Efficient and Systematic Partitioning of Large and Deep Neural Networks for Parallelization

Haoran Wang[1,2](✉), Chong Li[1], Thibaut Tachon[1], Hongxing Wang[1], Sheng Yang[1], Sébastien Limet[2], and Sophie Robert[2]

[1] Huawei Technologies, Paris, France
{wanghaoran19,ch.l,thibaut.tachon,hongxingwang, sheng.yang1}@huawei.com

[2] Université d'Orléans, Orléans, France
{sebastien.limet,sophie.robert}@univ-orleans.fr

Abstract. Deep neural networks (DNNs) are playing an increasingly important role in our daily life. Since the size of DNNs is continuously growing up, it is highly important to train them effectively by distributing computation on multiple connected devices. The efficiency of training depends on the quality of chosen parallelization strategy. Being able to find a good parallelization strategy for a DNN in a reasonable amount of time is not trivial. Previous research demonstrated the possibility to systematically generate good parallelization strategies. However, systematic partitioning still suffers from either a heavy preprocessing or poor quality of parallelization. In this paper, we take a purely symbolic analysis approach by leveraging the features of DNNs like dense tensor balanced computation. We propose the Flex-Edge Recursive Graph and the Double Recursive Algorithm, successfully limiting our parallelization strategy generation to a linear complexity with a good quality of parallelization strategy. The experiments show that our solution significantly reduces the parallelization strategy generation time from hours to seconds while maintaining the parallelization quality.

Keywords: Distributed algorithm · Distributed machine learning · Neural network partitioning

1 Introduction

The past decade has witnessed the dramatic development of deep learning in almost every domain in our daily ife. On one hand, DNN fra meworks like [1–3] increase the efficiency of DNN development by automating DNN training based on the user's description of the network. On the other hand, the increase of computing power, driven by the availability of new accelerators, enables the design of larger and more complex deep neural networks (DNNs) [4,5]. Deeper and wider DNNs enable new applications but require efficient distribution of computation on connected devices for accelerating the training process.

A DNN consists of hundreds or even thousands of operators whose inputs and outputs are tensors (i.e., multidimensional arrays). A DNN can be represented as a *computation graph* whose vertices are the operators and whose edges are the tensors. A *parallelization strategy* defines how to partition the data and the operators of a DNN into multiple devices. The most commonly used parallelization strategy approach is *Data Parallelism* [6]. In data parallelism, each device holds a replica of the entire network and trains it with a subset of training data. This approach is efficient because the subsets of training data are independent, hence there is no communication during the computation of the operators. Data parallelism is not suitable for the layers with large parameter tensors (e.g., *fully connected layers*) due to a long parameter synchronization time. Another widely used approach is *Model Parallelism* [7] where the DNN operators and tensors are distributed over the computing devices. For a given operator, there exist different model parallelisms that introduce different extra communications.

The *Hybrid Parallelism* approach [8,9] has been recently proposed to overcome the disadvantages of data and model parallelism. The hybrid parallelism implements either data or model parallelism on different operators to achieve better performance. By using hybrid parallelism, communication overhead caused by inconvenient parallelization strategies is reduced. Meanwhile, hybrid parallelism introduces data redistribution between operators if two connected operators are assigned different parallelization strategies. Based on the preceding information, the efficiency of hybrid parallelism depends on the parallelization strategy of each operator. Searching for an optimal parallelization strategy for a DNN is a combinatorial problem: the number of strategies grows exponentially with the number of operators and polynomially with the number of devices. Therefore, it is difficult to find the optimal parallelization strategy within an acceptable time regarding the size of the search space. Directly comparing all the possible parallelization strategies of a DNN by profiling is not realistic. For instance, profiling a 21-layer VGG16 [10] network already takes more than 24 h [11].

Many DNN frameworks provide good functionalities on data loading, computational graph execution, and fault tolerance. Some of the frameworks support hybrid parallelism with a manually configured parallelization strategy. However, automatically offering the optimal hybrid parallelization strategy is still one of the biggest challenges for these frameworks. In this paper, we thus focus on how to choose efficient hybrid parallelization strategies for DNNs.

FlexFlow [12] uses a randomized Markov Chain Monte Carlo algorithm to circumvent the parallelization strategy complexity. However, these approaches cannot guarantee the searching time nor the optimality of the result. OptCNN [11] proposes a dynamic programming searching algorithm that reduces the complexity w.r.t. the number of operators from exponential to polynomial. Analyzing a large DNN like ResNet [4] only takes few hours. The algorithm mixes profiling and cost model to estimate the global execution time of the training. The cost model is composed of the profiled execution time of each operator and the estimated communication time. However, the execution time of an operator may vary with DNN configuration changes as well as dataset changes. As a result, new profiling and searching need to be processed after each modification of the model or dataset. Moreover, extra profiling and searching time may offset the gain of DNN training time by using more devices. The estimated communication time in OptCNN is the product of the communication bandwidth and data quantity. However, the communication capacity of a parallel machine is not only dominated by the bandwidth but also other factors such as latency, network topology, etc. A unique bandwidth used in OptCNN may induce substantial errors in the evaluation of a strategy and lead to a wrong decision in the choice. Another drawback of OptCNN solution is that the dynamic programming algorithm is designed for handling *fork-join* graphs, which are suitable for convolution neural networks used in computer vision classification. But the algorithm cannot handle multi-input or multi-output graphs used in other areas like natural language processing, recommendation systems, and image segmentation.

