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

Lixin Liu, Tianji Liu, Bentian Jiang, and Evangeline F.Y. Young, *Fellow, IEEE*

***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 design flow, named *Parmesan*, to map the training of a large DNN onto a system with general device topology to maximize the throughput. 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 systems with different GPU topologies and is able to handle the mapping problem for heterogeneously interconnected architectures. We believe our proposed general device topology mapping algorithm will provide valuable information for architecture designers and assist them in designing a more DNN-friendly architecture.**

***Index Terms*—neural network acceleration, deep learning system, partitioning, mapping, optimization, parallelism**

I. Introduction

THE ever-increasing Deep Neural Network (DNN) model size [1]–[3] has stimulated the development of distributive learning scheme in pursuit of more efficient large-scale DNN training [4]–[6]. 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 effectiveness in terms of network and device topology coverage; and (3) turnaround efficacy for searching a desirable parallelism strategy.

However, designing high-performance distribution strategies for different neural network architectures over different device topologies is non-trivial and challenging, because of the astronomically large search space brought by the growing model size and complex device interconnects. Various paradigms, including but not limited to data parallelism [7]–[9], model parallelism [10]–[12] and pipeline parallelism [13]–[18], 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, etc., with the objective of finding the best tradeoff among device-utilization, memory footprint and training convergence (for asynchronous training).

Recent pipeline scheduling literature [13]–[18] 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 maximize training throughput. Although some existing works [17]–[19] 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 requires pre-processing. Layer-level partitioning assumes 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 with limited memory for each core, so a large number of partitions are required to fully utilize the cores. In such cases, higher flexibility can only be provided at the operator-level (op-level). Besides, since the intermediate representation (IR) graph of 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 the op-level. Such pre-processing also 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 partitioning a DNN.

Heuristic-based device mapping cannot be generalized to different hardware architectures. The most commonly-used heuristic for device mapping is to put consecutive stages of a partitioned DNN training on consecutive devices (we call it consecutive mapping). Although consecutive mapping works well in some cases, it cannot guarantee good performance. We demonstrate this using a simple example of mapping four stages on a 2×2 hierarchical topology, shown in Fig. 2.

This work was partially supported by a grant from the Research Grants Council of the Hong Kong SAR (Project No. CUHK14210923).

The authors are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong (e-mail: lxliu@cse.cuhk.edu.hk).

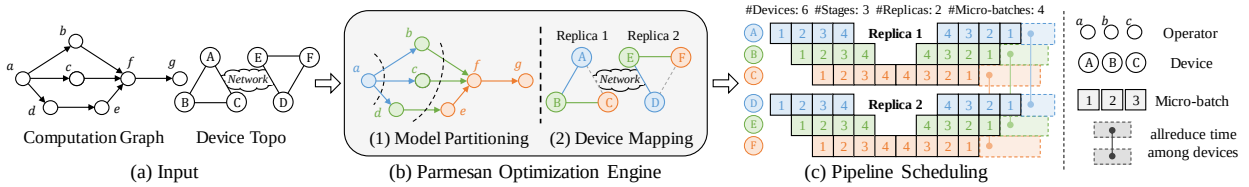


Fig. 1. The workflow of optimizing the pipelined training throughput of using Parmesan.

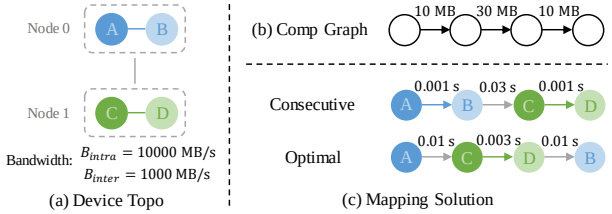


Fig. 2. An example to show consecutive mapping cannot guarantee optimality.

The total communication latency of consecutive mapping is around $1.4\times$ larger than the optimal solution. Admittedly, one may develop heuristics for device mapping to produce a near-optimal performance on some commonly-used hierarchical GPU architectures, but there are many dedicated hardwares other than GPUs for DNN training today which are of very different networking topologies or even integrating multiple processors and interconnections on a single chip (e.g., Google TPU, Cerebras CS-1). Existing heuristics may struggle in those scenarios, and new heuristics are needed to be invented. A good 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.

In light of the above, we proposed Parmesan, an efficient middleware for large-scale pipeline training based on the PyTorch. 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 Fig. 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 an 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. Experimental results on real-world training and simulations on non-hierarchical topologies indicate the effectiveness of Parmesan.

II. Related Work

Data Parallelism. Data parallelism partitions training data into batches and enables model training to scale up to a distributed system [7]–[9]. Each device maintains a *replica* of the entire model and computes gradient synchronously by a technique called *allreduce* [20]. The computed gradient is then applied to update the model parameters.

Model Parallelism. When the memory of a device is insufficient to maintain a DNN, model parallelism is an alternative option to address the memory issue. Model parallelism handles large model training by partitioning DNN into several disjoint sets and spreading them across different devices [10]–[12].

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. The split micro-batches and the partitioned DNN will be delicately scheduled, resulting in an increase in throughput. GPipe [13] synchronously schedules the forward and backward propagation during training. DAPPLE [17] introduces an one-forward-one-backward synchronous pipeline. PipeDream [16], [18] 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. In this work, we mainly consider synchronous pipelined training whose training accuracy is not affected theoretically. Fig. 1(c) shows an example of synchronous pipeline combined with data parallelism (also called *hybrid parallelism*).

DNN Partitioning and Mapping. Orthogonal to pipeline scheduling, balancing of DNN partitions and proper mapping of the partitions to a distributed multi-GPU system are very important factors to maximize the training throughput [1]. The works [21]–[23] utilized reinforcement learning to partition and map a DNN, using the resulting throughput as a training reward. However, these learning-based approaches often encounter challenges due to the extensive problem spaces of training modern DNNs and require time-consuming online

[1]In this work, we mainly focus on partitioning and mapping DNNs to GPUs to accelerate DNN training. We assume GPU devices are homogeneous while their interconnected bandwidths are heterogeneous. As for the CPU, in this work, it is mainly used for data transmission.

throughput measurements. These limitations not only restrict them from generating high-throughput solutions but also largely affect the overall pipeline training time. In contrast, dynamic-programming-based partitioning algorithms, such as PipeDream [18], DAPPLE [17], Piper [19], RaNNC [24], and Alpa [25], emerge as prominent choices and demonstrate their effectiveness in optimizing real-world pipeline training. However, PipeDream, Piper, and DAPPLE only focus on coarse-grained layer-level granularity. Alpa and RaNNC consider operator-level graphs, while their algorithms rely on a pre-processed topological order. As for device mapping, RaNNC only considers flattened device topologies of constant bandwidth. PipeDream, Piper, and Alpa exploit dynamic programming to address the hierarchical topology mapping problem, but they do not provide a general mapping algorithm to tackle arbitrary device topologies. DAPPLE proposes device mapping methods based on some heuristics which have no guarantee of solution quality.

III. 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 the 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 IV) and device mapping (in Section V). A discussion of decoupling the problem into two phases is given in Section VI. Section VII discusses the operator-level graph extractor, profiler, and the evaluation modules.

IV. 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 [18], 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 [17]–[19]. However, the practicality of layer-level partitioning is limited by some critical concerns outlined in Section I. Besides, modern DNNs exhibit more complex structures, such as skip connection and multi-head attention, which can result in multiple possible topological orders within the operator-level graph. Given that most layer-level algorithms primarily depend on a pre-sorted order, their partitioning solution qualities can be largely affected when adapting them to tackle operator-level graphs. [24] 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 [18] to tackle the operator-level partitioning problem in Section IV-A and then discuss two techniques in Section IV-B and Section IV-C to reduce the complexity of the DP while maintaining solution quality. The details of searching the number of stages S and the number of replicas R are given in Appendix B.

A. Dynamic Programming

Definition 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 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 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. $F_\mathcal{G} = \{\mathcal{F} \mid \mathcal{F} \text{ is a fronted subgraph of } \mathcal{G}\}$)

Theorem 1. For a DAG $\mathcal{G}(\mathcal{V}, \mathcal{E})$, $|F_\mathcal{G}| \geq |\mathcal{V}|$.

