Piper: Multidimensional Planner for DNN Parallelization

Anonymous Author(s)
Affiliation
Address
email

Abstract

The rapid increase in sizes of state-of-the-art DNN models and in the compute and memory requirements of training them has led to the development of many execution schemes such as data parallelism, pipelined model parallelism, tensor (intra-layer) model parallelism, and various memory-saving optimizations. However, no prior work has tackled the highly complex problem of finding the optimal partitioning of the DNN computation graph across many accelerators while combining the modes of parallelism and optimizations above. In this work we introduce Piper, an efficient optimization algorithm for this problem that is based on dynamic programming and a two-level approach. Our two-level approach is driven by the insight that being given tensor-parallelization techniques for individual layers (e.g. Megatron-LM) allows for a significant reduction of the search space, as opposed to considering arbitrary tensor-parallel configurations of the entire DNN operator graph, and makes the global problem tractable.

1 Introduction

Deep Neural Network (DNN) models have grown exponentially in size over the past two decades; consequently, modern DNNs are extremely computationally expensive. For example, a state of the art language model, GPT-3 [2], has an astounding 175-billion parameters, requiring 314 ZettaFLOPS (3.14×10^{23} FLOPS) of compute.

This trend of computationally expensive models with working-set sizes for training and inference that far exceed the memory capacities of individual DNN accelerators (GPUs, TPUs, FPGAs, ASICs, etc.) has reinvigorated the study of parallel training techniques that split DNN model state across different devices. In the "modern era", such model-parallel training techniques trace their roots back to AlexNet [12] and early influential systems such as DistBelief [6] and Project Adam [3]. Recent advances in model-parallel training involve *tensor (intra-layer) model parallelism* [18], where individual operators of the model are partitioned over multiple workers, hybrid parallelism that combines tensor and data parallelism across operators [11], and pipelined model parallelism that combines intra-batch (cross-layer model and data parallel) with inter-batch parallelism (pipelining) [15, 16, 9].

Given the importance of model-parallel training (and inference [7, 4]), partitioning a model across devices has received recent interest, evolving from a manual process driven by human experts to using algorithms for automated device placement of DNN operators. Focusing on algorithms, some take a black-box approach using learning-based techniques [14, 13, 8, 1, 21, 17]; others build a cost model and algorithmically solve an optimization problem [10, 15, 16, 11, 19]. However, each of the prior algorithmic approaches tackle a slightly different subset of parallelization techniques, system constraints, and DNN optimizations; for example, FlexFlow [11] does not consider pipelining, PipeDream [15] does not consider memory-saving optimizations that are useful for training large models, and PipeDream and PipeDream-2BW [16] both do not consider tensor parallelism.

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

1.1 Our contribution

In a first, this paper covers a broad search space of parallelization and optimizations techniques while determining placement of DNN layers on a given deployment (number of devices with memory capacity and interconnect performance constraints). Specifically, our optimization algorithm (Piper) considers three different dimensions of parallelism (data, tensor model, and pipeline model parallelism), and a general framework of memory-saving optimizations that trade off throughput for lower memory footprint. The focus of our work is on maximizing throughput (for training and large-batch inference of DNN models). The spirit of this paper is theoretical; our focus is on algorithms for DNN partitioning across devices and not on systems or frameworks to execute parallel training.

Piper finds parallelization configurations that always use pipelined model parallelism, optimizing partitioning the DNN across multiple devices jointly with non-trivial combinations of the other above modes of parallelism and memory-saving optimizations, carefully chosen for each stage of the partition. Piper supports general Directed Acyclic Graphs and is agnostic of the type of DNN accelerators used. It is efficient both in theory and in practice.

In Section 5 we evaluate Piper on inputs obtained from profiling real-world DNN workloads, and study the effects of combining the various modes of parallelism and optimizations on performance. We find that Piper's large search space combined with a principled algorithmic approach allows it to find parallelization configurations that are fine-tuned to the given instance in non-trivial ways and that it compares favorably to planners from prior work (PipeDream, PipeDream-2BW). For example, we find (as expected) that in many settings pipeline model parallelism combined with per-shard data parallelism is enough to obtain high throughput; however, tensor parallelism is also required when one needs the number of microbatches in a batch to be smaller than the number of devices, as well as in highly memory-constrained settings, which also require memory-saving optimizations. Piper's ability to form heterogeneous stages also gives it an advantage in throughput over prior work.

Broader impact. The aim of this research is to lower monetary, energy and climate costs of training. However, by doing so, it may amplify any negative impacts that the usage of large models entails.

2 Background

Model parallelism with pipelining. Pipeline model parallelism is a recent technique [15, 9] used to accelerate distributed training of both small and large models. The operators of a model are partitioned over the available devices. A batch is then split into smaller microbatches, and execution is pipelined across these microbatches: while the second device starts executing the first microbatch, the first device starts executing the second microbatch, and so on. Pipeline model parallelism features small amounts of point-to-point communication, which helps from a scaling perspective.

