Towards Optimal Placement and Scheduling of DNN Operations with Pesto

Ubaid Ullah Hafeez, Xiao Sun, Anshul Gandhi, Zhenhua Liu
Stony Brook University, Stony Brook, NY, USA
{ubaidullah.hafeez,xiao.sun,anshul.gandhi,zhenhua.liu}@stonybrook.edu

ABSTRACT

The increasing size of Deep Neural Networks (DNNs) has necessitated the use of multiple GPUs to host a single DNN model, a practice commonly referred to as model parallelism. The key challenge for model parallelism is to efficiently and effectively partition the DNN model across GPUs to avoid communication overheads while maximizing the GPU utilization, with the end-goal of minimizing the training time of DNN models. Existing approaches either take a long time (hours or even days) to find an effective partition or settle for sub-optimal partitioning, invariably increasing the end-to-end training effort. In this paper, we design and implement Pesto, a fast and near-optimal model placement technique for automatically partitioning arbitrary DNNs across multiple GPUs. The key idea in Pesto is to jointly optimize the model placement and scheduling at the fine-grained operation level to minimize inter-GPU communication while maximizing the opportunity to parallelize the model across GPUs. By carefully formulating the problem as an integer program, Pesto can provide the optimal placement and scheduling. We implement Pesto in TensorFlow and show that Pesto can reduce model training time by up to 31% compared to state-of-the-art approaches, across several large DNN models.

CCS CONCEPTS

• **Theory of computation** → **Integer programming**; **Linear programming**; **Scheduling algorithms**; • **Computing methodologies** → **Neural networks**.

KEYWORDS

systems for ML, model parallelism, DNN placement, scheduling, giant DNNs

ACM Reference Format:
Ubaid Ullah Hafeez, Xiao Sun, Anshul Gandhi, Zhenhua Liu. 2021. Towards Optimal Placement and Scheduling of DNN Operations with Pesto . In *22nd International Middleware Conference (Middleware '21), December 5–11, 2021, Québec city, QC, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3464298.3476132

1 INTRODUCTION

The past few years have seen a significant increase in the adoption of Deep Neural Networks (DNNs) for a wide range of machine learning applications, including image classification, language translation and modeling [19, 26, 36]. The success of DNNs is primarily due to the large models which have the capacity to learn complex features from big data. Modern DNN training frameworks, such as TensorFlow [9] and pyTorch [47], model DNNs as directed acyclic graphs (DAGs) where each node is a compute operation, typically run on a GPU, TPU, or CPU, and each edge represents data communication between operations (see Section 2).

Today's DNNs typically have a large number of layers resulting in billions of model parameters [29]. This tremendous growth in size of such "giant" DNN models complicates the already time- and resource-intensive DNN training process. An unavoidable and undesirable side-effect of the ever-increasing size of DNN models is the inability to fit these giant models on a single GPU. For instance, training Transformer [57] model for language translation with 103 layers requires more than 32 GB of GPU memory [29], suggesting that this model cannot fit in one (commodity) GPU [1, 3].

The growth in model sizes has necessitated the use of *multiple GPUs to host a single DNN model*, also referred to as *model parallelism* [15, 17, 38]. Under model parallelism, the model (or DNN graph) is partitioned among multiple GPUs such that each GPU evaluates only a subset of the model. Model parallelism is also attractive as it has the potential to improve resource (GPU) utilization and, consequently, reduce total training time [33, 44, 45].

The key challenge with model parallelism is that partitioning a model among GPUs is not easy as DNNs are not always structured as easily separable parallel branches. Further, when partitioning the graph, if two adjacent nodes are placed on different GPUs, a *communication overhead* is incurred. Manually finding the best partition for a given DNN model that minimizes the communication overhead while efficiently utilizing multiple GPUs is a challenging task, even for experienced domain experts or machine learning (ML) practitioners [45]. Typically, domain experts partition models across GPUs by assigning a subset of layers to each GPU [10, 31, 44, 45, 58], referred to as *Expert* strategy. However, because of the sequential nature of the layers of DNNs [29], such partitioning can result in under-utilization of compute resources while incurring significant communication overhead, resulting in large training times.

Prior work on partitioning DNN models has employed *learning-based* techniques to determine the placement of DNN operations on different GPUs [10, 44, 45]. While such learning-based techniques can outperform (manual) Expert strategies, they require *significant time* to find the placement, often on the order of days [45]. A recent work, Baechi [31], employed an algorithmic approach to significantly reduce the placement time of DNNs. However, the resulting

Middleware '21, December 5–11, 2021, Québec city, QC, Canada
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8534-3/21/12…$15.00
https://doi.org/10.1145/3464298.3476132

placements are not optimal, and can lead to training times that are worse than learning-based and Expert strategies.

This paper presents Pesto, a fast and near-optimal model placement technique for automatically partitioning arbitrary DNNs across multiple GPUs. In contrast to existing approaches, Pesto can find better placements in a few minutes. Pesto aims to maximize resource utilization by intelligently scheduling operations across GPUs while respecting their dependencies. Further, Pesto aims to minimize training time by opportunistically overlapping compute time with communication time (across GPUs) while taking memory constraints into account.

The key idea in Pesto is to jointly consider the model placement *and* scheduling of operations. Pesto formulates the placement and scheduling problem as an integer linear program (ILP). To model the system dynamics, we carefully craft communication congestion and memory constraints, a significant addition over similar graph scheduling approaches in the literature. By design, Pesto's ILP-generated placement and scheduling is *optimal*. To reduce the DNN training effort due to the time required to solve the ILP, we develop a cycle-free vertex merging technique to efficiently coarsen the model graph while still obtaining a near-optimal solution.

We implement Pesto in TensorFlow and experimentally evaluate its performance for eleven different variants across four giant DNN models. Compared to the Expert placement and Baechi, we find that Pesto reduces training time across all models by about 15.5% and 23.4%, respectively, on average. Further, we show that Pesto can outperform learning-based approaches in terms of the improvement in training time afforded over Expert. Finally, Pesto reduces the placement time from hours or days under learning-based approaches to a few minutes. To the best of our knowledge, Pesto is the first technique to find placements for RNNLM models [60] that reduce the training time significantly compared to the Expert strategy (see Section 5 for results).

To summarize, this paper makes the following contributions:

- We present a near-optimal, joint DNN placement and operation scheduling algorithm for arbitrary DNN models.
- We design an online model parallelism framework, Pesto, which minimizes model training time on multiple GPUs.
- We implement Pesto on TensorFlow and show that Pesto can reduce DNN model training time by 15.5% and 23.4%, on average, compared to the Expert strategy and Baechi.
- Given the increasing popularity of giant models, unlike existing DNN placement techniques [10, 31, 44, 45], we evaluate Pesto extensively across multiple giant models, each with different variants.

2 BACKGROUND

This section provides background on DNN training and why partitioning as well as scheduling of low level operations in DNN model graphs is important for parallel training.

2.1 Overview of DNN training

Deep Neural Nets (DNNs): DNNs are typically employed in applications such as image classification or language translation, and work by learning a relationship between the output variable(s) (such as detection of objects in the image) and the input features (such as pixels of images) [50, 61]. The relationship can be non-linear, and is parameterized using model weights or parameters, that are learned by the DNN, based on training over (typically) labeled data sets.

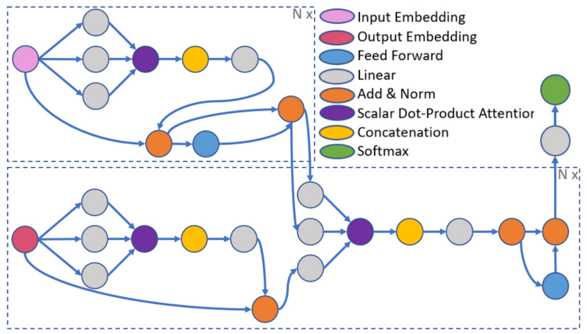


Figure 1: Illustration of the N-layered Transformer [57] model's DNN architecture.

The DNN model consists of sequence of layers of different types, with each layer consisting of several nodes, corresponding to computational tasks or operations. Figure 1 shows the simplified DAG of the Transformer model [57]. Here, the edges connecting different colored circles denote data flow between them. The colored circles in the figure represent a functional compute unit (see legend), which under TensorFlow could itself be composed of multiple smaller compute operations. As a result, the actual DAG is much more complex. For example, in our experiments, a 6 layered Transformer model results in a DAG of more than 19,000 operation nodes. Note the × multiplier within some of the (dotted boundary) layers; these indicate that the layer repeats N times in the model. Training generally consists of multiple iterations of the training data over the same model.