To avoid profiling issues, we introduce a purely symbolic cost model based on the semantics of each operator in Sect. 2. We observed that tensors in DNNs are dense multi-dimension arrays and the operators are typically balanced. Inspired by SGL [13], we proposed a 2-part recursive partitioning to eliminate the influence of a machine's communication capacity. Besides, we introduce a Flex-Edge Recursive Graph (FER Graph) to reduce the searching complexity in Sect. 3. We leverage DNN features to let the traversing of FER Graph be topology-independent. We intend to visit the vertices according to their *importance*. In this way, we can guarantee the quality of the generated strategies. Double Recursive Algorithm (D-Rec), presented in Sect. 4, includes Inner Recursion and Outer Recursion. Inner Recursion is designed to partition the computation graph into two parts. Outer Recursion recursively applies Inner Recursion p times to partition the neural network into 2^p parts.

2 Symbolic Cost Model

The cost of a distributed parallel program is the summation of local computation cost and the communication cost: $Cost = Cost_{comp} + Cost_{comm}$. The local computation is the process executed locally on each device without external data. The communication denotes data communicating between devices. Tensors in DNNs are dense multi-dimensional arrays. The operators (e.g., *Matmul*, *Conv*, *Add*, etc.) are massively parallelizable computations which are evenly computed among the devices. Therefore, the number of operations to perform is constant for any distributed strategy and a fixed number of devices. DNN platforms allow load-balancing of the computation operations among the devices such that $Cost_{comp}$ is equal for any chosen algorithm on a given number of devices. Therefore, our asymptotic analysis can focus only on $Cost_{comm}$.

The communication cost is determined by two factors: the communication capacity of the chosen machines, denoted by g, and the quantity of data needed to be transferred denoted by q. To achieve better performance, modern computer clusters, like supercomputers [14,15] and AI accelerator clusters [16,17], have a hierarchical architecture. A unique g cannot describe precisely the communication capacity of modern machines. Valiant [18] proposed to use different g for each hierarchical level. The communication cost can be calculated by the summation of each level contribution: $Cost_{comm} = \sum_i (g_i \times q_i)$, where i is the hierarchical level. We noticed that the hierarchical architectures are also typically symmetric [19]. Inspired by SGL [13], a hierarchical and symmetric machine can be abstracted in a recursive way. For example, a typical GPU architecture shown on the left of Fig. 1, can be described by an abstract machine on the right. The abstract machine has a tree structure, where the leaves are the computing devices and the branch nodes model the hierarchical structure. The communication is analyzed by a recursion. Each recursion step is a level of the tree whose communication capacity is shared as g_i. For each level, g_i does not affect the choice of the parallelization strategy. Therefore, the communication can be recursively analyzed with only the quantity of communicated data q_i, where i becomes the recursion step.

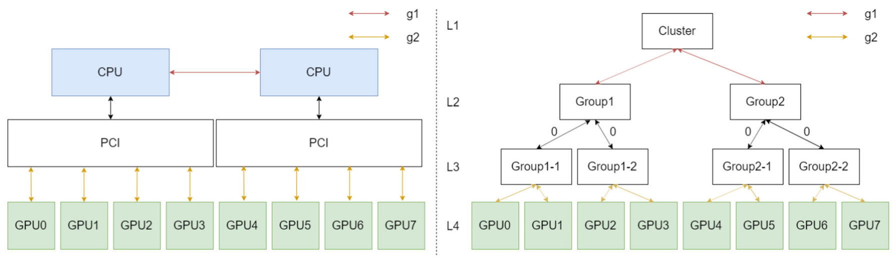


Fig. 1. A typical GPU architecture described by a recursive tree

Parallelization strategy determines how tensors are distributed into devices. We formalize our analysis by logically setting that input and output tensors of operators are evenly distributed among the devices (e.g., GPU0-7 in Fig. 1). The analysis of parallelization strategy is thus equivalent to the analysis of communication quantity. In other words, less communication quantity leads to a better parallelization strategy. Communication and computing overlap techniques [20,21] are orthogonal to our cost analysis: our goal is to find the minimized communication cost regardless of whether overlap techniques are applied.

For each level of the recursive tree, the number of branches depends on the architecture. A specific number of branches acquires a specific set of cost functions. Designing so many cost functions is not realistic. However, each level of the recursive tree is homogeneous, like GPU0-3 in Fig. 1. It can be transformed again into a multi-level tree. Besides, in real academic and industrial practice, the number of devices is usually a power of 2 to achieve the best performance. Therefore, the recursive tree can be transformed into a full binary tree and the partitioning of the symmetric architectures can be realized by recursively dichotomy. As a result, we choose 2-part cost functions to model the cost of partitioning an operator into two parts. Our goal is to find the optimal parallelism policy, so we assume that all devices operate normally and ignore small performance differences between the same devices. In addition, the heterogeneity of symmetric architecture can be decomposed. Based on the above assumptions, homogeneity is applied to all the 2-part analyses in this paper.

It is obvious that partitioning an operator evenly into 2^p part can be done by dichotomy with p recursions. Take a matrix as an example of a tensor, a matrix partitioned into four parts along columns can be the result of partitioning into two parts along column recursively twice; a matrix can be partitioned into 2×2 grid by firstly partitioning along column and then recursively along row. Therefore the recursive 2-part partitioning can be well mapped to the symmetric architecture.