A key problem with deploying pipeline parallelism is determining how to schedule forward and backward microbatches on the available devices, and how to stash state to ensure that the final model is high-quality. Different schedules come with different trade-offs along throughput, memory footprint, and semantics dimensions. For example, GPipe [9] maintains just a single weight version, but introduces periodic pipeline flushes where all microbatches in the pipeline are allowed to drain. This leads to idle resources and lower throughput, but also lower memory footprint. PipeDream [15], on the other hand, maintains a weight version for every in-flight input microbatch, but does not use pipeline flushes, leading to both high throughput and high memory footprint. PipeDream-2BW [16] tries to find a middle ground, where gradients are coalesced across microbatches, reducing the number of maintained weight versions, while still keeping throughput high.

Tensor (intra-layer) model parallelism. A model can be partitioned in other ways as well. Megatron-LM [18] slices Transformer blocks in NLP models like BERT and GPT across available devices. This is akin to performing distributed matrix multiplications. All-to-all communication in the form of scatters and gathers needs to be performed to ensure that these distributed operators retain their semantics. Megatron-LM uses a specific, hand-tuned strategy to distribute Transformer layers across multiple GPUs. An advantage of tensor parallelism is that some ways of slicing will result in reduced per-device memory usage, as model parameters will be split across devices.

Trying to find an optimal parallelization on the operator level is a very challenging problem, with a large search space, that in particular requires understanding the semantics of each operator. For example, a matrix multiplication can be split over the rows or columns of the left (input) matrix or of the right (weight) matrix, or a combination of these; some of these splits will require inserting an

addition operator to recover the result, some will subdivide the weights (others will not, requiring all-reduce synchronization), and some might yield lower latency than others. A matrix multiplication split over some dimension would accept input tensors that are also split over that same dimension, but e.g. a dropout operator requires the entire input tensor. Operator-granularity graphs are also very large and have considerable branching.

Data parallelism. Data parallelism is the de-facto way of parallel training today. With data parallelism, every worker has a copy of the full model. Inputs are sharded, and each worker computes weight gradients independently. Periodically, these weight gradients are aggregated using all-to-all communication or a parameter server. Vanilla data parallelism assumes that model parameters fit on a single device. For large models that do not, data parallelism is used on model shards; in this mode of usage, gradients need to be exchanged only among devices that are responsible for the relevant model shard. Data parallelism can be used to complement both pipeline model parallelism (as shown in PipeDream [15]) and tensor model parallelism (as shown in Megatron-LM [18]).

Other optimizations. When training large models, it is common to use memory-saving optimizations that trade off a reduction in throughput for a lower memory footprint. One example is **activation recomputation**, where the intermediate activations produced during the forward pass are not stored. Instead, the forward pass is performed a second time, just before the backward pass, from a stashed *input* activation, which is much smaller than the full set of intermediate activations. This drastically reduces the footprint of pipelined training, at the cost of executing the forward pass twice.

Related work on algorithms for partitioning and placement of DNN graphs. A number of systems have looked at the problem of partitioning DNN models across devices. However, each of these systems considers a subset of the parallelism dimensions considered in this paper. FlexFlow [11] considers many dimensions (data, tensor, and inter-layer model parallelism), but not pipelining; moreover, it uses a heuristic MCMC-based approach that requires domain knowledge to be encoded into the "parallel plan" generator. PipeDream [15] and Tarnawski et al. [19] use dynamic programming and consider a combination of pipeline model parallelism and data parallelism; they operate on linear (path) graphs and general DAGs, respectively. While PipeDream is oblivious to memory usage, its enhancement, PipeDream-2BW [16], targets large models that do not necessarily fit on a single accelerator. Exploiting the repetitive structure of some of these large models, such as Transformer-based language models, PipeDream-2BW's planner only considers configurations where every stage in the pipeline is replicated an equal number of times (equi-replicated stages), thus simplifying the search for parallelization configurations. PipeDream-2BW's planner also explicitly considers per-device memory capacity constraints and includes activation recomputation in its configuration search space. None of the three approaches above consider tensor parallelism.

Learning-based approaches (Mirhoseini et al. [14, 13], Spotlight [8], Placeto [1], GDP [21], REGAL [17]) treat the objective (latency or throughput) as a black box to be optimized using RL; runtimes are measured in an online fashion on the target system, which is very accurate but computationally expensive. Moreover, some of them optimize for latency or peak memory usage rather than throughput, and they do not explicitly consider splitting individual operators or pipelining.

All existing algorithms, except PipeDream-2BW, also do not consider memory-saving optimizations, that trade off throughput for reduction in memory footprint, in combination with different modes of parallelism; activation recomputation is one such important optimization. These optimizations are commonly needed for training models that exceed the memory capacities of modern accelerators.

3 Our model

In this section we describe the search space of parallelization configurations that Piper explores, as well as the objective function (cost model) that we minimize in order to find the best solution (global configuration). These give rise to the description of the optimization problem that Piper solves.

Piper is given as input a DNN workload which is described as a layer-granularity Directed Acyclic Graph (DAG) $G = (V, E)$. That is, each node corresponds to a single DNN layer such as LSTM or Transformer. Each edge (u, v) corresponds to a data transfer: node/layer v requires the output of u as its input. The input graph is annotated with quantities related to compute times, communication costs, and memory usage; we introduce and motivate these quantities in this section, and then formally list them in Section 3.4. To obtain an input to be supplied to Piper, these quantities should be profiled (or estimated). Furthermore, Piper is given: the number K of devices, available memory M per device, the network bandwidth B, and the maximum allowed number N of microbatches in a batch.

