Toward Multi-FPGA Acceleration of the Neural Networks

SAMAN BIOOKAGHAZADEH, PRAVIN KUMAR RAVI, and MING ZHAO,
Arizona State University

High-throughput and low-latency Convolutional Neural Network (CNN) inference is increasingly important for many cloud- and edge-computing applications. FPGA-based acceleration of CNN inference has demonstrated various benefits compared to other high-performance devices such as GPGPUs. Current FPGA CNN-acceleration solutions are based on a single FPGA design, which are limited by the available resources on an FPGA. In addition, they can only accelerate conventional 2D neural networks. To address these limitations, we present a generic multi-FPGA solution, written in OpenCL, which can accelerate more complex CNNs (e.g., C3D CNN) and achieve a near linear speedup with respect to the available single-FPGA solutions. The design is built upon the Intel Deep Learning Accelerator architecture, with three extensions. First, it includes updates for better area efficiency (up to 25%) and higher performance (up to 24%). Second, it supports 3D convolutions for more challenging applications such as video learning. Third, it supports multi-FPGA communication for higher inference throughput. The results show that utilizing multiple FPGAs can linearly increase the overall bandwidth while maintaining the same end-to-end latency. In addition, the design can outperform other FPGA 2D accelerators by up to 8.4 times and 3D accelerators by up to 1.7 times.

CCS Concepts: • **Hardware → Emerging technologies**; **Analysis and design of emerging devices and systems**; **Emerging architectures**; • **Computing methodologies → Machine learning**; • **Computer systems organization → Architectures**; **Other architectures**; **Reconfigurable computing**; **Heterogeneous (hybrid) systems**;

Additional Key Words and Phrases: FPGA, neural networks, distributed systems

ACM Reference format:
Saman Biookaghazadeh, Pravin Kumar Ravi, and Ming Zhao. 2021. Toward Multi-FPGA Acceleration of the Neural Networks. *J. Emerg. Technol. Comput. Syst.* 17, 2, Article 25 (April 2021), 23 pages.
https://doi.org/10.1145/3432816

1 INTRODUCTION

In recent years, FPGAs have received tremendous attention in the world of neural network acceleration. FPGAs can provide unique benefits to accelerate Convolutional Neural Networks (CNNs). First, FPGAs can guarantee tight latency bounds for incoming requests. Conventional CNN accelerators (i.e., GPUs) have shown the ability for the acceleration of a batch of requests by leveraging their farm of processing cores. Unfortunately, they lack the potential to guarantee

This work was supported by National Science Foundation awards CNS-1619653, CNS-1562837, CNS-1629888, and CNS-1955593.
Authors' address: S. Biookaghazadeh, P. K. Ravi, and M. Zhao, Arizona State University, School of Computing, Informatics, and Decision Systems Engineering, 699 S. Mill Avenue, Tempe, AZ 85281; emails: {sbiookag, pravi8, mingzhao}@asu.edu.

1550-4832/2021/04-ART25 $15.00
https://doi.org/10.1145/3432816

low-latency services for individual requests [1, 2]. In contrast to GPUs, FPGAs can leverage their reconfigurable deep pipeline to service the requests in a streaming fashion and provide a predictable low latency. Second, conventional processors are usually power hungry, which makes them challenging to deploy in power- or energy-constrained environments. Differently, FPGAs are highly power efficient due to their low operational clock frequency. In conclusion, FPGAs are considered as an excellent platform for accelerating CNNs for deployment.

The ever-increasing complexity of emerging CNNs requires FPGAs with a higher amount of resources, such as memory bandwidth and logical units, to achieve low-latency and high-throughput inferences. Even high-end FPGA chip technologies can host only a small section of a whole CNN model. For example, the Intel Stratix 10 FPGA can perform only 5,000 multiply-accumulation (MAC) operations per clock cycle, which is even less than the total number of operations for a single layer of a typical CNN, such as VGG-16 or ResNet. As a result, they fall short in handling heavier CNNs for ultra-low latency (less than 10 ms) and high-throughput (more than 60 images/frames per second). Such a problem is even more significant for accelerating more computationally intensive operations, such as 3D convolutions, which show great potential in video processing applications. This challenge can be potentially addressed by utilizing a cluster of FPGAs, connected through a high-bandwidth communication infrastructure.

Achieving linear speedup using a multi-FPGA solution is not straightforward. First, we need to have an efficient design on a single FPGA and achieve state-of-the-art performance. Such performance benefits should be reflected in the acceleration of various CNN operations. Second, the pipeline of multiple FPGAs should be correctly managed to ensure that all FPGAs are doing useful works to handle incoming requests. Third, CNN partitioning, which is the process of mapping different parts of the model onto different FPGAs, should be done intelligently to make sure the workload is balanced across the FPGAs.

Related works [1, 3] have studied the multi-FPGA acceleration of neural networks. These works come with several limitations. First, they do not provide a general architecture to accelerate various types of CNNs. For example, they are only able to accelerate either 2D or 3D convolutions, but not both. Second, they do not optimally exploit the FPGA acceleration resources, which leads to sub-optimal performance compared to the maximum theoretical performance of an FPGA. Third, they are designed and developed, using low-level hardware programming languages (Jiang et al. [3] used Xilinx HLS), which makes it difficult to extend and support by the widely used deep learning frameworks, such as TensorFlow [4] and Caffe [5].

In this article, we present a novel multi-FPGA CNN accelerator that can leverage a deep pipeline of FPGAs, connected through a high-performance I/O channel. First, we adopted the Intel Deep Learning Accelerator (DLA) [6] architecture and applied various optimizations to achieve an efficient design on a single FPGA. Using a novel systolic array design, our architecture has reduced the total resource consumption of the DLA by up to 25% and increased the overall performance by 24%. We developed this design using OpenCL, which enables convenient integration with widely used deep learning frameworks. In addition, it enables the integration of the accelerator in a heterogeneous environment, where the same OpenCL code can run across different processors. Second, we extended the design to support data communication with other FPGAs in the pipeline, using a 40-Gb/s QSFP+ I/O channel. Using a network of connected FPGAs enables temporal (distributing the layers onto different FPGAs) and spatial (splitting a single layer and mapping it onto multiple FPGAs) parallelization of the layers. Using this configuration, a user can allocate a set of FPGAs in a network, with no prior information about the network architecture. The user can interact with these FPGAs as a single FPGA with a large number of resources. Further, the user can select a neural network model and deploy it on these FPGAs. The framework can automatically split the model into several sub-models and deploy each sub-model onto an FPGA. This cluster of FPGAs can provide the same or better latency and energy efficiency compared to the available CPU or GPU solutions. Third, we extended the design to support 3D convolutions, on top of 2D convolutions, for certain types of emerging CNN applications. Fourth, we developed a model and strategy for optimizing the partitioning and the placement of the CNN layers on the set of available FPGAs in the pipeline.

To demonstrate the feasibility of our framework, we performed multiple experiments using different widely used CNN models. Our CNN models for the experiments are VGG-16, AlexNet, and ResNet, which are 2D models commonly used for image classification, and C3D, which is a 3D model commonly used for video processing. We deployed these models on single- and multi-FPGA pipelines. Our results show that using the multi-FPGA configuration can increase the throughput, almost linearly, with respect to the total number of FPGAs. In addition, our extended systolic array shows superior performance (up to 1.7 times), compared to other related works, for accelerating the 3D convolution-based CNN architectures.

The rest of the article is organized as follows. Section 2 introduces the background, Section 3 describes the methodology, Section 4 presents experimental results, and Section 5 presents the related works. Section 6 concludes the article.

2 BACKGROUND

2.1 Convolutional Neural Networks

CNNs are the main building blocks in many AI applications, such as image classification [7], reinforcement learning [8], and natural language processing. CNNs are also showing promising results in more complex domains such as video understanding [9, 10, 11, 12, 13]. Almost all CNNs are considered as a chain of various operations, such as convolution (2D or 3D), matrix multiplication (MM), pooling, and ReLU. In CNN, the data is processed by one operation, and the result is handed over to the next operation in the chain. In a 2D convolution operation, the input data is just composed of multiple input channels, where each input channel is a 2D structure of numerical values. In a 3D convolution operation, the input data is not only composed of multiple input channels, but each input channel contains data from different instances in a time frame, where the instances should be sequential in that specific time frame. Equation (1) and Equation (2) describe the 2D and 3D convolutions, where m represents a specific output channel, f represents a specific frame number, w and h represent width and height location, respectively, in the output, CH_{in} represents the total number of input channels, and n, k, i, and j represent the iterator indexes on the input channels, frames, and width and height locations of the convolution kernel.

$$OUT[m][w][h] = \sum_{n=0}^{CH_{in}} \sum_{i=0}^{K_w} \sum_{j=0}^{K_h} WEIGHT[m][n][i][j] \times IN[n][stride \times w + i][stride \times h + j] \quad (1)$$

$$\begin{aligned} OUT[m][f][w][h] = &\sum_{n=0}^{CH_{in}} \sum_{k=0}^{K_f} \sum_{i=0}^{K_w} \sum_{j=0}^{K_h} WEIGHT[m][n][f][i][j] \\ &\times IN[n][stride \times f + k][stride \times w + i][stride \times h + j] \end{aligned} \quad (2)$$

CNNs are highly computationally intensive, due to a large number of mathematical operations (hundreds of thousands and even up to millions) that each layer of the network involves. Among all of the widely used operations in the neural networks, convolutions and MMs (also known as fully connected (FC)) are the most significant contributors to the total execution time for one round of inference on a simple neural network. Table 1 reports the contribution of three primary operations in the VGG-16 model in terms of the total number of arithmetic operations (e.g., MAC, min, and max) and the parameter size. *Ops* and *Data* columns represent the total number of arithmetic

Table 1. Total Contribution of Major Operations in the VGG-16 CNN Model in Terms of Total Number of Arithmetic Operations and Input/Weight Parameters

Operation (2D)	Ops	Data
Convolution	99.19%	8.61%
Matrix multiplication	00.79%	91.38%
Pooling	00.00%	00.00%

Table 2. Total Contribution of Major Operations in C3D CNN Model in Terms of Total Number of Arithmetic Operations and Input/Weight Parameters

Operation	Ops	Data
Convolution (3D)	99.9%	26.72%
Matrix multiplication	00.1%	73.28%
Pooling	00.00%	00.00%
ReLU	00.00%	00.00%

operations and parameters (weights and inputs) involved in that operation, respectively. The convolution operations (2D) contribute more than 99% of the total arithmetic. The MM operations contribute more than 91% of input and weight data access from global memory, which can consume a considerable portion of the total runtime.