In order to realize the 2-part partitioning, tensors are partitioned along one of the tensor's dimensions at each recursion step. An operator has several input/output tensors, but only a few combinations of tensors' 2-part partitions may lead to optimal communication. For example, partitioning the two input tensors of *MatMul* along column dimension will never lead to optimal communication: all the data need to be communicated. These combinations of tensors' 2-part partitions are defined as *Possible Partition Dimensions (PPDs).* For each PPD, a cost function is defined to return the communication data quantity. The training of DNN is an iterative process that consists of forward and backward propagation. *Forward Propagation* computes the operators with the intermediate *parameters* from input to output and gets a *loss* that estimates the distance between the output and the expected value. *Backward Propagation* will update the intermediate parameters from output to input based on the loss using an optimizer like Adam [22].

There exist two kinds of data communication during the DNN training:

- Q_{op} is the quantity of data needed to be transferred inside an operator. It is composed of Q_f and Q_b. Q_f is the communication quantity between two groups of devices during the forward propagation. Q_b denotes the communication for updating the parameters during the backward propagation process.
- Q_{redist} is the communication quantity between two connected operators. In fact, the output tensor of the previous operator is the same as the input tensor of the second operator. However, as this tensor may have different parallelization strategies for the two connected operators, the data may need to be redistributed. Q_{redist} models this specific communication.

2.1 Communication Inside Operators: Q_{op}

An operator is defined by a type that describes its computational task (e.g., *Type* = *Add*, *Conv*, *MatMul*, *Relu*, etc.). It takes tensors as input and produces tensors as output. We denote $Type.\mathcal{D}_P = \{D_0, D_1, \dots D_k\}$ the set of PPDs for each type of operator. Each PPD can be converted to the partition dimensions of all the tensors in an operator. We denote $d_0, d_1, ...$ the dimensions of a tensor. For example for the *MatMul* (Matrix Multiplication) operator $MatMul.\mathcal{D}_P = \{i, k, j\}$. The two input tensors are respectively of $i \times k$ and $k \times j$ dimensions and the output tensor is of $i \times j$ dimension. As shown in Fig. 2 (bottom), PPD i corresponds to partition the first input tensor and output tensor along d_0 and the second input tensor along either d_0 or d_1.

We define the following notions:

- $shape(tensor)$ denotes the shape of a tensor (e.g., $shape(t) = [4; 4]$).
- $shape(D)$ denotes the shape of a PPD of an operator (e.g., $shape(i) = 4$).
- $shape(d)$ denotes the shape of a dimension of a tensor (e.g., $shape(d_0) = 4$).
- $input_n$ denotes an input tensor according to its index.
- $Q_{op}(D)$ denotes the Q_{op} of a PPD where $Q_{op}(D) = Q_f(D) + Q_b(D)$.
- $Q_{redist}(d, d')$ denotes the operator's Q_{redist} from the partition dimension d of its tensor and another dimension d' of the its connected operators' tensor.

$\boldsymbol{Q_f}$ Communication occurs when an operator is executed on multiple devices. Each device possesses only a part of data. Therefore, data need to be moved between devices to perform the whole computation. We detail here the most representative operators.

MatMul OP is the operator for Matrix Multiplication, described as:

$$output[i][j] \rightarrow \sum_k input_0[i][k] \times input_1[k][j].$$

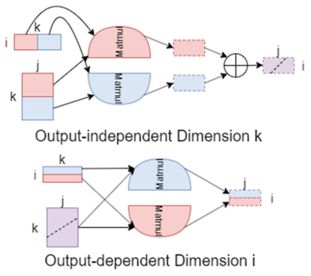


Fig. 2. MatMul semantics (Color figure online)

If we cut according to the output-independent dimension, a reduction still needs to occur to combine the partial results. The dimension cut is not specified for the output (represented with a diagonal dashed line on purple in Fig. 2) but still exists. Hence, for any dimension cut, each device is responsible for computing half of the output tensor. To this end, each device preserves half of its data and communicates the other half to the other device. As we assume both communications can happen simultaneously, the communication cost will be proportional to the amount of data communicated by one of them. The 2-part cost function is as follows:

$$Q_f(k) = \frac{shape(i) \times shape(j)}{2}.$$

If we cut according to an output-dependent dimension, partial results simply have to be concatenated. However, one input is wholly needed by each device to compute their partial result. As this input will be cut eventually, each device must receive the half that it does not possess. The 2-part cost functions are presented below:

$$Q_f(i) = \frac{shape(j) \times shape(k)}{2},$$

$$Q_f(j) = \frac{shape(i) \times shape(k)}{2}.$$

Conv OP represents N-dimension convolutions operators. We name one of the inputs as *kernel*. b, c_i, c_o are batch, input channel, output channel dimensions respectively. $\boldsymbol{x}$ and $\boldsymbol{z}$ are computing dimensions. $\boldsymbol{s}$ is stride and $\boldsymbol{d}$ is dilation rate. Bold italic refers to vector. The description is as follows:

$$output[b][\boldsymbol{x}][c_o] \rightarrow \sum_{\boldsymbol{z} c_i} (kernel[\boldsymbol{z}][c_i][c_o] \\ \times input_0[b][x_0 s_0 + d_0 z_0]...[x_{n-1} s_{n-1} + d_{n-1} z_{n-1}][c_i]).$$

