DESIGN AND PERFORMANCE ANALYSIS OF HARDWARE ACCELERATOR FOR DEEP NEURAL NETWORK IN HETEROGENEOUS PLATFORM.

by

Md Syadus Sefat, B.S.

A thesis submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Engineering
August 2018

Committee Members:

Semih Aslan, Chair

Apan Qasem, Co-chair

Bahram Asiabanpour

Damian Valles

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

Duplication Permission

DEDICATION

Dedicated to my parents whose love and sacrifice have brought me here.

ACKNOWLEDGEMENTS

I would like to express my gratitude towards Dr. Semih Aslan and Dr. Apan Qasem for their continuous support, guidance, and encouragement throughout the course of this thesis. I would also like to thank Dr. Damian Valles and Dr. Bahram Asiabanpour for agreeing to be in the thesis committee and for their valuable comments. I would like to express my gratitude to the CAPI team, IBM Austin, for their constant support. I would thank specially to JT Kellington, Thomas Fuchs, Curt Wollbrink of IBM Austin and Mark Paluszkiewicz of Xilinx for their guidance and support throughout the development of the accelerator.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

ABSTRACT

This thesis describes a new flexible approach to implementing energy-efficient DNN accelerator on FPGAs. Our design leverages the Coherent Accelerator Processor Interface (CAPI) which provides a cache-coherent view of system memory to attached accelerators. Computational kernels are accelerated on a CAPI-supported Kintex FPGA board. Our implementation bypasses the need for device driver code and significantly reduces the communication and I/O transfer overhead. To improve the performance of the entire application, we propose a collaborative model of execution in which the control of the data flow within the accelerator is kept independent, freeing-up CPU cores to work on other parts of the application. For further performance enhancements, we propose a technique to exploit data locality in the cache, situated in the CAPI Power Service Layer (PSL). Finally, we develop a resource-conscious implementation for more efficient utilization of resources and improved scalability. Compared with the previous work, our architecture achieves both improved performance and better power efficiency.

I. INTRODUCTION

1.1 Motivation

In recent years, the rapid growth of data in the digital world is occurring at an exponential rate. This increase in data (i.e., video, image, speech) from numerous sources such as social media is creating the need for extracting knowledge base from the data by using data analytics with Machine Learning (ML) tools. Among the different ML algorithms, deep learning algorithms are achieving the state-of-art in solving many real-time tasks such as image detection and classification, image recognition and tagging, natural language recognition, pattern recognition, fraud detection, targeted marketing, autonomous driving, intelligent gaming, fraud detection and monitoring, and financial forecasting [5] [6] [7]. Because of the ability to train and classify data with high accuracy, the deep neural network (DNN) methods have been demonstrated to be widely used methods for various applications including image classification, face detection, video analysis, speech recognition, and document processing [8]. Optimized DNN algorithms are becoming major components in many modern applications and they are attracting enthusiastic interest from both academia [5] [7] [9] and industry like Google [10], Facebook [11], and Baidu [12].

To process a sample of data (e.g., image), DNN requires to work on millions of parameters and billions of operations [13] [14]. As DNN includes one or more fully connected layers, it is a massive memory and computation intensive workload for the Central Processing Unit (CPU). To accelerate the computation process, numerous software platforms have been released, primarily targeting the power-hungry CPUs. Graphics Processing Units (GPUs), which are designed with higher throughput and memory bandwidth, generally require a considerable amount of power and can sometimes be constrained by memory capacity. Memory constraint and power requirement are critical issues for DNNs, particularly for those running

inferences tasks on edge devices. In this context, re-configurable FPGA hardware accelerators offer a potential alternative platform in the existing DNN system which exhibits a tunable balance among performance, power consumption, and programmability [15]. Hardware acceleration of neural networks is a promising research direction [16, 14, 2]. Although currently FPGAs cannot deliver the raw compute power demanded by large-scale deep learning applications, accelerated implementations generally yield better energy efficiency in the form of improved performance/watt [17].

1.2 Challenges

Hardware acceleration of DNNs poses significant challenges. Real-world DNNs are constructed with millions of model parameters requiring hundreds of megabytes of storage for each layer, which far exceeds the capabilities of current FPGAs. As a result, accelerated implementations incur high I/O overhead and performance is often dominated by data transfer time over a low-bandwidth I/O bus such as PCIe. In traditional HW-SW collaboration paradigm, the accelerator is attached as a memory-mapped I/O device. A device driver performs the virtual to physical address translation and delivers the addresses of the pinned kernel buffer to the accelerator. The developers need to develop a device driver according to the hardware specification.

Another major obstacle with FPGA acceleration, CNN or otherwise, is the time to development. There are two methods of designing an accelerator in FPGA. First one is designing an accelerator using a traditional Register Transfer Level (RTL) tool and the second one is using a High-Level Synthesis (HLS) tool. Designing in RTL is a tedious process but gives greater flexibility to the developers for creating lower-level decisions, which maximizes the efficiency and performance of the designed architecture. On the other hand, designing an accelerator in HLS is easier and it is less prone to errors.

Recent introduction of high-level synthesis tools, such as OpenCL and Vivado HLS has increased FPGA programmability with respect to data path represen-

tation. Nonetheless, a custom FPGAs still requires writing device driver code to access memory-mapped I/O and communicate with the CPU, which can significantly add to the development time.

In traditional systems, when an accelerator finishes its computation, it writes the data in the kernel space. Then a device driver copies the data from kernel space to the user space, generates a pointer to the data and passes it to the application. Thus, the same data is copied *twice*. The recent development in OpenCL environment provides the runtime platform for the FPGA which manages the SW-HW communication. However,in that case the underlying semantics in the FPGA needs to be those of the OpenCL. This does not give a good flexibility to incorporate customized hardware design within the FPGA.

1.3 Contributions

This thesis represents the design and implementation of a complete hardware accelerator solution for Deep Neural Network for training as well as inference phase. Our implementation is realized on a (1) System on Chip (SoC) FPGA device and on a (2) coherent FPGA which utilizes IBM's Coherent Accelerator Processor Interface (CAPI). The CAPI technology, recently introduced by IBM through its OpenPower initiative, enables coherent connection to custom acceleration engines within a heterogeneous compute unit. CAPI adds an Effective-to-Real-Address Translator (ERAT) within the Power Service Layer (PSL) that translates the addresses, eliminating the need for address translation at the FPGA end. Furthermore, in CAPI, the pointer to the user space data is sent from the application directly to the FPGA, thereby avoiding any extraneous copying of data from kernel to user space. We develop an efficient and flexible DNN implementation that leverages these CAPI features and bypasses the need for device driver code and significantly reduces the communication and I/O transfer overhead.

In both implementations, a software implements the generic neural network model then accelerates the computation by off-loading the computation to the FPGA hardware accelerator. Currently, both the RTL and HLS methods have

been used for designing the hardware accelerator separately. Since, among all the computations, the matrix multiplication calculation requires the most computational resources, on both heterogeneous platforms, hardware design is mostly focused on vector product calculations. Rest of the calculations have been performed on CPUs for both SoC and CAPI machines. Experimental evaluation on the two platforms show that implementation with coherent accelerators can not only yield significant performance improvements but also produce higher performance/watt.

To summarize, the main contributions of the thesis include:

- New flexible approach to implement DNN in a heterogeneous system
- A new hardware architecture design for DNN to leverage the Coherent Accelerator Processor Interface which provides a cache-coherent view of the system memory. To the best of our knowledge, this is the first such design
- Optimization of the hardware unit to implement a resource-conscious design for matrix-multiply on the FPGA
- Design of a batched computational unit to work with large weight matrices of DNN
- Optimize the hardware to work with weight matrices of any size
- A DNN solution that focuses on multiple computational kernels including vector product and activation layers
- Implementation of a general software-hardware framework that will enable many further optimizations

1.4 Thesis Overview

The thesis is presented in several chapters. Chapter 2 introduces neural networks and presents essential background on POWER8 CAPI architecture, Zynq SoC, and HLS. Chapter 3 explores related works and finds out the scope of contribution in DNN accelerator design. Chapter 4 describes the methodology for implementing software hardware co-work framework. The section also describes

our DNN implementation for Zynq SoC. Chapter 5 describes the hardware architecture design for AFU in CAPI system. Our core hardware design for matrix multiplication is described on chapter 5. Chapter 6 introduces resource-conscious and optimized hardware designs. Chapter 6 also talks about our approach towards batched computation. Chapter 7 evaluates the performance of our hardware architecture. Chapter 9 concludes the thesis and points out some future scopes.

II. PREVIOUS WORK

2.1 Evaluation of Neural Network Methods

Being inspired by nature such as biology, the human brain, etc., researchers have developed AI and ML techniques such as artificial neural network, evolutionary algorithms and cellular automata [18]. Among all the biology-inspired algorithms, CNNs are most successful regarding accuracy in classification. An experiment in neurobiology started the history of the CNN where, Hubel and Wiesel [19] found that neurons in the visual cortex at different stages activate some specific patterns and ignore others. Inspired by the findings of Hubel and Wiesel, Fukushima [20] proposed a model of multilayer neural network which succeeded in recognizing simple patterns. According to Schmidhuber [21], the first feedforward network that was successful in training neural network was developed by Lvakhnenko and Lapa in 1966. The first kind of neural networks were feedforward type, where information only flowed in the forward direction, from the output of one layer to input of next layer. Subsequently, researchers started using the multilayer neural network, using unsupervised learning approach through backpropagation [22].

Until mid-2000s, neural networks ware not heavily used in ML algorithms, as there were no good optimization tools available to minimize the error. It has been found that the elementary optimization tool, the gradient descent algorithm, performs really poorly in minimizing the error rate [23]. The accuracy of multilayer network increased when statistical methods like support vector machine were used for optimization [24]. The early 2010s have seen a rapid development of DNN-based applications [25] with the work of Microsoft's speech recognition systems in 2011 [26] and AlexNet system for image recognition in 2012 [27]. One of the factors to confluence the success of deep learning is believed to be the availability of compute capacity [25]. As a result of the current on-going research, there are several open-source frameworks for DNN that have been released by researchers

such as TensorFlow [28], Caffe [29], Theano [30], Torch [31], CNTK [32], etc. Among these frameworks, TensorFlow is the most popular one, which implements the DNN algorithm in C/C++ and python.

2.2 Acceleration of Deep Neural Networks

DNN, being a computationally intensive algorithm, makes FPGA as the best candidate to accelerate the computation. Dang and Skadron have shown that for a specific type of data mining application, frequent itemset mining (FIM), FPGA gets 3.2x speedup over a six-core CPU and also shows better energy efficiency [33]. Moreover, the contributions in [34], [35], [36] distributed the workload in FPGA and thus accelerated various algorithms such as KNN, K-means, SVM and produced higher efficiency.

Mahajan *et al.* proposed TABLA [37], a template-based framework for accelerating ML algorithms. They proposed to accelerate mathematical operations by creating and utilizing Verilog templates so that the high level of abstraction of those templates could be used in solving statistical operations by the programmer.

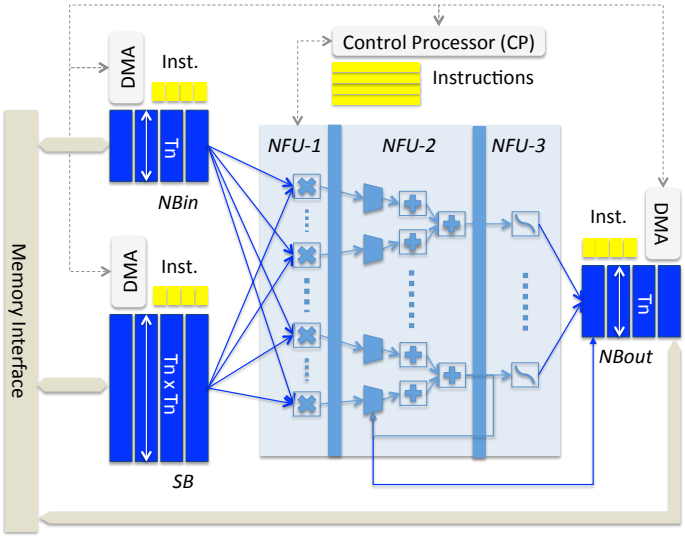


Figure 2.1: Neural Network Accelerator Architecture of DianNao [1]

In [1], the author presented an ASIC design for neural network acceleration called *DianNao*. As seen in Figure 2.1, the main components of the ASIC ac-

celerator include an input buffer for input neurons (NBin), an output buffer for output neurons (NBout), a buffer for weight matrices (SB), a computational unit (NFU) and the Control process (CP). Computation in the NFU is performed at NFU layers in different stages. In this implementation, the NFU unit is designed to accumulate only 16 output neurons which calculates the sum of products of 16 inputs *(x)* with 16 weights values. The buffers are connected with DMA units to transfer the data to the off-chip DRAM. CP generates instruction sets for each layer of computation.

Sharma *et al.* proposed a framework called DnnWeaver that automatically generates a synthesizable accelerator by the Caffe framework for a specific pair of DNN and FPGA from some given combinations. The proposed Instruction Set Architecture (ISA) toolset ensures a better FPGA hardware memory utilization and data reuse [16].

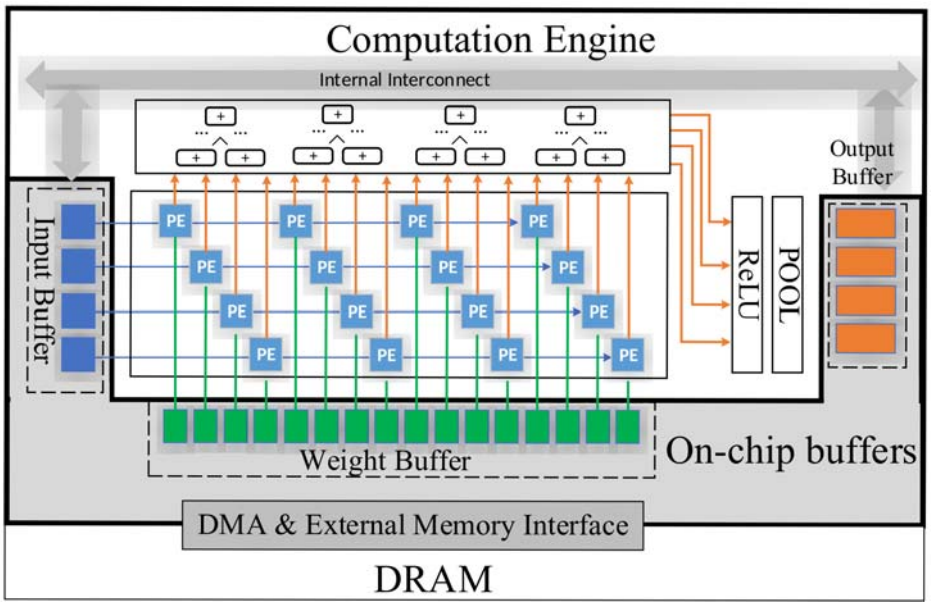


Figure 2.2: Neural Network Accelerator Architecture Caffeine [2]

Zhang *et al.* proposed Caffeine, a hardware/software co-designed library for accelerating the whole CNN on FPGA [2]. They introduced a uniform convolutional matrix-multiplication for the convolution layer as well as a fully connected layer. They focused on memory access organization. Figure 2.2 represents an overview

of Caffeine hardware architecture. The main components of the architecture are three buffer units: Input Buffer, Weight Buffer, Output Buffer, systolic arrays of PEs, ReLU unit, and POOL unit. Each PE is an arithmetic multiplication of input feature and corresponding weights. The ReLU of a max-pooling unit can be bypassed if there is no such layer in the neural network. Due to design complexity and FPGA resource constraints, they implemented only 32 PEs in the KU060 FPGA.

2.3 Distinction from Previous Work

Compared to previous research, the proposed approach in this thesis is different in methodology and perspective. The existing method in the previous research requires the design of Processing Engine (PE) that works as processing unit cores in the FPGA [16], [2]. These methods need to develop specific instructions set architectures for the PEs. The CPU maps the PEs for particular tasks and performs the task scheduling for accelerating the DNN computation. The process requires the design of device drivers for communication between the CPU and PEs. Then, the CPU communicates with the accelerator to control and manage data flow between the CPU and accelerator, and among PEs, thus the CPU remains busy while accelerator performs the computation operation.

In both implementations, instructions/configurations of PEs, input features, and weight matrices are kept in the FPGA device DRAM, and then from the DRAM, the data are transferred from DRAM to the on-chip BRAM and only then arithmetic operations are performed on the data. To get the data from the system memory to the DRAM of the FPGA data is first copied from the user space to the kernel space and then from kernel space, the data is transferred to the FPGA DRAM. Thus these conventional FPGA system creates I/O overhead for the DNN application.

There have been a plethora of previous works on FPGA deployment for the CNN computation. However, these works do not exploit the cache-coherent view of system memory with the attached accelerator to access the data. In [38], the au-

thors claimed that the CAPI technology simplifies the software design by enabling the FPGA to access to the main memory.

In this thesis work, instead of controlling the operations in the accelerator from the CPU, the control of the data flow within the accelerator is kept independent. In the proposed architecture, through the use of CAPI, CPU only needs to transfer the pointer to the user space data. The accelerator then initializes transferring the data from the system memory to the accelerator and upon receiving the data starts the computation. As the operation in the accelerator does not need control signals from the CPU, the proposed architecture reduces the workload of the CPU. Also, there is reduced I/O overhead as the system does not have to copy data from user space to kernel space, from kernel space to FPGA off-chip DRAM and from off-chip DRAM to off-chip BRAM.

III. BACKGROUND

3.1 Deep Neural Network (DNN)

The immense amount of increase in the data flow from numerous untapped sources has increased the extent of computation in today's digital computing systems and requires some analytics system to analyze these data to extract information. Most of the data analytic systems heavily depend on the ML algorithms. Any typical ML algorithm works in two phases. In the first phase, training phase, the system works on some known samples of the labeled dataset and forms a hypothesis iteratively. In the second phase, prediction phase, the system classifies a new test dataset using that hypothesis. Figure 3.1 shows the pictorial view of a typical neural network algorithm with fully connected layers. A typical deep learning network framework consists of several layers, each having some parameter matrices called weight matrices, which perform in sequence [39]. Each node of any layer is activated by some specific activation function such as sigmoid, tanh, ReLU (Rectified Linear Unit), Leaky ReLU, maxout and ELU (Exponential Linear Unit). After being triggered by the activation function, it produces an output. Each layer works on a specific feature from its previous layer to learn the feature. Each of the layers is initialized with the given weights and then goes through forward pass and backward pass. In forward propagation, a loss function is evaluated, an output is calculated from the layers which is fed to the next layer. In backward propagation, gradients of the parameter matrices are calculated (i.e., Jacobian matrix), weight matrices in different hidden layers are updated iteratively to minimize the loss function. In general, when a forward pass occurs a huge amount of data is cached in the memory, and during backpropagation, those values are used for updating the loss function and other parameters.

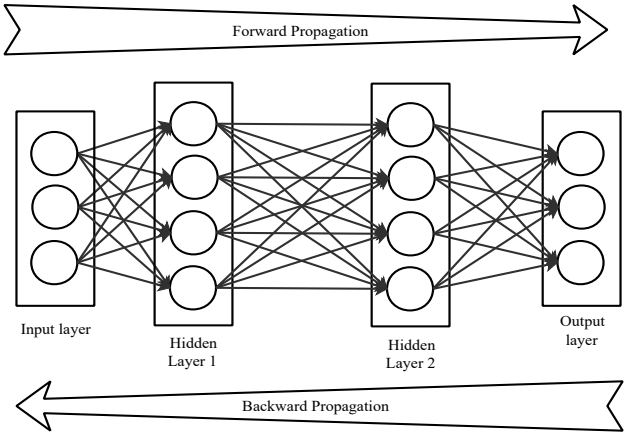


Figure 3.1: Neural Network Architecture

Let us consider a neural network with L layers. The weight matrix in the $(l-1)^{th}$ layer is denoted by w^l. The weight from the k^{th} neuron in the $(l-1)^{th}$ layer to the j^{th} neuron in the l^{th} layer is given by w^l_{jk}. The bias and activation in the l^{th} layer of j^{th} neuron is represented by b^l_j and a^l_j. The activation of the j^{th} neuron in l^{th} layer depends on the activation of $(l-1)^{th}$ layer and the weight matrix and is expressed as follows:

$$a^l_i = \sigma\left(\sum_k \omega^l_{j_k} a^{l-1} + b^l_j\right), \tag{3.1}$$

where, $\sigma(x)$ is the activation function. In vectorized notation form, equation 3.1 can be written as follows:

$$a^l = \sigma(w^l a^{l-1} + b^l). \tag{3.2}$$

While computing the activation function we calculate the intermediate quantity z where, in matrix form, $z^l \equiv w^l a^{l-1} + b^l$. We consider z^l to be the weighted input

to the l^{th} layer. In a layer l, z^l has the components:

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l. \tag{3.3}$$

The final target of a neural network is to find the weights and biases so that the output from the network approximates the training labels or the outputs. This target is achieved by calculation a cost function or loss function and minimizing that cost function.

$$C(w,b) \equiv \frac{1}{n} \sum_x \|y(x) - a\|^2. \tag{3.4}$$

Here, w is the notation for the weight matrices of the whole network, b is the notation for the biases of the whole network, n is the total number of output, and a is the output in vector format. The cost, $C\left(w,b\right)$ or the loss function is expressed in terms of mean squared error. To find out the minimum value of the loss function, gradient descent algorithm is applied where the gradient of different parameters in the network are repeatedly calculated, and the parameters are updated to the opposite direction of the loss function. For the weight matrices w and bias vectors b, the ∇C has the components $\partial C/\partial w^{(l)}$ and $\partial C/\partial b^{(l)}$. For a mini batch of data for m samples out of n samples, the updates of the parameters are performed by using the following rule:

$$w^l \rightarrow w^{l'} = w^l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w^l} \tag{3.5}$$

$$b^l \rightarrow b^{l'} = b^l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b^l}, \tag{3.6}$$

Here, η is the learning rate and the sums are performed within the mini batch samples. In a neural network, the network tries to understand how a change in weights and biases affects the cost function. In the forward pass, the weighted inputs and outputs are calculated and in backward pass, the gradient of the parameters in different layers are calculated. To find out the gradients of weights, biases and activation function, chain rule is applied from the rightmost layers to the leftmost layers. That is, it finds the partial derivatives of the weights and biases, $\partial C/\partial w^l_{jk}$, and $\partial C/\partial b^l_j$ from output layer (L) towards the first layer.

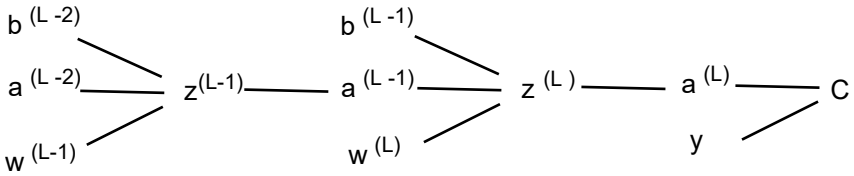


Figure 3.2: Gradient flow in layers

To demonstrate gradient flow in the chain rule, let us consider for a single training sample with a single neuron in the last layer, the cost function is defined by $C_o = \left(a^{(L)} - y\right)^2$ where y is the corresponding output and $a^{(L)} = \sigma(w^{(L)}a^{(L-1)} + b^{(L)})$ is activation in the last layer which is dependent on the activation of the previous layer and weight and bias of the layer L. To find out the change in the cost C with respect to w, chain rule is applied.

$$\frac{\partial C_o}{\partial w^{(L)}} = \frac{\partial C_o}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial w^{(L)}} \tag{3.7}$$

By knowing the cost function, derivative of C with respect to a is found by

$$\frac{\partial C_o}{\partial a^{(L)}} = 2\left(a^{(L)} - y\right) \tag{3.8}$$

$$\frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma' \tag{3.9}$$

$$\frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)} \tag{3.10}$$

Equation 3.9, expresses the the influence of w in layer L on the cost function for a single training sample. To find the overall influence for all training sample, average value is taken over all the partial derivatives of C with respect to w.

$$\frac{\partial C}{\partial w^{(L)}} = \frac{1}{n}\sum_{k=0}^{n-1}\frac{\partial C_k}{\partial w^{(L)}} \tag{3.11}$$

The left side of equation 3.11 is one of the components of the gradient vector ∇C. Similarly, we can find the influence of b on C as follows:

$$\frac{\partial C_o}{\partial b^{(L)}} = \frac{\partial C_o}{\partial a^{(L)}}\frac{\partial a^{(L)}}{\partial z^{(L)}}\frac{\partial z^{(L)}}{\partial b^{(L)}} \tag{3.12}$$

Here, $\partial z^{(L)}/\partial b^{(L)} = 1$

To find generalized equations for a network, let us consider layer L consists of n_L neurons and layer $L-1$ consists of n_{L-1} neurons.

$$C_o = \sum_{j=0}^{n_L-1}\left(a_j^{(L)} - y_j\right)^2 \tag{3.13}$$

$$z_j^{(L)} = \sum_k w_{jk}^{(L)} a_k^{(L-1)} + b_j^L. \tag{3.14}$$

$$\frac{\partial C_o}{\partial w_{jk}^{(L)}} = \frac{\partial C_o}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} \tag{3.15}$$

$$\frac{\partial C_o}{\partial a_k^{(L-1)}} = \sum_{j=0}^{n_L-1} \frac{\partial C_o}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial a_k^{(L-1)}} \tag{3.16}$$

To find out the fundamental operation that will be performed in the back propagation let us consider an intermediate quantity, δ_j^l which is a measure of error in the j^{th} neuron in the l^{th} layer.

