FPGA Acceleration of CNNs Using OpenCL

by

Pravin Kumar Ravi

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

Approved July 2020 by the
Graduate Supervisory Committee:

Ming Zhao, Chair
Baoxin Li
Fengbo Ren

ARIZONA STATE UNIVERSITY

August 2020

ABSTRACT

Convolutional Neural Network (CNN) has achieved state-of-the-art performance in numerous applications like computer vision, natural language processing, robotics etc. The advancement of High-Performance Computing systems equipped with dedicated hardware accelerators has also paved the way towards the success of compute intensive CNNs. Graphics Processing Units (GPUs), with massive processing capability, have been of general interest for the acceleration of CNNs. Recently, Field Programmable Gate Arrays (FPGAs) have been promising in CNN acceleration since they offer high performance while also being re-configurable to support the evolution of CNNs. This work focuses on a design methodology to accelerate CNNs on FPGA with low inference latency and high-throughput which are crucial for scenarios like self-driving cars, video surveillance etc. It also includes optimizations which reduce the resource utilization by a large margin with a small degradation in performance thus making the design suitable for low-end FPGA devices as well.

FPGA accelerators often suffer due to the limited main memory bandwidth. Also, highly parallel designs with large resource utilization often end up achieving low operating frequency due to poor routing. This work employs data fetch and buffer mechanisms, designed specifically for the memory access pattern of CNNs, that overlap computation with memory access. This work proposes a novel arrangement of the systolic processing element array to achieve high frequency and consume less resources than the existing works. Also, support has been extended to more complicated CNNs to do video processing. On Intel Arria 10 GX1150, the design operates at a frequency as high as 258MHz and performs single inference of VGG-16 and C3D in 23.5ms and 45.6ms respectively. For VGG-16 and C3D the design offers a throughput of 66.1 and 23.98 inferences/s respectively. This design can outperform other FPGA 2D CNN accelerators by up to 9.7 times and 3D CNN accelerators by up to 2.7 times.

ACKNOWLEDGEMENT

I would love to thank my advisor Dr. Ming Zhao for the opportunity, encouragement and support he extended to me. My Masters would not have been possible without his valuable advice. I would love to thank Dr. Fengbo Ren for his support throughout the time by granting me access to the hardware resources which were very crucial for this project. I would love to thank Dr. Baoxin Li for being a part of the committee and reviewing my work. I would love to thank Saman Biookaghazadeh for educating me the perks of research and for providing constant guidance on my work. His directions helped me move in the right direction.

This work is in honor of the incomprehensible superpower, God, for He is my strength and shield who surrounds me with shouts of deliverance. To me, Masters program was a period of spiritual growth, needless to say, the massive growth in technical knowledge. I owe my deepest gratitude to my father, mother and sister for their unconditional love, support, motivation, and for showering me with valuable advice and guidance. I would love to thank my friends, who are like my family in a foreign country, for making this journey a memorable one.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Neural Networks, inspired by the working of the brain, are used to model patterns and form predictions. Neural networks have leveled up the accuracy of predictions in applications like image processing, video understanding, forecasting etc. Convolutional Neural Network is a multilayer network where each neuron is connected with the neurons in a particular local region of previous layer. This local connectivity makes CNNs suitable for applications like computer vision, robotics etc. CNN is highly compute-intensive due to convolution operation which are their basic building blocks.

Multi-core CPUs and GPUs were extensively used to meet the tremendous computation needs of CNNs. However, the high power consumption of CPUs and GPUs makes them unsuitable for energy-constrained systems such as edge-computing Biookaghazadeh *et al.* (2018). Also, they cannot promise low single inference latency. While ASICs can provide energy-efficient, low latency inferences, they are not suitable for the evolving nature of CNNs. Recently, FPGAs have gained attraction in CNN acceleration since they are power efficient and provide high throughput than CPUs and GPUs while also being reconfigurable to support the evolution of CNNs.

CNNs learn new information during training phase and apply the knowledge during inference phase to make a prediction in real-time. In applications like self-driving cars, video surveillance etc. only inferences are performed by the accelerator. In such real-time scenarios, low inference latency is very crucial and failure to meet the latency constraint might be disastrous. This work is focused on building a low-latency CNN processor on FPGA which is generic to support a wide range of CNN architectures.

In this thesis, we present a novel design methodology that can leverage the parallel power of an FPGA. We implemented this work in OpenCL that allows easy integration with deep learning frameworks like Tensorflow Abadi *et al.* (2016) and Caffe Jia *et al.* (2014). We adopted the systolic array of processing elements and studied various inefficiencies that exist. Using a novel arrangement of processing elements we reduced the resource consumption with no loss of performance. We further explored optimizations that significantly reduced the resource consumption with a small loss in performance. We performed optimizations that improved the operating frequency by a large margin. Also, we extended the systolic array to support 3D convolutions to accelerate CNN applications that perform complex operations like video processing.

To demonstrate the performance of our framework, we performed experiments on different 2D CNNs like VGG-16, Alexnet, Resnet and the 3D CNN C3D. Our implementation shows superior performance compared with existing works.

Chapter 2

BACKGROUND

2.1 Convolutional Neural Network

Convolutional Neural Network is the backbone in applications like image classification Litjens *et al.* (2017), reinforcement learning Arulkumaran *et al.* (2017), natural language processing etc. CNN performs a sequence of operations, each considered as a layer. Each layer accepts an input from previous layer, performs the computation and produces output which becomes the input for the next layer. CNN comprises of layers such as convolution, pooling, fully-connected, ReLU etc. The neurons in each layer is activated based on the neurons in a particular local region of the previous layer. This local connection makes CNN suitable for learning the information in a local area along the spatial dimensions of the input.

CNNs are compute-intensive due to the arithmetic complexity involved in convolution layers and fully connected layers. The size of parameters is also large only for these two layers. Table 2.1 represents the percentage of operations and size of

Layer	Operations	Data size
Convolution	99.20%	8.61%
Fully connected	00.79%	91.38%
Pooling	00.00%	00.00%

Table 2.1: Contribution of VGG-16 Layers to Total Arithmetic Operations and Input/Weight Parameters

Layer	Operations	Data size
Convolution	99.96%	26.72%
Fully connected	00.03%	73.28%
Pooling	00.00%	00.00%

Table 2.2: Contribution of C3D Layers to Total Arithmetic Operations and Input/Weight Parameters

input/weight parameters contributed to the whole network by convolution and fully connected layers. In VGG-16, around 99.20% of the operations is performed in convolution layers while 0.79% is performed in fully-connected layers. The size of parameters contributed by convolution layers is 8.61% and fully-connected layers is 91.38%. Table 2.2 shows the similar information for C3D. Based on this observation, we focus mainly on accelerating convolution and fully-connected layers. Other layers do not pose a bottleneck and do not affect the latency because of which we skip explanation for them.

Figure 2.1 demonstrates a convolution layer. The input feature map consists of 3 channels, represented as red, green and blue. Each channel is a 2D array of dimension $I_w \times I_h$. There are m weight filters each of which has K_w width, K_h height, and the same number of channels as the input. In CNNs, a convolution operation involves performing dot-product of a weight filter and an input region to produce an output data element, as shown in the figure. The weight filter slides over the input along width and height dimensions to produce output data elements along those dimensions for a particular channel. Performing convolution with m weight filters produces m output channels. In a 3D convolution operation, input feature maps, output feature maps, and weight consists of an additional dimension often referred as frames. In

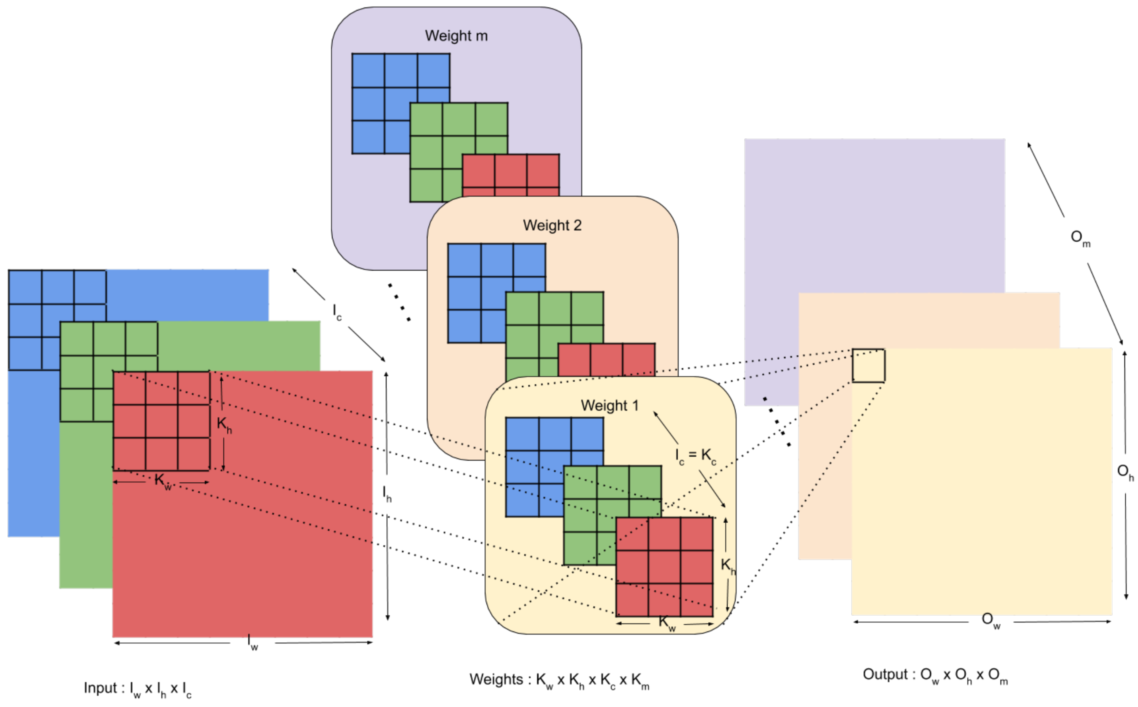
Weight m
Weight 2
Weight 1
I_c
K_h
K_w
I_h
I_w
$I_c = K_c$
K_h
K_w
O_m
O_h
O_w
Input : I_w x I_h x I_c
Weights : K_w x K_h x K_c x K_m
Output : O_w x O_h x O_m

Figure 2.1: Convolution Layer

3D CNNs, each weight filter slides along frame dimension of the input in addition to width and height dimension.

