Hardware Acceleration of Video Analytics on FPGA Using OpenCL

by

Akshay Dua

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved September 2019 by the
Graduate Supervisory Committee:

Fengbo Ren, Chair
Umit Ogras
Jae-sun Seo

ARIZONA STATE UNIVERSITY

December 2019

ABSTRACT

With the exponential growth in video content over the period of the last few years, analysis of videos is becoming more crucial for many applications such as self-driving cars, healthcare, and traffic management. Most of these video analysis application uses deep learning algorithms such as convolution neural networks (CNN) because of their high accuracy in object detection. Thus enhancing the performance of CNN models become crucial for video analysis. CNN models are computationally-expensive operations and often require high-end graphics processing units (GPUs) for acceleration. However, for real-time applications in an energy-thermal constrained environment such as traffic management, GPUs are less preferred because of their high power consumption, limited energy efficiency. They are challenging to fit in a small place.

To enable real-time video analytics in emerging large scale Internet of things (IoT) applications, the computation must happen at the network edge (near the cameras) in a distributed fashion. Thus, edge computing must be adopted. Recent studies have shown that field-programmable gate arrays (FPGAs) are highly suitable for edge computing due to their architecture adaptiveness, high computational throughput for streaming processing, and high energy efficiency.

This thesis presents a generic OpenCL-defined CNN accelerator architecture optimized for FPGA-based real-time video analytics on edge. The proposed CNN OpenCL kernel adopts a highly pipelined and parallelized 1-D systolic array architecture, which explores both spatial and temporal parallelism for energy efficiency CNN acceleration on FPGAs. The large fan-in and fan-out of computational units to the memory interface are identified as the limiting factor in existing designs that causes scalability issues, and solutions are proposed to resolve the issue with compiler automation. The proposed CNN kernel is highly scalable and parameterized by three architecture parameters, namely pe_num, reuse_fac, and vec_fac, which can

be adapted to achieve 100% utilization of the coarse-grained computation resources (e.g., DSP blocks) for a given FPGA. The proposed CNN kernel is generic and can be used to accelerate a wide range of CNN models without recompiling the FPGA kernel hardware. The performance of Alexnet, Resnet-50, Retinanet, and Light-weight Retinanet has been measured by the proposed CNN kernel on Intel Arria 10 GX1150 FPGA. The measurement result shows that the proposed CNN kernel, when mapped with 100% utilization of computation resources, can achieve a latency of 11ms, 84ms, 1614.9ms, and 990.34ms for Alexnet, Resnet-50, Retinanet, and Light-weight Retinanet respectively when the input feature maps and weights are represented using 32-bit floating-point data type.

ACKNOWLEDGMENTS

I would like to express my gratitude to me thesis Advisor Dr. Fengbo Ren for his patience and continuous guidance throughout the year that helped me shape my thesis work. Weekly meeting with him helped me keeping my work on track. I learned a lot from him not only in terms of technical aspects but also how to organize my work in a better way that is always going to help me even in my professional career.

I would also like to thank my lab mate Michael Riera for always motivating me and having all discussions that pushed my work to completion.

This work is supported by an unrestricted gift from RadiusAI,Inc. I would like to thank Dr. Aykut Dengi and Dr. Bobby Chowdary from RadiusAI, Inc for their technical suggestions and fruitful discussion on this work.

Finally, I would like to thank my family members, my mom, dad and my brother without their emotional support I would never have been able to complete this work.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Over the past few years, there have been many groundbreaking advancements in the field of deep learning. Numerous applications from different domains such as autonomous, medical, finance have adopted deep learning algorithms. Deep learning algorithms are even beating humans in different activities, as has been shown by the Alphabets deep mind alphago,[1] which defeated Go game grandmaster. Google translator, based on a recurrent neural network [2], translating multiple languages at high accuracy, is one of the success stories of the deep learning algorithms. However, the increase in accuracy comes at the cost of the increase in computation complexity and memory requirement. When Alexnet [3] was introduced in 2012, it requires 0.7 GFLOPS to perform image classification, which increased to 11 GFLOPS for Resnet-152 [4], which is an increase in computational complexity by almost 15 times to improve the accuracy of classifying an image by 12%.

Deep learning algorithms implementation can be divided into two phases. One is training, and the other is inference. For a given deep learning model, training of the model is typically done offline to define weights and is a one-time effort. So, training time is not a concern for real-time applications. However, the inference is where the application uses the deep learning models for the classification of real data; hence, the inference run time becomes a crucial parameter in determining the performance of the deep learning model.

The critical performance metrics to measure the performance of a real-time application, which uses the deep-learning algorithm as its backbone, includes latency and energy efficiency. Latency and energy efficiency depends on the model and hard-

ware selected to map deep learning algorithms. Graphics processing units (GPUs) having thousands of computing units, high memory bandwidth (GB/s), and running at a high frequency (in GHz) delivers high throughput when processing a batch of multiple images in parallel. GPU hardware is optimized to provide high-throughput for batch processing. However, for real-time applications that require the processing of streaming data at low latency and high energy efficiency, GPUs are not the ideal hardware [5].

Some research work has focused on developing custom application-specific integrated circuits (ASICs) based accelerator [6] for the deep learning algorithms to achieve high energy efficiency and low latency. However, ASICs are not flexible enough to keep up with the changes in deep learning algorithms. Also, the long design verification and fabrication time add to the disadvantages of using ASICs as the accelerator for deep learning algorithms. Recent studies [7], [8] have shown Field Programmable Gate Array (FPGAs) offering lower power consumption (therefore higher energy efficiency) as compared to GPUs for accelerating many applications at the cost of lower throughput. For example, [8] has shown Intel Arria 10 GX1150 FPGA achieving 4 times higher energy efficiency as compared to Tesla K40C GPU while the computing performance of FPGA is 0.3 times of GPU. FPGAs also offers more hardware flexibility and reduces the time to develop accelerator as compared to ASICs. So, higher energy efficiency and flexibility motivated us to design a convolution neural network (CNN) accelerator using FPGAs for this thesis.

1.1 Related Work

FPGAs offers high hardware flexibility and energy efficiency that have attracted many research teams to use FPGAs for accelerating CNN inference. [9] shows the computing power of FPGAs by achieving a throughput of over 1.3 TFLOPS. The

conventional method of programming FPGAs relies on designing at Register-transfer level (RTL), which allows fine-tuning of resource utilization to improve the performance of computing units and energy efficiency, as has been shown by [10]. But verification of design requires verifying functionality and timing issues, that adds to the design development cycle. High-level synthesis tools provided by FPGAs vendors such as Intel FPGA SDK for OpenCL enables faster verification of the design and hence decreases the design development cycle. [9], [11] designs have shown achieving high performance by programming FPGA using OpenCL.

Several proposed FPGA frameworks offer limited design flexibility [12] [13]. For example, [12] is designed for accelerating the YOLO CNN model [14], so the resource utilization is highly optimized for a particular CNN model, but the same design cannot be used for accelerating different CNN models. This reduces the flexibility of the design and increases the design effort to re-analyze and accelerate other CNN models. Many existing FPGA CNN accelerators fail to fully utilize all the available computing units [15], [16], which reveals scalability issues that exist in the design, which leads to reduced performance. To address the issue of scaling up the design, [17] discusses the advantages of adopting systolic array architecture [18]. [11] adopts 2-d systolic array architecture to achieve high frequency, scale up the design to maximize the utilization of the computing units, and achieve high performance. However, [11] fails to achieve 100% of computing unit utilization due to the high fan-in and fan-out issues that they fail to resolve completely.

1.2 Contributions

In this thesis, we mainly focused on the accelerating inference phase CNNs. After analyzing the shortcomings of the current work, as discussed in Section 1.1, we propose solutions to achieve higher performance. Key contributions of our CNN accelerator

design are encapsulated in the following.

- We optimize the throughput of the CNN models by implementing a highly pipelined 1-D systolic array architecture for convolution.
- We parameterize OpenCL design to enable design space exploration. This would enable the design to scale up and maximize the utilization of computing power.
- We optimize the external memory bandwidth utilization by reducing the frequency of external memory access. To reduce the external memory access, we implement loop tiling.
- We provide solutions for resolving high fan-in and fan-out issues to increase the scalability of the design.
- We generalized our CNN accelerator design to make it compatible with different CNN architectures. This makes our design more user-friendly and reduces the hardware design effort.

1.3 Thesis Organization

This thesis is divided into multiple chapters to explain various aspects of the CNN accelerator design along with the experimental setup, performance results and comparison with the state-of-the-art CNN accelerator designs.

- Chapter 2 gives essential background information about Deep learning. It describes different CNN layers and architecture that we used for our experiments. This chapter also discusses OpenCL based FPGA computing.
- Chapter 3 explains various features of our design and proposes solutions to the existing problem that we implement in our CNN accelerator design.

- Chapter 4 describes the experimental setup, with a description of hardware and software tools that we used to perform experiments.

- Chapter 5 provides an example of design space exploration based on the architectural parameters that we defined for our CNN accelerator design.

- Chapter 6 is where we present our result for four different CNN models, namely Alexnet and Resnet-50, Retinanet and Light-weight retinanet, to show the flexibility of our CNN accelerator design with variety of CNN models. We also compare our results with the state-of-the-art CNN acclerator designs.

- Chapter 7 is the conclusion of our thesis.

Chapter 2

BACKGROUND

In this chapter, we give a brief background of Deep-learning. Then we move on to give basic information about the CNNs, which is the core operation of video-analytics. After that brief introduction of OpenCL, a framework that we use for programming FPGA.

