

FPDeep: Scalable Acceleration of CNN Training on Deeply-Pipelined FPGA Clusters

Tianqi Wang*, Tong Geng*, Ang Li, Xi Jin, and Martin Herbordt

Abstract—Deep convolutional Neural Networks (CNNs) have revolutionized numerous applications, but the demand for ever more performance remains unabated. Scaling CNN computations to larger clusters is generally done by distributing tasks in batch mode using methods such as distributed synchronous SGD. Among the issues with this approach is that, to make the distributed cluster work with high utilization, the workload distributed to each node must be large; this implies nontrivial growth in the SGD mini-batch size. In this paper we propose a framework, called FPDeep, which uses a hybrid of model and layer parallelism to configure distributed reconfigurable clusters to train CNNs. This approach has numerous benefits. First, the design does not suffer from performance loss due to batch size growth. Second, work and storage are balanced among nodes through novel workload and weight partitioning schemes. Part of the mechanism is the surprising finding that it is preferable to store excess weights in neighboring devices rather than in local off-chip memory. Third, the entire system is a fine-grained pipeline. This leads to high parallelism and utilization and also minimizes the time that features need to be cached while waiting for back-propagation. As a result, storage demand is reduced to the point where only on-chip memory is used for the convolution layers. And fourth, we find that the simplest topology, a 1D array, is preferred for interconnecting the FPGAs thus enabling widespread applicability. We evaluate FPDeep with the Alexnet, VGG-16, and VGG-19 benchmarks. Results show that FPDeep has good scalability to a large number of FPGAs, with the limiting factor being the FPGA-to-FPGA bandwidth. But with 250 Gb/s bidirectional bandwidth per FPGA, which is easily supported by current generation FPGAs, FPDeep performance shows linearity up to 100 FPGAs. Energy efficiency is evaluated with respect to GOPs/J. FPDeep provides, on average, $6.4\times$ higher energy efficiency than comparable GPU servers.

Index Terms—Convolution Neural Network Training, Reconfigurable Computing, Parallel and Distributed Systems

1 INTRODUCTION

DEEP convolutional neural networks (CNNs) have revolutionized applications such as image classification and object recognition [28], [38], [39], [44]. But as there remains an open-ended demand for more complex networks and larger datasets, new computing solutions are critical. A challenging problem is that while large training sets are necessary for good generalization, they are also more computationally expensive. Therefore, nearly all of these neural networks are powered by the stochastic gradient descent algorithm (SGD).

Traditionally, distributed synchronous stochastic gradient descent (D-SGD) has enabled large-scale CNN training by partitioning SGD mini-batches into smaller data batches that can then be processed in parallel and so accelerate CNN training [13]. A drawback of this method is scalability: to enable continued high utilization as the number of nodes increases, each node must be allocated an ever-larger workload, which means that the mini-batch size must increase. Larger mini-batches, however, slow training convergence. Thus, while larger clusters provide increased computation capacity, the training time is not proportionally reduced [13], [21]. In [21] the authors demonstrate that increasing batch size increases improper convergence to sharp minimizers, which, in turn, gives rise to poor generalization and thus causes an increasing gap between test and training accuracy. Table 1 shows the performance of small-batch (SB) and large-batch (LB) variants of ADAM on six networks. Comparing LB and SB, we observe that LB does not decrease the accuracy derived from the training set, but does substantially reduce the testing accuracy.

Certain methods can somewhat reduce this loss of accuracy – e.g., using dynamic batch sizes and fine-tuning the learning rate – but they do not solve the problem [13]. SB limits the parallelism that can be exploited by high-end computing clusters, especially when data parallelism is used; SB is thus rarely used in large-scale training.

FPGA clusters are a competitive technology for CNN inference [7], [9], [29], [34], [35], [42], [43]. For CNN training, however, their efficacy is still an open question; one that is addressed in this work. Previous FPGA clusters for CNN training have generally worked in batch mode (*batch* in the computational sense), which uses the distributed synchronous SGD algorithm just described [8], [14], [15], [26], [31], [45]. In this approach, called *Data Parallelism* [2], each FPGA executes all layers of the CNN. This is done in sequential order, a layer at a time, with a new layer starting only after the previous layer has completed. Data Parallelism has three significant disadvantages. First, optimal FPGA configurations for different CNN layers vary greatly: either the FPGA is suboptimally configured, or the FPGA needs to be reconfigured repeatedly at run-time. Second, the storage required for weights and intermediate features is generally large enough that off-chip memory must be used. And third, this entire approach suffers from the scalability problem of the distributed synchronous SGD algorithm already described.

Another method, which we call *Layer Parallelism*, is to daisy-chain multiple FPGAs and map the entire CNN onto a single pipeline. Zhang, et al. [43] used Layer Parallelism to accelerate CNNs using FPGA clusters, but only for inference. Their approach, however, still leaves two problems. First, the pipeline is not seamless; a particular layer might

* marked authors have equal contribution and credit

TABLE 1

Performance of small-batch(SB) and large-batch(LB). Note that LB does not decrease training accuracy, but reduces the test accuracy [21]

Name	Network Type	Data Set	Training Accuracy		Test Accuracy	
			SB	LB	SB	LB
F_i	FC	MNIST	99.66% \pm 0.05%	99.92% \pm 0.01%	98.03% \pm 0.07%	97.81% \pm 0.07%
F_2	FC	TIMIT	99.99% \pm 0.03%	98.35% \pm 2.08%	64.02% \pm 0.20%	59.45% \pm 1.05%
C_1	Shallow Conv	CIFAR-10	99.89% \pm 0.02%	99.66% \pm 0.20%	80.04% \pm 0.12%	77.26% \pm 0.42%
C_2	Deep Conv	CIFAR-10	99.99% \pm 0.04%	99.99% \pm 0.01%	89.24% \pm 0.12%	87.26% \pm 0.07%
C_3	Shallow Conv	CIFAR-100	99.56% \pm 0.44%	99.88% \pm 0.30%	49.58% \pm 0.39%	46.45% \pm 0.43%
C_4	Deep Conv	CIFAR-100	99.10% \pm 1.23%	99.57% \pm 1.84%	63.03% \pm 0.50%	57.81% \pm 0.17%

stall until the previous layer finishes. All features must, therefore, be cached until the last feature of a layer is obtained. This requires large storage that necessitates the use of off-chip memory. Second, the computational load varies greatly among layers. A naive workload distribution can result in a large number of idle cycles due to inter-layer dependencies. These two problems are present in both inference and training but have a greater impact on the latter. In training, all features of the hidden layers must be cached until their corresponding errors arrive through Back Propagation (BP), thus requiring much larger memory. And due to BP, the number of operations per layer triples.

We propose FPDeep, a novel FPGA-cluster-based training framework for CNNs that solves the problems just described. FPDeep does this by using a hybrid of layer and model parallelism together with a number of new workload/weight balancing strategies. No reconfiguration is needed: each device computes only certain layers or a part of a single layer; each device is optimized independently with respect to its own computation. The cluster is now a single fine-grained pipeline so the batch size can be arbitrarily small. The amount of data that must be saved is drastically reduced eliminating most off-chip memory accesses. Internode communication is simple and pipeline utilization very high. To the best of our knowledge, our work is the first on CNN training with FPGA-based clusters using this method of parallelism and also the first with fine-grained workload/weight balancing.

The underlying theme of this work is to convert batch parallelism to pipeline parallelism, which has obvious benefits. Parallelism is equal to the depth of the pipeline, in this case, many thousands of stages across the cluster. Communication paths can be short so cycle times are as small as the designer can make them. Communication among devices is direct and contention-free with any latency having no effect on throughput. There is also the aforementioned benefit of having all of the latency reduction applied to individual problem instances and so obviating the algorithmic challenges that come with larger batches.

We find this approach to be effective with performance similar to that of GPU clusters of similar size and technology, but with far better power efficiency. The limiting factor is inter-FPGA bandwidth. But, somewhat surprisingly, we find that a 1D topology suffices and that, even using only six transceivers per FPGA (Stratix-V era), FPDeep achieves linear speed-up to 83 FPGAs. Overall, with 250 Gb/s bidirectional bandwidth per FPGA, easily supported by current generation FPGAs, FPDeep's performance shows linearity up to 100 FPGAs. The main contributions are as follows:

1) The possibility of breaking down the well-known scalability wall of CNN training and demonstrating FPGA

clusters to be a competitive technology for CNN training;

2) A novel framework for mapping CNN training logic to distributed FPGA clusters that achieves both high efficiency and scalability; that does not suffer from issues related to mini-batch size; and that needs only a simple interconnection network as is available in any multi-FPGA system with efficient inter-FPGA communication and reasonable bandwidth;

3) A fine-grained pipeline design that minimizes the time that features need to remain available while waiting for back-propagation, thus reducing the storage demand to the point where only on-chip memory is required for the convolution layers;

4) Fine-grained partitioning and mapping methodologies, which provide almost perfect workload and weight balancing among FPGAs; this is done by increasing the flexibility of workload and weight allocation, thus leading to improved utilization: multiple FPGAs can cooperatively compute the same layer, while multiple layers can also be mapped to the same device;

5) An RTL code generator that automatically creates RTL implementations based on the mapping scheme generated by FPDeep.

The organization of this paper is as follows. In Section 2, related work is discussed. In Section 3, the methodology of FPDeep is presented and the workload/parameter partition methods are defined. In Section 4, the overall system architecture is given. In Section 5, the implementation of each FPGA node's accelerator is introduced. In Section 6, the experimental results are presented. Discussion and further work are in Section 7.

2 BACKGROUND AND RELATED WORK

In this article we use VGG-16, a widely used neural network in image classification, as an example to demonstrate various FPDeep features.

2.1 Background

The computations for CNN training are shown in Fig. 1(A). The red datapath shows Forward-Propagation (FP). It calculates the errors of output features in the final layer. Starting with an input image (Cat), neurons in each layer are evaluated with parameters Pa_i . Errors are calculated by comparing inference results to the label in the training dataset. BP has two sub-steps: Error Back-propagation (EB-green) and Parameter Gradient (PG-orange). In EB, errors are back-propagated through the network. In PG errors of each layer are used to calculate gradients of the weights ($\frac{\partial Err}{\partial Pa_i}$). The convolution kernels are called parameters, the temporal convolution results are called activations. The notation is shown in Table 2.

TABLE 2
Definitions of symbols used in the equations.

Notation	Description
$A[b][l][c]$	Activation[batch-id][layer-id][channel-id]
$Pa[b][l][o][i]$	Parameter[batch-id][layer-id][output-channel-id][input-channel-id]
$dP[b][l][o][i]$	Differential of the parameter[batch-id][layer-id][output-channel-id][input-channel-id]
$E[b][l][c]$	Error of the layer[batch-id][layer-id][channel-id]
W	Size of the activation
K	Size of the convolution kernel
IC	Number of the input channels
OC	Number of the output channels

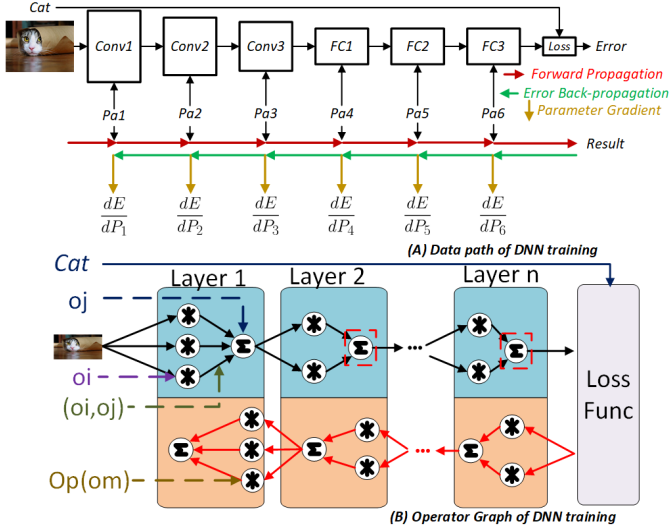


Fig. 1. (A) Illustration of computations involved in CNN training including datapaths for forward propagation and backward propagation and parameter gradient update. (B) An operator graph is used to represent DNN training. Nodes are operators and edges are tensors.

