Mixed-Precision Architecture for Flexible Neural Network Accelerators

by

Driss Hafdi

B.S., Massachusetts Institute of Technology (2018)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2019

Author .....
Department of Electrical Engineering and Computer Science
August 23, 2019

Certified by .....
Song Han
Assistant Professor, MIT EECS
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Mixed-Precision Architecture for Flexible Neural Network Accelerators

by

Driss Hafdi

Submitted to the Department of Electrical Engineering and Computer Science
on August 23, 2019, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Model quantization provides considerable latency and energy consumption reductions while preserving accuracy. However, the optimal bitwidth reduction varies on a layer by layer basis. This thesis suggests a novel neural network accelerator architecture that handles multiple bit precisions for both weights and activations. The architecture is based on a fused spatial and temporal micro-architecture that maximizes both bandwidth efficiency and computational ability. Furthermore, this thesis presents an FPGA implementation of this new mixed precision architecture and it discusses the ISA and its associated bitcode compiler. Finally, the performance of the system is evaluated on a Virtex-9 UltraScale FPGA by running state-of-the-art neural networks.

Thesis Supervisor: Song Han
Title: Assistant Professor, MIT EECS

Disclaimer

The work in this thesis was done in collaboration with another student, Yujun Lin. Most of the architecture in this thesis was developed during our discussions and brainstorming sessions. We both share credit for the design and system architecture. My responsibility extended beyond the design to the FPGA implementation and to the bit code compiler.

Acknowledgments

I would like to thank my thesis supervisor Song Han for giving me the opportunity to join his group and pursue the work presented in this thesis. I'm very grateful for all the guidance and support provided, and feel lucky to have been part of this community.

I would also like to thank Yujun for introducing me to this project and for all the help he has provided me throughout. I could not have imagined a better partner and friend in completing this research. I am also grateful for the assistance and friendship from other members of Hanlab. In particular, I thank Hanrui and Kwan for their cooperation and eagerness to help.

Thank you to all of my friends here at MIT and more particularly at Number Six for their support and for making this place feel like home. I thank Sule for carrying me through this work and constantly pushing me to do my best. I also dedicate a special thank you to my friends Edgar, Antonis, Jeremiah, Manos and Veronica for cheering me up when things looked bleaker. I could not have done it without all of you.

Lastly, I would like to thank my family for their ever-present support and love. I thank my mother for always believing in me and pushing me to better myself. I could not have done it without you. I also thank my sisters Selma and Ines and my cousin Abdou for their invaluable support. Finally, I want to reserve a very special thank you to my father, who even though is no longer here with us, continues to inspire me day after day. This work is the culmination of the path you have set me in, and I will always be grateful for that.

Contents

List of Figures

List of Tables

Chapter 1

Mixed-Precision Architecture

1.1 Introduction

The deployment of state-of-the-art neural networks to edge devices is hindered by their high computational and memory requirements, energy consumption, and latency. These issues have caused a resurgence in the interest of application-specific architectures [32]. New architectures opened the door to a wide variety of optimization routes. Attempts at reducing neural network (NN) models' size and computational complexity are particularly promising. Notably, recent attempts to quantize the weights and activations of NN models to lower precisions have shown to incur no loss of accuracy [10][16][28]. However, different layers have different redundancy; achieving constant accuracy requires the use of bitwidths that vary from layer to layer. Hence, there is a growing need to develop specialized hardware that can leverage these mixed precisions in a way that is currently impossible using state of the art CPUs or GPUs.

A primary limitation of the currently available accelerators that can leverage these variable bitwidths is their high energy consumption and resource utilization. Also, these designs suffer from high and inefficient memory bandwidth usage.

We propose a novel accelerator design, which will be referred to as **Mixed Precision Accelerator (MPA)** throughout this thesis. The accelerator fuses both temporal and spatial paradigms and is able to both simplify the logic of the processing el-

ements (PEs) and drastically reduce the memory bandwidth. In addition to the carefully designed PEs, we relocated all the shifting logic outside of the PEs, thus reducing their area overhead. We aim to provide a highly flexible and parametrized accelerator that can handle a wide variety of layers efficiently. Also, the flexible nature of our accelerator maps well onto FPGAs, and its high configurability opened the door to exciting Hardware-Software co-design opportunities.

In this chapter, we start by elaborating in section 1.2 on the schemes leveraged by state of the art accelerators to handle multiplications of variable precisions. Then, we draw our focus onto our Mixed Precision Architecture. Section 1.3 details the internals of the PEs and Section 1.4 describes the organization of the PEs into a Systolic Array. Section 1.5 introduces the shifting logic enabling operators of variable bitwidths. We summarize the resource utilization of the modules enabling operands of mixed precision in Section 1.7. Finally, Section 1.6 presents the Post Processing unit.

1.2 Bit-Level Fusion

Developing flexible hardware that can carry out binary multiplication with operands of variable bitwidths requires us to rethink about which sub-operations are getting carried out in the process. Let us consider the n-bit binary numbers $A = a_{n-1}....a_0$ and $B = b_{n-1}....b_0$. In the case where n = 4, we can show the multiplication [22] $A \times B$ as in Figure 1-1. We observe that it is possible to describe the product as the sum of multiple logical *and* operations. With regards to signed binary numbers, only the Most Significant bit (MSB) carries information about the sign, as seen below.

$$A_d = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$$

Hence, extending the previous description to signed numbers would only require using sign extensions in the partial products. Therefore the rest of this discussion will ignore signed logic.

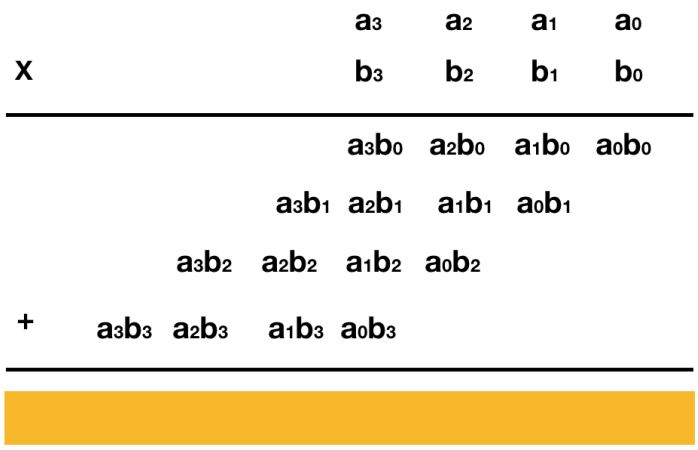

Figure 1-1: Unsigned Product Matrix

Given the above description, current research has attempted to provide variable bitwidth multiplication by breaking the operation down into these 2b-2b sub-multiplications[14]. The partial products can then be fused dynamically by applying shift-add operations. The operands can thus be decomposed into any 2*n-bit* wide binary number.

Traditionally, there have been two separate approaches as to how exactly dynamically fuse these partial products and hence achieve flexibility. The first one entails the development of **temporal** architectures, as exhibited by Stripes [27] or BISMO [34]. Their common approach is to subdivide the multiplication of the full precision operands into 1b-1b sub-multiplications, as shown in Figure 1-2 The results are then accumulated over time until the product is complete. The increased latency is compensated by the parallelization in space, thus achieving the same or improved throughputs over other fixed precision accelerators. However, the higher parallelization entails large increases in required bandwidth to keep the enlarged compute unit busy all the time. For example, in the case of Stripes, the compute unit's size is increased by a factor of 16x, requiring an equally larger input bandwidth. Additionally, the energy efficiency for BISMO and Stripes is affected by the high register usage, since every computation sub-unit requires its accumulator.

The second approach when designing mixed-precision accelerators was to use **spatial** multipliers. BitFusion [14] uses a distinctive multiplier architecture that can

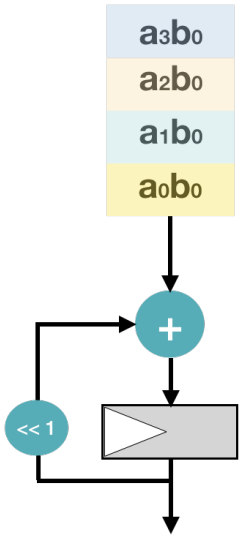

Figure 1-2: 1b-1b Temporal Multiplier

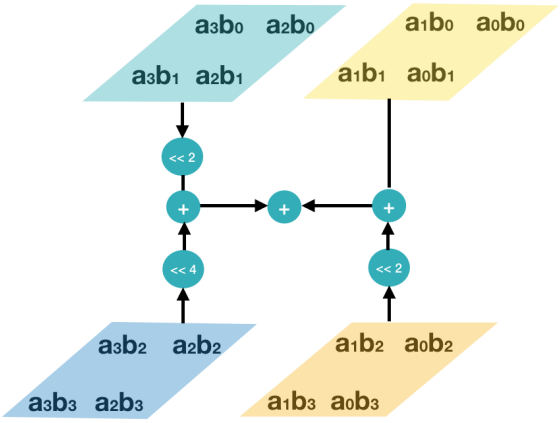

Figure 1-3: Spatial multiplier

subdivide a 16x16 bit multiplier into smaller power of 2 independent multipliers (e.g., two 8x8 or 16 2x2 multipliers). The basic units were 2b-2b multipliers called BitBricks. Figure 1-3 shows an example of how a spatial multiplier is organized, with each bitbrick highlighted in a particular color. The multipliers are organized in a systolic array, thus reducing the memory bandwidth requirement. However, the logic at each multiplier is overly complicated and hardly scalable. Each multiplier consumes a large number of large shifters and handles the complex routing of each 2-bit pair in the input to potentially any 2b-2b sub-multiplier. These requirements cause BitFusion to incur a considerable area and energy efficiency overhead. They also hinder BitFusion's adoption in FPGAs, as the large shifters and adders in the multipliers would not map efficiently onto the FPGA's LUTs.

In this thesis, we provided a new approach, which we call **pipelined** fusion. This approach consists in spatially unrolling and pipelining the loops obtained from temporally fusing the multiplication. This new idea can be seen in Figure 1-4.

The architecture presented in this thesis leverages all three fusion approaches. It can combine them in ways that are optimal for a given layer. The particular way in which the fusion approaches are combined will be discussed in the following sections.

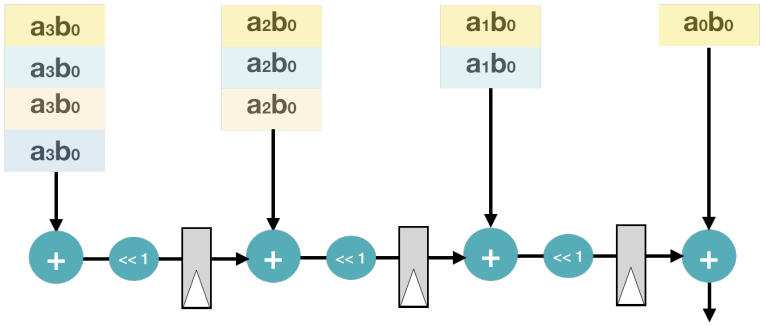

Figure 1-4: Pipelined Fusion

1.3 Processing Elements

A significant difference between MPA and other state-of-the-art accelerators is that no shifting logic is carried out within the Processing Element. This fact greatly simplifies the PEs' logic as they only need to perform Multiply-Accumulate (MACs) operations on low bit operands.

The Processing Elements' micro-architecture is shown in Figure 1.3. We see that the PEs are made out of a multitude of 2b-2b multipliers. The input bandwidth to each processing unit consists of 32-bit activations packets and 32-bit weight packets. Therefore, each PE is made out of 16 unique 2b-2b multipliers. The products are then added together in an adder tree. The resulting partial sum (psum) is returned as a 16b signed number.

Due to the low bitwidth, each multiplier will be implemented as a lookup table, which also turns out to be very efficient for FPGAs. Additionally, using lookup tables enables us to configure the multiplier to perform signed or unsigned multiplications easily. The signed bits are concatenated with the 2-bit operands to form the address to the lookup table. In other words, we use the 6-bit value $x_signed, x[1:0], y_signed, y[1:0]$ to index into a 64 location lookup table that gives us the result to their multiplication.

Since no shifting is performed within the PE, each set of 2-bit in each 32-bit input packet must belong to different values in a given layer. The most common case will be that each set will be originating from separate input channels, but could

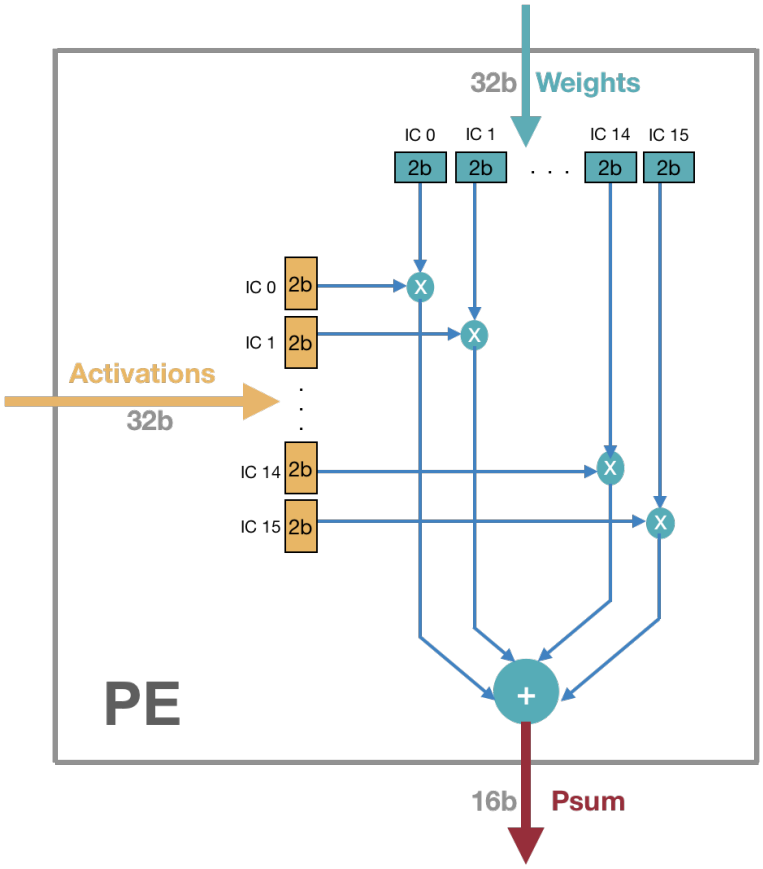

Figure 1-5: Processing Element

also sometime come from activations contributing to the same window. The origin of each set of 2-bit packets is something that the compiler will decide, based on the optimal layout for a given layer and will be furthered discussed in Chapter 3.

The PEs in our accelerator can carry out twice the amount of multiplications that the Bitfusion [14] PEs do due to the higher input bandwidth per operand (32b for MPA vs. 16b for Bitfusion). Additionally, the PEs presented in this section do so at a much lower area overhead, since the shifter logic is outsourced to a central location in the design, which will be discussed in more details in the following sections. Thus, we proposed a spatial PEs that provides high computational capabilities while drastically reducing the overhead that similar architectures incur.

1.4 Systolic Array

Research on Neural Network Accelerator has heavily focused on architectures that involve systolic arrays [19][14] as they have shown to be quite promising. Systolic Arrays combine a high computational ability with a relatively efficient bandwidth utilization and simple control logic.

The key idea behind a systolic array is that it is made up of a matrix of PEs. Each of those PEs is connected to its direct neighbors only, thus reducing the routing burden. Initial data and control input is done at the peripheral PEs. Inter-PE connections enable the transfer of the required control signals to each PE and move the data around in a limited manner. The MAC operations proceed through the systolic array in a pipelined manner. The limited data sharing scheme arising from the systolic array's dataflow allows for a reduced input bandwidth while maximizing data reuse. These characteristics are of particular relevance when dealing with large amounts of data and computations, as is the case with Neural Network operations.

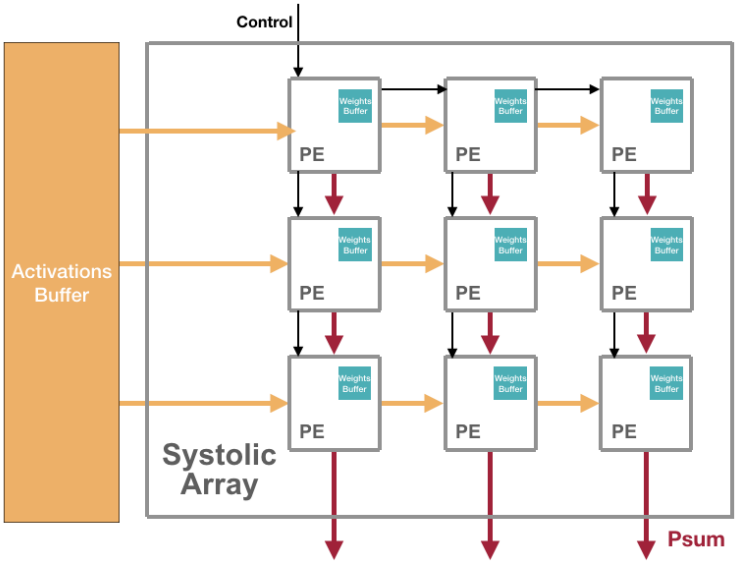

Figure 1-6: Systolic Array architecture for MPA

Figure 1-6 shows the particular architecture of the systolic array used in MPA. At first glance, we can see how each PE has a connection with its neighbors. Each

PE receives an activation from its left flank and then propagates it to it's right following its use in a computation. Similarly, every PE receives the current psum from its top neighbor. Following the local computation, the PE will add the new products onto the current psum. It will then propagate it to its southern neighbor. Not shown in this diagram are the registers at the PEs' input, guaranteeing a single clock cycle lifetime for incoming psums and activations. The registers enable a pipelined operation and essentially allow the systolic array to work at such.

