PrimePar: Efficient Spatial-temporal Tensor Partitioning for Large Transformer Model Training

Haoran Wang
wanghaoran20g@ict.ac.cn
Institute of Computing Technology, Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China

Lei Wang
wanglei21s@ict.ac.cn
Institute of Computing Technology, Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China

Haobo Xu
xuhaobo@ict.ac.cn
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China

Ying Wang
wangying2009@ict.ac.cn
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China

Yuming Li
liyuming22@mails.ucas.ac.cn
Institute of Computing Technology, Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China

Yinhe Han
yinhes@ict.ac.cn
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China

Abstract

With the rapid up-scaling of transformer-based large language models (LLM), training these models is becoming increasingly demanding on novel parallel training techniques. Tensor partitioning is an extensively researched parallel technique, encompassing data and model parallelism, and has a significant influence on LLM training performance. However, existing state-of-the-art parallel training systems are based on incomplete tensor partitioning space, where the distribution of partitioned sub-operators is limited to the spatial dimension. We discover that introducing the temporal dimension into tensor partitioning of LLM training instance provides extra opportunities to avoid collective communication across devices, saving memory space and also overlapping device-to-device communication with computation. In this paper, we propose a new tensor partition primitive that distributes sub-operators along both the spatial and temporal dimensions to further explore communication and memory overhead reduction over current solutions. This new primitive creates a broader parallelization space and leads to parallel solutions that achieve better training throughput with lower peak memory occupancy compared to state-of-the-art techniques. To efficiently deploy optimized parallel transformer model training to multiple devices, we further present an optimization algorithm that can find optimal parallel solutions from our spatial-temporal tensor partition space with acceptable search time. Our evaluation shows that our optimized tensor partitioning achieves up to 1.68 × training throughput with 69% peak memory occupancy compared to state-of-the-art distributed training systems when training LLMs. Upon scaling to 32 GPUs, the geo-mean speedup across benchmarks is 1.30 ×. When applied in 3D parallelism, up to 1.46 × training throughput can be achieved.

ACM Reference Format:
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 *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3620666.3651357

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0386-7/24/04.
https://doi.org/10.1145/3620666.3651357

1 INTRODUCTION

With the recent advances in artificial intelligence, the continuous scaling of transformer models [56] has significantly enhanced model accuracy, generalization, and self-supervised capabilities. Transformer-based large language models with tens or hundreds of billions of parameters have achieved state-of-the-art performance on a wide range of tasks [2, 7, 14, 43, 44, 54, 55, 60, 62, 64, 66].

With the exponential growth in parameter scale, training transformer models is becoming increasingly demanding on hardware resources, not only in terms of individual device computational power, but also in terms of interconnection bandwidth and memory. Responsively, NVIDIA has introduced NVSwitch to support high-performance communication between cross-node GPUs [6]. TPU v4 proposed optical circuit switching to improve networking, which can

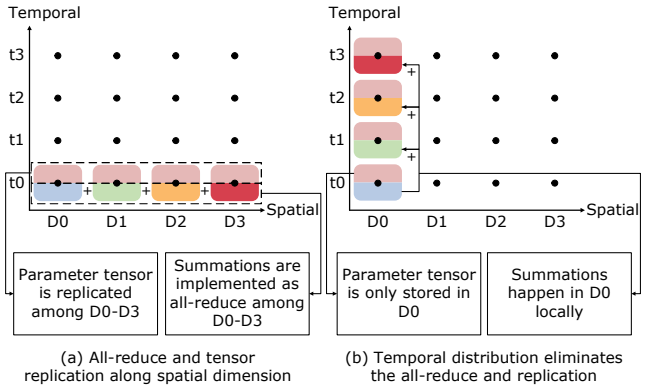


Figure 1. Illustration of distributing 4 sub-operators along spatial dimension (a) and temporal dimension (b). The horizontal axis represents 4 devices (D0-D3) while the vertical axis represents 4 temporal steps (t0-t3). Each rounded rectangle represents a sub-operator, with the upper and lower representing two tensors. The same color represents identical tensors, while different colors indicate distinct partial sums that require summation.

adjust interconnection topology to match the application and deliver more bandwidth [22].

Parallel training systems are required to adapt to the development of transformer models and hardware resources, so that the expensive computational power, interconnection bandwidth and memory resources are efficiently utilized. Current common practice for training transformer models is resorting to 3D parallelism [37, 45, 49], including data, model and pipeline parallelism. Amongst 3D space, the Pipeline parallelism [18, 35] features inexpensive point-to-point communication in the cost of pipeline bubbles caused by periodic flushes during training. Data and model parallelism [9, 12, 13, 21, 53, 57], which can be collectively represented by tensor partitioning, are more communication-intensive and favored in devices with relatively high interconnection bandwidth. The performance of tensor partitioning is critical to system performance. With the advancement of interconnection topology and bandwidth, tensor partitioning will have wider application scenarios and play a more important role in system performance.

However, existing tensor partitioning fails in fully utilizing hardware resources. The inadequacy is that current tensor partition primitive only distributes the partitioned sub-operators along spatial dimensions. It is often the case that sub-operators produce partial sum results that necessitate further aggregation from devices. As depicted in Fig.1, when distributing these sub-operators along spatial dimensions (i.e. to different devices), the summation of partial-sum leads to all-reduce communication traffic among mapped devices. On the contrary, distributing these sub-operators along the temporal dimension in a single device can avoid all-reduce since the summation is accomplished locally by accumulating outputs through the passage of time. If tactically handled, temporal partitioning can both mitigate memory/communication overhead and fully utilize the computation resource of multiple devices when combined with spatial partitioning. Our research reveals that adding the temporal dimension to the tensor partition space unlocks numerous possibilities, including avoiding all-reduce, reducing memory footprint and overlapping communication with computation.

Guided by these insights, we propose PrimePar, an automated parallel training framework targeting large transformer models. PrimePar incorporates the extended tensor partition space, resulting in parallel solutions that achieve superior training throughput with lower peak memory occupancy. To the best of our knowledge, this is the first research that introduces the temporal dimension into tensor partitioning for parallel training. Our major contributions can be summarized as:

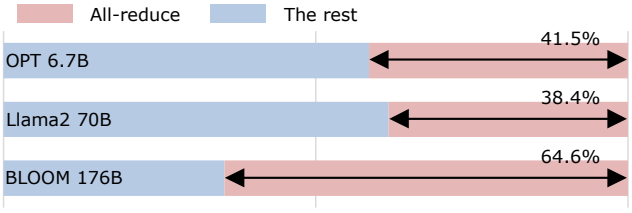
- We expand the tensor partition space for LLM training by introducing a new partition primitive, which leverages the previously ignored temporal dimension to distribute sub-operators in a fashion of more communication and memory efficiency.
- Regarding the lack of scalable tensor partition optimization algorithm for transformer models, we design a dynamic programming-based algorithm capable of identifying the optimal tensor partition strategy from our extended space with solvable complexity.
- We evaluate the effectiveness of PrimePar on training workloads of popular transformer models with a cluster of 32 GPUs. Compared to state-of-the-art distributed training systems Megatron-LM [37, 49] and Alpa [67], we can achieve up to 1.68 × speedup, with only 69% peak memory occupancy. When scaling to 32 GPUs, 1.30 × geo-mean speedup can be achieved. When applied to 3D parallelism, up to 1.46 × speedup over Megatron-LM can be achieved.

2 BACKGROUND AND MOTIVATION

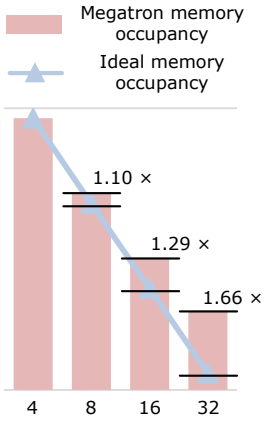
2.1 Tensor Partitioning for Transformers

Tensor partitioning has been extensively explored as a key technique to parallelize neural networks [5, 8, 12, 24, 63] to speedup the computation and amortize the memory overhead. Recursive tensor partitioning has been proposed [15, 20, 21, 33, 51, 52, 58] to formalize a relatively general parallel space and enable automatic search for the best parallel plans.

Tensor partitioning has been applied to accelerate training transformer models. Megatron-LM [37, 49] manually designed efficient tensor partitions for transformer models, where row or column dimensions of linear operators and head dimension of attention matrix multiplications were



(a) proportion of all-reduce latency during training



(b) memory occupation

	Collective	Replication
Proposed	×	×
Partition batch	√	√
Partition row	√	√
Partition column	√	√
Partition sequence	√	√

(c) advantage of proposed partitioning

Figure 2. The proportions of all-reduce latency during training OPT 6.7B, Llama2 70B and Bloom 176B on 16 V100 GPUs using Megatron-LM (model parallelism within a node and data parallelism across nodes) are shown in (a). Training Llama2 70B model with the same batch size on 4, 8, 16, 32 GPUs, the disparity in peak memory occupation per GPU between Megatron-LM and the ideal scenario is shown in (b). The advantage of proposed partition primitive compared with existing ones is shown in (c).

partitioned. Alpa [67] proposed a framework supporting automatic searching and deploying optimal tensor partition solutions in their space, achieving transformer model training performance that is comparable to Megatron-LM.

2.2 Motivation

Unfortunately, the tensor partition space explored so far is incomplete, missing opportunities to better utilize hardware resources. Take partitioning row dimension of linear operator weight as an example, which is one of the strategies proposed in Megatron-LM. Row dimension of weight W is partitioned into n slices, generating n tensor blocks $W_1, \cdots, W_n$. Therefore, the forward computation is divided into n sub-operators $I_1W_1, \cdots, I_nW_n$, where $I_1, \cdots, I_n$ are input tensor blocks partitioned along last dimension in accordance with partitioning W. These n sub-operators are distributed to n different devices with each of them computing a partial result, thus all-reduce among these n devices is necessary for computing the final result $I_1W_1 + \cdots + I_nW_n$. Also, after all-reduce, each device holds a copy of the output tensor, which is a waste of memory. The all-reduce and memory waste are not coincidences. When partitioning a dimension of an operator which is mathematically summed-over, distributing the partitioned sub-operators to different devices will inevitably induce all-reduce. When some tensor has no dimension partitioned, then it is replicated among devices' memory. As shown in Fig.2 (a), all-reduce communication takes up a significant proportion of training latency of Megatron-LM. Fig.2 (b) depicts the disparity in peak memory occupation between Megatron-LM and the ideal scenario, where the ideal scenario assumes no tensor replication among devices. It can be seen that the memory waste caused by tensor replication becomes progressively more severe as the parallelism size increases.

