MeshSlice: Efficient 2D Tensor Parallelism for Distributed DNN Training

Hyoungwook Nam
University of Illinois at Urbana-Champaign
Champaign, Illinois, USA
hn5@illinois.edu

Gerasimos Gerogiannis
University of Illinois at Urbana-Champaign
Champaign, Illinois, USA
gg24@illinois.edu

Josep Torrellas
University of Illinois at Urbana-Champaign
Champaign, Illinois, USA
torrella@illinois.edu

Abstract

In distributed training of large DNN models, the scalability of one-dimensional (1D) tensor parallelism (TP) is limited because of its high communication cost. 2D TP attains extra scalability and efficiency because it reduces communication relative to 1D TP. Unfortunately, existing algorithms for general matrix multiplication (GeMM) in 2D TP suffer from inefficiencies. Indeed, Cannon's algorithm incurs high traffic, SUMMA suffers from high synchronization overhead, and a 2D GeMM with collective communication operations does not overlap communication with computation. In addition, it is difficult to optimize the numerous parameters of 2D TP, including the dataflow, mesh shape, and sharding. As a result, human experts are needed to find efficient configurations of 2D TP. To address these problems, this paper proposes *MeshSlice*, a novel 2D GeMM algorithm for efficient 2D TP in distributed DNN training. The MeshSlice algorithm slices the collective communications into multiple partial collectives that allow overlapping communication with computation. As a result, MeshSlice hides most of the communication latency. We also present the *MeshSlice LLM autotuner*, which automates finding an efficient 2D GeMM dataflow configuration, mesh shape, and communication granularity for Large Language Model (LLM) training using analytical cost models. To evaluate MeshSlice, we simulate TPUv4 clusters training LLM models. We show that MeshSlice maintains good efficiency up to at least 256-way 2D TP. In a cluster of 256 TPUs, MeshSlice trains the GPT-3 and Megatron-NLG models 12.0% and 23.4% faster, respectively, than the state-of-the-art algorithm.

CCS Concepts

• **Computing methodologies → Distributed algorithms**; **Neural networks**.

Keywords

Large-scale Machine Learning, Large Language Model, Distributed Training, Collective Communication

ACM Reference Format:
Hyoungwook Nam, Gerasimos Gerogiannis, and Josep Torrellas. 2025. MeshSlice: Efficient 2D Tensor Parallelism for Distributed DNN Training. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3695053.3731077

This work is licensed under a Creative Commons Attribution 4.0 International License.
ISCA '25, Tokyo, Japan
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1261-6/25/06
https://doi.org/10.1145/3695053.3731077

1 Introduction

Deep Neural Network (DNN) models are growing in size dramatically – especially, transformer-based [31] Large Language Models (LLMs). These models must be trained using a distributed cluster, not only to satisfy their massive computing demands but also to hold their large memory footprint.

To parallelize a DNN running on a distributed cluster, three approaches exist: data parallelism (DP) [7, 37], pipeline parallelism (PP) [12], and tensor parallelism (TP) [16, 20]. Due to the large computing and memory demands of LLMs, most LLM training methods use all three types of parallelism together, forming 3D training clusters [5, 8, 22, 27]. Of the three types of parallelism, TP usually has the least parallelism because it incurs the most communication cost. This is because TP requires communicating the input or output matrix shards at every general matrix multiplication (GeMM), and GeMMs account for the majority of DNN computations. To continue to increase the size of DNN models, it is necessary to scale all three types of parallelism and, in particular, resolve the communication bottleneck of TP.

One way to mitigate the communication cost of TP is to make it two dimensional (2D). 2D TP distributes matrix shards into a 2D mesh of accelerators, and performs 2D distributed GeMM computations. In a 2D distributed GeMM, a shard of a matrix is communicated only to the accelerators in the same row or column of the 2D mesh. As a result, 2D GeMM incurs less traffic than 1D GeMM, where a matrix shard must be communicated to all accelerators. Consequently, by replacing 1D TP with 2D TP, one can attain higher parallelism at a similar communication cost, or the same parallelism at a smaller communication cost. Additionally, a higher degree of parallelism via 2D TP reduces the communication overhead for DP, as each accelerator holds smaller weight matrix shards.

In practice, optimizing 2D TP is a challenging problem that requires careful architectural considerations. The first difficulty is to find an efficient 2D GeMM algorithm, as existing algorithms suffer from inefficiencies. Specifically, Cannon's algorithm [4] is a traditional method for 2D GeMM that incurs high traffic because it requires skewing the matrix shards, and only supports square meshes. The SUMMA algorithm [30] has less traffic than Cannon by choosing an optimal mesh shape, but relies on fine-grain *broadcast* and *reduce* communication operations. These primitives pipeline the communications into small packet transfers, which result in many synchronizations. This is inefficient in the high-bandwidth inter-chip interconnects (ICI) of contemporary machine learning (ML) clusters with a large number of accelerators.

Google's TPU clusters compute 2D GeMMs using *AllGather* (AG) and *ReduceScatter* (RdS) collective communication operations [35]. They fully utilize the ICI bandwidth via high communication parallelism. However, this approach combines all partial communications into a single collective communication. As a result, it is not possible to overlap communications with computations via software pipelining. Wang et al. [34] partially solve this problem by partitioning a collective communication into multiple *SendRecv* operations to overlap communications with computations. 2D GeMM involves communication operations in two directions (i.e., row and column). Unfortunately, Wang's partitioning method can only be applied to one direction; applying the partitioning to both directions requires using Cannon's algorithm.

The second difficulty of 2D TP is to find the optimal configurations of the many hyperparameters in the 2D GeMM algorithm. The hyperparameters that affect the communication cost of 2D GeMM include how matrices are sharded in a mesh (i.e., the *sharding*), how each shard moves in the mesh of accelerators (i.e., the dataflow), and what the row and column sizes are (i.e., the mesh shape). Currently, finding the optimal configuration of these knobs requires expert knowledge and repeated experiments.

To address these challenges, this paper introduces *MeshSlice*, a new, efficient distributed GeMM algorithm for 2D TP. MeshSlice *slices* the AG/RdS collective communication operations into multiple partial AG/RdS operations in both row and column directions. With this approach, MeshSlice overlaps most of the communications with computations—unlike Wang's algorithm. Further, unlike Cannon, MeshSlice supports different mesh shapes to minimize the traffic cost. Finally, unlike SUMMA, MeshSlice uses efficient AG and RdS operations for high ICI bandwidth utilization.

We also introduce the *MeshSlice LLM autotuner*, which automatically optimizes MeshSlice's hyperparameters for distributed LLM training. The autotuner finds the hyperparameters in two phases. It begins by choosing an efficient 2D GeMM dataflow, and then uses analytical cost models to co-optimize the mesh shape and the communication granularity.

We evaluate MeshSlice and multiple baseline algorithms by simulating TPUv4 [13] clusters training the GPT-3 [3] and Megatron-NLG [27] LLM models. MeshSlice maintains high efficiency up to at least 256-way 2D TP. In a cluster of 256 TPUs, MeshSlice trains the GPT-3 and Megatron-NLG models 12.0% and 23.4% faster, respectively, than the state-of-the-art. We also develop an implementation of MeshSlice running on Google TPUv4 clusters. The resulting measurements show that MeshSlice's slicing incurs only a small overhead, and that the autotuner's cost models can accurately estimate the communication and compute costs.

This paper makes the following contributions:

- The novel MeshSlice algorithm for 2D TP, which solves major inefficiency problems in existing 2D GeMM algorithms.
- The MeshSlice LLM autotuner, which finds an efficient combination of dataflow, mesh shape, and communication granularity.
- The implementation of MeshSlice and its evaluation on simulated and real TPU clusters.

2 Background on Distributed DNN Training

2.1 Distributed Training Methods

Large-scale DNNs, especially LLMs, are typically trained in distributed systems of computing devices. In this paper, we will refer to a computing device as a chip, which can be an ML accelerator, a GPU, or a CPU. There are three major types of parallelism when distributing DNN computations among chips: data, pipeline, and tensor parallelism. To maximize the scalability of training, contemporary LLM training methods form 3D networks of chips by leveraging all three types of parallelism together [5, 22, 27, 29].

Data parallelism (DP) partitions the input data among different chips [7]. Because the DNN parameters (weights and biases) are replicated among the chips, the only communication happens when the parameters are updated and synchronized. The communication cost of DP can be effectively hidden because the parameter update communication of one layer can be done in parallel with the computation of another layer.

Pipeline parallelism (PP) gives different DNN layers to different chips [12]. The communication happens only at the boundary between pipeline stages. The scalability of PP is inherently limited by the network structure of the DNN model, and the overhead of PP increases with the number of pipeline stages.

Tensor parallelism (TP) partitions all matrices (weight, input, and output) of a DNN layer among different chips. Because all matrices are partitioned, TP requires the least memory footprint, but incurs the most communication traffic out of the three types of parallelism. TP generates communication traffic in every GeMM computation. In LLM training, this communication traffic is generated by the fully-connected (FC) layers [16].

In 1D TP GeMMs, the weight matrix is partitioned by either its input or output dimension into a 1D network of chips. In the former case, the input and weight shards are multiplied to compute the partial outputs, and the partial outputs are accumulated via a ReduceScatter (RdS) communication. In the latter case, the input shards are collected from all chips via an AllGather (AG) communication, and are then multiplied by the weight shards.

Unfortunately, the scalability of 1D TP is limited because the communication traffic grows linearly with the number of chips. This is because either the output shard must be accumulated across all chips during RdS, or the input shard must be copied to all chips during AG. Therefore, to achieve linear performance scaling with 1D TP, the communication bandwidth must scale quadratically with the number of chips. One example solution is NVIDIA's NVSwitch [19], which connects 8 GPUs in a fully-connected ICI network. This approach has limited scalability because it becomes quadratically harder to build a fully-connected switch with a larger number of GPUs. Consequently, most 3D LLM training clusters limit TP to 8-way [5, 8, 27, 29].

2.2 2D Tensor Parallelism

A more scalable solution for TP is 2D TP, where the matrices are partitioned among chips organized in a 2D mesh (connected as a 2D torus). Each chip locally holds one shard of each matrix. Then, the FC layers use a 2D distributed GeMM algorithm.