Compared to 2D convolutions, 3D convolutions have higher computational complexity, due to the existence of an extra dimension (usually frame), which enables spatio-temporal feature recognition in continuous video frames. Table 2 lists the total contribution of 3D convolutions and other operations in the C3D model. The convolution operations (3D) contribute more than 99% of the total operations. The MM contributes to more than 73% of data access.

Based on the preceding observations, in this article we mainly focus on the acceleration of the convolution (2D and 3D) and the MM operation, which are the main contributors to the end-to-end execution time. We omit the discussion of the other types of layers, such as pooling and ReLU, due to their simplicity and lack of impact on the runtime of the CNNs.

2.2 Winograd Algorithm

Winograd transformation [14] is a proven method to reduce the complexity of MAC operation in hardware design. Using this technique for convolutions can ultimately reduce the arithmetic complexity. Shen et al. [15] showed that using the Winograd algorithm can reduce the total number of multiplications by 58%. In addition, Winograd becomes more practical for smaller filter sizes, such as 3×3, which is quite common in many neural networks. In our design, we utilize the 2D Winograd algorithm to accelerate both 2D and 3D convolutions on the FPGAs. To demonstrate the Winograd algorithm, we will start with an example of a 1D convolution. In the Winograd algorithm, we denote a 1D convolution as $F(M, R)$, where M and R represent the size of the input and the filter. The typical convolution computation is given by

$$O_i = \sum_{r=0}^{R-1} W_r I_{i+r}, \tag{3}$$

where I, O, and W denote the input, output, and filter data. By using the Winograd algorithm, the output can alternatively be derived as follows [16] (we consider the Winograd algorithm for

$F(2,3)$):

$$O = M[(Sx) \cdot (Ww)], \tag{4}$$

where M, S, and W are transformation matrices with values of

$$S = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}, \quad W = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{-1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{-1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}, \quad M = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}. \tag{5}$$

The preceding method can be extended for 2D convolutions as well. Considering the 2D Winograd algorithm $F(m \times m, r \times r)$, it can be calculated using the following equation:

$$O = M[(SxS^T) \cdot (WwW^T)]M^T. \tag{6}$$

2.3 Hardware Acceleration

Hardware acceleration is a crucial enabler of the CNN applications. Due to the computational intensity of the CNNs (mainly convolution and MM), CPUs cannot deliver a reasonable performance for latency-critical applications, such as object detection for self-driving cars, due to the limited parallelism capability. This problem leads to the utilization of hardware accelerators, such as GPUs, FPGAs, and TPUs [17]. Unlike CPUs, hardware accelerators can exploit their massive parallelism to split major functions into thousands of parallel operations and ultimately reduce the overall computation time. GPUs have been extensively studied and utilized for the acceleration of both inference and training. Widely used deep learning frameworks, such as TensorFlow [4] and Caffe [5], rely on GPUs to deliver acceptable performance for both the training and the inference. Recently, FPGAs have captured the right amount of attention due to their flexibility and reconfigurability. FPGAs are proven to be able to provide much lower latency, compared to CPU and GPU, for applications with latency-critical conditions [1, 2]. In addition, they can provide much better energy efficiency compared to CPUs and GPUs, which is crucial for energy-restricted environments, such as edge computing [1, 18].

Recent advancements in high-level languages have made it easy to program and use FPGAs for various applications. For example, developers can use C or C++ to describe their algorithm and compile and deploy it on a target FPGA. FPGA vendors have integrated OpenCL, a heterogeneous parallel programming language, with their FPGAs. OpenCL has several benefits for software developers and systems designers. It provides ease of development by keeping a higher abstraction, at the cost of an acceptable performance loss. In addition, it enables software engineers to take advantage of the ultimate performance and the energy efficiency of FPGAs. Using OpenCL, developers can describe their algorithm in standard representation and target all available accelerators, such as GPUs, CPUs, and DSPs. To port OpenCL across different platforms, a developer needs only to make minor modifications to utilize the unique features of the target platform fully. OpenCL also enables further integration with widely used deep learning frameworks, such as Caffe and TensorFlow. These frameworks provide an OpenCL adapter (Figure 1), which enables the utilization of any OpenCL-compatible accelerator. This work is fully developed in OpenCL, using the Intel OpenCL SDK toolchain [19].

Several related works have studied the acceleration of the 2D [2, 6, 20–23] and 3D CNNs [10, 11, 13, 24] on FPGAs. Other related works [1, 3] have studied the feasibility of using multiple FPGAs for increasing the throughput or decreasing the latency of the CNN accelerators. However, these works cannot deliver state-of-the-art performance and are not designed to support different

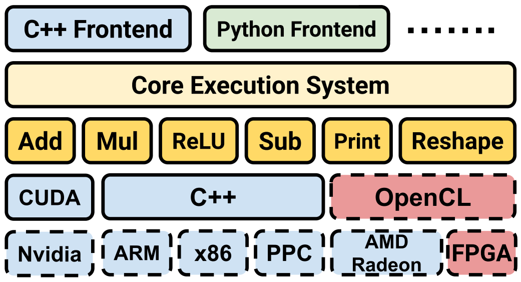


Fig. 1. General architecture of deep learning frameworks.

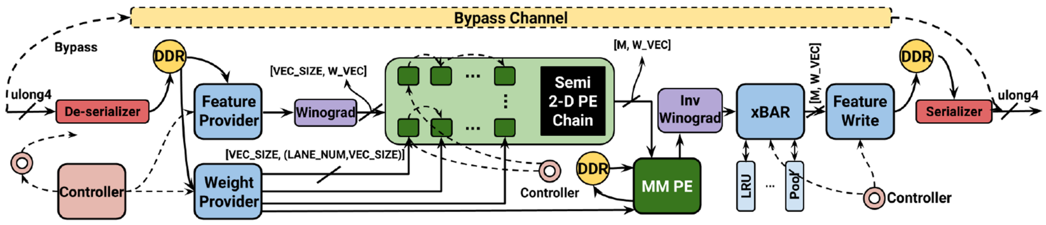


Fig. 2. CNN accelerator architecture.

types of convolutions in a single architecture. In addition, they are all implemented with low-level hardware languages, which makes them hard for further extensions and improvements. Our design is built on top of the DLA [6] architecture. Boutros et al. [25] made a comparison between widely known CNN accelerators on FPGA and showed that DLA is the fastest available solution. However, DLA lacks several important optimizations and critical features. For example, the systolic array needs enhancements for lower resource consumption and higher throughput. In addition, weight and input organization can be changed for better memory utilization. From the usability perspective, it works for only the default 2D convolution but cannot support the more complex 3D convolution. Finally, it does not support the multi-FPGA acceleration, which is important for complex CNNs. Our design is built on top of DLA while addressing all of the preceding limitations.

3 METHODOLOGY

In this section, we discuss the overall architecture of our CNN accelerator and the host side manager for distributing and accelerating of a target neural network model. More specifically, first, we discuss the anatomy of the basic building blocks of our CNN architecture. Second, we describe how this CNN architecture is further extended to support 3D convolutions for use cases such as video processing. We also explain our method for choosing the best strategy for mapping 3D convolution onto our native 2D convolution accelerator. Third, we describe the multi-FPGA support for our design architecture. Last, we discuss our algorithm for the efficient mapping of various layers of a CNN onto the available chain of the FPGAs.

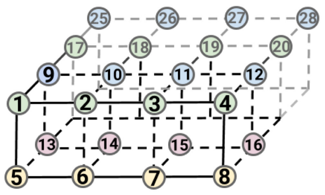
3.1 CNN Accelerator Architecture

Our CNN accelerator skeleton has adopted the 1D systolic array architecture from the DLA [6]. We applied various optimizations on the DLA to achieve higher performance and lower resource utilization. We also extended it to support 3D convolutions and data transmission with other FPGAs. Figure 2 depicts the overall architecture of our CNN accelerator design. Our design consists of several key components. First, the *Controller* acts as the coordinator between all other components.