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l} \tag{3.17}$$

The quantity δ^l denotes the vectors associated with errors in layer l. In the backpropagation algorithm, for each of the layers in neural network, δ^l is calculated and using this parameter $\partial C/\partial w_{jk}^l$ and $\partial C/\partial b_j^l$ is calculated. Using equation 3.9, error in the output layer (L), δ^L, is found by applying chain rule and is given by

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L). \tag{3.18}$$

Provided that the cost function is known, δ^l is calculated by using Hadamard product and z_j^L which is calculated during the forward propagation. In matrix format equation 3.18 can be written as

$$\delta^L = \nabla_a C \odot \sigma'(z^L). \tag{3.19}$$

Here, $\nabla_a C$ is a vector quantity whose component are given by the partial deriva-

tives, $\partial C/\partial a_j^L$. Error terms in the $L-1$ layer is found by using δ^L and weight matrix w^L.

$$\delta^{L-1} = ((w^L)^T \delta^L) \odot \sigma'(z^{L-1}) \tag{3.20}$$

Here, w^{LT} is the transpose of weight matrix w^L in the L^{th} layer. For any layer l, the error vector is found by δ^l where, $\delta^l = (w^{l+1})^T \delta^{l+1} \odot \sigma'(z^l)$. Here, $(w^{l+1})^T$ is the transpose of weight matrix w^{l+1} in the $(l+1)^{th}$ layer. Using the δ^l parameter the equation for rate of change of C with respect to bias. By applying the chain rule we find

$$\frac{\partial C}{\partial b_j^{(l)}} = \frac{\partial C}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} = \frac{\partial C}{\partial z_j^{(l)}} = \delta_j^l, \tag{3.21}$$

where, $\partial z_j^{(l)}/\partial b_j^{(l)} = 1$. Again, similarly using the chain rule, equating for the rate of change of cost with respect to weight is found by equation 3.22.

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l. \tag{3.22}$$

3.2 Zynq SoC

This section gives a brief introduction to FPGA, SoC, HLS and CAPI architectures. FPGA is a semiconductor device that includes configurable logic blocks, block Random Access Memories (RAMs) and Digital System Processing (DSP) blocks that are connected through programmable interconnects. FPGA has fine-grained, low-latency memory with higher bandwidth, which helps to make a flexible and customizable hardware accelerator.

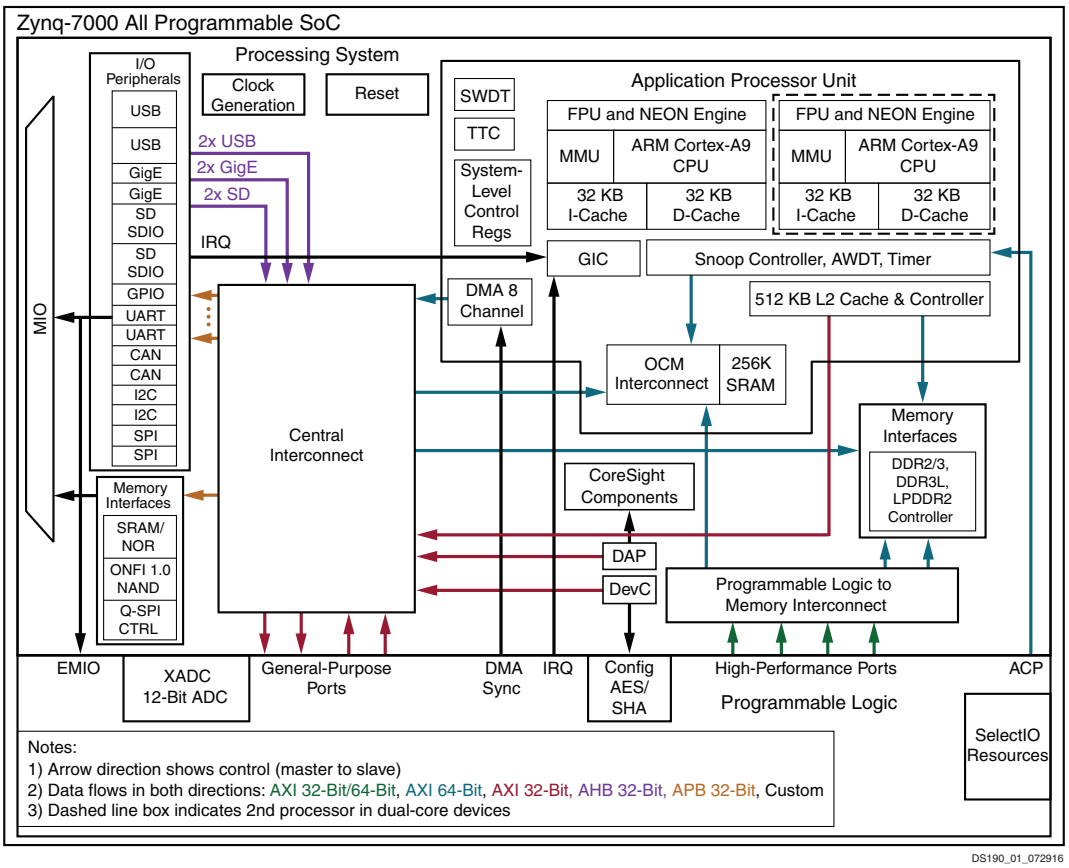


Figure 3.3: Zync SoC Architecture [3]

In this thesis, Xilinx Zynq-7000 family SoC device is used. The Zynq- 7000 SoC device enables the integration of software programmability and hardware programmability in an ARM-based processor with Xilinx FPGA [40]. The SoC is equipped with Cortex-A9 dual-core ARM processor with 28-nm Artix-7 FPGA. All the computational task of the neural network implementation is done in the Programmable System (PS), Cortex-A9 processor except for the dot-product calculation. The dot-product calculation task is done in the Programmable Logic (PL) in FPGA section of the SoC. In Figure 3.3, a block diagram of Zynq SoC architecture is shown. A central interconnect maintains interconnection between different parts of the PS and also between PS an PL sections of the SoC. There are four high-performance Advanced eXtensible Interface (AXI) master-slave interface ports which provide high bandwidth data transfer between PL and PS. The ARM processor initializes the data transfer (read and write requests) between PS and PL. There is an accelerator coherency port through which the AXI master in the

PL section can read and write data in the ARM memory address space. Each of the Cortex-A9 processors contains the snoop control unit which maintains level-1 caches and a shared level-2 cache, and the coherency between them.

3.3 HLS

Traditionally, hardware description language (HDL) such as Verilog, VHSIC Hardware Description Language (VHDL) is used to design a logic circuit for programming FPGA. The HDL code is abstracted to register transfer level (RTL) of the logic, which a hardware engineer designs to implement using parallel mode of operation with combinational and sequential logic. Programming in RTL is very close to programming in gate level logic, and thus gives the programmer a very good controllability of the hardware. Implementation of an algorithm in RTL requires a good amount of effort in formulating a finite state machine, break down the algorithm into the logic level, and parallelizing and pipelining different blocks of the algorithm, and thus the whole process is time-consuming. High-level synthesis (HLS) tool gives the opportunity to the programmer with a high level of abstraction to program in a high-level programming language such as C, C++, SystemC and compiles to HDL level under some programming constraints (i.e., without dynamic memory allocation) [41], [42], [43]. HLS tools also provide the capability for co-synthesis of the designed algorithm to verify the design. It also provides the option to optimize the design using parallelism and pipe-lining [44], [45], [46].

The HLS tool of Xilinx is integrated with the Xilinx's Vivado design methodology and the Intellectual Property (IP) block generated from the HLS tool is realized by the Vivado Design Suite for the Zynq 7000 series SoC. In this thesis, the HLS tool is used for designing the hardware accelerator using C language for the Zynq SoC, thus enabling the automated generation of RTL architecture to be implemented in the FPGA.

3.4 Power 8 and CAPI

In the fast-growing high-performance computing environment, acceleration is the key factor that researchers can use in performing data analytics. IBM's Power 8 processor architecture, which supports OpenPOWER environment (allows open licensing environment) [47], is designed by focusing on big data analysis. It has a larger memory bandwidth around the processor, which makes it possible to process big data in real-time. With lower latency and higher bandwidth, it can transfer data through its Generation 3 PCIe I/O bus. Multi-threading is another feature for which the Power 8 processor is being chosen by data centers. Also, being an "OpenPOWER" system, it attracts researchers to develop their application by collaborating with the greater community.

In a traditional SoC system, programmers have to take the overhead of managing I/O mapping for maintaining the communication and for maintaining memory mapping between PS and PL. On the other hand, the Power 8 system removes this overhead by introducing a coherent processor accelerator interface (CAPI), which allows part of any task to be completed in an accelerator. Also in a traditional SoC, there exists an overhead for any processor for translating between a virtual address and physical address. But in the Power 8 CAPI system, in the accelerator there exists an Effective-to-Real-Address Translator (ERAT) that translate between physical and virtual address, there is no need to translate between a virtual address and physical address. In the Power 8 system, the accelerator effectively acts as a thread. It has the flexibility of attaching devices across the PCIe bus coherently. The overhead of running things across the I/O bus is that the software stack needs to communicate with the device driver. In other systems, even if the I/O has a higher bandwidth, it still requires a large amount of CPU to communicate with the device driver.

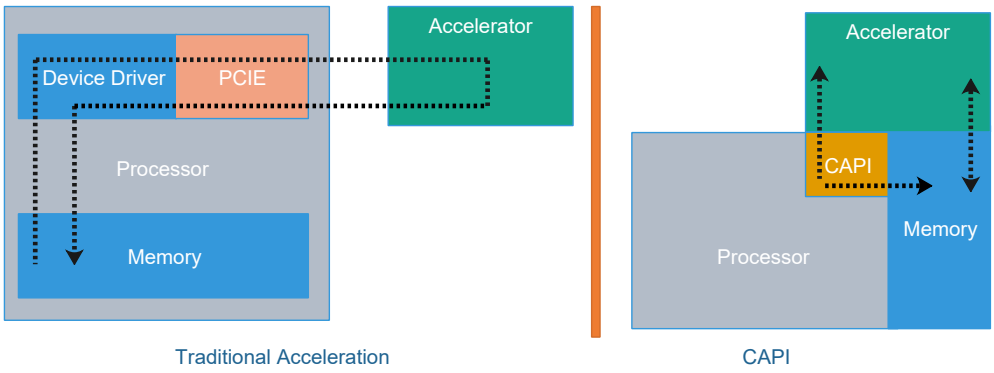


Figure 3.4: Memory access trends (modified and redrawn [4])

In the CAPI paradigm, the part of any algorithm that needs to be accelerated is kept in a functional unit of FPGA is called the Accelerator Function Unit (AFU). The whole purpose of an AFU is to facilitate an application with a higher-density computational unit to maximize the overall performance. The CAPI architecture in the Power 8 system enables the system to use the AFU unit as a coherent peer to the Power 8 processors. The Coherent Accelerator Processor Proxy (CAPP) unit maintains coherency protocol between the AFU and the cores of the Power 8 processor. The CAPP unit maintains architectural coherency for the AFU across the virtual memory space. The Power Service Layer (PSL) from IBM that is maintained in the FPGA, which maintains the connection between the CAPP unit across the PCIe bus. The communication between the processor core and PSL is maintained by the Power 8 processor and the PSL, thus allowing the programmer to concentrate on the algorithm. In Figure 3.5, a block diagram of the architecture is shown. The AFU contains a cache of 256 KB, and the accelerator can direct any algorithm running on the cores of Power 8 processors to access the cache. The accelerator uses an unchanged virtual address space that an application uses with full access. Any application needs to send a processing element to the AFU that contains a Work Element Descriptor (WED), which contains the whole description of the application that needs to be accelerated in the AFU. The WED element can also contain any pointer to memory on which the accelerator performs necessary

operations.

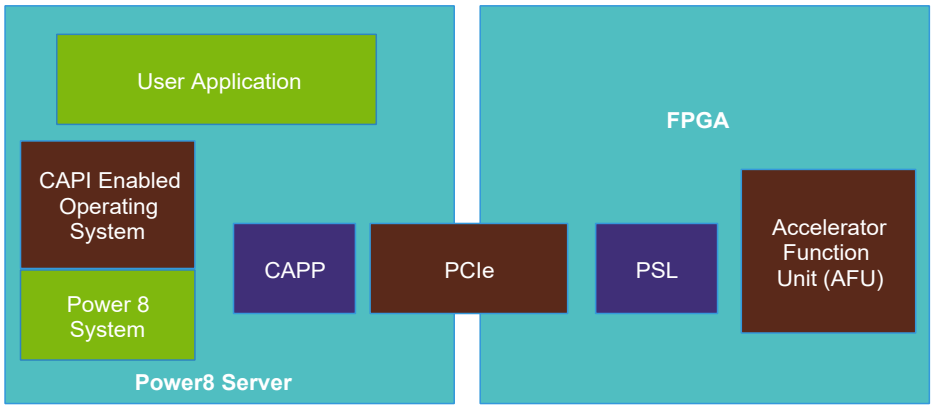


Figure 3.5: Power8 CAPI architecture

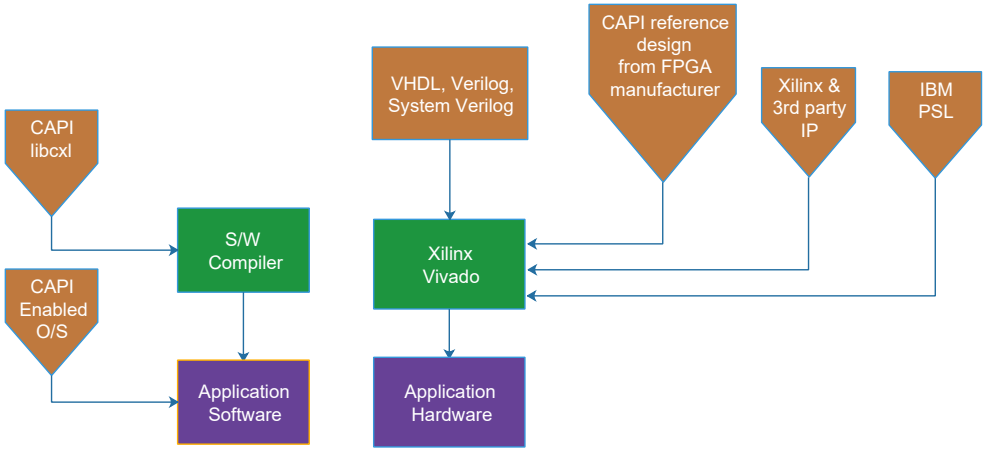


Figure 3.6: Software Hardware Components

On the software side programmers need to include a CAPI library called "libcxl" which facilitates several functions to connect and communicate with the CAPI device in the Power 8 system. Figure 3.6 shows all required software and hardware units. PSL communicates with the AFU through the following interfaces: AFU command interface, AFU buffer interface, PSL response interface, AFU Memory Mapped Input Output (MMIO) interface, and AFU control interface. Using AFU command interface the AFU makes service requests to the PSL, while through

the PSL response interface PSL reports about the service requests. Using the buffer interface PSL and AFU transfer data between them and through the MMIO interface software running on the host can get access to the registers of the AFU. AFU manages the control states of PSL using the AFU control interface.

IV. SOC IMPLEMENTATION

Accelerator for multilayer neural network is designed and implemented in two different heterogeneous systems, i.e., Xilinx Zynq SoC and IBM Power 8 system.

4.1 Two-Layer Fully Connected Neural Net

This thesis presents, hardware accelerator for two-layer neural network. The network takes N-dimensional inputs and classifies into C classes. It has a hidden layer of dimension H and Rectified Linear Unit (ReLU) nonlinearity layer after first the FC layer. The ReLU computes $f(x) = max(0, x)$.This nonlinear unit helps to trigger on and off the neurons of the previous layer. It helps to converge to a solution more quickly [48]. L2-regularization is used for updating weight matrices to reduce the variance in the model. The network is trained with softmax classifier [49] function. The softmax layer is added behind the last fully connected layer. This layer computes the raw class score i.e. C_i into a class probability P_i according to the computation $P_i = e^{C_i} / \sum_k e^{C_k}$.

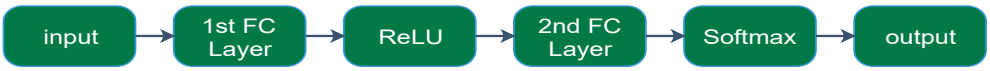


Figure 4.1: Two-layer neural network

Figure 4.1 shows the layers of a two-layer neural network considered in this thesis. A simplified algorithm used for training the neural network is given below:

In Algorithm 1, the weight matrices are initialized with randomly generated numbers from -1 to 1 and the biases are initialized with zero values. The training data is divided into two parts, training input data set and the validation dataset. The number of iteration that the training session will be run for optimization is pre-set by the value of iteration number. Batch size determines the number of training examples will be used per iteration. Fist minibatch of input data is created. Then the in the loss function, the gradient decent operation is performed

Algorithm 1 TWO LAYER NEURAL NET

function LOSS(X, y) ▷ X - input, y - output
    $N, D = shape\,(X)$
    $Temp1 = X$ dot $W1 + b1$
    $Temp2 = max\,(0, Temp1)$
    $Score = Temp2$ dot $W2 + b2$
    $Score\_Exp = exp\,(Score)$
    $Probs = Score\_Exp / sum\,(Score\_Exp)$
    $Log\_Prob = -log\,(probs\,(0..N)\,, y- = 1)$
    $Loss\_1 = sum\,(Log\_Prob)$
    $Loss\_2 = (sum\,(W1^2)) + sum\,(W2^2))\,/2$
    $Loss = Loss\_1 + Loss\_2$
    $Grad\_score = (probs[(0..N)\,, y]- = 1)\,/N$
    $Grad\_W2 = Transp\,(Temp2)$ dot $Grad\_score$
    $Grad\_b2 = sum\,(Grad\_score)$
    $Grad\_H = Grad\_score$ dot $Transp\,(W2)$
    $Grad\_H = 0$ if $Temp2 <= 0$
    $Grad\_W1 = Transp\,(X)$ dot $Grad\_H$
    $Grad\_b1 = sum\,(Grad\_H)$
    $Grad\_W2 = Grad\_W2 + reg\_coeff * W2$
    $Grad\_W1 = Grad\_W1 + reg\_coeff * W1$
end function
function TRAIN($X, y, X\_val, y\_val, num_of_iter, batch_size$) ▷ X - input, y - output
    $Num\_train = X.shape[0]$
    for $iter = 0$ to num_of_iter **do**
        $indx =$ random $((0..Num\_train)\,, batch\_size)$
        $X\_batch, y\_batch = X[indx], y[indx]$
        $Grad = LOSS\,(X\_batch, y\_batch)$
        $W1, W2, b1, b2+ = \eta * (Grad\_W1, Grad\_W2, Grad\_b1, Grad\_b2)$
    end for
end function

on the minibatch of data to find out the weight matrices that minimized the mean squared errors. In the loss function, in forward pass, the weight matrices are multiplied with the input matrices to perform the linear regression. This generates the class scores. The scores are then passed through the ReLU activation function and class probabilities are calculated. Then, in the backward pass the gradient on the weight matrices are calculated.

From the algorithm, it can be found that the highly computational unit of the two-layer net is the calculation of the dot-products used in the loss function, which is repeatedly performed to minimize the sum squared error. Therefore, acceleration is proposed for the dot-product calculations.

4.2 Acceleration in Xilinx Zynq SoC

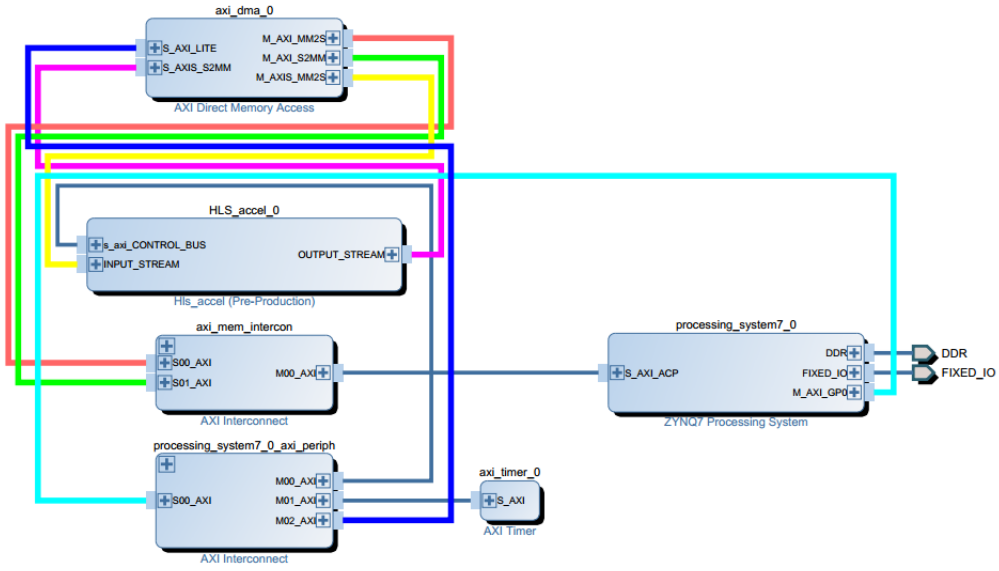


Figure 4.2: Architecture in zynq SoC

The proposed framework of the neural network distributes the computation of DNN between the CPU and the accelerator. The hardware accelerator architecture is proposed for two platforms, IMB CAPI and Zynq SoC. In this section, the accelerator architecture in Zynq SoC platform is proposed. In Figure 4.2, the main modules of the proposed accelerator and the interconnection and interface among the modules are shown. The most expensive computation part of multilayer

neural network, the dot product calculation in FP32 between two matrices is accelerated in the *Hls_accel* module. The accelerator is designed in HLS and imported into the design. The rest of the calculations are performed in the *ZYNQ Processing System* block, which includes a Cortex-A9 ARM processor CPU. The element of the two matrices is first written in the DDR3 memory using a DMA controller. When the data is written in the DDR3 memory, the data is passed to the accelerator, and the accelerator starts functioning. The accelerator writes its output to the Direct Memory Access (DMA) through AXI interconnect. The CPU controls the accelerator through an AXI control interface bus. All the modules in the accelerator is designed using HLS and realized in RTL for the accelerator. So, all the modules except the *ZYNQ Processing System* block are initialized as hardware in the FPGA. These modules are initialized to the cortex Cortex-A9 processor as AXI memory mapped devices. To communicate with these devices the processor needs to know the addresses of these devices. If the addresses are not correctly assigned to the devices, then the processor won't be able to communicate with them. The memory addresses of the memory mapped AXI devices are shown in Figure4.3. To be able to transfer data form the accelerator processor needs to write and read data in the address between 0x43C00000 to 0x43C0FFFF.

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
HLS_accel_0	s_axi_CONTROL...	Reg	0x43C0_0000	64K	0x43C0_FFFF
axi_timer_0	S_AXI	Reg	0x4280_0000	64K	0x4280_FFFF
axi_dma_0	S_AXI_LITE	Reg	0x4040_0000	64K	0x4040_FFFF
axi_dma_0					
Data_MM2S (32 address bits : 4G)					
processing_system7_0	S_AXI_ACP	ACP_DDR_LOWOCM	0x0000_0000	1G	0x3FFF_FFFF
processing_system7_0	S_AXI_ACP	ACP_M_AXI_GP0	0x4000_0000	1G	0x7FFF_FFFF
processing_system7_0	S_AXI_ACP	ACP_IOP	0xE000_0000	4M	0xE03F_FFFF
Data_S2MM (32 address bits : 4G)					
processing_system7_0	S_AXI_ACP	ACP_DDR_LOWOCM	0x0000_0000	1G	0x3FFF_FFFF
processing_system7_0	S_AXI_ACP	ACP_IOP	0xE000_0000	4M	0xE03F_FFFF
processing_system7_0	S_AXI_ACP	ACP_M_AXI_GP0	0x4000_0000	1G	0x7FFF_FFFF

Figure 4.3: Addresses of AXI memory mapped device

The addresses table that has been assigned to the modules are translated for the Zynq SoC. The hierarchy of the functions performed on the SoC is shown in the block diagram in Figure 4.4. The Cortex-A9 processor starts the main function and starts executing the neural network. As mentioned in the algorithm, for minimizing the squared error, the loss function is executed. In the loss function

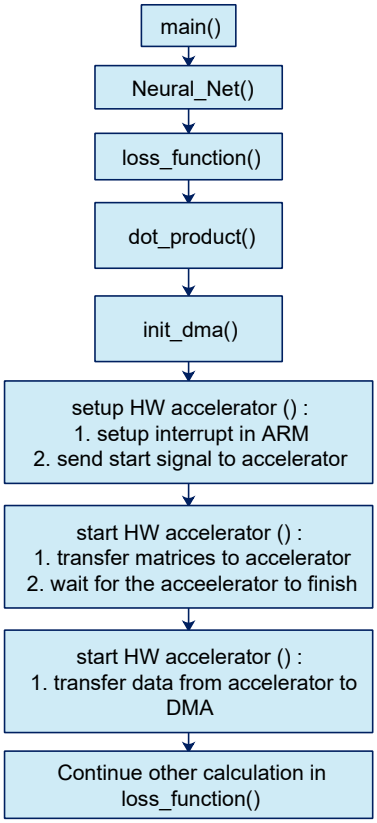


Figure 4.4: Zynq implementation

its necessary to calculate the dot-products. As, the dot-product is calculated in the hardware accelerator and the accelerator needs access to the memory. The *init_ dma()* function is called to initialize the AXI DMA interconnect to be able to read and write data form memory. The processor initializes its interrupt service routine for receiving interrupt from the hardware accelerator. Then the processor sends start signal to the accelerator to transfer data from the memory to the hardware accelerator. Upon receiving the data, the accelerator performs the computation and when it finishes the computation it sends a signal to the processor. Then the processor controls the accelerator to write the computed result in the memory. When the accelerator finishes writing the result, it sends an interrupt signal to the processor to continue the rest of the computation.

V. COHERENT ACCELERATOR IMPLEMENTATION

This chapter introduces the system architecture development and state machine organization for data transmission within CAPI interface. The POWER8 processor includes a symmetric multiprocessor (SMP) bus interconnection fabric which enables the various units to share system memory and communicate coherently. In the POWER8 processor system, PCIe Host Bridge (PHB) provides connectivity to PCIe Gen3 I/O links. There are memory controller (MC) blocks and a coherent accelerator processor proxy (CAPP) block along with the PHB which enable memory coherency, data transfer, as well as interrupt system and address translation for the PCIe attached accelerator [50]. The CAPI system architecture is depicted in Figure 5.1. This chapter describes the interface details of CAPI and the algorithm details for proper PSL-AFU communication.

