Overlap Communication with Dependent Computation via Decomposition in Large Deep Learning Models

Shibo Wang
shibow@google.com
Google
USA

Jinliang Wei
jlwei@google.com
Google
USA

Amit Sabne
asabne@google.com
Google
USA

Andy Davis
andydavis@google.com
Google
USA

Berkin Ilbeyi
berkin@google.com
Google
USA

Blake Hechtman
blakehechtman@google.com
Google
USA

Dehao Chen
dehao@google.com
Waymo
USA

Karthik Srinivasa Murthy
ksmurthy@google.com
Google
USA

Marcello Maggioni
maggioni@google.com
Google
USA

Qiao Zhang
zhangqiaorjc@google.com
Google
USA

Sameer Kumar
sameerkm@google.com
Google
USA

Tongfei Guo
tongfei@google.com
Google
USA

Yuanzhong Xu
yuanzx@google.com
Google
USA

Zongwei Zhou
zongweiz@google.com
Google
USA

ABSTRACT

Large deep learning models have shown great potential with state-of-the-art results in many tasks. However, running these large models is quite challenging on an accelerator (GPU or TPU) because the on-device memory is too limited for the size of these models. Intra-layer model parallelism is an approach to address the issues by partitioning individual layers or operators across multiple devices in a distributed accelerator cluster. But, the data communications generated by intra-layer model parallelism can contribute to a significant proportion of the overall execution time and severely hurt the computational efficiency.

As intra-layer model parallelism is critical to enable large deep learning models, this paper proposes a novel technique to effectively reduce its data communication overheads by overlapping communication with computation. With the proposed technique, an identified original communication collective is decomposed along with the dependent computation operation into a sequence of finer-grained operations. By creating more overlapping opportunities and executing the newly created, finer-grained communication and computation operations in parallel, it effectively hides the data transfer latency and achieves a better system utilization. Evaluated on TPU v4 Pods using different types of large models that have 10 billion to 1 trillion parameters, the proposed technique improves system throughput by 1.14 - 1.38x. The achieved highest peak FLOPS utilization is 72% on 1024 TPU chips with a large language model that has 500 billion parameters.

This work is licensed under a Creative Commons Attribution 4.0 International License.

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9915-9/23/03.
https://doi.org/10.1145/3567955.3567959

CCS CONCEPTS

• **Computer systems organization → Neural networks**; • **Software and its engineering** → *Compilers*; • **Computing methodologies** → *Parallel computing methodologies.*

KEYWORDS

Large scale machine learning, Compiler optimization, Collective communication hiding

ACM Reference Format:
Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, Sameer Kumar, Tongfei Guo, Yuanzhong Xu, and Zongwei Zhou. 2023. 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 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3567955.3567959

1 INTRODUCTION

Recent work in deep learning has demonstrated that larger models are significantly more sample efficient [27]. Scaling up the model size and also the amount of compute used for training has seen dramatic benefits across multiple application fields, such as natural language processing [12, 30, 46], recommendation [32], auto speech recognition [55], and computer vision [54]. As a result, the number of parameters in many deep learning models becomes larger and larger. For example, the size of the state-of-the-art natural language processing model has grown at an exponential rate in recent years [35]: the Megatron-Turing natural language generation model [37] with a dense architecture reaches 530 billion parameters; the recent GLaM [20] with a sparse mixture-of-experts architecture has 1.2 trillion parameters.

These large models are frequently well beyond the memory capacity of a single accelerator. Model parallelism then becomes necessary to fit these models in accelerator memories. Intra-layer (tensor) model parallelism [45, 46, 52] is one of the techniques to enable large models. It partitions individual layers in a model across multiple devices. As a result, data communication across devices is required with intra-layer model parallelism when the partitioning strategy changes between consecutive layers (e.g., each device needs to gather all of the data shards[1] that are distributed among device partitions if the partitioning strategy of the corresponding tensor changes from partitioned to replicated). This data communication can consume a significant proportion of the runtime, and become worse on clusters with a larger number of GPU or TPU nodes. Figure 1 shows the training time breakdown per minibatch of several large models on TPU v4 Pods [1]. Although these models have different types and amounts of data communication primitives depending on the model architecture and how partitioning is performed, they all spend a substantial percentage of the training time on data communication.

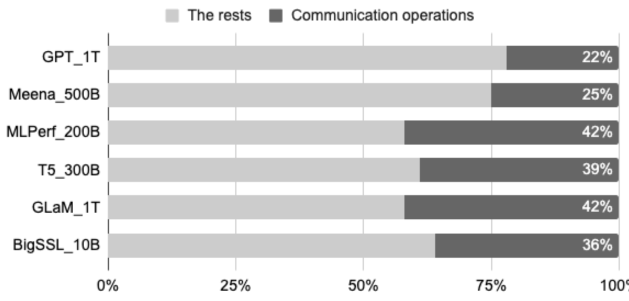

Figure 1: Training step time breakdown of large models. The models run on 128 - 2048 TPU chips depending on the model size. Details on the large models can be found in Section 6.

Leveraging the fact that computational units are mostly idle during the data communication, this paper proposes overlapping communication with computation to effectively hide the increasing communication overheads of large deep learning workloads with intra-layer model parallelism[2]. The sequential behaviour of most deep learning models make it hard to overlap communication and computation that have data dependence. To create more overlapping opportunities, the proposed technique decomposes the original computation and communication operations into finer-grained ones and directly work on each data shard individually instead of waiting until all of the data are ready (the data can be either operands or computation results). A communication operation that transfers data among multiple devices (e.g., *AllGather* and *ReduceScatter*) is decomposed into a sequence of finer-grained, point-to-point collectives (i.e., *CollectivePermutes*) based on the number of partitions specified by the model parallelism partitioning strategy. The consumer or producer computation operation (i.e., *Einstein Summation*) is also decomposed into a sequence of finer-grained ones (each of which performs only part of the computation) with a simple accumulation or concatenation operation to combine the partial results. The two sequences have the same length and together is semantically equivalent to the original collective-computation operation pair. As the sequences are generated in a way that there is no data dependence between the finer-grained communication and computation operations with the same index, more overlapping opportunities are created. To enable the overlapping, a new instruction scheduling scheme is proposed to asynchronously execute the decomposed collectives in parallel with the corresponding finer-grained computation operations in an iterative way. The collectives transfer either operands required by the partial computation in the next iteration or the accumulated partial results generated in the current iteration based on the model parallelism partitioning strategy. By performing decomposition and overlapping on the identified operations, the proposed technique dramatically reduces the communication cost and effectively improves the system utilization.

This paper makes the following contributions:

- We identify a new opportunity to overlap communication and computation via decomposition without violating data dependence. The target communication and computation patterns can be commonly found in distributed deep learning programs using intra-layer model parallelism.
- We propose a new scheme to maximize the overlapping via semantically equivalent graph transformation and decoupled asynchronous instruction scheduling. We also introduce a cost model to evaluate the trade-offs and automatically enable the overlapping feature based on the net benefits.
- We have implemented this technique in a ML compiler (XLA [3]) and it has been deployed in production. The optimization is performed automatically during compilation with no specific hardware nor model-level changes required. The general idea is hardware-agnostic and can be applied to other compilers or ML frameworks.
- We evaluated the proposed technique using various types of large production deep learning models on TPU v4 Pods with as many as 2048 chips. The evaluation results show

[1]Shard and partition are used interchangeably in this paper.

[2]The proposed technique can also be used by workloads with a combination of data, pipeline and intra-layer model parallelism if needed. A more detailed discussion can be found in Section 7.3.

1.2x training time speedup on average with 2-3x communication cost reduction. The system efficiency is significantly improved with as high as 72% peak system FLOPS utilization.

2 BACKGROUNDS ON INTRA-LAYER MODEL PARALLELISM

With intra-layer model parallelism, each layer of a model is partitioned over multiple devices. There are many ways to partition a model (along different dimensions and different number of dimensions in a layer), each of which may have different computational efficiency, memory consumption, and communication cost, especially when overlapped communication is taken into consideration. This section discusses the communication collectives and the common partitioning strategies with best performance, which are used throughout the paper.

2.1 Communication Primitives

MPI-style collective operations [21] are used to represent communication that occurs in intra-layer model parallelism with single program, multiple data (SPMD) paradigm. Some common collectives are:

- *CollectivePermute* has point-to-point communication patterns between each specified source-destination pair.
- *AllGather* has a many-to-many communication pattern that each logical device partition receives data from all of the partitions and concatenate them based on the data Partition IDs. It is mostly applied to replicate data along partitioned dimensions.
- *ReduceScatter* has a reverse communication pattern of *AllGather*. Each logical device partition performs element-wise reduction over the received data partitions with the same Partition ID.
- *AllReduce* can be considered as a *ReduceScatter* followed by an *AllGather*. After the operation, each logical device partition has the same tensor value from element-wise reduction over all of the data inputs, each of which comes from a different logical device partition.

