HyPar: Towards Hybrid Parallelism for Deep Learning Accelerator Array

Linghao Song[†], Jiachen Mao[†], Youwei Zhuo[‡], Xuehai Qian[‡], Hai Li[†], Yiran Chen[†]
[†]*Duke University*, [‡]*University of Southern California*
{linghao.song, jiachen.mao, hai.li, yiran.chen}@duke.edu, {youweizh, xuehai.qian}@usc.edu

ABSTRACT

With the rise of artificial intelligence in recent years, Deep Neural Networks (DNNs) have been widely used in many domains. To achieve high performance and energy efficiency, hardware acceleration (especially inference) of DNNs is intensively studied both in academia and industry. However, we still face two challenges: large DNN models and datasets, which incur frequent off-chip memory accesses; and the training of DNNs, which is not well-explored in recent accelerator designs. To truly provide high throughput and energy efficient acceleration for the training of deep and large models, we inevitably need to use multiple accelerators to explore the coarse-grain parallelism, compared to the fine-grain parallelism inside a layer considered in most of the existing architectures. It poses the key research question to seek the best organization of computation and dataflow among accelerators.

In this paper, we propose a solution HYPAR to determine layer-wise parallelism for deep neural network training with an array of DNN accelerators. HYPAR partitions the feature map tensors (input and output), the kernel tensors, the gradient tensors, and the error tensors for the DNN accelerators. A partition constitutes the choice of parallelism for weighted layers. The optimization target is to search a partition that minimizes the total communication during training a complete DNN. To solve this problem, we propose a communication model to explain the source and amount of communications. Then, we use a hierarchical layer-wise dynamic programming method to search for the partition for each layer. HYPAR is practical: the time complexity for the partition search in HYPAR is *linear*. We apply this method in an HMC-based DNN training architecture to minimize data movement. We evaluate HYPAR with ten DNN models from classic Lenet to large-size model VGGs, and the number of weighted layers of these models range from four to nineteen. Our evaluation finds that: the default Model Parallelism is indeed the worst; the default Data Parallelism is not the best; but hybrid parallelism can be better than either the default Data Parallelism or Model Parallelism in DNN training with an array of accelerators. Our evaluation shows that HYPAR achieves a performance gain of 3.39× and an energy efficiency gain of 1.51× compared to Data Parallelism on average, and HYPAR performs up to 2.40× better than "one weird trick".

1. INTRODUCTION

With the rise of artificial intelligence in recent years, Deep Neural Networks (DNNs) have been widely used because of their high accuracy, excellent scalability, and self-adaptiveness properties [1,2]. Many applications employ DNNs as the core technology, such as face detection [3], speech recognition [4], scene parsing [5].

DNNs are computation and memory intensive and pose intensive challenges to the conventional Von Neumann architecture where computation and data storage are separated. For example, AlexNet [6] performs 10^9 operations in processing just one image data. During processing, a large amount of data movements are incurred due to the large number of layers and millions of weights. Such data movements quickly become a performance bottleneck due to limited memory bandwidth and more importantly, an energy bottleneck. A recent study [7] showed that data movements between CPUs and off-chip memory consumes two orders of magnitude more energy than a floating point operations. Clearly, the cost of computation and data movement are serious challenges for DNNs.

To achieve high performance and energy efficiency, hardware acceleration of DNNs is intensively studied both in academia [8–90] and industry [91–101]. In particular, several major companies developed *1)* DNN accelerators, e.g., Google TPU [91,92], and neuro-processors, e.g., IBM TrueNorth [93–95]; *2)* corresponding standards, architectures, and platforms [96,98–100]. In academia, Eyeriss [15] is a representative design of spatial architecture to coordinate dataflow between processing engines (PEs). Neurocube [19] takes the advantage of in-memory processing by deploying PEs in hybrid memory cubes (HMCs) [102] with a programmable data packet scheme. Flexflow [20] is a systolic architecture with tiling optimization. Furthermore, inter-layer data flow for DNN acceleration are considered in [39,42,103].

Despite the explosion of neural network accelerators, there are still fundamental challenges. First, with large model size (e.g., for ImageNet dataset [104]), accelerators suffer from the frequent access to off-chip DRAM, which consume significant energy, e.g., 200× compared to on-chip SRAM [15,17], and can thus easily dominate the whole system power consumption. Second, almost all of the recently proposed DNN accelerators only focuses on DNN inference.

The *inference* of deep neural networks is a forward progress

2378-203X/19/$31.00 ©2019 IEEE
DOI 10.1109/HPCA.2019.00027

of input images from the first layer to the last layer. Kernels (weights) of a network are obtained through training before the inference. The computations are performed one layer after another. The *training* of deep neural networks is more complex than inference due to more computations and additional data dependencies. Besides data forward, error backward and gradient computation are two additional computation steps to generate new weights in training. With the increasing importance of deep learning, we argue that DNN training acceleration is a crucial problem. Currently, DNNs are typically trained by high-performance computer systems with high-end CPUs/GPUs, which are not performance and energy efficient. While many accelerators focused on acceleration for DNN inference, training was only considered in a few existing accelerators [19, 103] in restricted manner.

To truly provide high throughput and energy efficient acceleration for training deep and large models [105, 106], we inevitably need to use *multiple accelerators* as a general architecture to explore the coarse-grain parallelism, compared to the fine-grain parallelism inside a layer considered in the existing research [15, 19, 20, 92]. It poses the key research question to seek the best organization of computation and dataflow among accelerators.

The problem is challenging due to the complex interactions between the type of parallelism and different layers. Assume we have N accelerators, in *data parallelism*, a batch of data is partitioned into N parts, while the model (weights) are duplicated N times. Each accelerator holds one part of the partitioned data and a complete copy of the model. It incurs no communication in data forward and error backward. To update the weights, each accelerator requires remote access to the gradient in the other accelerators, thereby results in communication. In *model parallelism*, the kernel is partitioned into N parts, and feature maps are partitioned accordingly. Opposite to data parallelism, it incurs communications in data forward but no communication in error backward and weight updating. More details are discussed in Section 2.2.

The current accelerator designs *do not provide a good answer* on how to determine parallelism for multiple accelerators, because they focus on the acceleration of intra-layer (fine-grain parallelism) computation and assume that the needed data are already in memory. Clearly, the solution is nontrivial. We cannot simply partition all layers in data parallelism or model parallelism manner, because different layer types imply different choices. Moreover, networks also have various connection between different layers. Overall, it can be phrased as a complex optimization problem.

To systematically solve this problem, we propose a solution HyPar to determine layer-wise parallelism for deep neural network training with an array of DNN accelerators. The goal of HyPar is to partition the feature map tensors (input and output), the kernel tensors, the gradient tensors, and the error tensors among the DNN accelerators. A partition is determined by the choice of parallelism for all weighted layers. The optimization target is to search a partition that minimizes the total amount of communication during training a complete deep neural network. In HyPar, we propose a *communication model* to explain where the communication comes from in partitioned tensors (of feature maps and kernels), and determine the amount of communication. Then we uses a hierarchical layer-wise dynamic programming method to search for the partition for each layer. We show that HyPar is practical: the time complexity for the partition search in HyPar is *linear*. We apply this method in an HMC-based DNN training architecture with an array of sixteen accelerators organized in four hierarchical levels and minimize the data movement.

Based on the architecture, we evaluate HyPar with ten DNN models from classic Lenet to large-size model VGGs, and the number of weighted layers of these models ranges from four to nineteen. Our evaluation shows that: the default model parallelism is indeed worst; the default data parallelism is also not the best; but hybrid parallelism can be better than either the default data parallelism or model parallelism in DNN training with an array of accelerators. Our evaluation shows that HyPar achieves a performance gain of $3.39\times$ and a energy efficiency gain of $1.51\times$ compared to data parallelism on average. In addition, we also study the scalability and the effects of network topology to provide deeper understanding of the HyPar architecture.

This paper is organized as follows. Section 2 introduces DNN background, parallelism in DNN computation, DNN accelerators and our motivations. Section 3 proposes a communication model to quantify the communication in DNN computation. Section 4 proposes the partition algorithm based on the communication model to determines the parallelism for each layer in training. Section 5 presents an HMC-based accelerator array in which the partitions are generated by HyPar. Section 6 evaluates the performance, energy efficiency and communication of HyPar architecture in DNN training, and conducts studies to provide further insights. Section 7 concludes the paper.

2. BACKGROUND AND MOTIVATION

2.1 Inference and Training of DNNs

The *inference* of deep neural networks is a forward progress of input data (typically images) from the first layer to the last layer. Kernels (weights) of a network are obtained through training before the inference. Images are typically grouped into batches to ensure high throughput with efficient forward propagation. The computations are performed one layer after another. For a convolutional layer l, we use $\mathbf{F}_l$ to represent feature maps of this layer, and use B to denote the batch size. We assume that each feature map is a three-dimensional tensor, with a height of H, a weight of W and a depth of C_l (C_l is also the number of channels of Layer l). The size of the feature map slice is $[H_l \times W_l \times C_l]$. Thus, $\mathbf{F}_l$ is of size $B \times [H_l \times W_l \times C_l]$. The kernel $\mathbf{W}_l$ has a size of $[K \times K \times C_l] \times C_{l+1}$, where K is the height/width of kernels and C_{l+1} is the number of channels of next layer, Layer $l+1$. $f(\cdot)$ is an activation function, performing element-wise non-linear operations. We use $\otimes$ to denote convolutions. The inference (forward propagation) can be represented as,

$$\mathbf{F}_{l+1} = f(\mathbf{F}_l \otimes \mathbf{W}_l) \tag{1}$$

The *training* of deep neural networks is more complex than inference. The purpose of training is to tune the kernels to reduce the value of a loss function. The loss function computes the difference between the output of a neural network with the ground truth (i.e., labels) of a group of input images. L2-norm and softmax are two examples of loss functions. Besides forward, *error backward* and *gradient computation* are two additional computation steps to generate new weights in training. We use $\mathbf{E}_l$ to represent errors for Layer l, the error backward can be represented as Equation 2,

$$\mathbf{E}_l = (\mathbf{E}_{l+1} \otimes \mathbf{W}_l^*) \odot f'(\mathbf{F}_l) \tag{2}$$

where $\mathbf{W}^*$ is a reordered form of $\mathbf{W}$ (if $\mathbf{W}$ is a matrix then $\mathbf{W}^* = \mathbf{W}^\top$), $\odot$ is an element-wise multiplication and $f'(\cdot)$ is the derivative of $f(\cdot)$. The gradient computation is,

$$\triangle \mathbf{W}_l = \mathbf{F}_l^* \otimes \mathbf{E}_{l+1} \tag{3}$$

with $\triangle \mathbf{W}_l$, we can update $\mathbf{W}_l$.