If we cut according to an output-dependent dimension:

$$Q_f(b) = \frac{\prod shape(kernel)}{2},\ Q_f(k) = \frac{\prod shape(input_0)}{2},$$

$$\forall i \in \boldsymbol{x},\ Q_f(i) = \frac{\prod shape(kernel)}{2} + \frac{\prod shape(input_0)}{2}.$$

If we cut according to a output-independent dimension:

$$\forall i \in \boldsymbol{z},\ Q_f(i) = \frac{\prod shape(input_0)}{2} + \frac{\prod shape(output)}{2},$$

$$Q_f(q) = \frac{\prod shape(output)}{2}.$$

Elementwise OP computes each element independently without any communication, for example, *Add*, *Sub*, *Mul*, *ReLU*, *Log*, etc. The description of *Add* is: $output[\boldsymbol{\mu}] \rightarrow input_0[\boldsymbol{\mu}] + input_1[\boldsymbol{\mu}]$ and the cost will be $\forall i \in \boldsymbol{\mu},\ Q_f(i) = 0$.

$\boldsymbol{Q_b}$ is the communication quantity at the end of each backward propagation. Certain operators' need to update one of its tensors during the training. These tensors are referred as *Parameters*. When the *batch* dimension is chosen, parameters hosted by each device need to be communicated to compute the average value. We group the parameter tensors as *param*. The communication quantity of a 2-part partitioning is defined $Q_b = \frac{\prod shape(param)}{2}$.

2.2 Communication Between Operators: $\boldsymbol{Q_{redist}}$

For two connected operators, the output tensor of the first one is the same tensor as the input tensor of the second one. If the two operators choose different partition dimensions for this tensor, it needs to be redistributed which induces communication cost named Q_{redist}. Figure 3 shows two simple situations of redistribution cost.

If the partition strategy of an operator's input tensor and its connected output tensor of the previous operator are equal, $Q_{redist}(d_0, d_0) = 0$. Otherwise, $Q_{redist}(d_0, d_1) = (shape(d_0) \times shape(d_1))/4$. As shown in the second part of Fig. 3, the blue part and red part respectively represent the data stored in the first and second devices. In this situation, a half of the blue part needs to be transferred to the second device while a half of the red part needs to be transferred to the first device. The cost equals to the max value between them because both transfers happen simultaneously. More generally for the typical operators in DNNs, Q_{redist} is computed from the shape of the tensor.

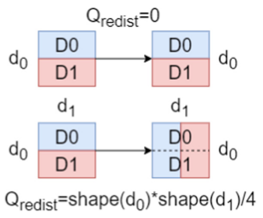


Fig. 3. Redistribution cost (Color figure online)

3 Flex-Edge Recursive Graph

Choosing the optimal strategy for a DNN necessitates to consider all the operators together. For each operator, the redistribution cost depends on the parallelization strategies of its connected operators. It leads to an exponential searching complexity with regard to the number of operators. We propose a topology-independent graph structure named Flex-Edge Recursive Graph (FER Graph) and a traversal order to avoid backtracking on the graph traversal in this section.

3.1 Preliminary Definitions

An operator in the computation graph is defined as *Op*. Each *Op* has its type *Op.Type*. The shape of *Op* is defined by $Op.Shape = [(d_i \in Op.Type.\mathcal{D}_P : int) \mid 0 \le i \le m-1]$ where m is the number of dimension names. Also taking *MatMul* as an example, if its shape is $[(i : 10), (k : 20), (j : 30)]$, the operator computes the product of a 10×20 matrix by a 20×30 matrix.

Definition 1. *The partition strategy of an operator is defined by*

$$Op.Strategy = [d_j \mid d_j \in Op.Type.\mathcal{D}_P]$$

Op.Strategy is a sequence of dimension names that indicates that the operator is partitioned firstly in its d_0 dimension, secondly in the d_1 dimension, and so on. The dimension names in *Op.Strategy* are not necessarily different.

Definition 2. *A computation graph is defined as $G = (V, E)$ where V is a set of Vertices and E is a set of Edges. A vertex $v \in V$ is a tuple $(Op.Type, Op.Shape, Op.Strategy)$. An edge e is defined as a tuple (v_1, v_2, i_1, i_2) where $v_1, v_2 \in V$ and $i_1, i_2 \in \mathbb{N}$. It means that the i_2 input tensor of the operator of v_2 corresponds to the i_1 output tensor of the operator of v_1.*

As an example, *MatMul* with two input matrices whose shapes are respectively *e.g.*, 20×30 and 30×40. It can be represented as such vertex in the computation graph: $(Op.Type : MatMul, Op.Shape : [(i : 20)(k : 40)(j : 30)]$, $Op.Strategy : \emptyset)$. Note that the MatMul operator has three PPDs $\{i, k, j\}$. *Op.Strategy* is empty at the beginning of the algorithm and will be filled with chosen parallelization strategies by the double recursive algorithm discussed in Sect. 4. A $Strategy = [i, i, i, k]$ indicates that the strategies chosen for this operator are along i dimension three times first and then along k dimension once.

3.2 FER Graph

Suppose $G = (V, E)$ a computation graph of a DNN to be partitioned.

Definition 3. *Let σ denote an order on the vertices $v \in V$ such that $\sigma_i(V) \in V$ is the i^{th} visited vertex. From σ order, the Flex-Edge Recursive Graph (FER Graph) G_f can be redefined as*

$$G_f = (\sigma(V), E)$$

Associated with the FER graph G_f a list of sub-graphs is defined in order to establish a traversal rule. This list is built thanks to a concatenation operator denoted $+\!\!+$.