In 2D GeMM, each shard is communicated only to the chips in the same row or in the same column—unlike in 1D distributed

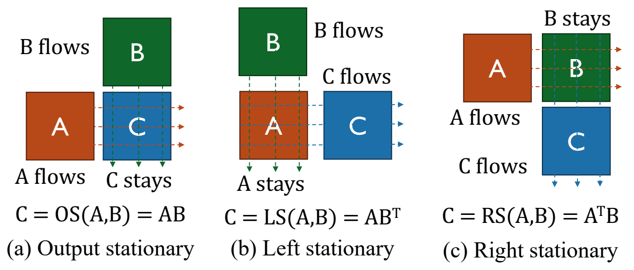

Figure 1: Three dataflows of 2D GeMM algorithms.

GeMM, where a shard is communicated to all other chips. This is possible because computing one element of the output matrix only depends on one row of the left input matrix and one column of the right input matrix. Therefore, 2D TP incurs less communication traffic than 1D TP. As a result, 2D TP is more scalable for a given communication cost, or has lower communication cost for a given number of chips. Moreover, the hardware cost to build a large mesh is lower than the cost to build a large fully-connected network.

The extra scalability of 2D TP can be exploited in different ways. For example, Llama 3 is trained using a 16K GPU cluster exploiting 3D parallelism (DP+PP+TP) with only 8-way 1D TP [8]. If we leverage 128-way 2D TP instead of 8-way 1D TP, we can build a 16× larger cluster with 256K chips. This not only attains a larger scale but also reduces the communication cost of DP. Indeed, because each chip now holds a shard of 1/128th of the weight matrix instead of 1/8th, the per-chip DP traffic is 16× smaller than before.

Alternatively, we can keep the same total number of chips, apply 128-way 2D TP, and decrease the degrees of both DP and PP by 4×. In this cluster, the per-chip DP traffic is 64× smaller than in the cluster using the 8-way 1D TP. As a result of the reduced communication cost and fewer pipeline stages, we can expect a better compute utilization than in the original cluster.

2.3 2D GeMM Algorithms

2.3.1 General Aspects. 2D GeMM is the heart of 2D TP in distributed DNN training. While there are a variety of 2D GeMM algorithms, they share many common aspects. Assume that we compute the output matrix $C = AB$ as the multiplication of the left input A and the right input B. All three matrices are partitioned in both their row and column dimensions into shards, and the shards are assigned to a 2D mesh of chips. That is, in a mesh with P_r rows and P_c columns, A is partitioned into the shards (i.e., sub-matrices) $A_{00} \ldots A_{(P_r-1)(P_c-1)}$. Then, A_{ij} is stored in chip (i, j) at the i-th row and j-th column of the mesh. The same applies to B and C.

2D GeMM algorithms can compute GeMMs in three possible dataflows [30, 36] as illustrated in Figure 1. In each dataflow, the shards of one of the three matrices (A, B, C) remain *stationary* in their chips, and the shards of the other two are communicated through either the vertical (inter-row) direction or the horizontal (inter-column) direction.

In the output-stationary (OS) dataflow (Figure 1a), the output C is stationary, each shard of A is transferred to the chips in the same row (inter-column), and each shard of B is transferred to the chips in the same column (inter-row). This results in $C = AB$. In the left-stationary (LS) dataflow (Figure 1b), the left input A is stationary, each shard of the right input B is transferred to the chips in the same column, and each shard of the output C is transferred and accumulated into the chips in the same row. This results in $C = AB^\top$. Finally, the right-stationary (RS) dataflow (Figure 1c) is the symmetric version of the LS dataflow, resulting in $C = A^\top B$. LS and RS dataflows are equivalent to input-stationary and weight-stationary dataflows in systolic arrays [24], respectively.

The 2D GeMM communication traffic depends on the dataflow and the shape of the mesh. Assume that we have a mesh of P_r rows and P_c columns and two matrices M_r and M_c that flow in the inter-row direction (i.e., vertically) and in the inter-column direction (i.e., horizontally), respectively. Each matrix shard must be communicated to all other chips in either its row or its column. We can compute the time taken to transfer the shards (excluding synchronization and other overheads) as follows. For the inter-row (i.e., vertical) transfers, the time is $(P_r - 1) \times sizeof(M_r)/(P_r \times P_c)/BW_{row}$; for the inter-column (i.e., horizontal) transfers, it is $(P_c - 1) \times sizeof(M_c)/(P_r \times P_c)/BW_{col}$. Here, BW_{row} is inter-row link bandwidth, and BW_{col} is inter-column link bandwidth. We refer to these times as the *traffic costs*. The 2D GeMM traffic cost is the maximum of the inter-row and inter-column traffic costs, as we need to wait until the communications in both directions complete.

If $BW_{row} = BW_{col}$, the traffic cost is minimized when $(P_r - 1)/(P_c - 1) = sizeof(M_c)/\ sizeof(M_r)$. However, the values of P_r and P_c that minimize the traffic cost may not minimize the overall communication cost due to synchronizations and other overheads.

2.3.2 Cannon's Algorithm. Cannon's algorithm [4] is one of the first 2D GeMM algorithms. It only works for square meshes. Before the computation begins, the matrix shards are shifted in a skewed manner. Then, Cannon systolically shifts the shards (using SendRecv communication operations) while computing the partial multiplications. Cannon is the base algorithm for systolic arrays [14] and 3D/2.5D GeMM algorithms [1, 28].

The major limitation of Cannon is that it has a higher traffic cost than other 2D GeMM algorithms for two reasons. First, skewing the matrix shards at the beginning incurs extra communication traffic that is not required in other algorithms. Second, while different mesh shapes change the traffic cost of 2D GeMM algorithms, Cannon only works for square meshes. Hence, Cannon incurs a higher traffic cost than other 2D GeMM algorithms when the matrix shapes are significantly imbalanced.

2.3.3 SUMMA Algorithm. SUMMA [30] solves the two limitations of Cannon in that it does not need to skew the matrix shards and it can support any mesh shapes. The pseudocode of SUMMA for different dataflows is shown in Figure 2a. SUMMA splits the matrices into $P \times P$ shards, where P is a common multiple of P_r and P_c. Then, it performs communications and computations in a loop of P iterations. SUMMA uses broadcast (bcast) and reduce communication operations on shards across the same row or column. Because each row or column of a 2D torus is connected in a ring topology, SUMMA runs ring bcast and reduce algorithms.

For example, assume that we run the SUMMA LS algorithm (Figure 2a, center) in a $P \times P$ mesh. A_{ij} is a shard of A located in the chip at the i-th row and j-th column of the mesh. In the

SUMMA OS

```
for p = 0…(P-1):
  $A' = bcast_{col}(A_{ip})$
  $B' = bcast_{row}(B_{pj})$
  $C_{ij}$ += $A'B'$
```

SUMMA LS

```
for p = 0…(P-1):
  $B' = bcast_{row}(B_{pj})$
  $C' = A_{ij}(B')^T$
  $reduce_{col}(C', C_{ip})$
```

SUMMA RS

```
for p = 0…(P-1):
  $A' = bcast_{col}(A_{ip})$
  $C' = (A')^T B_{ij}$
  $reduce_{row}(C', C_{pj})$
```

(a) SUMMA

Collective OS

```
$A_{i*} = AG_{col}(A_{ij})$
$B_{*j} = AG_{row}(B_{ij})$
$C_{ij} = A_{i*}B_{*j}$
```

Collective LS

```
$B_{*j} = AG_{row}(B_{ij})$
$C'_{i*} = A_{ij}(B_{*j})^T$
$C_{ij} = RdS_{col}(C'_{i*})$
```

Collective RS

```
$A_{i*} = AG_{col}(A_{ij})$
$C'_{*j} = (A_{i*})^T B_{ij}$
$C_{ij} = RdS_{row}(C'_{*j})$
```

(b) Collective

Figure 2: Pseudocode of SUMMA and Collective 2D GeMM algorithms for the three dataflows. In the code, expressions with *row* as subscript are inter-row communications in the same column, while those with *col* are inter-column communications in the same row. A_{ij} is a shard of A located in the chip at i-th row and j-th column of the mesh.

p-th iteration, the following occurs. First, the chips (p,j) for all j broadcast their B_{pj} shards to the chips in the j-th column. Then, in all chips, the local A_{ij} shard is multiplied with the transpose of B_{pj}, producing the partial result C'. Finally, for each row i, the partial results C' in all the chips in the row are reduced to the shard C_{ip} in the chip (i,p) in the p-th column.

Unfortunately, SUMMA's one-to-all bcast and all-to-one reduce communications are inefficient in a large mesh connected with high-bandwidth network links. Consider a bcast, shown in the left part of Figure 3. To utilize all the links in a ring (a row or column) during the transfer, the shard to be broadcasted is broken down into D packets that are streamed over the ring as fine-grain transfers. The streaming is done in $P+D-1$ pipeline stages. There are two sources of overhead: bubbles in the pipeline and synchronizations. Each link suffers $P-1$ bubbles—some at the beginning and some at the end of the transfer. Further, each pipeline stage requires a synchronization and therefore the broadcast needs $P+D-1$ synchronizations. The reduce operation has the same communication pattern and suffers from the same overheads. Since there are P iterations in SUMMA, the total synchronization overhead grows as $O(P^2)$.

2.3.4 Collective 2D GeMM. To avoid the SUMMA's overheads, a popular approach is to perform the 2D GeMM using the AllGather (AG) and ReduceScatter (RdS) [25, 35] collective communication operations. AG involves the parallel execution of all the per-column or per-row bcasts. Likewise, RdS is the parallel execution of all the per-column or per-row reduce operations. We call this approach *Collective* 2D GeMM.

For each dataflow of the SUMMA algorithm, there is a Collective 2D GeMM counterpart [25]. The algorithms for the three dataflows are shown in Figure 2b. For example, in the LS dataflow (Figure 2b, middle), multiple $bcast_{row}$ operations in SUMMA LS (Figure 2a, middle) are merged into a single AG_{row} operation, and multiple $reduce_{col}$ operations in SUMMA LS are merged into a single RdS_{col} operation. The computation is also done in a single step rather than in P iterations as in SUMMA.