Theorem 2. If a connected DAG $\mathcal{G}(\mathcal{V}, \mathcal{E})$ has a unique topological ordering, $|F_\mathcal{G}| = |\mathcal{V}|$.

After discussing these definitions, we then introduce the primary motivation behind utilizing DP in DNN partition. Given that a DNN is represented as a DAG $\mathcal{G}$, it is natural to consider all its fronted subgraphs $\mathcal{F} \in F_\mathcal{G}$ as DAGs. Consequently, the overall partitioning problem on $\mathcal{G}$ can be decoupled into smaller partitioning problems on each subgraph $\mathcal{F}$. This inherent structure leads us to choose DP to optimally solve the overall partitioning problem. Note that our DP is able to handle non-unique topological orders since fronted subgraphs already capture all possible sub-structures.

To solve Problem (1) by dynamic programming, we first define a DP table as $T \in \mathbb{R}^{(|F_\mathcal{G}|+1) \times D \times S}$, which represents the best timing (computation and communication) achievable to partition the computational graph represented by the first parameter with d devices and s stages where $d \leq D$ and $s \leq S$. 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

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

In the above formulation, the subgraph $\mathcal{F} - \mathcal{F}'$ is assigned to stage s with $d - d'$ devices (replicas). Note the range of d' represents the minimum and maximum number of devices required to partition the fronted subgraph $\mathcal{F}'$. The term $t_{\mathcal{F}-\mathcal{F}',d-d'}$ denotes the stage time of s, formulated as

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

where $c_{\mathsf{u}}(v)$ is the computation time for operator v and $c_{\mathsf{m}}(v,v')$ 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 C, is larger than $(d-d') \times DM$, where DM is the device memory, we will put $t_{\mathcal{F}-\mathcal{F}',d-d'} = +\infty$. Note that real-world training with different number of devices (replicas) for each stage will invoke expensive collective communication operators. We thus empirically put $d - d'$ a constant R, and detailed discussion will be provided in Appendix B.

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 degree of vertices $\overline{d} = O(|\mathcal{V}|)$), the dynamic programming in Equation (2) can run in $O(2^{|\mathcal{V}|}D^2S)$ time. If we assume that $\mathcal{G}$ is a connected DAG with a unique topological ordering so that Theorem 2 applies, the complexities of these two steps are $O(|\mathcal{V}|^2D)$ and $O(|\mathcal{V}|^2D^2S)$ respectively. However, in modern DNNs, the operator graphs always contain thousands of operators and have multiple topological orders, so it is intractable to directly apply DP to solve the operator-level partitioning problem.

B. 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. In practice, 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. Therefore, it is natural to group these lightweight operators within the same partition to balance the stage time. Such domain-specific knowledge guides us in developing *operator clustering*, which performs hyper-operator merging **before** DP on the vanilla computation graph $\mathcal{G}$ to significantly accelerate the DP.

Now, we introduce the details of this operator clustering. We maintain a hyper-operator graph $\hat{\mathcal{G}}_k(\hat{\mathcal{V}}_k, \hat{\mathcal{E}}_k)$ during the operator clustering, 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. One step of 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 with update of the corresponding dependencies in $\hat{\mathcal{E}}_k$, forming a new graph $\hat{\mathcal{G}}_{k-1}(\hat{\mathcal{V}}_{k-1}, \hat{\mathcal{E}}_{k-1})$. Note that we can consider a hyper-operator as one kind of operators because it also supports attributes like computation time, communication size and memory, etc.

We will next describe the criteria of selecting 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. 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.

To balance the computation cost and reduce the communication cost, Parmesan first enumerates all valid operator pairs that both satisfy the convexity and memory constraints. Then, Parmesan clusters the target pair $(\hat{u}, \hat{v})$ with the smallest cost,

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

where α indicates relative importance between computation and communication cost. The value of α is automatically selected from 0.01, 1, and 100 with the assistance of the offline throughput simulator. Starting from the vanilla graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, 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 input to DP. As $K \gg S$, operator clustering will not significantly affect DP to yield computation balanced yet communication reduced stages. Besides, $K \ll |\mathcal{V}|$, the whole operator clustering step will thus run in $O(|\mathcal{V}||\mathcal{E}|)$ time and the complexity of the DP in Section IV-A 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 2.

C. Iterative Refinement

After operator clustering and DP, we observe that it is possible to further balance computation cost or reduce communication between two adjacent partitions by simply moving a few operators from one to another. Thus, we perform iterative refinement to further fine-tune 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. After finding out all the valid candidates, Parmesan will calculate their movement gain according to the following three metrics:

1) decrease in the shortest path distance to the nearest reconverging operator (A reconverging vertex in a DAG is a cut vertex with at least one of its in-degree or out-degree larger than one.)
2) decrease in the total communication size
3) decrease in the maximum stage computation time.

Note that the gain values are the higher the better for all the three metrics and the $i+1$-st metric is used as a tie breaker for the i-th metric. A move with the highest gain is selected to perform one step of iterative refinement.

Parmesan 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. We set I as 100. Since $I \ll |\mathcal{V}|$, the iterative refinement will thus run in $O(|\mathcal{V}|(|\mathcal{V}|+|\mathcal{E}|))$ time.

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.

V. 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 under a mapping p 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 V-A), and then discuss two instantiations of $c_{\mathsf{stage}}(s, p)$ as well as the corresponding search algorithm (Section V-B). Equipped with a properly designed selection criterion, a combination of these two algorithms are able to efficiently produce high quality mappings for different NNs.

A. A General and Optimal Searching Algorithm

The method 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.

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 connection edges in $\mathcal{G}'$ and the instantiation

Algorithm 1 Device Mapping

Input: $\mathcal{G}'$ with c_{u} and message size as vertex and edge attributes, bandwidth B between each pair of devices.

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

Algorithm 2 Inner-level Search, $\text{search}(\mathcal{G}', B, t)$

Input: Graph $\mathcal{G}'$ with c_{u} and c_{m} as vertex and edge attributes, bandwidth B between each pair of devices, target of the maximum stage time t.

function dfs($(\mathcal{G}', B, t, p, t_p)$)
  if $|p| = |V(\mathcal{G}')|$ **then return** (p, t_p)
  $s \leftarrow \text{mapping\_order}(\mathcal{G}')[|p|]$
  $t_p^{\text{old}} \leftarrow t_p$
  for all d not assigned a stage in p **do**
    satisfied $\leftarrow$ true
    for all $s' \in \text{checking\_scheme}(s)$ **do**
      $t_{s'} \leftarrow c_{\mathsf{stage}}(s', p \cup \{(s, d)\})$
      if $t_{s'} > t$ **then**
        satisfied $\leftarrow$ false; **break**
      $t_p \leftarrow \max\{t_p, t_{s'}\}$
    if satisfied **then**
      $p^{\text{res}},\ t_p^{\text{res}} \leftarrow \text{dfs}(\mathcal{G}', B, t, p \cup \{(s, d)\}, t_p)$
      if $p^{\text{res}} \neq \emptyset$ **then return** $(p^{\text{res}}, t_p^{\text{res}})$
    $t_p \leftarrow t_p^{\text{old}}$
  return $(\emptyset, 0)$

$s_0 \leftarrow \text{mapping\_order}(\mathcal{G}')[0]$
for all $d \in \mathcal{D}$ **do**
  $p, t_p \leftarrow \text{dfs}(\mathcal{G}', B, t, \{(s_0, d)\}, 0)$
  if $p \neq \emptyset$ **then return** (p, t_p)
return $(\emptyset, 0)$

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 is an important pruning technique to ensure the practical effectiveness of the inner-level search. The recursion terminates when a full mapping is found. A detailed description of this algorithm is given by Algorithm 2.