Model parallelism: Today's DNN models can be very large, or "giant" [19, 29, 57], requiring significant amount of GPU memory to host the model and making it infeasible to host the giant model on a single GPU [29, 44, 45]. Model parallelism is a technique that is ideal for emerging and giant DNN models that do not fit on one GPU [10, 44, 58]. Under model parallelism, the DNN graph is partitioned into subsets, and each subset is placed on a different GPU. Note that this is different from data parallelism whereby the entire model is replicated on each GPU, but each GPU only processes a subset of the training data. For giant models, since the model cannot fit in one GPU, data parallelism is infeasible. Under model parallelism, data flows through the entire graph, across GPUs, thus incurring communication overhead. There could also be data flow between operations colocated on the same GPU, but this communication latency is negligible.

TensorFlow DNN training framework: TensorFlow [9] is a widely employed software framework for training machine learning models, such as DNNs. In TensorFlow, the DNN models are stored and executed as directed acyclic graphs (DAGs), with the graph nodes representing mathematical operations (e.g., multiplication, gradient, etc.) and the edges representing data arrays (or tensors) that flow between operations. Different DNN models can have different DAGs. In TensorFlow, when adjacent operations are placed on different devices, a pair of send and receive operations is added to the

DAG for data transfer between devices; this data transfer incurs some overhead.

In terms of scheduling, for each device, TensorFlow randomly picks an operation from the ready queue (those whose predecessors have completed execution) for execution. Note that all partitioning and scheduling decisions in TensorFlow have to be made *before* runtime, and can thus not be altered during execution. For the rest of the paper, we consider TensorFlow as our DNN training framework.

2.2 Significance of partitioning and scheduling of DNN operations across GPUs

Depending on the number of model parameters and batch size employed, a DNN model may require significant memory capacity. For example, the Transformer model (see Figure 1) with 6 layers (and hidden size of 512 units, 8192 filters, and 16 attention heads with a batch size of 32 or higher) typically cannot be hosted on a single commodity GPU, thus necessitating the partitioning of the DNN model across GPUs. Given the complexity of the model, as illustrated in Figure 1, it is not obvious which operations, or even whole layers, should be placed on a given GPU to maximize GPU usage. Worse, when neighboring operations (that share an edge) are placed on different GPUs, a non-trivial communication overhead is incurred when data is transferred between the operations. For example, in our experiments, the communication overhead can account for 20%–50% of the total model training time. Consequently, the placement of the model across GPUs can significantly impact model training time [10, 44].

Assuming that a near-perfect partitioning of the model across GPUs can be obtained, it is still not obvious how the individual operations should be *scheduled* on their respective GPUs. Since the operations and layers in DNNs are closely dependent on each other (see Figure 1), a sub-optimal scheduling of operations on one GPU can stall the execution of dependent operations on other GPUs. This would result in a cascading effect, delaying the entire DNN training.

Illustrative example: To illustrate the challenges involved in DNN execution under frameworks like TensorFlow, consider the DAG shown in Figure 2(a) for a toy example. Assuming we have two homogeneous GPUs, Figure 2(b) shows the result of a naive scheduling which prioritizes the longest critical path, without knowing the compute requirements of operations. By ignoring the compute requirements, the naive scheduling results in a sub-optimal solution. Note that the operations are aware of all placement decisions, and can thus transfer data to the successor node GPU immediately after execution. If we additionally consider a naive placement of operations, as shown in Figure 2(c), then the communication overheads can further increase the execution time.

If the compute and communication times can be estimated beforehand, then an optimal solution can be obtained for operation placement and scheduling, as illustrated in Figure 2(d) where the compute intensive operations F and G are scheduled before the smaller operations, A–E. The subsequent improvement in execution time (and resource efficiency) can be significant, to the tune of 22–26% in the above example.

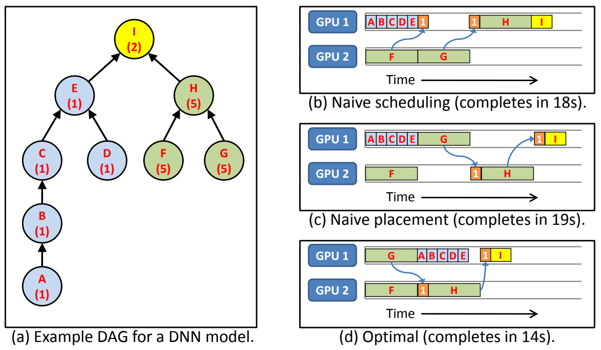

I (2)
E (1)
H (5)
C (1)
D (1)
F (5)
G (5)
B (1)
A (1)
GPU 1
GPU 2
Time
A B C D E 1 1 H I
F G
(b) Naive scheduling (completes in 18s).
A B C D E G 1 I
F 1 H
(c) Naive placement (completes in 19s).
G A B C D E 1 I
F 1 H
(a) Example DAG for a DNN model.
(d) Optimal (completes in 14s).

Figure 2: Illustration of the placement and scheduling challenges for DAGs in TensorFlow. In Figure (a), each circle (node) shows a compute operation. Directed arrows (edges) represent the direction of flow of data. Number in parentheses inside each node shows compute time. In Figures (b), (c), and (d), the orange boxes represent communication events. The white numbers inside each communication event denote the communication overhead.

3 PLACEMENT AND SCHEDULING VIA PESTO

This section presents our approach, Pesto, to intelligently place an arbitrary DNN DAG model across devices (GPUs, CPUs, etc.) with the goal of minimizing model training time without incurring significant placement determination time. Pesto jointly determines the placement *and* scheduling of DAG operations across devices. Pesto works by first estimating the compute time of operations and the communication time between adjacent operations in the DAG (Section 3.1) and then our algorithms (Section 3.2) make placement and scheduling decisions based on our integer programming formulation. Since integer programming can be time consuming, we introduce a graph coarsening technique (Section 3.3) that significantly speeds up our algorithm, thereby reducing the placement and scheduling overhead.

The core of Pesto is our DAG partitioning and scheduling. Scheduling DAGs with precedence (or dependency) constraints to minimize the makespan (time between the first task arrival and the last task completion) is an NP-hard problem [56]. As such, our ability to obtain near-optimal solutions remains limited. Algorithms based on List Scheduling (LS) are known to provide good approximation ratios for several DAG scheduling problems *if* the communication time between devices is negligible [21]. While the communication time for real systems is not negligible, LS-based solutions continue to be popularly employed [31, 51, 55]. For DAG scheduling with general communication times, there is no known polynomial-time approximation algorithm. In fact, even with constant communication times for all operations, the problem is known to be NP-hard [28].

The DNN DAG scheduling problem in Pesto differs from existing DAG scheduling literature in two aspects, as shown in Figure 3: communication time and constraints. First, most existing works assume unlimited bandwidth for data transfer [24, 39]. As a consequence, even if 100 operations on the same GPU send data to another GPU, it is assumed that they can proceed simultaneously without any

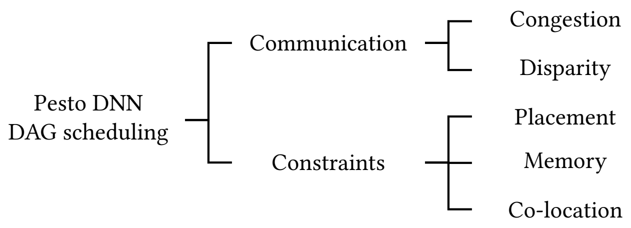
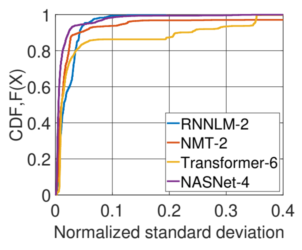
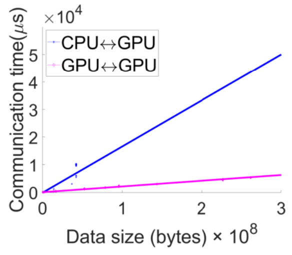


Figure 3: Algorithm challenges in Pesto scheduling.



(a) CDF of normalized standard deviation of compute times.



(b) Communication time as a function of the data transfer size.

Figure 4: Empirical results for the compute and communication time of operations for various TensorFlow models.

queueing delay and without slowing each other. However, this is not the case in real systems. In our Pesto scheduling, we perform data transfers sequentially since simultaneous transfers on the same link can result in additional delay. Further, existing models often assume that communication time is much faster than computation time [31], which is not necessarily true for DNN training. In fact, even when using the fast NVlink for inter-GPU data transfer, we find that communication time can be several orders of magnitude higher than the compute time of some operations. For constraints, our Pesto algorithm design considers several constraints to faithfully capture the system dynamics, including device affinity (e.g., CPU operation to be placed on CPU, GPU execution operations to be placed on GPU), GPU memory capacity constraints, and co-location constraints for specific set of DNN operations. DAG scheduling in the presence of such constraints has not been considered in prior work.