Since data accesses tend to be the bottleneck in digital systems, we designed the systolic array to maximize data reuse and locality of accesses. Hence, the weights originate from small local SRAM buffers located within each PE. These would be pre-loaded before the computation of any layer. Prior research has tended to create small local buffers in the PEs for the weights as opposed to the activations since these tend to be less numerous and incur a lot more reuse [32]. Using these small buffers make the deployment of a weight stationary (WS) dataflow[33] more propitious. WS dataflow would imply that the accelerator's scheduling policy relies on maximizing convolutional and filter reuse. It minimizes the number of weight reads and aims to use them at a given PE for as long as possible. From preliminary experiments, we have also noticed how MPA tended to achieve better performance by adopting a weight stationary dataflow. Besides the small weights buffers, there are also a few bigger SRAM activation buffers. There is one independent activation buffer per systolic array row, allowing for 1 clock cycle latency for each buffer access and simplifying the routing logic. There also exists a set of partial sum SRAM buffers on the outskirts of the systolic array. However, they are placed following the shifting logic and as such, will be discussed in section 1.5.

The systolic array's dataflow can be described as diagonally propagated, i.e., the data is propagated from the top left corner first, and at every new clock cycle, the current computation will move 1 step to the right and 1 step down. The diagonal propagation entails that there is a single point of entry for control signals, thus rendering the control of the systolic array extremely simple. The figure 1-6 shows how the control signals are propagated through the systolic array. These control

signals include but are not limited to, the buffer addresses, the valid bit, and the sign bits.

Using a systolic array as the core of MPA guarantees a high computational ability and simple control logic. Concerning temporal accelerators, a systolic array will induce significant improvements in terms of bandwidth efficiency since inputs are only provided to the flanks of the array while preserving similarly high computational capabilities.

1.5 Shifting Logic

MPA's main contribution and the fundamental idea behind its high performance is the innovative shifting logic. While Bitfusion[14] combines 2b-2b multiplications

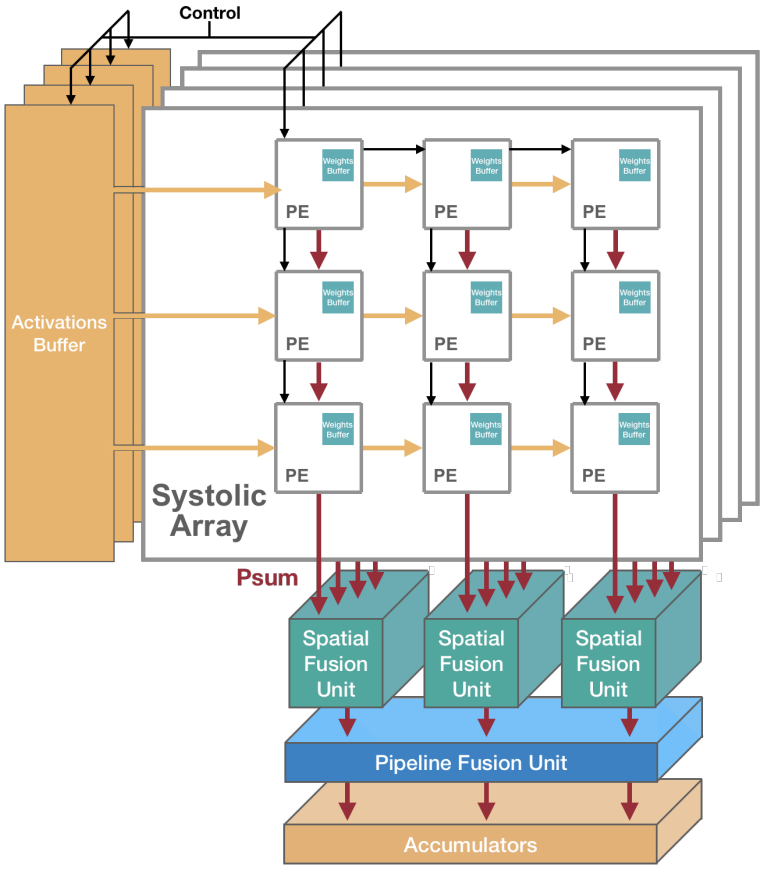

Figure 1-7: Compute Core

and shift-add operations in each PE, MPA proposes a novel approach by carrying out every single 2b-2b multiplication required in the PEs. As the partial sums exit the systolic array, they go through the external shifting logic where the dynamic fusion of the partial sums takes place. Hence, we can obtain fully-fledged activations as if they were computed using $2n$-bits wide operands.

MPA leverages two separate schemes to achieve activations and weights of variable precision. The activations are handled by the Spatial Fusion Unit (SFU). On the other hand, the weights are manipulated in the Pipeline Fusion Unit (PFU). Both components are independent, and a given implementation of MPA could be made to use either of the two units at a time or both. They will be discussed separately in the following two subsections. Both schemes, under their current implementation, limit the variable precision to be between 2b and 8b. Achieving a higher level of precision would require software scheduling adhering to the temporal fusion paradigm.

We can see in Figure 1-7 how the previously explored systolic array fits in with the shifter logic. In this section, we will explore all of these new components and modifications applied to the systolic array to achieve ideal mixed-precision performance. In this discussion, we will narrow down our focus on the cases that involve bit precisions ranging from 2-bits to 8-bits, i.e., we will ignore temporal fusion, since the latter involves a fair amount of software scheduling. Moreover, specific optimizations that were applied to the weight and psum SRAM buffer routers will also be discussed.

1.5.1 Architecture Support for Variable Activation Precision

We use the Spatial Fusion Unit (SFU) to operate on the activations of variable bitwidth. The main change that the unit required was for the systolic array rows to be folded, to provide a new dimension. If there were n rows, the SFU equipped MPA would now have four systolic arrays of $n/4$ rows each. The SFU will dynamically combine the systolic arrays' partial sums and will produce a single fused psum

output stream.

The set of systolic arrays can be seen in Figure 1-7. They each share the same control signals. Since the four systolic arrays were obtained by folding the original systolic array, each one of the resulting four systolic arrays will have an SRAM activation buffer. Similarly, each PE will still have its own small weights SRAM buffer. Folding the array would imply that there will now be 4 separate psums generated simultaneously per column. The increased output bandwidth from the set of systolic arrays is something that the SFU relies on to provide spatial fusion.

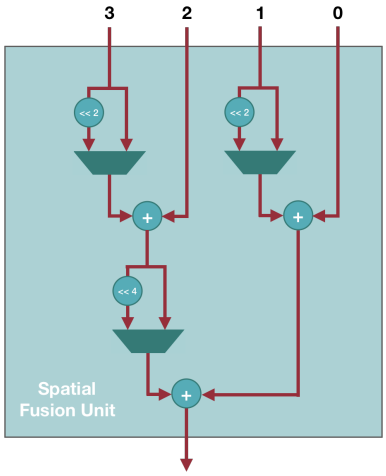

Figure 1-8: Spatial Fusion Unit

Figure 1-8 shows what is going on inside of the SFU. The numbers at the top of the figure denote the systolic array from which a given psum originated. Each 16-bit partial sum originating from a given systolic array is the sum of a few 2b-2b products. Hence, we can apply spatial fusion in a way similar to Figure 1-3, i.e., we can combine the partial products through shift-add operations. The SFU handles 2b to 8b precisions dynamically by using a set of three muxes. If all the muxes are off, we end up summing up the partial sums of all four systolic arrays. Hence, for n total rows, we get the sum of n psums computed from 2b activations. However, if we wish to operate on activations of say 4b, we would have to turn on the pair of muxes at the top level. Since the activation buffers are independent, we can store the lower 2-bits in the activation buffers located at systolic array 0 and 2. Then we would store the upper 2-bits in the activation buffers located at systolic array 1 and 3. This way, the following output partial sums will then be shifted by 2 and added with the partial sums originating from systolic array 0 and 2. Accordingly, we would obtain the sum of $n/2$ partial sums that were computed with 4-bit activations. Last, if we wish to operate on 8-bit activations, we would turn on all the muxes in the SFU. Then, we would store

bits [1:0] of the activations in the activation buffers from systolic array 0, bits[3:2] at the buffers from systolic array 1, bits [5:4] at systolic array 2 and finally bits[7:6] at systolic array 3.

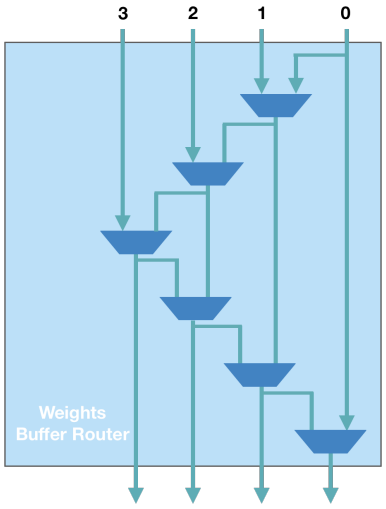

Figure 1-9: Weights Buffer Router

The muxes located at the SFU provide us with much flexibility at runtime. Altering the precision of the activations is greatly simplified and incurs no latency penalty. However, the design suffers from a reduced SRAM capacity when dealing with activations whose bitwidth is larger than two. As an example, consider the case when our activations are 4-b wide. Let $A = a_3...a_0$ and $B = b_1b_0$. Also, let a_3a_2 be stored at systolic array 1 and a_1a_0 at systolic array 0. Both a_3a_2 and a_1a_0 will need to be multiplied by B. Since they're located in separate systolic arrays, the PE at systolic array 1 and systolic array 0 will need to store B, thus wasting some precious SRAM space. In order to solve this design flaw, we've designed a router that enables the weights from the SRAM buffers in the same row to move freely between the four systolic arrays. The router is shown in Figure 1-9. The routing is set based on which precision mode we program the SFU with. An SFU set to 2-b mode will consider the weight buffers as four separate buffers. If set to 4-b mode, they will be seen as two sets of a weight buffer that is twice as big as usual, and if set to 8-bit will be considered as a unique weight buffer that is four times bigger.

As an example, refer to Figure 1-10 and Figure 1-11. These two figures consider the case where the router is programmed for 4-bit activations. The weights from the weights buffer in systolic array 0 are first broadcasted to PEs in systolic arrays 0 and 1. Then, once the weights from the buffer in systolic array 0 are exhausted, the buffer in systolic 1 takes over and starts broadcasting its weight contents.

The SFU, combined with re-configurable weight buffers allows MPA to maximize resource utilization by concentrating the shifting logic in a single location

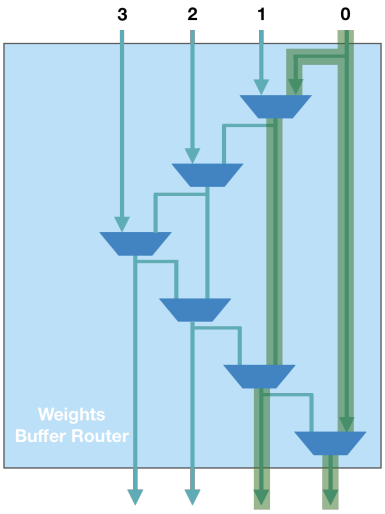

Figure 1-10: Buffer 0 broadcasts weights in 4b mode

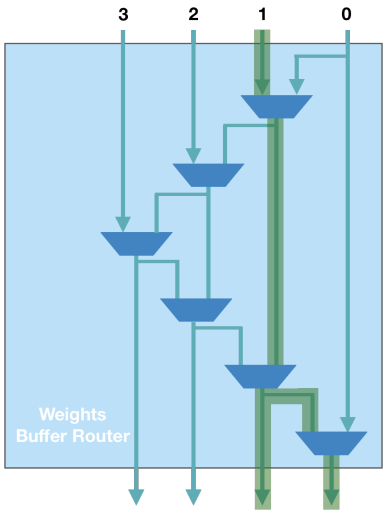

Figure 1-11: Buffer 1 broadcasts weights in 4b mode

while preserving and even enhancing the local memory capacity.

1.5.2 Architecture Support for Variable Weight Precision

We use the Pipeline Fusion Unit (PFU) to support variable weight precision. It theoretically can handle $2n$-bits weights for any $n <= m$, where m is the number of columns in the systolic array. However, for routing complexity purposes and in order to simplify this discussion, we will focus our description on an implementation that only supports 2-b, 4-b, and 8-b weights.

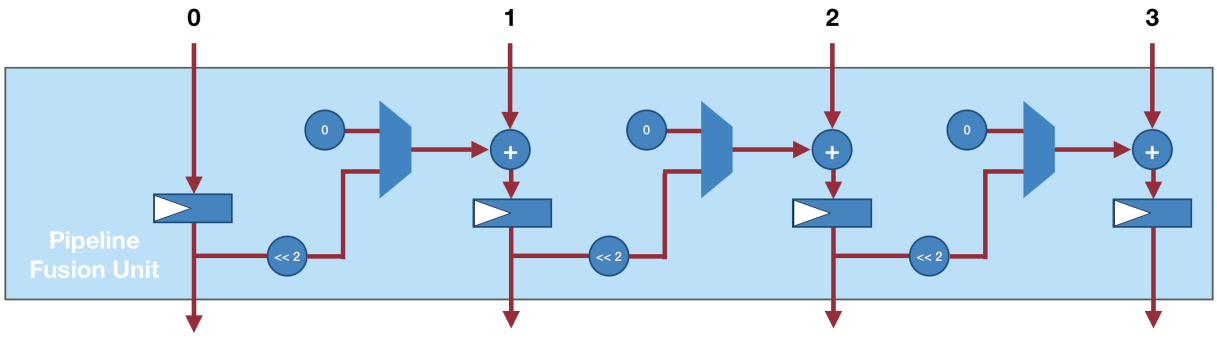

Figure 1-12: Pipeline Fusion Unit

PFU's architecture is shown in Figure 1-12. There is one partial sum stream

generated per column of the systolic array. Each stream is the result of multiplying activations with 2-bits weights. In this discussion, the streams will be referred by their numbers, as indicated in Figure 1-12.

The PFU achieves variable weight precision by dynamically turning on a path between the different columns. As a given psum travels to a column located to its right, it will be shifted by 2 and added to psum coming from above. This way, the PFU can extend the precision of the weight operand by two bits. However, in doing so, the PFU also reduces the output bandwidth by 2.

In the case where the weights are programmed to be 2-bits wide, every mux is turned off, and the input bandwidth exactly matches the output bandwidth. However, any raise in weight precision will lower the output bandwidth. In the case where we program the weights to be 4-bits wide, every set of two columns is merged. If we consider the set of columns made up of columns 0 and 1, only the output from column 1 will be considered as valid. Consequently, the PFU effectively halved the output bandwidth. In order to handle potentially significant accumulation results from large amounts of psums and variable operand's bitwidth, the psum bitwidth was set to 32.

The partial sums coming from the systolic array are, as their names inform us, partial. Hence, MPA requires a scheme to accumulate the incoming partial sums for as long as further computations take place in the systolic array. Thus, since the PFU is the last component in the computation pipeline, its columns are followed by an accumulator. The accumulators are primarily made out of a small SRAM buffer and an adder. However, as we've discussed previously, the PFU alters the psum bandwidth, requiring the accumulators to be modular and flexible. These new requirements make it more challenging to design an efficient router. For these reasons, we limit the capabilities of the PFU by subdividing every set of four columns into their sub-groups, i.e., there are no shift-adders in between the subgroups. The subgroups will look exactly as in Figure 1-12. The limit imposed on the PFU does not only simplify the accumulator's router, but it also reduces the logic within the PFU. The logic is reduced by only required three shift-adders for

every four columns.

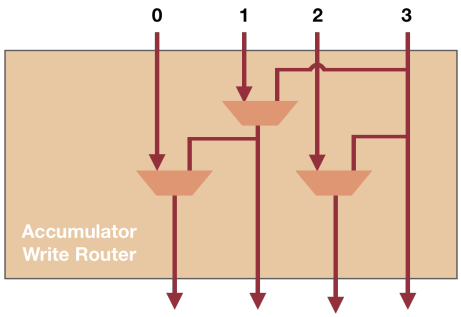

Figure 1-13: Accumulator Router Write Path

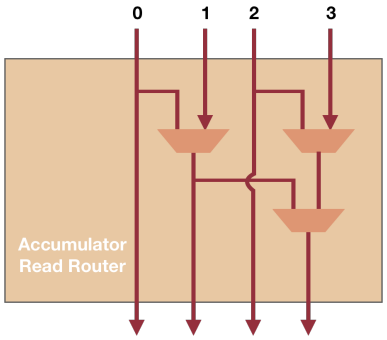

Figure 1-14: Accumulator Router Read Path

The accumulators are made modular for the only purpose of maximizing local memory utilization. Let us consider the case where we program the weights to be 4-bits. Then, the traffic from column 0 will become invalid, and we would leave the accumulator in the said column idle. Thus, in this situation, we need to provide a way to divert the traffic from column 1 to accumulator 0 until it fills up, and then back to its original destination to accumulator 1. This way, it would be as if we merged the two accumulators into a single one that has double the storage. The router design allows us not to waste hardware resources. Therefore, we enable MPA to provide variable weight precision while potentially enhancing memory capacity.