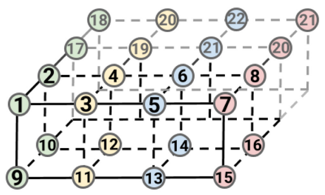
The *Controller* sends the respective configuration parameters to the *Feature Provider*, Processing Elements (PEs), and so forth. These parameters are usually the type of the layer, size of the input data for processing, and all other related necessary parameters for executing a layer. Second, the *Feature Provider* is responsible for reading the data in a fixed size and feeding it into the first PE. This component reads the data from the global memory, caches it in the respective local memory, and sends it over a channel to the first PE in the systolic array, which is a grid of connected PEs. The channel is a communication medium between two components in the same kernel or different kernels on separate FPGAs. Third, the *Weight Provider* takes care of updating the weight buffers in all of the PEs. Since the number of the PEs is usually less than the number of output channels, the PE needs to update the weight buffers multiple times while handling a single layer. Fourth, the Cross Bar (xBar) and all attached *activation layers* apply the required transformation (pooling, ReLU, etc.), after each convolution or MM for each layer. Fifth, the *Feature Writer* receives the output from the *xBar* and writes it back to the global memory. Latter layers further use this data. Sixth, the *Serializer* and the *Deserializer* are responsible for receiving/sending the intermediate results from/to the previous/next FPGA in the chain of the FPGA cluster, respectively. Seventh, the Winograd and inverse (inv) Winograd convert the data into Winograd format and revert the data into the normal representation, respectively. Finally, the controller activates the *Bypass Channel* if the FPGA handles a sub-section of a layer (sub-layer), instead of a whole layer or a group of layers. This channel is responsible for bypassing the partial results from processing a sub-layer to the next FPGA and potentially to the FPGA that handles the last sub-layer of a specific layer. The FPGA that handles the final sub-layer concatenates all partial results and generates the complete output for that layer. We need to mention that the *Bypass Channel* is deactivated if the FPGA is handling the first of the last sub-section of a layer.

Feature Provider. The *Feature Provider* is responsible for reading the input data from the global memory and feeding it into the first PE. Further, each PE forwards the received data to the next available PE in the chain. Reading data from the global memory is critical in the accelerator design. Non-optimized data read from the global memory can lead to major stalls (low throughput) in the pipeline. To maximize the throughput, our design leverages two main optimizations: (1) memory data access coalescing and (2) caching data in the local memory. First, every access to the global memory requires hundreds or thousands of clock cycles. At the same time, the global memory can provide only a chunk of data in every memory access. As a result, to reduce the overall memory waiting time, it is critical to reducing the memory access frequency by coalescing multiple load/store requests. By doing so, the *Feature Provider* can almost saturate the external memory bandwidth and, consequently, minimize the total number of memory transactions. Second, the local memory can be up to 100 times faster than global memory. Hence, caching and reusing data on the local memory can provide significant performance benefits.

For efficient global memory data access, we applied data rearrangement on the input data. Previous works [2, 6] demonstrated the importance of memory interface bit width, which is the total number of bits that can be accessed in one memory transaction, and burst length, which is the total amount of data that is going to be fetched from memory in multiple sequential memory accesses, to performance. For example, for the Intel Arria 10 FPGAs, a minimum bit width of 512 and a burst length of above 128 KB are required to saturate the 16 GB/s per bank memory bandwidth. The traditional row-major data representation (used in DLA [6]) has limitations to achieve efficient burst length due to discontinuous DRAM access. To alleviate this problem, we propose a partial input-channel-major data arrangement. Figure 3 represents both the traditional and the new data arrangement. In this new method, the host divides the input channel section into multiple chunks of size *VEC_SIZE*. Further, it iterates over the data in a row-major manner, but instead of storing



(a) Logical view of the traditional data layout



(b) Logical view of the new data layout

Feature Data Address	1	2	3	4	5	6	7	8	9	...	32
DRAM Data Address	0x00	0x08	0x01	0x09	0x02	0x10	0x03	0x11	0x04	...	0x31

(c) Traditional data layout arrangement on the DRAM

Feature Data Address	1	2	3	4	5	6	7	8	9	...	32
DRAM Data Address	0x00	0x01	0x02	0x03	0x04	0x05	0x07	0x08	0x09	...	0x31

(d) New data layout arrangement on the DRAM

Fig. 3. Traditional feature data arrangement versus new data arrangement. In the traditional arrangement, the data is first stored by rows and then the input channels. In the new arrangement, for each row we store a set of input channels, sequentially.

a single data item, it stores the whole *VEC_SIZE*. This data rearrangement is applied to the input data before it is streamed into the FPGA.

The *Feature Provider* needs to send the data in the right format and size to the PEs so that they can perform the convolution correctly. Starting from a convolution, each PE has to convolve a collection of weight parameters for a single output channel, with a section of the input data, which has the same width, height, and input channel size. We call this piece of data a *brick*, with a size of $Width \times Height \times IN\_CH\_SIZE$. Due to the utilization of Winograd, the $Width$ is always equal to *W_VEC* (in our case, it equals eight). The *Feature Provider* does not send the whole brick at once, but instead sends it in the granularity of $Width \times 1 \times VEC\_SIZE$, which we call a *plate*. Each plate has a width of 8, a height of 1, and a depth of VEC_SIZE. The total number of plates in each brick is equal to $Height \times (IN\_CH\_SIZEVEC\_SIZE)$. Due to the arrangement of the data in the global memory, reading each plate or multiple sequential plates leads to fully sequential data access in the memory. Sequential data access helps in reading more data in fewer transactions and leads to better utilization of the memory bandwidth. The new data arrangement places the required data sequentially in the memory and enables the *Feature Provider* to load the required data with the minimum number of memory accesses.

Weight Provider. The *Weight Provider* updates the weight buffers on all PEs while servicing a layer. In our design, each PE generates all particular features for a single output channel. Since the total number of PEs (it is 32 in our design, based on the available DSPs) is usually less than the actual number of output channels (up to 8,096 in different models), they can only generate the features for a certain number of output channels (32). For the rest of the outputs, the controller needs to refresh the PEs with the new set of weight parameters and initiate the same process as the previous round. Finally, similar to the *Feature Provider*, the *Weight Provider* should maximize the DDR bandwidth (the available data transfer rate between the global memory and the weight and input providers) to minimize the overall stall while reloading the weight buffers on the PEs.

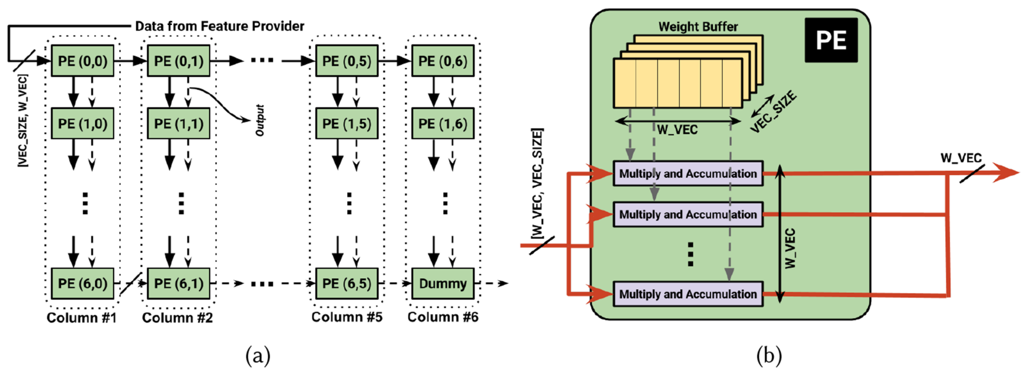


Fig. 4. (a) PE semi-1D structure. (b) PE architecture.

As a result, the weights are reordered, similar to the input data (Figure 3), to enable efficient burst length and bit width.

Processing element. PEs are the main building blocks of the design, for the computation of the convolutions and the MMs. In our design, PEs are all arranged in a semi-1D systolic array fashion. We call it semi-1D, since it adopts the original 1D systolic array while the outputs from the PEs are forwarded in a 2D fashion for area optimization purposes. Figure 4(a) represents the general architecture of the array of PEs. Each PE has a dedicated channel to the next PE in the same column, which forwards the arrived input data (a plate) to the next PE. In contrast, the first PE of each column also forwards the data to the next first PE of its neighbor column so that each column can have access to the input data. Doing so helps PEs avoid reading the same piece of data directly from memory, which leads to significant performance and area overhead. The *Feature Provider* streams the data only to the first PE of the first column, and the PEs transmit the same piece of data to the other PEs. Doing so reduces the effort for the wiring between the *Feature Provider* and the PEs, and makes the process of fitting the model on the FPGA more manageable during the compilation.

Our novel semi-1D systolic array architecture is mainly designed to reduce the resource consumption of the connections between every two consecutive PEs. In a traditional systolic array architecture, the total number of wires between PEs i and $i+1$ is equal to $(i+1) \times W\_VEC$. As we go through the PEs in the systolic arrays (increasing the i), we observe a higher number of wires consumption. The total number of wires for an architecture with P PEs will be equal to $P * (P+1))/2 * W\_VEC$. In the semi-1D architecture, we arrange the PEs in a grid fashion, with n rows and m columns ($n \times m =$ P), as shown in Figure 4. In this architecture, the number of wires between each two PEs in a row is following the same pattern as the traditional design, whereas going from one row to another resets the value of i to 1, which significantly reduces the number of wires. We have some extra wiring between PEs in a column for bypassing the computed data. In each column, the total number of wires between the PEs with indexes (i, j) and $(i, j+1)$ ((i, j) is the index of the PE in the grid) is equal to $(j+1) \times W\_VEC$. As a result, the total number of wires in the design is $C \times m + D$, where C is total number of wires in a column and is given by $(n \times (n+1))/2 \times W\_VEC$, and D is the total number of output wires in the last row and is given by $(m \times (m+1))/2 \times n \times W\_VEC$. The total number of wires in the semi-1D design is less than the traditional design number. Using the semi-1D arrangement for our design (32 PEs), compared to the traditional design in DLA [6], we can reduce the total number of output ports by 65%. Respectively, it reduces the total Flip-Flops (FFs) consumption by up to 25%.