Equation 2.1 and 2.2 represent 2D and 3D convolution operation respectively.

$$O[m][w][h] = \sum_{n=0}^{I_c} \sum_{i=0}^{K_w} \sum_{j=0}^{K_h} I[n][stride \times w + i][stride \times h + j] \times W[m][n][i][j] \quad (2.1)$$

$$O[m][f][w][h] = \sum_{n=0}^{I_c} \sum_{k=0}^{K_f} \sum_{i=0}^{K_w} \sum_{j=0}^{K_h} I[n][stride \times f + k][stride \times w + i][stride \times h + j] \times W[m][n][f][i][j] \quad (2.2)$$

where m represents the output channel; f represents frame; w and h represent width and height; I_c represents the number of input channels; K_w, K_h and K_f represent the dimensions of weight filter along width, height and frame dimension; n, k, i, and j represent the iterator along input channels, frames, width and height.

$$F(2,3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix}$$

$$m_1 = (d_0 - d_2)g_0 \qquad m_2 = (d_1 + d_2)\frac{g_0 + g_1 + g_2}{2}$$

$$m_4 = (d_1 - d_3)g_2 \qquad m_3 = (d_2 - d_1)\frac{g_0 - g_1 + g_2}{2}$$

Figure 2.2: F(2,3) Transformation of Winograd Algorithm

2.2 Winograd Transformation

Winograd transformation Lavin and Gray (2016) is a popular technique to reduce the arithmetic complexity involved in convolution operation. Winograd algorithm is practical for small filters because of which it is suitable for CNN models like VGG-16 and C3D which use 3×3 weight filters.

Winograd transformation is usually represented as F(m, r) where m represents the output size and r represents the convolution kernel size. Consider the convolution operation of data d and weight filter g, given by $d = [d_0\ d_1\ d_2\ d_3]$ and $g = [g_0\ g_1\ g_2]$. The Winograd transformations for this convolution, F(2,3), is shown in Figure 2.2.

The usual convolution takes 6 multiplications, while the F(2, 3) transformed equation requires only 4 multiplications. In our design, we have implemented F(6, 3) which reduced the number of multiplications by 58%.

2.3 Related Works

Since FPGAs are gaining attraction in CNN acceleration, there are several works which studied its acceleration using Hardware Description Language (HDL). However, OpenCL based acceleration is understudied. Moreover, these works do not support 3D CNNs to do video processing. PipeCNN Wang *et al.* (2017) is an OpenCL based implementation flexible enough to support various CNNs. It includes a complete design required to perform the inference phase fully on FPGA. It supports vectorization factors to explore the design space to find the optimal configuration. There are components to move data on/off the chip which are optimized enough to reduce the idle time of the processing elements. These components are connected using Intel channels. Moreover, it is the only related work which is open source. The design suffers from high fan-out due to which resource utilization is poor and also the design ends up achieving low frequency. Boutros *et al.* (2018) made a comparison between widely known CNN accelerators and showed that Deep Learning Accelerator (DLA) Aydonat *et al.* (2017) provides state-of-the-art performance. DLA adopts a systolic array of processing elements intended to simplify the routing and hence achieve high frequency. Systolic array achieves high DSP utilization which is crucial for a compute intensive application like CNN. We propose a novel arrangement of systolic array which consumes less resources than the trivial 1D systolic array. DLA caches the intermediate feature maps in the onchip buffers which is not possible for large networks. Moreover, DLA batches multiple inputs for fully-connected layers to enable weight reuse. This is not suitable for latency-critical applications. To reduce the overhead of fully connected layers we adopted a FC-processor alongside the convolution processor. Also we support 3D convolutions. Caffeine Zhang *et al.* (2018) uses a 2D systolic array for accelerating convolutions. It maps the fully connected layer computation onto the 2D systolic array. It also supports Caffe framework for deployment of the model onto the FPGA and performs batching to achieve weight reuse in fully-connected layers. However, the performance and frequency of Caffeine is low. Zhang *et al.* (2015) focuses much on exploring the design space and performs design optimizations to find the best performance with low FPGA resource consumption. It's design space exploration also aims to maximize throughput. Suda *et al.* (2016)

also performs design space exploration to achieve high throughput like the previous work. But these related works do not support 3D convolutions and also provide low performance.

There are several works that have studied the acceleration of 3D CNNs on FPGAs. Shen *et al.* (2018) and Liu *et al.* (2019) explore the uniform acceleration of 2D and 3D convolutions on FPGA with the same design methodology. Further, they explore the design space efficiently through a uniform analytical model for 2D and 3D CNNs. Hegde *et al.* (2018) provides acceleration specifically for 3D CNNs. They discuss various tiling and loop ordering techniques to optimize the computation of 3D convolution. Our design can achieve higher frequency and performance by extending the systolic array to support 3D convolutions.

Chapter 3

DESIGN

This chapter explains in detail the baseline architecture that we replicated from cutting-edge related works. The design was built using the source code of Wang *et al.* (2017), and so the format of PipeCNN are predominant in our design. The following sections explain the architecture of the host application running on the CPU and the device application programmed on the FPGA.

3.1 Host Application

The host process validates the configurations of each layer by performing condition checks on the dimensions of features and weights. Then the host process rearranges the input data into a format which allows efficient memory access on the FPGA, as explained in section 3.1.1. The host and device can operate concurrently, hence, while an inference is being made on the FPGA the host can rearrange the input for next inference. Then the host invokes the various kernels on the device explained in section 3.2.

3.1.1 Input Data Arrangement for Efficient DDR Access

Since FPGAs have significantly less DDR bandwidth than GPUs, optimal ways to access the DDR have been explored much in FPGA acceleration of CNNs. To utilize the DDR memory bandwidth efficiently each memory request should be wide enough and multiple memory accesses should be coalesced together so that the bus width is fully utilized. This can be achieved by rearranging the input, intermediate and output feature maps and weights of each layer in a suitable way. Weights are rearranged and loaded in the FPGA DDR memory only once during initialization of the device. The host application rearranges every input during run time before transferring it to the FPGA accelerator. The data is rearranged based on the access pattern of CNNs. In CNNs, a weight is convoluted with a region of input data, after which the weight slides along the width, and height of the input. A single convolution operation needs all the channels of a region of input data along width and height dimensions. Due to this, a partial channel-width major data arrangement as shown in 3.1, and as followed by Wang *et al.* (2017), is adopted to achieve high DDR efficiency. In this arrangement, the input channels are grouped together with each group consisting of VEC_SIZE number of channels, which is a compile time configuration parameter. The pixels which lie in the same width and height coordinate in all the VEC_SIZE input channels of one group are stored in consecutive memory addresses. This is called a lane of data and is shaded with dark red. The lanes which occur consecutively along width dimension are stored in consecutive memory addresses. Each consecutive row in a particular group is stored in consecutive memory addresses. This is followed by the group of channels which is behind the current group. The design accesses the memory in bursts of W_VEC (Winograd vectorization factor) requests which is determined by the Winograd transformation as explained in 3.2.6. Each DDR access on the device requests $VEC\_SIZE \times W\_VEC$ bytes of data which should be sufficient to fully utilize the bandwidth. The pixels shaded with dark red and light red form a $VEC\_SIZE \times W\_VEC$ volume called as plate. The pixels indicated in dark red, light red, and dark green form a $W\_VEC \times K_h \times K_c$ volume called as brick. Due to Winograd transformation, the size of weight filter is also $W\_VEC \times K_h \times K_c$. Performing a convolution of a brick of data and weight produces INV_VEC output pixels, where $INV\_VEC = W\_VEC - K_w + 1$. A brick is accessed from DDR by reading each plate separately.

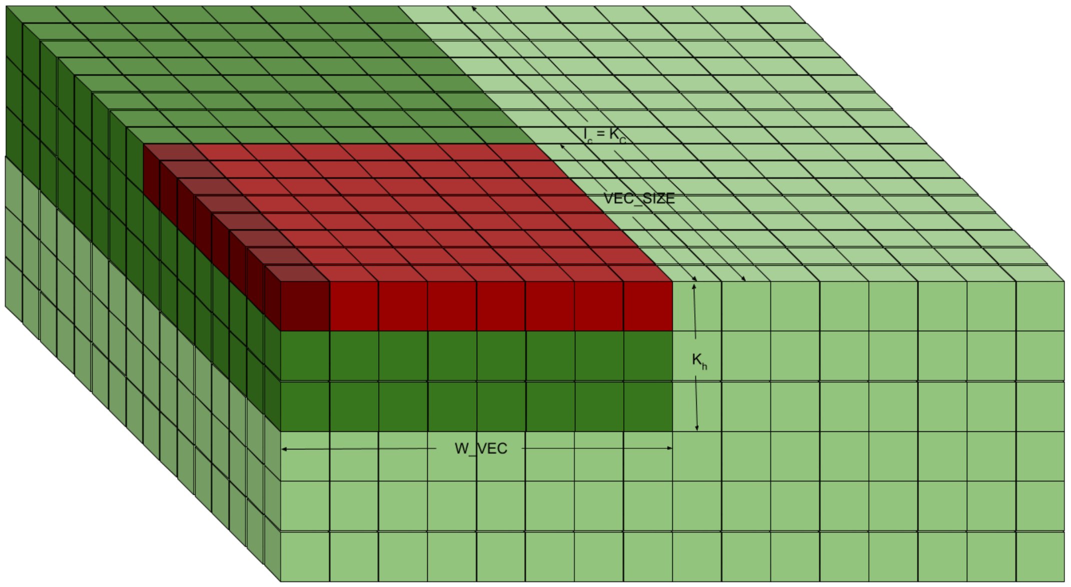

Figure 3.1: Data Rearrangement

3.2 CNN Accelerator Architecture

The accelerator consists of several modules each designated to perform a particular operation such as data movement, data transformation, processing etc. Each of these modules, implemented in OpenCL as individual kernels, form a separate stage in the pipeline and hence run concurrently. Communication between modules is enabled through Intel channels. Figure 3.2 shows an overview of the accelerator. At the core of the design is a systolic array of Processing Elements (PEs). To reduce the idle time of PEs, the design consists of data movement modules to move input features and weights from DDR to the PE array and the output produced by PE the array to DDR. Also, the design includes modules to perform the input/output transformation. The following sections explain the different OpenCL kernels.

