthetic features [6], were used for a scalability evaluation. This work makes the following contributions.

1. An analysis of system bottlenecks that manifest during GCN execution on conventional architectures (Section III). System bottlenecks such as memory bandwidth and capacity on CPU and GPU architectures should be mitigated by a scalable graph processing system. This section offers intuition about how PIUMA may overcome these bottlenecks.

2. A characterization of an optimized SpMM kernel on PIUMA (Section IV). An analytical model for SpMM is detailed. A DMA-based SpMM algorithm is compared against this model, highlighting its strong scaling characteristics. This characterization is important because it exposes the relationship between memory bandwidth and SpMM, demonstrating the viability of PIUMA for SpMM at scale.

3. A comparison of SpMM on PIUMA versus Xeon CPU and Ampere GPU, leading to a discussion about GCN scalability on PIUMA (Section V). The performance of at-scale GCN was restricted by memory capacity on GPUs, and memory bandwidth on CPU. PIUMA overcomes these bottlenecks and GCNs were effectively accelerated. Further GCN acceleration may be attained by improving the dense computation capabilities of PIUMA.

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Distribution Statement A – Approved for Public Release, Distribution Unlimited.

979-8-3503-9739-0/23/$31.00 ©2023 IEEE
DOI 10.1109/ISPASS57527.2023.00025

II. Background

A. Graph Convolutional Networks

Given a graph $G(V, E)$ where V is the set of vertices and E is the set of edges, the feature vectors of all vertices can be represented as a matrix $\boldsymbol{H}_0$. A single GCN layer transforms these features using learned weights $\boldsymbol{W}_0$, as:

$$\boldsymbol{H}_1 = \sigma(\tilde{\boldsymbol{A}}\boldsymbol{H}_0\boldsymbol{W}_0)$$

where $\tilde{\boldsymbol{A}}$ is the normalized adjacency matrix of G, and σ is a non-linear activation. A small number of these layers are stacked, where each state vector $\boldsymbol{H}_t$ feeds into the next state vector $\boldsymbol{H}_{t+1}$. While input and output features are dataset specific, the number and feature length of hidden layers are design parameters of the GCN model. As state information propagates through the hidden layers, features of each vertex exert influence on other vertices further away, similar to the growing receptive field effect observed in CNNs.

The sparse adjacency matrix, dense feature matrix and weight matrices result in a mix of sparse aggregation ($\tilde{\boldsymbol{A}}\boldsymbol{H}_*$), dense update ($(\square)\boldsymbol{W}$), and element-wise ($\sigma$) matrix operations. GCN aggregation uses sparse matrix multiplication and is referred to as SpMM, while update uses dense matrix multiplication, referred to as Dense MM throughout this paper.

B. Dataset Selection - Open Graph Benchmarks

The datasets used in this investigation are from the Open Graph Benchmark (OGB) [6]. This benchmark suite provides real-world problems featuring domains such as nature, society and information; scale from thousands of vertices, to millions of vertices; and tasks such as node, link and graph problems. In particular, the datasets considered are from the node and link classification tasks. Details about the specific graphs used are shown in Table I.

An RMAT generator provided by the Stanford Network Analysis Platform [7] was used for performing linear function sweeps, such as in Figure 2.

TABLE I
OGB Dataset Descriptions

Name	$\|V\|$	$\|E\|$
ddi	4,267	1,334,889
proteins	132,534	39,561,252
arxiv	169,343	1,166,243
collab	235,868	1,285,465
ppa	576,289	30,326,273
mag	1,939,743	21,111,007
products	2,449,029	61,859,140
citation2	2,927,963	30,561,187
papers	111,059,956	1,615,685,872

C. Sparse Matrix - Dense Matrix Multiplication (SpMM)

SpMM is an important part of the GCN execution flow as it represents the per-vertex accumulation of neighboring vertex feature vectors during the aggregation phase.

As shown in Algorithm 1, the kernel uses an input sparse matrix with dimensions $|V| \times |V|$ and two dense matrices (input and output) with dimensions $|V| \times K$ (K denotes the embedding dimension). It scales the rows of the input dense matrix by the values of the sparse matrix and accumulates them on the output. Thus, the non-zeros of the sparse matrix dictate the access pattern for the rows of the input dense matrix. SpMM can be parallelized at the granularity of individual elements of the sparse matrix or output matrix rows. These methods are referenced throughout this paper as *edge-parallel* or *vertex-parallel*, respectively. In the first case, edges are distributed across threads while in the second case each thread only processes the in-edges of the vertices assigned to it. Trade-offs of these two methods are discussed in Section IV-B.

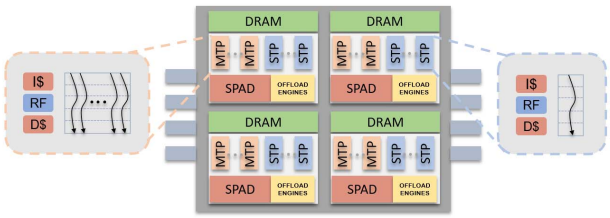

Fig. 1. PIUMA architecture overview showing 4 PIUMA cores, each with Multi-Threaded Pipelines (MTPs), Single-Threaded Pipelines (STPs), Scratchpad memory (SPAD), and DRAM slices.

Algorithm 1 SpMM

Input: Sparse $\tilde{\boldsymbol{A}} \in \mathbb{R}^{|V|\times|V|}$, Dense $\boldsymbol{H}_{in} \in \mathbb{R}^{|V|\times K}$
Output: Dense $\boldsymbol{H}_{out} = \tilde{\boldsymbol{A}}\boldsymbol{H}_{in}$

1: **procedure** SpMM
2: **for each** non-zero (u, v) in $\tilde{\boldsymbol{A}}$
3: $\boldsymbol{H}_{out}[u,:] = \boldsymbol{H}_{out}[u,:] + \tilde{\boldsymbol{A}}[u,v] * \boldsymbol{H}_{in}[v,:]$

D. PIUMA

Intel's Programmable Integrated Unified Memory Architecture (PIUMA) [5] is a specialized architecture developed to address graph analytic applications at scale. In contrast with conventional domain-specific accelerators, the PIUMA processing element pipelines implement a custom RISC ISA. Thus, PIUMA supports general purpose programming for offloading arbitrary workloads.

There are two types of PIUMA pipelines in the system, namely the Single-Threaded Pipelines (STPs) and the Multi-Threaded Pipelines (MTPs). Both the MTPs and STPs implement the same custom RISC ISA. The STPs are single issue, in-order, stall-on-use pipelines and are capable of exploiting intra-thread memory and instruction-level parallelism. Typically, they are used for single-threaded tasks such as memory/thread management.

The MTPs are single-issue, in-order and round-robin multi-threaded. Each thread can only have one in-flight instruction; thus, through fine-grained thread interleaving, long memory access latencies can be effectively hidden. For example, if a thread is stalled waiting for memory, pipeline resources may be utilized by other threads. The in-order single-issue design contributes to significantly reduced power consumption when compared to wide, out-of-order, superscalar cores.