The router architecture is shown in Figure 1-13 and Figure 1-14. The read path of router design is similar to the weight buffer routers seen in subsection 1.5.1. While the weight buffer routers needed to enable us to route the contents of any buffer to any PE within the subgroup, the accumulator routers only required us to divert the contents of a buffer to the leftmost member of the buffer's precision subgroup. For example, in a 4-bit configuration, the contents of accumulator 0 needed to appear in column 1 and those of accumulator 2 at column 3.

The PFU, combined with modular accumulators, lets MPA achieve optimal performance while drastically reducing the logic. The shift-add logic in the pipeline doesn't use any resources for the shifting, as the psums always get shifted when

moving to the neighboring columns. Additionally, the modular accumulators allow for the local storage capacity to increase when dealing with higher bit precisions.

1.6 Post-Processing Unit

The entirety of the convolution (CN) or fully-connected (FC) layer computations are carried out in the compute core. However, MPA also needs to handle activation function layers [23] and batch normalization[15] layers, which are essential in the execution of a neural network. Their implementation in MPA will be further discussed below in sub-section 1.6.2. Another essential process that needs to be carried out by MPA is the quantization process. The Post-Processing unit provides a method to quantize the outputs from the compute core. It does so by merging the quantization scale factor into the batch normalization (BN) layer, rather than merging batch normalization into the convolutional layer (as conventional methods did). This quantization mechanism will be discussed in subsection 1.6.1.

1.6.1 Output Quantization

Our Mixed Precision Accelerator represents data with fixed-point precision. Previous work has shown the representation's feasibility when used in Neural Network computations[25]. The representation consists of picking a given bit position with a bit string. Every bit on the left of the bit we picked is an integer power of two starting from 2^0. Then, the bit we picked itself and any other bit on its right will be seen as a fractional power of two in ascending order, i.e., $\frac{1}{2^0}$, $\frac{1}{2^1}$, $\frac{1}{2^2}$,... In the case of MPA, we clip the weights and activations to -1 and 1 when signed, and 0 and 1 when unsigned. Therefore, all the input bits will be perceived as fractional if unsigned and all but one if signed.

In order to quantize a given weight or activation from a floating-point precision representation to a fixed-point representation, we can simply multiply the value to be quantized by a pre-computed scaling factor s, seen in **??**, where n is the target bit

precision and t_h is the value with the highest magnitude within the group of values to quantize with the same scaling factor, e.g., the weights in a given layer. The product will then be rounded and made to fit the target precision, i.e., the rounded number's binary representation will be obtained.

$$s = \frac{t_h}{2^n - 1} \tag{1.1}$$

While the quantized representations of our activations fed to the first layer and of all the weights in the network can easily be obtained by using a dedicated software compiler, MPA needs to be able to quantize the outputs from the compute core dynamically. The quantization needs to be flexible and able to handle the quantization from an output where the fractional part starts at a variable location to activation of any precision between 2-bits and 8-bits. The output's fractional part can start at many possible locations since we chose to handle mixed-precision networks. For example, if we're using 2-b unsigned weights and 4-bit signed activations, the fractional part of the psums will start at the 6th least significant bit. The starting location varies depending on whether the inputs are signed (signed numbers will have one fractional bit less than unsigned numbers) and on the precision of the inputs. The variable starting point of the output's fractional part further complicates the quantization process.

Therefore, we needed to think of a way to carry out the quantization seamlessly in the hardware. Our final design merges the scaling factor with the batch normalization. The BN operation [15] is given by 1.2, where γ and β are the batch normalization parameter and are learned in the optimization process, $\mathbf{x}$ is the input vector, μ is the sample mean and σ^2 is the sample variance. Finally, ϵ is a small constant included for numerical stability.

$$\mathbf{y} = \gamma \frac{\mathbf{x} - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \tag{1.2}$$

Therefore, if we rearrange the equation such that we can clearly separate the input vector $\mathbf{x}$ with a multiplicative term, we get that:

$$\mathbf{y} = \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}}\mathbf{x} + \left(\beta - \frac{\mu}{\sqrt{\sigma^2 + \epsilon}}\right) \tag{1.3}$$

By observing 1.3, we can conclude that we can compute the batch normalization by using a dedicated high-precision hardware multiplier followed by an adder. Thus, we can merge the quantization process by multiplying the BN terms with the scaling factor s_{BNU}. Therefore, we can offload to the software compiler the duty to get the new augmented batch normalization parameters defined in 1.4 and 1.5. The new terms can be used to apply the linear transformation 1.6 in the hardware. Since the outputs of a Fully Connected layer do not feed into a batch normalization layer, FC layers use the BNU exclusively for scaling purpose: where the outputs from the FC layer only get multiplied by the appropriate scaling factor s_{BNU} within the BNU. The scaling factor s_{BNU} is defined as seen in 1.7, i.e., it is a combination of the scaling factors used for the weights and activation operands and of the scaling factor used for the next layer's activations. The former is added to cancel their effect on the obtained psum, and the later is added to quantize the psum into an activation that can be used in the following layer. Each scaling term is obtained as seen in **??**

$$\gamma_s = s_{BNU}\frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} \tag{1.4}$$

$$\beta_s = s_{BNU}\left(\beta - \frac{\mu}{\sqrt{\sigma^2 + \epsilon}}\right) \tag{1.5}$$

$$\mathbf{y} = \gamma_s\mathbf{x} + \beta_s \tag{1.6}$$

$$s_{BNU} = \frac{s_{act}s_{wgts}}{s_{next_acts}} \tag{1.7}$$

As discussed above, the batch normalization terms are learnable and obtained after training. However, we need to add another constraint to deal with the mixed-precision and the issues that come with it. To simplify the hardware, MPA guar-

antees that the fractional part of $\mathbf{y}$ always starts at the same bit location. During the training process, we constrain the values of the parameters γ_s and β_s so that $\mathbf{y}$ will meet the guarantee. The fixed fractional part location lets the hardware round and set the precision of $\mathbf{y}$ as needed by simply extracting the same subset of 8-bits following the integer part. To clarify, if set the fractional part to start at the n^{th} bit, we would extract the subset $\mathbf{y}[n-1:n-i]$, where $i \in [2, 4, 8]$, depending on the target precision.

In this implementation, γ_s and β_s will be stored as 32-bit values since it would provide ample flexibility to shift around the position of their fractional part. The psum $\mathbf{x}$ is stored as 32-bits in the accumulator and therefore $\mathbf{y}$ will be a 64-bit value. We arbitrarily picked bit $\mathbf{y}[31]$ to correspond to the start of the fractional part as discussed in the previous paragraph.

Given the bit precision of the operands, the resulting partial sums can have the length of their fractional part be from 16-bits down to 2-bits. For example, if the inputs were both using unsigned 8-bit fixed-point precision, the resulting psum will have a 16-bit fractional part, since the two inputs are multiplied together. Another example would be if we instead used signed 2-bit inputs. That would imply that their fractional part is only 1-bit wide, and as such, their product will have a 2-bit fractional part.

As discussed previously, we shift the relative fractional length of γ_s and β_s to accommodate for the variable partial sums' fractional length. Therefore, let us consider the case mentioned above, where both inputs used are unsigned 8-bit variables. Then, the batch normalization product using these 16-bit fractional psums would still need the overall fractional part to be located at bit $\mathbf{y}[\mathbf{31}$, as stated above. To enable that, we set the fractional length of the batch normalization parameters to be 16-bits, thus enabling the product's fractional part to be 32 bits. A similar scenario can be discussed in the context of the second example mentioned above, when using 2-bit signed inputs. Given the 2-bit fractional size of the psum and the need of the batch normalization's product to be 32-bits, we set the BN parameter' fractional bitwidth to 30-bits, thus ensuring a 32-bit fractional BN product.

Therefore, we can see how choosing the BN's output fractional part to be constantly set to 32-bit guarantees a sufficiently wide dynamic range for the parameters and allows for sufficiently accurate quantization results.

1.6.2 Post-Processing Unit Architecture

The outputs from the Convolutional (CN) or Fully Connected (FC) layers are stored in the accumulators at the end of the computation. However, before they get stored in DDR memory, we need to quantize the outputs from 32-bits to the precision of the next layer's activations. Additionally, we may need to apply some activation function on the results. The post-processing unit, shown in Figure 1-15, is in charge of carrying these computations out. The unit is replicated for every systolic array column. The unit was designed to operate in a streaming manner, i.e., it performs computations in a pipelined manner, taking a new partial sum in per clock cycle. As such, it does not form an integral part of the compute core discussed in section 1.5. Instead, it is attached to the output path and is, therefore, implementation-specific. The unit's interface will be further discussed in the next chapter.

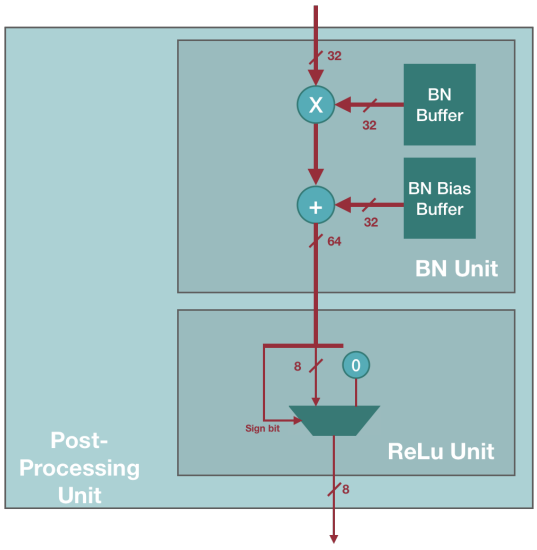

Figure 1-15: Post-Processing Unit

In its most basic form, the post-processing unit contains two sub-modules: a batch normalization unit and an activation function unit. The later could be

expanded to include a wide variety of activation function. However, since most modern Neural Network topologies solely use a Rectified Linear Unit (ReLu) [23], MPA currently only provides the ReLu activation function.

The Batch Normalization Unit (BNU) operates, as discussed in subsection 1.6.1, i.e., performs a merged batch normalization and quantization process. The hardware implementation is simple; it includes a single dedicated 32-bit multiplier and a 64-bit adder. An optional register can be placed in between the multiplier and adder, simplifying timing constraints. The main complexity arises from the necessity of storing a relatively large set of BN parameters. Since the batch normalization parameters are attached to a given CN or FC output channel, each BN unit will need to store the BN parameters linked to every output channel that will be streamed through the unit. Since there is one BNU per systolic array column, each BNU is equipped with two small SRAM buffers that will store the parameters γ_s and β_s.

Fully Connected layers could potentially require hundreds if not thousands of output channels per column, e.g., the VGG Neural Network topology [31], thus drastically increasing the required size of the buffers in the BNU. The higher number of output channel for FC layers stands in contrast with CN layers and their relatively low output channel numbers. However, the higher storage capacity would not affect the buffer storing the $\gamma_{s,oc}$ terms, since a BN layer does not follow FC layers. On the other hand, every single FC bias would need to be appropriately scaled. As such, it would be impossible to store all the biases in the buffer, storing the β_s, oc. We solve this problem by pre-loading the accumulators with the appropriately scaled FC bias before the FC computation. This way, we remove the need to use the dedicated 64-bit adder (the β_s, oc buffer will be pre-loaded with zeros) and instead use the pre-existing 32-bit accumulator adders.

The ReLu unit is simple in its operation. Out of the 64 bits generated at the BNU, we start by quantizing the result to 8-bits. We do so as described in section 1.6.1, i.e., by taking bits $\mathbf{y}[31:24]$, where $\mathbf{y}$ is the output from the BNU. We then get the sign bit out of $\mathbf{y}$ and use it to select between outputting the 8-bit subset of zero.

Depending on the output path implementation, we would further quantize the 8-bit result to possibly 4-bits or 2-bits, depending on the next layer's requirements.

1.7 Resource Utilization Analysis

In this section, we will delve into a detailed theoretical resource analysis of the logic enabling multiple operand precisions, as well as the optimizations that arose from the said logic. At the end of this section will be a table summarizing the resource utilization.

Let's first draw our focus on the core unit composed of the systolic array and the PEs that it contains. While the PE itself doesn't contain any registers, it contains about 16 2b-2b multipliers made from lookup tables. The 16 partial products then get summed up through an adder tree. Since it contains no shifting logic, there are no muxes within the PEs. Concerning the systolic array, it is made out of mostly registers that serve to connect and pipeline data between the different PEs.

The overhead due to handling mixed-precision starts with the SFU and its dependencies. By looking back at Figure 1-8, we can see that there are three shifting operations taking place per SFU. To implement them, we use a multiplexer per shift operation, to select between shifting the input psum by a certain fixed amount, or not to shift. Overall, we can conclude that the SFU uses 3 muxes, on top of 3 adders. Since the psums at the input of the SFU are 16 bit wide, we have two 18-bit adders on the first level producing a potential 19-bit output due to overflow. Lastly, the adder on the last level is a 23-bit adder due to the potential leftward shift by 4. The last adder's output will 24-bits due to possible overflow. Optional registers can be placed at the inputs and outputs of the SFU if the many adders turn out to be part of the critical path, thus making timing harder to meet. With respect to the weight buffer router, which resulted as an optimization to using the SFUs and folding the systolic array, we can conclude from Figure **??** that it uses 6 muxes to achieve its purpose.

The PFU shifting logic is based on a pipeline, adding a layer of shifts as the

Architecture	Modules		2-1 Mux		1-bit Registers		Adders	
			Count	Real	Count	Real	Count	Real
MPA	PE		-	-	-	-	$13nm$	6,656
	Systolic Array		-	-	$48nm$	24,576	-	-
	SFU		$59m$	1,888	-	-	$3m$	96
	Weights Buffer Router		$6\frac{n}{4}m$	640	-	-	-	-
	PFU		$87\frac{m}{4}$	696	$\frac{87m}{4}$	696	$\frac{3m}{4}$	24
	Acc. Router	Read	$32\frac{3m}{4}$	768	-	-	-	-
		Write	$32\frac{3m}{4}$	768	-	-	-	-
	Total		$\frac{3}{2}mn + \frac{515}{4}m$	4,888	$48nm + \frac{87m}{4}$	25,272	$13nm + \frac{15m}{4}$	6,776
Bitfusion	PE		$60nm$	30,720	-	-	$15nm$	7,680
	Systolic Array		-	-	$48nm$	24,576	-	-
	Weights Buffer Router		$12nm$	6,144	$32nm$	16,384	-	-
	Activations Buffer Router		$12n$	192	$32n$	512	-	-
	Total		$92nm + 12n$	47,296	$80nm + 32n$	41,472	$15nm$	7,680
BISMO	Total (Only from PE)		$96nm$	49,152	$32nm$	16,384	$32nm$	16,384

Table 1.1: Architectural Resource Utilization Summary. The real column assumes n = 16 and m = 32

psums progress leftwards. The shift operation is hardwired to always consists of a leftward shift by 2. However, since the PFU is divided into subgroups that can themselves be further divided into precision subgroups at runtime, we inserted a mux at each stage to provide the said flexibility. The mux provides the option to either forward the shifted psum being propagated or to send a zero. Sending a zero would indeed cut off the link in between columns in the PFU and create precision sub-groups. Therefore, due to the delimitation of columns into sub-groups of 4 columns in the PFU, there are 3 muxes and adders per 4 columns. Outputs are registered to contribute to psum propagation in the PFU and to make it easier to meet timing. Finally, the accumulator's router uses 6 muxes, which can be seen from Figure 1-13 and Figure 1-14. The accumulator, to handle a large number of partial sums, uses 32-bit adders, and so the muxes would need to be of the same width.

To summarize the resource utilization we can refer to table 1.1. The real column assumes n = 16 and m = 32. The table also includes an estimate regarding the resource utilization of Bitfusion [14] and BISMO [34]. Their descriptions are an approximation based on information found in their respective paper. One modification was made to the systolic array in Bitfusion, where we don't consider the activations to be broadcasted on a row basis but instead to be propagated one PE at

a time. The modification was done to achieve a fairer comparison with the architecture presented in this chapter as it doesn't do any form of broadcasting within the systolic array. BISMO was assumed to use a barrel shifter in each PE. For fairness, we assume the barrel shifter can achieve any shift amount from 1-16 since we assume up to 8-bit operands. In the analysis below, we consider a compute core with n rows and m columns. We also ignore the contributions from handling the control signals.

The overhead of the post-processing unit is not included in the table. The lack of overhead information can be explained by the fact that it is hard to compare against the resources used by similar modules in other accelerators. The architectural description of these accelerators often glances over these aspects, as they deem it to be left up to the implementation. However, we will briefly delve into the post-processing unit resource utilization. Given the requirement that the BN will also help carry out the quantization, we would need a 64-bit multiplier and a 64-bit adder. Then, we would also require a small buffer per column to hold the BN and BN bias terms of what could potentially be many different output channels. Finally, the implementation of the ReLu unit is simple and would only require eight 2-1 muxes per column.

