Final Year Thesis (2021 – 2022)
ELEC Final Report

Hardware Acceleration for AI Processing

Project ID: ZW01a-21

Supervisor: Professor Wei ZHANG

Author (Student ID): CHENG Yih (20566646)

Date: 13 September 2021

Main Objective

Field-programmable gate arrays (FPGAs) are highly configurable hardware that allows short prototyping and design periods. Moreover, the introduction of high-level synthesis (HLS) has taken a step further in abstracting hardware level programming and circuit designs. This thesis aims to implement a deep residual network (ResNet), a specific CNN architecture that has helped achieve superior image recognition accuracy, on an FPGA device. While trying to minimize overall latency, the accuracy shall be kept to the same level as when ResNet is deployed on CPUs. The ultimate goal is to export a fast ResNet FPGA core that can be embedded into larger hardware systems.

Objective Statements

1. Design an algorithm to optimize convolution layer computation on FPGA which can increase acceleration speed 5 times compared to for loop implementation.
2. To deploy ResNet on Zynq FPGA boards that results in an overall 25-30 times acceleration speed for CNN components when compared to implementing on 4 core ARM processors.
3. Test the ResNet model on existing datasets like CIFAR-10 and reach accuracies as existing research.

Contents

ABSTRACT iii
SECTION 1—INTRODUCTION 1
1.1 Background and Engineering Problem 1
1.2 Objectives 1
1.2.1 Objective Statements 1
1.3 Literature Review of Existing Solutions 2
SECTION 2—METHODOLOGY 3
2.1 Overview 3
2.1.1 Product/System Description 3
2.1.2 FPGA-based Hardware Accelerator Diagram 5
2.1.3 FPGA-based Hardware Accelerator Execution Workflow 6
2.1.4 Components List 7
2.1.5 ECE Knowledge 7
2.2 Objective Statement Execution—Plan for implementing ResNet architecture on FPGA 8
2.2.1 Accelerator Algorithm and Architecture Improvement 8
2.2.2 Implementing and Optimizing ResNet architecture on FPGAs 8
2.2.2.1 Evaluation of Objective 2.2.1 and Objective 2.2.2 24
2.2.3 Testing of Accelerator on Existing Image Datasets 25
2.2.3.1 Evaluation of Objective 2.2.3 26
2.3 Main Objective Evaluation & Discussion 27
SECTION 3—CONCLUSION 27
REFERENCES 29
APPENDICES 31
Appendix A. Project Schedule 31
Appendix B. Budget 33
Appendix C. Meeting Minutes 34

List of Illustrations

List of Figures

Figure 1. ResNet building block (from [1]) 3
Figure 2. ResNet18 network 4
Figure 3. Schematic of the FPGA accelerator system 5
Figure 4. ResNet18 execution workflow 6
Figure 5-1. Block array partitioning 9
Figure 5-2. Cyclic array partitioning 9
Figure 5-3. Complete array partitioning 9
Figure 6. Double buffer implementation timing 10
Figure 7. Conv layer hardware mapping 17
Figure 8. Batch normalization timing overview 18
Figure 9. ReLU layer timing overview 19
Figure 10. ReLU layer timing overview 19
Figure 11. Relationship between CPU and FPGA 20

List of Tables

Table 1. Result size of each ResNet18 layer 4
Table 2. List of Specifications 7
Table 3. Input reuse algorithm walkthrough process 14
Table 4. Input reuse algorithm performance (100 iterations) 16
Table 5. Input reuse algorithm resource utilization (FIL=8) 16
Table 6. Computation engine performance (100 iterations) 16
Table 7. Computation engine resource utilization 16
Table 8. CPU and FPGA timing 20
Table 9. Timing Performance 21
Table 10. Resource Utilization 21
Table 11. Timing Performance 22
Table 12. Resource Utilization 22
Table 13. Timing Performance 23
Table 14. Resource Utilization 23
Table 15. Accelerator performance on difference platforms ($N = 5$) 24
Table 16. Clock Performance 25
Table 17. Accelerator performance on difference platforms ($N = 5$) 26
Table 18. Resource Utilization 26
Table 19. CiFAR-10 testing results 26
Table 20. FPGA-based accelerator schedule. 31
Table 21. Expected budget 33
Table 22. Action Items from the Previous Meeting 34
Table 23. Action Items for Next Meeting 34

List of Code Blocks

Code Block 1. Primitie convolution layer implementation 12
Code Block 2. Tiled convolution layer implementation 12
Code Block 3. Input reuse convolution layer implementation 14

ABSTRACT

Field-programmable gate arrays (FPGAs) are highly configurable hardware that allows various parallelism designs and high performance computing. In this thesis, we aim to design a fast ResNet18 neural network FPGA accelerator to demonstrate the potential of deploying FPGAs into the machine learning field. Optimization methods are customized for each layer in view of minimizing latency and lowering unnecessary resource allocations. AWS cloud FPGA instances serve as the main deployment platform for testing and evaluating our results. In the end, we are able to reach a 12-13X speedup in convolution layer and a 5-6X speedup of the whole network when compared to even high end processors such as AMD Ryzen 9 CPUs. The implementation of each module are highly dynamic to provide scalibility and integration abilities with other systems.

SECTION 1—INTRODUCTION

1.1 Background and Engineering Problem

Neural networks have been widely adopted in the AI and machine learning field. They are the cornerstone of deep learning algorithms. As the name goes, neural networks are inspired by the way biological neurons signal each other in human brains, and therefore model structures are built to mimick this phenonenom. Among different types of neural networks, convolutional neural networks (CNNs) are proven to be one of the most common yet effective methods when encountering computer vision tasks, including image classification and object detection. However, the performance of traditional CNNs will be limited after the depth of the network reaches a certain threshold, and training these "deep" networks are very difficult [1]. Various researches have been conducted to identify and solve related problems.

The deployment effiency of CNNs in real world scenarios are greatly boosted thanks to advanced processors that are capable of handling parallel workflows, which is a key point in accelerating the training and inferencing of CNNs. Multicore central processing units (CPUs) and graphical processing units (GPUs) stand out as two of the best and common pieces of hardware that can assist in this aspect. However, the seemingly ever-growing demand of CNNs has also greatly pushed research and development regarding custom hardware accelerators in recent years, including field-programmable gate arrays (FPGAs) [2]. The high reconfigurability of FPGAs allows short design periods as well as prototyping various network structures, which aligns to the fast-evolving CNN architectures. The cost of large scale FPGA installations is also significantly lower compared to traditional CPUs and GPUs, making mass deployment of them a realistic method. Moreover, the introduction of high-level synthesis (HLS) have taken a step further in abstracting hardware level programming, simplifying redundant and repeating steps that may be encountered during register-transfer level (RTL) designs and allowing even software and algorithm engineers to be involved without sufficient circuit domain knowledge.

The ability to run CNN-based deep learning algorithms on FPGAs can provide a solution regarding the rapidly soaring computational power demands for machine learning and AI. Not only is it cost effective, but is also capable of providing accuracy and results of the same level regarding similar tasks compared to traditional hardware accelerators. In addition, with the development of cloud technology that can host and manage large-scale infrastructures, a practical business model of deploying massive quantities of FPGAs in data centers to further level up the computational power provided could be foreseen in the near future [3]. Therefore, proving that working with deep neural networks on configurable devices like FPGAs is very practical, and from a higher perspective, it may be a new solution towards the soaring demand of computing power for AI and machine learning.

