# Piper: Multidimensional Planner for DNN Parallelization

**Jakub Tarnawski**
Microsoft Research
jakub.tarnawski@microsoft.com

**Deepak Narayanan**
Microsoft Research
dnarayanan@microsoft.com

**Amar Phanishayee**
Microsoft Research
amar@microsoft.com

## Abstract

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

## 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 ($3.14 \times 10^{23}$ 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 (e.g., GPUs, TPUs, FPGAs, ASICs) 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 [13] and early influential systems such as DistBelief [6] and Project Adam [3]. Recent advances in model-parallel training include *tensor (intra-layer) model parallelism* [21], where individual operators of the model are partitioned over multiple workers, and pipeline model parallelism that combines inter-layer model parallelism with pipelining [16, 17, 10]. Various hybrid parallelism techniques that combine tensor, pipeline, and data parallelism across operators [12, 18] have also been developed to more efficiently train a wider range of models. Additionally, memory-saving optimizations like activation recomputation, which trade off throughput for memory, are also being used [17] to scale training to larger models such as GPT-3.

Combining these dimensions, however, is non-trivial [18], since each dimension has trade-offs with respect to computational efficiency, amount of communication, and memory footprint. Given the importance of *efficient* model-parallel training (and inference [7, 4]), partitioning a model across

35th Conference on Neural Information Processing Systems (NeurIPS 2021).

devices has thus 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 [15, 14, 8, 1, 25, 20, 24]; others build a cost model and algorithmically solve an optimization problem [11, 16, 17, 12, 22]. However, each of the prior algorithmic approaches tackle a slightly different subset of parallelization techniques, system constraints, and DNN optimizations; for example, FlexFlow [12] does not consider pipelining, PipeDream [16] does not consider memory-saving optimizations that are useful for training large models, and PipeDream and PipeDream-2BW [17] both do not consider tensor parallelism.

**Our contribution.** In a first, this paper covers a broad search space of parallelization and optimization techniques while determining placement of DNN layers on the provided hardware deployment (a given number of accelerators of a specific type with certain performance characteristics, memory capacity, and interconnect bandwidth). Specifically, our optimization algorithm, called 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. In this work, we consider pipeline parallelism schedules without pipeline flushes [16, 17]. We leave extensions to pipeline parallelism schedules with flushes [10, 18] to future work.

The focus of our work is on algorithms for DNN partitioning across devices to maximize throughput (for training and large-batch inference of DNN models). Piper uses non-trivial combinations of data, tensor, and pipeline parallelism, as well as memory-saving optimizations such as activation recomputation, carefully chosen for each stage of the partition. Piper supports general DAG (Directed Acyclic Graph) model topologies and is agnostic of the type of DNN accelerator used. Piper is given a target "safe" batch size (e.g., 1920 for large GPT-style models [19]) and returns a training configuration with at most (or exactly) this batch size. It is efficient both in theory and in practice.

In Section 5, we evaluate Piper on real-world DNN profiles, and study the effects of combining the various parallelism modes and memory-saving optimizations on performance. We find that Piper's large search space combined with its principled algorithmic approach allows it to find non-trivial high-quality parallelization configurations for each given problem instance. Piper 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 a batch size smaller than the number of devices, as well as in highly memory-constrained settings. Piper's ability to form heterogeneous stages also gives it an advantage over prior work.

## 2 Background

In this section, we briefly describe methods to train models at scale.

### 2.1 Pipeline model parallelism

Pipeline model parallelism is a recent technique [16, 10] used to accelerate the distributed training of both small and large models. The operators of a model are partitioned over the available devices (i.e., each device hosts a subset of the operators of the model; typically these subsets are contiguous). 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 cheap point-to-point communication.

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 a final high-quality model. Different schedules have different throughput, memory footprint, and semantics tradeoffs. For example, GPipe [10] maintains just a single weight version, but introduces periodic pipeline flushes where microbatches in the pipeline are allowed to drain. This leads to idle resources and lower throughput, but also lower memory footprint. PipeDream [16] maintains a weight version for every in-flight input microbatch, but does not use pipeline flushes, leading to both high throughput and memory footprint. PipeDream-2BW [17] tries to find a middle ground, where gradients are coalesced across microbatches, reducing the number of weight versions, while still keeping throughput high. In this work, we assume that a non-flushing pipelining scheme is used when considering pipelines with multiple stages (inter-layer model parallelism is never used in isolation).

### 2.2 Tensor (intra-layer) model parallelism

A model can be partitioned in other ways as well. Megatron-LM [21] slices transformer blocks in NLP models like BERT and GPT across available devices, in effect performing distributed matrix multiplications. All-to-all communication in the form of scatters, gathers and all-reductions 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. Tensor parallelism reduces per-device memory usage, as model parameters are split across devices, and shows good scaling performance when using high-bandwidth interconnects like NVLink [18].

Trying to find an optimal tensor parallelization strategy for the entire model by deciding how to split every operator is a highly challenging problem with a large search space, and requires understanding the semantics of each operator. Modern models consist of a large number of operators, often with considerable branching. Moreover, each operator can be split in many different ways. For example, a matrix multiplication can be split over the rows or columns of the left (input) matrix or right (weight) matrix, or a combination of these. Some splits will require inserting an all-reduction operator to recover the result, and some will require an all-gather. Some will divide weights across devices, which helps with memory usage. The optimal split in terms of latency depends on the matrix dimensions and also downstream operators; a matrix multiplication split over some dimension would accept input tensors split over the same dimension, but a dropout operator requires the entire input tensor.

### 2.3 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, data parallelism can be used on model shards; gradients need to only be exchanged among devices responsible for the relevant model shard. Data parallelism can be used with both pipeline (as shown in PipeDream [16]) and tensor model parallelism (as shown in Megatron-LM [21]).

### 2.4 Memory-saving 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.

### 2.5 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 [12] 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 [16] and Tarnawski et al. [22] use dynamic programming and consider a combination of pipeline model parallelism and data parallelism; however, they support different model topology classes. PipeDream supports only linear (path) graphs while [22] supports general DAGs. While PipeDream is oblivious to memory usage, its enhancement, PipeDream-2BW [17], 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 these three approaches consider tensor parallelism.

Learning-based approaches (Mirhoseini et al. [15, 14], Spotlight [8], Placeto [1], GDP [25], REGAL [20]) 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 Problem Setup

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. These give rise to the description of the optimization problem that Piper solves.

Piper is given as input a DNN workload (model) which is represented as a Directed Acyclic Graph (DAG), $G = (V, E)$. Each node corresponds to a single DNN layer such as LSTM or transformer (each layer is composed of lower-level operators, e.g., matrix multiplications or additions). 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 a number of attributes related to compute times, communication costs, and memory usage; we introduce and motivate these attributes in this section, and then formally list them in Section 3.6. To obtain an input to be supplied to Piper, these attributes should be profiled (or estimated). Furthermore, Piper is given: the number of devices ($K$), available memory per device ($M$), the network bandwidth ($B$), and the maximum allowed number of microbatches in a batch ($N$). $N$ is the ratio of the maximum batch size safe for convergence (obtained from past runs for similar models, e.g., 1024–2048 for large transformer-based LMs) to the provided microbatch size.