The inefficiencies mentioned above have become increasingly unacceptable considering the demanding requirements placed on computation, communication and memory by large transformer models training workloads. Although these inefficiencies seem unavoidable, in fact, they are addressed if we introduce a new dimension into tensor partitioning, which is the temporal dimension. Inspired by cannon parallel matrix multiplication [3], we consider distributing partitioned sub-operators along both spatial dimensions and temporal dimension (i.e. on the same device but at different time). Specifically, as illustrated in Fig.1, if the results of a group of sub-operators need to be aggregated (or general reduction), distributing them along temporal dimension eliminates the necessity for all-reduce communication. Moreover, distributing sub-operators which have some common tensor along temporal dimension can avoid physically replicating their common tensor in device memory.

Motivated by these considerations, we propose a novel spatial-temporal tensor partition primitive for parallel training. This is a challenging work in that the proposed partition primitive should meet the training requirements, such as tensor alignment between different training phases, while exhibiting better parallel performance. As summarized in Fig.2 (c), our proposed partition primitive avoids incurring the overhead associated with both collective communication and tensor replication, differing from conventional partitioning methodologies employed in transformer models.

3 PRIMEPAR PARTITION SPACE

3.1 Basic Formulation

PrimePar partitions over 2^n homogeneous devices with each device indexed by a ***Device ID*** $\mathbf{D} = (d_1, \cdots, d_n)$, where $d_i = 0, 1$. After tensor partitioning, an operator is partitioned into sub-operators with each sub-operator holding a part of the partitioned tensor. In PrimePar, two sub-operators can be allocated to different devices $\mathbf{D}$, or one single device at different temporal steps t, hence both $\mathbf{D}, t$ are necessary to identify a sub-operator. We introduce the ***Dimension Slice Index*** (DSI), which is a function of $\mathbf{D}, t$ and records which

slice of dimension the sub-operator $(\mathbf{D}, t)$ holds. Specifically, consider the linear operator in transformer models:

$$\begin{aligned}
&\textbf{Forward:} && \overset{[B,M,K]}{O} = \overset{[B,M,N]}{I} \cdot \overset{[N,K]}{W} \\
&\textbf{Backward:} && \overset{[B,M,N]}{dI} = \overset{[B,M,K]}{dO} \cdot \overset{[K,N]}{W^T} \\
&\textbf{Gradient:} && \overset{[N,K]}{dW} = \overset{[B,N,M]}{I^T} \cdot \overset{[B,M,K]}{dO}
\end{aligned} \tag{1}$$

Linear operator has four dimensions B, M, N, K (referring to batch, sequence, input/output hidden dimension), each dimension, say, dimension N maintains three DSIs I_N^F, I_N^B, I_N^G for Forward, Backward and Gradient phases respectively. If $I_N^F(\mathbf{D}, t) = 2$, then at step t of Forward phase, device $\mathbf{D}$ computes a sub-operator whose I, W tensors' dimension N is the 2-th slice among the slices of original dimension N generated by tensor partitioning. Any tensor partition plan in the space of PrimePar can be uniquely represented by specifying DSIs.

In the following of this section, we introduce PrimePar basic tensor partitions and their DSI formulations. Our tensor partition space is represented as the space of sequences of these basic partitions.

3.2 Existing Tensor Partitions

The current paradigm of tensor partition can be summarized as partition by dimension. This involves dividing a single dimension of the operator into two slices, resulting in the partitioning of tensors that include this dimension into two parts, each allocated to separate sub-operators. For tensors not containing this dimension, they are replicated in sub-operators. Recursively partition by dimension forms a large tensor partition space which covers data parallelism and model parallelism that have been applied to train transformer models. For example, in Megatron-LM linear operators are parallelized by recursively partitioning row or column (dimension N, K in Eq.1), while matrix multiplications in the attention block are partitioned by head dimension. PrimePar establishes a comprehensive tensor partition space encompassing conventional partition by dimension, with adaptation to transformer operators.

Algorithm 1 PrimePar Linear Operator DSIs

Input: Sequence of partitions $\mathscr{P}$
Output: DSIs I_X^F, I_X^B, I_X^G, X refers to B, M, N, K
Initialize: $i = 1$, $I_X^F = I_X^B = I_X^G = 0$
for p in $\mathscr{P}$ **do**
  if p is partitioning by dimension X **then**
    $I_X^F \leftarrow 2I_X^F + d_i$, $I_X^B \leftarrow 2I_X^B + d_i$, $I_X^G \leftarrow 2I_X^G + d_i$
    $i \leftarrow i + 1$
  else if p is $P_{2^k \times 2^k}$ **then**
    $r = 2^{k-1}d_i + 2^{k-2}d_{i+2} + \cdots + 2^0 d_{i+2k-2}$
    $c = 2^{k-1}d_{i+1} + 2^{k-2}d_{i+3} + \cdots + 2^0 d_{i+2k-1}$
    temporal index t varying from 0 to $2^k - 1$
    $I_M^F \leftarrow 2^k I_M^F + (r \bmod 2^k)$
    $I_M^B \leftarrow 2^k I_M^B + (r \bmod 2^k)$
    $I_M^G \leftarrow 2^k I_M^G + ((r + t) \bmod 2^k)$
    $I_N^F \leftarrow 2^k I_N^F + ((r + c + t) \bmod 2^k)$
    $I_N^B \leftarrow 2^k I_N^B + ((r + c - 1) \bmod 2^k)$
    $I_N^G \leftarrow 2^k I_N^G + ((r + c - 1 + \delta_{t,2^k-1}) \bmod 2^k)$
    $I_K^F \leftarrow 2^k I_K^F + (c \bmod 2^k)$
    $I_K^B \leftarrow 2^k I_K^B + ((c + t) \bmod 2^k)$
    $I_K^G \leftarrow 2^k I_K^G + ((c - 1 + \delta_{t,2^k-1}) \bmod 2^k)$
    $i \leftarrow i + 2k$
  end if
end for

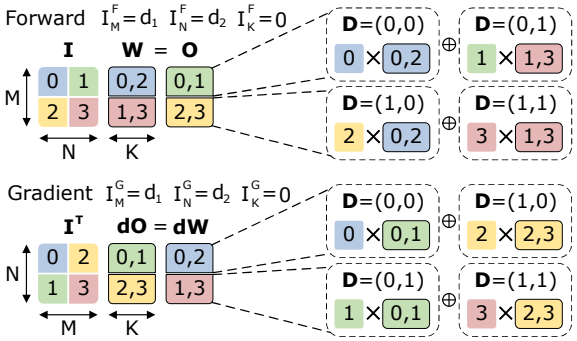


Figure 3. Parallelizing linear operator to 4 devices by partition dimension M and N. The colored blocks represent partitioned tensors, and the number (decimal device id) within indicate which devices are holding the tensor. Blocks with multiple numbers indicate the tensor replication among devices. Dashed boxes on the right show the sub-operators mapped on devices, where $\oplus$ represents all-reduce.

Linear Operator. PrimePar supports partitioning all four dimensions B, M, N, K of linear operator (Eq.1). We exemplify parallelizing linear operator to 4 devices ($\mathbf{D} = (d_1, d_2)$) by partitioning dimension M and N sequentially. Initially all DSIs are 0, indicating that no dimension has been partitioned. The first partition by dimension M can be represented as

$$I_M^F \leftarrow 2I_M^F + d_1, I_M^B \leftarrow 2I_M^B + d_1, I_M^G \leftarrow 2I_M^G + d_1 \tag{2}$$

where dimension M is partitioned into 2 slices and devices with $d_1 = 0$ hold the 0-th slice while others hold the 1-th slice. The second partition

$$I_N^F \leftarrow 2I_N^F + d_2, I_N^B \leftarrow 2I_N^B + d_2, I_N^G \leftarrow 2I_N^G + d_2 \tag{3}$$

partitions dimension N into 2 slices and devices with $d_2 = 0$ and $d_2 = 1$ hold the 0-th slice and 1-th slice respectively. The partitioned tensor distribution and the way they form sub-operators are depicted in Fig.3.

Consider devices whose device ids differ only in d_1 (devices $\mathbf{D}$=(0,0),(1,0) or $\mathbf{D}$=(0,1),(1,1) in Fig. 3), they hold different slices of dimension M while holding the same slice of any

Table 1. Sender device coordinates for receiver device (r, c) for ring communications. Table row distinguishes temporal intervals that have different communication directions. Note that each coordinate entry specifies a ring communication which overlaps with computation step t, and the blank entry means no communication is required.

Forward			
Temporal step	I	W	
$t < 2^k - 1$	$(r, c+1)$	$(r+1, c)$	
Backward			
Temporal step	dO	W	
$t < 2^k - 1$	$(r, c+1)$	$(r-1, c+1)$	
$t = 2^k - 1$		$(r, c+1)$	
Gradient			
Temporal step	I	dO	dW
$t < 2^k - 2$	$(r+1, c-1)$	$(r+1, c)$	
$t = 2^k - 2$	$(r+1, c)$	$(r+1, c+1)$	
$t = 2^k - 1$			$(r, c+1)$

other dimensions. Thus during Gradient phase, they compute the same dW (containing dimension N, K), but each of them obtains partial sum due to not having the complete dimension M. This induces all-reduce communication among them as shown in Fig.3 (Gradient phase). Also, tensors not containing dimension M (W, dW) are replicated among them (blue and red blocks with black borders in Fig.3).

Other Operators in Transformer. PrimePar supports partitioning all dimensions of matrix multiplications in attention, except for head embed dimension. The head embed dimension typically takes values of 64 or 128, partitioning which would significantly reduce the operational intensity and lower the throughput. For softmax operator, we partition all of its dimensions except for the dimension along which softmax is computed (usually the last dimension). For normalization operator, we support partitioning all of its dimensions, with potential all-reduce of expectations and gradient of parameters γ, β. For the remaining element-wise operators, we support partitioning all of their dimensions.

While partitioning by dimension dominates existing tensor partition, it suffers from significant shortcomings of needing collective communication and tensors replication. As transformer models continue to scale up, these shortcomings are becoming increasingly destructive to parallel training.

3.3 Novel Tensor Partition Primitive

We introduce a novel tensor partition primitive that effectively addresses the shortcomings of conventional partition by dimension. When incorporating this novel partition, we can identify parallel strategies that achieves higher throughput and lower peak memory occupancy simultaneously.

Formulation of Partition. We denote the proposed partition as $P_{2^k \times 2^k}$ ($k \geq 1$), which is parallelized across a group of 2^{2k} devices. $P_{2^k \times 2^k}$ logically sees these devices as a $2^k \times 2^k$ square. Each device $\mathbf{D}$ in the group has its relative row and column indices $r(\mathbf{D}), c(\mathbf{D})$ within the logical square.

