Parmesan: Efficient Partitioning and Mapping Flow for DNN Training on General Device Topology

Anonymous Author(s)
Affiliation
Address
email

Abstract

Recently, various pipeline parallelism strategies are proposed to tackle the scalability problem of training a large DNN model on a distributed system. However, most of the works focus on pipeline scheduling while lacking a general methodology to handle network partitioning and mapping to distributed systems with heterogeneous interconnection. In this work, we propose an efficient optimization flow, named *Parmesan*, to maximize the throughput of training a general DNN on a system with general device topology. Parmesan works in an end-to-end manner and solves the whole optimization problem in two phases. The first phase aims at producing well-balanced partitions, and the second phase works towards placing the DNN on devices connected by an arbitrary topology network, considering the heterogeneity of the interconnection bandwidth. We show that Parmesan speeds up the pipeline training throughput on two systems with different GPU topologies and is able to handle the mapping problem for the heterogeneously interconnected architectures.

1 Introduction

The ever-increasing Deep Neural Network (DNN) model size [2, 4] has stimulated the development of distributive learning scheme in pursuit of more efficient large-scale DNN training and serving. In response to such surging demands, both researchers and industrial practitioners have been actively exploring dedicated model parallel schemes, with the objectives of improving (1) model throughput over the given device instances; (2) flow robustness in terms of network and device topology coverage; and (3) turnaround efficacy for searching a desirable parallelism strategy.

However, designing robust and high-performance distribution strategies for different neural network architectures over different device topologies is non-trivial and challenging, since it is essentially seeking a favourable solution (with high throughput) within an astronomically large search space brought by the growing model size and complex device interconnects. Various paradigms, including but not limited to data parallelism [7, 13, 27], model parallelism [3, 11, 24] and pipeline parallelism [5, 10, 14, 15, 19, 25], have been extensively studied to enable decent parallel execution.

As one of the most prevailing techniques, pipeline parallelism mainly covers three types of optimization problems. (1) *Model Partitioning:* model parameters and associated activations are partitioned into a set of stages, where the workload of each stage is deployed on an individual device. This step aims to balance the workloads among all stages as well as to maximize the overall throughput. (2) *Device Mapping:* partitioned stages are physically placed on the devices with consideration of the hardware constraints, network topology and communication bandwidths. This step affects network traffic and computing resource utilization which turns out to be crucial for large-scale DNN training. (3) *Pipeline Scheduling:* this step schedules the pipelined stages for computation and communication considering other scheduling factors like pipeline flush, weight buffering and update mechanisms,

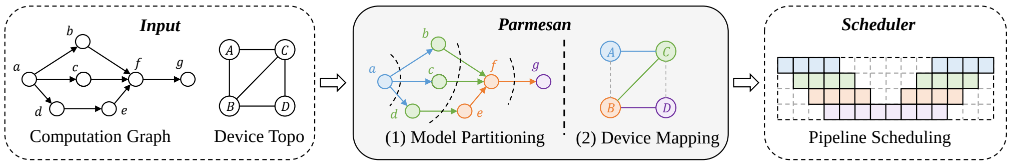


Figure 1: The workflow of optimizing the pipelined training throughput of using Parmesan.

etc., with the objective of finding the best tradeoff among device-utilization, memory footprint and training convergence (for asynchronous training).

Recent pipeline scheduling literature [10, 5, 19, 14] already demonstrates its superiority in accelerating DNN training. Orthogonal to previous works that focus on improving pipeline scheduling, balanced and properly mapped partitions are also important to maximizing training throughput. Although some existing works [25, 19, 26, 5] explore mechanisms such as dynamic programming-based model partitioning and heuristic-based device mapping, they only consider layer-level graphs and flattened/hierarchical topologies without considering the potential issues behind them.

Layer-level partitioning lacks flexibility and needs pre-processing. Layer-level partitioning relies on the assumption that the number of layers is larger than the number of stages. However, this may not be true in some cases. For example, some specialized hardware has many cores while limited memory for each core, so a large number of partitions are required to fully utilize the cores. In such cases, high flexibility can only be provided at the operator-level (op-level). Besides, since the intermediate representation (IR) graph in modern DNN compilers/frameworks is at op-level instead of layer-level, a non-trivial pre-processing is needed to generate the layer-level graph from op-level. Moreover, such pre-processing varies among different DNNs, which brings extra development effort. Hence, balancing the workloads among all stages and effectively handling the op-level graph should be considered when developing a DNN partitioner.

Heuristic-based device mapping cannot guarantee optimality and cannot be generalized. The most commonly-used heuristic for device mapping is to put the consecutive stages on consecutive devices (we call it consecutive mapping). Although consecutive mapping works well in some cases, it cannot guarantee optimality. We demonstrate this point using a simple example of mapping four stages on a 2×2 hierarchical topology, shown in Figure 2. The total communication latency of consecutive mapping is around $1.4\times$ larger than the optimal solution. Admittedly, one can develop some heuristics for device mapping to produce a near-optimal performance on the commonly-used hierarchical GPU servers. However, dedicated hardware other than GPUs for DNN training also exists and emerges that is of very different networking topologies or even integrates multiple processors and interconnections on a single chip (e.g., Google TPU, Cerebras CS-1). Existing heuristics may struggle in these scenarios, and new heuristics (if there exists a good one) need to be invented with effort. Therefore, an ideal device mapping algorithm should be able to solve the general device topology mapping problem optimally. Such a general topology mapping algorithm can also assist architecture designers in estimating their hardware performance during the design loop, and enable them to design a more DNN-friendly hardware architecture.