3.1 Estimating the compute and communication time of operations

For determining the placement and scheduling, Pesto requires accurate estimates of compute and communication times of DNN model operations.

Compute time: For estimating compute times, we use empirical data obtained by running 100 iterations of the DNN model on our experimental setup (Section 5). Using the average compute time for an operation across 100 runs as an estimate works well in practice because there is very little variability in the compute time of a *given* model-specific operation over different iterations, as has also been observed by prior works [22, 33, 44, 45]. Figure 4(a) shows our empirical results for the CDF of the normalized standard deviation (normalized by the mean) of the per-operation compute times for different DNN models. We see that, for all models, the normalized standard deviation is relatively small. For ease of illustration, we ignore very small operations in Figure 4(a) as they do not contribute significantly to model execution time. However, for our placement and scheduling algorithm, we do take *all* operations of the DNN graph into account.

Our estimation approach of employing a handful of iterations has very little overhead in practice, and similar approaches have been used in prior works (e.g., Baechi [31] and FlexFlow [33]). Since giant DNNs are typically trained for more than 100K training steps and multiple epochs [11, 18, 57, 61], the overhead of running 100 iterations for profiling translates to a less than 0.1% overhead. Further, the process of logging the compute times of a few iterations of the DNN model (before actually executing for the entire training data) can be automated and does not require manual effort. Finally, the profiling does not have to be repeated if some of the hyperparameters change, such as the learning rate. However, if the underlying DNN graph changes, for example, when the batch size is changed, then the compute times will have to be re-profiled.

Communication time: To estimate communication time between operations, we first classify communications into different types, i.e., between CPU and GPU (in both directions) and between two GPUs. Figure 4(b) shows our empirical results for the communication time across operations (of various models) as a function of the data transfer size (obtained from TensorFlow logs). We see that the communication time is almost linearly related to the data size. Based on the above analysis, we model the communication times for each communication type via a simple linear fit as $T_{comm} = \beta_0 + \beta_1 \times data$, where $data$ is the size of the data transfer (in bytes, in our case) and β_0, β_1 are constants that we determine via regression. We find that linear regression provides an accurate data fit with R^2 values of 0.92–0.99. Note that the linear communication model is independent of the DNN, and can thus be easily obtained via offline profiling of the communication operations of varying data sizes from *any* model (not necessarily the target DNN), as is the case for our evaluation results in Section 5.

3.2 Pesto algorithm design

3.2.1 Problem formulation. We represent the DNN model as a DAG, $G = (V, E)$, where V is the set of compute operations, and $E \subseteq V \times V$ is the set of precedence constraints among the operations. Each operation i has an execution time p_i (estimated as discussed in Section 3.1) and a designated device type O_i to be placed onto. Here we consider three types of devices, CPU, GPU, and Kernel, denoted by O_C, O_G, and O_K, respectively. Kernel operations are small pre-processing operations executed on the CPU before a GPU operation can be executed on the GPU. There is a communication link between each pair of devices. This could be PCIe for CPU-GPU communication, and PCIe or NVlink for GPU-GPU communication.

A constraint edge (i, j) requires that operation j can only start after operation i is completed and the data transfer process is completed on their corresponding communication link. The data transfer time between two operations $i, j \in V$ with $(i, j) \in E$ is estimated via the communication time model from Section 3.1, if the operations are placed on different devices; if the operations are placed on the same device, the transfer time is negligible and is ignored. To capture the congestion in the communication bus, we model inter-device communication links as a First-Come-First-Served queue.

For both devices and communication links, we do not allow preemptions, so there is only one operation that can be scheduled on each device or link at any given time. Our goal is to minimize the DAG completion time considering both compute time and communication congestion.

3.2.2 Pesto ILP. We employ an integer linear program (ILP) to find the DAG placement and scheduling. At a high level, the Pesto ILP algorithm consists of two steps. First, it augments the original DAG with additional edges to incorporate communication cost. It then formulates and solves an efficient 0-1 ILP to obtain the optimal placement and scheduling of operations across all devices.

Compared to existing DAG scheduling approaches, we explicitly model communication congestion and device and model constraints. For ease of presentation, we consider DAG scheduling with 2 identical GPUs. We discuss the extension to multiple GPUs at the end of this subsection.

DAG augmentation: In traditional DAGs, vertices represent compute time and edges represent communication time. However, such models often assume congestion-free communication, e.g., data transfer on edge 1 will not contend with data transfer on edge 2, even if the transfers are between the same pair of devices. This is not the case for DNN training in practice. We augment the DAG by converting edges into new nodes with specific labels and add constraints so that nodes with the same label, e.g., CPU-0→GPU-0, cannot be scheduled at the same time. This augmentation addresses the limiting assumption of unrestricted data transfer made in most existing works.

For a directed edge $e = (i, j) \in E$ where data is transferred from operation i to j before executing j, we augment G as follows. If i and j are both placed on GPUs, we add one vertex, k, and two edges $(i, k), (k, j)$ for the potential communication overhead between GPUs. Denote this set of added vertices by O_{GG}, representing new GPU-GPU communication vertices. For CPU-GPU (or GPU-CPU, respectively) precedence constraints, we augment the DAG in the same way and denote the new set of nodes by O_{CG} (or O_{GC}, respectively). Denote the augmented graph by $\bar{G} = (\bar{V}, \bar{E})$.

Placement and scheduling: We identify the optimal placement and scheduling for each operation in the DAG by solving the following ILP, referred to as Pesto ILP.

$$
\begin{array}{lll}
\min & C_{\max} & \text{(Pesto ILP)} \\
\text{s.t.} & C_i \le S_j, \quad (i,j) \in \bar{E} & (1) \\
& S_i + p_i = C_i, \quad \forall i \in \{\bar{V}/O_{GG}\} & (2) \\
& 0 \le C_i \le C_{\max}, \quad \forall i \in \bar{V} & (3) \\
& \text{Non-overlapping constraints} & (4) \\
& z_k = x_i \textbf{ XOR } x_j, \forall k \in O_{GG}, (i,k), (k,j) \in \bar{E} & (5) \\
& S_i + z_i p_i = C_i, \quad \forall i \in O_{GG} & (6) \\
& \text{Congestion constraints} & (7) \\
& \text{Memory constraints} & (8) \\
& x_i \in \{0,1\}, \forall i \in O_G, \quad z_j \in \{0,1\}, \forall j \in O_{GG} & (9)
\end{array}
$$

Here, $C_{\max}$ represents the execution time of the entire DAG, which is the per-iteration DNN model training time. C_i, S_i, and p_i are the completion time, starting time, and processing time of operation i, respectively. Note that p_i for $i \in O_{GG} \cup O_{CG} \cup O_{GC}$ is simply the communication time estimate, obtained as discussed in Section 3.1. Constraint (1) enforces the precedence constraint for all operations. Constraints (2) ensures that the finishing time of any operation equals the sum of its starting time and processing time.

The key decision variable is x_i for the placement of GPU operation $i \in O_G$. $x_i = 0$ (or $x_i = 1$) denotes the placement of i on GPU-0 (or GPU-1, respectively). Colocation constraints can be easily handled in our ILP formulation. To colocate operations $\{i_1, i_2, \ldots, i_k\}$, we set $x_{i_1} = x_{i_2} = \ldots = x_{i_k}$.

Non-overlapping constraints: ensure that operations placed on the same device do not have overlapping execution periods. For example, two operations i and j placed on CPU-0 core must have non-overlapping execution intervals, $[S_i, C_i)$ and $[S_j, C_j)$. Thus, $S_i \ge C_j$ **XOR** $S_j \ge C_i$. The conditions, for each CPU core, can be rewritten using indicator variables (to maintain the 0-1 ILP formulation) as $S_i \ge C_j - M\delta_{ij}$ and $S_j \ge C_i - M(1 - \delta_{ij})$, where M is a large number and δ_{ij} is a 0-1 indicator variable. To see the equivalence, if $\delta_{ij} = 1$, then the first inequality always holds, so we have $S_j \ge C_i$. Otherwise $\delta_{ij} = 0$, which means the second inequality always holds, so $S_i \ge C_j$. Note that, by Constraints (2), $C_i > S_i$ and $C_j > S_j$, so $S_i \ge C_j$ implies $C_i > S_i \ge C_j > S_j$, and so $S_j \ngeq C_i$, ensuring the **XOR** condition. Similarly, $S_j \ge C_i$ implies $S_i \ngeq C_j$.