Space of configurations. Piper's search space consists of partitions of G into consecutive contiguous subgraphs, also called stages, which are executed using disjoint sets of devices. Each node/layer is assigned to exactly one stage. Each stage is executed using some number $d \cdot t$ of devices, combining data parallelism of degree d with tensor parallelism of degree t. That is, we want to find a partition $V = S_1 \cup ... \cup S_\ell$ and, for each stage $i = 1, ..., \ell$: **(a)** the degree d_i of data parallelism, **(b)** the degree t_i of tensor parallelism, **(c)** further configuration: how (for each layer in the subgraph) the tensor parallelism is carried out, as well as what other optimizations are used.

Thus the i-th stage will be run on $d_i \cdot t_i$ devices, and so we must have $\sum_{i=1}^{\ell} d_i \cdot t_i \leq K$. Furthermore, each stage should be feasible in terms of memory usage, and the sum $\sum_{i=1}^{\ell} d_i$ of data-parallel degrees, which is related to the effective number of microbatches in a batch (see Appendix A), must be at most N. We call this output a **solution**. We use the definition of contiguity from [19]:

Definition 3.1 *We say that a set $S \subseteq V$ is* contiguous *if there do **not** exist nodes $u \in S$, $v \in V \setminus S$, and $w \in S$ such that v is reachable from u and w is reachable from v.*

Time-Per-Sample. Our objective is to maximize throughput; that is, we want to minimize its inverse, the **Time-Per-Sample (TPS)**. For pipelined execution schemes such as PipeDream or PipeDream-2BW, it has been argued [19] as well as shown in practice [15] that the Time-Per-Sample of a solution equals the maximum Time-Per-Sample of a device. The problem of finding a solution with minimal TPS (absent data or tensor parallelism and other optimizations) was addressed by [19]; in their model, which we extend, the TPS of a device/subgraph is given as the sum of compute times of all nodes in the subgraph, plus costs of communication along edges coming into or leaving the subgraph. We now discuss how to incorporate the modes of parallelism from Section 2 into the model.

3.1 Tensor Model Parallelism Configurations (TMPCs)

As we remarked in Section 2, the task of making a global, joint decision about how to tensor-parallelize every *operator* in a DNN computation graph is hugely complex. In this work we therefore **reduce** this task to the problem of coming up with good tensor parallelization techniques *for individual layers*. Solving the latter problem is not part of Piper; instead, we assume that we already know (i.e., Piper is given as input) some number of good tensor parallelism configurations (for some layers v and some degrees t of tensor parallelism). In practice, such configurations can be devised by hand (as done e.g. for Transformer layers in Megatron-LM [18]; we use their technique in our experimental evaluation), or the problem can be attempted using automated methods similar to e.g. FlexFlow [11] or Tofu [20] or RL-based approaches. We remark that this problem has to only be solved for each layer *type*, e.g., even if the input DNN has many Transformer layers, one just needs a good technique for *a Transformer layer*. It is also not necessary to provide configurations for all layers v and possible degrees t. On the other hand, there could be many reasonable tensor parallelism configurations for the same v and t; for example, there might be a trade-off between memory usage and runtime.

We assume that, for each available configuration, Piper gets as input certain quantities related to compute, communication and memory costs; we list them formally in Section 3.4. A tuple containing these quantities is called a *Tensor Model Parallelism Configuration (TMPC)*. Piper takes as input, for each node/layer v and each degree t of tensor parallelism, a (possibly empty) list $T(v, t)$ of TMPCs.

In order to make our model simple and unified, *we will treat non-tensor parallelized execution ($t = 1$) as a special case, and also use TMPCs to describe it*. We also use this abstraction to encode various auxiliary optimizations (see Section 2), e.g., activation recomputation.[1]

3.2 Data parallelism

Consider a stage that is data-parallelized across $d > 1$ devices. Following PipeDream [15], we model the effect of this on TPS as follows. The compute load of the stage is spread evenly across devices (with a factor $1/d$). However, data-parallel execution necessitates a periodic resync, whose total communication cost is accounted as $4(d-1)w$, where w is the total size of parameters in all layers in this stage/subgraph. This gives a cost of $4 \cdot \frac{d-1}{d} \cdot w$ bytes per device (and per d microbatches).

[1] As an example, a node v might have the following TMPCs as input: for $t = 1$, one TMPC corresponding to regular execution, and another corresponding to activation recomputation; and for $t = 8$, one TMPC corresponding to tensor parallelization in some dimension, another corresponding to another dimension, and two more that mirror the first two but with added activation recomputation.

3.3 Memory usage in pipelined configurations

With pipelining, every stage needs to store some number of weight versions and some number of activation versions; these depend on the number of in-flight microbatches (per data-parallel replica). Specifically, the memory usage of a stage can be modeled as an **affine function** of this number. That is, for every stage there are coefficients a, b such that if this stage has data-parallel degree d and the sum of data-parallel degrees of this and all further stages is s, then its memory usage would be $a \cdot \lceil s/d \rceil + b$. See Appendix A for an explanation. For example: in PipeDream-style execution, a would be the size of weights plus the size of all activations in the subgraph, and b would comprise e.g. optimizer state or temporary buffers. PipeDream-2BW uses only two stashed weight versions instead of $\lceil s/d \rceil$ many, so the size of weights would now contribute to b (multiplied by 2), not a. If activation recomputation is also used, then a reflects only the sizes of *input* activations of every layer, whereas all other activations move to b. This accounting is more precise than in PipeDream, which ignores memory usage, or in [19], which uses a constant function (equivalent to not having a above).