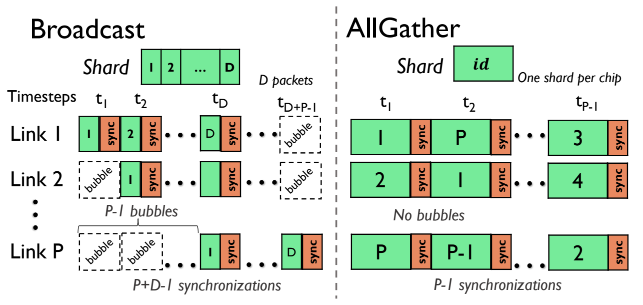

Figure 3: Communication patterns of Broadcast and All-Gather operations in a P-chip ring.

Using collective AG/RdS communications solves the inefficiency problems of the bcast/reduce operations. Consider the AG operation in a ring (a row or a column) of P chips, as shown in the right part of Figure 3. In each of the $P-1$ steps, each link transfers an entire shard to a neighbor. Hence, compared to bcast, AG eliminates pipeline bubbles, transfers larger packets, and invokes fewer synchronizations. The same is true for RdS compared to reduce. Moreover, since Collective 2D GeMM only calls AG or RdS once per each direction, its total synchronization overhead grows as $O(P)$. Hence, AG/RdS are more efficient and attain higher bandwidth utilization than bcast/reduce.

The major limitation of Collective 2D GeMM is that it cannot *overlap* communications with computations. One cannot apply software pipelining to overlap them because there are no loop iterations, and there are true dependencies between the computation and the communication operations.

Wang et al. [34] present a partial solution to this problem by splitting the collective communication in one direction into multiple *SendRecv* communications. Then, by applying software pipelining, the SendRecv communications are overlapped with the partial GeMM computations. Such a 2D GeMM is equivalent to a combination of FSDP [37] and 1D TP. However, this solution can partition and overlap only the communication operation in one direction; it cannot overlap the communication operation in the other direction. To partition the AG/RdS operations into multiple SendRecv operations in both directions, one needs to use Cannon, whose limitations are discussed in Section 2.3.2.

3 2D Tensor Parallelism with MeshSlice

In this work, we make two contributions to 2D TP. First, we propose a new 2D GeMM algorithm that solves the limitations of existing 2D GeMM algorithms. Second, we design an LLM autotuner that finds an efficient 2D TP configuration for LLM training. The LLM autotuner optimizes the configuration of the dataflow, mesh shape, and communication granularity.

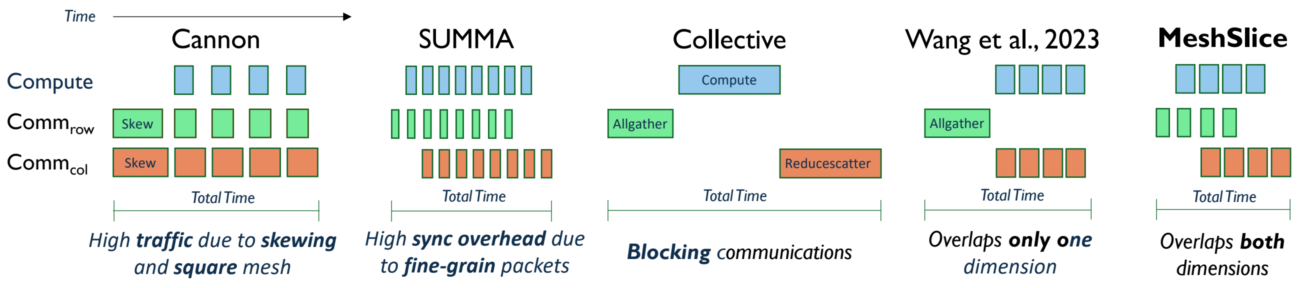

Figure 4: Comparing the timelines of five 2D GeMM algorithms: Cannon, SUMMA, Collective, Wang, and MeshSlice.

Our proposed 2D GeMM algorithm is called MeshSlice. Figure 4 visualizes the timelines of the previous algorithms, and compares them to MeshSlice. The figure shows the time progression of the computation, inter-row communications, and inter-column communications. Cannon requires skewing and only supports square mesh shapes. As a result, it has higher traffic than the other algorithms, which increases overall execution time. SUMMA uses inefficient bcast/reduce communication operations, which incur pipeline bubbles and synchronization overhead due to fine-grained packets. Collective does not overlap the collective communications with computation. Wang's algorithm partitions the collective communication in only one direction, so it leaves the communication in the other direction non-overlapped. Finally, MeshSlice is able to overlap the communication with computation in both directions and attains the fastest execution.

MeshSlice OS

```
for s = 0…(S-1):
  $A_s$=$slice_{col}$($A_{ij}$,S,s)
  $B_s$=$slice_{row}$($B_{ij}$,S,s)
  A'= $AG_{col}$($A_s$)
  B'= $AG_{row}$($B_s$)
  $C_{ij}$+= A'B'
```

MeshSlice LS

```
for s = 0…(S-1):
  $B_s$=$slice_{row}$($B_{ij}$,S,s)
  $C_s$=$slice_{col}$($C_{ij}$,S,s)
  B'= $AG_{row}$($B_s$)
  C'= $A_{ij}$(B')$^T$
  $C_s$= $RdS_{col}$(C')
```

MeshSlice RS

```
for s = 0…(S-1):
  $A_s$=$slice_{col}$($A_{ij}$,S,s)
  $C_s$=$slice_{row}$($C_{ij}$,S,s)
  A'= $AG_{col}$($A_s$)
  C'= (A')$^T$$B_{ij}$
  $C_s$= $RdS_{row}$(C')
```

Figure 5: Pseudocodes of the MeshSlice 2D GeMM algorithm in the three dataflows.

3.1 The MeshSlice 2D GeMM Algorithm

The MeshSlice algorithm has three characteristics. First, it partitions and overlaps communication operations in both directions. Second, it uses efficient AG and RdS communication operations instead of bcast, reduce, or SendRecv operations. Finally, it supports any mesh shape and a flexible granularity of communication.

To overlap communications with computations, we need to *partition* the collective communications into smaller communications and apply software pipelining. There are two existing methods to partition collective AG/RdS operations. The first one is to break them down into multiple SendRecv communications. Wang's algorithm [34] applies this approach to a single direction, and applying it to both directions requires using Cannon's algorithm [4]. The second method is to partition AG and RdS into multiple bcast and reduce operations, respectively. This approach results in SUMMA [30].

Instead, MeshSlice introduces a *new* partitioning method: partitioning AG and RdS operations into *partial* AG and RdS operations. The core of this algorithm is *slicing* the matrix shards into S sub-shards. In each iteration of an S-way loop, we apply a partial AG or RdS to a sub-shard, and compute a partial GeMM. The algorithm completes when all S sub-shards have been processed.

Figure 5 shows, for each of the three dataflows, the pseudocode of the MeshSlice algorithm running in every chip (i, j) of the mesh. Each algorithm executes in a loop with S iterations. In here, we give the high-level intuition of the algorithms; in subsequent sections, we will explain the operations in detail. In the OS algorithm (Figure 5, left), for each $s = 0 \ldots S-1$, the following occurs. First, each chip uses $slice_{col}$ to slice its local matrix shard A_{ij} along the column dimension to fetch its local s-th sub-shard, A_s. Similarly, it uses $slice_{row}$ to slice its local B_{ij} along the row dimension to fetch its local s-th sub-shard, B_s. Then, each chip collects A_s sub-shards from all the chips in the same row and B_s sub-shards from all the chips in the same column using AG_{col} and AG_{row} operations, respectively. Finally, each chip computes the partial GeMM with the collected sub-shards and accumulates the result into its local output shard C_{ij}. Crucially, the partial GeMM operation in one iteration is overlapped with the AG and slicing operations in another iteration via software pipelining of the loop.

We can apply a similar slicing method to LS and RS dataflows. For each iteration in the LS dataflow (Figure 5, center), B_{ij} and C_{ij} are sliced along their row and column dimensions, respectively. Then, the s-th sub-shards B_s in the chips in the same column are all-gathered to B'. Next, the partial multiplication result $C' = A_{ij}(B')^\top$ is computed. Finally, C' is reduce-scattered to the s-th sub-shards C_s in the chips in the same row. The RS dataflow (Figure 5, right) follows a similar flow.

Like Collective and SUMMA, MeshSlice can be applied to a mesh of any shape. Also, we can control the slice count S to adjust the granularity of communication. A small S (coarse granularity) results in a large non-overlapped prologue and epilogue during the software pipelining. A large S (fine granularity) reduces the size of the prologue and epilogue, but increases the total synchronization overhead by performing more communication operations. Given this trade-off, there are different optimal values of S for different 2D GeMM configurations and hardware architectures.

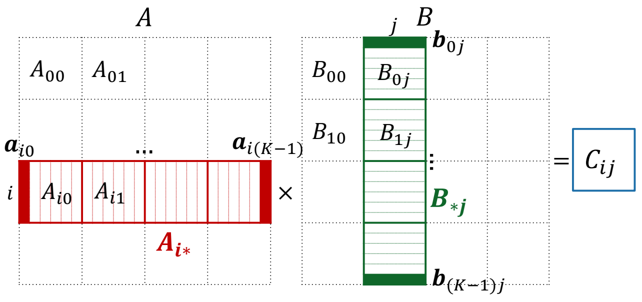

Figure 6: Generating an output shard in a 2D GeMM.

The major challenge of MeshSlice is designing a correct and efficient slicing mechanism. This is not a trivial problem: most arbitrary slicings result in an incorrect computation. In the following, we describe how we implement the correct slicing mechanism.