Fig. 2 zooms-in on the FP, EB, and PG operations of two layers. As shown in Tab. 2 we use A , Pa , dP , and E to represent Activations, Parameters, Differentials of the Parameters, and errors, respectively. The relationship among of these in CNN training is shown in Eqns. 1, 2, 3 and 4

1. For FP the activation of layer l 's channel c is generated by summing all related convolution results of layer $l-1$'s activations and layer l 's parameters (Eqn. 1).

2. For EB the error of layer l 's channel c is generated by summing all related convolution results of layer $l+1$'s errors and layer l 's parameters (Eqn. 2).

3. For PG the differentials of layer l 's parameters are the convolution results of this layer's error and the previous layer's activation (Eqn. 3). The differentials of the parameters are then used ($Pa[b] + dP[b]$) as the next batch's weights ($Pa[b+1]$, Eqn. 4).

$$A[b][l][c] = \sum_p A[b][l-1][p] * Pa[b][l][c][p] \quad (1)$$

$$E[b][l][c] = \sum_p E[b][l+1][p] * Pa[b][l][c][p] \quad (2)$$

$$dP[b][l][p][q] = A[b][l-1][q] * E[b][l][p] \quad (3)$$

$$Pa[b+1][l][p][q] = dP[b][l][p][q] + Pa[b][l][p][q] \quad (4)$$

2.2 Related Work

Much work has addressed the mapping of inference/training of CNNs to clusters with programmable ac-

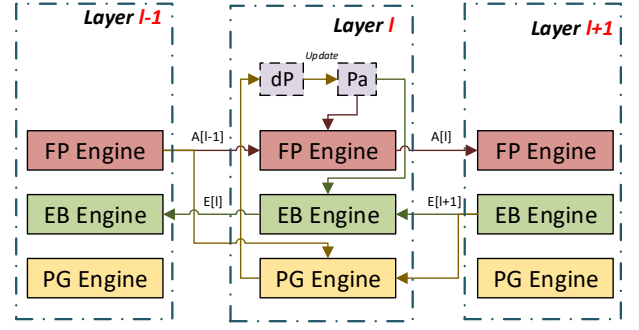


Fig. 2. Execution details of forward and backward propagation with zoom-in on adjacent CONV layers, operations and data dependencies.

celerators, including [3], [39]. Also, many frameworks and libraries have been deployed, e.g., MXNet [6], Caffe [19], and Tensorflow [1]. These systems hide the complexity of workload decomposition and provide friendly programmer interfaces, including Python, R, and Scala. In [30], Google proposed a method that uses reinforcement learning to optimize device placement for the Tensorflow computational graph. [16] introduced GPipe to solve the problem that the DNN model capacity increases to the point that the model is too big to fit in the memory of a single accelerator. GPipe uses a batch-splitting pipelining algorithm to map AmoebaNet onto eight GPUs. Microsoft proposed PipeDream [16], which exploits intra-batch parallelism to train CNNs with GPU clusters. PipeDream uses dynamic programming to find the optimal workload partition. A detailed comparison is given in Sections 3.2 and 4.4.

For FPGA-based clouds, the prior work is more limited. Microsoft's Catapult project [4], [33] includes a parameterized CNN accelerator cluster that can deliver over 1 TFLOPs with very high energy efficiency. Zhang's CDSC FPGA-Enabled Cluster accelerates CNNs on top of Spark and Hadoop [8], [43]. In [43], researchers build a deeply pipelined FPGA cluster with 6 Xilinx VC709 boards to accelerate CNNs. In [45], an FPGA-based framework of CNN training is proposed but focuses mainly on single-FPGA designs.

Most distributed CNN systems, including TensorFlow and CNTK, are based on the distributed synchronous SGD algorithm (Centralized Parallel SGD algorithm - C-PSGD). The Parameter Server Topology [25] uses a central parameter node connected with multiple worker nodes. There are multiple bottlenecks including communication load on the central node [27] and idle time while waiting for straggling worker nodes [5]. Also, for large-scale clusters, the growth in the SGD mini-batch size limits scalability. Lian, et al. use a decentralized parallel SGD algorithm (D-PSGD) to build a large-scale cluster [27]. Each node must maintain its own local copy of the model and data duplication is inevitable.

Fig. 4 shows the design space for mapping CNNs onto distributed nodes. We use the terminology introduced by [2]. Data parallelism (Fig. 4(A)) is the most popular approach in CPU and GPU clouds [1], [6]. It is also widely used in existing FPGA clouds, such as Catapult and CDSC [8]. This method has drawbacks as mentioned in Section I. In CNNs, the configurations of each layer, such as kernel size, pooling size, and stride size, vary greatly, requiring different

TABLE 3
Hardware Constraint Parameters

Notation	Description
Device Num	Number of FPGA devices in the cluster
LUT_{max}	Number of Look-up table per FPGA device
FF_{max}	Number of Flip-flop per FPGA device
$BRAM_{max}$	Number of Block-RAM per FPGA device
DSP_{max}	Number of DSP-slice per FPGA device
$Trans_{max}$	Number of available transceiver per FPGA device

hardware designs to obtain optimal performance. Thus, FPGAs need to be reconfigured between layers, leading to significant overhead. Also, as each FPGA executes all layers in sequential order, each layer starts only after the previous layer has completed. Thus, for all intermediate features, weights need to be stored to, and loaded from, the host upon completion of a layer, leading to heavy off-chip memory traffic.

Layer Parallelism (Fig. 4(B)) maps layers of the CNN onto individual nodes and pipelines the computation. It has been employed by both GPU [17, 18, 20, 41] and FPGA frameworks [43]. In [41], each LSTM layer is assigned to a different GPU. Since each layer is mapped to a certain GPU, workloads are not balanced among devices. For multi-FPGA systems, Zhang, et al. [43] only addresses inference; also, the parallelism is coarse-grained, the workload is unbalanced, and there is heavy off-chip memory communication. So while Layer Parallelism mitigates some of the problems with batch size and frequent reconfiguration, it suffers from other problems: load balancing and stalls as some nodes wait for others to finish.

In Model Parallelism (Fig. 4(C)), weights for each layer are distributed across nodes. Therefore, all intermediate results from all devices must be added up and then broadcast to every device leading to heavy communication. This method has been used for AlexNet [22].

3 FPDEEP FRAMEWORK

3.1 Overview

An operator graph G (Fig. 1(B)) is used to describe the operations in DNN training. Each node $o_i \in G$ is an operator (e.g., matrix multiplication or active function), and each edge $(o_i, o_j) \in G$ is a tensor (an n -dimensional array) that is an output of o_i and an input of o_j . Each node o_m has a weight $Op(o_m)$. Hardware constraint parameters (Tab. 3) are used to describe all available hardware devices.

FPDeep thus has two sets of input parameters, from the Operator Graph and the Hardware Constraint Parameters. The whole framework contains two parts: mapping and implementation (Fig. 3(A)). The Mapping Framework partitions the operator graph into several fine-grained segments and maps them onto FPGA clusters so that every FPGA gets a balanced workload and parameters. In the Implementation Framework, the RTL generator creates RTL implementations for each FPGA based on the parameterized mapping, and a cycle-accurate simulator gives measures of throughput, bandwidth demand, and percent idle stages.

3.2 Operator Graph Partitioning Methodology

As mentioned in Section 2, DNN training includes FP, EB, and PG phases. The inter-phase data dependencies lead to

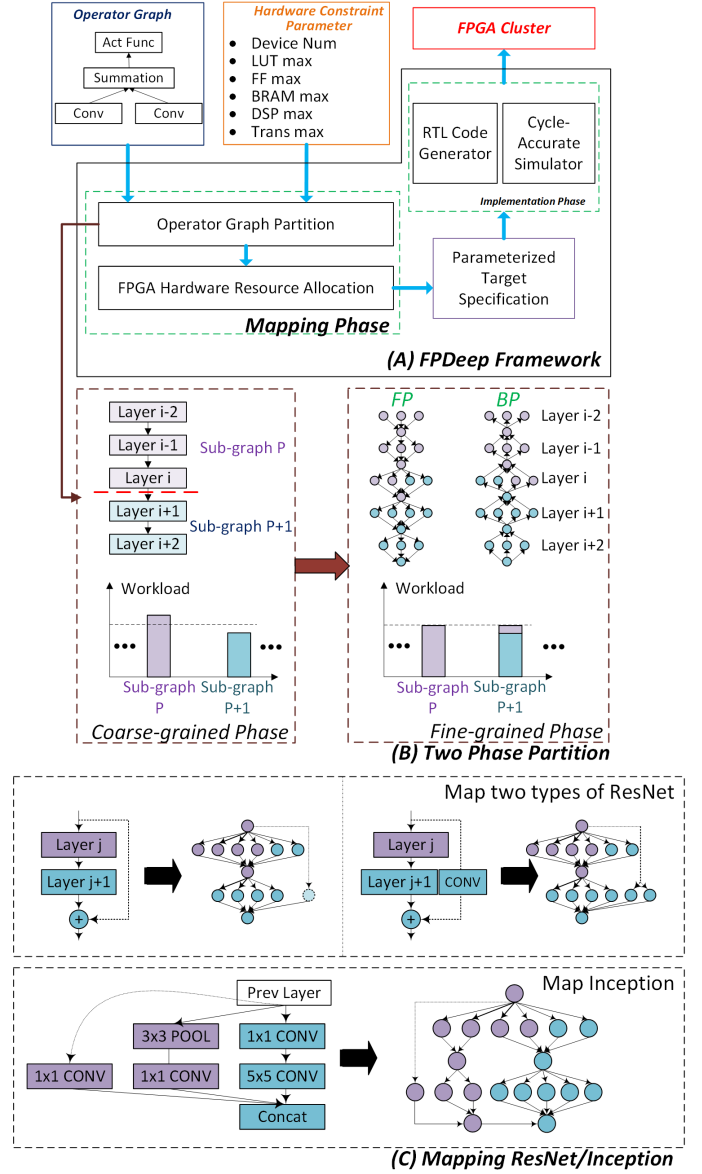


Fig. 3. (A) Overview of the FPDeep Framework. The operator graph and hardware constraints are input parameters. (B) FPDeep contains two phase: mapping and implementation. (C) The proposed DNN operation graph partition methodology with ResNets and Inception.

a complex operator graph G . General graph partitioning methods, such as Google's Reinforcement Learning (RL) method [30], are useful approaches in the partitioning tasks of DNN training, but not efficient enough. Finding the global optimal solution of graph partitioning is NP-hard, making the time to find the best partition comparable to the DNN training time [32]. FPDeep takes advantage of the fact that DNN training logic can be modeled as a computational pipeline consisting of groups of consecutive layers; this significantly simplifies the optimization algorithm and makes it possible to return the exact solution in polynomial time.