2.2 Typical Partitioning Strategies

Many large machine learning models can fit in the memory with high system throughput when partitioning is performed along only one dimension of a tensor. The first common partitioning strategy we used is similar to the one used by Megatron [46]. However, instead of keeping layer inputs and outputs replicated and performing *AllReduce* along the reduction dimension after partitioned computation, we keep a shard of activations and shapes on each device and construct the weights on-demand by employing collective operations during execution.

Figure 2 demonstrates the data communication patterns in this typical partitioning strategy using a two-layer multi-layer perceptron (MLP) as an example. In this and the following examples in the paper, *Einsum*[3] is used to express the matrix multiplication and the general dot product computation in deep learning models due

[3]Einstein summation (Einsum) is a notation convention to simplify summation over a set of indexed terms in a formula [51].

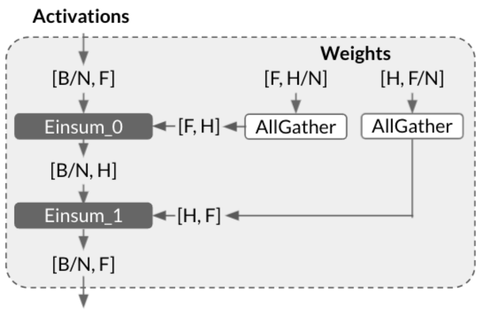

Figure 2: Forward-propagation of a two-layer MLP with N-way partitioning along one dimension. In the figure, each tensor is represented by a list of two elements, with *B*, *F*, and *H* respectively representing the size of the batch, the feature, and the hidden dimension. Notice that the activations and weights have already been partitioned, demonstrated by "/*N*" along a certain dimension.

this has more comms than megatron style TP

to its brevity. It can be seen from the figure that each device keeps a shard of an activation ([B/N, F] and [B/N, H]) during the entire process. An *AllGather* is performed before each *Einsum* computation to reconstruct the original, multi-dimensional weights ([F, H] and [H, F]). During backward-propagation (not shown in the figure), the *AllGathers* will become *ReduceScatters* to generate the partitioned gradients for weight update.

When a model scales up to a very large size, a larger number of partitions is required due to the device memory capacity limitations. In this case, devices are considered as forming a logical mesh or torus instead of a ring. The tensors are partitioned along two dimensions instead of a single one to avoid the low computation efficiency on GPUs or TPUs caused by small dimension sizes after partitioning. A different partitioning strategy is used to achieve both low peak memory usage and high effective computational efficiency. Figure 3 shows the new data communication patterns of the example two-layer MLP. Similar to the example shown in Figure 2, the batch dimension of the activations is kept partitioned. Input activations and weights are *AllGathere*d along different dimensions before each *Einsum*. In addition, output activations that are long-lived need to be kept fully partitioned to reduce memory usage — a subgroup *ReduceScatter* along x dimension is performed on the partially partitioned result of the second *Einsum*.

3 RELATED WORK

Overlapping communication with computation is a widely-accepted strategy to achieve high performance on distributed parallel systems. Prior works [15, 16, 23, 25, 31, 40] applies various of loop analysis and transformation techniques (such as loop fission, interchange, tiling, pipelining, strip-mining, etc.) to extract loops that contain only independent communication and computation for overlapping. The technique proposed in this paper targets at large deep learning models with intra-layer model parallelism and create new overlapping opportunities for dependent communication and computation operations by decomposing coarse-grained operations into finer-grained ones.

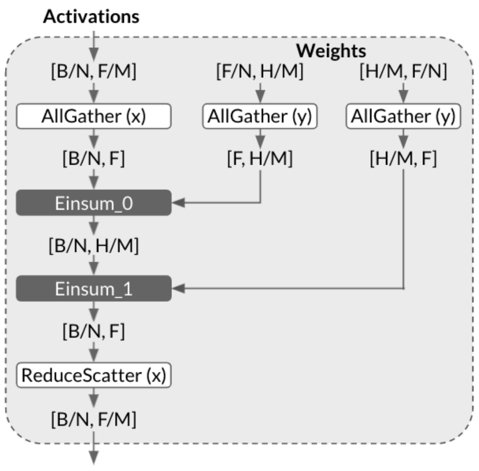

Figure 3: Forward-propagation of a two-layer MLP with N*M-way partitioning along two dimensions. In this example, the device mesh or torus has a shape of [M, N]. A tensor dimension divided by M or N means the tensor is respectively partitioned along x or y dimension. The (x) or (y) annotation at the end of each communication collective represents the corresponding data transfer dimension.

Generalized multipartitioning [17] demonstrated an algorithm that can find an optimal mapping of d-dimensional ($d > 2$) arrays onto an arbitrary number of processors that minimized the total communication cost in line sweep computations. This paper, however, aims at hiding the cost of communications by overlapping it with computations given a model partitioning that allows the ML model to fit in the accelerator memories. Using intra-layer model parallelism, the model partitioning used in this paper has already possessed the balance property, with different data communication patterns as compared to the line sweeps of multi-dimensional arrays. Finding the optimal partitioning based on an theoretical objective function considering different communication patterns is beyond the scope of this paper.

Another related area is distributed dense matrix multiplication. Cannon's algorithm [13] is a widely used 2D algorithm since its first introduction in 1969. By skewing the initial matrix block on each device along rows and columns of a square device mesh, the matrix blocks are aligned, and only a near-neighbor data transfer (circular shift) is required for correct block multiplication at each iteration. SUMMA [48] is another widely adopted one that logically sums up the vector outer products. As compared to Cannon's algorithm, it has higher communication costs due to row and column broadcasts. However, it's more flexible — neither the device mesh nor the matrix operands need to form a perfect square. 3D [9, 18] and more recent 2.5D [47, 50] algorithms were further proposed for different applications by trading memory for communication. Replications of matrix operands are required to avoid excessive data communication in 2D cases.

Georganas et al. proposed to overlap the forwarding shift of matrix blocks needed in the next iteration with the BLAS matrix multiplication in the current iteration [22]. This work leverages an idea similar to that used in this paper. However, we need to first identify the operations of interest in the large models that use model parallelism and to decompose the corresponding operations into finer-grained ones based on the number of partitions. Moreover, our technique provides a more general solution: 1) it supports overlapping communication that comes from not only gathering distributed operands, but also scattering the computed results; 2) it allows partitioning along different types of dimensions. The matrix multiplication with Cannon's algorithm can be considered as a special case in which the operands are partitioned along contracting dimensions and *AllGather*ed to compute the correct results. In addition, rather than using the complex 2D and above algorithms, the proposed scheme overlaps the data transfer and partial computation along a single dimension. This fits perfectly for partitioning strategies that have one partitioned dimension, and also improve the computation efficiency by increasing the sizes of the partitioned operands for partitioning strategies that have two or more partitioned dimensions.

ACE [43] has been recently proposed to accelerate communication collectives in deep learning training platforms. An accelerator is added alongside the network to handle the communication protocol. By offloading data communication to the accelerator, ACE can effectively improve the network utilization. As a comparison, the overlapping technique proposed in this paper is a pure software solution without hardware overheads. Rather than targeting small models that use only data parallelism, it identifies and creates more overlapping opportunities for large models with intra-layer model parallelism. Moreover, the proposed technique can be combined with ACE to further reduce the communication overheads that are not targeted in this paper.

4 OVERVIEW

Since communication collectives required by intra-layer model parallelism are quite expensive and block the dependent computations, this paper proposes to better utilize the idle computational units by overlapping communication with the dependent computation. For example, an *AllGather* can be overlapped with the consumer *Einsum*; a *ReduceScatter* can be overlapped with the producer *Einsum*. The key idea is to iteratively perform partial computation and asynchronous, non-blocking data communication over each data shard separately instead of doing them as a whole. To enable this, a target multi-step, blocking collective is broken into a sequence of single-step, non-blocking collectives. The computation operation that the collective wants to overlap with is also decomposed into a sequence of finer-grained computation operations with a simple accumulation or concatenation to combine the partial results. Both sequences are semantically equivalent to their original operation, and can be formed as a loop with the same number of iterations. Rather than executing the two sequences one after another, the proposed technique interleaves the communication and computation iterations. As the sequences are generated in a way that there is no data dependence between the communication and computation operations in the same iteration, each asynchronous, non-blocking collective is scheduled to execute in parallel with the corresponding decomposed computation.

Figure 4 uses an example to demonstrate how the proposed technique works for *AllGather* cases. In this example, each device partition has one shard of the operand *A* — *A0* on *Partition 0*, and *A1* on *Partition 1*, respectively — assuming *A* is partitioned at the beginning. In a current system, both devices start executing the computation after the *AllGather*, when the entire *A* (i.e., [*A0*, *A1*]) is available. The proposed overlapping scheme, however, does not wait until the entire data to be ready before the computation. Each device starts from asynchronously sending the data shard that originally resides in it to the other device [4], while computing the partial result using the available data shard at the same time. By the time that the first partial result is generated, the data shard required by the second half of the computation (*A1* on *Partition 0*; *A0* on *Partition 1*) has been received. Therefore the second partial computation can be continued immediately. An extra *DynamicUpdateSlice*[5] operation is needed for each partial result to update the corresponding slice in the final result, which is initialized with all zeros. The correct final result can be achieved by performing such steps several times. The exact number of steps depends on the number of partitions on *A*, and in this case is two. By decomposing and overlapping the data communication required to get *A* with the corresponding computation, the proposed scheme is able to achieve better hardware utilization and performance, as shown in the figure.