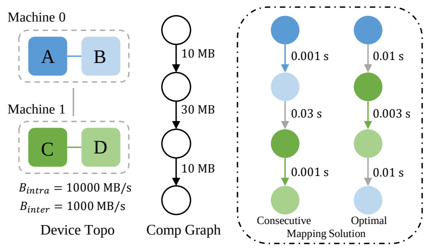


Figure 2: An example to show that consecutive mapping cannot guarantee optimality.

In the light of the above, in this paper, we proposed Parmesan, a robust and efficient middleware for large-scale pipeline training based on the PyTorch infrastructure. Given an *arbitrary* operator-level DNN (expressed as a directed acyclic graph) and general device topology as inputs, Parmesan automatically optimizes the pipeline training throughput in an end-to-end manner.

To the best of our knowledge, Parmesan is the first work to formulate general device mapping problem for pipeline parallelism. Our device mapping formulation considers general device topology, i.e., arbitrary topology with heterogeneous interconnect bandwidths, which can be proved as an NP-complete problem. To make the above challenges solvable, Parmesan decouples the model partitioning and device mapping problems, resulting in a two-phase optimization engine as depicted in Figure 1. In the model partitioning stage, Parmesan performs an operator-level optimization based on dynamic programming and two well-designed techniques, considering the computation and communication overheads due to the device hardware constraints. In the device mapping stage, Parmesan conducts customized searching with effective pruning strategy to find out the optimal placement solution efficiently. Different from previous methods, the proposed device mapping mechanism is capable of handling general topology with heterogeneous interconnect bandwidths. Our empirical evaluations on real-world training and simulations on non-hierarchical topologies indicate the robustness and the effectiveness of Parmesan.

2 Related Work

Pipeline Parallelism. Pipeline parallelism aims at scheduling DNN training more elaborately and boosting resource utilization. Extending from model parallelism, pipeline parallelism not only partitions a DNN to different devices but also divides a mini-batch into several micro-batches. In pipeline parallelism, the split micro-batches and the partitioned DNN will be delicately scheduled, resulting in an increase in throughput. GPipe [10] synchronously schedules the forward and backward propagation during training. DAPPLE [5] introduces a hybrid parallelism strategy (i.e., pipeline training combined with data parallelism) with an one-forward-one-backward synchronous pipeline. PipeDream [19, 20] generalizes the pipeline training to an asynchronous fashion and reduces the idling time (bubble) while sacrificing the model accuracy due to weight staling. Chimera [14] further extends the synchronous pipeline to a bidirectional pipeline and achieves impressive throughput.

DNN Partitioning and Mapping. Some works [18, 6, 17, 1, 22] partition and map a DNN with reinforcement learning but require time-consuming online measurement of throughput. PipeDream [19], Piper [26], DAPPLE [5], and RaNNC [25] develop dynamic-programming-based partitioning algorithms to optimize the training. However, PipeDream, Piper and DAPPLE only focus on coarse-grained layer-level granularity. As for device mapping, RaNNC only considers flattened device topologies of constant bandwidth. PipeDream and Piper assume that the network topologies are hierarchical and exploit dynamic programming to handle the problem. However, they do not explicitly provide an algorithm to map DNN stages to devices. DAPPLE proposes three heuristic-based policies to perform device mapping while lacking guarantee on the solution quality and optimality.

3 Overview of Parmesan

Given a neural network (NN) and a distributed system of GPU devices with heterogeneous interconnect bandwidths, our goal is to maximize the throughput of *hybrid* parallel training for this NN. We use the term *hybrid* to denote pipelined training combined with data parallelism. Since the throughput can be affected by many factors (e.g., how to partition the NN, how to map the partitions onto GPU devices) and the overall optimization problem is over-complicated, we decouple the problem into two phases, namely model partitioning (in Section 4) and device mapping (in Section 5). The detailed discussion of decoupling the problem into two phases is given in Section 6.

Besides model partitioning and device mapping, Parmesan also provides a unified operator-level graph extractor, profiler, deployment, and evaluation API. A task-graph-based simulator is also incorporated. Due to the limitation of the space, we leave the details in Appendix H.

4 Operator-Level Model Partitioning

Given an NN represented as a Directed Acyclic Graph (DAG), $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ denotes the set of operators (e.g. convolutions, additions, etc.) and $\mathcal{E}$ denotes the set of operator dependencies, the objective of our model partitioning algorithm is to find a partitioning solution to maximize the throughput of a pipeline training. Note that the objective is equivalent to finding a solution to minimize the maximum stage time [19], given by

$$\max_{\mathcal{S}} \text{ throughput} = \min_{\mathcal{S}} \max_{s \in \mathcal{S}} \{c_{\mathsf{u}}(s) + c_{\mathsf{m}}(s)\} \tag{1}$$

where $c_{\mathsf{u}}(s)$ and $c_{\mathsf{m}}(s)$ denote the computation time (including forward and backward execution time) and the communication time of stage s respectively, and $\mathcal{S}$ denotes a set of stages (i.e. a partitioning solution). Each stage $s \in \mathcal{S}$ contains a group of consecutive operators from $\mathcal{V}$.

The solution of model partitioning will be a set of disjoint stages $\mathcal{S}^*$ while satisfying: (a) $\mathcal{V} = \bigcup_{s \in \mathcal{S}^*} s$, (b) each stage s contains a group of consecutive operators from $\mathcal{V}$, (c) the total memory of each stage is less than the device memory. Besides, a stage-level graph $\mathcal{G}_S$ consisting of all the stages $s \in \mathcal{S}^*$ as vertices and inter-stage communications as edges is constructed and passed to the next phase (i.e. device mapping).