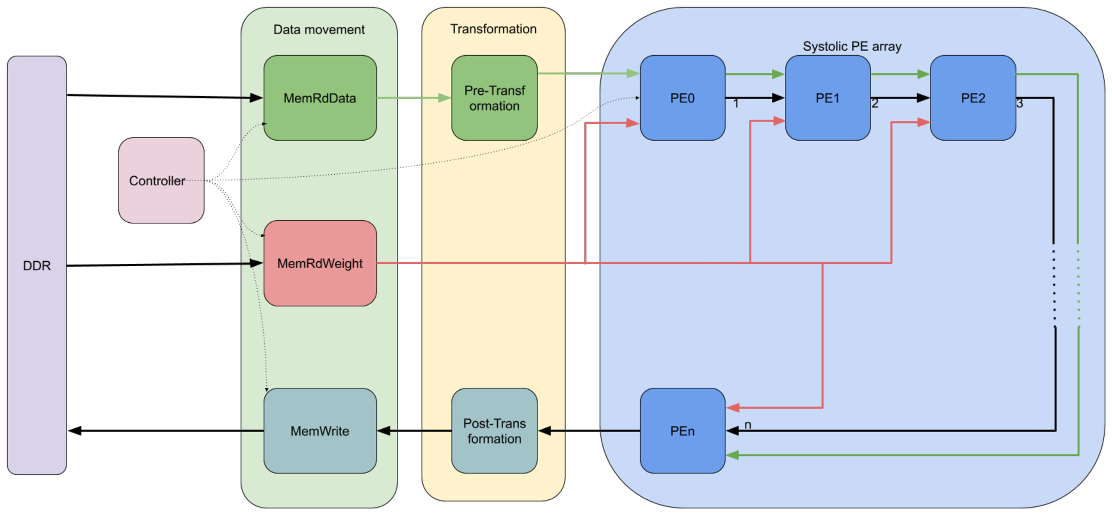

Figure 3.2: Architecture of CNN Accelerator

3.2.1 Controller

Controller initiates of the execution of a particular layer. Controller reads the configurations for every layer and communicates with every other kernel the dimensions of the input, output and weight. It is based on this information that the data movement kernels and PEs calculate the number of iterations that needs to be performed for a corresponding layer.

3.2.2 MemReadData

MemReadData module moves the data from DDR to PEs. MemReadData is optimized to operate on data which has been rearranged as explained in section 3.1.1. In every iteration, MemReadData reads a plate of feature from DDR and transfers to the PEs. Each lane of feature is read with an individual memory request because of which W_VEC number of DDR accesses are issued in one burst to read a plate. Intel OpenCL SDK implements a burst-coalesced cached LSU and coalesces W_VEC

reads into single burst thus reducing the number of requests and also utilizing the DDR bus width effectively. Also, the default cache enables data reuse and reduces the number of DDR read request.

3.2.3 MemReadWeight

MemReadWeight connects with every PE through dedicated channels and feeds the weight by transferring one plate of weight to one PE at a time. Each PE has a local weight buffer to store the weight filter and enable weight reuse by using them multiple times. Since the weight filters are read only once from DDR and are cached inside PE, MemReadWeight does not have a dedicated cache.

3.2.4 Systolic Array

Input feature reuse is possible along the output channel dimension, because of which the same feature plate needs to be transferred to all the PEs to be convoluted with different weight filters. Several accelerators follow a design where the memory module is connected to all the PEs. However, this design is inefficient due to the high fan-out which results in poor routing and low operating frequency.

At the core of this architecture is a systolic array which is an array of PEs. Figure 3.3 shows the architecture of a PE. Each PE in the systolic array performs I/O through interconnection with its neighbor PEs. This eases up the routing and the design achieves high frequency. Systolic arrays form the backbone of accelerators like TPU Abadi *et al.* (2016) and Intel DLA Aydonat *et al.* (2017). The array is made of $LANE\_NUM$ identical PEs, where $LANE\_NUM$ is the vectorization factor along output channels. Each PE handles one output channel, and hence uses one weight filter. PE_n receives input from PE_{n-1}, the PE immediately preceding it in the array, and passes it to PE_{n+1}, the PE following it in the array through Intel

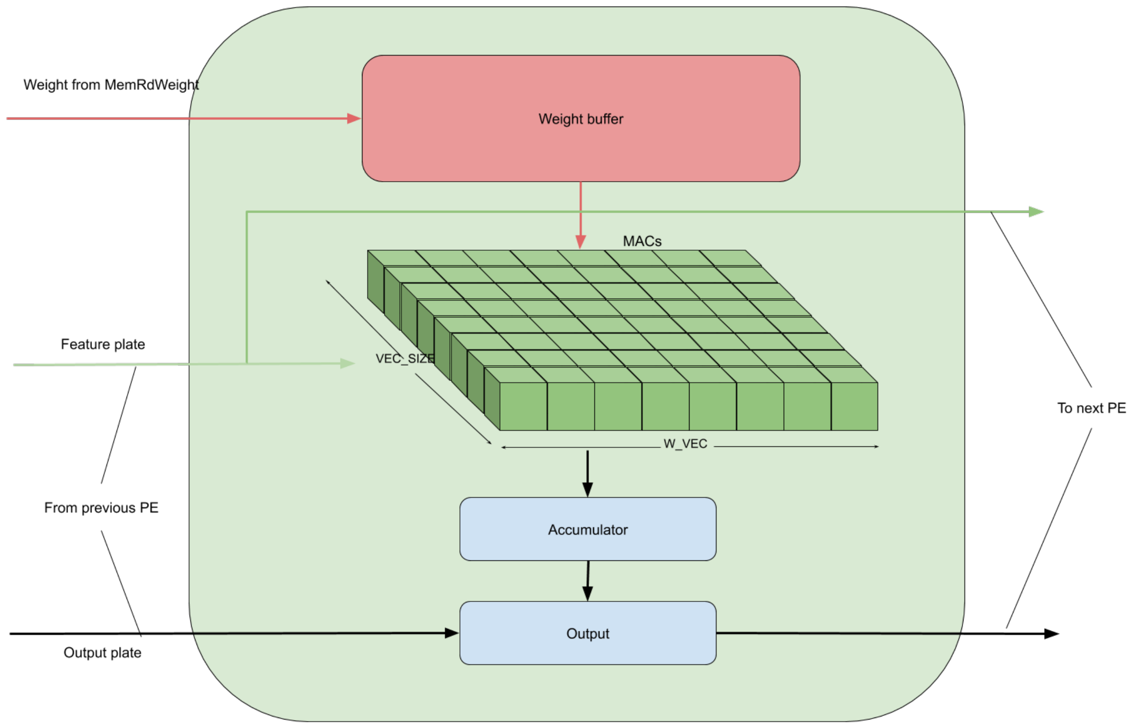

Figure 3.3: Architecture of PE

channels. PE_0 receives input data from MemReadData. Since the data channels should accommodate the transfer of a data plate every clock cycle, the data channels between any two subsequent PE is of width VEC_SIZE x W_VEC. Similarly, the output is also transferred in a systolic fashion. Each PE receives the output data from its previous PE, appends its output and forwards to the next PE. The last PE forwards the output to MemWrite. Each PE produces an output of dimension $1 \times W\_VEC$. To make the design fully pipeline, the output channels between any two subsequent PE should be wide enough to accommodate the output from all its previous PEs. Each PE is fed with weights through dedicated channels from MemReadWeight. The weight channel from MemReadWeight to any PE is of dimension $VEC\_SIZE \times W\_VEC$ to transfer a plate of weight every clock cycle.

3.2.5 MemWrite

MemWrite block transfers the output produced by PEs to DDR. The output channel of only the last PE in the systolic array is connected with MemWrite kernel. MemWrite rearranges the output data in an order as explained in section 3.1.1 and writes it to the DDR. The output data is in an arranged format as expected by MemReadData for next layer.

3.2.6 Winograd and Inverse Winograd Transformation

MemReadData transfers the data plates to a module which applies Winograd transformation on the data before feeding it to the PEs. The module takes data with W_VEC lanes and produces data with W_VEC lanes. The last PE in the systolic array transfer the output to a module which applies inverse Winograd transformation on the output before transferring it to the MemWrite block. The inverse module accepts Winograd output data with W_VEC lanes and produces inverse transformed output data with INV_W_VEC lanes, where $INV\_W\_VEC = W\_VEC - K_w + 1$

Chapter 4

3D CNN ACCELERATOR

3D CNNs are used in several video understanding applications (Hegde *et al.* (2018), Ji *et al.* (2012), Maturana and Scherer (2015), Sun *et al.* (2015), Tran *et al.* (2015)). While many works have studied the acceleration of CNNs, studies on 3D CNNs is limited. 3D CNNs consists of 3D convolution operations which employ an additional dimension in the feature maps and weights. In video processing, this dimension is used to learn the temporal information in the frames of video i.e. action or motion of the object. The input and intermediate feature maps have multiple frames and each frame has multiple channels. The weight filter, which also has multiple frames, slides over the frames of input to capture the temporal information. For instance, the first layer of C3D accepts an input of dimension $112 \times 112 \times 3 \times 16$ ($I_w \times I_h \times I_{ch} \times I_f$) and comprises weight filters of dimension $3 \times 3 \times 3 \times 3$ ($K_w \times K_h \times K_{ch} \times K_f$). The frames of weight filter slides along the frames of input feature map to capture the temporal information. This makes 3D convolutions more compute-intensive than 2D convolutions. The systolic array of processing elements can be extended to support these computationally complex operations. The challenges involved in extending a 2D CNN accelerator to support 3D CNN is entirely dependant on the underlying architecture. This chapter discuss the opportunities and challenges that are introduced by 3D CNNs while extending them support in our 2D CNN accelerator. First, the data reuse opportunity along the temporal dimension need to be exploited. Second, the order of computation of frames and output channels need to be studied since they can have an impact on performance and the amount of data read from main memory. Third, since 3D weights are large, we need techniques to alleviate the large weight buffer requirement.

4.1 Data Reuse Opportunity Along Frame Dimension

The temporal dimension introduces an additional reuse opportunity which needs to be exploited in order to minimize the number of main memory requests. The double data buffer, as explained in section 5.2.1, has been extended to cache multiple frames. The input feature map is partitioned into tiles along width, height and frame dimension. Each input tile of dimension $W \times H \times F$ can produce an output tile of dimension $(W - K_w + 1) \times (H - K_h + 1) \times (F - K_f + 1)$, where K_w, K_h and K_f represents the dimensions of weight along width, height and frame. For a constant onchip data buffer, caching along 3 dimensions allows higher data reuse, and hence lower main memory access, than caching along only 2 dimensions.

4.2 Order of Execution of Dimensions

The systolic array follows the optimal order of computation along width, height and channels for a 2D convolution. With the introduction of frame dimension, we explored how order of computation of frame dimension can be accommodated. The computation order of tiles and output channel dimension can affect the number of main memory access and computation time. In a output-channel major approach, all the output channels of a tile are computed before moving on to the next tile. The MemReadWeight loads each PE with a weight filter - total of n weight filters for a systolic array with n number of PEs. The MemReadData caches a tile of input block and streams to the PEs thus producing a n output channels for the particular tile. The weights are reloaded into the PEs to compute next n output channels for the same tile. This approach requires all weights to be loaded from main memory to compute each tile and hence increases the total amount of weights moved from DDR.