Chapter 2

Mixed Precision Accelerator FPGA Implementation

2.1 Introduction

The description of the Mixed Precision Accelerator (MPA) made in Chapter 1 was an architectural description. It focused on the aspects relevant to the contributions made to the field and the generally expected performance. The present chapter will turn its focus toward a complete working implementation of MPA. The guiding principle was high parametrization while maintaining equally high performance. MPA was envisioned to fit into a broader software-hardware co-design framework. The hardware could adapt to the neural network it was tasked to accelerate, optimizing for bandwidth usage and local storage. On the other hand, the software could alter the neural network topology as it receives hardware architectural details. This thesis will solely focus on the hardware part of this over-arching project. The neural network optimization framework with neural architecture search will be covered in another work from HanLab.

Given the requirements, the target platform for the accelerator was naturally embodied by an FPGA. The high configurability and parallelization they provide is unique, and show much promise in the deployment of low latency, high

throughput, and efficient neural network accelerators. The FPGA chosen for the deployment of MPA was the Virtex-9 Ultra-Scale+ designed by Xilinx, in the form of the VCU1525 Development Board[8]. Their high performance was ideal in the development phase, providing ample resources and fast PCIe connectivity with the host. Future implementations of MPA could be deployed on FPGAs targeting edge-devices, e.g., the Zynq family of SoCs developed by Xillinx[5].

This chapter will describe every interface and module that was designed to make our architecture functional. The overall design is shown in figure 2-1. This chapter starts by describing the PCIe and memory interfaces in Section 2.2. Then, the instruction fetch process will be presented in section 2.3. While an ISA had to be developed for this implementation, it will be discussed in Chapter 3. Section 2.4 focuses on the data movement scheme for both the inputs (weights and activations) and the outputs. It will also involve a discussion on the layer activations' memory layout. The various address generators used to map Convolutional and Fully Connected layers are presented in Section 2.5. Eventually, Section 2.6 closes this chapter by describing the FPGA resource utilization for a particular configuration.

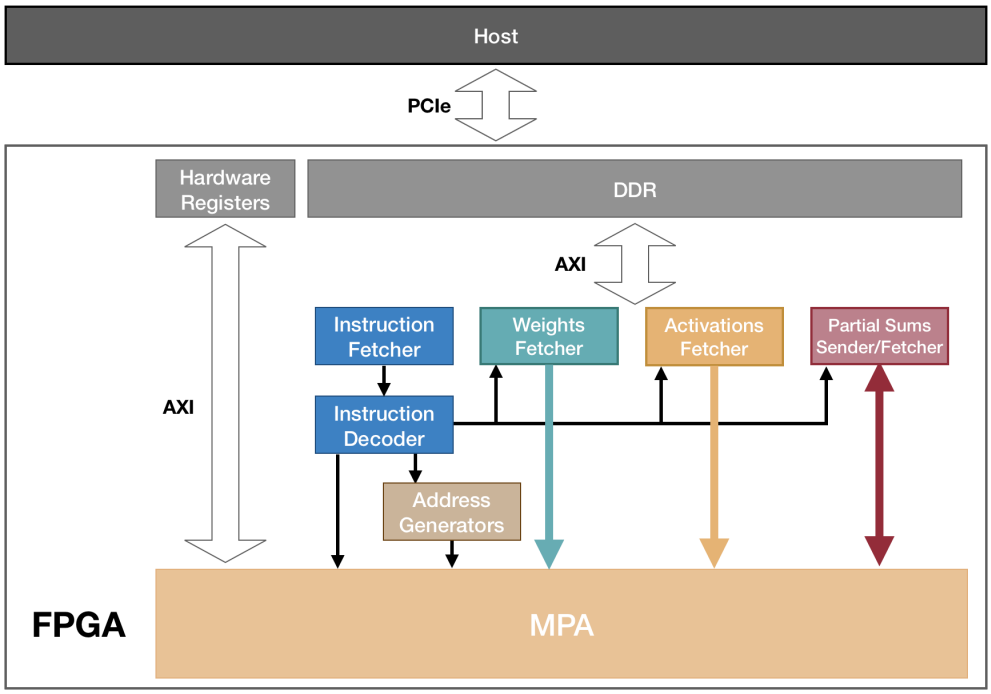

Figure 2-1: Mixed Precision Accelerator FPGA Implementation Architecture

2.2 Interfaces

In this section, the external interfaces used by MPA will be discussed. The overall interface architecture is shown in figure 2-2. In the design, the clock domain crossing and the arbitration is done by the SmartConnect Xilinx IP [4]. The IP core carries out clock domain crossings using asynchronous FIFOs and arbitration with round-robin arbiters.

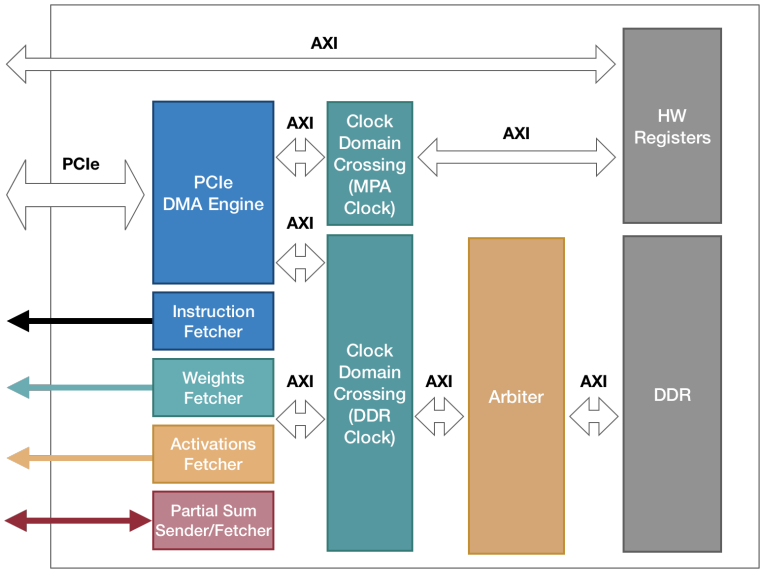

Figure 2-2: FPGA External Interfaces Configuration

This accelerator is an application-specific processor. Therefore, to facilitate its operation, a PCIe link between MPA and a host computer was established. This way, the host computer can send a compiled program to the FPGA. The process through which the host computer communicates with the FPGA will be described in Section 2.2.2. The accelerator will also need to have access to DDR memory to save temporary data and to write back the final results. The interaction between the DDR banks and the logic core will be the focus of Section 2.2.3. Each interface will have its particular clock domain due to their standard's differing specifications and will have to operate in conjunction with MPA's core logic. The description of the various clocks used through the system can be found in sub-section 2.2.1.

2.2.1 Clock Domains

Clock Frequency	Dependent Modules
150 MHz	Accelerator Core logic
333 MHz	DDR Controller
100 MHz	PCIe Link

Table 2.1: Clock Domains Summary

To properly operate and interact with the aforementioned external interfaces, the FPGA will need to generate a set of clocks for each interface and the core logic. Also, it will have to handle any necessary clock domain crossings. Table 2.1 summarizes the clock frequencies used through the design. They are all derived from the main FPGA system clock by using on-board Phase-Locked Loop (PLL) circuits. The PCIe's 150MHz clock was chosen due to it being a good trade-off between performance and ease of achieving timing closure. Eventually, this implementation can be further optimized to increase the core logic's clock frequency.

2.2.2 PCIe Interface

The host communicates with the FPGA as if it was writing to its own memory space. Within the FPGA, the communication is established in two different and complementary ways. The first one is essential in facilitating the computations carried out by MPA. The host computer does so not do so by establishing a synchronized channel with the FPGA, but instead by writing the program directly to DDR. Therefore, the host has a direct and independent path to the FPGA's on-board DDR. However, the host computer also needs to communicate synchronously with the FPGA, since at the end of the day, the host off-loads the inference task to MPA. Additionally, the accelerator also needs to be aware of the existence of a new program and needs to know when to start its computation. Therefore, a set of hardware registers were also included, and are accessible by the host through PCIe and locally available for the FPGA. Through these two separate communication channels, a high-throughput, high-latency path through direct DDR access

for sending program data to the FPGA was provided, in addition to a low-latency, low-throughput path through the hardware registers to synchronize the FPGA with the host computer.

Regarding the exact data path in the FPGA, we can refer to Figure 2-2. The FPGA's PCIe ports are directed to a Direct Memory-Mapped (DMA) engine. The engine maps DDR addresses and hardware registers to the host's address space. The Xilinx XDMA PCIe engine [3] was used as MPA's DMA engine. The output of the DMA engine follows the AXI protocol [2] and resides in the PCIe clock domain. The AXI signals are forwarded to the hardware registers and DDR. Before reaching their destinations, the AXI signals cross the required clock domains through asynchronous FIFOs.

2.2.3 DDR Interface

The local storage provided by the many local SRAM buffers in MPA's core logic is too small to hold most Neural Networks. Therefore, MPA was given access to a 16GB DDR4 bank, easily extensible to 64GB if needed. To hide the high latency of reading out of DDR, the FPGA provides a DDR bandwidth of 512-bits. The high bandwidth will direct a lot of the design choices that we will see in the rest of this chapter. From Figure 2-2, we see that MPA fetches and send data using dedicated modules. These use as their core technology the DataMover Xilinx IP [1] and a DMA AXI wrapper module developed by Zhekai, another member of Hanlab. The IP facilitates data movement, handling requests, and buffering. The IP also handles backpressure, aided by the correct use of the AXI protocol. MPA will inform the fetcher and sender modules how much data to fetch and where to fetch it from based on the compiled instructions. A small exception is made with regards to the instruction fetcher. The later receives the size and location of the instructions from a couple of hardware registers.

The DDR controller operates on its clock domain, as seen in Table 2.1. Therefore, since the AXI signals from the fetcher and sender modules reside in MPA's compute

core's clock domain, they will be synchronized by using an asynchronous FIFO. Additionally, the DDR controller can receive concurrent requests from potentially all the fetcher and sender modules in addition to the host through PCIe. Therefore, an arbiter is included to provide fair access to the DDR.

2.3 Instruction Fetch Pipeline

A neural network execution starts by having the host store instructions in the FPGA's DDR banks. As mentioned in sections 2.2, the FPGA is made aware of the existence of a new program by writing in a hardware register the location and size of the program, followed by the setting of the valid program register. From that moment onward, the Data Mover IP at the core of the Instruction Fetcher will start reading instructions out of DDR and forwarding them to a local FIFO instruction buffer. Every instruction packet read will be 512-bits wide. However, as we will see in Chapter 3, the ISA used in this implementation defines 128-bit instructions. Due to the DDR bandwidth, the local FIFO entry size was set to 512-bits. A 128-bit four-element shift register was instantiated at the FIFO's output. This way, the instruction fetcher can withstand the high DDR bandwidth while at the same time, provide each instruction to MPA at the appropriate rate. The instructions exiting the shift register will then be redirected to a decoder.

During execution, a scheme to control when a new instruction should be dispatched as needed. An instruction dispatcher module embedded within the instruction buffer was used. As an instruction exits the shift register, a scoreboard checks if this instruction can be dispatched based on the other instructions currently in-flight. In the case that it can be dispatched, the instruction is registered, and its valid bit is set. The later is then used by whichever component takes the instruction as an input and also by the scoreboard. When the instruction is completed, it is removed from the scoreboard by de-asserting the valid bit. The scoreboard creates a few dependencies between instructions. For example, an activation or weight fetch cannot be loaded if a computation is currently being carried on. If the

hardware were to be also optimized to include double buffering for the weights and the activations, then it would be possible to do without the latter requirement.

2.4 Data Movement

The data movement is the most complex piece of control logic developed for this implementation. The weights and BN parameters are easily controlled by software since they are only read from DDR by MPA. As such, they will be discussed in Chapter 3. On the other hand, the FPGA will be in charge of writing the partial sums back to memory. The psums will then need to be read as activations when computing the next layer - enabling seamless multiple-layer computations thus required co-designing a memory layout for the activations and the control logic arranging the data into the said layout in DDR.

Two main issues rendered this process complicated: the variable accelerator's output bandwidth due to the PFU (see Section 1.5.2) and the high DDR memory bandwidth (512-bits). Since memory accesses are the most energy-consuming operation[33], the design aims to use as much of the 512-bits as possible. However, under this implementation's configuration, the output bandwidth ranges from 16-bits (using 8-bit weights and setting the next layer's activations to 2-bits) to 256-bits (using 2-bit weights and setting the next layer's activations to 8-bits). This high variability, combined with a high output bandwidth, make it almost impossible to deal with memory writes in an online manner while keeping it easy to read back the output as activations when computing the next layer. Instead, the computation was decoupled from the writes by using ping-pong buffers in the accumulators. This way, the next layer's activation can be written back to DDR with full visibility on the output tile that was just computed. Additionally, due to the decoupling, the writes can happen simultaneously with the computation of a new tile. In the next few sections, we will be seeing how the activations are stored in memory in a way that is general to every layer and easily writable and readable. We can then draw our focus on the logic within the FPGA that parses and creates 512-bit DDR

packets made from activations.

2.4.1 Activations Memory Organization

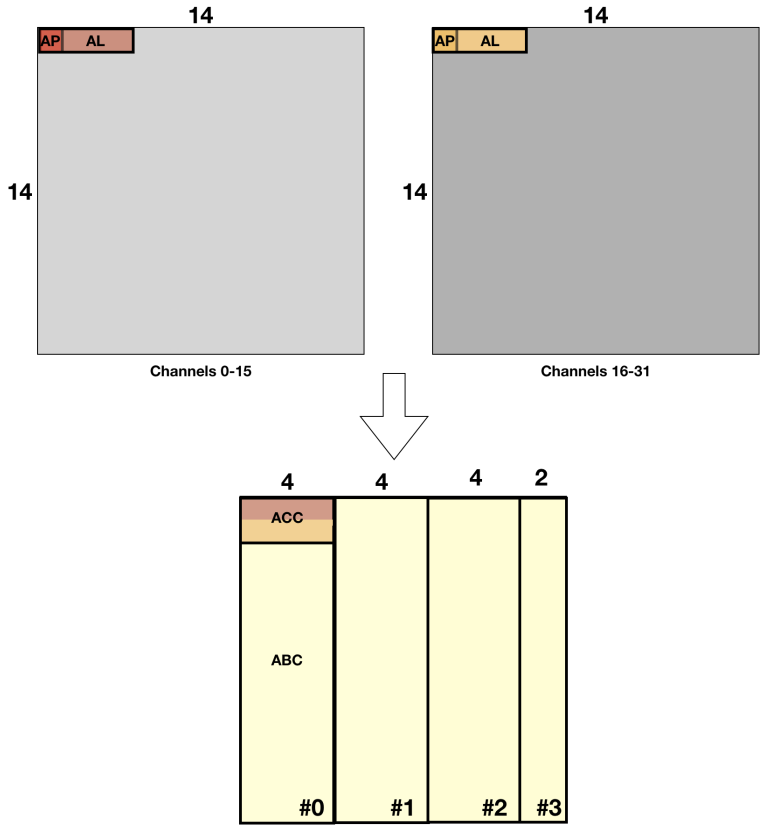

Figure 2-3: Activations' subdivisions in memory

The activations need to be laid out in memory in a way that is consistent and can be generalized to every layer encountered. The consistency is critical in enabling the outputs of a layer to be used as the inputs of the next layer. In this discussion, the activations considered are distributed in 3 dimensions: height, width, and channels. In the case of Fully-Connected layers, the memory organization will still hold and will consider that the height and width are set to 1. The first requirement is given by PE's use of the activations, as seen in Section 1.3. The PEs use activations packets, formed by a set of 16 different activations. A single packet sent to a PE store these activations in 2-bit subpackets. If the precision of the activations is higher than 2-bits, the activations will first be divided into 2-bit fields, and each

field will be part of a different activation packet. The 16 different activations found in a packet were chosen to originate from 16 different channels. The memory organization will use these 32-bit activation packets as the basic unit and will be referred by their acronym AP.

Given the DDR bandwidth of 512-bits per access, 16 different APs can be packed. Of those, the 16 APs would correspond to distinct height and width dimensions if 2-bit activations are used, 8 different APs if 4-bit activations are used and, finally, 4 different APs when using 8-bit activations. These set of APs from an activation line (AL), and are chosen from activation that differs in their width dimension only. Als are created until the width dimension is exhausted, without spilling over onto the height dimension. This way, the final AL could be partially filled in the cases where the width is not a multiple of 16,8 or 4, depending on the activation precision. In the case where the number of the channel exceeds 16, we may find multiple ALs for the same height and width dimensions. These ALs are stacked to form an Activation Channel Column (ACC). Finally, every ACC in the height dimension is taken to form Activation Block Columns (ABC). In memory, every ABC is stacked, until storing the totality of the activations. Figure 2-3 exposes this process in the case where a given layer is set to 8-bits and whose dimensions are 32x14x14, where 32 corresponds to the number of channels. The two blue shapes on the top of the diagram are already made out of APs, and so represent the contribution from 16 channels each. Lastly, Figure 2-4 shows how to stack the ABCs are stacked in DDR.