The number of PEs (P) can affect the area efficiency of the semi-1D architecture. For example, for a design with 29 PEs, we cannot have a perfect grid with exactly 29 PEs. The best grid will be an 8×4 grid, with 3 PEs that only work as bypassing data. This configuration would consume more resources than necessary and can lead into inefficient utilization of the available resources on the chip. We need to mention that this problem can be alleviated by configuring the extra PEs to take care of the next input dataset. This approach requires additional scheduling and management from the controller.

In another design architecture, we eliminated the output channels and instead used data channels to transfer the outputs. This design is not fully pipelined since data and output are sharing the same channel, and the data feeding process stalls while PEs are generating the output. This optimization increases the execution time by 4% for C3D and VGG-16 while reducing total FF consumption by 32%. Since our framework is ultimately focused on the overall performance, we prefer the previous design architecture.

Figure 4(b) depicts the overall architecture of the PEs. Each PE has *W_VEC* number of MACs. Each MAC receives an array of data with the length of *VEC_SIZE*. Further, it fetches the respective array of weights of the same size from the buffer and performs an element-wise MAC between these two arrays. The output of all MACs (an array of size *W_VEC*) is added and stored into a local buffer of the same size, which acts as a temporary buffer for the partial accumulation. To fully process a convolution, each PE iterates the preceding process for $Height \times (IN\_CH\_SIZE \div VEC\_SIZE)$ number of times. After fully iterating a single convolution, the PE streams the content of the intermediate results buffer to the output channel.

FC PE. CNNs usually consist of convolutional FC layers and simple FC layers. Convolutional FCs are the layers that connect the earlier convolution layers to the very first FC layers and have higher memory intensity. For example, the single convolutional FC layer in VGG-16 (Layer 13) has at least a 4.3 times higher number of parameters compared to all other layers. Simple FC layers receive the input from a previous FC layer and perform MM. Simple FCs have a higher data to computation ratio compared to the convolutions. Such a difference requires designers to optimize the FC operations on a target hardware architecture. Prior CPU and GPU implementations use the regular FC representation to utilize available libraries, such as MKL on Intel CPUs and cuBLAS on Nvidia GPUs. Unfortunately, using regular FC representation for convolutional FC layers can impact the performance and introduce significant data duplication overhead, which can overwhelm the FPGA bandwidth-limited DDR. Zhang et al. [2] showed around 25 times overhead for using the preceding approach.

The optimal acceleration of FC layers requires efficient mapping of these layers onto the FPGA hardware. A common approach is to map the FC MM onto the available systolic array. To map the FC layer onto the systolic array, the user has two options: (1) input-major and (2) weight-major mappings. Zhang et al. [2] studied the efficiency of these mappings onto their 2D systolic array architecture for a batch of input data. Since our design performs inferences on a single input at a time, the previous observations may not be applicable anymore. In addition, the architectural differences between the two designs can affect the choice of mapping. As a result, we experimented with both input-major and weight-major mappings for a set of FC layer configurations. For the input-major mapping, the input channel dimension is divided by *W_VEC*, where *W_VEC* number of arrays of the size *VEC_SIZE* are mapped onto each plate of the *Feature Provider*. Doing so enables efficient utilization of the computation capacity that exists in similar convolution layers and reduces the total number of iterations required for the output calculation. For the weight-major mapping, each *PE* fully loads the input data (instead of the weights), and the *Feature Provider* feeds

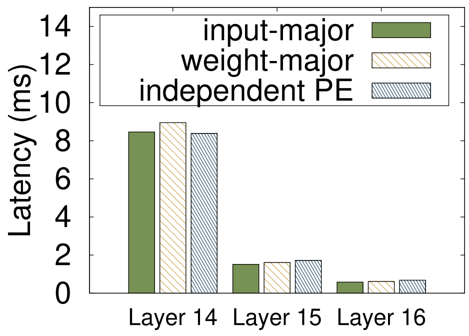


Fig. 5. Performance for different mappings of the VGG-16 MM operations.

Table 3. Performance Comparison between 32-PE, 1-PE, and Theoretical for Acceleration of the FC Layers

Input/Output	256	512	1024	2048	4096
	32-PE | 1-PE | Theor.				
2048	0.27 ms | 0.34 ms | 0.04 ms	0.26|0.39|0.08	0.42|0.46|0.17	0.58|0.65|0.33	0.94|1.06|0.67
4096	0.3|0.4|0.08	0.36|0.47|0.17	0.58|0.69|0.33	0.93|0.95|0.67	1.51|1.63|1.33
8192	0.43|0.54|0.17	0.51|0.65|0.33	0.94|0.96|0.67	1.56|1.63|1.33	2.89|2.94|2.67
16384	0.53|0.64|0.33	0.86|0.94|0.67	1.52|1.6|1.33	2.94|2.91|2.67	5.57|5.65|5.33
25088	0.77|0.78|0.51	1.24|1.42|1.02	2.27|2.29|2.04	4.36|4.3|4.08	8.46|8.3|8.12
32768	0.9|0.94|0.67	1.56|1.64|1.33	2.86|2.98|2.66	5.57|5.49|5.28	10.91|10.75|10.62

the weight parameters into the *PE* array, in the same fashion. Doing so enables higher data reuse rate for the input data and reduces the load from the external memory.

Both the input-major and weight-major mappings' performance are bounded by the global memory bandwidth. In contrast to convolutions, in MM operations, the *Weight Provider* needs to update the PEs in both input-major and weight-major mappings frequently. As a result, for single input inference, the total latency of the MM is bounded by the memory bandwidth. Figure 5 represents the latency of MM for the last three layers of the VGG-16 model. Based on these results, both weight-major and input-major mappings provide the same latency.

Unlike all previous related works, we propose the independent MM acceleration approach. This independent PE is responsible to accelerate the MM operations, without blocking the rest of the systolic array. As mentioned earlier, the computation of the FC layers is bounded by the total global memory bandwidth. As a result, the common strategy (used by all previous related works) of accelerating MMs on the available systolic array cannot fully exploit the parallelism of the PE systolic array. Table 3 represents the experimental performance evaluation of various FC layers and the theoretical maximum performance, considering the number of parameters and the effective bandwidth of the global memory (12 GBps). The input and output sizes are based on the typical FC layer dimensions in practical neural networks. The results confirm that using all available PEs cannot guarantee performance improvements compared to the theoretical performance cap. We need to mention that, on average, all of our results are 0.25 ms slower than the theoretical maximum performance. This difference is due to the overhead of the whole design, which includes the startup time of the OpenCL stack, on both the host and the device. We expanded our experiments to find the essential number of PEs to efficiently accelerate the FC layers and avoid wasting extra computing resources. We modified the total number of PEs and evaluated the performance of the design. Our results confirm (shown in Table 3) that a single PE is sufficient to saturate the memory bandwidth thoroughly and can provide the same performance as 32 PEs.

Using a separate PE for FC layers also enables our accelerator to co-locate the processing of the last FC layers of one input and the first convolutions of the next input in time-shared scenarios. As a result, the FPGA pipeline will not be blocked by the MM, which increases the delay for receiving new inputs for processing. For example, the new design reduces the interval of receiving new input requests for VGG-16 from 30.02 to 26.52 ms.

Winograd transformers. The Winograd transformation technique is used to increase the number of operations per clock cycle. Our design utilizes this technique by transforming the input feature data into the Winograd representation. In this design, we set the length of the Winograd data in the x-axis to 8. This stage is handled by the *Winograd Transformer.* The generated data is fed to the array of PEs for processing. The output from the final PE should be inversely transformed to represent the real final value. This stage is handled by the *Winograd Inverse Transformer.* For example, in our design, the output always has a constant length of 8, in the x-axis dimension. For the convolutions of sizes 3 and 7, which are the typical convolution dimension, the size of the output after the inverse transformation is 6 and 2. We need to mention that we do not apply any transformations on the weights since they are already preprocessed and transformed into the Winograd representation on the host for any number of inferences.

(De)Serializer. Our design utilizes a specific *Deserializer* and *Serializer* to receive/send data from the previous or to the next FPGA. For deserialization, the *Deserializer* receives the size of the incoming data from the controller and then starts receiving data in *long4* format, which is an array of four *long* values. Further, it stores the received data in a specific global memory buffer. Similarly, for serialization, the *Serializer* receives the size of the data to be sent and the location of the data from the controller. Afterward, it starts sending the data in that buffer to the next FPGA, again in *long4* data format.

xBar interconnect. Our design adopts the traditional *xBar* in DLA [6]. *xBar* is a custom interconnect to customize, connect, and configure the layers to the design. This component allows for adding more layers and achieves higher acceleration. Examples of these layers are ReLU, Pool, Norm, and Sigmoid.