2.2 Parallelisms

For DNNs training, the training data samples (images) are grouped into batches. For each epoch, a batch of data need to perform forward, error backward and gradient computation. Since training requires more computations, it is typically conducted with multiple DNN accelerators. In this context, parallelism needs to be considered among the accelerators. *Data Parallelism* [107,108] and *Model Parallelism* [109,110] are the two types of parallelism used in DNN training. In Data Parallelism, all accelerators hold a copy of model, but data (training samples) are partitioned into parts and each accelerator processes one part. In Model Parallelism, all accelerators process on the same data (training samples), but the whole model is partitioned and each accelerator holds a part of the model.

Whether to use Data Parallelism or Model Parallelism is currently determined empirically. For neural networks with rich convolutions, Data Parallelism is employed, while for neural networks with large model size, Model Parallelism is employed. Thus for the training of deep learning (which usually contains a lot of convolutions), Data Parallelism is the default setting [107,108]. Krizhevsky proposed "one weird trick" [111] to outperform the default Data Parallelism, where convolutional layers are configured with data parallelism and fully connected layers are configured with model parallelism. It is called "weird" because that method works but why is works was not known. We will show that trick is not "weird" with our communication model (Section 3), and HYPAR even has higher performance.

2.3 DNN Accelerators

Many DNN accelerators were proposed to optimize the data flow to improve performance and energy efficiency. Eyeriss [14,15] is a representative design which employs a spatial data flow to share data between processing engines (PEs). Neurocube [19] takes the advantage of in-memory processing by deploying PEs in hybrid memory cubes (HMCs) [102] with a programming data packet scheme. Flexflow [20] is a systolic architecture with tiling optimization. MAESTRO [112] explored even five types of fine-grained on-chip data flow for DNN accelerator. All of these accelerators focus on intra-layer computations in: computations for no more than one layer are performed at one time slot. They are all about the design of a stand-alone accelerator, which is orthogonal to this work.

In comparison, inter-layer data flow for DNN acceleration are considered in [39,42,103]. Alwani *et al.* [39] proposed the fused-layer pipelining for DNN accelerators. Shen *et al.* [42] further optimized the tiling parameters for inter-layers. Song *et al.* [103] proposed an inter-layer accelerator for DNN training.

2.4 Motivation

With the increasing importance of deep learning, we argue that DNN training acceleration is a crucial problem. Currently, DNNs are typically trained by high-performance computer systems with high-end CPUs/GPUs, which is not computation and energy efficient. For this reason, DNN training acceleration is of high interest to the companies with huge amount of data. For example, Google released a new version of TPU [113] for training after a first version of TPU [92] designed for inference. In research community, many accelerators focused on accelerating DNN inference, and training was only considered in a few existing accelerators [19,103] in restricted manner. Neurocube [19] partitions model into HMC vaults, but does not consider the parallelism between HMCs. Among vaults, it assumes fixed parallelism setting for all layers, which may not be the best for all networks. With inter-layer design (model parallelism), Pipelayer [103] performs the computations of different layers simultaneously in different processing units of the accelerator. Pipelayer also used a intra-layer parallelism, which is actually intra-layer data parallelism, to boost performance. However, the details of data movement for intra-layer and inter-layer parallelism was yet to be explored.

To truly support high throughput and energy efficient training acceleration of deeper and large models [105,106], we eventually need to use *multiple accelerators* to explore the coarse-grain parallelism, compared to the fine-grain parallelism inside a layer. It requires a systematic study to seek the best organization of compute and dataflow among an array of accelerators.

This problem is *unsolved* by existing solutions [10,11,15,20]. They only consider the acceleration of intra-layer computation and assumes that the needed data are already in memory. For a stand-alone accelerator that processes a layer separately, it is an acceptable assumption as the focus is the fine-grained computation inside the layer. With an array of accelerators, the input data of an accelerator (for the current layer) are potentially produced by other accelerators (for the previous layer), the computation and dataflow organization affect the *data movement*, which is a critical factor affecting the performance. That motivates us to find a solution to determine tensor partition and dataflow organizations among layers for neural network training with an array of DNN accelerators.

3. COMMUNICATION MODEL

We propose HYPAR to determine layer-wise parallelism for deep neural network training with an array of DNN accelerators. HYPAR partitions the feature map tensors $\mathbf{F}_l$ (input and output) and $\mathbf{F}_{l+1}$, the kernel tensor $\mathbf{W}_l$, the gradient tensor $\triangle \mathbf{W}_l$, and the error tensors $\mathbf{E}_l$ and $\mathbf{E}_{l+1}$ for the DNN accelerators. A partition constitutes the choice of parallelism for all weighted layers. The optimization target is to search a

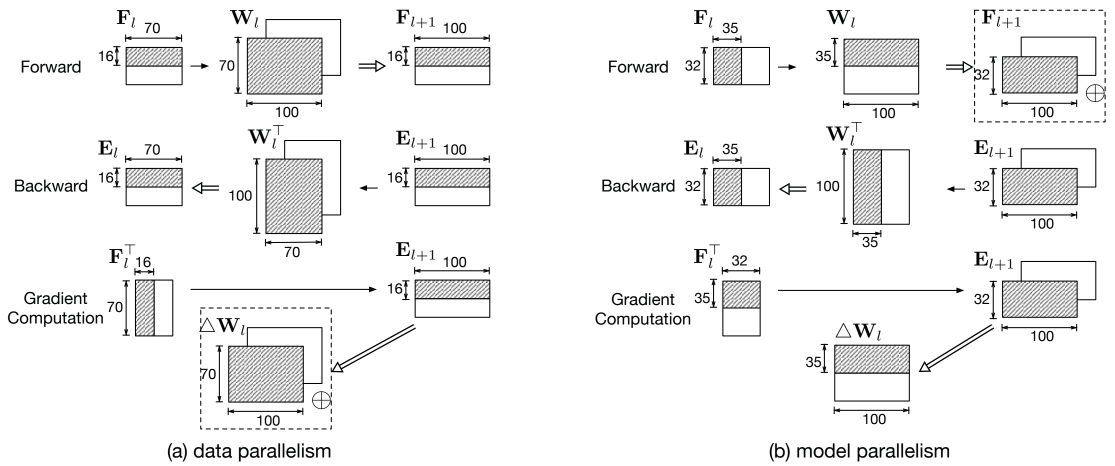

Figure 1: Forward, Backward and Gradient Computation in (a) data parallelism and (b) model parallelism. In data parallelism, intra-layer communication happens in kernel updating, and in model parallelism intra-layer communication happens in computation for output feature map, both marked by a $\oplus$.

partition that minimizes the total communication during training a complete deep neural network. Thus, we first develop a communication model that answers the questions such as: for an accelerator and various parallelism settings, where does the communication come from, and what is the amount of communication?

Before the technical discussion, we clarify some terminologies. When we use *lowercase* "data parallelism", we refer to the case where all accelerators have a copy of kernel (weight) of one specific layer, while feature maps associated with that layer are partitioned. When we use *lowercase* "model parallelism", we refer to the case where the kernels (weights) of one specific layer are partitioned and each accelerator has one partition. We discuss more details about data parallelism and model parallelism in the following Section 3.1. In contrast, we use *Uppercase* "Data Parallelism" and "Model Parallelism" to refer to cases where *all* layers of a neural network are in data parallelism or model parallelism, respectively.

3.1 Two Types of Parallelism

We discuss data parallelism and model parallelism using a concrete example. Assume we have two accelerators, the batch size is $B = 32$. Let us consider a fully-connected layer, where the number of input and output neurons are 70 and 100, respectively. Thus, the feature map $\mathbf{F}_l$ has a size of 32×70, the kernel (weight matrix) has a size of 70×100 and $\mathbf{F}_{l+1}$ has a size of 32×100.

3.1.1 data parallelism

In data parallelism, a batch of data is partitioned into two parts, while the kernels (weight matrix) are duplicated. Each accelerator holds one part of the partitioned data and a complete copy of the kernel.

Figure 1 (a) illustrates the shapes of tensors held by the two accelerators. All of the rectangles with shadow lines are held by one accelerator and all of the white rectangles are held by the other.

In forward, each accelerator performs the computation in Equation 1. Because $f(\cdot)$ is an element-wise operation, which only requires local data in the accelerator itself but does not require remote data from the other accelerator, we focus on the multiplication and represent Equation 1 as $\mathbf{F}_l \rightarrow \mathbf{W}_l \Rightarrow \mathbf{F}_{l+1}$. For the one holding the rectangles with shadow lines, it performs a multiplication with a size of $[16 \times 70] \rightarrow [70 \times 100] \Rightarrow [16 \times 100]$. Since no remote data are required, there is no communication between the two accelerators.

In error backward, the multiplication for each accelerator is $\mathbf{E}_{l+1} \rightarrow \mathbf{W}_l^\top \Rightarrow \mathbf{E}_l$, and the size of matrices in the multiplication is $[16 \times 100] \rightarrow [100 \times 70] \Rightarrow [16 \times 70]$. Still, no communication exists between the two accelerators.