Definition 4. *Let the FER graph $G_f = (\sigma(V), E)$, the list of sub-graphs $\mathcal{G} = [G_f^i = (\sigma(V_i), E_i)]$ is defined as $G_f^0 = (<>, \{\})$ and $G_f^i = G_f^{i-1} +\!\!+ \sigma_i(V)$ with*

$$\left| \begin{array}{c} V_i = V_{i-1} \cup \sigma_i(V) \\ E_i = E_{i-1} \cup \bar{E}_i \\ \bar{E}_i = \{e_j \in E \mid j < i, e_j = (\sigma_i(V), \sigma_j(V), k_1, k_2)\} \\ \cup \{e_j \in E \mid j < i, e_j = (\sigma_j(V), \sigma_i(V), k_1, k_2)\} \end{array} \right.$$

Figure 4 illustrates an example where the upper left corner is a FER Graph with ordered vertices $< v_1, v_4, v_2, v_3 >$. The traversal is a process of reconstructing the original FER Graph from an empty one. The vertex v_1 is added first. None of the other vertices connected to $v1$ is added, so no edge is added to G_f^1, then vertex v_4 is added. Similarly, no edge is together visited with v_4. When v_2 is added, their neighbors v_1 and v_4 are already in the graph. Therefore, e_1, e_3 are added with v_2. After all the concatenations, $\mathcal{G} = [(<>, \{\}), (< v_1 >, \{\}), (< v_1, v_4 >, \{\}), (< v_1, v_4, v_2 >, \{e_1, e_3\}), G)]$.

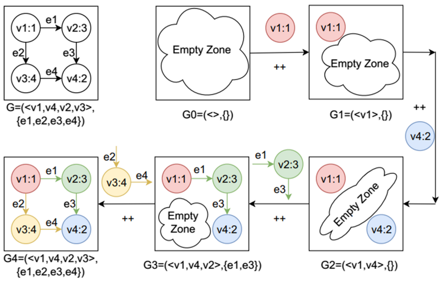


Fig. 4. Traversal of Flex-Edge graph (the number after the colon denotes the order of the vertex. $v2 : 3$ means $v2$ is ordered at the third place).

3.3 Traversing Order

Before discussing the traversal order, we first define the *minCost* function to compare the quality of strategies and to choose the optimal one. The function takes a vertex and its together visiting edges as inputs, then searches the possible partition dimensions and finds the optimal strategy which minimizes the communication cost.

Let $G_f^{i-1}, G_f^i \in \mathcal{G}$ such that $G_f^i = G_f^{i-1} \mathbin{+\!\!+} \sigma_i(V) = (\sigma(V_i), E_i)$, let *op* the operator associated to $\sigma_i(V)$. We defined the cost function for an operator: $Cost(d, \sigma_i(V), \bar{E}_i) = Q_{op}(d) + \sum_{e \in \bar{E}_i} Q_{redist}(e, d)$.

Function $minCost(\sigma_i(V), \bar{E}_i)$ returns the chosen strategy d_r which minimizes the cost function s.t. $Cost(d_r, \sigma_i(V), \bar{E}_i) = \text{MIN}_{d \in \sigma_i(V).Type.\mathcal{D}_P}$ $Cost(d, \sigma_i(V), \bar{E}_i)$.

The idea of our traversal order is to find the optimal strategy for the new sub-graph G_f^i when concatenating a vertex $\sigma_i(V)$ to a sub-graph G_f^{i-1}. So that by finding the optimal strategy for every sub-Graph recursively, we can ensure the optimal strategy for the whole graph.

For a vertex, d_{opmin} denotes a dimension in $\mathcal{D}_P$, such that $Q_{op}(d_{opmin}) = \text{MIN}_{d' \in \mathcal{D}_P} Q_{op}(d')$. If there is no Q_{redist} cost between $\sigma_i(V)$ and G_f^{i-1}, the optimal strategy of G_f^i is the union of the optimal strategy of G_f^{i-1} and the d_{opmin} of $\sigma_i(V)$. However, if Q_{redist} is large, either $\sigma_i(V)$ or G_f^{i-1} needs to change its strategy. In order to avoid backtracking, we define the order $\sigma(V)$ to ensure that it is always the strategy of $\sigma_i(V)$ that needs to be changed. This change of strategy is referred as a *compromise*. Recall that Q_{redist} is either 0 or a fixed positive value. The *compromise* consists in changing the d_{opmin} to a strategy d_{redist} *s.t.* $Q_{redist} = 0$. In this way, the price of reducing an operator's Q_{redist} to zero is the increment of its Q_{op}. Therefore, the *compromise price* of an operator (i.e., the price to change the strategy of an operator) is defined as $\gamma_{\sigma_i(V)} = Q_{op}(d_{redist}) - Q_{op}(d_{opmin})$. The order $\sigma(V)$ of the operators is in descending order of their *compromise price* γ_{op}.

Definition 5. *Let $G_f = (\sigma(V), E)$ a FER graph such that the number of V is n, such that*

$$\forall\ 0 \le j < k \le n,\ \sigma_j(V) \textit{ is ordered before } \sigma_k(V) \textit{ if } \gamma_{\sigma_j(V)} < \gamma_{\sigma_k(V)}$$

The list of sub-graphs of G_f is referred as $\mathcal{G} = [G_f^i = (\sigma(V_i), E_i)]$.