As shown in Fig. 3(B), FPDeep graph partitioning works in two phases: 1) Coarse-grained and 2) Fine-grained.

1. Coarse-grained phase: The whole graph G is abstracted and simplified as a one-way graph \bar{G} . Each node in \bar{G} presents the workload of forward and backward propagation of a certain layer. The coarse-grained graph \bar{G} is partitioned into multiple (number of FPGAs) sub-graphs with similar sizes of workloads. This simplifies the

TABLE 4

Operator Partition Choice. Different operator graph partition design choices make possible different parallelizability methods.

	Parallelizability Method		
	Sample	Layer	Model
Data Parallelism (DP)	Y	N	N
Layer Parallelism (LP)	N	Y	N
Model Parallelism (MP)	N	N	Y
Hybrid Parallelism (HP)	N	Y	Y

partitioning process of \bar{G} , but results in a coarse-grained partitioning solution with high variance in the workload size distribution of the sub-graphs.

2. Fine-grained phase: Each sub-graph \bar{G}_i of \bar{G} is replaced with the details of forward and backward propagation. As shown in Fig. 3, \bar{G}_i is presented with a finer tiling unit with nodes representing convolution operations at different channels. In this phase, FPDeep performs reallocation of FPGA resources in a finer-grained manner to reduce the variance of workload distribution in phase 1. In Section 6, we showcase the proposed graph partitioning method with practical DNNs e.g. AlexNet, VGG-16/19.

The proposed CNN training graph partitioning method is useful not only for Feed-Forward DNNs (FFDNNs), but also DNNs with more complex topologies, e.g. Residual Neural Networks, Inception, as they can also be modeled as pipelined groups of consecutive layers with some extra abstractions (Fig. 3(C)). For example, the parallel convolution and pooling kernels in Inception can be treated as additional output channels of a CONV layer in FFDNNs followed by distributed and pipelined concatenation kernels for data reduction. For Residual Neural Networks, a shortcut can be treated as extra channels of the convolution kernels being bypassed by the shortcut. Support for DNNs with more general topologies will be included in the next-generation FPDeep.

3.3 Design Choices in Operator Graph Partitioning

As shown in Fig. 4, we can use a cube to represent a node in the operator graph. For each operator, there are three parallelizable dimensions: Sample, Model, and Layer. All available partition choices are shown in Fig. 4 and Tab. 4. Four metrics are used to compare the partitioning methods:

1. FLOP Utilization. Maximum FLOPs can be achieved when every DSP slice processes a valid operation every clock cycle. Real performance is less than ideal because of workload imbalance or synchronization overhead. FLOP Utilization ($RealFlops/MaxFlops$) captures this behavior.

2. Storage Requirement. During DNN training, model parameters and temporal activations must be stored. The total Storage Requirement determines whether all necessary data can be stored in on-chip memory.

3. Communication Footprint. FPGAs need to synchronize data among co-workers. The Communication Footprint specifies the entire communication data throughput of one mini-batch SGD iteration.

4. Communication Bandwidth is the communication footprint divided by the time of one mini-batch SGD iteration. This metric is used to characterize burstiness.

Fig. 5 shows results for these four metrics for different parallelization methods and scales of FPGA clusters. For VGG-16, we set the batch size to 128 so that the DP method

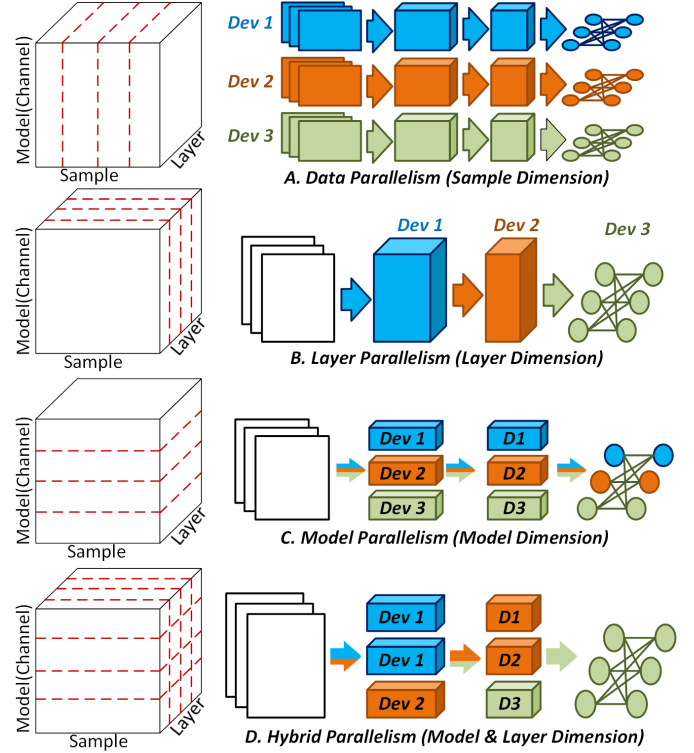


Fig. 4. Illustration of operator graph partition design choices: (A) Data parallelism, (B) Layer parallelism, (C) Model parallelism, and (D) Hybrid parallelism (Layer + Model)

works for clusters with fewer than 128 devices. There are 16 layers (13 convolution and 3 fully connection); thus the LP method only works for a cluster with less than 16 devices. Also, the minimum channel count is 64 (Layer CONV-1,2) so the MP method works for clusters with less than 64 devices.

A. Analysis of Flop Utilization. LP is the best choice because all devices work in a pipeline manner. However, because of variations in the DNN layer's operation count, LP still suffers from workload imbalance. DP is the second choice. In each mini-batch SGD iteration, DP must synchronize the DNN model globally, which causes serious communication overhead for large scale FPGA clusters. MP is the worst choice since it needs an additional layer for synchronization among different channels.

B. Analysis of Storage Requirement. LP is the best choice because of pipelining, which means the cluster does not need to store temporal activations off chip and the DNN model's parameters are distributed. **Clearly, when the cluster is small, storing all necessary data in the FPGAs' on-chip memory is a challenge.** But when the cluster is large enough, the size of on-chip memory is not a bottleneck. For DP and MP, each FPGA must keep its own copy of the DNN model's parameters. Also, all temporal activations must be maintained in local memory.

C. Analysis for Communication Footprint. DP is the best choice because all temporal activations are stored locally and only DNN model parameters need device-to-device communication. LP is the second choice because devices work in a pipeline manner and each device needs to synchronize activations with adjacent devices. MP is worse because it needs to both synchronize parameters globally and synchronize activations among channels.

D. Analysis for Communication Average Bandwidth.

TABLE 5

Qualitative comparison, from 1 (worst) to 4 (best), of different partition methods with respect to various parameters

Design / Metrics	Parallel Opt	Flop Util	Storage Req	Comm Foot-print	Comm Band-width
Data Parallel	2	2	2	4	4
Layer Parallel	1	3	3	2	2
Model Parallel	3	1	1	1	1
FPDeep	4	4	4	3	3

DP's bandwidth is the lowest due to the centralized burst communication pattern. It only synchronizes the model's parameters after all workers finished their jobs. When the workers are busy, the device-to-device links are idle. In LP, due to all devices working in a pipeline manner, they need to synchronize activations with the adjacent nodes. The communication is stable and the bandwidth of LP is larger than for DP.

FPDeep Summary: Rather than DP, LP, or MP, FPDeep uses a hybrid parallel method (Fig. 4). It works in a deeply pipelined manner with workload balanced among devices; this improves FLOP Utilization. The balanced allocation policy also reduces the Storage Requirement of the device-to-device activation buffer. As there is no "free lunch," all temporal activations must be transferred among devices. **Thus the communication bandwidth is the system's bottleneck for large scale clusters.** Tab. 5 compares the different partition methods. Performance details are presented in Section 6.

3.4 Mathematical Model of FPDeep

As shown in Fig. 3, the mapping phase of FPDeep has two parts: operator graph partitioning and FPGA resource allocation. We present a mathematical model for this process. We assume N FPGAs and an operator graph G .

3.4.1 Operator graph partitioning

In this step, the operator graph G is partitioned into a set of sub-graphs $\mathcal{G} = \{G_1, G_2 \dots G_N\}$. Function Op returns the operation count of a sub-graph. For example, $Op(G_i)$ is the operation count of operator graph G_i and $Op_{min}(\mathcal{G})$ is the minimum operation count of the sub-graph set \mathcal{G} . Because the FPGA cluster is pipelined, the variance of the sub-graph operation count V should be minimized:

$$V = \sum_i \frac{Op(G_i) - Op_{min}(\mathcal{G})}{Op_{min}(\mathcal{G})} \quad (5)$$

3.4.2 FPGA resource allocation

In this step, the FPGAs' hardware resources are allocated according to the sub-graph set \mathcal{G} . The resource allocation step is an optimization problem. The pipeline is constructed from Convolution Engines (CEs), which are used to handle compute-intensive convolution operations, and buffers (Buf), which are used to store CNN model parameters and temporal activations. Convolution engines are composed of 2-D systolic arrays that consume input features from shift-registers. Their design is similar to those in [40],

[45]. The FPGAs' resources can be expressed as a tuple: $(LUT, FF, BRAM, DSP)$.

As mentioned above, for large FPGA clusters, the goal is to maximize the cluster's throughput (T), while for small FPGA clusters, the goal is to minimize the storage requirement. In this context, *size* is relative and it depends on the ratio of the size of the neural network to the FPGA resources. The constraints lie in the hardware resources at each device and are denoted as $(LUT_{max}, FF_{max}, BRAM_{max}, DSP_{max})$.

For large clusters the number of CEs in device i is denoted as CE^i . The theoretical maximum performance of these CEs is $Perf(CE^i)$. These convolution engines need buffers Buf^i , which is the function of CE^i (Eq. 6). The overall throughput of the cluster is T and depends on the node with the lowest performance (Eq. 7).

$$Buf^i = f_1(CE^i) \quad (6)$$

In FPGAs these Convolution Engines or Buffers can be built with hard DSP-slices/Block-RAMs or distributed Lookup-Tables/Flip-Flops. We build some CEs (αCE^i) with hard DSP slices and other CEs $((1-\alpha)CE^i)$ with LUTs/FFs. Similarly, some buffers (βBuf^i) are built with hard BRAMs and others $((1-\beta)Buf^i)$ with LUTs/FFs. Equations 8, 9, 10, and 11 define the hardware resource constraints. Functions $f_2, f_3, f_4, f_5, f_6, f_7$ return the consumption of the corresponding hardware resource. The target function for a large cluster is the maximum throughput T :

$$T = \min(\frac{Op(G_i)}{Perf(CE^i)}) \quad (7)$$

subject to:

$$LUT^i = f_2(\alpha CE^i, \beta Buf^i, (1-\alpha)CE^i, (1-\beta)Buf^i) \leq LUT_{max}, \quad (8)$$

$$FF^i = f_3(\alpha CE^i, \beta Buf^i, (1-\alpha)CE^i, (1-\beta)Buf^i) \leq FF_{max} \quad (9)$$

$$BRAM^i = f_4(\beta Buf^i) \leq BRAM_{max} \quad (10)$$

$$DSP^i = f_5(\alpha CE^i) \leq DSP_{max} \quad (11)$$

For small clusters the target function minimizes the storage requirement S ; the constraints are the same as the large cluster case. To fit all DNN training logic into a small cluster, we propose a method called **parameter balancing**.

$$S = \max(f_6(\beta Buf^i) + f_7(\beta Buf^i, (1-\beta)Buf^i)) \quad (12)$$

Fig. 6(A) shows the number of model parameters and activations in VGG-16. Observe that from the first to the last layer the number of activations is decreasing while the number of parameters is increasing. The decrease in activations is because the dimensions of the feature maps are reduced by the pooling layers. The increase in the parameters is because the number of input and output channels increases in the later layers. In clusters with small numbers of FPGAs (to accelerate VGG-16) the memory demand of parameters for the later layers increases to the point where the on-chip memories in each FPGA are not big enough to cache the allocated parameters.