## 3.1 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 of data-parallel degrees ($\sum_{i=1}^{\ell} d_i$), which is related to the effective number of microbatches in a batch (see Appendix A), must be at most (or exactly) $N$. We call this output a **solution**. We use the definition of contiguity from [22] (see Fig. 6 in the Appendix for an illustration):

**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 $v$.*

## 3.2 Time-Per-Sample

Our objective is to maximize throughput; equivalently, we minimize its inverse, the **Time-Per-Sample (TPS)**. For pipelined execution schemes such as PipeDream or PipeDream-2BW, it has been argued [22] as well as shown in practice [16] 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 Tarnawski et al. [22]; in that 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.

## 3.3 Tensor Model Parallelism Configurations (TMPCs)

As we remarked in Section 2.5, the task of making a global, joint decision about how to tensor-parallelize every *operator* in a DNN computation graph is very 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 a goal 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 [21]; we use their technique in our experimental evaluation), or the problem can be attempted using automated methods similar to e.g., FlexFlow [12] or Tofu [23] or RL-based approaches. 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.

For each available configuration, Piper gets as input certain attributes related to compute, communication and memory costs for the target model (listed formally in Section 3.6). A tuple containing these attributes is called a *Tensor Model Parallelism Configuration (TMPC)*. Piper receives a (possibly empty) list $T(v,t)$ of TMPCs for each node $v$ and degree $t$ of tensor parallelism (including $t=1$). Memory-saving optimizations (e.g., activation recomputation) are modeled using TMPCs as well.[1]

### 3.4 Data parallelism

Consider a stage that is data-parallelized across $d > 1$ devices. Following PipeDream [16], 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 periodic synchronization of weight gradients, which has a total communication cost of $4(d-1)w$, where $w$ is the size of parameters in all layers in the stage. This gives a cost of $4 \cdot \frac{d-1}{d} \cdot w$ bytes per device (per $d$ microbatches).

### 3.5 Memory usage in pipelined configurations

With pipelining, every stage needs to store some number of weight and 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 with data-parallel degree $d$ and suffix sum of data-parallel degrees $s$, memory usage would be $a \cdot \lceil s/d \rceil + b$ for appropriate coefficients $a$ and $b$. 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 be the size of optimizer state and temporary buffers. PipeDream-2BW uses only two stashed weight versions instead of $\lceil s/d \rceil$ many, so the size of weights would contribute only to $b$ (multiplied by 2), not $a$. If activation recomputation is used, then $a$ accounts for only the sizes of *input* activations of every layer, and $b$ accounts for all other activations. This accounting is more precise than PipeDream's, which ignores memory usage, or Tarnawski et al.'s [22], which uses a constant function.

### 3.6 Input and 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 3.3).

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 start 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:

[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 with activation recomputation; and for $t = 8$, one TMPC corresponding to tensor parallelization in dimension 1 and another using dimension 2, with and without activation recomputation (four in total).

- 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 3.5).

Piper **outputs** the following 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 selected in $T(v, t_i)$.

All nodes 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 dynamic programming on *downsets*, extending recent work [22].

**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 $D' \subseteq D$.*

We rely on the domain insight that layer-granularity computation graphs of modern DNNs do not have many downsets, making tractable the approach of considering all downsets and all contiguous sets using dynamic programming. We define the following **dynamic programming table**: $dp[D][k][s]$ is 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) such 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 3.5) 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 $\min_{k=1,...,K} \min_{s=1,...,N} dp[V][k][s]$ (only use $s = N$ in order to exactly attain the maximum safe batch size).

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

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

where we **define** $\text{TPS}(S, t, d, s)$ 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 $s$. We now describe how we compute this quantity.

**The knapsack subproblem.** How to compute $\text{TPS}(S, t, d, s)$? Given parallelism degrees $d$ and $t$, we need 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

[2] Includes the runtime of both the forward and the backward passes.

memory usage of any combination can be expressed as sums of the TPSes and the memory usages of TMPCs of individual nodes $v \in S$ (more details 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 $+\infty$. This is the only non-exact component of the algorithm; however, we find that this heuristic 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. **Compute** is the simplest: $\text{compute} = \frac{1}{d} \sum_{v \in S} X(v).p$. For **communication**, we have costs stemming from sending activations into $S$, sending activations out of $S$, and costs of all-reductions from using data parallelism (all in bytes):

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

where we used the notation $\delta^-(S)$ and $\delta^+(S)$ for the sets of incoming and outgoing edges of $S$. **Memory usage** is given by: $\text{mem} = \sum_{v \in S} \left( X(v).a \cdot \lceil s/d \rceil + X(v).b \right)$. The final TPS for the selected $S$, $t$, $d$, $s$ and $\{X(v)\}_{v \in S}$ is $\text{compute} + \frac{\text{comm}}{B}$ if $\text{mem} \leq M$; otherwise we return $+\infty$.

**Running time.** Under some natural assumptions, 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.

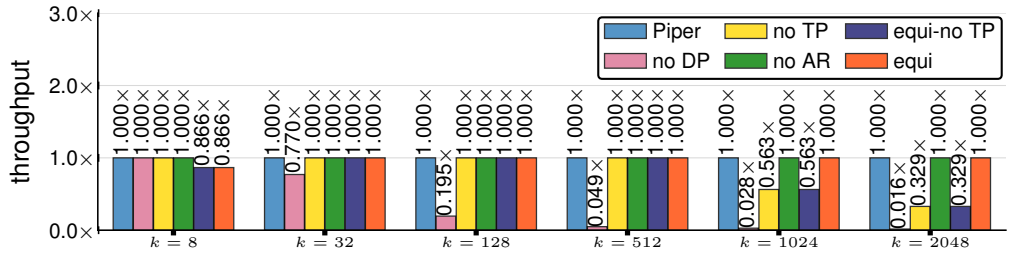
## 5 Evaluation

We now evaluate the Piper algorithm on real-world DNN models. 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-saving optimizations, as well as over best possible *equi-partitioned configurations* such as those found by PipeDream-2BW's planner algorithm. We stress that we focus on evaluating the Piper *partitioning algorithm* in this work, not any particular pipelined DNN training system. Moreover, with a fixed batch size and pipelining scheme, minimizing TPS is equivalent to minimizing time-to-convergence.

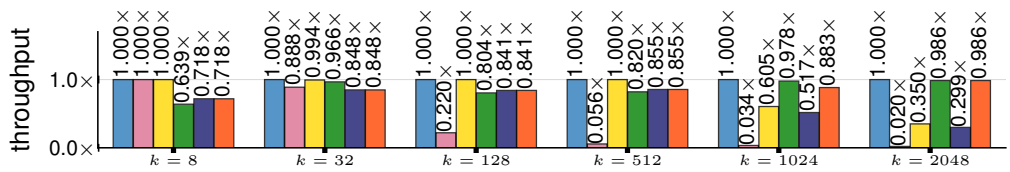
**Inputs.** For our comparisons, we use a BERT-32 model, which consists of an embedding layer, 32 transformer layers and a pooling layer. This model has 5.9 billion parameters, with similar dimensions to the GPT-3 model with 6.7B parameters [2], and does not fit on a single GPU. We provide TMPCs for non-tensor-parallelized ($t = 1$) and tensor-parallelized executions of transformer layers [21] ($t \in \{2, 4, 8\}$), each with and without activation recomputation. These TMPCs are obtained by profiling models implemented in PyTorch on NVidia A100 GPUs interconnected with a 300 GB/s bandwidth NVSwitch within a server, and 25 GB/s across servers. In our evaluation, we assume that all $K \leq 2048$ devices are connected by a flat network topology with a (full bisection) bandwidth of $B = 25$ GB/s (used for all non-tensor-parallelism communication). Our TMPCs model memory footprint assuming the PipeDream-2BW pipelining scheme.