2.1 Introduction to Deep Learning

Machine learning is a science that enables computers to work with real-data without any explicit instructions. Machine learning-based applications learn from real experiences and generate a statistical model that helps computers to classify random input data received from the real world. The generation of a statistical model is called the training phase. After the training phase is completed, which can take multiple days to finish, this machine learning model is used for classifications. This phase of classification is called the inference phase. For example, the machine learning model trained for image-classification application is trained using a data-set of thousands of images. In the inference phase, this model classifies images.

Machine learning is an umbrella term of which deep learning [19] is a part. The artificial neural networks with multiple hidden layers generate a deep neural network. An artificial Neural network imitates the brain by having many neurons that are connected to form an acyclic graph that maps the input features to the output features in a non-linear way. A Neural Network has multiple layers known as hidden layers between input and output mapping that consists of multiple neurons. Each of these hidden layers is responsible for extracting different features of the input data and

sending processed data to the next layer. After the features of the object get extracted by the hidden layers, such as the edges of an image, classifiers are used to identify image class.

In the training phase of a neural network, each neuron is assigned weights and bias through backward propagation functions such as gradient computing technique or stochastic gradient descent (SGD) [20]. For the inference phase, results propagate in the forward direction to the next layer only before they are sent to the classifier, which performs either binary classification or regression. The following are the advantages of using a deep neural network.

- **High accuracy:** Existing deep neural networks are achieving high accuracy. With network presented by [21], surpassing humans in face verification, deep neural networks are the ideal solutions to many applications such as image classification.

- **Knowledge transfer:** Once the deep neural network is trained, it is used by different applications working in the same domain for which that deep neural network is trained. This is what we have exploited in our work by using the open-source pre-trained deep neural network from the caffe2 framework [22].

2.2 CNN Architecture

The CNNs are the subcategory of deep learning, and designing an accelerator for CNNs is the primary focus of this thesis work. Image classification based applications mostly use CNNS. CNNs re-utilizes the weight across the input feature map to extract the spatial dependencies between different pixel values of the input feature map. In this thesis, we present the performance of the four CNN architecture, namely Alexnet [3], Resnet-50 [4], Retinanet [23], and Light-weight retinanet [24] using propose CNN

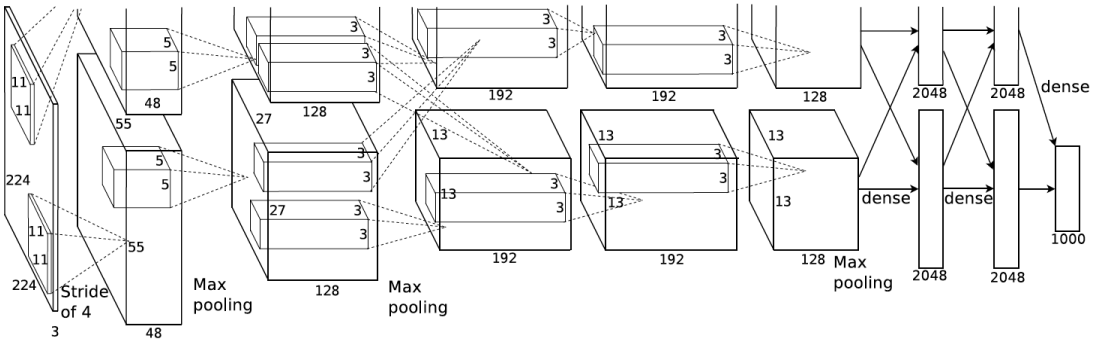

Figure 2.1: Architecture of Alexnet

accelerator design to demonstrate the compatibility of our design with a wide variety of CNN architectures.

Alexnet [3], introduced in 2012 by the research team of the University of Toronto, won the 2012 ImageNet LSVRC-2012 competition by achieving the maximum reduction in error rate by bringing error rate to 15.3%. The Alexnet CNN model consists of five convolution layers, two local response normalization layers (LRN), two max pooling, and three fully connected layers. This model contains over 62.4 million parameters and requires 1.1 billion computations for performing the inference on the image of 227 $\times$ 227 $\times$ 3. Figure 2.1 represents the architecture of Alexnet.

Resnet [4], introduced in 2015 by the Microsoft research team, won the ILSVRC 2015 competition with an error rate reduced to 3.5%. Multiple residual neural networks were proposed by the Microsoft research team, namely Resnet-18, Resnet-34, Resnet-50, Resnet-101, and resnet-152. The Resnet-50 architecture trained for 224 $\times$ 224 $\times$ 3 image size originally has around 25 million of parameters with forty-nine convolutions, one fully connected layer, one average pooling, and one max pooling. The significant advantage of using deep residual networks is the solution to the problem of degraded neural network performance when there is an increase in the number

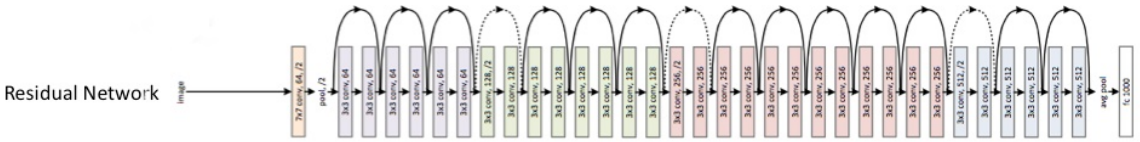

Figure 2.2: Architecture of Resnet

of layers. Figure 2.2 represents the architecture of Resnet-32.

Retinanet [23], developed by the Facebook research team, is the CNN architecture used by video-analytics based applications for object detection. Retinanet model's inference phase is divided into three parts. The first part acts as the backbone that proposes the regions for object detection, followed by the feature pyramid network (FPN) [25] that generates four feature maps by merging the top layers with the bottom ones. The output of the FPN is used to generate bounding box regression and classify objects in the image. Retinanet performs 156 GFLOPS on the image size of $800 \times 800 \times 3$ for detecting objects in a given image. Figure 2.3 represents the architecture of Retinanet.

Light-weight retinanet [24], proposed by the parallel system and computing (PSC) lab of Arizona state university, is the modified version of the Retinanet [23] CNN model. The underlying architecture of Light-weight Retinanet is similar to the original Retinanet and can be referred from Figure 2.3. The idea behind Light-weight Retinanet is to reduce the number of FLOPS by identifying the most computationally expensive layer and reducing filter size given there is no drop in the accuracy. This model reduces the number of FLOPS by 1.8 times as compared to the original Retinanet CNN architecture.

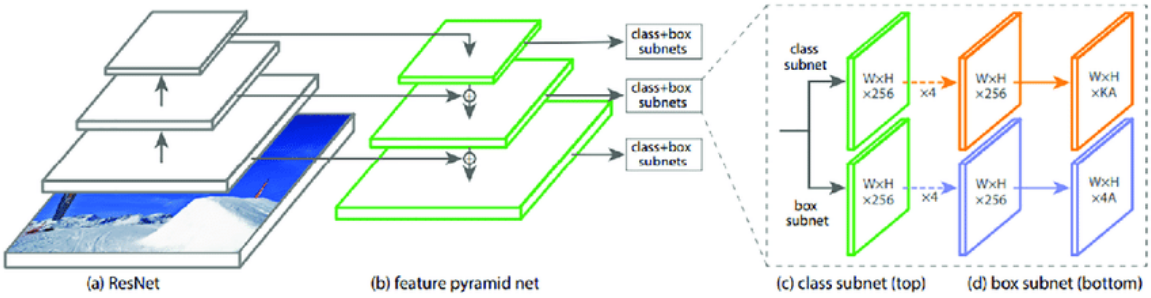

Figure 2.3: Architecture of Retinanet

2.3 CNN Layers

In the previous section, we discussed the CNN models that we use to measure the performance of our CNN accelerator design. To ensure that our CNN accelerator design is compatible with different CNN models, along with convolution we have added support of multiple CNN layers namely, batch normalization (BNORM), LRN, max pooling, average pooling, element-wise (ELTWISE), fully connected (FC), and rectified linear unit (RELU) layer for the inference phase of CNN models.

Convolution: The core operation of CNN based model is convolution. The number of layers in CNN models are increasing as they become dense, which increases the percentage of the convolution computations. The Alexnet CNN model, with five convolution layers, has 90% of its computations as convolution that increased in the Resnet-50 CNN model to 99%. As a result, accelerating convolution becomes crucial for the CNN accelerator, as has been evident from [26]. The convolution is a 3-D multiplication and accumulation of the input feature maps and weights and is known to be the computationally expensive layer. Equation 2.1 represents the formula for convolution,

$$output(oc, y, x) = \sum_{n=0}^{ic} \sum_{n1=0}^{K} \sum_{n2=0}^{K} inp(n, y + n1, x + n1) \times w(oc, n, n1, n2) \qquad (2.1)$$

where the output (oc,x,y) is the output generated at x row, y column at oc channel,

which is calculated after multiplying and accumulating the weights of K× K size with the input feature map at x row and y column. Another factor to consider with convolution is stride, which determines the shift of the window of the weight along the rows and columns of the input feature map.