Each pipeline is equipped with L1 data and instruction caches. The programmer selects whether a memory access will be cached or will bypass the cache. Caches are not coherent

across the whole system, meaning that it is the programmer's responsibility to ensure that coherency is retained for shared read-write data, such as by implementing the corresponding accesses as uncached.

PIUMA pipelines are organized in cores. Each core hosts a fraction of the global shared DRAM, STPs and MTPs, a local scratchpad and offload engines, as shown in Figure 1. The offload engines increase the available memory-level parallelism and reduce the pipeline pressure of the MTP's in-order designs. They support a collection of operations including fast remote atomics and atomic queue operations, Direct Memory Access (DMA) and efficient global collectives such as barriers and reductions.

PIUMA implements a hardware distributed global address space (DGAS) allowing each thread to access a *huge* pool of shared memory, distributed across the entire system. To decrease the latency and increase the interconnect bandwidth when accessing remote memory blocks, a Hyper-X network topology [8] is used in combination with optical links for node-to-node connections. The main characteristics of the PIUMA system that motivate its use for GCN acceleration can be summarized as follows:

- *Latency tolerance*: Pipelines in PIUMA cores hide long memory latencies by massive multithreading.
- *Enormous Parallelism*: A single PIUMA node supports concurrent execution of more than 16K threads.
- *Efficient remote atomics*: High thread count demands efficient atomic memory access for conflict resolution to prevent pipeline stalls.
- *Memory Performance*: DDR modules of each node provide aggregate TB capacities and TB/s bandwidths.
- *Multi-node scalability*: PIUMA's low-latency interconnection network and DGAS shared-memory abstraction provide scalability to large-scale graphs without the overheads related to graph partitioning and message-passing. This may enable high-performance GCN inference on large graphs which far exceed the capacity of existing CPUs/GPUs.

III. GCN Inference Performance on CPU and GPU

Prior work has shown GCN execution on both CPU and GPU [9]–[14]. We build upon prior work by characterizing the execution time of GCN kernels on conventional architectures by deeply investigating the scaling trends of embedding dimension. This section also provides intuition about why PIUMA may be an interesting architectural candidate for GCN at scale versus CPU or GPU solutions. For example, GPU systems offer high memory bandwidth, latency tolerance, and massive parallelism; however, GPUs are limited at scale by memory capacity. On the other hand, CPU systems offer terabytes of memory capacity and can support full graph operation; however, CPU systems will reach a memory-bandwidth peak at the node level, limiting the maximum performance of these systems.

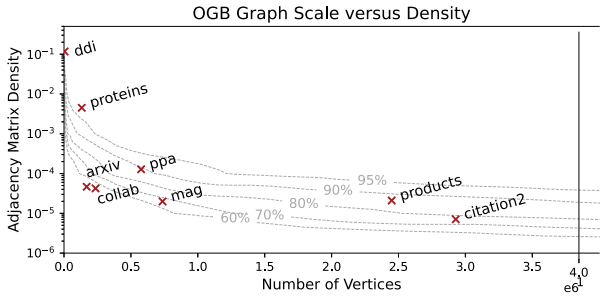

Fig. 2. Visualizing the relationship between the number of vertices, adjacency matrix density, and percentage execution time for SpMM of a GCN layer with input and output embedding dimensions of 256 on CPU. The fraction of sparse execution time is shown by the dotted contour lines.

A. GCN Profiling Setup

GCN inference was implemented in PyTorch-Geometric using the torch-sparse library. The characterization in this work used a three-layer GCN model, where the hidden feature dimensions were varied across experiments. Profiling on CPU was conducted on a dual-socket Intel(R) Xeon(R) Platinum 8380 with 40 cores per socket and 512 GB of main memory [15]. The highest instruction set available was AVX-512 with 2 AVX-512 FMA units. Profiling experiments on CPU were aligned to existing findings from [16] which used an NVIDIA-A100 GPU with 40 GB memory capacity and PCIe 4.0 between CPU host and GPU [17]. The CPU host was a dual-socket Intel(R) Xeon(R) Platinum 8358 with 32 cores per socket and 512 GB of main memory.

B. GCN Layer Execution Time Breakdown

For given embedding dimension(s), GCN layer performance can be modeled as a function of:

- *Scale* $|V|$: Determines capacity requirements and caching behavior of GCN kernels.
- *Sparsity* $\frac{|E|}{|V|^2}$: Affects the computational patterns and complexity of sparse operations.

The contour lines in Figure 2 represent the equivalence class of graphs that are expected to spend an equal fraction of the time in computing SpMM on the CPU. These lines were discovered through extensive experiments using RMAT graphs of uniform degree distributions with varied *scale* and *sparsity*. By marking the respective contour of a dataset, one can estimate the benefits of GCN execution on graph accelerators like PIUMA: datasets with a large fraction of sparse computation (SpMM) time are expected to benefit more from such accelerators. To the best of our knowledge, such analysis has not been shown in prior work.

The results in Figure 2 reveal for a given graph *scale*, the fraction of execution time spent in SpMM increases with the graph density. This is because the total number of non-zeros in $\tilde{\boldsymbol{A}}$ increases proportionally with density, while the dense matrix $\boldsymbol{W}$ has a fixed density.

Similarly, for a given graph *sparsity*, the fraction of execution time spent in SpMM increases with the graph *scale*. This is because the number of non-zeros in $\tilde{\boldsymbol{A}}$ increase quadratically

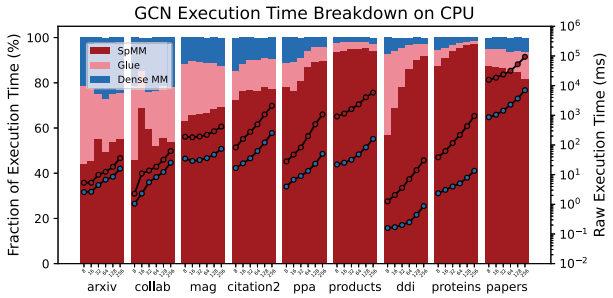

Fig. 3. Execution time breakdown on CPU for OGB workloads using a 3-layer GCN with hidden embedding dimensions ranging from 8 to 256 on orders of 2. Left Axis: Bar chart comparing percent execution times. Right Axis: Execution time of SpMM (red dots) versus Dense MM (blue dots).

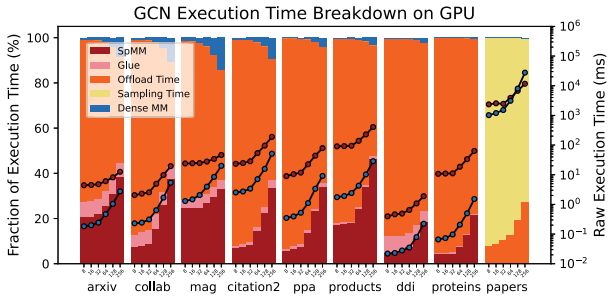

Fig. 4. Execution time breakdown on GPU [16]. All graphs except *papers* fit on a single-node GPU; thus *papers* required sampling on CPU. Offload time is the main contributor to execution time for graphs which fit on GPU while sampling is the main contributor for graphs which do not fit on GPU.

with vertices[1], whereas the complexity of Dense MM increases linearly.

