
# Unity: Accelerating DNN Training Through Joint Optimization of Algebraic Transformations and Parallelization

Colin Unger, *Stanford University;* Zhihao Jia, *Carnegie Mellon University and Meta;* Wei Wu, *Los Alamos National Laboratory and NVIDIA;* Sina Lin, *Microsoft;* Mandeep Baines and Carlos Efrain Quintero Narvaez, *Meta;* Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, and Jamaludin Mohd-Yusof, *Los Alamos National Lab;* Xi Luo, *SLAC National Accelerator Laboratory;* Dheevatsa Mudigere, Jongsoo Park, and Misha Smelyanskiy, *Meta;* Alex Aiken, *Stanford University*

https://www.usenix.org/conference/osdi22/presentation/unger

This paper is included in the Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-28-1

Open access to the Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

NetApp®


# Unity: Accelerating DNN Training Through Joint Optimization of Algebraic Transformations and Parallelization

Colin Unger[†♠] Zhihao Jia[‡♭♠] Wei Wu[*◇] Sina Lin[§] Mandeep Baines[♭]
Carlos Efrain Quintero Narvaez[♭] Vinay Ramakrishnaiah[*] Nirmal Prajapati[*]
Pat McCormick[*] Jamaludin Mohd-Yusof[*] Xi Luo[♯] Dheevatsa Mudigere[♭]
Jongsoo Park[♭] Misha Smelyanskiy[♭] Alex Aiken[†]

*Stanford University*[†] *Carnegie Mellon University*[‡] *Los Alamos National Lab*[*]
*NVIDIA*[◇] *Microsoft*[§] *Meta*[♭] *SLAC National Accelerator Laboratory*[♯]

## Abstract

This paper presents Unity, the first system that jointly optimizes algebraic transformations and parallelization in distributed DNN training. Unity represents both parallelization and algebraic transformations as substitutions on a unified *parallel computation graph* (PCG), which simultaneously expresses the computation, parallelization, and communication of a distributed DNN training procedure.

Optimizations, in the form of graph substitutions, are automatically generated given a list of operator specifications, and are formally verified correct using an automated theorem prover. Unity then uses a novel hierarchical search algorithm to jointly optimize algebraic transformations and parallelization while maintaining scalability. The combination of these techniques provides a generic and extensible approach to optimizing distributed DNN training, capable of integrating new DNN operators, parallelization strategies, and model architectures with minimal manual effort.

We evaluate Unity on seven real-world DNNs running on up to 192 GPUs on 32 nodes and show that Unity outperforms existing DNN training frameworks by up to 3.6× while keeping optimization times under 20 minutes. Unity is available to use as part of the open-source DNN training framework FlexFlow at https://github.com/flexflow/flexflow.

## 1 Introduction

Deep neural networks (DNNs) are becoming progressively larger and computationally more expensive to train, and as they have grown, so has interest in optimizing their execution to reduce training times and improve scalability. Two key classes of optimizations shown to yield significant performance improvements across diverse model architectures are algebraic transformations and parallelization.

*Algebraic transformations* exploit operator identities to perform the underlying computation in a more efficient way, but ignore parallelization and distribution of training. Common examples of algebraic transformations include operator fusion, which merges two operators into a single semantically-equivalent operator whose computation is more efficient, and operator reordering, where the associativity or commutativity of sets of operators allows them to be reordered into more efficient configurations or to expose further optimization opportunities. More explanation of algebraic transformations, along with examples, is provided in Section 2.2.

*Parallelization*, in contrast, distributes operators over multiple devices, but does not change the way in which the underlying computation is performed. DNN training exploits a class of parallelism named *partition-n-reduce* [59], in which every distributed subcomputation of an operator must perform the same computation, and may only differ in the input data it consumes. The tensor computations in DNN training are particularly well-suited to this form of parallelism, and many *parallelism dimensions* along which to divide distributed operators have been identified, such as data [6], model [13], spatial [27], reduction [50], and pipeline [39]. For a detailed overview of these various approaches, see Section 2.1.

When applied effectively, these two techniques can improve training times by more than an order of magnitude. However, effective application is nontrivial. Rewriting the computation graph for maximum speedup can require many transformations, some of which may harm performance except in the context of a longer sequence of transformations [26]. The optimal parallel execution strategy for a model often requires simultaneously exploiting multiple parallelization dimensions and using different parallelization schemes for each operator [24]. Early work relied on the programmer to manually determine the correct optimizations to apply [6]. While manual optimization allows fine-grained control over the model's performance, it requires many hours of tuning by experts to achieve good performance. As the pace of new developments in model design has increased, manual optimization has struggled to scale beyond the most commonly used models.

Recent work has focused on automating optimizations. MetaFlow [26], TASO [25], and PET [58] propose algorithms for automatically generating and applying algebraic transformations by posing optimization as a search problem.

♠ Contributed equally.

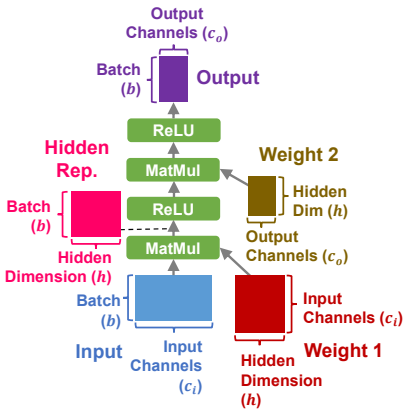
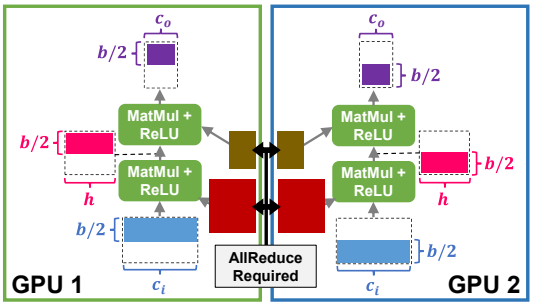



Figure 1: Computation graph for a 2-layer MLP.

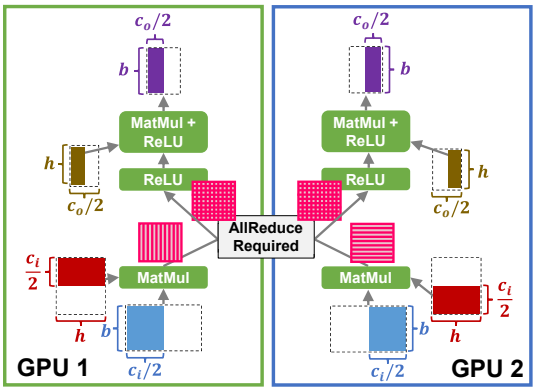
FlexFlow [27], automap [48], Tofu [59], and Whale [23] bring a similar approach to parallelism. These works present impressive benchmarks, yielding the impression that automating algebraic and parallelization optimization is a solved problem.

However, to reduce training time as much as possible, we want to apply both of these optimizations, but the most effective way to combine algebraic and parallelization optimizations is not obvious. The simplest solution is to apply them independently, in one of two orders: algebraic optimization followed by parallelization, or the reverse. The reverse order turns out to be problematic: since algebraic transformations can introduce new operations or replace existing ones, running algebraic optimization after parallelizations have been assigned can lead to the final solution having operations without assigned parallelizations (if the operation was created) or invalid parallelizations (if the operation was replaced). Workarounds can be used to fix invalid solutions by using default parallelization strategies or copying the strategies of nearby operators, but it is easy to find cases in which these workarounds lead to suboptimal solutions. As such, applying algebraic optimization before parallelization is the only option, but as we see in the next example, it can miss significant optimization opportunities.

Consider the computation graph shown in Figure 1, which represents a 2-layer multilayer perceptron (MLP). If we are optimizing independently (also referred to as "sequentially"), we start by applying algebraic transformations without considering parallelism. A typical algebraic optimizer will fuse the `MatMul` and `ReLU` operators to remove redundant memory loads and stores. The model is then parallelized (we consider only 2 GPUs for simplicity) resulting in Figure 2a: data parallelism is used for both operators and thus the weight gradients must be synchronized with an AllReduce. Since weight 1 has size $c_i h$ and weight 2 has size $hc_o$, the total communication is $2(c_i h + hc_o)$. Using a set of parameters for a basic image classification model for MNIST ($b = 64$, $h = 512$, $c_o = 10$, $c_i = 28 \times 28 = 784$) yields a total communication of $813,056d$ bytes, where $d$ is the element size.




(a) Sequential optimization.




(b) Joint optimization.

Figure 2: Comparing joint and sequential optimizations.

Instead of independently applying algebraic transformations and parallelization, we can combine them and solve a single joint optimization problem that discovers the solution in Figure 2b. By not fusing the first `MatMul` and `ReLU`, more efficient reduction parallelism can be used. This requires synchronizing the activation and gradient of the first `MatMul`'s output, but not the weights: a total inter-GPU communication of $4bh$, or $131,072d$ bytes for our MNIST example. Joint optimization reduces communication by $6\times$, which far exceeds the cost of not fusing the first `ReLU`.

As this example shows, joint optimization is necessary to maximize performance. However, it also poses significant challenges. The first is representation: existing frameworks perform optimizations on a model's computation graph. As discussed above, algebraic transformations can leave operators in the computation graph with unassigned or invalid parallelizations. To prevent such invalid solutions from arising during search, we need a representation that allows algebraic transformations to consider the current parallelization before being applied. Further discussion of the representation challenges is in Section 3.4.

The second challenge is scalability: existing search-based approaches already struggle to scale up to large models and GPU counts. Improvements have been made for algebraic transformations alone [62], but the complexity of these solutions makes adding parallelization a daunting task. Simultane-

ously considering both optimization classes only exacerbates this problem by exponentially increasing the search space size. For joint optimization to be practical, search algorithms must improve on the scalability of past techniques.