2.4.2 Input Path

The activation fetcher needs to maximize its usage of the available DDR bandwidth while keeping the latency of loading the local SRAM buffers to a minimum. In order to use the input bandwidth fully, we would need at least 16 separate activation buffers, since there are 16 activation packets stores per 512-bit DDR packet. Therefore, this discussion will be narrowed down to the case where the FPGA is programmed with 16 activation buffer. After folding the systolic array to enable

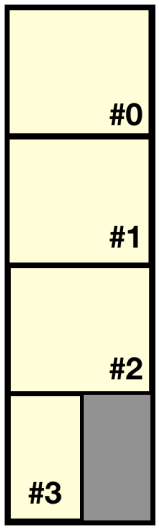

Figure 2-4: Activations written in DDR

activations mixed-precision, we have 16 separate points of entry with 2-bit activations, 8 with 4-bits and 4 with 8-bits. Due to the memory arrangement, it is easiest to forward each packet to the same row of the systolic array. To do so, a set of 16 shift register is deployed at each entry point. In the regular case, where input channels are plentiful and exceed 256 for 2-bit activations, 128 for 4-bits and 64 for 8-bits, it is possible to read from DDR without having to stall. Each DDR packet will correspond to an activation line, and so the dependencies between the various rows in MPA are removed. In the case that the DDR bandwidth is reduced, as would be the case if we were to use an edge FPGA instead of the Virtex-9 FPGA, the number of registers per shift register will be reduced.

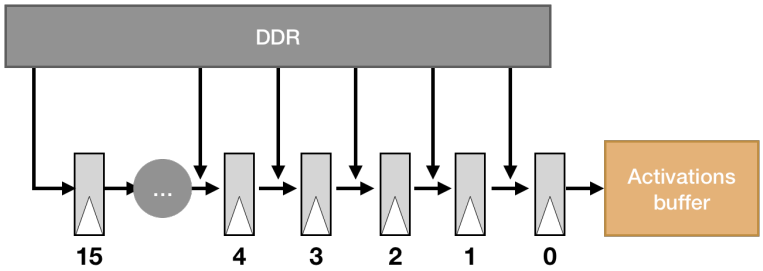

Figure 2-5: Input Path

A dedicated activation fetcher address generator is deployed to carry out the scheduling logic to each shift register, and to send the appropriate read requests

to DDR, depending on the size of the tile used in the current computations. Each register in the shift registers has a mux at its input, deciding whether to load a new value to the register or to keep propagating activations. The activation loaded is augmented with a valid bit, used at the output of each register to know whether the value should be loaded to an activation buffer. The valid bit also serves the purpose of helping with the activation buffer address computation. The instruction directing the activation fetching will program a dedicated activation buffer register to hold the address at which to load the data at the start of the loading process. The address register will get incremented by one after each valid bit that it sees. The muxes facilitating the loading process also enable to load multiple shift registers simultaneously in the case that the activation precision is higher than 2-bits. We can refer to Figure 2-5 to see how the shift registers are loaded with activation from DDR.

The large size of the layer computations carried out by MPA will most often than not force the subdivision of the computations in smaller tiles. The existence of tiles is due to the fact that the local storage provided by the SRAM buffers embedded in MPA is too small to hold most of the encountered layers. Therefore, it is also necessary to provide a partial sum fetcher, restoring the contents of the accumulator to complete the computation and obtain full-fledged activations to be used by the next layer. While the pathway through which we write both partial sums and activation to DDR will be furthered discussed in Section 2.4.3, the only information to know by now is that partial sums are stored in memory as a block, matching exactly how they are in the accumulators. Therefore, when the time comes to restore the said partial sums, it is possible to burst read from a specified DDR address the partial sums and write each incoming packet in a linearly increasing address to the accumulators. The large bandwidth enables a large number of accumulators. At least 16 accumulators will be needed to avoid a stall, and therefore 16 columns in the systolic array. However, in this work, MPA will be configured with 32 columns.

2.4.3 Output Path

From the previous section, we saw how MPA would need to write both partial sums and complete activations to DDR. Therefore, a module that can read 512 bits from the accumulators was designed. The module decides to either write the accumulator's outputs as they are to DDR or to make them go through logic that creates quantized activation packets that match the memory organization from Section 2.4.1. The later has the post-processing unit (PPU) discussed in Section 1.6 embedded in it to facilitate the quantization process, followed by a packet maker module. The architecture of the output sender module is shown in Figure 2-6. To reiterate previously discussed architectural aspects of the output path, MPA was configured with 32 columns for this implementation and uses a set of ping-pong buffers for the accumulators. These buffers decouple the writes from the computation, removing the need to carry out all of the write logic in a streaming manner.

Using 32 columns doubles the output bandwidth that can be handled at no cost. Therefore, the accumulator buffers were divided into two groups, the ones located at an even position and those at an odd position. These two groups are mutually exclusive when it comes to reading the buffer values, thus reducing the bandwidth to 512-bits. Due to the accumulator buffer coalescing when using precisions higher than 2-bits (see Section 1.5.2), the said division scheme would not cause any particular issues, as the outputs are naturally arranged in such a way. Some issues arise when using 2-bit weights, but they can be fixed through software.

The 16 distinct 32-bit accumulator values get quantized in the Post-Processing Unit, producing a 128-bit stream made out of 8-bit distinct values. The multiplier and the adder that the PPU contain are implemented using built-in DSP slices in the FPGA. The values will then be furthered quantized to 2-bit or 4-bit in the packet maker module if necessary. The values then get broken up into subgroups of 2-bits and get merged into separate 32-bit packets, as shown in Figure 2-7. All the bits

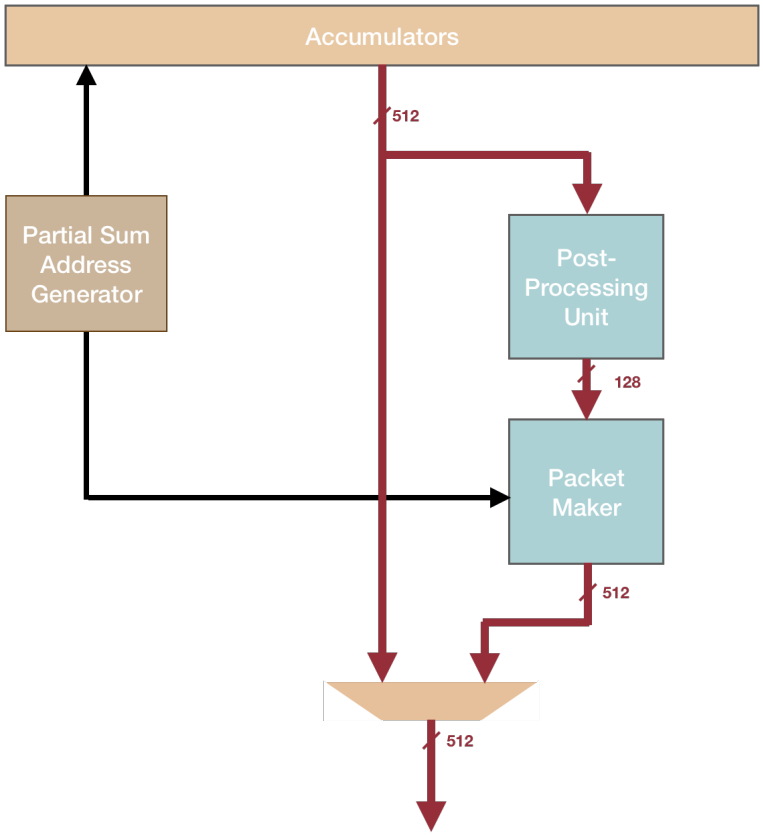

Figure 2-6: Output Path

from the quantized values at positions [7:6] get packed into a single activation packet, and so do the bits at positions [5:4], [3:2] and [1:0]. The relative position of these packets was chosen to make it easier to load the activations in the activation fetcher, achieving the lowest latency possible. The activation packets then get fed into a shift register. The first four registers form a single group and make sure that an entire 128-bit packet is loaded. Then, there is another set of shift registers that shift the said 128-bit packet. This scheme simplifies the loading logic for varying bandwidths caused by the mixed output precision. The dedicated psum address generator synchronizes this sequence of reading values out of the accumulators and enabling the shift registers in the packet maker module.

The packet maker also instantiates a shift register to form a byte mask, required by the DDR controller when making DDR write requests. Due to the varying dimensions of the tile, some DDR packets will contain less than the available

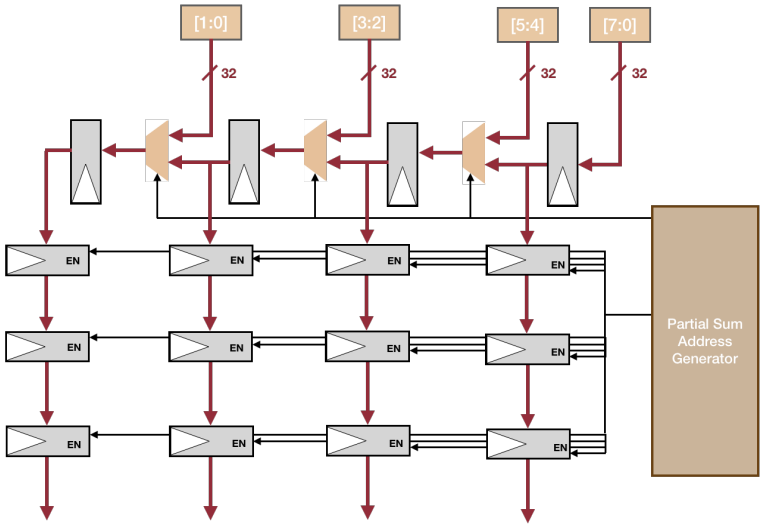

Figure 2-7: Output Packer Maker Logic

512-bits, as seen in Figure 2-3 where the last activation block column contains activation lines with only 2 out of available 4 activation packets. Hence, the byte mask generated will have to reflect this and only validate the appropriate number of bits.

2.5 Computation Address Generators

The compute core and its architecture were discussed extensively in Chapter 1. However, the activation and the weight buffers need to be accessed in a synchronized way to perform a convolution or a fully connected layer. This section will discuss how Convolutional and Fully-Connected layer mapping onto the core is performed at the hardware level. The mapping is carried out by a set of address generators, programmed through the program instructions. The convolution operation is defined as in Equation 2.1[33], where **O** is the output of the convolution, B is the bias vector, **I** is the activation vector, and **W** is the weight vector. The variable m is the iterator of the output channel variable M, x and y respectively correspond to the height, and the width of the output vector, S and C correspond respectively to the height and width of the kernel, finally C refers to the number of input channels, and finally U indicates the stride for the corresponding layer. Fully-Connected

layer computations can still be defined according to the same equation if we set $X, Y, R, S = 1$.

$$\mathbf{O}[m][x][y] = Activation\left(\mathbf{B}[m] + \sum_{R-1}^{i=0}\sum_{S-1}^{j=0}\sum_{C-1}^{k=0}\mathbf{I}[k][Ux+i][Uy+j] \times \mathbf{W}[m][k][i][j]\right) \tag{2.1}$$

The goal of the address generators is to map the above computation onto the compute core. The computation would have to be tiled first, to only give to the address generators the responsibility of carrying out the tiled computation. The computation is tiled in terms of the input channel (IC), output channel (OC), output height (OH), and width (OW) and kernel height (KH) dimensions. The relative loop ordering for the outer tile loop is variable and determined by the compiler based on performance metrics such as energy consumption and latency. The inner loop's relative loop ordering is fixed based on the dataflow provided by the program instructions. The only dataflow implemented thus far is the weight stationary dataflow, but the implementation could easily be extended also to use output or input stationary dataflows. As we will see in Chapter 3, the instructions governing the computation address generators were designed so as only to require 3 instructions to schedule a whole tile computation; one to govern the activations, another for the weights and finally one for the partial sums. The low instruction per layer guarantees a relatively small program size.

The address generation process for the activations and the weights is linked in the module. After providing a few parameters regarding the size of the tile via instructions, a set of interconnected counters keep track of the computation and forward the appropriate addresses to the activation and weight buffers. The addresses are input in a single location, the top left corner of the systolic array. The addresses then get propagated through the systolic array. Therefore, the address generator can simultaneously control an array of any arbitrary size, thinking that it only is communicating with a single PE. The buffers are loaded with the appropriate data in order for the computation to remain correct.

Finally, the address generation process for the output is quite different. Compared with the process for the activations and the weights, there is still a single address generator that also communicates with a single accumulator which then propagates the control signals to other accumulators. As such, we can see how the accumulators will behave in a FIFO like manner, and do not need to know anything about the state of the weight and activations buffer. All it needs to know is that a new output has been produced, and it should advance its programmed sequence. The system is therefore implemented as a latency-insensitive design [11]. For the weight stationary dataflow, the specific address sequence can be described as filling up a FIFO with a short sequence of push operations, carrying the initial partial sums, followed by a long sequence of simultaneous push-pop operations, in order to get a value from the accumulator, add onto it the incoming psum and finally store back the value in the accumulator. We do not need to pop the results from the accumulators at the end of the computation, since the buffers will be swapped with their ping-pong counterpart, and will be read by the output under the module.

2.6 Results

The implementation presented in this chapter achieves the desired goals. It provides an interface from the host to the FPGA in a way that provides both a low-latency path for synchronization purposes and a high-throughput path that enables the bulk transfer of program instructions and data. The goal of leveraging the high DDR bandwidth to its full potential within the FPGA was achieved, and a mechanism to read and write activations seamlessly was provided. Lastly, the computation scheduling is provided by address generators dedicated to the computation and is done in such a way to relieve the program from having to be too dense by requiring a few instructions to map each tile onto the compute core.

When configuring the FPGA and running synthesis and implementation, the various parameters were set according to what is shown in Table 2.2. Given this configuration, we should achieve state-of-the-art performance, since it is similar

Parameter	
Systolic Array	16x32
Weights Buffer	65KB
Activations Buffer	16KB
Accumulators	32KB

Table 2.2: MPA FPGA Configuration

to the configuration used by other mixed-precision accelerators such as [14]. With respect to the Virtex-9 (VU9P) resource utilization, we can refer to Table 2.3 to see the post-implementation resource summary as seen on Vivado. The table shows us that given the current configuration, we are far from reaching full potential provided by the VU9P FPGA. Hence, the potential for MPA to provide even better performance on the VU9P is high, and the exploration of bigger designs can be left for future work.

Resource	Count	Utilization (%)
LUT	253k	21
LUTRAM	12.3k	2
FF	179k	7
BRAM	626	29
DSP	182	3
I/O	140	21
GT	8	11
BUFG	25	1
MMCM	2	7
PLL	3	5
PCIe	1	17

Table 2.3: Post-Implementation Resource Utilization

Chapter 3

Software Framework

3.1 Introduction

An efficient and intuitive software framework is necessary to harness the full potential of our accelerator. The solution consists of the software framework presented in this chapter. The first thing that had to be done was to hide the complexity of interacting with a PCIe device to the end-user. Therefore, a kernel driver that knew exactly how to facilitate memory transaction with the XDMA engine and which would enable the end-user to communicate with the FPGA as if it were a regular file was deployed. The driver used is described in Section 3.2. Then, it was necessary to make the interface with the FPGA even more intuitive by developing a python library that provides a wrapper over the FPGA, making it easy to integrate with the compiler. The FPGA python library is discussed in Section 3.3. Once the communication infrastructure with the FPGA is set up, the next step was to integrate a popular machine learning framework into the one presented in this chapter. Doing so reduces the amount of work required from the end-user, as the later can re-use previously written neural network descriptions and use them directly to be accelerated on MPA. In this work, the Pytorch environment was integrated into MPA's software stack. This integration process starts by splitting the network's layers into smaller tiles. MPA's compiler can then use the tile parameters. A neural network tiling compiler developed in house by a fellow lab mate in HanLab was used. The

Neural Network tiling compiler is briefly presented in Section 3.4. Finally, a dedicated MPA compiler uses the tiling information and the neural network topology extracted from the Pytorch description to compile the necessary instructions that will schedule the computations. The compiler will be discussed in Section 3.6 and the ISA used by the compiler will be presented in Section 3.5.

3.2 PCIe Driver

Interacting with a PCIe device from a host computer requires the use of a kernel driver since we're attempting to access restricted system resources under the supervision of the kernel. Due to the complex nature of the driver, we use a driver provided by Xilinx [7] that was custom written to work in conjunction with the XDMA engine IP [3]. The driver generates high-throughput PCIe memory transactions between the host and the FPGA. The driver registers the FPGA device as a character device, which appears as a file accessible from its file system. The driver operates by mapping virtual addresses from the host's address space onto the FPGA. It maps the 16GB of DDR and a few KB to access the various hardware registers. Since the PCIe BARs do not contain such an extensive range of addresses, it is impossible to do the mapping directly. To circumvent that problem, the host maps the data that it desires to transfer onto a buffer located in its address space. Then, the host will also write in its memory a special descriptor containing information about the physical address location of the buffer containing the data to transfer, in addition to the size of the transfer and the desired FPGA address to write the data into. The XDMA engine in the FPGA is relied on to understand these carefully crafted descriptors. The engine will then start to fetch the data from the host's memory and directing it to the appropriate location in the FPGA. This scheme is used used by the host for both read and write transactions.

3.3 Host FPGA Framework

The goal of the host FPGA software framework is to make it easier to interact with the FPGA device file, in addition to provide debug capabilities. The part of the framework that deals with file operations are loosely inspired by the one deployed by DNNWeaver 2.0 [30]. It consists of wrappers around the OS file operations, providing easier reads and writes to hardware registers and DDR memory. The framework proposed by this thesis then leverages this library by building an abstraction of the FPGA. On top of the activation memory organization seen in Section 2.4, it proposes a memory organization for the weights, the biases, and the batch normalization parameters. Moreover, it provides methods that write the latter neural network layer terms into the said organizations in memory, given the appropriate memory addresses. The addresses are obtained through the compiler described in Section 3.6. Finally, the framework also provides a series of methods that enable us to control the FPGA program execution.