To make enable the mapping of big networks to small clusters of FPGAs, parameter balancing can be used. Figs. 6(B-D) show the method. Simply, parameters from the later layers are stored in FPGAs where there is room, even if those

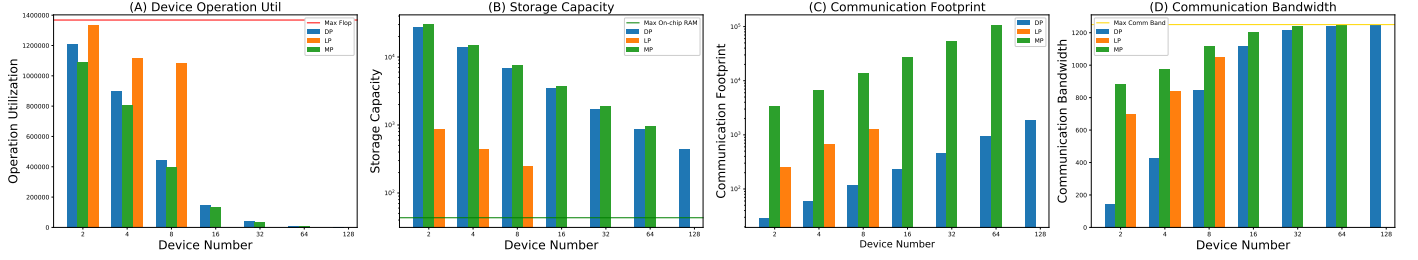


Fig. 5. Comparison of different operator graph partition methods accounting for four different metrics: flop utilization, storage requirement, communication footprint, and average communication bandwidth)

FPGAs are some distance away from where those parameters will eventually be used. For example, the parameters of layer 8 are stored in FPGAs 3 and 1. During computation, the parameters stored in FPGA 1 are transferred to FPGA 3 through the communication network together with activations. Note that the transport of parameters does not tighten the constraint on inter-FPGA communication. This is because the smaller number of activations in the later layers cancels out the added traffic for the parameters. Our experiments demonstrate the benefit of this approach: only on-chip memory is needed for the CONV layers.

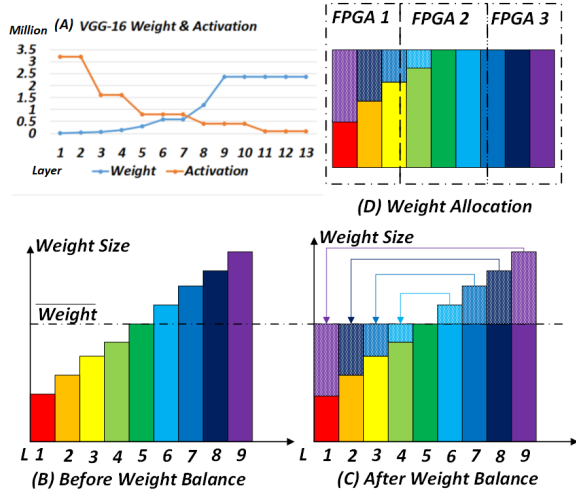


Fig. 6. Parameters and activations for VGG-16

4 SYSTEM DESIGN

4.1 Input/output channel partition implementation

FPDeep uses hybrid parallelism to partition the operator graph. We begin by noting that partitioning in the layer dimension is straightforward (Fig. 4(D)). The model dimension is more involved and is done via input/output channel partitioning. As shown in Fig. 7, each device executes the operations of a fraction of the input/output channels. Input feature maps, along with model parameters, are partitioned in the ic/oc dimension and allocated among FPGA devices. Each device generates the partial results and their sum is the final output activation.

There are two ways to partition the graph, by input (ICP) or by output (OCP) channels. These methods are shown in Figs. 7(A) and (B), respectively.

1) ICP: Layer 1 is partitioned and mapped to 4 devices. IC input activation channels and corresponding weights are partitioned into 4 segments each containing $IC/4$ channels. Each FPGA receives one of the 4 segments and calculates partial results of activations for all output channels. Each

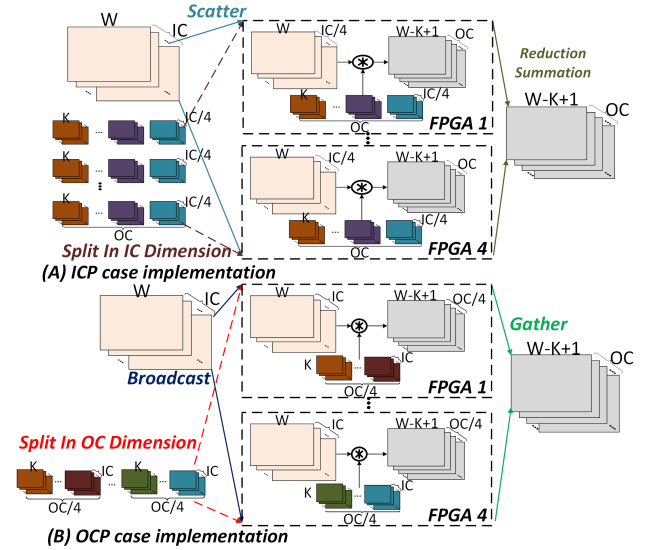


Fig. 7. Partitioning image input/output channels ic/oc

complete output activation is calculated by summing up the related partial results from the 4 FPGAs.

2) OCP: OC output activation channels are partitioned into 4 segments each containing $OC/4$ channels. Each FPGA is responsible for calculating a certain segment of output activations. The 4 segments' results are then gathered. In CNN training, all activations need to remain available while waiting for back-propagation. Therefore, all IC input feature maps are cached in every FPGA. This duplication leads to additional on-chip memory overhead. This defect of OCP does not exist in ICP, so FPDeep prefers to use ICP: OCP is only used when the number of the input channel is too small to provide sufficient parallelism. For example, the first layer of AlexNet only has 3 channels of input features but 96 channels of output features so OCP is used.

4.2 Dataflow Analysis and Interconnection Topology

Fig. 8(A) shows an N layer CNN mapped to an FPGA cluster with M devices. Each CNN layer contains O_i operations ($i \in [1, N]$). The computation capacity of each device is C operation per second. To balance compute workloads among devices, the workloads are mapped to FPGAs in proportion to the device's compute capacity. Each device needs to execute $\bar{W} = \frac{\sum C_i}{M}$ operations.

Fig. 8(B) zooms in on the CNN training procedure that turns the dataflow into an operator graph as the summation of all activation channels are cut into several pieces. Each piece only needs to add the local convolution result R_i to the previous node's intermediate result R_{i-1} . The workloads, i.e., the arithmetic operations, of a whole network are

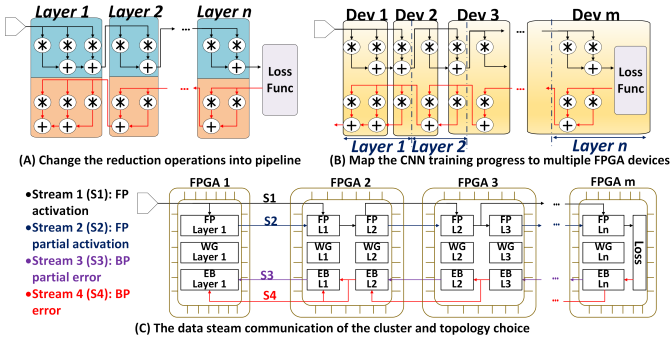


Fig. 8. Data flow analysis of CNN training. FPDeep pipelines the reduction operations and maps them to multiple FPGAs.

partitioned and mapped onto M FPGA nodes in proportion to their computation capacities. Fig. 8(C) shows the data streams in the FPGA cluster. Note that a 1-D interconnect topology is sufficient.

Fig. 9 illustrates the topology design choice by mapping VGG-16's CONV-3 - CONV-5 layers onto a cluster with eight VC709 FPGA boards (see Section 6) according to FPDeep's operator graph partition method and FPGA resource allocation policy. The red dotted box marks the communication bottleneck. Let us assume 10 board-to-board interconnections ports. First assume a 2D topology. We see that Dev-1 is the bottleneck: because the degree of Dev-1 is 5, each communication link of Dev-1 only has two ports. With a 1D topology, however, Dev-4 is the bottleneck. Some of the partial output activations need to be duplicated (dotted arrows in and out of Dev-4). But the degree of Dev-4 is only 2 while each communication link has 5 interconnection ports, making the communication more efficient.

Fig. 10 shows quantitatively how the choice of topology affects performance. For clusters larger than 5 nodes, the 1D topology is better. As the number of nodes increases, this advantage becomes even more apparent. For clusters with 4 nodes, the 2D topology is better because the degree of the bottleneck device is only 2.

A further advantage of 1D topology versus 2D is its simplicity. With only single links, different dataflow types are multiplexed and easily scheduled. Also, for 2D the reduction operation of each DNN layer is centralized, which incurs significant synchronization overhead and requires more data movement.

Another consideration is that, practically, FPGA accelerator boards almost always have less communication capability than the FPGAs themselves, both in BW and number of ports. This makes the choice of 1D even more crucial. An interesting exception is for accelerator boards with multiple tightly coupled FPGAs. For single boards with, say, four FPGAs, we have already noted that 2D is preferred. For clusters with multiple multi-FPGA boards, because internode connectivity is more limited than intranode, the preferred inter-node topology is again 1D. Within the node, however, the additional *in situ* connections remain useful leading to a hierarchical topology.