To solve Problem (1) and find a partitioning solution $\mathcal{S}^*$, several past works introduce dynamic programming-based layer-level model partitioning approaches [5, 19, 26]. However, the practicality of layer-level partitioning is limited by some critical concerns outlined in Section 1 (e.g., low flexibility and requiring pre-processing). [25] introduces a scheme to partition the operator-level graph but yields solutions with limited qualities. In this section, we firstly introduce a DP formulation extended from [19] to tackle the operator-level partitioning problem in Section 4.1 and then discuss two techniques in Section 4.2 and Section 4.3 to reduce the complexity of DP while maintaining solution quality. The details of searching the number of stages S and the number of replicas R are provided in Appendix E.

4.1 Dynamic Programming

Definition 4.1. A subgraph $\mathcal{H}(\mathcal{V}_\mathcal{H}, \mathcal{E}_\mathcal{H})$ of a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ is induced if for any two vertices $u, v \in \mathcal{V}_\mathcal{H}$, $(u, v) \in \mathcal{E}_\mathcal{H}$ if and only if $(u, v) \in \mathcal{E}$.

Definition 4.2. An induced subgraph $\mathcal{F}(\mathcal{V}_\mathcal{F}, \mathcal{E}_\mathcal{F})$ of a directed acyclic graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ is a fronted subgraph if for any vertex u in $\mathcal{V}_\mathcal{F}$, all predecessors of u in $\mathcal{V}$ are also in $\mathcal{V}_\mathcal{F}$.

Definition 4.3. We call a graph set $F_\mathcal{G}$ a *fronted subgraph set* if it contains all possible fronted subgraphs of $\mathcal{G}$ (i.e. $F_\mathcal{G} = \{\mathcal{F} \mid \mathcal{F}$ is a fronted subgraph of $\mathcal{G}\}$)

Theorem 4.4. *For a DAG $\mathcal{G}(\mathcal{V}, \mathcal{E})$, $|F_\mathcal{G}| \geq |\mathcal{V}|$.*

Theorem 4.5. *If a connected DAG $\mathcal{G}(\mathcal{V}, \mathcal{E})$ has a unique topological ordering, $|F_\mathcal{G}| = |\mathcal{V}|$.*

Now we will show how to solve Problem (1) by dynamic programming. The DP table is defined as $T \in \mathbb{R}^{(|F_\mathcal{G}|+1) \times D \times S}$, where D is the number of devices and S is the number of stages. We initialize $T_{\emptyset,d,s} = 0$ for any d, s. Each item $T_{\mathcal{F},d,s} \in T$, which represents the optimal solution of partitioning a fronted graph $\mathcal{F} \in F_\mathcal{G}$ to s stages with assignment to d devices, is given by

$$T_{\mathcal{F},d,s} = \min_{\mathcal{F}' \in F_\mathcal{F} \setminus \{\mathcal{F}\}} \min_{d'=s-1}^{d-1} \max\{T_{\mathcal{F}',d',s-1}, t_{\mathcal{F}-\mathcal{F}',d-d'}\} \tag{2}$$

which also implies that the subgraph $\mathcal{F} - \mathcal{F}'$ is assigned to stage s with $d - d'$ replicas. In Equation (2) $t_{\mathcal{F}-\mathcal{F}',d-d'}$ denotes the stage time of s, formulated as

$$t_{\mathcal{F}-\mathcal{F}',d-d'} = \sum_{v \in \mathcal{V}_{\mathcal{F}-\mathcal{F}'}} \{c_\mathsf{u}(v) + \sum_{v' \in \mathrm{adj}(v) \setminus \mathcal{V}_{\mathcal{F}-\mathcal{F}'}} c_\mathsf{m}(v, v')\}/(d - d') \tag{3}$$

in which $c_\mathsf{m}(v, v')$ denotes the communication time between operator v and operators v', and $\mathrm{adj}(v)$ denotes v's adjacent operators (i.e., a set of all operators that directly communicate with operator v) . Note that if the memory consumed by $\mathcal{F} - \mathcal{F}'$, formulated in Appendix F, is larger than $(d - d') \times DM$, where DM is the device memory, we will let $t_{\mathcal{F}-\mathcal{F}',d-d'} = +\infty$. Note that real-world training with different replicas for each stage will invoke expensive communication operators. We thus empirically consider $d - d'$ as a constant, and detailed discussion will be given in Appendix E.

For all $\mathcal{F} \in F_\mathcal{G}$, we recursively compute Equation (2) and then fill up the table T. The optimal value, $\min \max_{s \in \mathcal{S}} t(s)$ ($t(\cdot)$ denotes the stage time), is naturally given by $T_{\mathcal{G},D,S}$. The optimal solution $\mathcal{S}^*$ will then be derived from the computed table T.

All possible stage times described in Equation (3) can be pre-computed in $O(2^{|\mathcal{V}|}D)$ time. Under the assumption that $\mathcal{G}$ is a sparse DAG (i.e. the average vertex degree in $\mathcal{G}$ is small), the dynamic programming in Equation (2) can run in $O(2^{|\mathcal{V}|}D^2S)$ time. If we assume $\mathcal{G}$ is a connected DAG with a unique topological ordering (which is usually the case in practice) so that Theorem 4.5 applies, the complexities of these two steps are $O(|\mathcal{V}|^2D)$ and $O(|\mathcal{V}|^2D^2S)$ respectively. However, it is still intractable when the dynamic programming approach meets a modern DNN and a large cluster.