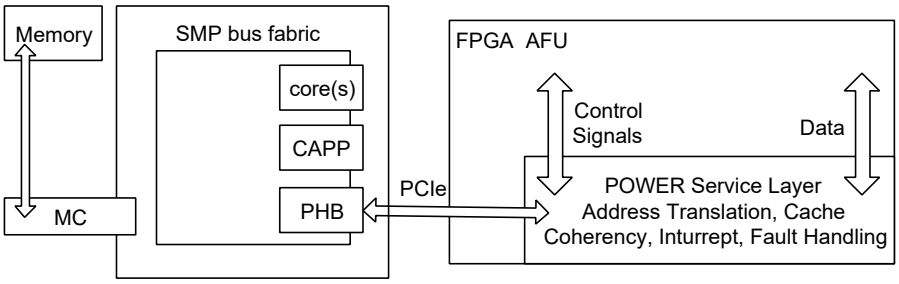


Figure 5.1: CAPI System Architecture.

5.1 The Psl-AFU Interface

The PSL-AFU interface communicates with the accelerator logic running on the FPGA. Through this interface, the PSL offers service to the FPGA and responses to any request made by the FPGA. In our accelerator design, the communication between PSL and AFU is maintained using five interface channels. The communication is established using different handshake signals. The hardware programmer has to generate and check the signals in different ports of the interfaces of AFU for maintaining the communication. The interface channels are broadly categorized as follows:

1. Accelerator Command Interface
2. Accelerator Buffer Interface
3. PSL Response Interface
4. Accelerator MMIO Interface
5. Accelerator Control Interface/ Job Interface

These interfaces allow the software stack to control the accelerator state and allow the accelerator to access the data in the system memory. Table 5.1 summarizes the interfaces with the source and destination information along with brief description of the interfaces.

Table 5.1: Interface details

Interface	From	To	Details
Command Interface	Accelerator	PSL	Send Service request
Buffer Interface	PSL, Accelerator	Accelerator, PSL	Transfer Data
Response Interface	PSL	Accelerator	Reports Status about service requests
MMIO Interface	Software	Accelerator	Reads Register within accelerator
Control Interface	PSL	Accelerator	Control state of accelerator

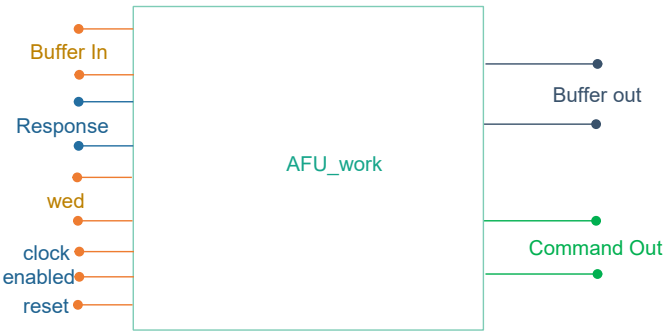


Figure 5.2: AFU Block Diagram

In our design, the interfaces are sub-grouped into two categories: Input, and Output. There is an unit inside the AFU named *AFU_work* which connects with either one or both of the 'in' and 'out' sub-categories of the interface. Some of the

important subdivisions of each of the interfaces are shown in Figure 5.3.

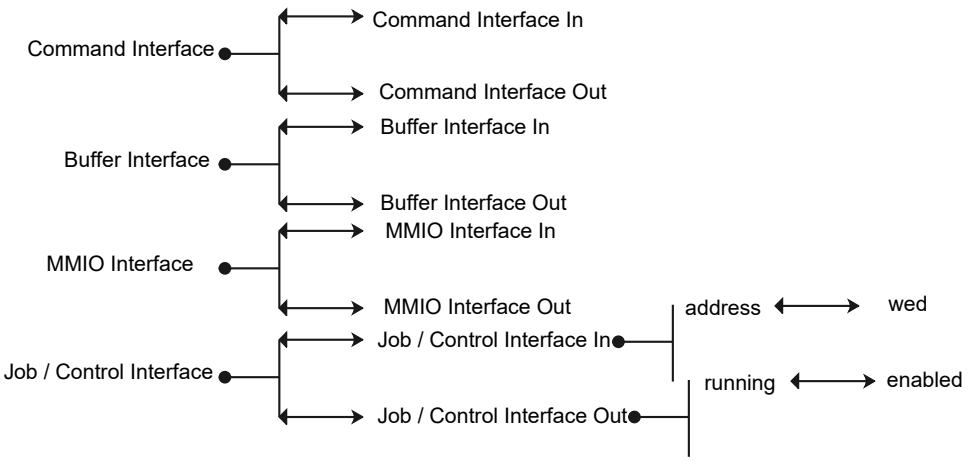


Figure 5.3: AFU Interface

5.1.1 Accelerator Command Interface

Through the accelerator command interface the AFU logic sends command to the PSL. This interface works in a synchronous way. The *AFU_work* unit communicates with the PSL via the *command Interface Out* bus. For each of the command, a valid signal is issued from the AFU. All the individual commands are assigned with 8-bit tag numbers. The PSL uses these tag numbers in its subsequent operations while serving for each of the commands. The tag number is used by the *Buffer Interface* and by the *Response Interface* for updating the status notification. The *command code* bus informs the PSL about the action it needs to perform. IBM has specified some specific codes for the PSL. The *command address* bus holds the effective addresses of shared memory. The lay out *Command Interface* is shown in Figure 5.4.

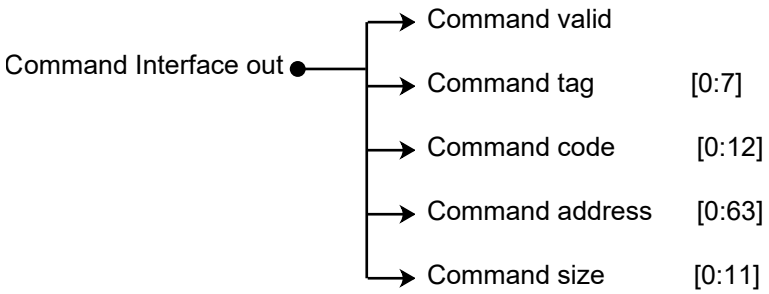


Figure 5.4: Command Interface

5.1.2 Accelerator Buffer Interface

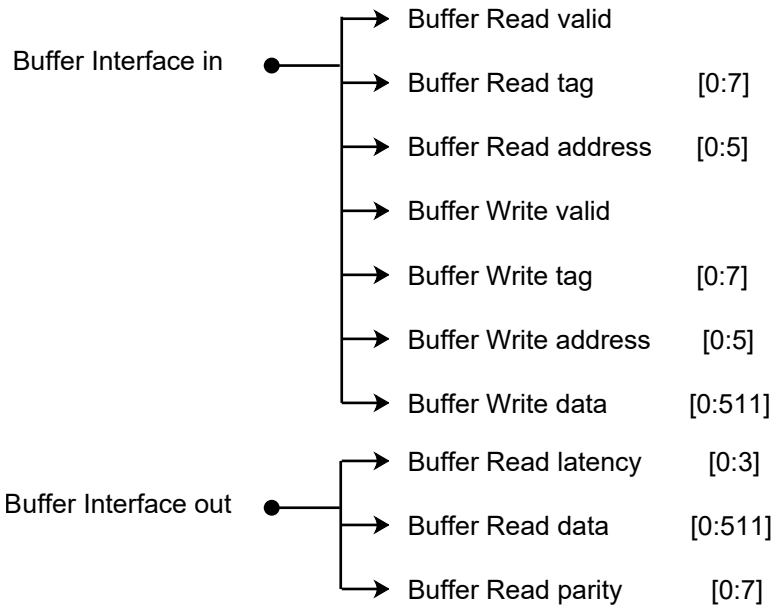


Figure 5.5: Accelerator Buffer Interface

Buffer Interface is responsible for transferring data between PSL and AFU. After receiving a valid command with a specific tag, buffer interface reads or writes data from and to the AFU. The buffer interface can read or write simultaneously. While transferring data half line of data (512-bits) is read or written through the buffer interface, *buffer interface in* or *buffer interface out*. The read operation is performed in pipe-lined style. The *Buffer Read/ Write valid* signal is issued when a valid data is present in the interface. The *Buffer Read/ Write tag* indicates the command tag number for which the PSL is responding through the buffer interface. The port description of the buffer interface is shown in Figure 5.5.

5.1.3 PSL Response Interface

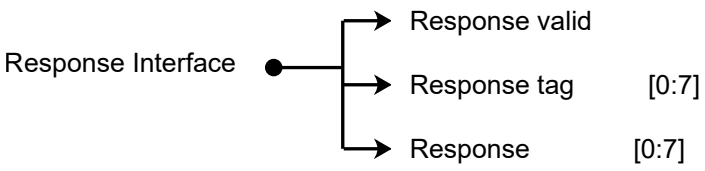


Figure 5.6: PSL Response Interface

The response interface is responsible for signaling the completion of different commands by the PSL requested from the AFU. This interface helps out in main-

taining the control flow. The structure of response interface is shown in Figure 5.6. The *response valid* signal is issued when a valid response is present in the interface. The *response tag* indicates the tag number of the command for which the response has been generated. In the *Response* bus, response codes are transferred. The definitions of the response codes are generated by IBM.

5.1.4 Accelerator Control Interface

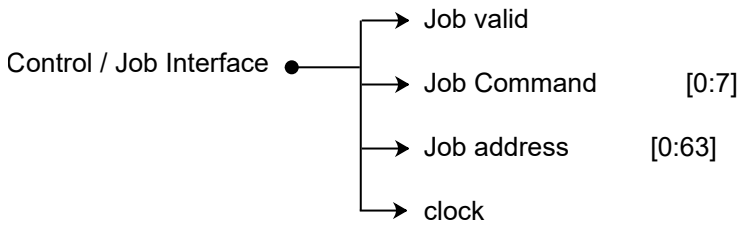


Figure 5.7: Accelerator Control Interface

Through the accelerator control interface, the state of the AFU is monitored and controlled. The working unit of AFU is also monitored and controlled through this interface. The valid signal is issued for one cycle for a command. The PSL sends the control commands to the work element unit of the PSL through the *Job Command Interface*. The PSL mainly resets and starts the AFU through this bus. Through the 64-bit *job address* bus, the WED information is passed from the PSL to the AFU and lastly to the *AFU_work* unit of AFU.

5.2 State Machine Design for CAPI Interface and Data transfer

A state machine has been designed to maintain the control flow in the AFU. For streaming the data from the host memory to the PSL and from the PSL to the host memory, different interfaces from PSL-AFU is used. By checking different signals and by setting values on different buses and ports the state machine translates from one state to another.

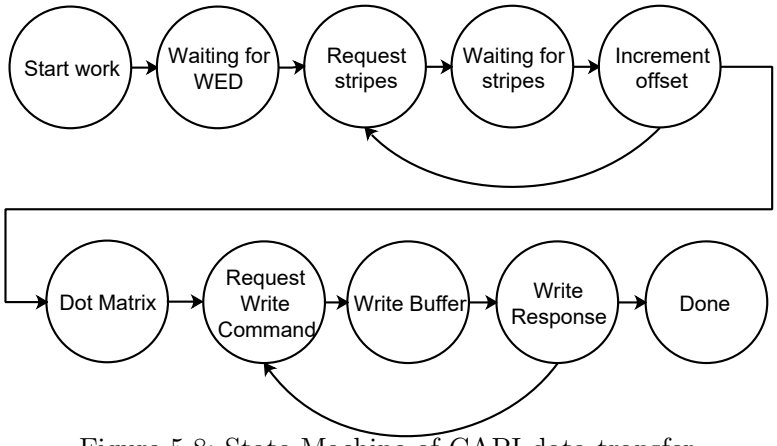


Figure 5.8: State Machine of CAPI data transfer

At first, the PSL resets the state machine via the *control interface* with a *RESET* command. After being reset by the PSL, the state machine of the AFU initializes its state in the *START WORK* state. Then, it waits for receiving the WED via the *control interface* in the *WAITING FOR WED* state. When it gets a WED element, it knows the effective address of the data on the shared memory. In the *REQUEST STRIPES* state, the AFU make requests through the *command interface* to read data from memory. After making the request, in the *WAITING FOR STRIPES* state, the state machine observes the ports and buses of the response interface and buffer interface. When it receives a *buffer valid* signal, it takes the data via the buffer interface. After receiving the data from shared memory, different register values are updated in the *INCREMENT OFFSET* state. By cycling the states, *REQUEST STRIPES, WAITING FOR STRIPES, and INCREMENT OFFSET*, the FPGA receives the data from the shared memory. When all the data is on the on-chip memory of FPGA, the state machine performs the computation in the *Dot Matrix* state. The work unit inside the AFU performs the computation and stores the result of the computation in a buffer. After the completion of the computation, the state machine transfers the state to the *Request write command* state. Through the command interface, it makes a write request to the PSL to write data from AFU buffer to the memory. After making the write request, the state machine waits for the buffer valid signal to be high for one cycle. When it gets the buffer valid signal it writes the data through the buffer interface. By looping

through several cycles of *REQUEST WRITE COMMAND, WRITE BUFFER and WRITE RESPONSE* states the state machine completes sending all the data from the buffer.

5.3 Detailed Implementation of the State Machine:

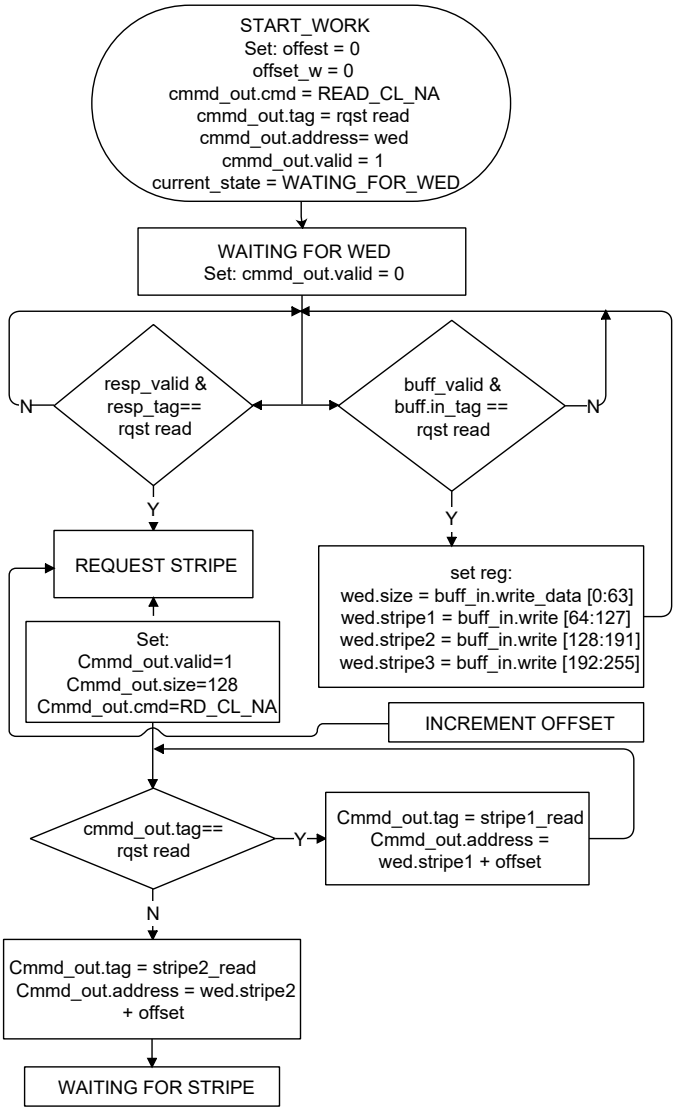


Figure 5.9: Detailed Flow diagram of START_WORK

After getting a reset signal from the PSL, the AFU resets its states. Several command tags are enumerated for different kinds of command requests at the beginning of the execution of the states. In the *START WORK* state, it sets the registers, offset and offset_w to their initial values. These two registers count the offset that needs to be added with the stripe1, srtripe2 and stripe3 addresses for reading and writing data from and to the correct addresses. This state also sets the different ports of the command out interface. First, the command is sent to read data from memory with a command READ_CL_NA. The command is attached with a tag that was enumerated in the beginning of execution. The address is set

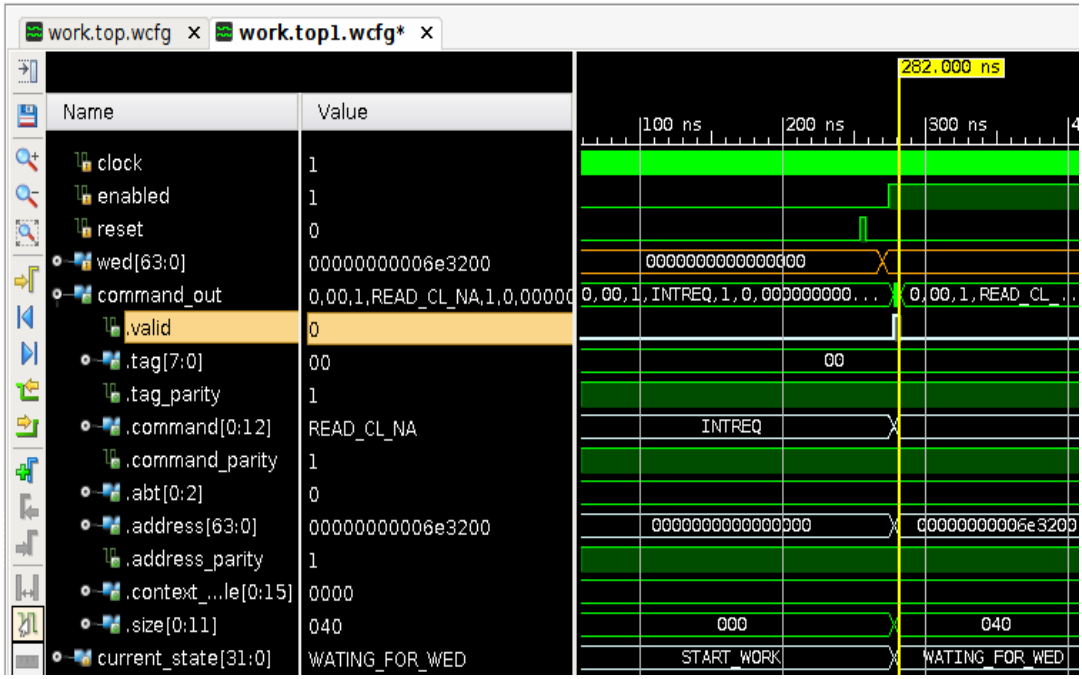


Figure 5.10: Timing Diagram for the state: *START_ WORK*

with the value that is received through the address bus of the job in interface while resetting the state machine. After setting all the ports of command interface the *command out valid* signal is set for one cycle. In Figure 5.10, *START_ WORK* state is shown. A 32-bit register, *current_ state*, holds the current state of the state machine. After getting a reset signal from the PSL, the WED information is passed to the *command_ out.address* bus to read the WED descriptor from memory. In this case, the WED address is 6e3200. The size of the WED descriptor is 64 (0x40) bytes which are also passed through the *command out* interface.

```
sefat@sefatVb:/media/sefat/E/psl/C$ LD_LIBRARY_PATH=/media/sefat/D/psl/pslse/lib
cxl/ ./test_alloc_all
128-byte aligned addr: 0x6b2c80
128-byte aligned addr: 0x6c2d00
128-byte aligned addr: 0x6d2d80
[wed structure
  wed: 0x6e3200
  wed->size: 16384
  wed->stripe1: 0x6b2c80
  wed->stripe2: 0x6c2d00
  wed->stripe3: 0x6d2d80
  wed->size_A: 16384
  &(wed->done): 0x6e3228
INFO:Connecting to host 'localhost' port 16384
Attached to AFU
Waiting for completion by AFU
```

Figure 5.11: Wed structure in host application

The address of the WED descriptor is a pointer which holds the WED structure. The pointer is generated in the host application and passed to the PSL. Here, in Figure 5.11, the WED pointer generated from the host application is 0x6e3200 which is the exact value that the PSL got in its *wed* register through the *Job in address* bus. The WED descriptor contains the total 5 pointers to different data: *size*, *stripe1*, *stripe2*, *stripe3*, and *size_A*. From the *START_WORK* state, the state machine transfers its current state to *WAITING_FOR_WED* state. In this state, buffer interface is checked for a *buffer in write valid* signal. When the WED descriptor is read from memory and is available on the buffer interface, the value of the descriptor is stored on some registers in the FPGA.

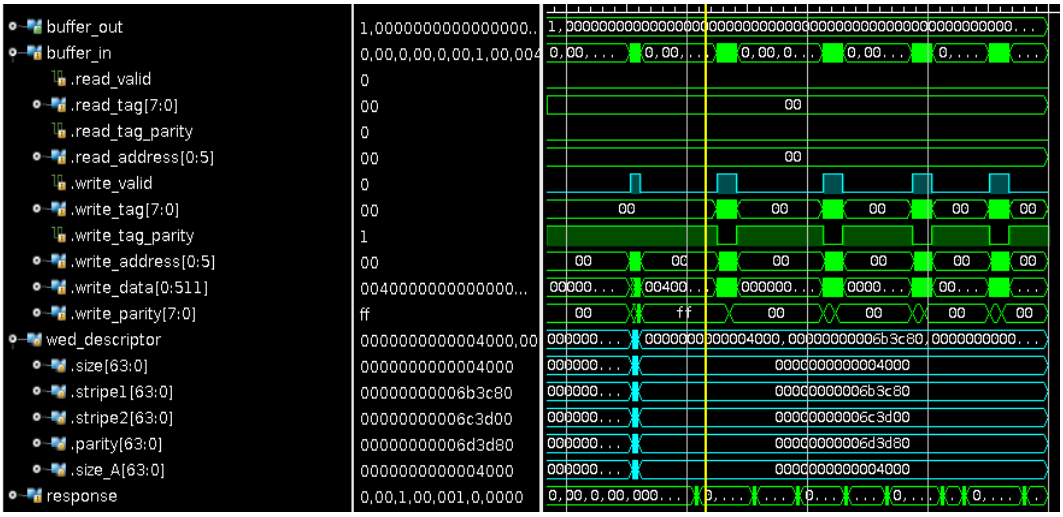


Figure 5.12: WED descriptor elements in simulation

In Figure 5.12, the contents of the *wed_descriptor* register are shown. When the *write valid* signal is high, the *wed_descriptor* register is updated. Values of the pointers for the data *size*, *stripe1*, *stripe2*, *stripe3* (which in this state machine has been declared as parity) and *size_A* are received as 0x4000 (DEC 16384), 0x6b3c80, 0x6c3d00, 0x6d3d80, and 0x4000 (DEC 16384) which exactly match with the WED descriptor that has been sent from the host application. When the response valid signal is high, the state machine transfers its state to the *REQUEST STRIPE* state. In Figure 5.13, it is shown that from this state, using the command interface, data read request is sent for reading stripe1 and stripe2 data from the

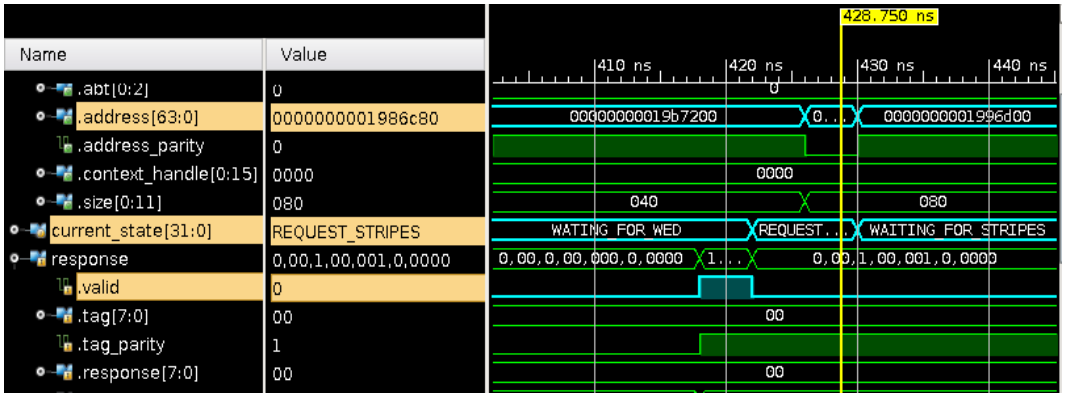


Figure 5.13: Command request for reading stripe1 and stripe2 data

shared memory. In Figure 5.14 the addresses of stripes are shown.

```
sefat@sefatVb:/media/sefat/E/psl/C$ LD_LIBRARY_PATH=/media/sefat/D/psl/pslse/lib
cxl/ ./test_alloc_all
128-byte aligned addr: 0x1986c80
128-byte aligned addr: 0x1996d00
128-byte aligned addr: 0x19a6d80
[wed structure
  wed: 0x19b7200
  wed->size: 16384
  wed->stripe1: 0x1986c80
  wed->stripe2: 0x1996d00
  wed->stripe3: 0x19a6d80
  wed->size_A: 16384
  &(wed->done): 0x19b7228
```

Figure 5.14: Pointers for Stripe1 and Stripe2 data

In Figure 5.13, the timing diagram shows that when a valid response is available in the response buffer, first the AFU makes a request to the PSL to read the data from memory pointed by the address of stripe1 register and then makes the request to read data for the stripe2 register. When both requests are made with two different command tags through the command interface, the state machine goes to the next state to wait for the data to come through the buffer interface.