3.2 3D CNN Accelerator Architecture

3D CNNs are composed of 3D convolution layers, which have higher computational intensity compared to the 2D convolutions. This higher intensity is due to the existence of an extra dimension (usually frames) in the input feature map, the intermediate feature map between the layers, and the weight filters. 3D CNNs rely on this extra dimension to learn the motion information across multiple frames of a video. Toward this goal, 3D convolution weights slide over the frame dimension on the input feature map, similar to the weights sliding over the width and height to generate the convolved output. The semi-1D CNN accelerator can be extended to support 3D convolutions. Unfortunately, mapping the 3D convolutions onto the semi-1D systolic array is not trivial and can lead to poor performance or low utilization of the FPGA resources. There are two main challenges in mapping the 3D convolutions onto the systolic array. First, the choice of order between processing the frames or the output channels can lead to different data and weight reuse, which can directly affect the performance. Second, the large weight size of the 3D convolutions may not fit inside the limited local buffer of the PEs. In the following, we discuss our solutions to address these two problems.

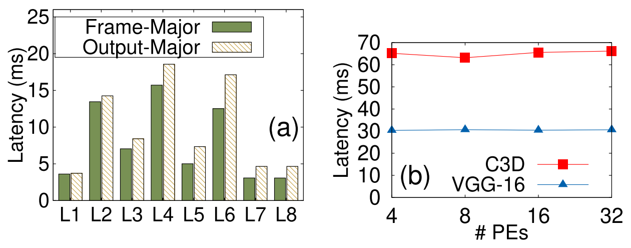


Fig. 6. (a) Latency of the layers of the C3D model, with *Frame-Major* and *Output-Major* approaches. (b) Performance of C3D and VGG-16, with different numbers of PEs. It is already included in the graph.

Proper order between dimensions. Choosing the right processing order between the frame dimension and the output channel dimension can affect the frequency of global memory accesses and the data reuse rate, and is thus critical for achieving the fastest implementation. There are two approaches for mapping the computation of the 3D convolution onto the systolic array. In the first approach, the *weight provider* uploads the weights isnto the PEs for a certain number of output channels (32). Further, the feature provider can slide over the frames of the input data and stream the data to the PE array. At the end of this round, we have all output frames for that specific number of output channels (32). The accelerator performs this process for "*OUT_CH_SIZE / NUM_PE*" rounds, where *NUM_PE* equals to 32, to get all relevant data for a set of output channels. In this approach, each round the accelerator generates all output frames for a set of output channels, which we call the *Output-Major* approach. In the second approach, the feature provider caches a frame of data in the local buffer and streams it into the PE array. The *weight provider* iteratively loads the weights into the PEs. Further, PEs convolve the input data with the weights and generate the outputs. For the next round, the feature provider slides one frame over the input data and starts over the same process. In this approach, each round the accelerator generates all output features for one output frame, which we call the *Frame-Major* approach.

Figure 6(a) represents the latency of the layers of a sample 3D CNN model (C3D) while experimenting with the preceding two approaches. In comparison, the frame-major approach outperforms the output-major, up to 1.3 times faster for specific layers. As a result, our design adopts the frame-major approach for 3D convolutions.

Large number of weight parameters. Unlike 2D convolutions, 3D convolutions have a large number of weight parameters, due to the existence of the extra frame dimension. As a result, the weight size might exceed the available local buffer in the PEs. There are three approaches to addressing this problem. First, we can split the weight features into *NUM_CHILDLAYER* child layers, where each child layer handles a portion of the weights, which can entirely fit in the PE's local buffer. The same approach is also adopted by Liu et al. [24]. The weights can be split from either the input-channel or the frame dimension, where both lead to the same performance. The outputs of all child layers have to be accumulated to get the final output of the convolution for the whole weight features. This accumulation is performed inside the *Feature Writer*, which writes back the data to the global memory (Figure 2). Second, we can trade the number of PEs with the total amount of memory available in each PE. By reducing the number of PEs, we can allocate a larger buffer for each PE, which can store the whole weight parameters. Third, we can use a combination of the preceding two approaches. Considering the *weight splitting* and *fewer PEs* as the two ends of the spectrum, it is possible to have an architecture that sits somewhere in between. For example, we can attempt to reduce the total number of splits while maintaining a minimum number of PEs.

We evaluated the preceding approaches by experimenting with various designs with different numbers of PEs and buffer size. We synthetically reduced the total size of the local memory on the

FPGA to make the PE buffer small for the weight parameters. Figure 6(b) represents the performance of the VGG-16 and C3D models while running on the FPGA with different configurations (number of PEs). On the two ends of the spectrum, we have the 32 PE configuration (maximum number of PEs) and the 4 PE configuration (enough buffer to hold all weight parameters). We also explored architectures with 8 and 16 PEs, which represent the trade-off between the number of PEs and the available buffer on each PE. We need to mention that reducing the number of PEs to increase the buffer size does not mean the allocation of less DSPs. In this configuration, we can assign more DSPs to each PE by increasing the value of *VEC_SIZE* in our design (each PE contains $VEC\_SIZE \times W\_VEC$ number of DSPs). As a result, we can maintain the same level of parallelization.

Experimental results show almost no difference between various configurations. As a result, we decided to stick with the default configuration (32 PEs, with smaller VEC_SIZE value), which requires weight splitting for layers with a large number of weight parameters. Increasing the value of VEC_SIZE can have adverse effects for layers with a small number of channels. In these cases, VEC_SIZE goes beyond the size of the channel, which leads to having some of the DSPs going idle, or doing dummy computations. As a result, choosing a relatively small VEC_SIZE value can make the design generic enough for all types of layers with different configurations, irrespective of their size.

3.3 Multi-FPGA Support

Our framework can split a single CNN design onto multiple FPGAs, connected through Intel-supported serial channels. Each serial channel utilizes a 40-Gbps Infiniband link. The FPGAs are connected in a daisy-chained fashion. The first port of each FPGA is connected to the second port of the previous FPGA in the chain, receiving the data. The second port of each FPGA is connected to the first port of the next FPGA in the chain, sending the data. The first port of the first FPGA and the second port of the last FPGA do not have any cables connected. We need to mention that the first FPGA can be connected to a streaming device (camera, cloud, etc.), and directly receives the data for processing.

The assumption for achieving linear speedup is the perfect load balance of the FPGAs while running the layers. In another words, each FPGA should handle an equal computation load of the whole execution. An imperfect load balance leads to one or more FPGAs handling a larger share of computation compared to the other FPGAs. Those FPGAs will become the bottleneck in the pipeline, and this stops the design to achieve perfect linear speedup. As a result, proper load balance is critical to enable optimal throughput improvement.

Figure 7 depicts a sample CNN deployed on a series of FPGAs in a row. In this example, we broke large layers into multiple smaller sub-layers. We also grouped multiple small layers, where the granularity of the group is equal to the other layers or sub-layers. This approach enables our design to balance the computation overhead among all FPGAs. During the execution, each FPGA receives the input or the intermediate data from the source or the previous FPGA, pushes the data through the assigned (sub)layers, and sends the output to the next FPGA or the sink. On each FPGA, the sub-layers or the layers are processed sequentially. After processing one input and forwarding the output, each FPGA grabs the next input. In this setup, all FPGAs operate in parallel to co-locate the execution of different layers of different inputs and increase the overall throughput. In other words, let us consider a network with an execution time of t on a single FPGA. With a configuration of two FPGAs, each FPGA can handle half of the network. In this configuration, the second FPGA can execute the second half of the network for an input, whereas the first FPGA can start processing the first half of the network for the next input. As a result, we can obtain twice the throughput compared to a single FPGA configuration.

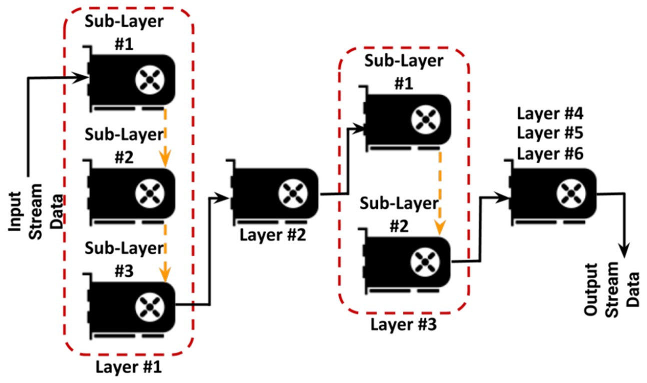


Fig. 7. Mapping a neural network onto a cluster of FPGAs.

As mentioned earlier, our framework breaks large layers into multiple sub-layers. Mapping original layers onto the FPGAs can lead to unbalanced computational overhead throughout the pipeline, and further prevents the framework from achieving linear speedup, while increasing the number of FPGAs. Our framework tackles this problem by splitting a layer from *output* channel into multiple sub-layers with 32 output channels. Any number below 32 is not going to have any benefit since we have 32 PEs that need to finish execution. In addition, it breaks a layer over the *frame* channel, where the number of the frames of the input is higher than the number of frames of the weight. Further splitting forces the design to pad the frame dimension, which introduces unnecessary extra overhead. After the splitting process, each sub-layer can run individually, with no dependency on any other sub-layer.

We need a proper model of our system to find the best mapping of sub-layers onto the FPGAs. We consider our multi-FPGA system as a pipeline of M stages, where M is the number of FPGAs. The framework splits the CNN network into M partitions (each partition is composed of layers and sub-layers) and distributes them sequentially on the chain of FPGAs. We define the *Latency (L)* of the multi-FPGA system, $L_{Multi-FPGA}$, as the time interval, after which the multi-FPGA system can accept the next input. The throughput of our multi-FPGA system is equal to $1/L_{Multi-FPGA}$, which is the total number of inputs it can process per unit of time. As mentioned before, our multi-FPGA system is a pipeline, where each FPGA is called a *stage*. Each stage handles part of the neural network. In a pipeline, the speed of the whole system is bounded by the speed of the slowest stage (highest latency). Thus, the partition with the highest latency stalls the whole pipeline and is called the *bottleneck stage*. The latency of the whole pipeline is equal to the latency of the bottleneck stage.