## 1.1 Unity's Approach

The key idea behind Unity is to represent both algebraic transformations and parallelization as graph substitutions on a unified *parallel computation graph*, and then to use a hierarchical search algorithm to efficiently identify which combination of substitutions yields the best performance. Figure 3b shows an overview of Unity, which differs from existing frameworks in the following ways:

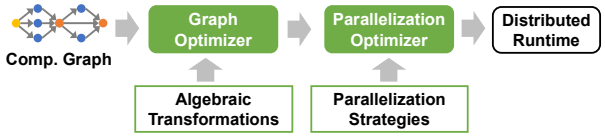
**Unified graph representation.** We introduce the *parallel computation graph* (PCG)[1] as a unified representation of distributed DNN training that simultaneously expresses computation, parallelism, and data movement. All parallelization strategies used in existing frameworks can be represented as specific PCGs, and parallelization and algebraic transformations as sequences of graph substitutions. pONNX [57] previously proposed merging computation and parallelism into a single graph, but certain design decisions prevent Unity-style joint optimization. For a detailed comparison, see Section 3.4.

**Transformation generation and verification.** Unity does not require users to explicitly define possible parallelization strategies for DNN training. Unlike prior work that automatically generates parallelization strategies [59] or algebraic transformations [25], by using the PCG Unity is able to generate both kinds of transformations with a single approach, as well as hybrid algebraic-parallelization optimizations absent in prior automated approaches. Automatically generating and verifying transformations greatly reduces the engineering effort required to support different parallelism dimensions and enables extensibility to new operators.

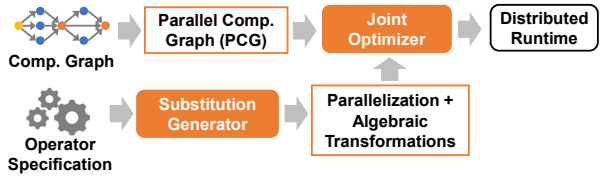
**Joint optimization.** Unity uses a hierarchical search algorithm to discover highly optimized PCG substitutions and device placements while maintaining scalability to models with hundreds of operators distributed over hundreds of GPUs. Unity's cost model includes both computation and communication time, and the search algorithm handles custom network topologies and heterogeneous compute devices. Despite the exponentially larger search space being considered, Unity outperforms existing search-based approaches (see Section 6).

The rest of this paper provides additional background (Section 2), discusses Unity's design and implementation (Sections 3, 4, and 5), and evaluates its performance on seven real-world DNNs (Section 6). For widely-used DNNs highly optimized by existing frameworks, such as BERT [14], Unity matches the performance of existing expert-designed strategies while being completely automated. For complex DNN architectures with a mixture of compute- and communication-intensive operators, such as DLRM [41] and CANDLE-Uno [1], Unity is up to 3.6× faster than existing frameworks.

[1]To prevent ambiguity, we use the term *computation graph* strictly to refer to the conventional computation graph used in prior work, and *parallel computation graph* or PCG to refer to Unity's new unified representation.




(a) Existing approach.




(b) Unity's approach.

Figure 3: Comparing existing DNN frameworks and Unity.

# 2 Background

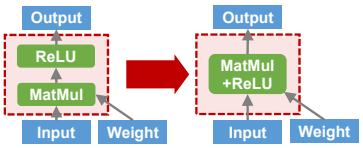
We first provide a brief overview of the two classes of optimizations that Unity exploits, parallelization (Section 2.1) and algebraic transformations (Section 2.2), as well as a discussion of how they are represented in existing systems (Section 2.3). For a discussion of how Unity interacts with other classes of optimizations, see Section 8.

## 2.1 Parallelization

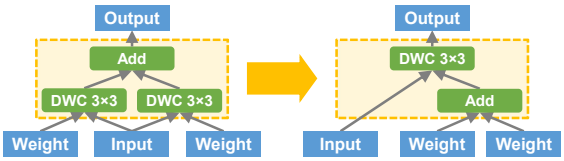
The massively parallel nature of tensor algebra creates many opportunities for parallelizing DNN training. We identify six primary forms of parallelism leveraged in DNN systems:

1. *Data parallelism* is the most common approach used in existing frameworks [6, 9, 42]. Data parallelism keeps a replica of the entire DNN model on every device and assigns each a subset of the training data.
2. *Model parallelism* divides a DNN model into disjoint sub-models and trains each sub-model on a dedicated device.
3. *Spatial parallelism*[2] divides the spatial dimensions of a tensor (e.g., the height and width of images) into multiple partitions, each of which is assigned to a specific device [27]. Spatial parallelism often requires synchronizing the shared elements (e.g., the shared pixels along the boundary of different sub-images) between devices.
4. *Reduction parallelism* exploits the linearity of tensor algebra operators. For a matrix multiplication $C = A \times B$, reduction parallelism splits $A$ along its columns and $B$ along its rows as follows: $A = [A_1, \ldots, A_n]$, $B = [B_1^T, \ldots, B_n^T]^T$ The matrix multiplication is distributed across $n$ devices, with the $i$-th device computing $C_i = A_i \times B_i$. An extra reduction afterward recovers the original result: $C = \sum_i C_i$.

[2]Spatial parallelism was called *attribute parallelism* in [27].




(a) Basic operator fusion.




(b) A more complex algebraic transformation.

Figure 4: Example algebraic transformations. `DWC` stands for `DepthwiseConv` (i.e., depth-wise separable convolution).

5. *Pipeline parallelism* exploits the opportunity to parallelize across different training iterations [39].
6. *Operator-specific parallelism.* The introduction of new DNN operators provides operator-specific parallelization opportunities. For example, the following equation shows the batched matrix multiplication used in Transformer [54]: $output(s,h,o) = \sum_i input(s,h,i) \times weight(h,o,i)$. This differs from typical matrix multiplication in that it applies a different weight for each input sample. As a result, these batched matrix multiplications across attention heads can be run in parallel (i.e., the $h$ dimension) without any tensor replication or synchronization.

Most parallelizations are not pure performance optimizations, but are instead trade-offs among different cost metrics. For example, applying data parallelism reduces per-device computation time at the cost of increased memory usage and data movement for storing and synchronizing model parameters. Thus, DNN operators typically require a combination of these forms of parallelism to achieve optimal performance.

## 2.2 Algebraic Transformations

Algebraic transformations are very diverse and are not as easily categorized as the forms of parallelism, so we instead provide examples. For a more comprehensive exploration of algebraic transformations, see [25].

The most basic algebraic transformation is operator fusion, shown in Figure 4a. Unfused, the device needs to load and store activations to and from memory twice, once before and after each operator. If the two operators are fused, however, the combined kernel can compute the `ReLU` operation as it stores the outputs of the `MatMul` back to memory.

For a more complex example, see Figure 4b. By exploiting `DepthwiseConv`'s linearity, a computation that previously required two `DepthwiseConv` operations now only requires one plus an additional `Add`, effectively halving the amount of computation needed.

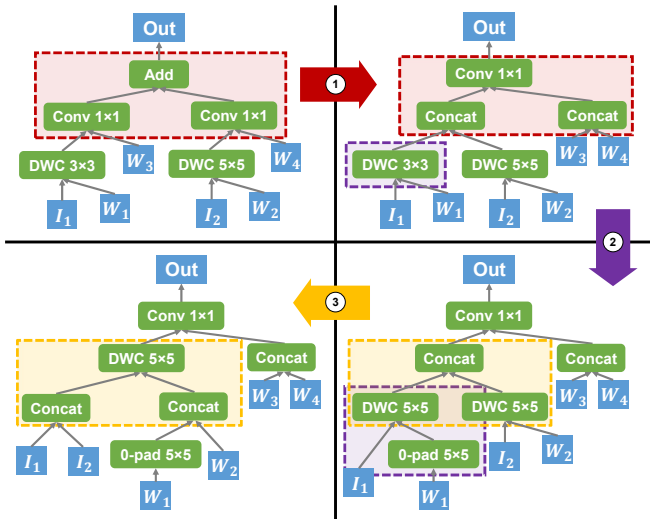



Figure 5: Compositions of small algebraic transformations can lead to significant changes.

Small algebraic transformations can be composed to create large changes. Consider the sequence of transformations shown in Figure 5: while each individual transformation is relatively small, the final output is radically different from the original computation graph. Also, notice that not all performance gains are realizable in a single transformation: for example, moving from graph 2 to graph 4 reduces the amount of computation by reducing the number of `DepthwiseConv` operations performed, but it is first necessary to pass through graph 3 which performs worse than either graph 2 or 4.

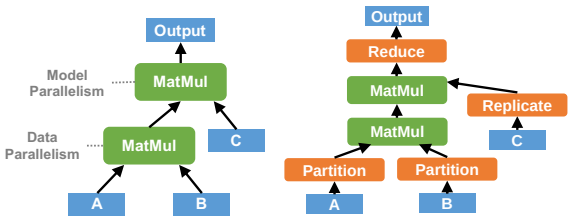
## 2.3 Intermediate Representations

Most existing optimizing frameworks represent a DNN architecture as a *computation graph*[3]: a node is a mathematical tensor operator (e.g., matrix multiplication, etc.), and an edge is a tensor (i.e., $n$-dimensional array) passed between operators. An example computation graph is shown in Figure 6a. Algebraic transformations are performed by iteratively applying graph substitutions, and the model is parallelized by assigning each node a set of parallelism annotations.

This representation has two limitations. First, while using distinct representations for algebraic transformations (i.e., graph substitutions) and parallelization (i.e., node annotations) is convenient, it hinders joint optimization. The key issue is that algebraic transformations can add or replace nodes in the graph, while parallelization views the computation graph as static and thus cannot handle these newly-created, unannotated nodes. This prevents interleaving the two search algorithms, since at any time a substitution can transform a valid parallelization into an invalid one.

Second, a computation graph does not explicitly capture

[3] Alternative representations are discussed in Section 3.4 and Section 7.




(a) Computation graph. (b) Parallel computation graph.

Figure 6: Comparing computation graph and PCG. Both graphs describe the same parallelization of two consecutive matrix multiplications $(A \times B) \times C$ (a simplified form of attention). The green and orange boxes denote regular DNN operators and Unity's new parallelization operators (see Section 3.3) respectively.