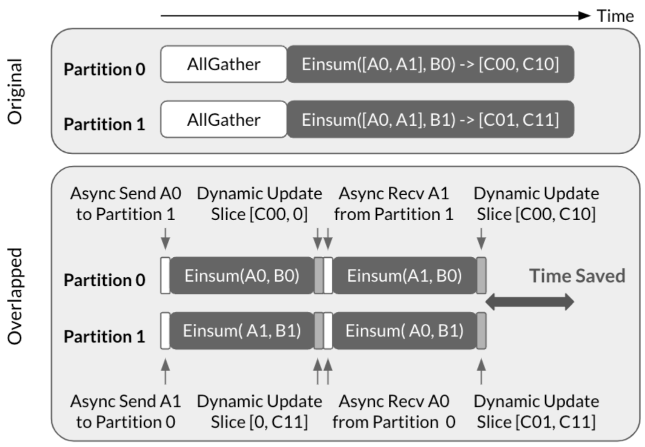
Time
Original
Partition 0
AllGather
Einsum([A0, A1], B0) -> [C00, C10]
Partition 1
AllGather
Einsum([A0, A1], B1) -> [C01, C11]
Overlapped
Async Send A0 to Partition 1
Dynamic Update Slice [C00, 0]
Async Recv A1 from Partition 1
Dynamic Update Slice [C00, C10]
Partition 0
Einsum(A0, B0)
Einsum(A1, B0)
Time Saved
Partition 1
Einsum(A1, B1)
Einsum(A0, B1)
Async Send A1 to Partition 0
Dynamic Update Slice [0, C11]
Async Recv A0 from Partition 0
Dynamic Update Slice [C01, C11]

Figure 4: An illustrative example on the key idea of *AllGather-Einsum* with 2-way intra-layer model parallelism. The upper original one illustrates how the operations are executed in a current system; the lower overlapped one illustrates how the proposed technique works. The latency of each operation shown in the figure (and the following one) is not accurate, and is just used to demonstrate the idea.

[4]Asynchronous *Send* and *Receive* operations are used in this and the following example for simplicity and easy understanding. The operations used in the the real implementation are asynchronous, non-blocking *CollectivePermutes*, which will be discussed in a later section.

[5]This operation [6] overwrites the value of a slice at the specified indices, which in this case is computed based on the device Partition ID.

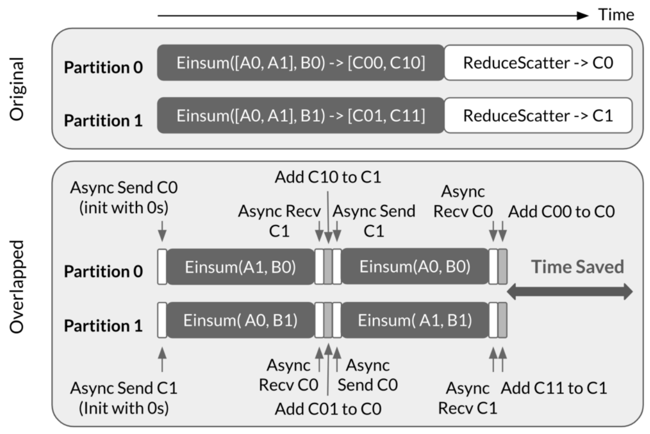

Figure 5: An illustrative example on the key idea of *Einsum-ReduceScatter* with 2-way intra-layer model parallelism.

Similarly, *ReduceScatter* can also be overlapped with the corresponding computation (shown in Figure 5). In this example, *C0* (the addition of *C00* and *C01*) and *C1* (the addition of *C10* and *C11*) are respectively the result shard of *C* after *ReduceScatter* on *Partition 0* and *Partition 1*. As the data communication is performed on the computation result, in this case, each device needs to asynchronously transfer the accumulation result shard rather than the operand shard. The accumulation result shard is initialized with zeros on each device. At the beginning of each iteration, each device asynchronously send the accumulation result shard (e.g., *C0* and *C1* on *Partition 0* and *Partition 1*, respectively, in the first iteration) to the other device, and start executing the partial *Einsum* in parallel with the data transfer. When the computation finishes, the partial result is added to the received accumulation result shard at the end of the iteration (e.g., *C1 += C10* and *C0 += C01* on *Partition 0* and *Partition 1*, respectively, in the first iteration). Note that the index used to perform *DynamicSlice* on the operand A[6] does not start from the same number as the device Partition ID. This is designed to make the final result shard ID align with its corresponding device Partition ID (i.e., *C0* and *C1* on *Partition 0* and *Partition 1*, respectively).

5 IMPLEMENTATION DETAILS

The proposed technique is implemented as compiler passes and works automatically for large deep learning models with intra-layer model parallelism. The decomposition part that generates sequences of finer-grained communication and computation operations is added as a dataflow graph transformation. Then a decoupled asynchronous, non-blocking *CollectivePermutes* generation and scheduling pass is added to perform the overlapping. At last, optimizations and a mechanism that automatically enables the feature

[6]Not explicitly shown in the figure, a *DynamicSlice* [5] on operand *A* to get the correct slice based on device Partition ID is required as *A* is replicated on each device. *A0* and *A1* shown in the figure correspond to the *DynamicSlice* result with index 0 and 1, respectively.

```
input  : $looped\_operand, local\_operand$
output : $result$
$result \leftarrow 0$                                                    /* Initialization. */
for $i \leftarrow 0$ to $N-1$ do                                         /* N equals to the number of partitions. */
    $local\_operand^* \leftarrow$ Select($local\_operand$, DynamicSlice($local\_operand$))
    $tmp \leftarrow$ Einsum($local\_operand^*$, $looped\_operand$)
    if $i < N-1$ or overlapping ReduceScatter then CollectivePermute(Select($looped\_operand, result$))
    $result \leftarrow$ Update($result, tmp$)
end
```

Algorithm 1: A generalized and simplified algorithm for *Looped CollectiveEinsum*. The *looped_operand* input refers to the initialized data shard that will be transferred among device partitions (e.g., Ax in Figure 4). The *local_operand* input refers to the data that will not be transferred (e.g., Bx in Figure 4). The `Update` can be either `Addition` or `DynamicUpdateSlice` depending on the actual case. The `Select` is not an operation that is actually used in the implementation, and is just used here to represent choosing among the two possible inputs.

for individual operation based on the estimated benefits are added to further improve the system performance. Details are discussed as follows.

5.1 Looped CollectiveEinsum

Following the idea shown in Section 4, the *Einsum* computation with the corresponding communication collective is broken into a sequence of operations with repetitive patterns, and is reconstructed as a loop (referred to as *Looped CollectiveEinsum* in later discussion for simplicity). Each iteration of the loop performs partial computation and related data transfer on different shards of data, with the data shard ID computed based on the loop index variable.

In the *AllGather-Einsum* example shown in Figure 4, the operand A is partitioned along a non-contracting dimension[7]. However, the actual partitioning strategy may choose another logical dimension for better system performance. Partitioning along a different logical dimension results in different handling of the partial computation input and output at each iteration. Assuming the partitioning strategy of the output is compatible with that of the Right-Hand Side operand (RHS), the three different cases to overlap *AllGather*s are as follows[8]:

- Case 1 — the Left-Hand Side operand (LHS) is partitioned along a non-contracting dimension. This is the simplest case (same as the one shown in Figure 4). At each iteration, the decomposed *Einsum* is directly computed using the received LHS and RHS, then a *DynamicUpdateSlice* is required to update the final result.
- Case 2 — the LHS is partitioned along a contracting dimension. In this case, the RHS needs to be *DynamicSlice*d first before feeding into the *Einsum*. The *DynamicSlice* is performed along the contracting dimension of the RHS with the slice size the same as the size of the corresponding, partitioned LHS dimension. The *DynamicUpdateSlice* after the *Einsum* in the previous case is replaced with an *Addition* to accumulate the partial result.
- Case 3 — the LHS is partitioned along a batch dimension. Similar to the Case 2, the RHS also needs a *DynamicSlice* to get the correct data shard for computation. The slice, however, is along the batch dimension with the size same as the RHS batch dimension. The *Einsum* computation is followed by a *DynamicUpdateSlice* along the batch dimension to update the final result.

discuss case 2 and 3 they are confusing

[7]The non-contracting dimension in this paper refers to a dimension that exists in only one of the operand and will not be "contracting" away in the output.
[8]The same applies if LHS and RHS are exchanged.

Einsum-ReduceScatter overlapping is performed when either the LHS or the RHS is partitioned along a contracting dimension, and the result is partitioned along a non-contracting dimension. Both the LHS and the RHS are examined to see which one has the non-contracting dimension that corresponds to the partitioned non-contracting dimension of the output. The found operand is then *DynamicSlice*d along the matched non-contracting dimension to get the correct input shard for the *Einsum*. The result update operation is an *Addition* to the corresponding output shard at the end of each iteration.