However, the input feature map is moved only once from main memory. In the early layers of C3D where the size of input is large and weight is small, output-channel major approach reduces the amount of data moved. In a tile-major approach, a set of output channels for all the tiles is computed before moving on to next set of output channels. In this approach, each PE is loaded with a weight filter. MemReadData streams all the input tiles and produces n output channels for all the output tiles. Then the PEs are reloaded with next set of weight filters and MemReadData streams all the input tiles again. This approach moves weights only once, but loads input features multiple times. Hence, this approach is suitable for later layers where weight is large and input feature map is small.

Figure 4.1 compares the latency of the convolution layers of C3D using the above two approaches. In comparison, the tile major approach outperforms the output-channel major approach by up to 1.3 times. We have adopted the tile major approach since low latency is our motive.

4.3 Large Weight Filters

The size of weight filters is larger in 3D convolutions, compared with 2D convolutions, due to the existence of additional dimension. The weight buffer in each PE should be large enough to accommodate the largest weight filter among all the layers. Hence, the weight buffer requirement for performing 3D convolutions is usually high. We performed experiments to alleviate this high memory requirement. First, we can follow a convolution layer split approach where each layer is split into as many sublayer layers as necessary such that the weight of each sublayer fits in the weight buffer. This technique, as explained in section 5.2.4, is also followed by Liu *et al.* (2019). However, this approach increases the amount of data moved from/to the main memory. Second, we can change the number of PEs to change the amount of onchip memory consumed.

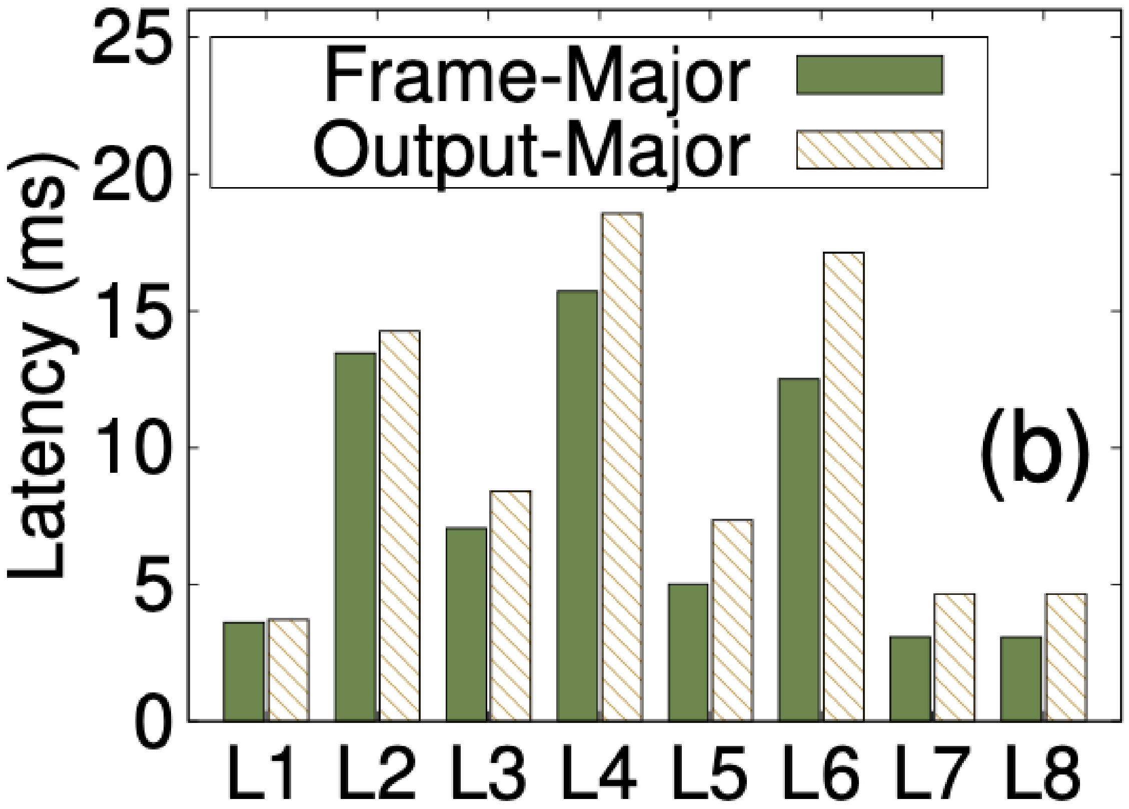

Figure 4.1: Performance Evaluation of Frame-major and Output Channel-major Designs

Since the weight buffer in each PE is constant for a given CNN architecture, total onchip memory consumed is given by $NUM_PEs \times WEIGHT_BUF_SIZE$. Changing the number of PEs, as explained in section 5.2.3, does not have a significant impact on the performance, given that the total number of DSPs utilized is the same. However, reducing the number of PEs increases the number of times input feature maps are loaded from the main memory. We can use a combination of these two approaches to minimize the amount of data loaded

Chapter 5

OPTIMIZATIONS

This chapter explains various optimizations which improve the latency of the design and also reduce the resource consumption. Section 5.1 explains techniques used to reduce the FF consumption of the design. Section 5.2 explains the onchip memory implementations to minimize offchip memory access and also to improve the latency. Section 5.3 provides an alternative design technique that achieves high operating frequency. Finally, section 5.4 provides an efficient mapping of fully connected layers and explains how a dedicated processor can increase the throughput of the accelerator.

5.1 Area Optimization

This section discusses the components which consume most resources and how performance can be traded-off for these resources. On Arria 10 GX 1150, the design consumes more than 500K FFs which is 30% of total FFs. Intel channels consume more than 300K FFs which is 18% of total FFs available and 60% of the total FFs consumed by the design. Our design extensively uses channels to (1) move configurations to all the kernels for every layer (2) move input feature, and weight to each PE and (3) move output feature from each PE. First, we propose a novel design which reduces the number of channels used while maintaining the performance. Second, we discuss an optimization which provides significant reduction in the number of channels with an insignificant increase in latency.

5.1.1 Semi-1D Systolic Array

We propose a novel arrangement of the systolic array where PEs are arranged in a 2D grid. The motive of this arrangement is to achieve a different output forwarding method which reduces the number of output channels required. Although the arrangement is 2D, the functionality of the systolic array resembles a 1D systolic array and hence we call it semi-1D systolic array.

In a 1D systolic array, each PE receives a block of output from its previous PE, appends its block of output and transfers the entire block to the next PE. The size of output block produced by each PE is of size W_VEC. For the systolic array to be fully pipelined, the output channel between any two neighbor PEs should be wide enough to transfer the output from all its previous PEs. In a systolic array with n PEs, the number of output channels between PE_0 and PE_1 is 1, PE_1 and PE_2 is 2, and $\mathrm{PE}_{\text{n-1}}$ and MemWrite is n. The total number of output channels is given by $W\_VEC * (1 + 2 + ... + n)$, or $W\_VEC * n * (n + 1)/2$, which becomes significantly large as the number of PEs increase. Designs with 16, 32 and 48 PEs consume 1088, 4224 and 9408 output channels respectively.

In the semi-1D systolic array, the PEs are arranged in a grid as shown in 5.1. In this arrangement, each $\mathrm{PE}_{\text{row,col}}$ receives feature plates from $\mathrm{PE}_{\text{row-1,col}}$ and forwards them to $\mathrm{PE}_{\text{row+1,col}}$. Each PE in the first row, $\mathrm{PE}_{\text{0,col}}$, receives feature plates from $\mathrm{PE}_{\text{0,col-1}}$, and forwards them to $\mathrm{PE}_{\text{0,col+1}}$ and and also to $\mathrm{PE}_{\text{1,col}}$. $\mathrm{PE}_{0,0}$ receives feature plates from memReadData. Each PE has a dedicated weight channel to receive weight plates from memReadWeight. Each $\mathrm{PE}_{\text{row,col}}$ receives a block of output from $\mathrm{PE}_{\text{row-1,col}}$, appends its output block and transfers the entire block to $\mathrm{PE}_{\text{row+1,col}}$. Each PE in the last row $\mathrm{PE}_{\text{nrow, col}}$ transfers the entire block of output data to $\mathrm{PE}_{\text{row,col+1}}$ which is the PE in the last row of the next column. The total number of output channels required now is given by $W\_VEC * (ncol * (nrow - 1) * nrow/2 + (nrow * ncol * (ncol + 1)/2)$ where $nrow = ncol = ceil(sqrt(n))$. Using semi-1D systolic array, designs with 16, 32 and 48 PEs consume 512, 1600 and 2744 output channels respectively, which is 53%, 62% and 71% reduction in output channels. The performance of this method is same as a 1D systolic array.

5.1.2 Timeshared Channels

Each PE loads into its weight buffer a weight filter to compute an output channel. Since weights are loaded plate by plate, it takes as many iterations as the number of weight plates to load an entire weight filter. Then, each PE reads a feature plate from its preceding PE, writes the feature plate to its subsequent PE, and starts computation on the feature plate. When PE_1 reads plate 3 from PE_0, it is also writing plate 2 to PE_2 and performing computation with plate 1. However, when weight plates are being loaded into a particular PE, the feature plate read/write of all the PEs preceding it in systolic array gets stalled. By profiling we observed that the PEs at the beginning of the systolic array stall more on feature channel write while PEs at the end of the systolic array stall more on weight channel read. This inability to concurrently utilize weight channels and feature channels can be alleviated by a double weight buffer which allows concurrent execution of computation of particular output channel and weight load for next output channel. Section 5.2.2 describes this more in detail. A design without double weight buffers can eliminate weight channels entirely and use the same channels to feed PEs with features and weight.