OGB graphs *scale* and *sparsity* coordinates are annotated on Figure 2. For example, *arxiv* and *collab* are expected to spend less than 60% execution time in SpMM for a layer with embedding dimension 256. This plot provides insight into the potential benefits of running on PIUMA. PIUMA is expected to do well on workloads bottlenecked by sparse computation; thus *arxiv* and *collab* may benefit less from PIUMA, whereas *proteins* and *products* may benefit more from PIUMA.

C. Execution Time Breakdown of GCN on CPU and GPU

GCN execution time was analyzed using three categories; SpMM, Dense MM, and Glue Code. Glue Code comprises auxiliary function calls, such as activation functions, kernel initialization, and other PyTorch wrapper functions.

A sweep study over the hidden-layer embedding dimension was performed. From an application perspective, the embedding dimension is useful for tuning model accuracy. From an architectural perspective, the embedding dimension influences the ratio of sparse to dense compute, memory capacity requirements, memory bandwidth utilization, and end-to-end latency. The impact of embedding dimension has not been deeply analyzed from an architecture perspective in prior work.

[1] $|E| = \delta|V|^2$, where $|E|$, $|V|$ and δ denote the number of edges, vertices, and density of the graph, respectively.

The relative execution time of SpMM, Dense MM and Glue Code, and the absolute execution time of SpMM and Dense MM on CPU is shown in Figure 3. These results corroborate the per-layer estimation methodology in Section III-B.

On CPU, the SpMM kernel dominated the execution time of GCN, especially for large and/or dense datasets such as *ppa*, *products*, *ddi*, *proteins*, and *papers* where more than 80% of time was spent in SpMM. As the embedding dimension increased, caching either maintained or decreased the influence of Dense MM, while the memory bandwidth bottleneck worsened, which increased the influence of SpMM. The memory bandwidth bottleneck had a greater influence because larger embedding dimensions meant fewer vertex embeddings were cached, which increased off-chip communication.

For example, graphs such as *ddi* and *proteins* entirely fit in the CPU cache at low embedding dimensions, which explains why the fraction of execution time for SpMM increased with higher embedding dimensions. Graphs such as *products* and *papers* did not have as much benefit from caching at low embedding dimensions, therefore, the fraction of off-chip communication did not increase significantly at higher embedding dimensions. In the extreme case of *papers*, the impact of the Glue Code actually increased because the activation inputs were evicted from the cache after being computed.

Key Takeaway 1: *Caching benefits diminish with growing dataset size. PIUMA trades off L2/L3 cache resources for latency hiding massive thread-count and high memory bandwidth, offering a potentially better architecture for at-scale GCN computation.*

The GPU results from [16] are presented in Figure 4 for comparison. All graphs except *papers* fit entirely on GPU. To enable GPU characterization for *papers*, layer-wise sampling was employed. The sampling technique used full-neighborhood sampling to provide a fair comparison.

For non-sampled workloads, the clear performance bottleneck for GPU was the offload time of the adjacency matrix and vertex embeddings. Data offload is an unavoidable runtime contribution in inductive graph problems.

As the embedding dimension increases, both the Dense MM and SpMM kernels increase in influence on GPU. This is because the volume of data-movement between CPU and GPU does not change, as the embedding dimension sweep only applies to the hidden-layers.

Papers, which does not fit on GPU, was largely bottlenecked by sampling; more than 75% of the execution time was spent sampling on CPU. The combined impact of sampling and offloading was more than 99% of the execution time, leading to considerably lower performance than CPU for at-scale GCN inference.

Key Takeaway 2: *GPUs are most effective at accelerating workloads which require a compute first, high FLOPS to Bytes ratio design. PIUMA takes a fundamentally different approach, designed from a memory-first perspective with a low FLOPS to Bytes ratio. The intent of PIUMA was to meet growing demand in data-set capacity, sparsity, and communication, whereas GPUs were designed to meet growing demand in compute, leading to a communication bottleneck for at-scale GCN.*

IV. Characterization of SpMM on PIUMA

In the previous section, it was shown that GCNs are largely dominated by SpMM on CPUs. In this section, a detailed analysis of SpMM on PIUMA is presented. Two implementations of SpMM on PIUMA are evaluated to show that a scalable DMA implementation can overcome longer latencies and saturate bandwidth at scale.

The evaluation makes use of the PIUMA architecture simulator [5], [18]–[22] which simulates the timing of all instructions in the pipelines, engines, memory and network, based on the hardware specifications.

A. SpMM Kernel Analytical Model on PIUMA

A bandwidth-bound analytical model was used to estimate the performance of SpMM, which aided in architecting a high-performing SpMM kernel for PIUMA. SpMM is a low arithmetic intensity kernel [9] and is primarily bound by the memory performance. Therefore, a bandwidth-bound analytical model was used, which assumed no reuse of input feature vectors. This is a fair assumption since PIUMA does not use an L2 or L3 cache unlike conventional CPUs.

$$B_{CSR} = (|V| + 1) * B_R + |E| * B_C + |E| * B_N \quad (1)$$

$$B_{Feature} = K * |E| * B_F \quad (2)$$

$$B_{Write} = K * |V| * B_F \quad (3)$$

$$FLOP = 2 * |E| * K \quad (4)$$

Total reads generated from the adjacency matrix assumes a CSR storage format, corresponding to the row array (vertices in the graph), the column array (edges in the graph), and the non-zero value array (weight associated with each edge) as shown in Equation 1. The format B_X indicates the size in bytes of variable X where R is row index, C is column index, N is non-zero value, and F is a feature vector element.

Feature reads account for the memory traffic generated from embedding vectors. When a vertex is processed, the neighbors' feature vectors are aggregated in a partial sum vector. Thus, for every edge, one embedding vector of dimension K is read from the memory. The total number of bytes read from the dense feature matrix is given by Equation 2. For the bandwidth-bound analytical model, the optimal scenario was assumed where the cached partial sum vector was written back to memory only once (output matrix row) after all neighbors of the vertex were aggregated. Thus, the total write traffic is given by Equation 3. The FLOP count to process one edge is $2 * K$ because each vertex aggregation is a multiply and add (MAC) operation; an adjacent vertex embedding is scaled by a value and accumulated. The total FLOP count is given by Equation 4.

$$Time = \frac{(B_{CSR} + B_{Feature})}{BW_{Read}} + \frac{B_{Write}}{BW_{Write}} \quad (5)$$

To calculate the overall execution time, the total volume of data read and written is divided by the respective read and write bandwidths. The execution time from Equation 5 and the FLOP count from Equation 4 is used to calculate the expected throughput in FLOPS for the SpMM kernel.