3.1.1 Mathematical Description of the MeshSlice Algorithm. Assume that we are computing a 2D GeMM of $A \in \mathbb{R}^{M\times K}$ and $B \in \mathbb{R}^{K\times N}$ in OS dataflow on a mesh of shape $P_r \times P_c$. In each chip (i, j), the computation result will be the shard $C_{ij} \in \mathbb{R}^{M/P_r\times N/P_c}$, which is the multiplication of $A_{i*} = AG_{col}(A_{ij}) \in \mathbb{R}^{M/P_r\times K}$ and $B_{*j} = AG_{row}(B_{ij}) \in \mathbb{R}^{K\times N/P_c}$. Figure 6 depicts A_{i*} and B_{*j} for a given i and j. Then, A_{i*} is broken down into K column vectors $\boldsymbol{a}_{i0}, \ldots, \boldsymbol{a}_{i(K-1)} \in \mathbb{R}^{M/P_r}$, and B_{*j} is broken down into K row vectors $\boldsymbol{b}_{0j}, \ldots, \boldsymbol{b}_{(K-1)j} \in \mathbb{R}^{1\times N/P_c}$. The figure highlights and labels four of these vectors: $\boldsymbol{a}_{i0}$, $\boldsymbol{a}_{i(K-1)}$, $\boldsymbol{b}_{0j}$, and $\boldsymbol{b}_{(K-1)j}$. Finally, $C_{ij} = A_{i*}B_{*j}$ is equivalent to the sum of K outer products of the column and the row vectors as follows.

$$C_{ij} = \boldsymbol{a}_{i0}\boldsymbol{b}_{0j} + \cdots + \boldsymbol{a}_{i(K-1)}\boldsymbol{b}_{(K-1)j}$$

Our algorithm slices this computation in a loop with S iterations, where an iteration computes every S-th outer products. For S=3, Figure 7 shows the vectors accessed by the first iteration of the loop (in purple) and those accessed by the second iteration (in yellow). Algorithm 1 shows the algorithm. For instance, the first iteration accumulates $\{\boldsymbol{a}_{i0}\boldsymbol{b}_{0j} + \boldsymbol{a}_{iS}\boldsymbol{b}_{Sj} + \boldsymbol{a}_{i2S}\boldsymbol{b}_{2Sj} + \ldots\}$ to the output shard C_{ij}. The column vectors accessed by this iteration $\{\boldsymbol{a}_{i0}, \boldsymbol{a}_{iS}, ...\}$ are gathered using AG from the shards $\{A_{i0}, A_{i1}, \ldots\}$ in the chips at the i-th row. Likewise, the row vectors $\{\boldsymbol{b}_{0j}, \boldsymbol{b}_{Sj}, ...\}$ are gathered from the shards $\{B_{0j}, B_{1j}, \ldots\}$ in the chips at the j-th column.

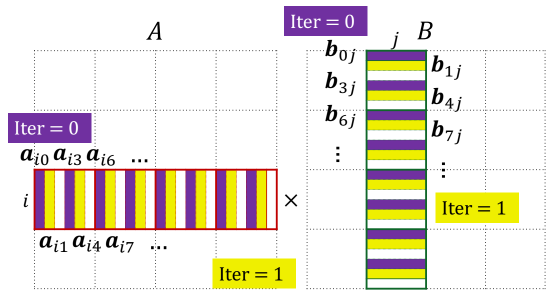

Figure 7: Slicing the shards with $S = 3$ in MeshSlice.

Algorithm 1 S-way sliced GeMM algorithm to compute C_{ij}

```
for $s = 0 \ldots S-1$ do
    $C_{ij}$ += $\boldsymbol{a}_{is}\boldsymbol{b}_{sj} + \boldsymbol{a}_{i(s+S)}\boldsymbol{b}_{(s+S)j} + \boldsymbol{a}_{i(s+2S)}\boldsymbol{b}_{(s+2S)j} + \ldots$
```

3.1.2 Detailed Implementation of the MeshSlice Algorithm. In this section, we describe the MeshSlice algorithm presented in Figure 5 in detail, and show that its partial GeMM computation is identical to the sliced GeMM computation in Algorithm 1. To begin with, note that A_{ij} contains K/P_c column vectors of A_{i*} and B_{ij} contains K/P_r row vectors of B_{*j} as follows:

$$A_{ij} = [\boldsymbol{a}_{i(j\times K/P_c)}, \boldsymbol{a}_{i(j\times K/P_c+1)}, \ldots, \boldsymbol{a}_{i((j+1)\times K/P_c-1)}]$$

$$B_{ij} = \begin{bmatrix} \boldsymbol{b}_{(i\times K/P_r)j} \\ \vdots \\ \boldsymbol{b}_{((i+1)\times K/P_r-1)j} \end{bmatrix}$$

At the s-th iteration of the MeshSlice OS algorithm (Figure 5, left), applying $slice_{col}$ to A_{ij} collects every S-th column vectors in A_{ij}, and applying $slice_{row}$ to B_{ij} collects every S-th row vectors in B_{ij}. We call A_s and B_s the s-th sub-shards of A_{ij} and B_{ij}, respectively.

$$A_s = slice_{col}(A_{ij}, S, s) = [\boldsymbol{a}_{i(j\times K/P_c+s)}, \boldsymbol{a}_{i(j\times K/P_c+s+S)}, \ldots]$$

$$B_s = slice_{row}(B_{ij}, S, s) = \begin{bmatrix} \boldsymbol{b}_{(i\times K/P_r+s)j} \\ \boldsymbol{b}_{(i\times K/P_r+s+S)j} \\ \vdots \end{bmatrix}$$

If we AllGather (AG_{col}) the A_s sub-shards from all the chips in the same row of the mesh, and AllGather (AG_{row}) the B_s sub-shards from all the chips in the same column, we obtain the following A' and B' matrices.

$$A' = AG_{col}(A_s) = [\boldsymbol{a}_{is}, \boldsymbol{a}_{i(s+S)}, \boldsymbol{a}_{i(s+2S)}, \ldots]$$

$$B' = AG_{row}(B_s) = \begin{bmatrix} \boldsymbol{b}_{sj} \\ \boldsymbol{b}_{(s+S)j} \\ \boldsymbol{b}_{(s+2S)j} \\ \vdots \end{bmatrix}$$

These $\boldsymbol{a}$ and $\boldsymbol{b}$ vectors are those shown in Figure 7. Then, computing C_{ij} += $A'B'$ is mathematically identical to computing the s-th iteration of Algorithm 1.

Our slicing operation may result in non-contiguous memory accesses. For example, the $slice_{col}$ operation accesses column vectors $\boldsymbol{a}_{is}, \boldsymbol{a}_{i(s+S)}, \ldots$, which are not contiguous in memory. This is inefficient in most memory subsystems. Therefore, we further optimize the slicing operations ($slice_{col}$ and $slice_{row}$) so that the $\boldsymbol{a} \in \mathbb{R}^{M/P_r}$ column vectors become matrices $\boldsymbol{a} \in \mathbb{R}^{M/P_r\times B}$, where B is an architecture-dependent block size (e.g., a cache line size). At the same time, the $\boldsymbol{b} \in \mathbb{R}^{1\times N/P_c}$ row vectors become matrices $\boldsymbol{b} \in \mathbb{R}^{B\times N/P_c}$. This design ensures contiguous memory accesses.

As an example, the blocked column slicing algorithm $slice_{col}$ is shown in Algorithm 2, where R and C are the dimensions of a local shard. The block size B is determined by the hardware architecture. For instance, since a TPU accesses its memory via 2D 128 × 8 chunks [10], we set $B = 8$ for TPUs. The user can then choose any slice count S from the divisors of C/B.

Algorithm 2 Blocked column slicing algorithm

```
function $slice_{col}(X : matrix < R, C >, S : int, s : int)$
    $B \leftarrow$ block size for efficient memory access
    // Splits $X$ by $B$ contiguous columns
    $X' \leftarrow X.reshape(< R, C/SB, S, B >)$
    // The $s$-th sub-shard has $< R, C/S >$ elements
    return $X'[:, :, s, :].reshape(< R, C/S >)$
```

3.2 The MeshSlice LLM Autotuner

The MeshSlice 2D GeMM algorithm has several parameters that determine its communication cost and efficiency. First, the shape of the mesh determines the traffic cost as discussed in Section 2.3.1. Next, there are numerous ways to partition the matrix into shards for the chips in the mesh (i.e., the sharding), and a different partitioning changes the 2D GeMM dataflow. Finally, we need to determine the slice count S of MeshSlice, which affects the synchronization overhead and the communication overlapping.

Finding an optimal configuration of these parameters is a hard problem, and usually relies on human inputs [35]. Manually finding the optimal parameters needs expert knowledge of the system architecture and expensive trial-and-error.

To address this problem, we design the *MeshSlice LLM autotuner*, which can find efficient parameter configurations for an LLM training. The inputs to the autotuner are the LLM architecture, the training hyperparameters (e.g., batch size and input sequence length), and the possible 2D mesh shapes of the cluster of chips. The autotuner runs in two phases. First, it determines efficient dataflows for the FC layers and the shardings of the LLM tensors. Next, it jointly optimizes the mesh shape of the cluster and the slice count S of each FC layer, using analytical cost models.

3.2.1 Phase 1: Dataflow and Sharding. The first optimization problem of 2D TP is choosing the right *sharding*. Given a mesh, a sharding of a tensor is a mapping from the mesh dimensions to the tensor dimensions that are to be partitioned. For example, given a 2D mesh and a 4D tensor, there are $4P2 = 12$ possible shardings, since we need to choose two tensor dimensions to split among the rows and columns of chips, respectively.

Automatically finding the optimal sharding is difficult for two reasons. First, there are many choices available. For example, consider a 2D GeMM of two 4D tensors. Since we have two input tensors and one output tensor, there are $(4P2)^3 = 1728$ possible sharding combinations. Next, estimating the performance for each sharding combination is hard. A change in one sharding may induce a different dataflow, require a possible re-sharding due to transposition, and result in a different compute efficiency. To make matters worse, we cannot just consider the shardings of the forward computations, but also the shardings of the backpropagation computations during training.

Choosing a dataflow. In this work, we approach this problem in the opposite direction: the autotuner chooses the dataflow first and then determines the sharding of each tensor. Take a 2D GeMM $Y = XW$, which multiplies the input X and the weight W to compute the output Y. There are three 2D GeMM dataflows for the forward pass computation, as shown in the *Forward* column of Table 1. The autotuner chooses the dataflow that makes the largest matrix of the three remain stationary. For instance, if the output Y is the largest matrix, the choice is Y-stn.