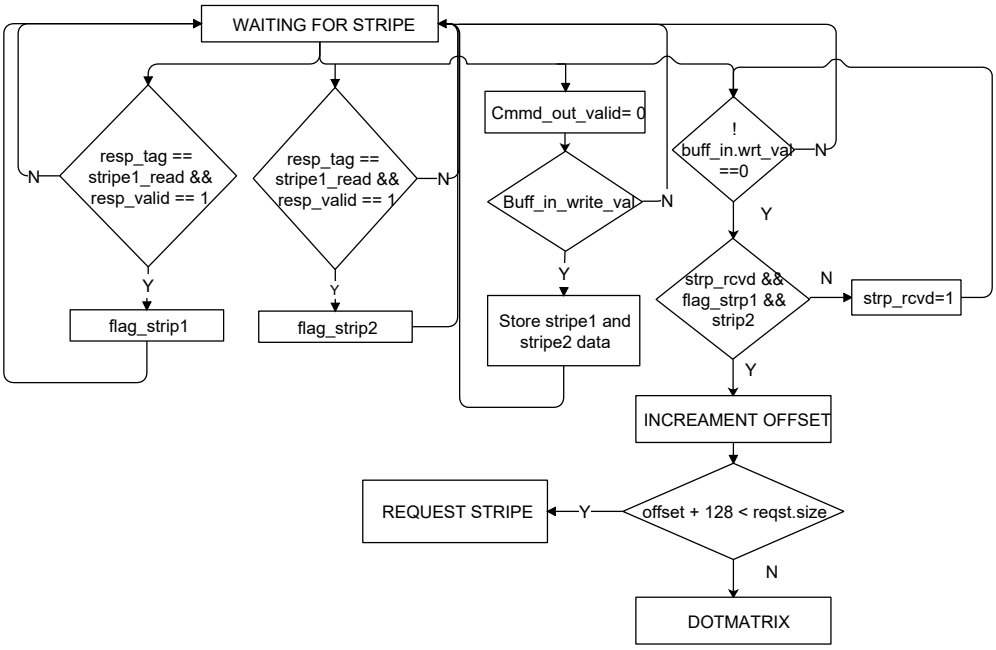


Figure 5.15: Algorithm implemented for data receiving

When the data are available in the buffer interface, the *response_valid* signal goes high. In the *WAITING FOR STRIPES* state, the state machine waits for the response valid signal to go high. When it finds that the response valid signal is high, depending on the value of the response tag it either receives data for stripe 1 or for stripe 2. After receiving the data from memory, it is buffered in the on-chip buffer of the FPGA.

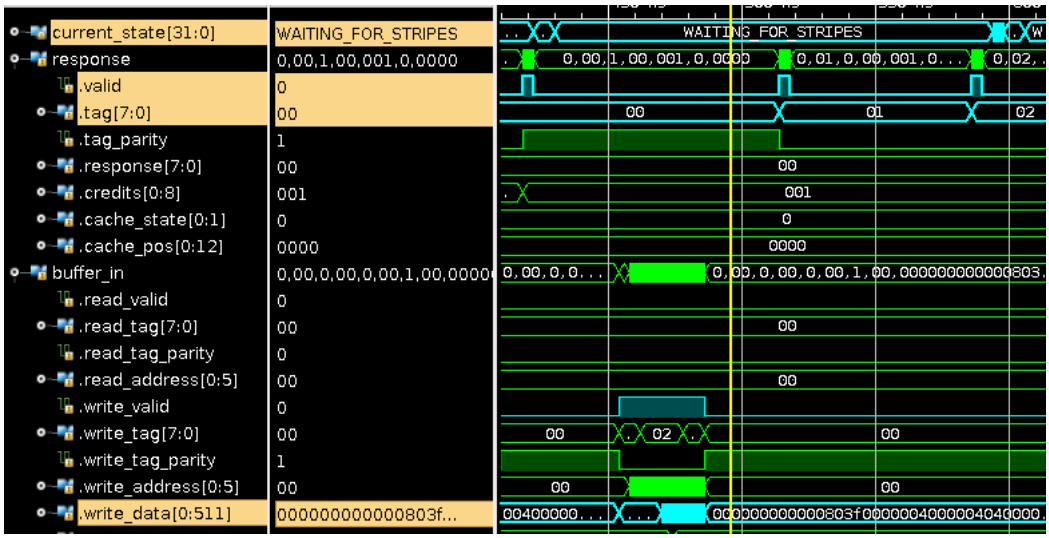


Figure 5.16: Timing diagram in the *WAITING FOR STRIPES* state.

After storing the data in the stripe1 buffer and stripe2 buffer, offset to the ad-

dresses are incremented for receiving next stripes. From the *INCREMENT OFFSET* state, the state is transferred to the *REQUST STRIPES* for requesting next stripes. This process repeats for receiving all the data for matrices. After receiving data for matrices, the state machine goes to the state of the computation. After finishing the computation, the result is stored in a buffer named stripe3.

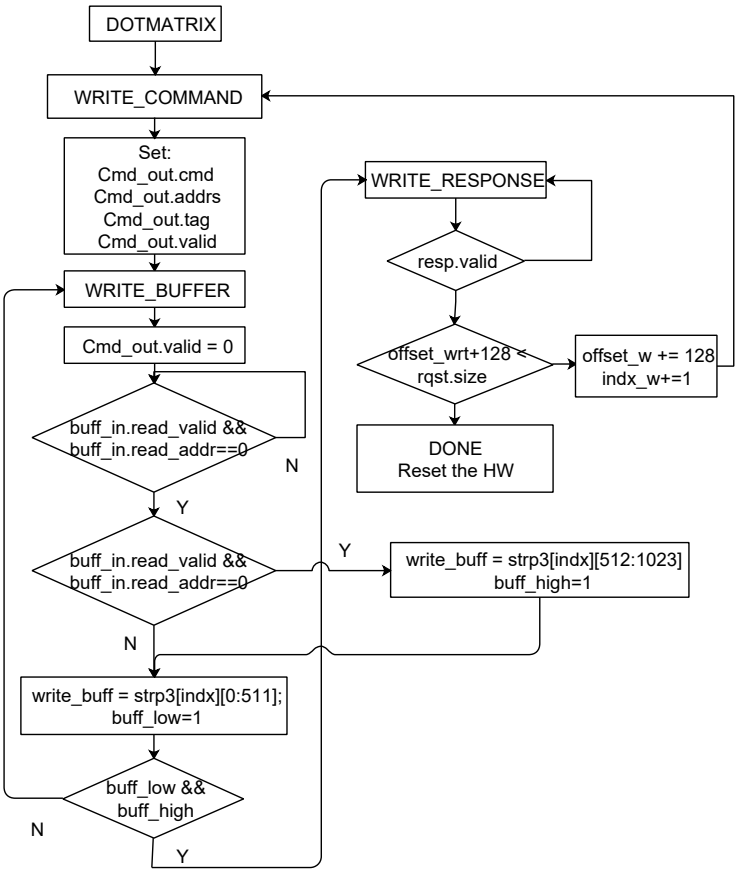


Figure 5.17: Algorithm to to write data for data

In the *WRITE COMMAND* state, using the command interface write request is made with the command code *WRITE_NA*. Command address is assigned with the value that was sent from the application in the WED descriptor. A valid command is created when a valid signal is made for one cycle. The timing diagram is presented in Figure 5.18.

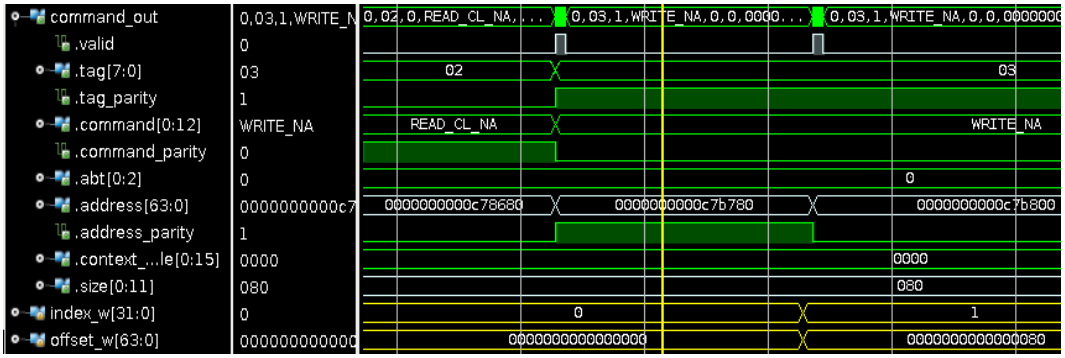


Figure 5.18: Valid command issue

Every time a *data write* command is completed, the value of the offset register, *offset_w*, is increased for finding the next address to where data is to be written. The register, *index_w* counts the index number for calculating the total amount of data that has been written in the memory.

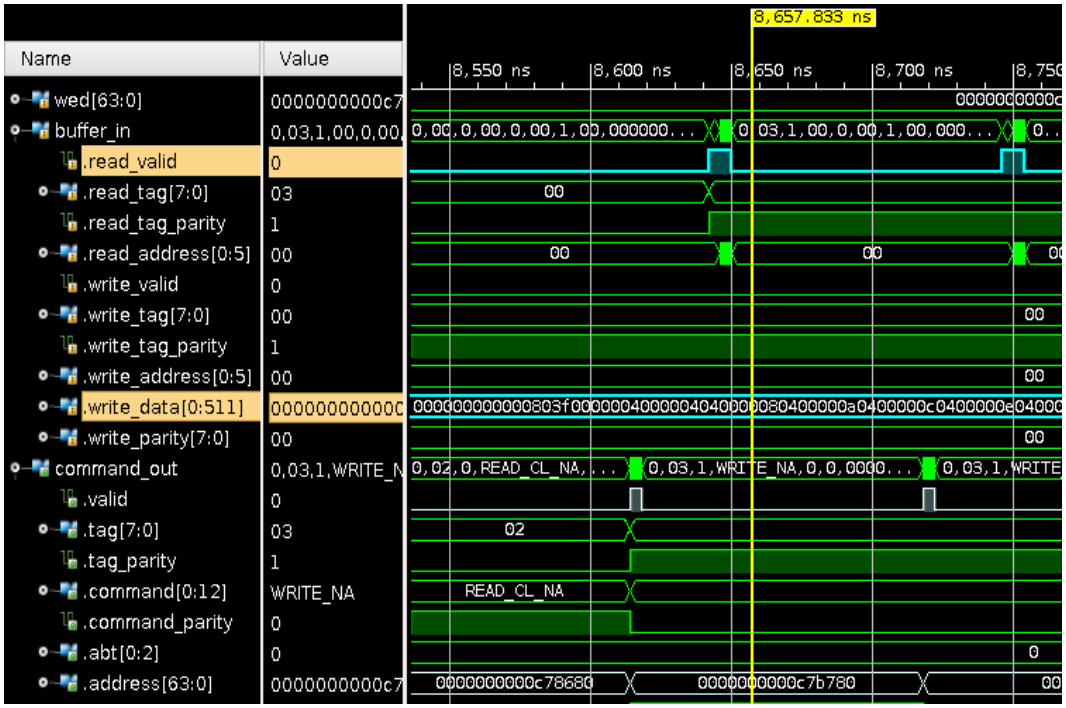


Figure 5.19: Write data timing diagram

After making the write request command through the command interface, the flag *buffer_in.read_valid* is made high to pass the data to the PSL through the buffer interface. In Figure 5.19, the timing diagram has been shown. When the data is passed through the buffer interface, *response valid* signal is checked for confirmation of data write validity. When a *response valid* signal is received, the

value in the *offset_ w* register is added to the current address, and *index_ w* register is also increased. The timing diagram is shown in Figure 5.20. Then, the state is transferred to to *COMMAND_ WRITE* state, and the three states are looped through until all the data is written in the memory.

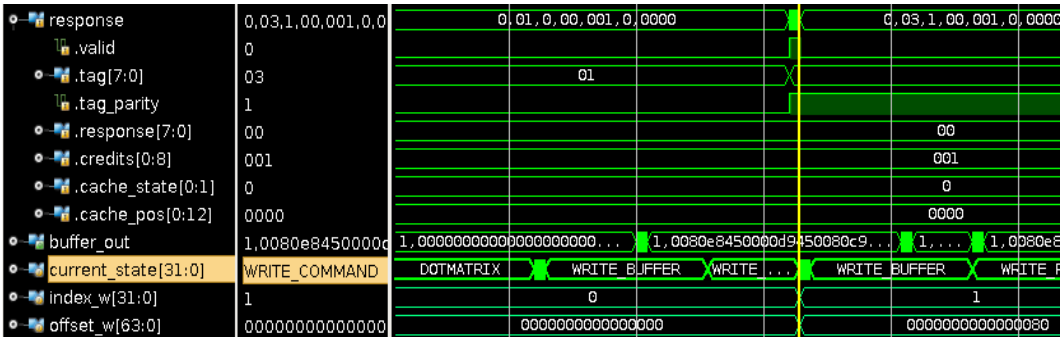


Figure 5.20: Write data timing diagram; response valid signal

5.4 Simplified Data-flow in Accelerator

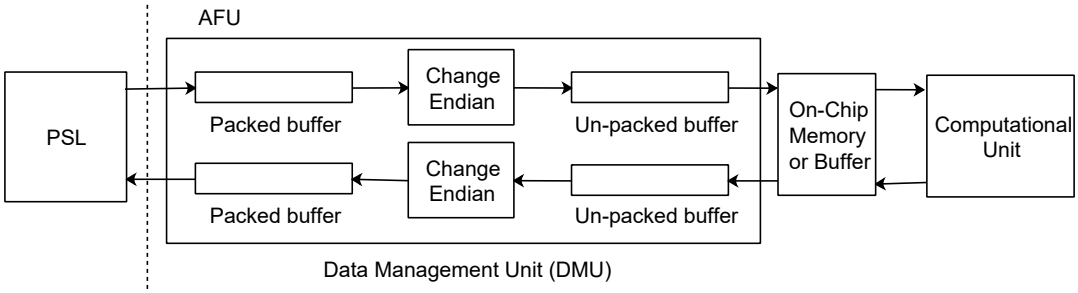


Figure 5.21: Simplified Data-flow in accelerator

In the hardware accelerator, data comes from the shared memory through the PSL to the on-chip memory of the FPGA. Depending on the design of the AFU, the memory can be BRAM type or sparse LUT RAM type. Inside the AFU there is a Data Management Unit (DMU) which manages the data to be structured and stored correctly in the AFU. The PSL represents the data in big-endian format whereas the accelerator works on the little-endian format. So the data is swapped to make it little-endian format. Through the buffer interface, the AFU receives the data in a packed format of m bytes where the maximum packed data size that PSL can buffer at a time is 128 bytes. To use the data in the computational unit, the data is transformed from the packed to the unpacked structure. After this

process, the data is stored in the on-chip memory of the FGPA. In Figure 5.21, simplified data flow in the DMU of AFU is shown.

5.5 Computational Unit Design

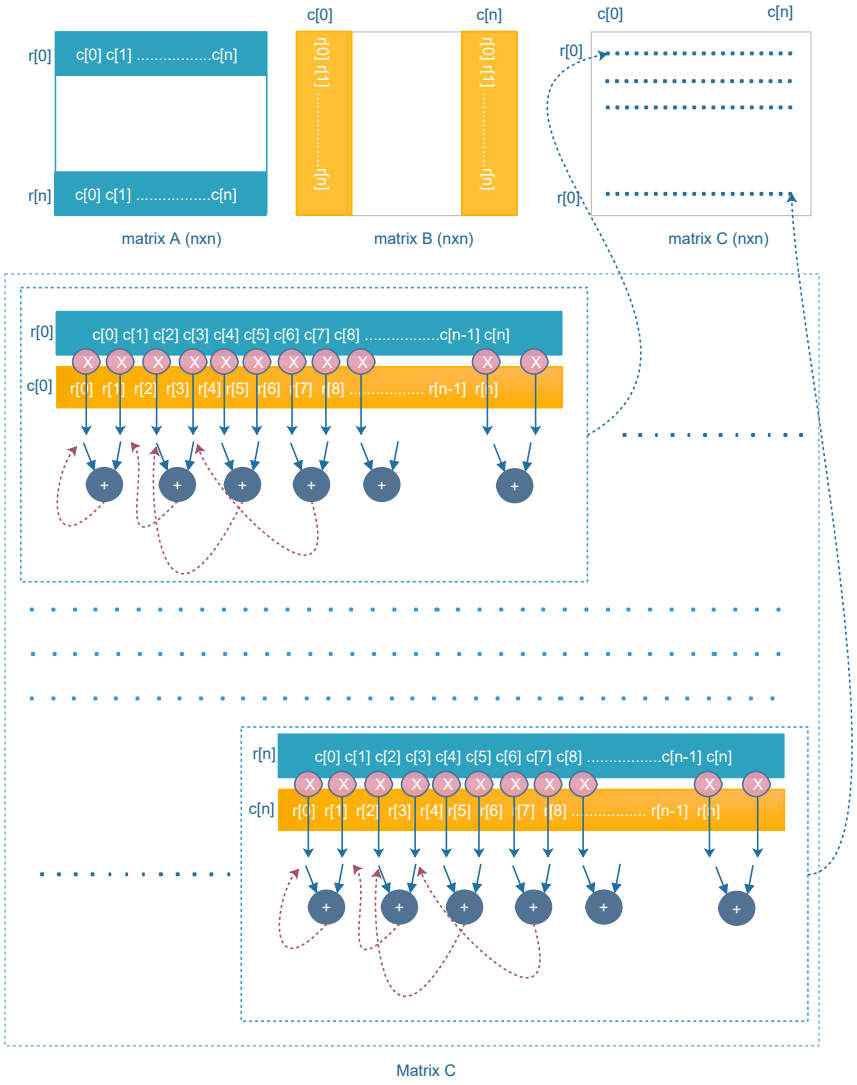


Figure 5.22: Matrix multiplier hardware design

The computational unit mainly contains computational blocks. The computational blocks perform the operation of multiplication and accumulation (MAC). One of the core computations of DNN algorithm is matrix multiplication. In this section hardware design for matrix multiplier is discussed.

Matrix Multiplication V.1: In matrix multiplication between two matrices, matrixA and matrixB, generate an element at $C(i, j)$ in the resultant matrix,

matrixC. The elements of i^{th} row in the first matrixAre multiplied with j^{th} column elements of second matrixAnd the element wise multiplication results are summed together to get the final element in the resultant matrix. In the first attempt of implementing matrix multiplier, for each of the elements in the resultant matrix, matrixC, a number of multiplier and adders are initialized in the hardware. For a $(n \times n)$ matrix, for each element in the resultant matrix, n number of multipliers are initialized and $n/2$ number of adders are initialized. The figure 5.22, shows the hardware architecture. In this implementation, each of the hardware block for the different elements in the output matrix works in parallel. It is mandatory for this design that all the data for matrixA and matrixB are present in the on-chip buffer of FPGA.

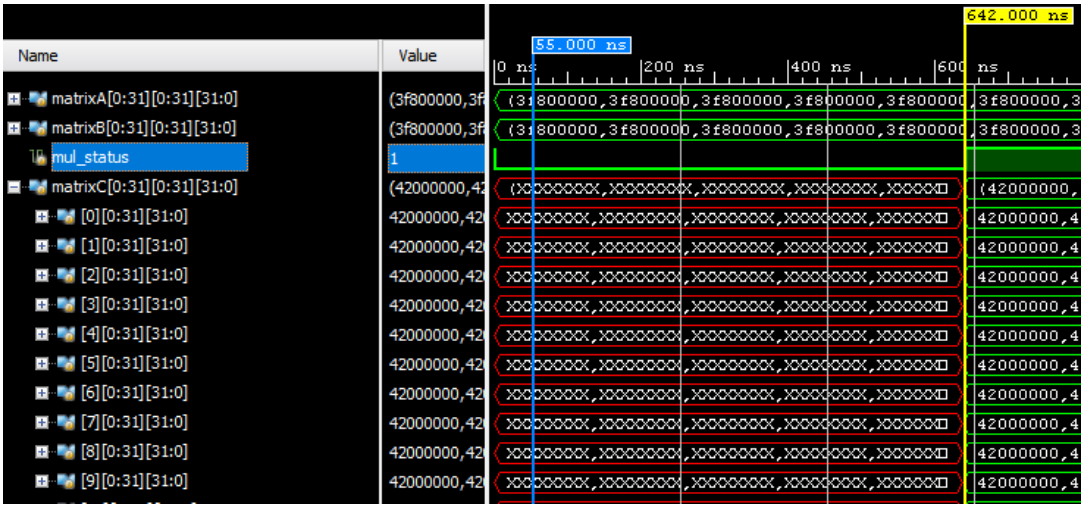


Figure 5.23: Data flow for matrix multiplication

Here, in this timing diagram dot matrix multiplication of two (32×32) matrices, *matrixA* and *matrixB* is performed. All the elements of *matrixA* and *matrixB* have the floating value 1.0 (in IEEE 754 single precision format 0x3f800000). The output status signal *mul_ status* is high when all the hardware blocks finish their jobs. In this case all the blocks finished their work at the same time. The result is buffered in *matrixC*.

To reduce the number of adders that are required to generate final result from the adders, the number is reused by using a buffering arrangement and re-using the adders. In this fashion number of stages required to complete the addition is

$log_2(n)$ where the total number of adders is n .

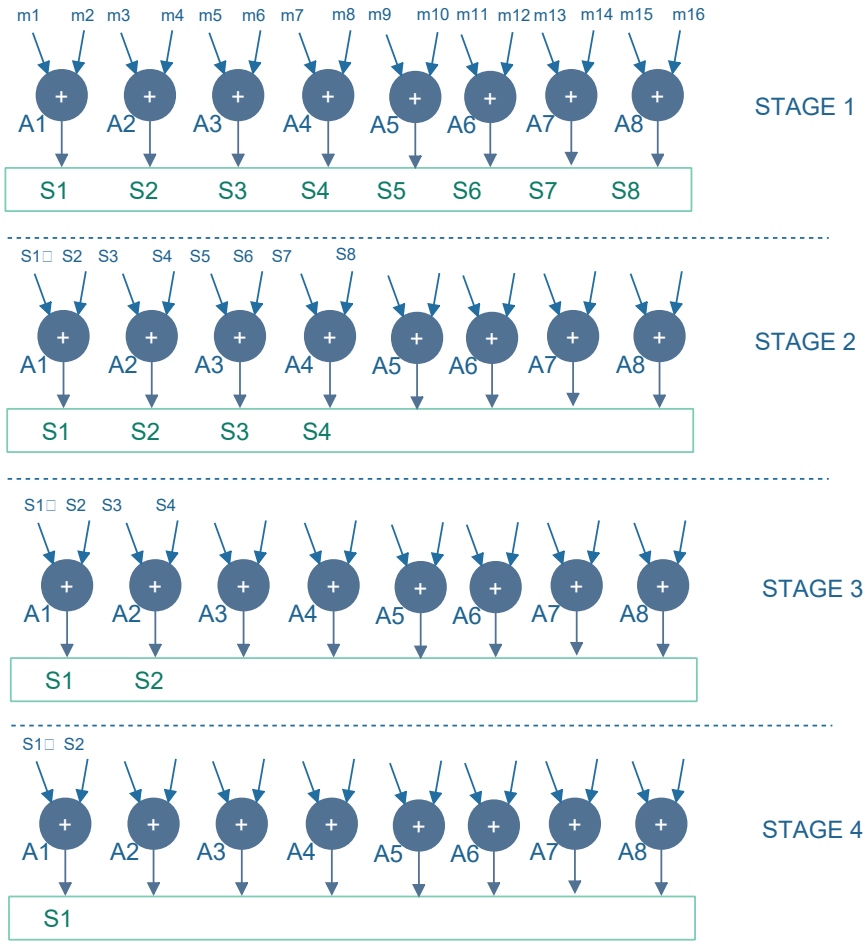


Figure 5.24: Addition stages in adder row

For a row of 8 adders, the stages are shown in the figure. The outputs of earlier stages are passed to the input in the next stages. Here, 8 adders arrangement is shown. For 8 adders total 4 stages are required to get the final summation result form the adders row. In the first stage, the input to the adders, *m1, m2, .. m16* are received from the output of the multipliers. The adders produce output *S1, S2 . . . . S8*. These outputs from stage 1 are passed to the input of the first 4 adders in the second stage. The second stage produces output *S1* through *S4*. These outputs are passed to the inputs of first 2 adders and thus in stage 4 the final result is found from the first adder, *A1's* output. For a $(n \times n)$ matrix, total number of multipliers required is $(n \times n \times n)$ and total number of adders required

is $(n \times n \times n/2)$.

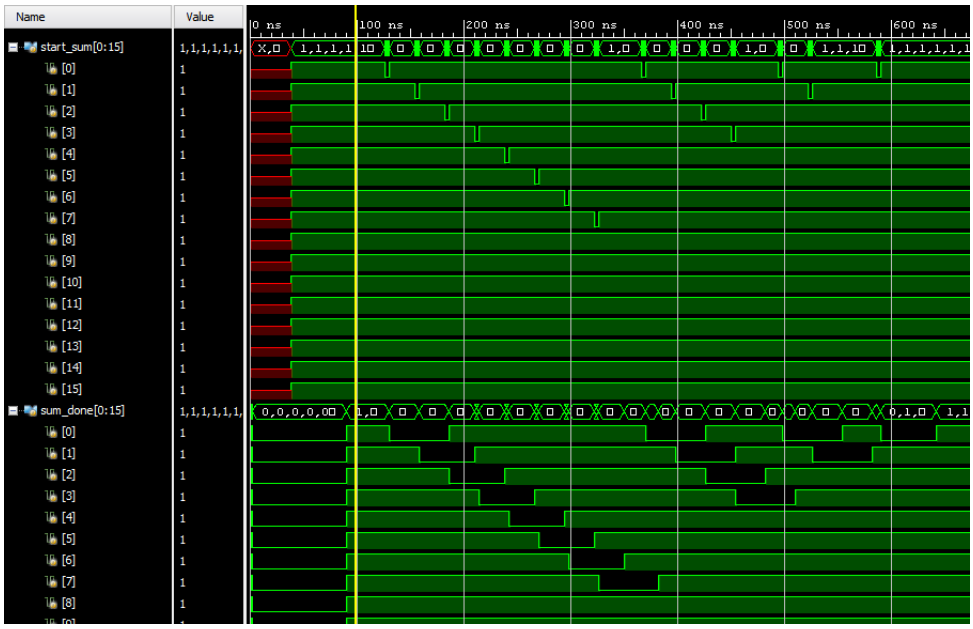


Figure 5.25: Control and status signal of adders

The timing diagram in Figure 5.25 shows the start time and completing of addition computation time for a 16 adders arrangement in a hardware block. Here, total number of stage is 5. The time of starting an adder is indicated by the signal *start_sum*, and the completion time is signaled by *sum_done*. When the *start_sum* signal is high for an adder it starts computation. When the result is available in an adder's output it generates a high signal. When the adders input ports get the data from multipliers output, the first stage starts. For the first stage, all the adders are started at the same time. And the *sum_done* signal is high almost at the same time for all the adders meaning all the adders completes their addition almost at the same time. Then the data are passed from output of the adders and transferred to the input of the adders and second stage starts. The data are passed from the output to the input in a sequential order. When a new input is available in a adder, the adders are made to reset by giving a low pulse in the *start_sum* signal. All the adders are reset in a sequential order one after another. In this fashion, in the first stage all the 16 adders work. In the second stage, 8 adders get reset pulse sequentially, in third stage, 4 adders get reset pulse sequentially, 4^{th} stage get 2 and finally in 5^{th} stage it finishes the computation.

