PaSE: Parallelization Strategies for Efficient DNN Training

Venmugil Elango
Baidu Research, Sunnyvale, USA
Email: elango.4@buckeyemail.osu.edu

***Abstract*—Training a deep neural network (DNN) requires substantial computational and memory requirements. It is common to use multiple devices to train a DNN to reduce the overall training time. There are several choices to parallelize each layer in a DNN. Exhaustively searching this list to find an optimal parallelization strategy is prohibitively time consuming and impractical. The standard practice is to use data parallelism because of its simplicity. However, data parallelism is often sub-optimal, and suffers from poor performance and high memory requirement. Expert-designed strategies have been proposed on a case-by-case basis using domain specific knowledge. These expert-designed strategies do not generalize well to DNNs other than the ones for which they were designed, and are not always necessarily the best choice.**

In this paper, we propose an approach to automatically find efficient parallelization strategies for DNNs from their computation graphs. We present an efficient algorithm to compute these strategies within a reasonable time in practice. We evaluate the effectiveness of our approach on various DNNs. We also compare the performance of the strategies identified by our approach against data parallelism, expert-designed strategies, and the state-of-the-art approaches. Our results show that the strategies found using our approach outperform the baseline data parallelism strategy in all the cases. In addition, our strategies achieve better performance than the expert-designed strategies and the state-of-the-art approaches.

***Keywords*-Machine learning, neural nets, parallelism, automatic parallelization, dynamic programming, optimization.**

I. Introduction

Deep neural networks are becoming increasingly sophisticated, and use larger and larger datasets for better accuracies. This has led to an increase in computational and memory requirements to train DNNs. It typically takes from several hours to days, and multiple GPUs to train a network. For instance, as noted in [1], Google's neural machine translation (GNMT) model takes around 6 days to train on WMT EN$\rightarrow$FR dataset with 96 NVIDIA K80 GPUs. Training a DNN involves three phases: *forward propagation*, *backward propagation* (or *backprop*), and *update* phase. First, the input dataset is split into multiple *mini-batches*. During a *step*, a mini-batch is passed through the layers of the network during forward propagation. At the end of the forward phase, the output is compared against the *ground truth*, and a *loss* is computed using an appropriate loss function. To minimize the loss, its *gradients* w.r.t. the model parameters are computed during backprop. Finally, the model parameters are updated during the update phase using the computed gradients. This process is repeated over several timesteps, called *epochs*, until the required accuracy is achieved.

DNN parallelization strategies can be broadly classified into three, namely, *data parallelism*, *parameter parallelism*, and *pipeline parallelism*. A strategy that combines these three approaches to parallelize each layer differently is often referred to as *hybrid parallelism*. Each has its own advantages and disadvantages, as described below.

In *data parallelism*, each of the p devices keeps a replica of the entire DNN, and a mini-batch is split into p shards and distributed to different devices. Each device performs forward and backward propagation independently on its shard of data. During the update phase, gradients from all the devices are accumulated, typically through an *all-reduce* operation. For large models, this communication becomes a major bottleneck. Further, as the model parameters are replicated (instead of being distributed), it might be impossible to train large models by just using data parallelism, due to memory constraints. Additionally, data parallelism is inefficient at small mini-batch sizes. Unfortunately, using a larger mini-batch size may not always be possible, owing to poor convergence and accuracy [2]. Despite these drawbacks, data parallelism remains popular due to its simplicity.

An alternative strategy is to divide the work along model parameter and attribute dimensions (e.g., image height/width, channels, filters, etc.), rather than the mini-batch dimension. This is the approach taken by *parameter parallelism*[1] strategy [3]. With this approach, the model parameters/attributes are distributed among different devices, and each device only computes a part of a layer's activations (and gradients) during forward (and backward) propagation. This strategy typically incurs *all-to-all* communication to accumulate the activations and gradients. Depending on the mini-batch and model parameter sizes, one strategy is more efficient than the other.

The third approach (*pipeline parallelism*) [8] is to place different layers of a network on different devices, without splitting the input data or model parameters along any dimension. Each device computes activations (and gradients) for the layers it owns, and sends the results to the devices that own the successive layers. This strategy has the advantage of not

[1]Some previous works [3]–[5] refer to this strategy as model parallelism, while others [1], [6] use the term model parallelism to refer to a different strategy. To avoid any confusion, we instead use the term parameter parallelism here. In terms of SOAP [7], parameter parallelism captures both attribute (A) and parameter (P) dimensions. Refer to Fig. 1 for an illustration.

©2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. DOI: 10.1109/IPDPS49936.2021.00111

needing to collectively communicate the model parameters, however, there needs to be sufficient inter-layer parallelism and the data needs to arrive at a specific rate through the pipeline for this strategy to be efficient.

A parallelization strategy that combines multiple strategies to parallelize different layers differently is typically referred to as *hybrid parallelism* [7]. In hybrid parallelism, each layer is parallelized differently using a mix of different strategies (e.g., data+parameter parallelism). There are several possibilities to choose how different layers need to be parallelized. Hence, it is impractical to exhaustively search for an optimal strategy. Based on domain specific knowledge, expert designed strategies [1], [5] have been proposed on a case-by-case basis for different DNNs. There also have been efforts in the past to automatically find good strategies. These works either (i) apply different heuristics [7], [9] to find a greedy solution, (ii) find an optimal solution restricted to a certain class of DNNs [10] (such as CNNs), or (iii) reduce the search space by restricting some choices to find an optimal strategy within the reduced search space [6], [9], [10]. In this paper, we take this third approach. We ignore *inter-layer* pipeline parallelism, and restrict ourselves to finding the best strategy to parallelize different layers of a DNN using a combination of parameter and data parallelism. In Section IV, we empirically show that our method works well in practice despite this restriction as it does not extensively prune the optimal strategies from the search space. We also compare our results against the state-of-the-art approach FlexFlow [7]. We formally define the problem in Section II, and provide a method to find efficient strategies in Section III. In Section VI, we summarize the previous works and discuss the differences between our approach and theirs. To summarize our contributions,

- We propose a formulation, and a vertex ordering strategy to enable efficient computation of the best parallelization strategies for DNNs.
- We develop an efficient algorithm based on our formulation to compute the best strategies for various DNNs. A prototype implementation of our approach is available at https://github.com/baidu-research/PaSE. Experimental results show that our algorithm finds efficient strategies within a few seconds for various DNNs.
- We evaluate the strategies found by our approach against data parallelism, expert-designed strategies, and the strategies proposed by the state-of-the-art framework FlexFlow [7]. Results show that our strategies outperform data parallelism by up to $1.85\times$ on a multi-node/multi-GPU system consisting of 1080Ti GPUs, and by up to $4\times$ on a system consisting of 2080Ti GPUs for various benchmarks. Our strategies also perform better than expert-designed strategies, and the strategies suggested by FlexFlow.

II. Problem representation

A DNN can be represented as a *computation graph*. A computation graph $G = (V, E)$ is a weakly connected directed graph, where each node $v \in V$ corresponds to a layer (e.g., fully-connected, convolution, etc.) in the DNN, and each edge $(u, v) \in E$ represents flow of a tensor that is an output of u and an input of v. Each node $v \in V$ has an associated *iteration space* [11] that captures the computation of v. Consider, for instance, a fully-connected layer that multiplies a matrix $A_{M\times K}$ with a matrix $B_{K\times N}$. Its iteration space is specified by the set $\{(i, j, k) \in \mathbb{Z}^3 \mid 0 \le i < M \wedge 0 \le j < N \wedge 0 \le k < K\}$.

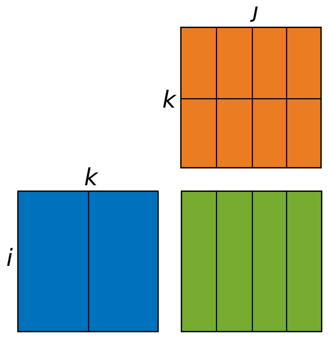


Fig. 1. Iteration space of a GEMM computation parallelized using the configuration $(1, 4, 2)$. j and k dimensions are split 4-ways and 2-ways, respectively, while the i dimension is not parallelized.