4.3 Deep Fine-Grained Pipeline

To illustrate data dependencies during training we use as an example of two CONV layers with 3×3 kernel size. The

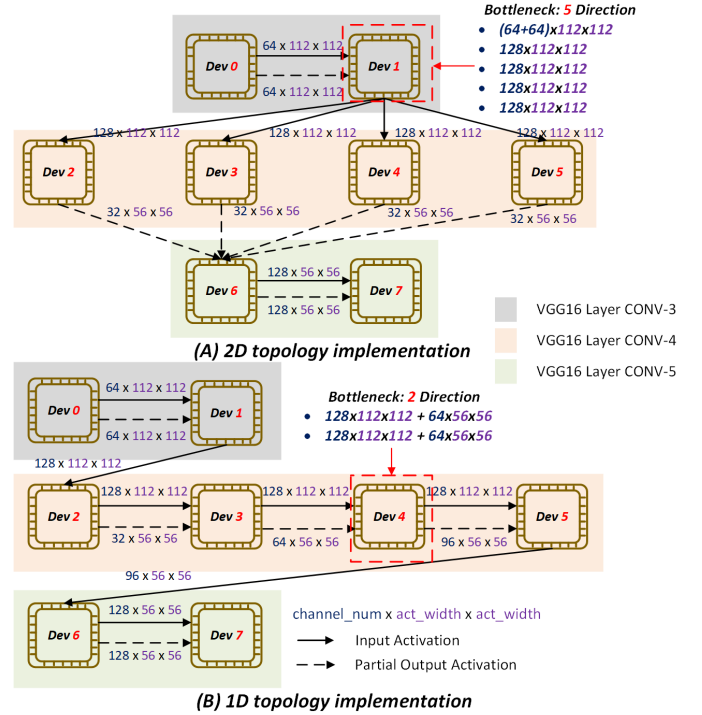


Fig. 9. 1D-2D topology design choice: while 2D seems the obvious choice clearly 1D has better performance

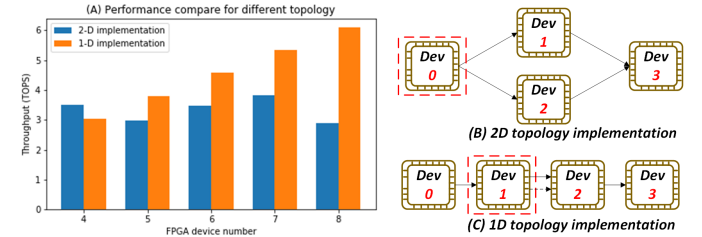


Fig. 10. 1D-2D topology performance comparison

operations of these two layers' forward/backward propagation are shown in Fig. 11(A). In forward propagation, a 7×7 feature map is fed into Layer 1 and a 5×5 feature map is generated. At layer 2 the 5×5 feature map is convolved with the parameters and inferred to a 3×3 feature map. In the backward propagation, the 3×3 error map is padded to 7×7 before it is fed to Layer 2. Next, the error map and corresponding parameters are convolved and another (5×5) error map is produced; this is used for Layer 2's weight/bias gradient calculation. At Layer 1, the 5×5 error map is padded to 9×9 and then convolved to 7×7 .

The Fig. 11(A) depicts the data dependency of forward and backward propagations during CNN training. For the forward propagation phase, the image is inferred through all layers. To determine the data dependency, we start from the four activations at the output feature map at the last layer, which are marked as black, red, blue, and yellow, respectively, and trace backward to find the region of the input feature maps on which each depends. For the backward propagation phase, errors calculated are propagated backward through the network. To calculate gradients at a particular layer, errors which are backward propagated from the next layer and activations of its feature maps are necessary. Hence, the feature maps, which are generated in the forward propagation phase, need to remain available

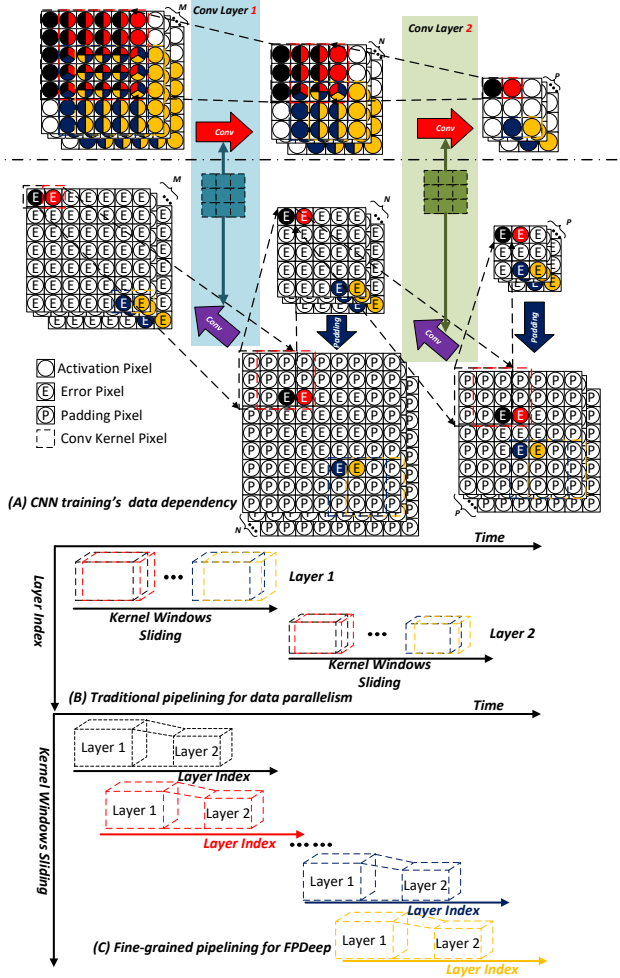


Fig. 11. FPDeep's deep fine-grained pipeline design. (A) shows the data dependency of CNN training; (B) shows traditional data parallelism's coarse-grained pipeline; (C) shows FPDeep's fine-grained pipeline.

awaiting backward propagation. As shown in Fig. 11, activations, and errors among CONV layers show only fine-grained dependencies. That is, to begin computing the value of a pixel in a layer, only a fraction of the pixels from the previous layer are needed. Therefore the computation of a layer can start much earlier before the previous layer is completely done. This provides the opportunity to process all CONV layers in parallel in a fine-grained pipeline.

The Fig. 11(B) shows the traditional method of accelerating CNN training. First, the N channels of the feature maps are fed into the convolution layer $L1$. Next, results from all M channels begin to be processed while the convolution kernel slides across the N -channel feature maps. Much storage capacity is needed to maintain all temporal feature maps. Clearly, this method is not efficient. The fine-grained alternative is shown in Fig. 11(C): the calculation of an activation/error starts as soon as its dependent activations/errors are propagated from the previous/next layer. The basic process unit of FPDeep is an activation/error of a feature/error map; this is in contrast to the traditional method's basic unit of the entire feature/error map. The result is both a large increase in parallelism through the added pipeline stages and a reduction in storage so that only on-chip memory is needed.

4.4 Parameter Alignment

In contrast to traditional DP, where centralized gradient aggregation and weight update need to be performed sequentially, FPDeep conducts all of these processes in parallel. In order to achieve full hardware efficiency, we use a distributed and slightly-unaligned weight update scheme: gradient calculation, aggregation, and weight update are always performed locally. With respect to the parameter alignment, after the last training sample in a certain mini-batch (round M) is forward propagated through the cluster, its backward propagation follows immediately. At the same time, the forward propagation of training samples in the next mini-batch (round $M + 1$) follows using the old parameters of round M until the last training sample (from round M) is backward propagated. The deep fine-grained pipeline used in FPDeep guarantees fast feature and error propagation and reduces the time that old parameters need wait for the weight update, i.e. it eases the parameter alignment issue. Based on our experiments, with the mini-batch size as 1K and a cluster with 100 FPGAs, only the first 5 training samples of each epoch suffer from the resulting slight non-alignment. Moreover, this slight parameter non-alignment does not affect the convergence rate (as discussed in Section 6.4 and shown in Fig. 16(B)(D)(F)).

Existing work considers parameter alignment and high throughput to require a trade-off. Google's GPipe [16] and Microsoft's PipeDream [32] use a similar pipeline scheme to build a distributed DNN training system but use different alignment methods. GPipe divides the input mini-batch into several smaller micro-batches, enabling different GPUs/TPUs to work on different micro-batches simultaneously. GPipe needs to flush the pipeline and synchronize the gradients among all accelerators after the computation of the whole mini-batch finishes. GPipe's solution introduces many bubbles in the pipeline. Moreover, GPipe focuses on fitting oversized DNN onto multiple accelerators, not solving the large-batch training problem. To the best of our knowledge, the approach used in GPipe makes the large-batch problem even more severe: more accelerators require more micro-batches and, in order to fill up each device, the size of micro-batches must be relatively large. As with GPipe, Microsoft's PipeDream also uses coarse-grained workload partitioning and pipelining. However, in contrast to GPipe, PipeDream suffers from the parameter alignment issue. The authors propose a technique called *weight stashing* to save multiple versions of the parameters and so align parameters on a slightly longer time scale.

The optimization target of this paper is throughput, i.e. $epoch/h$. Note that in FPDeep, higher throughput is equivalent to reduced training execution time as FPDeep, because of the small mini-batch size, does not require more epochs to converge.

5 HARDWARE ACCELERATOR ARCHITECTURE

5.1 Overall Architecture

The overall architecture of the multi-FPGA accelerator and the detailed architecture of each FPGA are shown in Fig. 12(B). For an l -layer CNN, FPDeep maps convolution layers $t \sim l$ to p FPGAs. All p FPGAs are connected in a

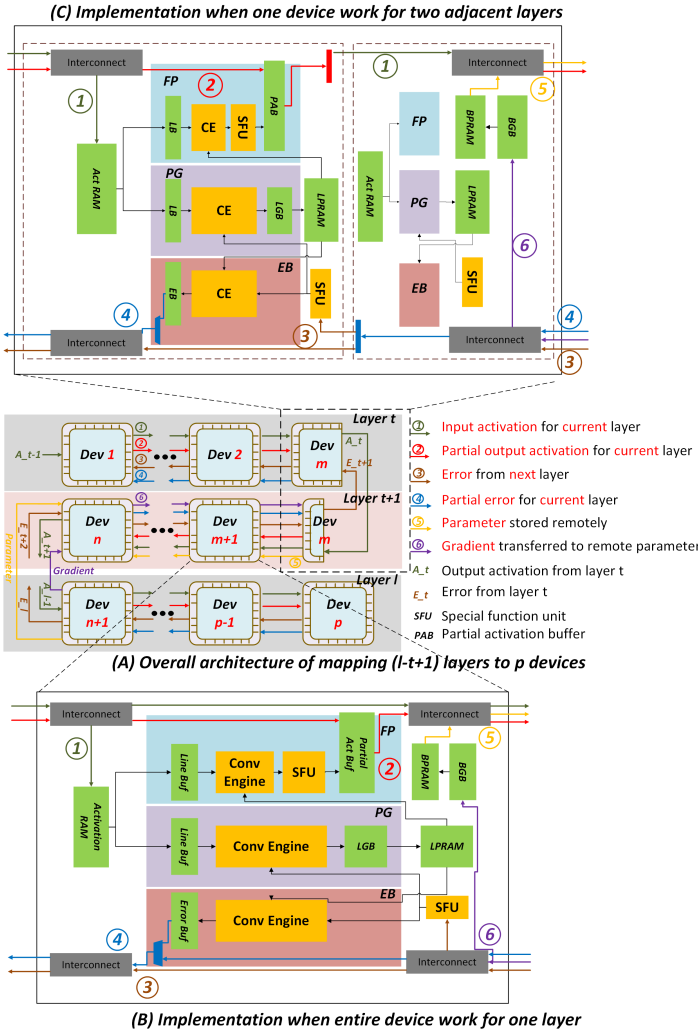


Fig. 12. Overall architecture of FPDeep accelerator and block design of each FPGA. (A) illustrates the overall architecture of FPDeep; FPGAs can cooperatively work on the same layer and multiple layers can be mapped on the same FPGA. (B) illustrates the architecture of FPGA $m+1$ which is fully allocated to layer t . (C) illustrates the architecture of FPGA m which is allocated to both layer t and layer $t+1$.

1-D topology. In Fig. 12, A_t denotes the output activation of layer t and E_t the errors backward propagated from layer t . There are six key data-paths. Steps 1, 2 and 5 are for FP, while 3, 4 and 6 are for BP.

