AccPar: Tensor Partitioning for Heterogeneous Deep Learning Accelerators

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

ABSTRACT

Deep neural network (DNN) accelerators as an example of domain-specific architecture have demonstrated great success in DNN inference. However, the architecture acceleration for equally important DNN training has not yet been fully studied. With data forward, error backward and gradient calculation, DNN training is a more complicated process with higher computation and communication intensity. Because the recent research demonstrates a diminishing specialization return, namely, "accelerator wall", we believe that a promising approach is to explore coarse-grained parallelism among multiple performance-bounded accelerators to support DNN training. Distributing computations on multiple heterogeneous accelerators to achieve high throughput and balanced execution, however, remaining challenging.

We present ACCPAR, a principled and systematic method of determining the tensor partition among heterogeneous accelerator arrays. Compared to prior empirical or unsystematic methods, ACCPAR considers the complete tensor partition space and can reveal previously unknown new parallelism configurations. ACCPAR optimizes the performance based on a cost model that takes into account both computation and communication costs of a heterogeneous execution environment. Hence, our method can avoid the drawbacks of existing approaches that use communication as a proxy of the performance. The enhanced flexibility of tensor partitioning in ACCPAR allows the flexible ratio of computations to be distributed among accelerators with different performances. The proposed search algorithm is also applicable to the emerging multi-path patterns in modern DNNs such as ResNet. We simulate ACCPAR on a heterogeneous accelerator array composed of both TPU-v2 and TPU-v3 accelerators for the training of large-scale DNN models such as Alexnet, Vgg series and Resnet series. The average performance improvements of the state-of-the-art "one weird trick" (OWT) and HYPAR, and ACCPAR, normalized to the baseline data parallelism scheme where each accelerator replicates the model and processes different input data in parallel, are 2.98×, 3.78×, and 6.30×, respectively.

1. INTRODUCTION

Advances in deep learning (DL) have become the main drivers of revolutions in various commercial and enterprise applications, such as computer vision [1–3], social network [4–6], financial data analysis [7–9], healthcare [10–12] and scientific computing [13–15]. Due to the high demand of computing power in DL applications, we have recently witnessed a phenomenal trend in which the landscape of computing has shifted from general-purpose processors to domain-specific architectures [16–110]. By sacrificing some flexibility, such domain-specific accelerators are specialized for executing kernels of modern DL algorithms and therefore, are capable to deliver high performance with low power budget.

While the latest advances are pushing the envelope of DL acceleration for higher performance and energy efficiency, a recent study of chip specialization [111] has predicted an ultimate *accelerator wall*. More specifically, due to the constraints in the exploration of mapping computational problems (e.g., DL) onto hardware platforms with fixed hardware resources, the optimization space of chip specialization is limited by a theoretical roofline. Combining with recent slow CMOS technology scaling, the gains from specific accelerator designs will gradually diminish and the execution efficiency will eventually hit an upper-bound. On the other hand, the computational demands of emerging DL applications continue increasing in order to adapt to more complex models with deeper structures [112, 113] or more sophisticated learning methods [114, 115].

To address the mismatch between the diminishing performance gains in hardware accelerators and the ever-growing computational demands, it is imperative to explore coarse-grained parallelism among multiple performance-bounded accelerators to support large-scale DL applications. In general, a deep neural network (DNN) model is a parametric function that takes a high-dimensional input and makes useful predictions (i.e., inference), such as a classification label. Model parameters, i.e., *kernels* or *weights*, are obtained through a large number of iterations in *training* process involving data forward, error backward and gradient calculation phases. The trained model can be used to perform inference function through only the data forward phase. Compared to inference, training is much more complicated and computational intensive. Hence, typically training is offloaded to high-end CPUs/GPUs and then the trained models are deployed to end/user devices. It is a natural need to have multi-accelerator architectures specialized for DNN training.

Given the high complexity of modern DNN models, find-

2378-203X/20/$31.00 ©2020 IEEE
DOI 10.1109/HPCA47549.2020.00036

ing the best distribution of computations on multiple accelerators is nontrivial. Moreover, due to the inherent performance difference of accelerators and the discrepancy of the communication bandwidth between them, ensuring high throughput and balanced execution is extremely challenging. The problem can be formulated as partitioning the model and data tensors among accelerators to enable parallel processing. There are two approaches: *data parallelism*, where each accelerator replicates the model and processes different input data in parallel before applying the calculated gradients to update the model; and *model parallelism*, where each accelerator keeps part of the model and performs a part of computation based on the same input. The choice between the two parallelism configurations affects the overall performance since it incurs different communication patterns between the accelerators.

The current solutions to this problem are either purely *empirical* or *incomplete* — both lacking optimal guarantee. For example, for a given DNN, "One Weird Trick" (OWT) [107] empirically suggests to use data parallelism for convolutional (CONV) layers and model parallelism for fully-connected (FC) layers. HYPAR [108] proposes a principled approach to search for the optimized parallelism configuration to minimize data communication. Although it can achieve a much better result than OWT, HYPAR suffers from several limitations: 1) the search is based on an incomplete design space; 2) it can only handle DNN architectures with linear structure; 3) it lacks a cost model and uses only communication as the proxy for performance optimization; and most importantly 4) it assumes an *homogeneous* execution environment — the performance of each accelerator and the bandwidth between them are all the same. A truly optimal solution for this critical problem still does not exist yet.

The combination of model and data parallelism is explored in deep learning accelerator architectures [60, 90] multi-GPU training systems [107, 116–119]. Recursive methods [74, 118] are proposed for tensor partitioning on multiple devices and dynamic programming methods [116, 118] are proposed for tensor partitioning layer-wisely. Inspired by those previous works [74, 107, 116–119], we present ACCPAR — a principled and systematic method of determining the tensor partition among heterogeneous accelerator arrays. Our solution is composed of several key innovations. First, we consider a *complete tensor partition space* in all three dimensions: batch size, input data size, and output data size. Hence, our solution is able to reveal previously unknown parallelism configuration. The completeness and optimality of the searching algorithm is also guaranteed. Second, in order to better optimize performance, we propose a *cost model* considering both computation and communication cost of a heterogeneous execution environment, instead of using communication as the proxy for performance as in HYPAR. Third, ACCPAR offers flexible tensor partition ratio between the accelerators to match their unique computing power and network bandwidth. Finally, we propose a technique to handle the emerging multi-path patterns in modern DNNs such as ResNet [113]. ACCPAR significantly outperform the state-of-the-art solutions, offering the *first complete solution* for tensor partitioning on heterogeneous accelerator arrays.

We simulate ACCPAR on a heterogeneous accelerator array composed of both TPU-v2 and TPU-v3 accelerators for training of large-scale DNN models such as Alexnet [112], Vgg series [120] and Resnet series [113]. The average performance of "one weird trick" (OWT) [107], HYPAR [108] and ACCPAR, normalized to the baseline data parallelism on the heterogeneous accelerator array is 2.98×, 3.78×, 6.30×, respectively. For Vgg series, ACCPAR can achieve a speedup up to 16.14×, while the highest speedup of OWT and HYPAR are 8.24× and 9.46×, respectively. For Resnet series, ACCPAR can achieve performance speedup from 1.92× to 2.20×, while the ranges of speedup achieved by OWT and HYPAR are 1.22× to 1.38× and 1.03× to 1.04×, respectively.

Notation	Description
$\mathbf{F}_l$	Input feature map to Layer l (output feature map of Layer $l-1$).
$\mathbf{E}_{l+1}$	Input error to Layer l (output error of Layer $l+1$).
$\mathbf{W}_l$	Kernel of Layer l.
$\triangle\mathbf{W}_l$	Gradient of Kernel in Layer l.
B	Mini-batch size.
$D_{i,l}$	Input data size(channel number) of Layer l.
$D_{o,l}$	Output data size(channel number) of Layer l.
c_i	The computation density of Accelerator i.
$p_{i,l}$	The partitioning of Accelerator i at Layer l.
b_i	The network bandwidth of Accelerator i.
$\mathbb{A}(\cdot)$	Function to return the size of a tensor.
$\mathbb{C}(\cdot)$	Function to return the amount of computation to be performed by an accelerator.
α, β	Partitioning ratios.

Table 1: Notations and descriptions.

2. BACKGROUND AND MOTIVATION

2.1 DNN Training

DNN training involves three tensor computing phases at each layer: *forward*, *backward* and *gradient*. The notations and descriptions are listed in Table 1. In the *forward* phase, at layer l, the input feature map tensor ($\mathbf{F}_l$) from a previous layer and the kernel/weight tensor ($\mathbf{W}_l$) are multiplied in fully-connected (FC) layers or convolved in convolutional (CONV) layers to generate the output feature map tensor, which is used as the input feature map tensor for the next layer ($\mathbf{F}_{l+1}$). Usually a non-linear activation $f(\cdot)$ is performed on each scalar of the feature map. Thus, the forward phase can be represented as $\mathbf{F}_{l+1} = f(\mathbf{F}_l \otimes \mathbf{W}_l)$, where $\otimes$ is either multiplication or convolution. In the *backward* phase, at layer l, the error tensor ($\mathbf{E}_l$) is computed by $\mathbf{E}_l = \left(\mathbf{E}_{l+1} \otimes \mathbf{W}_l^\top\right) \odot f'(\mathbf{F}_l)$, where $\mathbf{E}_{l+1}$ is the error tensor from layer $l+1$, $\odot$ is an element-wise multiplication, and $f'(\cdot)$ is the derivative function of $f(\cdot)$. In the *gradient* phase, the gradient to the kernel/weight is computed by $\triangle\mathbf{W}_l = \mathbf{F}_l^\top \otimes \mathbf{E}_{l+1}$.

The three tensor computation phases capture the common flow in many popular training algorithms, such as Gradient Descent, Stochastic Gradient Descent, Mini-batch Gradient Descent, Momentum [121] and Adaptive Moment Estimation (Adam) [122]. For example, Momentum method updates the parameter using $v_t = \gamma \cdot v_{t-1} + \eta \cdot \nabla_\theta J(\theta), \theta = \theta - v_t$, where θ is the parameter (weight), $\nabla_\theta J(\theta)$ is the gradient of a loss function $J(\cdot)$ with respect to θ, v is the velocity to record the historic gradient, γ is the momentum hyper parameter and η is the learning rate, respectively.