LRN: The LRN layer [3], used by earlier CNN models such as Alexnet, to normalize the input feature maps. Normalization also depends on the neighboring elements of the input feature map also. It generates the output feature map of the same dimension as the input feature maps with the normalized result written at the same coordinate position as the corresponding input feature map. The LRN layer uses Equation 2.2 to normalize the input feature maps.

$$output(i,y,x) = input(i,y,x)/(k + \alpha \times \sum_{j=max(0,i-n/2)}^{min(N-1,i+n/2)} (input(j,y,x)^2))^{\beta} \quad (2.2)$$

Parameters n,α,β, and k are trainable parameters, and are fixed for inference phase. The parameter N is the input channel dimension that varies layer to layer.

Pooling: The pooling layers are used by CNN models for down-sampling the input feature maps, such that the dimension of each feature map reduces by the factor, which is determined by the pooling filter dimension (Kp × Kp), and stride factor. The two types of pooling, which are generally used by CNN models and supported by our design, are 1) average-pooling, and 2) max pooling. Equation 2.3 and 2.4 shows the functionality of average and max pooling, respectively.

$$output(i,y,x) = \sum_{n=0}^{n=K_p} \sum_{n1=0}^{n1=K_p} (input(i,y+n,x+n1)/(K_p \times K_p) \quad (2.3)$$

$$output(i,y,x) = max(input(i,y+K_p,x+K_p) \quad (2.4)$$

BNORM: BNORM [27] is a more commonly used layer than LRN for normalization. It replaced the LRN layer mainly because of the higher learning rate. For example, in Resnet-50, every convolution layer is followed by the BNORM layer. BNORM output,

similar to LRN, is of the same dimension as the input feature map dimension. In LRN, the normalized output depends on the neighboring elements of the input. However, in the BNORM layer, the output at position (y,x) only depends on the input at the position (y,x). Equation 2.5 represents the formula for BNORM,

$$output(i, y, x) = \gamma((input(i, y, x) - \mu)/\sqrt{\sigma^2}) + \beta \tag{2.5}$$

where μ is running mean, and σ^2 is running variance. Parameters γ and β are trained parameters, and not changed in the inference phase. Thus, the BNORM layer can be fused with the previous convolution layer, as suggested by [28]. In our design, we have fused BNORM layer with the convolution layer.

ELTWISE: ELTWISE layers merge different branching layers, which is becoming common in many CNN models. It is evident from different variations of Resnet CNN models. ELTWISE performs an element-wise sum operation between two merging branches, and generates the output, which is of the same dimension as the inputs.

FC: For most CNN models, the FC layer is the last layer, which is used for classification of the image. FC layers generate a one-dimension result which connects all the input feature maps by multiplying the input feature maps with the weights and accumulating the results. In FC layers, there is no opportunity to re-utilize the weights because of which FC layers are memory intensive, and the available external board memory bandwidth limits their performances. Equation 2.6 shows the formula for calculating FC layer output,

$$output(i) = \sum_{n=0}^{n=N_if} w(i, n) \times inp(n) \tag{2.6}$$

where N_if is the dimension of the input feature map. The Alexnet CNN model contains three FC layers as compared to Resnet-50, which has one FC layer. In our design, we use same hardware for both convolution and FC layers.

Activation: The activation functions are the non-linear functions such as tanh, sigmoid. The function supported by our CNN accelerator design is RELU. As Equation 2.7 represents,

$$output(i) = (input(i) > 0)?input(i) : 0 \tag{2.7}$$

RELU generates the results of same dimension as the input.

2.4 OpenCL-based FPGA Computing

The OpenCL framework developed by the Khronos group is an open-source platform that enables parallel programming across heterogeneous hardware such as FPGAs, GPUs, and CPUs. OpenCL codes are portable across different hardware. OpenCL code consists of the host and device code, where the host code, written in C/C++, runs on the host processor, which communicates accelerator via PCIe. The host code is compiled by the standard C/C++ compiler, such as g++/gcc. The device code, also known as the kernel code, written in OpenCL C, is mapped to hardware by converting code to VHDL and synthesizing the generated VHDL code. Placement and routing along with static timing analysis are performed by vendor-specific OpenCL based HLS tools on the synthesized code, and the binary file is generated that programs the target hardware. The device code is compiled at run-time, where data which accelerator uses to do computation is known only at run-time and is send by host code via PCIe. For our thesis, we are using Intel OpenCL SDK for FPGA to program Intel ARRIA 10 GX1150 FPGA. Intel OpenCL offers following two ways of programming hardware.

First is, **NDRange programming**, where the thread-level parallelism is explored. But with this, available hardware resources such as registers are not utilized efficiently, and no loops are pipelined. The performance for this mode mainly depends

on the number of thread FPGA can execute in parallel.

Second is **Single-thread Programming**, where compiler pipelines loops to improve the throughput [29]. With Single-thread programming, the compiler can map some of the buffers to the available register or shift-register that optimizes memory bandwidth utilization.

Additional features of Intel OpenCL SDK for FPGA that we have exploited in our design implementation are:

- **Memory channels**, which are the first-in-first-out (FIFO) on-chip registers that transfer the data between different kernels. This pipelines data movement and hence can be used to increase temporal parallelism.

- **Autorun kernels** are the part of the device code that has no interface with the host code. So they are generally used for computation, which receives data, processes it, and distributes via memory channels. Autorun kernels are always running kernels and are not invoked from the host side. This reduces the latency of invoking the kernel from the host side and hence improves the performance. [8] shows that using autorun kernel also enables the compiler to optimize pipeline better and increases the frequency of the design.

Chapter 3

ARCHITECTURE DESIGN

This chapter provides an in-depth description of the architecture design of our CNN accelerator along with the optimization techniques for different layers of CNN, namely convolution, LRN, FC, ELTWISE, and pooling to accelerate the performance of CNNs. Figure 3.1 represents the proposed system architecture, along with the data flow. The input feature maps are read from the external memory and stored to an on-chip shift-register-based buffer, which we further describe in Section 3.2.6. The

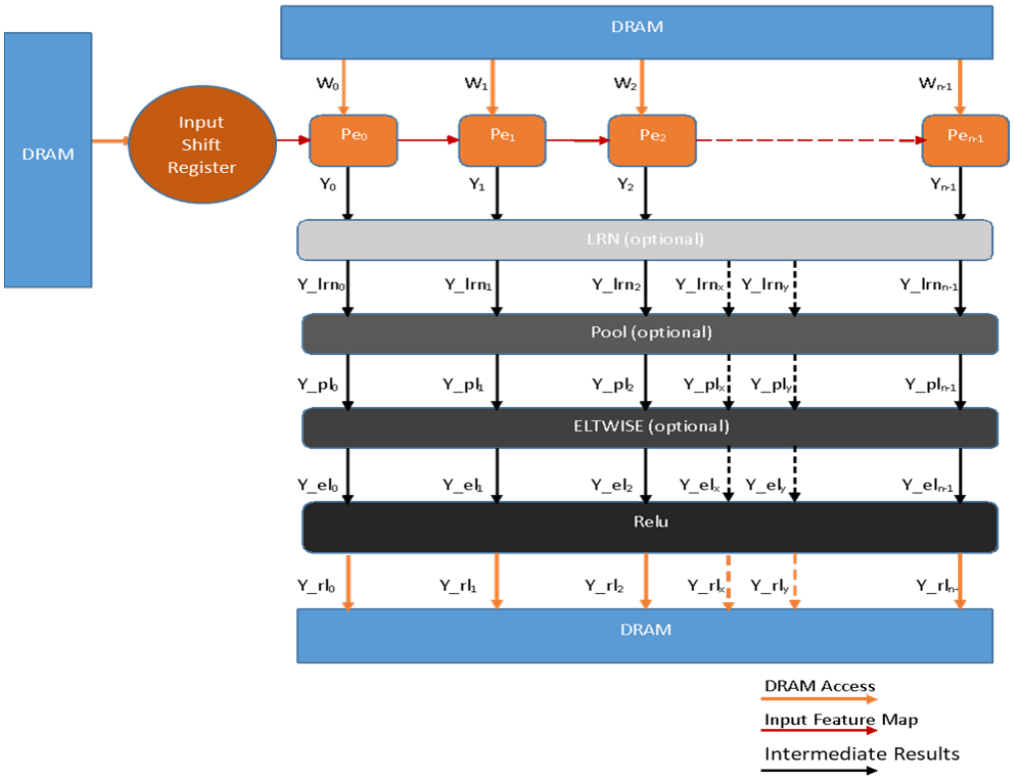

Figure 3.1: Proposed System Architecture

weights are also read from the external memory and are streamed to the processing elements (PEs), along with the input feature maps to perform convolutions. We adopt a 1-D systolic array architecture for accelerating convolution layers, which we discuss in Section 3.1. Depending on the subsequent layer of convolution, the convolution results are sent to either the LRN, pooling, ELTWISE or RELU modules. Similarly, for the following convolution layer, the input feature maps are read from the external memory along with the new weights, and the data flows in a similar direction as the first convolution. We parameterize the architecture design of our accelerator with three architectural parameters, namely pe_num, vec_fac, and reuse_fac. Pe_num determines the number of PEs in the 1-D systolic array that performs convolution in a deep pipeline, as shown in Figure 3.1. Reuse_fac determines the number of times convolution kernels reuses the same input feature map data loaded into the shift-register-based buffer. A high reuse_fac reduces the need for external memory access and improves bandwidth efficiency. Vec_fac defines the SIMD width of MAC units in each PE. These three parameters allow users to efficiently perform the design space exploration to fully utilize the available hardware resources on an FPGA. An example of the design space exploration based on the Intel Arria 10 development kit is discussed in Chapter 5.