1.2 Objectives

Field-programmable gate arrays (FPGAs) are highly configurable hardware that allows short prototyping and design periods. Moreover, the introduction of high-level synthesis (HLS) has taken a step further in abstracting hardware level programming and circuit designs. This thesis aims to implement a deep residual network (ResNet), a specific CNN architecture that has helped achieve superior image recognition accuracy, on an FPGA device. While trying to minimize overall latency, the accuracy shall be kept to the same level as when ResNet is deployed on CPUs. The ultimate goal is to export a fast ResNet FPGA core that can be embedded into larger hardware systems.

1.2.1 Objective Statements

4. Design an algorithm to optimize convolution layer computation on FPGA which can increase acceleration speed 5 times compared to for loop implementation.
5. To deploy ResNet on Zynq FPGA boards that results in an overall 25-30 times acceleration speed for CNN components when compared to implementing on 4 core ARM processors.
6. Test the ResNet model on existing datasets like CIFAR-10 and reach accuracies as existing research.

1.3 Literature Review of Existing Solutions

Research in deep learning CNN architectures have always been a main focus in machine learning. However, most of the effort are put towards creating new algorithms or model structures to further optimize the network, whereas relatively less work are done by utilizing hardware acceleration techniques. Three deep learning research papers, [2], [5], [6], that mainly focus on FPGA hardware domain will be reviewed below.

Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Network [2]

While FPGA have been proved to perform better in deep CNN trainings than CPUs, there are still a lot of design improvement that can be made to further level up the efficiency. A critical problem awaiting to be solved is the matching between computation throughput and memory bandwidth, as the same model design with different implementations could end up with 90% performance difference [2]. To address this problem, the authors introduced the Roofline model which is able to link the total solution performance together with computation-to-communication (CTC) ratio of off-chip memory and the peak performance that could be attained. According to this analysis, a CNN accelerator composed of parallel tree-shaped computation engine and a memory sub-system was proposed. The implementation result showed a 17.42x speedup compared to single thread CPU software solution and a 4.8x speedup compared to 16 thread CPU software solution.

This research paper provided an in-depth perspective for analyzing computation and memory operations of different implementations. Along with the above analysis and the Roofline model, the best solution for a CNN model that would result in best performance and high resource utilization can be identified. However, there are still limitations to the proposed accelerator, for instance the uniform unroll factor for all convolutional layers might not be optimal for all solutions and may degrade the performance. In addition, this paper only discussed acceleration and optimization of CNN layers with no mentioning of full connected (FC) layers. This may result in under estimation of memory access operations as CNN layers are compute-bound whereas FC layers are usually memory-bound.

A Programmable Parallel Accelerator for Learning and Classification [5]

This research paper proposes a FPGA-prototyped accelerator – MAPLE that is capable of handling both learning and classification tasks. The team identified that computational tasks in most deep learning scenarios involved both small-scale and large-scale matrices, which lead to the structure consisting of parallel streaming of data through 2D vector processing elements (PE). Each PE would then only be responsible of a partial part of the matrix multiplication operation, and every one of them matches to a smart memory block. The smart memory block is responsible of intermediate parameter calculations, for example, comparisions or finding maximum values. This allowing the PE to directly start the next vector operation in the next clock cycle without wasting time on additional workload. These data in the memory block can later be extracted and concatenated as the final parameter output.

By designing smart memory blocks, MAPLE utilizes in-memory processing for all PEs, therefore reducing the on-chip and off-chip memory communication. As for CNN workloads, it can reduce off-chip access traffic by as much as 76x. However, the significant performance improvement comes with the consequence of massive on-chip memory consumption. Therefore, it would not be possible for deep CNNs to be deployed in this hardware impelementation as the on-chip memory will not be able to hold on with massive amount of parameters.

Throughput-Optimized FPGA Accelerator for Deep Convolutional Neural Networks [6]

This research paper proposed a parallel framework for FPGA accelerators working with CNNs. Four parallelism levels for accelerating CNNs are identified, being task-level parallelism, layer-level parallelism, loop-level parallelism, and operator-level parallelism. Task-level parallelism could be deployed to execute multiple prediction tasks simultaneously. Layer-level parallelism allows the FPGA to pipeline and perform calculation of different network layers within the same clock cycle. Loop-layer parallelism does loop optimization to boost efficiency when encountering loop calculations. Operator-level parallelism is done on the sub-convolution process when doing image-kernel convolution. LeNet [7], AlexNet [8], and VGG-S [9] CNN architectures were implemented with this framework, and it proved to give a 14.84x, 6.96x, and 4.79x boost in performance while improving power efficiency by 51.84x, 24.69x, and 16.46x compared to CPUs (Intel Core-i7) respectively.

The four parallelism levels of this framework could work well in most CNN architectures given that the forward propagation and backward propagations are a one direction process and depends only on the previous layer outputs. However for network architectures like ResNet [1] or even U-Net [10], since short jumps and long jumps exist, a new step of adding the previous layer output with another identity input is required. In this case, layer-level parallelism would be rather inefficient as the pipeline now needs to wait for both layers to finish computing, and several clock cycles will also be consumed for addition.

All three of these papers show different approaches toward implementing CNNs on FPGAs. Some aimed to boost efficiency by utilizing large amount of on-chip memory, while some focused on parallelism techniques to optimize the CNN structures to suit FPGAs. Other papers such as [3] and [11] provide a different perspective of looking into FPGA hardware acceleration, as it focuses more on how large scale FPGA devices can work and collaborate together in cloud data centers.

This thesis aims to find an optimal implementation of ResNet on FPGA devices. By optimizing CNN computation tasks first, it allows the project to be built on top of a steady backbone. Through revising ResNet structure to suit FPGA devices by doing similar analysis like [2], it may further improve hardware efficiency. Last but not least, by utilizing hardware acceleration techniques such as that mentioned in [2] and [12], the unique advantages of FPGA devices can be maximized.

SECTION 2—METHODOLOGY

2.1 Overview

2.1.1 Product/System Description

In this thesis, a FPGA-based accelerator for a 18-layer ResNet will be built. Deep residual network (ResNet) [1] emerged as one of the most popular deep learning architectures as it is capable of achieveing superior accuracy while using significantly lower parameters and computational operations [13]. As the depth of different neural networks started to expand, experiments showed that deep networks started to encounter degradation problems, meaning that the training accuracy were lowered while the depth of the network grew. To solve this problem, ResNet introduced two ways of mapping method: residual mapping (the straight connections in Figure 1) and identity mapping (the rounded connection in Figure 1). The whole module in Figure 1 is the core "building block" of ResNet, and as the figure shows, a "short connection", or the identity mapping, is introduced to allow residual learning. The output of this building block is $y = F(x) + x$, where $F(x)$ is the residual mapping or the difference equaling $y - x$, and x is identity mapping, which is identical to the input to this building block. If training networks have reached the optimal results, then as the network expands even deeper, residual mapping $F(x)$ will be limited to near 0 while identity mapping x will dominate the input to the next building block. In theory, the network will always stay in the optimal stage once it reaches it rather than continue training as the network grows deeper, which resolves the above mentioned problem.