A *parallelization configuration* [10] C_v of a node v is a d-tuple of positive integers that defines how the iteration space of v is split and parallelized across different devices, where d is the dimension of the iteration space of v. For the fully-connected layer example above, a configuration $(1, 4, 2)$ states that the iteration space has to be split into 4 equal parts along the second dimension of its iteration space, and into 2 parts along the third dimension. (Refer Fig. 1.) Computationally, this configuration states: split columns of A and rows of B into two equal parts; split columns of B into four equal parts; perform the 8 GEMM computations that correspond to each part on 8 separate devices; and finally perform partial reduction of the intermediate results. Refer [3, Section 6] for a more concrete description of how different DNN layer types can be parallelized, and the communications they incur. Given a node v with d dimensional iteration space, and p devices, the set of valid configurations for v is given by $\mathcal{C}(v) = \{(c_1, \ldots, c_d) \in \mathbb{N}^d \mid \prod_{i=1}^{d} c_i \le p\}$.

A *parallelization strategy* ϕ is the set $\{(v, C_v) \mid v \in V \wedge C_v \in \mathcal{C}(v)\}$ that specifies a valid configuration for each node $v \in V$. Configuration for a node v in ϕ is given by $C_v = \phi(v)$. A *substrategy* $\phi_{|U}$ is a strategy restricted to the subset U, i.e., $\phi_{|U} = \{(u, \phi(u)) \mid u \in U\}$. An *optimal strategy* $\widehat{\phi}$ has the minimum cost over all possible strategies for V, under a given cost function $\mathcal{F}$, i.e., $\widehat{\phi} = \arg\min_{\phi\in\Phi} \mathcal{F}(G, \phi)$, where Φ is the set of all valid strategies for V. Given p devices with an average peak floating-point performance of F FLOPS per device, and an average communication bandwidth of B bytes per second per link, the cost function we use is:

$$\mathcal{F}(G, \phi) = \sum_{v\in V} t_l(v, \phi, r) + \sum_{(u,v)\in E} r \times t_x(u, v, \phi) \qquad (1)$$

where, $r = F/B$ is the FLOP-to-bytes ratio; layer cost $t_l(v, \phi, r)$ is the cost (in FLOP) of executing the layer v (such as fully-connected) using the parallelization configuration $\phi(v)$ – this cost includes both computation and communication that happens internally within a layer (such as all-reduce within a layer, halo communication for convolutions, etc., normalized to FLOP by multiplying it with r); and data transfer cost $t_x(u, v, \phi)$ is the communication cost (in bytes) needed to communicate the tensor that flows along the edge (u, v) or (v, u) during forward and backward propagation, where u and v are parallelized using configurations $\phi(u)$ and $\phi(v)$, respectively.[2]

Our cost function $\mathcal{F}$ is an approximation of the actual cost. It ignores any overlapping (or pipelining) of different layers by adding the costs $t_l(v_x, \cdot, \cdot)$ and $t_l(v_y, \cdot, \cdot)$ of any two layers (instead of taking a max where possible). Thus, it captures data and parameter parallelism, but ignores *inter-layer* pipeline parallelism. Note that this only ignores pipeline parallelism between layers, while any *intra-layer* pipeline parallelism opportunities within a layer can be accurately captured by accounting for it in the layer cost t_l. In return, this approximation allows us to devise a technique (described in Section III) to efficiently find the best strategies quickly in practice. Our approach is effective despite this simplification as most DNNs do not contain significant inherent pipeline parallelism opportunities due to data dependence constraints. Even though our approach finds the optimal solution $\widehat{\phi} = \arg\min_{\phi \in \Phi} \mathcal{F}(G, \phi)$ under the cost function $\mathcal{F}$, since $\mathcal{F}$ itself is an approximation, rather than referring to our solution as the *optimal* strategy, we refer to it as an *efficient* strategy or the *best* strategy, to avoid any confusion. Some previous works [6] have used *inter-batch pipeline parallelism* to improve parallel training throughput by making semantic modifications to the model. In this work, we do not perform any such semantic modifications to the model. Thus, the convergence rates and the final accuracies of the strategies proposed by our method are exactly same as the original model.

As we only have a handful of different types of DNN layers, we use analytically derived layer costs t_l (parametrized for problem sizes) for different types of layers. Communication cost t_x along an edge (u, v) is given by: $\max_d |A(v, d, \phi)| - |A(v, d, \phi) \cap A(u, d, \phi)|$, where, $A(v, d, \phi)$ and $A(u, d, \phi)$ are the volume of input tensor of v (parallelized using configuration $\phi(v)$) needed by a device d, and volume of output tensor of u (parallelized using $\phi(u)$) held by d, respectively. We ignore many low level details such as cache effects, etc., to keep the cost function simple, although such simplifications are not necessary for our method to work. Additionally, to select the best strategy, our approach only requires various strategies to be ranked in the correct order. For instance, if a strategy ϕ_1 is better than ϕ_2, the analytical cost of ϕ_1 computed by our cost function ($\mathcal{F}$) has to be lower than ϕ_2. However, precisely predicting their absolute runtime costs is not necessary for our method to work accurately. Our simplifying assumptions affect costs of all the strategies more or less alike, preserving most of the relative ordering.

Although a parallelization configuration only describes how to split an iteration space into multiple parts, while the actual device assignment for each part is not explicitly specified, our experience shows that once we have a complete parallelization strategy, a simple greedy assignment that maximizes data locality (i.e., a greedy assignment that maximizes $|A(v, d, \phi) \cap A(u, d, \phi)|$) works sufficiently well in practice. Additionally, frameworks such as GShard [12] can take user-specified parallelization strategies, such as the ones computed by our approach, and automatically perform efficient device assignment by simply aligning the sharding decisions of adjacent layers.

Even though our objective primarily focuses on minimizing the training time, this also indirectly minimizes the space requirements, since the memory footprint per device is a sum of (i) the space needed to hold the input and output tensors, and (ii) the space for the communication buffers. When a DNN layer is distributed among d devices, the space required for (i) often reduces by a factor proportional to d uniformly for all parallelization strategies (there are a few uncommon cases where this may not strictly hold), and the space required by (ii) is proportional to the amount of communication, which our objective indeed tries to minimize.

III. Computing Efficient Strategies

This section describes our approach to compute the best strategies for DNNs efficiently.

Notation: Given $G = (V, E)$ and a vertex $v \in V$, we let $N(v)$ denote its neighbors, i.e., $N(v) = \{u \in V \mid (u, v) \in E \vee (v, u) \in E\}$. Given some $U \subseteq V$, we also use the notation $N(U) = \bigcup_{u \in U} N(u)$ to refer to the neighbors of U. Let $\mathcal{V} = (v^{(1)}, \ldots, v^{(|V|)})$ be an (arbitrary) ordering of V. Then $v^{(i)}$ refers to the ith vertex in the sequence $\mathcal{V}$. We also use $\mathcal{V}_{\leq i}$, $\mathcal{V}_{\geq i}$, $\mathcal{V}_{<i}$, and $\mathcal{V}_{>i}$ to refer to vertex sets $\{v^{(1)}, \ldots, v^{(i)}\}$, $\{v^{(i)}, \ldots, v^{(|V|)}\}$, $\{v^{(1)}, \ldots, v^{(i-1)}\}$ and $\{v^{(i+1)}, \ldots, v^{(|V|)}\}$, respectively.