1. Output activations from layer $(t-1)$ are allocated to FPGAs of layer t according to the ICP results. Each FPGA caches a segment of the features allocated to it and propagates the rest to the next node.
2. Using the segment of features cached at Datapath 1, each FPGA calculates partial results of all output features at layer t . The partial features produced from FPGA m are propagated to node $m+1$ through Datapath 2. After adding up partial features produced by nodes m and $m+1$, the updated partial features are propagated to the next node.
3. In each cycle, errors from layer $(t+1)$ are back-propagated to the FPGAs of layer t through Datapath 3.
4. Using errors from Datapath 3, each FPGA calculates the errors of the features allocated to it at Datapath 1 and propagates them to the preceding node. Node m propagates the errors calculated by itself first and then the errors transferred from node $m+1$.
5. Parameters are transferred from the node where they are

cached for parameter load balancing to the node where they are used to compute the output features.

6. The gradients of parameters are transferred from the node where they are produced to the node where they are cached for parameter load balancing.

The proposed architecture is generally useful for SGD-based training of any feed-forward CNNs and can be extended to support other CNNs with more complex topologies such as ResNet and Inception. As described in Section 3.2, as long as a DNN can be described as a one-way graph with nodes representing pipelined groups of consecutive layers, FPDeep can efficiently partition and map its training logic to an FPGA cluster. New modules are needed as follows: *aggregation* to the filter concatenation in Inception; *duplication* of FP, PG, and EB for the parallel CONV and Pooling kernels in Inception; and *gather and bypass* for the various types of shortcuts of ResNet. Integrating these into FPDeep is straightforward and will be part of the next-generation system.

5.2 Single-FPGA Architecture

As shown in Fig. 12, each FPGA includes FP, PG, and EB modules, as well as a memory subsystem to cache parameters, gradients, and activations. Each accelerator has 6 interconnection modules to communicate with its neighbors (this number is selected because it is available on many boards used for FPGA clusters and is sufficient for good scaling). An FPGA can be allocated to multiple layers. Implementations with FPGAs working for single layer and for multiple layers are illustrated in Fig. 12(B) and (C).

5.2.1 Interconnection

There are two pairs of interconnection modules. **a)** The upper pair, used by Forward datapaths 1, 2 and 5, 1) receives input features and partial features propagated by the preceding node; 2) bypasses the input features which are not mapped to it to the succeeding node; 3) adds partial results produced by FP to the received partial features and propagates updated partial features to the succeeding node; 4) forwards the parameters and gradients from the node which caches them to the node which produces them. **b)** The bottom pair, used by backward datapaths 3, 4 and 6, 1) receives errors from the next layer bypassed by the succeeding node and passes them on to the preceding node; 2) receives errors of this layer calculated by the the succeeding node; 3) after errors are calculated by the EB module, propagates them to preceding node; 4) forwards the parameters and gradients from the node which calculates them to the node which caches them.

5.2.2 Memory Subsystem

The memory subsystem includes BRAM-based modules and stores activations, parameters, and gradients.

1. Activation RAM (Act-RAM) caches input activations mapped to the target FPGA until they are consumed in back-propagation and provides input activations as operators to FP and PG modules for output activation and parameter gradient calculation. After input activations are consumed in FP, they are kept in Act-RAM and wait to be reused during BP to calculate parameter gradients. Act-RAM is

implemented as a FIFO-based memory. In BP, when errors are calculated and propagated backward from the adjacent FPGAs to a certain device, the features stored earlier in Act-RAM are first consumed by the PG module for parameter gradient calculations.

2. Local Parameter RAM (LPRAM) caches parameters used as operators to produce the output activation at the local FPGA. For each FPGA, there are $SF \times K \times K \times OC$ parameters stored in LPRAM, where SF is the number of activations in the activation segment allocated to an FPGA. To provide enough concurrency of parameter access, LPRAM is designed as a $SF \times K \times K$ -bank line buffer. Each bank caches OC parameters.

3. Balanced Parameter RAM (BPRAM) caches the parameters mapped to the local device for parameter load balancing. These parameters are used as operators in other FPGA devices where on-chip memory is insufficient to cache all the required parameters. Both LPRAM and BPRAM are updated by PG. Similar to LPRAM, BPRAM is implemented as multi-bank line buffer. The number of banks and their depths are decided by the parameter balancing scheme.

4. Local Gradient Buffer (LGB) caches the gradients of the parameters stored in LPRAM. The gradients are cached and averaged at each iteration. At the point that a mini-batch size number of input figures are completely trained, the averaged gradients are forwarded to LPRAM and update the parameters stored in LPRAM.

5. Balanced Gradient Buffer (BGB) caches the gradients of the parameters stored in BPRAM. These gradients are generated by and transferred from the device where the corresponding parameters are consumed.

5.3 Forward Propagation (FP)

The Line Buffer (LB) reads input features from the Act-RAM and feeds them to the Convolution Engines (CEs). The CEs perform convolutions with parameters from the LPRAM and input activation from the LB. In the Special Function Unit (SFU), the output features are activated, normalized, and sampled based on network specifications. Afterwards, features are transferred to the Partial Activation Buffer (PAB) and added to the partial features produced by and propagated by the preceding node. Finally, the updated partial activations are propagated to the next device through the interconnection module.

We use ICP as an example to show how the FP module calculates partial activation results. Assuming a certain FPGA node has been allocated with S_{IC} channels, at each cycle, $K \times K \times S_{IC}$ activation are accessed from line buffers and broadcast to the CEs. Each CE consists of S_{IC} convolution tiles. Each convolution tile has $K \times K$ multiply-accumulate units and executes a $K \times K$ convolution operation per cycle. With all convolution tiles working on different input channels, at each cycle, each CE can finish calculating the partial results of one output channel. In the FP module, there are multiple CEs calculating the activations of different channels in parallel. The number of CEs, P , is determined by the number of DSPs allocated to FP operations during the offline ICP mapping. When partial activations are calculated, they are forwarded to partial activation buffers where they are used to update the partial results produced from the previous nodes.

5.4 Error Back-Propagation (EB)

The EB module consumes errors from the next layer and produces errors for the target layer.

It takes two steps to calculate the error of each input activation: (1) errors of all output feature maps are convolved, respectively, with their parameter filters and (2) their convolution outputs are summed. In FPDeep, for an FPGA allocated with S_{IC} input channels, the EB module calculates S_{IC} errors in parallel. In EB, CEs are used to perform the convolutions of errors of output channels and their parameters. This is different from the FP module where the number of convolution tiles in each CE is S_{IC} , rather, each EB module has S_{IC} CEs. Each CE has P tiles and each tile can perform a $K \times K$ convolution operation. The number of convolution tiles in each PE is pre-determined during ICP mapping. Taking P as the number of CEs, at each cycle, the errors from P from OC output channels are broadcast to, and consumed by, S_{IC} CEs. The outputs are partial results of errors at S_{IC} input channels. After OC/P cycles, all output channels are evaluated and the complete results of errors are forwarded to the Error Buffer.

5.5 Parameter Gradient Calculation (PG)

PG consumes errors of the next layer propagated from the succeeding neighbor and calculates gradients of the parameters. Errors of output activations are used as convolution filters on the input activations which are cached in the Act-RAM during forward propagation. Gradients are cached in the Gradient Buffer and used to update parameters in LPRAM when a mini batch of samples is trained.

In contrast to the convolutions at FP and EB where the filter size is normally smaller than 7, the filter sizes in PG (R and C) can be in the hundreds. This requires expensive convolution tiles – the resources can even exceed those of the FPGA. Even if this does not happen, the PG may still occupy most of the computing resources and result in a serious workload imbalance. In PG, $K \times K W \times W$ convolutions need to be performed. In FPDeep, we cut $K \times K$ large convolutions into $W \times W$ small convolutions ($K \times K$). The overall operation count stays the same. But in this case, the PG module always fits the DSP resources constraint and the CE array design of PG is similar to the ones in FP and EB.

6 EXPERIMENTS AND EVALUATION

In this section, we describe experiments performed to evaluate the efficiency of FPDeep. First, we evaluate the correctness and performance of the design on a small FPGA cluster with eight Xilinx VC709 boards. Based on these results, we validate a cycle-accurate software simulator. Because the small FPGA cluster is insufficient for complex neural networks, we use the software simulator to evaluate the performance of FPDeep on large-scale clusters.

6.1 Small Scale Cluster Experiments

The small scale cluster experiments use a cluster of eight Xilinx VC709 evaluation boards. As shown in Fig. 13(A), each VC709 motherboard contains one XC7VX690T FPGA, an FMC-HPC connector for the daughterboard extension,

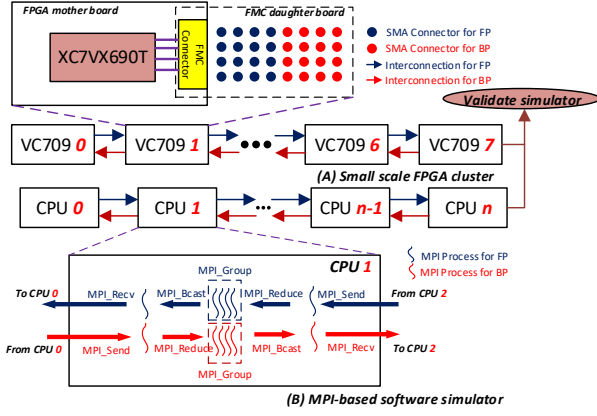


Fig. 13. Hardware evaluation and MPI-based software simulator

and an SMA FMC, which contains 32 SMA connectors. Some of the SMA connectors are used for forward propagation, others for backward propagation. The eight FPGA boards are connected in a 1-D daisy chain. This cluster is used to validate the parameterized hardware accelerators (Section 5), to perform the topology experiments (Section 4), and to validate the correctness of the software simulator.

6.2 Large Scale Cluster Experiments

The FPDeep software simulator is based on MPI version OpenMPI 2.1.1 (Fig. 13); n CPUs work in a pipelined manner. For each CPU, we have two MPI process groups, one each for FP and BP. Additional MPI processes handle data exchange with adjacent CPUs, broadcast the previous CPU's temporal activations, and reduce the current CPU's results. The simulator is parameterized to support various FPGA platforms. For example, the MPI processes which handle communication have configurable parameters that enable accurate simulation of the data exchange among FPGA boards with different interconnect bandwidths and latency.

The software simulator is currently used to evaluate large-scale VGG-19, VGG-16, and AlexNet training. For larger and more complex DNNs such as ResNet, GoogLeNet, and Recurrent Neural Networks, we will use emerging large-scale FPGA clusters. For example, the Open Cloud Testbed (associated with the Massachusetts Open Cloud), which was kicked off at the end of 2019, will in the first stage be equipped with at least 64 Xilinx Alveo 280 boards and be publicly available.

6.3 Utilization and Performance

6.3.1 FPGA Resource Utilization

For illustration, we map AlexNet and VGG-16/19 to a cluster with 15 FPGAs. Figs. 14(A-I) show resource utilization of each FPGA and resource allocations among the network layers. As shown in the DSP utilization reports (Figs. 14(C)(F)(I)), the mapping is well-balanced. The utilization of DSP slices is roughly 80% and the throughput of each FPGA is around 1 TOPS. On-chip BRAM is only used in the FPGAs that work solely on the CONV layers (FPGAs 1-14) and utilization of BRAMs is under 80%. The highest bandwidth requirement among these 15 FPGAs for these three networks is 18.6 Gb/s.