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

B. 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[2], 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 formulation imposes many constraints on the checking scheme used in Algorithm 2 that the time of a particular stage cannot be determined until a considerable number of later stages have also been assigned. This in turn will lead to delayed pruning and inefficient search, and thus can be hardly applied in large-scale settings that hundreds of devices are involved. To handle this issue, we propose two kinds of instantiation for $c_{\mathsf{stage}}(s,p)$ such that they together provides mapping results of similar qualities as the aforementioned formulation 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 that apply to the inter-stage and inter-replica dominant cases respectively as follows:

$$c^{\mathsf{m}}_{\mathsf{stage}}(s,p) = c_{\mathsf{u}}(s) + c_{\mathsf{m}}(s,p), \tag{6}$$

$$c^{\mathsf{AR}}_{\mathsf{stage}}(s,p) = c_{\mathsf{u}}(s) + c_{\mathsf{AR}}(s,p), \tag{7}$$

$$c_{\mathsf{m}}(s,p) = \sum_{s' \in \mathrm{adj}(s)} \frac{M(s,s')}{B(p(s),p(s'))}, \tag{8}$$

$$c_{\mathsf{AR}}(s,p) = \max_{\substack{s_i,s_{i+1} \\ \in \mathrm{ring}(\mathrm{repl}(s))}} \left\{ 2\frac{R-1}{R} \frac{P(s)}{B(p(s_i),p(s_{i+1}))} \right\} \tag{9}$$

where $M(s,s')$ is the communication size between stage s and s', $B(d_1,d_2)$ 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 $\mathrm{ring}(\mathrm{repl}(s)) = \{(s_0,s_1),(s_1,s_2),(s_2,s_0)\}$).

With the partial assignment of devices to stage $s \in \mathcal{V}'$ and its adjacencies, $\min_d c_{\mathsf{stage}}(s,d)$ can be derived and it is always smaller than or equal to the optimal value of Problem (5).

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

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

$$t^{\mathsf{m}}_{l0} = \max_{s \in \mathcal{V}'} \left\{ c_{\mathsf{u}}(s) + \min_d \min_{p^{\mathsf{m}}_{s,d}} \sum_{s' \in \mathrm{adj}(s)} \frac{M(s,s')}{B(d, p^{\mathsf{m}}_{s,d}(s'))} \right\} \tag{10}$$

and

$$t^{\mathsf{AR}}_{l0} = \max_{s \in \mathcal{V}'} \{ c_{\mathsf{u}}(s) + \min_d c^{\min}_{\mathsf{AR}}(s,d) \},$$
$$c^{\min}_{\mathsf{AR}}(s,d) = \min_{p^{\mathsf{AR}}_{s,d}} \max_{s' \in \mathrm{adj\_repl}(s)} \left\{ 2\frac{R-1}{R} \frac{P(s)}{B(d, p^{\mathsf{AR}}_{s,d}(s'))} \right\}, \tag{11}$$

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 taking the minimum value between a random mapping and a heuristic mapping.

Proposition 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. We also enable timeout for preventing this searching task from running over long, and the results for the early-stopped threads are regarded as not found.

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

VI. Discussions

Theorem 3. The device mapping problem is NP-complete.

As Theorem 3 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

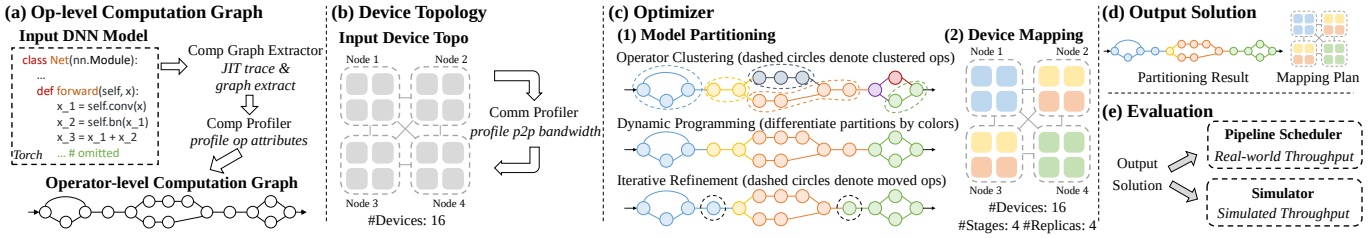


Fig. 3. The whole workflow of Parmesan. (a) Utilizing a graph extractor to extract an operator-level computation graph. (b) Measuring p2p bandwidth within the device topology. (c) An illustration of our optimization process, which takes the operator-level computation graph and device topology as input to perform model partitioning and device mapping. In model partitioning, operators sharing the same color either denote a specific hyper-operator (in op clustering) or represent a specific partition (in DP and refinement). In device mapping, a device marked with a particular color indicates its mapping to a replica of the corresponding colored partition. (d) The output solution of our optimizer. (e) Evaluating the output solution by a pipeline scheduler or a simulator.

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.

VII. Implementation

We implement Parmesan with PyTorch, Numba, and NetworkX. The core of Parmesan contains around 12k lines of Python. In this section, we will introduce other components in Parmesan. To simplify the notation, we will use the word *optimizer* to denote the whole optimization process (including model partitioning and device mapping).

Fig. 3 describes the whole workflow of Parmesan. Given a DNN model written in PyTorch, our graph extractor will first conduct just-in-time (JIT) tracing and automatically extract the operator-level computation graph. We will then launch the profiler to profile the operator attributes (including forward/backward time/memory), compute the total parameter size of this operator (allreduce time is highly correlated to the parameter size), and calculate inter-op communication size.

As for the device topology, we first adopt our communication profiler to measure the point-to-point (*p2p*) communication overhead between every pair of devices. Then we construct a p2p bandwidth look-up-table (LUT) based on the postal model [26], [27] to represent the device topology.

After building the operator-level computation graph and the device topology LUT, our optimizer will take them as input and output the partitioning result and mapping plan.

Parmesan's pipeline scheduler and simulator evaluate the solution quality and provide instructive information for further development. Inspired by FlexFlow [11], we develop a task-graph-based simulator to tackle the general device topology simulation problem. Our simulator considers pipelined forward/backward propagations, pipeline bubbles, and gradient allreduce. For the real-world evaluation, we design a GPipe [13] fashion pipeline scheduler in PyTorch with CUDA 11.3 and adopt NCCL 2.10.3 distributed backend for both the p2p communication between pipeline stages and allreduce between the stage replicas. Note that the Python code snippets executed by our pipeline scheduler are automatically generated from the operator-level graph and the optimized solution.

Besides, Parmesan supports writing/reading computation graphs, device topologies, and optimized solutions. Thus, one can further explore some more algorithms/flows and evaluate their performance based on Parmesan. Meanwhile, Parmesan's optimizer and simulator are independent of the deep learning framework. Provided the computation graph extracted by other DL frameworks (like Tensorflow), Parmesan can automatically conduct model partitioning and device mapping for the given network and simulate the solution performance.

TABLE I
A COMPARISON WITH RELEVANT BASELINES ON THE TWO-LEVEL HIERARCHICAL ARCHITECTURE. SPEEDUP BY OUR MAPPING IS MARKED IN BROWN.

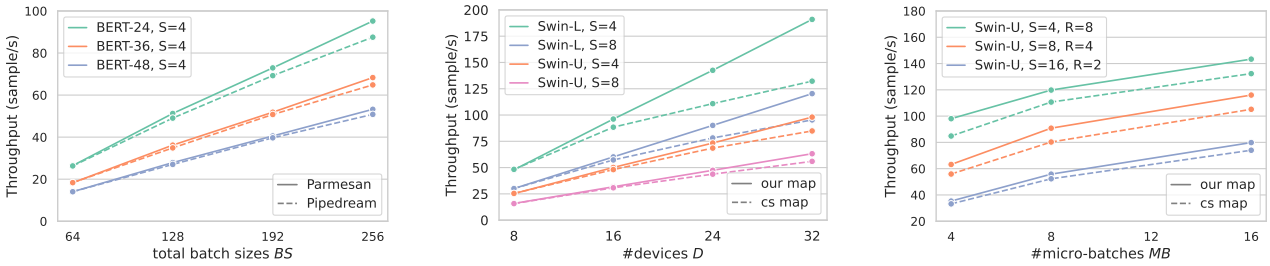
Task	Res152 Classifi.			BERT-L Pre-train			Swin-L Pre-train		
(S, R)	(4,4)	(8,2)	(16,1)	(4,4)	(8,2)	(16,1)	(4,4)	(8,2)	(16,1)
Pipedream	1.0x	1.0x	1.0x	1.0x	1.0x	1.0x	1.0x	1.0x	1.0x
+ our map	7.6x	2.4x	1.0x	1.0x	1.0x	1.0x	2.5x	1.4x	1.2x
RaNNC	0.9x	1.3x	0.8x	0.8x	0.9x	0.9x	-	-	-
+ our map	7.2x	2.5x	0.9x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

VIII. 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. We mainly conduct experiments on ResNet [1], BERT [2] and Swin Transformer [3]. Two real machine architectures and a series of synthetic architectures are considered in the experiments. The detailed settings of different NNs and different device topologies are described in Appendix D.