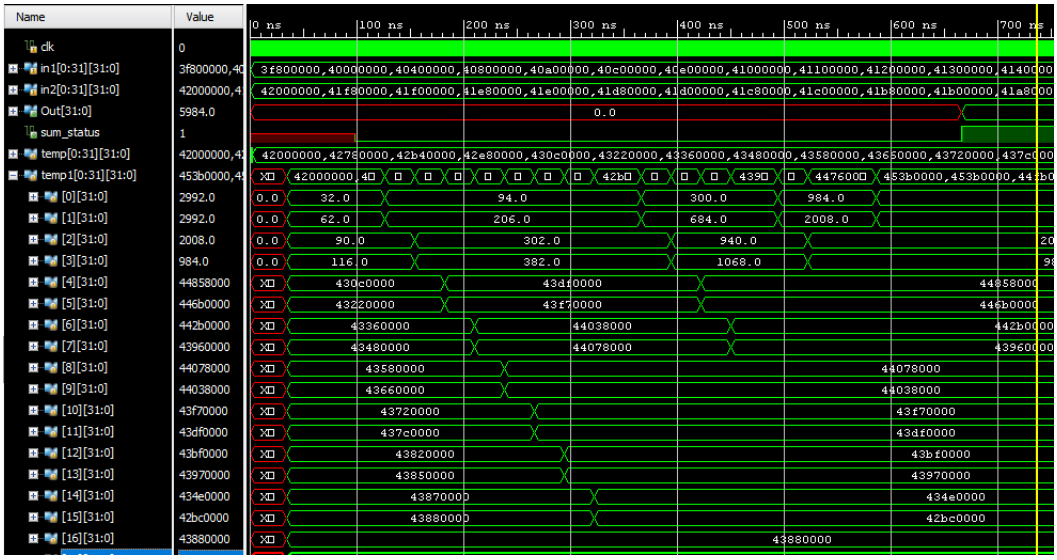


Figure 5.26: Data flow in adders row

In Figure 5.26, data flow in the input of adders are shown. The buffer *temp1* hold the input values for the adders. The buffers *temp1[0]* and *temp1[1]* are the inputs for adder1; similarly the others are also assigned to the rest of the adders.The buffers in1 and in2 hold the input for the hardware blocks. They are mapped to the input of multiplier array. The first input of adder 1 are from the multiplication of first element of the in1 and in2 buffer. The first element in the in1 buffer is 0x3f800000 (1.0 in IEEE 754 format) and the first element in the in2 buffer is 0x42000000 (32.0 in IEEE 754 format). The first input in the adder1 is 32.0 (32.0∗ 1.0). Similarly, the second element in the in1 buffer is 0x40000000 (2.0 in IEEE 754 format) and the second element in the in2 buffer is 0x41f80000 (31.0 in IEEE 754 format). The second input in the adder1 is 62.0 (31.0∗ 2.0). To understand the data flow easily, the radix of the first 4 register buffers are made floating point. When the addition in the first adder (32.0 + 62.0 = 94.0) and the addition in the second adder (90.0+116.0 = 206.0) are completed, the two input buffer registers to adder 1, *temp1[0]* and *temp1[1]* are updated with the output value from the adder 1 and adder 2. So, in the first stage, the contents 0f temp1[0] and temp1[1] are replaced by 0x42bc0000 (94.0) and 0x434e0000 (206.0). In a similar fission, in the first stage, 16 buffer registers in *temp1* are updated in a sequential manner 2 pairs at a time one after another. In the second stage, 8 buffers are updated and

in final stage the first 2 buffers for adder 1 are updated.

Matrix Multiplication V.2:

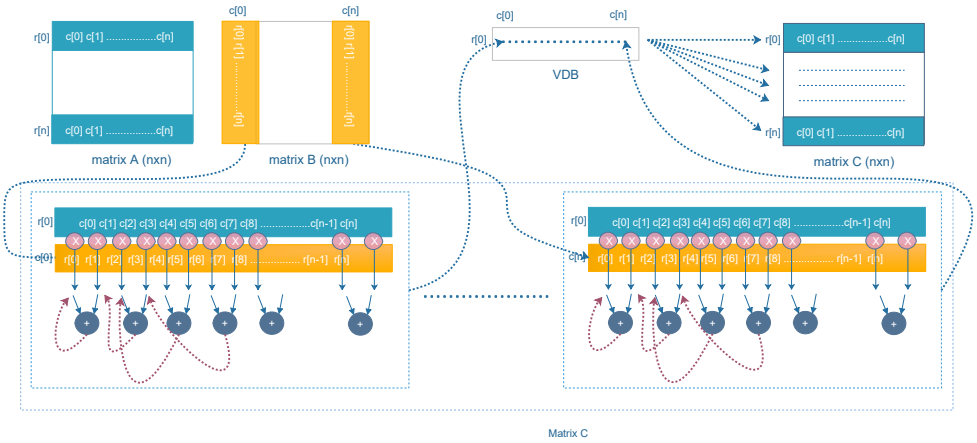


Figure 5.27: Hardware design of matrix-multiplier V2

In the V.1 matrix multiplier design, the number of adders and multipliers required becomes high when n is a big number. All the FPGAs are resource constrained. In most of the cases its impossible to instantiate this large number of adders and multipliers in the design. Keeping in mind that FPGAs are resource constrained the next design has been made. To our best knowledge the existing FPGA community has not applied the technique described here. Instead of having hardware blocks for each of the data points of the output matrix, adders and multipliers are instantiated for one single row. Then data are passed to the same hardware block for calculating the next row of the output matrix. For this implementation its necessary to have data of one multiplicand to be on the memory of FPGA.

The Figure 5.27 shows the basic configuration of the V2 multiplier. It has a computational row (VCB) which contains all the computational hardware blocks. For a $(n \times n)$ matrix multiplication, the computational row contains n number of hardware blocks. Each of the hardware blocks contains n number of multiplier and $n/2$ number of adders. Each of the hardware blocks contains two n dimensional buffers. The buffer contains the row elements and column elements of the two

matrices which are to be multiplied. Elements of the first buffer are multiplied with the elements of the second buffer. The second buffer of the hardware blocks (marked in orange) contains different column elements of the second matrices. The row of the first matrix is copied to all the first buffers of the hardware blocks. With one stage of multiplication and $log_2\,(n/2)$ stages of addition the final result is found in each for each of the blocks. In this design all the hardware blocks work in parallel to get the final result on the computational row. The result is buffered in the computational row. From the computational row buffer, the result is transferred to the store space of matrixC. After completion of the computation for the first row of matrixC, the second row of the matrixA is passed to the second buffer of each of the hardware blocks in the computation row. Then the computation is performed and the output from the computation row is passed to the memory space of matrixC.

In Figure 5.28 matrix multiplication of two 32×32 matrices, maritxA and matirxB is performed. All the elements of matrixA and matrixB have the floating value 1.0 (in IEEE 754 single precision format 0x3f800000). The output status signal *mul_status* is high when all the hardware blocks finish their jobs. The computational row computes for a single row at a time and passes the data to the rows of matrixC sequentially one after another. The result is buffered in matrixC.

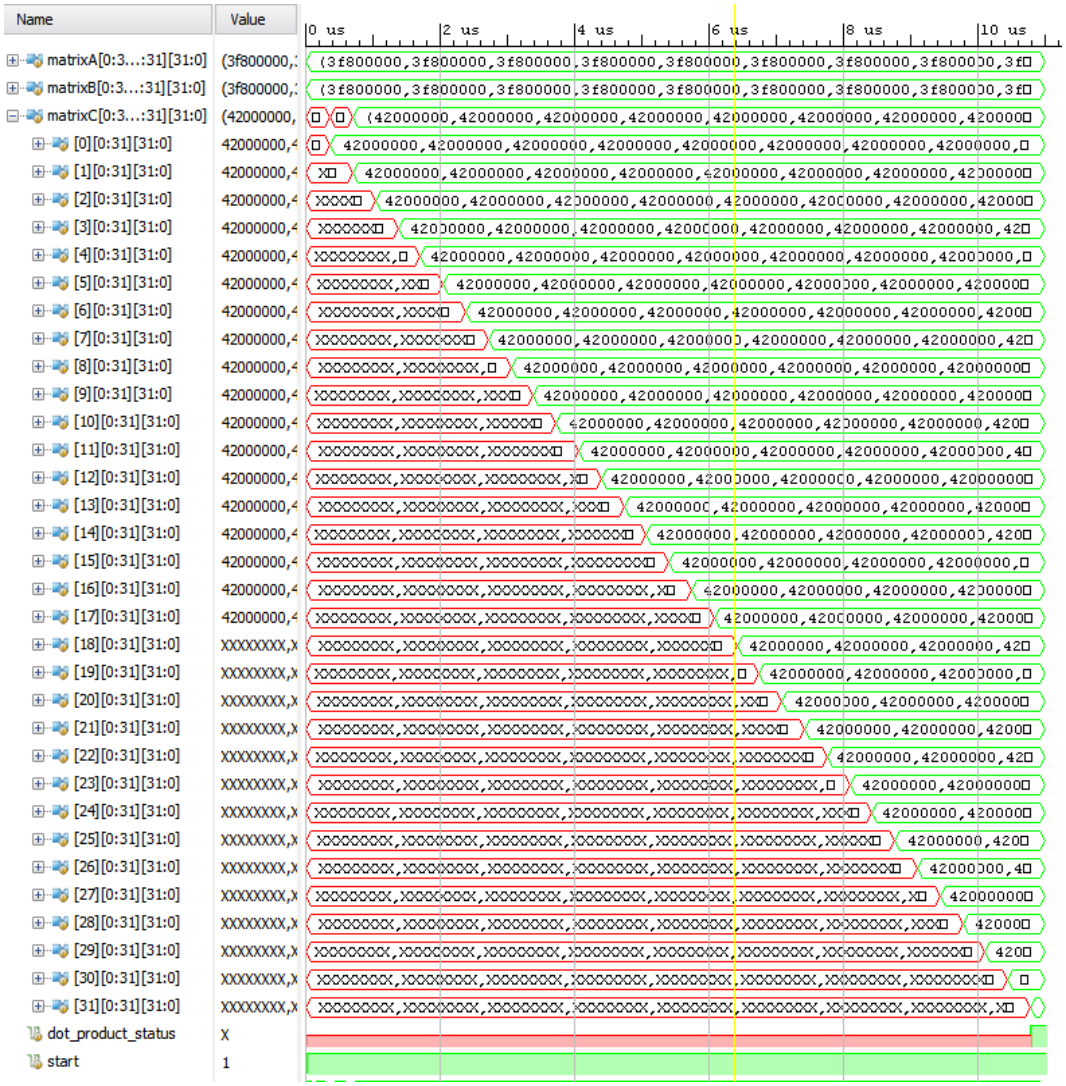


Figure 5.28: Data Flow in Dot Matrix Multiplier V2

There is an encoder block which supplies the correct row elements from matrixA to the hardware block. In this case, matrixA contains 1.0 to 32.0 in the first row, 33.0 to 64.0 in the second row and so forth in the subsequent rows. Depending on the value of the *select_ encoder* output of the encoder block changes. The hardware block has a counter to count which row of matrixA is being used for the current computation. The hardware block sends the *select* signal depending on the value of the counter to the encoder block and receive corresponding row of matrixA. Upon receiving the row elements of matrixA, it passes the data to *in1[0:31][31:0]* buffer and then the hardware block performs computation. When the computation is completed status signal goes high for one cycle and the data in the *out* buffer

get transferred to matrixC.

The second version design of our dot matrix multiplier has improved adders' utilization and arrangement. In the first version, in each stage of addition computation, the adder's enable signal (*start_sum*) were assigned sequentially one after another. A counter is enabled to count the stage and a second counter counts the adder numbers in different stages and resets and enables the adder one after another. In the second version of the design all the counters are enabled and reset at the same time and works parallelly. Thus, a greater parallelism is gained than the first version with in the hardware blocks.

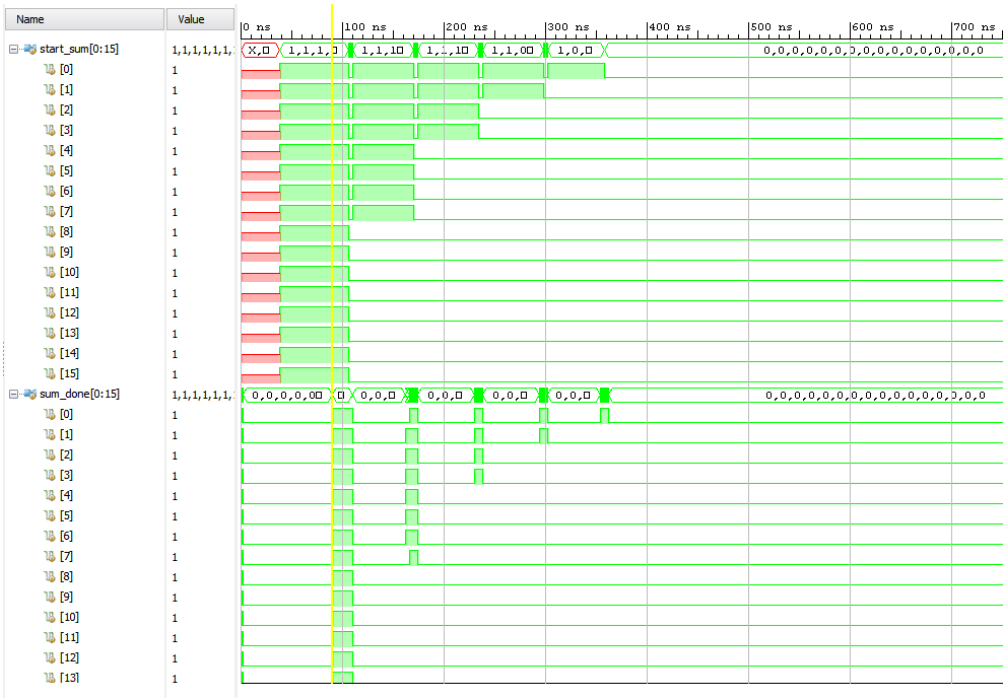


Figure 5.29: Control and status signals for the adders in hardware block

The timing diagram in Figure 5.29 shows the start time and completing of addition computation time for a 16 adders arrangement in a hardware block. Here, the total number of stage is 5. The enable signal of an adder is indicated by the signal *start_sum*, and the completion time is signaled by *sum_done*. When the *start_sum* signal is high for a adder it starts computation. When the result is available in an adder's output it generates a high signal in the *sum_done* bus. When the adders input ports get the data from multipliers output, the first stage

starts. For the first stage, all the adders are started at the same time. And the *sum_ done* is signal is high almost at the same time for all the adders meaning all the adders complete their addition almost at the same time. Then the data are passed from output of the adders and transferred to the input of the adders and second stage starts. The data are passed from the output to the input in a single clock cycle. When a new input is available in an adder, the adders are made to reset by giving a low pulse in the *start_ sum* signal. In the second stage the first 8 adders are reset parallely altogether at the same time. The remaining other 8 adders are made disabled by putting low signal in the *start_ sum* bus. In this fashion, in the first stage all the 16 adders work. In second stage, the remaining 8 adders (0 through 7) get reset pulse sequentially, in third stage, 4 adders (0 through 3) get reset pulse sequentially, 4^{th} stage get 2 adders (0 and 1) and finally in 5^{th} stage the 0^{th} adder gets reset pulse. When the adder completes the computation the *sum_ done[0]* becomes high and finally after transferring the data form *adder[0]* the adder is disabled by putting a low signal in the *start_ sum[0]* register.

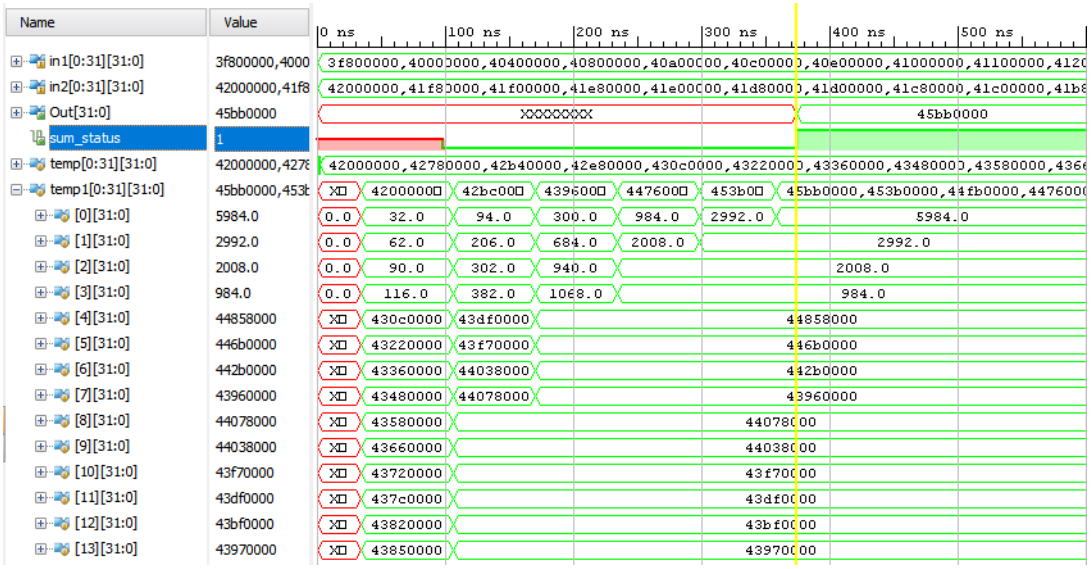


Figure 5.30: Data flow in adders row for V2. matrix multiplier

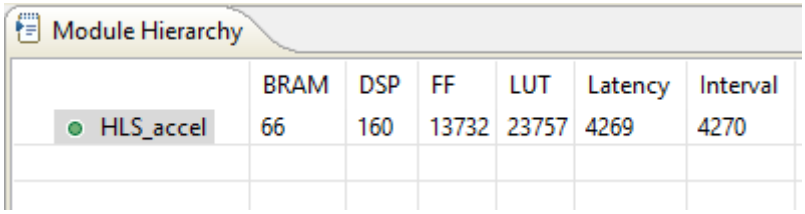
In this timing diagram, data flow in the input of adders are shown. The buffer *temp1* hold the input values for the adders. *temp1[0]* and *temp1[1]* are the inputs for adder1, similarly the others are also assigned to the rest of the adders. in1

and in2 holds the input for the hardware blocks. They are mapped to the input of multiplier array. The first inputs of adder 1 are from the multiplication of first element of in1 and in2 buffer. The first element in the in1 buffer is 0x3f800000 (1.0 in IEEE 754 format) and the first element in the in2 buffer is 0x42000000 (32.0 in IEEE 754 format). The first input in the adder1 is 32.0 (32.0×1.0). Similarly, the second element in the in1 buffer is 0x40000000 (2.0 in IEEE 754 format) and the second element in the in2 buffer is 0x41f80000 (31.0 in IEEE 754 format). The second input in the adder1 is 62.0 (31.0×2.0). To understand the data flow easily, the radix of the first 4 register buffers are made floating point. When the addition in the first adder (32.0 + 62.0 = 94.0) and the addition in the second adder (90.0+116.0 = 206.0) are completed, the two input buffer registers to adder 1, temp1[0] and temp1[1] are updated with the output value from the adder 1 and adder 2. So, in the first stage the, temp1[0] and temp1[1] are replaced by 0x42bc0000 (94.0) and 0x434e0000 (206.0). In a similar fission, in the first stage, 16 buffer registers in *temp1* are updated parallelly at the same time. In the second stage 8 buffers are updated altogether at the same time and in final the first 2 buffers for adder 1 are updated.

VI. DESIGN OPTIMIZATIONS

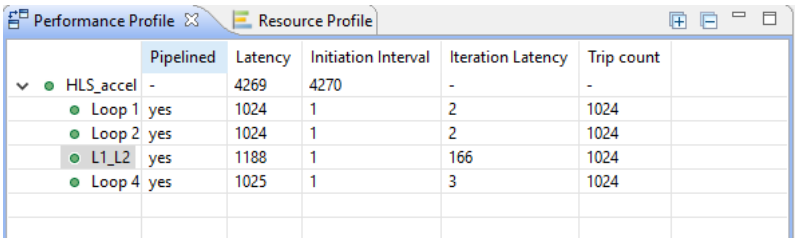
6.1 Optimization in HLS and Resource Hierarchy

For optimized accelerator implementation in SoC, loop pipe-lining optimization in the HSL is used. This is a key performance optimization technique in the HLS flow. This optimization facilitates parallelism across loop iterations. From the accelerator, data is streamed from and into the DDR3 memory. In this streaming process pipe-lining optimization is used. Another loop pipe-lining optimization is used while calculating the dot-product. The resource utilization in Figure 6.1 shows that the optimized PL design is expected to use 160 DSP blocks, 13723 Flip-Flops (FFs), 23757 Look-Up-Tables (LUTs) and it takes 4269 clock cycles to complete the computation.



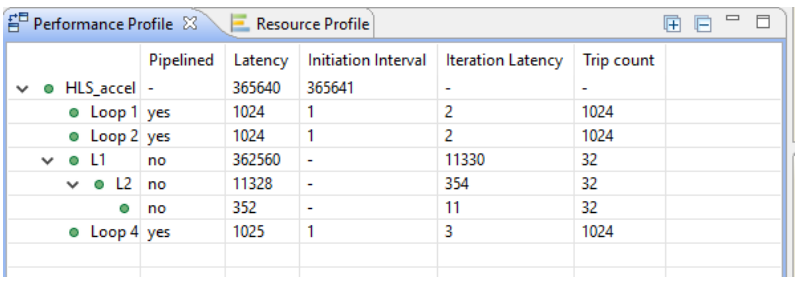
	BRAM	DSP	FF	LUT	Latency	Interval
HLS_accel	66	160	13732	23757	4269	4270

Figure 6.1: Module hierarchy for resource utilization



	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
HLS_accel	-	4269	4270	-	-
Loop 1	yes	1024	1	2	1024
Loop 2	yes	1024	1	2	1024
L1_L2	yes	1188	1	166	1024
Loop 4	yes	1025	1	3	1024

Figure 6.2: Performance profile in optimized pipe-lined design



	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
HLS_accel	-	365640	365641	-	-
Loop 1	yes	1024	1	2	1024
Loop 2	yes	1024	1	2	1024
L1	no	362560	-	11330	32
L2	no	11328	-	354	32
	no	352	-	11	32
Loop 4	yes	1025	1	3	1024

Figure 6.3: Performance profile in unoptimized design

From Figures 6.2 and 6.3, it is observed that the total 365,640 of clock cycles needed by the accelerator in case of the unoptimized accelerator, which is much bigger than for the optimized design with just 4,269 clock cycles. Unpipelined L1, L2 loops are responsible for higher latency in the unoptimized design. Multilayer neural network implementation in the Xilinx Zynq SoC shows about 13x times faster dot-product calculations in the loss function of the neural net, which results in the faster calculation in backpropagation.

6.2 Resource Conscious Matrix Multiplier Design (V3)

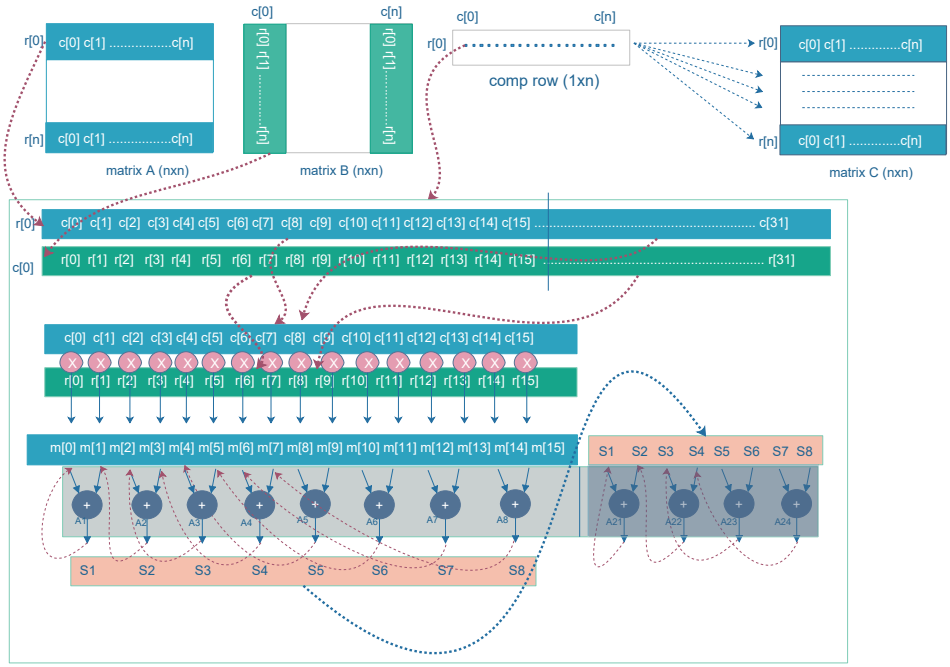


Figure 6.4: Matrix multiplier hardware design V3

In the resource conscious dot matrix multipliers design, the number of multiplier and adders are decreased. In a hardware block which computes for a 32x32 kernel, the number of multipliers and adders in each hardware blocks are reduced to 16 and 12 instead of 32 and 16. Thus, it utilizes less area in the FPGA and also as a result of utilizing less area it consumes less power.