4.2 Operator Clustering

The dynamic programming can optimally solve the partitioning problem but the running time will be extremely long if the input computation graph is large. To handle this scalability issue while maintaining quality, operator clustering will be performed before the dynamic programming process. It is observed that most of the operators in $\mathcal{V}$ are lightweight and can be clustered with other operators that are topologically closed to form hyper-operators while balancing the stage time. To this end, we perform hyper-operator merging **before** DP, called *operator clustering*, on the vanilla computation graph $\mathcal{G}$ to significantly accelerate the DP.

Now, we introduce the details of this operator clustering. We will maintain a hyper-operator graph $\hat{\mathcal{G}}_k(\hat{\mathcal{V}}_k, \hat{\mathcal{E}}_k)$ during the process, where $\hat{\mathcal{V}}_k$ denotes a set of hyper-operator nodes, $\hat{\mathcal{E}}_k$ denotes their dependency edges, and k is the number of hyper-operators in $\hat{\mathcal{V}}_k$. At the beginning, this hyper-operator graph is simply derived from the given operator graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ and k is equal to $|\mathcal{V}|$. Starting with $\hat{\mathcal{G}}_{|\mathcal{V}|}(\hat{\mathcal{V}}_{|\mathcal{V}|}, \hat{\mathcal{E}}_{|\mathcal{V}|})$, Parmesan will recursively cluster hyper-operators, and one step of operator clustering for a graph $\hat{\mathcal{G}}_k(\hat{\mathcal{V}}_k, \hat{\mathcal{E}}_k)$ will fuse two adjacent hyper-operators $\hat{u}, \hat{v} \in \hat{\mathcal{V}}_k$ into one hyper-operator with updating the corresponding dependencies. After one step clustering, a new graph $\hat{\mathcal{G}}_{k-1}(\hat{\mathcal{V}}_{k-1}, \hat{\mathcal{E}}_{k-1})$ will be given (A detailed description of this algorithm is given in Appendix B). Note that we can consider a hyper-operator as one kind of operators because it also supports vanilla operator attributes like computation time, communication size and memory, etc.

We will next describe how to select two adjacent hyper-operators $\hat{u}, \hat{v}$ to cluster. Before discussing the selection criteria, we first give the definition of subgraph convexity.

Definition 4.6. A subgraph $\mathcal{H}$ of a directed acyclic graph $\mathcal{G}$ is convex if for any two vertices $u, v \in \mathcal{H}$, there is no directed path between u, v in $\mathcal{G}$ lying outside $\mathcal{H}$.

To maintain $\hat{\mathcal{G}}_{k-1}$ as acyclic after clustering, it is not hard to see that the operators $\hat{u}, \hat{v} \in \hat{\mathcal{V}}_k$ selected to be clustered must be such that the induced subgraph $\mathcal{H}$ with $\mathcal{V}_{\mathcal{H}} = \{\hat{u}, \hat{v}\}$ of the DAG $\hat{\mathcal{G}}_k$ satisfies the **convexity constraint**. Besides, the **memory constraint** should also be satisfied, that is, the total memory consumed by the resulted hyper-operator should not exceed the device memory.

In order to balance the computation cost and reduce the communication cost, Parmesan first enumerates all valid operator pairs that satisfy the two aforementioned constraints and it will then cluster the target pair $(\hat{u}, \hat{v})$ with the smallest cost, given by

$$\text{cost}(\hat{u}, \hat{v}) = c_{\mathsf{u}}(\hat{u}) + c_{\mathsf{u}}(\hat{v}) - \alpha c_{\mathsf{m}}(\hat{u}, \hat{v}) \quad (4)$$

where α indicates relative importance between computation and communication cost. Parmesan recursively performs the operator clustering until the total number of hyper-operators left is no more than a parameter K, which is set to a value that is $S \ll K \ll |\mathcal{V}|$. The resultant hyper-operator graph $\hat{\mathcal{G}}_K(\hat{\mathcal{V}}_K, \hat{\mathcal{E}}_K)$ will then be passed to DP. Because $K \gg S$, operator clustering won't significantly affect DP to yield computation balanced yet communication reduced stages. Since $K \ll |\mathcal{V}|$, the whole operator clustering step will run in $O(|\mathcal{V}||\mathcal{E}|)$ time and the complexity of the DP in Section 4.1 is significantly reduced to $O(2^K D^2 S)$. It is also found that the topological ordering of $\hat{\mathcal{G}}_K$ is unique in most of the cases and the DP will thus run in $O(K^2 D^2 S)$ time according to Theorem 4.5.

4.3 Iterative Refinement

After operator clustering and DP, an iterative refinement will be performed to further improve the partitioning result. Iterative refinement aims at reducing the communication time and balancing the computation time by fine tuning the DP result.

In iterative refinement, atomic operators on a boundary, i.e., operators that have at least one edge connecting to an operator in another stage, may be moved to a neighboring stage. Each refinement step will first find out all the valid move candidates. A move is a tuple that consists of two elements: operator to be moved and its target stage. Only atomic operators on a boundary will be considered, and invalid moves that lead to non-convexity or memory violation will be filtered out. Then Parmesan will select one of candidates according to well-designed selection criteria and then iteratively refines the DP solution until no valid move can be chosen, or a limit I for the maximum number of refinement step is reached. Details of the selection criteria are given in Appendix C.