**Baselines.** We compare Piper to:

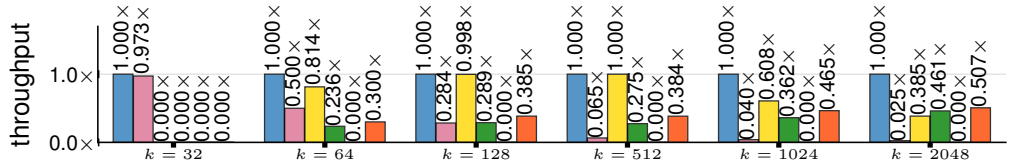
- **no DP**: a version of Piper that forgoes data parallelism (i.e., $d = 1$),



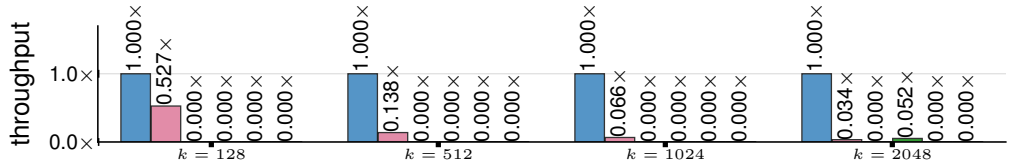

(a) BERT-32, 80GB of memory on each of $k$ devices, $N = 512$.




(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, $N = 512$.

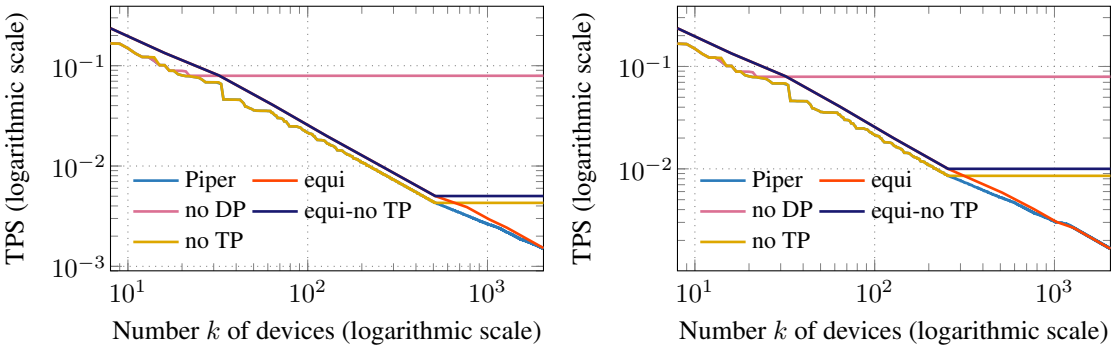
Figure 1: Comparison of Piper to baselines on the basis of throughput normalized to the best configuration. Throughput is the inverse of TPS (Time-Per-Sample); higher throughput is better. Piper always obtains the highest throughput ($1.000\times$). We vary the numbers of devices (from 8 to 2048) and the memory per device, which impacts the space of feasible solutions. For 1GB- or 2GB-devices, baselines are often unable to find *any* memory-feasible configuration ($0.000\times$). 1GB models a scenario where TP and AR are both required to fit even a single layer on a device.

- **no TP**: a version of Piper that forgoes tensor parallelism (i.e., $t = 1$),
- **no AR**: a version of Piper that forgoes activation recomputation,
- **equi**: an algorithm that equi-partitions layers into $w$ equal groups and uses $d$-data parallelism and $t$-tensor parallelism, choosing the lowest-TPS among all valid $(w, d, t)$ triples; it also considers performance with and without activation recomputation (for the entire model), and places the embedding and pooling 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 (i.e., $t = 1$); we show this as a surrogate for PipeDream-2BW's planning algorithm.

Both **no TP** and **no AR** are more sophisticated than PipeDream's partitioning algorithm, which forgoes tensor parallelism, activation recomputation, and also assumes $M = \infty$.

### 5.1 Results

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.




(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 (TPS) for every number of devices and every sum of data-parallel degrees up to $N$ (batch size = $N \cdot$ microbatch size). Here, we show the best TPS for Piper and the baselines versus the number of devices, for two different $N$ values. Lower TPS is better. Tensor parallelism is needed to use more than $N$ devices, as seen by the **no TP** baseline plateauing after $k = N$. **equi** obtains a 10-15% worse TPS than Piper for most $k$.

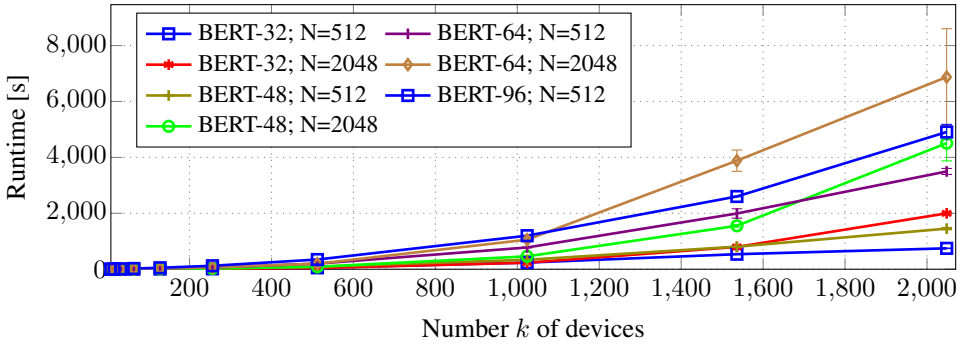



Figure 3: Piper's running time as a function of the number of devices for various DNN model sizes. We show runtimes for different values of $N$ (and consequently batch sizes). Bars show standard deviation. Single-core, unoptimized implementation executed on a mobile Intel i7-8665U CPU.

**Data parallelism (DP).** Data parallelism is crucial, as expected. Without it, scaling to more devices is impossible; 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 essential 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).** Activation recomputation is required to obtain *any* memory-feasible solution in certain cases. 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 in memory. For example, in Fig. 1c, without AR, one is forced to use small values of $s$ to stay within memory, which limits the number of stages and data-parallel degrees, and hence performance (using TP as a replacement yields worse throughput). In scenarios with enough memory relative to DNN size (Fig. 1a), AR is not useful.

**Power of Piper's search space.** Piper has the ability to return configurations with stages that are different from each other (e.g., different numbers of layers, degrees $d$ and $t$, memory optimizations). We found that this leads to a lower optimal TPS than the baseline **equi** even for extremely repetitive DNNs such as BERT (see Figs. 1 and 2). This is useful for two reasons. First, in earlier stages, the parameter $s$ that controls memory usage (Section 3.5) is higher (as $s$ is a "suffix sum" of DP degrees

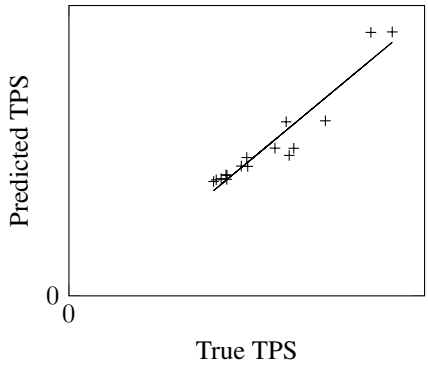



Figure 4: Real vs. Piper-predicted Time-Per-Sample for 16 configurations of pipeline, data, and tensor parallelism for BERT-32 with $K = 64$ devices. Training times were measured on a system with 8 NVidia DGX A100 machines, each with 8 GPUs. The Pearson correlation coefficient is 0.95.