A generalized pseudocode of *Looped CollectiveEinsum* for both *AllGather* and *ReduceScatter* cases is shown in Algorithm 1. The above discussion has already covered most of it except the *CollectivePermutes*. The *CollectivePermutes* are used to exchange data shard (operand and result of *Einsum* for *AllGather* and *ReduceScatter* cases, respectively) among device partitions. To have a modular structure and provide better code generation, the proposed technique decouples the generation and scheduling of asynchronous, non-blocking *CollectivePermutes*s from the decomposition and loop generation. As shown in Algorithm 1, the default synchronous, blocking *CollectivePermutes* are used during the loop generation. A later compiler pass will transform these *CollectivePermutes* into asynchronous, non-blocking ones and schedule them to overlap with the corresponding *Einsums* (discussed in Section 5.2). The {*source*, *destination*} pairs of a *CollectivePermute* at each iteration are constructed as {0, N - 1}, {1, 0}, {2, 1}, ... {N - 1, N - 2}. Figure 6 and 7 respectively illustrate how data shards are transferred among device partitions at each iteration in the *AllGather* and *ReduceScatter* cases. The *Dx* blocks (with *x* being the shard ID) in a column represent the data shards used for computation by the corresponding device partition at each iteration. As can be seen in Figure 6 and 7, data shards are circular-shift left in both cases. However, they have slightly different patterns: data shard IDs are aligned with device Partition IDs at the beginning of the loop in the *AllGather* case, while the same pattern appears at the end in the *ReduceScatter* case. This is because the data shards are used as the operands of both *Einsum* and *CollectivePermute* in the *AllGather* case, but used after the *CollectivePermute* to accumulate the partial result in the *ReduceScatter* case.

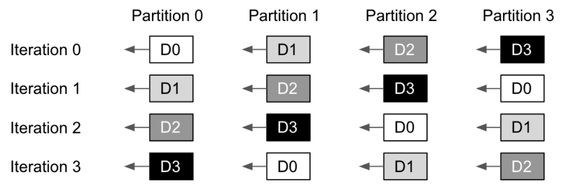

Figure 6: An illustrative example of data shard transfer in a looped *AllGather-Einsum* with 4-way intra-layer model parallelism.

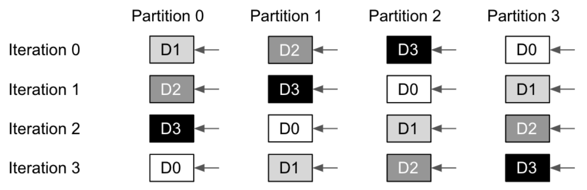

Figure 7: An illustrative example of data shard transfer in a looped *Einsum-ReduceScatter* with 4-way intra-layer model parallelism.

5.2 Asynchronous CollectivePermute and Scheduling

To enable the asynchronous, non-blocking behaviour, each *CollectivePermute* is broken into a pair of *CollectivePermuteStart* and *CollectivePermuteDone*. The *CollectivePermuteStart* simply starts the data transfer between each *source*, *destination* device pair. It does not block the following instructions[9], and takes almost no execution time. The *CollectivePermuteDone* is used to indicate the finish of the data transfer. Although the actual time spent on data transfer does not change with this asynchronous, non-blocking instruction pair, the observed execution time varies depending on how much data transfer can be overlapped with the computation. In one extreme case where no overlapping happens, the execution time of *CollectivePermuteStart* + *CollectivePermuteDone* is the same as that of the original *CollectivePermute*. On the other hand, when data communication takes no longer than the time spent on the overlapped computation, the asynchronous, non-blocking *CollectivePermuteStart/Done* pair consumes negligible execution time. Therefore, the scheduling of asynchronous *CollectivePermtueStart* and *CollectivePermuteDone* in an instruction sequence is vital to the overlapping performance.

To avoid dramatically changing the liveness of variables, the proposed scheduling algorithm takes the instructions that have been scheduled by the existing instruction scheduling pass (which uses an algorithm that tries to minimize the memory usage) as the inputs. Two approaches have then been tried to maximize the overlapping. One approach adopts a bottom-up way — instructions are reversely scheduled from the roots of an dataflow graph. Its simplified pseudocode is shown in Algorithm 2. The *ready_queue* keeps the instructions that are ready to be scheduled and prioritizes

[9]Instructions and operations are used interchangeably in this paper.

```
input  : $instr\_sequence$
output : $new\_instr\_sequence$
/* Initialization.                                                        */
create $new\_instr\_sequence$
$sched\_graph \leftarrow$ BuildSchedGraph($instr\_sequence$)
create $ready\_queue$
create $pending\_queue$
$async\_instr\_count \leftarrow 0$
$current\_time \leftarrow 0$
for each $root\_instr$ of $sched\_graph$ do
    $root\_instr.ready\_time \leftarrow 0$
    $ready\_queue.enqueue(root\_instr)$
end
/* Finds instructions to be scheduled (in reverse order) and updates states. */
while $ready\_queue$ is not empty or $pending\_queue$ is not empty do
    $candidate \leftarrow$ SelectNodeFromReadyQ($ready\_queue$, $async\_instr\_count$)
    if $candidate$ is NULL then
        $candidate \leftarrow$ SelectNodeFromPendingQ($pending\_queue$, $async\_instr\_count$)
    end
    $new\_instr\_sequence.append(candidate)$
    $async\_instr\_count \leftarrow$ UpdateAsyncInstrCount($candidate$, $async\_instr\_count$)
    $candidate.ready\_time \leftarrow 0$
    for each $instr$ in Successors($candidate$) do
        $candidate.ready\_time \leftarrow$ Max($instr.ready\_time + instr.latency$, $candidate.ready\_time$)
    end
    $current\_time \leftarrow candidate.ready\_time + candidate.latency$
    $ready\_queue, pending\_queue \leftarrow$ UpdateReadyQandPendingQ($current\_time$, $ready\_queue$, $pending\_queue$)
end
reverse $new\_instr\_sequence$
```

ReadingQueue prioritizies CollectiveDone ops

if nothing is actually ready, we must schedule something, so pull from pending

Algorithm 2: A bottom-up scheduling algorithm to hide communication overheads. The details of each helper function is ignored for simplicity.

*CollectivePermuteDone*s and their users[10]. The *pending_queue* keeps the instructions that have all of the user instructions scheduled but themselves are still not ready to be scheduled. It prioritize instructions that will get ready first. The readiness of an instruction is determined by comparing its estimated ready time to the *current* time. After initialization, the scheduling heuristic tries to find the next instruction to be scheduled in *ready_queue* (`SelectNodeFromReadyQ`) and *pending_queue* (`SelectNodeFromPendingQ`) following the priority rules. In the case that the chosen instruction is a *CollectivePermuteDone* and the total number of in-flight asynchronous instructions exceeds the budget (constrained by the limited number of synchronization flags in a system), the next instruction in the queue will be selected[11]. When the next scheduled instruction has been found, both *ready_queue* and *pending_queue* are updated based on the readiness and the data dependence of instructions (`UpdateReadyQandPendingQ`).

Another approach uses a top-down method with a simple-yet-effective idea — to schedule a *CollectivePermuteStart* as early as possible after its operand instructions haven been scheduled, and to schedule a *CollectivePermuteDone* as late as possible just before

[10]If none of the priority conditions is triggered, the original instruction order will be applied to keep reducing memory pressure.
[11]This may reduce the amount of overlapping. But fortunately, the maximum number of in-flight asynchronous *CollectivePermutes* generated by the *Looped CollectiveEinsum* can mostly be controlled under the budget.

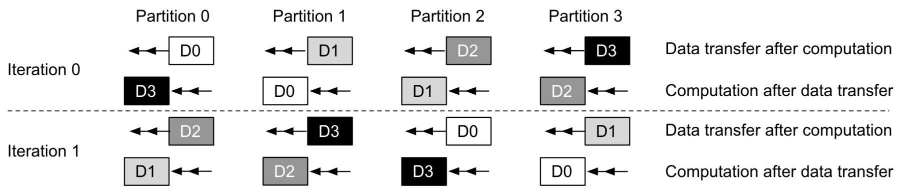

Figure 8: An illustrative example of a looped *Einsum-ReduceScatter* with unrolling degree of 2 and 4-way intra-layer model parallelism. Similar to Figure 7, a *Dx* block in the figure represents the data shard with ID *x* that is used for computation on the corresponding device partition at that particular iteration.

its usage. Instruction scheduling is quite straightforward following this rule. However, the original input instruction order may significantly reduce the chance of overlapping. To address this issue, the top-down scheme first rebalances the instructions among each *CollectivePermute* interval based on the runtime cost. After a chain of *CollectivePermutes* has been identified, the estimated average instruction execution time between each *CollectivePermute* in the chain is calculated, and used to redistribute *CollectivePermutes* in the instruction sequence. In addition, certain instruction that feeds into a *CollectivePermute* chain start is moved to an earlier position in the sequence to create more overlapping opportunities.