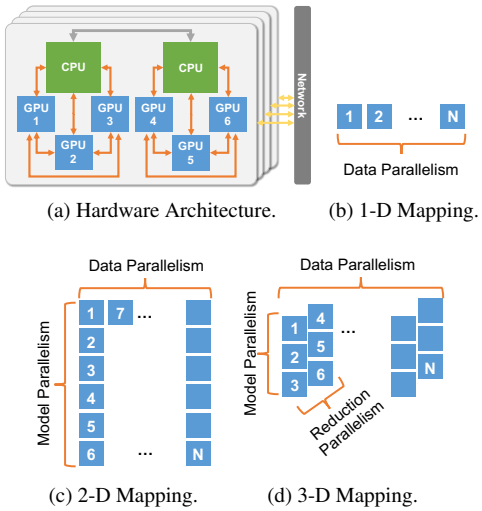
the communication costs associated with parallelism. This absence makes it difficult for algebraic transformations to reason about the impact on the performance of the final model.

## 3 Parallel Computation Graph

To solve the shortcomings of the existing model representations described in Section 2.3, we introduce the *parallel computation graph* (PCG) as a unified representation of distributed DNN training that is capable of simultaneously expressing computation, parallelism, and communication. The PCG allows Unity to consider both algebraic transformations and parallelization as graph substitutions on a common graph. While the PCG is not the first to merge computation and parallelization into a single graph, the PCG is tailored for optimization and as such differs from prior unified graph representations in key aspects, which we discuss in Section 3.4.

PCGs extend the existing computation graph representation by allowing nodes to represent changes in parallelization in addition to mathematical tensor operations, and edges to represent distributed movement of tensor data in addition to data dependence. A set of *parallelization operators* are added that allow PCGs to express all existing parallelization strategies and provide an explicit representation of data movement and its associated costs during training. Additionally, each operator in a PCG is associated with a *machine mapping*, denoting how the execution of the operator is mapped to individual processors in a parallel machine. Figure 6b shows an example of a PCG.

Sections 3.1, 3.2, and 3.3 provide a brief description of the tensor representation, machine mappings, and parallelization operators, respectively. Finally, Section 3.4 discusses the design decisions that make the PCG uniquely suited for joint optimization, and how it differs from alternative unified graph representations.




(a) Hardware Architecture. (b) 1-D Mapping.

(c) 2-D Mapping. (d) 3-D Mapping.

Figure 7: Example machine mapping for a compute node in our evaluation. (a) shows the node's hardware architecture, where and orange and grey arrows denote NVLink and X-Bus. Numbers in mapping examples denote GPU ids.

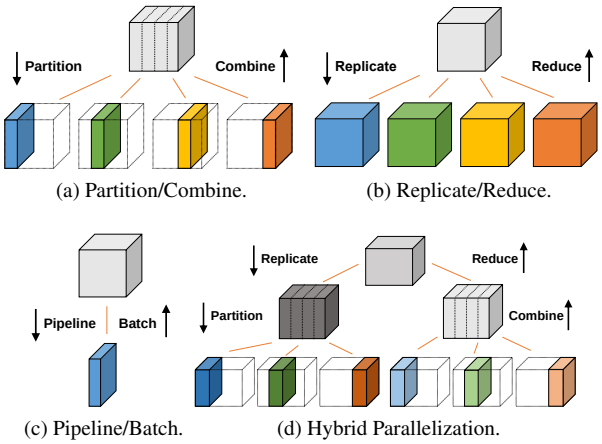
### 3.1 Tensor Representation

Unity models tensors as a set of data dimensions, each of which has two fields: a size and a degree. The degree field specifies the number of partitions the tensor has been divided into along that dimension. Every tensor also includes a special *replica dimension*, which represents the number of replicas of that tensor's data.

### 3.2 Machine Mappings

Each operator in a PCG is associated with a *machine mapping*, an $n$-dimensional array of devices/processors that specifies on which device to run each piece of the operator's computation. More formally, given an operator and a set of $n$ applicable parallel dimensions with degrees $d_1, \ldots, d_n$, Unity divides the operator into $d_1 \times d_2 \times \ldots \times d_n$ parallel tasks, which we reference with tuple indices of the form $(i_1, \ldots, i_n)$ where $0 \leq i_k < d_k$. A machine mapping is a map from task indices $(i_1, \ldots, i_n)$ to individual GPUs that will be used to run that parallel task. For convenience, we also define the machine mapping of an entire PCG to be the set of machine mappings of each of its constituent operators.

Figure 7 shows some example machine mappings for the Summit compute nodes [55] used in our evaluation. The hardware architecture is depicted in Figure 7a. Figure 7b shows a basic 1-D machine mapping for data parallelism, while Figure 7c shows a 2-D machine mapping of a hybrid parallelization strategy combining data and model parallelism, where model parallelism is applied across GPUs within the same compute node and data parallelism across distinct compute nodes. Figure 7d shows a 3-D machine mapping where we apply model parallelism across GPUs attached to the same



(a) Partition/Combine. (b) Replicate/Reduce.

(c) Pipeline/Batch. (d) Hybrid Parallelization.

Figure 8: Parallelization operators in Unity.

CPU, reduction parallelism across GPUs attached to different CPUs but on the same compute node, and data parallelism across different compute nodes.

Unity includes a comprehensive set of machine mappings that capture effective usages of a parallel machine. In addition, developers can register custom machine mappings tailored to specific hardware architectures. For example, when node pairs in a cluster have different network bandwidths and latencies, an extra dimension can be added to the existing machine mappings to represent node-level locality.

Machine mappings provide two key desirable properties: *expressiveness* and *scalability*. All effective distributions of parallel tasks in a PCG can be captured in just a few machine mappings, and complex features of a machine's hardware architecture can be easily leveraged through adding additional machine mappings. Machine mappings also allow Unity to capture all effective device assignments while remaining linear in the number of devices and aid Unity's search algorithm by removing inefficient assignments from consideration.

## 3.3 Parallelization Operators

Unity uses six parallelization operators to capture the computation and communication costs associated with different parallelization strategies. These six are further divided into three pairs, where one operator is the "back propagation" of the other (e.g., when back propagation is done on `Partition` it becomes semantically equivalent to `Combine`, and the same in reverse). The three pairs are:

1. *Partition and Combine*: `Partition` and `Combine` change a tensor's degree of parallelism. More specifically, `Partition` increases the parallelism degree of a tensor dimension by splitting the dimension into multiple equal-sized partitions, as shown in Figure 8a. `Combine` performs the reverse: reducing a tensor's degree of parallelism by concatenating multiple partitions into one.
2. *Replicate and Reduce*: `Replicate` and `Reduce` control the parallelism degree of the replica dimension by copying and summing tensors, as shown in Figure 8b. Parameter synchronization is naturally captured as the back propagation of `Replicate` operations applied to weight tensors.
3. *Pipeline and Batch*: `Pipeline` splits a tensor dimension into equal size partitions and processes one partition at a time, while `Batch` aggregates tensors across iterations (see Figure 8c). Note that `Pipeline` does not modify the parallelism degree of a tensor dimension, but instead reduce its size.

As a basic demonstration of the PCG's expressiveness, Figure 9 illustrates how Unity's six parallelization operators can represent some example parallelization strategies from Section 2.1. These parallelization operators can also be composed to create hybrid parallelism. Figure 8d shows an example that applies `Replicate` and `Partition` on the same tensor dimension, replicating the tensor and partitioning each replica. To improve efficiency, Unity replaces particular sequences of parallelization operators with fused versions at run time (e.g., a `Reduce` followed by a `Replicate` can be implemented as an AllReduce).

## 3.4 Discussion and Comparison

Unity's decision to use the PCG instead of an annotated computation graph is driven by how easily the representations lend themselves to joint search and not a fundamental limitation of annotated computation graphs. Theoretically, there exist annotation languages isomorphic to the PCG, but attempts to design such a language quickly lead to a number of difficulties.

First, because each operator can use different forms of parallelism, including operator-specific forms of parallelism, the number of annotations quickly grows prohibitively large. Which annotations are supported by which operators, along with their semantics and composition, must then be baked into the representation itself. By comparison, Unity's PCG moves this knowledge into the PCG substitutions, which are generated automatically. This separation of concerns makes the core of Unity simpler and easier to maintain.

Second, not explicitly representing communication forces communication patterns along dataflow edges to be reconstructed from their source and destination node annotations, which is difficult due to the expressive forms of parallelism Unity considers. Specifically, supporting $n$ parallelism dimensions requires considering up to $2^n$ different subsets of these dimensions and thus $2^n \times 2^n = 4^n$ potential communication patterns between operators. Unity explicitly represents communication patterns throughout search, obviating the need for a complex analysis to reconstruct them. Representing these patterns via a small set of parallelization operators also allows Unity to easily recognize and optimize common communication patterns, such as executing a pair of `Reduce-Replicate`

operators as an AllReduce. In an annotated computation graph, these optimizations become entangled with the code for reconstructing the communication patterns themselves, adding significant complexity and implementation effort.

Finally, jointly optimizing an annotated computation graph is challenging, as algebraic transformations can introduce new operators which, since they have not yet been parallelized, lack annotations. As such, the internal representation becomes underspecified and the cost becomes undefined. It is possible to add an additional mechanism to "fill in" these missing annotations such as inserting a random annotation, a fixed value, a value from a neighboring node (though this becomes challenging when neighboring nodes have differing parallelizations), or evaluating the valid parallelizations and choosing the best one. However, Unity's PCG avoids this additional complexity by representing each parallelization strategy for the new operator as one or multiple PCG substitutions, offering an efficient and uniform approach to joint optimization.

**pONNX.** Unity is not the first to integrate computation and parallelism into a single graph: pONNX [57] proposed doing so using `Split`, `Concat`, and custom operators `Send` and `Recv`. However, Unity focuses on optimization while pONNX is designed as a serialization format, leading to critical differences.