However, the remote data access, which leads to communication, is required in kernel updating. The multiplication is $\mathbf{F}_l^\top \rightarrow \mathbf{E}_{l+1} \Rightarrow \triangle\mathbf{W}_l$, and the size of matrices in the multiplication in gradient computation is $[70 \times 16] \rightarrow [16 \times 100] \Rightarrow [70 \times 100]$. Different accelerators compute the gradient with different half of $\mathbf{E}_{l+1}$ and different half of $\mathbf{F}_l^\top$. Therefore, elements in the computed gradient matrix ($[70 \times 100]$) are just partial sums, and the actual gradient is the summation of the two partial sums from the two accelerators. To update the weights, each accelerator requires remote accesses to the gradient partial sum in the other accelerator, and adds with the local gradient partial sum. We use a $\oplus$ to indicate a remote accesses (and the addition of the partial sum), which incurs communication.

3.1.2 model parallelism

In model parallelism, the kernel is partitioned, and feature maps are partitioned accordingly. In forward, each accelerator performs computation for the matrices $\mathbf{F}_l \rightarrow \mathbf{W}_l \Rightarrow \mathbf{F}_{l+1}$ with sizes of $[32 \times 35] \rightarrow [35 \times 100] \Rightarrow [32 \times 100]$. This scenario is similar to gradient computation in data parallelism. To get the result for $\mathbf{F}_{l+1}$, remote accesses to the partial sum feature maps from the other accelerator is required. In error backward, the multiplication for each accelerator is $\mathbf{E}_{l+1} \rightarrow \mathbf{W}_l^\top \Rightarrow \mathbf{E}_l$, and the size of matrices in the multiplication is $[32 \times 100] \rightarrow [100 \times 35] \Rightarrow [32 \times 35]$. No communication exists between the two accelerators. To compute gradient, the multiplication is $\mathbf{F}_l^\top \rightarrow \mathbf{E}_{l+1} \Rightarrow \triangle\mathbf{W}_l$, and the size of matrices in the multiplication in gradient computation is $[35 \times$

Table 1: Intra-layer communication amount in data parallelism and model parallelism.

data parallelism	model parallelism
$\mathbb{A}(\triangle \mathbf{W}_l)$	$\mathbb{A}(\mathbf{F}_{l+1})$

$32] \rightarrow [32 \times 100] \Rightarrow [35 \times 100]$.

In model parallelism, the gradient computed by one accelerator is the exact gradient needed for updating the kernel held by itself. Therefore, no communication is required. However, the communication between two accelerators happens in the computation for output feature maps in forward, which is marked by a $\oplus$ in Figure 1 (b).

3.2 Intra-Layer Communication

As shown in Figure 1, each layer performs three multiplications for forward, error backward, and gradient computation. In each multiplication, three tensors are involved. Thus, in total nine tensors need to be considered. Notice that $\mathbf{F}_l$ and $\mathbf{F}_l^\top$ are the same, which is also true for $\mathbf{W}_l$ and $\mathbf{W}_l^\top$, $\mathbf{E}_{l+1}$ and $\mathbf{E}_{l+1}^\top$.

For one layer, we call the two tensors on the left hand side of kernel or gradient tensors as *L* tensors; and the two tensors on the right hand side of kernel or gradient tensors as *R* tensors. As shown in Figure 1, $\mathbf{F}_l$ ($\mathbf{F}_l^\top$) and $\mathbf{E}_l$ are *L* tensors, and $\mathbf{F}_{l+1}$ and $\mathbf{E}_{l+1}$ are *R* tensors.

Following the idea of intra layer and inter layer from [103], we decouple the communication into two parts: *1) intra-layer communication* by kernel updates within a layer, marked by a $\oplus$ within Figure 1; and *2) inter-layer communication* by conversions of *L* and *R* tensors of feature maps and errors between layers.

In data parallelism, we can see that communication for kernel updating happens when one accelerator remotely accesses $\triangle \mathbf{W}_l$ in the other to perform partial sum $\oplus$. We use $\mathbb{A}(\triangle \mathbf{W}_l)$ to denote the amount of data in $\triangle \mathbf{W}_l$. Therefore, the intra-layer communication in data parallelism is $\mathbb{A}(\triangle \mathbf{W}_l)$, while the intra-layer communication in model parallelism is 0. In model parallelism, partial sum is to be performed to get $\triangle \mathbf{F}_{l+1}$, so the intra-layer communication is $\mathbb{A}(\mathbf{F}_{l+1})$. The intra-layer communication for data parallelism (dp) and model parallelism (mp) are summarized in Table 1.

3.3 Inter-Layer Communication

Next, we calculate the inter-layer communication due to accessing feature maps and errors. Essentially, we calculate the communication for conversion of *L* and *R* tensors, as shown in Figure 2. Since the parallelism for each layer is either data parallelism or model parallelism, to calculate the communication for each layer, we should consider four cases: dp-dp, dp-mp, mp-mp and mp-dp.

dp-dp In Figure 2 (a), the *R* tensors $\mathbf{F}_{l+1}$ and $\mathbf{E}_{l+1}$ belong to a previous layer, Layer *l*, and *L* tensors belong to Layer $l+1$. For the accelerator which holds the dashed-line tensors, no remote access to the accelerator is necessary because the *R* and *L* tensors have the same shape. Thus, the inter-layer communication in dp-dp is 0.

dp-mp The *R* and *L* tensors in Figure 2 (b) have different shapes, which causes communication between two accelerators. For the accelerator which holds the dashed-line tensors, it needs the *L* tensor $\mathbf{F}_{l+1}$, but the *R* tensor $\mathbf{F}_{l+1}$ held by this accelerator (from the computation it performs on Layer *l*) has a different shape compared to the *L* tensor $\mathbf{F}_{l+1}$. Thus, this accelerator needs to remotely read part of the white *R* tensor $\mathbf{F}_{l+1}$ from the other accelerator. The shape of the communication is the overlaps of the white *R* tensor $\mathbf{F}_{l+1}$ and the dashed-line *L* tensor $\mathbf{F}_{l+1}$, i.e., the black tensor. The black tensor is 1/4 of the tensor $\mathbf{F}_{l+1}$, so the inter-layer communication between *R* $\mathbf{F}_{l+1}$ and *L* $\mathbf{F}_{l+1}$ is $0.25\mathbb{A}(\mathbf{F}_{l+1})$. Similarly, we can calculate and obtain that the inter-layer communication between *R* $\mathbf{E}_{l+1}$ and *L* $\mathbf{E}_{l+1}$ is $0.25\mathbb{A}(\mathbf{E}_{l+1})$. Therefore, the inter-layer communication in dp-mp is $0.25\mathbb{A}(\mathbf{F}_{l+1}) + 0.25\mathbb{A}(\mathbf{E}_{l+1})$.

Table 2: Inter-layer communication amount for the transition of dp-dp, dp-mp, mp-mp and mp-dp.

dp-dp	0
dp-mp	$0.25\mathbb{A}(\mathbf{F}_{l+1}) + 0.25\mathbb{A}(\mathbf{E}_{l+1})$
mp-mp	$0.5\mathbb{A}(\mathbf{E}_{l+1})$
mp-dp	$0.5\mathbb{A}(\mathbf{E}_{l+1})$

mp-mp Because the dashed-line *R* $\mathbf{F}_{l+1}$ already contains the dashed-line *L* $\mathbf{F}_{l+1}$, the communication for $\mathbf{F}_{l+1}$ is 0. But for $\mathbf{E}_{l+1}$, the dashed-line *R* $\mathbf{E}_{l+1}$ requires remote access to obtain the black part, the communication is $0.5\mathbb{A}(\mathbf{E}_{l+1})$.

mp-dp Similar to mp-mp, we can calculate that the inter-layer communication for mp-dp is $0.5\mathbb{A}(\mathbf{E}_{l+1})$.

The results based on the above inter-layer communication calculation are summarized in Table 2.

From Table 1 and Table 2, we can see that, for DNN inference, the best option is Data Parallelism, i.e., data parallelism for every layers. It is because the intra-layer communication is zero since no gradient computation is necessary in inference, and the inter-layer communication of dp-dp is also zero. However, the assumption is different for DNN training, and parallelism becomes a critical concern.

3.4 Parallelism in Training

We use the example in Figure 1 to compare the communication of data parallelism and model parallelism in our communication model. In Figure 1, for a fully-connected layer, we see that the communication amount between the two accelerators in data parallelism is 56KB($= 2 \times 70 \times 100 \times 4$B)

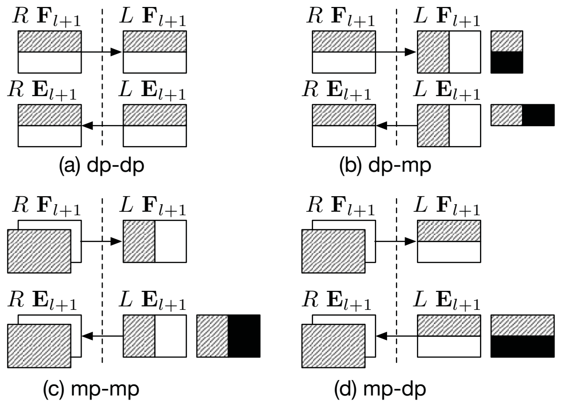

Figure 2: Inter-layer communication for (a) dp-dp, (b) dp-mp, (c) mp-mp and (d) mp-dp.

assuming a precision of 32-bit floating point and a batch size of 32. The communication amount in model parallelism is 25.6KB($= 2 \times 32 \times 100 \times 4$B). In this case, model parallelism is better than data parallelism. However, if we consider computations in a convolutional layer, where the $\mathbf{F}_l$ has a size of $[12 \times 12 \times 20]$, $\mathbf{W}_l$ has a size of $[5 \times 5 \times 20] \times 50$ and $\mathbf{F}_{l+1}$ has a size of $[8 \times 8 \times 50]$, the communication amount in data parallelism is 200KB($= 2 \times 5 \times 5 \times 20 \times 50 \times 4$B), while communication amount in model parallelism is 819KB($= 2 \times 32 \times 8 \times 8 \times 50 \times 4$B). In this different scenario, data parallelism is better than model parallelism. From the simple example, we see that a default approach that choose either data parallelism or model parallelism for all layers in a network would not get the highest performance because both convolutional and fully-connected layers exist in most widely used deep neural networks. That also explains why fully-connected layers are configured with model parallelism and convolutional layers are configured with data parallelism in the empirical "one weird trick" [111], which outperforms default Data Parallelism and Model Parallelism.