Segment	DNN Accelerators & Architectures
Neuro Co-processor	SpiNNaker [16], Neuromorphic Acc. [17], TrueNorth [18–20], MT-spike [21], PT-spike [22]
DNN Co-processor	NPU [23], DianNao-family [24–27], Cambricon [28], Cambricon-x [29], TPU [30, 31], ScaleDeep [32], Stripes [33], Neural Cache [123, 124], Diffy [125]
FPGA	FPGA-DCNN [34], Embedded-FPGA-CNN [35], FPGA-Exploration [36], OpenCL-FPGA-CNN [37] [126], Caffeine [38], DeepBurning [39], FPGA-DPCNN [40], TABLA [41], DNNWEAVER [42], FP-DNN [43], FPGA-LSTM [44], ESE [45], FPGA-Dataflow [46], FPGA-BNN [47], FPGA-Utilization [48], FFT-CNN [49], VIBNN [50], iSwitch [127], Shortcut-Mining [128], FA3C [129], E-RNN [130]
Dataflow	Neuflow [51], Eyeriss [52–54], Flexflow [55], Fused-CNN [56], CNN-Paritition [57], GANAX [58], UCNN [59], TANGRAM [131], Sparse-Systolic [132], MAERI [133]
PIM	Neurocube [60], XNOR-POP [61], DRISA [62], 3DICT [63], NAND-NET [134], SCOPE [135], Promise [136]
Light Models	EIE [64], SC-DCNN [65], SCNN [66], Escher [67], LookNN [68], Bit-Pragmatic DNN [69], Bit Fusion [70], Cnvlutin [99], TIE [137], Laconic [138], ADM-NN [139], Gist [140]
Co-Design	DPS-CNN [71], Minerva [72], MoDNN [73], MeDNN [74], AdaLearner [75], Stitch-X [76], PIM-DNN [77], Scalpel [78], CirCNN [79], CoSMIC [80], SnaPEA [81], OLAcce [82], Prediction-based DNN [83], PERMDNN [84], MnnFast [141], DNN Computation Reuse [142], vDNN [143], Compressing-DMA-Engine [144], AxTrain [145], Eager pruning [146], Bit-Tactical [147], GENESYS [148]
Emerging Tech.	TETRIS [85], RENO [88], PRIME [86], ISAAC [87], Memristive Boltzmann Machine [89], PipeLayer [90], Atom-layer [91], ReCom [92], ReGAN [93], ReRAM ACC. [94], EMAT [95], ReRAM-BNN [96], ZARA [97], SNrram [98], Sparse ReRAM Engine [149], RedEye [150], Quantum-SC-NN [151], FloatPIM [152], PUMA [153], FPSA [154]
Toolset/Framework	Data Parallelism [106], OWT [107], NEUTRAMS [100], Perform-ML [101], Adaptive-Classifier [102], DNNBuilder [103], Group Scissor [104], FFT-CNN [105], HyPar [108], NNest [155]

Table 2: Landscape of DNN accelerators (*accelerators highlighted in cyan are designed for training*).

2.2 Deep Learning Accelerator Architectures

Domain-specific computing architectures [156–159] are considered as a promising solution to accommodate the ever-growing intensive computing in various deep learning applications. This view has also been confirmed by a flurry of DNN accelerators [16–110] that have emerged in recent years. Compared with general-purposed CPUs/GPUs, these custom architectures achieved better performance and higher energy efficiency.

As summarized in Table 2, many designs include a vertical integration practice across algorithm and hardware levels [71–84] where DNN models are typically optimized prior to being deployed for inference. Some designs investigate the dataflow (or data reuse pattern) in DNN workloads [51–59], among which Eyeriss [52–54] is a representative design that explores many data reuse opportunities existed in DNN execution. Processing-in-memory (PIM) based designs are also proposed to reduce costly off-chip memory accesses [60–63]. Lightweight models are also introduced [64–70] to reduce computational effort. Many designs based on emerging memory technologies, such as resistive random access memory (ReRAM) technology [86–98, 160] with 3D stacking technology [60, 63, 85], are also proposed.

2.3 Motivation

Although domain-specific architectures have effectively addressed the challenges of the ending of Moore's law [161], the recent study of chip specialization [111] has clearly demonstrated the diminishing specialization returns and the ultimate *accelerator wall*. In other words, it is unlikely to further achieve fine-grained optimizations on a single accelerator. To satisfy the computation and memory requirement for large DNN models and datasets that typically cannot be satisfied by a single accelerator, a natural solution is coarse-gained DNN execution on an accelerator array. On the other hand, as highlighted in cyan in Table 2, only a few of the existing DNN accelerators are designed for training. Among these designs, strict constraints are often applied to the models that can be supported. For example, [106–108] only considered *homogeneous* platforms, where the computation capability and the network bandwidth for each accelerator are identical. In reality, however, it is more important to explore solutions for an array of *heterogeneous* accelerators with various computation capacity and network bandwidth. For example, though a more powerful TPU-v3 was released, the early deployed TPU-v2 may not retire immediately considering the deployment cost and the need for supporting various acceleration workloads. They are in fact both available off-the-shelf [162]. It is important to optimize large-scale DNN training acceleration when both TPU versions are used simultaneously. To achieve high throughput and balanced execution, we need to efficiently distribute data and model tensors among accelerators with the awareness of heterogeneous computation capability and network bandwidth. A principled and systematic approach is needed to overcome the challenge of handling the complexity of DNN models and heterogeneous hardware execution environment,

3. TENSOR PARTITIONING SPACE

Compared with DNN inference, DNN training is more complicated because of the three computation phases involved in training, i.e., forward, backward and gradient. The tensors and computations in the three phases are closely coupled together. We need first construct a complete set of the basic tensor partitioning types.

3.1 Problem Statement

We first consider FC layers and later show that the solution can be naturally extended to CONV layers. In FC layers, DNN training involves three tensor computing phases:

Forward: $\mathbf{F}_{l+1} = f(\mathbf{F}_l \times \mathbf{W}_l)$,

Backward: $\mathbf{E}_l = \left(\mathbf{E}_{l+1} \times \mathbf{W}_l^\top\right) \odot f'(\mathbf{F}_l)$,

Gradient: $\triangle \mathbf{W}_l = \mathbf{F}_l^\top \times \mathbf{E}_{l+1}$.

Using the notations in Table 1, the shape of the tensors in the above three phases are as below. Here we do not include the element-wise multiplications $\odot$ in the space relations

since they can be performed in place.

Forward: $(B, D_{o,l}) \leftarrow (B, D_{i,l}) \times (D_{i,l}, D_{o,l})$,
Backward: $(B, D_{i,l}) \leftarrow (B, D_{o,l}) \times (D_{o,l}, D_{i,l})$,
Gradient: $(D_{i,l}, D_{o,l}) \leftarrow (D_{i,l}, B) \times (B, D_{o,l})$.

For illustration purpose, this section considers a simple case of an array with two accelerators. The *problem* is to *exhaustively and systematically enumerate all possible partitions* of the tensors involved in the three phases among the two accelerators, and understand the corresponding data communication and replication requirements. This is critical because the partition will determine the communication between the accelerators and affect overall training performance. We will also explain why the current solutions [107, 108] failed to provide a complete and comprehensive solution.

3.2 Partitioning in Three Dimensions

We note that the two matrices in each of the pairs ($\mathbf{F}_l$,$\mathbf{E}_l$) and ($\mathbf{F}_{l+1}$,$\mathbf{E}_{l+1}$) have the same shape. We assume that $\mathbf{F}_l$ and $\mathbf{E}_l$ (also $\mathbf{F}_{l+1}$ and $\mathbf{E}_{l+1}$) are partitioned in the same manner. This constraint is intuitive since otherwise additional communication will be unnecessarily incurred, contradicting our goal of minimizing communication between the accelerators.

We see only *three dimensions* appear in the three tensor computing phases: B (batch size), $D_{o,l}$ (output data size of layer l), and $D_{i,l}$ (input data size of layer l). Therefore, we can naturally focus on the partition in these three dimensions. For the partitions in one dimension, we assume that the same partition parameter is used for this dimension in every tensor to avoid additional communication.

Key observation: The dimensions are *not independent*. In fact, only *one* dimension can be "free" in a partition.

We explain observation using an example: consider the forward phase and the partition in B dimension, Since we will have only two partitions, for $(B, D_{i,l})$ ($\mathbf{F}_l$), the $D_{i,l}$ dimension should not be partitioned. This also determines that $(D_{i,l}, D_{o,l})$ ($\mathbf{W}_l$) should not be partitioned in $D_{i,l}$ dimension, otherwise the matrix multiplication cannot be performed. The only case left is the $D_{o,l}$ dimension of $\mathbf{W}_l$. Suppose we partition that, the combination of multiplication of the local partitions in each accelerator does not lead to a complete result of $\mathbf{F}_l$ with shape $(B, D_{i,l})$. Specifically, depending on the partition, only the upper left and lower right sub-matrix, or upper right and lower left sub-matrix are computed. Therefore, $D_{o,l}$ dimension of $\mathbf{W}_l$ cannot be partitioned. In fact, the whole $\mathbf{W}_l$ needs to be replicated on the two accelerators to compute the complete $\mathbf{F}_l$. The other scenarios can be considered similarly.

With the assumption that $\mathbf{F}_l$ and $\mathbf{E}_l$ (also $\mathbf{F}_{l+1}$ and $\mathbf{E}_{l+1}$) use the same partition, and the fact that only one dimension is free in a partition, there are only three partition types. We discuss them one by one in the following.

3.2.1 Type-I: Partitioning B Dimension

In Type-I, we partition the B dimension in the three tensor computing phases, as shown in Figure 1(a).

In forward phase, $\mathbf{F}_{l+1} = \mathbf{F}_l \times \mathbf{W}_l$. The element $\mathbf{F}_{l+1}[b, qo]$ in $\mathbf{F}_{l+1}$ can be computed as

$$\mathbf{F}_{l+1}[b, qo] = \sum_{qi \in \{1, \cdots, D_{i,l}\}} \mathbf{F}_l[b, qi] \times \mathbf{W}_l[qi, qo], \quad (1)$$

where $b \in \{1, 2, \cdots, B\}, qo \in \{1, 2, \cdots, D_{o,l}\}$. We assume the ratio of computation to be assigned to one accelerator is α, $0 \leq \alpha \leq 1$, and the ratio for the other is β, $\beta = 1 - \alpha$. The set $\{1, 2, \cdots, B\}$ is partitioned into two subsets $\{1, 2, \cdots, \alpha B\}$ and $\{\alpha B + 1, 2, \cdots, B\}$. Specifically, $\mathbf{F}_l[1 : \alpha B, :]$ is assigned to one accelerator and $\mathbf{F}_l[\alpha B + 1 : B, :]$ is assigned to the other. The two accelerators process disjoint subsets of the batch, and perform the computation indexed by the corresponding b of the two subsets.

As discussed before, to ensure the validity of matrix multiplication and to get the complete results of $\mathbf{F}_{l+1}$, $\mathbf{W}_l$ is *replicated* in the two accelerators. After matrix multiplication, each accelerator produces a portion of results based on the same partition in B dimension. Specifically, $\mathbf{F}_{l+1}[1 : \alpha B, :]$ is produced by one accelerator and $\mathbf{F}_{l+1}[\alpha B + 1 : B, :]$ is produced by the other.

In backward, an element $\mathbf{E}_l[b, qi]$ in $\mathbf{E}_l$ can be computed as

$$\mathbf{E}_l[b, qi] = \sum_{qo \in \{1, \cdots, D_{o,l}\}} \mathbf{E}_{l+1}[b, qo] \times \mathbf{W}_l^\top[qo, qi]. \quad (2)$$

Due to the constraint that $\mathbf{F}_l$ and $\mathbf{E}_l$ (also $\mathbf{F}_{l+1}$ and $\mathbf{E}_{l+1}$) use the same partition, one accelerator keeps $\mathbf{E}_{l+1}[1 : \alpha B, :]$ and produces $\mathbf{E}_{l+1}[1 : \alpha B, :]$ with replicated $\mathbf{W}_l^\top$. The other accelerator handles the other portion: $\mathbf{E}_l[\alpha B + 1 : B, :]$ and $\mathbf{E}_{l+1}[\alpha B + 1 : B, :]$.