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 will be loose in the last stage, which only stores activations for one microbatch. In contrast, Piper will effectively adjust all stages to ensure that almost all available memory is used on every device. Second, Piper is also able to adjust to different layer types; 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.

**Quality of cost model.** Fig. 4 shows that Piper's cost model is very close to real performance. In this work, we assumed a flat network topology. While this already yields precise estimates, Piper can be extended to handle hierarchical network topologies (see Appendix E).

## 6 Conclusions and Future Work

In this work, we presented a two-level dynamic programming approach that determines how to partition a provided model over a given number of accelerators, while optimally composing tensor and pipeline model parallelism with data parallelism and memory-saving optimizations. In our evaluation, we found that Piper finds high-throughput parallelization strategies that in many cases improve upon planners from prior work, and that Piper's power of finding non-trivial combinations of parallelism modes is especially beneficial in large-scale or memory-constrained scenarios.

While we consider pipeline parallelism schedules without pipeline flushes in this paper, one way of extending Piper to consider schemes with pipeline flushes is by adding the pipeline depth to Piper's dynamic programming state. However, this would worsen Piper's time complexity. We leave such extensions and improvements to the algorithm for future work.

The main open question resulting from this work is automatically finding good tensor-parallelism schemes for each individual layer type (rather than the entire DNN operator graph). Piper can then determine how to best compose these per-layer tensor-parallelism schemes to obtain a high-performance training configuration for the entire model.

## Acknowledgments and Disclosure of Funding

This work was done in the context of Project Fiddle at MSR. During this work, the second author was at Stanford University, where he was supported in part by NSF Graduate Research Fellowship grant DGE-1656518. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors alone.

## 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] LLC Gurobi Optimization. Gurobi Optimizer Reference Manual, 2019.

[10] 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.

[11] 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.

[12] 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.

[13] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, 2012.

[14] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V. Le, and Jeffrey Dean. A Hierarchical Model for Device Placement. *ICLR*, 2018.

[15] 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.

[16] 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.

[17] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-Efficient Pipeline-Parallel DNN Training, 2020.

[18] 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. *arXiv preprint arXiv:2104.04473*, 2021.

[19] NVIDIA and Microsoft. Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, the World's Largest and Most Powerful Generative Language Model, 2021.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] Yanqi Zhou, Sudip Roy, Amirali Abdolrashidi, Daniel Wong, Peter Ma, Qiumin Xu, Hanxiao Liu, Phitchaya Phothilimtha, Shen Wang, Anna Goldie, Azalia Mirhoseini, and James Laudon. Transferable Graph Optimizers for ML Compilers. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 13844–13855. Curran Associates, Inc., 2020.

[25] 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.

| | d | s | s/d | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Stage 1, device 1 | 3 | 6 | 2 | 1 | 1 | 1 | | | | 4 | 4 | 4 | 1 | 1 | 1 | 7 | 7 | 7 | 4 | 4 | 4 | 10 | 10 | 10 | 7 | 7 | 7 | | | | |
| Stage 1, device 2 | 3 | 6 | 2 | | | 2 | 2 | 2 | | | | 5 | 5 | 5 | 2 | 2 | 2 | 8 | 8 | 8 | 5 | 5 | 5 | 11 | 11 | 11 | 8 | | | | |
| Stage 1, device 3 | 3 | 6 | 2 | | | | | 3 | 3 | 3 | | | | 6 | 6 | 6 | 3 | 3 | 3 | 9 | 9 | 9 | 6 | 6 | 6 | 12 | 12 | | | | |
| Stage 2, device 4 | 1 | 3 | 3 | | | | 1 | | 2 | | 3 | 1 | 4 | 2 | 5 | 3 | 6 | 4 | 7 | 5 | 8 | 6 | 9 | 7 | 10 | 8 | 11 | | | | |
| Stage 3, device 5 | 1 | 2 | 2 | | | | | 1 | | 2 | 1 | 3 | 2 | 4 | 3 | 5 | 4 | 6 | 5 | 7 | 6 | 8 | 7 | 9 | 8 | 10 | 9 | | | | |
| Stage 4, device 6 | 1 | 1 | 1 | | | | | | 1 | 1 | 2 | 2 | 3 | 3 | 4 | 4 | 5 | 5 | 6 | 6 | 7 | 7 | 8 | 8 | 9 | 9 | 10 | | | | |

Figure 5: Example of a pipelined schedule (in particular, PipeDream-2BW's schedule), with non-equal data-parallel degrees for different stages. Stage 1 has $d_1 = 3$ and uses devices numbered 1–3; stages 2–4 have $d_2 = d_3 = d_4 = 1$ and use devices numbered 4–6 respectively. Tensor parallelism is not used in this example. The suffix sums of data-parallel degrees ($s$) are shown in the figure. The forward pass, marked in blue, is assumed to take the same time as the backward pass, marked in green. The runtime of a single microbatch in stage 1 is assumed to be 3 times larger than the runtime in each of the stages 2, 3, and 4 (which makes for a balanced schedule since stage 1 has a data-parallel degree of 3). The number $\lceil s/d \rceil$ is the number of in-flight microbatches of every device in steady state. Stashed activations for these microbatches must be kept between their arrival (blue) and departure (green). For example, device 2 maintains stashed activations for microbatch sets $\{2\}, \{2, 5\}, \{5\}, \{5, 8\}, \{8\}, \{8, 11\}$ and $\{11\}$ at different points in time.

# A Batch size and memory usage considerations

**Batch size considerations.** When a pipeline is in steady state, the number of microbatches "in flight" and being processed by some worker in the pipeline is equal to the sum $s = \sum_i d_i$ of data-parallel degrees of all stages. This is because with data parallelism, multiple devices are processing disjoint microbatches (devices process the same microbatch, possibly disjoint shards of it, with tensor parallelism).

In the pipelining schemes that we consider [16, 17], the number of microbatches in a batch is a multiple of $s$. Namely, the total batch size is given as $g \cdot s \cdot b'$, where $b' \geq 1$ is the microbatch size and $g \geq 1$ is the degree of gradient accumulation ($g = 1$ means no additional gradient accumulation).

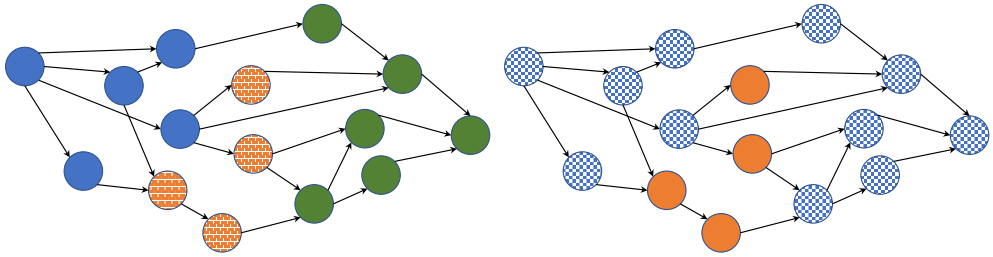
Values of hyperparameters such as $g$ and $b'$ should be chosen before Piper is run; we envision that single-layer runtimes are profiled for multiple $b'$ and then Piper is run for multiple possible settings of $(g, b')$. Usually, a maximum safe batch size $B'$ (verified to not compromise statistical accuracy and model convergence) is known; for large modern language models, this could be on the order of 1024–2048 [19]. Thus, for a fixed $(g, b')$, we require $g \cdot s \cdot b' \leq B'$, and so would set $N = \frac{B'}{g \cdot b'}$ (recall that $N$ is the upper bound for the sum $s$ of data-parallel degrees that Piper will enforce).