Each module of the proposed system architecture is generalized to make our accelerator compatible with a variety of CNN models. Taking advantage of the high generality, we tested the proposed CNN accelerator with four different CNN models of various sizes, including Alexnet, Resnet-50, Retinanet, Light-weight Retinanet. We discuss measure performances in Chapter 6.

```
Loop1 : for i = 0 to op_dim                                          /// op_dim is output feature map dimension
Loop2 :   for j = 0 to ic_dim                                        /// ic_dim is input feature map dimension
Loop3 :     for y = 0 to col_dim                                     /// col_dim is column of input feature
Loop4 :       for x = 0 to row_dim                                   /// row_dim is row of input feature
Loop5 :         for m1 = 0 to K                                      /// K is filter size
Loop6 :           for m2 = 0 to K                                    /// K is filter size
                    output[i][y][x] += ip[j][y+m2][x+m1] * w[i][j][m1][m2]   /// ip is the input feature and w is the filter
```

Figure 3.2: C Code of Convolution

3.1 Convolution Architecture

The convolution, defined in Equation 2.1, is the most computationally expensive layer, and it accounts for more than 90% of the total computations of a CNN. Thus, optimizing the convolution computations is the key to accelerate the performance of the CNNs. Figure 3.2 represents the C code of convolution. A convolution result is obtained after six nested loops are executed. The basic principles of accelerating convolution are to 1) parallelize the execution of data-independent loops with spatial parallelism in hardware, and 2) pipeline the execution of data-dependent loops with temporal parallelism in hardware. Loop1, loop3, and loop4 of Figure 3.2 have no loop-carried data dependency. Thus they can be parallelized in execution. The other loops of Figure 3.2 have loop-carried data dependency. They can be partially parallelized due to the associative property of addition and multiplication and then further pipelined to resolve all the data dependency. For our design, we apply these principles and unroll loop1, loop2, and loop4 to accelerate the convolution performance.

However, the following challenges remain and must be resolved for accelerating the CNNs on the system level:

- FPGA acceleration cards have **limited on-chip memory. They suffer from long access latency** for transferring the weights and input feature maps between the external memory and on-chip buffers, which significantly limits the system-level performance of CNN acceleration. The size of the weights and in-

put feature maps are often large in CNNs, and the on-chip memories in FPGA are usually not large enough to store. For example, the weight size for the second convolution layer of Alexnet is 4 MB, making it impossible to store. This is why both the weights and input feature maps must be stored in the external memory. However, this results in transferring a large amount of data from the external memory onto the on-chip memory. As a result, the long external memory access latency becomes the performance bottleneck of the system. The solution to this challenge is to optimize the memory accesses by loop tiling [16] and using shift-register-based buffers to store the input feature maps, which promises to reduce the amount of external memory access and improve the external memory bandwidth efficiency. We discuss the memory access optimization scheme in Section 3.2.6.

- **Scalability of the accelerator**: Digital signal processing (DSP) blocks are embedded in FPGAs to perform fast multiplication and accumulation. So, the number of DSP blocks represents the computing power of a given FPGA. Fully utilizing these blocks at high efficiency is one of the targets of our accelerator design. It is evident from [29], [15] that achieving 100% utilization of the available DSP blocks is challenging. Reasons that prevent scaling up of the accelerator include the following:

  - **Long interconnect routes** results as DSP blocks are spread across the FPGA; in addition, routing of the LUTs requires additional interconnect resources, which increases the routing congestion, and hence it is difficult for the compiler to route the design. This leads to a longer critical path, which decreases the frequency of the accelerator. One of the solutions is a 1-D systolic array architecture for convolution. We discuss the 1-D systolic

array architecture in Section 3.1.1.

- **The large size of multiplexer** makes it difficult for the compiler to fit the multiplexer. For example, parallelizing loop2 or loop5 of Figure 3.2 generates a different number of output results in parallel, which determines the size of the output multiplexer. One of the solutions is to reorganize the reading of the input feature maps and to parallelize loop2 to reduce the size of the output multiplexer. We further discuss the solution in Section 3.2.7.
- **High fan-in and fan-out** make it difficult for the compiler to complete the placement and routing stage. We propose the solution to this issue Section 3.2.8.

3.1.1 1-D Systolic Array Architecture for Convolution

For convolution, we implemented the idea of highly pipelined 1-D systolic array architecture [30], which can be seen in Figure 3.3. The accelerator consists of an array of PEs, with each PE receiving different sets of the weights, and shifting the input feature maps to the next PE, with the first PE receiving the input feature maps from the input buffer. Each PE consists of multiple MAC units that perform convolution and sends out the result. Partial sum (PS) stays inside each PE and is used for accumulation with the result generated by the MAC units. This minimizes the movement of PS. The result at the position (oc, y, x) of Equation 2.1 is ready to be used by the other layers after loop2, loop5, and loop6 of Figure 3.2 are finished. Each PE generates independent results, which are either directly transferred to the external memory or used by the other layers such as ELTWISE. Pe_num determines the number of PEs that exists in the accelerator and hence, the depth of the pipeline, which is shifting the input feature maps. For each PE, reuse_fac determines the

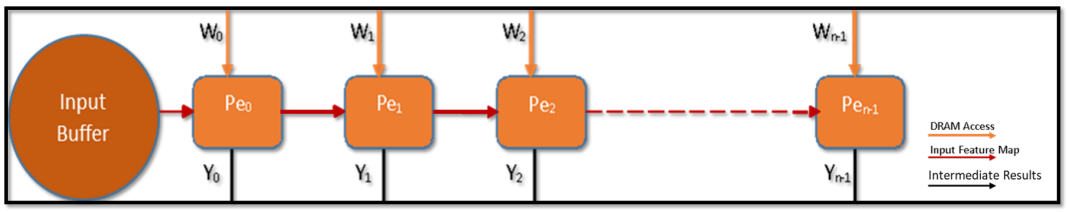

Figure 3.3: Systolic Array Architecture

number of independent results generated by each PE in parallel. So the total number of outputs generated in parallel is given by Equation 3.1.

$$number\_output = pe\_num \times reuse\_fac \tag{3.1}$$

Equation 3.1 also determines the size of the output multiplexer, which collects the output from multiple PEs.

This architecture addresses the scalability challenge of the CNN accelerator that we discussed in Section 3.1 because of the following key features:

- **Short and local interconnects**: 1-D systolic array architecture results in short and local interconnects between different PEs. With short and local interconnects, routing congestion is reduced along with the critical path, which increases the frequency of the design.

- **Elimination of multiplexer**: The input feature maps are shifted across from one PE to another PE, along with the weights, which are routed to a fixed PEs. For example, W_0 always routes to PE_0, so no multiplexer is needed for shifting the input feature maps and the weights to different PEs, thus eliminating the need for a multiplexer.

- **Reduced input feature map fan-out**: Each PE shifts the input feature maps to the next PE only. Thus, for shifting the input feature maps, PE_n is driven

only by PE_{n-1}. As a result, 1-D systolic array architecture reduces the fan-out and hence increases the scalability of the CNN accelerator.

- **Reducing the external memory access**: As the same input feature maps are shifted across different PEs, architecture brings down the external memory access for the same input feature maps by pe_num times. Thus relaxing the board memory bandwidth and reducing the external memory access.

- **Concurrent execution**: As each PE receives both shifted input feature maps and weights, each PE generates results after the same number of cycles and hence increases the concurrent execution of convolution by pe_num factor.

Overall, the proposed 1-d systolic array architecture for convolution improves both the scalability and operating frequency, thus improving the performance of the CNN accelerator.

3.1.2 PE Architecture

Figure 3.4 shows PE architecture, which is replicated based on the pe_num parameter. It consists of a highly pipelined MAC tree, which operates on the weights and input feature maps received per cycle. The stored PS is accumulated per cycle, with the result generated by the pipelined MAC tree.

The number of DSP blocks utilized by each PE is determined by the vec_fac and reuse_fac. Each PE explores spatial parallelism by generating reuse_fac number of output in parallel. Equation 3.2 expresses the number of DSP blocks utilized per PE.

$$DSP\_units = vec\_fac \times reuse\_fac + reuse\_fac \tag{3.2}$$

The first term accounts for the MAC units of Figure 3.4, and the second term is for the last stage adder unit to accumulate the PS.