The common pattern for forward and backward phases is that after replicating $\mathbf{W}_l$, accelerators can perform computation locally and produce disjoint parts of the result matrix, which can be combined to obtain the complete result. However, this pattern does not exist in gradient phase as the two accelerators are not able to complete the computation individually. In gradient phase, an element $\triangle\mathbf{W}_l[qi, qo]$ in $\triangle\mathbf{W}_l$ can be computed as

$$\triangle\mathbf{W}_l[qi, qo] = \sum_{b \in \{1, \cdots, B\}} \mathbf{F}_l^\top[qi, b] \times \mathbf{E}_{l+1}[b, qo]. \quad (3)$$

Based on the same partition of B dimension in $\mathbf{F}_l^\top$ and $\mathbf{E}_{l+1}$, the two accelerators can perform local matrix multiplications. Each of them will produce the result matrix of shape $(D_{i,l}, D_{o,l})$, the same as $\triangle\mathbf{W}_l$. To get the final results in Equation (3), element-wise additions need to be performed to combine the partial results:

$$\begin{aligned} \triangle\mathbf{W}_l[qi, qo] = & \sum_{b \in \{1, \cdots, \alpha B\}} \mathbf{F}_l^\top[qi, b] \times \mathbf{E}_{l+1}[b, qo] \\ & + \sum_{b \in \{\alpha B + 1, \cdots, B\}} \mathbf{F}_l^\top[qi, b] \times \mathbf{E}_{l+1}[b, qo]. \end{aligned} \quad (4)$$

The computation pattern in gradient phase implies that communication is needed to obtain the final results of $\triangle\mathbf{W}_l$, because one of the accelerators needs to perform the partial sum. We call it as *intra-layer* communication. We can see that such communication happens at different phases for different types of partition.

3.2.2 Type-II: Partitioning $D_{i,l}$ Dimension

The partition in $D_{i,l}$ dimension is shown in Figure 1(b). To perform matrix multiplication, $D_{i,l}$ dimension of $\mathbf{F}_l$ is partitioned in the same way. Based on this, in forward phase, each

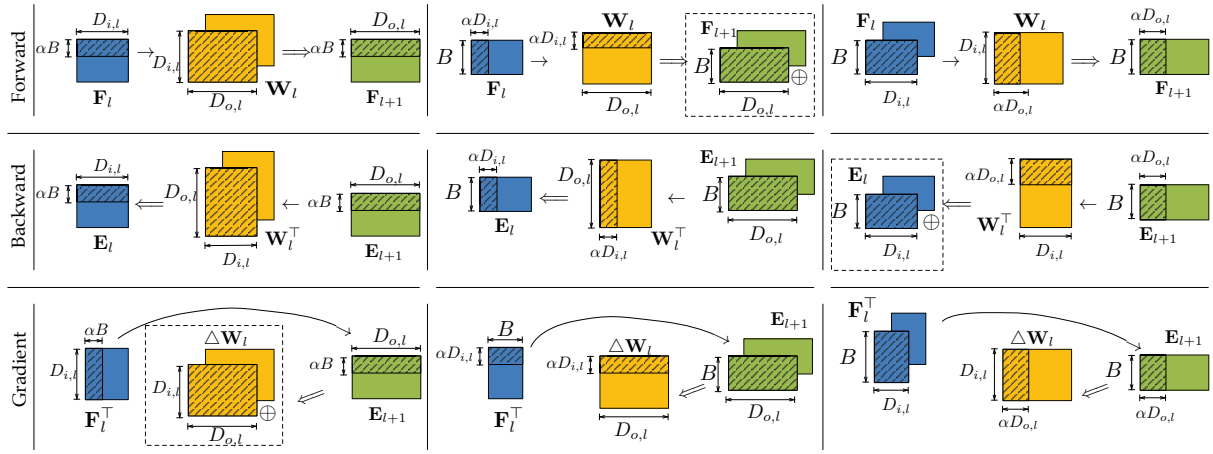

Figure 1: Three basic tensor partitioning types. The partitioning ratio for one accelerator is α and the partitioning ratio for the other accelerator is $\beta = 1 - \alpha$. Shadow tensors are assigned to one accelerator and non-shadow tensors are assigned to the other accelerator. $\otimes$ denotes element-wise addition of two tensors.

accelerator will compute a result matrix of shape $(B, D_{o,l})$. Similar to the case of $\triangle \mathbf{W}_l$ in Type-I, computing the complete $\mathbf{F}_{l+1}$ requires element-wise addition in this partitioning:

$$\mathbf{F}_{l+1}[b, qo] = \sum_{qi \in \{1, \cdots, \alpha D_{i,l}\}} \mathbf{F}_l[b, qi] \times \mathbf{W}_l[qi, qo] + \sum_{qi \in \{\alpha D_{i,l}+1, \cdots, D_{i,l}\}} \mathbf{F}_l[b, qi] \times \mathbf{W}_l[qi, qo]. \quad (5)$$

Since $\mathbf{F}_{l+1}$ and $\mathbf{E}_{l+1}$ follow the same partition, in backward phase, $\mathbf{E}_{l+1}$ is replicated in the two accelerators. This allows each of the accelerators produces a disjoint part of result of $\mathbf{E}_l$. The partition and replication are similar to that used in gradient phase. A key difference between Type-I and Type-II is that the intra-layer communication incurs at forward phase, instead of gradient phase.

3.2.3 *Type-III: Partitioning $D_{o,l}$ Dimension*

The partitioning $D_{o,l}$ dimension is shown in Figure 1(c). $\mathbf{F}_l$ needs to be replicated to compute complete $\mathbf{F}_{l+1}$ in forward phase. It is the case overlooked by all previous solutions. Essentially, it means that the input feature maps of the same batch are *replicated* into the two accelerators, instead of partitioning B. It may sound not intuitive since we want to have accelerators process the *same* data. However, we show that it is an important partition in the design space that presents the same trade-off in terms of communication just as in Type-I and Type-II.

Similar to gradient phase of Type-I and forward phase of Type-II, the backward phase of Type-III requires element-wise addition of partial results:

$$\mathbf{E}_l[b, qi] = \sum_{qo \in \{1, \cdots, \alpha D_{o,l}\}} \mathbf{E}_{l+1}[b, qo] \times \mathbf{W}_l^\top[qo, qi] + \sum_{qo \in \{\alpha D_{o,l}+1, \cdots, D_{o,l}\}} \mathbf{E}_{l+1}[b, qo] \times \mathbf{W}_l^\top[qo, qi]. \quad (6)$$

3.3 Extension to CONV

In the previous sessions, we used matrix-matrix multiplication in FC to illustrate the three types of partitions, which can be conceptually visualized in Figure 1. For CONV, the three partitioning types are still valid. However, $\mathbf{F}_l[b, qi]$, $\mathbf{F}_{l+1}[b, qo]$, $\mathbf{E}_l[b, qi]$ and $\mathbf{E}_{l+1}[b, qo]$ are no longer scalars but are 2-dimensional matrices. Therefore, $\mathbf{F}_l$, $\mathbf{F}_{l+1}$, $\mathbf{E}_l$ and $\mathbf{E}_{l+1}$ are 4-dimensional tensors, i.e., (batch, channel)$\times$(width, height). Similarly, $\triangle \mathbf{W}_l[qi, qo]$ and $\mathbf{W}_l[qi, qo]$ are also 2-dimensional matrices rather than scalars. Thus, $\triangle \mathbf{W}_l$ and $\mathbf{W}_l$ are 4-dimensional tensors, i.e., (input channel, output channel)$\times$(kernel width, kernel height). The multiplication ($\times$) in Equation (1), (2), (3), (4), (5), (6) then become convolution ($\otimes$). The additional dimensions (4D vs. 2D) and more complex operations ($\times$ vs. $\otimes$) only imply different amount of computation but not affect the partition types based on existing dimensions (B, $D_{i,l}$, and $D_{o,l}$).

3.4 Completeness

In the three phases, only three dimensions appear and we have shown that only one dimension can be partitioned at a time. Thus, the three types we derived constitute the *complete partition space*. Table 3 summarized the key features of the partitions. The *LHS Shape* and *RHS Shape* respectively indicate the shapes of the metrics on the left-hand and right-hand side of the equation for each phase. The *Psum Shape* is the shape of the matrices containing partial results in two accelerators that need to be combined using element-wise additions. It happens when the matrix appear on the LHS. This is also the shape of the matrix that needs to be replicated if it appears on the RHS. From Table 3, we can observe a *rotational symmetry* on each column.

3.5 Problems of "One Weird Trick" & HyPar

Two solutions were recently proposed to address the same problem that is addressed by this paper, — communication and parallelism between accelerators. However, neither of these two solutions is complete.

Krizhevsky [107] proposed "one weird trick" (OWT) to

Multiplication	L Shape	R Shape	Partition Dim	Psum Shape	Basic Type
$\mathbf{F}_{l+1} = \mathbf{F}_l \times \mathbf{W}_l$	$(B, D_{o,l})$	$(B, D_{i,l})$, $(D_{i,l}, D_{o,l})$	$D_{i,l}$	$(B, D_{o,l})$	Type-II
$\mathbf{E}_l = \mathbf{E}_{l+1} \times \mathbf{W}_l^\top$	$(B, D_{i,l})$	$(B, D_{o,l})$, $(D_{i,l}, D_{o,l})$	$D_{o,l}$	$(B, D_{i,l})$	Type-III
$\triangle \mathbf{W}_l = \mathbf{F}_l^\top \times \mathbf{E}_{l+1}$	$(D_{i,l}, D_{o,l})$	$(B, D_{i,l})$, $(B, D_{o,l})$	B	$(D_{i,l}, D_{o,l})$	Type-I

Table 3: Rotational Symmetry of the Three Tensor Multiplications.

configure CONV layers with data parallelism and FC layers with model parallelism to get a higher performance. It is certainly better than just using data parallelism for all layers, however it does not provide any insight on why this trick works and whether it is the best we can do. Therefore, this solution is fundamentally empirical.

HyPar [108] is a more recent and principled approach to optimize the parallelism configurations also by partitioning the layers based on the intra-layer and inter-layer communication. However, it only considers the same two basic partitions in OWT, — data parallelism and model parallelism. In fact, they correspond to Type-I and Type-II in Figure 1, respectively. Therefore, the parallelism setting in HyPar is *not complete*. Even if it is based on a more systematic approach to explore the partition space, it cannot find the optimal solution based on incomplete basic partition types. Specifically, HyPar will miss one intra-layer communication pattern (Type-III) and five inter-layer communication patterns (see more details in Section 4.1). Moreover, HyPar always partitions the tensors equally, so it cannot capture the performance heterogeneity among accelerators.

4. ACCPAR COST MODEL

To search the optimal partition of layers, we propose a cost model for multiple accelerators. We consider the computation by individual accelerator and the communication between accelerators as two major affecting DNN training performance. Compared to HYPAR [108], which uses communication cost as the proxy for performance, the cost model of ACCPAR takes both communication cost E_{cm} and computation cost E_{cp} into consideration. The optimization target is to minimize overall cost.