Both of the approaches are evaluated for comparison. As compared to the top-down approach, the bottom-up one is more general with the cost of implementation complexity. Starting from a single root instead of possibly multiple starting parameters that scatter in the beginning of a sequence, it avoids certain pattern-matching based instruction reordering in the top-down approach, and may find more overlapping opportunities. With its slightly better performance, the bottom-up one is used for the final overall evaluation.

5.3 Implementation in XLA Compiler

Both the decomposition and overlapping schemes discussed above are implemented in XLA, which is a domain-specific production compiler designed for linear algebra [3]. It transforms a machine learning model into a backend agnostic HLO graph, which is a dataflow graph that represents operations as nodes and tensors as edges. Many frontend frameworks such as Tensorflow [7], JAX [11], PyTorch [38], and Julia [10] have already had the lowering logic to get the intermediate representation. If a hardware platform has an compiler backend for XLA, the HLO graph can then be compiled into executable and run on the target devices (e.g., CPUs, GPUs and TPUs).

Implemented in XLA, the proposed technique automatically performs graph transformation and communication overlapping with the supported frontends and backends mentioned above. However, it is not restricted to XLA, and the idea can be applied to other machine learning systems and frameworks.

5.4 Performance Optimizations

This section describes the major optimizations on top of a naïve implementation to further reduce overheads and improve performance.

5.4.1 Loop Unrolling. A direct implementation of the proposed *Looped CollectiveEinsum* as described in Algorithm 1 creates extra nontrivial *Copy* operations in the loop. This is due to loop-carried aliasing of the transferred data. To eliminate the explicit copy overheads, loop unrolling with degree of 2 is added to provide a similar effect as double buffering. This provides extra benefit to the *Einsum-ReduceScatter* case. Without having loop unrolling, the asynchronous *CollectivePermuteDone* has to give up overlapping with the big *Einsum* as the result-update accumulation is fused with the *Einsum* and adds data dependence between the *CollectivePermuteDone* and the fusion node. Although this specific fusion can be disabled to allow overlapping, it also hurts the performance due to extra memory accesses.

Figure 8 illustrates how to improve the performance of looped *Einsum-ReduceScatter* with loop unrolling. Instead of keeping one copy of each partial result shard that is accumulated along each device partition, the proposed scheme separates the accumulation chain into two interleaved parts. For example, the first partial result of *D0* is obtained by accumulating data from *Partition 0* and *2*; the second half is from *Partition 1* and *3*. The final result shard is computed by adding these two halves together. At each unrolled iteration, the first part transfers result after the accumulation, while the second part performs the accumulation after the data transfer. As the two separate paths are independent from each other, the asynchronous *CollectivePermuteDones* can be successfully overlapped with the *Einsums* (from the other path), even if an *Einsum* is fused with the corresponding accumulation. An extra epilogue process needs to be added for correctness (not shown in the figure)[12]. After iteration 1, the partial accumulated result pairs that end up in device *Partition 0 - 3* are {*D0*, *D1*}, {*D1*, *D2*}, {*D2*, *D3*}, {*D3*, *D0*}, respectively. An extra *CollectivePermute* with {*source*, *destination*} pairs of {0, 1}, {1, 2}, {2, 3}, {3, 0} is required for the second item of each result pair to align the partial results, which are then summed up as the final result shard on each device partition. The overhead of the epilogue process is small as compared to the loop. In addition, the asynchronous, non-blocking *CollectivePermtue* in the epilogue can also be overlapped with other computation to further reduce the cost.

5.4.2 Bidirectional Data Transfer. With the increasing number of partitions, the amount of computation within each partitioned *Einsum* may not be large enough to overlap with the corresponding data communication. In this case, bidirectional data transfer is applied to mitigate the problem. By concatenating the operands of two partial *Einsums* and executing them as a single operation, it

[12]This is *Einsum-ReduceScatter* specific requirement and not needed in the *AllGather-Einsum* case.

effectively doubles the amount of computation at each iteration. Furthermore, the bidirectional data transfer can fully utilize the dedicated InterChip Interconnects (ICI) links on TPUs that provide high bandwidth on both directions [26, 36], which helps improving the system performance with the proposed overlapping technique in general.

Figure 9 and 10 respectively illustrate the data communication pattern in the *AllGather-Einsum* and *Einsum-ReduceScatter* overlapping cases. Loop unrolling has also been applied in the actual implementation, but not shown in the examples for simplicity. In both examples, each device partition works on two *Dx* data blocks and transfers them in different directions (shown by the arrows): the one on the left side is sent counterclockwise; the other one on the right side is sent clockwise. For the *AllGather-Einsum* case, a prologue process is added before the loop. It simply shifts each local data shard clockwise by one to initialize the correct data shards on each device partition. For the *Einsum-ReduceScatter* case, an epilogue is required to align the result shards and perform the summation. With the clockwise accumulated partial result shard (the one on the right side) further shifted clockwise by one, the two partial result shards on each device partition have the same data shard ID as the device Partition ID. The final scattered result can be easily obtained with an extra *Addition* operation. When combined with loop unrolling optimization, the amount of work required by the epilogue will still be the same with no more data communication.

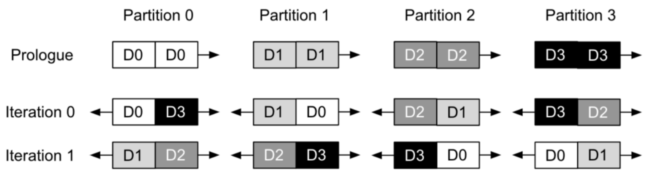

Figure 9: An illustrative example of data shard transfer in a looped *AllGather-Einsum* with bidirectional data communication and 4-way intra-layer model parallelism.

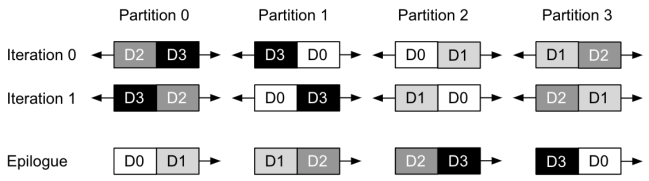

Figure 10: An illustrative example of data shard transfer in a looped *Einsum-ReduceScatter* with bidirectional data communication and 4-way intra-layer model parallelism.

5.4.3 Fusion Optimization. Operation fusion is an effective way to reduce slow main memory accesses and kernel launch overheads. As one of the most important optimizations, operation fusion is automatically performed in XLA with heuristics. To allow more memory-bound element-wise operations to be fused in the generated *CollectiveEinsum* loop, some local graph rewrites are performed to be more fusion-friendly. With bidirectional data transfer, the local operand of the *Einsum* is typically constructed by concatenating two data shards each of which is *DynamicSlice*d based on the runtime shard ID. Directly using *DynamicSlice*s and *Concatenation* operation makes it hard to be used with the *Einsum* in the current XLA fusion pass. Instead, the proposed scheme first extends the dimension of each operand by one, and then explicitly replaces the *Concatenation*(a, b) with a semantically equivalent operation collection of *Max*(*PadLow*(a), *PadHigh*(b)) on the extended dimension. The original *DynamicSlice* on a or b is inserted right before the *Max* on each padded value to make the final result the same as before. With this change, the entire local operand pre-processing can be fused with the corresponding *Einsum* user. For the other looped operand, more operations can be fused by simply exchanging the order of corresponding *Reshape* and *Concatenation*. In addition, the fusion of result accumulation post-processing in the *Einsum-ReduceScatter* case is enabled by exchanging the order of involved *Reshape* and *Slice*.

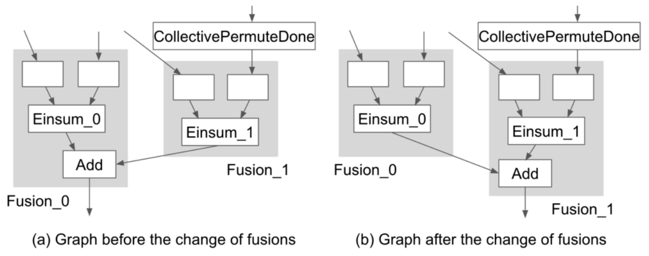

Figure 11: Example HLO graphs with different fusion decisions.

Some fusions constructed based on the default heuristic may hurt the overlapping performance. Figure 11 (a) illustrates a simplified graph with default fusion decisions. In the figure, a grey box represents a fusion node; a white blank box represent one or more fused HLO instructions (names are not shown for simplicity). The graph shows a typical pattern in an unrolled loop — the results of two *Einsum*, one of which has asynchronous *CollectivePermuteDone* as input, are accumulated as the updated result at that iteration. As the *Einsum_0* is independent from the *CollectivePermuteDone*, it is expected to execute in parallel with the asynchronous data communication for overlapping. However, the *Addition* fused with the *Einsum_0* creates a data dependence between *Fusion_0* and *CollectivePermuteDone*, resulting in sequential execution of the three nodes. To avoid this bad fusion, the fusion heuristic is changed to prioritize fusing the *Addition* with the *Einsum* that has an operand from asynchronous *CollectivePermuteDone*. With the newly generated graph (shown in Figure 11 (b)), the data communication can successfully overlap with the *Fusion_0*.