First, an operator in pONNX with a parallelism degree of $n$ is duplicated $n$ times, requiring an optimizer to reconstruct the operator from multiple nodes. Unity simply adds a parallelism operator so the operator remains a single node in a PCG.

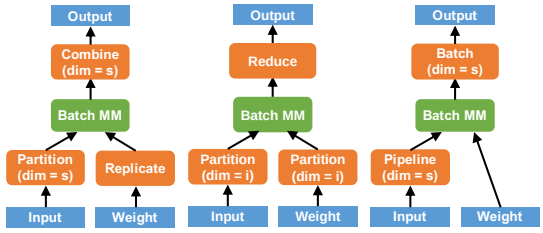
Second, pONNX assigns every communication its own `Send`/`Recv` node, which dramatically increases the size of the graph. Since communication patterns in DNN training are highly regular, Unity eschews materializing every communication in favor of optimizing communication patterns (e.g., `Reduce`, `Replicate`, etc.), which allows Unity to represent communication costs without reasoning about individual communications.

Finally, pONNX makes device placement part of the operator, while Unity represents it separately as a machine mapping. This allows Unity's search to optimize device assignments separately and to ignore the symmetries created by a large number of compute devices with identical capabilities.

Additional unified representations have been proposed [47, 48], which are discussed in Section 7.

## 4 Graph Substitutions

Since Unity represents both algebraic transformations and parallelization as graph substitutions, the effectiveness of joint optimization relies on having an appropriate set of graph substitutions. The number of potential substitutions increases exponentially with size, so Unity represents large and complex algebraic transformations and parallelization strategies as compositions of small PCG substitutions. For example, Figure 10 shows the sequence of substitutions for the hand-tuned parallelization strategy used in Megatron-LM [50].




(a) Data/Sample. (b) Reduction. (c) Pipeline.

Figure 9: Representing different parallelization strategies for batched matrix multiplication with a PCG. $s$, $i$, $o$, and $h$ indicate the sample, input channel, output channel, and attention head dimensions, respectively.

**Substitution generation.** To reduce the engineering effort to support new parallelization strategies, Unity automatically generates and formally verifies all valid PCG substitutions up to a fixed size to serve as a "basis set" from which the search algorithm can construct sophisticated optimizations. This also allows Unity to not only automatically discover algebraic transformations and parallelization strategies, but also to find novel hybrids of the two missed by prior approaches. To do so, Unity adopts TASO's super-optimization approach [25].

As in TASO, Unity discovers substitutions in two steps: first it uses a fast heuristic to identify candidate substitutions, and then it uses a more expensive formal verification to ensure correctness. To find candidate substitutions, Unity enumerates all possible PCGs up to a fixed size. Note that this fixed size does not limit the size of the transformations Unity can apply, as many larger substitutions are compositions of smaller ones.

For each generated PCG, Unity computes a *fingerprint*: a hash of the PCG's output tensors generated by evaluating the PCG on some fixed input tensors. To allow Unity to account for parallelization, we extend the fingerprint function in TASO [25] to include the parallelism degree of each tensor dimension. A pair of PCGs is considered a candidate substitution if both PCGs have an identical fingerprint. The addition of parallelism causes Unity to discover 651 new candidate substitutions beyond the 743 previously identified by TASO.

**Substitution verification.** Similar to TASO, Unity formally verifies the new substitutions using an automated theorem prover (Z3 [12] in our implementation). Operator specifications are provided in first-order logic, where an operator is represented as a function of its inputs and configuration parameters. For example, $\texttt{Reduce}(d,x)$ defines a `Reduce` operator with input $x$ and parallelism degree $d$. The fact that `Reduce` commutes with matrix multiplication is captured by the following operator property (where $\texttt{Replicate}(d,y)$ represents a `Replicate` with input $y$ and parallelism degree $d$):

$$\forall d,x,y.\ \texttt{Matmul}(\texttt{Reduce}(d,x),y) = \texttt{Reduce}(d,\texttt{Matmul}(x,\texttt{Replicate}(d,y)))$$

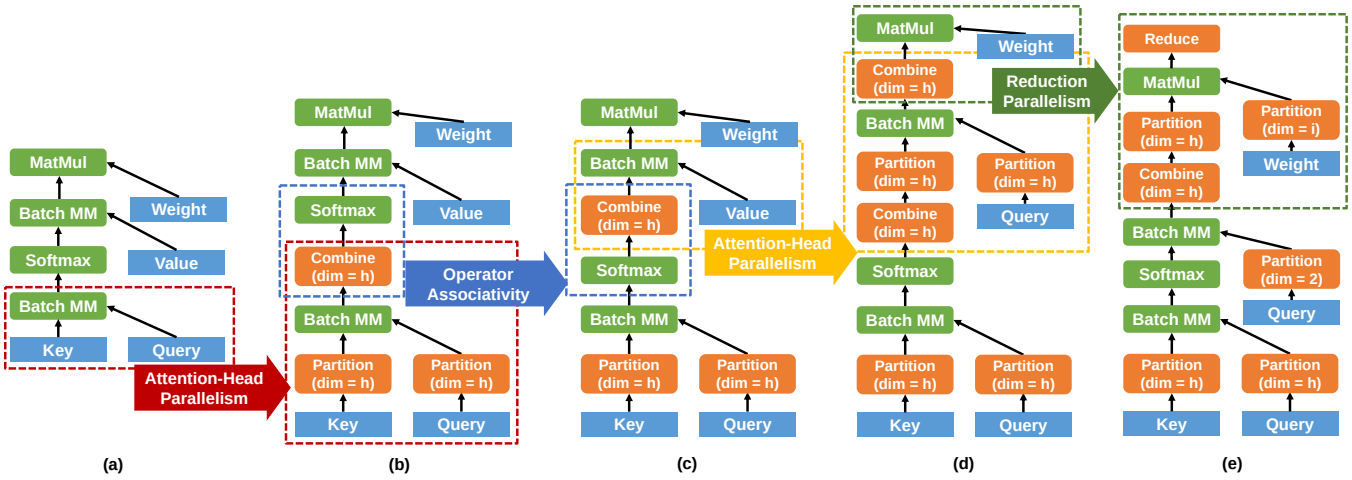



Figure 10: Representing the hand-tuned parallelization strategies used in Megatron-LM [50] as a sequence of basic graph substitutions in Unity. `BatchMM` and `MatMul` are batched and regular matrix multiplications, respectively. Each arrow denotes a graph substitution, where the dotted subgraphs in the same color are the source and target graph of the substitution. For `Partition` and `Combine`, the parentheses indicate the data dimension for which they are performed.

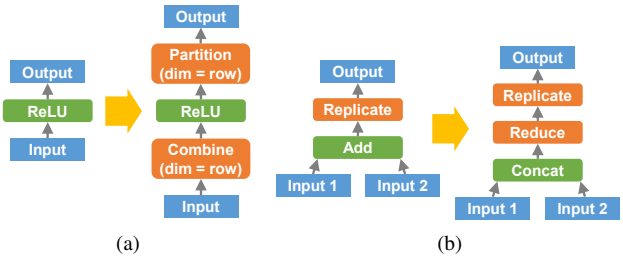



Figure 11: Substitution (a) shows that spatial parallelism is valid for `ReLU`. Substitution (b) demonstrates a hybrid algebraic-parallel transformation: transforming an `Add` into a `Concat` followed by a `Reduce` allows Unity to use the more efficient AllReduce communication pattern.

We follow TASO's methodology for developing operators' parallelization properties: we attempt to formally verify all candidate substitutions using Z3, and when a substitution cannot be verified but is correct, we add the missing operator properties. This procedure was repeated until all 651 new substitutions discovered by Unity were verified. Overall, we introduced 33 operator properties in addition to the 43 properties from TASO [25, Table 2] to verify all PCG substitutions.

Combined, the substitution generation and verification process takes a total of 30 minutes. Since the available substitutions only change on the addition of new operators or forms of parallelism, this process can be run entirely offline so as not to impact the execution time of Unity's joint search algorithm.

**Example Substitutions.** Most new substitutions generated by Unity simply state the parallelism valid for an operator. For instance, the substitution in Figure 11a indicates that `ReLU` supports spatial parallelism in the row dimension. However, combining algebraic transformations and parallelization also yields novel hybrids, such as the example shown in Figure 11b, where Unity identifies that an `Add` operator is equivalent to a `Concat` followed by a `Reduce`. In the left PCG, when `Input 1` and `Input 2` are located on separate devices and `Output` is required to be replicated across those same devices, `Input 1` and `Input 2` would have to be sent to and from a single device to be added. By applying this transformation, Unity is able to merge the input tensors into a single distributed tensor through a `Concat` (which moves no data) and replace the communication with a `Reduce` followed by a `Replicate` (which is implemented as an AllReduce).

## 5 Joint Optimization

This section describes Unity's search algorithm for jointly optimizing algebraic transformations and parallelization. The core problem is as follows: given a PCG (Section 3), a set of operator-level machine mappings (Section 3.2), and a set of PCG substitutions (Section 4), find (1) a sequence of PCG substitutions and (2) a machine mapping for the resulting PCG that minimize the per-iteration training time. A key challenge is the exponentially larger search space created by unifying algebraic transformations and parallelization. The search must also scale to both complex DNNs (i.e., a large input PCG) and large numbers of compute devices (i.e., a large set of operator-level machine mappings).

Unity uses a three-level hierarchical search algorithm, depicted in Figure 12. In simplified form, Unity breaks an input PCG into subgraphs, determines an optimized sequence of substitutions for each subgraph (which requires determining the optimized machine mapping for each candidate), and then

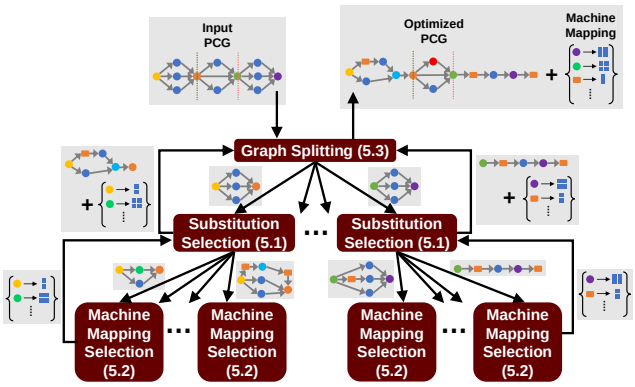



Figure 12: High-level depiction of Unity's hierarchical search.