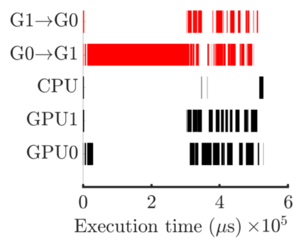
For GPUs, the non-overlapping condition is more complicated as non-overlapping constraints are needed when i and j are on the same GPU. Nevertheless, we can employ 0-1 indicator variables to express these constraints as:

$$
\begin{cases}
S_i \ge C_j + M_s\delta_{ij} - M_l(2 - x_i - x_j) \\
S_j \ge C_i + M_s(1 - \delta_{ij}) - M_l(2 - x_i - x_j) \\
S_i \ge C_j + M_s\delta_{ij} - M_l(x_i + x_j) \\
S_j \ge C_i + M_s(1 - \delta_{ij}) - M_l(x_i + x_j),
\end{cases}
\tag{10}
$$

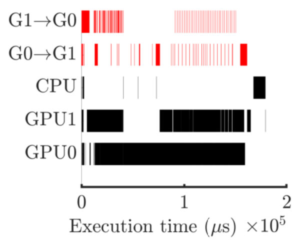
where $M_l \gg M_s \gg 0$. The placement constraints for kernel operations are similar to those of GPU operations.

Congestion constraints: handle communication congestion, which can critically impact DNN training times under model parallelism. For GPU-GPU communication, the existence of processing time for the augmented vertex, say k, depends on whether the two communicating GPU operations are placed on the same GPU. We thus add a 0-1 integer z_k for $k \in O_{GG}$. Constraint (5) sets $z_k = 1$ if the two corresponding GPU operations are placed on different GPUs. The **XOR** condition in Constraint (5) can be reformulated (since the logical **XOR** condition is not supported directly by solvers) by four linear constraints: $z_k \le x_i + x_j$, $z_k \ge x_i - x_j$, $z_k \ge x_j - x_i$, $z_k \le 2 - x_i - x_j$. From Equation (6), if $z_k = 1$, then $z_k p_k = p_k$ amount of communication time is required to process the transfer; if $z_k = 0$, then $S_k = C_k$, meaning negligible transfer.

For two GPU-GPU communication operations $i, j \in O_{GG}$, assume i connects $a, b \in O_G$ and j connects $c, d \in O_G$. For the GPU-1 $\longrightarrow$ GPU-0 one-way traffic, congestion occurs *only* when $x_a = 1, x_b = 0$ and $x_c = 1, x_d = 0$, meaning that both data transfers exist and are from GPU-1 to GPU-0. Then we have $x_a + x_c - x_b - x_d = 2$. The non-overlapping constraints can thus be written, similar to



(a) Congestion-oblivious ILP solution.



(b) Congestion-aware ILP solution.

Figure 5: Illustrating the importance of congestion constraints in our Pesto ILP for the RNNLM-2-2048 DNN model.

Equation (10), as:

$$\begin{cases} S_i \geq C_j - M\delta_{ij} + M_s(x_a + x_c - x_b - x_d - 2) \\ S_j \geq C_i - M(1-\delta_{ij}) + M_s(x_a + x_c - x_b - x_d - 2) \end{cases}$$

where $M_s \gg M \gg 0$. Similar constraints exist for GPU$-0 \longrightarrow$ GPU-1 traffic or GPU-CPU congestion.

We emphasize that communication congestion constraints are crucial to our DNN DAG scheduling. Figure 5 compares the DNN execution for an RNNLM model (via our simulator, see Section 5.4) under the Pesto ILP without and with congestion constraints; the x-axis denotes the execution timeline and the y-axis denotes the significant execution components. Without congestion constraints, multiple data transfers proceed in parallel on the GPU-0 ⟶ GPU-1 link (Figure 5(a)), causing a significant communication delay. With the constraints in place, the ILP finds a communication-aware placement that results in fewer and staggered inter-GPU communication events, resulting in a nearly 3× reduction in execution time (see x-axis in Figure 5(b)).

Memory constraints: are approximated in our ILP formulation by ensuring that the cumulative memory footprint of operations on each GPU is balanced. For each GPU device, we compute the memory footprint of the resident operations by summing up the input and output tensor memory sizes; these sizes can be obtained from TensorFlow via the `tf.profiler` API. We find, in our experiments (see Section 5), that this simple memory constraint suffices to avoid out of memory (OOM) errors. However, we note that the constraint can be strengthened, if needed, to also account for precise allocation and deallocation of temporary memory [31].

ILP optimality, extensions, and solution: Any valid placement and scheduling is a feasible solution to our ILP, and any feasible solution obtained from the ILP is also a valid placement and scheduling, by construction of our ILP. Given the 1-1 correspondence, the solution found by solving the ILP is the optimal placement and schedule. This is because the solution to the ILP is by definition, optimal.

Theorem 3.1 (Optimality). *For DNN DAG scheduling problem with 2 GPUs, the solution generated by Pesto ILP formulation is optimal.*

Our Pesto ILP can be extended to the case of arbitrary number of devices. The first step in this extension is to add additional indicator variables for each new device. For example, for 4 GPUs, the placement of operation i can be indicated by the pair $\{x_i, y_i\}$ of 0-1 variables, which together encode all four GPU placement possibilities. The non-overlapping congestion constraints can then be adapted accordingly. Our ILP framework also supports hierarchical and heterogeneous communication models, which are useful for modeling communication between hosts or over different communication fabrics (e.g., PCIe and NVlink).

By solving this 0-1 integer programming using standard optimization software like CPLEX [30], we obtain both the starting time and the placement for each operation. This scheduling and placement is then implemented in TensorFlow, see Section 4. However, solving the 0-1 ILP is a time-consuming step. We next discuss our approach to substantially speed up (by as much as 10,000×) the ILP solving time.

3.3 Graph Coarsening

Modern DNNs often consist of tens of thousands of operations, most of which are very small in terms of compute time, as shown in Table 1. It is challenging to optimize the DAG scheduling problem at this large scale, especially since simplified versions of our DAG scheduling problem have already been shown to be NP-hard [21, 56]. We propose a novel approach for graph coarsening to speed up Pesto significantly by efficiently shrinking the graph size while avoiding generating cycles and maintaining the parallelizability.

Model	Execution Time		
	$< 10\mu s$	$10-100\ \mu s$	$> 100\mu s$
Transformer-6	14511	4119	763
RNNLM-2	3401	973	240
NASNet-4	14191	1677	1033
NMT-2	21243	8114	703

Table 1: Most DNN compute operations are small.

Cycle prevention: The key challenge when shrinking a graph by merging vertices is to avoid creating cycles since otherwise the resulting graph is no longer a DAG (and so no valid schedule can be found). We start with a theorem stating necessary and sufficient conditions for merging two adjacent vertices. The theorem can be easily proved via contradiction, and is omitted due to lack of space.

Theorem 3.2 (Necessary and sufficient conditions for merging two paired operations without generating a cycle). *Given a DAG $G = (V, E)$ with $u, v \in V$ and $(u, v) \in E$, the new graph G' obtained by removing edge (u, v) and merging u, v is acyclic if and only if (u, v) is the only path from u to v on G.*

Corollary 3.3. *Given a DAG $G = (V, E)$ with $u, v \in V$ and $(u, v) \in E$. Let $prec(v)$ and $succ(v)$ denote the set of all predecessors and successors, respectively, of v. If $|prec(v)| = 1$ or $|succ(u)| = 1$, (u, v) is the unique path from u to v.*

Note that Theorem 3.2 cannot be used to merge multiple pairs of vertices simultaneously. Consider the simple example in Figure 6 where both (A, C) and (B, D) satisfy the condition in Theorem 3.2. However, when merging them simultaneously, a loop is generated between the two new merged-vertices, as illustrated in Figure 6. Thus, if merge pairs of vertices sequentially, we need to test the condition in Theorem 3.2 each time we merge a pair. This would take at least $O(|V| + |E|)$ time for each merge, resulting in a prohibitively

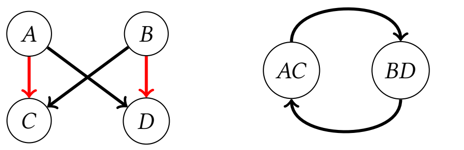


Figure 6: Merging (A, C) and (B, D) simultaneously creates a cycle, violating the DAG requirement.

high running time given that we have several tens of thousands of vertices in our giant DNN DAGs.