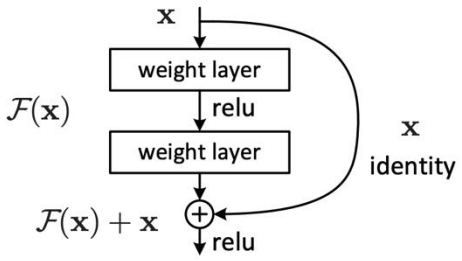

Figure 1. ResNet building block (from [1])

With the ResNet building block, we are able to construct the full ResNet18 network according to [1]. Figure 2 shows the detail construction of the whole network structure. Input images will first go through a convolution layer, a batch normalization layer, and a relu layer before traversing through 9 ResNet building blocks. Different colors in the blocks of figure 2 represents a downsampling in the network. Unlike

conventional downsampling methods in machine learning models, downsampling in ResNet18 is done by increasing the stride of a convolution layer to 2. As an example, downsampling is performed between $Block\ 2.1$ and $Block\ 3.0$ (neighboring blue and yellow block) by increasing the stride of the convolution layer in $Block\ 3.0$ to 2. Downsampling between $Block\ 5.1$ and $Block\ 6.0$ acts in accordance too. For residual layers that face inconsistency between identity mapping x and residual mapping $F(x)$, zero padding is performed to x. All other convolution operations are performed with stride 1 and pad 1. A global average pooling layer followed by a full connected layer is inserted at the end of the network. Table 1 shows each layer's parameter size for reference supposing that the input size to the first $Conv$ layer is $[N][Channel][H1][W1]$ and the weight of the $Conv$ layer is $[FIL1][Channel][CONV\_H][CONV\_W]$. $H2$, $W2$, and $H3$, $W3$ are all calculated according to the input size of the previous layer and stride 1 pad 1.

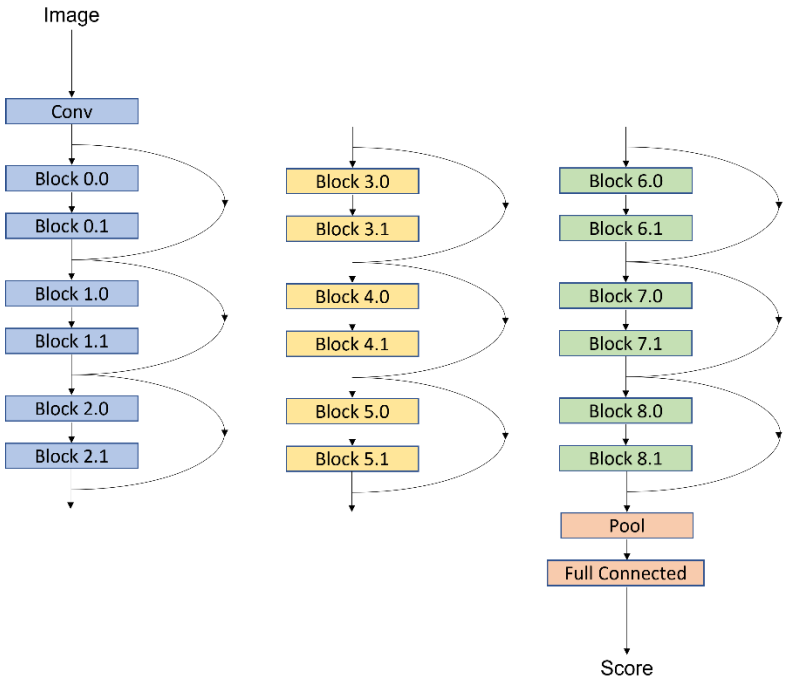

Figure 2. ResNet18 network

Block ID	Result size	Block ID	Result size	Block ID	Result size
Conv	$[N][FIL1][H1][W1]$				
Block 0.0 … Block 2.1	$[N][FIL1][H1][W1]$	Block 3.0 … Block 5.1	$[N][FIL2][H2][W2]$	Block 6.0 … Block 8.1	$[N][FIL3][H3][W3]$
				Pool	$[N][FIL3]$
				FC	$[N][Class]$

Table 1. Result size of each ResNet18 layer

Given the structure of ResNet, its implementation on FPGA hardware can be constructed. The design process will be seperated into three phases. First, revise the structure of ResNet, especially focusing on convolution layers. Since on-chip RAM is limited on FPGAs, reducing parameters by implementing techniques such as pruning [14] is key to acceleration efficiency. The second step is to readjust CNN to make training more capatible on FPGAs. For example, fix-point-arithmetics may be needed to utilize digital-signal-processing (DSP) units on FPGAs, and in this case parameters must be processed to accommodate with FPGA requirements. Third, different convolution computing algorithms will be utilized and disassembled in order

to maximize efficiency and resource utilization in FPGAs. Some methods have been provided in [2] and [15] to accelerate the convolution process during FPGA computation. Convolution computation improvement will be measured by clock cycles here to compare with existing traditional algorithms. In addition, the whole process would also be monitored and the clock cycle that is consumed will be compared with that of a traditional CPU. Last but not least, the trained model will be deployed and tested with existing CIFAR-10 image dataset to evaluate the efficiency and accuracy.

2.1.2 FPGA-based Hardware Accelerator Diagram

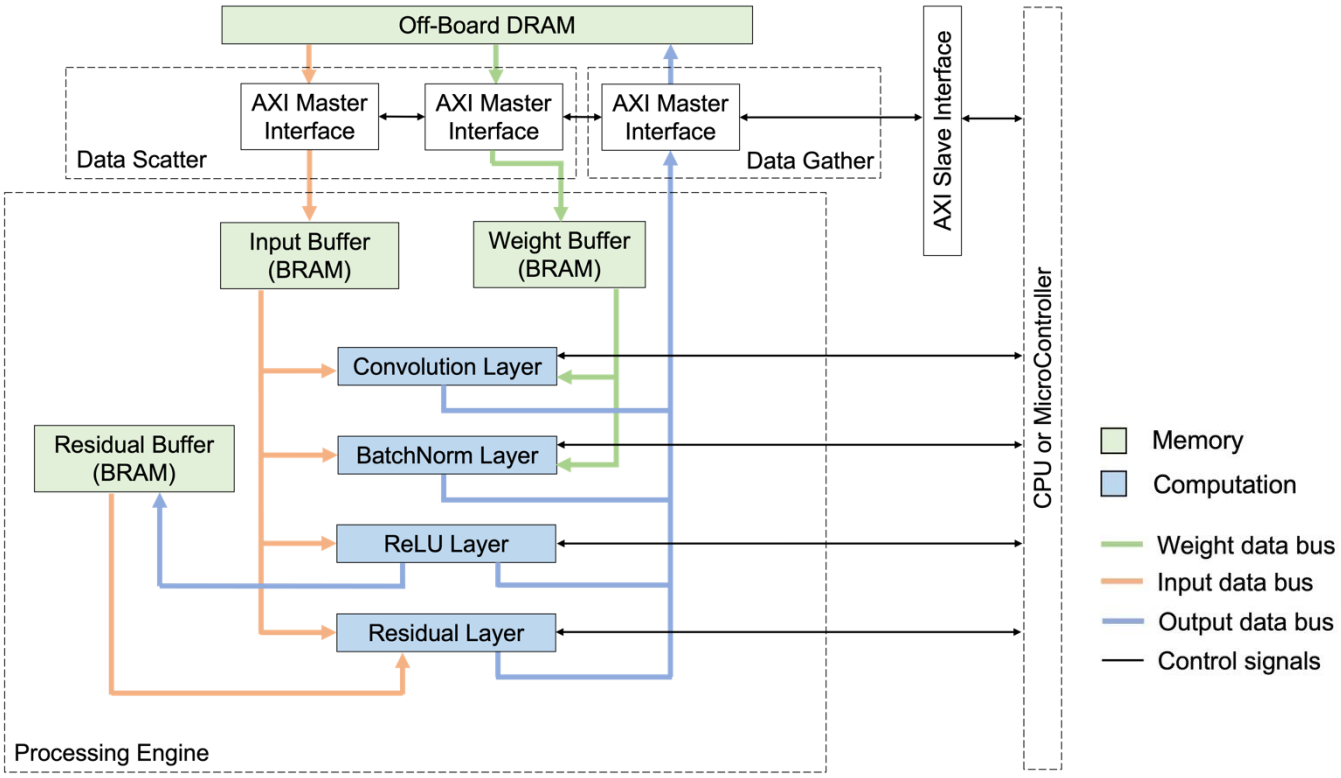

Figure 3. Schematic of the FPGA accelerator system

Figure 3 is the basic design of the FPGA accelerator for ResNet18. The structure of the accelerator can be logically divided into two parts, with the green part being memory components and the blue part being computation components. To further elaborate the design idea behind, the memory component can be further categorized into off-chip and on-chip memory.