A. A naïve approach

A brute-force method to find an efficient strategy for a computation graph is to enumerate all possible combinations of configurations of the vertices, and choose the one with the least cost. Combinatorial nature of this method makes it impractical to use even on small graphs such as AlexNet. However, a straight-forward observation shows that the parallelization configuration chosen for a layer only affects the cost of computing the layer itself, and its neighbors. This is also evident from Equation (1), where, changing the configuration for a vertex v from C to C' only affects the layer cost $t_l(v, \cdot, \cdot)$ of the vertex itself, and its data transfer costs with its neighbors $t_x(u, v, \cdot)$, where $u \in N(v)$. One way to exploit this property is to sequence V in a *breadth-first* traversal order $\mathcal{V} = (v^{(1)}, \ldots, v^{(|V|)})$, and find the best strategy for G as follows: For an ith vertex $v^{(i)}$ in $\mathcal{V}$, we define its *dependent set* $D_B(i)$ as the set of neighbors of $\mathcal{V}_{\leq i}$ that are in $\mathcal{V}_{>i}$, i.e., $D_B(i) = N(\mathcal{V}_{\leq i}) \cap \mathcal{V}_{>i}$. Let $\phi \in \Phi_{|D_B(i)}$ be any valid

[2]Note that t_x captures communication cost along both directions (forward and backward), and is edge-direction agnostic, i.e., for an edge $(u, v) \in E$, $t_x(u, v, \phi) = t_x(v, u, \phi)$.

substrategy for $D_B(i)$. Then, for $1 \le i \le |V|$, a configuration C for $v^{(i)}$ that minimizes the overall parallel training cost is given by the following recurrence:

$$\begin{aligned} \mathcal{B}_{\mathcal{V}}(i,\phi) &= \min_{C \in \mathcal{C}(v^{(i)})} \mathcal{H}_{\mathcal{V}}(i,\phi') + \mathcal{B}_{\mathcal{V}}(i-1,\phi'') \\ \mathcal{B}_{\mathcal{V}}(0,\phi) &= 0 \end{aligned} \quad (2)$$

where, $\phi' = \phi \cup \{(v^{(i)}, C)\}$, $\phi'' = \phi'_{|D_B(i-1)}$, and

$$\mathcal{H}_{\mathcal{V}}(i,\phi) = t_l(v^{(i)},\phi,r) + \sum_{v \in N(v^{(i)}) \cap \mathcal{V}_{>i}} r \times t_x(v^{(i)},v,\phi) \quad (3)$$

The function $\mathcal{H}_{\mathcal{V}}(i,\cdot)$ captures the layer cost of $v^{(i)}$ and its data transfer cost with its neighbors that appear after $v^{(i)}$ in $\mathcal{V}$. (Data transfer costs with its remaining neighbors are captured within $\mathcal{B}_{\mathcal{V}}(i-1,\cdot)$.) An efficient parallelization strategy for G is the one that achieves the minimum cost $\mathcal{B}_{\mathcal{V}}(|V|,\varnothing)$. As the recurrence (2) has an optimal substructure, a dynamic programming (DP) based algorithm can be used to find the best strategy. However, as we show later in Table I, computing efficient strategies using this recurrence is still quite expensive, and takes significant amount of time to find the best strategies for graphs other than simple path graphs such as AlexNet. In the next subsection, we will derive a more efficient approach to find the best strategies.

B. An efficient approach

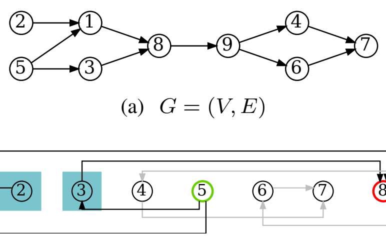
From recurrence (2), we can observe that as $\mathcal{B}_{\mathcal{V}}(i,\phi)$ needs to be computed for all possible $\phi \in \Phi_{|D_B(i)}$, the computational complexity for finding the best strategy using (2) is at least $O(K^{M+1})$, where $K = \max_{v \in V} |\mathcal{C}(v)|$ is the maximum number of configurations for any vertex in G, and $M = \max_{i \in [1,|V|]} |D_B(i)|$ is the size of the largest dependent set. It is important to note that dependent sets (and thus M) are a function of sequence $\mathcal{V}$. Thus, by carefully arranging the vertices in $\mathcal{V}$, it is possible to reduce M and thus the overall computational complexity.

If computation graphs are fully sparse, any arbitrary ordering would be adequate to keep M sufficiently small, whereas if they are fully dense, no possible ordering can help reduce M. However, a unique property of DNN graphs is that they are mostly sparse with a few high degree nodes. Thus, if an arbitrary ordering, or a simple breadth-first ordering is used, these dense locations form the bottlenecks, leading to high computational overhead in finding the best strategies. By carefully ordering the vertices, the size M can be reduced, allowing us to compute the parallelization strategies efficiently. In this subsection, we develop an algorithm that exploits this property to order the vertices in a sequence that keeps the sizes of these dependent sets to the minimum. We first introduce a few definitions and reformulate the recurrence (2) in terms of subgraphs and connected components that provides us the flexibility to efficiently compute costs from arbitrary sequences. Let $G = (V,E)$ be the computation graph for a DNN, and $\mathcal{V}$ be an ordering of V. For a vertex $v^{(i)}$:

a) Connected set: A *connected set* $X(i) \subseteq \mathcal{V}_{\le i}$ of $v^{(i)}$ is the set of vertices in $\mathcal{V}_{\le i}$ that are connected to $v^{(i)}$ through a path $(v_1 \in \mathcal{V}_{\le i}, \dots)$ that only goes through the vertices in $\mathcal{V}_{\le i}$. Note that $v^{(i)} \in X(i)$. For e.g., in Fig. 2, $X(5) = \{v^{(1)}, v^{(2)}, v^{(3)}, v^{(5)}\}$. Intuitively, any vertex $v \notin X(i)$ is irrelevant, both directly and indirectly, for finding the best configuration for $v^{(i)}$, and thus can be ignored allowing us to use a smaller dependent set as defined below.

b) Dependent set: We redefine the *dependent set* of $v^{(i)}$ as the neighbors of $X(i)$ that are in $\mathcal{V}_{>i}$, i.e., $D(i) = N(X(i)) \cap \mathcal{V}_{>i}$.

c) Connected subsets: Given a connected set $X(i)$, consider the vertex set $U = X(i) - \{v^{(i)}\}$, and its induced subgraph $G' = (U, E \cap U \times U)$. G' is composed of one or more connected components $\{(V_1,E_1),\dots\}$. We refer to the set of vertices of these connected components $\{V_1,\dots\}$ as the *connected subsets* $S(i)$ of $v^{(i)}$. In Fig. 2, $S(5) = \{\{v^{(1)}, v^{(2)}\}, \{v^{(3)}\}\}$. Intuitively, the problem of finding the best configuration for $v^{(i)}$ is broken down into smaller subproblems in terms of $S(i)$, that allows us to use dynamic programming to solve our new recurrence (4) defined below.



(a) $G = (V,E)$

(b) $\mathcal{V}$: an ordering of V from left to right. ($\mathcal{V}$ is edge direction agnostic. Edges can go either forward or backward.)

Fig. 2. A toy computation graph G, and an ordering $\mathcal{V}$ of its vertices. For the vertex $v^{(5)}$ (marked in green), its connected set $X(5) = \{v^{(1)}, v^{(2)}, v^{(3)}, v^{(5)}\}$, and its dependent set $D(5) = \{v^{(8)}\}$ (marked in red). Its connected subsets $S(i) = \{\{v^{(1)}, v^{(2)}\}, \{v^{(3)}\}\}$ are represented by blue boxes in the figure. A similar, but more elaborate, structure appears in InceptionV3 (refer Fig. 5) and Transformer models.

Let $v^{(i)}$ be the ith vertex in $\mathcal{V}$, and $\phi \in \Phi_{|D(i)}$ be a valid substrategy for the vertices in $D(i)$. Then,

$$\mathcal{R}_{\mathcal{V}}(i,\phi) = \min_{C \in \mathcal{C}(v^{(i)})} \mathcal{H}_{\mathcal{V}}(i,\phi') + \sum_{X(j) \in S(i)} \mathcal{R}_{\mathcal{V}}(j,\phi'') \quad (4)$$

where, $\phi' = \phi \cup \{(v^{(i)}, C)\}$, $\phi'' = \phi'_{|D(j)}$, and $\mathcal{H}_{\mathcal{V}}$ is defined in (3). We show in Theorem 1 below that a parallelization strategy that minimizes $\mathcal{R}_{\mathcal{V}}(|V|,\varnothing)$ corresponds to an efficient strategy to parallelize G. We provide a proof for Theorem 1 in Appendix A.