Batch merging: To process the graph faster, we build on recent graph partitioning work [25] to merge thousands of vertices in a single batch. Compared to recent work [25], our batch merging allows for additional edges to be take into consideration, resulting in faster coarsening. We now present an efficient technique for batch merging while avoiding cycles.

Definition 3.4 (Height). The height of a vertex v, $H(v)$, is the longest distance in terms of number of vertices it takes to go from a root node to v. The height of root nodes, i.e., the nodes without any precedence constraints, are defined as 1.

The height of all vertices can be calculated using a modified version of topological sorting[1] in $O(|V| + |E|)$. Using the concept of height, we present a theorem stating the sufficient conditions for merging pairs of vertices without introducing a cycle. Due to lack of space, we only provide a proof sketch.

Theorem 3.5 (Sufficient conditions for merging multiple pairs in a batch). *Given a DAG $G = (V, E)$ with $u, v \in V$ and $(u, v) \in E$. Define set $M = \{(u_1, v_1), \ldots, (u_k, v_k)\}$ where each $(u_i, v_i) \in E$ for $i \in [k]$. Let $d_i = H(v_i) - H(u_i)$. Then the new graph G' obtained by merging every edge in M is acyclic if*

(i) There are no repeated vertices in M, i.e., M is a match;
(ii) $|succ(u_i)| = 1$ or $|prec(v_i)| = 1$ or $H(v_i) = H(u_i) + 1$ for all $i \in [k]$ or $H(w_i) > H(u_i) + d_i$ for all $w_i \in succ(u_i)$ and $w_i \neq v_i$; and
(iii) $\forall i, j$ and $i \neq j$, $h(u_i) \neq h(v_j) + d_j$ or $(u_i, v_j) \notin E$.

Proof Sketch. The proof is by contradiction. Assume a cycle is generated after merging set M. We first find a minimal cardinality cycle, thus eliminating conditions such as $|succ(u_i)| = 1$ or $|prec(v_i)| = 1$ where a smaller cycle can be found. We can also eliminate cases where either $u_i \rightsquigarrow u_{i+1}$, $v_i \rightsquigarrow u_{i+1}$ or $v_i \rightsquigarrow v_{i+1}$, where $\rightsquigarrow$ represents a directed path. We then consider branches where $u_i \rightsquigarrow v_{i+1}$. For both of the following cases: $H(v_i) = H(u_i) + 1$ or $H(w_i) > H(u_i) + d_i$ for all $w_i \in succ(u_i)$ and $w_i \neq v_i$, we show that $H(v_1) < H(v_2) < \ldots < H(v_1)$, implying the existence of a cycle in the unmerged graph, creating a contradiction. □

The conditions outlined above can be verified easily within constant time as for each vertex we only need to check or compare its height, outdegree, and indegree. We can thus efficiently coarsen the DAG without introducing a cycle. For more complex environments (e.g., more GPUs or communication links) where the number of variables and/or constraints is large, the DAG may require much coarsening to obtain the ILP solution in a reasonable amount of time. Corollary 3.6 below shows that we can always reduce the graph size as desired via our coarsening algorithm.

Corollary 3.6. *Given a DAG $G = (V, E)$, for any $T \geq 1$, the acyclic merging in Theorem 3.5 can obtain a coarsened graph $G' = (V', E')$ with $|V'| \leq T$ in finite steps.*

Maintaining parallelizability: In theory, any DAG can be coarsened into a single merged-vertex by carefully merging vertices; however, this would result in the complete loss of parallelizability of the DAG. To reduce the graph size while facilitating parallelizability across GPUs, we must carefully manage the graph coarsening. There are several parameters to consider for coarsening that can impact the parallelizability, such as which pairs of vertices should be merged. In practice, we prioritize the merging of pairs of vertices based on the size of their connecting edge, i.e., the estimated communication time. Intuitively, if the data transfer between two operations is large, it may be best to place both operations on the same GPU to avoid the high communication time.

Our coarsening algorithm: In each iteration, our coarsening algorithm merges all feasible edges (based on the conditions in Theorem 3.5). Each iteration takes $O(|E| \log |E|)$ time. We continue iterating until the DAG reaches the desirable size or until no feasible edges can be found. In practice, each iteration removes 30–70% of the existing edges, depending on the sparsity of the DAG. Thus, a few iterations are sufficient to adequately reduce the graph size. For the giant DNN models we consider in Section 5, we find that coarsening the graph to ~200 vertices for our specific experimental setup provides a good tradeoff between the time required to solve the ILP and the reduction in DNN training time. When we coarsen beyond 200 vertices, we find that the graph is too simplified for the models we consider, resulting in minimal reduction in training time. If we do not coarsen till about 200 vertices, the ILP solution time is substantially high and the resulting placement and scheduling does not reduce the training time by much; see Section 5.3 for a specific example.

The coarsened graph is used as the input for our ILP algorithm in Section 3.2.2. If the ILP suggests placing merged-vertex v on, say, GPU-0, then all vertices that were merged during coarsening to form v will be placed on GPU-0. In terms of scheduling, individual vertices of a merged-vertex are scheduled sequentially on the same device, following the precedence constraints of the DAG. For some of the models we evaluate in Section 5, when the DAG is very large, each vertex in the final coarsened graph may contain hundreds of operations. In such cases, we lose out on scheduling opportunities due to coarsening, and thus instead employ the default TensorFlow scheduling. The Pesto placement of the coarsened DAG operations still provides substantial benefits, so we employ that placement.

4 IMPLEMENTING PESTO ON TENSORFLOW

We implement Pesto by integrating directly with TensorFlow. The source code for Pesto is publicly available for reference [5]. There are three challenges that need to be addressed to realize the implementation of Pesto. For operation placement and scheduling, Pesto requires the compute time estimates of operations. Further, for scheduling, Pesto requires the structure of the DAG. Finally, the placement results from Section 3.3 must be enforced when running the DAG.

[1]At each step, we remove a set of vertices without predecessors instead of removing just one in Kahn's topological sorting algorithm [35].

To obtain estimates of compute time, we exploit the observation (see Figure 4(a)) that there is little variability in the per-operation compute time of DNN models. As such, compute times can be estimated using a handful of iterations of the DNN model training or via offline runs, as discussed in Section 3.1. These estimates are then used as input for our Pesto ILP to obtain the optimal placement and scheduling.

The key challenge in implementing Pesto is to enforce the placement and scheduling in TensorFlow. In TensorFlow, a graph API, `tf.Graph`, holds information about the structure of the DAG of the DNN model being trained. A session API, `tf.Session`, takes as input this DNN graph and identifies a sub-graph which needs to be executed next. TensorFlow then executes all the operations in the sub-graph based on available resources (CPUs, GPUs, etc.) according to some internal scheduling and placement policies. The `tf.Graph` API only allows user-specified placement and scheduling constraints (typically specified as control-flow-dependencies) at the time when an operation is being added to the graph [8]. Because of the static nature of the constraints, the placement and scheduling suggested by our algorithm cannot be realised without modification of the user code of the model.

To enable TensorFlow to support placement and control-flow-dependencies *after* the graph is created and *without* requiring the user to modify their code, we modify the `tf.Session` API to place low-level graph operations using the `tf.Node.set_assigned_device` function for each operation node in the graph. We also modify the scheduler in `tf.Session` to take into account the post-graph-creation control-dependencies while scheduling graph operations. Specifically, we modify the TensorFlow runtime to add scheduling constraints for each node in the graph using the `tf.Node.add_control_dependency` function to enforce Pesto-suggested scheduling.

5 EVALUATION RESULTS

We now discuss our evaluation results. We first describe our experimental setup and methodology, including the models we experiment with and the baselines we compare against. We then present our implementation results. Finally, we present a simulator-based analysis to evaluate Pesto under different GPU and communication link settings.

5.1 Experimental Setup

We run our experiments on a server with an Intel Xeon Silver 4116 processor and two NVIDIA Tesla V100 SXM2 16GB GPUs. Each GPU is connected to the CPU with a dedicated PCIe [53] connection. GPUs communicate with each other directly via NVlink [4]. We use TensorFlow r1.15 as our DNN model training framework.

5.2 Evaluation Methodology

Given the increasing popularity of giant models, unlike prior works, we evaluate Pesto across multiple giant models.

RNNLM [34, 60]: Recurrent Neural Network Language Model with multiple LSTM [27] layers is used for language modeling. We fix the batch size to 128 and experiment with three variants: 2-layered with 2048 hidden units, 4-layered with 2048 hidden units, and 16-layered with 1024 hidden units. We use the Penn Treebank [43] dataset for training.