While operator clustering is a bottom-up approach to handle scalability, iterative refinement is a top-down approach to consider explicitly the influence of a move on stage partitioning to improve quality. It can remarkably mitigate the sub-optimality brought by operator clustering and can enhance the solution quality significantly. The whole model partitioning flow in Parmesan is: (1) operator clustering, (2) dynamic programming and (3) iterative refinement.

5 Device Mapping for General Device Topology

In the previous phase of model partitioning, a stage-level NN graph $\mathcal{G}_S$ containing S vertices along with a constant R indicating the number of replicas per stage are computed. For simplicity, we define a new graph $\mathcal{G}'(\mathcal{V}', \mathcal{E}')$ that is composed of R identical copies of $\mathcal{G}_S$. The objective of the device mapping problem is to obtain a bijective mapping $p : \mathcal{V}' \to \mathcal{D}$ that assigns each $s \in \mathcal{V}'$ to a unique device $d \in \mathcal{D}$ under the heterogeneous network settings, such that the min-max stage time objective

$$\min_p \max_{s \in \mathcal{V}'} c_{\mathsf{stage}}(s, p) \tag{5}$$

is optimized. Note that $c_{\mathsf{stage}}(s, p)$ is a general notation of the stage turnaround time and can be instantiated differently under different scenarios.

In this section, we firstly introduce a general search algorithm that optimally finds a solution for Problem (5) (Section 5.1), and then discuss two instantiations of $c_{\mathsf{stage}}(s, p)$ as well as the corresponding search algorithm (Section 5.2). Equipped with a properly designed selection criterion, a combination of these two algorithms are able to efficiently produce high quality mappings for different NNs.

5.1 A General and Optimal Search Algorithm

The algorithm shown in Algorithm 1 is formulated as a nested two-level searching. The outer-level is essentially a binary search that manages an interval $[t_l, t_r]$ in which the optimal value of Problem (5) resides. In each iteration, the inner-level search is invoked to find whether there exists a mapping p such that the maximum stage time $t_{\max}(p) := \max_{s \in \mathcal{V}'} c_{\mathsf{stage}}(s, p) \leq t$, where t is the mid-point of the interval. The interval shrinks according to the existence of such p at an exponential rate with respect to the number of invocations of the inner-level search. When the length of the interval becomes small enough and no more feasible mappings can be found, the algorithm returns the last found mapping.

Algorithm 1 Device Mapping

```
$p \leftarrow \emptyset,\ t_l \leftarrow t_{l0},\ t_r \leftarrow t_{r0}$
**while** $t_r - t_l > \epsilon > 0$ **do**
    $t \leftarrow (t_l + t_r)/2$
    $p, t_p \leftarrow \mathrm{search}(\mathcal{G}', B, t)$
    **if** $p = \emptyset$ **then**
        $t_l \leftarrow t$
    **else**
        $t_r \leftarrow \min\{t, t_p\}$
**return** $p$
```

The inner-level is a recursive, depth-first search based backtracking algorithm. A partial mapping p is taken as an input state and for each time the algorithm is invoked, it tries to assign the next unmapped stage s by checking all the unallocated devices. For each candidate device d, before starting a new recursive pass with the mapping $p \cup \{(s, d)\}$, the algorithm firstly checks the stage times $c_{\mathsf{stage}}(s', p \cup \{(s, d)\})$ of some previously assigned stages s' that can only be determined after s is assigned. The planning of what stages to be checked when assigning s is called the checking scheme of s. It only depends on the topology of $\mathcal{G}'$ and the instantiation of $c_{\mathsf{stage}}(s, p)$, so it can be precomputed before the overall mapping algorithm. If any of the stage time is larger than the target t, any mapping that includes $p \cup \{(s, d)\}$ is infeasible and need not be further enumerated and checked. This serves as a pruning technique and ensures practical effectiveness of the inner-level search. The recursion terminates when a full mapping is found. A detailed description of this algorithm is given as Algorithm 2 in Appendix D.

We provide a proof in Appendix A.3 showing that Algorithm 1 always gives the optimal solution, provided that the initial interval contains the optimal value.

5.2 Two Instantiations

In the scenario of hybrid pipeline training, a natural thought on formalizing $c_{\mathsf{stage}}(s, p)$ is to include the computational time, inter-stage communication time and inter-replica allreduce time[1], i.e., $c_{\mathsf{stage}}(s, p) = c_{\mathsf{u}}(s) + c_{\mathsf{m}}(s, p) + c_{\mathsf{AR}}(s, p)$. Despite its theoretical exactness, this form imposes so many constraints on the checking scheme used in Algorithm 2 in Appendix D that the time of a particular stage cannot be determined until a considerable number of stages after it have also been assigned. This in turn will lead to delayed pruning and inefficient searching, and thus can be hardly applied in large scale settings in which hundreds of devices are involved. To handle this issue, we

[1]In this work, we mainly consider the ring-allreduce.

propose two kinds of instantiation for $c_{\mathsf{stage}}(s,p)$ such that they, combined, provides mapping results of similar qualities as the aforementioned form but consumes less computational time in practice.