Off-chip memory or external memory such as DRAM mainly stores pre-calculated weights and intermediate parameter values as these values are too big to all be stored on on-chip memory, especially when the networks grows deeper.

On-chip memory like BRAM mainly takes advantage of locality and acts like a buffer between external memory and the processing engine. It will be useful when implementing streaming input algorithms while processing engines are doing parallel computation.

Similar to [4], AXI4 Interface ports are designed to transfer data between BRAM and DRAM. The Slave port is responsible for the control signals while the Master ports read input from DRAM and write to corresponding BRAM blocks. A data scattering module is implemented to read the requested inputs and weights through read channels and writes them to specified on-chip memory blocks. A data gathering module is implemented to gather the calculated results and controls write channels on the AXI4 Master port to store the results back

to DRAM. Multiple AXI4 Master ports can be implemented for parallel accessing of input feature maps. The total number of AXI Master ports can be calculated with the below equation:

$$N_{ports} = \max(N_{map}, N_{gather}) + 1 .$$

N_{map} is the read ports that for data mapping of input feature maps from DRAM to BRAM, while N_{gather} is the write ports for data gathering of results to DRAM. Since read and write channels on AXI4 Master ports are independent, the maximum of these two is the required number of ports. In addition, one AXI4 Master port is needed for reading the layer weights and bias.

The processing engine (PE) is designed according to each building block of ResNet. Depending on the control signal from the CPU, one of the layers will be run per device kernel invoke. All layers read and write from the same BRAM buffers. In addition, a residual buffer is added to cache intermediate results for the special residual operations in ResNet.

Similar to most traditional network architectures, ResNet18 includes a convolution layer followed by batch normalization layer to improve training efficiency and ReLU activation layer to introduce some nonlinearity to the model. The scale operation is to align the dimensions of identity mapping x and residual mapping $F(x)$ since element-wise addition can be only done on same channel dimension matrix. An additional adder is implemented, which will perform addition of the two mappings when needed. After going through 17 basic building blocks, the output will be send into a full connected layer to conclude the 18 layer ResNet. In the PE architecure, it can be observed that even though layers are run in a stream line direction, different layers may use similar hardware resources. For example, both batch normalization layer and pooling layers utilizes adders, and the multiplier functions are also similar in batch normalization layer and scale layer. Therefore, instead of assigning separate resources to each layer, we can make each module layer flexible and share resources through some pipeline designing. In this way, the network can expand deeper without the concern of running out of resources.

2.1.3 FPGA-based Hardware Accelerator Execution Workflow

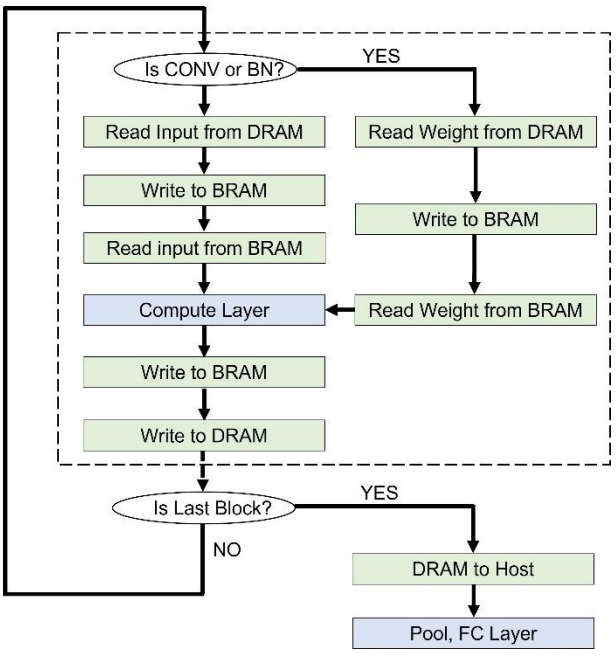

Figure 4. ResNet18 execution workflow

Figure 4 indicates the execution workflow of the accelerator and the host device. Depending on the control signal, the accelerator performs respective computation intensive tasks. Convolution and batch normalization

layers require pre-trained parameters, and their address in the device global memory will be known to the accelerator during runtime by control signals. Data scatter and gather modules are responsible for the input data and the weight data transfer between BRAM and DRAM.

After the FPGA task is done, host will copy the output from device DRAM to host memory. Pooling and full connected layer are performed on host processor for the final results. Overall, most of the repetitive and computation intensive jobs are performed by the FPGA device. Detailed relationship between the host and the device are discussed in **2.2.2 *Part 8***.

2.1.4 Components List

Table 2. List of Specifications

Items	Specifications/Model
Field Prgrammable Gate Array (FPGA)	Xilinx Virtex UltraScale FPGA VCU108/ AWS-FPGA Instance
Personal Computer with a minimum 8Gb memory	ROG Strix Scar 17 G733
Operating System	Ubuntu LTS 18.04 / Windows 10/11
Development Software	Vitis High Level Synthesis (HLS), Vivado Design Suite 2020.02

2.1.5 ECE Knowledge

ELEC 4320 --- FPGA-based Design: From Theory to Practice

The major knowledge base of this thesis will come from this course since all the theorems and hands-on impelementations of FPGAs are taught in it. From initializing a FPGA project to doing simulation, synthesis, implementation, then generating bit-stream to program on FPGA, this design flow will be deployed throughout the whole thesis. In addition, high level synthesis (HLS) is also introduced, and it will play a major role as it is an abstraction of complex algorithms for RTL design. Hence, this course will serve as the backbone for this thesis.

ELEC 4240 --- Deep Learning in Computer Vision

This course introduced various CNN architectures, especially focusing on computer vision. In fact, the main deep learning algorithm, ResNet, that will be implemented in this thesis was taught in this course. The math and also the theorem behind these different CNN architectures would play a major role in optimizing and implementing neural networks on FPGA-based accelerators.

ELEC 2350 --- Introduction to Computer Organization and Design

Understanding in-depth of how a microprocessor works is vital for this thesis, as multiple acceleration engines and ALUs might be implemented to fulfill the objectives. In addition, floating point arithmetics for computers were also taught in this course, which is key knowledge as it requires a massive amount of float point calculation in this thesis. Last but not least, the idea of pipelining to construct multi-cycle processors is also one of the most important concepts when doing CNN accelerations. Multiple research papers, including some of them in the literature review part, also proved this to be a very important technique.

ELEC 3300 --- Introduction to Embedded Systems

FPGA devices include all kinds of components as that of a typical microprocessor control board, and therefore the knowledge taught in this embedded system course will be useful. It will require understanding of bus communications, processors, along with DRAM, BRAM memory structures and access methods to complete this thesis.

2.2 Objective Statement Execution—Plan for implementing ResNet architecture on FPGA

In this subsection, detailed procedures and algorithms will be introduced in view of meeting set objectives on target and on time. Note that **2.2.1** and **2.2.2** will be discussed together with the full optimization methodology of the FPGA accelerator.