For AlexNet (only 8 layers), the 15-node cluster does not require parameter balancing to achieve the best performance

so CLB and BRAM utilization have been left unbalanced. For VGG-16/19, however, parameter balancing is required.

6.3.2 Performance and Power Efficiency

Table 6 compares performance and power efficiency among the Titan X GPU [43], the Tesla K80 GPU [23], a previous FPGA implementation [43], and this work.

[43] uses a workstation with an 8-core 3.8GHz AMD A10-5800K CPU and an Nvidia Titan X GPU. We use a server with Nvidia Tesla K80 GPUs as the golden model and baseline design. OpenBLAS and cuDNN libraries are used in software implementations. [43]'s CPU & GPU and our GPU implementation are all based on data parallelism, while [43]'s FPGA design is based on layer parallelism; the latter results in inter-board workload imbalance. The power consumption of all baseline and FPDeep systems are board-level and measured with a power meter.

FPDeep provides performance $5\times$ higher than previous FPGA work and comparable to the Titan X GPU. We evaluate energy efficiency with respect to GOPs/J. FPDeep provides $8.8\times$ better energy efficiency than the Titan X and $5.6\times$ better than the previous FPGA work. Compared with the K80, FPDeep provides $5.7\times$ better energy efficiency.

6.3.3 Load Balance and Optimization

Alexnet, VGG-16, and VGG-19 are mapped onto clusters of sizes 5 to 85 with the cycle-accurate simulator. To demonstrate workload balance among FPGAs in different sized clusters, we present the proportions of idle stages. Figs. 15(B)(D)(F) show that this is always under 5%. When the number of FPGAs is more than 30, this number is stable with fluctuation between 0.5% to 1%. Generally, as the number of FPGAs increases, the proportion of idle stages decreases. The reason is that during ICP and OFP, the number of DSPs allocated to each layer is rounded to a multiple of $K \times K$. With more FPGAs and DSP resources, this effect is reduced.

Computation and communication are critical constraints in system throughput. The roofline plots of AlexNet, VGG-16, and VGG-19 are shown in Figs. 15(A)(C)(E). Note that the throughput has linear scaling up to the constraint imposed by inter-FPGA communication. For example, with 150 Gbps as the inter-board communication constraint, FPDeep shows linearity up to 83, 56, and 70 FPGAs for Alexnet, VGG-16 and VGG-19, respectively. As each transceiver (of that generation) can reach a maximum rate of 28 Gb/s, using 6 transceivers per FPGA achieves this number [12], [36], [37].

Since high-end FPGAs frequently have more than 50 transceivers, scaling to much larger clusters is possible. The reason that bandwidth required by VGG-16 is larger than VGG-19 is straightforward: VGG-19 has more layers and thus more workload. During partitioning, with the same overall hardware resources, each layer of VGG-19 is allocated fewer resources. Thus, fewer batch features in each layer can be computed and transferred in parallel, which results in a smaller bandwidth requirement.

6.4 DNN Model Convergence

Figs. 16(A)(C)(E) show the number of epochs that can be trained per hour. FPDeep provides a linear speedup of

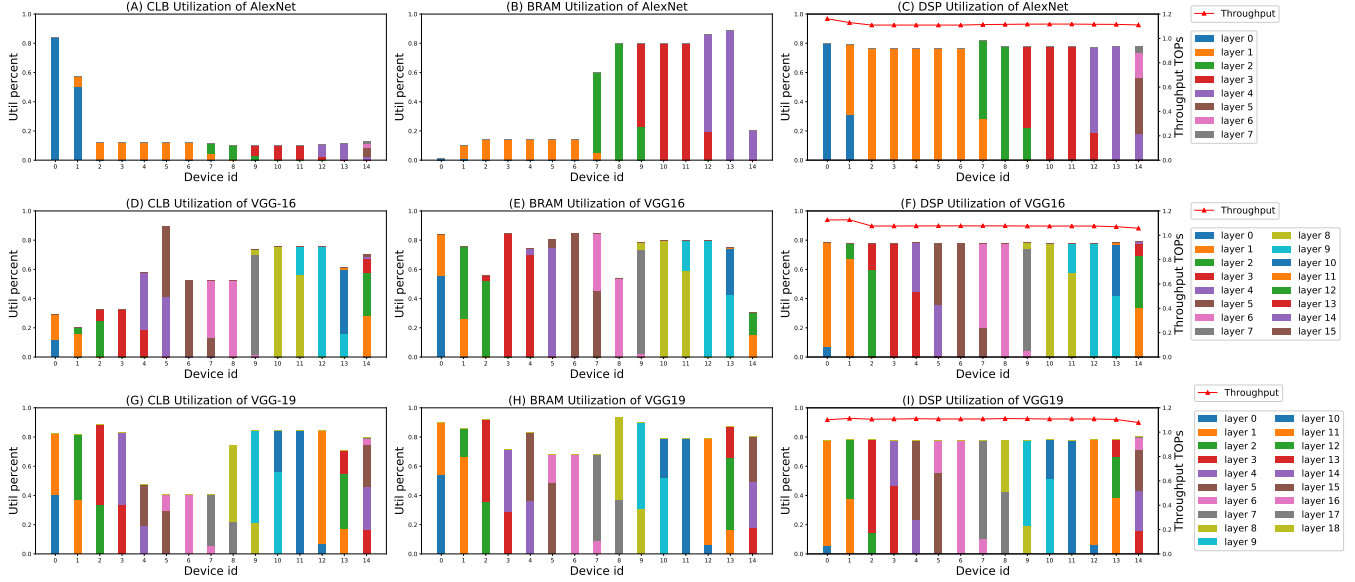


Fig. 14. Experimental results and utilization report when mapping AlexNet and VGG-16/19 to a cluster with 15 FPGAs

TABLE 6

Cluster-Level experimental result. All of these CPU/GPU/FPGA implementations use single floating point precision. [43], [23], and [24] do not show the experiment result of training time per epoch

Device	CPU [43]	GPU [43]	GPU		FPGA [43]		FPDeep		
	AMD A10	Titan X	Tesla K80		Xilinx XC7VX690T				
CNN Model	AlexNet	AlexNet	AlexNet [23]	VGG-16 [24]	AlexNet	VGG-16	AlexNet	VGG-16	VGG-19
Config	1 CPU	1 GPU	1 GPU	1 GPU	4 FPGAs	1 FPGA	15 FPGAs	15 FPGAs	15 FPGAs
Perf (GOPS)	34.23	1385	2330	2018	207 (Per FPGA)	290 (Per FPGA)	1157(Per FPGA)	1197(Per FPGA)	1220(Per FPGA)
Training time (H)/Epoch	NA	NA	NA	NA	NA	NA	0.17	2.19	2.76
Power efficiency (GOPS/J)	0.39	4.22	7.87	6.86	6.55	8.28	37.09	37.88	38.13

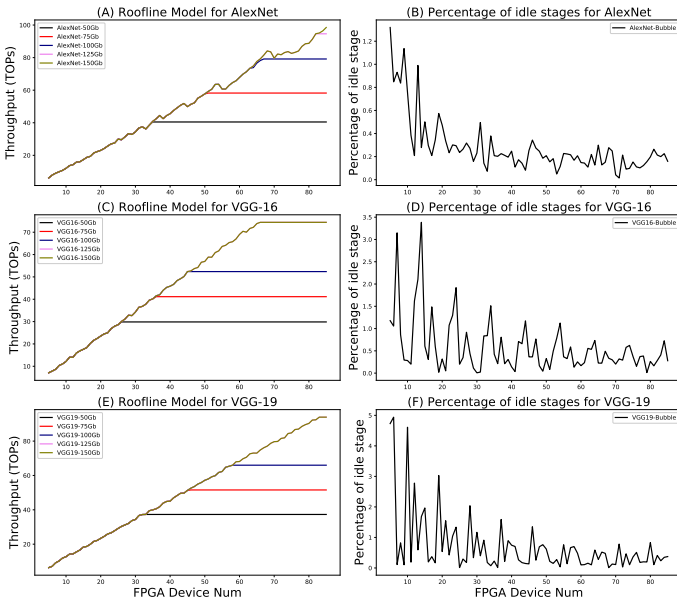


Fig. 15. Roofline Models, Percent Idle stages, and Epochs per hour of AlexNet, VGGNet-16 and VGGNet-19

training per epoch. As hybrid model/layer parallelism does not constrain the choice of mini-batch size, the optimal learning rate and mini-batch size can always be applied in SGD, leading to the minimum number of epochs needed

for training of a given accuracy. Hence, the linear speedup of training per epoch results in a linear speedup of CNN training.

Figs. 16(B)(D)(F) show the convergence rates of FPDeep and the traditional centralized DP training using the same small mini-batch size in SGD. The results show that FPDeep has similar convergence rates compared with the traditional centralized DP method, demonstrating that the slightly-unaligned weight update of FPDeep does not introduce additional training epochs. For the centralized case, we use a Sugon W740-G20 GPU server, which contains two Tesla K80 GPUs. The experiment is based on the Darknet framework; the CUDA library is cuDNN 5.0. For the FPDeep case, we use a Sugon CX50-G20 CPU cluster with an Intel Xeon E5-2680 v3 CPU. The FPDeep software simulator is compiled with gcc 7.1 and OpenMPI 2.1.1. The training dataset is CIFAR-10.

7 DISCUSSION AND FUTURE WORK

We propose a framework, FPDeep, which maps training logic of DNNs to multi-FPGA clusters with high efficiency and also automatically generates RTL implementations for target networks and clusters.

With FPDeep, clusters of FPGAs work in a deeply-pipelined manner using a 1-D topology; this enables the ac-

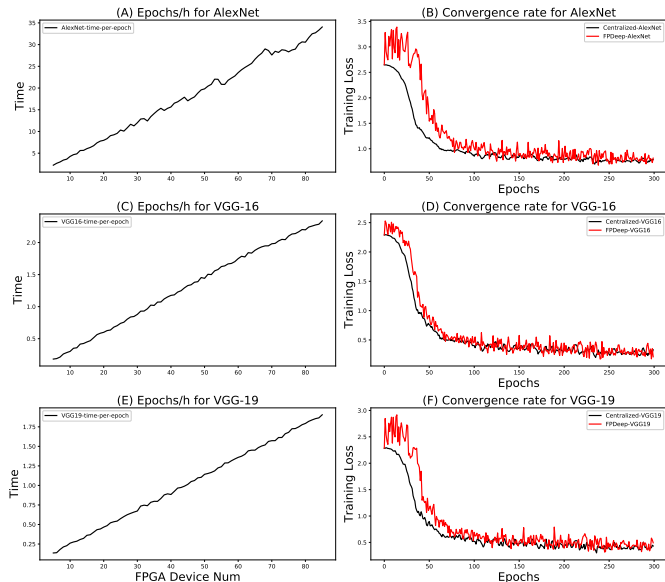


Fig. 16. Evaluation experimental setup

celerators to map directly onto existing platforms, including Catapult, Catapult2, and almost any tightly-coupled FPGA cloud or cluster. FPDeep uses two mechanisms to facilitate high-performance and energy-efficiency: 1) various fine-grained partition and mapping strategies to balance workloads among FPGAs and 2) training of CNNs is executed in a fine-grained inter- and intra-layer pipelined manner, which reduces the time that features need for backward propagation and leads to a reduction in storage demand to the point where only on-chip memory is required for CONV layers. Experiments show that FPDeep has good scalability to a large number of FPGAs. The bottleneck is inter-FPGA communication bandwidth. However, we find that with 250 Gb/s bidirectional bandwidth per FPGA, which is easily supported by current generation FPGAs, FPDeep’s performance shows linearity up to 100 FPGAs. For example, using Alexnet and the VGGNets as benchmarks, with 6 transceivers per FPGA (e.g., using a 2014-era Altera Stratix-V), FPDeep shows linearity up to 83 FPGAs. We evaluate energy efficiency with respect to GOPs/J and find that FPDeep provides 5.7x to 8.8x higher energy efficiency than GPU servers.