Table 1: Three dataflows for the 2D GeMM $Y = XW$. Each dataflow makes Y stationary (Y-stn), X stationary (X-stn), or W stationary (W-stn).

Dataflow	Forward	Backward Data	Backward Weight
Y-stn	$Y = OS(X, W)$	$X' = LS(Y', W)$	$W' = RS(X, Y')$
X-stn	$Y = LS(X, W^\top)$	$X' = OS(Y', W^\top)$	$W'^\top = RS(Y', X)$
W-stn	$Y = RS(X^\top, W)$	$X'^\top = LS(W, Y')$	$W' = OS(X^\top, Y')$

To train a DNN layer, there are two passes of computation: forward pass to compute the outputs, and backward pass to compute the gradients for backpropagation. A forward pass of $Y = XW$ generates two backward pass computations, namely *backward data* and *backward weight*. The backward data computation computes the input gradient $X' = Y'W^\top$ as a multiplication of the output gradient Y' and the transpose of W. The backward weight computation computes the weight gradient $W' = X^\top Y'$ by multiplying the transpose of X and Y'. The compute and communication demands of each of the backward data and the backward weight computations are almost identical to those of the forward pass.

Given the forward pass dataflow, the autotuner chooses the dataflows for the backward pass computations from the same row of Table 1. This ensures the following properties. First, the largest matrix remains stationary in all three (forward and two backward) computations. Also, each matrix and its gradient matrix flow in the same direction in all three computations. Finally, none of the matrices needs to be transposed to compute the backward pass.

As an example, assume we choose the X-stn dataflow for the forward pass ($Y = LS(X, W^\top)$). Hereby $W^\top$ is statically transposed during its initialization. In the forward computation, Y flows horizontally, and $W^\top$ flows vertically. The dataflow for backward data computation is $X' = OS(Y', W^\top)$. As with the forward pass, X' is stationary, Y' flows horizontally, and $W^\top$ flows vertically. The dataflow for backward weight computation is $W'^\top = RS(Y', X)$, which makes X be stationary, Y' flow horizontally, and $W'^\top$ flow vertically. Overall, *across all computations*, X and X' remain stationary as they are the largest matrices, Y and Y' flow horizontally, and $W^\top$ and $W'^\top$ flow vertically.

For each row of Table 1, we can transpose all the matrices and flip the dataflow directions of the two non-stationary matrices to obtain the corresponding *transposed* dataflow. For example, the transposed version of $Y = OS(X, W)$ is $Y^\top = OS(W^\top, X^\top)$. Therefore, for each layer, there are two dataflow choices (non-transposed and transposed) that make the largest matrix stationary. Finding the absolute optimal dataflow choices for an L-layer neural network is a search problem with 2^L possibilities. Hence, the MeshSlice LLM autotuner uses a simple heuristic: for each layer, choose the non-transposed dataflow as a default, unless the layer's input X needs to be transposed to keep the non-transposed dataflow. In most LLMs, this heuristic eliminates transpositions between the layers.

Sharding. Once the autotuner chooses the dataflows, the shardings of the three matrices are automatically determined. The matrix rows are sharded among the rows of chips, and the matrix columns are sharded among the columns of chips.

In practice, LLMs use 4-D tensors of shape (B, S, H, D), where B is the batch size, S is the sequence length, H is the number of attention heads, and D is the per-head hidden dimension. In an FC layer, the 4D tensor is reshaped into a 2D matrix of $(B \times S, H \times D)$ dimensions. MeshSlice follows the simple principle of partitioning the two outermost dimensions of a 2D matrix. Hence, the B dimension of the 4D tensor is sharded among the mesh rows, and the H dimension is sharded among the mesh columns.

Besides the FC layers, an LLM network has many other operations. The shardings of these other operations have minimal performance impact as they incur no communication traffic [16]. Therefore, once the shardings of the FC layer tensors are determined, we let other tensors follow the same shardings to avoid resharding traffic.

3.2.2 Phase 2: Mesh Shape and Slice Count. This phase configures the two remaining parameters of MeshSlice: the mesh shape and the slice count. To this end, we design analytical cost models that estimate the GeMM execution time for each configuration. Then, we use these cost models to co-optimize the two parameters.

We construct an analytical cost model of the communication from offline measurements of the synchronization latency, network bandwidth, and communication operation launch overheads in a small ML accelerator cluster. The cost of a collective communication operation is defined as follows:

$$cost_{op} = t_{launch} + (P - 1) \times (t_{sync} + sizeof(shard)/bw)$$

Hereby t_{launch} is the overhead of operation launch, P is the number of chips in the row or column, t_{sync} is the synchronization latency, $sizeof(shard)$ is the size of a shard to be transferred, and bw is the measured bandwidth of a link. This linear model fits well for the AG/RdS communications on a row or column ring. This is because in an AG or RdS on a ring (Figure 3, right), the shard transfers are synchronized and there is no network contention.

To estimate the compute times, our analytical compute cost model divides the total FLOP count of the local GeMM by the effective FLOPS throughput of the ML accelerator. The effective FLOPS is measured by benchmarking a few GeMM operations on a single accelerator chip. This computation model is accurate enough for LLM training because most GeMMs in LLM training are large enough to fully saturate the compute throughput of ML accelerators. For better accuracy, one can measure the compute execution time in a single accelerator chip instead of using the analytical model.

For each FC layer, the autotuner breaks down the compute plus communication execution time of the MeshSlice algorithm into three parts: prologue, steady-state, and epilogue. The prologue and epilogue are the operations in the first and last loop iterations, respectively, that cannot be overlapped by software pipelining. For instance, in our OS algorithm of Figure 5, the two all-gather operations in the first iteration form the prologue, and the partial GeMM computation in the last iteration forms the epilogue. We assume that a communication in one direction can execute in parallel with a communication in the other direction. Hence, the prologue time is time of the longest of the two AG operations, and the steady-state time per iteration is the time of the longest of three operations: the

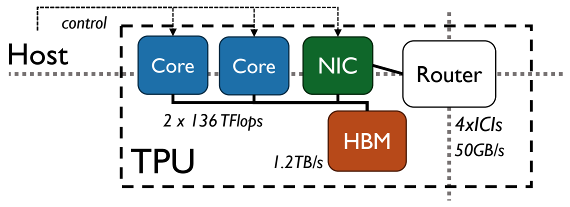

Figure 8: Architecture of the simulated TPU.

two AG operations and the partial GeMM. The total estimated execution time of the S iterations is: prologue + $(S - 1)\times$ steady-state per iteration + epilogue.

Using the cost models, the autotuner co-optimizes the mesh shape of the cluster and the slice count S_i of each FC layer i using an exhaustive search. For each possible mesh shape, the autotuner tunes S_i for each FC layer i by searching through all possible S values. Since the optimal S_i values of the different FC layers are independent of each other, the autotuner optimizes the S_i value one layer at a time. Finally, it picks the configuration with the shortest execution time. The search space for the mesh shape and the slice counts is small because there are only a few possible integer choices for them. Therefore, the autotuner finishes in a few seconds thanks to the small search space and the simple analytical cost models.

4 Experimental Setup

4.1 Simulation Setup

Our target ML accelerator to evaluate is Google's TPU [14] because TPUs can build a 2D torus cluster connected with ICI links. Hence, we have built an implementation of MeshSlice on Jax [9] that runs on TPU instances in Google Cloud. Hereby we use Jax's *shard_map* feature to partition the computations into the mesh, and the *dynamic_slice* operation to implement MeshSlice's blocked slicing algorithm. This implementation does not overlap computations with communications because Google only supports asynchronous (i.e., overlapped) communication for SendRecv operations and not for the AG and RdS operations needed by MeshSlice yet.

For this reason, for the most part, we evaluate our algorithm by simulating clusters of TPUs connected with 2D torus topologies of different sizes and shapes. Figure 8 shows the architecture of the simulated TPU, which models Google's TPUv4 [13]. A node in the cluster consists of a TPU and a host. The TPU has two cores and a network interface controller (NIC) that share an HBM memory. Each core has a 64MB scratchpad memory and four 128×128 systolic arrays to compute matrix multiplications. The NIC is connected to a router with four ICI links that connect the TPUs in a 2D torus network. The NIC processes direct communications between the TPUs. It can directly read from and write to the HBM memory. The TPU cores and the NIC are controlled by the host and can execute in parallel. The only performance interference between the cores and the NIC comes from any contention for the shared HBM memory.

We customize the SST simulator [23] to simulate the cluster and network architecture. We modify the SST's rdmaNic simulator to simulate the NIC. The modified rdmaNIC simulator implements one controller per each ICI link, so that the four ICI links can execute communication operations in parallel. The HBM memory is simulated using DRAMSim3 [18].

To model the TPU cores, we implement a custom accelerator in SST. We focus on accurately modeling their memory access behavior, to be able to capture the memory contention between the cores and the NIC. The simulated TPU takes a GeMM ($C = AB$) request from the host and breaks it down into tiled sub-matrix multiplications. An output tile C_t is computed in a loop, where each iteration prefetches the input tiles A_t and B_t from HBM to the scratchpad memory, and computes $C_t += A_tB_t$. We apply software pipelining in that the prefetches overlap with the multiplications. Once the loop finishes, C_t is written back to the HBM memory and the core transitions to the next tile.

We use the Google Cloud [11] to benchmark four real TPUv4 chips connected in a ring, to calibrate the parameters of our simulator (e.g., frequency, bandwidth, latency, and tile size). We profile the communication operations to configure the NIC and the router. The simulator is calibrated until the difference in benchmark execution time between the simulation and the real hardware is less than 10%.

4.2 2D GeMM Implementations

We evaluate the MeshSlice algorithm against four 2D distributed GeMM baselines: Cannon [4], SUMMA [30], Collective 2D GeMM (Collective) [25], and Wang's algorithm (Wang) [34]. All 2D GeMM algorithms are executed in two parts: (i) prologue plus epilogue, which execute the operations that are not overlapped, and (ii) a single iteration of the steady state, where communications are overlapped with computations.