5.5 Enabling the Overlap Feature Based on Estimated Benefits

In the proposed scheme, data are asynchronously transferred along logical rings formed by the device partitions[13]. When the original

[13]More complicated 2D or 3D algorithms can also be used. We choose the 1D algorithm for implementation simplicity.

Table 1: Evaluated Applications.

	GPT_1T [12]	**Meena_500B** [8]	**MLPerf_200B** [2]	**T5_300B** [41]	**GLaM_1T** [20]	**BigSSL_10B** [55]
Number of Parameters	1.03T	507B	199B	290B	1.16T	10.4B
Number of Layers	142	120	66	64	32	48
Size of Model Dimension	24576	18432	12288	12288	8192	3072
Size of Feedforward Dimension	98304	65536	98304	36864	32768	12288
Batch Size	4096	2048	4096	3072	1024	64
Number of TPU Chips	2048	1024	1024	512	1024	128

Table 2: Evaluated GPT models scaled from 32 billion to 1 trillion parameters.

	32B	**64B**	**128B**	**256B**	**512B**	**1T**
Number of Parameters	32.2B	64.2B	128.6B	257.7B	513.4B	1.0T
Number of Layers	40	51	71	80	102	142
Size of Model Dimension	8192	10240	12288	16384	20480	24576
Size of Feedforward Dimension	32768	40960	49152	65536	81920	98304
Batch Size	512	512	1024	2048	3072	4096
Number of TPU Chips	64	128	256	512	1024	2048

AllGather or *ReduceScatter* is performed on a mesh or torus, the overall runtime of the decomposed sequence of *CollectivePermute*s might be longer than that of the original one, as it utilizes only half of the interconnect bandwidth (along a single dimension). Although the asynchronous, non-blocking *CollectivePermute*s are executing in parallel with computation, the system performance becomes worse when the amount of computation is not large enough to cover the extended data communication time. To prevent this, a general mechanism is added to automatically enable the overlap feature for each *AllGather-Einsum* or *Einsum-ReduceScatter* only when we see benefits. The estimation is based on equation $comp\_t + comm\_t >= max(comp\_t,\ comm\_t\_ring) + extra\_t$. $comp\_t$ and $comm\_t$ respectively represent the time spent on the original computation and communication. $comm\_t\_ring$ represents communication time along the logical ring. The time spend on prologue or epilogue is also added as $extra\_t$, conservatively assuming the extra *CollectivePermute* outside the loop cannot find overlapped computation. Both the computation and communication time are simply estimated against the peak FLOPS and interconnect bandwidth. If the above equation is not satisfied, the original operations will remain unchanged instead of being decomposed and overlapped.

When an *Einsum* has two *AllGather* (or one *AllGather* and one *ReduceScatter*) candidates to overlap with, the proposed scheme chooses the one that leads to higher benefits. If the *Einsum* is estimated to be faster than both communication collectives, the collective operation that has smaller data shard size will be selected as it has smaller extra *CollectivePermute* overhead outside of the loop (assuming it cannot be overlapped in the worst case). Otherwise, the collective operation with longer estimated execution time will be selected for decomposition and overlapping.

6 EVALUATION

Different types of large models (Table 1) are used for evaluation to show the effectiveness of the proposed technique. For this table and the next Table 2, size of the model dimension is the number of units after embedding and in each bottleneck layer. Size of feedforward dimension the width of the feedforward layers. These are representative hyperparameters that determine the size of each model. We use the same terminologies as the GPT-3 paper [12]. The GPT_1T uses the model architecture of the well-known autoregressive language model GPT-3 and scales it up to the size of 1T parameters. The Meena_500B builds on the original conversational chatbot but uses much more parameters to provide better quality. The MLPerf_200B uses the same giant BERT [19] model that Google submitted to MLPerf v1.1 open category. The T5_300B is a unified text-to-text Transformer [49] with the classical encoder-decoder architecture. Similar to the GPT and the Meena, it is also scaled up from the original size. The GLaM_1T is a new language model with a sparsely activated mixture-of-experts architecture. We use its largest configuration with 64 experts. The last one — BigSSL_10B — is a relatively small automatic speech recognition model. It's slightly larger than the one in the original paper, and we use intra-layer instead of pipeline model parallelism to partition the model. Because of its relatively small size, partitioning along one dimension (size of 8 on the 128-chip device mesh) is enough to fit the model in memory. All of the rest models are fully partitioned using intra-layer model parallelism across the entire device mesh with partitioning strategies that have two partitioned dimensions. The partitioning strategies were chosen to deliver the best performance of the baselines. Both the baseline and the proposed technique are evaluated with the same number of chips and the same partitioning strategy for each model.

Among these six applications, GPT_1T, MLPerf_200B, and T5 _300B uses JAX as the frontend framework, and the rest three are implemented with Tensorflow. We use TPU v4 Pods [1] with an XLA compiler backend to run all of the models. The number of TPU chips used for each model varies based on the actual requirement, as shown in Table 1.

6.1 Overall Performance

Figure 12 presents the system performance with and without the proposed technique. The achieved system throughput of each application is normalized to the corresponding peak FLOPS (which varies based on the number of used TPU chips). On average, the proposed technique improves the performance by 1.2x, with the highest speedup of 1.38x on MLPerf_200B. Three out of the four dense models with two partitioned dimensions have higher than 60% FLOPS utilization (the highest one is 72% achieved by Meena_500B). The

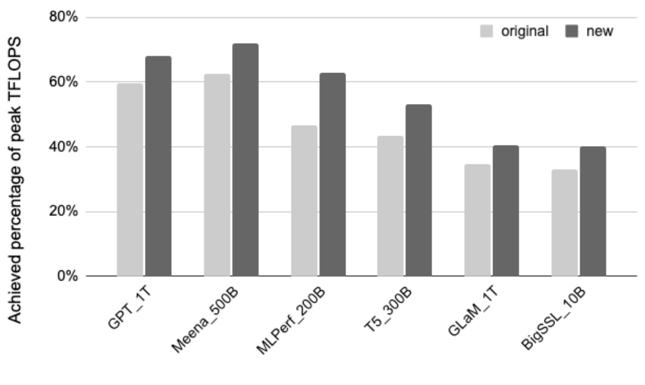

Figure 12: Performance of the evaluated applications.

slightly lower utilization of T5_300B is due to the *AlltoAll* communications generated in the backward propagation, which contributes to about 10% of the runtime. With a better partitioning configuration that can remove the *AlltoAll* operations, it can achieve similar FLOPS utilization as other large dense models. The GLaM_1T with sparse mixture-of-experts architecture and the BigSSL_10B using 1D partitioning achieve around 40% FLOPS utilization. As compared to the large dense models, they have narrower model architecture that the computation is not large enough to cover the data communication even they are overlapped. In addition, there are still data communication operations that cannot be overlapped with the current technique (e.g., *AlltoAll*s and *AllGather*s that cannot be decomposed with dependent *Einsum*s). These communication costs can be further reduced by combining with other techniques such as offloading independent communications [43], which is left for future work.

6.2 Weak Scaling Case Study

To better understand how the proposed technique scales with model sizes, we conduct a weak scaling study using the GPT model as an example. Table 2 lists the evaluated GPT models with number of parameters ranging from 32 billion to 1 trillion. With 1.1 - 1.4x speedup, the proposed technique consistently improves the performance across all sizes, shown in Figure 13.

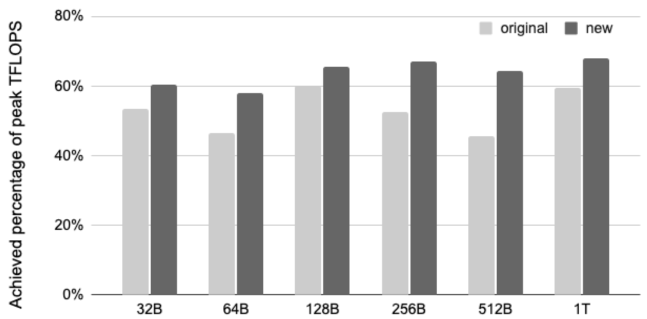

Figure 13: Performance of the weakly scaled GPT models.

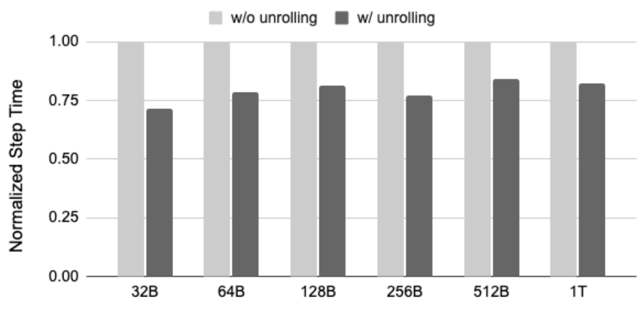

Figure 14: Performance improvements provided by loop unrolling.