Algorithm 2 SpMM Edge-Parallel

Input: CSR matrix $\tilde{\boldsymbol{A}}$ - row offset array $row \in \mathbb{Z}^{|V|}$, non-zero array $col \in \mathbb{Z}^{|E|}$, feature matrix $\boldsymbol{H}_{in} \in \mathbb{R}^{|V| \times K}$
Output: $\boldsymbol{H}_{out} \in \mathbb{R}^{|V| \times K}$

1: **procedure** SpMM
2: **for each** thread id $t \in \{1, 2, ...T\}$ **do in parallel**
3: $start, end \leftarrow \frac{t*|E|}{T}, \frac{(t+1)*|E|}{T}$
4: $u \leftarrow \arg\min i$ s.t. $row[i] \geq start$ ▷ *Binary Search*
5: $B[0 : K-1] \leftarrow 0$ ▷ *Accum. buffer*
6: **for each** $e \in \{start, ..., stop\}$
7: **if** $e \geq row[u+1]$ **then** ▷ *Vertex completed*
8: $\boldsymbol{H}_{out}[u, :] \leftarrow \boldsymbol{H}_{out}[u, :] + B[:]$ ▷ *Atomic write*
9: $u \leftarrow u + 1,\ B[:] \leftarrow 0$
10: $v \leftarrow col[e]$ ▷ *Neighbor vertex*
11: $B[:] \leftarrow B[:] + \tilde{\boldsymbol{A}}[u, v] * \boldsymbol{H}_{in}[v, :]$ ▷ *Accumulate*

B. SpMM Algorithms on PIUMA

As discussed in Section II-C, SpMM can use a vertex-parallel or edge-parallel algorithm. Three key tradeoffs exist between these strategies.

First, since the edge-parallel execution divides work (i.e., edges) across the column pointer, a binary search is required to find the first vertex/row in the row pointer assigned to a thread. Second, the edge-parallel algorithm requires atomic memory writes to prevent conflicts between threads updating the same vertex embedding. This is unlike the vertex-parallel algorithm where each vertex is assigned exclusively to only one thread. Finally, the vertex-parallel algorithm may exhibit load imbalance.

The performance bottleneck of atomic write-backs in the edge-parallel algorithm is addressed by PIUMA with its highly optimized remote atomic instructions and tolerance to memory latency. Therefore, the edge-parallel algorithm is an attractive option for PIUMA which can improve load balance.

The fundamental edge-parallel algorithm, detailed in Algorithm 2, uses a for-loop to iterate over the embedding values of each adjacent vertex embedding vector (Line 11), which is scaled and accumulated in the correct index of the accumulation buffer. PIUMA does not have an AVX unit for SIMD operations, therefore loop unrolling is employed to group memory accesses and pipeline MAC operations. The compiler is persuaded to unroll up to eight embedding values at once, requesting a fully aligned, 64-byte cache line, which is brought into the core's data cache.

This loop-unrolling method was evaluated using the PIUMA simulator and compared against the analytical model, revealing inefficiency in memory bandwidth utilization. As the number of cores increased, the memory bandwidth utilization decreased dramatically because the cores were waiting for NNZ (non-zero edge) reads which were on average $6\times$ higher latency for a 32-core system compared to a single-core system. Since the same scalar pipeline of MTPs that issued the memory operations also issued all other operations, with the default 8-byte load/store operation, the relative overhead of each NNZ read was significant when the memory latency increased.

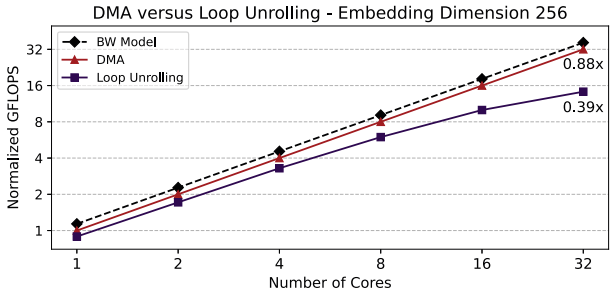

Fig. 5. Comparing SpMM algorithms on PIUMA to the bandwidth model for embedding dimension 256, normalized to single core DMA performance. The Direct Memory Access implementation (red) was within 85% percent of the bandwidth model performance while the Loop Unrolling implementation (purple) was challenged with scaling past 8 cores. These scaling trends were similar for embedding dimensions 8 and 64.

With elevated latency, while accessing remote memory slices at a finer granularity (8-byte), it was challenging to hide latency and saturate the memory bandwidth. This motivated a new solution using the DMA offload engine which allowed for requesting large chunks of memory at once, as well as perform in-memory add and multiply operations, freeing MTP pipelines to do other operations.

The PIUMA DMA controller provides specialized hardware support for offloading compute to the DMA engine. First, the DMA controller initializes a buffer of size embedding dimension K with the edge weight. Next, a DMA multiply operation atomically reads the feature vector from memory and multiplies it by the vectorized weight. Finally, the DMA engine executes a copy-add between the feature vector and the accumulation buffer. After processing all edges for a vertex, the DMA engine atomically writes the new embedding vector to memory. Using the DMA controller ensured high memory bandwidth utilization.

The strong scaling analysis shown in Figure 5 revealed that the DMA implementation was within 10-20% of the analytical model performance for embedding dimensions 8, 64 and 256, while the loop-unrolled version was less than 40% of the expected performance for higher numbers of cores. The DMA implementation of SpMM was used in subsequent studies.

Key Takeaway 1: *The DMA-based SpMM implementation fully saturates the available memory bandwidth and gives the highest performance on PIUMA.*

C. Latency Sensitivity of SpMM on PIUMA

The scalability beyond single-node PIUMA configurations is tightly coupled with the ability of the cores to tolerate longer network latency to remote memory slices. To validate the latency tolerance property of the pipelines, higher memory access latency scenarios were modeled by sweeping the DRAM access latency. Figure 6 (bottom) displays the simulated results of this sweep, for a varying number of cores and embedding dimensions. The embedding dimensions were purposely selected at the two extremes of this range in order to capture the worst-case scenarios. DRAM latency tolerance is

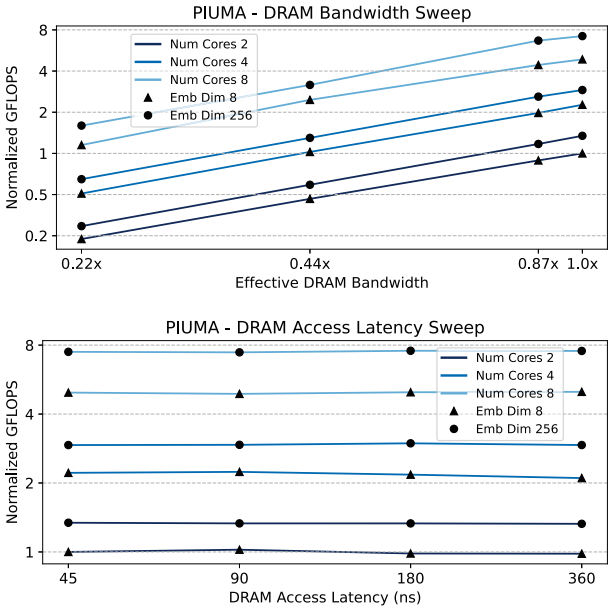

Fig. 6. Impact of DRAM bandwidth and latency across 2, 4 and 8 core PIUMA system for embedding dimensions 8 and 256. Top: Bandwidth sweep showing linear performance with respect to bandwidth. Bottom: Latency sweep showing latency-insensitivity up to 360 ns DRAM latency.

displayed for SpMM across a range of embedding dimensions and number of PIUMA cores.