Figure 6.4 shows the basic configuration of the resource conscious multiplier, V3. It has a computational row (*comp row*) which contains all the computational hardware blocks. For a $n \times$n matrix multiplication, the computational row con-

tains n number of hardware blocks. Each of the hardware blocks contains $n/2$ multipliers and $3n/8$ adders. Each of the hardware blocks contains two n dimensional input buffers. The buffer contains the row elements and column elements of the two matrices which are to be dot producted. Elements of the first buffer are multiplied by the elements of the second buffer. The second buffer of the hardware blocks (marked in green) contains different columns elements of the second matrix, matrixB. The row of the first matrix is copied to the first buffer of a hardware blocks. The column elements are copied to the second buffer of a hardware block. The adder row is divided into two parts the first part, the AD1 contains n/4 adders and second part, the AD2 contains n/8 adders. Initially, the first half of the input data is passed to the input of the multipliers. After completing the computation in the multipliers in one cycle, the result is transferred to the AD1 adders input. The result from the AD1 adders output are transferred to the AD2 adders input. In the meantime, the second portion of the input data is transferred to the input buffer of the multipliers. The output of the multiplier row is available to pass to the input of the the AD1 adders. After transferring the data from the AD1 adders output to the AD2 adders input, the AD1 adders row takes data from the output of the multiplier row. The final result is available after two stages of multiplication and $log_2(n/4) + log_2(n/8)$ stages of addition. In this design all the hardware blocks work in parallel to get the final result on the computational row. The result is buffered in the computational row. From the computational row buffer, the result is transferred to the store space of matrixC. After completion of the computation for the first row of matrix C, the second row of matrixA is passed to the second buffer of each of the hardware blocks in the computation row. Then the computation is performed and the output from the computation row is passed to the memory space of matrixC.

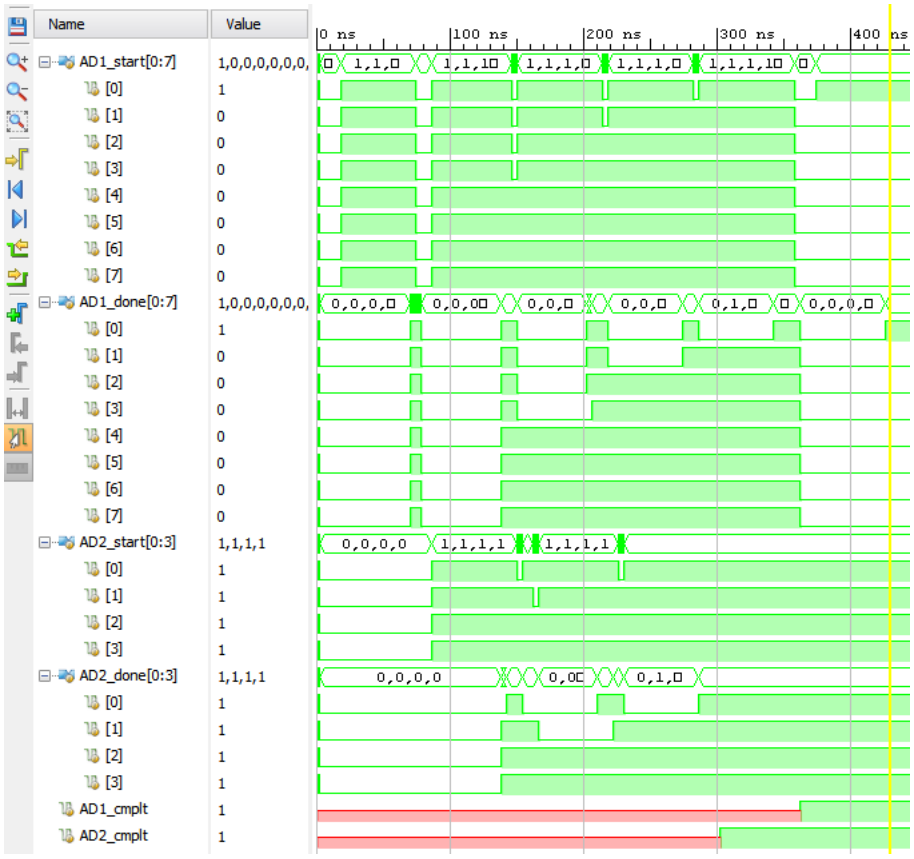


Figure 6.5: Control and Status signal in adders row in V3 hardware block

Figure 6.5 shows the timing diagram of a 32×32 matrix multiplication computation. The timing diagram is generated while computing dot matrix multiplication of the dimension of 32×32. The timing diagram shows the start time and completion time for a 12 adders arrangement (8 in the AD1 and 4 in the AD2 section) in a hardware block. Here, the total number of addition computation stages in the AD1 is 3 and the AD2 is 2. The enable signal of an adder is indicated by the signal *AD1_start* & *AD2_start*, and the completion time is signaled by *AD1_done* & *AD2_done*. The first stage computation begins when the AD1 adders input ports get the data from the multipliers output. For the first stage, all the adders are started at the same time. The *AD1_done* signal is high almost at the same time. When the first stage computation is complete, the result from the output of the AD1 adders are passed to the input of the AD2 adders. The second stage addition starts in the AD1 adders and the first stage computation starts in the AD2 adders. The data are passed from the output to the input in a single clock cycle. When a

new input is available in an adder, the adders are made to reset by giving a low pulse in the *AD1_ start* & *AD2_ start* registers. In the second stage, after receiving the data from the output of the multiplier in the second phase, the adders in the AD1 are reset in parallel. In the third stage, the remaining 4 adders (0 through 3) are reset sequentially, in third stage, 2 adders (0 and 1) are reset in parallel. When the adder completes the computation the *AD1_ done[0]* and *AD2_ done[0]* becomes high. The registers *AD1_ cmplt* and *AD2_ cmplt* are high when both the AD1 adders and the AD2 adders complete the computation.

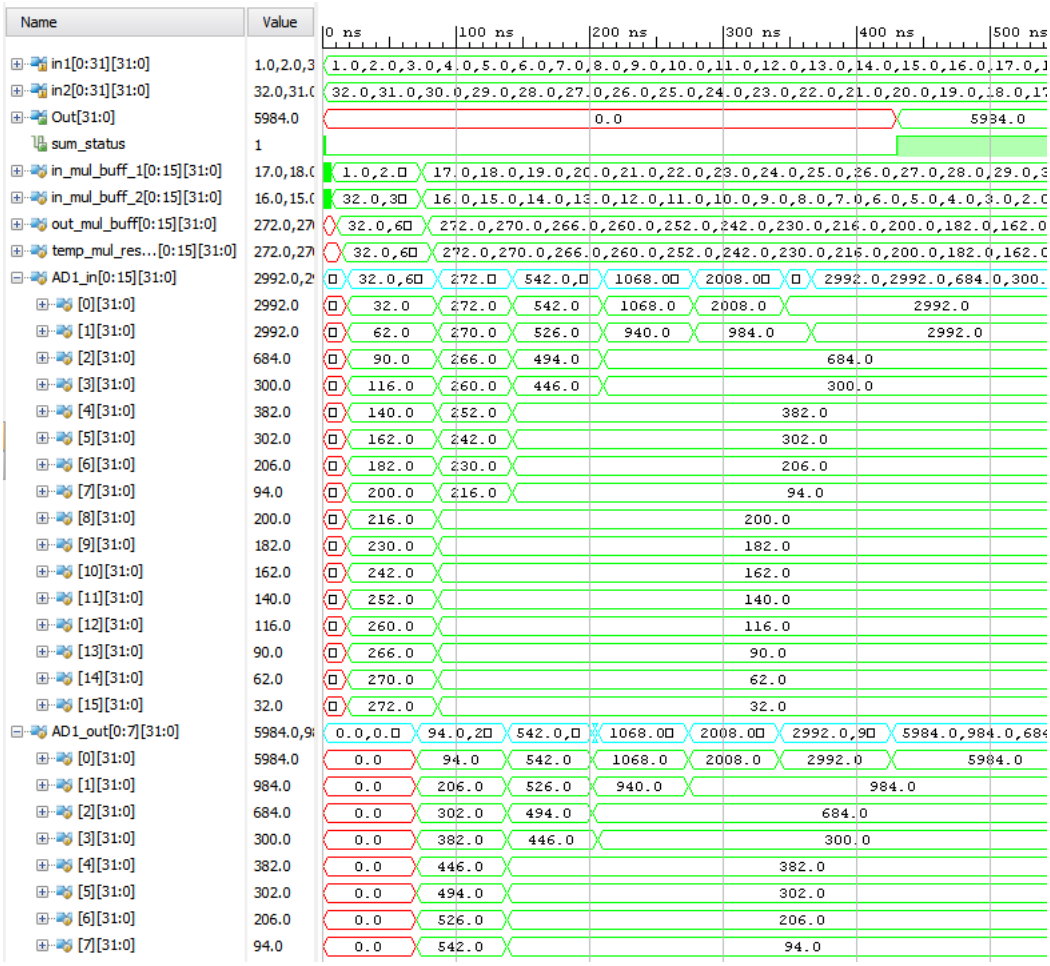


Figure 6.6: Data flow in the AD1 adder row

In Figure 6.6, data flow in the input and output of the AD1 adders are shown. The buffer *AD1_ in* holds the input values for the adders. *AD1_ in[0]* and *AD1_ in[1]* are the inputs for adder1; the remaining adders are assigned in the same way. The buffers *in1* and *in2* hold the input for the hardware blocks. The buffer *in1* con-

tains the floating point numbers 1.0 through 32.0 and *in2* contains the numbers 32.0 through 1.0. In the first phase, the result of multiplication in *in1* and *in2* is fed to the input of adder. The buffers *In_mul_buff_1* and *In_mul_buff_2* are the input buffers for the multiplier row. The multiplier row contains 16 multipliers in the row. The input of the multiplier array receives the data in two phases. In the first phase, the first 16 elements from the buffer *in1* and *in2* are copied to the in_mul_buffer_1 and in_mul_buff_2 respectively. The output from the multipliers are buffered in the buffer *out_mul_buff*. The element wise multiplication of 1.0 through 16.0 from buffer *in1* with 32.0 to 17.0 from buffer *in2* produces the result { 32.0, . . . , 272.0} in the buffer *out_mul_buff* which is transferred to the *AD1_in* buffer. The result of the first state multiplication is produced and buffered in the AD1_out buffer and then which is transferred to the input of the AD2 adders.

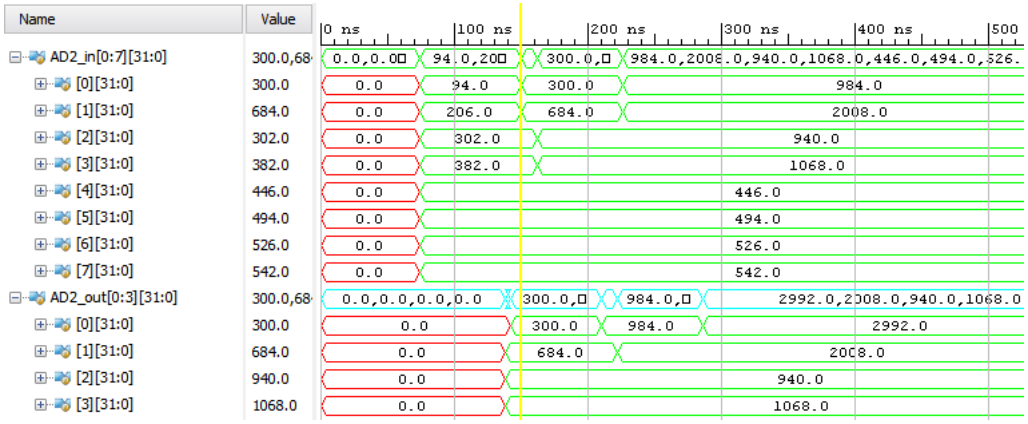


Figure 6.7: Data flow in AD2 adder row

The first stage inputs of adder 1 are 32.0 and 62.0 totalling 94.0, which is assigned to the input of first adder in the AD2. In similar fashion the other inputs of the AD2 adders are assigned. In Figure 6.7, data flow in AD2 is shown. The addition operation in the AD1 adders and in the AD2 goes in parallel and finally the output is received from the *AD1_out[0]* buffer.

6.3 Batched Computation

The designed matrix multipliers in the computational unit described in the earlier chapter utilize a number of resources (LUTs & FFs) in the FPGA. In all designs, when the size of the computational unit increases to support a larger matrix, the resource utilization also increases. However, FPGAs have a fixed and limited number of resources. Due to these resource constraints, it is impossible to instantiate a computational unit in the FPGA beyond a certain limit in the matrix size. But in reality, we may need to compute for a larger size matrix. To serve this purpose, a batched version of matrix multiplication strategy is developed in this section. It is assumed that the FPGA has a computational unit that can compute matrix multiplication with a specific matrix size. Let's define this computational unit as a kernel. The kernel would be used for computing the larger matrix multiplication. For developing this strategy, let's consider two (4×4) matrices, X and Y. The first matrix contains capital letters from A through P and the second matrix contains small letters from a through p. The matrices are shown in Figure 6.8.

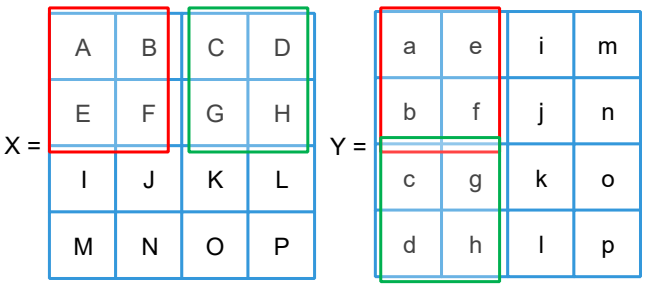


Figure 6.8: Dummy matrices.

Each matrix is subdivided into 4 batches. The first matrix is batched into X_1, X_2, X_3, and X_4; the second matrix is batched into Y_1, Y_2, Y_3, and Y_4. The batched matrices are shown in Figure 6.9.

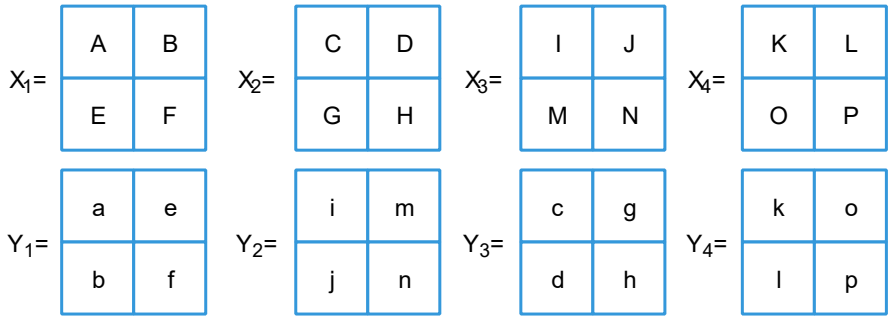


Figure 6.9: Subdivision of matrices.

The result of the matrix multiplication between the two $4\times$ 4 matrices, X and Y, is found by Out matrix shown in Figure 6.10.

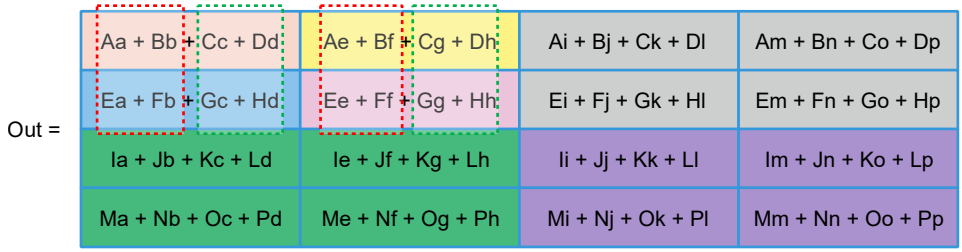


Figure 6.10: Resultant Out matrix.

If we examine the expressions for $Out\,(1,1)$ and $Out\,(2,1)$ elements in the Out matrix, we will find that the $Out\,(1,1)$ element is given by the expression $Aa+Bb+Cc+Dd$ and the $Out\,(2,1)$ element is given by the expression $Ea+Fb+Gc+Hd$. If we group the first two elements in the *(1,1)* expression with the first two elements in the *(2,1)* expression, we will find that this combination matches the result of matrix multiplication of X_1 and Y_1. In the same way, if we group the second two elements in the $Out\,(1,1)$ expression with the first two elements in the $Out\,(2,1)$ expression, we will find out that this matches with the result of matrix multiplication of X_2 and Y_3. The resultant matrix multiplications are shown in Figure 6.11.

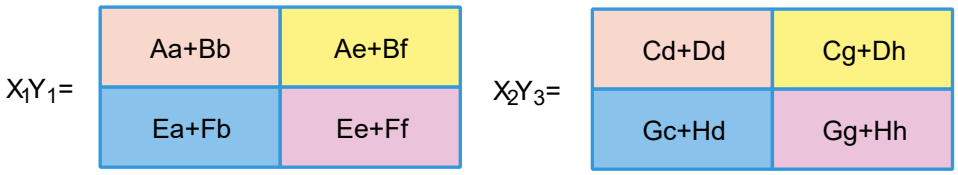


Figure 6.11: Resultant matrix multiplications.

Finally, it turns out that performing element-wise addition between $X_1 Y_1$ and $X_2 Y_3$ generates the 2× 2 matrix whose elements are exactly the same as the elements of the 1st quadrant of the matrix *Out*. The 2nd, 3rd, and 4th quadrant elements are found by element-wise addition between the matrices $X_1 Y_2$ and $X_2 Y_4$, $X_3 Y_1$ and $X_4 Y_3$, $X_3 Y_2$ and $X_4 Y_4$. The elements of the *Out* matrix are shown in Figure 6.12.

Figure 6.12: Elements of *Out* matrix

If the kernel size is $n \times n$ and the matrix size is $N \times N$, then total number of matrix multiplications needed for computing $N \times N$ matrix with $n \times n$ kernel is found by k^3 , where $k = N/n$. The number of additions required for the batched computation is found by $(k-1)\,k^2$.

In batched computation, the multiplicand matrices are chunked into sub-matrices. The total number of sub-matrices is found by $2k^2$. In Table 6.1, number of multiplications and additions for different k is shown. In table 6.2 and 6.3, the distribution of computation is shown.

Table 6.1: Batched matrix multiplication distribution numbers

#k	**# Dot Matrix Multiplication**	**# Addition Stages**
2	8	4
3	27	18
4	64	48
5	125	100
6	216	180
7	343	294
8	512	448

Table 6.2: Matrix multiplication distribution when K = 3

X1Y1 + X2Y4 + X3Y7	X1Y2 + X2Y5 + X3Y8	X1Y3 + X2Y6 + X3Y9
X4Y1 + X5Y4 + X6Y7	X4Y2 + X5Y5 + X6Y8	X4Y3 + X5Y6 + X6Y9
X7Y1 + X8Y4 + X9Y7	X7Y2 + X8Y5 + X9Y8	X7Y3 + X8Y6 + X9Y9

Table 6.3: Matrix multiplication distribution when K = 4

X1Y1 + X2Y5 + X3Y9 + X4Y13	X1Y2 + X2Y6 + X3Y10 + X4Y14	X1Y3 + X2Y7 + X3Y11 + X4Y15	X1Y4 + X2Y8 + X3Y12 + X4Y16
X5Y1 + X6Y5 + X7Y9 + X8Y13	X5Y2 + X6Y6 + X7Y10 + X8Y14	X5Y3 + X6Y7 + X7Y11 + X8Y15	X5Y4 + X6Y8 + X7Y12 + X8Y16
X9Y1 + X10Y5 + X11Y9 + X12Y13	X9Y2 + X10Y6 + X11Y10 + X12Y14	X9Y3 + X10Y7 + X11Y11 + X12Y15	X9Y4 + X10Y8 + X11Y12 + X12Y16
X13Y1 + X14Y5 + X15Y9 + + X16Y13	X13Y2 + X14Y6 + X15Y10 + X16Y14	X13Y3 + X14Y7 + X15Y11 + X16Y15	X13Y4 + X14Y8 + X15Y12 + X16Y16

6.4 Batched Matrix Multiplication Implementation in CAPI

Depending on the availability of resources in an FPGA, the batched matrix multiplication in CAPI can be designed in two different ways. Let's consider the intermediate matrix multiplication results as intermediate matrices. While designing the multiplier, depending on the value of n, k, and the resources in the FPGA, it is determined whether the intermediate matrices will be buffered or not. The first method considers that there is not enough FFs or BRAM available that intermediate matrices and all the input chunks of memory can not be stored into the FPGA. The second method considers there is enough resources available for storing all the intermediate data and input matrices.

6.4.1 Batched Matrix Multiplication V1

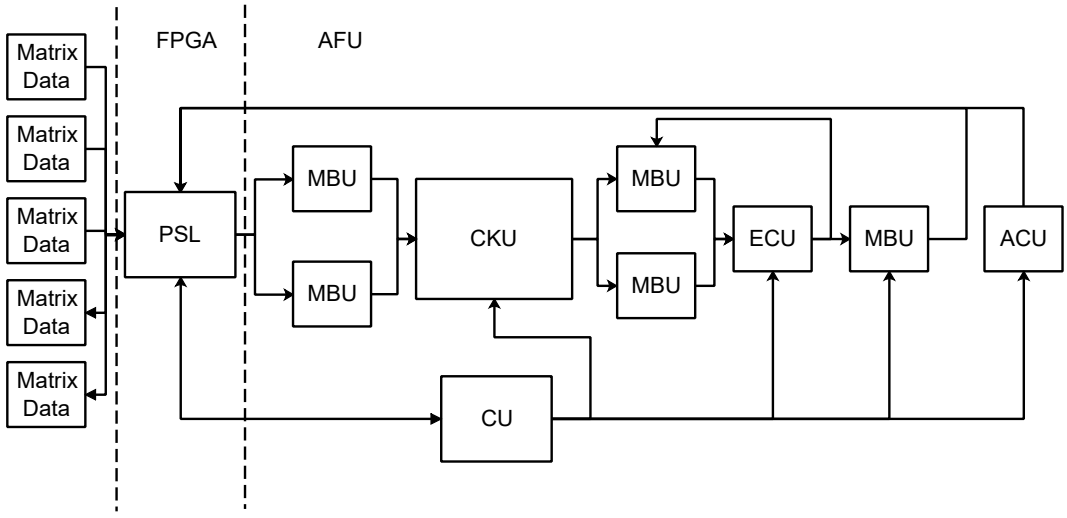


Figure 6.13: Batched computation V1

In the V1 batched computation, it is assumed that the FPGA does not have enough resources to hold entire input matrices and intermediate matrices into the FPGA. In V1, the hardware architecture (Fig. 6.13) inside the AFU consists of computational kernel unit (CKU), a controller unit (CU), multi-ported buffer unit (MBU), extra computational unit (ECU) and an address computing unit (ACU). The CU unit in the AFU controls the data flow and communication between

the AFU and PSL layer. The two buffers, MBUs, for holding the data of input matrices. The CKUs have specialized computational units such as dot block (DB) and ReLu block (RB) for tackling the most common DNN operations. The DBs are the largest computational blocks in the accelerator. An $n \times n$ DB kernel consists of n vector dot block (VDB) units. Each VDB unit computes the vector dot product of two vectors. The CKU unit completes the computation of matrix multiplication. The CU unit controls the batched computation. As in this version there is not enough memory to hold intermediate results, the CU unit uses the PSL's cache unit to read and write the data from and to the system memory while working on the chunks of the data. After computing the intermediate matrix multiplication, data is passed to the EKU unit for element wise multiplication. Depending on the addition stages requirements, the control unit in the AFU requests input data matrices through the PSL layer form the memory. When the addition is completed, the control unit directs the PSL layer to write back the result into the memory in a correct address. As storage buffer is limited the the AFU won't be able to use the same input chunk multiple time. So, the PSL may require reading the same data i.e. X1 several times form the memory. The cache effect in the PSL layer plays an important role in reading data from memory quickly.

The CU is responsible for extracting information from meta-data in WED and maintaining the state of the hardware. The ECU is used in the batched computation for performing addition operations on intermediate data. The ACU calculates the virtual address for reading and writing user space data to- and from- the shared memory.

6.4.2 Batched Matrix Multiplication V2

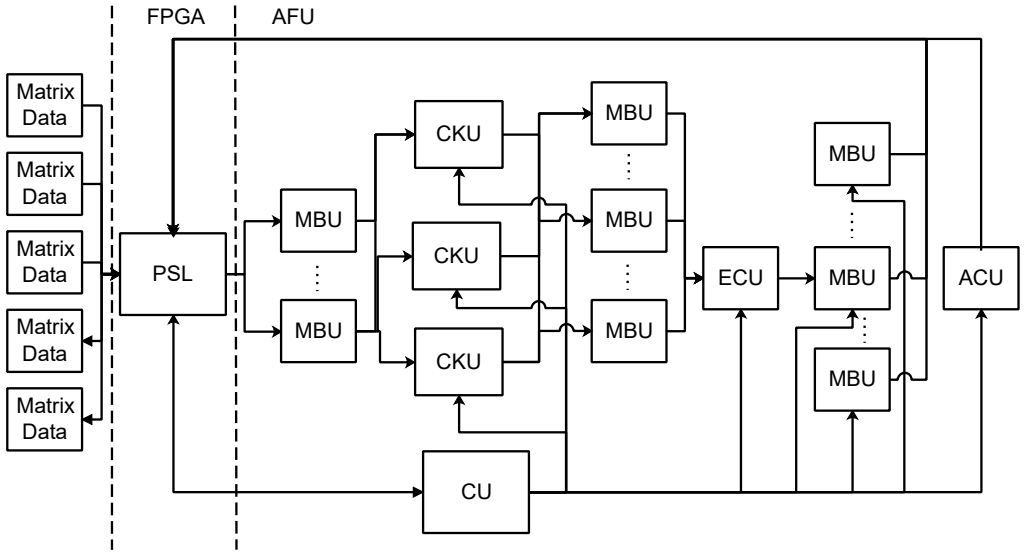


Figure 6.14: Batched computation V2

In the V2 multiplication, it is assumed that the FPGA has enough resources to hold entire input matrices and intermediate matrices into the FPGA. In V2, the hardware architecture (Fig. 6.14) inside the AFU consists of 3 computational kernel units (CKUs), a controller unit (CU), several multi-ported buffer unit (MBUs), extra computational unit (ECU) and an address computing unit (ACU). The CU unit in the AFU controls the data flow and communication between the AFU and PSL layer. The MBUs are implemented as BRAMs for holding the data of input chunk matrices, intermediate matrices and output chunk matrices. The CKU unit has the same functionality as described in the previous section. In this version the CKU units works in parallel on different chunk of input matrices. The CU unit controls the batched computation. As in this version there is enough memory to hold intermediate results, the MBUs holds the intermediate matrix multiplications data. After computing the intermediate matrix multiplications, data is passed to the EKU unit for element wise multiplication. Depending on the addition stages requirements, the CU unit controlls the EKU unit to write the results into the MBUs. From the MBUs data are passed to the PSL to write back to the system

memory.