Theorem 1: Let $G = (V,E)$ be a computation graph for a DNN that is executed on p devices with average FLOP-to-bytes ratio r. Let $\mathcal{V}$ be a sequence for V, and Φ be the set of all possible strategies for G. Then,

$$\mathcal{R}_{\mathcal{V}}(|V|,\varnothing) = \min_{\phi \in \Phi} \mathcal{F}(G,\phi).$$

Note that the recurrence (4) by itself does not reduce the complexity of finding the best strategies. For instance, with a simple breadth-first ordering, for any vertex $v^{(i)}$, $X(i) = \mathcal{V}_{\leq i}$, and thus $D(i) = D_B(i)$. However, recurrence (4) provides the flexibility to efficiently compute costs if the sequence $\mathcal{V}$ has the potential. For instance, in Fig. 2, $|D_B(5)| = |\{v^{(7)}, v^{(8)}, v^{(9)}\}| = 3$, while $|D(5)| = |\{v^{(8)}\}| = 1$. Since the computational complexity of the recurrence is exponential in terms of the sizes of dependent sets, using recurrence (4) instead of (2) exponentially reduces the computation time for finding the best strategy.

We will now develop an approach to generate a sequence $\mathcal{V}$ that will maintain the sizes of dependent sets of the vertices as small as possible, thus allowing us to efficiently compute the recurrence (4). The complete algorithm GENERATESEQ is shown in Fig. 3. GENERATESEQ generates a sequence that maintains the sizes of dependent sets $D(i)$ (referred to as $v.d$ in Fig. 3) as small as possible. In Line 1, the dependent

```
Procedure GENERATESEQ ($G = (V, E)$)
1:  $\forall_{v \in V}, v.d \leftarrow N(v)$
2:  $U \leftarrow V$                                        // Unsequenced nodes
3:  $\mathcal{V} = (\perp_1, \ldots, \perp_{|V|})$
4:  for $i = 1$ to $|V|$ do
5:      $v^{(i)} \leftarrow \arg\min_{u \in U} |u.d|$        // Assign $i^{\text{th}}$ element of $\mathcal{V}$
6:      $U \leftarrow U - \{v^{(i)}\}$
7:      for all $v \in v^{(i)}.d$ do
8:          $v.d \leftarrow v.d \cup v^{(i)}.d - \{v^{(i)}\}$
9:      end for
10: end for
11: return $\mathcal{V}$
```

Fig. 3. Algorithm to generate a sequence $\mathcal{V}$ such that sizes of dependent sets are small.

set of a vertex v is initialized to its neighbors. In Line 5, at an iteration i, a node u that has the least cardinality $|u.d|$ is picked from the set of nodes U that are yet to be sequenced. This node becomes $v^{(i)}$ in $\mathcal{V}$. Once u has been added to $\mathcal{V}$, $v.d$ for all the nodes in $v^{(i)}.d$ are updated in Line 8. This makes sure that $|v.d|$ that is checked in Line 5 is correctly maintained. In Theorem 2 (proof available in Appendix B), we show that the dependent sets computed by GENERATESEQ are indeed correct. Computational complexity of GENERATESEQ is $O(|V|^2)$.

Theorem 2: Given a computation graph $G = (V, E)$, and a sequence $\mathcal{V}$ computed by GENERATESEQ in Fig. 3, for any $v^{(i)} \in V$, $v^{(i)}.d = D(i)$.

A dynamic programming (DP) based algorithm for recurrence (4) that uses GENERATESEQ to compute an efficient strategy is shown in Fig. 4. In Line 1, V is sequenced using GENERATESEQ. In Line 5, all possible valid substrategies Φ for the set $D(i)$ are computed. The function DFS(G, U, v) performs depth-first search on subgraph of G induced by U, starting from the vertex v, to obtain the vertices reachable from v passing only through U. This function is used to compute $X(i)$ and $S(i)$ in lines 6 and 7, respectively. For each $\phi \in \Phi$, a configuration $C \in \mathcal{C}(v^{(i)})$ that minimizes the cost $\mathcal{R}_{\mathcal{V}}(i, \phi \cup \{(v^{(i)}, C)\})$ is computed in lines 10–22: Line 12 computes $\mathcal{H}_{\mathcal{V}}(i, \phi')$; lines 13–17 use DP tables $v^{(j)}.tbl$ to get the costs $\sum_{X(j) \in S(i)} \mathcal{R}_{\mathcal{V}}(j, \phi'')$. For a substrategy ϕ, if a better configuration C is found for $v^{(i)}$, $v^{(i)}.cfg$ and $v^{(i)}.tbl$ are updated in lines 19–20. Finally, in Line 25, the minimum cost of the best strategy for G is returned. For simplicity, we do not show the details of extracting the best strategy from the stored configurations $v^{(i)}.cfg$, but a simple back-substitution, starting from $v^{(|V|)}.cfg$ provides the best strategy found by FINDBESTSTRATEGY. Overall computa-

```
Procedure FINDBESTSTRATEGY ($G = (V, E)$)
1:  $\mathcal{V} \leftarrow$ GENERATESEQ$(G)$                        // Refer Fig. 3
2:  $\forall_{v \in V}, v.tbl \leftarrow \varnothing$                       // DP table
3:  $\forall_{v \in V}, v.cfg \leftarrow \varnothing$                      // Best configs
4:  for $i = 1$ to $|V|$ do
5:      $\Phi \leftarrow \prod_{v \in v^{(i)}.d}\{(v, C) \mid C \in \mathcal{C}(v)\}$       // $\Phi_{|D(i)}$
6:      $X \leftarrow$ DFS$(G, \mathcal{V}_{\leq i}, v^{(i)})$                      // X(i)
7:      $S \leftarrow \bigcup_{v \in X - \{v^{(i)}\}}$ DFS$(G, \mathcal{V}_{<i}, v)$          // S(i)
8:      for all $\phi \in \Phi \vee \{\varnothing\}$ do
9:          $min\_cost \leftarrow \top$
10:         for all $C \in \mathcal{C}(v^{(i)})$ do
11:             $\phi' \leftarrow \phi \cup \{(v^{(i)}, C)\}$
12:             $cost \leftarrow \mathcal{H}_{\mathcal{V}}(i, \phi')$
13:             for all $X' \in S$ do
14:                 $j \leftarrow \max_{v^{(k)} \in X'} k$
15:                 $\phi'' \leftarrow \{(v, \phi'(v)) \mid v \in v^{(j)}.d\}$
16:                 $cost \leftarrow cost + v^{(j)}.tbl(\phi'')$
17:             end for
18:             if $cost < min\_cost$ then
19:                 $v^{(i)}.tbl(\phi) \leftarrow min\_cost \leftarrow cost$
20:                 $v^{(i)}.cfg(\phi) \leftarrow C$
21:             end if
22:         end for
23:     end for
24: end for
25: return $v^{(|V|)}.tbl(\varnothing)$
```

Fig. 4. Dynamic programming based algorithm to compute an efficient strategy for a computation graph G.

tional complexity of FINDBESTSTRATEGY is $O(|V|^2 K^{M+1})$, where $K = \max_{v \in V} |\mathcal{C}(v)|$ is the maximum number of configurations for a layer, and $M = \max_{v^{(i)} \in V} |D(i)|$ is the size of the largest dependent set.