Output channels are greatly underutilized in the systolic PE array. A PE generates one output for every K_hxK_c/VEC_SIZE number of feature plates. On average, in VGG-16 and C3D respectively, each PE generates an output for every 122 and 316 input feature plate. This shows that eliminating output channels and utilizing the

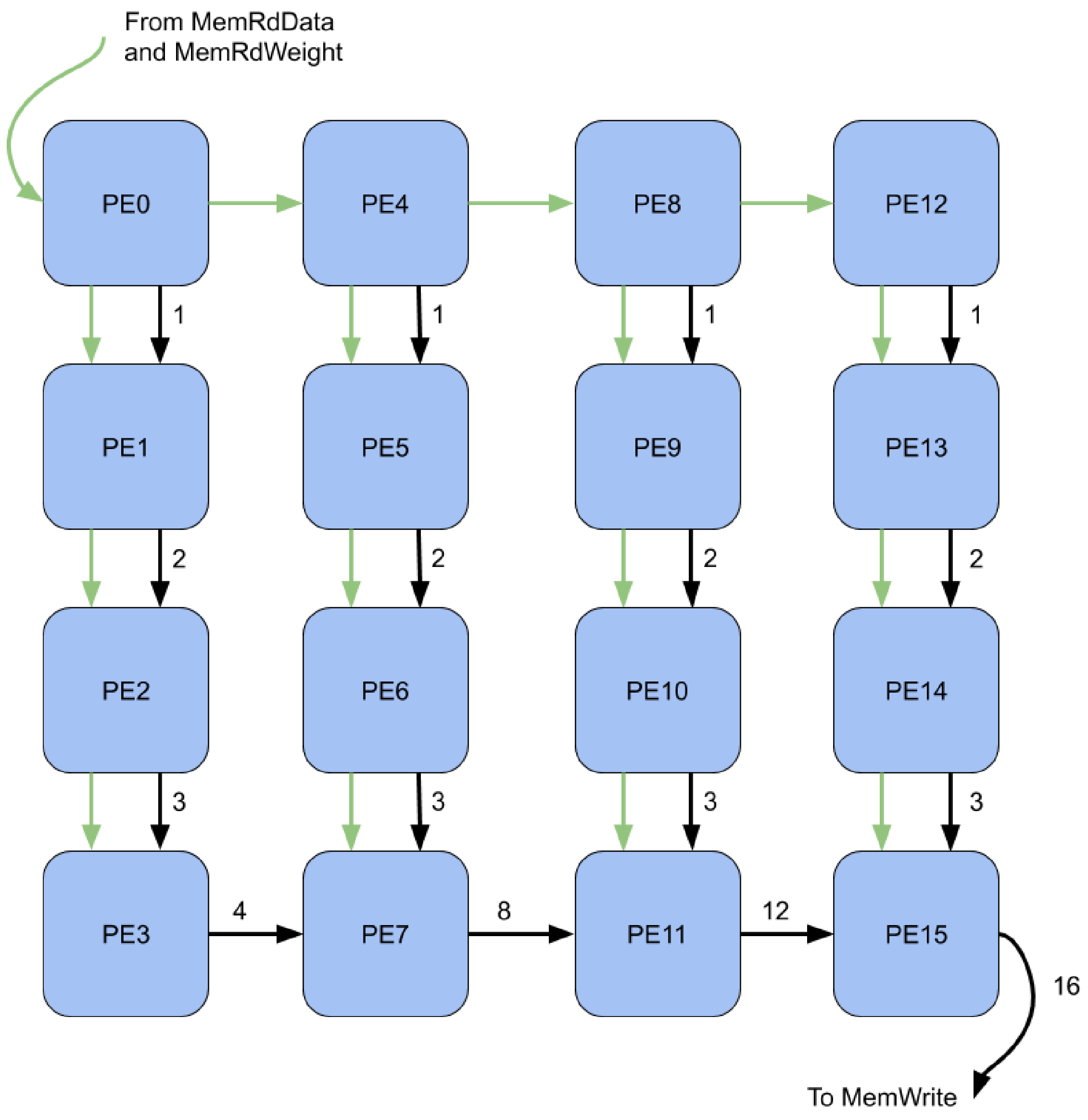

Figure 5.1: Semi-1D Systolic Array

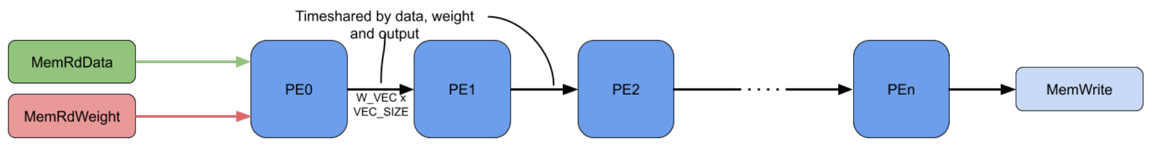

Figure 5.2: Timeshared Channels

same channels to transfer feature and output might reduce FFs usage significantly while having insignificant effects on performance.

These factors motivated us to implement a systolic array without weight channels and output channels. Figure 5.2 shows the architecture with timeshared channels. Each PE has a dedicated channel from/to its preceding/succeeding PE. This channel is used to transfer features, weights and outputs. In this approach MemReadWeight transfers weights only to the first PE_0. First, the weights are loaded into the PEs. Each PE in the systolic array either stores the weight plate in its buffer if it is the recipient or forwards the weight plate to its succeeding PE. Second, feature plates are transferred through the same channels. Each PE receives a feature plate, forwards it to the succeeding PE and also performs computation with it. Third, the output produced by PEs are transferred through the same channels. This approach resulted in an increase in single inference latency by 4% for C3D. However, the number of channels were reduced by 83% and the total FFs consumed by the design were reduced by 49%. The loss of performance is very small which makes this optimization a suitable candidate for area-performance trade-off.

5.2 Memory Optimization

CNNs involve data reuse due to which an onchip cache implementation can significantly reduce the number of DDR accesses. However, the number of BRAMs available in current FPGAs is not sufficient to hold the entire input feature or weight.

An optimal caching technique achieves maximum data reuse while consuming small amount of onchip memory. This section explains the double buffering implementations for feature maps and weights which enable reuse and hide DDR read time. Also, this section discusses optimizations to reduce the onchip memory consumption.

5.2.1 Double Data Buffer

Input features are reused when weights are moved across their width, height and frame dimensions. When a weight filter with width K_w is moved by *conv_stride* steps along input width, $K\_w - conv\_stride$ number of pixels overlap. An efficient memory module in MemReadData can cache the overlapping feature pixels and decrease the number of redundant accesses to DDR. Ma *et al.* (2018) implements shift register to enable reuse of features. Shift registers shift the current pixels conv_stride times and loads new pixels from DDR to produce the next chunk of data to be fed to the PE. DLA Aydonat *et al.* (2017) includes a fully on-chip implementation which might be suitable only for small networks like Alexnet. PipeCNN Wang *et al.* (2017) has implemented a double buffer policy which buffers a tile of input feature map and enables reuse inside the tile. Double buffer comprises a load-compute buffer pair to overlap loading of next tile with computation on current tile. First, we analyze the effectiveness of the default cache implemented along with LSU. Second, we implement double buffer policy as followed by PipeCNN and evaluate its inefficiency. Third, we optimize it for high performance.

The OpenCL SDK for Intel generates a burst-coalesced cached LSU for the global memory read in memReadData. MemReadData reads each feature plate as W_VEC separate lanes. Since the data is rearranged such that every adjacent lane is stored in subsequent memory addresses, W_VEC number of DDR read requests are coalesced. Size of the associated cache is 64KB with size of the cache line being 64 bytes and number of cache lines is 1024. Since this is a generic cache implementation, it is not very optimal for the data access pattern used in CNN. Moreover, there is no prefetching in this LSU cache. Since we know the data access pattern of CNN, applying this knowledge into the fetching logic can improve the performance of the LSU further.

We implement a load-compute buffer pair in MemReadData as shown in 5.3. The input feature map is split into tiles along width, height and frame dimension. Each tile can be used to compute multiple output pixels along width, height and frame dimension. For ex. an input tile of dimension W x H x F produces an output tile (W - K_w + 1) x (H - K_h +1) x (F - K_f +1) which is also the reuse factor of the inputs. The dimension of the tile is chosen such that there is maximum reuse while also ensuring it fits entirely in the onchip buffer. The load-compute buffer pair overlaps data load time from DDR and computation to a certain extent. While the current data tile in compute-buffer is used to feed PEs, the next data tile is loaded from DDR into load-buffer. Once this is completed, the buffers swap their purpose.

PipeCNN implements double buffer using a loop as follows:

```
int loop_limit = max(compute_loop_limit, load_loop_limit)
for(int i = 0; i <loop_limit; i++) {
      if(i < load_loop_limit)
1:          \\ read next input feature plate from DDR and load in buffer[0]
      if i $<$ compute\_loop\_limit:
2:        \\ read next input feature plate from buffer[1] and send to PE
}
```

Each iteration of the loop (1) reads a feature plate from DDR for the next data tile and stores it in load-buffer (2) reads a feature plate from compute buffer and writes