NMT [11, 58]: Neural Machine Translation model with attention is used for automatic language translation. The high level structure of NMT is similar to that of RNNLM, but because of the large number of hidden states and attention mechanism, NMT is far more complex. We experiment with 2- and 4-layered model variants and fix the number of hidden units to 1024 for each LSTM layer and use a batch size of 128 for training. We train NMT on the WMT16 dataset [7].

Transformer [57] model is another sequence-to-sequence model with multi-head-attention mechanism for language translation. In contrast to NMT, Transformer employs feed-forward networks instead of LSTM layers. We consider 3 variants: 10 layers with 8 heads and 1024 hidden units, 12 layers with 8 heads and 1024 hidden units, and 6 layers with 16 heads and 2048 hidden units. We set the batch size to 32 sentences and train the model on the WMT14 [12] dataset.

NASNet [61] is a convolutional neural net (CNN) with multiple cells. Each cell in NASNet is composed of multiple branches for convolution, addition, and other compute operations, providing an opportunity for parallel execution. We train NASNet using ImageNet [18] data with a batch size of 32 under 3 settings: 4 cells with 212 filters each, 6 cells with 148 filters each, and 6 cells with 168 filters each.

Except for RNNLM 2-layered and NMT 2-layered (which we include for comparison with other approaches), none of the model variants fit on one GPU in our setup.

Baselines for comparison. To fully evaluate Pesto, we compare our results with those obtained by manual placement via domain experts as well as state-of-the-art automated placement approaches from prior works [10, 31, 44, 45]. We discuss prior approaches in detail in Section 6.

Expert: The widely used comparison baseline is the manually obtained placement found by domain experts, referred to as *"Expert"* strategy. For NMT, Expert places different layers of the sequentially stacked LSTM layers on multiple GPUs [58]. In addition, attention and softmax layers are placed on the same GPU as the last LSTM layer while embedding layer is colocated with the first LSTM layer. Given the similarity in DNN graph structures, the Expert placement for RNNLM and Transformer is similar to that of NMT. For NASNet, Expert places parallel branches within each cell across different GPUs [10]. Each cell in NASNet has multiple convolution and addition operations which are divided evenly among available GPUs.

RNN-based: Mirhoseini et al. [45] use a Recurrent Neural Net (RNN) to optimize device placement. This approach was further improved by using hierarchical models for better grouping and placement of compute operations [44].

Placeto [10] is also a learning-based algorithm that uses graph embeddings and reinforcement learning to iteratively improve the learned placement policies.

Baechi [31] employs traditional job scheduling algorithms to find memory-aware feasible placements for DNN model graphs across

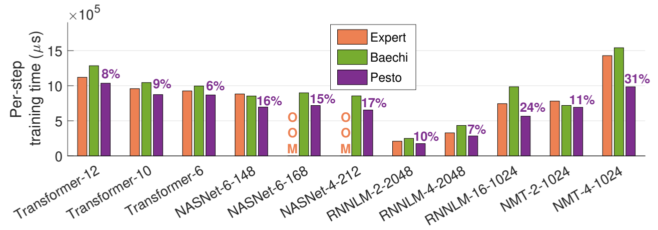


Figure 7: Experimental per-step model training time under different strategies. NASNet-6-168 and NASNet-4-212 run out of memory (OOM) when using Expert strategy.

multiple GPUs. We evaluate Baechi using its open-sourced version [2]. While Baechi employs three heuristics, we primarily compare with mSCT, that reportedly outperform the other two heuristics, mETF and mTOPO [31]. mSCT modifies the classical SCT [23] job scheduling algorithm to take into account large communication times, finite number of GPUs, as well as limited memory.

5.3 Implementation Results

For all the models and variants discussed in Section 5.2, Figure 7 compares the (per-step or per-iteration) DNN training time using the placement and scheduling suggested by Pesto with that under the Expert placement strategy and Baechi. When referring to Baechi in our evaluation, we use results for the best Baechi heuristic; in our experiments, for all models, mSCT always outperforms mETF and mTOPO under Baechi with respect to DNN training time. In Figure 7, the numbers above the bars denote the percentage reduction in training time achieved by Pesto as compared to the best alternative strategy for each model variant. Across all variants, Pesto reduces DNN training time by 14%, on average, compared to the best alternative approach.

Owing to the grid like structure of LSTM cells in NMT and RNNLM, Pesto finds placements with high GPU utilization, resulting in a significant reduction in training time by about 21% and 18%, on average, compared to Expert, and by about 20% and 35%, compared to Baechi. Importantly, *state-of-the-art approaches (listed in Section 5.2) are unable to find placements for RNNLM that are superior to Expert.*

For the Transformer models, Baechi is unable to find placements which perform better than the Expert strategy. However, Pesto achieves moderate but still non-trivial reductions of about 8% (on average, across all variants) in training time over Expert. While the 8% reduction does not appear significant, it can translate to substantial savings in the training effort. For example, to achieve high accuracy for German-to-English translation on the WMT14 [12] data set, Google [57] suggests training the giant Transformer models for 300K training steps. This corresponds to ∼93 hours of training time (for 12-layered Transformer model) when using the Expert strategy. Savings of 8% afforded by Pesto translates to a reduction of ∼8 hours in total training time, reducing total compute cost of training by ∼16 (8×2) GPU-hours.

For NASNet, across 3 different variants, Pesto enables 16% faster training than Baechi, which is the best among the alternative placement strategies. For two of the configurations (6 cells with 168 filters and 4 cells with 212 filters), the *Expert strategy encounters the out of memory (OOM) error.* By contrast, since Pesto aims to balance

Models	Spec.	Placement time (minutes)			
		Baechi	RNN-Based	Placeto	Pesto
NMT	2-layer	1	2859	788	30
	4-layer	3	2714	4120	51
NASNet	6-cells	3	241	50	24

Table 2: Comparison of placement time across approaches.

the memory footprint across the GPUs, it successfully partitions the larger variants of NASNet without encountering the OOM error.

Comparison of DNN training time improvement with learning-based approaches: Open-source implementations of learning-based approaches (RNN-based [44, 45], Placeto [10]) are not available for comparison. To conduct a fair comparison (despite the differences in experimental setup), we compare Pesto's improvement over Expert with the reported improvement, from Addanki et al. [10], of existing learning-based approaches over Expert. This is feasible as the Expert strategy employed in our work and that employed by existing approaches is consistent. However, most existing approaches only consider a subset of the giant models we consider in our evaluation, so we only report those results.

For the NASNet-6-148 model, Placeto provides no improvement over Expert, whereas the RNN-based learning approaches [44, 45] result in ∼3.5% higher training times than Expert. By contrast, Pesto reduces training time for NASNet-6-148 by 21% compared to the Expert placement. Placeto [10] also evaluates placements for NMT (but not RNNLM or Transformer); however, in Placeto's experimental setup, the Expert strategy throws an OOM error for NMT so we are unable to compare the relative improvement over Expert.

Comparison of placement time improvement with existing approaches: Table 2 shows the average time reported by existing approaches [10, 44, 45] to find the placement (referred to as placement time) for NMT and NASNet models. For Baechi, we obtain placement times based on our experiments with the open-source Baechi implementation using the NMT and NASNet model variants listed in Section 5.2.

Pesto finds the placement (and schedule) for 4-layered NMT in about 51 minutes. By contrast, Placeto and RNN-based approaches take more than 45 hours to find placements, with Placeto requiring almost 3 days. We see that Baechi is able to find placements quickly for NMT models. However, as noted above, Baechi results in higher training times for NMT compared to Expert. Similarly, for NASNet, Pesto finds the placement with reduced training time in about 24 minutes as compared to learning-based approaches (Placeto and RNN-based) that require much more time (up to 4 hours) to find the placement. In contrast to Pesto, Baechi finds placements for NASNet in about 3 minutes but only achieves 3% improvement in training time as compared to Expert.

Comparison of total training effort with existing approaches: In actual deployments, the DNN *training effort* includes both the overhead of finding a placement (or placement time) and the actual training process which is repeated several times until a desired accuracy is reached. We now compare the total training effort by leveraging the reported results relative to Expert for learning-based approaches from Addanki et al. [10]; we assume that the Expert placement strategy is known a priori and so has zero placement time. For NMT models, as suggested by prior work [6], we consider 350K training steps. Table 3 shows that the end-to-end training effort for Baechi is 0.94× and 1.08× that of Expert, respectively, for the 2-layer and 4-layer NMT models. The corresponding numbers for Pesto are 0.89× and 0.7× that of Expert, representing a significantly reduced effort. For NASNet, prior work [54, 61] reports that training NASNet on ImageNet data requires 375K training steps per epoch. The training effort for one epoch of ImageNet data for NASNet is roughly the same for Expert and Placeto. For RNN-based and Baechi, the training effort relative to Expert is 1.03× and 0.97×, respectively. By contrast, Pesto only requires 0.81× times the effort of Expert. These results show that Pesto significantly (20−30%) improves the end-to-end training effort compared to Expert and existing approaches.