4.1 Communication Cost Model

Assuming the network bandwidth for accelerator i is b_i, and $\mathbf{T}$ is the accessed tensor needs to be transferred from one to the other, we define the communication cost E_{cm} for the tensor transfer as

$$E_{\text{cm}} = \frac{\mathbb{A}(\mathbf{T})}{b_i}. \tag{7}$$

The tensor size $\mathbb{A}(\mathbf{T})$ is defined as the product of the lengths of all dimensions. For example, the size of a 4-by-5 matrix is 20, and the size of a kernel whose input channel is 16, kernel window width is 3, kernel window length is 3 and output channel is 32, is $4,608 = 16 \times 3 \times 3 \times 32$. Next, we will determine what the remotely-accessed tensor $\mathbf{T}$ is.

4.1.1 Intra-layer Communication Cost

As discussed in Section 3, for each of the three basic tensor partitioning types, there is one and only one computation phase requires remote accessing.

Basic Type	Intra-layer Communication Cost
Type-I	$\frac{\mathbb{A}(\mathbf{W}_l)}{b_i}$
Type-II	$\frac{\mathbb{A}(\mathbf{F}_{l+1})}{b_i}$
Type-III	$\frac{\mathbb{A}(\mathbf{E}_l)}{b_i}$

Table 4: Intra-layer communication cost of the three basic tensor partitioning types. Note that intra-layer communication cost is not dependable on the partitioning ratio α because intermediate results are accumulated locally and partial sum tensors are accessed remotely.

In Type-I, the gradient phase requires remote accessing (Equation (4)). For the accelerator whose partitioning ratio is α, for each $b \in \{1, \cdots, \alpha B\}$, the intermediate tensor size is $\mathbb{A}(\mathbf{F}_l^\top[:, b] \times \mathbf{E}_{l+1}[b, :]) = D_{i,l} \cdot D_{o,l} = \mathbb{A}(\triangle \mathbf{W}_l) = \mathbb{A}(\mathbf{W}_l)$. Those intermediate tensors ($\forall b \in \{1, \cdots, \alpha B\}$) are accumulated locally ($\sum_{qi \in \{1, \cdots, \alpha D_{i,l}\}}(\cdot)$) by the accelerator i to reduce remote accessing by the other accelerator j. Also, the accelerator j performs local accumulation $\sum_{qi \in \{\alpha D_{i,l}+1, \cdots, D_{i,l}\}}(\cdot)$. Thus, the size of the tensor remotely accessed by accelerator i from accelerator j is $\mathbb{A}(\mathbf{W}_l)$ rather than $(1-\alpha) \cdot B \cdot \mathbb{A}(\mathbf{W}_l)$. With Equation (7), we get the intra-layer communication cost for accelerator i to remotely access the partial sum tensor in accelerator j is $\frac{\mathbb{A}(\mathbf{W}_l)}{b_i}$.

For Type-II and Type-III, readers can follow the similar idea to get the intra-layer communication cost for the two basic tensor partitioning types. We list the intra-layer communication cost of the three basic tensor partitioning types in Table 4.

4.1.2 Inter-layer Communication Cost

Since each layer is assigned a basic tensor partitioning type, when switching content from one layer to the next layer, an accelerator may require remote accessing. That is the inter-layer communication. There are two tensor conversions may require remote accessing, i.e., (1) the conversion of the output feature map tensor $\mathbf{F}_{l+1}$ in layer l to the input feature map tensor $\mathbf{F}_{l+1}$ in layer $l+1$ and (2) the conversion of the output error tensor $\mathbf{E}_{l+1}$ in layer $l+1$ to the input error tensor $\mathbf{E}_{l+1}$ in layer l. As we have three basic tensor partitioning types, there are nine inter-layer communication patterns between the basic types, as shown in Figure 2. Tensors in layer l are colored in green and Tensors in layer $l+1$ are colored in blue.

(a) Type-I to Type-I, (f) Type-II to Type-III and (h) Type-III to Type-II. In the tensor conversion of the three patterns, since the (green) tensors in layer l and the (blue) tensors in layer $l+1$ has the same partitioning, there is no conversion, and the inter-layer communication cost is 0.

(c) Type-I to Type-III, (d) Type-II to Type-I, (e) Type-

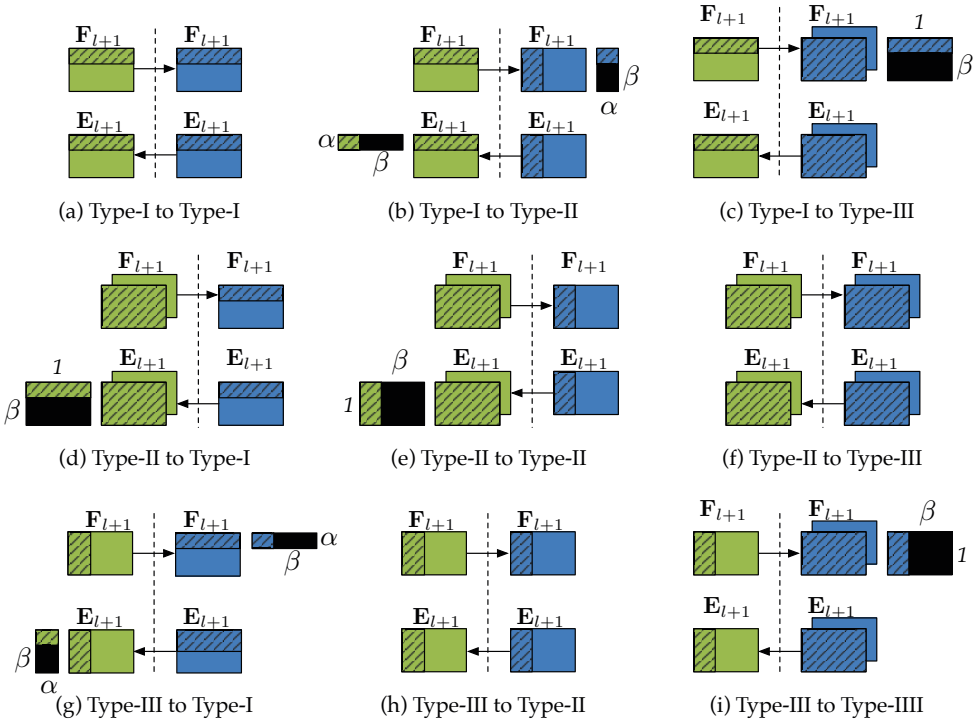

Figure 2: Inter-layer communication patterns between three basic tensor partitioning types. Shadow tensors are held by one accelerator (whose partitioning ratio is α) and non-shadow tensors are held by the other accelerator (whose partitioning ratio is β).

		Layer $l+1$		
		Type-I	Type-II	Type-III
Layer l	Type-I	0	$\frac{\alpha\beta\mathbb{A}(\mathbf{F}_{l+1})+\alpha\beta\mathbb{A}(\mathbf{E}_{l+1})}{b_i}$	$\frac{\beta\mathbb{A}(\mathbf{F}_{l+1})}{b_i}$
	Type-II	$\frac{\beta\mathbb{A}(\mathbf{E}_{l+1})}{b_i}$	$\frac{\beta\mathbb{A}(\mathbf{E}_{l+1})}{b_i}$	0
	Type-III	$\frac{\alpha\beta\mathbb{A}(\mathbf{F}_{l+1})+\alpha\beta\mathbb{A}(\mathbf{E}_{l+1})}{b_i}$	0	$\frac{\beta\mathbb{A}(\mathbf{F}_{l+1})}{b_i}$

Table 5: Inter-layer communication cost between the three basic tensor partitioning types.

II to Type-II and (i) Type-III to Type-III. We take Figure 2(c) as an example. In the tensor conversion from Type-I to Type-III, in the forward phase, the accelerator i (whose partitioning ratio is α) holds green shadow tensor $\mathbf{F}_{l+1}$ in layer l, but in the next layer $l+1$, the accelerator need the whole blue shadow tensor $\mathbf{F}_{l+1}$. The difference is the black part, and the black tensor incurs remote accessing to the other accelerator j (whose partitioning ratio is β). Thus the inter-layer communication amount is $(\beta B)\times D_{o,l}=\beta\mathbb{A}(\mathbf{F}_{l+1})$, and the inter-layer communication cost by accelerator i to remotely access the black tensor in accelerator j is $\frac{\beta\mathbb{A}(\mathbf{F}_{l+1})}{b_i}$. Reversely, the inter-layer communication cost for the accelerator j is $\frac{\alpha\mathbb{A}(\mathbf{F}_{l+1})}{b_j}$ in this case. Note that the inter-layer communication cost for (c) Type-I to Type-III, (d) Type-II to Type-I, (e) Type-II to Type-II and (i) Type-III to Type-III are the same, but the shapes of conversion tensors are not the same.

(b) Type-I to Type-II and (g) Type-III to Type-I. We take Figure 2(b) as an example. In the tensor conversion from Type-I to Type-II, in the forward phase, the accelerator i (whose partitioning ratio is α) holds green shadow tensor $\mathbf{F}_{l+1}$ $(\alpha B, D_{o,l})$ in layer l, but in the next layer $l+1$, the accelerator need the whole blue shadow tensor $\mathbf{F}_{l+1}$ $(B, \alpha D_{o,l})$. The difference is the black part, and again the black tensor incurs remote accessing to the other accelerator j (whose partitioning ratio is β). Thus the inter-layer communication amount is $(\beta B)\times\alpha D_{o,l}=\alpha\beta\mathbb{A}(\mathbf{F}_{l+1})$, and the inter-layer communication cost by accelerator i to remotely access the black tensor in accelerator j is $\frac{\alpha\beta\mathbb{A}(\mathbf{F}_{l+1})}{b_i}$. Reversely, the inter-layer communication cost for the accelerator j is $\frac{(1-\alpha)(1-\beta)\mathbb{A}(\mathbf{F}_{l+1})}{b_j}=\frac{\beta\alpha\mathbb{A}(\mathbf{F}_{l+1})}{b_j}$ in this case. Note that the inter-layer communication cost for (b) Type-I to Type-II and (g) Type-III to Type-I are the same, but the shapes of conversion tensors are not the same.

We list the inter-layer communication cost for the nine patterns of the tensor conversion between the three basic partitioning types in Table 5.

4.2 Computation Cost Model

We assume the tensor computation density of an accel-

Multiplication	# FLOP
$\mathbf{F}_{l+1} = \mathbf{F}_l \times \mathbf{W}_l$	$\mathbb{A}(\mathbf{F}_{l+1}) \cdot (D_{i,l} + D_{i,l} - 1)$
$\mathbf{E}_l = \mathbf{E}_{l+1} \times \mathbf{W}_l^\top$	$\mathbb{A}(\mathbf{E}_l) \cdot (D_{o,l} + D_{o,l} - 1)$
$\triangle \mathbf{W}_l = \mathbf{F}_l^\top \times \mathbf{E}_{l+1}$	$\mathbb{A}(\mathbf{W}_l) \cdot (B + B - 1)$

Table 6: The amount of floating point operations (FLOP) in the three multiplications.

erator i is c_i and the amount of floating point operations to perform the multiplication of the multiplication of two tensors $\mathbf{T}_1 \times \mathbf{T}_2$ is $\mathbb{C}(\mathbf{T}_1 \times \mathbf{T}_2)$. For an accelerator with a partitioning ratio α, the effective amount floating point operations performed is $\alpha \cdot \mathbb{C}(\mathbf{T}_1 \times \mathbf{T}_2)$. We can define the computation cost E_{cp} for an accelerator i to perform the computation as

$$E_{\text{cp}} = \frac{\alpha \cdot \mathbb{C}(\mathbf{T}_1 \times \mathbf{T}_2)}{c_i}. \tag{8}$$

The most important step to get the computation cost is to calculate the number of floating point operations (FLOP) of a tensor multiplication. In the forward phase, to get the output tensor $\mathbf{F}_{l+1}$, the number of FLOP is $(B \cdot D_{o,l}) \cdot (D_{i,l} + D_{i,l} - 1) = \mathbb{A}(\mathbf{F}_{l+1}) \cdot (D_{i,l} + D_{i,l} - 1)$. The numbers of FLOP for the three multiplications are listed in Table 6.