Unlike partitioning by dimension, $P_{2^k \times 2^k}$ assigns each device with 2^k sub-operators, which are executed sequentially in temporal steps with $0 \leq t < 2^k$ indexing these steps. We design $P_{2^k \times 2^k}$ to satisfy the following features:

1. Collective communication free: Collective communication, which is expensive and hard to overlap with computation, is not required throughout the whole training process.
2. Memory efficient: No replication of tensors among the memory of devices.
3. Support training: Forward, Backward and Gradient phases can be executed periodically without additional operations between them.

Feature 1 and 2 endow $P_{2^k \times 2^k}$ with superior performance and memory efficiency compared to partitioning by dimension. Feature 3 requires two kinds of tensor distribution alignments. For the tensor stashed in memory in an earlier phase which will be used in some latter phase, its distribution at the last step of the earlier phase must align with that at the first step of the latter phase. Also, the weight distribution at the first step of Forward phase must align with that at the last step of Gradient phase. If feature 3 is not satisfied, frequent redistribution of weight, gradient and other intermediate tensors is required between training phases.

We now specify the details of $P_{2^k \times 2^k}$. For device (r, c) at temporal step t, its DSIs are:

$$\textbf{Forward:} \begin{cases} I_M = r \bmod 2^k \\ I_N = (r + c + t) \bmod 2^k \\ I_K = c \bmod 2^k \end{cases} \tag{4}$$

$$\textbf{Backward:} \begin{cases} I_M = r \bmod 2^k \\ I_N = (r + c - 1) \bmod 2^k \\ I_K = (c + t) \bmod 2^k \end{cases} \tag{5}$$

$$\textbf{Gradient:} \begin{cases} I_M = (r + t) \bmod 2^k \\ I_N = (r + c - 1 + \delta_{t,2^k-1}) \bmod 2^k \\ I_K = (c - 1 + \delta_{t,2^k-1}) \bmod 2^k \end{cases} \tag{6}$$

where $\delta_{t,2^k-1}$ takes value 1 if $t = 2^k - 1$ otherwise 0. It can be verified that features 1,2,3 are satisfied.

Feature 1: During Forward phase, the summed-over dimension of matrix multiplication is dimension N. In Eq.4, for an arbitrary device (r, c), I_N takes all 2^k values as t varies from 0 to $2^k - 1$ while I_M, I_K are fixed to r, c. Thus after 2^k steps of computation, device (r, c) computes the (r, c)-th block of output tensor, with no device yielding partial sum because the partial sums of all slices of dimension N are summed locally in device (r, c) during these 2^k steps (see Fig.4). The

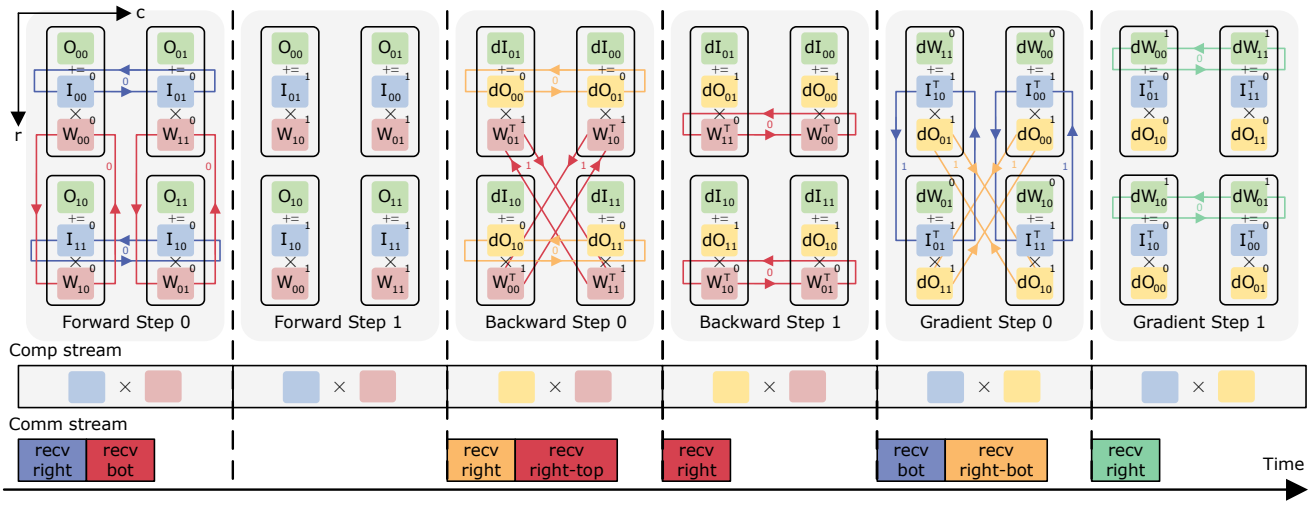


Figure 4. The process of training with partition $P_{2\times2}$. Details of orchestration among 2×2 devices are shown in the upper half, while the corresponding execution timeline is shown in the lower. The 0-1 numbers to the upper-right of tensor block or within communication ring indicate the tensor or communication source is in which double buffer.

same conclusion can be verified for Backward and Gradient phases similarly.

Feature 2: Consider arbitrary two devices (r, c) and (r', c') in Gradient phase (Eq.6), suppose their corresponding I_N and I_K are the same at temporal step t

$$(r + c - 1 + \delta_{t,2^k-1}) \equiv (r' + c' - 1 + \delta_{t,2^k-1}) \bmod 2^k$$
$$(c - 1 + \delta_{t,2^k-1}) \equiv (c' - 1 + \delta_{t,2^k-1}) \bmod 2^k$$

The latter equation yields $c = c'$ which further yields $r = r'$ in the former. Thus at any step t during Gradient phase, no two devices have the same I_N and I_K simultaneously, which means W, dW are never replicated among devices. Similar reasoning can be done for all tensors in all phases.

Feature 3: Let $t = 2^k - 1$ in Eq.4 and $t = 0$ in Eq.6, where we can verify that I_M, I_N at the last step of Forward match with that at the first step of Gradient. This means the distribution of tensor I does not change when transition from Forward to Gradient. Thus each device can stash its I at the end of Forward and directly use the stashed I for computation when entering Gradient phase (see the blue blocks in Forward step 1 and Gradient step 0 in Fig.4). Other alignments can be verified similarly.

Formulation of Communication. Due to the variation of DSIs with t, there exists tensor communication between temporal steps. In Eq.4, I_N changes with t. Since both tensors I and W contain dimension N, the communication for I, W is necessary. Similarly from Eq.5, 6 it can be inferred that in Backward phase, communication is required for tensors dO, W, while in Gradient phase, communication is necessary for tensors I, dO, dW.

However, unlike all-reduce communication, these communications are not data-dependent on the computation result and can be carried out simultaneously with computation. This plays a crucial role in effectively harnessing both computational power and communication bandwidth.

Specifically, in Forward phase, tensors I, W require communication between steps. Leveraging double buffer, each device can compute with I, W of the current step while receiving I, W for the next step and storing them into double buffer. The communications of dO, W in Backward phase and I, dO in Gradient phase are similar. As for dW in Gradient phase, it is the result tensor accumulating the computation result of each step. Since I_N, I_K only change from step $t = 2^k - 2$ to $t = 2^k - 1$ when $\delta_{t,2^k-1}$ turns from 0 to 1, dW accumulated in previous steps should be redistributed during the last step of computation. Afterwards, each device adds the redistributed dW with its last step computation result to yield the final dW result. This dW redistribution is necessary for weight alignment between start of Forward and end of Gradient.

Here we explicitly derive the communication pattern of W in Backward phase. In Eq.5, I_N, I_K are the DSIs of W. At step $t + 1$, device (r, c) holds this part of W:

$$I_N = (r + c - 1) \bmod 2^k$$
$$I_K = (c + (t + 1)) \bmod 2^k$$

To identify which device was holding the part of W specified by the above I_N, I_K at step t, rewrite the above equations as:

$$I_N = ((r - 1) + (c + 1) - 1) \bmod 2^k$$
$$I_K = ((c + 1) + t) \bmod 2^k$$

which indicates that device $(r-1, c+1)$ was holding this part of W. Note here $r-1 = 2^k-1$ if $r = 0$ and $c+1 = 0$ if $c = 2^k-1$. Therefore, we can conclude that for $t < 2^k - 1$ in Backward phase, each device receives W from its right-top neighbor,

establishing a ring point-to-point communication pattern, where each device only communicates with its neighbors in the ring (Fig.4 Backward step 0). During the last step of Backward, W should be redistributed to align with the W distribution at the start of Forward. At the start of Forward, device (r, c) holds:

$$I_N = (r + c) \bmod 2^k$$
$$I_K = c \bmod 2^k$$

which can be rewritten into:

$$I_N = (r + (c + 1) - 1) \bmod 2^k$$
$$I_K = ((c + 1) + (2^k - 1)) \bmod 2^k$$

comparing these equations with Eq.5 shows that during the last step of Backward, device $(r, c + 1)$ holds the part of W which device (r, c) holds at the start of Forward. Thus during the last step of Backward, each device should receive W from its right neighbor to ensure the W distribution alignment (Fig.4 Backward step 1).

Similar reasoning can derive all communication patterns listed in Table.1, all of which are ring communication and can be effectively overlapped with computation.

4 COST MODEL

4.1 Intra-operator Cost

The cost of executing a parallelized operator based on a tensor partition strategy is the intra-operator cost, which is dependent on both the operator configuration and the tensor partition strategy. We model two aspects of intra-operator cost, training latency and peak memory occupancy. The intra-operator cost is a weighted sum of these two metrics, enabling joint optimization of performance and memory occupation. We introduce the intra-operator cost for an operator n that is partitioned with partition sequence $\mathscr{P}$ (refer to Alg.1).

Intra-operator Communication. PrimePar has two kinds of intra-operator communication: all-reduce communication caused by conventional partition by dimension and ring communication caused by $P_{2^k \times 2^k}$. Both all-reduce and ring communication happens in groups.

The grouping pattern for all-reduce is determined by the positions of partitioning the summed-over dimension in $\mathscr{P}$. Consider parallelizing linear operator to 8 devices ($\mathbf{D} = (d_1, d_2, d_3)$), we determine the grouping pattern of all-reduce in Forward phase. Suppose I_N^F changes with (d_1, d_3), while the rest of device id (d_2) does not affect the value of I_N^F. Two devices within one group must have the same (d_2) and different (d_1, d_3). The same (d_2) ensures that they compute the same part of result tensor, while different (d_1, d_3) ensures that they hold different slices of dimension N to compute different partial sums. We call (d_1, d_3) as the group indicator. In this way, all devices are divided into several disjoint groups, whose union is the complete set of devices.