A. Validation on Different Device Topologies

Two-level hierarchical architecture. We compare the throughput of Parmesan with RaNNC (operator-level) and



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

Fig. 4. Throughput of different BERTs and Swins on 4x DGX-1 Architecture.

Pipedream (layer-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. The performance of these two baselines, RaNNC and Pipedream, are then compared with the full flow of Parmesan. 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 I).

We measure the relative speedup of the synchronous pipeline training on a two-level hierarchical architecture (described in Appendix D-B), and the results are shown in Table I. By default, we apply CS map for Pipedream and RaNNC. To demonstrate the 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. However, Parmesan still obtains $1.1\times$ speedup compared to Pipedream due to a better-balanced model partitioning for BERT. 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), CS map, as a kind of allreduce-first mapping, is no longer suitable for these cases. In contrast, our mapping algorithm can optimally map the partitions without human-in-the-loop. Note that our mapping results on Pipedream's Swin-L $(8, 2)$ and $(16, 1)$ and RaNNC's ResNet-152 $(16, 1)$ are neither consecutive mapping nor p2p-first, which further demonstrates the effectiveness of our mapping algorithm comparing to human-designed heuristic. Besides, 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. Finally, the results of our full-flow algorithm compared to other baselines demonstrate the superiority of Parmesan.

TABLE II
A COMPARISON WITH PIPEDREAM ON 4X DGX-1. "PD" IS SHORT FOR PIPEDREAM AND "PAR" IS SHORT FOR PARTITIONING.

Task	Swin-L Pre-train			Swin-U Pre-train		
(S, R)	(4,8)	(8,4)	(16,2)	(4,8)	(8,4)	(16,2)
PD Par + CS map	1.0x	1.0x	1.0x	1.0x	1.0x	1.0x
PD Par + Our map	**2.7x**	**2.5x**	1.3x	**1.9x**	**1.3x**	1.2x
Our Par + CS Map	1.9x	1.9x	1.2x	1.6x	1.2x	1.4x
Parmesan	**2.7x**	**2.5x**	**1.5x**	**1.9x**	**1.3x**	**1.5x**

4x DGX-1 architecture. We conduct larger-scale experiments on the Alibaba cloud 4x DGX-1 architecture, in which the throughput under different settings are compared. 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 (described in Appendix D-B). To examine the scalability of Parmesan, we enlarge the BERT-24 (i.e. BERT-L) to BERT-48 with 48 transformer layers with 667M parameters. We also enlarge Swin-L 2.15x to construct Swin-U which contains 424M parameters. The detailed settings of our enlarged models are given in Appendix D-A.

Fig. 4 shows a comparison of the throughput under different settings on the 4x DGX-1 architecture. In Fig. 4(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 change the number of replicas R and the number of BERT layers to measure the throughput under different total batch sizes BS ($BS = mBS \times R \times MB$). This experiment demonstrates the effectiveness of our full-flow algorithm when scaling up the BERT size and the total batch size compared to Pipedream + CS Map. In Fig. 4(b), we use our partitioning algorithm and fix $mBS = 16, MB = 4$, then we modify the number of devices D ($D = S \times R$) to evaluate the throughput of different Swin models under different mapping algorithms. Our mapping algorithm consistently achieves better solution qualities when scaling to large systems and large models compared to CS Map since our mapping algorithm takes the heterogeneity of the intra-DGX-1 connection into account. In Fig. 4(c), we keep using our partitioning algorithm and measure the Swin-U throughput under various (S, R, MB)

TABLE III
SIMULATION RESULTS FOR DIFFERENT TOPOLOGY. SPEEDUP BY OUR MAPPING OVER THE 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

configurations and different mapping algorithms while maintaining $mBS = 16, D = 32$. The results show that our mapping algorithm consistently speeds up the throughput of CS Map under different (S, R, MB) configurations. All these experiments demonstrate the high extensibility and high efficiency of Parmesan.

Besides, we compare the throughput with Pipedream on 4x DGX-1 architecture shown in Table II. Compared to the CS map, our mapping algorithm consistently obtains remarkable speedup on the partitioning results generated by both two algorithms (ours and Pipedream's). Since Pipedream's partition needs to communicate between stages more intensively while 4x DGX-1 architecture has extremely large intra-node bandwidth (double NVLink), the inter-stage communication overhead of Pipedream's partitions is significantly reduced by applying our mapping. The remaining factor that mainly affects the throughput is whether the computation is balanced or not. Since Pipedream's partition and our partition are both computationally balanced, the throughput of Pipedream's partitioning + our mapping looks close to ours. Nevertheless, our partitioning algorithm still considers communication overhead and achieves significant speedup compared to Pipedream when fixing the mapping as CS map. The results further demonstrate the effectiveness of our mapping algorithm and our operator-level partitioning scheme.

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 hardwares due to unavailability or incompatibility with the commonly used programming interface. 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. The experiments are conducted on SemanticFPN [28] whose partitioning results include more skip-connections, bringing more challenges to solving the problem. We randomly generate three kinds of fully heterogeneous topologies, named random_blk_1, random_blk_2 and uniform_dist (described in Appendix D-B). As shown in Table III, our mapping algorithm consistently yields significant speedup over CS mapping on various topologies. The results show that our proposed general topology mapping algorithm can solve the mapping problem once and for all without human-in-the-loop.

TABLE IV
THE RELATIVE SPEEDUP OF OUR REFINEMENT TECHNIQUE. "RF" IS SHORT FOR REFINEMENT.

	Res152 Classfi.			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)
w/o RF	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x	1.00x
w/ RF	**1.04x**	**1.09x**	**1.14x**	**1.00x**	**1.17x**	**1.25x**	**1.38x**	**1.28x**	**1.12x**

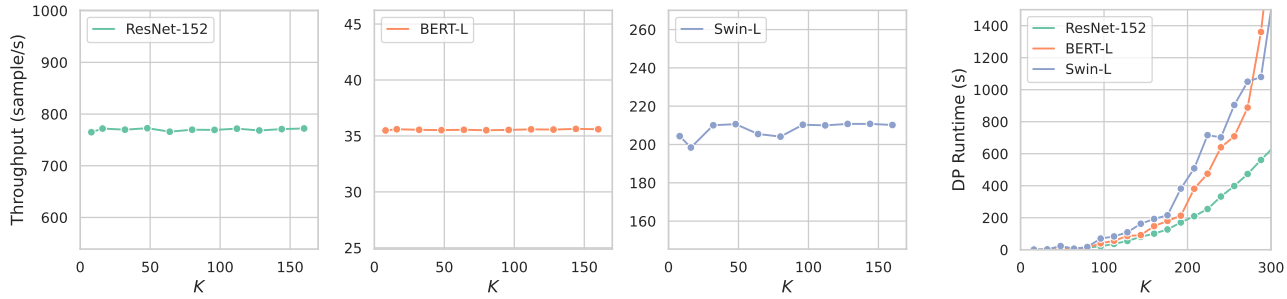
B. Ablation Studies on Model Partitioning

Effectiveness of Clustering. We measure the pipeline training throughput under different settings of K on the two-level hierarchical architecture. We fix the device mapping for each group with the same value of $S = 4, R = 4$. As shown in Fig. 5(a), changing K has a modest impact on throughput, remaining within a relatively narrow range. When K reaches a sufficiently large value, the alterations in throughput are negligible. This observation indicates that $K \gg S$ can ensure sufficient space for DP to yield high-quality solutions. Meanwhile, we also measure the impact on the DP running time of changing K to demonstrate that operator clustering can significantly improve the efficiency of DP. As shown in Fig. 5(b), the DP runtime (s) increases drastically with increasing the K. These two experiments illustrate that operator clustering is able to reduce the running time of DP with very minimal effect on the partitioning quality.