A key observation is that the ratio between inter-stage communication cost c_{m} (determined by intermediate feature maps) and inter-replica communication cost c_{AR} (determined by the number of NN parameters) varies with different $\mathcal{G}'$. For example, in general, CNN image models have fewer parameters than Transformer-based language models thanks to convolutions, but their intermediate tensors are larger due to the 2D nature of images. In light of this, we design the two instantiations as follows:

$$c^{\mathsf{m}}_{\mathsf{stage}}(s,p) = c_{\mathsf{u}}(s) + c_{\mathsf{m}}(s,p), \ \ c_{\mathsf{m}}(s,p) = \sum_{s' \in \mathrm{adj}(s)} \frac{M(s,s')}{B(p(s),p(s'))}, \tag{6}$$

$$c^{\mathsf{AR}}_{\mathsf{stage}}(s,p) = c_{\mathsf{u}}(s) + c_{\mathsf{AR}}(s,p), \ \ c_{\mathsf{AR}}(s,p) = \max_{\substack{s_i, s_{i+1} \\ \in \mathrm{ring}(\mathrm{repl}(s))}} \left\{ \frac{2 \cdot (R-1) \cdot P(s)}{R \cdot B(p(s_i), p(s_{i+1}))} \right\}, \tag{7}$$

that apply to the inter-stage and the inter-replica dominant cases respectively. $M(s,s')$ is the communication size between stage s and s', and $B(d_1,d_2)$ denotes the bandwidth between device d_1 and d_2. $P(s)$ is the total parameter size in s, and $\mathrm{ring}(\mathrm{repl}(s))$ represents the set of adjacent pairs of replicas that appear in the allreduce ring of s (e.g., if $\mathrm{repl}(s) = \{s_0, s_1, s_2\}$, then $\mathrm{ring}(\mathrm{repl}(s)) = \{(s_0,s_1),(s_1,s_2),(s_2,s_0)\}$).

The initial lower bounds t_{l0} in Algorithm 1 of the two instantiations are computed respectively as

$$t^{\mathsf{m}}_{l0} = \max_{s \in \mathcal{V}'} \left\{ c_{\mathsf{u}}(s) + \min_{d} \min_{p^{\mathsf{m}}_{s,d}} \sum_{s' \in \mathrm{adj}(s)} \frac{M(s,s')}{B(d, p^{\mathsf{m}}_{s,d}(s'))} \right\} \tag{8}$$

and

$$t^{\mathsf{AR}}_{l0} = \max_{s \in \mathcal{V}'} \{c_{\mathsf{u}}(s) + \min_{d} c^{\min}_{\mathsf{AR}}(s,d)\}, \ c^{\min}_{\mathsf{AR}}(s,d) = \min_{p^{\mathsf{AR}}_{s,d}} \max_{s' \in \mathrm{adj\_repl}(s)} \left\{ \frac{2 \cdot (R-1) \cdot P(s)}{R \cdot B(d, p^{\mathsf{AR}}_{s,d}(s'))} \right\}, \tag{9}$$

where $p^{\mathsf{m}}_{s,d} : \mathrm{adj}(s) \to \mathcal{D} \setminus \{d\}$ (resp. $p^{\mathsf{AR}}_{s,d} : \mathrm{adj\_repl}(s) \to \mathcal{D} \setminus \{d\}$) represents a partial assignment of the adjacent stages (resp. two adjacent replicas on the ring) of s to devices other than d. These can be computed in polynomial time using a sorting-based approach. The initial upper bounds are generated by the minimum value between random mapping and heuristic mapping.

Proposition 5.1. *$t^{\mathsf{m}}_{l0} \le \min_p \max_{s \in \mathcal{V}'} c^{\mathsf{m}}_{\mathsf{stage}}(s,p)$ and $t^{\mathsf{AR}}_{l0} \le \min_p \max_{s \in \mathcal{V}'} c^{\mathsf{AR}}_{\mathsf{stage}}(s,p)$. Hence, the initial interval obtained as above always contain the optimal value for the respective instantiation.*

Although these two instantiations of $c_{\mathsf{stage}}(s,p)$ enables effective pruning during the inner-level search and make the search trees shallower, in the worst case the inner-level search still has a complexity of $O(D!)$ (as the mapping problem is NP-complete). Therefore, we perform an enhancement and a heuristic on Algorithm 1 to further accelerate the overall device mapping. The enhancement is to launch a batch of inner-level searches with different targets (t) using multi-threading. The targets are selected such that the interval is evenly divided. The lower bound would be updated as the largest value of all the not-found targets, and the upper bound would be the smallest value of all the found targets. As a heuristic, we enable timeout for preventing search tasks from running over long, and the results for the early-stopped threads are regarded as not found. Admittedly, this heuristic would affect the optimality of device mapping, but as the experiments on our instantiations in Appendix J show, in general the result quality is rarely affected.

For a certain $\mathcal{G}'$, one of the two instantiations is automatically selected based on the ratio between the allreduce communication size and total inter-stage communication size. If this ratio is larger than 1, then $c^{\mathsf{AR}}_{\mathsf{stage}}(s,p)$ is selected, otherwise $c^{\mathsf{m}}_{\mathsf{stage}}(s,p)$ is selected.

6 Discussions

Theorem 6.1. *The device mapping problem is NP-complete.*

As Theorem 6.1 shown, the device mapping problem (phase 2) for general topology is an NP-complete problem. If we consider the various bandwidth inside the general topology during model partitioning (phase 1), that is, do phase 1 and phase 2 simultaneously, the overall complexity will be too high to be solved effectively. To make the problem solvable, we decouple the whole problem into two phases (model partitioning and device mapping). However, such decoupling unavoidably brings an issue to phase 1: we cannot foresee which two devices a partition and its adjacent partition will be mapped to, so the bandwidth between these two devices is hard to determine beforehand, especially for a general topology. However, the communication size between two adjacent partitions can be captured.