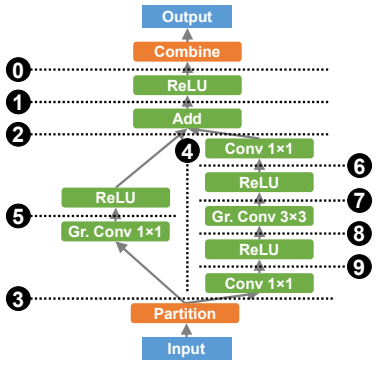



Figure 13: Applying sequence and parallel graph splits on a data-parallel ResNeXt module. Horizontal and vertical dotted lines refer to sequence and parallel splits, respectively, and numbers indicate the order they are applied.

combines these sub-solutions to produce the final output. This allows Unity to scale to DNNs with over 300 operators and machines with 192 GPUs while keeping search times below 20 minutes, which is negligible compared to the hours or days needed to train modern DNNs.

In the following section, we provide a more detailed description of Unity's search algorithm. Sections 5.1, 5.2, and 5.3 describe the three levels of Unity's search algorithm, starting from the middle layer (*substitution selection*), then the lowest (*machine mapping selection*), and finally introducing the highest level (*graph splitting*) as an optimization to help Unity scale to large DNNs. Afterward, we briefly address Unity's cost estimation and how the search algorithm can be tweaked to integrate pipeline parallelism.

## 5.1 Substitution Selection

Unity uses the cost-based backtracking search algorithm from TASO [25] to identify a sequence of substitutions that minimizes the execution time of an input PCG. Unity maintains a queue of candidate PCGs sorted by their execution times, and until the queue is emptied or a fixed budget is exceeded, Unity iteratively removes the best candidate from the queue and uses it to generate new candidates by applying every available substitution at every location in the PCG whenever applicable. Candidate PCGs with execution times that are a threshold factor times worse than the best candidate PCG seen so far are pruned, while the rest are inserted into the queue. The threshold factor allows the user to balance the search time and amount of exploration. In our experiments, we use a threshold factor of 1.05.[4]

This algorithm allows Unity to explore arbitrary sequences of substitutions, but requires an accurate cost estimator to evaluate the execution time of each candidate PCG. Since a PCG contains only the parallelization of each operator but not the devices to which it is assigned (i.e., the machine mapping), this cost estimator must first determine an optimized machine mapping. An efficient algorithm must be used to identify this mapping, as the cost estimator is called for every candidate PCG. Section 5.2 introduces our algorithm to find optimized machine mappings.

[4]This specific value was chosen to match [25].

## 5.2 Finding Optimized Machine Mappings

The lowest level of Unity's search algorithm identifies the optimized machine mapping for a candidate PCG. The key observation behind this level is that most modern DNN architectures consist of linear chains of independent strands of parallel computation. For example, ResNeXt [19] is built around two parallel strands of convolutions (see Figure 13), which are repeated to form the final model. Unity leverages this structure by recursively decomposing these linear chains and parallel strands into independent subgraphs through *sequence* and *parallel graph splits* respectively. Figure 13 demonstrates how sequence and parallel graph splits can be iteratively applied to decompose a ResNeXt module into recursive sub-problems which can be solved via dynamic programming.

A *sequence graph split* partitions an input PCG $\mathcal{G}$ by finding a postdominator node $n$, such that all paths from the inputs to the outputs of $\mathcal{G}$ go through $n$. This post-dominator node splits $\mathcal{G}$ into two disjoint subgraphs $\mathcal{G}_1$ and $\mathcal{G}_2$. Since all of $\mathcal{G}_2$ depends on $n$, and $n$ depends on all of $\mathcal{G}_1$, every operator in $\mathcal{G}_1$ must complete before any in $\mathcal{G}_2$ can start. This reduces the task of finding an optimized machine mapping for $\mathcal{G}$ to optimizing machine mappings for $\mathcal{G}_1$, $\mathcal{G}_2$, and $n$. For example, for split ❶ in Figure 13, assuming no other splits (such as ❼) had already been applied, $n$ would be the `Add` node, $\mathcal{G}_1$ would be all the nodes from `Input` up to but not including `Add`, and $\mathcal{G}_2$ would be all the nodes after the `Add` until `Output`.

A *parallel graph split* partitions a PCG $\mathcal{G}$ into independent subgraphs whose computations can be performed in parallel. In this case, Unity considers two potential ways of running the sides $\mathcal{G}_1$ and $\mathcal{G}_2$: in sequence (with access to the full machine resources) or in parallel (with each side given a disjoint share of the available resources) and chooses the faster one. Unity

does not allow combinations of serial and parallel execution, in which branches are run partially in parallel and partially in serial. While this eliminates certain strategies, considering them would significantly reduce Unity's scalability as it requires analyzing exponentially many interleavings of operators, and as evidenced by the results in Section 6, these strategies are not necessary to achieve good performance. To determine how to partition the available resources when running in parallel, Unity iterates over all possible resource quantities that can be assigned to each side. By considering resource quantities, Unity ignores redundant divisions that differ only in which GPUs are assigned and not in the number and location of these GPUs, replacing an exponential search over all subsets of devices with a quadratic search over resource quantities.

As an additional optimization, Unity maintains a cache of the selected machine mappings for all subgraphs. Since substitution selection generates a new candidate for each substitution, and each substitution modifies only a small part of a PCG, many candidate PCGs have most of their subgraphs in common with other candidates. This allows Unity to skip computing the cost and machine mapping of all but the part of the PCG modified by the substitution under consideration.

## 5.3 Scaling to Large Graphs

Even with the dynamic programming algorithm and cross-invocation caching, the search algorithm described so far fails to scale to large models. To understand why, we examine how the number of candidate PCGs in substitution selection scales with the size of the input PCG.

As described in Section 5.1, at each iteration Unity generates a candidate PCG for every possible application of each substitution. In the worst case this would require examining $O(2^{gs})$ candidates, where $g$ is the number of nodes in the PCG and $s$ is the number of substitutions Unity considers. In practice $s$ has limited impact on search time as only a small fraction of the substitutions Unity considers can be applied to any one model, but for large models the exponential behavior of $g$ becomes problematic.

To solve this, we borrow from Section 5.1 and decompose the PCG into independent sequential subgraphs. However, this approach prevents applying substitutions across these splits, which is problematic since Unity uses substitutions to represent parallelization. Thus, naive graph splitting would reduce the parallelism degree across all splits to 1, eliminating many common and important parallelization strategies, such as using data parallelism across the entire model.

Unity addresses this issue by explicitly searching for the optimal parallelization across every split location. More specifically, for every possible partitioning of the tensor communicated across the split, Unity optimizes the resulting two subgraphs under the condition that the first subgraph's output and the second subgraph's input must both match the partitioning under consideration. When either subgraph does not meet this condition, parallelization operators are inserted to ensure any communication cost arising from a change in partitioning is accounted for.

This method works for `Partition` and `Combine` but encounters a problem with `Replicate` and `Reduce`. For example, consider the case of the tensor crossing the split location having its replica degree fixed to 2 by the search algorithm. To coerce the first subgraph to output a tensor in this format, the search algorithm could insert a `Replicate` as its final operation, and the algorithm similarly could insert a `Reduce` as the first operation of the second subgraph. However, this will incorrectly scale the tensor by a factor of 2! The core issue is that unlike `Partition` and `Combine`, `Replicate` and `Reduce` are not inverses of each other. Fortunately, since reduction parallelism that spans many nodes of a computation graph is rarely useful in practice, we limit the partitionings across splits to only those with a replica degree of 1.

To reduce the number of algebraic transformations these splits prevent, Unity follows MetaFlow [26] and chooses split locations that disrupt the fewest substitutions while maintaining a minimum subgraph size $k$.[5] Thus graph splitting reduces the worst-case number of candidate PCGs from exponential in $g$ to linear in $g$, specifically from $O(2^{gs})$ to $O\left(\frac{gp}{k} \times 2^{ks}\right)$ where $p$ is the number of valid tensor partitionings.

**Cost estimation.** To estimate operator run times and communication costs we use similar methods as prior work [24, 27]. More accurate cost models are possible [47], but we have not noticed any issues caused by inaccuracies in our model.

**Pipeline parallelism.** When considering pipeline parallelism, Unity adopts the 1F1B schedule (i.e., interleaving forward and backward micro-batches on each device) and the weight update semantics from PipeDream-2BW [40], which achieves both high training throughput and low memory footprint. To reduce the search space, Unity only considers strategies where pipeline parallelism is applied to *all* operators in a PCG, since a non pipeline-parallel operator in the PCG would disable the benefits of pipeline parallelism. In addition, similar to prior work [20, 39, 65], Unity only considers sequential pipeline parallelism where each stage only communicates with a single next stage in the pipeline (except for the last stage, which directly performs back propagation after forward processing). Unity also follows prior work [16, 20, 53] in assuming that the number of micro-batches in a mini-batch is much larger then the number of pipeline stages so the additional latency introduced by pipeline initialization can be ignored. These constraints allow Unity to explore a comprehensive search space that includes existing pipeline parallelism strategies while maintaining reasonable search time. The search algorithm is also slightly modified: instead of using per-iteration run time as a proxy for throughput, we

[5] Our experiments use $k = 10$ as it strikes a balance between keeping subgraph sizes small enough for good scalability while blocking relatively few substitutions.

| Task | Architecture | Dataset |
|---|---|---|
| Image Classification | ResNeXt-50 [60] | ImageNet [46] |
| | Inception-v3 [51] | ImageNet [46] |
| Language Models | BERT-Large [14] | WikiText-2 [35] |
| Recommendation Systems | DLRM [41] | Criteo Kaggle [4] |
| | XDL [28] | Criteo Kaggle [4] |
| Precision Medicine | CANDLE-Uno [3] | Dose response data [1] |
| Regression | MLP [17] | Synthetic data |

Table 1: Overview of the seven DNNs evaluated.

maximize the throughput directly.