C. Example: InceptionV3

As described earlier, DNN graphs are generally sparse with a few high degree nodes. For instance, the computation graph of InceptionV3 (Fig. 5) has 218 nodes, of which 206 of them have a node degree of < 5 and the remaining 12 nodes have degree ≥ 5. Number of parallelization configurations per vertex of InceptionV3 vary between 10 and 30 for $p = 8$ GPUs, and the maximum number of configurations reaches up to 100 (i.e., $K = 100$) for $p = 64$ GPUs. Our experiments show that when breadth-first ordering is used, the sizes of dependent sets reach up to 10, leading to $K^{M+1} \geq 10^{11}$ (for $p = 8$) combinations to be analyzed to find the best configuration, making it prohibitively expensive in practice, in terms of both time and space (Refer Table I). However, by ordering the vertices using GENERATESEQ, $|D(i) \cup \{v^{(i)}\}|$ for any i is maintained to be ≤ 3, and the maximum number of combinations analyzed per vertex by the algorithm, $K^{M+1} \leq 25200$, for $p = 8$, enabling us to find the best configurations within a few seconds. This is because GENERATESEQ makes sure that the nodes with high degree (nodes 171 and 193 in Fig. 5) are placed in the sequence only after their (low degree) neighbors, and their ancestors/descendants are placed in the sequence, thus ensuring that the sizes of dependent sets of these high degree nodes remains small.

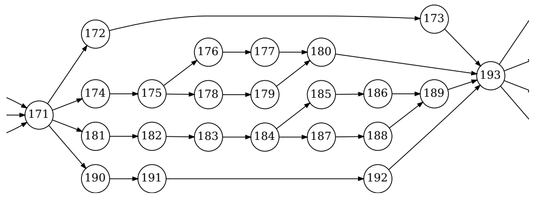


Fig. 5. Computation subgraph corresponding to InceptionE module of InceptionV3. A similar structure repeats throughout the graph. Nodes 171 and 193 have high degree, while the rest of the nodes are sparse.

IV. EXPERIMENTAL RESULTS

We evaluate our technique on four different benchmarks, each having a different graph structure: a) AlexNet [13] is a image classification *convolutional network* whose computation graph is a simple path graph, where each layer is only connected to the next layer; b) InceptionV3 [14] is a *deep CNN* that uses inception modules to increase the number of layers while maintaining a reasonable computational budget. The nodes are split and concatenated (refer Fig. 5) at the beginning and end of each inception module, respectively, leading to a few high degree nodes; c) RNNLM [15] is a two-layer *recurrent* neural network consisting of LSTM cells, used for *language modeling* tasks; and finally, d) Transformer [16] model is a *non-recurrent neural machine translation* model, whose computation graph is quite different from recurrent networks such as RNNLM. We used ImageNet-1K [17] dataset for CNNs, Billion-Word [18] for RNNLM, and WMT EN$\rightarrow$DE [19] for the NMT (Transformer) task. A batch size of 128 was used for CNNs, and 64 was used for the rest of the benchmarks. We compare our results against data parallelism, expert-designed strategies, and the strategies suggested by FlexFlow [7].

FlexFlow: FlexFlow is a deep learning framework that automatically finds fast parallelization strategies. It uses a general Markov Chain Monte Carlo (MCMC) search algorithm to explore the search space. When the search finishes, the framework returns the best strategy it has discovered. As this approach is based on meta-heuristics, the framework could get stuck in a local minima, returning a sub-optimal strategy. An initial candidate from the search space needs to be provided to MCMC to begin the search process, and the efficiency of the strategy found by FlexFlow might also vary depending on the initial candidate. As suggested in [7, Section 6.2], we use expert-designed strategies as the initial candidates in our evaluation, so that FlexFlow can improve upon them.

Expert strategies: Expert-designed parallelization strategies were developed by domain experts on a case-by-case basis. Since not all DNNs have well-defined expert-designed strategies proposed, we chose the ones that were the most relevant, as also used by the previous works [7], [10]. For CNNs, [5] proposes using data parallelism for the convolution layers, and switching to parameter parallelism for the fully-connected layers. This technique is referred to as *"one weird trick" (OWT)*. We use this technique to evaluate both AlexNet and InceptionV3. For RNNs, a data+pipeline parallelism strategy was proposed in [1], where different layers of RNN are placed on different devices to achieve pipeline parallelism, and each layer is replicated on the remaining devices for data parallelism. We compare against this strategy for RNNLM. For the Transformer model, we compare against the hybrid parallelism strategy suggested in [3], which primarily focuses on training large Transformer models within the memory constraints, while also achieving good parallel execution efficiency. Since neither our technique, nor FlexFlow or various expert-designed strategies used in our experiments perform any semantic changes to the DNN, the final trained accuracy of all the strategies match the accuracy of the original model.

A. Runtime overhead

In this subsection, we measure the time taken by different approaches to find the best strategies for the four benchmarks. We compare the running time of our approach that uses GENERATESEQ to order the vertices, against breadth-first (BF) ordering (Subsection III-A), and FlexFlow that uses meta-heuristics to find efficient strategies. We implemented our approach in a prototype tool written in Python, available at https://github.com/baidu-research/PaSE. The measurements were performed on a machine with Intel Xeon E5 (Sandy-Bridge) processor and 1080Ti GPUs. Unlike our approach that uses analytical costs, FlexFlow microbenchmarks the operators on GPUs and uses the execution results to find best strategies. Table I shows the time taken by different approaches to find the best strategies. For measuring the running times, as suggested in [7, Section 6.2], we stop FlexFlow's search algorithm either when it is unable to improve the best discovered strategy for half the search time, or when it has reached 250,000 iterations.

As the computation graph of AlexNet is a simple path graph, sizes of both $D_B(i)$ and $D(i)$ are just one for different vertices. Hence, both BF and GENERATESEQ ordering are able to efficiently compute the best strategy in similar time. However, for InceptionV3, BF ordering runs out of memory due to high node degree of a few vertices as detailed in Subsection III-C.

For RNNLM, since an RNN operator (with LSTM cells) can be efficiently represented in a single iteration space, we represent the complete RNN operator (including the recurrent steps) as a single vertex in the computation graph. The iteration space of an RNN operator is a five-dimensional space consisting of layer, batch, sentence sequence (recurrent

TABLE I
TIME TAKEN BY DIFFERENT ALGORITHMS TO FIND EFFICIENT PARALLELIZATION STRATEGIES. (UNIT: *mins:secs.msecs*)

p	**AlexNet**			**InceptionV3**			**RNNLM**			**Transformer**		
	BF	FlexFlow	Ours	BF	FlexFlow	Ours	BF	FlexFlow	Ours	BF	FlexFlow	Ours
4	0:00.234	0:02.54	0:00.226	OOM	1:09.21	0:14.398	0:00.070	1:47.07	0:00.057	OOM	NA	0:09.752
8	0:00.260	0:02.77	0:00.253	OOM	2:26.51	0:20.018	0:00.084	2:44.29	0:00.086	OOM	NA	0:28.798
16	0:00.303	0:06.98	0:00.295	OOM	6:05.82	0:39.791	0:00.109	7:12.19	0:00.069	OOM	NA	2:10.882
32	0:00.385	0:07.92	0:00.361	OOM	15:37.96	1:26.039	0:00.167	11:08.22	0:00.131	OOM	NA	9:13.022
64	0:00.485	4:17.30	0:00.475	OOM	37:17.04	3:16.253	0:00.271	17:21.98	0:00.215	OOM	NA	31:23.187

p: Number of GPUs; BF: Breadth-First ordering; OOM: Out-Of-Memory; NA: Not Available.

steps are captured by this dimension), output, and hidden dimensions. This is different from the way RNN is modelled in FlexFlow. In FlexFlow, the recurrent dimension is unrolled (we use a unroll factor of 40, same as [7]), and each iteration is represented as a vertex in the graph. By representing the whole RNN operator as a single vertex in our approach, in addition to tremendously reducing the graph size, it also allows our approach to analyze configurations that take advantage of inherent pipeline parallelism present within an RNN operator. Configurations that split the 'layer' and 'sentence sequence' dimensions capture intra-layer pipeline parallelism in a RNN layer. With this representation, the computation graph of RNNLM reduces to a simple path graph. Hence, both BF and GENERATESEQ orderings efficiently find the best strategies within a second.