2.2.1 Accelerator Algorithm and Architecture Improvement

Objective: Design an algorithm to optimize convolution layer computation on FPGA which can increase acceleration speed for 5 times compared to for loop implementation.

2.2.2 Implementing and Optimizing ResNet architecture on FPGAs

Objective: To deploy ResNet on Zynq FPGA boards that results in an overall 25-30 times acceleration speed for the total training process when compared to implementing on 4 core ARM processors.

To achieve our objectives in **2.2.1** and **2.2.2**, various optimizations to the accelerator are deployed. The following part consists of the thorough design flow and all optimization methods that were implemented in our ResNet18 accelerator. In addition, we will also discuss the interactions between the FPGA and the CPU.

Part 1. Task Arrangement

According to the architecture of ResNet18, the workload can be distributed for running on the CPU and the FPGA. Generally the CPU is more suitable with less repetive computations and more sequential signal controls, while the FPGA is more capable of running computation intensive calculations.

1. CPU : We assigned the global average layer and the full connected layer to be run on the CPU as they will only be executed once. In addition, these two layers take up relatively more memory than any other layers due to the parameters amounts. Deploying them on the FPGA will result in memory waste as additional idle buffer space must be set up that can only be used by these two layers in the end. Therefore, we decided to place these two layers in the CPU.
2. FPGA : As mentioned in **2.1.1**, ResNet18 is mainly constructed of repetitive residual blocks, each in a Conv-BatchNorm-ReLU-Conv-BatchNorm-Residual-ReLU structure. We implemented these four layer tasks, convolution, batch normalization, relu, and residual, on FPGAs as they are highly parallelly configurable. In addition, these residual blocks are executed multiple times during a ResNet18 inferencing, which makes it a huge advantage for resource reutilization. As a result, the full structure of the residual block is implemented on the FPGA.

Part 2. Array Partitioning

Array partitioning increases read and write (r/w) ports of a specific random accessed data. Normally arrays are stored as BRAM blocks in the accelerator, providing only one independent read and one independent write channel for data blocks smaller than 18Kbs. This allows at most one r/w request in a clock cycle, which greatly limits parallel computing abilities. Array partitioning increases r/w ports by mapping the same data block to different BRAMs. This provides multiple r/w in the same clock cycle, and depending on the partitioning factor and dimemsion, we can greatly increase the parallel computing ability. Below we briefly introduce the three partitioning methods provided by Xilinx and their implementations in our accelerator.

1. Block partitioning :

   This cuts the original array into consecutive subarrays that will be mapped to different BRAM blocks. Details are shown in figure 5-1.

   The weight buffer of the batch normalization layer is block partitioned with a factor of 4 to divide it into the mean buffer, variance buffer, gamma buffer, and beta buffer. This allows a batch normalization computation to fininsh in one clock cycle.

2. Cyclic partitioning :

   This creates smaller subarrays by interleaving data in the original array. The subarrays will then be mapped to different BRAM blocks. Details are shown in figure 5-2.

   The weight buffer of the convolution layer is cyclic partitioned with a factor of 9. We will leave the details of this part to the convolution optimization in ***part 4***.

3. Complete partitioning :

   This maps every data element in the original array to a different BRAM block, which corresponds to individual registers. Details are shown in figure 5-3.

   Each data element in the computation engine of the convolution layer, including a portion of the weight buffer and the input buffer, is completely partitioned. We will leave the details of this part to the convolution optimization in ***part 5***.

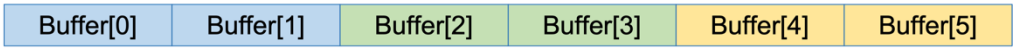

Figure 5-1. Block array partitioning

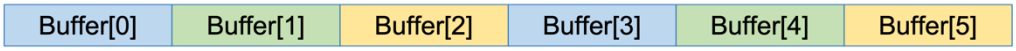

Figure 5-2. Cyclic array partitioning

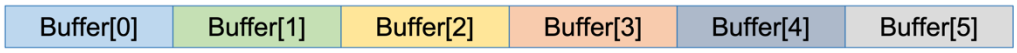

Figure 5-3. Compete array partitioning

**Different color represents a different BRAM block being mapped to.*

Part 3. Data Scatter/Gather Modules Optimization

Memory access can greatly bottleneck the accelerator's performance if not carefully designed. In this subpart, we will focus on the data read and write between off-chip DRAM and on-chip BRAM, which corresponds to the data scatter and data gather models in the hardware diagram of figure 3. There are two main interfaces to transfer data between DRAM and BRAM, being streaming interface and bursting interface. Streaming interface works as an infinite FIFO that works with continuous data in the stream sequentially. Burst interface, on the other hand, tries to reduce latency by read or write maximum amount of data per request.

In this ResNet18 accelerator, streaming is not as useful since calculations does not align with the streaming sequence in convolution and batch normalization layers. Furthermore, streaming interface requires the expected data size d_size_{ex} to match exactly with the streamed data size d_size_{stream}, which is a major challenge as every layer and different blocks require different parameter values. Redundant padding of $(d\_size_{ex\_max} - d\_size_{stream}) \times sizeof(float)$ bytes can be concatenated at the end of the stream to align d_size_{ex} and d_size_{stream}, but these additional data will all be disgarded later on. An alternative method to implement customized data access modules for each streaming interface is also possible, but it will result in poor resource utilization since every layer will require a new interface. On the other hand, burst interface does not require an initial data size as it simply burst reads or writes a fix amount of data, $max\_read\_burst\_lenth$ and $max\_write\_burst\_lenth$ respectively, until the data block ends. At most only $(max\_read\_burst\_length - 1) \times sizeof(float)$ or $(max\_write\_burst\_length - 1) \times sizeof(float)$ of data access time will be wasted, which is significantly smaller compared to the redundant padding of streaming interface. As a result, burst interfaces are chosen for this accelerator.

1. Burst read/write between DRAM and BRAM :

   Burst memory access is a method to maximize data throughput bandwidth and to minimize latency. Bursting typically provides a 4-5 time performance improvement compared traditional memory accessing methods depending on the burst size [18]. AXI bus interfaces are implemented to provide burst read and write from DRAM to BRAM and vice versa. This allows the data scatter and gather models to directly work with a whole cache line when provided one address instead of reading one word, wait for the next word's address, then reading the next word. For burst interfaces, $max\_read\_burst\_length$ and $max\_write\_burst\_length$ are fixed through pragma clauses and implies the amount of data to deal with per burst operation. In our accelerator, the $max\_read\_burst\_length$ and $max\_write\_burst\_length$ are both fixed to 256 as according to [18], the effective bandwidth increases as the burst transmission length increases.

2. Double buffering :

   In view of lowering the data transfer latency between BRAM and DRAM, double buffering, or ping pong buffering is implemented in both two data mapping modules. With this design, the process of moving data from DRAM and BRAM can be divided into two sub-parts. The first is the calculation of address offset to read from DRAM and write to median buffer, while the second is to copy the data from median buffer to BRAM blocks. With two additional median buffer, the latency between the first step and the second step above can be overlapped. $FIL \times H \times W$ bytes of data are being read from DRAM and stored to median buffer 0 in the current clock cycle, while $FIL \times H \times W$ bytes of data are being written to input/weight buffer from median buffer 1 in the same clock cycle. A detailed clock timing is provided for better illustration in figure 6.