To further investigate the reasons behind the latency insensitivity property of PIUMA, the latency sweep experiments were repeated with a varying number of threads per MTP, ranging from 1 to 16. One characteristic of the DMA engine is that DMA requests from threads belonging to the same core are directed to the same DMA engine and are serialized on the order of arrival. Consequently, if a thread could determine its DMA requests without having to wait for the sparse NNZs to be first fetched from DRAM, then it would be capable of saturating the DMA engine (which is latency tolerant) without the need for more threads in the same core.

The results displayed in Figure 7 (top) suggest that when the number of threads is reduced, the latency insensitivity property is lost for smaller embedding dimensions, while it is retained for higher embedding dimensions. This is attributed to the fact that for smaller embedding dimensions, fewer DMA requests are generated, thus the NNZ reads constitute a larger percentage of the total memory accesses. Regardless of the embedding dimension for all single-threaded cases, the latency of the NNZ reads appears on the critical path.

This means, that before being able to issue a new DMA request, the thread is blocked by the NNZ read, and the DMA is unused in this time window. In smaller leading dimensions, this time window is closer to the latency of the fine-grained DMA operation while in higher leading dimensions it is significantly smaller (refer to the PIUMA execution time breakdown displayed in Figure 8). Thus, when the latency increases for smaller embedding dimensions and the MTPs

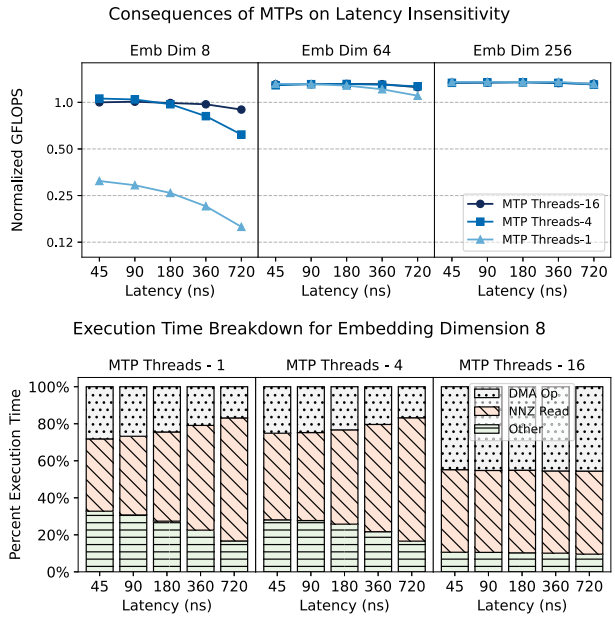

Fig. 7. An 8 core PIUMA system (1 die) was evaluated, sweeping the DRAM latency from 45 ns to 720 ns and the number of threads per MTP from 1 to 16. The default configuration uses 16 threads per MTP. Even extreme DRAM latency can be tolerated with sufficient MTP threads.

consist of a single thread, the DMA engines are underutilized and the performance is degraded.

Increasing the number of threads makes PIUMA more latency tolerant, especially for small embedding dimensions. This is because if a thread is blocked, waiting for the NNZ reads to complete, another thread can utilize the DMA engine providing full utilization of the DMA engine and available memory bandwidth (Figure 7 bottom).

Key Takeaway 2: *The PIUMA pipelines displayed significant DRAM latency tolerance for SpMM across a range of embedding dimensions and number of PIUMA cores. Using a large number of threads per MTP makes PIUMA highly latency tolerant, which especially benefits execution with smaller embedding dimensions.*

D. Bandwidth Sensitivity of SpMM on PIUMA

In order to highlight the ability of the PIUMA cores to saturate the available memory bandwidth, a DRAM bandwidth sweep study was performed. The results shown in Figure 6 (top) used the DMA implementation with 16 threads per MTP.

The system performance, as measured by the GFLOPs throughput, scales linearly as the available bandwidth of a single DRAM slice increases. The bandwidth is normalized with respect to the actual bandwidth provided by a single DRAM-slice memory controller. This behavior is preserved as the number of cores increases, suggesting that the network is fine-tuned and can provide adequate throughput for accesses that are directed to remote memory controllers.

Key Takeaway 3: *SpMM is limited by the memory bandwidth and not by the network interconnect bandwidth at scale on PIUMA.*

V. PIUMA Scalability versus CPU and GPU

In this section, the DMA-based SpMM kernel on PIUMA is compared to SpMM on CPU. Then, the performance of GCN on PIUMA is compared to CPU and GPU, revealing the speedup of one PIUMA node versus one CPU / GPU node, highlighting the scalability of PIUMA through an execution time breakdown.

A. Strong Scaling of SpMM on PIUMA versus CPU

SpMM is a memory bound kernel, therefore, a memory bandwidth comparison between CPU and PIUMA is an indicator for speedup. A stream benchmark was used to capture the effective memory bandwidth on the Xeon CPU. The control of threads and memory was maintained using *numactl* flags and *OpenMP* variables. The results of this bandwidth study are shown in Figure 8 (Left) which includes a comparison of PIUMA bandwidth. Notice the memory bandwidth of PIUMA exceeds CPU after $\sim$16 cores. Furthermore, the memory bandwidth for CPU decreased past 80 cores because the dual-socket Xeon only has 40 physical cores per socket; thus, more than 80 cores leads to hyper-threading which actually causes contention on the memory bandwidth.

Next, a strong scaling analysis was conducted using the DMA implementation of SpMM for PIUMA and an optimized CPU vertex-parallel implementation with dynamic load balancing using OpenMP. The edge-parallel algorithm was slower than vertex-parallel on CPU due to the overheads of atomic operations. The *products* graph was used for this study to show how at-scale graphs with minimal cache reuse would perform on CPU versus PIUMA.

Two parameters were swept in this study: the number of cores and the embedding dimension. The strong scaling study results using embedding dimension 256 are shown in the middle plot of Figure 8, while an execution time breakdown for an embedding dimension sweep is shown in the right plot.

The performance of SpMM on both CPU and PIUMA scale according to the available memory bandwidth for large graphs. This was expected as SpMM is a bandwidth bound kernel. At 16 cores, the PIUMA SpMM implementation is expected to be slightly higher performing than the CPU SpMM implementation according to the bandwidth comparison; however, Figure 8 (middle) shows that PIUMA is actually slightly lower performing than CPU.