Without tensor parallelism, $s$ is always equal to the number of devices used. Therefore, if more devices are available than the number $N$ determined as above, tensor parallelism (and Piper's fine-grained control over the sum $s$ of data-parallel degrees) is required to use all devices. In particular, even if $g = b' = 1$, tensor parallelism is required to use more than $B'$ devices.

**Memory usage.** Consider a stage $j$ (out of $\ell$ total stages). In steady state, the number of microbatches being processed by stages $j, ..., \ell$ is equal to $\sum_{i=j,...,\ell} d_i$ (in other words, $s$ from Piper's DP table for stage $j$). For each in-flight microbatch, we need to store all activations (if using PipeDream-2BW [17] and not using activation recomputation) or just input activations (if using PipeDream-2BW [17] and using activation recomputation) or stashed weights and activations (if using PipeDream [16]). If stage $j$ has data-parallel degree $d_j$, then these activations are partitioned equally among the devices assigned to this stage, which results in each of these devices needing to store $\lceil s/d_j \rceil$ such activation/weight stashes. See Fig. 5 for an example.

Besides this, each stage has other memory costs that do not depend on the number of in-flight microbatches (such as optimizer state, temporary buffers, or two sets of weights if using PipeDream-2BW [17]). Thus, for all these modes of execution, the memory usage can be modeled as

$$\sum_{v \in S} a_v \cdot \lceil s/d_j \rceil + \sum_{v \in S} b_v \,,$$



(a) Contiguous split (b) Non-contiguous split

Figure 6: (a) Contiguous and (b) non-contiguous splits. Note that the brick-patterned orange nodes in (a) form a contiguous subgraph despite not being connected, and checked blue nodes in (b) form a non-contiguous subgraph despite being connected. Figure borrowed from Tarnawski et al. [22].

where $a_v$ and $b_v$ are some coefficients that are functions of the node/layer $v$ and of the TMPC selected for this node. (For instance, if some tensor parallelism scheme partitions the activations among devices, $a_v$ will decrease; if it [also] partitions the weights, $b_v$ will [also] decrease, and so on.)

## B Data flow in tensor-parallelized stages

In this section, we discuss our model for the *edge-related* communication costs in the presence of tensor parallelism. Consider a subgraph/stage that is being tensor-parallelized. For each layer $v \in S$, there are certainly communication costs between the $t$ devices that are incurred in the course of the actual execution of $v$; these costs are modeled in the Time-Per-Sample quantity $X(v).p$ (whose value is obtained via profiling) and this section is not concerned with them. Rather, we are interested in modeling the transfers of tensors that correspond to edges. For simplicity let us concentrate on the forward pass; the backward pass is symmetric.

Let subgraph/stage $S \subseteq V$ be tensor-parallelized on $t > 1$ devices; for each $v \in S$, let $X(v)$ be the TMPC selected for that node/layer.

**Layers finish with full output activations on each device.** Let $v \in S$. We will assume (and require from the schemes that the TMPCs describe) that, once $v$ is done executing, *each of the $t$ devices will hold the same, full, output activations*. This may necessitate adding a final synchronization step between the $t$ devices to the tensor parallelization configuration if, for instance, each device were to end up computing a $1/t$ part of the final output activation, partitioned along some dimension; now these need to be exchanged so that every device holds the entire output activation.

We believe that this assumption is natural, as most tensor parallelization strategies in practice already have this property; in particular, this is true for the Megatron-LM strategy [21] that we use in our evaluations. Furthermore, the assumption allows us to guarantee that our cost model is simple and tractable. Indeed, an alternative might be to allow strategies that leave $1/t$ of the output tensor on each device, which would give benefits in communication if the subsequent layer is also tensor-parallelized among $t$ devices and the first operation in that layer is partitioned in such a way that it could take $1/t$ of the input activation on each device. However, even then, we would not know whether the last operation in the previous layer and the first operation in the next one would have inputs and outputs partitioned *along the same dimension*. In order to know this, we would need to maintain additional state (e.g., the shape of the input and output tensors of each Tensor Model Parallelism Configuration of each layer), and only select compatible configurations of TMPCs in a stage, which would be computationally costly and would require replacing the efficient heuristic for the knapsack subproblem with a more involved procedure.[3]

**Data flow over edges.** Now, consider an edge incoming to $v$ from some other node $u$. Which devices should send outputs to the $t$ devices that hold $v$? There are optimization/scheduling-type

[3] Of course, if it turns out to be the case that two consecutive layers can be parallelized in such an advantageous way, then the ML compiler can still make a local optimization to perform communication more efficiently in the runtime; we simply do not model such possible gains in our cost model.

decisions to be made here. If, for instance, $u$ is also parallelized on some number $k \geq t$ devices, then it would make sense to send activations in a matching-like manner (e.g. if $k = t = 3$ and $u$ is processed on devices 1–3 while $v$ is processed on devices 4–6, then one could communicate activations like $1 \to 4$, $2 \to 5$, $3 \to 6$). We can certainly do this in the runtime; however, to make the problem structure tractable for dynamic programming, we proceed somewhat differently in the cost model. There are two cases depending on whether $u$ is also in the same subgraph $S$ as $v$, or not:

- **Case $u \notin S$:** We cannot afford to know (keep in the dynamic programming state) information about how many devices the subgraph where $u$ lies is partitioned on. Thus we must assume the worst case, which is that only one device holds the outputs of $u$. Now, we could have this single device send the outputs to each of the $t$ devices of $v$, but this would be difficult to account for when computing the cost of $u$'s subgraph, as we cannot [afford to] know $t$ at that time. (Of course, if this is the optimal mode of communication, then we can use it in the runtime; this gain will just not be reflected in the cost model.) What we do instead is:
  - If each of the $t$ devices needs the entire input activation, then we will have that single $u$-device send the output to the first of the $t$ devices, and then let them distribute those activations among themselves, so that they begin the actual processing each having the entire input activation. The TMPC should account for this synchronization by means of a communication cost $c^{\mathsf{fw}}(u, v)$ (in bytes) for each incoming edge that each of the $t$ devices will pay.
  - An exception: if each device only needs a $1/t$ fraction of that activation (e.g., because the first operation is already parallelized along an input-activation-splitting dimension), then the $u$-device sends $1/t$ of that activation to each device; in total, it will have sent one activation, so our prior accounting for the $u$-device's communication cost (one output tensor sent) was correct. In this case, the TMPC can have $c^{\mathsf{fw}}(u, v) = 0$ (when proposing a tensor parallelization where this input activation is only needed in shards).
  - For the special case of $t = 1$ (non-tensor-parallelized execution), we have $c^{\mathsf{fw}}(u, v) = 0$ as there are no communication costs in this case.
- **Case $u \in S$:** In this case, $u$ is tensor-partitioned among *the same* $t$ devices as $v$. Recall that we assumed above that after $u$ is done executing, each of these devices holds the entire output activation. Thus, in this case, nothing needs to be sent. (The TMPC will contain some $c^{\mathsf{fw}}(u, v)$ value, as it does not "know" whether $u \in S$ or not, but the algorithm will not use this value.)