Effectiveness of Refinement. We measure the relative speedup of pipeline training throughput on the two-level hierarchical architecture to demonstrate the effectiveness of refinement. For each DNN model, we use the same K to generate the model partitions and fix the device mapping for each group (S, R). As shown in Table IV, refinement can significantly increase the throughput for most cases. The results demonstrate the effectiveness of the refinement step in fixing the partitioning result obtained by operator clustering and DP, and mitigating the sub-optimality gap brought by operator clustering.

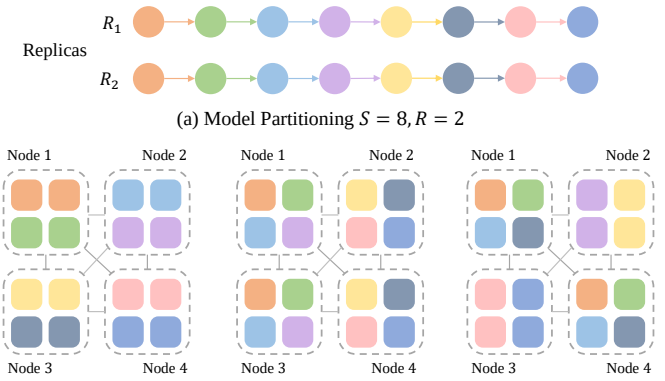
C. Comparison to Heuristic Mappings

Our previous discussion mainly compares with CS map, which is a kind of allreduce-first mapping. However, some networks are not allreduce-heavy, so applying the CS map to them may not be suitable. For example, a model with large inter-stage (p2p) communication (e.g., ResNet and Swin) may further speed up by other mappings. Intuitively, the most straightforward way is to develop a heuristic-based p2p sequential mapping (p2p map) for such cases. However, it is still non-trivial for humans to automatically select which heuristics to map. Although the ensemble of CS map (allreduce-first) and p2p map (p2p-first) works well in most cases on the two-level hierarchy experiments in Section VIII-A, for some cases, they are not the best. In contrast, our mapping algorithm generally generates the best mapping solution among all the aforementioned mappings. As cases shown in Table V, our mapping



(a) Throughput (sample/s) (b) DP runtime (s)

Fig. 5. (a) The impact of different K on the pipeline throughput for different models. (b) The DP runtime (s) of different models with setting different K.



(a) Model Partitioning $S = 8, R = 2$

(b) CS Mapping 1.00 × (c) p2p Mapping 1.04 × (d) Our Mapping 1.38 ×

Fig. 6. A visualization of different mapping algorithms on a two-level hierarchical architecture for Pipedream's Swin-L $(8, 2)$ partitioned results.

algorithm maps the model partitioning in a non-regular manner and produces a better solution than the previous two heuristics. Note that p2p map is identical to CS map when $R = 1$. A visualization of different mapping algorithms for Pipedream's Swin-L $S = 8, R = 2$ partitioned results are illustrated in Fig. 6. It is also non-trivial for humans to invent a heuristic to produce a solution with similar performance as Fig. 6(d).

Admittedly, if we compare our mapping with the integration of the CS map and p2p map, except in the cases mentioned above, the other two-level hierarchy experiments might be seemingly underwhelming. However, when tackling the non-regular architecture (grid-based and random-based), our mapping algorithm shows its effectiveness compared to these two heuristics. We compare the ensembling solution (the solution with the best simulated throughput) of the two instantiations of our mapping algorithm with the ensembling results of the two heuristics (CS map and p2p map) under SemanticFPN on the non-regular architecture. As shown in Table VI, our mapping successfully speeds up the simulated throughput of most cases compared to the ensemble heuristic. The results not only indicate the limitation of human-designed heuristics but demonstrate the strong demand for our effective general topology mapping algorithm.

D. Flow Runtime

Parmesan successfully generates partitioning and mapping solutions within 60 minutes for all the abovementioned experiments (Section VIII). The average runtime of these experiments is around 8 minutes. Compared to the pipelined training time of recently advanced DNNs which requires hundreds of hours, our optimization process only consumes a small portion of that time but brings significant speedup.

TABLE V
THE CASES THAT OUR MAPPING ALGORITHM MAPS THE MODEL PARTITIONING IN A NON-REGULAR MANNER (I.E., NEITHER P2P-SEQUENTIAL NOR ALLREDUCE-FIRST).

Model	(S, R)	Partition	CS Map	p2p Map	Our Map
Res152	(16,1)	RaNNC	1.00x	1.00x	**1.12x**
Swin-L	(8,2)	Pipedream	1.00x	1.04x	**1.38x**
Swin-L	(16,1)	Pipedream	1.00x	1.00x	**1.17x**

TABLE VI
SIMULATION RESULTS FOR DIFFERENT TOPOLOGIES. SPEEDUP BY OUR MAPPING OVER THE ENSEMBLING RESULTS OF CS MAP AND P2P MAP 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	1.0x	1.0x	1.2x	7.0x	2.5x	1.4x	-	-	-
2d torus	1.0x	1.0x	1.0x	1.0x	1.1x	1.0x	-	-	-
3d mesh	1.0x	1.0x	1.2x	-	-	-	1.6x	1.8x	1.0x
3d torus	1.0x	1.0x	1.0x	-	-	-	1.0x	1.0x	1.0x
random_blk_1	1.1x	1.0x	1.0x	1.3x	1.0x	1.0x	1.2x	1.0x	1.0x
random_blk_2	2.1x	1.8x	1.0x	1.1x	1.1x	2.1x	1.2x	1.1x	1.7x
uniform_dist	16.0x	11.7x	6.7x	12.1x	5.1x	9.8x	17.5x	7.7x	5.5x

E. Simulator's Accuracy

We evaluate the simulator's accuracy by comparing the simulated results with real-world execution. We measure the real-world training throughputs on different models, different (S, R) configurations, and different mappings on the two-level hierarchy and compare them with the simulated results. The results are shown in Fig. 7. Spearman's rank correlation coefficient between simulated and real-world throughput is around 0.95. There indeed exists small variations for some cases, but it should be understandable since there could be many factors that affect the real measured throughput due to, e.g., non-determinism in networking. Our simulator aims to give a reference value that is useful for developing an optimizer and exploring new partitioning and mapping algorithms.

With our well-designed optimizer, simulator, pipeline scheduler, and unified IO format, we believe our proposed general

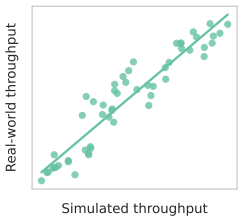


Fig. 7. A comparison between the simulated and real-world throughput.

device topology mapping algorithm will have a high potential to provide architecture designers with valuable insights for pre-evaluating their hardware and assist them in designing more DNN-friendly architectures.

IX. 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 these DP-based partitioning works still lack consideration of 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 design flow 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 will explore more pruning strategies for device mapping, exploit learning-assisted device mapping techniques, consider co-optimization of model partitioning and device mapping, and take pipeline bubble into account. We also plan to consider the heterogeneity for both device computational behaviors and interconnections in the future.