This may be attributed to the fact that *products*, although large, can make use of the CPU caches. Frequently used feature vectors can be cached which reduces the DRAM read traffic and improves the performance of GCN inference on CPU. Compiler-aided vectorization and prefetching may have helped as well.

The results shown in Figure 8 (right) reveal the execution time attributed to reading non-zero values decreases as the embedding dimension increases. For embedding dimension 8, 2-NNZs are read for every 8 DMA reads and writes, while for embedding dimension 256, 2-NNZs are read for every 256 DMA reads and writes. This validates the latency sensitivity study in Section IV-C which revealed that the NNZ

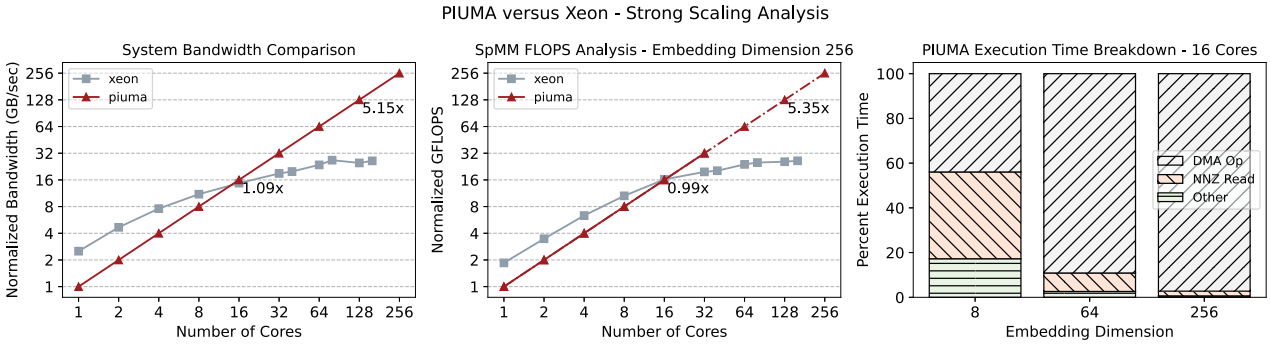

Fig. 8. A strong scaling analysis of SpMM on PIUMA versus Xeon using the *products* graph, normalized to single core PIUMA. Left: System bandwidth comparison. Middle: SpMM FLOPS comparison. Right: Execution time breakdown of a 16 core PIUMA system for three embedding dimensions.

reads were the primary reason for latency sensitivity and why embedding dimension 256 may need fewer threads per MTP than embedding dimension 8.

These results offer insight into the fundamental scalability challenges of CPU systems which are overcome by PIUMA. Traditional CPU systems such as Xeon can not scale their memory bandwidth by increasing the number of systems. Therefore, the scaling of SpMM on a shared-memory system is limited by the SoC memory-bandwidth. PIUMA does not have this limitation.

Key Takeaway 1: *PIUMA overcomes CPU scalability bottlenecks for SpMM by offering scalable memory bandwidth. As the number of nodes in a PIUMA system increases, the DGAS memory capacity and effective bandwidth increase proportionally. Thus, the performance of SpMM on PIUMA scales perfectly for SpMM while CPU systems must rely on distributed-memory solutions at-scale.*

Prior work has investigated distributed-memory solutions for sparse computation [12], [23]. Although this provides a means of scaling on CPU-based systems, communication overheads of MPI significantly reduces performance relative to an at-scale DGAS system [24].

B. GCN on PIUMA versus CPU and GPU

The expected performance of a single PIUMA node was compared to the performance of a Xeon CPU system. Equivalent GPU performance data from [16] was also included. This comparison is meaningful because it compared a single PIUMA node against two of the largest non-distributed memory CPU and GPU systems. The expectation is that PIUMA will scale well beyond the single-node comparison because of its scalable memory bandwidth and DGAS; therefore, this comparison is only a taste of what PIUMA is capable of for at-scale GCN acceleration.

Prior work has characterized the performance of Dense MM on PIUMA [21]. The observed peak FLOPS was used to calculate Dense MM time for GCN in this section. The results shown in the top plot of Figure 9 revealed the speedup of a single PIUMA node (red), the speedup of an A100-GPU (blue), normalized against a dual-socket Xeon CPU.

As the embedding dimension increased, the achieved speedup on PIUMA decreased, while the achieved speedup on GPU increased, relative to the CPU baseline. GPU speedup increased because it spent less time offloading data and more time computing. Since the input and output GCN dimensions do not change, the volume of data transferred between CPU and GPU does not change. However, since the hidden layer embedding dimensions increased, the GPU memory footprint requirement and amount of computation on GPU increased. GPUs have higher on-chip memory bandwidth than CPUs and higher compute capacity; thus, GPUs accelerate SpMM and Dense MM regions better than CPU, leading to improved performance. GPUs actually performed worse than CPUs for lower embedding dimensions due to the offloading overhead.

PIUMA speedup decreased with larger embedding dimensions because the amount of dense computation increased. Dense multiplication is a challenge for PIUMA as it does not have a SIMD unit. The execution time breakdown shown in Figure 10 offers deeper insights into why PIUMA performed worse at higher embedding sizes.

Workloads such as *arxiv*, *collab*, *mag*, *citation2* and *papers* spent over 75% execution time on Dense MM at an embedding dimension of 256. Even workloads such as *ppa* and *products* which had between $80-90\%$ percent SpMM on CPU were dominated by $50-60\%$ Dense MM on PIUMA for embedding dimension 256. PIUMA effectively accelerated SpMM; however, the dense computation demands at higher embedding dimensions limited PIUMA's performance.

Key Takeaway 2: *Increasing the embedding dimension tends to shift computational pressure of GCN from SpMM to Dense MM on PIUMA. A single PIUMA node always outperforms the CPU system due to its greatly improved sparse support, whereas a GPU has similar performance to PIUMA at large embedding dimension due to its strong Dense MM throughput.*

Understanding the implications of embedding size is important from an architecture perspective because systems such as PIUMA should be prepared to handle a spectrum of GCNs.

The GPU characterization for *papers* highlights the fundamental limitation of using GPUs for GCN. A threshold for GPU performance exists where a graph does not fit in GPU memory. Since the GPU relies on CPU sampling, the GPU performance degrades significantly. This memory-capacity limitation is a key reason why someone would desire a PIUMA system over a GPU system for at-scale GCN.