4.3 Discussion on Convolutions

We can easily expand the communication cost and computation cost from fully-connected layers to convolutional layers. In convolutions, $\mathbf{F}_l$, $\mathbf{F}_{l+1}$, $\mathbf{E}_l$ and $\mathbf{E}_{l+1}$ are 4-dimensional tensors, i.e., (batch, channel, height, width). We can view the four dimensional tensors as three dimensional tensors, but the third and fourth dimension is a meta dimension, i.e., (batch, channel, [height, width]). The kernel $\mathbf{W}_l$ are also 4-dimensional tensors, i.e., (input channel, output channel, kernel height, kernel width), and we can also view it as a three dimensional tensor, and the second dimension is a meta dimension, i.e., (input channel, [kernel height, kernel width], output channel). Thus, the communications costs listed in Table 4 and 5 keep the same formats.

In a matrix-matrix multiplication $\mathbf{M}_C = \mathbf{M}_A \times \mathbf{M}_B$, assume the shape of $\mathbf{M}_C$, $\mathbf{M}_A$, $\mathbf{M}_B$ is (M_C, N_C), (M_C, P), (P, N_C) respectively. The idea to to calculate the number of floating points performed is to multiply the number of output elements and the the number of floating points for each element. In the matrix-matrix multiplication, the number of output elements is $M_C \times N_C = \mathbb{A}(\mathbf{M}_C)$. For each output element, the number of multiplications performed is P and the number of additions performed is $P-1$. So the total number of FLOP is $\mathbb{A}(\mathbf{M}_C) \cdot (P + P - 1)$. To find the number of FLOP for a convolutional layers, we need only to find the number of FLOP for the convolution for one element in the output tensor because the number of elements of a tensor $\mathbf{T}_{\text{out}}$ is always $\mathbb{A}(\mathbf{T}_{\text{out}})$ no matter what the dimension it is. The number of multiplications performed is (input channel) $\times$ (kernel height) $\times$ (kernel width) and number of additions performed is ((input channel) $\times$ (kernel height) $\times$ (kernel width) - 1). Note that (input channel) is $D_{i,l}$, $D_{o,l}$ or B in the three multiplications respectively, and (kernel height) $\times$ (kernel width) is actually the 2D feature map or kernel size, i.e., the size of the feature map or kernel except the input and output channel. So for convolutional layers, the number of floating point operations is the entries in Table 6 multiply the 2D feature map or kernel size.

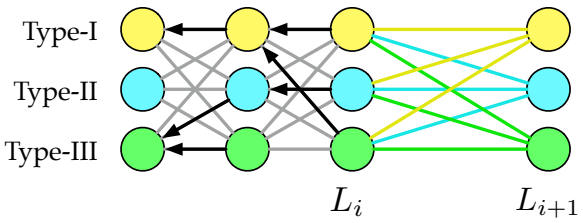

Figure 3: Layer-wise partitioning is determined by dynamic programming to minimize the computation cost and the communication cost.

5. ACCPAR PARTITIONING ALGORITHM

In this section, we explain the ACCPAR partitioning method. Like recent work HyPar [108], we determine the partitioning for each layer in a DNN model by a layer-wise dynamic partitioning scheme. However, ACCPAR is much more general for three reasons: 1) the algorithm considers the *complete* search space discussed in Section 5; 2) it can be parameterized with arbitrary partitioning ratio based on heterogeneous compute, communication cost and effective bandwidth between accelerator groups; 3) it can handle multiple paths in DNNs. As a result, we will see in Section 6 that ACCPAR achieves considerable speedups over HYPAR.

5.1 Layer-wise Partitioning

To find the best partitioning for each layer in a DNN to minimize communication and improve performance, an intuitive way is to enumerate all possible configurations by brute force. Unfortunately, it will result in a $O(3^N)$ complexity for a DNN with N layers — not a practical solution. Following the dynamic programming approaches [108, 116, 118], we reduce search complexity to $O(N)$ by dynamic programming.

Figure 3 illustrates the layer-wise partitioning procedure. For each layer, we determine the minimum cost based on the three basic partitioning types *from the first layer till the current layer*. We denote the accumulative cost up to layer L_i when it is in state (L_i, t) as $c(L_i, t)$ — layer L_i chooses a basic partitioning type $t \in \mathscr{T}$ ={Type-I, Type-II, Type-III}. Based on the cost model in Section 4, the accumulative cost given partition choice t of the current layer L_{i+1} ($c(L_{i+1}, t)$) can be calculated *inductively* with the cost of the previous layer L_i ($c(L_i, tt)$):

$$c(L_{i+1}, t) = \min_{tt \in \mathscr{T}} \{c(L_i, tt) + E_{\text{cp}}(t) + E_{\text{cm}}(tt, t)\}. \tag{9}$$

Here, $E_{\text{cp}}(t)$ is the computation cost for the current layer L_{i+1} for a type t, and $E_{\text{cm}}(tt, t)$ is the sum of the *intra-layer* communication cost for a type t and the *inter-layer* communication cost when *transition from state (L_i, tt) to state (L_{i+1}, t)*. For each basic partitioning type, during the algorithm execution we need to record the path to a previous layer for backtracking after going through all layers, shown as the black arrows in Figure 3. In this manner, after we compute the accumulative cost of the last layer, we have obtained the

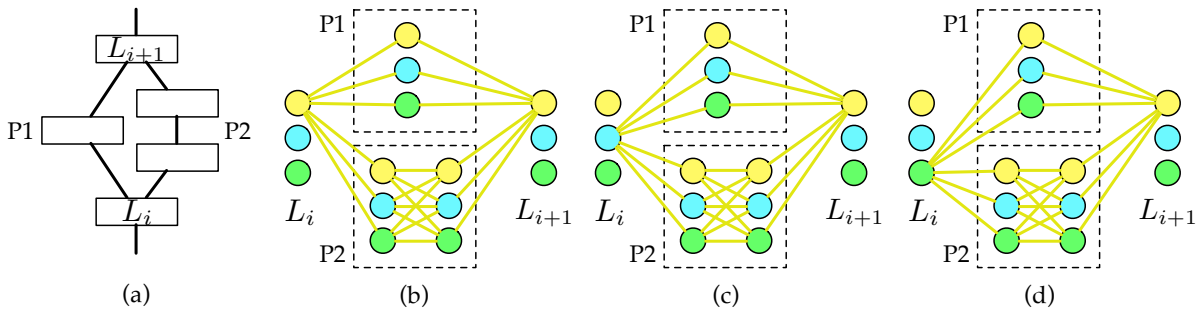

Figure 4: (a) Layer L_i and L_{i+1} are connected by path P1 and path P2. (b) Partitioning for (L_i,t = Type-I) to (L_{i+1},t = Type-I), (c) Partitioning for (L_i,t = Type-II) to (L_{i+1},t = Type-I), and (d) Partitioning for (L_i,t = Type-III) to (L_{i+1},t = Type-I).

cost state of the last layer and all previous layers with backtracking. The algorithm starts by initializing $c(L_0,t)$ for the three basic partitioning types in layer L_0 to 0.

We employ a hierarchical (recursive) partition for multiple hierarchies similar to [74, 108, 118]. The idea is to apply the layer-wise partitioning recursively on a partitioned hierarchy to partition on an accelerator array.

5.2 Handling Multiple Paths

Different from HYPAR, ACCPAR is able to determine the partition for the multi-path typologies that are common in Resnet [113]. Figure 4 shows an example where there are two paths between layer L_i and L_{i+1}. Path P1 consists of one weighted layer and Path P2 consists of two weighted layers. The key idea for multi-path partitioning is to (1) enumerate the partition state of layer L_{i+1} (L_{i+1},t), $t \in \mathcal{T}$, (2) enumerate the partition state of layer L_i (L_i,tt), $tt \in \mathcal{T}$, (3) perform individual layer-wise partitioning for each path between the two states (L_i,tt) and (L_{i+1},t) for each combination, (4) determine the lowest cost for the state (L_{i+1},t).

We need to determine the lowest cost for each state, i.e., enumerate the three colored circles in Figure 4 in Layer L_{i+1}. For example, (L_{i+1},t = Type-I) is one of the three possible states shown as the yellow circle at Layer L_{i+1} in Figure 4. We then enumerate the three colored circles at Layer L_i. Figure 4(b) starts from (L_i,t = Type-I), we search the three paths in P1 and the three paths in P2 between the two states (L_i,t = Type-I) and (L_{i+1},t = Type-I). We then select the lowest-cost path in P1, and the the lowest-cost path in P2 to (L_{i+1},t = Type-I), and combine them as the lowest-cost path from (L_i,t = Type-I) to (L_{i+1},t = Type-I). Similarly, we compute the lowest-cost path from (L_i,t = Type-II) to (L_{i+1},t = Type-I) as shown in Figure 4(c), and the lowest-cost path from (L_i,t = Type-III) to (L_{i+1},t = Type-I) as shown in Figure 4(d). With the three paths, i.e., the lowest-cost paths (1) from (L_i,t = Type-I) to (L_{i+1},t = Type-I), (2) from (L_i,t = Type-II) to (L_{i+1},t = Type-I) and (3) from (L_i,t = Type-III) to (L_{i+1},t = Type-I), we can finally determine the lowest cost to reach state (L_{i+1},t = Type-I) and record the lowest-cost path from Layer L_i to state (L_{i+1},t = Type-I). Following the similar procedure, we can determine the lowest-cost path to state (L_{i+1},t = Type-II) and state (L_{i+1},t = Type-III). Since we have searched the optimal states for Layer L_{i+1}, the optimal states in the last layer of the DNN model will be satisfied.

	TPU-v2	TPU-v3
Cores	4×2	4×2
FLOPS	180T	420T
HBM Memory	64GB	128GB
Memory Bandwidth	2400GB/s	4800GB/s
# Accelerators	128	128

Table 7: The specifications of the accelerators.