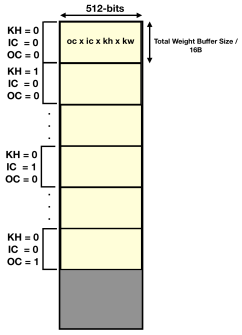

Figure 3-1: Weights

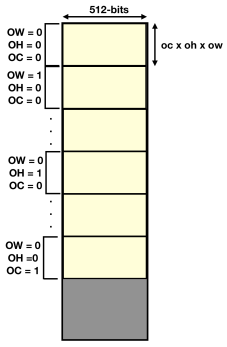

Figure 3-2: Partial Sums

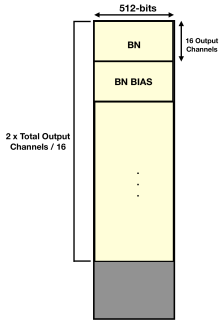

Figure 3-3: BN Parameter

Let us first examine the memory organization for all of the neural network terms. We've previously seen how activations are stored in memory in Section 2.4. The situation for the other terms is different in many ways since we have full software control over how we write organize these terms, as long as when reading from the FPGA, the computations can be carried out properly. We can see in Figure 3-1 how we arrange the weights. The simplest scheme to deal with them in term of hardware is that every new tile requires refreshing every single weight buffer. The scheme reduces the need for any complex routing and is instead common to every tile in every layer. Thus, we organize the weights in a set of weight blocks (WB). The WB map directly onto the weight buffer array in MPA. Therefore, if a tile doesn't fill every weight buffer, the resulting WB will have a few unused locations. Every weight block thus contains the tiled kernels in terms of their heigh, input channels, and output channels. We see in the figure that the WB are placed according to their kernel height dimension first, followed by the input channel dimensions and finally by their output channel dimension. Regarding the partial sums that need to be stored in memory temporarily, they also map the contents of the accumulators exactly. We can refer to Figure 3-2 to see how they are stored in memory. They are formed by reading sequentially all of the used accumulator buffers located at even positions, followed by the odd positions. To not wasted to much bandwidth when writing and reading partial sums, the buffers located at odd positions are sometimes not accessed when configuring MPA with 2-bit weights, since the latter conditions might lead them to be completely unused. Every partial sum block thus contains the tiled output channel, height, and width dimensions. These are stored in memory according to the width dimension first, followed by the height and then by the output channels. Lastly, we draw our attention to the Batch Normalization parameters. The parameters are first stored in a block of 16 batch normalization factors, followed by the 16 biases associated with them. Then, they are stacked with further BN blocks until the total output channels are exhausted. Figure 3-3 illustrates this disposition. Given appropriate compiled tiling parameters, the FPGA software framework exposes a few methods that will split the program data

into these tiles, and write them into memory addresses provided by the compiler. An exception is made for the activations form the first layer in the case that it is a convolutional layer. These tend to have a small number of input channels, thus leaving a lot of the computer core idle. The software framework proposes to convert the convolution into a matrix multiplication [13], and does so by performing a transformation of the input activations into a 2D matrix.

Section 2.2 covered the various paths between the host and the FPGA. While the usefulness of the high-throughput path to DDR has been clearly stated above, no clear purpose besides initiating the computation was described with regards to the low-latency path to the hardware registers. The FPGA software framework additionally provides a few methods that enable to reset the FPGA and a method to continuously poll the FPGA to see if it exhausted all of the instructions. It also lets it read information from the registers, such a the hardware configuration that the FPGA was programmed with, and potentially obtain some performance metrics as measured by the FPGA itself in a cycle-accurate manner.

The last utility that the software framework provides is a debug mode. The mode works in conjunction with the compiler. The later will add instructions that will force every layer to write non-quantized activation results in the partial sum address space. The FPGA software framework will then, after the execution of the program, iterate over every layer, and will read out of DDR the results from every one of those layers. It will then compare them against a software golden model of the layer execution that it ran in the background. The debugger will check for the non-quantized results in addition to the overall complete layer results, including quantization, batch normalization, and the activation function application. Whenever it encounters an error, it will print a debug statement that indicates at which layer the error occurred and their exact location.

3.4 Neural Network Compiler

A major issue to solve when developing a compiler for a neural network accelerator is how to deal with the tiling. Some strides have been made in attempting to make the solutions more general, with the TVM compiler [12] as a major example. For the purpose of this work, an MPA dedicated tiling compiler was developed at HanLab. The compiler relies on acquiring an approximate cost model for carrying computations within MPA's systolic array and for memory transactions. We've obtained these figures by synthesizing the PEs using Cadence's Design Compiler and obtained figures for energy consumption and latency. These are figures for an ASIC implementation and depend on a particular technology node. In this work, we've used a TSMC 45nm library. While the figures correspond to ASIC figures, they could still be used for the FPGA implementation presented in this thesis, as it would provide a good enough approximation in devising an appropriate tiling. With respect to the memory transactions, we use HewlettPackard's GitHub distribution of the Cacti DRAM library to extract an approximate memory access cost model. An MPA python simulator that uses these figures was developed. It attempts to, given a neural network graph, optimize the tiling based on the cost models that are available to us. It carries out the optimization process by iterating all possible tile sizes and all possible outer-loop tiling order exhaustively. The tiling compiler then reports the optimal timing and is saved in a configuration file. This compilation process is extremely time-consuming but only needs to be run once for a given neural network. The compiler from Section 3.6 and by the FPGA software framework from Section 3.3.

3.5 Instruction Set Architecture

The instruction set architecture was designed to involve the minimum number of instructions per layer scheduling. The main elements that contribute to the achievement of the said goal are the Load and Store Buffer instructions. Their

expressivity is enough to convey enough information to direct MPA to carry out the entire inner tile loop through three instructions; two Load Buffer instructions for the activations' and weights' address generators and one Store Buffer instruction for the partial sums' address generator. The ISA uses 128-bit instructions. While this is on the higher side and can be further improved, it was necessary so as not to split up the Load and Store Memory instructions, which contain large DDR address, DDR offsets, and buffer addresses embedded in them. Further work will revisit the ISA to reduce the bitwidth of every instruction to 64-bits. However, since programs will generally be small and contain few instructions, the larger instructions should not affect the computation by much. Furthering this point is the fact that instruction fetching and decoding represents such a small fraction of time compared to the other computations, thus hiding their latency. As for the relatively low instruction buffer density, it won't be an issue since the instructions are not fetched at once but are instead fetched every time the instruction buffer is half empty.

3.5.1 Setup Instruction

Figure 3-4: Setup Instruction

The Setup Instruction contains general information related to the layer to be computed. It contains what kind of dataflow the layer will adhere to and the respective precisions used by each of the operands int he layer, in addition to the precision for the next layer's activations. Finally, it contains information as to whether to consider the weights or the activations to be signed.

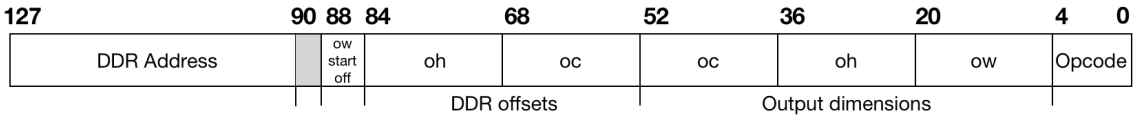

Figure 3-5: Store Memory Instruction

3.5.2 Store Memory Instruction

The Store Memory Instruction facilitates write transactions to the DDR. The data to be transferred is found in the accumulators. It directs the output data sender to either store the contents of the accumulator as partial sums or as activations. The instruction also provides information regarding the output tile parameters. The ow start offset field refers as to whether the tile to be written is part of a partially written Activation Block Column (ABC). The instruction indicates the base DDR address of the transfer. Finally, the DDR offsets enable multiple Activation Block columns of a given tile to be written by providing the relative DDR offset incurred from switching from a partially written ABC to another.

3.5.3 Load Activation Instruction

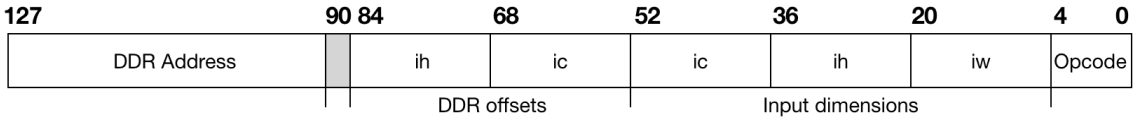

Figure 3-6: Load Activation Instruction

Since the activations are written to the activation buffers according to the memory organization described in Section 2.4.1, it is necessary to use a separate instruction for loading activations and other layer terms such as the weights or the batch normalization parameters. The instruction includes similar information as the Store Memory instruction, that is, it contains tile parameters, DDR offsets, and a DDR base address.

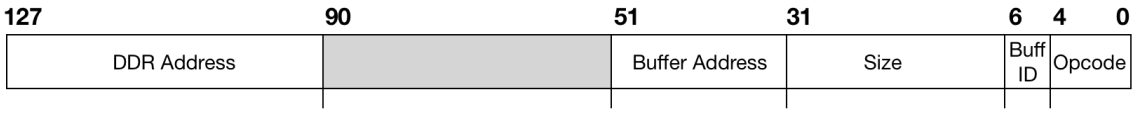

Figure 3-7: Load Memory Instruction

3.5.4 Load Memory Instruction

The weights and the batch normalization parameters (BN) are written as a block directly from memory and are organized in memory by software. Therefore, the instruction solely needs to provide a DDR base address and the size of the transfer. Additionally, the instruction provides the destination buffer through the Buffer ID and the base buffer address.

3.5.5 Store Buffer Instruction

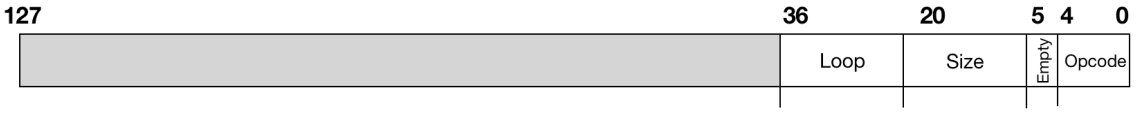

Figure 3-8: Store Buffer Instruction

Programming the accumulator's address generator is relatively simple. The instruction first provides the size of each partial sum inner tile loop computation. Then, it indicates how many loops will be performed. This scheme acknowledges the fact that the accumulators do not need to know the size of the individual components of the partial sum tile. Lastly, the instruction also provides information as to whether the accumulator buffers are empty or not, enabling the pre-loading of values such as Fully-Connected biases or previously computed partial sum values.

3.5.6 Load Buffer Instruction

The Load Buffer instruction programs the activations and weights address generators. They are distinguished through their unique Buffer ID. Then, information is provided relative to the tile size. The width and the height corresponding to

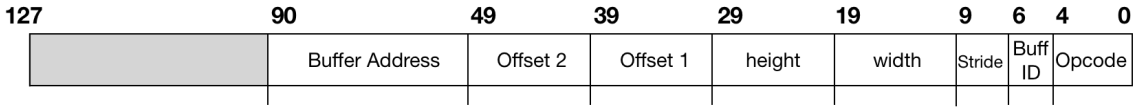

Figure 3-9: Load Buffer Instruction

the iteration size per loop for these dimensions. The two offset fields have different meanings if used for the activations or the weights address generators. In the former case, offset 1 indicates the width of the overall size of the tile's width dimension, and offset 2 refers to the overall height size. With respect to the weights address generator, offset 1 corresponds to the number of input channels to iterate on and offset 2 to the output channels to iterate on. Moreover, the instruction also indicates the stride of the convolution to be computed.

3.5.7 Compute Instruction

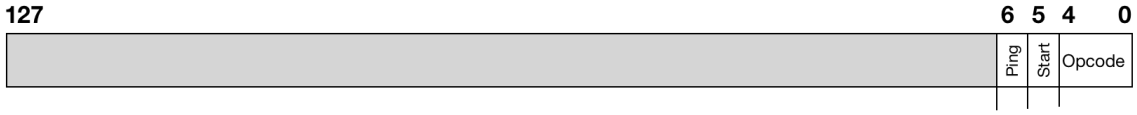

Figure 3-10: Compute Instruction

The Compute Instruction is relatively simple and could potentially be merged with some other instruction. It is called exactly once per inner tile loop computation and triggers its start. The instruction also provides software control on the state of the accumulator's ping-pong buffers.

3.6 Compiler

The inputs to the compiler are a graph object representing a given neural network, in addition to the tiling configuration file obtained from the neural network compiler from Section 3.4. The compiler attempts to schedule the various tiled computations onto the FPGA. The first step is to allocate DDR memory for each layer's terms, including the activations, the partial sums if necessary, the weights, and the BN

parameters. Then, it consists in generating a set of instructions according to the ISA that schedule each DDR read and write transactions according to the tile's outer-loop and programming the various computation address generators to control that will control the systolic array. An example program obtained using the compiler from this section exposing what a layer computation would look like is shown in Listing 3.1.

```
1  # Initiating computation
2  Setup data_flow:WGT_STATIONARY, wgt_prec:8, act_prec:8, next_act_prec:8,
       signed_acts:True, signed_wgts:True
3
4  # Fetching data from DDR Memory
5  LdMem dst_buff:ACTS, iw:196, ih:1, ic:2, ic_ddr_offset:0x80, ih_ddr_offset:0x80,
       ddr_addr:0x1010000
6  LdMem dst_buff:WGTS, size:0x10000, buff_addr:0x0, ddr_addr:0x1000000
7  LdMem dst_buff:BN, size:0x80, buff_addr:0x4000, ddr_addr:0x1015780
8
9  # Inner loop computation
10 LdBuf dst_buff:WGTS, Kw:1, Kh:1, ic:1, oc:2, buff_addr:0
11 LdBuf dst_buff:ACTS, stride:1, w:196, h:1, iw_off:196, ih_off:196
12 StBuf empty:True, size:196, loop:1
13 Comp start:1, ping:True
14
15 # For ping True, Writing back next layer's activations to DDR
16 StMem partial:False, oc:1, oh:14, ow:14, ow_start:0, oc_ddr_offset:0x40,
       oh_ddr_offset:0x40, ddr_addr:0x1014980
17 Comp start:0, ping:True
```

Listing 3.1: Sample code for a single tile computation

The scheme used for memory allocation is simple. It uses a single pointer to keep track of the lowest available address. The instructions are stored starting from the DDR address 0x0. The program data is then stored at a fixed offset above the instructions. Storing the program data is done by incrementing the value of the current DDR address pointer by the said fixed amount. As the compiler attempts to compile the program, it will iterate over each layer sequentially. As it iterates

over a given layer, it will look at every term in the layer, and allocate as much memory as the term needs. It then increments the current DDR address pointer to the address just above the term it just allocated continuously until exhausting every layer in the neural network. This scheme can be further improved to additionally be able to free memory that will no longer be used, as is the case for a layer once its computation was carried out. However, the ample address space provided by the 16GB DDR banks, which is easily extensible to 64GB, means that the optimization will most probably not be warranted. This is compounded by the fact that our low operand precision will further compress the neural networks that MPA operates on.

The compiler uses the tiling configuration file in its scheduling process. It has to handle the variable relative outer-loop ordering. This means that it takes care of inserting a DDR read or write requests whenever a tile is switched out. It optimizes the process by not carrying it out in case there are two or fewer output tiles, as MPA has ping-pong accumulator buffers. It would only need to keep track of the ping-pong state to know which buffer will be written to, and which will send DDR to write requests if done. Regarding the inner loop ordering, the compiler does not need to know anything about the dataflow, and solely configures the address generators with the inner-loop tiling parameters. The hardware address generators will take care of carrying out the mapping onto the compute core.

While pooling layers represent important operations for modern neural networks, the hardware for MPA currently does not handle it. Therefore, the way this compiler deals with them is to transform max-pooling and average pooling layer into convolutional layers with a stride. Once the hardware will be able to handle them, this compiler can easily be extended to provide support for them.

Regarding any specific optimization, the compiler currently only handles one regarding the maximization of the accumulator's buffer usage. In the case where we're dealing with a low weight bit precision, say 2-bits, the compiler will convert the weight precision to 4-bits or 8-bits if the number of output channels is low. This would enable the same tile to fit onto the accelerator but would increase the

accumulator's buffers' capacity by a factor of 2 or 4 due to the buffers' configurable routing seen in Section 1.5.2.

Chapter 4

Performance Analysis

The performance of the design was thoroughly evaluated on a functional implementation of MPA on a Xilinx Virtex-9 UltraScale FPGA, as seen in Chapter 2. For the particular configuration used, refer to Section 2.6. The analysis will include latency figures and power consumption for state-of-the-art networks used for the Imagenet Dataset. Namely, it will include ResNet-18, ResNet-50 and VGG16. The reason for focusing this analysis on the Imagenet Dataset is the fact that the networks that target the said dataset are orders of magnitude more complex than smaller datasets, such as MNIST or CIFAR-10. The latency is measured on a cycle-accurate manner by instantiating a set of hardware registers recording the exact cycle count needed to run each layer of the neural network. The individual layer latencies are then added up to provide full execution latency result.