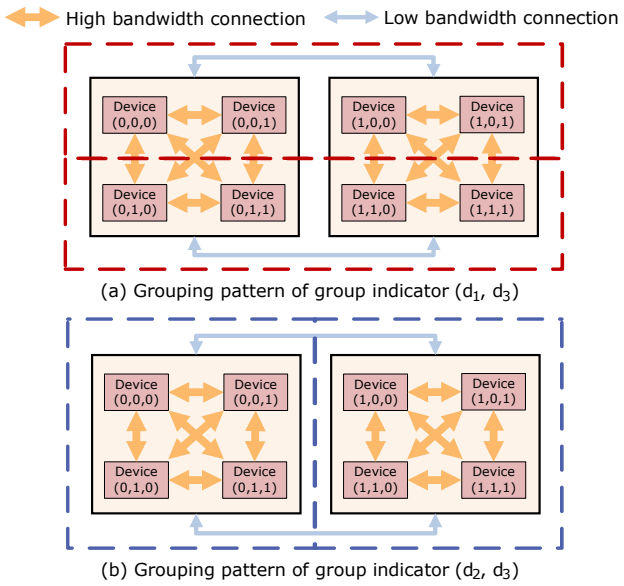


Figure 5. Illustration of grouping patterns of group indicators (a) (d_1, d_3) and (b) (d_2, d_3), where devices within the same dashed box form a group. When executing all-reduce or ring communication, (a) exhibits longer latency than (b) because low bandwidth interconnections are contained in each group of (a).

All-reduce happens within each group, hence the overall all-reduce latency is determined by the slowest among these groups, which can be modeled as a linear function of the tensor size being all-reduced. Fig.5 visualizes the grouping patterns when the group indicators are (d_1, d_3) and (d_2, d_3). For each group pattern, we obtain its linear function coefficients by profiling real system latency with different all-reduce tensor sizes and apply linear regression. Note this profiling is scalable for large systems since the number of group indicators is less than the number of devices (group indicator is a sub-sequence of device id $\mathbf{D}$). The modeling of ring communication latency is similar. In this manner, we establish functions for computing the total all-reduce latency and ring communication latency at step t, denoting them as $allreduce(n, \mathscr{P})$, $ring(n, \mathscr{P}, t)$ for convenience.

Computation. We model the computation latency of operators as linear functions of the amount of floating point operations and memory access. We also obtain the linear coefficients through real system latency profiling and linear regression. The coefficients are profiled separately for different types of operators. We denote the computation latency at step t as $compute(n, \mathscr{P}, t)$.

Peak Memory Occupancy. For each operator, its peak memory occupancy during training is modeled as the total size of its parameter tensors and tensors stashed in Forward phase for use in Backward and Gradient phases. Denote the peak memory occupancy as $memory(n, \mathscr{P})$.

The overall intra-operator cost can be modeled as:

$$intraC(n, \mathscr{P}) = \sum_t max(compute(n, \mathscr{P}, t), ring(n, \mathscr{P}, t)) + allreduce(n, \mathscr{P}) + \alpha \cdot memory(n, \mathscr{P}) \tag{7}$$

where α is the adjustment coefficient between latency and memory metrics.

4.2 Inter-operator Cost

Consider two consecutive operators n_1, n_2 partitioned with $\mathscr{P}_1, \mathscr{P}_2$, where the output of n_1 serves as the input of n_2. On a given device **D**, the locally held output tensor after execution of n_1 may not cover the input tensor of n_2. In such cases, it becomes necessary to redistribute the missing part of tensor from other devices [51, 67].

For each dimension X of the tensor passing from n_1 to n_2, there exist two DSIs, I_X^1, I_X^2, associated with n_1 and n_2 respectively. Note that I_X^1, I_X^2 are determined by $\mathscr{P}_1, \mathscr{P}_2$ as shown in Alg.1. Let s_X^1, s_X^2 be the lengths of each segment of dimension X of n_1, n_2 after partitioning. We can determine the part of dimension X held by each device **D** at the last and first temporal steps of n_1 and n_2 respectively:

$$\begin{aligned} S_X^1(\mathbf{D}) &= [s_X^1 I_X^1(\mathbf{D}, t=-1) : s_X^1 I_X^1(\mathbf{D}, t=-1) + s_X^1] \\ S_X^2(\mathbf{D}) &= [s_X^2 I_X^2(\mathbf{D}, t=0) : s_X^2 I_X^2(\mathbf{D}, t=0) + s_X^2] \end{aligned} \tag{8}$$

By evaluating DSIs for all dimensions of the tensor, the intersection lengths in each dimension can be computed. The product of these intersection lengths is the size of the part of n_1 output tensor computed by device **D**, which also serves as part of device **D**'s input tensor for computing n_2. Consequently, the total redistribution communication traffic of all devices when running from n_1 to n_2 during forward computation is:

$$\sum_{\mathbf{D}} (V - \prod_X |S_X^1(\mathbf{D}) \cap S_X^2(\mathbf{D})|) \tag{9}$$

where V is the partitioned size of n_2 input tensor on each device and || represent taking the length of dimension slice. The redistribution traffic from n_2 to n_1 during backward computation can be computed similarly. We model the inter-operator communication latency between n_1 and n_2 as a linear function of the sum of forward and backward redistribution communication traffic. The linear coefficients are obtained by profiling real system latency. Following this method, we establish the inter-operator cost function, denoted as $interC(n_1, n_2, \mathscr{P}_1, \mathscr{P}_2)$.

4.3 Overall Cost

Consider the computation graph $G =< N, E >$ of a model. Based on the intra-operator and inter-operator cost formulations, the overall cost of the whole model with each node n_i specified with partition strategy $\mathscr{P}_i$ is:

$$C = \sum_{n_i \in N} intraC(n_i, \mathscr{P}_i) + \sum_{(n_i, n_j) \in E} interC(n_i, n_j, \mathscr{P}_i, \mathscr{P}_j) \tag{10}$$

The optimization goal is to find the $\mathscr{P}_i$ for each n_i such that they collectively minimize the overall cost C.

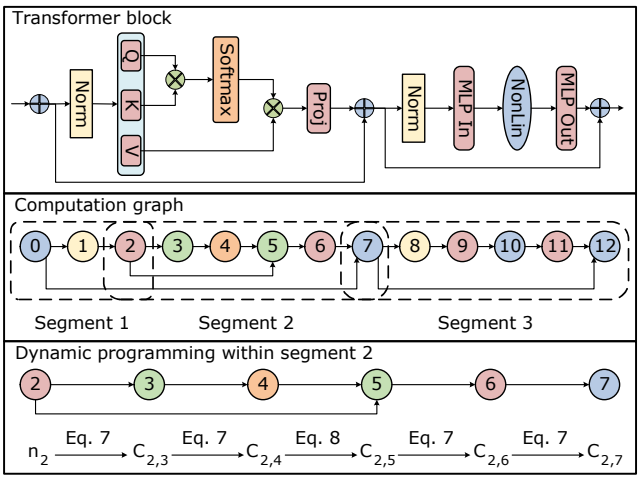


Figure 6. Architecture and computation graph of a transformer block in popular LLMs. The dynamic programming process within one of the segment is illustrated.

5 OPTIMIZATION ALGORITHM

PrimePar establishes a comprehensive tensor partition space, which contains better solutions to release more system performance. However it is not trivial to find optimal partition strategy, given that our partition space is extended by adding temporal dimension.

To optimize the tensor partition strategy for a model, it is necessary to find the optimal strategy for each operator in the model's computation graph such that they collectively achieve the minimal cost. This forms a search space which expands exponentially as the number of operators increases, with the base being the size of operator partition space.

Previous optimization algorithm applicable to transformer models formalized the problem as an integer linear programming (ILP) [67]. However, due to the fact that ILP is NP-hard, this optimization algorithm scales poorly. In this section, we propose an optimization algorithm to efficiently search for the optimal tensor partition strategy for transformer models, which minimize the sum of intra-operator (node) and inter-operator (edge) costs according to the cost model established in previous section.

5.1 Segmented Dynamic Programming

Dynamic programming that defines the sub-problem as finding optimal solution for sub-models can address the exponential complexity with respect to the number of operators.

However, due to the non-linear model structure of transformer models, it is challenging to formalize tractable Bellman equations. Thus we propose ***segmented dynamic programming*** that divides model into segments and carry out dynamic programming within each segment.

Consider two nodes n_i, n_j in the computation graph ($i < j$), we define the sub-model $Model_{i,j}$ consisting of nodes $n_i, n_{i+1}, ...n_j$ (listed in increasing topological order of computation graph) and edges whose source and destination nodes lie within $n_i, n_{i+1}, ...n_j$. Define the optimal cost of sub-model $Model_{i,j}$ when n_i, n_j are under partition states p_i, p_j as $C_{i,j}(p_i, p_j)$, which is the optimal sub-structure in our dynamic programming formulation.

Assumption 1. For all edges that connects n_j and belongs to $Model_{i,j+1}$, the only one that is not contained in $Model_{i,j}$ is $e_{j,j+1}$.

Assumption 2. Among all nodes in $Model_{i,j}$ only n_i and n_j can connect with n_{j+1}.

Under these two assumptions we can derive the Bellman equation between optimal cost of $Model_{i,j}$ and $Model_{i,j+1}$

$$C_{i,j+1}(p_i, p_{j+1}) = \min_{p_j}\{C_{i,j}(p_i, p_j) + n_{j+1}(p_{j+1}) + e_{j,j+1}(p_j, p_{j+1})\} \tag{11}$$

or

$$\min_{p_j}\begin{Bmatrix} C_{i,j}(p_i, p_j) + n_{j+1}(p_{j+1}) + e_{j,j+1}(p_j, p_{j+1}) \\ + e_{i,j+1}(p_i, p_{j+1}) \end{Bmatrix} \tag{12}$$

where $n_{j+1}(p_{j+1})$ represents the intra-operator cost of node n_{j+1} under partition state p_{j+1} and $e_{j,j+1}(p_j, p_{j+1})$ represents the inter-operator cost between n_j and n_{j+1} when they are under p_j and p_{j+1} respectively. Eq.12 is for the case when both n_i, n_j connect with n_{j+1}. It is desired that the dynamic programming for the whole model should be accomplished with only these two Bellman equations.

However, it is not possible to straightforwardly iterate from the first operator of transformer to the last with only Eq.11, 12. As illustrated in Fig.6, when iterating from $Model_{0,4}$ to $Model_{0,5}$, n_2 in $Model_{0,4}$ also connects with n_5, thus Assumption 2 is violated. Similar violation also happens when iterating from $Model_{0,11}$ to $Model_{0,12}$.

With segmented dynamic programming, we can avoid these violations. Since n_0, n_2, n_7 have extended edge whose destination is not the subsequent node, not regarding them as starting node of iteration will inevitably violate Assumption 2 when iterating to the destination node of their extended edge. Thus we apply dynamic programming within segments $Model_{0,2}$, $Model_{2,7}$ and $Model_{7,12}$ separately, where Assumptions 1,2 are never violated. In this way, optimal sub-structures $C_{0,2}, C_{2,7}, C_{7,12}$ can be computed with Eq.11, 12.