Appendix A
Proofs

A. Proof of Theorem 1

Proof. For any vertex u in $\mathcal{G}$, there is always a fronted graph $\mathcal{F}_u$ whose $\mathcal{V}_{\mathcal{F}_u} = \{u\} \cup \{v \mid v \text{ is a predecessor of } u\}$. We claim that $\forall\ u, v \in \mathcal{V}$, $\mathcal{F}_u \neq \mathcal{F}_v$. Then we can easily derive $|F_{\mathcal{G}}| \geq |\mathcal{V}|$ because $\mathcal{G}$ has $|\mathcal{V}|$ unique vertices. Now we prove the claim that $\forall\ u, v \in \mathcal{V}$, $\mathcal{F}_u \neq \mathcal{F}_v$. Suppose $\exists\ u, v$, s.t. $\mathcal{F}_u = \mathcal{F}_v$. Since $\mathcal{F}_u = \mathcal{F}_v$, we can get $u \in \mathcal{V}_{\mathcal{F}_v}$ and $v \in \mathcal{V}_{\mathcal{F}_u}$, then we have $\{u, v\} \subseteq \mathcal{V}_{\mathcal{F}_u}$ and $\{u, v\} \subseteq \mathcal{V}_{\mathcal{F}_v}$. This implies v is a predecessor of u and u is a predecessor of v, that is, there is a cycle in $\mathcal{G}$. This contradicts that $\mathcal{G}$ is a DAG. Thus $\forall\ u, v \in \mathcal{V}$, $\mathcal{F}_u \neq \mathcal{F}_v$. □

B. Proof of Theorem 2

Proof. Suppose v_i is the i-th vertex in the unique topological ordering of $\mathcal{G}$, we have $\mathcal{V} = \{v_1, v_2, ..., v_{|\mathcal{V}|}\}$. For any $\mathcal{F} \in F_{\mathcal{G}}$, if $v_i \in \mathcal{F}$ and $i \geq j$ for any $v_j \in \mathcal{V}_{\mathcal{F}}$, we have $\mathcal{V}_{\mathcal{F}} = \{v_1, ..., v_i\}$, which implies the vertex with the largest topological ordering will determine $\mathcal{F}$ uniquely. Because $\mathcal{G}$ has $|\mathcal{V}|$ vertices and their topological ordering are distinct and unique, we have $|F_{\mathcal{G}}| = |\mathcal{V}|$. □

C. Proof for Optimality of Algorithm 1

Proposition 2. Algorithm 1 always returns the optimal solution p^* of Problem (5), if the initial interval $[t_l^{(0)}, t_r^{(0)}]$ contains the optimal value $t_{\max}(p^*) = \max_{s \in \mathcal{V}'} c_{\mathsf{stage}}(s, p^*)$ of Problem (5).

Proof. We firstly prove that $t_{\max}(p^*) \in [t_l^{(i)}, t_r^{(i)}], \forall i$ by induction, where i is the iteration number of the while-loop in Algorithm 1. The base case $i = 0$ is already ensured as the precondition. For the inductive step, suppose $t_{\max}(p^*) \in [t_l^{(i)}, t_r^{(i)}]$ after iteration i. Consider two possible cases in iteration $i+1$: the inner-level search either founds or is unable to found a solution for target $t \in [t_l^{(i)}, t_r^{(i)}]$. For the former case, $t_{\max}(p^*) \leq t_p \leq \min\{t, t_p\} = t_r^{(i+1)}$, and since $t_{\max}(p^*) \geq t_l^{(i)} = t_l^{(i+1)}$, we have $t_{\max}(p^*) \in [t_l^{(i+1)}, t_r^{(i+1)}]$. For the latter case, there is no solution p such that $t_p \leq t$, i.e., $t_{\max}(p^*) \geq t = t_l^{(i+1)}$. Combined with $t_{\max}(p^*) \leq t_r^{(i)} = t_r^{(i+1)}$, we obtain $t_{\max}(p^*) \in [t_l^{(i+1)}, t_r^{(i+1)}]$.

It is easy to see that the interval $[t_l^{(i)}, t_r^{(i)}]$ is always shrinking with the increment of i, i.e., $t_l^{(i+1)} \geq t_l^{(i)}, t_r^{(i)} \leq t_r^{(i+1)}, \forall i$. When the length of the interval decreases to a value small enough (ϵ in Algorithm 1) such that no more solution can be found that has an objective value within this interval (since there is only a finite number of possible solutions), the last found solution is the optimal one. □

D. Proof of Proposition 1

Proof. We firstly prove $t_{l0}^{\mathsf{m}} \leq \min_p \max_{s \in \mathcal{V}'} c_{\mathsf{stage}}^{\mathsf{m}}(s, p)$. For any stage $s \in \mathcal{V}'$ and any mapping p, we have

$$\min_d \min_{p_{s,d}^{\mathsf{m}}} \sum_{s' \in \mathrm{adj}(s)} \frac{M(s, s')}{B(d, p_{s,d}^{\mathsf{m}}(s'))} \leq \sum_{s' \in \mathrm{adj}(s)} \frac{M(s, s')}{B(p(s), p(s'))}.$$

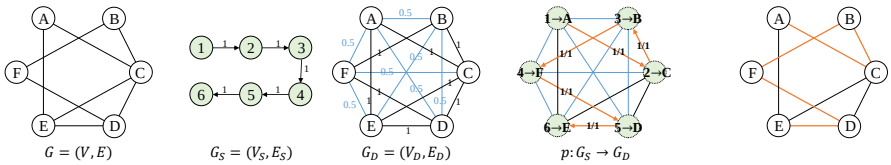
This is because the LHS is essentially the smallest possible inter-stage communication time of s (note that the LHS is independent of p), and the RHS is the inter-stage communication time of s under p. Since this inequality holds for any $s \in \mathcal{V}'$, we get

$$\begin{aligned} t_{l0}^{\mathsf{m}} &= \max_{s \in \mathcal{V}'} \left\{ c_{\mathsf{u}}(s) + \min_d \min_{p_{s,d}^{\mathsf{m}}} \sum_{s' \in \mathrm{adj}(s)} \frac{M(s, s')}{B(d, p_{s,d}^{\mathsf{m}}(s'))} \right\} \\ &\leq \max_{s \in \mathcal{V}'} \left\{ c_{\mathsf{u}}(s) + \sum_{s' \in \mathrm{adj}(s)} \frac{M(s, s')}{B(p(s), p(s'))} \right\} \\ &= \max_{s \in \mathcal{V}'} c_{\mathsf{stage}}^{\mathsf{m}}(s, p). \end{aligned}$$

Also, the above inequality holds for any p, and thus we obtain $t_{l0}^{\mathsf{m}} \leq \min_p \max_{s \in \mathcal{V}'} c_{\mathsf{stage}}^{\mathsf{m}}(s, p)$ as desired.

For the second inequality, we claim that

$$\begin{aligned} &\min_d \min_{p_{s,d}^{\mathsf{AR}}} \max_{s' \in \mathrm{adj\_repl}(s)} \left\{ 2 \frac{R-1}{R} \frac{P(s)}{B(d, p_{s,d}^{\mathsf{AR}}(s'))} \right\} \\ \leq &\max_{\substack{s_i, s_{i+1} \\ \in \mathrm{ring}(\mathrm{repl}(s))}} \left\{ 2 \frac{R-1}{R} \frac{P(s)}{B(p(s_i), p(s_{i+1}))} \right\} \end{aligned}$$



(a) Hamiltonian Path Problem on G (b) Device Mapping Problem between graph G_S and graph G_D (c) Solution $p: G_S \to G_D$ for Device Mapping Problem on G_S, G_D (d) Solution for Hamiltonian Path Problem on G

Fig. 8. The Hamiltonian path problem on G can be reduced to the device mapping problem between the stage graph G_S and the device topology graph G_D. The mapping solution p of DMP with a maximum stage time (= communication data size / bandwidth) less than or equal to 1 will give a path connecting all the corresponding vertices in G.

holds for any stage $s \in \mathcal{V}'$ and any mapping p, since the LHS is no greater than the smallest possible inter-replica communication time of s (note that on the LHS, the maximization is over a subset of the ring only), and the RHS is the inter-replica communication time of s under p. The remaining part of the proof essentially follows the argument in the proof of the first inequality. □

E. Proof of Theorem 3

Proof. We are going to prove the NP-completeness of the decision problem version of the Device Mapping Problem (DMP) by reducing from the Hamiltonian Path Problem on undirected graph $G(V, E)$. The decision problem version of DMP is to decide whether there is a mapping p that the maximum stage time is less than or equal to a threshold t,

$$\max_{s \in \mathcal{V}'} \ c_{\mathsf{stage}}(s, p) \leq t. \tag{12}$$

First of all, the decision version of DMP is NP since the correctness of a mapping can be checked in polynomial time. Next, we can reduce the Hamiltonian Path Problem to the decision version DMP by (1) putting $n = |V|$ and there is a corresponding device d_i for each node $v_i \in V$ for $i = 1 \dots n$, (2) setting the computation times of all the stages to zero; (3) for any edge $(v_i, v_j) \in E$, put the bandwidth between device d_i and d_j as 1, while the bandwidths of all the remaining pairs of devices are 0.5; (4) putting the threshold t as 1 and having the set of stage S just n stages $\{s_1, \dots, s_n\}$ such that there is data communication of size 1 from s_i to s_{i+1} for all $i = 1 \dots n-1$ only. It is not hard to see a Hamiltonian path in G will correspond to a mapping p of the n stages to the set of n devices with a maximum stage time less than or equal to 1, and the vice versa that a mapping p will give a path connecting the n corresponding vertices in G. □

Fig. 8 gives an example of the above reduction.