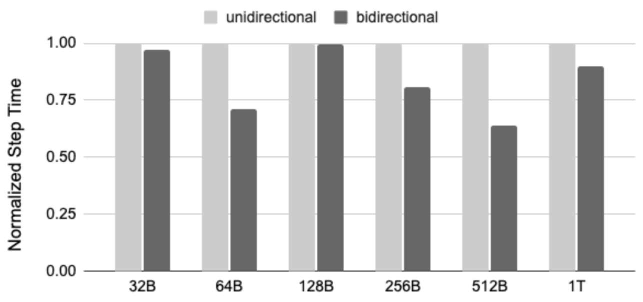

Figure 15: Performance improvements provided by bidirectional data transfer.

6.3 Effectiveness of Performance Optimization

Figure 14 and 15 respectively show the performance improvements brought by the loop unrolling and bidirectional data transfer optimizations. For simplicity, we also use the set of scaled GPT models for the evaluations. Execution time per step (normalized to the baseline) is used as y-axis in each figure. The loop unrolling achieves similar performance improvements across different model sizes (Figure 14). However, the effectiveness of bidirectional data transfer varies among models with different sizes. For example, both GPT_32B and GPT_128B see smaller than 5% improvement with bidirectional data transfer. This is because the number of partitioning along the dimension that applies the overlapping technique is relatively small in these two cases. Therefore the time spent on computation is long enough to cover most of the data communication, even with unidirectional data transfer.

We also compare the system performance with two different scheduling approaches discussed in Section 5.2. As shown in Figure 16, the bottom-up approach performs slightly better with 5% speedup on average. As it performs better and is more general, we choose the second approach to hide the communication latency.

6.4 Energy Consumption Reduction

The proposed technique does not increase the power consumption of the evaluated system. As the executed operations frequently switch between communication and computation when running a model, the computational units cannot be put into sleep and save

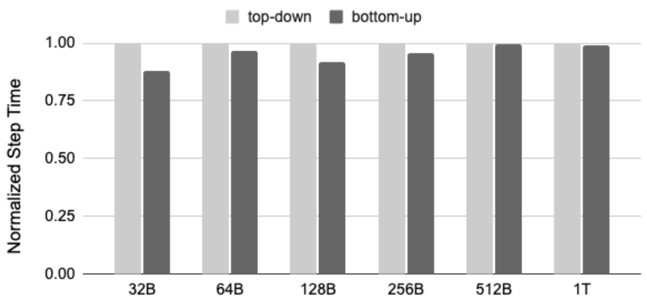

Figure 16: Performance comparison of the two scheduling approaches.

energy even if they are idle and waiting for synchronous communication results (without the capability of overlapping). Following the methodology in [39], the proposed technique achieves 1.14 - 1.38x energy consumption reduction, which comes from the end-to-end execution time improvement.

7 DISCUSSION

This section discusses some qualitative analyses and future works.

7.1 Application to Inference Tasks

Although the proposed overlapping scheme is mainly evaluated for training tasks to show the improvements on system throughput, it can also be applied to inference tasks to effectively hide the communication latency. For example, an in-house recommendation inference model with 2-way intra-layer model parallelism achieves 2x latency improvement with the proposed overlapping technique. A thorough study on different types of inference workloads is left for future work.

7.2 Application to Other Models and Systems

The models evaluated in this paper are mostly natural language processing (NLP) and speech models, as they are the most well-known large-scale deep learning models. Computer vision models that employ convolutional neural network (CNN) architectures are typically small enough to fit in the memory of a single accelerator, therefore do not need the proposed technique to reduce communication overheads. However, for emerging multilayer-perceptron (MLP)-based or attention-based computer vision models [54] and image generation models [42] that are compute-intensive and require model parallelism to fit in accelerator memories, the proposed technique can be applied to improve the performance. For deep learning models that spend most of the time on large embedding layers, the proposed technique may not work well as these models are more likely to have different model partitioning strategies with different communication patterns. These studies are left for future work.

Though the proposed technique is implemented and evaluated in XLA (details can be found in Section 5.3), the idea can also be applied to other compilers (e.g., TVM [14], Glow [44]) or ML frameworks (e.g., PyTorch [38], Mesh-Tensorflow [45]) that use similar communication primitives to run large scale deep learning models. In general, the part that generates the *Looped CollectiveEinsums* can be implemented as a graph rewrite or optimization. The back-end support on the asynchronous *CollectivePermute* with proper scheduling is also needed to demonstrate performance benefits.

Similarly, the overlapping idea can be applied to other hardware ML systems, such as GPU clusters connected via high-bandwidth and low-latency NVLink Network interconnects [4]. For systems that employ interconnects with low performance and therefore have very long data communication time that cannot be covered by the concurrent computation, the benefits of the proposed technique will be reduced. In this case, other methods to reduce the amount of required communication should be considered to further improve the performance.

7.3 Other Model Parallelism Patterns

Pipeline model parallelism [24, 29, 33, 34, 53] is another technique that have been proposed for large deep learning models to address the scaling challenges. With pipeline model parallelism, the (sub)layers of a model is partitioned into stages across multiple devices. A batch is split into several smaller microbatches, and executed as a pipeline through the partitioned stages: result of a microbatch flows downstream in the forward pass and upstream in the backward pass. To preserve strict weight update semantics in synchronous training (consistent weights in both forward and backward pass for a particular batch), the pipeline needs to be periodically flushed at the end of each batch [24]. This introduces *pipeline bubble* when devices are idle during the training time. Increasing the ratio of the number of microbatches to the number of pipeline stages can improve training efficiency. However, the required large global training batch size may further increase memory pressure and lead to poorer generalization and model quality degradation [28]. Asynchronous weight update and bounded weight delay techniques [29, 33, 34, 53] were proposed to avoid completely flushing the pipeline, but the relaxed weight update semantics may affect model convergence.

Some recent works (e.g., [35]) combine both intra-layer and pipeline model parallelism with data parallelism to achieve efficient training on GPU clusters. The proposed overlapping technique can still be applied in these cases to further improve the system performance. By reducing the communication cost introduced by intra-layer model parallelism, the proposed technique changes the performance trade-offs between different types of parallelism. This also provides new optimization opportunities to find a better parallelism combination going forward.

8 CONCLUSION

When running large deep learning models with intra-layer model parallelism, current systems spend a significant amount of time waiting for expensive communication collectives to prepare data for the following dependent computation operations. This paper propose a novel technique to address the issue. By overlapping the non-blocking, point-to-point decomposed communication collectives with the corresponding computation operations, the proposed technique can significantly reduce the communication cost and improve system throughput. Although the performance is evaluated on TPU systems, the idea can be applied to other distributed accelerator systems (e.g., GPU clusters) as well. We conclude that the proposed technique holds the potential to effectively improve FLOPS utilization in future large deep learning systems.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers and our shepherd, Prof. Chen Ding, for their valuable feedback.

REFERENCES

[1] 2020. *Google breaks AI performance records in MLPerf with world's fastest training supercomputer.* https://cloud.google.com/blog/products/ai-machine-learning/google-breaks-ai-performance-records-in-mlperf-with-worlds-fastest-training-supercomputer

[2] 2021. *MLPerf Training v1.1.* https://mlcommons.org/en/training-normal-11/

[3] 2021. *XLA: Optimizing Compiler for TensorFlow.* https://www.tensorflow.org/xla

[4] 2022. NVIDIA H100 Tensor Core GPU Architecture. https://www.hpctech.co.jp/catalog/gtc22-whitepaper-hopper_v1.01.pdf

[5] 2022. *XLA DynamicSlice Semantics.* https://www.tensorflow.org/xla/operation_semantics#dynamicslice

[6] 2022. *XLA DynamicUpdateSlice Semantics.* https://www.tensorflow.org/xla/operation_semantics#dynamicupdateslice

[7] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA, 265–283.

[8] Daniel Adiwardana, Minh-Thang Luong, David R. So, Jamie Hall, Noah Fiedel, Romal Thoppilan, Zi Yang, Apoorv Kulshreshtha, Gaurav Nemade, Yifeng Lu, and Quoc V. Le. 2020. Towards a Human-like Open-Domain Chatbot. *CoRR* abs/2001.09977 (2020). arXiv:2001.09977 https://arxiv.org/abs/2001.09977

[9] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. 1995. A Three-Dimensional Approach to Parallel Matrix Multiplication. *IBM J. Res. Dev.* 39, 5 (sep 1995), 575–582.

[10] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2014. Julia: A Fresh Approach to Numerical Computing. *CoRR* abs/1411.1607 (2014). arXiv:1411.1607 http://arxiv.org/abs/1411.1607

[11] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs.* http://github.com/google/jax

[12] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. *CoRR* abs/2005.14165 (2020). arXiv:2005.14165 https://arxiv.org/abs/2005.14165

[13] Lynn Elliot Cannon. 1969. *A Cellular Computer to Implement the Kalman Filter Algorithm.* Ph. D. Dissertation. USA. AAI7010025.

[14] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: End-to-End Optimization Stack for Deep Learning. abs/1802.04799 (2018). arXiv:1802.04799 http://arxiv.org/abs/1802.04799

[15] A. Danalis, Ki-Yong Kim, L. Pollock, and M. Swany. 2005. Transformations to Parallel Codes for Communication-Computation Overlap. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing.* 58–58. https://doi.org/10.1109/SC.2005.75

[16] Anthony Danalis, Lori Pollock, Martin Swany, and John Cavazos. 2009. MPI-Aware Compiler Optimizations for Improving Communication-Computation Overlap. In *Proceedings of the 23rd International Conference on Supercomputing* (Yorktown Heights, NY, USA) *(ICS '09)*. 316–325.