As depicted in Figure 4, for MeshSlice and SUMMA, the prologue plus epilogue are the parts of the first and last iterations that are not overlapped, and the steady state is the execution of the remaining loop iterations. In Cannon, the prologue is the skewing communication that shifts the matrix shards prior to the computation. There is no epilogue, and the steady state is the execution of the iterations, which overlap computations with SendRecv communications. For Collective, the prologue plus epilogue are the two collective communication operations, and the steady state is the execution of the GeMM without communication. For Wang, the prologue or the epilogue is the execution of a communication operation that is not overlapped, and the steady state is the execution of the iterations, which overlap computations with SendRecv communications.

For each algorithm, the optimal mesh shape is different because the communication pattern is different. Hence, for fairness, we compare the performance with optimal mesh shapes for each algorithm.

Wang et al. [34] apply loop unrolling to their algorithm to have a smaller iteration count. This helps the computational efficiency by merging small GeMMs into larger GeMMs. Hence, we apply loop unrolling to SUMMA and Wang, as they have large iteration counts. We set the loop iteration counts of both algorithms to be the slice count of MeshSlice given by our LLM autotuner.

4.3 1D Distributed GeMM Baselines

While MeshSlice is a method for 2D TP, we also compare it against two 1D baselines. The first one is 1D TP following the Sequence Parallelism method [16], which is the most popular method to apply TP in LLMs. The second 1D baseline is Fully-Sharded Data Parallelism (FSDP) [37], which is a type of DP. FSDP is often considered an alternative to TP, as it also partitions the weight matrix into multiple shards and collects the shards right before the computation. It has a memory footprint similar to 1D TP.

The 1D baselines are simulated on a ring of TPU chips. Hence, each TPU is connected to only two ICI links. This halves the total bandwidth compared to a 2D mesh with the same number of chips. For both 1D TP and FSDP, we overlap communications with computations using Wang et al.'s method [34]—i.e., their AG or RdS communications are broken down into multiple SendRecv communications that are overlapped with partial GeMMs.

4.4 Target LLMs

We evaluate the training performance of two LLM models: OpenAI's GPT-3 [3] and NVIDIA's Megatron-NLG [27]. GPT-3 has 175B parameters and is the most popular LLM. Megatron-NLG has 530B parameters and requires a larger-scale distributed training.

LLMs follow the Transformer [31] DNN architecture, which consists of multiple stacks of identical neural network blocks. Each block consists of two sub-networks: multi-head attention and feed-forward. There are four FC layers in a block: two in the multi-head attention and two in the feed-forward network.

For the different 2D GeMM algorithms, only the FC layers have different implementations, while the other layers remain the same. Therefore, to compare the different GeMM algorithms, we only evaluate the FC layers. The other layers are benchmarked with TPUv4 chips in Google Cloud. They can be benchmarked with a single TPU because they do not incur communication cost: they are executed independently in each TPU chip [16]. The execution times of the FC layers and the other layers are combined to estimate the end-to-end performance of LLM training.

4.5 Building Analytical Cost Model

The communication cost model is a linear function with three parameters: bw, t_{sync}, and t_{launch} (Section 3.2.2). We benchmark the collective communication operations in 2-chip and 4-chip TPUv4 clusters with shard sizes ranging from 8KB to 512MB. t_{sync} is set by comparing the execution times with different numbers of chips. bw and t_{launch} are found with a linear regression on the execution times with different shard sizes.

The computation cost model only requires the effective compute throughput of the TPU, which can be measured by profiling a few GeMM operations on TPUv4.

The detailed implementation of the cost models in the MeshSlice LLM autotuner are presented in the source code repository[1], together with the TPU implementation of MeshSlice.

5 Evaluation

In this section, we evaluate the performance of LLM training using different distributed GeMM algorithms (Section 5.1), the autotuner and the cost models (Section 5.2), and MeshSlice running on a small real TPUv4 cluster (Section 5.3).

[1]https://github.com/hwnam831/meshslice

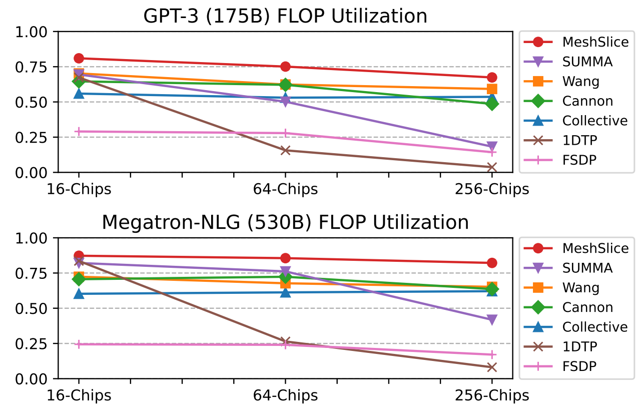

Figure 9: FLOP utilization of the FC layers with different distributed GeMM algorithms under weak scaling. The charts correspond to training GPT-3 (top) and Megatron (bottom).

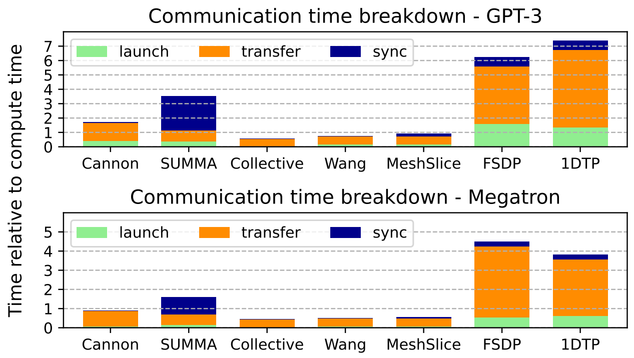

Figure 10: Breakdown of the communication time in the FC layers for the different algorithms relative to the algorithms' own computation time. The charts correspond to training GPT-3 (top) and Megatron (bottom) in 256-chip clusters.

5.1 Distributed GeMM Algorithm Performance

5.1.1 Weak Scaling Performance. We compare the MeshSlice algorithm to four 2D baselines (Cannon, SUMMA, Collective, and Wang) and two 1D baselines (1D TP and FSDP). We start by evaluating the performance under weak scaling, which is the most common scenario in distributed DNN training. This is because adding more chips to a cluster results in more available memory, which enables an increase in the input batch size for faster training. We set the batch size to half the number of chips in the cluster, and the input data sequence length to 2048. These configurations are selected to follow the setup of Megatron-NLG. [27]

We compare the performance of the algorithms in the Fully-Connected (FC) layers, combining all three training computations: forward, backward data, and backward weight. We report the FLOP utilization of each algorithm, which is computed as achieved GeMM compute throughput divided by the maximum compute throughput of the cluster (which is 272 TFLOPS per TPUv4). Because all the distributed GeMMs perform the same amount of compute, the FLOP utilization of an algorithm is proportional to its performance.

Figure 9 shows the FLOP utilization of the different algorithms for training GPT-3 (top) and Megatron-NLG (bottom). We see that MeshSlice is both the fastest method in all cases, and maintains good efficiency as the number of chips increases. For 256-way parallelism, MeshSlice is 13.8% and 26.0% faster than the state-of-the-art Wang algorithm in GPT-3 and Megatron, respectively. If we also include the performance of the non-FC layers of the LLM models, the *end-to-end speedups* of MeshSlice over Wang for 256-chips are 12.0% and 23.4% for GPT-3 and Megatron, respectively.

MeshSlice maintains high efficiency at 256-way parallelism. Going from 16-way to 256-way, MeshSlice only loses 16.8% and 5.8% of its efficiency for GPT-3 and Megatron-NLG, respectively.

Collective is always slower than MeshSlice because it can fall back to Collective by setting $S = 1$ whenever it is more efficient to do so. Wang lies in between MeshSlice and Collective, as it partially overlaps one of the two collective communications in Collective. The other algorithms (i.e., SUMMA, Cannon, 1DTP, and FSDP) are inefficient in large clusters. They are even slower than Collective, which cannot overlap communications with computations.

5.1.2 Communication Cost Breakdown. To understand the inefficiencies of the algorithms, Figure 10 breaks down their total communication time (overlapped plus non-overlapped). For each algorithm, the figure shows the communication time relative to its GeMM computation time—which is almost the same in all the algorithms. The bars are broken down into three parts: time spent launching a communication operation (launch), transfering the shards (transfer), and synchronizing the chips (sync). In an algorithm, it is theoretically possible to hide all the communication time if the total relative time is less than 1.

Cannon has a relatively high communication time because it incurs a large traffic cost. Its extra traffic comes from two sources: the requirement for a square mesh shape and for skewing the matrix shards. The shortcoming of needing a square mesh shape becomes more pronounced at large cluster sizes. This is because, as we increase the cluster size, the weight matrix size remains constant but the batch size increases—which induces larger input and output matrices. As the matrix sizes become more imbalanced, the square mesh becomes more inefficient relative to the optimal mesh shape.

SUMMA has even higher communication time, due to its large synchronization overhead. As discussed in Section 2.3.3, SUMMA's synchronization overhead grows quadratically with the number of rows or columns (whichever is larger). Its synchronization overhead becomes dominant in large meshes. Hence, SUMMA is efficient only in small clusters.

The high communication times in the 1D methods (1DTP and FSDP) show why we need 2D methods to scale DNNs efficiently. 1D methods are inefficient for two reasons. First, they can only utilize two ICI links to form a ring topology, rather than the four links of a 2D mesh. Second, 1D algorithms intrinsically have higher traffic than 2D algorithms, as discussed in Section 2.2. As a result, 1DTP and FSDP incur higher communication time than 2D GeMM algorithms, showing significantly lower performance in large accelerator clusters.

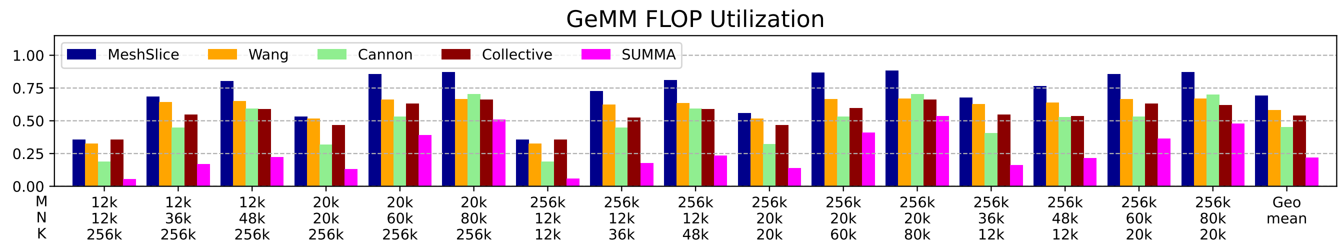

Figure 11: FLOP utilization of different 2D GeMM algorithms with different matrix shapes (M,N,K) in a 256-chip cluster.