The latency (L) of each layer i can be calculated as follows:

$$\begin{aligned} L_i = {} & (WEIGHT\_SIZE_i \ / \ DDR\_bandwidth) \\ & +(Total\_multiplications_i \ / \ (Frequency \\ & \times VEC\_SIZE \times W\_VEC \times Num\_PEs)). \end{aligned} \tag{7}$$

In Equation (7), the latency of each (sub)layer is specified as the time it takes to read the weights into the PEs, plus the total time it takes to finish the overall calculations. We need to mention that we omit the overhead of the network communication since FPGAs are directly connected to each other through 40-Gbps QSFP+ Infiniband cables, which makes the communication latency ($OUTPUT\_SIZE/Network\_bandwidth$) negligible. For example, for the C3D model, the maximum communication latency between two FPGAs is less than 0.1 ms.

Next, we need to find the appropriate mapping of the (sub)layers onto the FPGAs in our multi-FPGA setting. Assuming we have M FPGAs, connected in a daisy-chain fashion, we aim to map a CNN composed of N (sub)layers onto these FPGAs in a linear fashion. The most appropriate mapping should lead to the highest possible throughput. The overall throughput in this architecture is typically bounded by the FPGA with the highest latency for handling the assigned layers. We can formulate this problem as balancing the load across the FPGAs in the chain.

To find the most appropriate mapping, we first calculate the processing time (latency) and latency of each (sub)layer using Equation (7). Using these latencies, we can design an algorithm to perform a brute-force enumeration to find the best mapping. For the brute-force enumeration, the algorithm requires to evaluate $\binom{N}{M-1}$ configurations. The time complexity of this method is $O(N^{min(M,N-M)})$, which is exponential to the number of (sub)layers N. We also developed this algorithm in C++ to calculate the best mapping using a different number of layers and FPGAs. For example, a configuration of 100 layers and 10 FPGAs takes around 960 hours to finish, which is quite expensive.

To address the preceding problem, we developed a polynomial-time load-balancing algorithm using dynamic programming. Equation (8) presents the overall solution for the optimal mapping of the layers:

$$L_{j,k} = \begin{cases} \sum_{l=1}^{j} L_l & \text{if } k = 1 \\ min_{r=1}^{j-1}(max(L_{r,k-1}, \sum_{l=r+1}^{j} L_l)) & \text{if } k > 1 \text{ and } k \leq M. \end{cases} \tag{8}$$

In Equation (8) $L_{j,k}$ represents the latency of the first k FPGAs while handling the (sub)layers from 1 to j. For the case with only one FPGA, the latency for accepting new inputs is equal to the latency of handling all (sub)layers, from 1 to j, on a single FPGA. For the cases with multiple FPGAs, the latency is calculated based one assigning the last few (sub)layers (from $r + 1$ to j) in the set to the last FPGA. The latency of the final FPGA equals to the total latency of the layers. Further, the rest of the first (sub)layers (from 1 to r) are mapped to the other FPGAs, which is the sub-problem in our dynamic programming algorithm. Finally, the latency of the whole pipeline is the latency of the bottleneck stage. The time complexity of this method is $O(M^2 \times N)$, which is linear to the number of (sub)layers. Finally, we can obtain the overall throughput of the system by calculating the latency of the bottleneck stage.

4 EXPERIMENTAL RESULTS

To confirm and quantify the benefits of our new framework, we implemented and evaluated *VGG-16*, *AlexNet*, *ResNet*, *C3D*, and *I3D*. *VGG-16* is widely used for image classification. *ResNet* is a well-known CNN model for object recognition. *C3D* and *I3D* are both video analytics 3D CNNs. We evaluate the performance of our framework in comparison to the available single-FPGA and multi-FPGA solutions. All experiments were conducted on a set of Intel Fog Reference Design units [26], each equipped with two Nallatech p385a FPGA acceleration cards. Each host has one Intel Xeon CPU E5-1275, with 32 GB of main memory. Each FPGA card has an Intel Arria 10 FPGA, with 8 GB of DDR3 SDRAM. FPGAs are serially connected through QSFP+ 40-Gb/s InfiniBand cables. The OpenCL kernels were compiled using Intel FPGA SDK for OpenCL (version 19.1) with Nallatech *p385a_sch_ax115* board support packages (BSP). We performed GPU experiments on two Nvidia GPUs: (1) Nvidia RTX 2080Ti, which is a server class GPU, and (2) Nvidia Tesla T4, which is a small-scale GPU for edge systems. In addition, we performed our CPU experiments on a host, equipped with two Intel Xeon Silver 4210 octa-core CPUs, and 64 GB of main memory. We report the latency of our design, which is the time it takes for the design to accept a new input request, following the previous request. In addition, we report the throughput as the number of images per second (img/s) that the system can accept. The results include the energy efficiency of our experiments. We used *nvidia-smi* command line utility and the Nallatech mempoy-mapped device layer API to query instant board-level power consumption. For the CPU, we used the *power-stat* toolkit to measure the power consumption. Finally, we performed post-training quantization with Pytorch [27] on the VGG-16 model to enable it running with a lower bit precision (8 bit) on the CPU.

In the single-FPGA experiments, we evaluated the performance of our design for accelerating different CNN models. It achieves state-of-the-art performance on a single FPGA, which is crucial for any further extensions. In the multi-FPGA experiments, we accelerated the same CNN models but with more than one FPGA. We designed the experiment to show the correlation between the number of FPGAs and throughput.

4.1 Single-FPGA Performance Evaluation

In this section, we present the effectiveness and performance of our design on a single FPGA. Having a state-of-the-art single-FPGA solution is the basis for extending that solution to multiple FPGAs. Table 4 presents the performance, energy efficiency, and resource utilization of our design and the related works.[1] Our solution can provide 27 ms of latency for single input inference on VGG-16, which is faster than all other related works [2, 20, 22, 23]. For ResNet-50, which is a widely used object detection model, our solution can achieve 21 ms of latency per single input, which is 6 ms faster than the solution of Ma et al. [28].

DLA [6] can achieve an average of 1-ms latency for each image while processing a batch of images. This low latency is achieved by reading the weight and input data from the local memory, which eliminates the overhead of accessing global memory. We replicated the DLA experiment with VGG-16 and achieved 37 ms of latency per single image. By offloading the FC layers onto a dedicated PE, we can achieve 27 ms of latency, which is 10 ms faster than our clone of the DLA.

Our approach achieves similar or better energy efficiency than the related works, except the AlexNet on the Intel DLA. In this specific example, the design avoids reading/writing from/to the external memory (the AlexNet model can fit inside the on-chip memory) and applies inference on a batch of data. External memory exclusion significantly reduces the power consumption, and the utilization of a batch instead of a single input increases the MM operation's performance. However, our design targets streaming applications, requiring real-time service of each input.

We also demonstrate the performance of our framework for accelerating CNN models with 3D convolutions. Table 5 presents the performance of some standard video processing CNNs from our FPGA solution and the performance reported by the related works. Using our design for 3D convolutions, we can achieve almost 1.7 times better latency compared to these related works. This lower latency enables processing of a higher number of frames per second, for latency-critical applications.

Finally, Table 6 presents the performance and energy-efficiency of our design compared to the state-of-the-art CPU and GPU implementations. The CPU shows a high latency while handling a single input for inference, which makes it inferior to the accelerators. It also results in the lowest energy efficiency. The GPU can outperform the FPGA, but the energy efficiency of the FPGA is much better. We need to mention that the FPGA can operate individually, whereas the GPU requires a host, which adds to the overall power consumption. In addition, FPGAs can ingest data directly into the pipeline, which is useful to many stream data processing scenarios such as edge computing [18], whereas GPUs require the data to be first stored in the host memory and then copied to the GPU memory.

[1]Because the design details and source code are often not available, comparing to the performance numbers reported in the related works is the standard practice in the FPGA community.