We define *compromise price* of the sub-graph G_f^{i-1} as $\gamma_{G_f^{i-1}}$. It is obvious that $\gamma_{G_f^{i-1}} \ge \gamma_{\sigma_{i-1}(V)} \ge \gamma_{\sigma_i(V)}$. As a result, if we can order the vertices in descending order according to its *compromise price*, the minimized communication cost can be guaranteed.

However, it is not trivial to find the d_{redist} because Q_{redist} relies on the connected vertices. It seems that we return back to the original complexity problem, but the features of DNN help us to handle it. Actually, what we really need is the value of $Q_{op}(d_{redist})$ instead of d_{redist}. For typical operators, we can find their *compromise price* γ because of the characteristics of their semantics.

MatMul OP. MatMul has three PPDs $\{i, j, k\}$. It needs to *compromise* when its d_{opmin} leads to a large Q_{redist}. However, no matter d_{opmin} is i, j or k, when it *compromises* to the other two dimensions, Q_{redist} becomes 0.

The *compromise price* of MatMul is defined as $\gamma = min(Q_{op}(d_0), Q_{op}(d_1)) - Q_{op}(d_{opmin})$. d_0,d_1 are defined as the other two PPDs except d_{opmin}.

Conv OP. Although Conv has many possible partition dimensions, in current real Convolution Neural Networks (e.g., VGG [10], ResNet [4]), only batch dimension b and input channel dimension k will be chosen to cut. The reason is that in a DNN, the size of the kernels is very small so that partitioning kernel tensor usually leads to a super large communication cost. Besides, the channel number increases from input to output of DNN, so that the size of the output tensor is always much bigger than the input.

As there remain only two possible partition dimensions, let d_0 denotes the other dimension except d_{opmin}. The *compromise price* is defined as $\gamma = Q_{op}(d_0) - Q_{op}(d_{opmin})$.

Elementwise OP. Q_{op} of Elementwise OP is always 0, it is evident they do not have *compromise price*. When an Elementwise OP is located between two operators who have Q_{redist} between them, it will hide Q_{redist} between the two neighbors. However, it is not true since the Elementwise OP cannot be adapted to both neighbor operators. To avoid this problem, Elementwise OP are eliminated before the strategy searching. They will reuse one of neighbor's strategy.

Other OP. Except MatMul, Conv, and Elementwise OP, all the other operators (MaxPool, ReduceMean, ReduceSum, ReduceMax, Squeeze... etc.), we noticed in the real DNNs, may have multiple dimensions but they only have two values of Q_{op}. In other words, Q_{op} of several dimensions has the same value. Let d_0 denotes the dimension which has a different Q_{op} as d_{opmin}. The *compromise price* is defined as $\gamma = Q_{op}(d_0) - Q_{op}(d_{opmin})$.

4 Double Recursive Algorithm

Algorithm 1 describes D-Rec composed of Inner Recursion and Outer Recursion. The traversing of FER Graph is called Inner Recursion which takes charge of choosing a dimension in each vertex to partition it into two parts while Outer Recursion is responsible for extending this 2-part partitioning to all devices.

Outer Recursion takes a FER Graph G_f with an empty strategy and the number of partition times N as inputs and returns the strategy assigned Graph as the output. The initial N is obtained from the number of devices. The function *Reorder* sorts the vertex in FER Graph G_f according to the *compromise price* (see Sect. 3.3). At each Outer Recursion step, all the operators in the graph are partitioned into two parts with Inner Recursion. The function *ShapeUpdate* updates the *Shape* of each *Vertex* in G_f according to the chosen *Strategy*. N is decreased by one at each recursion step. Outer Recursion ends when $N = 0$.

Inner Recursion takes the sub-graph list $\mathcal{G}$ and an empty FER Graph G_{f_in} as inputs at each Outer Recursion step. *pop_end*() denotes the operation on $\mathcal{G}$

that pops the last graph in the list: $G = pop\_end(\mathcal{G}), \mathcal{G} \leftarrow \mathcal{G} - G$. In Algorithm 1, v_G denotes the visited vertex to construct G from its predecessor and $\bar{E_G}$ denotes the added new edges. At each step of Inner Recursion, a sub-graph G is popped, and the strategy of its vertices will be chosen by $minCost(v_G, \bar{E_G})$ according to the symbolic cost model. The reconstructed Graph G'_{f_in} is composed by concatenating the strategy updated vertex v_G. The process is recursively applied on the sub-graph list $\mathcal{G}$. The recursion ends when all vertices have been visited.

Algorithm 1. Double-Recursive Algorithm

```
Input: FER Graph $G_f$ whose Strategy is empty. The number of partition times $N$.
Output: FER Graph $G_f$ with chosen strategy.
 1: function OuterRecursion($G_f, N$)
 2:     if $N = 0$ then
 3:         return $G_f$
 4:     else
 5:         $(\sigma, \mathcal{G}) = Reorder(G_f)$
 6:         $G_{f\_in}$ = InnerRecursion($\mathcal{G}, (\varnothing, \varnothing)$)
 7:         $G_f^{'} = ShapeUpdate(G_{f\_in})$
 8:         return OuterRecursion($G_f^{'}, N - 1$)
 9:     end if
10: end function
11:
12: function InnerRecursion($\mathcal{G}, G_{f\_in}$)
13:     if $\mathcal{G} = \varnothing$ then
14:         return $G_{f\_in}$
15:     else
16:         $G = pop\_end(\mathcal{G})$
17:         $d_r = minCost(v_G, \bar{E_G})$
18:         $v_G.OP.Strategy + [d_r]$
19:         $G'_{f\_in} = G_{f\_in} \mathbin{+\!\!+} v_G$
20:         return InnerRecursion($\mathcal{G}, G'_{f\_in}$)
21:     end if
22: end function
```

5 Experiments

This section aims at evaluating the searching efficiency of D-Rec and the quality of the found strategy. The accuracy and training loss of the DNN are not discussed because our approach does not change the semantics of the DNN. These two metrics remain the same as training on a single node.