With $C_{0,2}, C_{2,7}, C_{7,12}$ computed, optimal sub-structure for larger sub-models can be obtained by ***merging***. Fig.6 shows that $e_{0,7}$ is not contained in $Model_{0,2}$ nor $Model_{2,7}$, whose cost needs to be added when merging $C_{0,2}, C_{2,7}$. Also, $Model_{0,2}$ and $Model_{2,7}$ have a common node n_2 whose cost needs to be subtracted.

$$C_{0,7}(p_0, p_7) = \min_{p_2}\begin{Bmatrix} C_{0,2}(p_0, p_2) + C_{2,7}(p_2, p_7) + \\ e_{0,7}(p_0, p_7) - n_2(p_2) \end{Bmatrix} \tag{13}$$

Furthermore, the optimal sub-structure of a whole Transformer layer can be obtained by merging $C_{0,7}, C_{7,12}$.

$$C_{0,12}(p_0, p_{12}) = \min_{p_7}\{C_{0,7}(p_0, p_7) + C_{7,12}(p_7, p_{12}) - n_7(p_7)\} \tag{14}$$

Due to the stacking of identical transformer layers in transformer models, it is sufficient to compute $C_{0,12}$ once. In Fig.6, n_0, n_{12} are the last operator of previous layer and current layer respectively, meaning that optimal sub-structures of neighboring layers actually has one node in common and can be merged like Eq.14. With $log(\#layers)$ steps of recursive merging, we can compute optimal sub-structure for 2, 4, 8 layers and so on.

5.2 Optimality Proof

5.2.1 Correctness of Bellman equation.

We prove the correctness of Eq.12 by induction where the induction hypothesis is that $C_{i,j}(p_i, p_j)$ is the minimal cost for $Model_{i,j}$. The proof for Eq.11 is similar and omitted. Based on assumptions 1,2 it can be inferred that $Model_{i,j+1}$ differs from $Model_{i,j}$ by the addition of a single node n_{j+1} and two edges $e_{j,j+1}, e_{i,j+1}$. For simplicity, we will abbreviate n for n_{j+1}, e for $e_{j,j+1}$ and e' for $e_{i,j+1}$ in the proof. Suppose there exists some $C^*_{i,j+1}(p_i, p_{j+1})$ for $Model_{i,j+1}$ which is smaller than $C_{i,j+1}(p_i, p_{j+1})$ and the partition state of n_j in $C^*_{i,j+1}(p_i, p_{j+1})$ is p^*_j. It is equivalent to say

$$C^*_{i,j}(p_i, p^*_j) + n(p_{j+1}) + e(p^*_j, p_{j+1}) + e'(p_i, p_{j+1}) < \min_{p_j}\{C_{i,j}(p_i, p_j) + n(p_{j+1}) + e(p_j, p_{j+1}) + e'(p_i, p_{j+1})\}$$

Since the R.H.S takes the minimal for all possible p_j, it is obviously no greater than fixing p_j to p^*_j, thus

$$C^*_{i,j}(p_i, p^*_j) + n(p_{j+1}) + e(p^*_j, p_{j+1}) + e'(p_i, p_{j+1}) < C_{i,j}(p_i, p^*_j) + n(p_{j+1}) + e(p^*_j, p_{j+1}) + e'(p_i, p_{j+1})$$

which yields $C^*_{i,j}(p_i, p^*_j) < C_{i,j}(p_i, p^*_j)$ contradicting the induction hypothesis.

5.2.2 Correctness of Merging.

We also only prove the correctness of Eq.13 because the proof for Eq.14 is similar. Suppose there exists some $C^*_{0,7}$ which is better than $C_{0,7}$ and the partition state of n_2 in $C^*_{0,7}$ is p^*_2. It is equivalent to say

$$C^*_{0,2}(p_0, p^*_2) + C^*_{2,7}(p^*_2, p_7) + e_{0,7}(p_0, p_7) - n_2(p^*_2) < \min_{p_2}\{C_{0,2}(p_0, p_2) + C_{2,7}(p_2, p_7) + e_{0,7}(p_0, p_7) - n_2(p_2)\}$$

Similar to section 5.2.1, fixing p_2 to p^*_2 in R.H.S

$$C^*_{0,2}(p_0, p^*_2) + C^*_{2,7}(p^*_2, p_7) + e_{0,7}(p_0, p_7) - n_2(p^*_2) < C_{0,2}(p_0, p^*_2) + C_{2,7}(p^*_2, p_7) + e_{0,7}(p_0, p_7) - n_2(p^*_2)$$

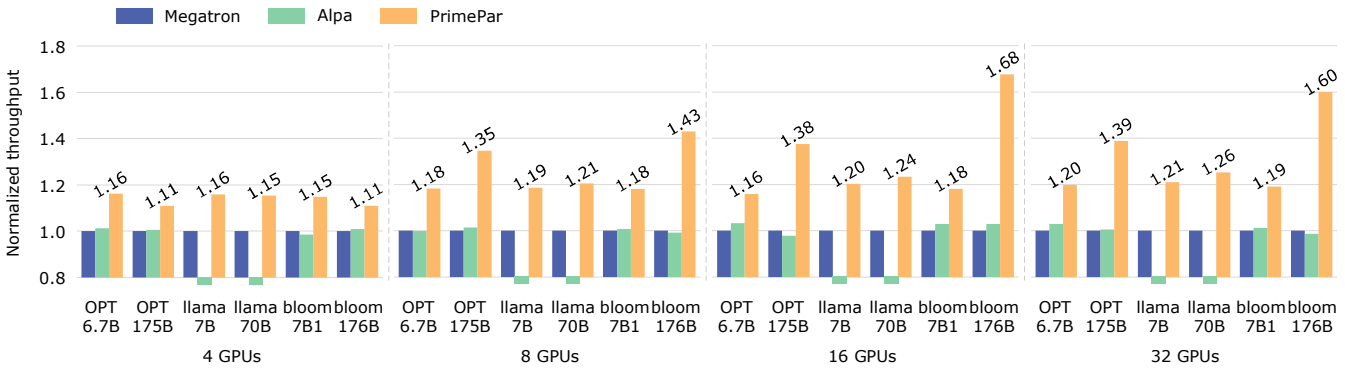


Figure 7. Normalized training throughputs of Megatron, Alpa and PrimePar. (The downward pillar indicates that the model structure is not supported by the current version of the framework.)

Thus $C^*_{0,2}(p_0, p_2^*) + C^*_{2,7}(p_2^*, p_7) < C_{0,2}(p_0, p_2^*) + C_{2,7}(p_2^*, p_7)$ contradicting the optimality of $C_{0,2}, C_{2,7}$, which is proved in section 5.2.1.

5.3 Generality and Complexity

This optimization algorithm is applicable to most transformer models, with only the segmentation different from model to model. The complexity is $O(P^3)$, where P is the size of operator partition space. When optimizing for system with 32 devices, modern CPU can solve it in seconds. Moreover, the main computation comes from Eq.11-14, which are parallelizable and can be implemented as CUDA kernels to enable fast optimization for potentially larger parallel systems.

6 EVALUATION

Implementation. We implement PrimePar with PyTorch [41]. Specified with searched optimal tensor partition strategy or any other strategies, it can automatically deploy models to multiple GPUs. We utilize PyTorch distributed communication package with NCCL [38] backend to implement collective and inter-operator communications. For proposed novel tensor partition primitive, we implement in C++ code utilizing CUDA and CUDA-aware MPI APIs [39, 40], which can be invoked from PyTorch code as a C++ extension.

Environment and models. We evaluate PrimePar on a cluster of 8 nodes where each node consists of 4 NVIDIA V100-SXM2 32GB GPUs and Intel Xeon Gold 5218 32-core CPU. The GPUs within each node is connected via 300 GB/s NVLink while nodes are connected via 100 GB/s InfiniBand. We evaluate the training workloads of 6 sets of popular transformer-based language models, including OPT 6.7B, 175B, Llama2 7B, 70B and BLOOM 7B1, 176B [55, 60, 66].

Baseline and metrics. We choose Megatron-LM [37, 49] as the baseline. We also compare with previous state-of-the-art automated parallel training framework Alpa [67]. Since PrimePar parallelism rigorously preserves the mathematical semantics of original training, we focus on evaluating the training throughput and memory occupancy.

6.1 Performance Analysis

To assess the efficacy of PrimePar tensor partitioning, we evaluate the training throughput of transformer models scaling to 4, 8, 16, 32 GPUs by different tensor partition strategies. Here, we refrain from utilizing pipeline parallelism to control variables. When evaluating Megatron-LM on n GPUs, we enumerate all possible data parallelism (partition batch dimension) size d and employ Megatron-LM's model parallelism (partition head/row/column dimensions) with size $\frac{n}{d}$. We select the configuration that exhibits the best performance of Megatron-LM to compare with PrimePar. For Alpa, we follow a similar process, where the partition strategy for non-batch dimensions are determined by the model parallelism strategy searched by Alpa on our machine. Please note that PrimePar incorporates partitioning all dimensions to its search space, including batch dimension, thus the enumeration of data parallelism size is unnecessary for PrimePar.

Fig.7 illustrates the training throughput of 6 models under 4 parallelism scales. In all testcases, PrimePar achieves better throughput than Megatron-LM and Alpa. Note that Megatron-LM and Alpa demonstrate close performance as they are both state-of-the-art within conventional tensor partition space. PrimePar achieves 1.16 - 1.20 × throughput over Megatron-LM in models with parameter scales around 7B. For large models exceeding 100B, 1.11 - 1.68 × throughput can be achieved. When scaling to 32 GPUs, the geo-mean speedup across benchmarks is 1.30 ×. It can be observed that the speedup increases as the number of GPUs grow, and large models get significant promotion when scaling to 16, 32 GPUs.