[17] A. Darte, D. Chavarria-Miranda, R. Fowler, and J. Mellor-Crummey. 2002. Generalized multipartitioning for multi-dimensional arrays. In *Proceedings 16th International Parallel and Distributed Processing Symposium.* 10 pp–.

[18] Eliezer Dekel, David Nassimi, and Sartaj Sahni. 1981. Parallel Matrix and Graph Algorithms. *SIAM J. Comput.* 10, 4 (1981), 657–675.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805 http://arxiv.org/abs/1810.04805

[20] Nan Du, Yanping Huang, Andrew M. Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, Barret Zoph, Liam Fedus, Maarten Bosma, Zongwei Zhou, Tao Wang, Yu Emma Wang, Kellie Webster, Marie Pellat, Kevin Robinson, Kathy Meier-Hellstern, Toju Duke, Lucas Dixon, Kun Zhang, Quoc V Le, Yonghui Wu, Zhifeng Chen, and Claire Cui. 2021. GLaM: Efficient Scaling of Language Models with Mixture-of-Experts. arXiv:2112.06905 [cs.CL]

[21] Message P Forum. 1994. *MPI: A Message-Passing Interface Standard.* Technical Report. USA.

[22] Evangelos Georganas, Jorge González-Domínguez, Edgar Solomonik, Yili Zheng, Juan Touriño, and Katherine Yelick. 2012. Communication Avoiding and Overlapping for Numerical Linear Algebra. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '12)*. Article 100, 11 pages.

[23] Jichi Guo, Qing Yi, Jiayuan Meng, Junchao Zhang, and Pavan Balaji. 2016. Compiler-Assisted Overlapping of Communication and Computation in MPI Applications. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. 60–69. https://doi.org/10.1109/CLUSTER.2016.62

[24] Yanping Huang, Yonglong Cheng, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, and Zhifeng Chen. 2018. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. *CoRR* abs/1811.06965 (2018). arXiv:1811.06965 http://arxiv.org/abs/1811.06965

[25] Kazuaki Ishizaki, H. Komatsu, and T. Nakatani. 2004. A Loop Transformation Algorithm for Communication Overlapping. *International Journal of Parallel Programming* 28 (2004), 135–154.

[26] Norman P. Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. 2020. A Domain-Specific Supercomputer for Training Deep Neural Networks. *Commun. ACM* 63, 7 (jun 2020), 67–78.

[27] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. *CoRR* abs/2001.08361 (2020). arXiv:2001.08361 https://arxiv.org/abs/2001.08361

[28] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. 2016. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. *CoRR* abs/1609.04836 (2016). arXiv:1609.04836 http://arxiv.org/abs/1609.04836

[29] Atli Kosson, Vitaliy Chiley, Abhinav Venigalla, Joel Hestness, and Urs Köster. 2020. Pipelined Backpropagation at Scale: Training Large Models without Batches. *CoRR* abs/2003.11666 (2020). arXiv:2003.11666 https://arxiv.org/abs/2003.11666

[30] 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. *CoRR* abs/2006.16668 (2020). arXiv:2006.16668 https://arxiv.org/abs/2006.16668

[31] Nilesh Mahajan, Sajith Sasidharan, Arun Chauhan, and Andrew Lumsdaine. 2012. Automatically Generating Coarse Grained Software Pipelining from Declaratively Specified Communication. (05 2012).

[32] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie Amy Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, K. R. Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. 2021. High-performance, Distributed Training of Large-scale Deep Learning Recommendation Models. *CoRR* abs/2104.05158 (2021). arXiv:2104.05158 https://arxiv.org/abs/2104.05158

[33] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized Pipeline Parallelism for DNN Training *(SOSP '19)*. New York, NY, USA, 1–15. https://doi.org/10.1145/3341301.3359646

[34] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. 2020. Memory-Efficient Pipeline-Parallel DNN Training. *CoRR* abs/2006.09503 (2020). arXiv:2006.09503 https://arxiv.org/abs/2006.09503

[35] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient Large-Scale Language Model Training on GPU Clusters. *CoRR* abs/2104.04473 (2021). arXiv:2104.04473 https://arxiv.org/abs/2104.04473

[36] Thomas Norrie, Nishant Patil, Doe Hyun Yoon, George Kurian, Sheng Li, James Laudon, Cliff Young, Norman Jouppi, and David Patterson. 2021. The Design Process for Google's Training Chips: TPUv2 and TPUv3. *IEEE Micro* 41, 2 (2021), 56–63.

[37] Ali Alvi Paresh Kharya. 2021. *Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, the World's Largest and Most Powerful Generative Language Model.* https://developer.nvidia.com/blog/using-deepspeed-and-megatron-to-train-megatron-turing-nlg-530b-the-worlds-largest-and-most-powerful-

generative-language-model/

[38] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. 8024–8035.

[39] David A. Patterson, Joseph Gonzalez, Quoc V. Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David R. So, Maud Texier, and Jeff Dean. 2021. Carbon Emissions and Large Neural Network Training. *CoRR* abs/2104.10350 (2021). arXiv:2104.10350 https://arxiv.org/abs/2104.10350

[40] Simone Pellegrini, Torsten Hoefler, and Thomas Fahringer. 2012. Exact Dependence Analysis for Increased Communication Overlap. In *Recent Advances in the Message Passing Interface*. Springer Berlin Heidelberg, Berlin, Heidelberg, 89–99.

[41] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2019. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *CoRR* abs/1910.10683 (2019). arXiv:1910.10683 http://arxiv.org/abs/1910.10683

[42] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. 2021. Zero-Shot Text-to-Image Generation. https://arxiv.org/abs/2102.12092

[43] Saeed Rashidi, Matthew Denton, Srinivas Sridharan, Sudarshan Srinivasan, Amoghavarsha Suresh, Jade Nie, and Tushar Krishna. 2021. Enabling Compute-Communication Overlap in Distributed Deep Learning Training Platforms. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (Virtual Event, Spain) *(ISCA '21)*. 540–553.

[44] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Nadathur Satish, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. 2018. Glow: Graph Lowering Compiler Techniques for Neural Networks. abs/1805.00907 (2018). arXiv:1805.00907 http://arxiv.org/abs/1805.00907

[45] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake A. Hechtman. 2018. Mesh-TensorFlow: Deep Learning for Supercomputers. *CoRR* abs/1811.02084 (2018). arXiv:1811.02084 http://arxiv.org/abs/1811.02084

[46] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *CoRR* abs/1909.08053 (2019). arXiv:1909.08053 http://arxiv.org/abs/1909.08053

[47] Edgar Solomonik and James Demmel. 2011. Communication-Optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II* (Bordeaux, France) *(Euro-Par'11)*. 90–109.

[48] Robert A. van de Geijn and Jerrell Watts. 1995. *SUMMA: Scalable Universal Matrix Multiplication Algorithm.* Technical Report. USA.

[49] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, Vol. 30. https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[50] Boxiang Wang, Qifan Xu, Zhengda Bian, and Yang You. 2021. 2.5-dimensional distributed model training. *CoRR* abs/2105.14500 (2021). arXiv:2105.14500 https://arxiv.org/abs/2105.14500

[51] Wikipedia. 2022. Einstein notation — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Einstein%20notation&oldid=1083457917. [Online; accessed 21-June-2022].

[52] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, Ruoming Pang, Noam Shazeer, Shibo Wang, Tao Wang, Yonghui Wu, and Zhifeng Chen. 2021. GSPMD: General and Scalable Parallelization for ML Computation Graphs. arXiv:2105.04663 [cs.DC]

[53] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher R. Aberger, and Christopher De Sa. 2019. PipeMare: Asynchronous Pipeline Parallel DNN Training. *CoRR* abs/1910.05124 (2019). arXiv:1910.05124 http://arxiv.org/abs/1910.05124

[54] Xiaohua Zhai, Alexander Kolesnikov, Neil Houlsby, and Lucas Beyer. 2021. Scaling Vision Transformers. arXiv:2106.04560 [cs.CV]

[55] Yu Zhang, Daniel S. Park, Wei Han, James Qin, Anmol Gulati, Joel Shor, Aren Jansen, Yuanzhong Xu, Yanping Huang, Shibo Wang, Zongwei Zhou, Bo Li, Min Ma, William Chan, Jiahui Yu, Yongqiang Wang, Liangliang Cao, Khe Chai Sim, Bhuvana Ramabhadran, Tara N. Sainath, Françoise Beaufays, Zhifeng Chen, Quoc V. Le, Chung-Cheng Chiu, Ruoming Pang, and Yonghui Wu. 2021. BigSSL: Exploring the Frontier of Large-Scale Semi-Supervised Learning for Automatic Speech Recognition. arXiv:2109.13226 [eess.AS]

Received 2022-03-31; accepted 2022-06-16