However, the empirical configuration in [111] will *not* always work. With our general communication model, we can see the trick [111] only considered intra-layer communication, but did not consider the other communication source, inter-layer communication. We will show that HyPar performs better than the trick in Section 6.5.2.

Realizing this fact, one may naturally choose a more comprehensive approach: choosing a parallelism for each layer and enumerating all possibilities to determine the best choice. Unfortunately, it is not feasible, because the time complexity for such enumeration is $O(2^N)$ for a neural network with N weighted layers. It motivates us to find a more practical partitioning algorithm.

4. LAYER PARTITION

In this section, we discuss HyPar, which partitions the feature map tensors $\mathbf{F}_l$ (input and output) and $\mathbf{F}_{l+1}$, the kernel tensor $\mathbf{W}_l$, the gradient tensor $\triangle\mathbf{W}_l$, and the error tensors $\mathbf{E}_l$ and $\mathbf{E}_{l+1}$ for the DNN accelerators. The parallelism for one layer actually determines the tensor partitioning for two accelerators, as shown in Figure 1. A partition constitutes the choice of parallelism for each layer in a deep neural network. The optimization target is to search a partition that minimizes the total communication during training a complete deep neural network. Rather than $O(2^N)$ brute-force search, HyPar is practical: the time complexity for the partition search in HyPar is *linear*, i.e. $O(N)$ for a neural network with N weighted layers.

4.1 Partition Between Two Accelerators

From Section 3.3, we can see that: *1)* each layer is configured with either data parallelism or model parallelism; *2)* the calculation for inter-layer communication only depends on two adjacent layers; *3)* the intra-layer communication only depends on the parallelism of that layer, but does not depend on any other layer. Thus, to minimize the total amount of communication, we can use a *layer-wise dynamic programming* method to search for the partitions for each layer. The time complexity of this method is $O(N)$ for a neural network with N weighted layers.

Algorithm 1 Partition Between Two Accelerators.

```
Input:
    1. Batch size, $B$,
    2. The number of weighted layers in a DNN model, $L$,
    3. A list of hyper parameters (layer type: conv or fc,
    kernel sizes, parameter for pooling, activation function),
    $HP[l], l = 0,..,L$-1.
Output:
    1. Total communication, $com$,
    2. A list of parallelism for each weighted layer, $P[l], l =$
    $0,..,L$-1.

 1: Generate tensor shapes for $\mathbf{F}_l$, $\mathbf{W}_l$, $\triangle\mathbf{W}_l$, $\mathbf{E}_l$ for each
    layer.
 2: Initiate $com\_dp[0] = 0$, $com\_mp[0] = 0$, $P\_dp = [\ ]$,
    $P\_mp = [\ ]$.
 3: for ($l = 0$; $l < L$; $l$++) do
 4:     Compute $intra\_dp$, $intra\_mp$ using Table 1 and
        $inter\_dp\_dp$, $inter\_mp\_dp$, $inter\_dp\_mp$,
        $inter\_mp\_mp$ using Table 2.
 5:     $com\_dp[l] = \min(com\_dp[l\text{-}1]+inter\_dp\_dp,$
                $com\_mp[l\text{-}1]+inter\_mp\_dp) + intra\_dp$.
 6:     Update $P\_dp$.
 7:     $com\_mp[l] = \min(com\_dp[l\text{-}1]+inter\_dp\_mp,$
                $com\_mp[l\text{-}1]+inter\_mp\_mp)+intra\_mp$.
 8:     Update $P\_mp$.
 9: end for
10: return min($com\_dp[L\text{-}1]$, $com\_mp[L\text{-}1]$)
    and the corresponding parallelism list ($P\_dp$ or $P\_mp$).
```

The idea is to use dynamic programming, for each layer, to compute the intra-layer communication of dp and mp and inter-layer communication of dp-dp, mp-dp, dp-mp, mp-mp using the results in Table 1 and Table 2, and then calculate the minimum *accumulated* communication for data parallelism or model parallelism in this layer.

The pseudocode of the partition algorithm between two accelerators is given in Algorithm 1. The inputs of our partition algorithm are identical to the parameters that are needed for a normal mini batch training process. As shown in the input section of Algorithm 1, the inputs include the batch size (B), the number of model layers (L), and the necessary hyper parameters (layer type: conv or fc, kernel sizes, parameter for pooling, activation function: $HP[l], l = 0,..,L-1$). The outputs of our partition algorithm are composed of the minimal total communication between two accelerators and a list parallelism methods we should chose to realize such minimal communication for each layer in the model.

4.2 Hierarchical Partition

So far, we assume only two accelerators. To expand to partition for an array of accelerators, we use a hierarchical approach.

Although Algorithm 1 performs partitions based on two accelerators, we can view the two accelerators as two groups of accelerators. Then, the Algorithm 1 can be used to partition between two groups. Based on this insight, we have a hierarchical partition algorithm.

Figure 3 illustrates an example to partition eight acceler-

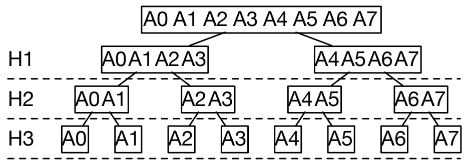

Figure 3: Hierarchical partition for three levels.

ators with three levels. Originally, there are 8 accelerators waiting to be assigned with the training workloads. As shown in Figure 3, at hierarchy level H1, we first view A0 to A3 as a group of accelerators and A4 to A7 as the other group. By utilizing Algorithm 1, the workloads can be assigned to each of these two groups. After that, the groups of A0 to A3 and A4 to A7 are further divided in to 4 groups at hierarchy level H2. Last, as illustrated in Figure 3, each accelerator is assigned its own workloads at hierarchy level H3 with the same partition method. Such binary tree structure of our hierarchical partition method enables the partition of 8 accelerators in logarithmic times of iteration, which equals 3 ($log_2 8$). With Figure 3, at each hierarchy level, we have a parallelism list for every weighted layers for layers to be partitioned into the two subarrays. In total three parallelism lists are generated by the partition algorithm and we can use the lists to determine the parallelism setting for each accelerator, and the tensors they hold.

Algorithm 2 Hierarchical Partition.

```
Input:
    1. The number of hierarchy levels, $H$,
    2. Batch size, $B$,
    3. The number of weighted layers in a DNN model, $L$,
    4. A list of hyper parameters (layer type: conv or fc,
    kernel sizes, parameter for pooling, activation function),
    $HP[l], l = 0, .., L$-1.
Output:
    1. Total communication, $com$,
    2. A list of parallelism for each weighted layer at each
    hierarchy level, $P[h][l], h = 0, ..., H$-1$, l = 0, .., L$-1.

1: if ($H == 0$) then
2:     return (0, [ ]).
3: else
4:     ($com\_h, P\_h$) = PartitionBetweenTwoAccelerators().
5:     ($com\_n, P\_n$) = HierarchicalPartition($H$-1).
6:     Update $P$.
7:     $com = com\_h + 2 * com\_n$.
8:     return ($com$, $P$).
9: end if
```

In the hierarchical partition algorithm, for one specific hierarchy, we first partition an array of the accelerators into two subarrays by Algorithm 1, and then recursively apply the hierarchical partition algorithm to the subarray until there is only one accelerator in one subarray. The hierarchical partition can be summarized as Algorithm 2.

The inputs of Algorithm 2 are similar to those of Algorithm 1 except that the hierarchical partition algorithm also needs the hierarchy levels(H) as the input to denote how many times we divide the training workload into two groups. For example, if hierarchy levels H equals 4, the total number of accelerators in the array are 2^H. The outputs of hierarchical partition algorithm include the total communication through all the hierarchy levels (com) and a list containing all the partition strategies for each layer of all the hierarchy levels in the accelerator array.

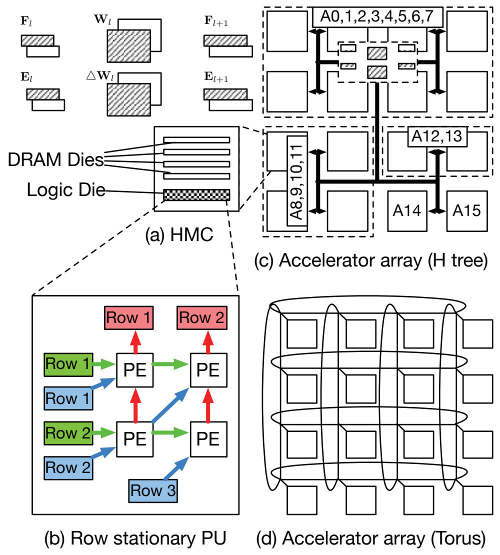

Figure 4: Overall view of (a) an HMC-based accelerator, (b) a row stationary processing unit, (c) an array of sixteen accelerators in H tree, and (d) the accelerator array in torus.

Our hierarchical partition algorithm is a recursive function. In each recursion, Algorithm 2 first calculates the minimal communication (com_h) for current hierarchy level using 1. Then, it calls itself with the input hierarchy levels(h) changed to hierarchy levels($h-1$) and obtains the total minimal communication (com_n) for the lower hierarchy levels. At last it returns the total communication com by adding current communication com_h and $2 \times com\_n$.