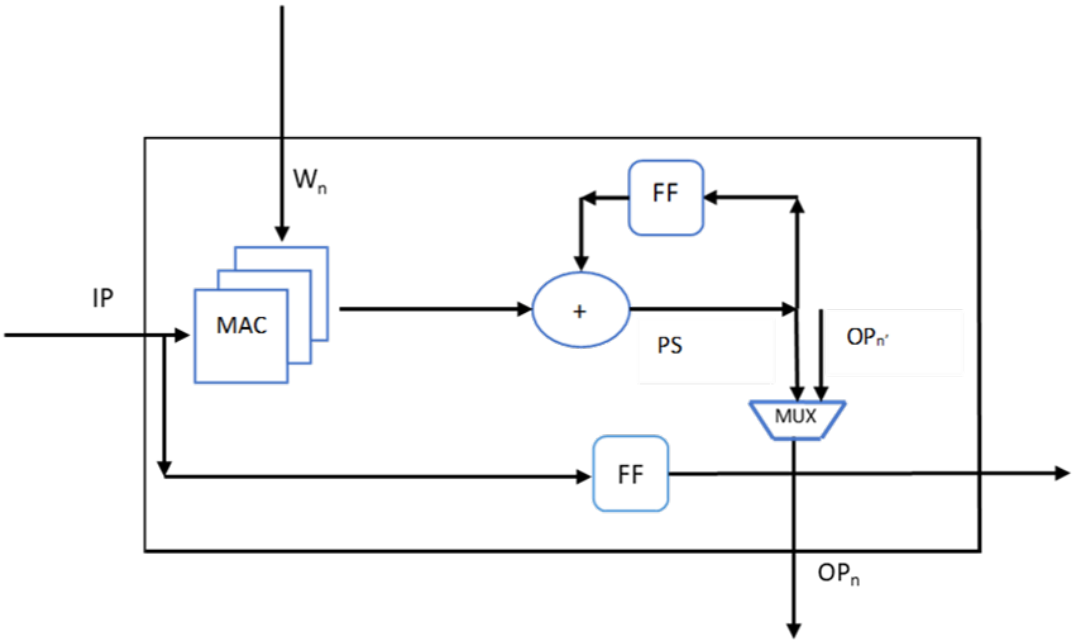

Figure 3.4: PE Architecture, with MAC Units. Ip and W_n Are Shifted Input Data and Weights to PE, Respectively, Flip-flop (FF) Updates the PS per Cycle, Multiplexer (Mux) Sends out Results to the next Layer

3.2 OpenCL Kernel Design

We use the Intel OpenCL SDK for FPGA to implement our accelerator. Along with the convolution, other layers, namely LRN, max pooling, RELU, and ELTWISE, are separated into different kernels. For layers, namely FC and average pooling, the same convolution kernel is reused to increase the hardware efficiency.

3.2.1 Convolution Kernel Design

Our OpenCL kernel design consists of multiple kernels that communicate through the Intel's OpenCL memory channels (discussed in Section 2.4) to implement the proposed 1-D systolic array architecture for the convolution. Figure 3.5 represents the

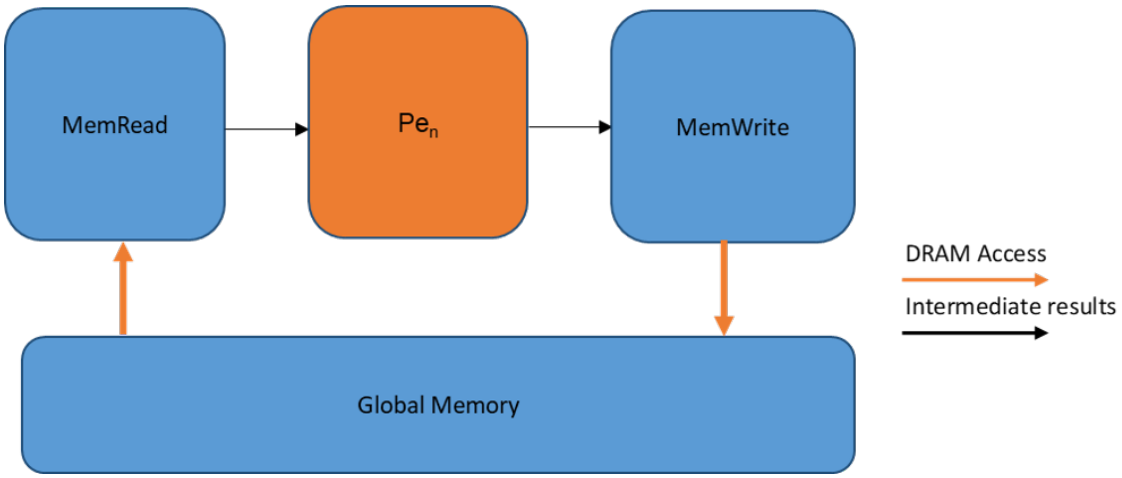

Figure 3.5: Kernel Design, with One PE, Single-Thread Memread and Memwrite Kernel Transferring Data to and from the External Memory Respectively.

convolution kernel design with one PE kernel for computation. Two single-threaded kernels, namely, memread and memwrite, are responsible for transferring the data from and to the external memory, respectively. The memread kernel loads the input feature maps and the weights and shifts them to the PE kernel through memory channels. The shift-register-based on-chip buffer is implemented in the memread kernel to store the input feature maps. The PE kernel is an autorun kernel (discussed in Section 2.4), which receives the data, performs MAC operations, and streams out the result to the memwrite kernel, where the results are transferred back to the external memory. Pe_num parameter defines the size of the PE kernel array, which sends out the results to the memwrite kernel in parallel.

The key advantages of adopting this kernel architecture are :

- **Deep pipelined processing**: Multiple kernels communicating with each other through memory channels increases the level of temporal (pipeline) parallelism.
- **Optimal initiation interval (one cycle)**: Separating the convolution operation into multiple cascaded kernels allows the Intel FPGA OpenCL SDK

compiler to resolve all the serial dependencies that exist in the design. As a result, the compiler efficiently pipelines the design with an initiation interval of one clock cycle.

- **High utilization of the hardware resources**: Other layers, namely pooling, ELTWISE, and RELU, also use memread and memwrite kernels, thus increasing the efficiency of hardware resources utilization.

The first two advantages of this kernel architecture are the key to improving the throughput of the CNN accelerator.

3.2.2 LRN Kernel Design

The LRN layer normalizes the input feature maps and is defined by Equation 2.2. Out of all the CNN models that we use for our experiments, only Alexnet uses this layer. LRN layers require an exponential function, which consumes a large amount of FPGA resources. This limits the resource utilization of the convolution layer, which is more crucial to accelerate the performance. To reduce the additional utilization of the resources, we implement exponential operation using a piece-wise linear approximation function. LRN could also be reformulated as Equation 3.3, where f(x) represents the exponential term of Equation 2.2. To achieve the required accuracy, we defined fifty points for the piece-wise linear approximation of the exponential function.

$$out(x) = in(x) * f(x) \tag{3.3}$$

The LRN kernel is implemented in a single-threaded OpenCL kernel mode, with vec_fac determining the SIMD width for the LRN layer. The number of outputs generated in parallel is equal to the vec_fac parameter.

3.2.3 Pooling Kernel Design

Average and max pooling are the two types of pooling supported by our design. For max pooling, defined by Equation 2.4, we have added a single-threaded kernel that receives the input feature maps from a memread kernel through the memory channel and performs a comparison to find the maximum value and write back the result to the external memory. The vec_fac parameter determines the SIMD width for max pooling.

For average pooling, defined by Equation 2.3, where K_p $\times$ K_p is the pooling filter size, our CNN accelerator uses the same kernel design as the convolution kernel. The average pooling output is generated by accumulating the input feature maps and multiplying the accumulated result with $\frac{1}{K\_p \times K\_p}$. So, by defining the weights as $\frac{1}{K\_p \times K\_p}$ and using the distributive property of addition and multiplication, we are able to use the convolution kernels for average pooling, which further increases the hardware utilization.

3.2.4 FC Kernel Design

The FC layer flattens the input feature map to one dimension and is a special case of the convolution with the filter size being the input feature map dimension. With loop3, loop4, and loop5 removed, the C code of the convolution in Figure 3.2 can be used for describing the computations in the FC layer. Therefore, we reuse the convolution kernel to also support the computations in the FC layer. Since the FC layer is a memory bounded layer (discussed in Section 2.3) and requires fewer computations as compared to the convolution layer, the performance of the FC layer is limited by the available external memory bandwidth of the FPGA board. The two solutions that our accelerator design supports to reduce the impact of the limited

board memory bandwidth are, 1) quantization of the FC layer weights to reduce memory bandwidth requirement, 2) batch-processing multiple images when the FC layers start in parallel, which increases the reusability of weights hence improves the throughput of the design.

3.2.5 Other Layers Kernel Design

Other layers, namely ELTWISE and RELU, are implemented in the memwrite kernel of Figure 3.5. The RELU and ELTWISE layers are implemented as optional functional units and are only activated when needed.

3.3 Optimization for Memory Access

Loop tiling [31] is an optimization technique that we adopt to reduce the amount and frequency of the external memory access, thus relaxing the external memory bandwidth requirements and improving the performance of the CNN accelerator. Loop tiling optimizes the utilization of the on-chip memory by storing a block of input data onto the on-chip memory and reusing the block of data stored locally as cached data. Since shift-register-based buffers are the most efficient buffering scheme for pipelined processing in a systolic array architecture, it is implemented as the buffering scheme in our accelerator for storing the block of input feature maps. One should note that the Intel FPGA OpenCL SDK compiler only synthesizes shift-register-based buffers efficiently in single-threaded kernel mode. Therefore, the memread kernel is implemented in the single-threaded kernel mode in our accelerator. Figure 3.6 represents a shift-register-based buffer. With shift-register-based buffer implementation, all the elements are reused until they are pushed out of the buffer. The shift-register-based buffering scheme eliminates the need for using wide multiplexers to feed the data into PEs and significantly simplifies the interconnections between the memory

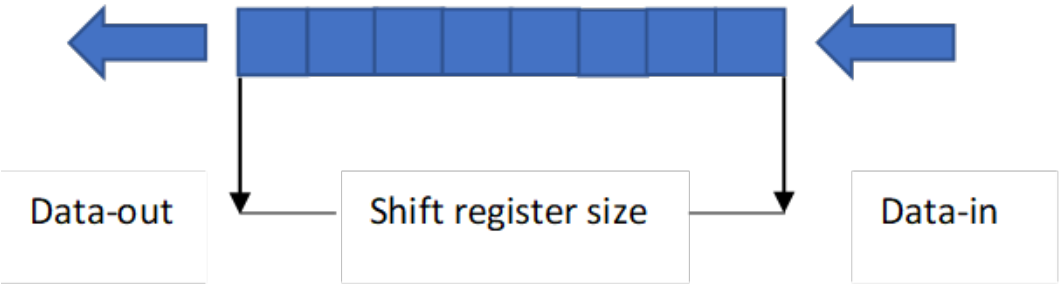

Figure 3.6: Loop Tiling Implemented Using Shift-Register-based Buffer Where Data Is Available for 8 Cycles and Can Be Reused for 8 Cycles

buffers. This reduces the critical path delay and hence increases the frequency of the accelerator. The size of the shift-register-based buffer is determined by the reuse_fac and vec_fac and is given by Equation 3.4,

$$SR\_size = reuse\_fac \times vec\_fac \tag{3.4}$$

with the step size of shifting per clock cycle equaling to vec_fac bits. The shift-register-based buffer decreases the overall external memory access by the factor indicated by Equation 3.4, thus improving the memory bandwidth efficiency and the overall system-level performance of CNN accelerator.