ResNet18 and **ResNet50** [21] are modern architectures that use at their core a residual block, which feeds the partial sums of a given layer to a few layers below. It enables convolutional neural networks to achieve higher depths while achieving higher accuracies, in addition to lowering their overall cost due to the relatively smaller model size. **VGG-16** [31] is also a performant modern CNN that is mostly characterized by its large size. It is made out of a series of groups containing convolutional layers and a max-pooling layer and terminated by a series of large Fully Connected layers. A detailed analysis for each layer in the respective networks can be found in Appendix A. Every model was tested using dummy data

Precision	ResNet-18			ResNet-50			VGG16		
	Latency **(ms)**	Total Ops **(GOP)**	Performance **(GOP/s)**	Latency **(ms)**	Total Ops **(GOP)**	Performance **(GOP/s)**	Latency **(ms)**	Total Ops **(GOP)**	Performance **(GOP/s)**
2	14.03	3.89	277.19	37.6	8.82	234.56	45.83	19.15	417.85
4	20.85		186.57	77.3		114.1	96.96		197.5
8	44.58		87.26	139.29		63.3	219.8		87.12

Table 4.1: MPA FPGA Performance

and checked using the FPGA debugger described in 3.3.

Table 4.1 summarizes the overall latencies achieved by the networks mentioned above for MPA. Each model was compiled using either 2-bit, 4-bit, or 8-bit operands. The total number of operations (GOP) was extracted from the tiling compiler software. It is based on the layers present in the Pytorch computation graph. The GOPs from the models are slightly different from what can be found in other sources because the MaxPooling layers were replaced with Convolutional layers of stride equals 2, and in the case of VGG16, they were merged with the convolutional layers preceding them. In the case of ResNet-50, the average pooling layer was replaced with a 7x7 Convolution with no padding and stride of 1.

The performance from table 4.1 can better be seen in the context of the Roofline model[29]. The model for this FPGA implementation of MPA can be seen in figure 4-1. The bandwidth was evaluated at 9.552GB/s and is based on the single bank DDR4 bandwidth[6]. The source provided different figures for the read and write bandwidth, and so the figure used was an average of the two, thus assuming a 50/50 access pattern. With respect to the operational intensity, it was derived from the configuration of the systolic array, which in this case is set to be 16x32. The roofline model contains three separate ceilings, each corresponding to the maximal theoretical, computational performance under 2-bit, 4-bit and 8-bit configurations. The lowest one corresponded to the 8-bit configuration and was calculated to be 153.6 GOP/s. With respect to the 4-bit configuration, the ceiling would be 614.4 GOP/s, and finally, the 2-bit configuration could theoretically achieve up to 2.458 TOP/s. The ResNet-18 2b model is located precisely underneath the VGG-16 2b model.

From the Roofline model, we can see how ResNet-50 suffers heavily from the

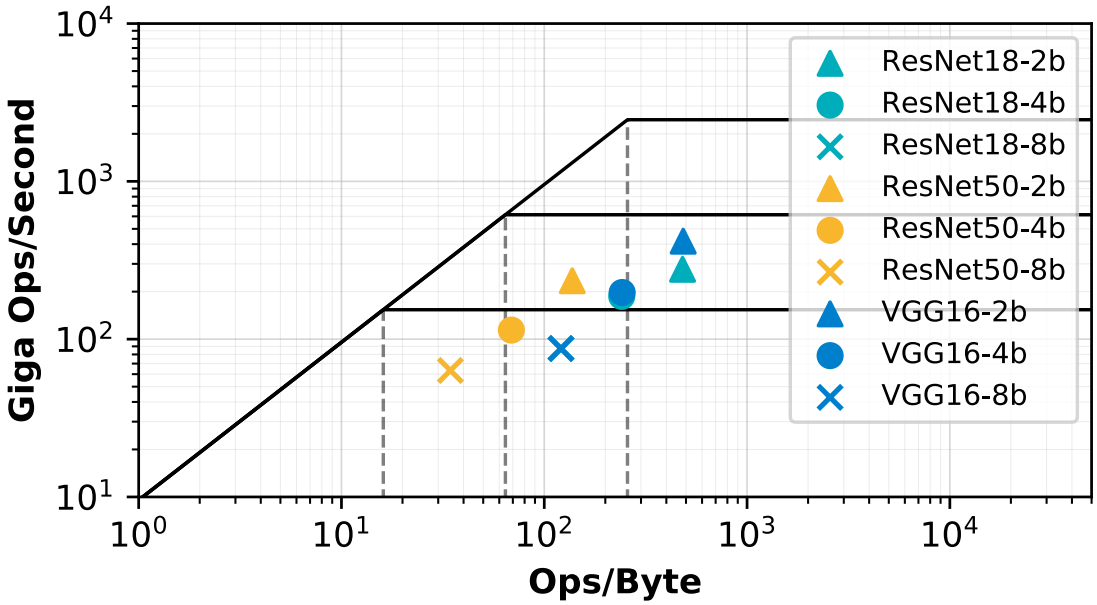

Figure 4-1: Roofline model for the FPGA implementation of MPA

abundance of the 1x1 memory bounded convolutions. The latter issue also affects ResNet-18 but is lower in severity. We also notice how the 8-bit computation approaches the computation bounded threshold and thus suggests that a more extensive array size will be extremely beneficial. We also notice how while the 8b-bit models approach the ceiling of the Roofline model, reaching on average up to 75% to 80% utilization, while the 2-bit and 4-bit configurations struggle to utilize the compute array fully. This can be explained with the fact that memory operations take an increasingly larger proportion of execution time for lower bit precisions. While it is true that lower bit precision will require smaller memory accesses, their relative size decreases linearly, while the computational ability increases polynomially. Thus, it becomes increasingly harder for lower bit precision configurations to take full advantage of the compute array. This effect can be better observed from looking at the tables in Appendix A. Additionally, this problem is compounded by the fact that the implementation suffers from the lack of activation fetch and weight fetch pipelining, i.e., the fact that the fetches don't overlap the computation.

With respect to power consumption and efficiency, we can see the power consumption and computation efficiency for each of the previous models and bit

Precision	ResNet-18			ResNet-50			VGG16		
	Power consumption **(W)**	Total Ops **(GOP)**	Efficiency **(GOP/s/W)**	Power consumption **(W)**	Total Ops **(GOP)**	Efficiency **(GOP/s/W)**	Power consumption **(W)**	Total Ops **(GOP)**	Efficiency **(GOP/s/W)**
2	28	3.89	9.9	28	8.82	8.38	27	19.15	15.5
4	27		6.9	27		4.23	27		7.3
8	26		3.36	26		2.4	25		3.5

Table 4.2: MPA FPGA Power Efficiency

precision configuration for MPA in table 4.2. The power was measured using a power meter connected to the PC's power supply. The power for the machine with the on-board PCIe FPGA programmed was measured, and compared against the power seen when the FPGA is running of the previously mentioned models. The difference between the two would correspond to the dynamic power consumption of the FPGA. The main thing that is noticeable is the slightly higher powers consumed by lower bit precisions. While it can be counter-intuitive, we notice how the relatively higher efficiencies for lower bit precisions put that into perspective. The higher powers can be explained by the lower utilization achieved by, the lower bit configurations, as seen in the tables in Appendix A.

	VGG-SVD[17]	**VGG16**[26]	**VGG16**[20]	**Syntegy**[35]	**DoReFa**[18]	**AlexNet**[24]	**FINN-R**[9]	**This Work**			
Platform	Zynq XC7Z045	Stratix-V	Zynq 7Z020	Zynq ZU3EG	Zynq 7Z020	Stratix-V	Virtex-9 (AWS F1)	VCU1525			
Clock (MHz)	150	120	214	250	200	150	155	150			
Power (W)	9.63	25.8	3	5.5	2.26	26.2	-	25-28			
DSP	780	727	198	37	89	201		183			
LUT	183k	120k	27k	24k	44k	231k	477k	151k			
FF	127k	-	-	30k	-	-	-	65.4k			
BRAM	486	1480	(272KB)	70	105.5	2210	1332	311			
Precision	16b	8b-16b	8b	4b	2b	2b-1b	2b-1b	2b	4b	8b	
								277.19	186.57	87.26	(ResNet-18)
Performance (GOP/s)	136.97	117.8	123	47.09	410.2	1963.96	11431	234.56	114.1	63.3	(ResNet-50)
								417.85	197.5	87.12	(VGG-16)
								9.9	6.9	3.36	(ResNet-18)
Power Efficiency (GOP/s/W)	14.22	4.57	41	8.56	181.51	74.96	-	8.38	4.23	2.4	(ResNet-50)
								15.5	7.3	3.5	(VGG-16)

Table 4.3: Performance comparison between MPA and other state-of-the-art FPGA accelerators

A side to side comparison for this implementation with other FPGA neural network accelerators for a wide variety of bit precision configurations can be seen in table 4.3. The logic cell count for MPA differs from what is seen on 2.6 since this table includes the results from the synthesis report. The reason in doing so is that these results do not include the PCIe logic and solely focus on user-defined logic, which results in a fairer comparison with other designs. We notice how MPA is

generally competitive, particularly when compared with models providing lower bit precisions. However, MPA seems to be on the higher side with respect to power consumption. The later is due to the fact that the VCU1525 board used in this implementation uses PCIe blocks, which consume more power than an equivalent implementation that could access data locally.

Chapter 5

Conclusion

5.1 Contribution

Large modern Neural Network models represent a particular computational challenge for currently available processors. To reduce their computational overhead, using lower bit precisions has proven to be a very efficient scheme while preserving the overall accuracy of the network. This works presented a novel mixed-precision architecture that is more efficient in terms of resource usage and computational ability than other mixed-precision neural network accelerators. The architecture was then deployed on a Virtex-9 Ultrascale+ FPGA, and a software framework was built on top of it to enable ease of integration with modern neural network tools. The system was then evaluated on modern neural network models such as ResNet-18, ResNet-50, and VGG-16. The implementation was shown to perform at up to 417GOP/s for the 2-bit configuration, 197.5GOP/s under the 4-bit configuration and up to 87.26GOP/s for the 8-bit configuration. Additionally, peak performances of up to 1.4TOP/s were seen for the 2-bit configuration, 364GOP/s for the 4-bit configuration and finally up to 121.76GOP/s for the 8-bit configuration. In terms of power efficiency, the FPGA implementation operates between 25-28W and achieves power efficiencies between 2.4 to 15.5 GOP/s/W.

5.2 Future Work

The implementation presented in this thesis has the potential to provide even better performance results. The main issue hindering the performance is the lack of pipelined activation and weight fetches. It particularly affects the 2-bit and 4-bit configurations since the memory operations take an increasingly larger role in the computation time. Doing so could provide up to 1.5x-2x latency reductions and enable state-of-the-art performance. Additionally, while the server class Virtex-9 Ultra-Scale+ FPGA was chosen as the main development platform, the implementation does not currently take full advantage of the capabilities provided by the FPGA. Mainly, the compute array size could be substantially increased to 16x64 or even 16x128 configurations to lift even higher the ceiling of maximal computational capabilities. On the other hand, the clock speed could be increased past the 200MHz threshold with further micro-optimizations and further enhance the performance of the design. With respect to the ISA, it could be further optimized to reduce the instructions to 64-bits. The current implementation also doesn't make use of the special activation fetcher routing logic, thus limiting the maximum capacity of the buffers when the number of input channels would otherwise enable further usage of the buffers. Finally, many fetched activations are left unused when carrying out a convolution with a stride, since the system is not currently optimized to skip specific DDR addresses when fetching the required inputs to the convolution.

Another main incentive to develop an FPGA implementation of the mixed-precision architecture presented in this thesis was to merge the high reconfigurability of the FPGA with a software framework that would enable co-design when facing a given neural network. The current system has the potential to be tied in with such a framework due to the exposed parameters of the FPGA design and shared software API used by the members of Hanlab currently working on the aforementioned software.

Appendix A

Detailed Layer by Layer Analysis for Select Networks

A.1 ResNet-18

Layer	Precision	Latency (ms)	Total Ops (GOP)	Performance (GOP/s)	Utilization	Arithmetic Intensity (Ops/Byte)
conv1	2	1.17	0.24	200.99	8.18%	355.16
	4	1.44		164.43	26.76%	177.65
	8	3.82		61.75	40.20%	88.84
maxpool.1	2	2.81	0.23	82.37	3.35%	875.11
	4	3.98		58.09	9.45%	437.98
	8	6.31		36.63	23.85%	219.10
layer1.0.conv1	2	0.65	0.23	355.36	14.46%	2033.02
	4	1.12		206.43	33.60%	1018.80
	8	2.06		112.20	73.04%	509.98
layer1.0.conv2	2	0.71	0.23	323.47	13.16%	2033.02
	4	1.19		194.64	31.68%	1018.80
	8	2.13		108.76	70.81%	509.98
layer1.1.conv1	2	0.65	0.23	355.80	14.48%	2033.02
	4	1.12		206.32	33.58%	1018.80
	8	2.06		112.20	73.05%	509.98
	2	0.71		325.75	13.25%	2033.02

layer1.1.conv2	4	1.19	0.23	195.09	31.75%	1018.80
	8	2.12		109.01	70.97%	509.98
layer2.0.conv1	2	0.82	0.12	140.77	5.73%	1197.64
	4	1.09		105.72	17.21%	602.01
	8	1.68		68.79	44.78%	301.81
layer2.0.conv2	2	0.44	0.23	529.70	21.55%	2519.30
	4	1.36		170.50	27.75%	1266.72
	8	1.93		119.78	77.98%	635.14
layer2.0.shortcut	2	0.18	0.01	72.89	2.97%	315.57
	4	0.87		14.77	2.40%	159.80
	8	0.24		53.40	34.77%	80.41
layer2.1.conv1	2	0.32	0.23	730.94	29.74%	2519.30
	4	0.69		333.29	54.25%	1266.72
	8	1.90		121.76	79.27%	635.14
layer2.1.conv2	2	0.43	0.23	533.44	21.71%	2519.30
	4	1.35		170.99	27.83%	1266.72
	8	1.93		119.90	78.06%	635.14
layer3.0.conv1	2	0.42	0.12	273.68	11.14%	1003.24
	4	0.61		188.02	30.60%	506.12
	8	1.11		104.11	67.78%	254.20
layer3.0.conv2	2	0.74	0.23	313.67	12.76%	1295.79
	4	1.02		226.61	36.88%	651.64
	8	2.02		114.34	74.44%	326.76
layer3.0.shortcut	2	0.05	0.01	236.47	9.62%	442.08
	4	0.23		55.83	9.09%	229.11
	8	0.14		92.67	60.33%	116.69
layer3.1.conv1	2	0.41	0.23	562.93	22.91%	1295.79
	4	0.70		329.57	53.64%	651.64
	8	2.01		115.30	75.07%	326.76
layer3.1.conv2	2	0.73	0.23	314.93	12.81%	1295.79
	4	1.02		227.13	36.97%	651.64
	8	2.02		114.30	74.41%	326.76
layer4.0.conv1	2	0.15	0.12	793.38	32.28%	361.63
	4	0.43		269.95	43.94%	181.98
	8	1.62		71.24	46.38%	91.28
	2	0.92		251.08	10.22%	378.69

layer4.0.conv2	4	1.49	0.23	155.07	25.24%	189.98
	8	2.45		94.45	61.49%	95.15
layer4.0.shortcut	2	0.10	0.01	134.33	5.47%	277.60
	4	0.39		33.29	5.42%	145.23
	8	0.34		38.22	24.88%	74.33
layer4.1.conv1	2	0.25	0.23	921.82	37.51%	378.69
	4	0.67		346.91	56.46%	189.98
	8	2.37		97.58	63.53%	95.15
layer4.1.conv2	2	0.91	0.23	252.76	10.28%	378.69
	4	1.49		154.93	25.22%	189.98
	8	2.45		94.51	61.53%	95.15
Avg-pool.1	2	0.44	0.03	58.81	2.39%	7.98
	4	0.87		29.45	4.79%	3.99
	8	1.81		14.19	9.24%	2.00
fc	2	0.02	0.00	63.79	2.60%	7.98
	4	0.03		31.98	5.21%	3.99
	8	0.07		15.58	10.14%	1.99

Table A.1: Detailed layer by layer ResNet-18 performance

A.2 ResNet-50

Layer	Precision	Latency **(ms)**	Total Ops **(GOP)**	Performance **(GOP/s)**	Utilization	Arithmetic Intensity **(Ops/Byte)**
conv1	2	1.17	0.24	202.16	8.23%	355.16
	4	1.43		164.65	26.80%	355.16
	8	3.82		61.75	40.20%	88.84
maxpool.1	2	2.80	0.23	82.58	3.36%	875.11
	4	3.97		58.18	9.47%	875.11
	8	6.35		36.44	23.72%	219.10
layer1.0.conv1	2	0.28	0.03	90.23	3.67%	252.14
	4	0.26		97.13	15.81%	252.14
	8	0.49		52.40	34.12%	63.27
layer1.0.conv2	2	0.65	0.23	354.40	14.42%	2033.02
	4	1.12		206.44	33.60%	2033.02