Similar to InceptionV3, a Transformer model has a large number of sparse vertices with a very few dense vertices. However, unlike InceptionV3, these high degree vertices (such as the final output of encoder) have long live ranges, that eliminate possible orderings that can reduce the dependent sets as effectively as in InceptionV3. This causes FINDBESTSTRATEGY to take longer to find the best strategy for Transformer. As with InceptionV3, BF ordering fails to find the best strategy for Transformer. We were unable to successfully implement and analyze the Transformer model with FlexFlow for comparison.

B. Comparison of performances of different strategies

We compare the actual parallel training throughputs of the best strategies proposed by our approach against data parallel, FlexFlow and expert designed strategies. The experiments were performed on varying number of GPUs ranging from 4 (on a single node) to 64 (spread across 8 nodes), incremented in powers of 2. The nodes are connected to each other using InfiniBand interconnection network. We evaluated our results on the following two processing environments: a) a multi-node/multi-gpu system where each node contains 8 GeForce GTX 1080 Ti GPUs (with sm_61 compute capability) fully-connected using PCIe links; b) a multi-node/multi-gpu system where each node contains 8 GeForce RTX 2080 Ti GPUs (with sm_75 compute capability) fully-connected using PCIe links. We implemented all the benchmarks in Mesh-TensorFlow [3] framework for evaluation. Although our cost function (in Equation 1) ignores accounting for certain optimizations such as inter-layer communication and computation overlap when computing costs for the best strategies, all such feasible optimizations were allowed to be performed by Mesh-TensorFlow

TABLE II
BEST STRATEGIES FOUND BY FINDBESTSTRATEGY FOR A SYSTEM OF 4 NODES, EACH CONSISTING OF 8 1080TI GPUS ($p = 32$).

	Layers	**Dimensions**	**Configuration**	**Legend**
Alexnet	Conv 1–4	$bchwnrs$	$(32, 1, 1, 1, 1, 1, 1)$	
	Conv 5	$bchwnrs$	$(16, 2, 1, 1, 1, 1, 1)$	b: batch
	FC 1, FC 3	bnc	$(1, 4, 8)$	h: height
	FC 2	bnc	$(1, 8, 4)$	w: width
	Softmax	bn	$(1, 4)$	c: in-channel
Inception	Modules A–D[†]	$bchwnrs$	$(32, 1, 1, 1, 1, 1, 1)$	n: out-channel
	Module E[†]	$bchwnrs$	$(16, 1, 1, 1, 2, 1, 1)$	r: filter height
	FC	bnc	$(1, 2, 16)$	s: filter width
	Softmax	bn	$(1, 2)$	
RNNLM	Embedding	$bsdv$	$(1, 1, 1, 32)$	b: batch
	LSTM	$lbsde$	$(2, 4, 1, 2, 2)$	s: sequence len
	FC	$bsvd$	$(1, 1, 32, 1)$	d: embed dim
	Softmax	bsv	$(1, 1, 32)$	e: hidden dim
Transformer	Embedding	$bsdv$	$(1, 1, 1, 16)$	v: vocab size
	Multihead attn[†]	$bshck$	$(16, 1, 2, 1, 1)$	l: RNN layers
	Feed forward[†]	$bsde$	$(16, 1, 1, 2)$	h: heads
	FC	$bsvd$	$(1, 1, 16, 1)$	c: query channels
	Softmax	bsv	$(1, 1, 16)$	k: kv channels

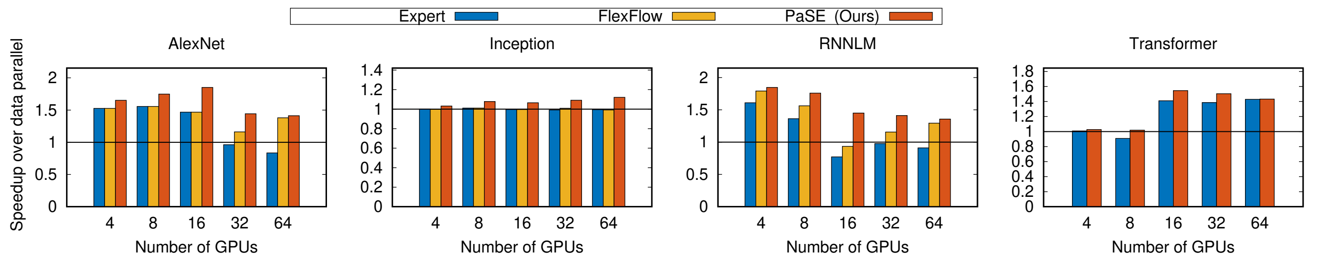
[†]For simplicity, we report configurations at module level. In practice, they are further broken down into their constituent layers such as conv, FC, etc.

in our experiments. Fig. 6 shows the speedups achieved by various strategies over data parallelism on 1080Ti and 2080Ti systems. On 1080Ti machines, the strategies proposed by our approach achieve a speedup of up to $1.85\times$ over data parallelism. As shown in Fig. 6a, our strategies consistently perform better than expert-designed strategies, and the strategies proposed by FlexFlow. On 2080Ti machines, our strategies achieve up to $4\times$ speedup over data parallelism, and outperform both expert-designed strategies and the strategies from FlexFlow. 2080Ti GPUs do not support peer-to-peer data access over PCIe links, leading to poor hardware communication efficiency, while having a higher computational peak than 1080Ti GPUs. This leads to a very low machine balance (ratio between peak communication bandwidth and peak GFLOPS). Thus, inefficiencies in parallelization strategies are much more pronounced on 2080Ti nodes, allowing us to achieve up to $4\times$ performance improvement over data parallelism.

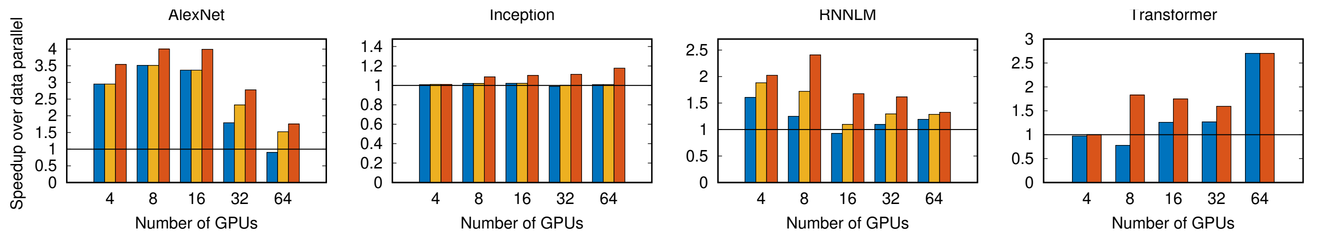
C. Analysis of computed strategies

Table II shows the best strategies found by FINDBESTSTRATEGY for training various DNNs on $p = 32$ 1080Ti GPUs (spread across 4 nodes).

AlexNet has five convolution layers, followed by three fully-connected layers. On 32 GPUs, our technique suggests to use data parallelism for the first four convolution layer, and to split



(a) Speedup on 1080Ti GPUs.



(b) Speedup on 2080Ti GPUs.

Fig. 6. Speedup achieved by various parallelism strategies over data parallelism.

the in-channel dimension of the last convolution layer into two. For the first and third fully-connected layers, the algorithm suggests to split the out-channel and in-channel dimensions by 4 and 8, respectively, while for the second fully-connected layer, it suggests to split them by 8 and 4, respectively. This alternating pattern effectively eliminates any inter-layer communication among the fully-connected layers. This differs from OWT [5], where only the out-channel dimension is parallelized for fully-connected layers, leading to high volume of all-gather communication between fully-connected layers.

Inception network has a sequence of inception modules (A – E) composed of convolution layers, followed by a single fully-connected layer. Our approach suggests to use data parallelism for modules A–D, but for the module E, the algorithm suggests a hybrid of data+parameter parallelism. This is because as the modules get deeper, their output channels get larger, and our algorithm finds pure data parallelism to be less effective here.