As a result, in phase 1, we assume the flattened device topology and aim at balancing the computation time while reducing the communication size. In phase 2, we take the device topology into account explicitly, and optimally map the partitions generated by phase 1 onto a user given general topology, and consider the various communication bandwidths between different devices. Empirically, such a two-phase paradigm works well within an acceptable runtime.

7 Experimental Results

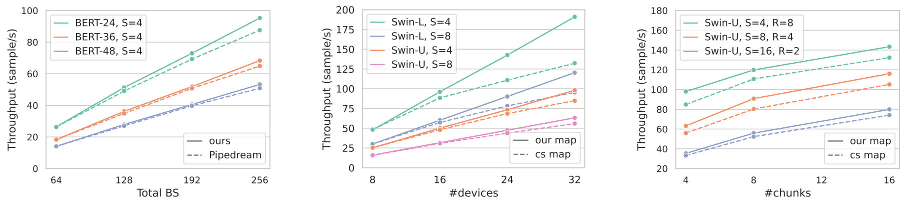
We evaluate our proposed method through measuring (1) the latency of running a real scheduled pipeline training, and (2) the simulated pipeline running time for non-regular topology. We use a synchronous pipeline where each training step consists of four micro-batches followed by an allreduce and parameter update, with activation recomputing enabled. The implementation details of our pipeline training scheduler can be found in Appendix H. Unless specified, the following settings are adopted by default: $I = 100$, $\alpha = 0.001$. We mainly conduct experiments on ResNet[9], BERT[4] and Swin Transformer[16]. The settings of different NNs and different device topologies are described in Appendix I.

Two-level hierarchical architecture. We compare the throughput with Pipedream (layer-level) and RaNNC (operator-level) since they both provide I/O interface and support configuring the number of stages. As device mapping is not provided in these baselines, we put consecutive stages on consecutive devices (CS map). Note that CS map is the most commonly used heuristic for device mapping, which aims to alleviate the allreduce time. The whole evaluation flow can be summarized as follows: (1) acquire the DNN partitioning results from the partitioner of these baselines; (2) adopt the CS map/our map to these partitioning; (3) invoke our synchronous pipeline scheduler to measure the training latency and throughput. We fix the number of micro-batches MB as 4. Note that Pipedream fails to partition the given operator-level graph (BERT and Swin) within 2 hours, so we develop several pre-processing techniques for Pipedream to generate the layer-level graph input from the operator-level graph. Meanwhile, RaNNC reports an unknown error when partitioning Swin (marked as "-" in Table 1).

Table 1: A comparison with relevant baselines. Speedup by our mapping is marked in brown.

Task	Res152 Classifi.			BERT-L Pre-train			Swin-L Pre-train		
(S, R)	(4,4)	(8,2)	(16,1)	(4,4)	(8,2)	(16,1)	(4,4)	(8,2)	(16,1)
Pipedream	1.0x	1.0x	1.0x	1.0x	1.0x	1.0x	1.0x	1.0x	1.0x
+ our map	7.6x	2.4x	1.0x	1.0x	1.0x	1.0x	2.5x	1.4x	1.2x
RaNNC	0.9x	1.3x	0.8x	0.8x	0.9x	0.9x	-	-	-
+ our map	7.2x	2.5x	0.9x	0.8x	0.9x	0.9x	-	-	-
Ours	**8.1x**	**2.6x**	**1.2x**	**1.1x**	**1.1x**	**1.0x**	**2.5x**	**1.6x**	**1.3x**

We measure the relative speedup of the synchronous pipeline training on a two-level hierarchical architecture, and the results are shown in Table 1. By default, we apply CS map for Pipedream and RaNNC. To demonstrate the orthogonal improvement of our mapping algorithm, we also apply our method to map their model partitions (+ our map). Since BERT is built with repeating blocks and is allreduce-heavy, our mapping algorithm produces similar solutions as CS map. Note that our algorithm still obtains $1.1\times$ speedup compared to Pipedream due to a better-balanced model partitioning. For the vision model ResNet-152 and Swin-L, our mapping algorithm speeds up the throughput of PipeDream and RaNNC significantly compared to CS map. Unlike BERT, ResNet and Swin are both p2p-heavy (Swin-L is also allreduce-heavy). As a kind of allreduce-first mapping, CS map is no longer suitable for these cases. In contrast, our mapping algorithm can optimally map the partitions via p2p-first without a human in-loop. Note that our mapping on Pipedream's Swin-L $(8, 2)$ and $(16, 1)$ and RaNNC's ResNet-152 $(16, 1)$ is neither consecutive mapping nor p2p-first, which further demonstrates the effectiveness of our mapping algorithm comparing to the human-designed heuristic. Moreover, our results achieve around 10% speedup for ResNet and Swin compared to other partitioning algorithms with our mapping, which indicates the better quality of our operator-level model partitioning algorithm. The results of our full-flow algorithm compared to other baselines show the superiority of Parmesan and demonstrate its robustness.



(a) $mBS = 8, MB = 4$ (b) $mBS = 16, MB = 4$ (c) $mBS = 16, D = 32$

Figure 3: Throughput of different BERTs and Swins on DGX-1 V100 Architecture.