3.4 Input & output

Now we can formally (re-)state Piper's **input**:

- a Directed Acyclic Graph $G = (V, E)$: layer graph of a DNN,
- for every edge $(u, v) \in E$, an associated communication cost $c(u, v)$ (in bytes),
- number K of devices, memory M per device, network bandwidth B, maximum number N of microbatches in a batch,
- for every node/layer $v \in V$, and every degree $t = 1, ..., K$ of tensor parallelism: a (possibly empty) list $T(v, t)$ of TMPCs (see Section 3.1).

The description of a TMPC in $T(v, t)$ must correspond to a tensor parallelism configuration where, for both the forward and the backward pass, each of the t devices starts execution already with a full copy of each input activation, and must end execution having in its memory a full copy of every output activation (see Appendix B for a discussion). Each TMPC description $X \in T(v, t)$ contains:

- a Time-Per-Sample $X.p$ that encompasses the compute time together with the communication costs of tensor parallelism (data transfers between the t devices),[2]
- for every incoming edge $(u, v) \in E$, a communication cost $X.c^{\mathsf{fw}}(u, v)$ (in bytes) that it would take to synchronize the tensor incoming over that edge between t devices (going from a state where one device has the tensor to a state where all t have it); for more explanation and motivation of this, see Appendix B,
- an analogous quantity $X.c^{\mathsf{bw}}(v, w)$ for every outgoing edge $(v, w) \in E$,
- the size $X.w$ (in bytes) of parameters/weights on each device,
- memory usage: two amounts $X.a$, $X.b$ of bytes such that if layer v is located on a stage with data-parallel degree d and data-parallel degrees of this and later stages sum up to s, then the memory consumption on each device is $X.a \cdot \lceil s/d \rceil + X.b$ (see Section 3.3).

Piper **outputs** the following description of a low-TPS solution: a collection of contiguous subgraphs/stages $S_1, ..., S_\ell$ such that $V = S_1 \cup ... \cup S_\ell$, and for each stage i: **(a)** the degree d_i of data parallelism, **(b)** the degree t_i of tensor parallelism, **(c)** for each node/layer $v \in S_i$: the index of the TMPC in $T(v, t_i)$ that is selected.

Note that all nodes/layers in a stage use the same degrees of data and tensor parallelism, but different nodes (even if all correspond to e.g. Transformer layers) can use different TMPCs. For example, some but not all layers in a stage might employ activation recomputation.

4 Algorithm

Our algorithm is based on a dynamic programming on *downsets* approach which extends that of [19].

Definition 4.1 *We call a set $D \subseteq V$ of nodes a* downset *if $(u, v) \in E$ and $u \in D$ implies $v \in D$.*

As proved there, going from downset to downset we can obtain every possible contiguous subgraph:

Fact 4.2 *A set $S \subseteq V$ of nodes is contiguous (see Definition 3.1) if and only if it is the difference of two downsets: $S = D \setminus D'$ where $D' \subseteq D$.*

[2] This is what one gets from profiling. It should be the total runtime of the forward and the backward pass.

A domain-knowledge insight that we rely on is that computation graphs of modern DNNs (and especially layer-granularity graphs) do not have many downsets, making tractable the approach of considering all downsets and all contiguous sets using dynamic programming.

We define the **meaning of the dynamic programming table** as follows: $dp[D][k][s]$ shall be the minimum TPS of a solution that partitions the downset D among k devices (recall that the TPS of a solution is the maximum TPS over all k devices) so that the sum of data-parallel degrees of all stages is s. A key novel point in our approach is that using the single variable s, which ranges from 0 to N, we can precisely control both memory usage (see Section 3.3) and batch size (Appendix A).

We **initialize** the table as follows: $dp[\emptyset][k][s] = 0$ for every k and s. The answer (minimum TPS) when using any number k of devices and sum s of data-parallel degrees will be held in $dp[V][k][s]$, and the final answer is $\min_{k=1,\ldots,K} \min_{s=1,\ldots,N} dp[V][k][s]$.

We proceed to the main part: **the recursion**. For every possible sub-downset D', we consider partitioning the contiguous subgraph $S = D \setminus D'$ with data-parallel degree d and tensor-parallel degree t, so that $d \cdot t$ devices are used for S and the rest for D'. For $\emptyset \subsetneq D \subseteq V$, we have

$$dp[D][k][s] = \min_{\text{downset } D' \subseteq D} \min_{d=1}^{s} \min_{t=1}^{\lfloor k/d \rfloor} \max(dp[D'][k - d \cdot t][s - d], \mathrm{TPS}(D \setminus D', t, d, s)),$$

where we **define** $\mathrm{TPS}(S, t, d, s)$ **as** the minimum Time-Per-Sample when optimally partitioning S with degrees d and t of data and tensor parallelism, respectively, and so that the sum of data-parallel degrees of this and all further stages is s. It remains to compute this quantity.