We briefly discuss future work. One area is supporting more complex NN models. Here, two additions are needed. First, while the current graph partitioning method supports ResNet and Inception (as described in Section 3.2), RNN and other new models require more complex graph structures. Second, support needs to be added for additional modules as described in Section 5.1. A second area is investigating benefits of hierarchical communication networks as arise when the nodes are multi-FPGA boards. Finally, another interesting question is use of off-chip memory. Currently, we only use off-chip memory when we are processing the fully connected layer. In the case of small clusters and large networks, where off-chip memory would be an option, we instead use the weight balancing scheme described in Section 3.4. In the future, as HBM becomes widespread, for very large networks it could make sense to use off-chip memory as an intermediate buffer to store activations and parameters.

ACKNOWLEDGMENTS

This research was partially supported by the DMC-CFA project and DS-HPC project under PNNL’s Laboratory Directed Research and Development Program. It was also partially supported by the U.S. DOE Office of Science, Office of Advanced Scientific Computing Research, under award 66150: “CENATE - Center for Advanced Architecture Evaluation”. The Pacific Northwest National Laboratory is operated by Battelle for the U.S. Department of Energy under contract DE-AC05-76RL01830. This work was also supported, in part, by the NSF through Awards CNS-1405695 and CCF-1618303/7960; by the NIH through Award 1R41GM128533; by grants from Microsoft and Red Hat; and by Intel through donated FPGAs, tools, and IP.

Preliminary versions of parts of this work appeared in FCCM 2018 [11] and FPL 2018 [10].

REFERENCES

- [1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.
- [2] Tal Ben-Nun and Torsten Hoefer. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *arXiv preprint arXiv:1802.09941*, 2018.
- [3] Michaela Blott, Thomas B Preußer, Nicholas Fraser, Giulio Gambardella, Kenneth O’Brien, Yaman Umuroglu, and Miriam Leeser. Scaling neural network performance through customized hardware architectures on reconfigurable logic. In *ICCD 2017*, pages 419–422, 2017.
- [4] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A cloud-scale acceleration architecture. In *MICRO 2016*, pages 1–13, 2016.
- [5] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981*, 2016.
- [6] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.
- [7] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. Serving dnns in real time at datacenter scale with project brainwave. *IEEE Micro*, 38(2):8–20, 2018.
- [8] Jason Cong, Vivek Sarkar, Glenn Reinman, and Alex Bui. Customizable domain-specific computing. *IEEE Design & Test of Computers* 2011, 28:6–15, 2011.
- [9] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. A configurable cloud-scale dnn processor for real-time ai. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pages 1–14. IEEE Press, 2018.
- [10] Tong Geng, Tianqi Wang, Ahmed Sanaullah, Chen Yang, Rushi Patel, and Martin Herbordt. A framework for acceleration of cnn training on deeply-pipelined fpga clusters with work and weight load balancing. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 394–394. IEEE, 2018.
- [11] Tong Geng, Tianqi Wang, Ahmed Sanaullah, Chen Yang, Rui Xu, Rushi Patel, and Martin Herbordt. Fpdeep: Acceleration and load balancing of cnn training on fpga clusters. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 81–84. IEEE, 2018.

- [12] Alan D George, Martin C Herbordt, Herman Lam, Abhijeet G Lawande, Jiayi Sheng, and Chen Yang. Novo-g#: Large-scale reconfigurable computing with direct and programmable interconnects. In *High Performance Extreme Computing Conference (HPEC)*, 2016 IEEE, pages 1–7. IEEE, 2016.
- [13] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.
- [14] Yijin Guan, Hao Liang, Ningyi Xu, Wenqiang Wang, Shaoshuai Shi, Xi Chen, Guangyu Sun, Wei Zhang, and Jason Cong. Fp-dnn: An automated framework for mapping deep neural networks onto fpgas with rtl-hls hybrid templates. In *FCCM 2017*, pages 152–159, 2017.
- [15] Gopalakrishna Hegde, Nachiket Kapre, et al. CaffePresso: Accelerating Convolutional Networks on Embedded SoCs. *TECS 2018*, 17(1):15, 2018.
- [16] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, Hyounjoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, pages 103–112, 2019.
- [17] Zhouyuan Huo, Bin Gu, and Heng Huang. Training neural networks using features replay. In *Advances in Neural Information Processing Systems*, pages 6660–6669, 2018.
- [18] Zhouyuan Huo, Bin Gu, Qian Yang, and Heng Huang. Decoupled parallel backpropagation with convergence guarantee. *arXiv preprint arXiv:1804.10574*, 2018.
- [19] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia 2014*, pages 675–678. ACM, 2014.
- [20] Zhihao Jia, Sina Lin, Charles R Qi, and Alex Aiken. Exploring hidden dimensions in parallelizing convolutional neural networks. *arXiv preprint arXiv:1802.04924*, 2018.
- [21] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.
- [22] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.
- [23] LeaderGPU. Tensorflow Alexnet benchmark. <https://www.leadergpu.com/articles/428-tensorflow-alexnet-benchmark>, 2018. [Online; accessed 19-July-2018].
- [24] LeaderGPU. Tensorflow VGG-16 benchmark. <https://www.leadergpu.com/articles/430-tensorflow-vgg16-benchmark>, 2018. [Online; accessed 19-July-2018].
- [25] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *OSDI*, volume 14, pages 583–598, 2014.
- [26] Ruo Long Lian. *A framework for FPGA-based acceleration of neural network inference with limited numerical precision via high-level synthesis with streaming functionality*. PhD thesis, University of Toronto, 2016.
- [27] Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and Ji Liu. Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 5330–5340, 2017.
- [28] Shuang Liang, Shouyi Yin, Leibo Liu, Wayne Luk, and Shaojun Wei. FP-BNN: binarized neural network on FPGA. *Neurocomputing* 2018, 275:1072–1086, 2018.
- [29] Liqiang Lu, Yun Liang, Qingcheng Xiao, and Shengen Yan. Evaluating fast algorithms for convolutional neural networks on fpgas. In *Field-Programmable Custom Computing Machines (FCCM)*, 2017 IEEE 25th Annual International Symposium on, pages 101–108. IEEE, 2017.
- [30] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yufeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2430–2439. JMLR.org, 2017.
- [31] Duncan JM Moss, Eriko Nurvitadhi, Jaewoong Sim, Asit Mishra, Debbie Marr, Suchit Subhaschandra, and Philip HW Leong. High performance binary neural networks on the Xeon+ FPGA platform. In *FPL 2017*, pages 1–4, 2017.
- [32] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.
- [33] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper 2015*, 2(11), 2015.
- [34] Ahmed Sanaullah, Chen Yang, Yuri Alexeev, Kazutomo Yoshii, and Martin C Herbordt. Trip: An ultra-low latency, teraflops/s reconfigurable inference processor for real-time multi-layer perceptrons. *2017 The International Conference for High Performance Computing, Networking, Storage and Analysis (Supercomputing)*, 2017.
- [35] Ahmed Sanaullah, Chen Yang, Yuri Alexeev, Kazutomo Yoshii, and Martin C Herbordt. Application aware tuning of reconfigurable multi-layer perceptron architectures. In *2018 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9. IEEE, 2018.
- [36] Jiayi Sheng, Chen Yang, and Martin C Herbordt. High performance communication on reconfigurable clusters. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 219–2194. IEEE, 2018.
- [37] Jiayi Sheng, Chen Yang, Tianqi Wang, and Martin Herbordt. High performance dynamic communication on reconfigurable clusters. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 219–219. IEEE, 2018.
- [38] Xing Sun, Zhimin Xu, Nan Meng, Edmund Lam, and Hayden K-H So. Data-driven light field depth estimation using deep convolutional neural networks. In *IJCNN 2016*, pages 367–374, 2016.
- [39] Swagath Venkataramani, Ashish Ranjan, Subarno Banerjee, Dipankar Das, Sasikanth Avancha, Ashok Jagannathan, Ajaya Durg, Dheemanth Nagaraj, Bharat Kaul, Pradeep Dubey, et al. Scaleddeep: A scalable compute architecture for learning and evaluating deep networks. *sigarch comput. archit. news* 45, 2 (june 2017), 13–26, 2017.
- [40] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. Automated systolic array architecture synthesis for high throughput cnn inference on fpgas. In *Proceedings of the 54th Annual Design Automation Conference 2017*, page 29. ACM, 2017.
- [41] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google’s neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.
- [42] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170. ACM, 2015.
- [43] Chen Zhang, Di Wu, Jiayu Sun, Guangyu Sun, Guojie Luo, and Jason Cong. Energy-efficient cnn implementation on a deeply pipelined fpga cluster. In *ISLPED 2016*, pages 326–331. ACM, 2016.
- [44] Ruizhe Zhao, Xinyu Niu, Yajie Wu, Wayne Luk, and Qiang Liu. Optimizing cnn-based object detection algorithms on embedded FPGA platforms. In *ARC 2017*, pages 255–267, 2017.
- [45] Wenlai Zhao, Haohuan Fu, Wayne Luk, Teng Yu, Shaojun Wang, Bo Feng, Yuchun Ma, and Guangwen Yang. F-cnn: An fpga-based framework for training convolutional neural networks. In *ASAP 2016*, pages 107–114, 2016.



Tianqi Wang received his Bachelor of Applied Physics from the University of Science and Technology of China in 2012 and is pursuing a PhD degree from University of Science and Technology of China. He is currently a visiting scholar at Boston University and is also working on accelerating N -body problems on FPGA-centric clusters.



Tong Geng received the BS from Zhejiang University, China in 2013 and the MS from Eindhoven University of Technology in 2015. He is working toward the PhD degree in Electrical & Computer Engineering at Boston University. His research interest includes machine learning, computer architecture, parallel computing, and reconfiguration computing.



Ang Li received his Bachelor degree from the CS department of Zhejiang University, Hangzhou, China, in 2010. From 2010 to 2012, he worked in industry as a software developer. In 2016, he received joint PhD degrees from the ECE department of National University of Singapore and the EE department of Eindhoven University of Technology. He is now a research scientist in the HPC group of the Pacific Northwest National Laboratory, USA.



Xi Jin is Associate Professor of Physics Department at University of Science and Technology of China where he directs the SoC Lab. His group has worked in accelerating scientific computing applications with FPGAs and using FPGA to build scientific instruments.



Martin Herbordt is Professor of Electrical and Computer Engineering at Boston University where he directs the Computer Architecture and Automated Design Lab. His group has worked for many years in accelerating HPC applications with FPGAs and on system aspects of FPGA clusters and clouds.