5. HYPAR ARCHITECTURE

This section presents the HYPAR architecture composed of an accelerator array, where the parallelism setting is determined by HYPAR. The individual accelerator is based on Hybrid Memory Cube (HMC) [102], as shown in Figure 4 (a). An HMC consists of stacked DRAM dies and logic die, and they are connected by through silicon vias (TSVs), Processing units (PUs) can be integrated on the logic die. HMC provides high memory bandwidth (320 GB/s) [102], which is suitable for the in-memory processing for DNNs. Since DNN accelerators incur heavy memory accessing and intensive computation operations, recent works [19, 22] demonstrated that HMC-based neural network accelerator could drastically reduce data movements.

For the PUs, as shown in Figure 4 (b), we implement a row stationary design as [15]. In such design, weight rows (green) are shared by processing engines horizontally, feature map rows (blue) are shared by processing engines diagonally, and partial sum rows (red) are accumulated vertically. The row stationary design is suitable for convolution computations.

Figure 4 shows the overall HYPAR architecture composed of a 2-D array of sixteen accelerators. Thus, the number of

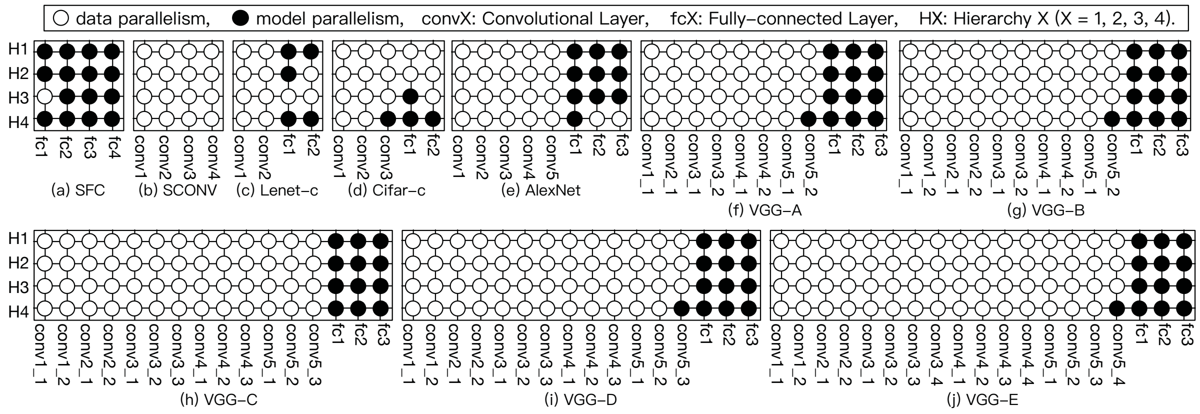

Figure 5: Optimized parallelism for weighted layers in four hierarchy levels of ten networks in HYPAR.

hierarchy levels for the accelerator array is four. To support efficient hierarchical communication, the accelerators in the array are connected with certain network topology. Notice that HYPAR algorithm hierarchically partitions the accelerators. In each level, the two subarrays of accelerators (or two accelerators in the last level) communicate between each other. The communication relationship is represented by the two edges from their ancestor subarray. For example, as shown in Figure 3, in level H2, the ancestor subarray {A0, A1, A2, A3} has edges connected to the child subarray {A0, A1} and {A2, A3}. That means subarray {A0, A1} and {A2, A3} communicate with each other. For subarray {A0, A1} to communicate with subarray {A4, A5}, the communication "backtracks" to level H1. That is natural because in the hierarchical partition Algorithm 2, subarray {A0, A1} and {A4, A5} have no direct communication in level H2, but they may communicate in level H1.

We consider two network topologies. Figure 4 (c) shows the H tree topology, which can match the communication patterns. As shown in Figure 4 (b), the tensors with shadow lines are assigned to subarray {A0-7}, while the white tensors are assigned to subarray {A8-15}, and the tensors in each subarray are recursively assigned to sub-subarrays according to the hierarchical partition.

Figure 4 (d) shows the torus topology for the accelerator array. While torus is a common topology, it performs worse than the H tree. It is because the tensor partition pattern generated by Algorithm 2 does not match torus topology as well as the H tree.

6. EVALUATION

6.1 Evaluation Setup

In the evaluation, we use three datasets, including small, medium and large size: MNIST [114], CIFAR-10 [115] and ImageNet [104]. We use ten deep neural network models in the evaluation: `SFC`, `SCONV`, `Lenet-c`, `Cifar-c`, `AlexNet`, `VGG-A`, `VGG-B`, `VGG-C`, `VGG-D` and `VGG-E`. `SFC` is a fully-connected network (no convolutional layers), and `SCONV` is a convolutional network without any fully-connected layers. The hyper parameters of `SFC` and `SCONV` are shown in Table 3. Notice that `SFC` and `SCONV` are valid networks, which have an accuracy of 98.28% and 98.71%, respectively. We build the two "strange" networks as extreme cases to demonstrate the effects of data and model parallelism. `Lenet-c` is a convolutional neural network for MNIST, `Cifar-c` is for Cifar-10. `AlexNet` and `VGG`s are for ImageNet, and the model hyper parameters can be found in [6] and [105] respectively.

Table 3: Hyper parameters for `SFC` and `SCONV`.

SFC	784-8192-8192-8192-10
SCONV	20@5×5, 50@5×5(2×2 max pool), 50@5×5, 10@5×5(2×2 max pool)

For the accelerator array, we employ sixteen accelerators (as shown in Figure 4). The number of partition hierarchy levels is four. Each accelerator is based on an HMC cube. The batch size is 256. We use an event-driven simulation. Within an HMC vault (i.e., an Eyeriss accelerator and its local memory), we modeled the computation cost and the memory access between vaults, we also considered the tensor communication. For the HMC, the DRAM bandwidth is 320 GB/s and each HMC has 8 GB memory [102]. The PUs used in the evaluation have an Eyeriss-like [15] row stationary architecture, and each processing unit has 168 (12×14) processing engines, 108 KB on-chip buffer and 84.0 GPOS/s computation density. The accelerator works at 250 MHz and the link bandwidth is 1600 Mb/s (i.e. the total network bandwidth is 25.6 Gb/s). The energy consumption for a 32-bit float ADD operation is 0.9 pJ, a 32-bit float MULT operation is 3.7 pJ, a 32-bit SRAM accessing is 5.0 pJ and a 32-bit DRAM accessing is 640 pJ [116]. We use a precision of 32-bit floating point in the computation.

We compare the default Model Parallelism (where all layers at the four hierarchy levels are assigned to model parallelism), the default Data Parallelism (where all layers at the four hierarchy levels are assigned to data parallelism) and HYPAR in the evaluation.

6.2 Overall Results

6.2.1 Optimized Parallelism in HYPAR

Figure 5 shows the optimized parallelisms for weighted layers in the ten networks at four hierarchy levels. For most networks, especially large-scale networks, such as `AlexNet` and `VGG`s, in the convolutional layers, the parallelisms are usually data parallelism, and in fully-connected layers, the parallelisms usually are model parallelism. That is consistent to our analysis in Section 3.2, i.e., convolutional layers favor data parallelism while fully connected layers prefer model parallelism to keep the communication as low as possible. The optimization of extreme cases is a slightly different. For `SFC`, because all layers are fully-connected layer, except `fc1`@H3 is optimized to data parallelism, all other layers at the four hierarchy levels are optimized to model parallelism. For `SCONV`, a network with all convolutional layers, all layers at the four hierarchy levels are optimized to data parallelism. We also see that except `SCONV`, the optimized parallelisms for layers at four hierarchy levels consist of both data parallelism and model parallelism, leading to hybrid parallelism.

6.2.2 *Performance*

The performance of the default Model Parallelism, the default Data Parallelism and HyPar are shown in Figure 6. The performance results are normalized to the default Data Parallelism.

HyPar achieves a 3.39× performance gain compared to Data Parallelism on average. We can also find that the performance of Model Parallelism is almost always worse than Data Parallelism. Thus, among these two, we should mostly prefer Data Parallelism in DNN training. For the extreme case `SFC`, Model Parallelism performs better than Data Parallelism, but HyPar still performs slightly better than Model Parallelism. Although fully-connected layers prefer Model Parallelism, as shown in Figure 5 (a), `fc1`@H3 is optimized to Data Parallelism. Therefore, the optimized parallelisms in HyPar are not fully Model Parallelism, and this explains why HyPar performs better than Model Parallelism (23.48× v.s. 22.19×). It validates the partitioning algorithm of HyPar. For the other extreme case `SCONV`, HyPar performs the same as Data Parallelism. For other eight networks, HyPar achieves performance gains ranging from 1.23× to 4.97× compared to Data Parallelism.

6.2.3 *Energy Efficiency*

The energy efficiency of the default Model Parallelism, the default Data Parallelism and HyPar are shown in Figure 7. The energy efficiency is the the energy saving normalized to the default Data Parallelism.

HyPar achieves a 1.51× energy efficiency compared to Data Parallelism on average. Again, Model Parallelism is almost always less energy efficiency than Data Parallelism. For the extreme case `SFC`, the energy efficiency of Model Parallelism is higher than that of Data Parallelism, but HyPar performs better than Model Parallelism, and HyPar has higher energy efficiency (10.27×) than Model Parallelism (9.96×). For the other extreme case `SCONV`, HyPar has a 1.00× energy efficiency, the same as Data Parallelism. For other eight networks, HyPar achieves energy efficiencies ranging from 1.03× to 1.81× compared to Data Parallelism.

6.2.4 *Total Communication per Step*

In HyPar, the total communication of a network is optimized to improve the performance and energy efficiency. We show the total communication per step of the default Model Parallelism, the default Data Parallelism and HyPar in Figure 8.