For the backward pass, we proceed in the same way, but we call the relevant quantity $c^{\mathsf{bw}}(v, w)$, and it pertains to the *outgoing* edges of $v$. Note that in the backward pass, the communication on this edge goes in the reverse direction $w \to v$. That is, $c^{\mathsf{bw}}(v, w)$ is the additional communication cost that needs to be paid by $v$ after having received the backward tensor from $w$ on one device, in order to have this tensor on each device.

# C Optimality gap of the knapsack heuristic

Let us remark that the knapsack heuristic is the only non-exact component of Piper. That is, if the heuristic did in fact yield always optimal solutions, then Piper would be an exact algorithm (that is, one finding optimal solutions) for the problem of finding the lowest-TPS global configuration in Piper's search space, subject to our cost model. Note that this is a sufficient condition, but far from necessary; for example, it would be enough that the knapsack heuristic be exact on the $O(|V|)$ instances $(S, t, d, s)$ that are part of some optimal global solution. Then, since the heuristic cannot *under*estimate the TPS of any (other) configuration, Piper would find that solution (or some other optimal solution).

Moreover, we investigated whether the knapsack heuristic does in fact return suboptimal solutions in a practical workload. To this end, we extracted a random subset of around 27000 knapsack subproblem instances that arose when running Piper on our BERT-32 workload from Section 5 (8 GB RAM per GPU, $K = 512$ devices, $N = 512$). We then solved them to optimality using the Gurobi 8.1 [9]

Integer Programming solver. We have determined that the heuristic indeed found the optimal solution in all instances.[4]

Finally, with minor modifications, one could also prove theoretical approximation guarantees for the heuristic (similar to Csirik et al. [5], for example).

# D Running time analysis

Let us denote by $P$ the number of $t$-values for which there are TMPCs in the input; that is, $P := |\{t = 1, ..., K : (\exists v \in V)\ T(v,t) \neq \emptyset\}|$. Next, let us denote by $T$ the maximum number of TMPCs for one $(v,t)$ pair: $T := \max_{v,t} |T(v,t)|$.

We can implement Piper as follows. An outer loop iterates over downsets $D$ and then over sub-downsets $D'$: this gives $O(\mathcal{D}^2)$ iterations, where $\mathcal{D}$ is the number of downsets in $G$. Fix $D$ and $D'$ and define $S := D \setminus D'$.

We will first compute all $\mathrm{TPS}(S, t, d, s)$ values (for all $t$, $d$, $s$), and then use these values to update the $dp$ table. We first focus on the former.

We start by precomputing, for all $v \in S$, $t$ and TMPC $X \in T(v,t)$, the contribution of $X$ (if selected) to TPS and to memory usage, the latter in the form of quantities $a$ and $b$ to be used in the memory usage formula (see Section 3.5). We take all communication (except all-reductions for weight gradients when using data parallelism) into account here (in particular, the communication over the boundary edges of $S$). Note that we can perform such a precomputation as both TPS and memory usage are additive over nodes $v \in S$. Note also that this precomputation is independent of parameters $s$ and $d$. For fixed $S$, this step takes $O(P \cdot T \cdot (|S| + |E|))$ time, as we look at every edge in $E$ at most twice (once for each endpoint that it has in $S$).

Then, we iterate over all degrees $t$ and $d$ of tensor and data parallelism, respectively; as we are using $t \cdot d \leq K$ devices here, $d$ ranges from 1 to $\lfloor K/t \rfloor$. For each $(t,d)$, we need to solve one knapsack subproblem instance per $s$ (to compute $\mathrm{TPS}(S, t, d, s)$; recall that $s$ ranges from 1 to $N$). Here we can take advantage of the fact that memory usage only depends on $s$ through $\lceil s/d \rceil$, which means that it does not change within groups $s \in \{1, ..., d\}$, $s \in \{d+1, ..., 2d\}$, $s \in \{2d+1, ..., 3d\}$, and so on. Therefore there are only $\lceil N/d \rceil$ distinct knapsack subproblem instances that we need to solve. Finally, the knapsack heuristic can be implemented to run in time $O(|S|T^2 \log |S|)$. The total running time of this step (for fixed $S$) is thus

$$\sum_{t=1}^{P} \sum_{d=1}^{\lfloor K/t \rfloor} \lceil N/d \rceil \cdot |S| T^2 \log |S| \,.$$

As $\sum_{d=1}^{\lfloor K/t \rfloor} 1/d = O(\log(K/t))$, basic estimations give that $\sum_{t=1}^{P} \sum_{d=1}^{\lfloor K/t \rfloor} 1/d = O(P \log(K/P))$. Thus the running time of this step can be written as $O(P \log(K/P) N |S| T^2 \log |S|)$.

Finally we consider updating the $dp$ table. We loop $t = 1, ..., K$ (but at most $P$ distinct values of $t$) and $d = 1, ..., K/t$ (as we must have $t \cdot d \leq K$). There are at most $K + K/2 + K/3 + ... + K/P = O(K \log P)$ such pairs $(t,d)$. We also need to iterate over $k = t \cdot d, ..., K$ and $s = d, ..., N$ (as $t \cdot d \leq k$ and $d \leq s$), which gives $O(KN)$ pairs $(k,s)$. Then we perform the update of $dp[D][k][s]$ with $\max(dp[D'][k - t \cdot d][s - d], \mathrm{TPS}(D \setminus D', t, d, s))$, which takes constant time. Overall this step takes time $O(K^2 N \log P)$.

Taken together, we get a time complexity of

$$O\left(\mathcal{D}^2 \cdot \left(P \cdot T \cdot (|V| + |E|) + P \log(K/P) N |V| T^2 \log |V| + K^2 N \log P\right)\right) .$$

**Simplification.** In practice, we expect $\mathcal{D} = O(|V|)$, $|E| = O(|V|)$, and $T, P = O(1)$. We also use the $\tilde{O}$-notation to suppress log factors. We then finally get a runtime of

$$\tilde{O}(|V|^2 N(|V| + K^2)) \,.$$

We also expect to usually have $|V| \leq K^2$, which yields $\tilde{O}(|V|^2 N K^2)$.

[4] We do note that in our evaluation, we always have $|T(v,t)| \leq 2$ for all $(v,t)$, and most layers $v$ are very similar (transformer layers). This might be making the knapsack subproblem easier. At the same time, we expect these properties to hold for many real-world workloads.

# E Hierarchical network topologies

We have thus far assumed a network model where bandwidth constraints are placed on the incoming/outgoing communication of each device – that is, a flat network topology (one can think that these devices are all connected with an infinite-bandwidth switch). However, it is possible to extend Piper to a hierarchical model where we are given some $K_L$ devices ("superdevices") of level $L$, each of which consists of $K_{L-1}$ devices of level $L-1$, and so on. The devices at each level $\ell = 1, ..., L$ (that are part of the same level-$(\ell+1)$ device, if $\ell < L$) are interconnected with a network with bandwidth $B_\ell$. As an example, the infrastructure used for Fig. 4 was 8 NVidia DGX-A100 machines, interconnected (between machines) with a 25 GB/s Infiniband, and each containing 8 A100 GPUs, connected (inside the machine) with a 300 GB/s NVSwitch. The GPUs are level-1 devices, and DGX machines are level-2 devices. This corresponds to having $L = 2$, $K_1 = K_2 = 8$, $B_1 = 300$ GB/s and $B_2 = 25$ GB/s.