6.2 Peak Memomry Occupation Analysis

We evaluate the effect of memory saving of PrimePar tensor partitioning. Under the same tensor partition configurations that achieve the throughputs illustrated in Fig.7, we profile the peak memory occupation during the entire training. Due to the Single Program Multiple Data (SPMD) nature of

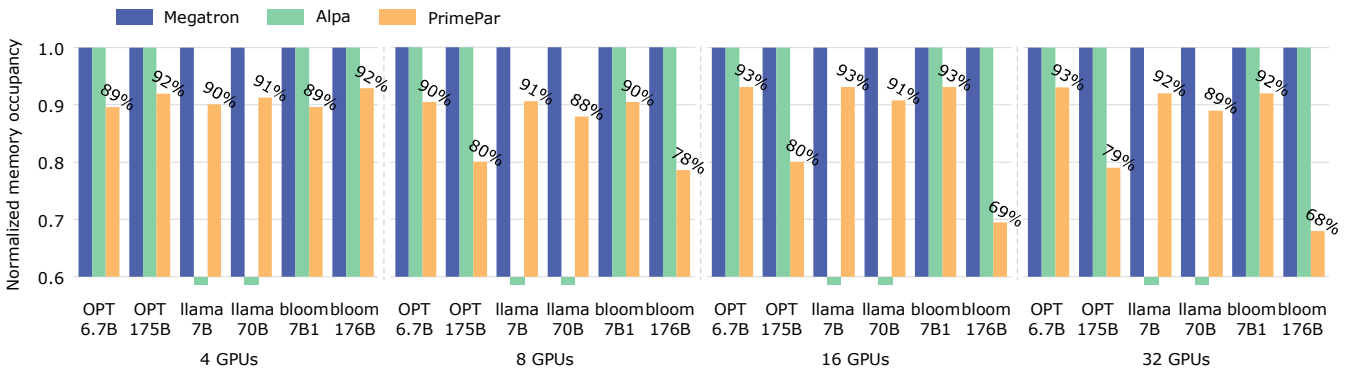


Figure 8. Normalized peak memory occupancy during training of Megatron, Alpa and PrimePar.

Megatron-LM, Alpa and PrimePar, the memory occupation is consistent across different GPUs, thus it is sufficient to profile the memory of one GPU for evaluation.

As shown in Fig.8, PrimePar presents lower peak memory occupation in all testcases. PrimePar consumes around 90% of the memory footprint compared to Megatron-LM when training models with a scale around 7B. It is worth highlighting that PrimePar demonstrates notable memory saving for large models. Specifically when training BLOOM 176B with parallelism size of 16 or 32, PrimePar consumes only 68% of the memory footprint compared to Megatron-LM. This memory saving originates from reduced tensor replication when applying the novel tensor partition.

6.3 Ablation Study

We explain the source of effectiveness of PrimePar by comparing details of latency breakdown and partition strategies with Megatron-LM. As shown in the left part of Fig.9, the major speedup originates from the reduction in collective communication latency, where PrimePar consumes 19.9% - 62.2% the collective communication latency compared to Megatron-LM. Also, the ring point-to-point communication originating from the novel partition introduced in PrimePar has short latency and can be fully overlapped with computation. Note that Megatron-LM and PrimePar share roughly the same computation latency, which means PrimePar does not trade computation efficiency for communication efficiency. The right part of Fig.9 shows the partition strategies and corresponding kernel execution timelines for OPT 175B MLP block when parallelizing to 8 GPUs. Note that we only demonstrate the timeline of one device and this shows the system latency due to the SPMD nature.

We elucidate the correspondence between partition strategies and collective communication kernels. Here we number the 8 GPUs from 0 to 7 ($\mathbf{D} = (d_1, d_2, d_3)$), where GPUs 0 to 3 constitute one node and GPUs 4 to 7 constitute another node. For example kernel ① of Megatron-LM arises from the innermost two partitions of fc2.$\mathscr{P}$, where dimension N is partitioned twice and distributed with group indicator (d_2, d_3). This leads to all-reduce of output tensor O within each group of $(0, 1, 2, 3)$ and $(4, 5, 6, 7)$, which means intra-node all-reduce. Since fc2.$\mathscr{P}$ contains a partition along dimension B, the size of tensor O for all-reduce is half of the original size.

Kernel ① of PrimePar arises from the outermost partition of fc2.$\mathscr{P}$ where dimension N is partitioned once and distributed with group indicator (d_1). This induces all-reduce of output tensor O within each group of $(0, 4)$, $(1, 5)$, $(2, 6)$, $(3, 7)$ via inter-node communication. Since fc2.$\mathscr{P}$ contains a $P_{2\times2}$, the size of tensor O for all-reduce is 1/4 of the original size. As for $P_{2\times2}$, it induces ring point-to-point communications which happen in groups with group indicator (d_2, d_3). Thus ring communications happen within groups of $(0, 1, 2, 3)$ and $(4, 5, 6, 7)$. It can be seen that these ring communications have small latency and can be fully overlapped with computation kernels. In this example, PrimePar demonstrates minimal communication overhead within each group of $(0, 1, 2, 3)$ and $(4, 5, 6, 7)$ attributed to the utilization of $P_{2\times2}$. When scaling to 8 GPUs, PrimePar entails mild collective communication involving only 1/4 the size of input and output tensor of the MLP block due to the fact that $P_{2\times2}$ partitions both input and output tensors.

The primary source of speedup of PrimePar is the introduction of novel partition and its appropriate position in the partition sequence searched by our optimization algorithm. PrimePar strategies successfully trade expensive collective communication for point-to-point communication that is relatively cheap and convenient to be overlapped with computation.

6.4 Impact on 3D Parallelism

Since 3D parallelism is the prevailing training technique for large transformer models, it is important to know the improvement that PrimePar brings to 3D parallelism. We use the notation (p, d, m) to represent the parallelism sizes of pipeline, data and model parallelism respectively. We fit with pipeline parallelism in the same way as introduced in Megatron-LM. For example when $(p, d, m) = (4, 1, 8)$, 32

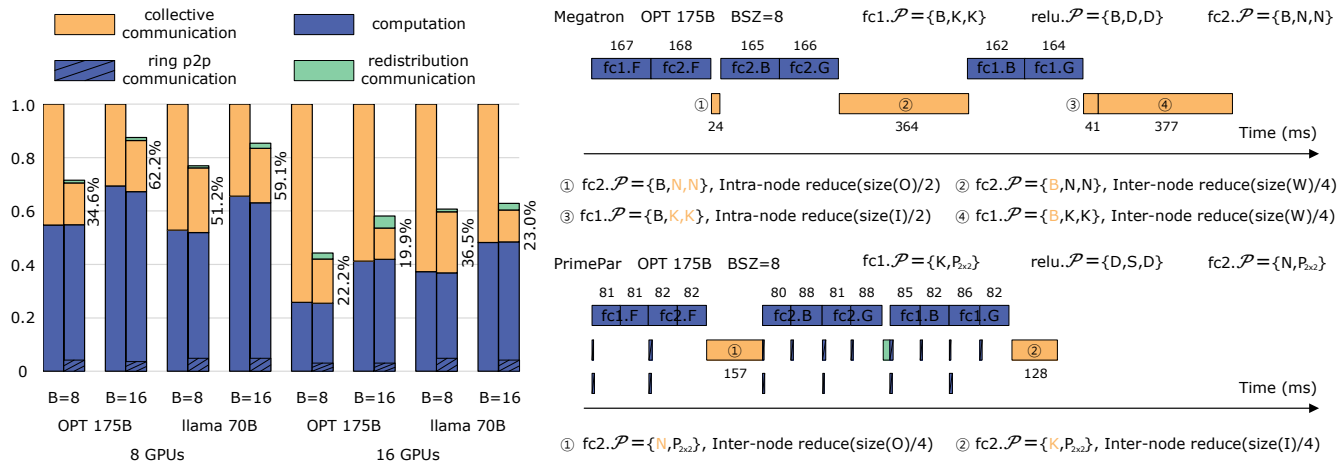


Figure 9. Left: normalized latency breakdown of MLP blocks with configurations of batch size 8, 16 and scaling to 8, 16 GPUs. Each pair of pillars represent Megatron-LM and PrimePar respectively, where collective communication latency reductions are labeled. Right: detailed partition sequence $\mathscr{P}$ of operators in OPT 175B MLP block of Megatron-LM and PrimePar and corresponding kernel execution timelines. Capital letters in $\mathscr{P}$ represent dimensions being partitioned, where notions for fc1 and fc2 are the same as Eq.1 and B, S, D represent batch, sequence, hidden dimensions for relu operator. We use F, B, G to label the forward, backward, gradient computation phases. The correspondence between collective communication kernels and partitions are listed.

devices are first separated into 4 groups, with each group consisting 8 devices computing a pipeline stage. 8-way model parallelism is further implemented within each group. We evaluate training each model with 32 GPUs using 3D parallelism of all possible (p, d, m) configurations ($p > 1$). Under each (p, d, m) configuration, we evaluate the training throughputs of Megatron-LM and PrimePar by applying their model parallel strategies with size m respectively, where the pipeline and data parallelism configurations are controlled to be consistent. In order to control d, we disable partitioning batch dimension in PrimePar and search for optimal partitioning non-batch dimensions strategy to generate model parallelism with size m.

Fig.10 demonstrates that 3D parallelism incorporating PrimePar consistently achieves superior throughput across different models and (p, d, m) configurations. For models around 7B, both Megatron-LM and PrimePar exhibits highest throughput under the configuration $(p = 2, d = 4, m = 4)$, where PrimePar slightly outperforms Megatron-LM. In the case of OPT 175B, Llama2 70B and Bloom 176B, PrimePar performs significantly better than Megatron-LM, achieving 1.46, 1.27 and 1.40 × highest throughput than Megatron-LM achieves. Different from models around 7B, large models exceeding 100B achieve peak performance under the configuration $(p = 2, d = 1, m = 16)$, where model parallelism is preferred over data parallelism. This is attributed to the expensive all-reduce communication induced by data parallelism when the weight scale grows large.

Table 2. Optimization time in milliseconds for OPT, Llama2 and Bloom model structures for parallelism sizes 4, 8, 16, 32.

Models	Parallelism size			
	4	8	16	32
OPT	85.3	86.5	170.9	5,357.3
Llama2	86.5	88.8	185.9	6,070.3
Bloom	85.4	80.2	165.8	4,153.0

6.5 Optimization Time

We report the time consumption of the proposed segmented dynamic programming algorithm. Our optimization algorithm runs in a single thread of Intel Xeon Gold 5218 (2.3GHz) CPU. As Table.2 shows, searching the optimal solution for OPT, Llama2 and Bloom models only consumes seconds even if the parallelism size scales to 32.