The geometric means of total communication for Model Parallelism, Data Parallelism and HyPar are 8.88 GB, 1.83GB and 0.318 GB respectively. Model Parallelism mostly has much higher amount of total communication than Data Parallelism and HyPar. However, for the extreme case `SFC`, the total communication of Model Parallelism is lower than that of Data Parallelism. HyPar has lower amount of total communication (0.681 GB) than Model Parallelism (0.723 GB) in `SFC`. For the other extreme case `SCONV`, HyPar has the same amount of total communication as Data Parallelism, and the amount is lower than Model Parallelism. For other eight networks, especially the large size networks, i.e., `AlexNet`, `VGG-A`, `VGG-B`, `VGG-C`, `VGG-D` and `VGG-E`, the total communication in Data Parallelism is almost ten times lower than that of Model Parallelism, and HyPar is about another ten

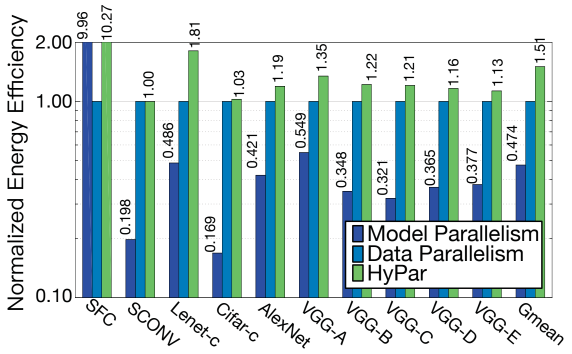

Figure 7: Energy efficiency of Model Parallelism, Data Parallelism and HyPar normalized to Data Parallelism.

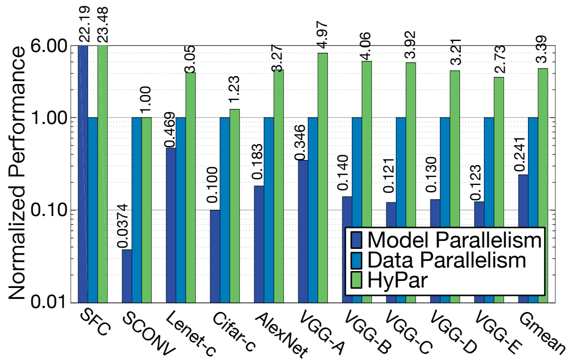

Figure 6: Performance of Model Parallelism, Data Parallelism and HyPar normalized to Data Parallelism.

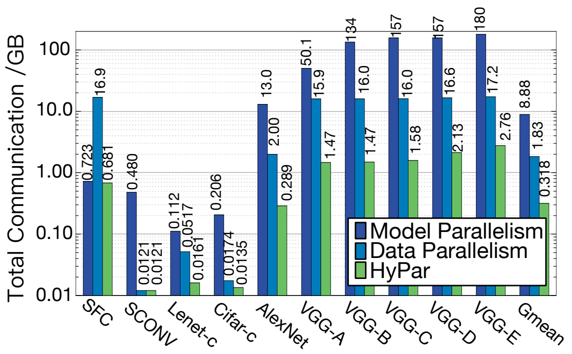

Figure 8: Total communication (in GB) of Model Parallelism, Data Parallelism and HyPar per step.

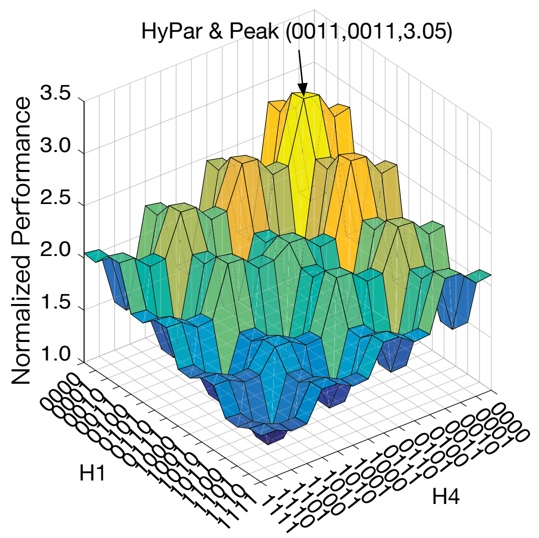

Figure 9: Normalized performance (to Data Parallelism) in parallelism space exploration for `Lenet-c`. H2 and H3 are fixed and parallelism for layers at H1 and H4 are explored. 0 indicates data parallelism while 1 indicates model parallelism.

times lower than Data Parallelism. We can find that the communication is a key factor that determines the performance and energy efficiency of an array of accelerators.

6.3 Case Studies

6.3.1 *Parallelism Space Exploration for Lenet-c*

We explore the parallelism space for `Lenet-c` to find the maximum performance. `Lenet-c` has four weighted layers, and four partition hierarchy levels, so the capacity of searching space would be $2^{4\times4} = 65536$, which is too large. As an alternative, we fix the parallelisms of all four layers at two hierarchy levels H2 and H3, and explore the possible parallelisms for all four layers at two hierarchy levels H1 and H4. The parallelisms of layers at H2 and H3 are fixed as the optimized ones, as shown in Figure 5 (c). So the capacity of the searching space is now $2^{2\times4} = 256$.

The results are shown in Figure 9. The results are normalized to the default Data Parallelism. As we can see, the peak of the normalized performance is 3.05×, at H1 = 0011 and H4 = 0011, which means the parallelisms for four layers at H1 are dp, dp, mp, mp and the four layers at H4 are dp, dp, mp, mp. That is exactly the performance gain of `Lenet-c` with the parallelisms optimized by HyPar.

6.3.2 *Parallelism Space Exploration for VGG-A*

We explore the parallelism space for `VGG-A` to find the maximum performance. While the capacity of full searching space would be $2^{4\times11} = 17.6$ T, which is never possible to enumerate every point. As an alternative, we fix the parallelisms of nine layers in the network, except `conv5_2` and `fc1`. We then explore the possible parallelisms for `conv5_2` and `fc1` at the four hierarchy levels. The parallelisms of the other layers are fixed as the optimized ones, as shown in Figure 5 (f). The capacity of the searching space is reduced

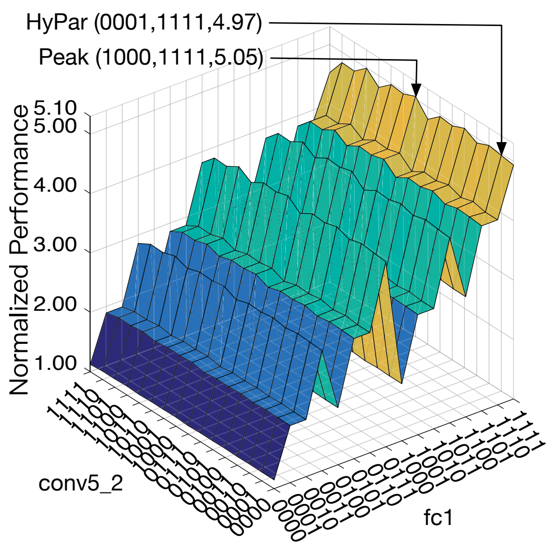

Figure 10: Normalized performance (to Data Parallelism) in parallelism space exploration for `VGG-A`. All weighted layers are fixed except `conv5_2` and `fc1`. Parallelism for H1 to H4 of `conv5_2` and `fc1` are explored.

to $2^{4\times2} = 256$.

The exploration results are shown in Figure 10. The results are normalized to the default Data Parallelism. As we can see, the peak of the normalized performance is 5.05×, at `conv5_2` = 1000 and `fc1` = 1111, which means the parallelisms for `conv5_2` at four hierarchy levels (H1 to H4) are mp, dp, dp, dp and `fc1` in four hierarchy levels (H1 to H4) are mp, mp, mp, mp. However, the performance optimized by HyPar is 4.97×, and the corresponding parallelisms for `conv5_2` at four hierarchy levels (H1 to H4) are dp, dp, dp, mp. That is because HyPar optimizes the total communication as a proxy for optimizing performance. Even HyPar failed to provide the maximum performance with the optimized setting, the performance of HyPar is very close to the maximum (4.97× vs. 5.05×), and is still much higher than the baseline Data Parallelism (4.97× vs. 1.00×).

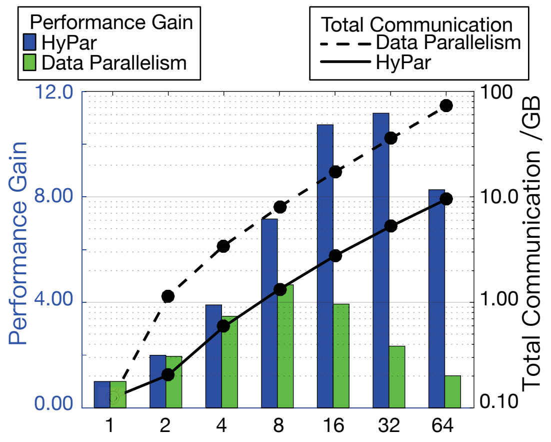

Figure 11: Comparison of scalability of HyPar and Data Parallelism. Left Y axis: performance gain normalized to one accelerator, right Y axis: total communication.

6.4 Scalability

We explore the scalability of HyPar using `VGG-A` as an example. In the exploration, we scale the number of accelerators from 1 to 64. We compare the performance gains of HyPar and the default Data Parallelism. The gains are normalized to the performance of the performance of one accelerator.

The results are shown in Figure 11. HyPar always performs better than the default Data Parallelism. The performance gains of the default Data Parallelism become decreasing after the number of accelerators exceeds 8. But the gains of HyPar increases until the number of accelerators exceeds 32. We can see that HyPar scales better than the default Data Parallelism. HyPar is always better than the default Data Parallelism in performance gains, and HyPar always has lower total communication.

6.5 Comparisons

6.5.1 *Comparison of H Tree and Torus Topology*

We compare the performance of H tree and torus typologies. The parallelisms for each layers are the optimized choices by HyPar, but the only difference is the connection topology of the sixteen accelerators. For H-tree, it is physically it is a fat-tree, and switches are placed at each parent node. The bandwidth between groups in a higher hierarchy are doubled compared to that of a lower hierarchy (but the number of links is halved). In a torus, the bandwidth for a link is the same.