# 6 Evaluation

## 6.1 Implementation and Experimental Setup

Unity is implemented on top of FlexFlow [27], a distributed multi-GPU runtime for DNN training. We modified FlexFlow to represent models with PCGs, added support for Unity's additional forms of parallelism, and replaced FlexFlow's randomized search with the algorithm described in Section 5. The substitution generator (Section 4) is implemented on top of TASO [25], and extends its fingerprint function to consider parallelization. We also add 33 parallelization-specific properties that are used by the substitution verifier as axioms capturing the semantics of the parallelization operators.

All experiments were performed on the Summit supercomputer [2, 56]. Each compute node is equipped with two IBM POWER9 CPUs, 512 GB main memory, and six NVIDIA Volta V100 GPUs. Three of the GPUs within a node are connected to the same CPU and interconnected via NVLink. Nodes are connected with Mellanox EDR 100Gb InfiniBand.

**DNNs.** Table 1 summarizes the seven DNN models used in our evaluation. ResNeXt-50 [60] and Inception-v3 are commonly used DNNs for image classification. BERT [14] is a language model with state-of-the-art accuracy on a spectrum of language tasks. DLRM [41] and XDL [28] are deep learning recommendation models for personalization and ads recommendation. CANDLE-Uno [3] is a DNN architecture for precision medicine. Multi-layer perceptron [17] (MLP) is a widely used architecture for a variety of regression tasks and a core component in many DNNs.

We follow prior work in setting hyperparameters for training (e.g., batch sizes, learning rates) [3, 14, 38, 41, 60]. We report per-GPU minibatch size $B$: for runs with $n$ GPUs, the global minibatch size is $n \times B$. The global minibatch sizes are consistent with those reported in the literature. We use a per-GPU minibatch size of 64 for ResNeXt-50 and Inception-v3, 4 for BERT-Large, 1024 for DLRM and XDL, and 256 for CANDLE-Uno and MLP. The MLP model includes 16 dense layers, each of which has a hidden dimension of 8192. We use Adam [29] with a learning rate of 0.0001 for BERT-Large, and SGD [18] with a learning rate of 0.01 for the other DNNs.

Unless stated, pipeline parallelism is disabled when comparing against frameworks that do not support this feature. We evaluate the impact of pipeline parallelism in Figure 15a.

**Search Time.** For all DNNs except Inception-v3, Unity's search times are under 10 minutes even for the largest GPU count (i.e., 192). Even for Inception-v3, the most complex architecture in our evaluation with 323 operators, search terminates within 20 minutes. These times are negligible compared to the hours or days needed to train these DNNs.

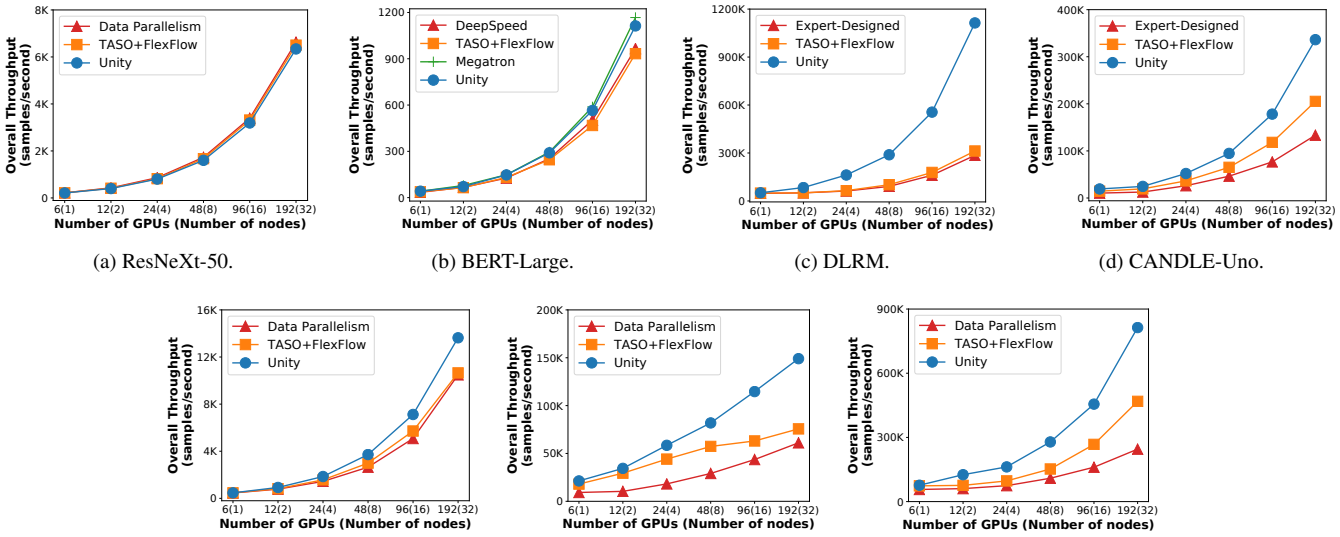
## 6.2 End-to-end Evaluation

We compare the end-to-end training performance of Unity and existing frameworks such as Megatron [50] and DeepSpeed [43]. We also compare against using TASO [25] and FlexFlow [27] to perform sequential optimization (i.e., TASO first and FlexFlow second). Since Megatron [50] and DeepSpeed [43] require the user to manually optimize each model, these baselines are only present for a subset of the models, while the automated approaches of FlexFlow and Unity can be used across all seven. Figure 14 shows the results.

BERT-Large has been highly optimized by existing frameworks such as Megatron and DeepSpeed which use expert-designed strategies combining multiple forms of parallelism. As such, Unity is not expected to outperform these strategies. Instead, the primary purpose of this evaluation is to determine if Unity can re-discover these hand-tuned strategies within a few minutes of automated search. Note that since Megatron and DeepSpeed require users to manually specify all parallelism degrees for data, tensor-model, and pipeline parallelism, we explore different combinations of the supported parallelism degrees and report the best performance.

Unity achieves on-par performance with Megatron and outperforms both DeepSpeed and FlexFlow. We find that the best strategy discovered by Unity is almost the same as the expert-designed strategy in Megatron: the only difference is that for some matrix multiplications Megatron uses reduction parallelism while Unity uses data parallelism, which has a negligible impact on overall training performance. This shows that even on highly-optimized models Unity is able to automatically generate parallelization optimizations that match those manually designed by domain experts. Megatron is customized for Transformer-based language models and does not support the other DNNs in our evaluation. The fact that the parallelization strategy discovered by Unity matches the expert-designed strategy in Megatron is, in our view, a positive outcome of Unity.

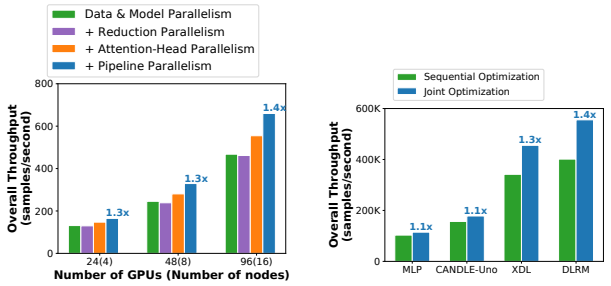
DLRM and CANDLE-Uno both exceed the memory capacity of a single GPU, preventing data parallel training. For both models we use the expert-designed strategy proposed in [38] as a baseline, which parallelizes communication-intensive operators (e.g., embedding tables) in model parallelism and compute-intensive operators (e.g., matrix multiplications) in data parallelism. Unity outperforms both expert-designed strategies and TASO+FlexFlow by up to 3.6× on DLRM and 1.6× on CANDLE-Uno. For all other models, we compare Unity against data parallelism and TASO+FlexFlow.




(a) ResNeXt-50. (b) BERT-Large. (c) DLRM. (d) CANDLE-Uno.

(e) Inception-v3. (f) MLP. (g) XDL.

Figure 14: Training throughput comparison among existing frameworks and Unity. The experiments were performed on the Summit supercomputer [2] with 6 GPUs per node. All numbers were measured by averaging 1,000 training iterations.




(a) (b)

Figure 15: (a) End to end performance of BERT-Large integrating different parallelization dimensions. Speedups relative to data+model parallelism. (b) Speedups solely attributable to joint vs sequential optimization on 96 V100 GPUs (16 nodes). Search space and algorithm are fixed to remove effects from Unity's larger search space and improved search scalability.

Unity outperforms the best existing approaches by 1.0× on ResNeXt-50, 1.3× on Inception-v3, 2.0× on MLP, and 1.9× on XDL. The lack of improvement on ResNeXt-50 is expected as the model's optimal strategy (data parallelism) is already the default used by most frameworks.

We observe that the performance improvement is achieved by (1) supporting operator-specific parallelism and (2) jointly optimizing algebraic transformations and parallelization. We further analyze these details in the following experiments.

## 6.3 Parallelism Dimensions

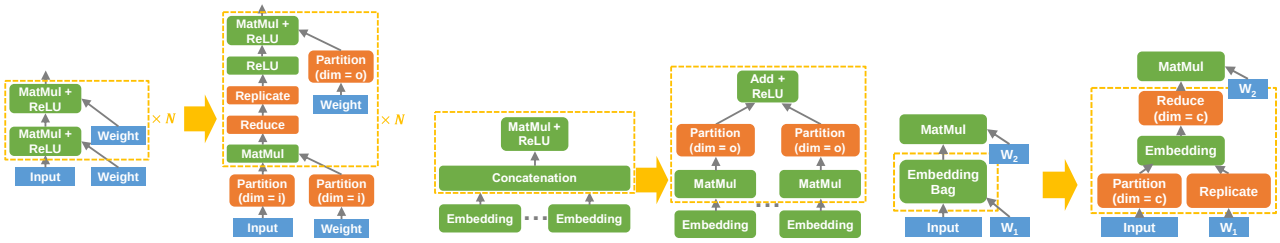
To evaluate how different parallelism dimensions improve training performance, we perform an ablation study of Unity on BERT-Large by iteratively adding new dimensions to Unity and measuring the training throughput. Figure 15a shows the results. Compared to data and model parallelism, adding reduction parallelism does not improve training performance, but combining reduction and attention-head parallelism increases performance by up to 1.2× because optimizing the attention operators in BERT-Large requires both reduction and attention-head parallelism, as shown in Figure 10. Enabling pipeline parallelism achieves an overall speedup of 1.4×. This result shows that hybrid strategies and operator-specific dimensions are critical for DNN training performance.