RNNLM is composed of an embedding layer, two layers of LSTM cells, and a final projection layer, whose computations are dominated by GEMM. The embedding layer has a huge vocabulary dimension v, and a much smaller embedding dimension d. FINDBESTSTRATEGY prefers fully splitting the vocabulary dimension for the embedding and projection layers. For the LSTM cells, the algorithm suggests to fully split the LSTM layer dimension l (thus utilizing intra-layer pipeline parallelism), and partially split the other three dimensions – batch, hidden, and output dimensions – to varying degrees.

Transformer is a non-recurrent self attention based NMT model. A hybrid parallelism strategy was suggested in [3], where the batch dimension of all the layers are split m-way, and model dimensions of different layers – vocabulary dimension, feed-forward hidden layer dimension, and attention heads – are split n-way. Our approach suggests to use parameter parallelism for embedding and softmax layers, and to use a hybrid data+parameter parallelism for the remaining layers.

V. Limitations and Future Work

We discuss some of the limitations of our approach, and possible directions for future work.

Computational complexity of our algorithm FINDBESTSTRATEGY is $O(|V|^2 K^{M+1})$, where $K = \max_{v \in V} |\mathcal{C}(v)|$, and $M = \max_{v^{(i)} \in V} |D(i)|$. DNN graphs are typically sparse allowing us to carefully order the vertices and efficiently compute parallelization strategies in practice. However, there do exist a few DNNs (such as DenseNet [20]) whose graphs are uniformly dense. No possible arrangement of vertices can effectively reduce the size M for such graphs, leading to high runtime overhead for our algorithm.

Inter-layer pipeline parallelism is ignored in our approach. Since DNNs typically do not have sufficient pipeline parallelism potential (without semantic changes), this does not severely affect our solutions. However, this prevents us from capturing the effects of overlapping computation and communication between different layers. (Computation / communication overlap within layers are accounted for in layer costs t_l.) Thus, it would be beneficial to incorporate pipeline parallelism into our formulation to improve its accuracy further.

Although our method is applicable to *heterogeneous architectures*, it does not explicitly include heterogeneity into the cost model. In case of heterogeneous systems, the peak FLOP and bandwidth, of the weakest computation node and communication link, respectively, are used to compute t_l and t_x, as they form the primary bottlenecks. In future work, we plan to extend the model to include heterogeneity.

For simplicity, we ignored several *low level details* such as cache effects in our cost model. This is not an inherent limitation of our approach as such, and in future work, we plan to fine-tune the cost model further by including these low level details to improve its accuracy.

VI. Related Work

Data parallelism has been widely used as the standard technique to parallelize DNNs. Data parallelism requires model parameters to be fully replicated on all devices. This typically leads to poor performance and scalability for large models. Some previous works [21], [22] have tried to address this by storing the parameters sharded across different devices, while still using data parallelism for computation. This leads to additional communication. Although these techniques propose methods to efficiently perform these communications, the minimum volume of data that needs to be communicated remains the same, leading to a fundamental bottleneck. In contrast, our method splits computation of each layer along different dimensions, minimizing the actual communication volume. Memory and communication optimizations proposed by [21], [22] are thus orthogonal to ours, and can be independently applied on top of ours to further improve the performance.

One weird trick (OWT) was introduced in [5] to parallelize CNNs, where data parallelism is used for convolutional layers, and parameter parallelism is used for the rest. This trick, while applicable only for CNNs, works reasonably well in practice. However, as evident from Fig. 6, even better performance can be achieved by a more sophisticated hybrid parallelism. A DP based approach to automatically find efficient strategies for CNNs was presented in [10]. The method exploits the fact that CNNs typically have nodes with single in-/out-edges, and computes efficient strategies by reducing the graph through *node and edge eliminations*. However, this technique fails on other tasks such as LM and NMT whose graphs do not have this special property. In contrast, our method is not limited to CNNs, and can find efficient strategies for various types of networks like RNNs and Transformers within a few minutes. Additionally, we define our parallelization configuration to split any dimension in the iteration space, while in [10], only the output tensor dimensions are split. This heavily restricts the search space, since some of the dimensions are not considered as possible choices for parallelization. *Tofu* [23] also uses the same DP formulation proposed in [10] to find the best strategies to parallelize fine-grained dataflow graphs, and thus suffers from the same limitation as [10], preventing them from being able to handle models such as Transformer, whose graphs do not have a linear structure.

FlexFlow [7] uses a general Markov Chain Monte Carlo (MCMC) meta-heuristic search algorithm to explore the search space to discover the best strategy. The strategy returned by their framework need not necessarily be optimal. While their method takes the whole search space into consideration, our method ignores inter-layer pipeline parallelism. In return, our method is able to find an efficient strategy for various DNNs much faster than FlexFlow, and is not subject to the limitation of getting stuck at a local minima. REINFORCE [9] and [24] use machine learning to find efficient device placement for various layers to achieve efficient pipeline parallelism. They ignore data and parameter parallelism in their search process. Further, they require multiple GPUs and take several hours to find an efficient strategy. [25] and [26] use polyhedral compilation techniques to optimize execution of individual DNN operators on a single GPU. These techniques can be orthogonally used with ours to further improve the performance within each GPU.

PipeDream [6] allows for semantic changes to the model to improve parallel training times. In our approach, we do not consider semantic modifications. While our technique heavily relies on parameter parallelism, this is completely ignored in [6]. Thus, we find their approaches to be complementary to ours: the computation graph can be first split into multiple stages using the formulation proposed in [6] to achieve inter-batch pipeline parallelism, and the subgraphs from each stage can be further parallelized with data+parameter parallelism using our approach.

Several expert-designed strategies [1], [5] have been proposed for different networks based on domain specific knowledge. Each network has to be individually analyzed manually to come up with an efficient strategy. Further, these strategies need not be necessarily optimal. The method proposed in this paper automates this process, and can point the expert towards the right direction for parallelization. While the focus of this paper is to find the best parallelization strategies for DNNs, frameworks such as Mesh-TensorFlow [3] and GShard [12] enable automatically converting these user-specified strategies into efficient parallel programs.

VII. Conclusion

In this paper, we presented a method to automatically find efficient parallelization strategies for DNNs. We proposed a recurrence formulation to compute the minimum cost of a computation graph, and presented a technique to efficiently compute the best parallelization strategies within a few minutes. We evaluated our results against data parallelism, expert designed strategies, and the strategies proposed by a deep learning framework, FlexFlow. Results show that the strategies proposed by our method outperform the standard data parallelism by a factor of up to $1.85\times$ and $4\times$ on multi-node systems with 1080Ti and 2080Ti GPUs, respectively. In addition, strategies from our method perform better than expert-designed strategies, and the ones proposed by FlexFlow.

Acknowledgment

We thank Kenneth Ward Church, Prashant Singh Rawat, and the anonymous reviewers for their valuable feedback and suggestions that helped improve the paper.

References

[1] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, Łukasz Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, "Google's neural machine translation system: Bridging the gap between human and machine translation," *CoRR*, vol. abs/1609.08144, 2016.

[2] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima," *CoRR*, vol. abs/1609.04836, 2016.

[3] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young, R. Sepassi, and B. Hechtman, "Mesh-tensorflow: Deep learning for supercomputers," in *Advances in Neural Information Processing Systems 31*, 2018, pp. 10 414–10 423.

[4] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. aurelio Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems 25*, 2012, pp. 1223–1231.

[5] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *CoRR*, vol. abs/1404.5997, 2014.

[6] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: generalized pipeline parallelism for DNN training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019.* ACM, 2019, pp. 1–15.

[7] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks," *CoRR*, vol. abs/1807.05358, 2018.

[8] Y. Huang, Y. Cheng, D. Chen, H. Lee, J. Ngiam, Q. V. Le, and Z. Chen, "GPipe: Efficient training of giant neural networks using pipeline parallelism," *CoRR*, vol. abs/1811.06965, 2018.