Figure 12 shows the performance of torus and H tree topology, normalized to Data Parallelism. For `SFC`, both the two typologies have a speedup of more than 10×, because all layers in `SFC` are fully-connected layers, and Data Parallelism has lower performance regardless of connection typologies. For the other networks, we can see, H tree outperforms torus topology, because the parallelism and tensor partition are determined in a binary tree pattern, and H tree is naturally more suitable for the pattern. The geometric mean of performance of torus and H tree typologies are 2.23× and 3.39×.

While three are many different possible topologies for the accelerator array, HyPar is topology independent. It is a simplification, but the topology-independent communication model and the dynamic programming method indeed reduced the total communication between accelerators. From the comparison of H tree and torus, we can see the partition also works for torus although HyPar prefers H tree, so the simplification is reasonable.

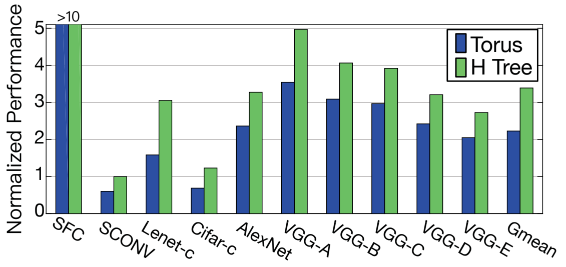

Figure 12: Normalized performance (to Data Parallelism) of torus and H tree topology.

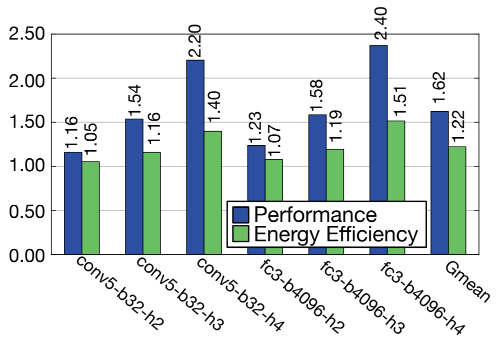

Figure 13: Performance and energy efficiency of HyPar compared to the trick in [111].

6.5.2 *Comparison of* HyPar *and the Trick in [111]*

Batch size is an important hyper parameter in DNN training, and batch size should be customized for specific purposes rather than a default setting. For for larger training throughput [117], a larger batch size (eg. 4096) should be selected, while for higher testing accuracy and ability to generalize [118], a small batch size (eg. 32) should be selected.

Thus we use the two batch size, i.e. `b32` and `b4096`, to evaluate HyPar and the Trick in [111]. We use the fully-connected and convolutional layers `fc3` and `conv5` in `VGG-E`, under three hierarchy levels `h2`, `h3` and `h4`. Figure 13 shows the performance and energy efficiency of HyPar compared to the Trick. We can see, the performance of HyPar is 1.62× better than the Trick and HyPar is 1.22× more energy efficient on average. HyPar can be 2.40× faster than the Trick.

So one may ask, what is wrong with the Trick [111]? Let's get back to our communication model (Section 3). With the intra-layer communication model, for `conv5`, $\mathbb{A}(\triangle\mathbf{W}_l) = C_iC_oK^2 = 512 \times 512 \times 3^2 = 2,359,296$, while $\mathbb{A}(\mathbf{F}_{l+1}) = BC_oWH = 32 \times 512 \times 14 \times 14 = 3,211,264$. Because $\mathbb{A}(\triangle\mathbf{W}_l) < \mathbb{A}(\mathbf{F}_{l+1})$, here `conv5` should be configured to model parallelism rather than data parallelism in the Trick. For `fc3`, $\mathbb{A}(\triangle\mathbf{W}_l) = C_iC_o = 4096 \times 1000 = 4,096,000$, while $\mathbb{A}(\mathbf{F}_{l+1}) = BC_o = 4096 \times 1000 = 4,096,000$. $\mathbb{A}(\triangle\mathbf{W}_l)$ and $\mathbb{A}(\mathbf{F}_{l+1})$ are the same, we can not clearly see which parallelism is better than the other, but we can further use inter-layer communication model to explain. According to Table 2, the communication of dp-dp is 0 while that of either mp-mp or mp-dp is not 0. So we should choose data parallelism for that layer, but unfortunately, the Trick chose model parallelism.

7. CONCLUSION

We propose HyPar to determine layer-wise parallelism for deep neural network training with an array of DNN accelerators. HyPar partitions the feature map tensors (input and output), the kernel tensor, the gradient tensor, and the error tensors for the DNN accelerators. A partition constitutes the choice of parallelism for all weighted layers. The optimization target is to search a partition that minimizes the total communication during training a complete deep neural network. HyPar is practical: the time complexity for the partition search in HyPar is *linear*. We apply this method in HyPar architecture, an HMC-based DNN training architecture to minimize data movement. We evaluate HyPar with ten DNN models from classic Lenet to large-size model VGGs, and the number of weighted layers of these models ranges from four to nineteen. Our evaluation shows that HyPar achieves a performance gain of 3.39× and an energy efficiency gain of 1.51× compared to the default data parallelism on average, and HyPar performs up to 2.40× better than the trick [111].

ACKNOWLEDGEMENT

We thank the anonymous reviewers of HPCA 2019, MICRO 2018, ISCA 2018 for their constructive and insightful comments. This work is supported in part by NSF 1725456, 1615475, and DOE DE-SC0018064. This work is also supported by the National Science Foundation grants NSF-CCF-1657333, NSF- CCF-1717754, NSF-CNS-1717984, and NSF-CCF-1750656.

8. REFERENCES

[1] I. Goodfellow *et al.*, *Deep learning*. MIT press Cambridge, 2016.
[2] Y. Bengio *et al.*, "Deep learning," *Nature*, 2015.
[3] Y. Sun *et al.*, "Deepid3: Face recognition with very deep neural networks," *arXiv*, 2015.
[4] G. Hinton *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Processing Magazine*, 2012.
[5] C. Farabet *et al.*, "Learning hierarchical features for scene labeling," *PAMI*, 2013.
[6] A. Krizhevsky *et al.*, "Imagenet classification with deep convolutional neural networks," in *NIPS*, 2012.
[7] A. Farmahini-Farahani *et al.*, "Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules," in *HPCA*, 2015.
[8] S. Liu *et al.*, "Cambricon: An instruction set architecture for neural networks," in *ISCA*, 2016.
[9] S. Zhang *et al.*, "Cambricon-x: An accelerator for sparse neural networks," in *MICRO*, 2016.
[10] T. Chen *et al.*, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ASPLOS*, 2014.
[11] Y. Chen *et al.*, "Dadiannao: A machine-learning supercomputer," in *MICRO*, 2014.
[12] Z. Du *et al.*, "Shidiannao: Shifting vision processing closer to the sensor," in *ISCA*, 2015.
[13] D. Liu *et al.*, "Pudiannao: A polyvalent machine learning accelerator," in *ASPLOS*, 2015.
[14] Y.-H. Chen *et al.*, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ISCA*, 2016.
[15] Y.-H. Chen *et al.*, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *JSSC*, 2017.
[16] Y.-H. Chen *et al.*, "Using dataflow to optimize energy efficiency of deep neural network accelerators," *IEEE Micro*, 2017.
[17] S. Han *et al.*, "Eie: efficient inference engine on compressed deep neural network," in *ISCA*, 2016.
[18] Z. Du *et al.*, "Neuromorphic accelerators: A comparison between neuroscience and machine-learning approaches," in *MICRO*, 2015.
[19] D. Kim *et al.*, "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory," in *ISCA*, 2016.
[20] W. Lu *et al.*, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *HPCA*, 2017.
[21] A. Ren *et al.*, "Sc-dcnn: Highly-scalable deep convolutional neural network using stochastic computing," in *ASPLOS*, 2017.
[22] M. Gao *et al.*, "Tetris: Scalable and efficient neural network acceleration with 3d memory," in *ASPLOS*, 2017.
[23] S. B. Furber *et al.*, "Overview of the spinnaker system architecture," *IEEE Transactions on Computers*, 2013.
[24] C. Zhang *et al.*, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *FPGA*, 2015.
[25] J. Qiu *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *FPGA*, 2016.
[26] M. Motamedi *et al.*, "Design space exploration of fpga-based deep convolutional neural networks," in *ASP-DAC*, 2016.
[27] N. Suda *et al.*, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," in *FPGA*, 2016.
[28] C. Zhang *et al.*, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *ICCAD*, 2016.
[29] S. Han *et al.*, "Ese: Efficient speech recognition engine with sparse lstm on fpga.," in *FPGA*, 2017.
[30] Y. Ma *et al.*, "Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks," in *FPGA*, 2017.
[31] R. Zhao *et al.*, "Accelerating binarized convolutional neural networks with software-programmable fpgas.," in *FPGA*, 2017.
[32] J. Zhang and J. Li, "Improving the performance of opencl-based fpga accelerator for convolutional neural network.," in *FPGA*, 2017.
[33] H. Esmaeilzadeh *et al.*, "Neural acceleration for general-purpose approximate programs," in *MICRO*, 2012.
[34] C. Farabet *et al.*, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *CVPRW*, 2011.
[35] S. Venkataramani *et al.*, "Scaledeep: A scalable compute architecture for learning and evaluating deep networks," in *ISCA*, 2017.
[36] Y. Ji *et al.*, "Neutrams: Neural network transformation and co-design under neuromorphic hardware constraints," in *MICRO*, 2016.
[37] T. Na and S. Mukhopadhyay, "Speeding up convolutional neural network training with dynamic precision scaling and flexible multiplier-accumulator," in *ISLPED*, 2016.
[38] A. Parashar *et al.*, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *ISCA*, 2017.
[39] M. Alwani *et al.*, "Fused-layer cnn accelerators," in *MICRO*, 2016.
[40] Y. Shen *et al.*, "Overcoming resource underutilization in spatial cnn accelerators," in *FPL*, 2016.
[41] Y. Shen *et al.*, "Escher: A cnn accelerator with flexible buffering to minimize off-chip transfer," in *FCCM*, 2017.
[42] Y. Shen, M. Ferdman, and M. Peter, "Maximizing cnn accelerator efficiency through resource partitioning," in *ISCA*, pp. 535–547, ACM, 2017.
[43] A. Mirhoseini *et al.*, "Perform-ml: Performance optimized machine learning by platform and content aware customization," in *DAC*, 2016.
[44] M. S. Razlighi *et al.*, "Looknn: Neural network with no multiplication," in *DATE*, 2017.
[45] Z. Takhirov *et al.*, "Energy-efficient adaptive classifier design for mobile systems," in *ISLPED*, 2016.
[46] J. H. Ko *et al.*, "Design of an energy-efficient accelerator for training of convolutional neural networks using frequency-domain computation," in *DAC*, 2017.
[47] H. Sharma *et al.*, "From high-level deep neural models to fpgas," in *MICRO*, 2016.
[48] B. Reagen *et al.*, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *ISCA*, 2016.
[49] C. Zhang and V. K. Prasanna, "Frequency domain acceleration of convolutional neural networks on cpu-fpga shared memory system," in *FPGA*, 2017.
[50] Z. Fan *et al.*, "Red: A reram-based deconvolution accelerator," in *DATE*, 2019.
[51] F. Chen *et al.*, "Regan: A pipelined reram-based accelerator for generative adversarial networks," in *ASP-DAC*, 2018.
[52] F. Chen and H. Li, "Emat: an efficient multi-task architecture for transfer learning using reram," in *ICCAD*, 2018.
[53] B. Li *et al.*, "Reram-based accelerator for deep learning," in *DATE*, 2018.
[54] H. Ji *et al.*, "Recom: An efficient resistive accelerator for compressed deep neural networks," in *DATE*, 2018.
[55] X. Qiao *et al.*, "Atomlayer: a universal reram-based cnn accelerator with atomic layer computation," in *DAC*, 2018.
[56] J. Mao *et al.*, "Modnn: Local distributed mobile computing system for deep neural network," in *DATE*, 2017.
[57] J. Mao *et al.*, "Mednn: A distributed mobile system with enhanced partition and deployment for large-scale dnns," in *ICCAD*, 2017.
[58] J. Mao *et al.*, "Adalearner: An adaptive distributed mobile learning system for neural networks," in *ICCAD*, 2017.
[59] T. Tang *et al.*, "Binary convolutional neural network on rram," in *ASP-DAC*, 2017.
[60] L. Jiang *et al.*, "Xnor-pop: A processing-in-memory architecture for binary convolutional neural networks in wide-io2 drams," in *ISLPED*, 2017.