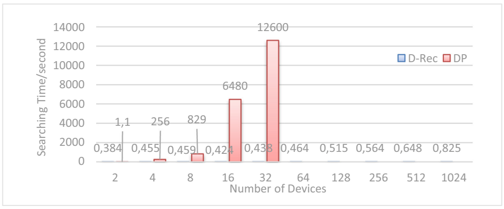
5.1 Environment Setup

The experiments in this section were run on either an Atlas 900 AI cluster [23] or a GPU cluster. Each node of the Atlas cluster is composed of two ARM CPUs and eight Huawei Ascend910 accelerators. Each Ascend 910 accelerator is equipped with a network module, and all Ascend 910 accelerators are interconnected directly even from a different node. Each node of the GPU cluster is composed of two Intel Xeon E5-2680 CPUs and eight NVIDIA V100 GPUs. All GPUs of a node communicate with each other via the PCIe (e.g., Fig. 1). Our D-Rec was run on CPU, and the DNN training was run on accelerators. We used MindSpore[1] as the DNN training platform to implement our proposal[2]. We also implemented a dynamic programming (DP) algorithm of OptCNN [11] to compare with. The Imagenet dataset[3] was used to train image classification DNNs like ResNet and VGG.

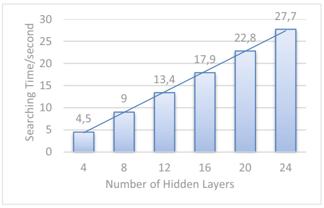
5.2 Searching Efficiency

We took ResNet101 [4] and BERT [5], two representative DNNs, to validate the strategy searching speed of D-Rec. The computation graph of ResNet101 was fixed, and we varied the number of devices from 2 to 1024 (Fig. 5(a)). The searching time of D-Rec on ResNet101 increases linearly from 0.383 s to 0.825 s. DP took nearly 2 h to find a strategy for 16 devices, 3.5 h for 32 devices and failed to find any strategy for 64 devices after hours.

We then fixed the number of devices to 8 and varied the number of hidden layers of BERT from 4 to 24 (Fig. 5(b)) since the number of operators in proportion to the number of hidden layers. The searching time of D-Rec on the variants of BERT is between 4.5 s and 27.7 s. DP does not work on these multi-input graph networks. The experiments showed that D-Rec could handle general large computation graphs in few seconds with a linear growth trend.



(a) Searching Time of ResNet101



(b) Searching Time of BERT

Fig. 5. Training efficiency

5.3 Strategy Quality

Training throughput, often defined as the capacity of processing *Images Per Second* (IPS), is used to evaluate the quality of a parallelization strategy. DP is

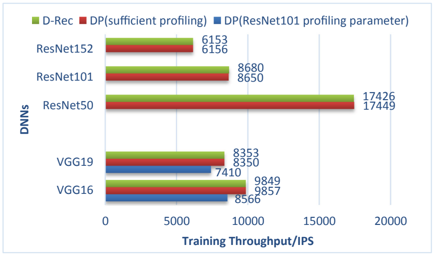
[1] https://www.mindspore.cn/en.

[2] https://github.com/mindspore-ai/mindspore/tree/master/mindspore/ccsrc/frontend/parallel/auto_parallel/rec_core.

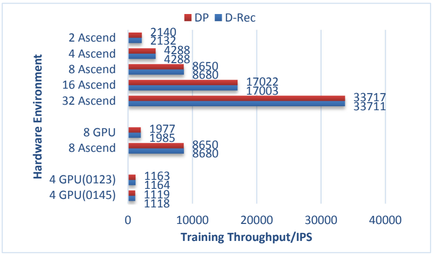
[3] http://image-net.org/.

used as the benchmark because with sufficient profiling its result can be regarded as the state of the art.

The IPS of VGG16, VGG19, ResNet50, ResNet101, and ResNet152 were similar between the parallelization strategies generated by D-Rec and by sufficient-profiled DP (Fig. 6(a)). It validates the quality of the parallelization strategy generated by D-Rec for different DNNs. However, the strategies generated by insufficient-profiled DP on VGG led worse IPS (blue bars in Fig. 6(a)). Thanks to our symbolic approach, D-Rec does not rely on such time-consuming profiling that DP requires.



(a) on Different DNNs



(b) on Different Hardware

Fig. 6. Strategy quality (Color figure online)

We then fixed the DNN model as ResNet101 and varied the architecture of the training machine. We first varied the number of Ascend 910 accelerators from 2 to 32. Then we used the GPU cluster to compare with the Atlas cluster. Lastly we varied the communication topology on the GPU cluster (Fig. 6(b)). In all the above cases, D-Rec obtained a similar IPS as DP. The experiments consistently validate the strategy quality of D-Rec. It shows that our 2-part partitioning recursion on symmetric architectures could eliminate the communication capacity g without impact on the strategy quality.