7 DISCUSSION

PrimePar's design supports modeling and optimizing for general computational environments with various interconnection topologies. Since PrimePar's novel partition primitive only induces ring communication, PrimePar is expected to achieve more efficient scaling in interconnection topologies with torus. For example, TPU v4 [22] chips are connected with twistable tori. It is expected that this interconnection is suitable for PrimePar's novel partition because the tori can be configured to be horizontal, vertical and parallel to diagonal to cater to PrimePar's ring communications such

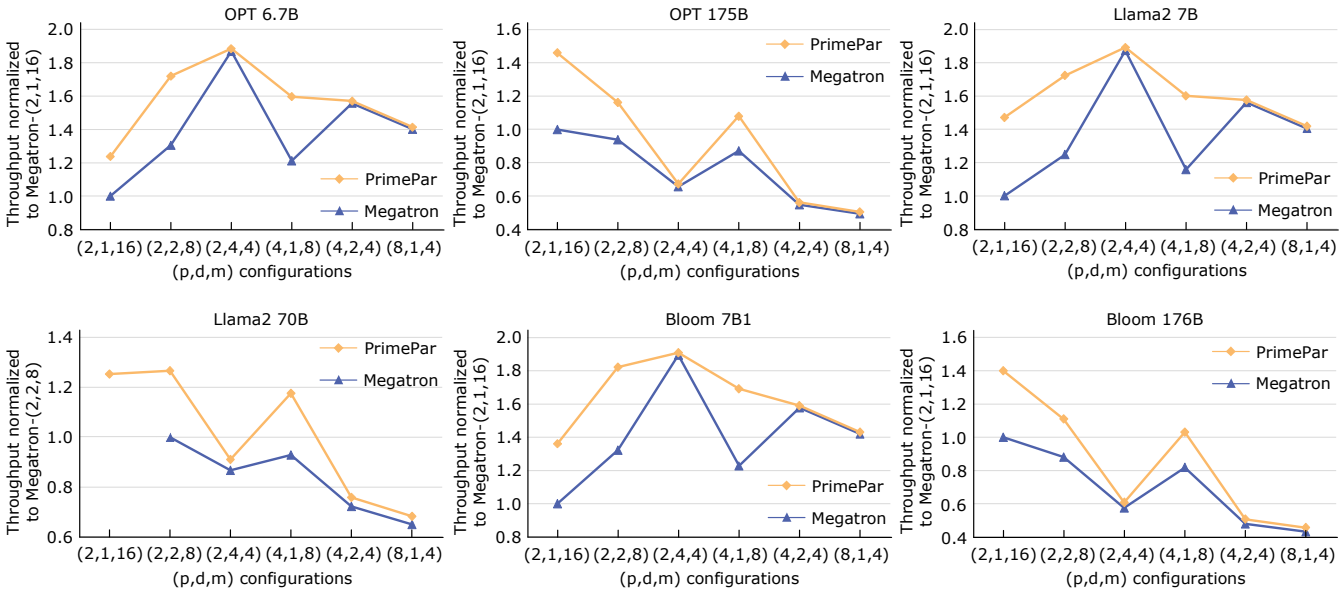


Figure 10. Normalized 3D Parallelism throughput of Megatron-LM and PrimePar with different (p, d, m) configurations, parallelizing to 32 GPUs.

that bandwidth are fully utilized. PrimePar is expected to achieve linear scaling on hardware platforms with balanced bandwidth and computation resources, as long as the ring communication latency per step is no longer than computation latency. Moreover, it is possible to utilize PrimePar to guide the co-design of computation and interconnection architecture.

8 RELATED WORKS

Tensor partition and automatic searching. Tensor partitioning has been explored to enable parallel deployment of models for acceleration and reducing memory overhead on individual device [15, 20, 21, 24, 63]. Optimizing tensor partitioning with minimum overhead [1, 17, 19, 20, 23, 25, 27, 31, 32, 34, 42, 47, 48] has been a highly anticipated topic. Recursive tensor partitioning has emerged as a critical formulation for automating tensor partitioning [33, 51, 52, 58], which covers a vast tensor partition space and is optimization friendly. Nevertheless, as far as we know, no research has introduced the temporal dimension into tensor partitioning for parallel training. The potential benefits and challenges brought by temporal dimension has not been studied.

Transformer parallel training systems. 3D parallelism has been extensively optimized to support large transformer model training [4, 26, 28, 37, 45, 46, 49, 50, 61, 67]. Data parallelism [9–12, 16, 29, 59, 65], model parallelism [13, 21, 53, 57] were applied to distribute input data or model parameter to multiple devices, each offering their own trade-offs. Pipeline parallelism [18, 30, 35, 36] were proposed to strip layers of a model over multiple GPUs, where pipeline execution is scheduled over micro-batches. ZeRO [45, 46] tackled tensor replication problem and proposed splitting optimizer states, in the cost of collective communication reduce-scatter and all-gather. These parallel training systems overlooked the temporal dimension in tensor partitioning (data, model parallelism), and can be further enhanced through the application of our proposed technique.

9 CONCLUSION

PrimePar introduces previously ignored temporal dimension into tensor partitioning, which provides abundant opportunities for better utilizing hardware resources and improving the performance of parallel training. We design a novel tensor partition primitive and apply it to improve LLM training performance, which is a successful practice in unlocking the potentials of spatial-temporal tensor partitioning. We provide a formalization which encodes the spatial-temporal distribution of sub-operators and exposes the essence of computation and communication, facilitating the design of efficient spatial-temporal tensor partitioning primitives. Further explorations into spatial-temporal tensor partition space are worthwhile.

ACKNOWLEDGMENTS

This work was supported in part by the National Natural Science Foundation of China (Grant No. 62025404, 62104229, 62104230, 61874124, 62222411), and in part by the Strategic Priority Research Program of Chinese Academy of Sciences (Grant No. XDB44030300, XDB44020300), Zhe jiang Lab underGrants 2021PC0AC01. The corresponding authors are Haobo Xu and Ying Wang.

References

[1] David Bau, Induprakas Kodukula, Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. Solving alignment using elementary linear algebra. In *Languages and Compilers for Parallel Computing: 7th International Workshop Ithaca, NY, USA, August 8–10, 1994 Proceedings 7*, pages 46–60. Springer, 1995.

[2] 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. Language models are few-shot learners, 2020.

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

[4] Yang Cheng, Dan Li, Zhiyuan Guo, Binyao Jiang, Jinkun Geng, Wei Bai, Jianping Wu, and Yongqiang Xiong. Accelerating end-to-end deep learning workflow with codesign of data preprocessing and scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 32(7):1802–1814, 2020.

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

[6] Jack Choquette, Edward Lee, Ronny Krashinsky, Vishnu Balan, and Brucek Khailany. 3.2 the a100 datacenter gpu and ampere architecture. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 64, pages 48–50. IEEE, 2021.

[7] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways, 2022.

[8] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. Deep learning with cots hpc systems. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1337–1345, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[9] Valeriu Codreanu, Damian Podareanu, and Vikram Saletore. Scale out for large minibatch sgd: Residual network training on imagenet-1k with improved accuracy and reduced time to train. *arXiv preprint arXiv:1711.04291*, 2017.

[10] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R Ganger, Phillip B Gibbons, et al. Exploiting bounded staleness to speed up big data analytics. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 37–48, 2014.

[11] Henggang Cui, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the eleventh european conference on computer systems*, pages 1–16, 2016.

[12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. *Advances in neural information processing systems*, 25, 2012.

[13] Nikoli Dryden, Naoya Maruyama, Tim Moon, Tom Benson, Marc Snir, and Brian Van Essen. Channel and filter parallelism for large-scale cnn training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–20, 2019.

[14] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *The Journal of Machine Learning Research*, 23(1):5232–5270, 2022.

[15] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. Tetris: Scalable and efficient neural network acceleration with 3d memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 751–764, 2017.

[16] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[17] Chua-Huang Huang and Ponnuswamy Sadayappan. Communication-free hyperplane partitioning of nested loops. *Journal of Parallel and Distributed Computing*, 19(2):90–102, 1993.

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

[19] David E Hudak and Santosh G Abraham. Compiler techniques for data partitioning of sequentially iterated parallel loops. *ACM SIGARCH Computer Architecture News*, 18(3b):187–200, 1990.

[20] Zhihao Jia, Sina Lin, Charles R Qi, and Alex Aiken. Exploring hidden dimensions in parallelizing convolutional neural networks. In *ICML*, pages 2279–2288, 2018.

[21] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019.

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

[23] Y J Ju and H Dietz. Reduction of cache coherence overhead by compiler data layout and loop transformation. In *Languages and Compilers for Parallel Computing: Fourth International Workshop Santa Clara, California, USA, August 7–9, 1991 Proceedings 4*, pages 344–358. Springer, 1992.

[24] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 380–392, 2016.

[25] Kathleen Knobe, Joan D Lukas, and Guy L Steele Jr. Data optimization: Allocation of arrays to reduce communication on simd machines. *Journal of parallel and Distributed Computing*, 8(2):102–118, 1990.

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

[27] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

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

[29] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on operating systems design and implementation (OSDI 14)*, pages 583–598, 2014.

[30] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. Terapipe: Token-level pipeline parallelism for training large-scale language models. In *International Conference on Machine Learning*, pages 6543–6552. PMLR, 2021.

[31] Qingda Lu, Christophe Alias, Uday Bondhugula, Thomas Henretty, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, Ponnuswamy Sadayappan, Yongjian Chen, Haibo Lin, et al. Data layout transformation for enhancing data locality on nuca chip multiprocessors. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 348–357. IEEE, 2009.

[32] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 553–564. IEEE, 2017.

[33] Jiachen Mao, Zhongda Yang, Wei Wen, Chunpeng Wu, Linghao Song, Kent W Nixon, Xiang Chen, Hai Li, and Yiran Chen. Mednn: A distributed mobile system with enhanced partition and deployment for large-scale dnns. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 751–756. IEEE, 2017.

[34] Igor Z Milosavljevic and Marwan A Jabri. Automatic array alignment in parallel matlab scripts. In *Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. IPPS/SPDP 1999*, pages 285–289. IEEE, 1999.

[35] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[36] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning*, pages 7937–7947. PMLR, 2021.

[37] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[38] NVIDIA. Nvidia collective communications library (nccl). https://developer.nvidia.com/nccl, 2018.

[39] NVIDIA. Nvidia cuda. https://developer.nvidia.com/cuda-zone, 2023.

[40] OPEN-MPI. Cuda aware mpi. https://www.open-mpi.org/faq/?category=runcuda, 2023.

[41] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[42] Michael Philippsen. Automatic alignment of array data and processes to reduce communication time on dmpps. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 156–165, 1995.

[43] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[44] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.

[45] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

[46] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[47] J Ramanujam and P Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Transactions on parallel and distributed systems*, 2(4):472–482, 1991.

[48] Jagannathan Ramanujam and Ponnuswamy Sadayappan. A methodology for parallelizing programs for multicomputers and complex memory multiprocessors. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 637–646, 1989.

[49] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[50] Jaeyong Song, Jinkyu Yim, Jaewon Jung, Hongsun Jang, Hyung-Jin Kim, Youngsok Kim, and Jinho Lee. Optimus-cc: Efficient large nlp model training with 3d parallelism aware communication compression. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 560–573, 2023.

[51] Linghao Song, Fan Chen, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. Accpar: Tensor partitioning for heterogeneous deep learning accelerators. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 342–355. IEEE, 2020.

[52] Linghao Song, Jiachen Mao, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. Hypar: Towards hybrid parallelism for deep learning accelerator array. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 56–68. IEEE, 2019.