[61] Q. Deng *et al.*, "Dracc: a dram based accelerator for accurate cnn inference," in *DAC*, 2018.

[62] Q. Lou *et al.*, "3dict: a reliable and qos capable mobile process-in-memory architecture for lookup-based cnns in 3d xpoint rerams," in *ICCAD*, 2018.

[63] H. Ji *et al.*, "Hubpa: High utilization bidirectional pipeline architecture for neuromorphic computing," in *ASP-DAC*, 2019.

[64] T. Liu *et al.*, "Mt-spike: A multilayer time-based spiking neuromorphic architecture with temporal error backpropagation," in *ICCAD*, 2017.

[65] T. Liu *et al.*, "Pt-spike: A precise-time-dependent single spike neuromorphic architecture with efficient supervised learning," in *ASP-DAC*, 2018.

[66] X. Zhang *et al.*, "Dnnbuilder: an automated tool for building high-performance dnn hardware accelerators for fpgas," in *ICCAD*, 2018.

[67] P. Wang *et al.*, "Snrram: an efficient sparse neural network computation architecture based on resistive random-access memory," in *DAC*, 2018.

[68] S. Li *et al.*, "Drisa: A dram-based reconfigurable in-situ accelerator," in *MICRO*, 2017.

[69] C.-E. Lee *et al.*, "Stitch-x: An accelerator architecture for exploiting unstructured sparsity in deep neural networks," in *SysML*, 2018.

[70] J. Liu *et al.*, "Processing-in-memory for energy-efficient neural network training: A heterogeneous approach," in *MICRO*, 2018.

[71] Q. Yang *et al.*, "A quantized training method to enhance accuracy of reram-based neuromorphic systems," in *ISCAS*, 2018.

[72] X. Liu *et al.*, "Reno: a high-efficient reconfigurable neuromorphic computing accelerator design," in *DAC*, 2015.

[73] H. Yan *et al.*, "Celia: A device and architecture co-design framework for stt-mram-based deep learning acceleration," in *ICS*, 2018.

[74] Y. Wang *et al.*, "Group scissor: Scaling neuromorphic computing design to large neural networks," in *DAC*, 2017.

[75] J. Albericio *et al.*, "Cnvlutin: ineffectual-neuron-free deep neural network computing," in *ISCA*, 2016.

[76] P. Judd *et al.*, "Stripes: Bit-serial deep neural network computing," in *MICRO*, 2016.

[77] D. Mahajan *et al.*, "Tabla: A unified template-based framework for accelerating statistical machine learning," in *HPCA*, 2016.

[78] M. N. Bojnordi and E. Ipek, "Memristive boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning," in *HPCA*, 2016.

[79] J. Yu *et al.*, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," in *ISCA*, 2017.

[80] J. Albericio *et al.*, "Bit-pragmatic deep neural network computing," in *MICRO*, 2017.

[81] C. Ding *et al.*, "Circnn: accelerating and compressing deep neural networks using block-circulant weight matrices," in *MICRO*, 2017.

[82] J. Park *et al.*, "Scale-out acceleration for machine learning," in *MICRO*, 2017.

[83] R. Cai *et al.*, "Vibnn: Hardware acceleration of bayesian neural networks," in *ASPLOS*, 2018.

[84] A. Yazdanbakhsh *et al.*, "Ganax: A unified mimd-simd acceleration for generative adversarial networks," in *ISCA*, 2018.

[85] V. Akhlaghi *et al.*, "Snapea: Predictive early activation for reducing computation in deep convolutional neural networks," in *ISCA*, 2018.

[86] K. Hegde *et al.*, "Ucnn: Exploiting computational reuse in deep neural networks via weight repetition," in *ISCA*, 2018.

[87] E. Park *et al.*, "Energy-efficient neural network accelerator based on outlier-aware low-precision computation," in *ISCA*, 2018.

[88] M. Song *et al.*, "Prediction based execution on deep neural networks," in *ISCA*, 2018.

[89] H. Sharma *et al.*, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *ISCA*, 2018.

[90] C. Deng *et al.*, "Permdnn: Efficient compressed deep neural network architecture with permuted diagonal matrices," in *MICRO*, 2018.

[91] "Google supercharges machine learning tasks with tpu custom chip." `https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html`.

[92] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *ISCA*, 2017.

[93] P. A. Merolla *et al.*, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, 2014.

[94] S. K. Esser *et al.*, "Backpropagation for energy-efficient neuromorphic computing," in *NIPS*, 2015.

[95] S. K. Esser *et al.*, "Convolutional networks for fast, energy-efficient neuromorphic computing," *PNAS*, 2016.

[96] "Intel nervana platform delivers deep learning analytics." `https://www.intel.com/content/www/us/en/financial-services-it/deep-learning-delivers-advanced-analytics-brief.html`.

[97] "Intel's new self-learning chip promises to accelerate artificial intelligence." `https://newsroom.intel.com/editorials/intels-new-self-learning-chip-promises-accelerate-artificial-intelligence/`.

[98] "Nvidia ai." `https://www.nvidia.com/en-us/deep-learning-ai/`.

[99] "Qualcomm machine learning." `https://www.qualcomm.com/invention/cognitive-technologies/machine-learning`.

[100] "Deephi tech." `https://www.xilinx.com/products/design-tools/deephi.html`.

[101] "Microsoft unveils project brainwave for real-time ai." `https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave/`.

[102] "Hybrid memory cube specification 2.1." `http://hybridmemorycube.org`.

[103] L. Song *et al.*, "Pipelayer: A pipelined reram-based accelerator for deep learning," in *HPCA*, 2017.

[104] J. Deng *et al.*, "Imagenet: A large-scale hierarchical image database," in *CVPR*, 2009.

[105] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *ICLR*, 2015.

[106] K. He *et al.*, "Deep residual learning for image recognition," in *CVPR*, 2016.

[107] M. Li *et al.*, "Scaling distributed machine learning with the parameter server," in *OSDI*, 2014.

[108] M. Li *et al.*, "Communication efficient distributed machine learning with the parameter server," in *NIPS*, 2014.

[109] A. Coates *et al.*, "Deep learning with cots hpc systems," in *ICML*, 2013.

[110] J. Dean *et al.*, "Large scale distributed deep networks," in *NIPS*, 2012.

[111] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv*, 2014.

[112] H. Kwon *et al.*, "Maestro: An open-source infrastructure for modeling dataflows within deep learning accelerators," *arXiv*, 2018.

[113] "Build and train machine learning models on our new google cloud tpus." `https://www.blog.google/topics/google-cloud/google-cloud-offer-tpus-machine-learning/`.

[114] Y. LeCun *et al.*, "The mnist database of handwritten digits," 1998.

[115] A. Krizhevsky, V. Nair, and G. Hinton, "The cifar-10 dataset," 2014.

[116] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *ISSCC*, 2014.

[117] P. Goyal *et al.*, "Accurate, large minibatch sgd: Training imagenet in 1 hour," *arXiv*, 2017.

[118] D. Masters and C. Luschi, "Revisiting small batch training for deep neural networks," *arXiv*, 2018.