The lower training effort under Pesto is partly due to our coarsening algorithm from Section 3.3. For example, without coarsening, the ILP solver that Pesto employs fails to complete even after one week of runtime on a commodity server for the smallest model we consider, RNNLM-2-2048. Thus, without coarsening, the placement time can be prohibitively high. With coarsening, we solve the ILP for all models we consider in less than one hour. For the RNNLM-2-2048 model, we coarsen the graph to ∼200 vertices and solve the ILP for the coarsened graph in 10 minutes; the resulting per-step training time under Pesto is 171ms. If we instead consider a coarser graph with ∼240 or ∼280 vertices, the ILP solution time is about 2 hours and 24 hours, respectively, and does not result in any noticeable decrease in training time.

Analysis of results: The training time results under Pesto can be analyzed from various perspectives. First, the structure of the DAG dictates the parallelization opportunity. Consequently, since Transformer models have significant communication overheads, they do not provide much opportunity for parallelization, leading to only moderate benefits. Second, by carefully staggering the communication events (via our ILP scheduling and congestion constraints), Pesto avoids communication overhead and congestion, which are commonly encountered by alternative approaches for NMT models. Finally, we find that the alternative approaches perform poorly for NASNet and RNNLM models because of unbalanced compute load across GPUs. By contrast, Pesto provides a much more balanced placement for these models, resulting in significant improvements.

While all approaches employ some form of graph coarsening, our rigorous coarsening approach (Section 3.3) has greater flexibility, resulting in a coarsened graph that can be better partitioned across GPUs. As a specific example, alternative approaches merge pairs based only on the outdegree information, whereas Pesto also leverages the indegree and height information (see Theorem 3.5), providing greater flexibility during coarsening.

Note that Pesto does not alter the compute graph for a given DNN model. Thus, the achieved model accuracy and convergence is not affected. For example, the top-5 accuracy for a 10-layered Transformer model after training for 100K steps on WMT14 [12] (English - German) dataset is similar (∼0.84) for both Expert and Pesto placement, as expected.

Models	Spec.	Training effort, relative to Expert			
		Baechi	RNN-Based	Placeto	Pesto
NMT	2-layer	0.94×	-	-	0.89×
	4-layer	1.08×	-	-	0.7×
NASNet	6-cells	0.97×	1.03×	1.0008×	0.81×

Table 3: Comparison of training effort of various approaches with Expert. The indirect comparison with RNN-Based and Placeto for NMT is omitted because Expert results in OOM error as reported by Addanki et al. [10].

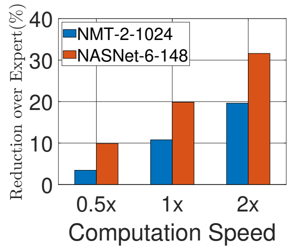
5.4 Exploratory Results via Simulation

Thus far our Pesto experimental results were obtained via implementation. To explore the benefits of Pesto under different hardware settings, we resort to simulation results based on an accurate DNN training simulator that we developed. Our simulator consists of three components: (i) it takes the placement and scheduling as input to our DAG and adds precedence constraints to the existing DAG to achieve the correct ordering and placement; (ii) it then employs a communication prediction model (such as the linear model from Section 3.1) to estimate the communication cost for each data transfer; and (iii) it employs the specified scheduling to place operations on simulated compute devices and communication links, using empirical estimates (from Section 3.1) as compute times. Our simulator takes only a few seconds to estimate the training time of a DNN model for a given placement and scheduling strategy.

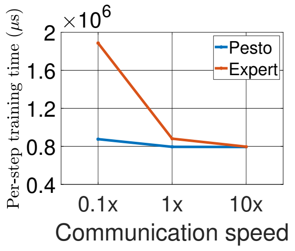
We validate our simulator by comparing the simulation-reported training times with the implementation-reported training times under Pesto for all DNN models from Section 5.2. The difference in reported training times ranges from 0.1% to 11.3%, with an average error of about 5%.

Using our simulator, we consider GPUs and CPUs with different compute power/speed (as a proxy for how fast the computation is) and interconnect links with different communication latencies by scaling our compute and communication time estimates accordingly. We simulate the per-iteration training time under Pesto and Expert, and report the training time reduction over Expert afforded by Pesto. Figure 8(a) shows that the improvement afforded by Pesto scales with the compute speed, suggesting improved benefits for future GPU models. As the compute speed increases, communication becomes a larger bottleneck, and so Pesto adapts by aggressively placing operations to prevent inter-GPU communication. The compute speed on the x-axis is relative to the GPU (NVIDIA Tesla V100) and CPU used in our implementation results.

Figure 8(b) compares the training time for Pesto and Expert under different interconnect speeds (for 1× compute speed) for the NMT-2-1024 model. Note that the 1× communication speed refers to the NVlink setting used in our implementation-based experiments; the 0.1× is on the order of PCIe [42] We see that Pesto successfully adapts its placement based on the communication speed, highlighting its robustness to the interconnect; this is because of the congestion constraints we design for our ILP formulation (Section 3.2.2). By contrast, the Expert strategy is oblivious to the communication speed, thus negatively impacting its performance for slower interconnect.



(a) Reduction in training time afforded by Pesto over Expert under different compute speeds.



(b) Training time for Pesto and Expert under different interconnect speeds for the NMT-2-1024 model.

Figure 8: Simulation results for various hardware settings.

6 RELATED WORK

Improving training time under model parallelism. ColocRL [45] employs a Recurrent Neural Network (RNN) based sequence-to-sequence model to find the optimal partitioning strategy through repeated trials. The authors followed up on this work and proposed HierarchicalRL [44] to further improve the effectiveness of the RNN-based learning technique. Placeto [10] is also a learning-based technique to find the optimal DNN placement but it strives to make the approach generalizable for a given model. However, the above learning-based approaches can take a long time (few hours to a couple of days) to find the DNN model placement, as shown in Table 2. Further, as discussed in Section 5.3, Pesto can outperform these techniques.

FlexFlow [33] employs Monte Carlo simulations to find operation placements for a given DNN model. However, FlexFlow has limited support for mainstream DNN training frameworks [9, 13, 47]; as such, using FlexFlow requires rewriting the model code. Further, FlexFlow ignores memory requirements of different operations and can result in out of memory (OOM) errors for giant DNNs.

Popular distributed DNN training frameworks provide only limited support for finding effective placement strategies under model parallelism. By default, TensorFlow [9] tries to fit the entire DNN on a single GPU and throws OOM error even for hosts equipped with multiple GPU devices. As such, users have to manually place different subsets of the DNN model on different devices (GPU, CPU, etc.). While frameworks such as DistBelief [16] and STRADS [37] support model parallelism, they still require the user to provide the partitioning manually. There are also recent works [38, 40] which support model parallelism for specific DNNs; however, these cannot be used for training arbitrary DNNs.

Other giant DNN training frameworks. DeepSpeed [48, 49] employs optimizations to reduce the memory footprint and memory redundancy of training giant DNNs. However, the consequent partitioning of memory (across GPUs or between GPU and host DRAM) introduces additional communication overheads that can be as high as 1.5× the overhead of data parallelism [48]. Prior works have shown that the communication overhead of data parallelism for giant models is already prohibitively high [29, 46]. Mesh-TensorFlow [52] enables model parallelism by executing each graph operation partially on a different GPU with the goal of reducing the GPU memory footprint. However, in contrast to Pesto, employing Mesh-Tensorflow requires rewriting the DNN model code. Further, Mesh-Tensorflow requires a domain expert to decide on the dimensions for splitting the graph operations.

Graph partitioning and scheduling algorithms. With the growth in the size of DNN models, graph partitioning is a necessary tool for speeding up DNN DAG scheduling algorithms. Purely graph-theoretic approaches, such as Scotch [14], exploit the DAG structure and balance the computational load while reducing the cost of communication. However, such approaches often result in local optimality, and perform poorly for large DNN models. Domain knowledge can be used to improve the graph partitioning, for example, by treating specific layers of the DNN as disjoint subsets [58], but this approach is not generalizable to arbitrary DNNs. Baechi [32] leverages label combinations provided by TensorFlow to merge operations. As discussed in Section 5.3, Baechi can find feasible placements faster than Pesto. However, the resulting placements are suboptimal with respect to training time, resulting in higher total training effort as compared to Pesto, as shown in Table 3.