DGX-1 V100 architecture. We conduct larger-scale experiments on the Alibaba cloud DGX-1 V100 architecture, in which the throughput under different settings are compared. In Figure 3(a), we compare the solution quality of different BERTs with Pipedream. We fix the number of micro-batches MB as 4 and the size of each micro-batch mBS as 8, and we change the number of replicas R and the number of BERT layers to measure the throughput under different total batch sizes BS ($BS = mBS \times R \times MB$). In Figure 3(b), we keep $mBS = 16, MB = 4$ and modify the number of devices D ($D = S \times R$) to evaluate the throughput of different Swin models under different mapping algorithms. In Figure 3(c), we measure the throughput under various (S, R, MB) configuration and different mapping algorithms for Swin-U while maintaining $mBS = 16, D = 32$. Our model partitioning algorithm is applied for all experiments shown in Figure 3(b) and Figure 3(c). Note that DGX-1 is not a completely hierarchical architecture. Although inter-DGX-1 connection still relies on Ethernet, there are three types of intra-DGX-1 connections, namely double NVLinks, single NVLink, and PCI-e. All these experiments demonstrate the high extensibility and the robustness of Parmesan.

Table 2: Simulation results for different topology. Speedup by our mapping over CS mapping is shown. "-" denotes the number of devices D ($D = S \times R$) cannot form a specific torus/mesh architecture.

(S,R)	(4,16)	(8,8)	(16,4)	(4,64)	(8,32)	(16,16)	(4,128)	(8,64)	(16,32)
2d mesh	1.1x	1.0x	2.7x	5.6x	2.5x	1.4x	-	-	-
2d torus	1.1x	1.0x	2.6x	1.6x	1.5x	1.0x	-	-	-
3d mesh	1.0x	1.1x	1.1x	-	-	-	1.3x	1.8x	1.2x
3d torus	1.0x	1.1x	1.0x	-	-	-	1.1x	1.0x	2.6x
random_blk_1	1.5x	1.8x	1.5x	1.1x	1.4x	1.9x	1.1x	1.7x	1.9x
random_blk_2	2.1x	1.6x	3.0x	1.4x	1.3x	3.7x	1.5x	1.6x	4.5x
uniform_dist	33.5x	11.4x	6.7x	8.1x	5.1x	7.2x	23.3x	8.9x	5.5x

Non-regular architecture. We mentioned above that DGX-1 has some intra-node heterogeneity, but it still resembles the traditional hierarchical architecture to some extent. To evaluate the performance of our mapping algorithm for general device topology, architectures of specialized hardware can be considered. Unfortunately, it is difficult to verify our algorithm on those hardware due to unavailability or incompatibility with the commonly used programming interfaces. Hence, we developed a simulator to simulate the performance and conduct a comparison with the human-designed heuristic on popular grid-based architectures (mesh and torus) and fully heterogeneous topologies. Also, the experiments are conducted on SemanticFPN [12] whose partitioning results include more skip-connections, bringing more challenges to solving the problem. Note that we randomly generate three kinds of fully heterogeneous topologies, named random_blk_1, random_blk_2 and uniform_dist (details are in Appendix I). As shown in Table 2, our mapping algorithm yields significant speedup over CS mapping. Our proposed general topology mapping algorithm can solve the mapping problem once and for all without human in the loop and provide valuable information for architecture designers.

More experimental results, including ablation studies, are given in Appendix J.

8 Conclusion

In this work, we investigate the model partitioning and device mapping problem for DNN training. First, we observe that existing works mainly focus on layer-level partitioning. From the DNN framework's and specialized hardware's perspective, partitioning at the operator level is more reasonable. Second, we notice that previous works lack considering the importance of mapping, and their mapping algorithm is based on some human-designed heuristic or even hand-craft. To this end, we present Parmesan, an efficient yet robust optimization framework to maximize the training throughput for operator-level DNNs on systems with general topology. We show the superiority of our mapping algorithm when tackling some non-hierarchical architecture compared to the existing heuristic. Future work would explore more pruning strategies for device mapping, consider co-optimization of model partitioning and device mapping, and take pipeline bubble into account.

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*, 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. In *Advances in Neural Information Processing Systems*, pages 1877–1901, 2020.

[3] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 571–582, 2014.

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[5] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. Dapple: a pipelined data parallel approach for training large models. *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021.

[6] Yuanxiang Gao, Li Chen, and Baochun Li. Spotlight: Optimizing device placement for training deep neural networks. In *International Conference on Machine Learning*, pages 1676–1684. PMLR, 2018.

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

[8] William Gropp, Luke N Olson, and Philipp Samfass. Modeling mpi communication performance on smp nodes: Is it time to retire the ping pong test. In *Proceedings of the 23rd European MPI Users' Group Meeting*, pages 41–50, 2016.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[10] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, pages 103–112, 2019.

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

[12] Alexander Kirillov, Ross Girshick, Kaiming He, and Piotr Dollár. Panoptic feature pyramid networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6399–6408, 2019.

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

[14] Shigang Li and Torsten Hoefler. Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

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

[16] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 10012–10022, 2021.

[17] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V Le, and Jeff Dean. A hierarchical model for device placement. In *International Conference on Learning Representations*, 2018.

[18] 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 *International Conference on Machine Learning*, pages 2430–2439. PMLR, 2017.

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

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

[21] The nvidia collective communication library (nccl). `https://developer.nvidia.com/nccl`, 2021.

[22] 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*, 2019.

[23] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019.

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

[25] Masahiro Tanaka, Kenjiro Taura, Toshihiro Hanawa, and Kentaro Torisawa. Automatic graph partitioning for very large-scale deep learning. *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1004–1013, 2021.

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

[27] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex Smola. Parallelized stochastic gradient descent. In *Advances in Neural Information Processing Systems*, volume 23, 2010.