The knapsack subproblem. How to compute $\mathrm{TPS}(S, t, d, s)$? The parallelism degrees d and t are already decided, and what remains is to choose one TMPC $X \in T(v, t)$ for each $v \in S$. Some TMPCs may have better TPS, but others may use less memory. More concretely, we want a combination that minimizes the total TPS of the stage while staying under the memory limit M. Both the total TPS and the total memory usage of any combination can be expressed as sums of the TPSs and the memory usages of TMPCs of individual nodes $v \in S$ (see the subsection below). However, there are still exponentially many $\left(\prod_{v \in S} |T(v, t)|\right)$ possibilities, and in fact this subproblem is NP-hard, as it generalizes the 0-1 knapsack problem. At the same time, we require a very fast solution as it needs to be solved for all S, t, d, s. We use a simple bang-per-buck heuristic:

- start by picking the lowest-TPS (fastest) TMPC for each $v \in S$,
- as long as the memory usage is too high, pick the TMPC in $\bigcup_{v \in S} T(v, t)$ whose ratio of decrease in memory usage (compared to the currently picked TMPC for v) to increase in TPS is the highest, and pick it for v (instead of the currently picked one).

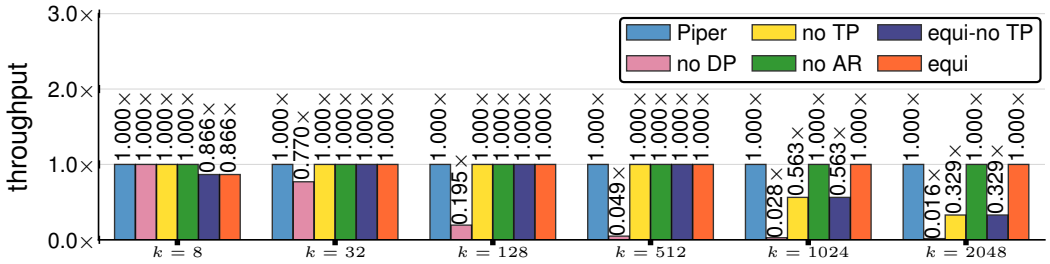
If no combination of TMPCs is memory-feasible, we return $+\infty$. Note that this is the only non-exact component of the algorithm; however, this heuristic in fact almost always finds the optimal solution in practice. See Appendix C for more discussion.

TPS of a fixed configuration. Finally, how do we compute the TPS of a stage for given S, t, d, s and a selection of an TMPC $X(v) \in T(v, t)$ for every $v \in S$? We need to take compute, communication and memory into account. The **compute** part is the simplest: $\mathrm{compute} = \frac{1}{d} \sum_{v \in S} X(v).p$.
For **communication**, we have costs stemming from sending activations into S, sending activations out of S, and costs of data parallelism resyncs (all in bytes):

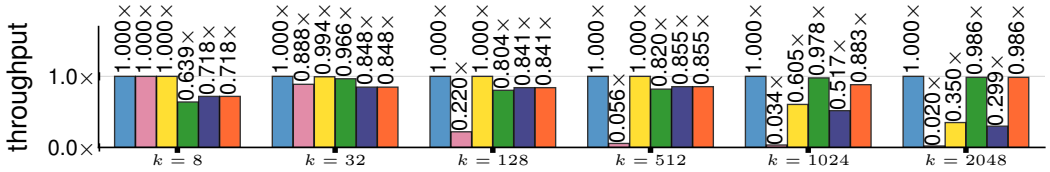
$$\mathrm{comm} = \frac{1}{d} \left(\sum_{(u,v) \in \delta^-(S)} 2 \left(c(u, v) + X(v).c^{\mathsf{fw}}(u, v) \right) + \sum_{(v,w) \in \delta^+(S)} 2 \left(c(v, w) + X(v).c^{\mathsf{bw}}(v, w) \right) + 4 \cdot \frac{d-1}{d} \cdot \sum_{v \in S} X(v).w \right)$$

where we used the notation $\delta^-(S)$ and $\delta^+(S)$ for the set of incoming and outgoing edges of S, respectively. Finally, the **memory usage**: $\mathrm{mem} = \sum_{v \in S} (X(v).a \cdot \lceil s/d \rceil + X(v).b)$.
Our final TPS for the selected S, t, d, s and $\{X(v)\}_{v \in S}$ is $\mathrm{compute} + \frac{\mathrm{comm}}{B}$ as long as $\mathrm{mem} \leq M$; otherwise we return $+\infty$.

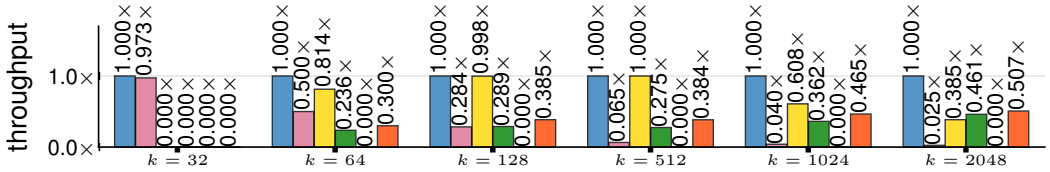
Running time. Under some natural assumptions on the workload, Piper's running time is $\tilde{O}(|V|^2 N K^2)$ (recall that $|V|$ is the number of nodes, K is the number of devices, and $N \leq K$ is the maximum sum of data-parallel degrees). A detailed analysis can be found in Appendix D.