	8	2.06		112.29	73.11%	509.98
layer1.0.conv3	2	2.17	0.10	47.30	1.92%	399.81
	4	1.75		58.67	9.55%	399.81
	8	1.75		58.65	38.18%	100.55
layer1.0.shortcut	2	0.57	0.10	179.21	7.29%	399.81
	4	0.72		143.59	23.37%	399.81
	8	1.02		100.28	65.29%	100.55
layer1.1.conv1	2	0.30	0.10	337.24	13.72%	402.21
	4	0.39		262.87	42.79%	402.21
	8	1.22		84.28	54.87%	100.70
layer1.1.conv2	2	0.65	0.23	355.63	14.47%	2033.02
	4	1.12		207.03	33.70%	2033.02
	8	2.06		112.42	73.19%	509.98
layer1.1.conv3	2	2.17	0.10	47.40	1.93%	399.81
	4	1.76		58.42	9.51%	399.81
	8	1.75		58.65	38.18%	100.55
layer1.2.conv1	2	0.31	0.10	336.79	13.70%	402.21
	4	0.39		262.80	42.77%	402.21
	8	1.22		84.20	54.82%	100.70
layer1.2.conv2	2	0.65	0.23	354.53	14.43%	2033.02
	4	1.12		206.58	33.62%	2033.02
	8	2.06		112.37	73.16%	509.98
layer1.2.conv3	2	1.82	0.10	56.60	2.30%	399.81
	4	1.36		75.48	12.29%	399.81
	8	1.36		75.71	49.29%	100.55
layer2.0.conv1	2	0.52	0.21	398.44	16.21%	662.39
	4	0.56		364.42	59.31%	662.39
	8	1.91		107.37	69.90%	166.01
layer2.0.conv2	2	1.05	0.23	220.63	8.98%	1384.96
	4	1.09		212.03	34.51%	1384.96
	8	2.58		89.59	58.33%	347.84
layer2.0.conv3	2	1.09	0.10	94.51	3.85%	704.22
	4	0.81		126.35	20.56%	704.22
	8	1.49		69.10	44.99%	179.84
	2	0.26		778.27	31.67%	1096.74

layer2.0.shortcut	4	0.50	0.21	412.01	67.06%	1096.74
	8	2.67		76.93	50.09%	278.76
layer2.1.conv1	2	0.17	0.10	620.28	25.24%	719.37
	4	0.30		347.27	56.52%	719.37
	8	0.93		110.41	71.88%	180.81
layer2.1.conv2	2	0.32	0.23	728.64	29.65%	2519.30
	4	0.69		333.41	54.27%	2519.30
	8	2.29		100.94	65.72%	635.14
layer2.1.conv3	2	1.09	0.10	94.47	3.84%	704.22
	4	0.81		127.24	20.71%	704.22
	8	1.49		69.09	44.98%	179.84
layer2.2.conv1	2	0.17	0.10	620.63	25.25%	719.37
	4	0.30		347.93	56.63%	719.37
	8	0.93		110.47	71.92%	180.81
layer2.2.conv2	2	0.32	0.23	731.83	29.78%	2519.30
	4	0.69		334.10	54.38%	2519.30
	8	2.29		100.89	65.68%	635.14
layer2.2.conv3	2	1.09	0.10	94.54	3.85%	704.22
	4	0.81		126.37	20.57%	704.22
	8	1.49		69.17	45.03%	179.84
layer2.3.conv1	2	0.17	0.10	620.23	25.24%	719.37
	4	0.29		348.61	56.74%	719.37
	8	0.93		110.37	71.85%	180.81
layer2.3.conv2	2	0.32	0.23	730.51	29.72%	2519.30
	4	0.69		334.04	54.37%	2519.30
	8	2.29		100.88	65.68%	635.14
layer2.3.conv3	2	0.91	0.10	112.83	4.59%	704.22
	4	0.62		166.94	27.17%	704.22
	8	1.48		69.24	45.08%	179.84
layer3.0.conv1	2	0.41	0.21	504.13	20.51%	1108.86
	4	0.46		444.81	72.40%	1108.86
	8	2.07		99.08	64.50%	279.53
layer3.0.conv2	2	0.43	0.23	537.21	21.86%	1071.06
	4	1.04		222.93	36.28%	1071.06
	8	2.88		80.31	52.29%	269.68
	2	0.88		116.48	4.74%	753.11

layer3.0.conv3	4	3.13	0.10	32.82	5.34%	753.11
	8	1.13		91.27	59.42%	197.16
layer3.0.shortcut	2	0.93	0.21	221.45	9.01%	958.01
	4	5.73		35.84	5.83%	958.01
	8	9.41		21.85	14.22%	246.57
layer3.1.conv1	2	0.08	0.10	1342.45	54.62%	788.62
	4	0.38		270.34	44.00%	788.62
	8	1.01		101.80	66.28%	199.51
layer3.1.conv2	2	0.41	0.23	563.60	22.93%	1295.79
	4	1.04		222.02	36.14%	1295.79
	8	2.81		82.16	53.49%	326.76
layer3.1.conv3	2	0.88	0.10	116.50	4.74%	753.11
	4	3.13		32.88	5.35%	753.11
	8	1.13		90.64	59.01%	197.16
layer3.2.conv1	2	0.08	0.10	1350.33	54.95%	788.62
	4	0.38		269.38	43.84%	788.62
	8	1.01		101.56	66.12%	199.51
layer3.2.conv2	2	0.41	0.23	564.54	22.97%	1295.79
	4	1.04		222.60	36.23%	1295.79
	8	2.82		81.93	53.34%	326.76
layer3.2.conv3	2	0.88	0.10	116.59	4.74%	753.11
	4	3.13		32.81	5.34%	753.11
	8	1.13		90.66	59.02%	197.16
layer3.3.conv1	2	0.08	0.10	1348.33	54.86%	788.62
	4	0.38		269.02	43.79%	788.62
	8	1.01		101.83	66.30%	199.51
layer3.3.conv2	2	0.41	0.23	564.18	22.96%	1295.79
	4	1.04		222.12	36.15%	1295.79
	8	2.83		81.71	53.19%	326.76
layer3.3.conv3	2	0.88	0.10	116.59	4.74%	753.11
	4	3.13		32.85	5.35%	753.11
	8	1.13		90.75	59.08%	197.16
layer3.4.conv1	2	0.08	0.10	1349.86	54.93%	788.62
	4	0.38		269.60	43.88%	788.62
	8	1.01		101.69	66.21%	199.51

layer3.4.conv2	2	0.41	0.23	563.87	22.94%	1295.79
	4	1.04		222.90	36.28%	1295.79
	8	2.82		82.13	53.47%	326.76
layer3.4.conv3	2	0.88	0.10	116.50	4.74%	753.11
	4	3.13		32.80	5.34%	753.11
	8	1.14		90.48	58.91%	197.16
layer3.5.conv1	2	0.08	0.10	1335.48	54.34%	788.62
	4	0.38		270.43	44.02%	788.62
	8	1.01		101.68	66.20%	199.51
layer3.5.conv2	2	0.41	0.23	563.16	22.92%	1295.79
	4	1.04		221.99	36.13%	1295.79
	8	2.83		81.69	53.19%	326.76
layer3.5.conv3	2	0.79	0.10	129.54	5.27%	753.11
	4	3.00		34.22	5.57%	753.11
	8	1.13		91.01	59.25%	197.16
layer4.0.conv1	2	0.16	0.21	1281.62	52.15%	976.66
	4	0.76		270.65	44.05%	976.66
	8	2.02		101.90	66.34%	247.78
layer4.0.conv2	2	0.29	0.23	791.31	32.20%	367.59
	4	0.86		270.06	43.95%	367.59
	8	3.24		71.34	46.45%	92.35
layer4.0.conv3	2	0.30	0.10	342.66	13.94%	331.61
	4	0.44		235.71	38.36%	331.61
	8	1.28		80.57	52.45%	86.32
layer4.0.shortcut	2	0.54	0.21	378.91	15.42%	355.39
	4	3.56		57.72	9.39%	355.39
	8	5.67		36.24	23.59%	90.78
layer4.1.conv1	2	0.17	0.10	587.38	23.90%	345.30
	4	0.43		237.36	38.63%	345.30
	8	1.17		87.68	57.08%	87.22
layer4.1.conv2	2	0.25	0.23	910.21	37.04%	378.69
	4	0.66		349.42	56.87%	378.69
	8	2.34		98.89	64.38%	95.15
layer4.1.conv3	2	0.30	0.10	340.38	13.85%	331.61
	4	0.44		236.03	38.42%	331.61

	8	1.28		80.43	52.37%	86.32
layer4.2.conv1	2	0.17	0.10	587.34	23.90%	345.30
	4	0.43		236.94	38.56%	345.30
	8	1.18		87.31	56.84%	87.22
layer4.2.conv2	2	0.25	0.23	936.15	38.09%	378.69
	4	0.66		350.43	57.04%	378.69
	8	2.34		98.90	64.39%	95.15
layer4.2.conv3	2	0.29	0.10	357.78	14.56%	331.61
	4	0.43		237.07	38.59%	331.61
	8	1.28		80.46	52.38%	86.32
Avg-pool.1	2	6.97	0.41	58.97	2.40%	8.00
	4	13.95		29.48	4.80%	8.00
	8	29.00		14.18	9.23%	2.00
fc	2	0.06	0.001	66.16	2.69%	7.99
	4	0.13		32.41	5.28%	7.99
	8	0.27		15.39	10.02%	2.00

Table A.2: Detailed layer by layer ResNet-50 performance

A.3 VGG-16

Layer	Precision	Latency **(ms)**	Total Ops **(GOP)**	Performance **(GOP/s)**	Utilization	Arithmetic Intensity **(Ops/Byte)**
conv1_1	2	4.42	0.17	39.25	1.60%	151.79
	4	3.96		43.74	7.12%	75.91
	8	7.57		22.91	14.92%	37.96
conv1_2	2	11.49	0.92	80.51	3.28%	906.33
	4	13.74		67.32	10.96%	453.28
	8	22.89		40.40	26.30%	226.67
conv2_1	2	6.58	1.85	281.02	11.43%	2941.62
	4	11.00		168.11	27.36%	1472.01
	8	18.47		100.17	65.22%	736.30
conv2_2	2	4.08	0.92	226.48	9.22%	1691.23
	4	5.51		167.98	27.34%	846.41
	8	12.15		76.14	49.57%	423.40

conv3_1	2	2.64	1.85	700.42	28.50%	4815.29
	4	5.60		330.53	53.80%	2414.08
	8	16.20		114.15	74.32%	1208.66
conv3_2	2	2.80	3.70	1321.36	53.77%	6541.73
	4	15.05		245.74	40.00%	3276.80
	8	32.58		113.53	73.91%	1639.89
conv3_3	2	2.18	0.92	424.72	17.28%	2268.91
	4	4.85		190.72	31.04%	1137.31
	8	11.26		82.14	53.48%	569.37
conv4_1	2	1.30	1.85	1420.38	57.80%	4047.81
	4	7.81		236.70	38.53%	2033.02
	8	17.04		108.52	70.65%	1018.80
conv4_2	2	3.59	3.70	1029.69	41.90%	4570.11
	4	11.85		312.25	50.82%	2290.85
	8	34.27		107.95	70.28%	1146.88
conv4_3	2	1.07	0.92	866.84	35.27%	1272.74
	4	3.40		272.26	44.31%	638.17
	8	10.23		90.37	58.84%	319.54
conv5_1	2	0.83	0.92	1117.01	45.45%	1418.96
	4	2.80		329.99	53.71%	711.72
	8	9.12		101.37	66.00%	356.42
conv5_2	2	0.83	0.92	1116.84	45.44%	1418.96
	4	2.80		329.78	53.67%	711.72
	8	9.12		101.36	65.99%	356.42
conv5_3	2	0.34	0.23	674.01	27.43%	367.59
	4	0.88		262.75	42.76%	184.40
	8	3.28		70.52	45.91%	92.35
fc6	2	3.06	0.21	67.09	2.73%	8.00
	4	6.39		32.18	5.24%	4.00
	8	12.99		15.82	10.30%	2.00
fc7	2	0.51	0.03	65.28	2.66%	8.00
	4	1.05		31.83	5.18%	4.00
	8	2.13		15.79	10.28%	2.00
fc8	2	0.13	0.01	65.05	2.65%	7.99
	4	0.26		31.39	5.11%	4.00

	8	0.53		15.47	10.07%	2.00

Table A.3: Detailed layer by layer VGG-16 performance

Bibliography

[1] AXI DataMover v5.1 : LogiCORE IP Product Guide. https://www.xilinx.com/support/documentation/ip_documentation/axi_datamover/v5_1/pg022_axi_datamover.pdf. Accessed: 08-07-2019.

[2] AXI Reference Guide. https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf. Accessed: 08-07-2019.

[3] DMA/Bridge Subsystem for PCI Express v4.1 : Product Guide. https://www.xilinx.com/support/documentation/ip_documentation/xdma/v4_1/pg195-pcie-dma.pdf. Accessed: 08-07-2019.

[4] SmartConnect v1.0 : LogiCORE IP Product Guide. https://www.xilinx.com/support/documentation/ip_documentation/smartconnect/v1_0/pg247-smartconnect.pdf. Accessed: 08-07-2019.

[5] SoCs with Hardware and Software Programmability. https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html. Accessed: 08-07-2019.

[6] VCU1525 Reconfigurable Acceleration Platform. https://www.xilinx.com/support/documentation/boards_and_kits/vcu1525/ug1268-vcu1525-reconfig-accel-platform.pdf. Accessed: 08-07-2019.

[7] Xilinx PCI Express DMA Drivers and Software Guide. https://www.xilinx.com/Attachment/Xilinx_Answer_65444_Linux.pdf. Accessed: 08-07-2019.

[8] Xilinx Virtex UltraScale+ FPGA VCU1525 Acceleration Development Kit. https://www.xilinx.com/products/boards-and-kits/vcu1525-a.html. Accessed: 08-07-2019.

[9] PreuÃ§er T.B. Fraser N.J. Gambardella G. OâĂŹbrien K. Umuroglu Y. Leeser M. Blott, M. and K. Vissers. Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks. ArXiv e-prints. arXiv:1809.04570, 2018.

[10] H. Mao C. Zhu, S. Han and W. Dally. Trained ternary quantization. ICLR '17.

[11] McMillan K. Carloni, L. and A. Sangiovanni-Vicentelli. Theory of latency-insensitive design. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol. 20, no. 9, pp. 1059-1076, Sep 2001.

[12] Moreau T. Jiang Z. Zheng L. Yan E. Cowan M. Shen H. Wang L. Hu Y. Ceze L. Guestrin C. Chen, T. and A. Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 578âĂŞ594, 2018.

[13] Woolley C. Vandermersch P. Cohen J. Tran J. Catanzaro B. Chetlur, S. and E. Shelhamer. cudnn: Efficient primitives for deep learning. arXiv, 2014. arxiv.org/abs/1410.0759.

[14] N. Suda L. Lai B. Chau V. Chandra H. Sharma, J. Park and H. Esmaeilzadeh. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network. ISCA '18.

[15] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. ICML '15.

[16] J. Lu J. Lin, Y. Rao and J. Zhou. Runtime neural pruning. NIPS '17.

[17] S. Yao K. Guo B. Li E. Zhou J. Yu T. Tang N. Xu S. Song Y. Wang J. Qiu, J. Wang and H. Yang. Going deeper with embedded fpga platform for convolutional neural network. FPGA '16.

[18] Cao W. Zhou X. Jiao L., Luo C. and Wang L. Accelerating low bit-width convolutional neural networks with embedded fpga. FPL '17.

[19] N. P. et al Jouppi. In-datacenter performance analysis of a tensor processing unit. ISCA '17.

[20] S. Yao Y. Wang Y. Xie K. Guo, S. Han and H. Yang. Software-hardware codesign for efficient neural network acceleration. MICRO '17.

[21] S. Ren K. He, X. Zhang and J. Sun. Deep residual learning for image recognition. CVPR '16.

[22] S. Krithivasan and M. J. Schulte. Multiplier architectures for media processing. ACSSC '03.

[23] Bottou L. Bengio Y. LeCun, Y. and P. Haffner. Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11):2278âĂŞ2324, November 1998a.

[24] Liu L. Luk W. Liang S., Yin S. and Wei S. Fp-bnn: Binarized neural network on fpga. Neurocomputing 2018.

[25] Talathi S. Lin, D. and S. Annapureddy. Fixed point quantization of deep convolutional networks. ICML '16.

[26] G. Dasika A. Mohanty Y. Ma S. B. K. Vrudhula J. Seo N. Suda, V. Chandra and Y. Cao. Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks. FPGA '16.

[27] T. Hetherington T. Aamodt A. Moshovos P. Judd, J. Albericio. Stripes: Bit-serial deep neural network computing. MICRO '16.

[28] H. Mao S. Han and W. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. ICLR '16.

[29] A. Waterman S. Williams and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. Commun. ACM, 52(4):65-76, 2009.

[30] Park K. Mahajan D. Amaro E. Kim K. Shao C. Mishra A. Sharma, H. and H. Esmaeilzadeh. From high-level deep neural models to fpgas. MICRO '16.

[31] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. ICLR '15.

[32] T. Yang V. Sze, Y. Chen and J. Emer. Efficient processing of deep neural networks: A tutorial and survey. arXiv preprint arXiv:1703.09039, 2017.

[33] J. Emer Y. Chen and V. Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. ISCA '16.

[34] L. Rasnayake Y. Umuroglu and M. Sjalander. Bismo: A scalable bit-serial matrix multiplication overlay for reconfigurable computing. FPL '18.

[35] B. Wu T. Zhang L. Ma G. Gambardella M. Blott L. Lavagno K. Vissers J. Wawrzynek et al. Y. Yang, Q. Huang. Synetgy: Algorithm-hardware co-design for convnet accelerators on embedded fpgas. FPGA '19.