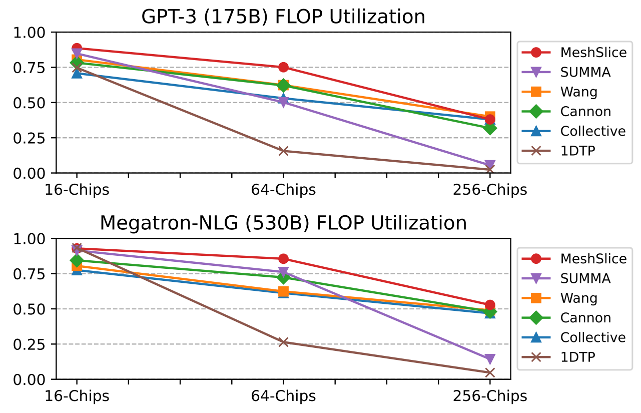

Figure 12: FLOP utilization of the FC layers with different distributed GeMM algorithms under strong scaling. The charts correspond to training GPT-3 (top) and Megatron (bottom).

The figure shows that Collective has the least communication time, as it executes the smallest number of large collective communications. However, this communication time cannot be overlapped with computation. Wang and MeshSlice have slightly higher communication times than Collective. Wang has extra launch overhead to call multiple SendRecv operations, while MeshSlice adds extra synchronization overhead because each AG/RdS operation invokes more synchronizations than a SendRecv operation. Nonetheless, the communication time in MeshSlice can be mostly hidden by overlapping it with computation.

5.1.3 Strong Scaling Performance. Figure 12 compares the FLOP utilization of the different algorithms under strong scaling. In the experiments, the batch size is fixed to 32, which is the configuration for the 64-chip cluster in weak scaling. While strong scaling is not a realistic scenario in distributed DNN training, the results provide some insights. Note that FSDP cannot support strong scaling because DP assumes the batch size increases with the chip count.

The 16-chip results show a compute-bound scenario. Because there is not much communication cost to overlap, all the algorithms exhibit a relatively high efficiency. However, such a compute-bound scenario is becoming less common, as the compute power of ML accelerators is growing faster than the bandwidth of ICIs.

In the 256-chip results, the communication cost is now dominant. Hence, MeshSlice's gain from communication overlapping diminishes, and MeshSlice shows a utilization similar to Collective and Wang. Still, MeshSlice has higher utilization than 1D TP and SUMMA, which incur more traffic and synchronization overhead, respectively. Overall, MeshSlice is a safe choice regardless of whether it is compute-bound or communication-bound.

Table 2: FLOP utilizations in FC layer training without and with MeshSlice dataflow optimization in a 256-chip cluster.

LLM	Not optimized	Optimized	Speedup
GPT-3	55.6%	67.4%	21.2%
Megatron	78.2%	82.2%	5.1%

5.1.4 Performance for Different Matrix Shapes. During the forward and backward passes of the FC layers in LLM training, there are eight distinct GeMM operations with different M,N,K matrix shapes. Because we run GPT-3 and Megatron-NLG, this means we have a total of 16 GeMM variants. Figure 11 compares the FLOP utilization of these 16 GeMMs using the different 2D GeMM algorithms in a 256-chip cluster. We see that MeshSlice is consistently faster than the other algorithms in all 16 GeMMs. On average, MeshSlice is 27.8% and 19.1% faster than Collective and Wang, respectively. The speedups are higher in larger GeMMs, which take longer times to execute during LLM training.

5.2 LLM Autotuner and Cost Model

In this section, we validate the efficacy of the MeshSlice LLM autotuner. The autotuner optimizes three MeshSlice parameters: the dataflow, the mesh shape, and the slice count S. The shardings are automatically determined by the dataflows. We evaluate each parameter optimization in turn.

The first parameter configured by the autotuner is the dataflows of the FC layers. Without a dataflow optimization, the default approach is to use Y-stn dataflows from Table 1 for all FC layers. This is because the Y-stn dataflow does not transpose any of the matrices in the FC layers. Table 2 compares the FLOP utilizations in FC layer training without and with the MeshSlice dataflow optimization. The dataflow optimization brings 21.2% and 5.1% speedup in GPT-3 and Megatron, respectively. In Megatron, MeshSlice is already fast without the dataflow optimization because most of the communication cost can be hidden by overlapping it with the computation. However, GPT-3 performs a smaller amount of computation because the model is smaller. As a result, some of the extra communication cost caused by the unoptimized dataflow cannot

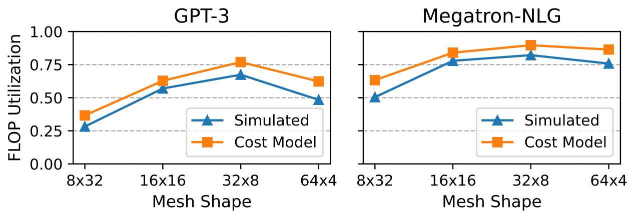

Figure 13: FLOP utilization estimated by the autotuner's cost models and utilization obtained with simulations, for different mesh shapes of a 256-chip cluster.

be overlapped, bringing a significant slowdown over the optimized dataflow. Overall, the performance gains delivered by the dataflow optimization of the MeshSlice LLM autotuner are significant.

The MeshSlice LLM autotuner uses analytical cost models of communication and computation to find efficient configurations for the mesh shape and the slice count. The accuracy of the cost models is key to the autotuner effectiveness. Here, we compare the cost model estimations to the simulation results, to evaluate the accuracy of the analytical cost models. Note that what matters is that the models correctly estimate if one configuration delivers higher performance than another, not that they correctly estimate the actual performance of each configuration.

Figure 13 compares the FLOP utilization estimated by the autotuner's communication and computation analytical cost models to the results obtained through simulations, for different mesh shapes of 256-chips. The autotuner's cost models estimate the execution times of the FC layers. The simulated and estimated execution times are converted to FLOP utilizations for comparison. We see that the mesh shape has a large impact on performance: the optimal mesh shape can bring a 2.4x speedup over a non-optimal one in GPT-3. From the figure, we see that, for both LLM models, the cost models accurately identify the optimal shape.

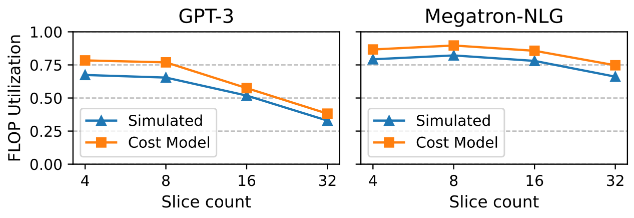

Figure 14: FLOP utilization estimated by the autotuner's cost models and utilization obtained with simulations, for different Slice Counts S in a 32×8 mesh.

Another hyperparameter that requires an accurate cost model is the Slice Count S. Figure 14 compares the FLOP utilization estimated by the autotuner's communication and computation analytical cost models to the results obtained through simulations, in a 32×8 mesh with different S values. We see that the optimal slice counts found by the autotuner's cost models are the same as the ones found by simulating the clusters. Overall, the MeshSlice LLM autotuner has simple but accurate analytical cost models that can find high-performance configurations of MeshSlice.

Table 3: FC layer FLOP utilization of 2D GeMM algorithms in a real 4x4 TPUv4 cluster. MeshSlice-Overlap shows the *estimations* if AG/RdS were overlapped with computation.

LLM	Collective	Wang	Mesh-Slice	MeshSlice Overlap (Estim.)
GPT-3	47.4%	47.7%	45.5%	65.7%
Megatron	49.4%	46.4%	47.1%	65.6%

5.3 MeshSlice Performance on Real Hardware

In this section, we run MeshSlice on a real 4x4 TPUv4 cluster. We note that current TPUv4 clusters *do not allow* the overlap of AG/RdS operations with computations—although they support overlapping asynchronous SendRecv operations as used in Wang et al. [34]. As a result, MeshSlice is slower than Collective and about as fast as Wang. Nevertheless, we conduct these experiments to 1) measure MeshSlice's intrinsic overheads and 2) validate the accuracy of our communication cost model.

5.3.1 Performance Comparison on a 4×4 TPU Mesh. The first three columns of Table 3 show the FLOP utilization of the FC layers implemented with Collective, Wang, and MeshSlice on the 4x4 TPUv4 cluster. Note that all the FLOP utilizations are lower than those obtained with simulations in Figure 9. The reason is that Google Cloud's 4x4 TPU clusters only utilize the *uni-directional bandwidth* of the bi-directional inter-node ICI links.

In this environment where MeshSlice cannot benefit from overlapping AG/RdS operations with computations, we see that MeshSlice adds ≈4.5% execution time overhead over Collective. Out of this 4.5% extra overhead, it can be shown from the traces that only 1.3% comes from the MeshSlice slicing operations; the majority of the remaining overhead comes from the less efficient fine-grain partial GeMMs and fine-grain partial AG/RdS operations. Therefore, MeshSlice's slicing mechanism is efficient and only adds very small performance overheads.

Wang is only marginally faster than Collective in GPT-3 and is slower than Collective in Megatron. Wang's speedup is lower than expected. This is because Google's current Jax compiler optimizations create dependencies that prevent most of Wang's communication operations from being overlapped with the computations.

The last column of Table 3 (MeshSlice Overlap) shows the estimated FLOP utilization in MeshSlice if the AG/RdS operations were overlapped with GeMM computations. We see that, if AG/RdS operations were overlapped, we can expect 38.6% and 32.8% speedups of MeshSlice over Collective for GPT-3 and Megatron, respectively.