6.5 Analysis of Batched Computation in V2

In this section we will find out the number of buffers required and execution time for batched computation. We will find out the values in terms of k. For a batched computation having the parameter k, number of batched input sub-matrices are found by k^2. So, to hold all the batched inputs we need $2 \times k^2$ buffers unit. Total number of intermediate matrix multiplication is k^3; we need k^3 buffers to hold the intermediate results. We need k^2 buffers to hold the output chunks. So the total number of buffers required is $3k^2 + k^3$. In Table 6.4, total number of buffer required and computation time for different k is shown.

Table 6.4: Batched computation Analysis

k	**#of int. mul**	**# of add.**	**#of buff for chunks**	**Total buff**	**Matrix size (N)**	**Comp. time(ns)**
2	8	4	12	23	64	3368
3	27	18	27	57	96	11272
4	64	48	48	115	128	26664
5	125	100	75	203	160	52040
6	216	180	108	327	192	89896
7	343	294	147	493	224	142728
8	512	448	192	707	256	213032
9	729	648	243	975	288	303304
10	1000	900	300	1303	320	416040
11	1331	1210	363	1697	352	553736
12	1728	1584	432	2163	384	718888
13	2197	2028	507	2707	416	913992

The KU115 FPGA has 4.5KB 2160 BRAM blocks. If we consider 32×32 size

batched matrices, a chunk-ed batched matrix takes 4.096KB space. So, using the V2 version, without re-utilizing the MBUs, our design can compute batched multiplication of size 352×352 having k=11.

6.6 Variable Size Matrix Multiplication

In the earlier chapters, fixed size matrix multiplier kernels are designed. A $n \times n$ computational kernel works on a $n \times n$ matrix data. But, it is not always the case in DNN computation that the kernel will receive $n \times n$ size matrix data. Depending on the batch size in the DNN layers and number of hidden layers, the matrix size may vary. This section of the thesis discusses the approach to compute a reduced size matrix multiplication in the existing kernel. If we look into a computational row in our designed matrix multiplier, we will find out that it consists of n number of computational blocks which computes the vector dot product of two n dimensional vectors. Figure 6.15 shows a computational block. The buffers, *buf1* and *buf2*, hold the vectors, between which dot product is calculated.

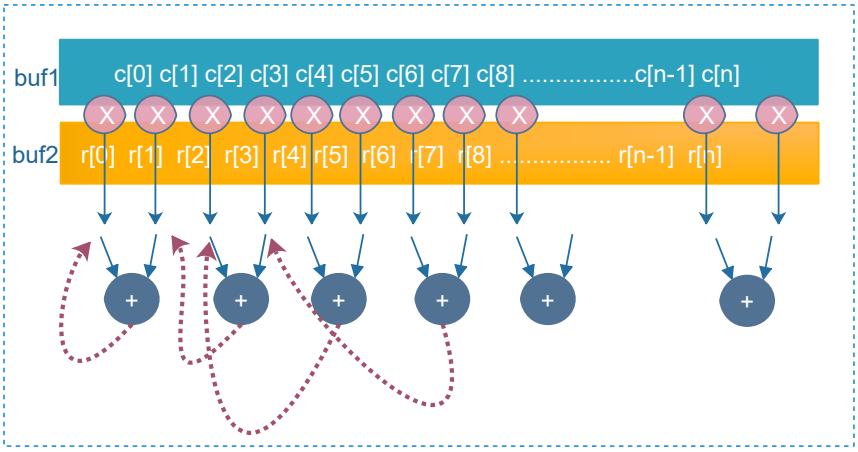


Figure 6.15: Computational block

Let us consider that we are trying to compute matrix multiplication between two matrices of size $m \times m$ where, $m < n$. Then the buffers in the computational units will hold m number of elements. So, $(n - m)$ number of elements will be in unknown logical state. To use the kernel with m elements, our approach is to zero

(0) padding those extra $(n-m)$ elements. So, $(n-m)$ number of elements after the mth element are zero padded in the buffers $buf1$ and $buf2$.

To reduce the latency of zero padding in a buffer in sequential manner, in our design, an *or* gate is used for padding zeros in the buffer in parallel fashion. Performing *or* operation with 0 keeps the buffer unchanged where the data is present and places zeros in other places where the buffer has high z value, or the buffer is in unknown logical state(X). Figure 6.16 shows the zero padding structure with *or* gates.

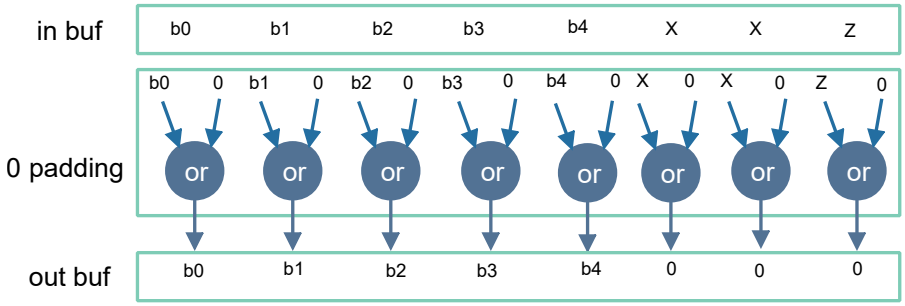


Figure 6.16: Zero padding of elements

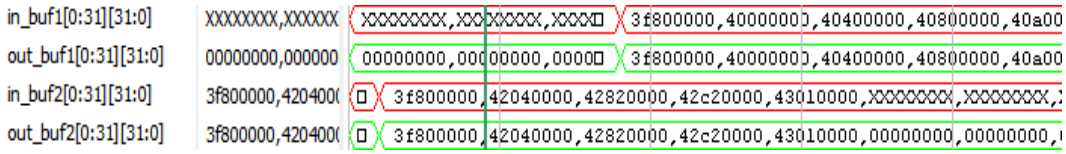


Figure 6.17: Zero padding of elements in simulation

Figure 6.17 shows the zero-padding effect in simulation. The buffer *in_ buf1* takes the non-padded data and produces 0 padded data in the buffer *out_ buf1*. Similarly, in buffer *in_ buf2*, the data that are in unknown stage (X) are made zero by using padding in *out_ buf2*.

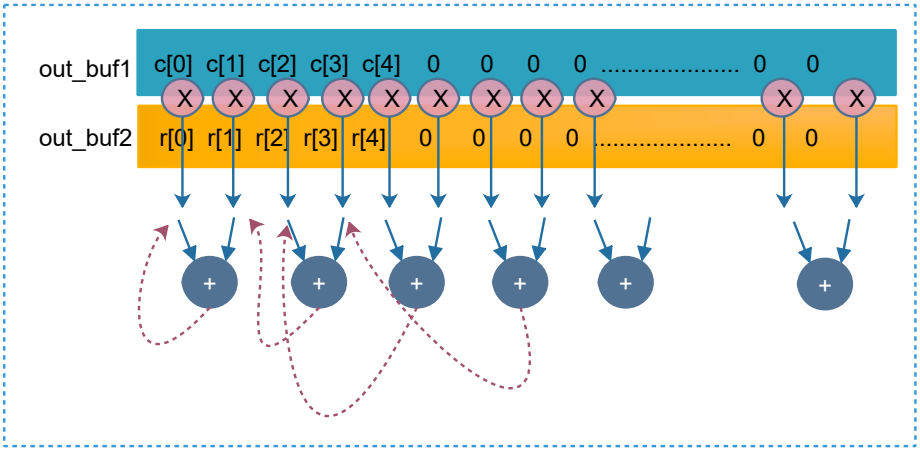


Figure 6.18: Computational block with zero padded data

When the buffers are padded with zero, the computational unit can perform computation on small size matrix with the existing hardware arrangement. In Figure 6.18, the computational block with zero padded buffer is shown. In this arrangement, the buffer *out_ buf1* and *out_ buf2* are the output of zero padding arrangement and these buffers hold the values of the two vectors. In the current arrangement of computational row, if the computational block computes dot product between vectors of size n, then it takes $log2\,(n)$ stages for the adders to compute the output. For a reduced size vector with size m, it will take $\lceil log2\,(m1)\rceil$ stages. Thus, the total computational time decreases in calculating a vector dot product is about $(\ \lceil log2\,(m1)\rceil\ /\ log2\,(n) \times t$ times where, t = latency of 1 vector dot product computation. If m is greater than n/2 then it takes same amount of time as it takes for computing a vector dot product of size n . To compute a $m \times m$ matrix it will take $(n/m) \times (\lceil log2\,(m1)\rceil / log2\,(n)) \times t$ amount of cycles to compute the final resultant matrix.

Let us consider the following matrices. We will find the matrix multiplication of

$$matrixA = \begin{bmatrix} 1.0 & 2.0 & 3.0 & 4.0 & 5.0 \\ 33.0 & 34.0 & 35.0 & 36.0 & 37.0 \\ 65.0 & 66.0 & 67.0 & 68.0 & 69.0 \\ 97.0 & 98.0 & 99.0 & 100.0 & 101.0 \\ 129.0 & 130.0 & 131.0 & 132.0 & 133.0 \end{bmatrix}$$

$$matrixB = \begin{bmatrix} 1.0 & 2.0 & 3.0 & 4.0 & 5.0 \\ 33.0 & 34.0 & 35.0 & 36.0 & 37.0 \\ 65.0 & 66.0 & 67.0 & 68.0 & 69.0 \\ 97.0 & 98.0 & 99.0 & 100.0 & 101.0 \\ 129.0 & 130.0 & 131.0 & 132.0 & 133.0 \end{bmatrix}$$

$$matrixC = matrixA * matrixB$$

$$matrixC = \begin{bmatrix} 1295.0 & 1310.0 & 1325.0 & 1340.0 & 1355.0 \\ 11695.0 & 11870.0 & 12045.0 & 12220.0 & 12395.0 \\ 22095.0 & 22430.0 & 22765.0 & 23100.0 & 23435.0 \\ 32495.0 & 32990.0 & 33485.0 & 33980.0 & 34475.0 \\ 42895.0 & 43550.0 & 44205.0 & 44860.0 & 45515.0 \end{bmatrix}$$

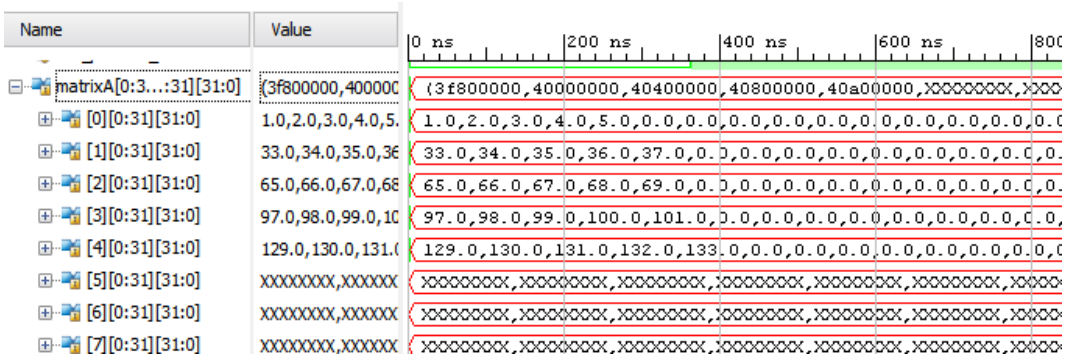


Figure 6.19: matrixA in simulation

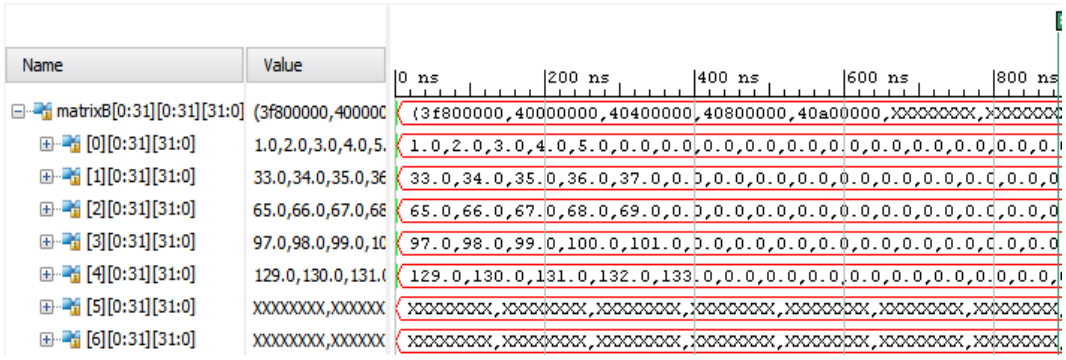


Figure 6.20: matrixB in simulation

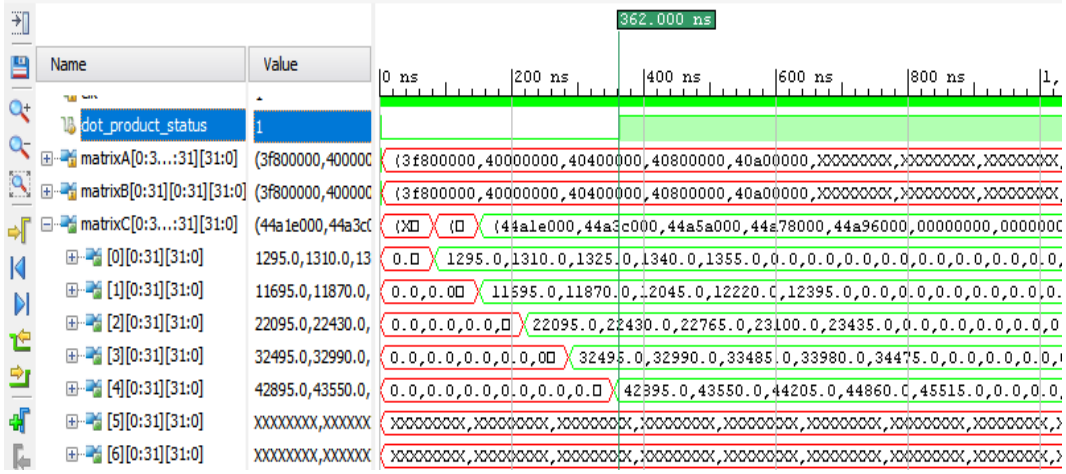


Figure 6.21: matrixC in simulation

The content of *matrixA*, *matriB* and *matrixC* is shown in Figure 6.19, 6.20 and 6.21. It took 362ns for the kernel to compute the multiplication computation. It took about 68ns to compute the elements in each rows of matrix C with the padding arrangement.

6.7 ReLU

As mentioned in chapter IV, a nonlinear function is applied after each FC layer or convolution layer in DNN. These include sigmoid or hyperbolic tangent as well as ReLU. ReLU has become popular in recent years due to its simplicity and ability to enable fast training [25]. We designed a hardware ReLU unit for our accelerator. As in our accelerator, we are working with IEEE 754 format single precision data, the ReLU unit only checks the sign bit and passes the input data to the output if the sign bit is 0 otherwise it replace the input data with 32-bit zeros. Thus it accomplishes the functioning of a ReLU unit.

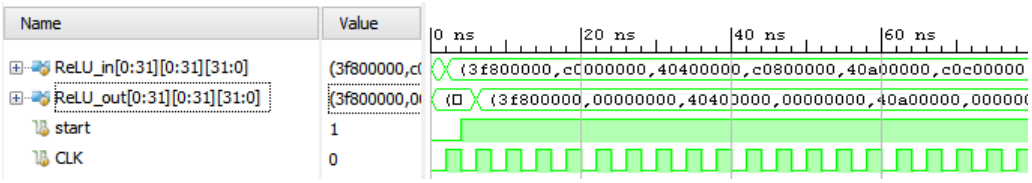


Figure 6.22: Hardware ReLU functioning timing diagram

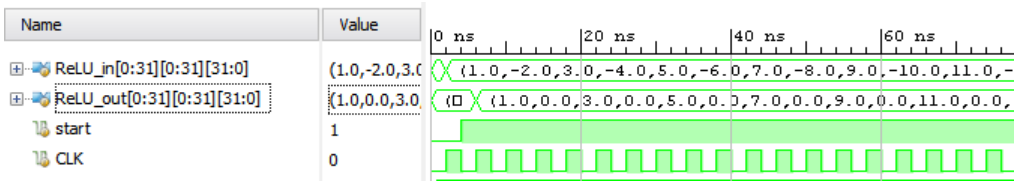


Figure 6.23: Hardware ReLU functioning timing diagram

In Figures 6.22 and 6.23, the functioning of a 32×32 ReLU unit have been shown with HEX and floating-point data. The hardware ReLU unit takes only one clock cycle to complete its functioning.

6.8 PSL-Cache

To facilitate the cache coherent access to the system memory, the PSL and CAPP unit contains cache lines which are used by the AFU. The CAPP unit snoops the SMP fabric on behalf of the PSL. The latency to the fabric to response is the same as other on-chip cache. In batched computational model (discussed in the next chapter), there is an expectation of temporal locality. The AFU-PSL

command interface supports cache enabled reading. We explored the cache enabled reading and found out that the cache enabled reading brings the data from system memory to the AFU with 3x or more speed than the non cache enabled reading.

VII. EVALUATION

7.1 Experimental Setup

We evaluated our algorithm on two platforms. The first FPGA platform is a CAPI-enabled POWER8 system which hosts a Xilinx Kintex Ultrascale 115 FPGA, with xcku-115-flva-2-3 chip on an Alpha Data ADM-PICE-KU115 board with an X16 PCI-Express interface. The FPGA has 663360 CLB LUTs. The second FPGA platform is an SoC system which houses a Xilinx Zynq SoC two ARM cortex A9 processors.

We used Xilinx Vivado's Out of Context (OOC) synthesis and implementation feature for synthesis and route results for some of the modules. Utilizing the OOC feature enabled us to synthesize the RTL design for which the pin numbers exceeds the maximum pin numbers of the FPGA board. For nativity offloading, implementing routing and placement of the architecture, non-project batch mode of design flow has been used by using tcl scripting. To simulate the software-hardware interaction in CAPI, PSLSE implements a client-server model, conceptually similar to distributed system object component (DSOC) [51].

7.2 Validation

- We initialized our design with randomly generated weights and dimensions for fully connected layers and hidden layers. For a fully connected layer, the dimensions are number of input and output.
- We generated the golden output using a reference software implementation that has been previously verified.
- We collected the test output with the accelerated hardware implementation.
- We compared the test output against the golden output collected from the software implementation.

7.3 Resource Utilization

The primary metric to quantify the capacity in FPGA is the CLB utilization. For our design, the CLB utilization is a function of the computational unit in the architecture and data buffers to hold the data. When the design complexity increases to the point that the FPGA's dedicated routing blocks are insufficient, Vivado uses LUTs for routing. Many a time increased CLB utilization can cause a failure to meet the timing constraints. We have designed two kinds of implementation: LUT based and BRAM based. In our design, CLB utilization varies in LUT based and BRAM based design. The BRAM based design fewer CLBs than the LUT based design because in the LUT based design, data are stored in the LUTs.

Table 7.1: Resource utilization config-1

Component Type	**Available**	**Used**	**% Util**
CLB LUTs*	663360	12645	1.46
—LUT as Logic	663360	9668	1.46
—LUT as Memory	293760	0	0.00
CLB Registers	1326720	4290	0.32
—Register as Flip Flop	1326720	3298	0.25
—Register as Latch	1326720	992	0.07
CARRY8	82920	294	0.35
F7 Muxes	331680	164	0.05
F8 Muxes	165840	80	0.05
F9 Muxes	82920	0	0.00
DSP48E2 only	5520	64	1.16
Block RAM Tile	2160	0	0.00
—RAMB36/FIFO*	2160	0	0.00
—RAMB18	4320	0	0.00

We have reported resource utilization of computational block for three kinds of implementation in our design. The three different kinds of computational blocks perform the same arithmetic operations but with different hardware configurations. The first kind of implementation considers adders tree in the design, the

Table 7.2: Resource utilization config-2

Component Type	**Available**	**Used**	**% Util**
CLB LUTs*	663360	7681	1.16
—LUT as Logic	663360	7681	1.16
—LUT as Memory	293760	0	0.00
CLB Registers	1326720	4397	0.33
—Register as Flip Flop	1326720	3901	0.29
—Register as Latch	1326720	496	0.04
CARRY8	82920	248	0.30
F7 Muxes	331680	89	0.03
F8 Muxes	165840	12	<0.01
F9 Muxes	82920	0	0.00
DSP48E2 only	5520	64	1.16
Block RAM Tile	2160	0	0.00
—RAMB36/FIFO*	2160	0	0.00
—RAMB18	4320	0	0.00

second one has one row of adders, and the third one has two rows of adders. Table 7.1, shows the resource utilization for computation block with adder tree configuration. In this section, we will call the three configurations as config-1, config-2, and config-3 respectively.

In Table 7.2 and 7.3, we presented the resource utilization for config-2 and cnfig-3, respectively. Among all the resource types CLB utilization is an important parameter as it plays a vital role in the complexity of the circuit when it is placed and routed in the FPGA. It is observed from the tables that the implementation with adder tree, config-1, utilizes more CLBs than the other two configurations. It utilizes the same number of DSP blocks that the second configuration uses. It is interesting to note that config-3 utilizes fewer CLB units as well as fewer DSP blocks than the other two configurations. This kind of utilization behavior is expected as config-3 uses fewer adders and multipliers and implements more parallelism than the other two configurations.

Table 7.3: Resource utilization config-3

Component Type	**Available**	**Used**	**% Util**
CLB LUTs*	663360	5373	0.81
—LUT as Logic	663360	5373	0.81
—LUT as Memory	293760	0	0.00
CLB Registers	1326720	4397	0.33
—Register as Flip Flop	1326720	3901	0.29
—Register as Latch	1326720	496	0.04
CARRY8	82920	248	0.30
F7 Muxes	331680	89	0.03
F8 Muxes	165840	12	<0.01
F9 Muxes	82920	0	0.00
DSP48E2 only	5520	32	0.58
Block RAM Tile	2160	0	0.00
—RAMB36/FIFO*	2160	0	0.00
—RAMB18	4320	0	0.00

7.4 Power Analysis

We back-annotated the switching activity using Vivado's RTL simulation. Vivado's power estimation flow supports power estimation in-between synthesizing and routing the design. We used both vectors-based and vector-less power estimation in our design flow. The RTL simulation estimates dynamic leakage power consumption at an ambient temperature of 25°C. The power result also includes device static power which measures the power consumption as a result of transistor leakage current. The dynamic design power measures the dynamic power dissipation due to the input data pattern and the design's internal activity, total on-chip power and off-chip power.

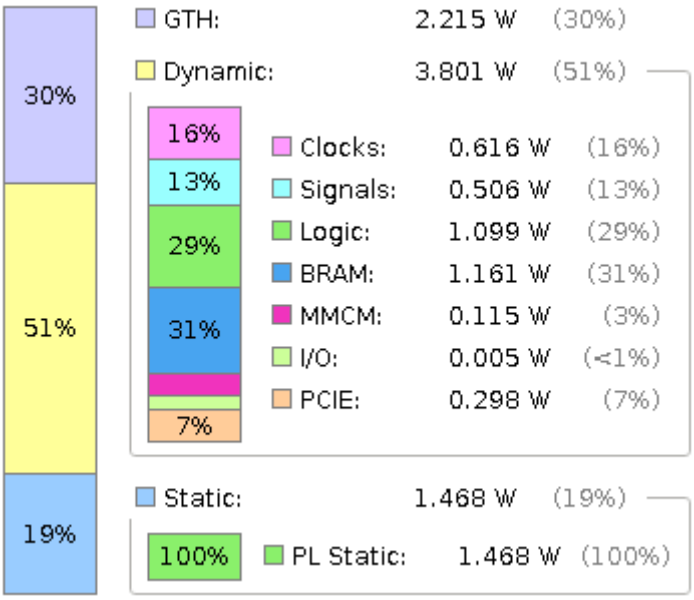


Figure 7.1: Power analysis of PSL-AFU hardware

In Figure 7.1, power analysis summary has been shown for the PSL-AFU architecture. The generated power estimation does not consider the power consumption of the computational unit. The estimation power summary report has been generated by Vivado using the vector-less approach. Total estimated power consumption by the architecture is 7.502 W. About 50% of the estimated total power consumption is dynamic power consumption. One interesting result of the estimated power analysis is the observation that the BRAM portion consumes the most significant portion of the dynamic power. The probable reason is that BRAMs are much larger circuits than the LUTs and powering them at high frequency is quite expensive. Another probable reason is that while the placement and routing process carries on, the BRAM cells requires using many long-distance wires which may dissipate significant amount of energy [52]. The power consumption in the PCI-E bus is estimated as 296 mW which is 7% of total estimated dynamic power consumption.