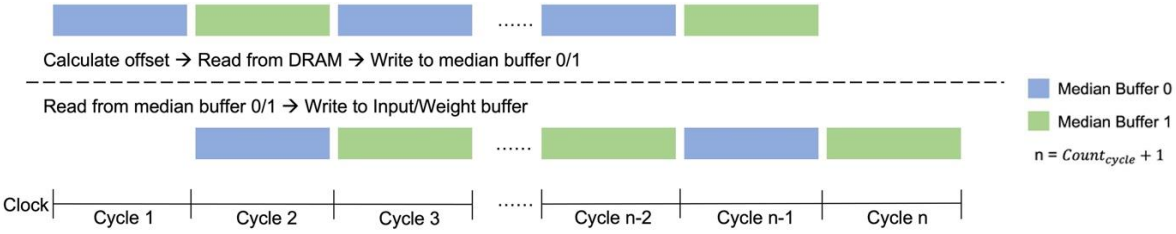

Figure 6. Double buffer implementation timing

For each additional median buffer, the memory it allocates is:

$$Buffer_{median} \geq LayerInput_{max} = \begin{cases} (FIL \times H \times W)_{max}, & Input\ buffer \\ (FIL)_{max} \times H_{wei} \times W_{wei}, & Weight\ buffer \end{cases}$$

As an example, given the ResNet18 network structure in [1] and using Cifar10 as testing dataset, the median buffer for input buffer should be at least $max(16 \times 32 \times 32, 32 \times 16 \times 16, 64 \times 16 \times 16) \times sizeof(float)$ bytes, while it must be at least $max(16, 32, 64) \times 3 \times 3 \times sizeof(float)$ bytes for weight buffer of the convolution layer.

Since AXI4 Master port reads DRAM in burst mode, the latency of reading greatly depends on the set burst length. Suppose $length$ bit of data is being transferred through a AXI4 Master port with $burstlength$ burst length and a clock frequency of $freq$. The latency of such a transmission is:

$$Latency_{DRAM\_to\_buf} = \frac{length}{freq \times sizeof(float) \times C_{burstlength}},$$

where $C_{burstlength}$ is the respective effective bandwidth coefficient of such $burstlength$ port.

Let the latency of the first sub-part, i.e. the steps of calculating address offset and reading from DRAM to write to median buffer, be $Latency_{pp\_1}$. In this case, the value of $Latency_{pp\_1}$ is:

$$Latency_{pp\_1} = T_{offset\_calculation} + Latency_{DRAM\_to\_buf}$$

Let the latency of the second sub-part, i.e. reading from median buffer and writing to input/weight buffer, be $Latency_{pp\_2}$. In this case, the value of $Latency_{pp\_2}$ is:

$$Latency_{pp\_2} = \frac{length}{freq \times sizeof(float)}.$$

The total clock cycle needed for retrieving all input/weight data is:

$$Count_{cycle} = \begin{cases} N_{input_{count}} = \dfrac{N \times FIL \times H \times W}{Buffer_{median}}, & Input\ buffer \\ FIL_{layer} = \dfrac{FIL \times FIL \times H_{wei} \times W_{wei}}{Buffer_{median}}, & Weight\ buffer \end{cases}$$

As a result, the total latency of implementing a ping pong buffer is:

$$Latency_{pp_{total}} = 1 \times Latency_{pp_1} + \left(Count_{cycle} - 1\right) \times \max\left(Latency_{pp\_1}, Latency_{pp\_2}\right) + 1 \times Latency_{pp\_2}.$$

Compared to the traditional streaming read latency, which is:

$$Latency_{stream} = Count_{cycle} \times (Latency_{pp\_1} + Latency_{pp\_2}).$$

The below conclusion can be made:

$$Latency_{pp\_total} \geq Latency_{stream}.$$

Furthermore, double buffering works perfectly with burst interfaces as each pipelined process can directly be matched to one fixed $max\_read\_burst\_length$ or $max\_write\_burst\_length$ burst operation.

Part 4. Convolution Layer Optimization

Since over 99% of computation powers in CNN architectures are spent on convolution operations, its algorithm is a primary key to the acceleration efficiency and speed of the system [16]. The acceleration strategy will not only effect the computation mechanism but also memory access, and to lower memory access while improving fully utilizing computation resources is the main goal of this objective.

The detailed convolution layer optimization is divided into three parts in the following subsections. We first introduce a new input reuse algorithm that is built from the traditional tiling method. Hardware implementation is provided in the second part, followed by additional parallelism optimization. Last, we will briefly discuss the sum tree to calculate the final output.

1. Input reuse algorithm and convolution tiling :

   Primitive convolution layer implementation is rather intuitive. The pseudo code of this version is shown in code 1. It does not take into account any resource reuse or performance optimization.

```
// initialize output with bias
// load input
// load weight
for (n = 0; n < N; n++) {
  for (fil = 0; fil < F; fil++) {
    for (row = 0; row < R; row++) {
      for (col = 0; col < C; col++) {
        for (channel = 0; channel < CH; channel++) {      # pipeline
          for (hh = 0; hh < WEI_H; hh++) {          # unroll
            for (ww = 0; ww < WEI_W; ww++) {    # unroll
              output[n][fil][row][col] += weight[fil][channel][hh][ww]×input[n][channel][row][col];
} } } } } } }
// store output
```

Code Block 1. Primitie convolution layer implementation

However, on-chip BRAM resources are typically much more limited compared to the computation needs of convolution tasks on FPGAs. For larger and deeper networks like ResNet18, it is not possible to simply load all data onto BRAM in one instance. Therefore, tiling is a common method to address the above problem and optimize convolution layers. Traditional tiling proposed in [2] and [19] takes advantage of data locality during convolution. It partitions input and weight data so that only a portion is read from DRAM at an instance. Intermediate outputs are calculated based on the existing on-chip data portion and cached on BRAM. With tiling, on-chip data can be reused xduring calculation, which avoids redundant memory accesses. The process is repeated with the other partitions, and results will be written back to DRAM only when the intermediate results are all completed. In addition, local memory promotion is also applied, which rearranges the writing sequence to the output memory. The pseudo code of traditional tiling from [2] is shown in code 2.

```
for (n = 0; n < N; n += Tn) {
  for (f = 0; f < M; f += Tf) {
    for (row = 0; row < R_out; row += Tr) {
      for (col = 0; col < C_out; col += Tc) {
// load intermediate output
        for (ch = 0; ch < FIL_total; ch += Tch) {
// load tile input
// load tile weight
                                                          On-Chip tile computation
          for(hh = 0; hh < H_wei; hh++) {
            for(ww = 0; ww < W_wei; ww++) {
              for (tn = n; tn < min(n + Tn, N); tn++) {
                for (tf = f; tf < min(f + Tf, M); tf++) {
                  for (tr = row; tr < min(row + Tr, R_out); tr++) {    # pipeline
                    for (tc = col; tc < min(col + Tc, C_out); tc++) {    # unroll
                      for(tch = 0; tch < min(ch + Tch, FIL_total); tch++) {      # unroll
                        tile_output[tn][tf][tr][tc] += tile_weight[tf][tch][hh][ww] ×
                                                       tile_input[tn][tch][stride×tr + hh][stride×tc + ww];
          } } } } } } }
        }
// store tile output
} } } }
```

Code Block 2. Tiled convolution layer implementation

The tiling method can further be adjusted to reuse data that are previously read, which leads to our input reuse algorithm. For the above tiling method, intermediate tiling output are cached on BRAM and written to off-chip memory only when an output element is fully calculated. However, duplicate reads of inputs and weights from DRAM still exist, which can be further reduced through our input reuse algorithm. The input reduce algorithm rearranges the intermediate convolution calculations so that input data are reused to the fullest before a new read is initiated. Pseudo code is shown in code 3. Detailed steps are provided below along with table 3 for better interpretation. Table 4 and 5 will then show the results of the input reuse algorithm compared to

the normal tiling method. The timing performance is done by running 100 iterations of the convolution operation on different implementations.