Table 4. Performance Comparison of State-of-the-Art Single-FPGA Implementations

Design	Zhang et al. [21]	Ma et al. [23]	Ma et al. [28]	Suda et al. [22]	Zhang et al. [2]		Wang et al. [20]	Aydonat et al. [6]		Ours			
CNN Model	Alex Net	VGG -16	ResNet -5	VGG -16	VGG -16		VGG -16	Alex Net	VGG -16	Alex Net	ResNet -50	VGG-16	
FPGA	Virtex 480t	Arria 10	Arria 10	Stratix V	KU060		Arria 10	Arria 10	Arria 10	Arria 10			
Clock Frq. (MHz)	100	200	150	120	200		190	303	215	212			
Precision (bits)	32	16	16	16	16	8	8	8	8	8		8	16
Latency/Image (ms)	43.23	43.2	27.2	117.8	101.15	25.3	225	1	37	8.8	20.9	26.52	30.3
Throughput (GOPS)	61.62	715.9	285.07	262.9	266	1.17K	N/A	1,300	N/A	990		990	866.25
Throughput (Img/s)	23.13	23.14	36.7	8.48	9.88	39.52	13.45	1,000	27	113.6	47.8	37.7	32.9
Power (watt)	18.61	N/A	N/A	N/A	25		27.3	45		23			
Energy Efficiency (GOPS/watt)	3.31	N/A	N/A	N/A	10.64	46.8	N/A	28.88	22	43.04		43.04	37.66
Energy Efficiency (Image/J)	1.24	N/A	N/A	N/A	0.39	1.58	0.49	22.2	0.6	4.93	2.07	1.63	1.44
DSP Util. (Used/Total)	2.2K/2.8K	1.5K/1.5K	1K/1.5K	0.8K/1.9K	1K/2.7K	0.1K/2.7K	0.5K/1.5K	1.5K/1.5K	1.5K/1.5K	1.4K/1.5K		1.4K/1.5K	1.2K/1.5K
BRAM Util (Used/Total)	1K/2K	1.5K/2.7K	2.1K/2.7K	1.6K/2.5K	0.7K/2.1K	0.7K/2.1K	N/A	2.4K/2.7K	1K/2.7K	1.3K/2.7K		1.3K/2.7K	1.7K/2.7K
LUT Util. (Used/Total)	186K/303K	N/A	N/A	N/A	100K/320K	200K/320K	N/A	N/A	278K/854K	290K/854K		290K/854K	420K/854K
FF Util. (Used/Total)	205K/607K	N/A	N/A	N/A	80K/727K	140K/700K	N/A	N/A	725K/1708K	762K/1708K		762K/1708K	933K/1708K

Note: Each column represents one of the related works, including ours. Each row represents some of the configurations of the experiments, and the performance and resource utilization of the related works.

Table 5. Performance Comparison of Single-FPGA Video Processing CNN Acceleration

Design	Shen et al. [15]	Liu et al. [24]	Ours
CNN Model	C3D	C3D	C3D
FPGA	VC709	VC709	Arria 10
Clock Frq. (MHz)	N/A	120	210
Precision	N/A	N/A	8 bit
Latency (ms)	89.4	115.5	66.08
Throughput (GOPS)	427.5	667.7	990
Throughput (Input/s)	11.18	8.65	15.13
Power (watt)	25	25	23
Energy Efficiency (GOPS/watt)	17.1	26.7	43.04
Energy Efficiency (Input/J)	0.44	0.34	0.65
DSP Util.	1.5K/3.6K	3.5K/3.6K	1.5K/1.5K
BRAM Util	1.5K/2.9K	0.3K/1.4K	1.3K/2.7K
LUT Util.	242K/432K	272K/432K	290K/854K
FF Util.	286K/866K	434K/866K	762K/1708K

Note: Columns and rows are the same as in Table 4.

Table 6. Performance Comparison Between CPU, GPU, and Our FPGA Implementation, Running the VGG-16 Model

Accelerator	RTX 2080Ti	Tesla T4	Xeon CPU		Ours	
Clock Frq. (MHz)	1,545	1,087	2200		212	
Precision (bits)	32	32	32	8	8	16
Latency/Image (ms)	8.43	13.14	128.39	58.35	26.52	30.3
Throughput (Img/s)	118.6	76.10	7.78	17.13	37.7	32.98
Power (watt)	250	70	86		23	
Energy Efficiency (Image/J)	0.47	1.08	0.09	0.19	1.6	1.4

4.2 Multi-FPGA Performance Evaluation

In this section, we evaluate the performance of our framework while scaling out the CNN deployment on multiple FPGAs. We performed all experiments on a pipeline of FPGAs (from one to eight) connected through I/O channels. We evaluated the performance of both 2D and 3D CNNs on our multi-FPGA platform, using VGG-16, C3D, and I3D. Figure 8 presents the performance improvements while accelerating the three CNN models mentioned previously, using multiple FPGAs. For all three models, the throughput increases linearly by increasing the number of FPGAs from one to eight. We can observe that using the load balancer in Section 3.3 can lead to a near-perfect partitioning scheme with a fully balanced workload across the FPGA.

We also made a comparison between our solution and the other available multi-FPGA related works [1, 3]. In comparison, our solution provides up to 5.8 times better throughput. Jiang et al. [3] experimented using 16-bit variables, which has higher precision than our configuration (8 bit). They can achieve similar performance to ours by utilizing a lower-precision configuration, but they are using a better FPGA (much more resources compared to Arria 10). Their solution cannot scale linearly beyond four FPGAs for the VGG-16 model. In addition, our design supports a diverse set of convolutional operations, and it is based on the high-level OpenCL language. Finally, our solution achieves much higher energy efficiency. We need to mention that the power consumption of the

Table 7. Performance Comparison of Multi-FPGA Acceleration Solutions

Design	Zhang et al. [1]		Jiang et al. [3]				Ours			
CNN Model	VGG-16		VGG-16				VGG-16			
FPGA	VC709		ZCU102				Arria 10			
Clock Frq. (MHz)	150		200				210			
Precision	16 bit		16 bit				8 bit			
Num. FPGAs	1	2	1	2	4	8	1	2	4	8
Throughput (Input/s)	6.5	13	14	35.28	74.2	84	37.7	67.8	150.8	232
Throughput (GOPS)	203.9	407.8	N/A	N/A	N/A	N/A	990	1,980	3,960	7,920
Power (watt)	25	50	27.2	54.4	108.8	217.6	23	46	92	184
Energy Efficiency (GOPS/watt)	8.16		N/A				43			
Energy Efficiency (Input/J)	0.26		N/A				1.63			

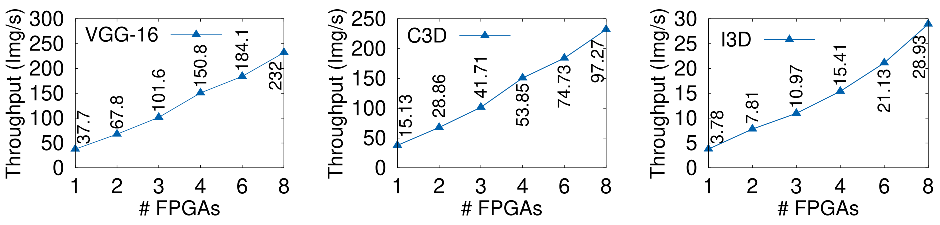


Fig. 8. Acceleration of 2D and 3D CNN models using multiple FPGAs.

related work [3] for four and eight FPGAs was estimated since the work provided experiments for only two FPGAs and simulated the results of more than two FPGAs.

5 RELATED WORKS

Several works have studied the use of FPGAs for accelerating 2D CNNs. Caffeine [2] explored using 2D systolic arrays for accelerating convolutions and MMs. It also studied the feasibility of mapping MM onto the convolution systolic array in both *input-major* and *weight-major* formats. Finally, it provided the support of the Caffe framework for the automatic deployment of the model onto the FPGA. Compared to our design, it provides lower performance and adopts a different approach for mapping MMs that is suitable for only processing for a batch of input requests. PipeCNN [20] is a generic OpenCL-based implementation for accelerating the 2D CNNs. It is the only related solution with the source code available online. Similar to our work, PipeCNN utilizes multiple components for fetching the data, computation, and writing back the data to the memory. All components are connected using Intel channels. Unlike other works, PipeCNN adopts a simple implementation for the convolution and MM operations, which leads to significant performance overhead. In addition, it cannot fully utilize the resources on an FPGA. Zhang et al. [21] observed that the accelerator design space had not been well exploited, and the computation throughput of a design may not well match the available memory bandwidth. As a result, they proposed an analytical design scheme using the roofline model. Using this model, they explored various design optimizations such as loop tiling and transformations to identify the best performance and lowest FPGA resource requirements. Similar to the previous work, Suda et al. [22] proposed an end-to-end large-scale CNN accelerator on FPGA. They also performed a design space exploration to maximize throughput. Compared to our work, none of these related works have studied the multi-FPGA acceleration of CNNs. In addition, they lack the support for the emerging 3D convolutions and operate slower compared to our solution. Most similar to our work, DLA [6] is the state-of-the-art design with the highest performance available. DLA adopted a 1D systolic array design architecture. In contrast, we proposed a novel semi-1D systolic array, which consumes a lower amount of resources compared to the original 1D systolic array. Our design supports more complex convolutions (3D) and multi-FPGA acceleration. As well, we improved the overall performance by introducing an MM accelerator alongside the systolic array.

Several other related works have studied the acceleration of 3D CNNs on FPGAs. Shen et al. [15] and Liu et al. [24] studied the uniform acceleration of 2D and 3D CNNs on FPGAs. Further, they demonstrated a uniform analytical model to facilitate efficient design space explorations of 2D and 3D CNNs. Morph [10] provided a specific design for accelerating 3D CNNs. This work mainly discusses various approaches for tiling and reordering the loops for a better acceleration of the 3D convolution computation. Compared to these works, our design can achieve higher performance by efficiently mapping 3D convolutions onto the systolic array and enables the multi-FPGA acceleration of the CNNs.