We refer to the dynamic programming algorithm from the main body of the paper as *single-level Piper*.

At a high level, the main technique needed to extend the dynamic program to multiple levels involves computing the optimal TPS for every number of devices *of some level*, not only for every downset, but for every contiguous set of nodes (that is, for every difference of two downsets).

The bottleneck of the system in such a hierarchical setting is either a level-1 device (whose load consists of compute and communication) or a higher-level device (whose load consists solely of communication).

**Basic version.** Let us first ignore tensor parallelism, memory usage, and batch size. Without these, we are essentially in the setting of PipeDream's partitioning algorithm [16], albeit with an arbitrary DAG rather than a path-graph of layers. We compute values

$$dp^{\ell}[D, D'][k] = \text{best TPS (max-load) achievable by placing } D \setminus D' \text{ on } k \text{ devices of level } \ell.$$

The final result will then be $\max_{k=1,...,K} dp^{L}[V, \emptyset][k]$.

There are two kinds of communication to account for:

- Communication over edges: note that an edge that crosses multiple levels needs to be accounted for at each level. One might worry that in the dynamic program, when considering a level $\ell$, a subgraph $D \setminus D'$ and an edge coming into it "from afar", we do not know (from the DP state) whether the other endpoint of that edge is placed on the same level-$\ell$ (or level-$(\ell+1)$, level-$(\ell+2)$, etc.) device as $D \setminus D'$. However, this is in fact not an issue: with this network model, we only need to account for the boundaries of the level-$\ell$ devices that we are currently forming, and we know that in this way, this edge will be accounted for at the boundary of each level that it actually crosses.

- All-reduce communication between data-parallel replicas: we can form stages that are replicated across multiple levels (e.g., level 1 and then again on level 2). We assume that in this case, the all-reduce implementation (say, a distributed parameter server approach) is cognizant of the hierarchical structure: for instance, the weights will be synchronized among layer-1 devices within the same level-2 device, and then among layer-2 devices (and so on, if there are more levels). Thus, here as well, at each level we are able to account for the data-parallel communication happening over that level's boundary. We also do not need to know how many devices are on the lower levels.

With this, we can write the DP recursion, in which we choose the boundary $D^\circ$ of the last stage[5] and its data-parallel degree $d$:

$$dp^\ell[D, D'][k] = \min_{\substack{D^\circ:\text{ downset} \\ D' \subseteq D^\circ \subsetneq D}} \min_{d=1,\ldots,k} \max\Big(dp^\ell[D^\circ, D'][k-d], \frac{1}{d}\left[\frac{4 \cdot \sum_{v \in D \setminus D^\circ} w_v \cdot \frac{d-1}{d} + 2 \cdot \sum_{(u,v) \in \delta(D \setminus D^\circ)} c(u,v)}{B_\ell}\right], \frac{1}{d} \cdot dp^{\ell-1}[D, D^\circ][K_{\ell-1}]\Big) .$$

Here:

- $w_v$ is the size of weights associated with node $v$ (in single-level Piper, this information is kept in TMPCs, but we are ignoring tensor parallelism so far).
- $\delta(S)$ is the set of all edges crossing $S$.
- For $D^\circ = D'$, we have $dp^\ell[D^\circ, D'][k] = 0$ (for any $k$, $d$) as we are partitioning an empty set $D^\circ \setminus D' = \emptyset$ of nodes.
- The recursion arises as follows. We are using $d$ level-$\ell$ devices, in a data-parallel fashion, for the node set $D \setminus D^\circ$, and $k - d$ level-$\ell$ devices (recursively) for the remaining node set $D^\circ \setminus D'$. Then, the bottleneck (device attaining the maximum load) could be either
  - one of the $k - d$ level-$\ell$ devices or one of the lower-level devices inside them (this is the first $dp^\ell$ term),
  - one of the $d$ level-$\ell$ devices; recall from the above discussion that for these, we only look at communication, which is of two kinds: the first term corresponds to data parallel resync costs and the second term corresponds to communication over edges (here we account for the cost of bringing these activations one level down),
  - one of the lower-level devices inside the $d$ level-$\ell$ devices (this is the $dp^{\ell-1}$ term); we recursively partition $D \setminus D^\circ$ on all $M_{\ell-1}$ many level-$(\ell-1)$ devices inside each level-$\ell$ device, and we account for the data-parallelism speedup with the $\frac{1}{d}$ term.
- If $\ell = 1$, then instead of the last $dp^{\ell-1}$ term, we should account for compute costs, as in single-level Piper. We take the compute costs as $\sum_{v \in D \setminus D^\circ} p_v$, where $p_v$ is the latency of node $v$ (in single-level Piper, this information is kept in TMPCs), and we use the sum of this with the communication-related (second) term.

**Running time.** If we first precompute all sums of $w$, $c$ and $p$ values for all contiguous sets, which takes $O(\mathcal{D}^2 \cdot (|V|+|E|))$ time, then filling the $dp$ table on level $\ell$ requires computing $O(\mathcal{D}^2 K_\ell)$ values, each taking time $O(\mathcal{D} K_\ell)$, for a total time complexity of $O(\mathcal{D}^3 \sum_{\ell=1}^{L} K_\ell^2)$. If $\mathcal{D}, |E| = O(|V|)$, then we get a total time of $O(|V|^3 K^2)$.

**An improvement.** Note that, for the highest level, we in fact only need to compute $dp^L[D, D']$ for $D' = \emptyset$ (as we are using the same $D'$ for the recursive term $dp^L[D^\circ, D'][k-d]$). This can be very helpful if e.g. $L = 2$ and $K_2 \gg K_1$, as the time complexity will have terms $\mathcal{D}^{\mathbf{2}} K_2^2 + \mathcal{D}^3 K_1^2$ (rather than $\mathcal{D}^3 K_2^2$).

**Crude memory accounting.** We note that the above "basic version" can be augmented with an approximate memory accounting: namely, at level 1, we only consider forming a stage if its memory usage is at most $M$, and memory usage is counted as in Section 3.5, but we use $K$ as an upper bound for $s$. That is, for a subgraph $S$ and data-parallel degree $d$ we want to have

$$\left(\sum_{v \in S} a_v\right) \cdot \lceil K/d \rceil + \sum_{v \in S} b_v \le M ,$$

where $a$, $b$ are the memory usage quantities that in single-level Piper are stored in TMPCs. This approach is somewhat akin to ignoring the fact that later stages have lower memory usage since they need to store fewer activations.

[5] We are forming a level-$\ell$ "superstage", which might be split further at lower levels.

**Precise memory accounting and batch size considerations.** Now we would like to extend the above "basic version". Recall that in single-level Piper, these are controlled by a parameter $s$, which is a suffix sum of data-parallel degrees over stages. Let us remark that now, for a "superstage" that is replicated over more than one level, we should take the product of its data-parallel degrees.

We start with a rather heavy-handed extension of the above dynamic program to take the parameter $s$ into account. Consider, as previously, forming a new stage for the subgraph $D \setminus D^\circ$. To correctly estimate the memory usage of this (super)stage, we need to know the sum of data-parallel degrees from this stage to the last stage (not just the last stage of the split of $D \setminus D^\circ$, but the last stage of the split of $D$). Also, the recursive call for $D^\circ \setminus D'$ will need to know its similar sum of data-parallel degrees, but we would not know this unless we also control the sum of data-parallel degrees in the split of $D \setminus D^\circ$ itself. This means that unfortunately we need to use another parameter, which we will denote by $x$.