[53] Jakub M Tarnawski, Deepak Narayanan, and Amar Phanishayee. Piper: Multidimensional planner for dnn parallelization. *Advances in Neural Information Processing Systems*, 34:24829–24840, 2021.

[54] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

[55] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert

Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

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

[57] Minjie Wang, Chien-chin Huang, and Jinyang Li. Unifying data, model and hybrid parallelism in deep learning via tensor tiling. *arXiv preprint arXiv:1805.04170*, 2018.

[58] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–17, 2019.

[59] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, and Eric P Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 381–394, 2015.

[60] BigScience Workshop, :, Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, Jonathan Tow, Alexander M. Rush, Stella Biderman, Albert Webson, Pawan Sasanka Ammanamanchi, Thomas Wang, Benoît Sagot, Niklas Muennighoff, Albert Villanova del Moral, Olatunji Ruwase, Rachel Bawden, Stas Bekman, Angelina McMillan-Major, Iz Beltagy, Huu Nguyen, Lucile Saulnier, Samson Tan, Pedro Ortiz Suarez, Victor Sanh, Hugo Laurençon, Yacine Jernite, Julien Launay, Margaret Mitchell, Colin Raffel, Aaron Gokaslan, Adi Simhi, Aitor Soroa, Alham Fikri Aji, Amit Alfassy, Anna Rogers, Ariel Kreisberg Nitzav, Canwen Xu, Chenghao Mou, Chris Emezue, Christopher Klamm, Colin Leong, Daniel van Strien, David Ifeoluwa Adelani, Dragomir Radev, Eduardo González Ponferrada, Efrat Levkovizh, Ethan Kim, Eyal Bar Natan, Francesco De Toni, Gérard Dupont, Germán Kruszewski, Giada Pistilli, Hady Elsahar, Hamza Benyamina, Hieu Tran, Ian Yu, Idris Abdulmumin, Isaac Johnson, Itziar Gonzalez-Dios, Javier de la Rosa, Jenny Chim, Jesse Dodge, Jian Zhu, Jonathan Chang, Jörg Frohberg, Joseph Tobing, Joydeep Bhattacharjee, Khalid Almubarak, Kimbo Chen, Kyle Lo, Leandro Von Werra, Leon Weber, Long Phan, Loubna Ben allal, Ludovic Tanguy, Manan Dey, Manuel Romero Muñoz, Maraim Masoud, María Grandury, Mario Šaško, Max Huang, Maximin Coavoux, Mayank Singh, Mike Tian-Jian Jiang, Minh Chien Vu, Mohammad A. Jauhar, Mustafa Ghaleb, Nishant Subramani, Nora Kassner, Nurulaqilla Khamis, Olivier Nguyen, Omar Espejel, Ona de Gibert, Paulo Villegas, Peter Henderson, Pierre Colombo, Priscilla Amuok, Quentin Lhoest, Rheza Harliman, Rishi Bommasani, Roberto Luis López, Rui Ribeiro, Salomey Osei, Sampo Pyysalo, Sebastian Nagel, Shamik Bose, Shamsuddeen Hassan Muhammad, Shanya Sharma, Shayne Longpre, Somaieh Nikpoor, Stanislav Silberberg, Suhas Pai, Sydney Zink, Tiago Timponi Torrent, Timo Schick, Tristan Thrush, Valentin Danchev, Vassilina Nikoulina, Veronika Laippala, Violette Lepercq, Vrinda Prabhu, Zaid Alyafeai, Zeerak Talat, Arun Raja, Benjamin Heinzerling, Chenglei Si, Davut Emre Taşar, Elizabeth Salesky, Sabrina J. Mielke, Wilson Y. Lee, Abheesht Sharma, Andrea Santilli, Antoine Chaffin, Arnaud Stiegler, Debajyoti Datta, Eliza Szczechla, Gunjan Chhablani, Han Wang, Harshit Pandey, Hendrik Strobelt, Jason Alan Fries, Jos Rozen, Leo Gao, Lintang Sutawika, M Saiful Bari, Maged S. Al-shaibani, Matteo Manica, Nihal Nayak, Ryan Teehan, Samuel Albanie, Sheng Shen, Srulik Ben-David, Stephen H. Bach, Taewoon Kim, Tali Bers, Thibault Fevry, Trishala Neeraj, Urmish Thakker, Vikas Raunak, Xiangru Tang, Zheng-Xin Yong, Zhiqing Sun, Shaked Brody, Yallow Uri, Hadar Tojarieh, Adam Roberts, Hyung Won Chung, Jaesung Tae, Jason Phang, Ofir Press, Conglong Li, Deepak Narayanan, Hatim Bourfoune, Jared Casper, Jeff Rasley, Max Ryabinin, Mayank Mishra, Minjia Zhang, Mohammad Shoeybi, Myriam Peyrounette, Nicolas Patry, Nouamane Tazi, Omar Sanseviero, Patrick von Platen, Pierre Cornette, Pierre François Lavallée, Rémi Lacroix, Samyam Rajbhandari, Sanchit Gandhi, Shaden Smith, Stéphane Requena, Suraj Patil, Tim Dettmers, Ahmed Baruwa, Amanpreet Singh, Anastasia Cheveleva, Anne-Laure Ligozat, Arjun Subramonian, Aurélie Névéol, Charles Lovering, Dan Garrette, Deepak Tunuguntla, Ehud Reiter, Ekaterina Taktasheva, Ekaterina Voloshina, Eli Bogdanov, Genta Indra Winata, Hailey Schoelkopf, Jan-Christoph Kalo, Jekaterina Novikova, Jessica Zosa Forde, Jordan Clive, Jungo Kasai, Ken Kawamura, Liam Hazan, Marine Carpuat, Miruna Clinciu, Najoung Kim, Newton Cheng, Oleg Serikov, Omer Antverg, Oskar van der Wal, Rui Zhang, Ruochen Zhang, Sebastian Gehrmann, Shachar Mirkin, Shani Pais, Tatiana Shavrina, Thomas Scialom, Tian Yun, Tomasz Limisiewicz, Verena Rieser, Vitaly Protasov, Vladislav Mikhailov, Yada Pruksachatkun, Yonatan Belinkov, Zachary Bamberger, Zdeněk Kasner, Alice Rueda, Amanda Pestana, Amir Feizpour, Ammar Khan, Amy Faranak, Ana Santos, Anthony Hevia, Antigona Unldreaj, Arash Aghagol, Arezoo Abdollahi, Aycha Tammour, Azadeh HajiHosseini, Bahareh Behroozi, Benjamin Ajibade, Bharat Saxena, Carlos Muñoz Ferrandis, Daniel McDuff, Danish Contractor, David Lansky, Davis David, Douwe Kiela, Duong A. Nguyen, Edward Tan, Emi Baylor, Ezinwanne Ozoani, Fatima Mirza, Frankline Ononiwu, Habib Rezanejad, Hessie Jones, Indrani Bhattacharya, Irene Solaiman, Irina Sedenko, Isar Nejadgholi, Jesse Passmore, Josh Seltzer, Julio Bonis Sanz, Livia Dutra, Mairon Samagaio, Maraim Elbadri, Margot Mieskes, Marissa Gerchick, Martha Akinlolu, Michael McKenna, Mike Qiu, Muhammed Ghauri, Mykola Burynok, Nafis Abrar, Nazneen Rajani, Nour Elkott, Nour Fahmy, Olanrewaju Samuel, Ran An, Rasmus Kromann, Ryan Hao, Samira Alizadeh, Sarmad Shubber, Silas Wang, Sourav Roy, Sylvain Viguier, Thanh Le, Tobi Oyebade, Trieu Le, Yoyo Yang, Zach Nguyen, Abhinav Ramesh Kashyap, Alfredo Palasciano, Alison Callahan, Anima Shukla, Antonio Miranda-Escalada, Ayush Singh, Benjamin Beilharz, Bo Wang, Caio Brito, Chenxi Zhou, Chirag Jain, Chuxin Xu, Clémentine Fourrier, Daniel León Periñán, Daniel Molano, Dian Yu, Enrique Manjavacas, Fabio Barth, Florian Fuhrimann, Gabriel Altay, Giyaseddin Bayrak, Gully Burns, Helena U. Vrabec, Imane Bello, Ishani Dash, Jihyun Kang, John Giorgi, Jonas Golde, Jose David Posada, Karthik Rangasai Sivaraman, Lokesh Bulchandani, Lu Liu, Luisa Shinzato, Madeleine Hahn de Bykhovetz, Maiko Takeuchi, Marc Pàmies, Maria A Castillo, Marianna Nezhurina, Mario Sänger, Matthias Samwald, Michael Cullan, Michael Weinberg, Michiel De Wolf, Mina Mihaljcic, Minna Liu, Moritz Freidank, Myungsun Kang, Natasha Seelam, Nathan Dahlberg, Nicholas Michio Broad, Nikolaus Muellner, Pascale Fung, Patrick Haller, Ramya Chandrasekhar, Renata Eisenberg, Robert Martin, Rodrigo Canalli, Rosaline Su, Ruisi Su, Samuel Cahyawijaya, Samuele Garda, Shlok S Deshmukh, Shubhanshu Mishra, Sid Kiblawi, Simon Ott, Sinee Sang-aroonsiri, Srishti Kumar, Stefan Schweter, Sushil Bharati, Tanmay Laud, Théo Gigant, Tomoya Kainuma, Wojciech Kusa, Yanis Labrak, Yash Shailesh Bajaj, Yash Venkatraman, Yifan Xu, Yingxin Xu, Yu Xu, Zhe Tan, Zhongli Xie, Zifan Ye, Mathilde Bras, Younes Belkada, and Thomas Wolf. Bloom: A 176b-parameter open-access multilingual language model, 2023.

[61] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, et al. Gspmd: General and scalable parallelization for ml computation graphs. arxiv e-prints, art. *arXiv preprint arXiv:2105.04663*, 2021.

[62] Ikuya Yamada, Akari Asai, Hiroyuki Shindo, Hideaki Takeda, and Yuji Matsumoto. Luke: deep contextualized entity representations with entity-aware self-attention. *arXiv preprint arXiv:2010.01057*, 2020.

[63] Xuan Yang, Jing Pu, Blaine Burton Rister, Nikhil Bhagdikar, Stephen Richardson, Shahar Kvatinsky, Jonathan Ragan-Kelley, Ardavan Pedram, and Mark Horowitz. A systematic approach to blocking convolutional neural networks, 2016.

[64] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive

pretraining for language understanding. *Advances in neural information processing systems*, 32, 2019.

[65] Sixin Zhang, Anna E Choromanska, and Yann LeCun. Deep learning with elastic averaging sgd. *Advances in neural information processing systems*, 28, 2015.

[66] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models, 2022.

[67] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.