5.3 Partitioning Ratio

ACCPAR allows the partition ratio to be adjusted for heterogeneous accelerators to balance the communication and computation costs of the individual accelerators For an accelerator with a partitioning ratio α, the computation and communication cost are both a function of α and a partitioning $p_{i,l}$, i.e., $E_{\text{cp}}(\alpha, p_{i,l})$ and $E_{\text{cm}}(\alpha, p_{i,l})$. To calculate the partition ratio for achieving the best performance, we need to find the ratio to balance the sum of computation cost and communication cost among two accelerator groups. From Equation (7), (8) and Table 4, 5 we can see that computation and communication cost are both linear with respect to the partition ratio: $E_{\text{cp}}(\alpha, p_{i,l}) = \alpha \cdot E_{\text{cp}}(p_{i,l})$ and $E_{\text{cm}}(\alpha, p_{i,l}) = \alpha \cdot E_{\text{cm}}(p_{i,l})$. For the accelerator with a partitioning ratio β and a partitioning $p_{j,l}$, we can get $\beta \cdot E_{\text{cp}}(p_{j,l})$ and $\beta \cdot E_{\text{cm}}(p_{j,l})$. To determine the partitioning ratio, we just need solve the linear equation

$$\begin{aligned} &\alpha \cdot E_{\text{cp}}(p_{i,l}) + \alpha \cdot E_{\text{cm}}(p_{i,l}) \\ &= \beta \cdot E_{\text{cp}}(p_{j,l}) + \beta \cdot E_{\text{cm}}(p_{j,l}). \end{aligned} \tag{10}$$

6. EVALUATION

6.1 Evaluation Setup

We use nine DNNs to evaluate ACCPAR: Lenet [163], Alexnet [112], Vgg11, Vgg13, Vgg19 [120], and Resnet18, Resnet34, Resnet50 [113]. We train Lenet on MNIST [164] dataset and other eight DNN models are trained with ImageNet [165].

We build a in-house simulator to model the performance of the accelerator array in tensor processing unit TPU-v2 and TPU-v3. In the simulation, we derive the tensor accessing traces (loading and storing) and partial sum computation (MULT and ADD) traces for the simulation and then we

calculate the time consuming for the computation and data accessing. The trace granularity for FC layer is element-wise (i.e., 1) and for CONV is kernel-wise (e.g., 3x3). While TPU-v1 is designed for DNN inference [31], TPU-v2 and TPU-v3 target DNN training. Table 7 lists the specifications for these two accelerators. An accelerator is a board that holds the processing units — 2 cores per chip and 4 chip per board. The peak floating point operations per second (FLOPS) are 180T for TPU-v2 and 420T for TPU-v3, and the memory for the two accelerators are 64GB high bandwidth memory (HBM) and 128GB HBM [162]. The memory bandwidth for TPU-v2 is 2400GB/s. Since the memory bandwidth for TPU-v3 is not available, we assume a 4800GB/s memory bandwidth for TPU-v3. For the network, the maximum data rate per core is 2Gb/s [166]. We set the network data rate for TPU-v2 as 8Gb/s and that for TPU-v3 as 16Gb/s. The number of accelerators for TPU-v2 and TPU-v3 are both 128. The data format used in the DNN training is bfloat, Google's 16-bit floating point data format for training. We also set the mini batch size to be 512.

To evaluate the effectiveness of ACCPAR, we compare it against data parallelism (DP) [106], "One Weird Trick" (OWT) [107] and HYPAR [108]. In DP, each accelerator maintains a local copy of DNN model, while training samples are partitioned among these accelerators. In OWT, the CONV layers in a DNN model is configured for data parallelism, and the FC layers are configured for model parallelism. In HYPAR, both CONV and FC layers can be configured for data parallelism or model parallelism to minimize overall communication. Data communication, operational computation and hardware heterogeneity are jointly considered in ACCPAR for collaborative optimization. Here we use DP as the baseline, and the performance and training throughput of OWT, HYPAR and ACCPAR are all normalized to the DP design.

6.2 Heterogeneous Array

We fist evaluate the performance for a heterogeneous accelerator array using same number of accelerators with different performance: 128 TPU-v2 accelerators and 128 TPU-v3 accelerators.

The normalized performance of DP, OWT and HYPAR are shown in Figure 5. The geometric mean of speedup (the throughput improvement) in DP, OWT, HYPAR and ACCPAR are 1.00×, 2.98×, 3.78×, 6.30×, respectively. For Vgg series, ACCPAR can get a speedup up to 16.14×, while the highest speedup of OWT and HYPAR are 8.24× and 9.46×. For Resnet series, ACCPAR can get speedups from 1.92× to 2.20×, while the ranges of speedup achieved by OWT and HyPar are 1.22× to 1.38× and 1.03× to 1.04×, respectively.

We see that the speedups achieved by ACCPAR is significantly higher than OWT and HYPAR. In Figure 5, on the four Vgg series, the speedups of OWT range from 5.33× to 8.25×, and the speedups of HYPAR range from 5.92× to 9.46×. However, the speedups of ACCPAR range from 9.75× to 16.14×, which are significantly higher than OWT and HYPAR. The improvements of ACCPAR come from: (1) the complete partitioning type search space, which includes all three types of tensor partitioning settings to reduce communication; and (2) the communication and computation joint optimization considering heterogeneity. With the communication and computation by different accelerators balanced by certain partitioing ratio, the idle time due to heterogeneous communication/computation capability under equal partitioning in OWT [107] and HyPar [108] are greatly alleviated.

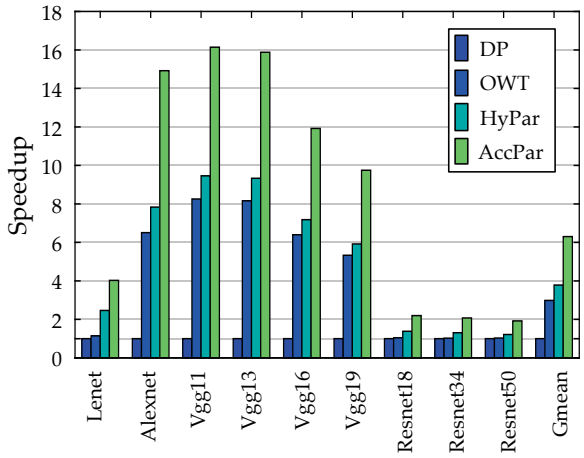

Figure 5: The speedup of data parallelism (DP), "one weird trick" (OWT) [107], HyPar [108] and AccPar in a heterogeneous accelerator array.

From Figure 5, we also notice that the the speedups of Resenet series are lower than that of Vgg series. Between the two series, the main difference is that the model sizes of Vgg series are larger than those of Resnet series, while the computation densities of Resnet series are higher than those of Vgg series. Among the three basic tensor partitioning types, Type-II and Type-III partitions the weight of layers, i.e., the model, while Type-I partitions the feature maps, i.e., the data. In data parallelism, all layers are configured by Type-I. Thus, for DNNs with a large model size, such as Vgg series, Type-II and Type-III are favored due to the potential greater reduction of communication when partitioning the model. On the other side, for DNNs with higher computation density, Type-I is favored for greater potential reduction of the communication when partitioning feature maps. Moreover, because the difference of mode sizes among different layers is smaller than the difference of feature maps, the relative benefits achieved by Type-II and Type-III are smaller than those of Type-I. The above analysis explains the reason why speedups of Resenet series are lower than that of Vgg series, since DNNs in former series mainly benefit from Type-II and Type-III — model partition, the trend is true for not only ACCPAR but also HYPAR and OWT. However, even for Resnet series, the speedups of ACCPAR are considerably higher than OWT and HYPAR: the highest speedup by OWT and HyPar on the Resnet series are 1.04× and 1.38× respectively, but the highest speedup by ACCPAR is 2.20×. This means that ACCPAR is 112% better than OWT and 59% better than HYPAR.

6.3 Homogeneous Array

We then evaluate the performance for a homogeneous accelerator array, where 128 accelerators employed are of the

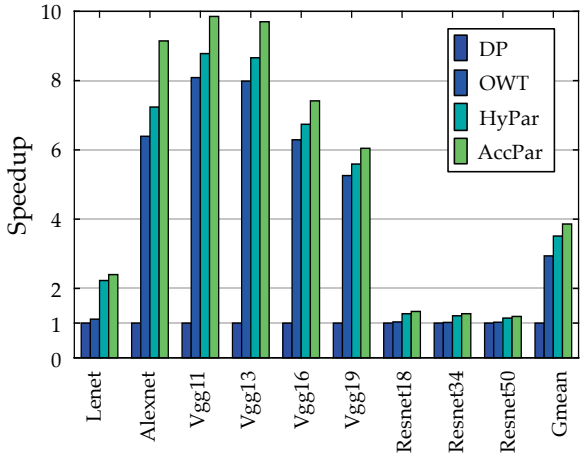

Figure 6: The speedup of data parallelism (DP), "one weird trick" (OWT) [107], HyPar [108] and AccPar in a homogeneous accelerator array.

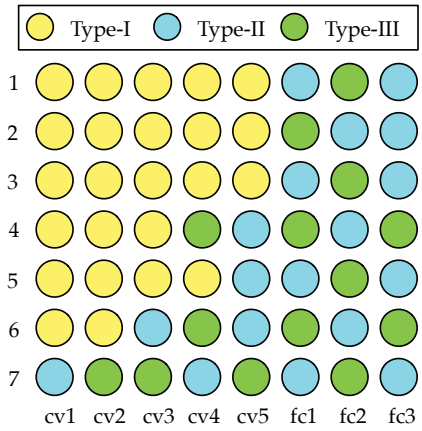

Figure 7: The selected partitioning types for the weighted layers in Alexnet. The number of hierarchies is 7 and the batch size is 128.

same type, i.e., TPU-v3. The speedups of data parallelism (DP), "one weird trick" (OWT), HYPAR and ACCPAR are shown in Figure 6. The results are normalized to data parallelism. The geometric mean of DP, OWT, HYPAR and ACCPAR are 1.00×, 2.94×, 3.51×, 3.86×, respectively. For Resent series, the DNNs are very "deep" (i.e., the number of layers is very large), and all layers except the last fully-connected layer are convolutional layers. While OWT, HYPAR, and ACCPAR all try to explore other parallelism settings or partitions rather than static data/model parallelism, we observe in the results that these three schemes evantually configure most of layers in Resnet series with data parallelism or Type-I partition. This is the reason why they achieve lower speedups on Resnet series than Vgg series. In comparison, the DNNs in Vgg series contain a variety of layers, so OWT, HYPAR, and ACCPAR can effectively explore larger search space with more diverse parallelism/partition settings. With homogeneous accelerator array, for a specific DNN model, we observe the *increasing* speedups of DP, OWT, HyPar and ACCPAR. It is the direct consequence of the increasing flexibility.

In Figure 7, we also show the selected partitioning types by ACCPAR for the weighted layers in Alexnet. In the three fully-connected layers, i.e., fc1, fc2 and fc3, Type-II and Type-III partitions are used to minimize the communication by maintaining a part of weight locally and communicating the feature maps. In the convolutional layers, i.e., cv1 to cv5, we can see that Type-I partition are mostly but *not solely* selected. ACCPAR allows all Type-I, Type-II and Type-III to be selected to reduce the total communication further. Especially, with the increase of hierarchy level, more layers are configured with Type-II or Type-III. This illustrates the importance of having a complete search space as in ACCPAR.

6.4 Scalability with Hierarchy Levels

In this section, we study the scalability of ACCPAR on various hierarchies on the heterogeneous accelerator array. Figure 8 shows the speedups of DP, OWT, HYPAR and ACCPAR on Vgg19 for hierarchy level from $h = 2$ to $h = 9$.