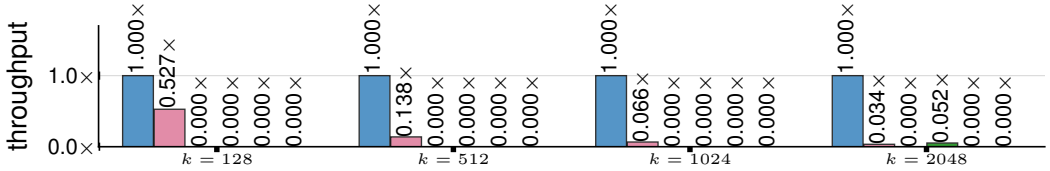
(a) BERT-32, 80GB of memory on each of k devices, $N = 512$



(b) BERT-32, 8GB of memory on each of k devices, $N = 512$



(c) BERT-32, 2GB of memory on each of k devices, $N = 512$



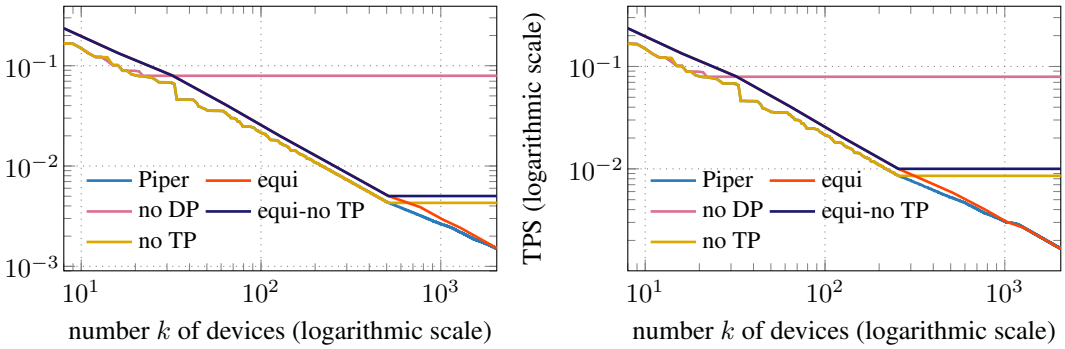
(d) BERT-32, 1GB of memory on each of k devices, $N = 512$

Figure 1: Comparison of Piper to baselines, in terms of the throughput of the best found configuration. Throughput is the inverse of TPS (Time-Per-Sample): higher throughput is better. Piper always obtains the highest throughput; values in each block are scaled so that Piper is $1.000\times$. We consider numbers k of devices from 8 to 2048 and varying assumptions on the memory per device, which has a large impact on the space of feasible solutions. Indeed, for 1GB or 2GB per device, baselines are often unable to find *any* memory-feasible configuration (this is shown as $0.000\times$). 1GB models a scenario where TP and AR are both required to fit even a single layer on a device.

5 Evaluation

In this section we evaluate the Piper algorithm on real-world DNN workloads. Besides the efficiency and scalability of Piper, we are also interested in the advantages of the configurations produced by Piper over best possible configurations that forgo at least one of the modes of parallelism or memory-related optimizations, as well as over best possible *equi-partitioned configurations* such as those found by PipeDream-2BW's planner algorithm. We stress that in this work we focus on evaluating the Piper *partitioning algorithm*, not any particular pipelined DNN training system. However, Fig. 4 also shows that Piper's cost model is very close to real performance. Moreover, with a fixed batch size and pipelining scheme, minimizing TPS is equivalent to minimizing time-to-convergence.

Inputs. For our comparisons we use a BERT-32 model, which consists of an embedding layer, 32 Transformer layers and a pooler layer. This model has 5.9 billion parameters, with similar dimensions to the GPT-3 model with 6.7B parameters [2], and does not fit on a single GPU. We provide TMPCs for non-tensor-parallelized execution ($t = 1$) as well as for the parallelizations of Transformer layers proposed in Megatron-LM [18] ($t \in \{2, 4, 8\}$): each with or without activation recomputation. The



(a) BERT-32, 8 GB RAM per device, $N = 512$ (b) BERT-32, 8 GB RAM per device, $N = 256$

Figure 2: A *single run* of Piper produces a 2D table of optimal Times-Per-Sample for every number k of devices up to K and every sum s of data-parallel degrees up to N. Here we show the best TPS for Piper and the baselines as a function of the number of devices, for two different values of N. It is clear that tensor parallelism is needed to use more than N devices, which shows as the **no TP** baselines plateauing after $k = N$. **equi** obtains a 10-15% worse TPS than Piper for most k.

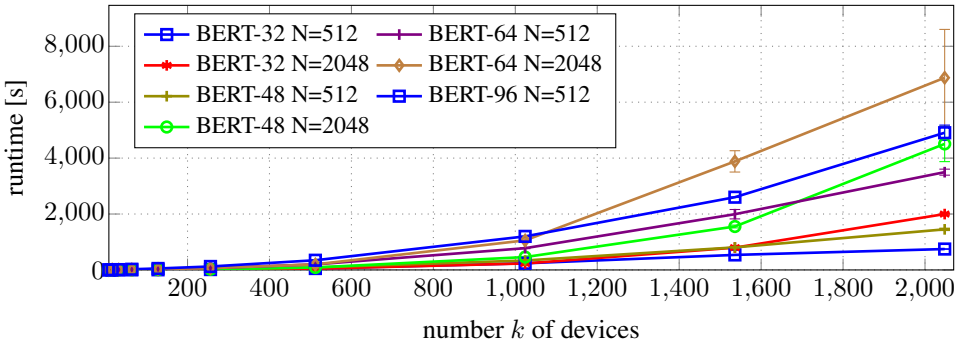


Figure 3: The running time of our algorithm, as a function of the DNN workload size and the number k of devices. Standard deviation is denoted by bars. Single-core, unoptimized implementation executed on a mobile Intel i7-8665U CPU.