3.4 Reading of Input Feature Map

The input feature maps can be considered as 3-dimensional data, and the vec_fac parameter determines the size of the input feature maps transferred from the external memory per cycle. The objective of our accelerator design is to minimize the number of output results generated by each PE to reduce the size of the output multiplexer, which collects the output results and transfers them to the external memory. The reading of the input feature maps whether along the channels or the rows results in

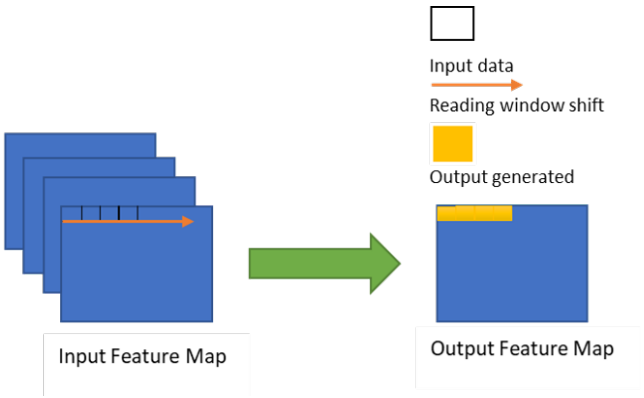

Figure 3.7: Reading along the Row with Vec_fac Equal to 4 Results in Generation of Vec_fac Number of Outputs in Parallel

a different number of outputs generated in parallel. So, optimizing the reading of the input feature maps becomes vital to reducing the size of the output multiplexer.

Memory accesses are coalesced to increase the efficiency of available memory bandwidth. With vec_fac defining the amount of data transferred from the external memory per clock cycle, there are two possible ways of reading the input feature maps:

- **Reading along the row**: Reading vec_fac size of data along the row of the input feature maps per cycle, as shown in Figure 3.7, results in parallelizing loop4 of Figure 3.2 to increase parallel computations, thus generating the output size of vec_fac per PE.

- **Reading along the channels**: Reading vec_fac size of data along the channels of input feature maps, as shown in Figure 3.8, results in parallelizing loop2 of Figure 3.2 to increase parallel computations, and thus generating the output size of one per PE.

Comparing the two reading pattern, reading the input feature maps along the channels generates outputs, which are $\frac{1}{vec\_fac}$ time fewer outputs as compared to reading along the row. Thus, to reduce the size of the output multiplexer in our

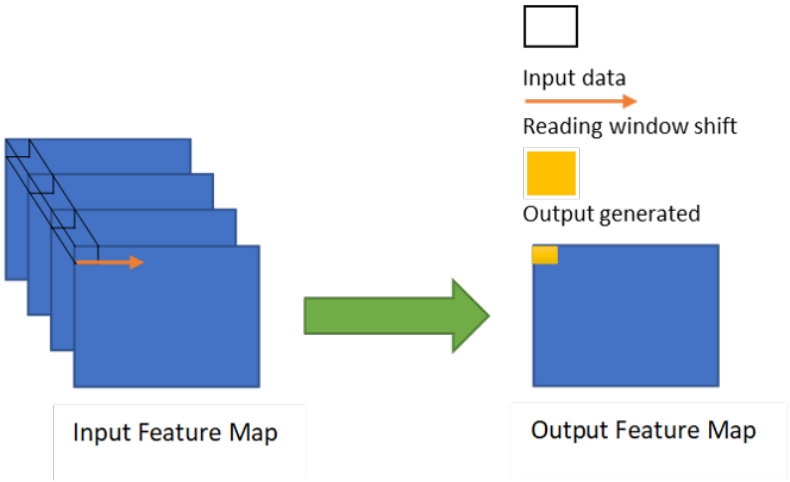

Figure 3.8: Reading along the Channel with Vec_fac Equal to 4 Results in Generation of One Output

accelerator, data is read along the channels. The size of the output multiplexer depends on the pe_num and vec_fac parameters and is given by Equation 3.1.

3.5 Design for Scalability

The proposed 1-D systolic array architecture in Section 3.1.1 reduces routing congestion and removes the input multiplexer from the design, which increases the scalability of the design. However, as the design scales up, fan-in of the storage units and fan-out of the load unit issues become the bottleneck that prevents the further upscaling up of the design.

The high fan-in issue exists in the storage unit that collects the output, which it receives from multiple PEs in the memwrite kernel of Figure 3.5. To resolve this issue, we propose the solution discussed in Section 3.3 by reading the input feature maps along the channels to minimize the size of the output multiplexer and hence reduce fan-in of the storage unit.

The high fan-out issue exists in the load units that are used to transfer the input

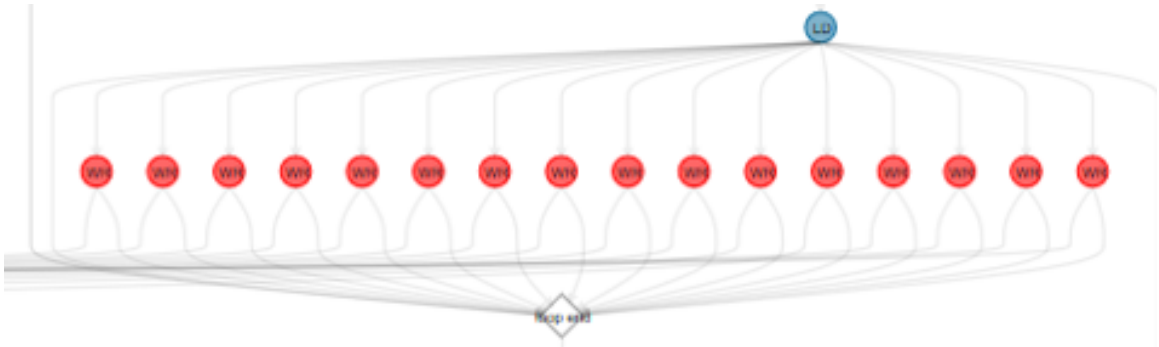

Figure 3.9: Fan-out Issue for Load Unit as Load Unit Driving 16 PEs

feature maps and the weights from the external memory to the on-chip buffer. The full utilization of resources is not possible because the high fan-out of load units creates a routing congestion problem as the parallelism of computation increases. That is why resolving fan-out issues in the design is critical, enabling the design to scale up.

For our design, the high fan-out issue exists in the input feature maps and weights load units, which stream the data to the pe_num number of PEs through memory channels. The proposed 1-D systolic array convolution architecture shifts the input feature maps from one PE to another PE. Thus, it resolves the fan-out issue for the input feature map load unit. However, the fan-out of the weight load units driving pe_num PEs increases proportionally as the number of PE. For resolving the high fan-out of the weight load unit, we propose the solution to generate multiple load units to transfer the weights from the external memory to the on-chip buffer. Figure 3.9 illustrates the fan-out issue where one load unit driving sixteen PEs results in the high fan-out for the load unit. By replacing the one large load unit with four load units, each with the fan-out factor of four, as shown in Figure 3.10, we are able to resolve the routing congestion problem of the tool as well as improve the operating frequency of the accelerator design by 1.1 times. This enables the accelerator to scale up to increase the utilization of DSP blocks.

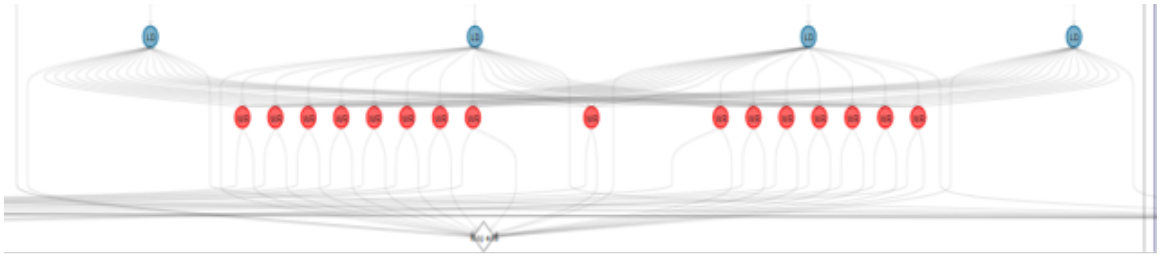

Figure 3.10: Fan-out of Load Unit Reduced by Generating 4 PEs Driving 16 PEs

We also experimented to determine the maximum bit width of the data port that a load unit can support without having a routing congestion issues. Equation 3.5 determines the number of load units for Arria 10 GX115 FPGA, where 2048-bit is found to be the maximum bit width suitable for one load unit, regardless of whether the data type is single-precision floating number or fixed-point representation.

$$Numberofloadunits = (vec\_size \times pe\_num \times bitsizeofdatatype)\%2048 \qquad (3.5)$$

The automated generation of the load units enables us to scale up our design freely and easily achieve 100% utilization of the DSP resources on FPGAs.