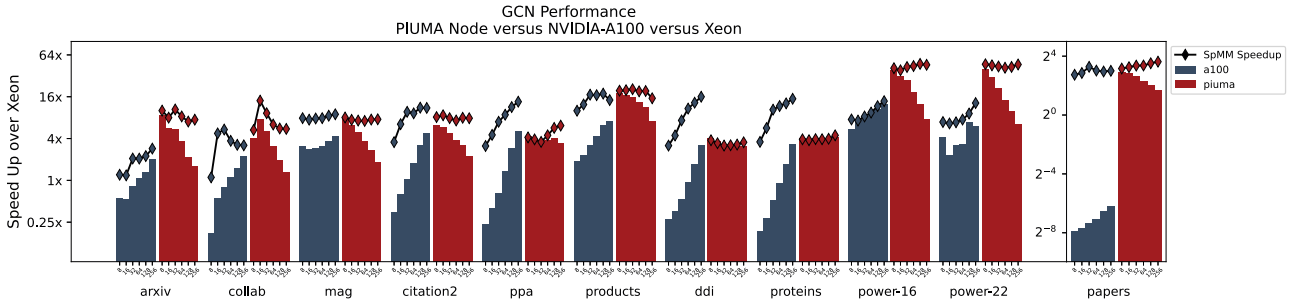

Fig. 9. Single-node performance of PIUMA against dual-socket performance of Xeon CPU and NVIDIA A100 GPU with embedding dimension sweep. PIUMA always outperformed CPU while GPU only outperformed CPU at higher embedding dimensions. Bars represent GCN speedup against Xeon while diamonds represent SpMM kernel speedup. PIUMA had similar SpMM speedups as GPU for large graphs (products), but significantly outperformed GPU on SpMM for graphs with low locality (power-16/power-22). GPU outperformed PIUMA on small graphs with good locality (ddi, proteins).

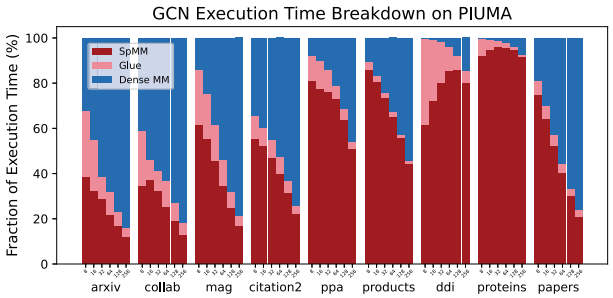

Fig. 10. Execution time breakdown for PIUMA, complimenting CPU and GPU results shown in Figure 3 and Figure 4.

Key Takeaway 3: *At-scale GCN is a challenge for GPUs because when a graph does not fit in GPU memory, CPU sampling and offload time causes a severe performance bottleneck. PIUMA does not require sampling and avoids this problem entirely through its DGAS, offering continued performance improvement through scalable memory bandwidth and Terabytes of memory capacity.*

VI. Discussion and Future Work

The execution of GCNs on PIUMA may be improved by changing how dense matrix multiplication is computed or the underlying algorithms.

Heterogeneous SoC: One approach could be to design a heterogeneous SoC combining PIUMA dies with dense compute accelerators that can improve the dense matrix multiplication performance on PIUMA. The ratio of PIUMA dies to dense units will largely depend on the application requirements.

Graph Partitioning: The DGAS of PIUMA may be beneficial for GNNs to avoid graph partitioning needed for large input graphs [10]. In distributed GNNs, a graph is partitioned so that each partition can fit in the local memory of each node. This graph partitioning is often done using a vertex cut or edge cut technique, which can be expensive. PIUMA's shared memory abstraction helps avoid such partitioning.

Graph Clustering and Sampling: Subgraph-based GCN training methods utilize graph clustering and sampling to create mini-batches [25], [26]. PIUMA can significantly accelerate graph clustering methods such as Louvain [27]. Neighborhood sampling-based GNN algorithms such as pinSAGE [28] and graphSAGE [29] often use random-walk for neighbor sampling. The random-walk algorithm is known to be latency bound, and PIUMA being latency optimized, has been shown to greatly accelerate random-walk over standard CPUs [5]. It would be interesting to explore the overall performance gains on PIUMA for such GCN methods during training.

VII. Related Work

Graphite presents a SW/HW co-design featuring layer fusion and DMA-based aggregation [9]. Layer-fusion demonstrated a 1.3x speedup for SpMM and is an interesting software optimization for PIUMA. Graphite's proposed DMA implementation is similar to the PIUMA DMA implementation. However, Graphite's DMA engine only supports aggregation; whereas, the PIUMA DMA engine is used for PIUMA specific operations and other basic arithmetic operations. The PIUMA DMA study is also distinguished through several in-depth sensitivity analyses.

Optimizations and characterizations for GNNs on GPU have also been investigated. GE-SpMM [11] proposed a method that uses Coalesced Row Caching to access sparse and dense data, improving the utilization of GPUs memory bandwidth. Characteristics of execution patterns of GCN training and inference on GPUs have been investigated as well [13], [14].

Several other works noted the importance of accelerating GCNs and proposed accelerator designs [30]–[32]. Although these accelerators demonstrated significant speedup over traditional computing methods, they are not comparable to PIUMA because they were not designed for at-scale graph analytics; thus, PIUMA holds a competitive edge for at-scale GCN.

VIII. Conclusion

This work characterized the scalability of GCN on the PIUMA architecture. Existing system bottlenecks in CPU and GPU systems were explained. The architectural implications of embedding dimension were discussed for both CPUs and GPUs, which lead to a discussion about why PIUMA may be beneficial for GCNs at scale.

An optimized DMA-based GCN implementation on PIUMA was developed and characterized, achieving up to 88% of the theoretical peak performance. Finally, a detailed comparison of performance scaling on PIUMA, CPUs and GPUs as a function of GCN architectural parameters was investigated.

REFERENCES

[1] W. Fan *et al.*, "A Graph Neural Network Framework for Social Recommendations," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 5, pp. 2033–2047, May 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9139346/

[2] S.-H. Wang, V. V. Govindaraj, J. M. Górriz, X. Zhang, and Y.-D. Zhang, "Covid-19 classification by FGCNet with deep feature fusion from graph convolutional network and convolutional neural network," *Information Fusion*, vol. 67, pp. 208–229, Mar. 2021. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1566253520303705

[3] S. Ye, J. Liang, R. Liu, and X. Zhu, "Symmetrical Graph Neural Network for Quantum Chemistry with Dual Real and Momenta Space," *The Journal of Physical Chemistry A*, vol. 124, no. 34, pp. 6945–6953, Aug. 2020. [Online]. Available: https://pubs.acs.org/doi/10.1021/acs.jpca.0c03201

[4] T. N. Kipf and M. Welling, "Semi-Supervised Classification with Graph Convolutional Networks," Feb. 2017, number: arXiv:1609.02907 arXiv:1609.02907 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1609.02907

[5] S. Aananthakrishnan *et al.*, "PIUMA: Programmable Integrated Unified Memory Architecture," *arXiv:2010.06277 [cs]*, Oct. 2020, arXiv: 2010.06277. [Online]. Available: http://arxiv.org/abs/2010.06277

[6] W. Hu *et al.*, "Open Graph Benchmark: Datasets for Machine Learning on Graphs," Feb. 2021, number: arXiv:2005.00687 arXiv:2005.00687 [cs, stat]. [Online]. Available: http://arxiv.org/abs/2005.00687