quantities in all these TMPCs are obtained by profiling models implemented in PyTorch on NVidia A100 GPUs interconnected with a 300 GB/s bandwidth NVSwitch. In our evaluation of Piper we consider a scenario where, other than this fast intra-cluster communication for tensor parallelism, all $K \leq 2048$ devices are connected with a flat network topology at a (full bisection) bandwidth of $B = 25$ GB/s. Our TMPCs model a PipeDream-2BW-style pipelining scheme.

Baselines. We compare Piper to the following:
• **no DP**: a version of Piper that forgoes data parallelism (i.e. we always use $d = 1$),
• **no TP**: a version of Piper that forgoes tensor parallelism (use $t = 1$),
• **no AR**: a version of Piper that forgoes activation recomputation (ignore the corresponding TMPCs),
• **equi**: an algorithm that equi-partitions layers into w equal groups and uses d-data parallelism and t-tensor parallelism, choosing the lowest-TPS of all valid choices of (w, d, t); it also tries both using or not using activation recomputation (in the entire DNN), as well as placing the embedding and pooler layers either on separate stages or with the first and last Transformer layer respectively,
• **equi-no TP**: a version of **equi** that forgoes tensor parallelism (use $t = 1$); we show this as it closely reflects the behavior of PipeDream-2BW's planner algorithm.

Both **no TP** and **no AR** are more sophisticated than the partitioning algorithm of PipeDream (which can be seen as a version of Piper that forgoes tensor parallelism, activation recomputation, and also does not take memory usage into account (assumes $M = \infty$)).

5.1 Results

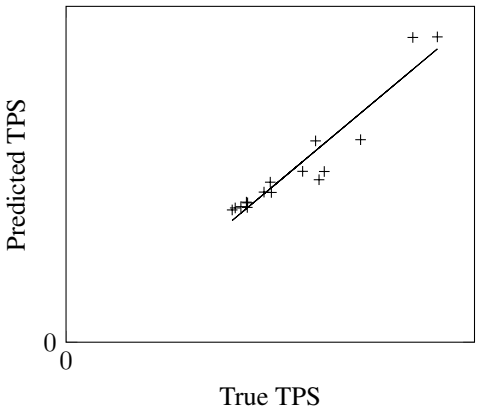


Figure 4: Real vs. Piper-predicted Time-Per-Sample for 16 configurations of pipeline, data, and tensor parallelism for BERT-32 with $K = 64$ devices. Training times were measured on a system with 8 NVidia DGX A100 machines, each with 8 GPUs. The Pearson correlation coefficient is 0.95.

Results of our evaluation in terms of the quality (TPS) of the obtained configurations are given in Figs. 1 and 2. The running times of Piper are shown in Fig. 3. We discuss the main takeaways.

Data parallelism (DP) is crucial, as expected. Without it, one cannot hope for a good throughput; e.g. in Fig. 2, the TPS of **no DP** stops decreasing beyond $k = 22$ devices.

Tensor model parallelism (TP). We found that, in a vacuum, the speedup obtained by TP is much less than that of DP due to frequent and costly synchronization. However, TP also shrinks the parameter size per device, which helps with memory feasibility, and does not increase s (in contrast to DP), which enables one to stay below the specified number N of microbatches in a batch. In particular, TP is simply required to use $k > N$ devices (see the **no TP** baseline in e.g. Fig. 1a or Fig. 2). TP is also required in cases when even a single layer cannot fit onto a device (Fig. 1d).

Activation recomputation (AR) can also be required to obtain *any* memory-feasible solution. Even when it is possible to do so without AR, foregoing it can force quite imbalanced (in terms of TPS) splits in order to fit within memory limits. For example, in Fig. 1c, without AR, one is forced to use small s-values to stay within memory, which limits the possible number of stages and data-parallel degrees, and hence performance (as using TP in lieu of DP or pipeline model splits yields worse throughput). In scenarios with ample memory in relation to DNN size (Fig. 1a), AR is not useful.

Power of Piper's search space. Piper has the ability to create configurations with stages that are different from each other (different numbers of layers, degrees d and t, memory optimizations, etc.). Does this lead to a lower optimal TPS (than the baseline **equi**) even for extremely repetitive DNNs such as BERT? The answer is yes, as seen from Figs. 1 and 2. Let us explain two ways in which this extra power is useful. First, in earlier stages, the parameter s that controls memory usage (see Section 3.3) is higher (as s is a "suffix sum" of DP degrees over stages), which means that it is optimal to have larger degrees d or t or smaller numbers of layers in the earlier stages. In other words, in an optimal *equi-partitioned* configuration, if memory is a bottleneck, then the memory limit will be attained almost tightly in the first stage, which needs to store activations for many in-flight microbatches, but it will be loose for the last stage, which only stores activations of one microbatch. In contrast, Piper will effectively adjust parameters of all stages so as to use almost all the available memory on every device. Secondly, Piper is also able to treat different types of layers differently; for instance, the embedding layer can use a different TP degree t than Transformer layers.