Chapter 4

EXPERIMENTAL SETUP

To run our experiments, we have used Intels Arria 10 GX115 FPGA, shown in Figure 4.1. This FPGA has 1518 hardened floating-point DSP blocks that perform computation. FPGA board has an external 2GB of DDR4 memory, with the memory bandwidth of 19GB/s. Intel FPGA SDK for OpenCL version 18.0 is used to compile device code. Host code is written in C/C++ and device code in OpenCL C.

After mapping design, to demonstrate our design flexibility we present performance of multiple CNN models, Alexnet [3], Resnet-50 [4], Retinanet [23] and Lightweight retinanet [24] in Chapter 6. OpenCV [32] is used to pre-process images before it is sent to FPGA. We use a 32-bit floating-point and fixed-point representation for weights, input feature maps, and output feature maps. For fixed-point representa-

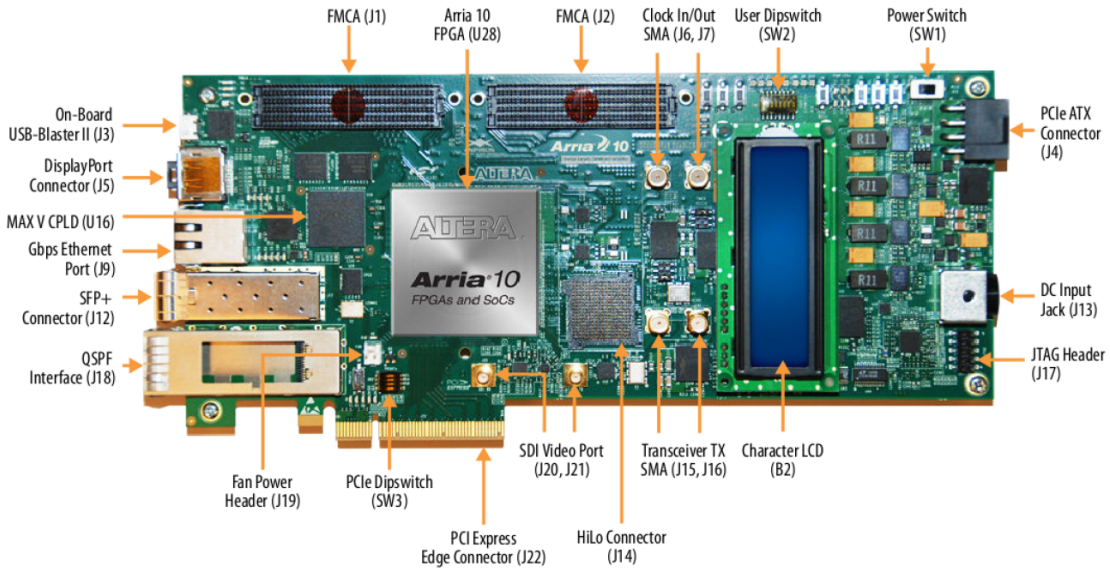

Figure 4.1: Intel Arria 10 FPGA Board Equipped with Arria 10 FPGA

tion, we measure the performance for the 8-bit data type of both the weight and input feature maps. The architecture parameters are used to explore the available FPGA resources, and the impact of parameters in scaling up the design is discussed in Chapter 5.

Chapter 5

DESIGN SPACE EXPLORATION

The target of our CNN accelerator is to fully utilize the available DSP blocks to maximize the parallel computations that would enable the CNN accelerator to achieve the maximum performance. The three architectural parameters defined for the accelerator, namely pe_num, reuse_fac, and vec_fac, are explored to scale up the DSP blocks utilization.

The scaling up of the architectural parameters increases the size of the input shift register-based buffer (Equation 3.4) and the weights buffer (Equation 5.1), which increases the utilization of the on-chip memory blocks and the demand of the external board memory bandwidth.

$$weightsbuffersize = pe\_num \times vec\_size \tag{5.1}$$

Equation 3.5 is used to determine the number of load units synthesized for transferring the weights from the external memory to the on-chip buffer. Architecture parameters reduce the external memory accesses for both the weights and input feature maps. The total reduction in the external memory accesses is given by Equation 5.2.

$$Reductioninexternalmemoryaccess = 2 \times Pe\_num \times vec\_fac \times reuse\_fac \tag{5.2}$$

Each architecture parameter has a different impact on the increase in the demand for the external board memory bandwidth, on-chip memory blocks utilization, and DSP blocks utilization. So, we explore different combinations of the architectural parameters to maximize the DSP blocks utilization such that on-chip memory blocks

Resources	Available Units
Logic Elements(k)	427,200
On-chip Memory Blocks	2,713
DSP Blocks	1,518

Table 5.1: Available FPGA Resources

utilization and the external board memory bandwidth is not the limiting factor in the scaling up of the accelerator.

We perform our experiments on the Intel Arria 10 GX1150 FPGA with the available DSP and on-chip memory blocks, given in Table 5.1. The performance of the Alexnet CNN model is measured to explore the values of the architectural parameters.

The vec_fac determines per-cycle data transferred from the external memory to the input shift register-based buffer. As a result, the vec_fac parameter depends on the burst-size of data per-cycle by the external memory and bit-size of the input feature map and weights. Vec_fac is determined by Equation 5.3

$$vec\_fac = \frac{(burst\_size)}{(data - bit - size)} \tag{5.3}$$

The vec_fac obtained by Equation 5.3 enables the accelerator to achieve high memory bandwidth utilization. So, based on Equation 5.3, and with the available external board memory burst-size of 512 bits, we fixed the vec_fac parameter to be 16 for 32-bit single-precision floating input feature maps and weights.

Increasing pe_num parameter value adds more PEs to the accelerator and increases the transfers of the weights from the external memory onto an on-chip buffer thus increasing the demand of the external memory bandwidth. Thus, pe_num value is limited by the available external board memory bandwidth. To determine the op-

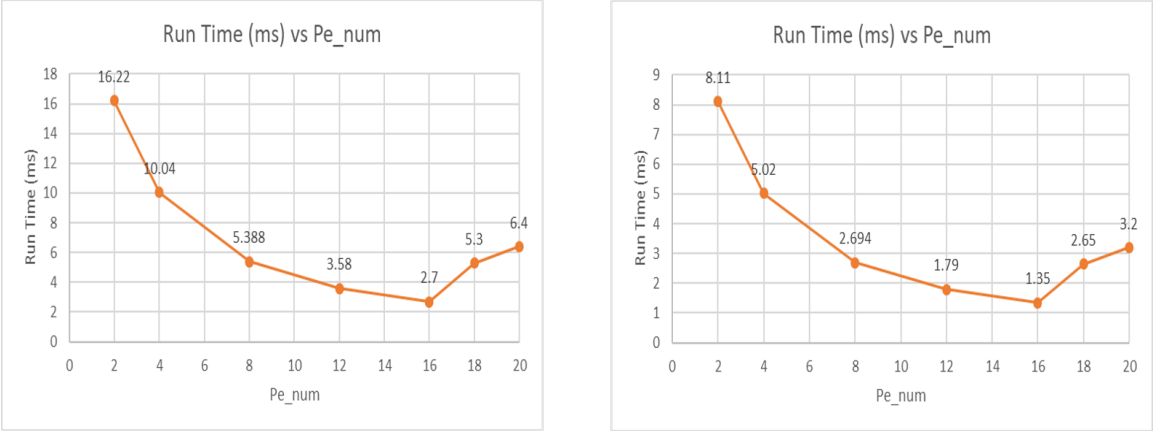

Figure 5.1: Change in Run Time (ms) of FC6 and FC7 layers by changing pe_num from 2 to 16 while reuse_fac and vec_fac fixed to 1 and 16 respectively

timized value of pe_num, the run time of the most memory intensive layers of the Alexnet CNN model, that is first two FC layers (FC6 and FC7), are measured, as shown in Figure 5.1 with pe_num changing from 1 to 20 while vec_fac fixed to the optimized value of 16 and reuse_fac fixed to 1. As can be seen from Figure 5.1, the run time of FC6 and FC7 layers increases as pe_num value becomes greater than 16. This is because of the available external board memory bandwidth limitation. So, pe_num value is fixed to 16.

The reuse_fac parameter determines the size of the shift register-based buffer (Equation 3.4) to store the input feature maps, which reduces the external memory accesses, and the number of results generated in parallel by each PE. Increasing this parameter is not limited by the external board memory burst-size or memory bandwidth but depends on the available on-chip memory blocks. Figure 5.2 shows the improvement in the performance of the Alexnet CNN model as DSP blocks utilization increases with the increase in reuse_fac parameter from 1 to 4 while pe_num and vec_fac parameters fixed to their optimized value of 16.