(1) Suppose the convolution weights have FIL_{total} channels. The computing engine reads $FIL \times H_{wei} \times W_{wei}$ elements from the first filter map's first FIL number of channels of both input and weight buffers. After performing convolution multiplication, the result is the first intermediate result of the first data element in the first filter map's first channel of the output buffer, i.e. $output[1][1][1][1]$. Then, instead of reading from both input and weight buffers, the inputs are cached in the computing engine while only $FIL \times H_{wei} \times W_{wei}$ elements of the second filter map's first FIL number of channels are read from the weight buffer to the computing engine. The result of the second convolution operation is the first portion of the final sum of $output[1][2][1][1]$. The above process repeats until we reach the last filter map's first FIL number of channels of the weight buffer, which provides intermediate results for all channels of the first filter map in the output buffer.

(2) Now, we update the input data in the computing engine for the first time with $FIL \times H_{wei} \times W_{wei}$ elements from the first filter map's second FIL number of channels of the input buffer. The weight data is also renewed the with $FIL \times H_{wei} \times W_{wei}$ elements from the first filter map's second FIL number of channels of the weight buffer. Performing convolution results in the second portion of $output[1][1][1][1]$ and is summed with the previously calculated intermediate value. The input is then cached again while we update the weight data just like in ***(1)***. The whole process above is performed repetitively until the last filter map's last FIL number of channels of the output buffer are calculated. Now, we have finished the convolution calculations for all $output[1][1][1][1]$ to $output[1][FIL_{total}][1][1]$.

(3) Next, the computing engine moves the convolution window for the first time, where the second $FIL \times H_{wei} \times W_{wei}$ elements from the first filter map's first FIL number of channels of the input buffer are read. The computing engine reads the same weight data as in the first iteration, which is $FIL \times H_{wei} \times W_{wei}$ elements from the first filter map's first FIL number of channels of the weight buffer. The convolution result contributes to the second data element of the first filter map's first channel of the output buffer, i.e. $output[1][1][1][2]$. We then repeat all the process of ***(2)*** until all $output[1][1][1][2]$ to $output[1][FIL_{total}][1][2]$ are calculated.

(4) Repeat all the above steps until the convolution task is finished.

```
for (n = 0; n < N; n++) {
  for (fil = 0; fil < F; fil += FIL) {
    for (row = 0; row < R; row += stride) {
      for (col = 0; col < C; col += stride) {

        for (i = 0; i < FIL; i++) {# pipeline
          for (j = 0; j < H_wei; j++) {
            for (k = 0; k < W_wei; k++) {
// read input
// initialize tile output with bias
        } } }

        for (w_n = 0; w_n < w_map; w_n++) {      # pipeline
// burst read tile weight
          for (i = 0; i < FIL; i++) {        # unroll
            for (j = 0; j < H_wei; j++) {    # unroll
              for (k = 0; k < W_wei; k++) {# unroll
                tile_output[i][j][k] = tile_weight[i][j][k] ×tile_input[i][j][k];
          } } }
// sum tree
          output[n][f][row][col] += sum_tree_result;
        }

} } } }
// store output
```

Code Block 3. Input reuse convolution layer implementation

Table 3. Input reuse algorithm walkthrough process

$\boldsymbol{Input}\ (\boldsymbol{N}, \boldsymbol{FIL_{total}}, \boldsymbol{R_{in}}, \boldsymbol{C_{in}})$	$Weight\ (M, FIL_{total}, H_{wei}, W_{wei})$	$Output\ (N, M, R_{out}, C_{out})$
$[\boldsymbol{1}][\boldsymbol{1}\sim\boldsymbol{FIL}]$ $[\boldsymbol{1}\sim\boldsymbol{H_{wei}}][\boldsymbol{1}\sim\boldsymbol{W_{wei}}]$	$[1][1\sim FIL][1\sim H_{wei}][1\sim W_{wei}]$	[1][1][1][1]
	$[2][1\sim FIL][1\sim H_{wei}][1\sim W_{wei}]$	[1][2][1][1]
	$[3][1\sim FIL][1\sim H_{wei}][1\sim W_{wei}]$	[1][3][1][1]
	……	……
	$[M][1\sim FIL][1\sim H_{wei}][1\sim W_{wei}]$	$[1][M][1][1]$
$[\boldsymbol{1}][\boldsymbol{FIL}\sim\boldsymbol{2}\times\boldsymbol{FIL}]$ $[\boldsymbol{1}\sim\boldsymbol{H_{wei}}][\boldsymbol{1}\sim\boldsymbol{W_{wei}}]$	$[1][FIL\sim 2\times FIL][1\sim H_{wei}][1\sim W_{wei}]$	[1][1][1][1]
	$[2][FIL\sim 2\times FIL][1\sim H_{wei}][1\sim W_{wei}]$	[1][2][1][1]
	$[3][FIL\sim 2\times FIL][1\sim H_{wei}][1\sim W_{wei}]$	[1][3][1][1]
	……	……
	$[M][FIL\sim 2\times FIL][1\sim H_{wei}][1\sim W_{wei}]$	$[1][M][1][1]$
……	……	……
$[\boldsymbol{1}][(\boldsymbol{FIL_{total}}-\boldsymbol{1})\times\boldsymbol{FIL}\sim\boldsymbol{FIL_{total}}\times\boldsymbol{FIL}]$ $[\boldsymbol{1}\sim\boldsymbol{H_{wei}}][\boldsymbol{1}\sim\boldsymbol{W_{wei}}]$	$[1][(FIL_{total}-1)\times FIL\sim FIL_{total}\times FIL][1\sim H_{wei}][1\sim W_{wei}]$	[1][1][1][1]
	$[2][(FIL_{total}-1)\times FIL\sim FIL_{total}\times FIL][1\sim H_{wei}][1\sim W_{wei}]$	[1][2][1][1]
	$[3][(FIL_{total}-1)\times FIL\sim FIL_{total}\times FIL][1\sim H_{wei}][1\sim W_{wei}]$	[1][3][1][1]
	……	……
	$[M][(FIL_{total}-1)\times FIL\sim FIL_{total}\times FIL][1\sim H_{wei}][1\sim W_{wei}]$	$[1][M][1][1]$