## 6.4 Joint Optimization

To evaluate Unity's joint optimization, we compare against sequential optimization of algebraic transformations and parallelization. Results are shown in Figure 15b. Unlike the TASO+FlexFlow baseline in Figure 14, in Figure 15b we include Unity's additional parallelism dimensions and improved scalability to isolate the effects of joint optimization. As a result, the performance improvement (up to 1.4× speedup) comes solely from the ability to optimize jointly rather than sequentially. We study three examples in detail.

The first (Figure 16a) is a slight generalization of the example introduced in Figure 2. By not fusing the first `MatMul` and `ReLU`, which would be done in sequential optimization as the algebraic optimizer would ignore parallelism, Unity is able to significantly reduce the amount of communication by using reduction parallelism and a more efficient AllReduce (represented by the `Reduce` followed by `Replicate`).

The second is shown in Figure 16b. `Concatenation` is the main performance bottleneck in DLRM and XDL, since it can-




(a) Multi-layer perceptron. (b) `Concatenation`. (c) `EmbeddingBag`.

Figure 16: Example joint optimizations of computation graph and parallelization discovered by Unity. For `Partition`, *i* and *o* indicate the input and output channel dimensions of a matrix multiplication.

Table 2: Search algorithm ablation study. "Scaled" numbers are relative to the 2 GPU time with all optimizations enabled.

| | All | | w/o Split | | w/o Cache+Split | |
|---|---|---|---|---|---|---|
| | Time | Scaled | Time | Scaled | Time | Scaled |
| 6 GPUs (1 nodes) | 57s | 1× | 4m 01s | 4.3× | 37m 01s | 38.5× |
| 12 GPUs (2 nodes) | 1m 47s | 1.9× | 11m 15s | 16.8× | > 1h | n/a |
| 24 GPUs (4 nodes) | 3m 00s | 3.1× | > 1h | n/a | > 1h | n/a |
| 48 GPUs (8 nodes) | 5m 55s | 6.1× | > 1h | n/a | > 1h | n/a |

not be parallelized in the same dimension as the `Embedding` operators and requires an all-to-all synchronization. The optimization eliminates the `Concatenation` by replacing the subsequent `MatMul` with independent `MatMul`s executed using the same model parallel strategy as the `Embedding` operators, which reduces communication costs as the `Embedding` operators' outputs are only used locally.

The third optimization is shown in Figure 16c. An `EmbeddingBag` [15] operator computes the sum of a bag of embeddings for each training sample. Unity discovers a joint optimization that transforms an `EmbeddingBag` to a normal `Embedding` to enable additional parallelization opportunities.

## 6.5 Search Algorithm

To evaluate the impact of the three search optimizations (graph splitting, cross-invocation cache, and dynamic programming) presented in Section 5, we perform an ablation study of the search time for ResNeXt-50. With all three techniques enabled (the "All" column), we see roughly linear scaling as we move from 6 to 48 GPUs. This, along with Figure 14, demonstrates that Unity's search algorithm scales to nontrivial node counts.

Disabling graph splitting increases search times by 4.3-8.8× and causes them to scale nonlinearly, while disabling the cross-invocation cache adds an additional 8.9×. Disabling the dynamic programming algorithm causes even the smallest cases to time out. These results indicate that the three proposed techniques are necessary for adequate performance.

# 7 Related Work

**Manually-designed parallelization strategies.** Manually-designed parallelization strategies are used in most existing DNN frameworks to optimize distributed DNN training [5, 6, 42, 43, 49]. For example, Neo [38] optimizes DLRM by using data parallelism for compute-intensive operators and model parallelism for communication-intensive operators. Megatron-LM [50] proposes a model-specific customized strategy that combines data, reduction, and attention-head parallelism for training large language models. These strategies only work for specific DNN models and do not generalize. We use these expert-designed strategies as baselines in our evaluation and show that Unity can automatically discover strategies with improved performance.

**Automated DNN parallelization.** Recent work has proposed automated approaches to optimizing distributed DNN training. For example, ColocRL [36, 37] and Placeto [7] use reinforcement learning to find efficient device placement for model parallelism. Baechi [22] achieves fast device placement for model parallelism using two memory-constrained algorithms. FlexFlow [27] uses randomized search to optimize data, model, and spatial parallelism. GSPMD [61], a generalization of GShard [34], finds parallelization strategies based on user-provided hints. PipeDream [39] uses dynamic programming to find optimized strategies combining pipeline and data parallelism. Tofu [59] uses recursive search to minimize communication time and automatically discovers parallelization dimensions via interval analysis. Tarnawski et al. [52, 53] propose a two-level dynamic programming algorithm to partition a DNN computation graph across devices by combining data, pipeline, and tensor model parallelism. Alpa [65] automates inter-operator (i.e., pipeline) parallelism using dynamic programming and intra-operator (i.e., data and tensor model) parallelism using integer linear programming. Whale [23] uses computation-balanced partitioning to accomodate heterogenous compute devices and allows specifying parallelization strategies through small parallelization primitives. TensorOpt [8] introduces the *cost frontier* to simultaneously reason about multiple objectives (e.g., execution time and cloud resource cost) in automatic parallelization. Finally,

AutoSync [63] learns to optimize synchronization strategies for data-parallel training from a few thousand samples. However, existing approaches (except Tofu) only support limited parallelism dimensions and none jointly optimizes algebraic transformations and parallelization. Unity supports all existing parallelism dimensions, is extensible to new operators and forms of parallelism, and jointly optimizes algebraic transformations and parallelization.

**Automated algebraic transformations.** TASO [25] automatically discovers algebraic transformations for DNNs but does not support parallelization. Unity adopts the superoptimization idea from TASO to generate and verify PCG substitutions. However, unlike the algebraic transformation task considered by TASO, Unity deals with a significantly larger search space and considers additional tasks, such as device assignments. We observe that TASO's search algorithm alone is incapable of exploring the larger search space. To address this challenge, Unity introduces three novel elements of the search technique: the dynamic programming algorithm for finding optimized machine mappings, the subgraph cache for exploiting the locality of graph substitutions, and the additional parallelism-compatible divide-and-conquer approach to enabling scalability to complex models.

**Automated DNN code generation.** Recent work has proposed approaches for generating hardware-specific code for DNN operators. TVM [10, 11] uses a learning-based algorithm to generate optimized code for a diverse set of hardware backends. Ansor [64] extends TVM by utilizing a hierarchical search algorithm to explore a much larger search space of program candidates. Unity optimizes DNN computation at a higher level than these approaches. Therefore, Unity's optimizations are orthogonal and can be combined with existing code generation techniques. We leave integrating code generation into Unity as future work.

**Intermediate representations for DNN parallelization.** TensorFlow [**?**], MLIR [31, 32], Relay [45], and ONNX [33] represent DNN computation with graph-based intermediate representations (IRs). Distributed training of a model is represented by annotating each operator with a parallelization strategy describing how the operator is parallelized across devices. These approaches represent algebraic transformations and parallelization separately and optimize them sequentially, missing joint optimizations. pONNX [57], automap [48], and DistIR [47] propose IRs that express both computation and communication, but are too low-level to be used for Unity-style joint optimization (see Section 3.4 for details). Unity uses a higher-level representation better suited to optimization, the PCG, and represents both parallelization and algebraic transformations as graph substitutions on PCGs.

## 8 Limitations and Future Work

To scale to large DNNs and machines, Unity's search algorithm exploits the sequential and parallel structure of modern DNNs (see Section 5.2). However, there exist DNN architectures (e.g., NASNet [66]) that violate this structure. Extending Unity to include these DNNs would improve generality, but potentially at the cost of decreased scalability.

While Unity successfully optimizes two of the most prominent classes of optimizations (i.e., algebraic transformations and parallelization), there are a variety of additional optimizations currently not considered, such as tensor offloading and rematerialization [21, 30, 44]. The PCG can be extended to represent these optimizations, but the search algorithm as presented in Section 5 does not reason about memory usage and therefore may generate parallelization strategies that violate memory constraints. While these invalid strategies can be made valid by applying the necessary tensor offloading and rematerialization afterward, not including these optimizations in Unity's joint search potentially leads to suboptimal performance. Thus, integrating memory optimizations into Unity's search algorithm is a promising area for future research.

Another limitation of Unity is its support for pipeline parallelism. While PCGs are capable of representing parallelization strategies that interleave pipeline-parallel and non-pipeline-parallel operators in a PCG, our search algorithm excludes these cases to reduce the search space. In addition, Unity's search algorithm does not consider non-sequential pipeline parallelism strategies, where a stage can have multiple predecessor/successor stages.

## 9 Conclusion

This paper presents Unity, the first system that jointly optimizes algebraic transformations and parallelization in distributed DNN training. Unity represents both parallelization and algebraic transformations as substitutions on a unified graph representation, uses a novel hierarchical search algorithm to identify an optimized sequence of substitutions, and scales to large numbers of GPUs and complex DNNs.

Our evaluation with seven real-world DNN benchmarks on up to 192 GPUs show that Unity outperforms state-of-the-art parallelization approaches by up to 3.6× while keeping optimization times under 20 minutes. As nearly half of this speedup is attributable solely to the use of joint optimization over sequential optimization, Unity demonstrates that joint optimization is practical and that future systems will need to include it or else miss significant performance gains.