After determining the optimized values of the three architecture parameters, the

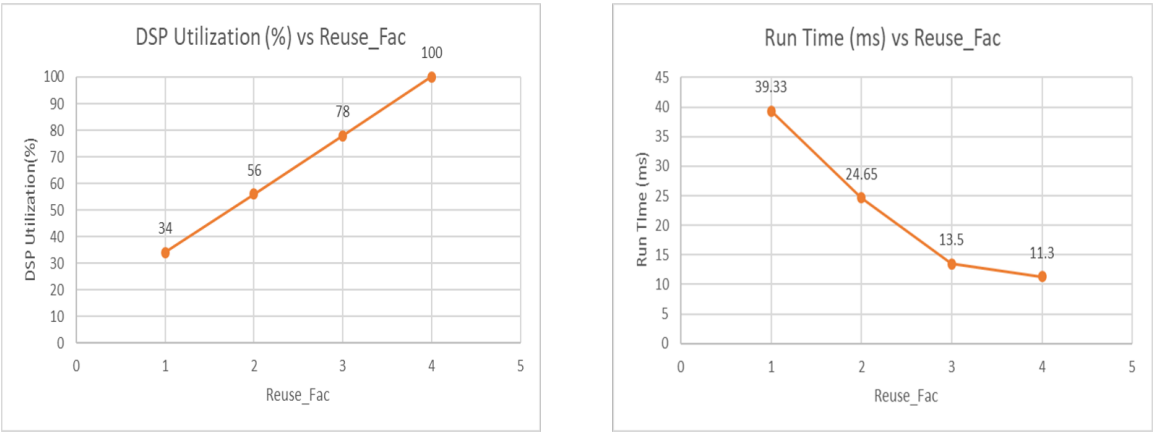

Figure 5.2: Change in Run Time (ms) of the Alexnet CNN model with increase in DSP blocks utilization (%) by increasing Reuse_fac from 1 to 4

pe_num	reuse_fac	vec_fac
16	4	16

Table 5.2: Parameters Value

CNN accelerator utilizes 100% of the available computing resources on an Intel Arria 10 GX1150 FPGA with architecture parameters values shown in Table 5.2.

Chapter 6

RESULTS

In this chapter, we present the performance of the CNN models, namely, Alexnet, Resnet-50, Retinanet, and Lightweight Retinanet. We compare our CNN accelerator performance with state-of-the-art CNN accelerators performances for the Alexnet CNN model in Table 6.1. We also compare FPGA resource utilization in Table 6.1.

We use latency to process one image as the parameter to compare the performance of CNN accelerators. [15] is the open-source CNN accelerator. We implemented this

Design	[11]	[15]	[16]	This work
FPGA	Arria 10 GT1150	Arria 10 GX1150	P395-D8	Arria 10 GX1150
CNN	Alexnet	Alexnet	Alexnet	Alexnet
Precision	Float(32 bit)	Fixed(8 bit)	Fixed(8-16 bit)	Float(32 bit)
Logic Utilization	350K(82%)	105K(25%)	NA	250K(59%)
BRAM Utilization	2360(86%)	641(24%)	NA	2472(91%)
DSP Utilization	1290(85%)	377(25%)	NA	1518(100%)
Latency/image(ms/img)	4.03	22.21	20.21	11.3
Frequency(MHz)	239	250	150	202
Winograd Convolution	Yes	No	No	No

Table 6.1: Performance and resource utilization Comparison with Existing Accelerators

CNN Model	Resnet-50	Retinanet	Lightweight Retinanet
Latency/image(ms/img)	84	1614.9	990.34
FLOPS(GFLOPS)	4	156	89

Table 6.2: Performance of Resnet-50, Retinanet and Lightweight Retinanet with 32-bit floating point representation of the weights and input feature maps

CNN Model	Alexnet	Resnet-50	Retinanet	Lightweight Retinanet
Latency/image(ms/img)	6.03	32	745.2	414

Table 6.3: Performance of Alexnet, Resnet-50, Retinanet and Lightweight Retinanet with 8-bit fixed-point representation of the weights and input feature maps

CNN accelerator on the available FPGA to measure the maximum performance, as shown in Table 6.1. DSP blocks utilization for [15] is limited to 25% of the available DSP blocks, mainly because of the fan-in and fan-out issues discussed in section 3.5. It can be seen from Table 6.1, our CNN accelerator outperforms the existing work [15], [16] except for [11]. The CNN accelerator of [11] and [9] shows the advantage of adopting Winograd transformation [33], which decreases the number of computations required to perform convolution. So, to further improve the performance, Winograd transformation needs to be incorporated as the part of future work. By comparing the resource utilization from Table 6.1, no other CNN accelerators have been able to utilize 100% of the DSP blocks, except for our CNN accelerator.

Table 6.2 shows the performance measured using the same CNN accelerator, as we used for comparing the performance of the Alexnet CNN model (in Table 6.1), with a 32-bit floating-point representation of the input feature maps and weights.

The performance shown in Table 6.2 demonstrates the flexibility of our CNN accelerator with multiple CNN models. We also estimate the performance of the

CNN models using the 8-bit fixed-point representation of the input feature maps and weights. To support the fixed-point representation of the input feature maps and weights, we use Intel's OpenCL support for the arbitrary precision data type. It also requires recompiling OpenCL kernels with the fixed-point representation of the input feature maps and weights and with the CNN architecture, which is the same as we proposed in Chapter 3. The CNN accelerator with the 8-bit fixed-point representation of the input feature maps and weights, running at a frequency of 233.09 MHZ, utilizes 65% (1001) of the available DSP blocks, and 63% of(1718) of on-chip memory blocks. Table 6.3 shows the performance of the CNN models that we have used for our experiments with the 8-bit representation of the input feature maps and weights.

Chapter 7

CONCLUSION

In this thesis, we presented a generalized CNN accelerator using OpenCL. We identified the core operation of video analysis -CNN and proposed a 1-d systolic array architecture to achieve high utilization of the available FPGA resources. We resolved the challenges that exist in state-of-the-art CNN accelerators to achieve high performance. We automated the scaling up of the CNN accelerator by introducing the three architectural parameters and proposed a strategy to scale up the accelerator using three architectural parameters that optimizes the external memory bandwidth utilization and computing units utilization. We presented the performance of the multiple CNN models to show the flexibility of our CNN accelerator.

REFERENCES

[1] Steven Borowiec. Alphago seals 4-1 victory over go grandmaster lee sedol. *The Guardian*, 15, 2016.

[2] 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.

[3] 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.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[5] Saman Biookaghazadeh, Ming Zhao, and Fengbo Ren. Are fpgas suitable for edge computing? In {*USENIX*} *Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.

[6] Erwei Wang, James J Davis, Ruizhe Zhao, Ho-Cheung Ng, Xinyu Niu, Wayne Luk, Peter YK Cheung, and George A Constantinides. Deep neural network approximation for custom hardware: Where we've been, where we're going. *ACM Computing Surveys (CSUR)*, 52(2):40, 2019.

[7] Kamel Abdelouahab, Maxime Pelcat, Jocelyn Serot, and François Berry. Accelerating cnn inference on fpgas: A survey. *arXiv preprint arXiv:1806.01683*, 2018.

[8] Hamid Reza Zohouri, Artur Podobas, and Satoshi Matsuoka. Combined spatial and temporal blocking for high-performance stencil computation on fpgas using opencl. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 153–162. ACM, 2018.

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

[10] Roman A Solovyev, Alexandr A Kalinin, Alexander G Kustov, Dmitry V Telpukhov, and Vladimir S Ruhlov. Fpga implementation of convolutional neural networks with fixed-point calculations. *arXiv preprint arXiv:1808.09945*, 2018.

[11] 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.

[12] D. T. Nguyen, T. N. Nguyen, H. Kim, and H. Lee. A high-throughput and power-efficient fpga implementation of yolo cnn for object detection. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(8):1861–1873, Aug 2019. doi: 10.1109/TVLSI.2019.2905242.

[13] Ruizhe Zhao, Xinyu Niu, Yajie Wu, Wayne Luk, and Qiang Liu. Optimizing cnn-based object detection algorithms on embedded fpga platforms. In *International Symposium on Applied Reconfigurable Computing*, pages 255–267. Springer, 2017.

[14] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[15] Dong Wang, Ke Xu, and Diankun Jiang. Pipecnn: An opencl-based open-source fpga accelerator for convolution neural networks. In *2017 International Conference on Field Programmable Technology (ICFPT)*, pages 279–282. IEEE, 2017.

[16] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. 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*, pages 16–25. ACM, 2016.

[17] Jiaxi Zhang, Wentai Zhang, Guojie Luo, Xuechao Wei, Yun Liang, and Jason Cong. Frequency improvement of systolic array-based cnns on fpgas. In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4. IEEE, 2019.

[18] H Kung. Algorithms for vlsi processor arrays. *Introduction to VLSI systems*, pages 271–292, 1980.

[19] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[20] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[21] Chaochao Lu and Xiaoou Tang. Surpassing human-level face verification performance on lfw with gaussianface. In *Twenty-ninth AAAI conference on artificial intelligence*, 2015.

[22] caffe2, 2018. URL `https://github.com/facebookarchive/caffe2`.

[23] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 2980–2988, 2017.

[24] Yixing Li and Fengbo Ren. Light-weight retinanet for object detection. *arXiv preprint arXiv:1905.10011*, 2019.

[25] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature pyramid networks for object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2117–2125, 2017.

[26] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 45–54. ACM, 2017.

[27] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[28] Ross Girshick, Ilija Radosavovic, Georgia Gkioxari, Piotr Dollár, and Kaiming He. Detectron. `https://github.com/facebookresearch/detectron`, 2018.

[29] 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.

[30] Hsiang-Tsung Kung. Why systolic architectures? *IEEE computer*, 15(1):37–46, 1982.

[31] Jingling Xue. *Loop tiling for parallelism*, volume 575. Springer Science & Business Media, 2012.

[32] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. " O'Reilly Media, Inc.", 2008.

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