Appendix B
Selection Criteria for S and R

In Section IV, we discuss how to partition a computation graph into S stages with R replicas. In this section, we will discuss how to search for the number of stage S and the number of replicas R.

Before introducing the searching method, we first give the definition of stage replica number. Stage $s \in \mathcal{S}$ has R_s stage replicas implies there are R_s devices maintaining a replica of the entire stage s. However, a different replica number R_s for each stage s will invoke two extra collective communication operators, *allscatter* and *allgather*, for transmitting tensors between two adjacent stages, which will cause high communication overhead. Such communication overhead will negatively affect the real training throughput, especially for synchronous pipeline training. Therefore it is better to have the same stage replica number for all stages. In our implementation, we consider R_s as a constant R among all the stages so the DP in Equation (2) can be simplified as follows,

$$T_{\mathcal{F},d,s} = \min_{\mathcal{F}' \in F_{\mathcal{F}} \setminus \{\mathcal{F}\}} \min_{\substack{\forall d' \in [s-1, d-R] \\ s.t.\ d' \in \mathbb{N}}} \max\{T_{\mathcal{F}',d',s-1}, t_{\mathcal{F}-\mathcal{F}',R}\} \tag{13}$$

With Equation (13), our dynamic programming can optimally partition the DNN into S stages with R replicas. Note that Equation (13) solves a sub-problem of Equation (2). Since the extra collective communication overhead caused by having different stage replicas is expensive in real-world training, Equation (13) is applied in Parmesan.

As for searching the best configuration (S, R), we develop a simulator to estimate the pipeline training latency offline. We can then choose a configuration having the shortest estimated latency as the best configuration. Note that the pipeline bubble is also considered during simulation. Since different configurations can be executed independently, such a search can be well-parallelised by multi-threading and can be finished within a reasonable amount of time.

Appendix C
Peak Memory Consumption Model

Given the number of micro-batches MB and a computation graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, the peak memory consumption of a subgraph $\mathcal{H} \subseteq \mathcal{G}$ is given as,

$$m(\mathcal{H}) = \frac{1}{MB} \sum_{v \in \mathcal{V}_\mathcal{H}} \{\max\{m_{\mathsf{fwd}}(v), m_{\mathsf{bwd}}(v)\} + \sum_{\substack{v' \in \mathrm{adj}(v) \\ v' \notin \mathcal{V}_\mathcal{H}}} M(v, v')\} \tag{14}$$

where $m_{\mathsf{fwd}}(v)$ and $m_{\mathsf{bwd}}(v)$ denote operator v's peak memory consumption during forward and backward propagation respectively, $M(v_i, v_j)$ denotes the communication size between operator v_i and operator v_j. This model approximates the peak memory usage of a subgraph $\mathcal{H} \subseteq \mathcal{G}$ in real-world training with *activation recomputing*.

TABLE VII
DNN MODELS USED IN TWO-LEVEL HIERARCHICAL AND NON-REGULAR ARCHITECTURES

	ResNet-152	Swin-L	SemanticFPN	BERT-L
Layers	[3, 8, 36, 3]	[2, 2, 18, 2]	Backbone: ResNet-50	24
Embed Dim	/	192	/	1024
Heads	/	[6, 12, 24, 48]	/	16
K	32	32	32	48
#Params	60M	197M	30M	365M
#Classes	1000	1000	150	/
Image Size	224x224x3	224x224x3	512x512x3	/
Vocab Size	/	/	/	30533
Seq Length	/	/	/	512
Micro-Batch Size	64	32	16	4

TABLE VIII
DNN MODELS USED IN CLOUD 4X DGX-1 ARCHITECTURE

	Swin-L	Swin-U	BERT-24	BERT-36	BERT-48
Layers	[2, 2, 18, 2]	[2, 2, 24, 2]	24	36	48
Embed Dim	192	256	1024	1024	1024
Heads	[6, 12, 24, 48]	[8, 16, 32, 64]	16	16	16
K	32	48	48	72	96
#Params	197M	424M	365M	516M	667M
#Classes	1000	1000	/	/	/
Image Size	384x384x3	384x384x3	/	/	/
Vocab Size	/	/	30533	30533	30533
Seq Length	/	/	512	512	512
Micro-Batch Size	16	16	8	8	8

APPENDIX D
EXPERIMENTAL SETTINGS

A. Model Settings

We conduct experiments on ResNet and SwinTransformer, BERT and SemanticFPN[3]. The model settings for experiments on two-level hierarchical architecture and simulation on non-regular architecture are given in Table VII, and the model settings for experiments on Alibaba cloud 4x DGX-1 architecture are given in Table VIII. Note that we generate Swin-U, BERT-36 and BERT-48 by modifying the open source code, and BERT-24 is identical to BERT-L. We use SGD as these DNNs' optimizers.

B. Device Topologies

Two-level hierarchical architecture is a four-machine system that has two Xeon 4114 CPUs and four Titan V GPUs per node. GPUs are connected by PCI-e intra-node. The inter-node connection is Ethernet. According the profiled results, intra-node bandwidth is 11GB/s, inter-node bandwidth is 1.1GB/s. An illustration of two-level hierarchical architecture is shown in Fig. 9(a).

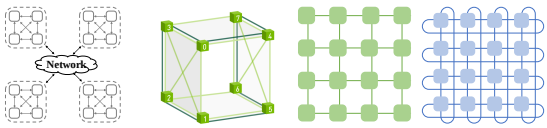
Alibaba cloud 4x DGX-1 architecture consists of 4 ecs.gn6e-c12g1.24xlarge virtual machine instances, where each machine contains 8 V100 GPUs. There are three types of bandwidth in intra-node, that is, PCI-e, single NVLinks and double NVLinks, their profiled bandwidth are 10.1GB/s, 21.4GB/s and 43.2GB/s separately. The inter-node connection relies on Ethernet and its profiled bandwidth is 1.4GB/s. Fig. 9(b) illustrates its intra-node connections[4]. Each corner

[3]Codes of ResNet, SwinTransformer, BERT are from TorchVision library, https://github.com/microsoft/Swin-Transformer, and https://github.com/codertimo/BERT-pytorch respectively. And we re-implement SemanticFPN following MMSegmentation (https://github.com/open-mmlab/mmsegmentation) and TorchVision library.

[4]Fig. 9(b) is from DGX-1 Whitepaper in https://www.nvidia.com/en-us/data-center/dgx-1/.

TABLE IX
SYNTHESISED BANDWIDTHS MESH/TORUS ARCHITECTURE

#Hops	1	2	3	4	5	6	7	8
Bandwidth GB/s	78.1	39.0	24.4	14.6	9.77	7.81	5.86	4.4
#Hops	9	10	11	12	13	14	15	16
Bandwidth GB/s	2.93	1.46	0.88	0.78	0.68	0.59	0.49	0.39
#Hops	17	18	19	20	21-31	31-51	≥ 52	
Bandwidth GB/s	0.29	0.19	0.098	0.098	0.088	0.078	0.068	



(a) Two-level hierarchy (b) intra-DGX-1 connection (c) 2d mesh (d) 2d torus

Fig. 9. Visualization of different device topologies.

denotes one V100 GPU, a single edge denotes a single NVLink connection, and a double-edge denotes a double NVLink connection. Device pairs without direct edge in Fig. 9(b) are connected by PCI-e.