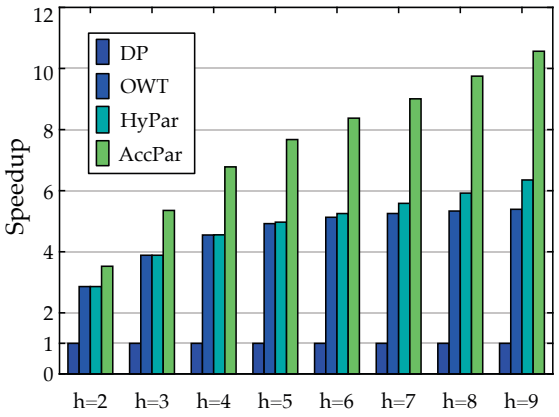

Figure 8: The speedup of data parallelism (DP), "one weird trick" (OWT) [107], HyPar [108] and AccPar under various partitioning hierarchies on Vgg19.

With the increase of hierarchy level, we partition the tensor for a finer-grained level. From Figure 8 we see that the speedups of OWT and HYPAR tend to be saturated, while the speedups of ACCPAR continue to increase with the increase of the partitioning hierarchies. For OWT, the convolutional layers are always configured with data parallelism and the fully-connected layers are always configured with model parallelism. While it provides more flexible than configuring all layers with data parallelism in DP, it is still a *static* configuration. HYPAR and ACCPAR are *dynamic* configurations because they do not statically set a class of layer to be a specific parallelism or partition, instead they dynamically explore the possible configuration to minimize the communication or cost. Compared with HYPAR, ACCPAR benefits from a *complete* partitioning type sets and the heterogeneity-aware configuration. The flexibility of the four schemes from low to high is: DP $\prec$ OWT $\prec$ HyPar $\prec$ ACCPAR. Table 8 shows the comparison of DP, OWT, HYPAR, and ACCPAR.

DP	OWT	HyPar	ACCPAR
Static	Static	Dynamic	Dynamic
Low	⟶		**High**

Table 8: The caparison of flexibility of DP, OWT, HyPar and ACCPAR.

7. CONCLUSION

In this paper we present ACCPAR, a principled and systematic method to determining the tensor partition among heterogeneous accelerator arrays for achieving optimal performance. ACCPAR considers the complete tensor partition space and reveals a new and previously unknown parallelism configuration. It optimizes performance based on a cost model considering both computation and communication cost of heterogeneous execution environment. The general search algorithm is applicable for the emerging multi-path patterns in modern DNNs such as ResNet. We simulate ACCPAR on a heterogeneous accelerator array composed of both TPU-v2 and TPU-v3 accelerators for training of large-scale DNN models such as Alexnet, Vgg series and Resnet series. The average performance of ACCPAR and previous state-of-the-art "one weird trick" (OWT) and HYPAR normalized to data parallelism in the heterogeneous accelerator array is 6.30×, 2.98×, 3.78×, respectively.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their constructive and insightful comments. This work is supported in part by NSF CCF-1910299, NSF CSR-1717885, ARO W911NF-19-2-0107 and AFRL FA8750-18-2-0121. This work is also supported by the National Science Foundation grants NSF CCF-1657333, NSF CCF-1717754, NSF CNS-1717984, NSF CCF-1750656, and NSF CCF-1919289.

8. REFERENCES

[1] S. Ren *et al.*, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *NIPS*, 2015.
[2] W. Liu *et al.*, "Ssd: Single shot multibox detector," in *ECCV*, 2016.
[3] X. Liu *et al.*, "Dpatch: An adversarial patch attack on object detectors," *arXiv:1806.02299*, 2018.
[4] B. Perozzi *et al.*, "Deepwalk: Online learning of social representations," in *KDD*, 2014.
[5] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv:1609.02907*, 2016.
[6] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *KDD*, 2016.
[7] T. Fischer and C. Krauss, "Deep learning with long short-term memory networks for financial market predictions," *European Journal of Operational Research*, 2018.
[8] X. Ding *et al.*, "Deep learning for event-driven stock prediction," in *IJCAI*, 2015.
[9] Y. Deng *et al.*, "Deep direct reinforcement learning for financial signal representation and trading," *IEEE TNNLS*, 2016.
[10] R. Miotto *et al.*, "Deep learning for healthcare: review, opportunities and challenges," *Briefings in bioinformatics*, 2017.
[11] T. Ching *et al.*, "Opportunities and obstacles for deep learning in biology and medicine," *Journal of The Royal Society Interface*, 2018.
[12] O. Faust *et al.*, "Deep learning for healthcare applications based on physiological signals: A review," *Computer methods and programs in biomedicine*, 2018.
[13] L. Song *et al.*, "Deep learning for vertex reconstruction of neutrino-nucleus interaction events with combined energy and time data," in *ICASSP*, 2019.
[14] P. Baldi *et al.*, "Searching for exotic particles in high-energy physics with deep learning," *Nature communications*, 2014.
[15] J. Wu *et al.*, "Galileo: Perceiving physical object properties by integrating a physics engine with deep learning," in *NIPS*, 2015.
[16] S. B. Furber *et al.*, "Overview of the spinnaker system architecture," *IEEE Transactions on Computers*, 2013.
[17] Z. Du *et al.*, "Neuromorphic accelerators: A comparison between neuroscience and machine-learning approaches," in *MICRO*, 2015.
[18] P. A. Merolla *et al.*, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, 2014.
[19] S. K. Esser *et al.*, "Backpropagation for energy-efficient neuromorphic computing," in *NIPS*, 2015.
[20] S. K. Esser *et al.*, "Convolutional networks for fast, energy-efficient neuromorphic computing," *Proceedings of the National Academy of Sciences (PNAS)*, 2016.
[21] T. Liu *et al.*, "Mt-spike: A multilayer time-based spiking neuromorphic architecture with temporal error backpropagation," in *ICCAD*, 2017.
[22] T. Liu *et al.*, "Pt-spike: A precise-time-dependent single spike neuromorphic architecture with efficient supervised learning," in *ASP-DAC*, 2018.
[23] H. Esmaeilzadeh *et al.*, "Neural acceleration for general-purpose approximate programs," in *MICRO*, 2012.
[24] T. Chen *et al.*, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ASPLOS*, 2014.
[25] Y. Chen *et al.*, "Dadiannao: A machine-learning supercomputer," in *MICRO*, 2014.
[26] Z. Du *et al.*, "Shidiannao: Shifting vision processing closer to the sensor," in *ISCA*, 2015.
[27] D. Liu *et al.*, "Pudiannao: A polyvalent machine learning accelerator," in *ASPLOS*, 2015.
[28] S. Liu *et al.*, "Cambricon: An instruction set architecture for neural networks," in *ISCA*, 2016.
[29] S. Zhang *et al.*, "Cambricon-x: An accelerator for sparse neural networks," in *MICRO*, 2016.
[30] "Google supercharges machine learning tasks with tpu custom chip." `https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html`.
[31] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *ISCA*, 2017.
[32] S. Venkataramani *et al.*, "Scaledeep: A scalable compute architecture for learning and evaluating deep networks," in *ISCA*, 2017.
[33] P. Judd *et al.*, "Stripes: Bit-serial deep neural network computing," in *MICRO*, 2016.
[34] C. Zhang *et al.*, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *FPGA*, 2015.
[35] J. Qiu *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *FPGA*, 2016.
[36] M. Motamedi *et al.*, "Design space exploration of fpga-based deep convolutional neural networks," in *ASP-DAC*, 2016.
[37] N. Suda *et al.*, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," in *FPGA*, 2016.
[38] C. Zhang *et al.*, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *ICCAD*, 2016.
[39] Y. Wang *et al.*, "Deepburning: Automatic generation of fpga-based learning accelerators for the neural network family," in *DAC*, 2016.
[40] C. Zhang *et al.*, "Energy-efficient cnn implementation on a deeply pipelined fpga cluster," in *ISLPED*, 2016.
[41] D. Mahajan *et al.*, "Tabla: A unified template-based framework for accelerating statistical machine learning," in *HPCA*, 2016.
[42] H. Sharma *et al.*, "From high-level deep neural models to fpgas," in *MICRO*, 2016.
[43] Y. Guan *et al.*, "Fp-dnn: An automated framework for mapping deep neural networks onto fpgas with rtl-hls hybrid templates," in *FCCM*, 2017.
[44] Y. Guan *et al.*, "Fpga-based accelerator for long short-term memory recurrent neural networks," in *ASP-DAC*, 2017.
[45] S. Han *et al.*, "Ese: Efficient speech recognition engine with sparse lstm on fpga," in *FPGA*, 2017.
[46] Y. Ma *et al.*, "Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks," in *FPGA*, 2017.
[47] R. Zhao *et al.*, "Accelerating binarized convolutional neural networks with software-programmable fpgas.," in *FPGA*, 2017.
[48] Y. Shen *et al.*, "Overcoming resource underutilization in spatial cnn accelerators," in *FPL*, 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] R. Cai *et al.*, "Vibnn: Hardware acceleration of bayesian neural networks," in *ASPLOS*, 2018.
[51] C. Farabet *et al.*, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *CVPRW*, 2011.
[52] Y.-H. Chen *et al.*, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ISCA*, 2016.
[53] Y.-H. Chen *et al.*, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE JSSC*, 2017.
[54] Y.-H. Chen *et al.*, "Using dataflow to optimize energy efficiency of deep neural network accelerators," *IEEE Micro*, 2017.
[55] W. Lu *et al.*, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *HPCA*, 2017.
[56] M. Alwani *et al.*, "Fused-layer cnn accelerators," in *MICRO*, 2016.
[57] Y. Shen *et al.*, "Maximizing cnn accelerator efficiency through resource partitioning," in *ISCA*, 2017.
[58] A. Yazdanbakhsh *et al.*, "Ganax: A unified mimd-simd acceleration for generative adversarial networks," in *ISCA*, 2018.
[59] K. Hegde *et al.*, "Ucnn: Exploiting computational reuse in deep neural networks via weight repetition," in *ISCA*, 2018.
[60] D. Kim *et al.*, "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory," in *ISCA*, 2016.
[61] L. Jiang *et al.*, "Xnor-pop: A processing-in-memory architecture for binary convolutional neural networks in wide-io2 drams," in *ISLPED*, 2017.
[62] S. Li *et al.*, "Drisa: A dram-based reconfigurable in-situ accelerator," in *MICRO*, 2017.
[63] 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.
[64] S. Han *et al.*, "Eie: efficient inference engine on compressed deep neural network," in *ISCA*, 2016.
[65] A. Ren *et al.*, "Sc-dcnn: Highly-scalable deep convolutional neural network using stochastic computing," in *ASPLOS*, 2017.
[66] A. Parashar *et al.*, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *ISCA*, 2017.
[67] Y. Shen *et al.*, "Escher: A cnn accelerator with flexible buffering to minimize off-chip transfer," in *FCCM*, 2017.
[68] M. S. Razlighi *et al.*, "Looknn: Neural network with no multiplication," in *DATE*, 2017.
[69] J. Albericio *et al.*, "Bit-pragmatic deep neural network computing," in *MICRO*, 2017.
[70] H. Sharma *et al.*, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *ISCA*, 2018.
[71] T. Na and S. Mukhopadhyay, "Speeding up convolutional neural network training with dynamic precision scaling and flexible multiplier-accumulator," in *ISLPED*, 2016.
[72] B. Reagen *et al.*, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *ISCA*, 2016.
[73] J. Mao *et al.*, "Modnn: Local distributed mobile computing system for deep neural network," in *DATE*, 2017.
[74] J. Mao *et al.*, "Mednn: A distributed mobile system with enhanced partition and deployment for large-scale dnns," in *ICCAD*, 2017.
[75] J. Mao *et al.*, "Adalearner: An adaptive distributed mobile learning system for neural networks," in *ICCAD*, 2017.
[76] C.-E. Lee *et al.*, "Stitch-x: An accelerator architecture for exploiting unstructured sparsity in deep neural networks," in *SysML*, 2018.
[77] J. Liu *et al.*, "Processing-in-memory for energy-efficient neural network training: A heterogeneous approach," in *MICRO*, 2018.
[78] J. Yu *et al.*, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," in *ISCA*, 2017.
[79] C. Ding *et al.*, "Circnn: accelerating and compressing deep neural networks using block-circulant weight matrices," in *MICRO*, 2017.
[80] J. Park *et al.*, "Scale-out acceleration for machine learning," in *MICRO*, 2017.
[81] V. Akhlaghi *et al.*, "Snapea: Predictive early activation for reducing computation in deep convolutional neural networks," in *ISCA*, 2018.
[82] E. Park *et al.*, "Energy-efficient neural network accelerator based on outlier-aware low-precision computation," in *ISCA*, 2018.
[83] M. Song *et al.*, "Prediction based execution on deep neural networks," in *ISCA*, 2018.
[84] C. Deng *et al.*, "Permdnn: Efficient compressed deep neural network architecture with permuted diagonal matrices," in *MICRO*, 2018.
[85] M. Gao *et al.*, "Tetris: Scalable and efficient neural network acceleration with 3d memory," in *ASPLOS*, 2017.
[86] P. Chi *et al.*, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *ISCA*, 2016.
[87] A. Shafiee *et al.*, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *ISCA*, 2016.
[88] X. Liu *et al.*, "Reno: a high-efficient reconfigurable neuromorphic computing accelerator design," in *DAC*, 2015.
[89] M. N. Bojnordi and E. Ipek, "Memristive boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning," in *HPCA*, 2016.
[90] L. Song *et al.*, "Pipelayer: A pipelined reram-based accelerator for deep learning," in *HPCA*, 2017.
[91] X. Qiao *et al.*, "Atomlayer: a universal reram-based cnn accelerator with atomic layer computation," in *DAC*, 2018.
[92] H. Ji *et al.*, "Recom: An efficient resistive accelerator for compressed deep neural networks," in *DATE*, 2018.
[93] F. Chen *et al.*, "Regan: A pipelined reram-based accelerator for generative adversarial networks," in *ASP-DAC*, 2018.
[94] B. Li *et al.*, "Reram-based accelerator for deep learning," in *DATE*, 2018.
[95] F. Chen and H. Li, "Emat: an efficient multi-task architecture for transfer learning using reram," in *ICCAD*, 2018.
[96] T. Tang *et al.*, "Binary convolutional neural network on rram," in *ASP-DAC*, 2017.
[97] F. Chen *et al.*, "Zara: A novel zero-free dataflow accelerator for generative adversarial networks in 3d reram," in *DAC*, 2019.
[98] P. Wang *et al.*, "Snrram: an efficient sparse neural network computation architecture based on resistive random-access memory," in *DAC*, 2018.
[99] J. Albericio *et al.*, "Cnvlutin: ineffectual-neuron-free deep neural network computing," in *ISCA*, 2016.
[100] Y. Ji *et al.*, "Neutrams: Neural network transformation and co-design under neuromorphic hardware constraints," in *MICRO*, 2016.
[101] A. Mirhoseini *et al.*, "Perform-ml: Performance optimized machine learning by platform and content aware customization," in *DAC*, 2016.
[102] Z. Takhirov *et al.*, "Energy-efficient adaptive classifier design for mobile systems," in *ISLPED*, 2016.
[103] X. Zhang *et al.*, "Dnnbuilder: an automated tool for building high-performance dnn hardware accelerators for fpgas," in *ICCAD*, 2018.
[104] Y. Wang *et al.*, "Group scissor: Scaling neuromorphic computing design to large neural networks," in *DAC*, 2017.
[105] J. H. Ko *et al.*, "Design of an energy-efficient accelerator for training of convolutional neural networks using frequency-domain computation," in *DAC*, 2017.
[106] M. Li *et al.*, "Scaling distributed machine learning with the parameter server," in *OSDI*, 2014.
[107] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv:1404.5997*, 2014.
[108] L. Song *et al.*, "Hypar: Towards hybrid parallelism for deep learning accelerator array," in *HPCA*, 2019.
[109] Y. S. Shao *et al.*, "Simba: Scaling deep-learning inference with multi-chip-module-based architecture," in *MICRO*, 2019.
[110] M. Zhu *et al.*, "Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus," in *MICRO*, 2019.
[111] A. Fuchs and D. Wentzlaff, "The accelerator wall: Limits of chip specialization," in *HPCA*, 2019.
[112] A. Krizhevsky *et al.*, "Imagenet classification with deep convolutional neural networks," in *NIPS*, 2012.
[113] K. He *et al.*, "Deep residual learning for image recognition," in *CVPR*, 2016.
[114] I. Goodfellow *et al.*, "Generative adversarial nets," in *NIPS*, 2014.
[115] A. Radford *et al.*, "Unsupervised representation learning with deep convolutional generative adversarial networks," *arXiv:1511.06434*, 2015.
[116] Z. Jia *et al.*, "Exploring hidden dimensions in parallelizing convolutional neural networks," in *ICML*, 2018.
[117] M. Wang *et al.*, "Unifying data, model and hybrid parallelism in deep learning via tensor tiling," *arXiv:1805.04170*, 2018.
[118] M. Wang *et al.*, "Supporting very large models using automatic dataflow graph partitioning," in *EuroSys*, 2019.
[119] Z. Jia *et al.*, "Beyond data and model parallelism for deep neural networks," in *SysML*, 2019.
[120] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv:1409.1556*, 2014.
[121] N. Qian, "On the momentum term in gradient descent learning algorithms," *Neural networks*, 1999.
[122] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv:1412.6980*, 2014.