Running time of Piper. Fig. 3 shows the scalability of Piper's algorithm as a function of k and $|V|$, as well as N, on larger BERT inputs created by expanding our BERT-32 profiles. For $K \leq 512$, Piper terminates in minutes. Even at scale, for the largest K values of 2048 in our evaluation, Piper is able to arrive at solutions within 13 minutes for BERT-32 ($N = 512$) to 2 hours for BERT-64 ($N = 2048$). Such solving times are acceptable, given they are a very small percentage of the actual training times of the DNNs (days to weeks). While we do not focus on an optimized implementation for the paper, we note that Piper's algorithm is embarrassingly parallel, as for every I, computing the table $\mathrm{TPS}(I \setminus I', \cdot, \cdot)$ and updating dp can be done in parallel over all sub-downsets $I' \subseteq I$. With such an implementation, runtimes for the algorithm would scale linearly on a multi-core CPU server.

6 Extensions and future work

We had assumed a flat network topology. While this already yields precise estimates (Fig. 4), Piper can be extended to handle hierarchical network topologies (see Appendix E).

The main open question prompted by this work is how to automatically find good tensor-parallelism schemes, which can then be used by Piper to find global configurations. Thanks to Piper, now this task must be solved only for individual layers (layer types), rather than entire DNN operator graphs.

References

[1] Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. Learning generalizable device placement algorithms for distributed machine learning. In *Advances in Neural Information Processing Systems 32*, pages 3981–3991. Curran Associates, Inc., 2019.

[2] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[3] Trishul M 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)*, volume 14, pages 571–582, 2014.

[4] Eric S. Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian M. Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Maleen Abeydeera, Logan Adams, Hari Angepat, Christian Boehn, Derek Chiou, Oren Firestein, Alessandro Forin, Kang Su Gatlin, Mahdi Ghandi, Stephen Heil, Kyle Holohan, Ahmad El Husseini, Tamás Juhász, Kara Kagi, Ratna Kovvuri, Sitaram Lanka, Friedel van Megen, Dima Mukhortov, Prerak Patel, Brandon Perez, Amanda Rapsang, Steven K. Reinhardt, Bita Rouhani, Adam Sapek, Raja Seera, Sangeetha Shekar, Balaji Sridharan, Gabriel Weisz, Lisa Woods, Phillip Yi Xiao, Dan Zhang, Ritchie Zhao, and Doug Burger. Serving dnns in real time at datacenter scale with project brainwave. In *IEEE Micro*, 2018.

[5] J. Csirik, J. B G Frenk, M. Labbe, and S. Zhang. Heuristics for the 0-1 min-knapsack problem. *Acta Cybernetica*, 10(1-2):15–20, January 1991.

[6] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.

[7] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steve Reinhardt, Adrian Caulfield, Eric Chung, and Doug Burger. A configurable cloud-scale dnn processor for real-time ai. In *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA '18)*, 2018.

[8] Yuanxiang Gao, Li Chen, and Baochun Li. Spotlight: Optimizing device placement for training deep neural networks. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1676–1684, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[9] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, 2019.

[10] Zhihao Jia, Sina Lin, Charles R Qi, and Alex Aiken. Exploring hidden dimensions in parallelizing convolutional neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML '18)*, 2018.

[11] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In *Proceedings of the 2nd SysML Conference, SysML '19*, Palo Alto, CA, USA, 2019.

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, 2012.

[13] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V. Le, and Jeffrey Dean. A hierarchical model for device placement. *ICLR*, 2018.

[14] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2430–2439. JMLR. org, 2017.

[15] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, Phil Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for DNN training. In *Proc. 27th ACM Symposium on Operating Systems Principles (SOSP)*, Huntsville, ON, Canada, October 2019.

[16] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel dnn training, 2020.

[17] Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. Reinforced genetic algorithm learning for optimizing computation graphs. In *International Conference on Learning Representations*, 2020.

[18] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[19] Jakub Tarnawski, Amar Phanishayee, Nikhil R. Devanur, Divya Mahajan, and Fanny Nina Paravecino. Efficient algorithms for device placement of DNN graph operators. In *NeurIPS*, 2020.

[20] 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*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[21] Yanqi Zhou, Sudip Roy, Amirali Abdolrashidi, Daniel Wong, Peter C. Ma, Qiumin Xu, Ming Zhong, Hanxiao Liu, Anna Goldie, Azalia Mirhoseini, and James Laudon. Gdp: Generalized device placement for dataflow graphs, 2019.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes]
   (c) Did you discuss any potential negative societal impacts of your work? [Yes]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes]
   (b) Did you include complete proofs of all theoretical results? [Yes]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes]
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [N/A]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] (where appropriate)
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [No] The experiments were run on a laptop; the total time can be inferred from Fig. 3 if needed. For Fig. 4 we only profiled a small number of training steps for each configuration.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [N/A]
   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A] (we include one sample input file for Piper obtained by profiling BERT)

(d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A] (no personal data used)

(e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A] (no personal data used)

5. If you used crowdsourcing or conducted research with human subjects...

(a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]

(b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]

(c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]