Mesh/torus architectures. Since it's difficult for us to access a real mesh/torus kind system, we set the bandwidth between any two devices according to the number of hops between them. Note that Table IX is generated by us as an example and it may not be the same as the real mesh/torus architectures. Examples of 4×4 2d mesh/tours architectures are shown in Fig. 9(c) and Fig. 9(d).

Randomly generated architectures. We randomly generated three kinds of fully heterogeneous device topologies, named random_blk_1, random_blk_2 and uniform_dist. (1) *random_blk_1* contains several nodes with randomly generated intra-node bandwidth and fixed inter-node bandwidth. Suppose the number of devices is D, we firstly generate a list of random integer $[b_1, b_2, ..., b_L]$ whose summation is D, where L denotes the number of nodes and b_i denotes the number of devices in node n_i. We then randomly generated L intra-node bandwidths for all nodes following a uniform distribution $\mathcal{U}_{[1e-4,1e-2]}$ MB/us (1 GB/s = $1024/10^6$ MB/us) and set the inter-node bandwidth as $1e-4$ MB/us. Note that the only homogeneity in this architecture is the intra-node connection for each node n_i. (2) *random_blk_2* also needs a list of random integer $[b_1, b_2, ..., b_L]$ similar to random_blk_1,. Besides, the intra-node bandwidths in random_blk_2 are randomly generated following $\mathcal{U}_{[1e-5,1e-2]}$ MB/us. Note that the intra-node bandwidths for each node n_i are heterogeneous. Then we compute the average value of all the intra-node bandwidths as $\overline{BW}_{intra}$, and let the base inter-node bandwidth as $BW_{inter} = \overline{BW}_{intra}/10$. The inter-node bandwidth between node n_i and n_j is set as $BW_{inter}/|i-j|$. (3) *uniform_dist*: all the device-to-device bandwidths are randomly generated following $\mathcal{U}_{[1e-5,1e-2]}$ MB/us.

Visualizations for different device topologies' bandwidth lookup-tables (a table to represent the device-to-device bandwidths) are shown in Fig. 10.

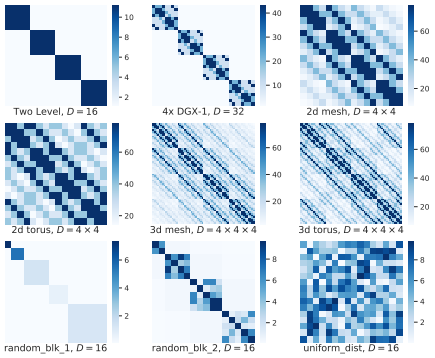


Fig. 10. Visualizations for different architectures' bandwidth look-up-tables (a table to represent the device-to-device bandwidths).

REFERENCES

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. CVPR*, 2016.

[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[3] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin transformer: Hierarchical vision transformer using shifted windows," in *Proc. ICCV*, 2021.

[4] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, "A unified architecture for accelerating distributed dnn training in heterogeneous gpu/cpu clusters," in *Proc. OSDI*, 2020.

[5] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *Proc. SIGKDD*, 2020.

[6] S. Athlur, N. Saran, M. Sivathanu, R. Ramjee, and N. Kwatra, "Varuna: Scalable, low-cost training of massive deep learning models," in *Proc. EuroSys*, 2022.

[7] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch sgd: Training imagenet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.

[8] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[9] M. Zinkevich, M. Weimer, L. Li, and A. Smola, "Parallelized stochastic gradient descent," in *Advances in Neural Information Processing Systems*, 2010.

[10] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project adam: Building an efficient and scalable deep learning training system," in *Proc. OSDI*, 2014.

[11] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks." in *Proc. MLSys*, 2019.

[12] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *arXiv preprint arXiv:1909.08053*, 2019.

[13] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and z. Chen, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," in *Advances in Neural Information Processing Systems*, 2019.

[14] S. Li and T. 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*, 2021.

[15] Z. Li, S. Zhuang, S. Guo, D. Zhuo, H. Zhang, D. Song, and I. Stoica, "Terapipe: Token-level pipeline parallelism for training large-scale language models," in *Proc. ICML*, 2021.

[16] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia, "Memory-efficient pipeline-parallel dnn training," in *Proc. ICML*, 2021.

[17] S. Fan, Y. Rong, C. Meng, Z. Cao, S. Wang, Z. Zheng, C. Wu, G. Long, J. Yang, L. Xia, L. Diao, X. Liu, and W. Lin, "Dapple: a pipelined data parallel approach for training large models," in *Proc. PPoPP*, 2021.

[18] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. A. Zaharia, "Pipedream: generalized pipeline parallelism for dnn training," in *Proc. SOSP*, 2019.

[19] J. M. Tarnawski, D. Narayanan, and A. Phanishayee, "Piper: Multidimensional planner for dnn parallelization," in *Advances in Neural Information Processing Systems*, 2021.

[20] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," *Journal of Parallel and Distributed Computing*, 2009.

[21] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device placement optimization with reinforcement learning," in *Proc. ICML*, 2017.

[22] Y. Gao, L. Chen, and B. Li, "Spotlight: Optimizing device placement for training deep neural networks," in *Proc. ICML*, 2018.

[23] A. Paliwal, F. Gimeno, V. Nair, Y. Li, M. Lubin, P. Kohli, and O. Vinyals, "Reinforced genetic algorithm learning for optimizing computation graphs," in *International Conference on Learning Representations*, 2019.

[24] M. Tanaka, K. Taura, T. Hanawa, and K. Torisawa, "Automatic graph partitioning for very large-scale deep learning," *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021.

[25] L. Zheng *et al.*, "Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning," in *Proc. OSDI*, 2022.

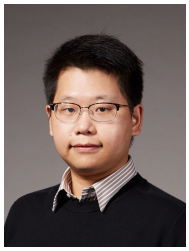
[26] A. Bar-Noy and S. Kipnis, "Designing broadcasting algorithms in the postal model for message-passing systems," in *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, 1992.

[27] W. Gropp, L. N. Olson, and P. 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*, 2016.

[28] A. Kirillov, R. Girshick, K. He, and P. Dollár, "Panoptic feature pyramid networks," in *Proc. CVPR*, 2019.

Lixin Liu received his B.Eng. degree from Electronics Science and Technology, South China University of Technology in 2019, and Ph.D. degree from the Department of Computer Science and Engineering, The Chinese University of Hong Kong (CUHK) in 2023. His research interests include electronic design automation and distributed deep learning systems.

Tianji Liu received the B.Eng. degree from Tsinghua University in 2019 and M.Sc. degree from University College London in 2020 respectively. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, The Chinese University of Hong Kong. His research interests are parallel and efficient algorithms for logic synthesis.



Bentian Jiang received his B.Eng. degree from Electronics and Information Engineering, Sichuan University in 2017, and Ph.D. degree from the Department of Computer Science and Engineering, The Chinese University of Hong Kong (CUHK) in 2021. His research interests include VLSI datapath subsystem optimization, design for manufacturability and physical design.

Evangeline F.Y. Young (Fellow, IEEE) received the B.Sc. degree in computer science from The Chinese University of Hong Kong (CUHK), Hong Kong, and the Ph.D. degree from The University of Texas at Austin, Austin, TX, USA, in 1999.

She is currently a Professor with the Department of Computer Science and Engineering, CUHK. Her research interests include EDA, optimization, algorithms and AI. Her research focuses on floorplanning, placement, routing, DFM and EDA on physical design in general.

Dr. Young's research group has won championships and prizes in numerous renown EDA contests, including the CAD Contests at ICCAD, DAC and ISPD. She has served on the organization committees of ICCAD and ISPD, and on the program committees of conferences, including DAC, ICCAD, ISPD, DATE and ASP-DAC. She also served on the editorial boards of IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, ACM Transactions on Design Automation of Electronic Systems, and Integration, the VLSI Journal. She is a Fellow of IEEE.