[9] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device placement optimization with reinforcement learning," in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ser. ICML'17, 2017, pp. 2430–2439.

[10] Z. Jia, S. Lin, C. R. Qi, and A. Aiken, "Exploring hidden dimensions in parallelizing convolutional neural networks," in *ICML*, 2018.

[11] M. Wolfe, "More iteration space tiling," in *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '89, 1989, pp. 655–664.

[12] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen, "Gshard: Scaling giant models with conditional computation and automatic sharding," *CoRR*, vol. abs/2006.16668, 2020.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, 2012, pp. 1097–1105.

[14] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[15] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur, "Recurrent neural network based language model," in *Eleventh annual conference of the international speech communication association*, 2010.

[16] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30*, 2017, pp. 5998–6008.

[17] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[18] C. Chelba, T. Mikolov, M. Schuster, Q. Ge, T. Brants, and P. Koehn, "One billion word benchmark for measuring progress in statistical language modeling," *CoRR*, vol. abs/1312.3005, 2013.

[19] "Workshop on statistical machine translation." [Online]. Available: http://www.statmt.org/wmt14/

[20] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[21] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimization towards training A trillion parameter models," *CoRR*, vol. abs/1910.02054, 2019.

[22] Y. Xu, H. Lee, D. Chen, H. Choi, B. A. Hechtman, and S. Wang, "Automatic cross-replica sharding of weight update in data-parallel training," *CoRR*, vol. abs/2004.13336, 2020.

[23] M. Wang, C.-c. Huang, and J. Li, "Supporting very large models using automatic dataflow graph partitioning," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19, 2019.

[24] A. Mirhoseini, A. Goldie, H. Pham, B. Steiner, Q. V. Le, and J. Dean, "A hierarchical model for device placement," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018.

[25] V. Elango, N. Rubin, M. Ravishankar, H. Sandanagobalane, and V. Grover, "Diesel: DSL for linear algebra and neural net computations on GPUs," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, ser. MAPL 2018, 2018, p. 42–51.

[26] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. Devito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "The next 700 accelerated layers: From mathematical expressions of network computation graphs to accelerated gpu kernels, automatically," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 4, Oct. 2019.

APPENDIX

A. *Optimality of* FINDBESTSTRATEGY

Theorem 1: Let $G = (V, E)$ be a computation graph for a DNN that is executed on p devices with average FLOP-to-bytes ratio r. Let $\mathcal{V}$ be a sequence for V, and Φ be the set of all possible strategies for G. Then,

$$\mathcal{R}_{\mathcal{V}}(|V|, \varnothing) = \min_{\phi \in \Phi} \mathcal{F}(G, \phi).$$

Proof: From Equation (4), we have,

$$\begin{aligned}
&\mathcal{R}_{\mathcal{V}}(|V|, \varnothing) \\
&= \min_{C \in \mathcal{C}(v^{(|V|)})} \mathcal{H}_{\mathcal{V}}(|V|, \{(v^{(|V|)}, C)\}) + \sum_{\substack{X(j) \in \\ S(|V|)}} \mathcal{R}_{\mathcal{V}}(j, \{(v^{(|V|)}, C)\}) \\
&= \min_{\phi \in \Phi} \sum_{v^{(i)} \in V} \mathcal{H}_{\mathcal{V}}(i, \phi) \qquad (5) \\
&= \min_{\phi \in \Phi} \sum_{v^{(i)} \in V} \left[t_l(v^{(i)}, \phi, r) + \sum_{v^{(j)} \in N(v^{(i)}) \cap \mathcal{V}_{>i}} r \times t_x(v^{(i)}, v^{(j)}, \phi) \right] \\
&= \min_{\phi \in \Phi} \sum_{v^{(i)} \in V} \left[t_l(v^{(i)}, \phi, r) \right] + \sum_{(v^{(i)}, v^{(j)}) \in E} \left[r \times t_x(v^{(i)}, v^{(j)}, \phi) \right] \qquad (6) \\
&= \min_{\phi \in \Phi} \mathcal{F}(G, \phi)
\end{aligned}$$

Equality (5) is due to the fact that for any vertex $v^{(i)} \in V$, $X(i) = (\bigcup_{U \in S(i)} U) \cup \{v^{(i)}\}$, and the connected sub-components in $S(i)$ are pairwise disjoint, i.e., for $U_1, U_2 \in S(i)$, $U_1 \neq U_2 \implies U_1 \cap U_2 = \varnothing$. Further, a computation graph G is weakly connected. Thus, $X(|V|) = V$.

Equality (6) is due to the fact that the union of the pairwise disjoint sets $\{\{v^{(i)}, v^{(j)}\} \mid v^{(j)} \in N(v^{(i)}) \cap \mathcal{V}_{>i}\}$ is the set of edges of (undirected) $G = (V, E')$, where $E' = \{\{u, v\} \mid (u, v) \in E\}$. ■

B. *Correctness of* GENERATESEQ

Theorem 2: Given a computation graph $G = (V, E)$, and a sequence $\mathcal{V}$ computed by GENERATESEQ in Fig. 3, for any $v^{(i)} \in V$, $v^{(i)}.d = D(i)$.

Proof: We will show this by induction. Let $\mathcal{V}$ be the sequence generated by GENERATESEQ. We define the dependent set of a vertex $v^{(j)}$ restricted to the first k vertices in the sequence, $D(j)_{|k} = N(X(j) \cap \mathcal{V}_{\leq k} \cup \{v^{(j)}\}) \cap \mathcal{V}_{>\min(j,k)}$. Clearly, for any $j \leq k$, $D(j)_{|k} = D(j)$. We will show that at the end of any iteration i in GENERATESEQ, the invariant $v^{(j)}.d = D(j)_{|i}$ holds. This will prove that at the end of the algorithm, for any j, $v^{(j)}.d = D(j)_{||V|} = D(j)$.

Induction base: The invariant trivially holds just before the first iteration (where $i = 0$) due to the initialization in Line 1.

Induction step: As a hypothesis, consider that the invariant is true at the end of an iteration $i - 1$. Let $v^{(i)}$ be the vertex chosen at iteration i in Line 5. For any j s.t. $v^{(i)} \notin X(j)$, $D(j)_{|i} = N(X(j) \cap \mathcal{V}_{\leq i} \cup \{v^{(j)}\}) \cap \mathcal{V}_{>i} = N(X(j) \cap \mathcal{V}_{<i} \cup \{v^{(j)}\}) \cap \mathcal{V}_{\geq i} = D(j)_{|i-1}$. For any j s.t. $v^{(i)} \in X(j)$ (i.e., if $v^{(j)} \in v^{(i)}.d$),

$$\begin{aligned}
D(j)_{|i} &= N(X(j) \cap \mathcal{V}_{\leq i} \cup \{v^{(j)}\}) \cap \mathcal{V}_{>i} \\
&= N((X(j) \cup X(i)) \cap \mathcal{V}_{\leq i} \cup \{v^{(j)}\}) \cap \mathcal{V}_{>i} \qquad (7) \\
&= N((X(j) \cup X(i)) \cap \mathcal{V}_{<i} \cup \{v^{(i)}, v^{(j)}\}) \cap \mathcal{V}_{>i} \\
&= N((X(j) \cap \mathcal{V}_{<i} \cup \{v^{(j)}\}) \\
&\quad \cup (X(i) \cap \mathcal{V}_{<i} \cup \{v^{(i)}\})) \cap \mathcal{V}_{>i-1} - \{v^{(i)}\} \\
&= D(j)_{|i-1} \cup D(i)_{|i-1} - \{v^{(i)}\}
\end{aligned}$$

Equality (7) is due to the fact that $X(i) \subseteq X(j)$. The update in Line 8 in Fig. 3 indeed performs this exact operation, making sure that the invariant is correctly maintained at the end of iteration i. Thus, at the end of $|V|$ iterations, $v^{(j)}.d = D(j)$ for any $v^{(j)} \in V$. ■