## Acknowledgement

We thank the anonymous reviewers for their comments, and are grateful to our shepherd Byung-Gon Chun for his feedback. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1656518, and an NSF award CNS-2147909. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

## References


[1] CANDLE Benchmarks. https://github.com/ECP-CANDLE/Benchmarks, 2018. 3, 11

[2] Summit supercomputer. https://www.olcf.ornl.gov/summit/, 2018. 11, 12

[3] Uno: Predicting tumor dose response across multiple data sources. https://github.com/ECP-CANDLE/Benchmarks/tree/master/Pilot1/Uno, 2018. 11

[4] Criteo 1tb click logs dataset. https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset/, 2021. 11

[5] Optimize and accelerate machine learning inferencing and training. https://www.onnxruntime.ai/, 2021. 13

[6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2016. 1, 3, 13

[7] Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. Placeto: Learning generalizable device placement algorithms for distributed machine learning. *CoRR*, abs/1906.08879, 2019. 13

[8] Zhenkun Cai, Xiao Yan, Kaihao Ma, Yidi Wu, Yuzhen Huang, James Cheng, Teng Su, and Fan Yu. TensorOpt: Exploring the Tradeoffs in Distributed DNN Training with Auto-Parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2021. 13

[9] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015. 3

[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *CoRR*, abs/1802.04799, 2018. 14

[11] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems 31*, NeurIPS'18. 2018. 14

[12] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, 2008. 7

[13] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012. 1

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. 3, 11

[15] EmbeddingBag in PyTorch. https://pytorch.org/docs/stable/generated/torch.nn.EmbeddingBag.html, 2021. 13

[16] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. Dapple: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, page 431–445, New York, NY, USA, 2021. Association for Computing Machinery. 10

[17] Matt W Gardner and SR Dorling. Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences. *Atmospheric environment*, 32(14-15):2627–2636, 1998. 11

[18] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017. 11

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR, 2016. 9

[20] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism, 2018. 10

[21] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 497–511, 2020. 14

[22] Beomyeol Jeon, Linda Cai, Pallavi Srivastava, Jintao Jiang, Xiaolan Ke, Yitao Meng, Cong Xie, and Indranil Gupta. Baechi: Fast device placement of machine learning graphs. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 416–430, New York, NY, USA, 2020. Association for Computing Machinery. 13

[23] Xianyan Jia, Le Jiang, Ang Wang, Jie Zhang, Xinyuan Li, Wencong Xiao, Langshi chen, Yong Li, Zhen Zheng, Xiaoyong Liu, and Wei Lin. Whale: Scaling Deep Learning Model Training to the Trillions. *arXiv:2011.09208 [cs]*, August 2021. 2, 13

[24] Zhihao Jia, Sina Lin, Charles R. Qi, and Alex Aiken. Exploring hidden dimensions in accelerating convolutional neural networks. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*. PMLR, 2018. 1, 10

[25] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 47–62, New York, NY, USA, 2019. Association for Computing Machinery. 1, 3, 4, 7, 8, 9, 11, 14

[26] Zhihao Jia, James Thomas, Todd Warzawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. Optimizing dnn computation with relaxed graph substitutions. In *Proceedings of the 2nd Conference on Systems and Machine Learning*, SysML'19, 2019. 1, 10

[27] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In *Proceedings of the 2nd Conference on Systems and Machine Learning*, SysML'19, 2019. 1, 2, 3, 10, 11, 13

[28] Biye Jiang, Chao Deng, Huimin Yi, Zelin Hu, Guorui Zhou, Yang Zheng, Sui Huang, Xinyang Guo, Dongyue Wang, Yue Song, Liqin Zhao, Zhi Wang, Peng Sun, Yu Zhang, Di Zhang, Jinhui Li, Jian Xu, Xiaoqiang Zhu, and Kun Gai. Xdl: An industrial deep learning framework for high-dimensional sparse data. In *Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data*, DLP-KDD '19, New York, NY, USA, 2019. Association for Computing Machinery. 11

[29] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 11

[30] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. Dynamic tensor rematerialization. *CoRR*, abs/2006.09616, 2020. 14

[31] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021. 14

[32] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A compiler infrastructure for the end of moore's law. *CoRR*, abs/2002.11054, 2020. 14

[33] Tung D. Le, Gheorghe-Teodor Bercea, Tong Chen, Alexandre E. Eichenberger, Haruki Imai, Tian Jin, Kiyokuni Kawachiya, Yasushi Negishi, and Kevin O'Brien. Compiling ONNX neural network models using MLIR. *CoRR*, abs/2008.08272, 2020. 14

[34] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *CoRR*, abs/2006.16668, 2020. 13

[35] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models, 2016. 11

[36] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V. Le, and Jeff Dean. A hierarchical model for device placement. In *International Conference on Learning Representations*, 2018. 13

[37] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. 2017. 13

[38] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie Amy Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong

Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, KR Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. Software-hardware co-design for fast and scalable training of deep learning recommendation models, 2021. 11, 13

[39] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 1–15, New York, NY, USA, 2019. Association for Computing Machinery. 1, 4, 10, 13

[40] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel dnn training, 2020. 10

[41] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019. 3, 11

[42] Tensors and Dynamic neural networks in Python with strong GPU acceleration. https://pytorch.org, 2017. 3, 13

[43] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimization towards training A trillion parameter models. *CoRR*, abs/1910.02054, 2019. 11, 13

[44] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. Zero-offload: Democratizing billion-scale model training, 2021. 14

[45] Jared Roesch, Steven Lyubomirsky, Marisa Kirisame, Josh Pollock, Logan Weber, Ziheng Jiang, Tianqi Chen, Thierry Moreau, and Zachary Tatlock. Relay: A high-level IR for deep learning. *CoRR*, abs/1904.08368, 2019. 14

[46] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 2015. 11

[47] Keshav Santhanam, Siddharth Krishna, Ryota Tomioka, Tim Harris, and Matei Zaharia. DistIR: An Intermediate Representation and Simulator for Efficient Neural Network Distribution. *arXiv:2111.05426 [cs]*, November 2021. 7, 10, 14

[48] Michael Schaarschmidt, Dominik Grewe, Dimitrios Vytiniotis, Adam Paszke, Georg Stefan Schmid, Tamara Norman, James Molloy, Jonathan Godwin, Norman Alexander Rink, Vinod Nair, and Dan Belov. Automap: Towards Ergonomic Automated Parallelism for ML Models. *arXiv:2112.02958 [cs]*, December 2021. 2, 7, 14

[49] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. Mesh-TensorFlow: Deep Learning for Supercomputers. *arXiv:1811.02084 [cs, stat]*, November 2018. 13

[50] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019. 1, 7, 8, 11, 13

[51] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016. 11

[52] Jakub Tarnawski, Amar Phanishayee, Nikhil R. Devanur, Divya Mahajan, and Fanny Nina Paravecino. Efficient algorithms for device placement of dnn graph operators, 2020. 13

[53] Jakub M Tarnawski, Deepak Narayanan, and Amar Phanishayee. Piper: Multidimensional planner for dnn parallelization. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 24829–24840. Curran Associates, Inc., 2021. 10, 13

[54] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. 4

[55] Sudharshan S Vazhkudai, Bronis R de Supinski, Arthur S Bland, Al Geist, James Sexton, Jim Kahle,

Christopher J Zimmer, Scott Atchley, Sarp Oral, Don E Maxwell, et al. The design, deployment, and evaluation of the coral pre-exascale systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC. IEEE, 2018. 5

[56] Sudharshan S. Vazhkudai, Bronis R. de Supinski, Arthur S. Bland, Al Geist, James Sexton, Jim Kahle, Christopher J. Zimmer, Scott Atchley, Sarp Oral, Don E. Maxwell, Veronica G. Vergara Larrea, Adam Bertsch, Robin Goldstone, Wayne Joubert, Chris Chambreau, David Appelhans, Robert Blackmore, Ben Casses, George Chochia, Gene Davison, Matthew A. Ezell, Tom Gooding, Elsa Gonsiorowski, Leopold Grinberg, Bill Hanson, Bill Hartner, Ian Karlin, Matthew L. Leininger, Dustin Leverman, Chris Marroquin, Adam Moody, Martin Ohmacht, Ramesh Pankajakshan, Fernando Pizzano, James H. Rogers, Bryan Rosenburg, Drew Schmidt, Mallikarjun Shankar, Feiyi Wang, Py Watson, Bob Walkup, Lance D. Weems, and Junqi Yin. The design, deployment, and evaluation of the coral pre-exascale systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18. IEEE Press, 2018. 11

[57] Fei Wang, Guoyang Chen, Weifeng Zhang, and Tiark Rompf. Parallel Training via Computation Graph Transformation. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 3430–3439, December 2019. 3, 7, 14

[58] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. PET: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 37–54. USENIX Association, July 2021. 1

[59] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting Very Large Models using Automatic Dataflow Graph Partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 1–17, New York, NY, USA, March 2019. Association for Computing Machinery. 1, 2, 3, 13

[60] Saining Xie, Ross B. Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. *CoRR*, abs/1611.05431, 2016. 11

[61] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, Ruoming Pang, Noam Shazeer, Shibo Wang, Tao Wang, Yonghui Wu, and Zhifeng Chen. GSPMD: General and Scalable Parallelization for ML Computation Graphs. *arXiv:2105.04663 [cs]*, May 2021. 13

[62] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. Equality Saturation for Tensor Graph Superoptimization. *Proceedings of Machine Learning and Systems*, 3:255–268, March 2021. 2

[63] Hao Zhang, Yuan Li, Zhijie Deng, Xiaodan Liang, Lawrence Carin, and Eric Xing. Autosync: Learning to synchronize for data-parallel distributed deep learning. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 906–917. Curran Associates, Inc., 2020. 14

[64] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor : Generating high-performance tensor programs for deep learning. *CoRR*, abs/2006.06762, 2020. 14

[65] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. *CoRR*, abs/2201.12023, 2022. 10, 13

[66] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018. 14