Multi-FPGA CNN acceleration is an underexplored topic, with only a few related works available. Zhang et al. [1] demonstrated a multi-FPGA acceleration solution to increase the throughput of the CNN applications. In this work, FPGAs are connected through serial channels, where each FPGA receives one piece of data, processes it, and then sends the data to the next FPGA. However, the FPGA cluster is managed by an external FPGA, which makes it hard for integration in heterogeneous systems with various types of processors. In contrast, we use a typical host system for the management of the FPGAs. In addition, our solution provides much higher throughput. Jiang et al. [3] proposed a multi-FPGA solution for reducing the latency of the inference operations rather than the overall throughput. It can achieve a low-latency inference through spatially parallelizing CNN layers on multiple FPGAs. All experiments in this work are limited to two FPGAs. The latency speedup of the configurations with more than two FPGAs is mathematically modeled. This work is not scalable to more FPGAs due to the limited number of ports on the FPGAs. They can alleviate this issue by using network switches and integrating network communication and control stack on the FPGAs, which would, however, affect their core low-latency data transmission by introducing an extra overhead and diminish the overall latency improvement. Differently, our design adopts a low-level data transmission protocol (direct serial channel communication between the FPGAs), which unlocks the theoretical bandwidth between the communication channels. Complementary to this solution, our work parallelizes the CNN models temporally, which helps increase the overall throughput of the system, as demonstrated in our experimental results, and also scale beyond two FPGAs. Microsoft Brainwave [29] is another notable multi-FPGA solution in the cloud. Brainwave relies on pruning and quantization of the model to fully store the input and model in on-chip storage and achieve the optimal performance, which is suitable for RNNs with small intermediate data but not for CNNs with large intermediate data. As a result, it is suitable for only a particular set of neural networks that can fully fit in on-chip BRAM and benefit from the preceding compression techniques.

Although our work utilizes the basic techniques of the early works, it introduces novel contributions. First, our work is the first work solely designed and developed using a high-level heterogeneous programming language (OpenCL) and achieves state-of-the-art performance. Second, to the best of our knowledge, it is the first to propose a semi-1D systolic array for further area efficiency of the design on the chip. Third, our design supports multi-FPGA acceleration of the complex neural networks such as 3D CNNs, with a novel and efficient algorithm for balanced distribution of the layers, onto the FPGAs.

6 CONCLUSION

In this work, we demonstrated a high-throughput multi-FPGA acceleration solution for a wide variety of CNNs. We first extended the DLA [6] architecture to achieve lower latency and better resource utilization for single-FPGA configuration. Second, we extended the architecture to support 3D convolutions for the video understanding applications. It includes studying the optimal deployment of the 3D convolutions on the semi-1D systolic array. Third, we enabled communication between the FPGAs to support multi-FPGA setups. Finally, we developed an algorithm for automatic deployment of the CNN layers onto the FPGAs in the multi-FPGA setup. Our results show that utilizing multiple FPGAs can linearly increase the overall throughput of the CNN inference. In addition, optimizing the PE systolic array and FC operations can lead to better area utilization (up to 25%) and higher overall throughput (up to 24%) compared to the state-of-the-art CNN accelerator. Finally, we studied the efficient mapping of 3D convolutions on our novel semi-1D systolic array to achieve the highest overall throughput.

7 FUTURE WORK

In future work, we plan to extend our platform in several ways. First, we plan to support multi-FPGA parallelism, where the FPGAs are connected through a network switch. This model enables easier scalability of the FPGA clusters in data centers and server racks. Implementation of this model requires the addition of the network connection and control layers onto the FPGAs. It also requires eliminating the extra resource consumption and the latency overhead to guarantee efficient scalability while using multiple FPGAs. Second, we plan to extend our PE design to support sparse operations. Recent advancement in model compression has made quantization (bit-width reduction) and sparsification getting more adopted in the model deployment process. Our design already supports different bit-precision configuration. Supporting sparse operations enables integration with available post-training quantization and sparsification methods. In case of the multi-FPGA support for sparsity, our design can still divide the layers onto multiple FPGAs and increase the throughput linearly by increasing the number of FPGAs, although it may require a different approach to estimate the computational intensity of the layers. Finally, we will extend our system to support heterogeneous acceleration. In many CNNs, particular layers with specific configurations may run faster on a GPU or a CPU. We will extend our framework to support these accelerators in a pipeline fashion and enable faster inference time. Our multi-FPGA accelerator is open source and publicly available (https://github.com/saman-aghazadeh/GNU-DLA).

REFERENCES

[1] Chen Zhang, Di Wu, Jiayu Sun, Guangyu Sun, Guojie Luo, and Jason Cong. 2016. Energy-efficient CNN implementation on a deeply pipelined FPGA cluster. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design.* ACM, New York, NY, 326–331.

[2] Chen Zhang, Guangyu Sun, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2018. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 11 (2018), 2072–2085.

[3] Weiwen Jiang, Edwin H.-M. Sha, Xinyi Zhang, Lei Yang, Qingfeng Zhuge, Yiyu Shi, and Jingtong Hu. 2019. Achieving super-linear speedup across multi-FPGA for real-time DNN inference. *ACM Transactions on Embedded Computing Systems* 18, 5s (2019), 1–23.

[4] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16).* 265–283.

[5] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia.* ACM, New York, NY, 675–678.

[6] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C. Ling, and Gordon R. Chiu. 2017. An OpenCL deep learning accelerator on Arria 10. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* ACM, New York, 55–64.

[7] Geert Litjens, Thijs Kooi, Babak Ehteshami Bejnordi, Arnaud Arindra Adiyoso Setio, Francesco Ciompi, Mohsen Ghafoorian, Jeroen Awm Van Der Laak, et al. 2017. A survey on deep learning in medical image analysis. *Medical Image Analysis* 42 (2017), 60–88.

[8] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. 2017. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine* 34, 6 (2017), 26–38.

[9] Daniel Maturana and Sebastian Scherer. 2015. VoxNet: A 3D convolutional neural network for real-time object recognition. In *Proceedings of the 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '15).* IEEE, Los Alamitos, CA, 922–928.

[10] Kartik Hegde, Rohit Agrawal, Yulun Yao, and Christopher W. Fletcher. 2018. Morph: Flexible acceleration for 3D CNN-based video understanding. In *Proceedings of the 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'18).* IEEE, Los Alamitos, CA, 933–946.

[11] Du Tran, Lubomir Bourdev, Rob Fergus, Lorenzo Torresani, and Manohar Paluri. 2015. Learning spatiotemporal features with 3D convolutional networks. In *Proceedings of the IEEE International Conference on Computer Vision.* 4489–4497.

[12] Lin Sun, Kui Jia, Dit-Yan Yeung, and Bertram E. Shi. 2015. Human action recognition using factorized spatio-temporal convolutional networks. In *Proceedings of the IEEE International Conference on Computer Vision.* 4597–4605.

[13] Shuiwang Ji, Wei Xu, Ming Yang, and Kai Yu. 2012. 3D convolutional neural networks for human action recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35, 1 (2012), 221–231.

[14] Andrew Lavin and Scott Gray. 2016. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 4013–4021.

[15] Junzhong Shen, You Huang, Zelong Wang, Yuran Qiao, Mei Wen, and Chunyuan Zhang. 2018. Towards a uniform template-based architecture for accelerating 2D and 3D CNNs on FPGA. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* ACM, New York, NY, 97–106.

[16] Shmuel Winograd. 1980. On multiplication of polynomials modulo a polynomial. *SIAM Journal on Computing* 9, 2 (1980), 225–229.

[17] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA'17).* IEEE, 1–12.

[18] Saman Biookaghazadeh, Ming Zhao, and Fengbo Ren. 2018. Are FPGAs suitable for edge computing? In *Proceedings of the USENIX Workshop on Hot Topics in Edge Computing (HotEdge'18).*

[19] Intel. n.d. *Intel FPGA SDK for Open CL Programming Guide.* Intel.

[20] Dong Wang, Ke Xu, and Diankun Jiang. 2017. PipeCNN: An OpenCL-based open-source FPGA accelerator for convolution neural networks. In *Proceedings of the 2017 International Conference on Field Programmable Technology (ICFPT'17).* IEEE, Los Alamitos, CA, 279–282.

[21] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* ACM, New York, NY, 161–170.

[22] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-Sun Seo, and Yu Cao. 2016. Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* ACM, New York, NY, 16–25.

[23] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-Sun Seo. 2018. Optimizing the convolution operation to accelerate deep neural networks on FPGA. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26, 7 (2018), 1354–1367.

[24] Zhiqiang Liu, Paul Chow, Jinwei Xu, Jingfei Jiang, Yong Dou, and Jie Zhou. 2019. A uniform architecture design for accelerating 2D and 3D CNNS on FPGAs. *Electronics* 8, 1 (2019), 65.

[25] Andrew Boutros, Sadegh Yazdanshenas, and Vaughn Betz. 2018. You cannot improve what you do not measure: FPGA vs. ASIC efficiency gaps for convolutional neural network inference. *ACM Transactions on Reconfigurable Technology and Systems* 11, 3 (2018), 20.

[26] Intel. n.d. Fog Reference Unit. Retrieved February 22, 2021 from https://www.intel.com/content/www/us/en/internet-of-things/fog-reference-design-overview.html

[27] Pytorch. n.d. Home Page. Retrieved February 22, 2021 from https://pytorch.org

[28] Yufei Ma, Minkyu Kim, Yu Cao, Sarma Vrudhula, and Jae-Sun Seo. 2017. End-to-end scalable FPGA accelerator for deep residual networks. In *Proceedings of the 2017 IEEE International Symposium on Circuits and Systems (ISCAS'17).* IEEE, Los Alamitos, CA, 1–4.

[29] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, et al. 2018. A configurable cloud-scale DNN processor for real-time AI. In *Proceedings of the 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18).* IEEE, Los Alamitos, CA, 1–14.

Received May 2020; revised September 2020; accepted October 2020