[123] C. Eckert *et al.*, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *ISCA*, 2018.

[124] X. Wang *et al.*, "Bit prudent in-cache acceleration of deep convolutional neural networks," in *HPCA*, 2019.

[125] M. Mahmoud *et al.*, "Diffy: a déjà vu-free differential deep neural network accelerator," in *MICRO*, 2018.

[126] J. Zhang and J. Li, "Improving the performance of opencl-based fpga accelerator for convolutional neural network.," in *FPGA*, 2017.

[127] Y. Li *et al.*, "Accelerating distributed reinforcement learning with in-switch computing," in *ISCA*, 2019.

[128] A. Azizimazreah and L. Chen, "Shortcut mining: Exploiting cross-layer shortcut reuse in dcnn accelerators," in *HPCA*, 2019.

[129] H. Cho *et al.*, "Fa3c: Fpga-accelerated deep reinforcement learning," in *ASPLOS*, 2019.

[130] Z. Li *et al.*, "E-rnn: Design optimization for efficient recurrent neural networks in fpgas," in *HPCA*, 2019.

[131] M. Gao *et al.*, "Tangram: Optimized coarse-grained dataflow for scalable nn accelerators," in *ASPLOS*, 2019.

[132] H. Kung *et al.*, "Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization," in *ASPLOS*, 2019.

[133] H. Kwon *et al.*, "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," in *ASPLOS*, 2018.

[134] H. Kim *et al.*, "Nand-net: Minimizing computational complexity of in-memory processing for binary neural networks," in *HPCA*, 2019.

[135] S. Li *et al.*, "Scope: A stochastic computing engine for dram-based in-situ accelerator," in *MICRO*, 2018.

[136] P. Srivastava *et al.*, "Promise: An end-to-end design of a programmable mixed-signal accelerator for machine-learning algorithms," in *ISCA*, 2018.

[137] C. Deng *et al.*, "Tie: Energy-efficient tensor train-based inference engine for deep neural network," in *ISCA*, 2019.

[138] S. Sharify *et al.*, "Laconic deep learning inference acceleration," in *ISCA*, 2019.

[139] A. Ren *et al.*, "Admm-nn: An algorithm-hardware co-design framework of dnns using alternating direction methods of multipliers," in *ASPLOS*, 2019.

[140] A. Jain *et al.*, "Gist: Efficient data encoding for deep neural network training," in *ISCA*, 2018.

[141] H. Jang *et al.*, "Mnnfast: A fast and scalable system architecture for memory-augmented neural networks," in *ISCA*, 2019.

[142] M. Riera *et al.*, "Computation reuse in dnns by exploiting input similarity," in *ISCA*, 2018.

[143] M. Rhu *et al.*, "vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *MICRO*, 2016.

[144] M. Rhu *et al.*, "Compressing dma engine: Leveraging activation sparsity for training deep neural networks," in *HPCA*, 2018.

[145] X. He *et al.*, "Joint design of training and hardware towards efficient and accuracy-scalable neural network inference," *IEEE JETCAS*, 2018.

[146] J. Zhang *et al.*, "Eager pruning: algorithm and architecture support for fast training of deep neural networks," in *ISCA*, 2019.

[147] A. Delmas Lascorz *et al.*, "Bit-tactical: A software/hardware approach to exploiting value and bit sparsity in neural networks," in *ASPLOS*, 2019.

[148] A. Samajdar *et al.*, "Genesys: Enabling continuous learning through neural network evolution in hardware," in *MICRO*, 2018.

[149] T.-H. Yang *et al.*, "Sparse reram engine: Joint exploration of activation and weight sparsity in compressed neural networks," in *ISCA*, 2019.

[150] R. LiKamWa *et al.*, "Redeye: Analog convnet image sensor architecture for continuous mobile vision," in *ISCA*, 2016.

[151] R. Cai *et al.*, "A stochastic-computing based deep learning framework using adiabatic quantum-flux-parametron superconducting technology," in *ISCA*, 2019.

[152] M. Imani *et al.*, "Floatpim: In-memory acceleration of deep neural network training with high precision," in *ISCA*, 2019.

[153] A. Ankit *et al.*, "Puma: A programmable ultra-efficient memristor-based accelerator for machine learning inference," in *ASPLOS*, 2019.

[154] Y. Ji *et al.*, "Fpsa: A full system stack solution for reconfigurable reram-based nn accelerator architecture," in *ASPLOS*, 2019.

[155] L. Ke *et al.*, "Nnest: Early-stage design space exploration tool for neural network inference accelerators," in *ISLPED*, 2018.

[156] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development," *Turing Lecture*, 2018.

[157] D. Patterson, "50 years of computer architecture: From the mainframe cpu to the domain-specific tpu and the open risc-v instruction set," in *ISSCC*, 2018.

[158] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, 2019.

[159] J. Dean *et al.*, "A new golden age in computer architecture: Empowering the machine-learning revolution," *IEEE Micro*, 2018.

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

[161] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, 1965.

[162] "Ai & machine learning products cloud tpu." `https://cloud.google.com/tpu/`.

[163] Y. LeCun *et al.*, "Lenet-5, convolutional neural networks," *URL: http://yann. lecun. com/exdb/lenet*, 2015.

[164] Y. LeCun *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, 1998.

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

[166] "Vpc resource quotas." `https://cloud.google.com/vpc/docs/quota`.