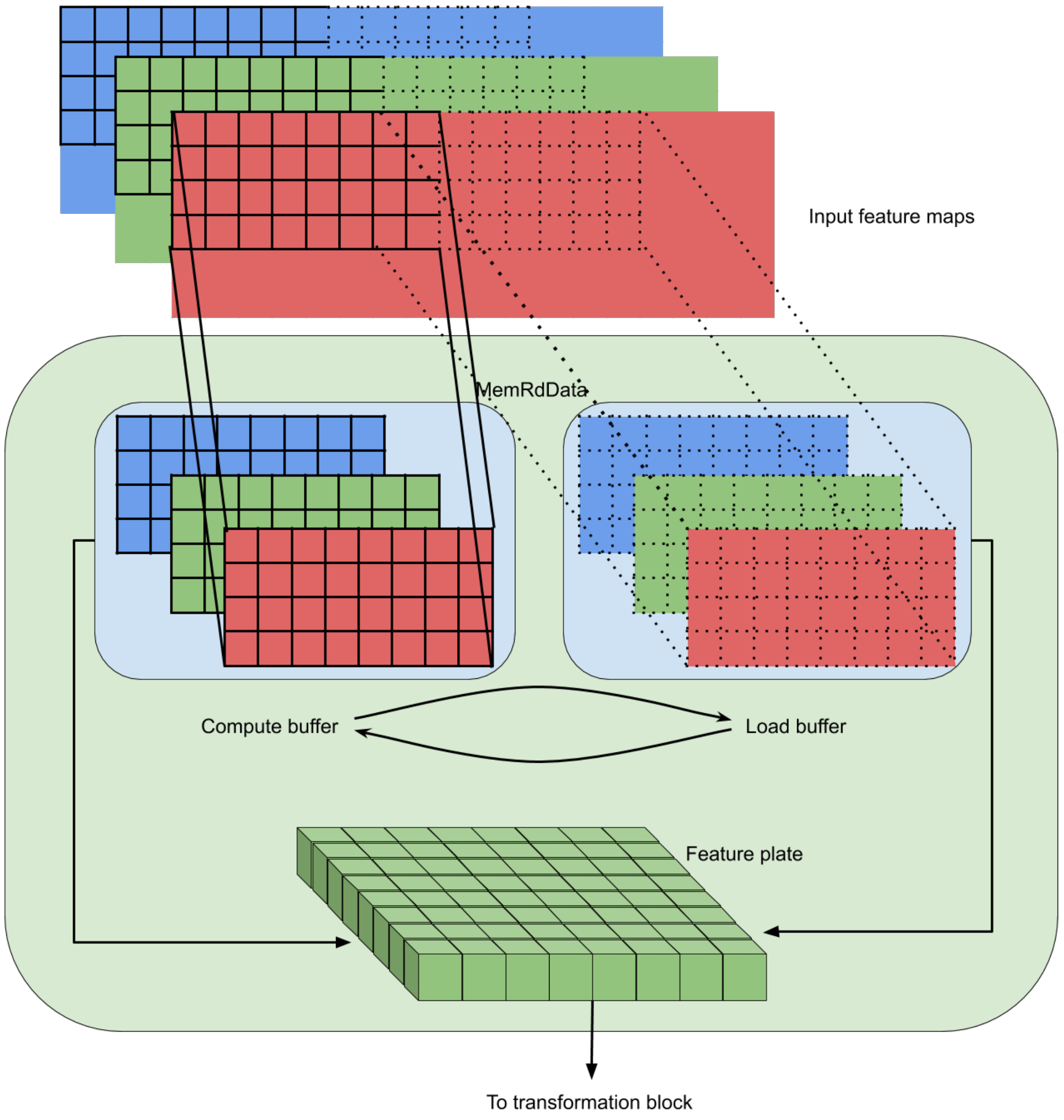

Figure 5.3: Double Data Buffer

to feature_channel connected to PE_0. Instruction (1) is executed load_loop_limit number of times which represents the number of plates read from DDR. Instruction (2) is executed compute_loop_limit number of times which represents the number of plates sent to PE. Since DDR read is the high latency operation, the performance of the loop is bound by the DDR read.

Usually, load_loop_limit and compute_loop_limit are not the same because of the reuse factor, and compute_loop_limit is generally greater than the load_loop_limit. The loop performs as many iterations as compute_loop_limit in most cases where DDR read access is made only during the first load_loop_limit iterations. We isolated high latency DDR read and low latency channel write by splitting these operations into separate kernels.

We implemented a kernel MemReadData_DDR which reads feature plates from the DDR and writes to a channel memreaddata_channel as shown in 5.4. The loop in memReadData reads features plate from memreaddata_channel, instead of DDR. Since the loop performs only channel read and write both of which are low latency operations, the overall performance of the loop improved. The loops in MemReadData_DDR and MemReadData are as follows:

```
// MemReadData_DDR
for(int i = 0; i < load_loop_limit; i++) {
     if(i < load_loop_limit)
1:         \\ read next input feature plate from DDR and write to
   memreaddata_channel
}
```

```
// MemReadData
int loop_limit = max(compute_loop_limit, load_loop_limit)
```

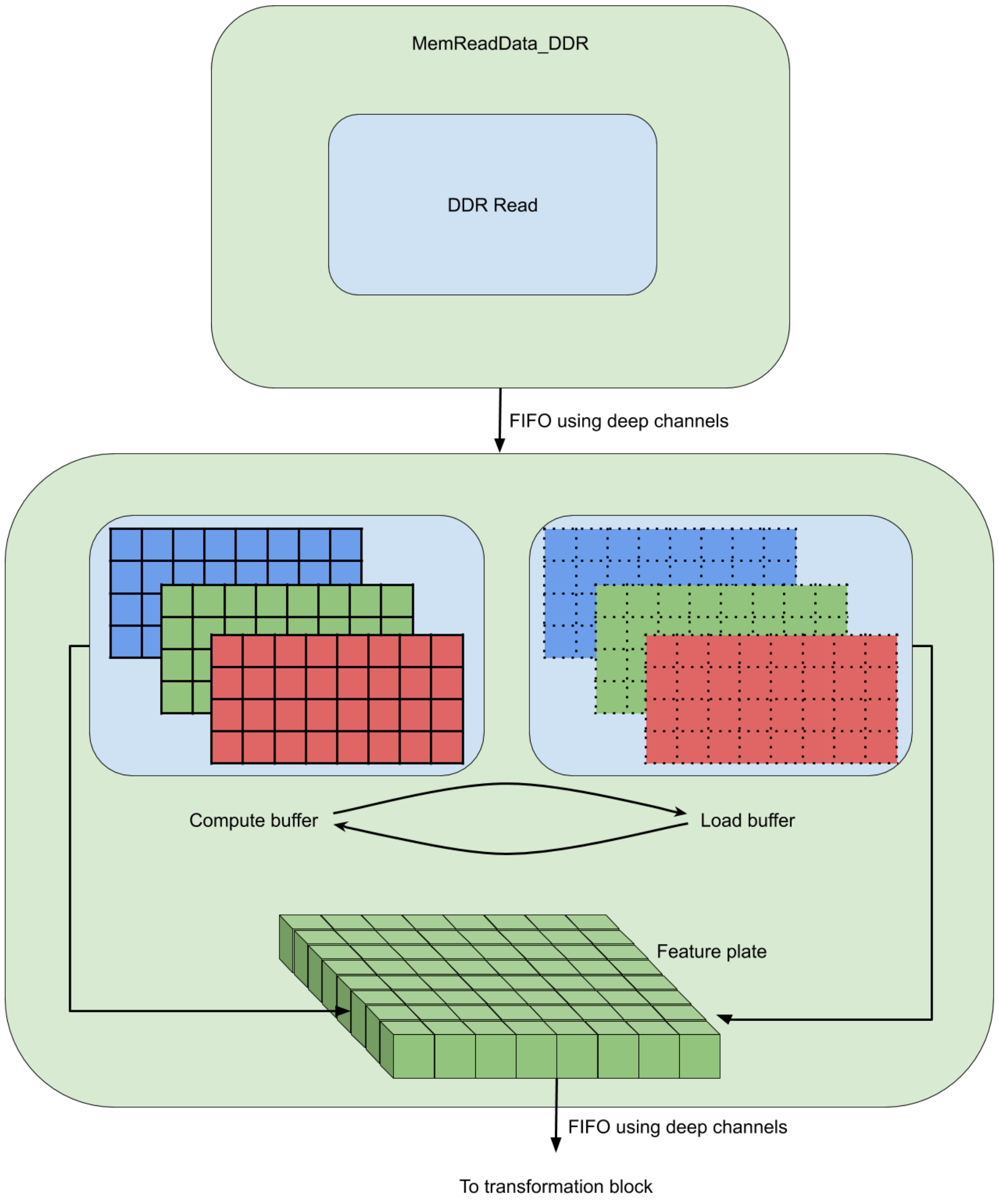

Figure 5.4: Double Data Buffer with DDR Read Isolation

```
for(int i = 0; i <loop_limit; i++) {
      if(i < load_loop_limit)
1:          \\ read next input feature plate from memreaddata\_channel and
    load in buffer[0]
      if i $<$ compute\_loop\_limit:
2:        \\ read next input feature plate from buffer[1] and send to PE
}
```

The DDR reads in memReadWeight and memReadData, which have variable latency based on the availability in cache, create stalls in the PE. To alleviate this problem we implemented a prefetching logic in memReadData with deep channel. We increased the depth of (1) data channel from memReadData to PE0, (2) weight channel from memReadWeight to PE0. This design improved the frequency of the design, and hence reduced the latency, by 7%.

5.2.2 Double Weight Buffer

The baseline architecture does not overlap weight loads with computation. Since the amount of weights in convolution layers is typically small for a network like VGG-16, the baseline architecture's performance does not suffer much for these networks. For a network like C3D, the convolution layer involves huge weights and hence the performance suffers by a large margin. To overlap weight loading and computation, we implemented double buffering policy for weights. While each PE is performing computation for a particular set of output channels using one weight buffer, weight for the next set of output channels is loaded into the other weight buffer. Every clock cycle each PE (1) reads a data plate to perform MAC operation with a weight plate from compute weight buffer for computing current output channel (2) reads a weight plate to either store in the load weight buffer for computing next output channel or forwards it to the next PE. Double weight buffers enabled the design to hide weight load time.

The above design performs multiple if..else condition checks on different variables inside the critical loop of PE. For a particular data plate and weight plate, the if..else clause in all the PEs will evaluate to the same result. Hence we perform all the if..else condition checks in PE0, store the boolean results in a 32-bit word where each bit represents a if..else result, and attach this condition-word to the data and weight plate. Each PE operates based on this bit value. This has increased the performance by appx. 5% (3ms for C3D).

5.2.3 Number of #PEs

The size of weight buffer inside each PE is kept large enough to accommodate the largest weight filter of the CNN model. CNNs with huge weight filters place a large requirement on the onchip memory. The total amount of memory consumed by weight buffers is given by the product of the number of PEs and the size of largest weight filter of the CNN. Reducing the number of PEs can lead into a design which consumes low onchip memory. However, as the number of PEs go down, the number of times entire data is read from DDR increases. For instance, if number of output channels is 64, a #PEs=32 design will take two rounds to compute. In each round, it loads all the PEs with weights, streams the entire data from DDR memory to perform computation. Whereas, a #PEs=16 design will perform 4 rounds and hence the input feature map is read 4 times from the DDR.

We evaluated designs with 4, 8, 16 and 32 PEs (VEC_SIZE adjusted accordingly to utilize same number of DSPs) and observed no significant difference in performance. Figure 5.5 shows the performance comparison of designs with different number of

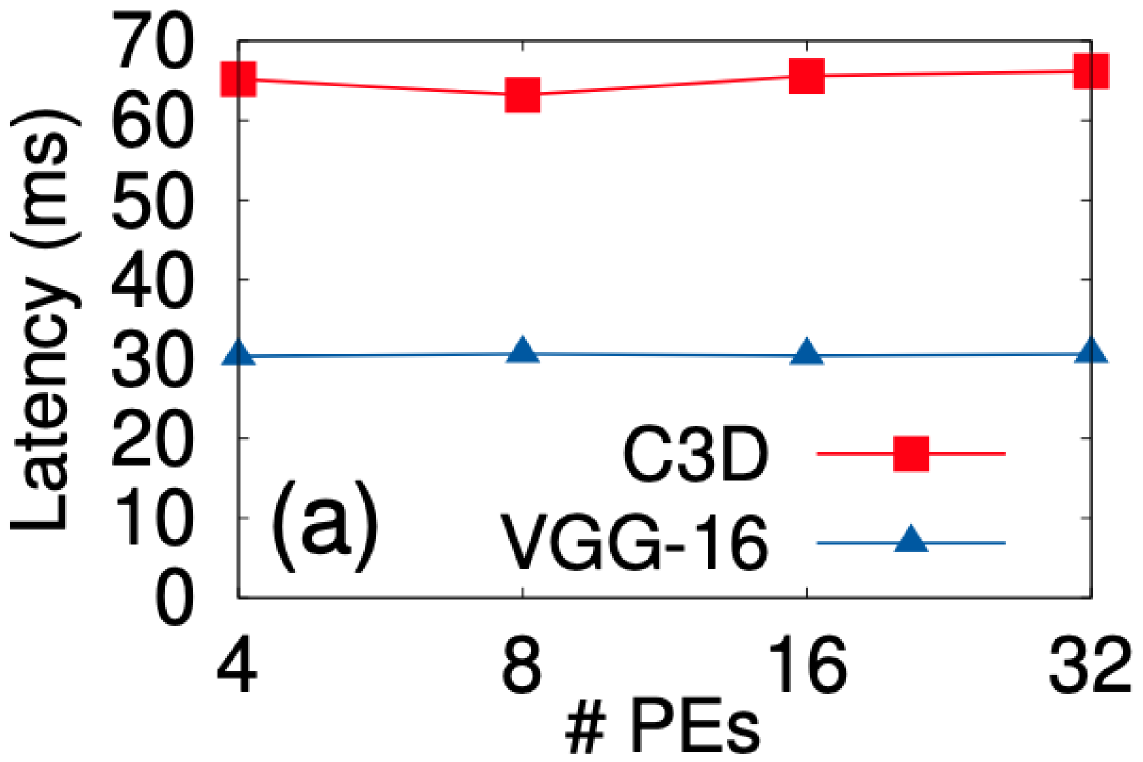

Figure 5.5: Impact of Number of PEs on Latency

PEs. To reduce onchip memory consumption, the number of PEs can be modified.