5.3.2 Validating the Accuracy of the Communication Cost Model. Figure 15 validates the accuracy of our communication cost model using hardware measurements. The figure compares the estimated and measured communication times (overlapped plus non-overlapped) for different FC layers. We report the total communication time of one forward and backward pass per FC layer. We see that our communication cost model accurately estimates the communication

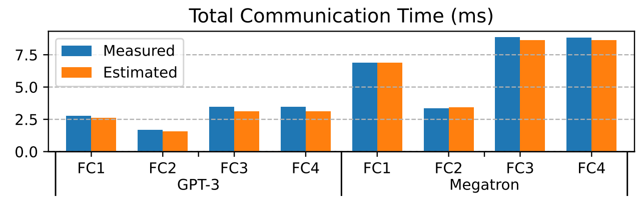

Figure 15: Estimated and measured total communication times of 8 different FC layers in MeshSlice.

times of the FC layers, with only 5.1% average error. The high accuracy of the communication cost model is expected because AG/RdS operations in a row or column suffer no network contention.

6 Discussion and Future Work

This work focuses on applying MeshSlice to a physical mesh of ML accelerators, which is currently available only with TPUs. To apply MeshSlice to other types of networks, we can construct a logical mesh on top of the existing network. This would allow us to apply MeshSlice to GPU clusters. However, with a logical mesh, MeshSlice becomes less efficient because AG/RdS operations will incur network contention that does not exist in physical meshes. In this case, the MeshSlice LLM autotuner needs to be modified to estimate the performance impact of the network contention.

In this paper, we have applied MeshSlice to training. MeshSlice can also be applied to inference. Indeed, the Wang algorithm has been used in an LLM inference cluster [21]. For inference, MeshSlice and its LLM autotuner may need to be modified, since inference computations are more likely to be memory bound.

While this paper focuses on FC layers as used in LLMs, MeshSlice can also be applied to other types of DNN layers. One example is a convolution layer, which can be implemented as a GeMM operation [6]. MeshSlice could also be used for GNNs [15, 32]: there has been work to perform 2D distributed sparse GeMMs for GNNs [2], and MeshSlice could be applied to optimize them.

MeshSlice can also be combined with mixture-of-experts (MoE) methods [26]. Applying MoE in LLMs builds a significantly larger model and adds another dimension of parallelism called expert parallelism (EP) [17]. MeshSlice 2D TP could be combined with EP.

All these ideas are possible ways to extend this work.

7 Related Work

There are previous works that try to hide the communication cost of TP, but most of them target 1D TP. For example, Pati et al. [20] decompose a 1D TP GeMM into fine-grain computations and communications to overlap them. Centauri [5] applies communication partitioning and operation scheduling to a 3D training cluster that uses FSDP+PP+1DTP to efficiently overlap communications.

PrimePar [33] is the prior work most relevant to MeshSlice. It uses Cannon's algorithm for 2D TP and adds an optimization algorithm that finds optimal sharding strategies on a given GPU topology. MeshSlice is a 2D GeMM algorithm that is more efficient and scalable than Cannon's algorithm. Moreover, MeshSlice's LLM autotuner differs from PrimePar's optimization algorithm in four ways. First, MeshSlice focuses only on optimizing 2D TP, while PrimPar's optimizer co-optimizes the partitioning using DP, PP, and TP together. Next, the MeshSlice LLM autotuner optimizes the cluster topology, while PrimePar works for a fixed cluster topology. Further, MeshSlice can choose from different 2D dataflows, while PrimePar only uses Cannon's OS algorithm. Finally, MeshSlice allows configuring the communication granularity.

To further reduce the communication cost of 2D GeMM, algorithms have been proposed to compute a GeMM in a 3D cluster. The popular methods are 3D GeMM [1] and 2.5D GeMM [28]. 2.5D GeMM is the most popular method because 3D GeMM only works for a cubic 3D torus (i.e., a $P \times P \times P$ shape), while the 2.5D GeMM algorithm can work for any 3D torus with a $P \times P \times c$ shape. 2.5D GeMM makes c copies of the input matrices along the last dimension to reduce the communication cost. As an alternative, we can compute a GeMM in a 3D cluster by combining MeshSlice with DP, where DP also copies the weight matrices along the third dimension.

Because the 2.5D GeMM algorithm is based on Cannon's algorithm, it suffers from the same limitations: it incurs high traffic due to skewing and the fact that it can only support a square shape for the base mesh (i.e., $P \times P$). As an example, assume that we build a 3D cluster of 1024 accelerators and use either 2.5D GeMM or MeshSlice plus DP to compute an FC layer of GPT-3 whose (M, N, K) is $1(1024K, 12K, 48K)$. In 2.5D GeMM, the only possible 3D torus shape is $16 \times 16 \times 4$, where the per-chip communication traffic becomes 1.6GB. On the other hand, with MeshSlice plus DP, we can choose the better shape of $32 \times 8 \times 4$, and only incur 336MB of per-chip communication traffic. Therefore, MeshSlice+DP has much less communication traffic than the 2.5D GeMM algorithm.

8 Conclusion

This paper proposed the *MeshSlice* algorithm for efficient 2D tensor parallelism in distributed DNN training. MeshSlice solves the inefficiencies of previous 2D GeMM algorithms: it supports multiple mesh shapes, uses efficient AG/RdS primitives, and efficiently overlaps communication with computation in both dimensions. We also proposed the MeshSlice LLM autotuner, which picks an efficient 2D GeMM dataflow and uses a cost model to co-optimize the accelerator mesh shape and the communication granularity. In our evaluation, we showed that, in a simulated cluster of 256 TPUv4s, MeshSlice trains the GPT-3 and Megatron-NLG models 12.0% and 23.4% faster end-to-end, respectively, than the state of the art.

Acknowledgments

This work was supported in part by NSF under grants CNS 1956007, CCF 2107470, and CCF 2316233; by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; and by Google through the TPU Research Cloud (TRC).

References

[1] Ramesh C Agarwal, Susanne M Balle, Fred G Gustavson, Mahesh Joshi, and Prasad Palkar. 1995. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development* 39, 5 (1995), 575–582.

[2] Vivek Bharadwaj, Aydın Buluç, and James Demmel. 2022. Distributed-Memory Sparse Kernels for Machine Learning. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 47–58. doi:10.1109/IPDPS53621.2022.00014

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[4] Lynn Elliot Cannon. 1969. *A cellular computer to implement the Kalman filter algorithm.* PhD dissertation, Montana State University.

[5] Chang Chen, Xiuhong Li, Qianchao Zhu, Jiangfei Duan, Peng Sun, Xingcheng Zhang, and Chao Yang. 2024. Centauri: Enabling Efficient Scheduling for Communication-Computation Overlap in Large Model Training via Communication Partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 178–191.

[6] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. CuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[7] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project Adam: Building an efficient and scalable deep learning training system. In *11th USENIX symposium on operating systems design and implementation (OSDI 14)*. 571–582.

[8] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[9] Roy Frostig, Matthew James Johnson, and Chris Leary. 2018. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning* 4, 9 (2018).

[10] Google. 2024. *Cloud TPU performance guide.* https://cloud.google.com/tpu/docs/performance-guide

[11] Google. 2025. *Cloud Tensor Processing Units (TPUs).* https://cloud.google.com/tpu

[12] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems* 32 (2019).

[13] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, et al. 2023. 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*. 1–14.

[14] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*. 1–12.

[15] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[16] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2023. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems* 5 (2023), 341–353.

[17] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2020. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668* (2020).

[18] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. 2020. DRAMsim3: A cycle-accurate, thermal-capable DRAM simulator. *IEEE Computer Architecture Letters* 19, 2 (2020), 106–109.

[19] NVIDIA. 2023. *DGX H100: AI for Enterprise.* https://www.nvidia.com/en-gb/data-center/dgx-h100/

[20] Suchita Pati, Shaizeen Aga, Mahzabeen Islam, Nuwan Jayasena, and Matthew D Sinclair. 2024. T3: Transparent Tracking & Triggering for Fine-grained Overlap of Compute & Collectives. *arXiv preprint arXiv:2401.16677* (2024).

[21] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2022. Efficiently scaling transformer inference. *arXiv preprint arXiv:2211.05102* (2022).

[22] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3505–3506.

[23] Arun F Rodrigues, K Scott Hemmert, Brian W Barrett, Chad Kersey, Ron Oldfield, Marlo Weston, Rolf Risen, Jeanine Cook, Paul Rosenfeld, Elliot Cooper-Balis, et al. 2011. The structural simulation toolkit. *ACM SIGMETRICS Performance Evaluation Review* 38, 4 (2011), 37–42.

[24] Ananda Samajdar, Jan Moritz Joseph, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. 2020. A Systematic Methodology for Characterizing Scalability of DNN Accelerators using SCALE-Sim. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 58–68.

[25] Martin D Schatz, Robert A Van de Geijn, and Jack Poulson. 2016. Parallel matrix multiplication: A systematic journey. *SIAM Journal on Scientific Computing* 38, 6 (2016), C748–C781.

[26] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538* (2017).

[27] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, et al. 2022. Using DeepSpeed and Megatron to train Megatron-Turing NLG 530B, a large-scale generative language model. *arXiv preprint arXiv:2201.11990* (2022).

[28] Edgar Solomonik and James Demmel. 2011. Communication-optimal parallel 2.5 D matrix multiplication and LU factorization algorithms. In *European Conference on Parallel Processing*. Springer, 90–109.

[29] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[30] Robert A Van De Geijn and Jerrell Watts. 1997. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience* 9, 4 (1997), 255–274.

[31] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[32] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rJXMpikCZ

[33] Haoran Wang, Lei Wang, Haobo Xu, Ying Wang, Yuming Li, and Yinhe Han. 2024. PrimePar: Efficient Spatial-temporal Tensor Partitioning for Large Transformer Model Training. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 801–817.

[34] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, et al. 2022. Overlap communication with dependent computation via decomposition in large deep learning models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 93–106.

[35] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, et al. 2021. GSPMD: general and scalable parallelization for ML computation graphs. *arXiv preprint arXiv:2105.04663* (2021).

[36] Ahmed Sherif Zekri and Stanislav G Sedukhin. 2006. The general matrix multiply-add operation on 2D torus. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 8–pp.

[37] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. 2023. Pytorch FSDP: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277* (2023).