We observed from Fig. 6(b) that by increasing the number of devices, IPS increases while the training time decreases. However, Fig. 5 shows that using more training devices makes strategy searching slower. The searching time may thus overcome the training time. Thanks to the efficiency of D-Rec, DNNs can be trained on large clusters without such issues.

6 Conclusion

We presented a symbolic cost analysis with FER Graph and D-Rec to generate a parallelization strategy of DNN training. The FER Graph data structure and its traversal ordering successfully guarantee the quality of generated parallelization strategy. Meanwhile, D-Rec reduces the searching complexity dramatically from exponential (i.e., OptCNN [11]) down to linear while preserving the parallelization strategy quality with FER Graph. Our experiments validate our claims and show that the optimal parallelization strategies can be generated in seconds. Not only CNNs but also general large DNNs can now be trained efficiently in parallel.

Our symbolic cost analysis could be used to discover better parallel algorithms for DNN training. The main limitation of our approach is that we do not consider inter-layer partitioning (e.g., pipeline parallelism). So we may obtain sub-optimal strategies for very large natural language processing networks like GPT-3 [24]. Extending our symbolic cost analysis for pipeline parallelism is planned for future work. It could also be extended to exploit new possibilities to accelerate DNN computing such as operator fusion in the future. Further studies to find out the way to cover heterogeneous architectures are desirable too.

References

1. Abadi, M., Barham, P., Chen, J., et al.: Tensorflow: a system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016), Savannah, GA, November 2016, pp. 265–283. USENIX Association (2016)
2. Paszke, A., et al.: Pytorch: an imperative style, high-performance deep learning library. In: Advances in Neural Information Processing Systems, pp. 8026–8037 (2019)
3. MindSpore. https://www.mindspore.cn/
4. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770–778 (2016)
5. Devlin, J., Chang, M.-W., Lee, K., Toutanova, K.: BERT: pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Minneapolis, Minnesota, June 2019, vol. 1 (Long and Short Papers), pp. 4171–4186. Association for Computational Linguistics (2019)
6. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems, pp. 1097–1105 (2012)
7. Dean, J., Corrado, G.S., Monga, R., et al.: Large scale distributed deep networks. In: Proceedings of the 25th International Conference on Neural Information Processing Systems - vol. 1, NIPS 2012, pp. 1223–1231, Red Hook, NY, USA. Curran Associates Inc (2012)
8. Krizhevsky, A.: One weird trick for parallelizing convolutional neural networks. arXiv preprint arXiv:1404.5997, 2014
9. Shazeer, N., Cheng, Y., Parmar, N., et al.: Mesh-tensorflow: deep learning for supercomputers. In: Advances in Neural Information Processing Systems, pp. 10414–10423 (2018)
10. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. In: Bengio, Y., LeCun, Y. (eds.), 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, 7–9 May 2015, Conference Track Proceedings (2015)

11. Jia, Z., Lin, S., Qi, C.R., Aiken, A.: Exploring hidden dimensions in accelerating convolutional neural networks. In: Volume 80 of Proceedings of Machine Learning Research, PMLR, 10–15 Jul 2018, pp. 2274–2283 (2018)
12. Jia, Z., Zaharia, M., Aiken, A.: Beyond data and model parallelism for deep neural networks. In: Talwalkar, A., Smith, V., Zaharia, M. (eds.) Proceedings of Machine Learning and Systems, vol. 1, pp. 1–13 (2019)
13. Li, C., Hains, G.: SGL: towards a bridging model for heterogeneous hierarchical platforms. Int. J. High Perform. Comput. Netw. **7**(2), 139–151 (2012)
14. Dongarra, J.: Report on the Fujitsu Fugaku system. University of Tennessee-Knoxville Innovative Computing Laboratory, Tech. Rep. ICLUT-20-06-2020 (2020)
15. Yuksel, A., et al.: Thermal and mechanical design of the fastest supercomputer of the world in cognitive systems: IBM POWER AC 922. In: ASME 2019 InterPACK (2019)
16. Liao, H., Tu, J., Xia, J., Zhou, X.: Davinci: a scalable architecture for neural network computing. In: 2019 IEEE Hot Chips 31 Symposium (HCS), pp. 1–44. IEEE (2019)
17. Nvidia. NVIDIA DGX-1 System Architecture White Paper (2017)
18. Valiant, L.G.: A bridging model for multi-core computing. J. Comput. Syst. Sci. **77**(1), 154–166 (2011)
19. Li, A., et al.: Evaluating modern GPU interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. IEEE TPDS **31**(1), 94–110 (2019)
20. Zhang, H., Zheng, Z., Xu, S., et al.: Poseidon: an efficient communication architecture for distributed deep learning on GPU clusters. In: 2017 USENIX Annual Technical Conference (USENIX ATC 17), pp. 181–193 (2017)
21. Sergeev, A., Balso, M.D.: Horovod: fast and easy distributed deep learning in tensorflow. arXiv preprint arXiv:1802.05799 (2018)
22. Chilimbi, T., Suzue, Y., Apacible, J., Kalyanaraman, K.: Project adam: building an efficient and scalable deep learning training system. In Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI 2014, USA, pp. 571–582. USENIX Association (2014)
23. Atlas900. https://e.huawei.com/en/products/cloud-computing-dc/atlas/atlas-900-ai
24. Brown, T., Mann, B., Ryder, N.: et al.: Language models are few-shot learners. In: Advances in Neural Information Processing Systems, vol. 33, pp. 1877–1901. Curran Associates Inc (2020)