5.2.4 Convolution Layer Split

For CNNs with large weight filters, a smaller buffer can be allocated for each PE and the convolution layer (and also the weight) can be split into as many sub-layers as needed so that every sub-layer's weight fits completely inside the PE weight buffer. When K_c is the number of input channels and weight channels, and when weight doesn't fit in the on-chip memory of size WBS, the convolution layer is split into sub-layers each with $I_{csub} = I_c/ceil((K_h \times K_w \times K_c)\ /WBS)$ number of channels and the number of sub-layers will be I_c/I_{csub}. Each of these sub-layers will produce output with the same dimension as the original convolution layer. The updated new dimension of input is $I_w \times I_h \times I_{csub}$, weight is $K_w \times K_h \times K_{csub} \times m$. $I_c/I_{csub} - 1$ sum layers will be needed to accumulate the output results of all the sub-layers. Sum layer takes two volumes of data with the same dimension as the input and accumulates the corresponding elements to produce an output with the same dimension. The logic for sum layer is implemented in MemWrite while writing each sub-layer output to DDR.

Why split along channel dimension?: The amount of data read from DDR will be the same as usual convolution layer without splitting. The first set of I_{csub} channels of data and weight will be read for the first sub-layer, second set of $I_c sub$ channels for the second sub-layer and so on. If the weights are split along other dimensions, we have to access the data as many times as the number of sublayers.

In C3D, 2 layers were split into 2 sublayers each and 3 layers were split into 3 sublayers each, (totally 19 layers) the execution time increased by less than 5%

5.3 Frequency Optimization

5.3.1 Systolic Weight Channels

Convolution layers involve great opportunity for data and weight reuse. Our CNN accelerator exploits data reuse by transferring the same data plate to each PE. Each PE works on a single weight to produce a particular output channel and hence there is no sharing of weights between PEs. Weight reuse is exploited by storing weight in the local weight buffer inside each PE. Connecting memReadData and memReadWeight modules to each PE results in a high fan-out design with low frequency. We adopted a systolic array of PEs, as shown in 5.6, where the memReadData transfers data only to the first PE in the array, and every PE in the array uses the data and forwards it to the next PE. Since PEs do not share weights with each other, we connected memReadWeight module individually to every PE. However this led to

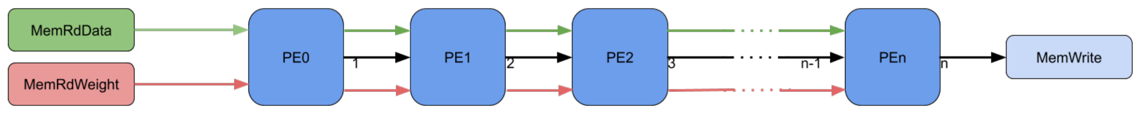

Figure 5.6: Systolic Weight Channels

a low frequency design. The design achieves 17% increase in frequency when the weights are also forwarded in systolic manner. In this approach, memReadWeight transfers $NUM\_WEIGHT\_PLATES \times NUM\_PEs$ weight plates to PE_0. Each PE stores its corresponding NUM_WEIGHT_PLATES number of weight plates in its buffer and forwards the rest to subsequent PE.

5.4 Fully Connected Layers

In fully connected layers (FC) the dimensions of input data and weight are the same. This calls for a different approach to execute FC layers. First, we map the FC layers onto the existing convolution systolic PE array. Second, we add logic to the existing PE array to improve the efficiency of FC layers. Third, we discuss the necessity of pipelined architecture.

A fully-connected (FC) layer differs from a convolution layer in few aspects like data reuse, computation to memory ratio. FC layers can be run on the convolution PE with only a few modifications to the logic. FC layers do not need Winograd transformation and hence the input data being fed to the PEs will bypass the Winograd's transformation module. The PEs are designed with W_VEC number of MAC units, where each MAC unit accepts VEC_SIZE number of input pixels. For a 1x1 FC layer to fully utilize all the MAC units, we consider the dimension of FC layer input and weight as W_VEC x 1 x (inp_ch/W_VEC) and W_VEC x 1 x (inp_ch/W_VEC) x outp_ch respectively, where usual dimensions are 1 x 1 x inp_ch and 1 x 1 x inp_ch x outp_ch. This does not need any rearrangement of input feature or weight since

ip/op	256	512	1024	2048	4096
2048	0.27|0.34|0.04	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	10.9|10.7|10.6

Table 5.1: Comparison of Latency on 32 PEs, 1 PE and Theoretical

both these dimensions have the same storage representation on physical memory. The weight is loaded into the PE buffer and data is streamed through systolic channel.

In a convolution layer, a single weight being loaded into PE buffer is reused several times to compute output pixels across the input width, height and frame. There is no such reuse in FC layers which makes loading weights into PE buffer as an unnecessary step. DLA Aydonat *et al.* (2017) has worked around this by loading input features in the buffer inside PE, which is usually used to store weight for convolution layers. To enable weight reuse, DLA batches multiple input images to execute FC layer on them. The buffer in each PE stores a different input while weights are being streamed through PEs in systolic channel. This enables weight reuse since all the PEs are using the same weight to compute the output of a different input image. However, since our design is focused on minimizing the latency of single inference we do not perform batching. The same input is loaded into the buffer inside each PE. memReadWeight module streams weight to all the PEs with which MAC operations are performed. These weights are dropped after computation since there is no reuse.

For a given FC layer, it takes insignificant time to load input features into the PE buffers. Streaming the weights from memReadWeight to PE is entirely overlapped

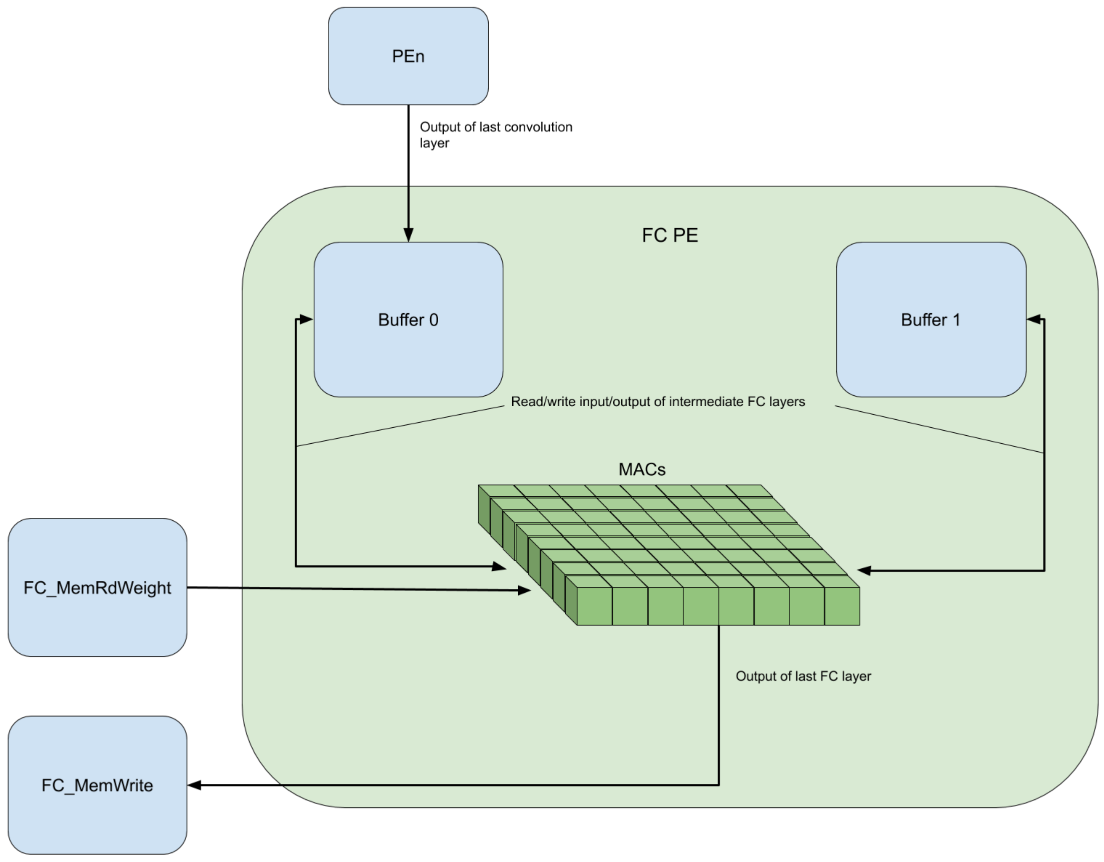
PEn
Output of last convolution layer
FC PE
Buffer 0
Buffer 1
Read/write input/output of intermediate FC layers
MACs
FC_MemRdWeight
Output of last FC layer
FC_MemWrite

Figure 5.7: Dedicated PE for Fully Connected Layers

with the computation. Because of this, we observe that the FC layer is memory bound. Since memReadWeight loads only one PE at a time, the rest of the PEs are idle which results in poor PE utilization. We conducted an experiment with only one PE. This design provided almost the same performance as the systolic PE array, which again proves that, for FC layers, only one PE is being utilized at a time in systolic PE array. Table 5.1 shows how a single PE design and 32 PE design achieve almost same performance as the theoretical maximum.