Namely, we can compute

$$\begin{aligned} dp^\ell[D, D'][k][s, x] = {} & \text{best TPS (max-load) achievable by placing } D \setminus D' \text{ on } k \text{ devices of level } \ell \\ & \text{assuming that the sum of data-parallel degrees in } D\text{'s split is } s \\ & \text{and that the sum of data-parallel degrees in } (D \setminus D')\text{'s split is } x. \end{aligned}$$

Now, in the DP recursion, we will also select a sum $x^\circ$ of data-parallel degrees for the new stage (for $D \setminus D^\circ$). Note that $x^\circ$ must be divisible by $d$. Then, the sum of degrees for $(D^\circ \setminus D')$'s split will become $x - x^\circ$, and the sum of degrees for $D^\circ$'s split will become $s - x^\circ$. The lower-level split (of $D \setminus D^\circ$) needs to have sum of degrees $x^\circ/d$. Moreover, since this is replicated $d$-wise, the effect of the $s$ microbatches waiting in the pipeline on memory usage can be obtained by dividing by $d$ (taking the ceiling; similarly as in Appendix A). Let us write the full formula:

$$\begin{aligned} dp^\ell[D, D'][k][s, x] = \min_{\substack{D^\circ:\text{ downset} \\ D' \subseteq D^\circ \subsetneq D}} \min_{d=1,\ldots,k} \min_{\substack{x^\circ = d, 2d, \ldots \\ x^\circ \leq \min(x, d \cdot K_{\ell-1})}} & \max\Big(dp^\ell[D^\circ, D'][k-d][s - x^\circ, x - x^\circ], \\ & \frac{1}{d}\left[\frac{4 \cdot \sum_{v \in D \setminus D^\circ} w_v \cdot \frac{d-1}{d} + 2 \cdot \sum_{(u,v) \in \delta(D \setminus D^\circ)} c(u,v)}{B_\ell}\right], \\ & \frac{1}{d} \cdot dp^{\ell-1}[D, D^\circ][K_{\ell-1}][\lceil s/d \rceil, x^\circ/d]\Big) \ . \end{aligned}$$

As previously, we adjust the recursive relations for $\ell = 1$ appropriately (using the parameter $s$ to account for memory usage, as in single-level Piper, and ensuring that the parameter $x$ is equal to the sum of data-parallel degrees of the split).

**Improvements.** The proliferation of parameters leads to a running time that is still polynomial (as long as $\mathcal{D} = \text{poly}(|V|)$), but of a very high degree, and most likely no longer practical. However, we can again optimize for the typical scenario we are targeting, where $L = 2$ and $K_1$ is small. To that end, notice that for $\ell = L$:

- we still only need to consider $D' = \emptyset$,
- and then, the parameter $x$ is unnecessary, as in this case, $D \setminus D' = D$ and so we must have $x = s$.

If $L = 2$, then only level $\ell = 1$ remains, and if $K_1$ is small (e.g., $K_1 = 8$), the above dynamic program should be feasible to run.

We note that another possible, albeit lossy, optimization is to upper-bound $s$ by $s \leq N$ whenever counting memory usage, similar to the approach proposed above for "crude memory accounting"; then the parameter $s$ can be removed, but we still control the sum of data-parallel degrees with the parameter $x \leq N$.[6]

[6]Of course, there is no reason to control the sum of data-parallel degrees unless we are using tensor parallelism, since absent tensor parallelism, this sum is always equal to the total number of devices $K = \prod_{\ell=1}^{L} K_\ell$.

**Tensor parallelism.** Finally, we extend the above with tensor parallelism. Following recent work [17, 18], we focus on tensor parallelization within a single level-2 device (e.g., an NVidia DGX machine), as very fast communication is necessary for tensor parallelization techniques to be efficient. This means that we only need to consider tensor parallelism for $\ell = 1$, and thus level 1 essentially becomes single-level Piper, but

- extended by computing the best configuration for every contiguous set $D \setminus D'$ (rather than every downset $D$), as above,
- extended by including the parameter $x$, as above,
- but executed for a smaller number $K_1 \ll K$ of devices.

# F Further evaluation results

In Table 1, we present a comparison of Piper to the **equi** baseline for two other DNNs models, albeit without tensor parallelism. These results demonstrate that equi-partitioning often fails to recover high-throughput partitionings when compared to more involved (e.g., dynamic programming based) partitioning algorithms such as Piper or other recent work [16, 22].

| **Number of devices ($K$)** | **Memory per device (GB)** | **Piper (TPS)** | **equi (TPS)** | **equi normalized to Piper (throughput)** |
|---|---|---|---|---|
| GNMT | | | | |
| 2 | 2.5 | 263.354 | 325.026 | 0.810x |
| 2 | 3.5 | 260.268 | 260.268 | 1.000x |
| 4 | 1.2 | 148.848 | OOM | 0.000x |
| 4 | 2.5 | 131.702 | 162.533 | 0.810x |
| 8 | 1.2 | 69.5705 | 137.518 | 0.505x |
| 8 | 2.4 | 65.1633 | 81.2712 | 0.801x |
| 16 | 1.2 | 33.4478 | 55.8757 | 0.598x |
| 32 | 0.8 | 17.0167 | 71.1943 | 0.239x |
| Resnet50 | | | | |
| 8 | 4 | 74.2513 | OOM | 0.000x |
| 8 | 8 | 63.4363 | 164.159 | 0.386x |
| 8 | 16 | 58.5656 | 108.886 | 0.537x |
| 16 | 2 | 76.5213 | OOM | 0.000x |
| 16 | 4 | 35.4642 | OOM | 0.000x |
| 16 | 8 | 31.636 | 82.0797 | 0.385x |
| 16 | 16 | 29.2406 | 54.4429 | 0.537x |
| 32 | 1.5 | 61.2086 | OOM | 0.000x |
| 32 | 2 | 26.4449 | OOM | 0.000x |
| 32 | 4 | 17.5978 | OOM | 0.000x |
| 32 | 8 | 15.8181 | 41.0398 | 0.385x |
| 32 | 16 | 14.5413 | 27.2214 | 0.534x |
| 64 | 1.5 | 16.9921 | OOM | 0.000x |
| 64 | 2 | 11.5094 | OOM | 0.000x |
| 64 | 4 | 8.64699 | OOM | 0.000x |
| 64 | 4 | 8.64699 | OOM | 0.000x |
| 64 | 8 | 7.83803 | 20.5199 | 0.381x |
| 64 | 16 | 7.27067 | 13.6107 | 0.534x |
| 128 | 1 | 19.8685 | OOM | 0.000x |
| 128 | 1.5 | 7.02514 | OOM | 0.000x |
| 128 | 2 | 5.50138 | OOM | 0.000x |
| 128 | 4 | 4.29729 | OOM | 0.000x |
| 128 | 8 | 3.90292 | 10.26 | 0.380x |
| 128 | 16 | 3.63534 | 6.80536 | 0.534x |

Table 1: Results of comparing Piper to the baseline **equi** on two further DNN models, GNMT and Resnet50, without tensor parallelism. We use profiles (on an nVidia GTX 1080Ti GPU) from prior work [22]. For both, we use multiple settings for the number of devices and the memory limit. For GNMT, the throughput of **equi** was on average 68% of that of Piper, and in one scenario **equi** could not find any memory-feasible solution while Piper was able to. For Resnet50, the throughput of **equi** was on average 46% of that of Piper, and we found more scenarios where **equi** could not find any feasible solution.