Table 7.4: Power analysis of 32x32 computation unit hardware

Total On-Chip Power (W)	14.918
Dynamic (W)	13.335
Device Static (W)	1.583

In Table 7.5, estimation of power consumption by a 32x32 computational unit has been shown. The computational unit performs a 32x32 matrix multiplication. The computational unit consists of 32 computational blocks. Each of the computational blocks consists of several adders, multipliers, and several buffer rows. The total estimated on-chip power consumption is 14.918 W, and total dynamic power consumption is 13.335 W.

Table 7.5: Power consumption estimation on on-chip components

On-Chip Components	**Power (W)**
Clocks	0.606
CLB Logic	1.427
—LUT as Logic	1.382
—Register	0.031
—CARRY8	0.013
—F7/F8 Muxes	<0.001
—Others	0.000
Signals	1.034
DSPs	0.261
I/O	10.002
Static Power	1.583
Total	14.912

Table, 7.5, shows the on-chip power consumption estimation on a computational unit. The report is generated from an OOC synthesized architecture. One interesting observation is that a large portion of the total dynamic power is consumed in the I/O bus. As it is an OOC synthesized model, all input and output port declarations of the computational unit got mapped onto the I/O pins. Although, in actual implementation, the module will be instantiated in the PSL and there will be no I/O power cost for the module. So, we can eliminate the I/O power cost from the table. The total estimated power consumption will be 4.91 W. So with a base performance of 23.88 GFLOPS, our design will have 4.863 GFLOPS/Watt performance, and with a base performance of 77.5384 GFLOPS, our design will have 15.791 GFLOPS/Watt performance.

7.5 Timing Analysis and Timing Constraints

A considerable amount of time has been spent to fix the timing violation in the routed and placed design. The precise timing information of a critical path is obtained only after placement and routing (PR). Placement process maps CLBs to a valid location on the cells of the FPGA. The routing process allocates the routing resources to nets to form the necessary wired connection among the CLBs. It takes a considerable amount of time for a synthesis tool to complete the placement and routing. The more complex a design is, the more it is prone to timing violation.

A typical sequential circuit's functionality is governed by some timing constraints, i.e., setup-time and hold-time constraints. The setup-time constraint ensures that no signal arrives at its destined flip-flop after the clock event and hold-time constraint ensures that a signal does not arrive too early at its destination. The setup and hold-time violations are fixed by rerouting the design and inserting buffers to minimize delay of the critical source and sink path. We focused on the critical paths and redesigned the architecture to avoid a timing violation.

The timing report generated by Vivado shows the design improves the worst negative slack (WNS) and total negative slack (TNS). The statistics for timing violation data statistics are shown in Table 7.6, where *Design Name* denotes the different customization of the design to minimize the timing violation. The table reports TNS, WNS along with the TNS Failing Endpoints and the TNS Total Endpoints. One interesting result we observed is that BRAM-based implementation is less prone to timing violation than an LUT-based implementation. When the worst negative slack is negative, the architecture violates the timing constraints. The design V9 meets the timing constraints where WNS is positive, and TNS is zero.

Table 7.6: Timing statistics to optimize and fix timing violations

Design Name	**WNS(ns)**	**TNS(ns)**	**TNS Failing Endpoints**	**TNS Total Endpoints**
v1	-0.069	-8.411	369	342569
v2	-0.741	-3626.4	14757	342607
V3	-0.197	-147.13	2508	345785
V4	-0.741	-3626.4	14757	342607
V5	-0.235	-124.37	1073	215135
V6	-0.132	-16.741	337	277057
V7	-0.92	-4298.9	14366	277051
V7	-0.313	-205.61	2483	277045
V8	-0.238	-90.52	885	218993
V9	0.007	0	0	221062

7.6 Performance Analysis

The purpose of the designed hardware is to provide a performance benefit to the CPU in computing DNN algorithm. The recognized way of measuring computing performance is to measure the floating-point operations per second (FLOPS). The FLOP is defined as either multiplication or addition of single or double precision floating point numbers. The theoretical limit of computation is found by peak FLOP rating which is found by the sum of adders and multipliers multiplied by the maximum operating frequency [53].

The performance time is derived from full-platform RTL behavioral simulation of the architecture in CAPI HW-SW simulation using PSLSE. The design implements both DSP based implementation and non-DSP based implementation. For designing the adders, we have used the blocking and clocked non-blocking adders. The blocking adder's core uses the DSP blocks that are available in the FPGA and non-blocking adder cores are generated using System Verilog code.

In the previous chapters, we have discussed the design of computational blocks which include five types of matrix multipliers for the DNN computation along with the ReLU units. In the first category, the design has one adder and one multiplier row with the un-parallel reset sequence in the adder (V1_unpa). The second category includes the design where it has one adder and multiplier row with parallel reset sequence in the adders (V1_pa). The third and fourth category includes the resource-conscious design. The third one includes two rows of adders and one row of the multiplier with un-parallel (V2_upna) reset sequence and the fourth one with parallel sequence (V2_pa). The fifth one utilizes most resources. It implements one row of multipliers and several rows of adders making an adder tree (V3_tree).

In our design, a computational kernel unit includes 32 computational blocks. Depending on the category of the computational blocks, the number of adder and multiplier units changes. For the first and second category, each of the computa-

tional blocks includes 32 multipliers and 16 adders and for the third and fourth category, each of the computational blocks includes 16 multipliers and 12 adders.

The computation time taken by the computational blocks are shown in the following figures. Figure 7.2 and 7.3 presents the timing diagram with blocking and non-blocking strategy and reports the execution time.

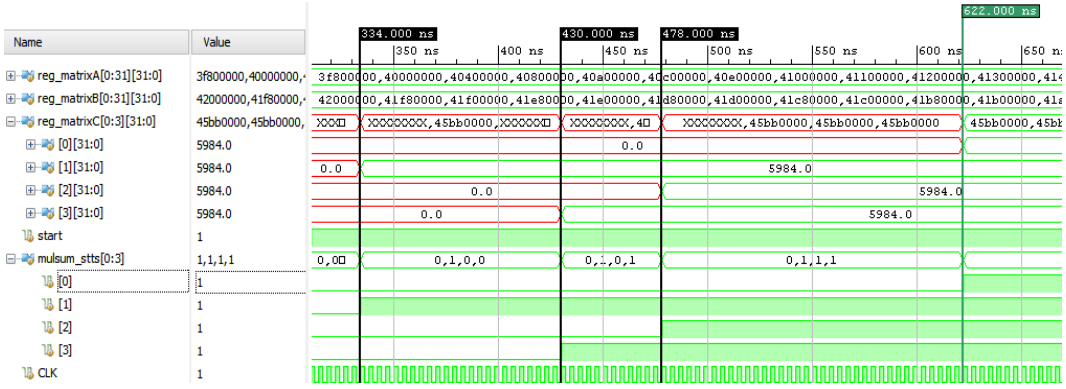


Figure 7.2: Execution timing in non-blocking strategy

In Figure 7.2, the register buffer *reg_matrixA* contains the data of first vector and *reg_matrixB* contains the data of second vector. Both vectors contain 32 single-precision floating-point data. Data of the first vector is 1.0 through 32.0 and data of the second vector is 32.0 down to 1.0. The flag register *mulsum_stts* shows the computation status of the four categories of the computational blocks. The first category with sequential resetting takes the longest time of 622.0 ns and the second category takes the shortest time of 334.0 ns.

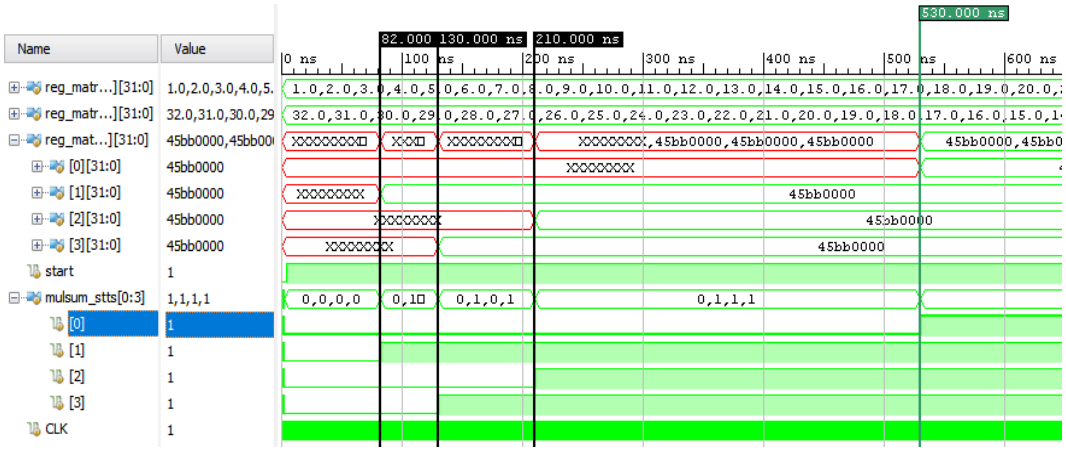


Figure 7.3: Execution timing in blocking strategy

As described in the earlier paragraph, *reg_ matrixA* contains the data of first vector and *reg_ matrixB* contains the data of second vector. The timing diagram is shown in Figure 7.3. Both vectors contain 32 single precision floating point data. Data of the first vector is 1.0 through 32.0 and data of the second vector is 32.0 down to 1.0. The flag register *mulsum_ stts* shows the computation status of the four categories of the computational blocks. The first category with sequential resetting takes the longest time of 530.0 ns and the second category takes the shortest time of 82.0 ns.

The fifth category with adder tree configuration takes 26 ns to finish similar computation.

Table 7.7: Performance with non-blocking implementation

Type	**Execution Time(ns)**	**Peak Performance (GFLOPS)**	**Performance (GFLOPS)**
V1_unpa	622.0	70.7	3.24
V1_pa	334.0	70.7	6.03
V2_unpa	478.0	30.4	4.21
V2_pa	430.0	30.4	4.688

Table 7.8: Performance with blocking implementation

Type	**Execution Time(ns)**	**Peak Performance (GFLOPS)**	**Performance (GFLOPS)**
V1_unpa	530.0	96.0	3.80
V1_pa	82.0	96.0	24.59
V2_unpa	210.0	44.8	9.60
V2_pa	130.0	44.8	15.50
v3_tree	26.0	128	77.54

Table 7.7 and 7.8 shows the theoretical peak and gained performance of different variants of a single computational unit. With the resource utilization shown in Table 7.1, 7.2 and 7.3, we designed out architecture two initiations of V1 and V3 computational units and three initiations of V2 computational units. With multiple initiations, for non-blocking implementations the performance values are shown in Table 7.9.

Table 7.9: Performance with multiple computational unit

Type	**Number of Comp Unit**	**Peak Performance (GFLOPS)**	**Performance (GFLOPS)**
V1_unpa	2	192.0	7.60
V1_pa	2	192.0	49.18
V2_unpa	3	134.4	28.8
V2_pa	3	134.4	46.50
v3_tree	2	256	155.08

7.7 Design Comparison

We compare out design with previous works [54] [55] [56] [57]. Compared with the previous work, our architecture achieved highest performance and has better power efficiency. At present out architecture uses 32-bit floating point data. The projected results of previous implementation with different quantizations are presented in Table 7.10. Theoretically the fixed point operations requires less amount of FPGA resources. Our future plat would be to use fixed point quantization to gain better performance.

Table 7.10: Comparison with previous work

	[54]	[55]	[56]	[57]	Ours
year	2010	2014	2015	2016	2018
Platform	Virtex5 SX240t	Zynq XC7Z045	Virtex7 VX458t	Zynq XC7Z045	CAPI KU115
Clock (MHz)	120	150	100	150	250
Quantization	48-bit fixed	16-bit fixed	32-bit float	16-bit fixed	32-bit float
Performance (GOP/s)	16	23.18	61.62	136.97	155.08
Power(W)	14	8	18.61	9.63	9.82
Power Efficiency (GOP/s/W)	1.14	2.90	3.31	14.22	15.80

VIII. CONCLUSIONS

We implemented a DNN accelerator, which works on both training and inference phases. The accelerator co-operates with the POWER8 CAPI system. The accelerator is designed for implementing both the dot product and the ReLU operation in parallel. The CAPI interface is used for transferring data between the shared memory and the on-chip buffer of the FPGA. The accelerator brings the data from the shared memory, performs computation on the data and writes back the result to the shared memory independently, without putting I/O overhead on the processors. In traditional systems, the SW stack assists the accelerator device each time the hardware reads data from or writes data to the system memory. Consequently, the running application (and the CPU) remains busy communicating with the hardware accelerator while the accelerator performs the memory transactions. In our CAPI-based implementation, the accelerator can perform I/O transactions independently. This allows the SW stack and the CPUs to perform other useful tasks in the application while the HW is busy in I/O transactions. This collaborative model, which improves overall utilization of computing resources, can also be particularly beneficial for heterogeneous supercomputers running large-scale, deep-learning applications. The designed hardware architecture offers better performance/watt than previous implementations. Our optimized design with batched computation model exhibits a high degree of temporal locality, and the use of PSL cache significantly reduces the memory traffic and speeds up the PSL-AFU transmission by 3x.

The accelerators presented in [2] [1] (accelerators which target to accelerate the DNN computation) are similar in operations compared to our design. However, these devices are meant to be used standalone with very lightweight processors to control the flow, and our architecture is designed to work in an environment of a shared memory cluster. Thus, it provides a unique design choice where the concern of storing and moving data is essential.

Currently, our implementation works with FC layers though the hardware is compatible to work on a CONV layer. The future work would be to implement fully functional accelerators to work with CONV as well as FC layer. In this thesis, we work on a randomly generated DNN model. One future goal would be to incorporate our design with established DNN platforms like AlexNet, GoogleNet, etc. Another useful direction of our design is to develop software and hardware stack to accelerate the DNN computation on multiple FPGAs. Collaborative work of multiple FPGAs can effectively increase the throughput of the network. Currently, we are working on 32-bit floating-point data. Recent research has shown that reduced quantization of weight data can achieve nearly state-of-the-art accuracy and performance [58]. Future work will investigate the feasibility of using our accelerator to accelerate the DNN computation with such representation. A recent study [59] showed that the numerical values used in DNN could be compressed significantly by pruning and encoding the non-zero-value indices in sparse weight matrices; the experiment also showed that the storage requirement could be reduced by 35x by using this kind of compression approach. In the future, we will work with sparse representation of DNN values and accelerate the computation of DNN with sparse values in FPGA hardware.

APPENDIX SECTION

Performance Computation

In this section, the calculation for performance gain has been shown. While calculating, the calculation takes into consideration of number of clock cycles required for addition and multiplication operation along with the clock frequency. The calculation also takes into consideration of total number of arithmetic operations performed within execution time. Calculation:

Non-Blocking Computation:

V1_unpa peak:

Let, number of computational block, N_C = 32

For a computational block,

number of multipliers in a row,N_M= 32; multipliers in a row work n parallel;

number of clock cycles required to complete one multiplication operation C_M = 1;

number of adders in a row, N_A = 16; adders in a row work in parallel;

number of clock cycles required to complete one addition operation C_A = 16;

clock frequency, CLK = 250 Mhz;

number of cycles required to get data from the output of the multiplier to the input of the adders and also from the output of adders to the input of adders (adders row is re-utilized); input latency, In_L = 3;

peak performance =

$N\_C \times ((N\_M/(C\_M + In\_L)) + (N\_A/(C\_A + In\_L))) \times CLK$

= 32×((32/(1+3))+(16/(16+3)))×250 MHz = 70736 MFLOPS = 70.7 GFLOPS

V1_unpa :

32× (32+16 + (8+4+2+1))FLOP in 622 ns

= 2016 FLOP in 622 ns

= 3.24 GFLOPS

V1_pa peak:

Same as V1_unpa peak.

V1_pa :

32× (32+16 + (8+4+2+1))FLOP in 334 ns

= 2016 FLOP in 334 ns

= 6.03 GFLOPS

V2_unpa peak:

$32 \times ((16/(1+4)) + (12/(16+4))) \times 250$ MHz = 30400 MFLOPS = 30.40 GFLOPS

V2_unpa:

32× (16 + 16 + 8+ 8 + 4 + 2 + 1 +1 + (4+2+1)) FLOP in 430 ns

= 2016 FLOP in 478 ns

= 4.2175 GFLOPS

V2_pa:

32× (16 + 16 + 8+ 8 + 4 + 2 + 1 +1 + (4+2+1)) FLOP in 430 ns

= 2016 FLOP in 430 ns

= 4.688 GFLOPS

Blocking Computation:

V1_unpa peak:

$32 \times ((32/(1+3)) + (16/(1+3))) \times 250$ MHz = 96000 MFLOPS = 96 GFLPOS

V1_unpa :

32× (32+16 + (8+4+2+1)) FLOP in 530ns

= 2016 FLOP in 530 ns

= 3.80 GFLOPS

V1_pa peak:

Same as V1_unpa peak.

V1_pa :

32× (32+16 + (8+4+2+1)) FLOP in 82ns

= 2016 FLOP in 82 ns

= 24.59 GFLOPS

V2_unpa peak:

$32 \times ((16/(1+4)) + (12/(1+4))) \times 250$ MHz = 44800 MFLOPS = 44.8 GFLOPS

V2_unpa:

32× (16 + 16 + 8+ 8 + 4 + 2 + 1 +1 + (4+2+1)) FLPO in 210 ns

= 2016 FLOP in 210 ns

= 9.6 GFLOPS

V2_pa:

32× (16 + 16 + 8+ 8 + 4 + 2 + 1 +1 + (4+2+1)) FLPO in 130ns

= 2016 FLOP in 130 ns

= 15.50 GFLOPS

V3_tree peak:

32× ((32 + (16 + 8 + 4 + 2 + 1))/2) × 250 MHz = 252000 MFLOPS = 252 GFLOPS

V3_tree:

32× (32 + 16 + 8 + 4 + 2 + 1) FLPO in 26 ns

= 2016 FLOP in 26 ns

= 77.54 GFLOPS

REFERENCES

[1] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM Sigplan Notices*, vol. 49, no. 4, pp. 269–284, 2014.

[2] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards Uniformed Representation and Acceleration for Deep Convolutional Neural Networks," in *Proceedings of the 35th International Conference on Computer-Aided Design*, ser. ICCAD '16. New York, NY, USA: ACM, 2016, pp. 12:1–12:8. [Online]. Available: http://doi.acm.org/10.1145/2966986.2967011

[3] X. Zynq, "7000 all programmable soc overview," *DS190 (v1. 10)[(accessed on 2 November 2016)]*, 2016.

[4] "Accelerating applications in the enterprise with CAPI." [Online]. Available: https://openpowerfoundation.org/blogs/capi-drives-business-performance/

[5] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, "Deepface: Closing the gap to human-level performance in face verification," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 1701–1708.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[7] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 580–587.

[8] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, "Going Deeper with Embedded FPGA Platform for Convolutional Neural Network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16. New York, NY, USA: ACM, 2016, pp. 26–35. [Online]. Available: http://doi.acm.org/10.1145/2847263.2847265

[9] S. Ji, W. Xu, M. Yang, and K. Yu, "3d convolutional neural networks for human action recognition," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 1, pp. 221–231, 2013.

[10] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with cots hpc systems," in *International Conference on Machine Learning*, 2013, pp. 1337–1345.

[11] O. Yadan, K. Adams, Y. Taigman, and M. Ranzato, "Multi-gpu training of convnets," *arXiv preprint arXiv:1312.5853*, 2013.

[12] K. Yu, "Large-scale deep learning at baidu," in *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. ACM, 2013, pp. 2211–2212.

[13] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2015, pp. 1–9.

[14] N. Suda, 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*, ser. FPGA '16. New York, NY, USA: ACM, 2016, pp. 16–25. [Online]. Available: http://doi.acm.org/10.1145/2847263.2847276

[15] S. I. Venieris, A. Kouris, and C.-S. Bouganis, "Toolflows for mapping convolutional neural networks on fpgas: A survey and future directions," *arXiv preprint arXiv:1803.05900*, 2018.

[16] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to FPGAs," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2016, pp. 1–12.

[17] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra *et al.*, "Can fpgas beat gpus in accelerating next-generation deep neural networks?" in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 5–14.

[18] D. Floreano and C. Mattiussi, *Bio-inspired artificial intelligence: theories, methods, and technologies*. MIT press, 2008.

[19] D. H. Hubel and T. N. Wiesel, "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex," *The Journal of Physiology*, vol. 160, no. 1, pp. 106–154.2, Jan. 1962. [Online]. Available: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1359523/

[20] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological Cybernetics*, vol. 36, no. 4, pp. 193–202, Apr. 1980. [Online]. Available: https://link.springer.com/article/10.1007/BF00344251

[21] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85–117, Jan. 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893608014002135

[22] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[23] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015. [Online]. Available: http://www.nature.com/nature/journal/v521/n7553/full/nature14539.html?foxtrotcallback=true

[24] D. Decoste and B. Schölkopf, "Training invariant support vector machines," *Machine learning*, vol. 46, no. 1-3, pp. 161–190, 2002.

[25] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[26] L. Deng, J. Li, J.-T. Huang, K. Yao, D. Yu, F. Seide, M. Seltzer, G. Zweig, X. He, J. Williams *et al.*, "Recent advances in deep learning for speech research at microsoft," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, 2013, pp. 8604–8608.

[27] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[28] M. Abadi, 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 *OSDI*, vol. 16, 2016, pp. 265–283.

[29] Y. Jia, 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*, ser. MM '14. New York, NY, USA: ACM, 2014, pp. 675–678. [Online]. Available: http://doi.acm.org/10.1145/2647868.2654889

[30] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley, and Y. Bengio, "Theano: new features and speed improvements," *arXiv preprint arXiv:1211.5590*, 2012.

[31] R. Collobert, C. Farabet, and K. Kavukcuoğlu, "Torch: Scientific computing for luajit," in *NIPS Workshop on Machine Learning Open Source Software*, 2008.

[32] F. Seide and A. Agarwal, "Cntk: Microsoft's open-source deep-learning toolkit," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2016, pp. 2135–2135.

[33] V. Dang and K. Skadron, "Acceleration of Frequent Itemset Mining on FPGA using SDAccel and Vivado HLS," in *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Jul. 2017, pp. 195–200.

[34] I. Stamoulias and E. S. Manolakos, "Parallel architectures for the knn classifier design of soft ip cores and fpga implementations," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 2, p. 22, 2013.

[35] H. M. Hussain, K. Benkrid, H. Seker, and A. T. Erdogan, "FPGA implementation of K-means algorithm for bioinformatics application: An accelerated approach to clustering Microarray data," in *2011 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, Jun. 2011, pp. 248–255.

[36] M. Papadonikolakis and C. S. Bouganis, "A Heterogeneous FPGA Architecture for Support Vector Machine Training," in *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2010, pp. 211–214.

[37] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh, "TABLA: A unified template-based framework for accelerating statistical machine learning," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Mar. 2016, pp. 14–26.

[38] M. Ito and M. Ohara, "A power-efficient fpga accelerator: Systolic array with cache-coherent interface for pair-hmm algorithm," in *Low-Power and High-Speed Chips (COOL CHIPS XIX), 2016 IEEE Symposium in*. IEEE, 2016, pp. 1–3.

[39] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?" in *Advances in neural information processing systems*, 2014, pp. 3320–3328.

[40] A. P. U. APU, "Zynq-7000 all programmable soc overview," 2012.

[41] S. M. Loo, B. E. Wells, N. Freije, and J. Kulick, "Handel-C for rapid prototyping of VLSI coprocessors for real time systems," in *Proceedings of the Thirty-Fourth Southeastern Symposium on System Theory (Cat. No.02EX540)*, 2002, pp. 6–10.

[42] D. Pellerin and S. Thibault, *Practical Fpga Programming in C*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2005.

[43] J. I. Villar, J. Juan, M. J. Bellido, J. Viejo, D. Guerrero, and J. Decaluwe, "Python as a hardware description language: A case study," in *2011 VII Southern Conference on Programmable Logic (SPL)*, Apr. 2011, pp. 117–122.

[44] J. Xu, N. Subramanian, A. Alessio, and S. Hauck, "Impulse C vs. VHDL for Accelerating Tomographic Reconstruction," in *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2010, pp. 171–174.

[45] G. Lhairech-Lebreton, P. Coussy, and E. Martin, "Hierarchical and Multiple-Clock Domain High-Level Synthesis for Low-Power Design on FPGA," in *2010 International Conference on Field Programmable Logic and Applications*, Aug. 2010, pp. 464–468.

[46] F. B. Muslim, A. Qamar, and L. Lavagno, "Low power methodology for an ASIC design flow based on high-level synthesis," in *2015 23rd International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, Sep. 2015, pp. 11–15.

[47] "Openpower." [Online]. Available: https://openpowerfoundation.org/

[48] B. Xu, N. Wang, T. Chen, and M. Li, "Empirical evaluation of rectified activations in convolutional network," *arXiv preprint arXiv:1505.00853*, 2015.

[49] W. Liu, Y. Wen, Z. Yu, and M. Yang, "Large-margin softmax loss for convolutional neural networks." in *ICML*, 2016, pp. 507–516.

[50] J. Stuecheli and et al., "CAPI: A coherent accelerator processor interface," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 7–1, 2015.

[51] P. G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, D. Lyonnard, O. Benny, B. Lavigueur, D. Lo, G. Beltrame, V. Gagné *et al.*, "Parallel programming models for a multiprocessor soc platform applied to networking and multimedia," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 7, pp. 667–680, 2006.

[52] T. Xie, V. Dang, J. Wadden, K. Skadron, and M. Stan, "Reapr: Reconfigurable engine for automata processing," in *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*. IEEE, 2017, pp. 1–8.

[53] M. Parker, "Understanding peak floating-point performance claims," *Technical White Paper WP-012220-1.0*, 2014.

[54] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 247–257.

[55] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 g-ops/s mobile coprocessor for deep neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2014, pp. 682–687.

[56] C. Zhang, 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*. ACM, 2015, pp. 161–170.

[57] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016, pp. 26–35.

[58] M. Courbariaux, Y. Bengio, and J.-P. B. David, "Training deep neural networks with binary weights during propagations. arxiv preprint," *arXiv preprint arXiv:1511.00363*, 2015.

[59] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.