It is viable to overlap compute-bound convolution layers and memory-bound FC layers to efficiently utilize the DDR bandwidth and reduce PE idle cycles. While the FC layers of an inference are being carried out, convolution layers of the next inference can be performed concurrently. We designed a dedicated FC module, as shown in 5.7 which consists of (1) one PE, FC_PE, whose structure is similar to the PE in systolic array (2) FC_memRead kernel to stream weights to the FC_PE. FC_PE has a buffer to store the input feature. It performs MAC operations on the weight being streamed from FC_memRead kernel. The convolution module and FC module are independent of each other. While the latency of a single inference remains the same, the number of inferences per second increases from 30 to 43, and 15 to 16 for VGG-16 and C3D respectively.

Chapter 6

RESULTS

To prove the flexibility of the design with different neural networks and also perform comparisons with related works, we executed the 2D CNNs VGG-16, Alexnet, Resnet, and the 3D CNN C3D on our design. All the experiments were run on a set of Intel Fog Reference Design units , each equipped with 2 Nallatech p385a FPGA acceleration cards. Each FPGA card has an Intel Arria 10 FPGA and DDR3 SDRAM of size 8GB. The host machines have an Intel Xeon CPU E5-1275 and DDR memory of size 32GB. The OpenCL kernels were compiled using Intel FPGA SDK for OpenCL (version 19.1) with Nallatech p385a_sch_ax115 board support packages (BSP). We report the single inference latency, which is the time taken to process all the layers of the CNN model. We report input latency, which is the time taken by the convolution-fc processor pipeline to accept a new input. Throughput, which is a measure of the input latency, is reported as the number of inferences per second (inference/s) that the accelerator can perform.

Table 6.1 compares the performance of baseline architecture and the improvements made by this thesis. Due to the contributions of this work, the design achieves 27% increase in operating frequency resulting in 24% and 31% reduction in single inference latency, and 105% and 58% increase in throughput (inferences per second) for VGG-16 and C3D respectively. Table 6.2 compares our resource utilization and performance with the related works. Our design achieves 23.5ms single inference latency for VGG-16 which is better than the related works on FPGA (Ma *et al.* (2018), Suda *et al.* (2016), Wang *et al.* (2017), Zhang *et al.* (2018)), and on CPU and GPU Zhang *et al.* (2018). DLA Aydonat *et al.* (2017) caches the intermediate feature maps fully onchip

Design	Baseline		This thesis	
CNN Model	VGG-16	C3D	VGG-16	C3D
FPGA	Arria 10		Arria 10	
Clock Frq (MHz)	203		258	
Precision (bits)	8		8	
Latency/Inference(ms)	31.2	66.6	23.52	45.6
Input latency (ms)	31.2	66.6	15.1	41.7
Throughput (GOPS)	831		1056	
Throughput (Inferences/s)	32.02	15.01	66.2	23.98
DSP Util. (Used /Total)	1.1K / 1.5K		1.2K / 1.5K	
BRAM Util. (Used / Total)	1.4K / 2.7K		1.7K / 2.7K	
LUT Util. (Used /Total)	328K / 854K		330K / 854K	
FF Util. (Used /Total)	733K / 1708K		743K / 1708K	

Table 6.1: Performance Comparison of Baseline Architecture and Optimized Architecture

and batches multiple inferences in fully-connected layers. Since our goal is low latency single inference, we do not perform batch processing. And our design does not cache intermediate feature maps in the onchip memory since our design is targeted for large networks. Due to these reasons our design has higher latency than DLA for Alexnet.

Table 6.3 presents the effectiveness of our accelerator for the 3D CNN C3D and compares the resource utilization and performance with related works. We achieve from 2.1 to 2.7 times improvement in throughput compared with related works.

Design	Zhang	Ma	Ma	Suda	Zhang	Wang	Aydonat	Ours
CNN Model	Alex Net	VGG-16	ResNet-5	VGG-16	VGG-16	Alex Net | VGG-16	Alex Net | VGG-16	VGG-16 | Alex Net | Res Net
FPGA	Virtex 480t	Arria 10	Arria 10	Stratix V	KU060	Arria 10	Arria 10	Arria 10
Clock Frq (MHz)	100	200	150	120	200	190	303|215	258
Precision (bits)	32	16	16	16	16|8	8	8|8	8
Latency /Img(ms)	43.23	43.2	27.2	117.8	101.15|25.3	74.3|225	1|37	23.52|6.9|16.44
Throughput (GOPS)	61.62	715.9	285.07	262.9	266|1.17K	N/A|N/A	1300|990	1056
Throughput (Img/s)	23.13	23.14	36.7	8.48	9.88|39.52	4.44|13.45	1000|27	66.2|282.43|21
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 | 0.5K / 1.5K	1.5K / 1.5K | 1.5K / 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 | N/A	2.4K / 2.7K | 1K / 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	N/A | 278K/ 854K	330K / 854K
FF Util. (Used /Total)	205K / 607K	N/A	N/A	N/A	80K / 727K | 140K/ 700K	N/A | N/A	N/A | 725K/ 1708K	743K / 1708K

Table 6.2: Performance Comparison of State-of-the-art 2D CNN Accelerators

Design	Shen *et al.* (2018)	Liu *et al.* (2019)	Ours
CNN Model	C3D	C3D	C3D
FPGA	VC709	VC709	VC709
Clock Frq (MHz)	N/A	120	258
Precision (bits)	N/A	N/A	8
Latency/Inference(ms)	89.4	115.5	45.6
Input latency (ms)	89.4	115.5	41.7
Throughput (GOPS)	427.5	667.7	1056
Throughput (Inferences/s)	11.18	8.65	23.98
DSP Util. (Used /Total)	2.2K / 2.8K	1.5K / 1.5K	1.2K / 1.5K
BRAM Util. (Used / Total)	1K / 2K	1.5K / 2.7K	1.7K / 2.7K
LUT Util. (Used /Total)	186K / 303K	278K/ 854K	330K / 854K
FF Util. (Used /Total)	205K / 607K	80K / 727K	743K / 1708K

Table 6.3: Performance Comparison of State-of-the-art 3D CNN Accelerators

Chapter 7

CONCLUSION

CNNs are extensively finding applications in latency-constrained environments with power restrictions, due to which FPGA acceleration is gaining attraction. We present a low-latency FPGA acceleration solution generic enough to support different CNN models. First, we designed an architecture in OpenCL to perform the entire inference phase on FPGA. Then, the limitations posed by high fan-out while scaling-up were identified and addressed using a systolic array of processing elements. Improvements were made to reduce the resource consumption of the systolic array without sacrificing performance. Optimizations were performed for further reduction in resource consumption with an acceptable loss of performance. Optimizations were performed to overlap memory accesses and computation to achieve better performance. Then, the arithmetic complexity was reduced by using Winograd transformation. The performance-area trade-off and the performance-memory trade-off were studied. The throughput of the accelerator was significantly improved by a pipeline of convolution processors and a dedicated FC processor. Further extensions were added to the design to support 3D convolutions and perform video processing. These optimizations synergistically helped our design achieve high performance.

REFERENCES

Abadi, M., P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning", in "12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)", pp. 265–283 (2016).

Arulkumaran, K., M. P. Deisenroth, M. Brundage and A. A. Bharath, "Deep reinforcement learning: A brief survey", IEEE Signal Processing Magazine **34**, 6, 26–38 (2017).

Aydonat, U., S. O'Connell, D. Capalija, A. C. Ling and G. R. Chiu, "An opencl^{TM} deep learning accelerator on arria 10", in "Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays", pp. 55–64 (2017).

Biookaghazadeh, S., M. Zhao and F. Ren, "Are fpgas suitable for edge computing?", in "{USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 18)", (2018).

Boutros, A., S. Yazdanshenas and V. Betz, "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 (TRETS) **11**, 3, 1–23 (2018).

Hegde, K., R. Agrawal, Y. Yao and C. W. Fletcher, "Morph: Flexible acceleration for 3d cnn-based video understanding", in "2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)", pp. 933–946 (IEEE, 2018).

Ji, S., W. Xu, M. Yang and K. Yu, "3d convolutional neural networks for human action recognition", IEEE transactions on pattern analysis and machine intelligence **35**, 1, 221–231 (2012).

Jia, Y., E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding", in "Proceedings of the 22nd ACM international conference on Multimedia", pp. 675–678 (2014).

Lavin, A. and S. Gray, "Fast algorithms for convolutional neural networks", in "Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition", pp. 4013–4021 (2016).

Litjens, G., T. Kooi, B. E. Bejnordi, A. A. A. Setio, F. Ciompi, M. Ghafoorian, J. A. Van Der Laak, B. Van Ginneken and C. I. Sánchez, "A survey on deep learning in medical image analysis", Medical image analysis **42**, 60–88 (2017).

Liu, Z., P. Chow, J. Xu, J. Jiang, Y. Dou and J. Zhou, "A uniform architecture design for accelerating 2d and 3d cnns on fpgas", Electronics **8**, 1, 65 (2019).

Ma, Y., Y. Cao, S. Vrudhula and J.-s. Seo, "Optimizing the convolution operation to accelerate deep neural networks on fpga", IEEE Transactions on Very Large Scale Integration (VLSI) Systems **26**, 7, 1354–1367 (2018).

Maturana, D. and S. Scherer, "Voxnet: A 3d convolutional neural network for real-time object recognition", in "2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)", pp. 922–928 (IEEE, 2015).

Shen, J., Y. Huang, Z. Wang, Y. Qiao, M. Wen and C. Zhang, "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", pp. 97–106 (2018).

Suda, N., V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo and Y. 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", pp. 16–25 (2016).

Sun, L., K. Jia, D.-Y. Yeung and B. E. Shi, "Human action recognition using factorized spatio-temporal convolutional networks", in "Proceedings of the IEEE international conference on computer vision", pp. 4597–4605 (2015).

Tran, D., L. Bourdev, R. Fergus, L. Torresani and M. Paluri, "Learning spatiotemporal features with 3d convolutional networks", in "Proceedings of the IEEE international conference on computer vision", pp. 4489–4497 (2015).

Wang, D., K. Xu and D. Jiang, "Pipecnn: An opencl-based open-source fpga accelerator for convolution neural networks", in "2017 International Conference on Field Programmable Technology (ICFPT)", pp. 279–282 (IEEE, 2017).

Zhang, C., P. Li, G. Sun, Y. Guan, B. Xiao and J. 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", pp. 161–170 (2015).

Zhang, C., G. Sun, Z. Fang, P. Zhou, P. Pan and J. Cong, "Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **38**, 11, 2072–2085 (2018).