[7] J. Leskovec and R. Sosič, "SNAP: A General-Purpose Network Analysis and Graph-Mining Library," *ACM Transactions on Intelligent Systems and Technology*, vol. 8, no. 1, pp. 1–20, Oct. 2016. [Online]. Available: https://dl.acm.org/doi/10.1145/2898361

[8] 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 - SC '09*. Portland, Oregon: ACM Press, 2009, p. 1. [Online]. Available: http://dl.acm.org/citation.cfm?doid=1654059.1654101

[9] Z. Gong *et al.*, "Graphite: optimizing graph neural networks on CPUs through cooperative software-hardware techniques," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*. New York New York: ACM, Jun. 2022, pp. 916–931. [Online]. Available: https://dl.acm.org/doi/10.1145/3470496.3527403

[10] V. Md *et al.*, "DistGNN: scalable distributed training for large-scale graph neural networks," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. St. Louis Missouri: ACM, Nov. 2021, pp. 1–14. [Online]. Available: https://dl.acm.org/doi/10.1145/3458817.3480856

[11] 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*. Atlanta, GA, USA: IEEE, Nov. 2020, pp. 1–12. [Online]. Available: https://ieeexplore.ieee.org/document/9355302/

[12] O. Selvitopi *et al.*, "Distributed-memory parallel algorithms for sparse times tall-skinny-dense matrix multiplication," in *Proceedings of the ACM International Conference on Supercomputing*. Virtual Event USA: ACM, Jun. 2021, pp. 431–442. [Online]. Available: https://dl.acm.org/doi/10.1145/3447818.3461472

[13] Z. Zhang *et al.*, "Architectural Implication of Graph Neural Networks," *IEEE Computer Architecture Letters*, pp. 1–1, 2020. [Online]. Available: https://ieeexplore.ieee.org/document/9075391/

[14] M. Yan *et al.*, "Characterizing and Understanding GCNs on GPU," *IEEE Computer Architecture Letters*, vol. 19, no. 1, pp. 22–25, Jan. 2020. [Online]. Available: https://ieeexplore.ieee.org/document/8976117/

[15] "Intel® Xeon® Platinum 8380 Processor." [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/212287/intel-xeon-platinum-8380-processor-60m-cache-2-30-ghz/specifications.html

[16] M. Adiletta, D. Brooks, and G.-Y. Wei, "Architectural Implications of Embedding Dimension during GCN on CPU and GPU," 2022, publisher: arXiv Version Number: 1. [Online]. Available: https://arxiv.org/abs/2212.00827

[17] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky, "NVIDIA A100 Tensor Core GPU: Performance and Innovation," *IEEE Micro*, vol. 41, no. 2, pp. 29–35, Mar. 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9361255/

[18] S. Eyerman, W. Heirman, Y. Demir, K. Du Bois, and I. Hur, "Projecting Performance for PIUMA using Down-Scaled Simulation," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. Waltham, MA, USA: IEEE, Sep. 2020, pp. 1–7. [Online]. Available: https://ieeexplore.ieee.org/document/9286184/

[19] B. Seshasayee, J. Fryman, and I. Hur, "Hash Table Scalability on Intel PIUMA," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. Waltham, MA, USA: IEEE, Sep. 2020, pp. 1–2. [Online]. Available: https://ieeexplore.ieee.org/document/9286204/

[20] K. Lakhotia, F. Petrini, R. Kannan, and V. Prasanna, "Accelerating Allreduce With In-Network Reduction on Intel PIUMA," *IEEE Micro*, vol. 42, no. 2, pp. 44–52, Mar. 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9665261/

[21] J. J. Tithi, F. Checconi, D. Doerfler, and F. Petrini, "SU3 Bench on a Programmable Integrated Unified Memory Architecture (PIUMA) and How that Differs from Standard NUMA CPUs," in *High Performance Computing*, A.-L. Varbanescu, A. Bhatele, P. Luszczek, and B. Marc, Eds. Cham: Springer International Publishing, 2022, vol. 13289, pp. 65–84, series Title: Lecture Notes in Computer Science.

[22] J. J. Tithi and F. Petrini, "A New Parallel Algorithm for Sinkhorn Word-Movers Distance and Its Performance on PIUMA and Xeon CPU," Apr. 2022, number: arXiv:2107.06433 arXiv:2107.06433 [cs]. [Online]. Available: http://arxiv.org/abs/2107.06433

[23] V. Bharadwaj, A. Buluç, and J. Demmel, "Distributed-Memory Sparse Kernels for Machine Learning," Mar. 2022, number: arXiv:2203.07673 arXiv:2203.07673 [cs]. [Online]. Available: http://arxiv.org/abs/2203.07673

[24] F. McSherry, M. Isard, and D. G. Murray, "Scalability! But at what COST?" in *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. Kartause Ittingen, Switzerland: USENIX Association, May 2015. [Online]. Available: https://www.usenix.org/conference/hotos15/workshop-program/presentation/mcsherry

[25] W.-L. Chiang *et al.*, "Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. Anchorage AK USA: ACM, Jul. 2019, pp. 257–266. [Online]. Available: https://dl.acm.org/doi/10.1145/3292500.3330925

[26] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "GraphSAINT: Graph Sampling Based Inductive Learning Method," 2019, publisher: arXiv Version Number: 4. [Online]. Available: https://arxiv.org/abs/1907.04931

[27] J. J. Tithi, A. Stasiak, S. Aananthakrishnan, and F. Petrini, "Prune the Unnecessary: Parallel Pull-Push Louvain Algorithms with Automatic Edge Pruning," in *49th International Conference on Parallel Processing - ICPP*. Edmonton AB Canada: ACM, Aug. 2020, pp. 1–11. [Online]. Available: https://dl.acm.org/doi/10.1145/3404397.3404455

[28] R. Ying *et al.*, "Graph Convolutional Neural Networks for Web-Scale Recommender Systems," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, Jul. 2018, pp. 974–983, arXiv:1806.01973 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1806.01973

[29] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive Representation Learning on Large Graphs," Sep. 2018, number: arXiv:1706.02216 arXiv:1706.02216 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1706.02216

[30] J. Li, H. Zheng, K. Wang, and A. Louri, "SGCNAX: A Scalable Graph Convolutional Neural Network Accelerator with Workload Balancing," *IEEE Transactions on Parallel and Distributed Systems*, pp. 1–1, 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9645224/

[31] X. Chen *et al.*, "Rubik: A Hierarchical Architecture for Efficient Graph Neural Network Training," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 4, pp. 936–949, Apr. 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9428002/

[32] J. Li, A. Louri, A. Karanth, and R. Bunescu, "GCNAX: A Flexible and Energy-efficient Accelerator for Graph Convolutional Neural Networks," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. Seoul, Korea (South): IEEE, Feb. 2021, pp. 775–788. [Online]. Available: https://ieeexplore.ieee.org/document/9407104/