In the algorithm design community, DAG scheduling problems are known to be NP-hard. While recent breakthroughs have been made in several simplified settings of the DAG scheduling problem [20, 39, 41], the resulting approximation algorithms are either too expensive to implement for giant DNNs or make limiting assumptions (such as congestion-free communication, see Section 3.2.2) that do not hold in practice. As a consequence, ad-hoc heuristics, such as dominant sequence clustering [59], Heterogeneous Earliest-Finish-Time, and Critical-Path-on-a-Processor [55], are commonly employed in different systems, including Baechi [31]. While these heuristics can provide efficient solutions, the resulting performance can be far from optimal as shown in our evaluation results comparing Baechi with Expert and Pesto.

7 CONCLUSION

This paper highlights the opportunity to improve DNN training time in frameworks like TensorFlow by employing smarter model placement and operation scheduling strategies. By leveraging concepts from integer programming and graph theory, we propose an efficient and optimal algorithm, Pesto, for joint model placement and operation scheduling. We integrate Pesto with TensorFlow and demonstrate non-trivial improvements (up to 31%, compared to Expert) in training time across modern, giant DNN models without incurring significant placement determination time. We also report the first placements for the giant Transformer and RNNLM models that provide significant training time reduction over the Expert strategy, representing a substantial saving in the training effort required for these models in practice.

8 ACKNOWLEDGMENT

This work was supported by NSF CNS grants 1750109, 1730128, and 1717588.

REFERENCES

[1] [n.d.]. Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types. Accessed: 2020-11-28.
[2] [n.d.]. Baechi: Fase Device Placement on Machine Learning Graphs (SoCC 2020). https://github.com/beomyeol/baechi. Accessed: 2021-03-28.
[3] [n.d.]. NVIDIA V100 TENSOR CORE GPUs. https://www.nvidia.com/en-us/data-center/v100/. Accessed: 2020-11-28.

[4] [n.d.]. NVLINK FABRIC: A FASTER, MORE SCALABLE INTERCONNECT. https://www.nvidia.com/en-us/data-center/nvlink/.

[5] [n.d.]. Pesto Source-code. https://github.com/PACELab/TF-Pesto. Accessed: 2020-11-28.

[6] [n.d.]. TensorFlow NMT GitHub. https://github.com/tensorflow/nmt. Accessed: 2020-11-28.

[7] [n.d.]. WMT 2016. http://www.statmt.org/wmt16/. Accessed: 2020-11-28.

[8] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. http://tensorflow.org/ Software available from tensorflow.org.

[9] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283.

[10] Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. 2019. Placeto: Learning generalizable device placement algorithms for distributed machine learning. In *Proceedings of the 33rd Conference on Neural Information Processing Systems (NeurIPS)*. 3983–3993.

[11] Dzmitry Bahdanau, Kyung Hyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations, ICLR 2015*.

[12] Ondrej Bojar, Christian Buck, Christian Federmann, Barry Haddow, Philipp Koehn, Johannes Leveling, Christof Monz, Pavel Pecina, Matt Post, Herve Saint-Amand, Radu Soricut, Lucia Specia, and Aleš Tamchyna. 2014. Findings of the 2014 Workshop on Statistical Machine Translation. In *Proceedings of the Ninth Workshop on Statistical Machine Translation*. Association for Computational Linguistics, Baltimore, Maryland, USA, 12–58. http://www.aclweb.org/anthology/W/W14/W14-3302

[13] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[14] Cédric Chevalier and François Pellegrini. 2008. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel computing* 34, 6-8 (2008), 318–331.

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

[16] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. 2012. Large scale distributed deep networks. *Advances in neural information processing systems* 25 (2012), 1223–1231.

[17] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems*. 1223–1231.

[18] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. 4171–4186.

[20] Shashwat Garg, Janardhan Kulkarni, and Shi Li. 2019. Lift and project algorithms for precedence constrained scheduling to minimize completion time. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 1570–1584.

[21] Ronald L. Graham. 1969. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics* 17, 2 (1969), 416–429.

[22] Ubaid Ullah Hafeez and Anshul Gandhi. 2020. Empirical Analysis and Modeling of Compute Times of CNN Operations on AWS Cloud. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 181–192.

[23] Claire Hanen and Alix Munier. 1995. An approximation algorithm for scheduling dependent tasks on m processors with small communication delays. In *Proceedings 1995 INRIA/IEEE Symposium on Emerging Technologies and Factory Automation. ETFA'95*, Vol. 1. IEEE, 167–189.

[24] Claire Hanen and Alix Munier. 1998. Performance of Coffman-Graham schedules in the presence of unit communication delays. *Discrete applied mathematics* 81, 1-3 (1998), 93–108.

[25] Julien Herrmann, Jonathan Kho, Bora Uçar, Kamer Kaya, and Ümit V Çatalyürek. 2017. Acyclic partitioning of large directed acyclic graphs. In *2017 17th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGRID)*. IEEE, 371–380.

[26] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Brian Kingsbury, et al. 2012. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal processing magazine* 29 (2012).

[27] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[28] JA Hoogeveen, Jan Karel Lenstra, and Bart Veltman. 1994. Three, four, five, six, or the complexity of scheduling with communication delays. *Operations Research Letters* 16, 3 (1994), 129–137.

[29] Yanping Huang, Yonglong Cheng, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, and Zhifeng Chen. 2018. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. *arXiv preprint arXiv:1811.06965* (2018).

[30] IBM. [n.d.]. *CPLEX Optimizer*. https://www.ibm.com/analytics/cplex-optimizer

[31] Beomyeol Jeon, Linda Cai, Pallavi Srivastava, Jintao Jiang, Xiaolan Ke, Yitao Meng, Cong Xie, and Indranil Gupta. 2020. Baechi: fast device placement of machine learning graphs. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 416–430.

[32] Beomyeol Jeon, Linda Cai, Pallavi Srivastava, Jintao Jiang, Xiaolan Ke, Yitao Meng, Cong Xie, and Indranil Gupta. 2020. Baechi: fast device placement of machine learning graphs. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 416–430.

[33] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond data and model parallelism for deep neural networks. *SysML 2019* (2019).

[34] Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. 2016. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410* (2016).

[35] Arthur B Kahn. 1962. Topological sorting of large networks. *Commun. ACM* 5, 11 (1962), 558–562.

[36] N Kalchbrenner, E Grefenstette, and Philip Blunsom. 2014. A convolutional neural network for modelling sentences. In *52nd Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics.

[37] Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth A Gibson, and Eric P Xing. 2016. STRADS: a distributed framework for scheduled model parallel machine learning. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–16.

[38] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997* (2014).

[39] Janardhan Kulkarni, Shi Li, Jakub Tarnawski, and Minwei Ye. 2020. Hierarchy-based algorithms for minimizing makespan under precedence and communication constraints. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2770–2789.

[40] Quoc V Le. 2013. Building high-level features using large scale unsupervised learning. In *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE, 8595–8598.

[41] Elaine Levey and Thomas Rothvoss. 2019. A (1+ epsilon)-approximation for makespan scheduling with precedence constraints using LP hierarchies. *SIAM J. Comput.* 0 (2019), STOC16–201.

[42] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R Tallent, and Kevin J Barker. 2019. Evaluating modern GPU interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2019), 94–110.

[43] Mitchell Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. (1993).

[44] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V Le, and Jeff Dean. 2018. A Hierarchical Model for Device Placement. In *International Conference on Learning Representations (ICLR)*.

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

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

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

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

[49] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3505–3506.

[50] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. 2019. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, Vol. 33. 4780–4789.

[51] Yassir Samadi, Mostapha Zbakh, and Claude Tadonki. 2018. E-HEFT: enhancement heterogeneous earliest finish time algorithm for task scheduling based on load balancing in cloud computing. In *2018 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 601–609.

[52] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. 2018. Mesh-TensorFlow: deep learning for supercomputers. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. 10435–10444.

[53] Kenneth W Stufflebeam Jr. 2006. Configurable PCI express switch which allows multiple CPUs to be connected to multiple I/O devices. US Patent 7,058,738.

[54] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2818–2826.

[55] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems* 13, 3 (2002), 260–274.

[56] J.D. Ullman. 1975. NP-complete scheduling problems. *J. Comput. System Sci.* 10 (1975), 384–393.

[57] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[58] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[59] Tao Yang and Apostolos Gerasoulis. 1994. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems* 5, 9 (1994), 951–967.

[60] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. 2014. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329* (2014).

[61] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. 2018. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 8697–8710.