$[\mathbf{1}][\mathbf{1}\sim\boldsymbol{FIL}]$ $[\mathbf{1}\sim\boldsymbol{H_{wei}}][(\mathbf{1}+\boldsymbol{stride})\sim(\boldsymbol{W_{wei}}+\boldsymbol{stride})]$	$[1][1\sim FIL][1\sim H_{wei}][1\sim W_{wei}]$	$[1][1][1][2]$
	$[2][1\sim FIL][1\sim H_{wei}][1\sim W_{wei}]$	$[1][2][1][2]$
	$[3][1\sim FIL][1\sim H_{wei}][1\sim W_{wei}]$	$[1][3][1][2]$
	……	……
	$[M][1\sim FIL][1\sim H_{wei}][1\sim W_{wei}]$	$[1][M][1][2]$
$[\mathbf{1}][\boldsymbol{FIL}\sim\mathbf{2}\times\boldsymbol{FIL}]$ $[\mathbf{1}\sim\boldsymbol{H_{wei}}][(\mathbf{1}+\boldsymbol{stride})\sim(\boldsymbol{W_{wei}}+\boldsymbol{stride})]$	$[1][FIL\sim 2\times FIL][1\sim H_{wei}][1\sim W_{wei}]$	$[1][1][1][2]$
	$[2][FIL\sim 2\times FIL][1\sim H_{wei}][1\sim W_{wei}]$	$[1][2][1][2]$
	$[3][FIL\sim 2\times FIL][1\sim H_{wei}][1\sim W_{wei}]$	$[1][3][1][2]$
	……	……
	$[M][FIL\sim 2\times FIL][1\sim H_{wei}][1\sim W_{wei}]$	$[1][M][1][2]$
……	……	……
$[\mathbf{1}][(\boldsymbol{FIL_{total}}-\mathbf{1})\times\boldsymbol{FIL}\sim\boldsymbol{FIL_{total}}\times\boldsymbol{FIL}]$ $[\mathbf{1}\sim\boldsymbol{H_{wei}}][(\mathbf{1}+\boldsymbol{stride})\sim(\boldsymbol{W_{wei}}+\boldsymbol{stride})]$	$[1][(FIL_{total}-1)\times FIL\sim FIL_{total}\times FIL][1\sim H_{wei}][1\sim W_{wei}]$	$[1][1][1][2]$
	$[2][(FIL_{total}-1)\times FIL\sim FIL_{total}\times FIL][1\sim H_{wei}][1\sim W_{wei}]$	$[1][2][1][2]$
	$[3][(FIL_{total}-1)\times FIL\sim FIL_{total}\times FIL][1\sim H_{wei}][1\sim W_{wei}]$	$[1][3][1][2]$
	……	……
	$[M][(FIL_{total}-1)\times FIL\sim FIL_{total}\times FIL][1\sim H_{wei}][1\sim W_{wei}]$	$[1][M][1][2]$
……	……	……
$[N][\mathbf{1}\sim\boldsymbol{FIL}]$ $[(\boldsymbol{R_{in}}-\boldsymbol{H_{wei}}+\mathbf{1})\sim\boldsymbol{R_{in}}][(\boldsymbol{C_{in}}-\boldsymbol{W_{wei}}+\mathbf{1})\sim\boldsymbol{C_{in}}]$	$[1][1\sim FIL][1\sim H_{wei}][1\sim W_{wei}]$	$[N][1][\boldsymbol{R}_{out}][C_{out}]$
	$[2][1\sim FIL][1\sim H_{wei}][1\sim W_{wei}]$	$[N][2][\boldsymbol{R}_{out}][C_{out}]$
	$[3][1\sim FIL][1\sim H_{wei}][1\sim W_{wei}]$	$[N][3][\boldsymbol{R}_{out}][C_{out}]$
	……	……
	$[M][1\sim FIL][1\sim H_{wei}][1\sim W_{wei}]$	$[N][M][\boldsymbol{R}_{out}][C_{out}]$
$[N][\boldsymbol{FIL}\sim\mathbf{2}\times\boldsymbol{FIL}]$ $[(\boldsymbol{R_{in}}-\boldsymbol{H_{wei}}+\mathbf{1})\sim\boldsymbol{R_{in}}][(\boldsymbol{C_{in}}-\boldsymbol{W_{wei}}+\mathbf{1})\sim\boldsymbol{C_{in}}]$	$[1][FIL\sim 2\times FIL][1\sim H_{wei}][1\sim W_{wei}]$	$[N][1][\boldsymbol{R}_{out}][C_{out}]$
	$[2][FIL\sim 2\times FIL][1\sim H_{wei}][1\sim W_{wei}]$	$[N][2][\boldsymbol{R}_{out}][C_{out}]$
	$[3][FIL\sim 2\times FIL][1\sim H_{wei}][1\sim W_{wei}]$	$[N][3][\boldsymbol{R}_{out}][C_{out}]$
	……	……
	$[M][FIL\sim 2\times FIL][1\sim H_{wei}][1\sim W_{wei}]$	$[N][M][\boldsymbol{R}_{out}][C_{out}]$
……	……	……
$[N][(\boldsymbol{FIL_{total}}-\mathbf{1})\times\boldsymbol{FIL}\sim\boldsymbol{FIL_{total}}\times\boldsymbol{FIL}]$ $[(\boldsymbol{R_{in}}-\boldsymbol{H_{wei}}+\mathbf{1})\sim\boldsymbol{R_{in}}][(\boldsymbol{C_{in}}-\boldsymbol{W_{wei}}+\mathbf{1})\sim\boldsymbol{C_{in}}]$	$[1][(FIL_{total}-1)\times FIL\sim FIL_{total}\times FIL][1\sim H_{wei}][1\sim W_{wei}]$	$[N][1][\boldsymbol{R}_{out}][C_{out}]$
	$[2][(FIL_{total}-1)\times FIL\sim FIL_{total}\times FIL][1\sim H_{wei}][1\sim W_{wei}]$	$[N][2][\boldsymbol{R}_{out}][C_{out}]$
	$[3][(FIL_{total}-1)\times FIL\sim FIL_{total}\times FIL][1\sim H_{wei}][1\sim W_{wei}]$	$[N][3][\boldsymbol{R}_{out}][C_{out}]$
	……	……
	$[M][(FIL_{total}-1)\times FIL\sim FIL_{total}\times FIL][1\sim H_{wei}][1\sim W_{wei}]$	$[N][M][\boldsymbol{R}_{out}][C_{out}]$

Table 4. Input reuse algorithm performance (100 iterations)

	With Input Reuse (FIL=8)	Without Input Reuse	Speedup
Time	5.983 s	77.15 s	$12.89\times$

Table 5. Input reuse algorithm resource utilization (FIL=8)

Resource	Available	With Input Reuse	Without Input Reuse
BRAM	4320	264 (6%)	193 (4%)
DSP	2280	405 (17%)	134 (5%)
FIFO	788160	67536 (8%)	50371 (6%)
LUT	394080	41726 (10%)	42852 (10%)
URAM	320	0 (0%)	0 (0%)

2. Computation engine of input reuse algorithm

The input reuse algorithm rearranges the whole convolution process to reduce input buffer access counts. The computation engine of the input reuse algorithm is responsible for the calculation intensive parts, mainly multiplication between input and weight data and the sum tree for the final result.

$FIL \times H_{wei} \times W_{wei}$ number of multipliers are implemented, allowing full parallelism of all $FIL \times H_{wei} \times W_{wei}$ input and weight element multiplications once read. In addition, arrays $tile\_output$, $tile\_weight$, and $tile\_input$ in code 3 are all partitioned with complete method, hence working similarly as registers, to allow the computation engine to access all data in just one clock cycle. Testing results of different parallel computing units are shown in table 6 and 7. The timing performance is done by running 100 iterations of the convolution operation with different implementations.

Table 6. Computation engine performance (100 iterations)

	$FIL = 2$	$FIL = 4$	$FIL = 8$
Time	8.930 s	6.702 s	5.983 s

Table 7. Computation engine resource utilization

Resource	Available	$FIL = 2$	$FIL = 4$	$FIL = 8$
BRAM	4320	210 (4%)	228 (5%)	264 (6%)
DSP	2280	135 (5%)	225 (9%)	405 (17%)
FIFO	788160	35716 (4%)	46325 (5%)	67536 (8%)
LUT	394080	25721 (6%)	31057 (7%)	41726 (10%)
URAM	320	0 (0%)	0 (0%)	0 (0%)

After $tile\_output$ is fully calculated, all $FIL \times H_{wei} \times W_{wei}$ elements in this array is added to form the final partial output. A sum tree with depth $\lceil \ln(FIL \times H_{wei} \times W_{wei} + 1) \rceil$ is implemented to finish this task as fast as possible. Figure 7 illustrates an example of the hardware design in the convolution layer. Note that memory access will not be a bottleneck as $tile\_output$ is also partitioned completely.

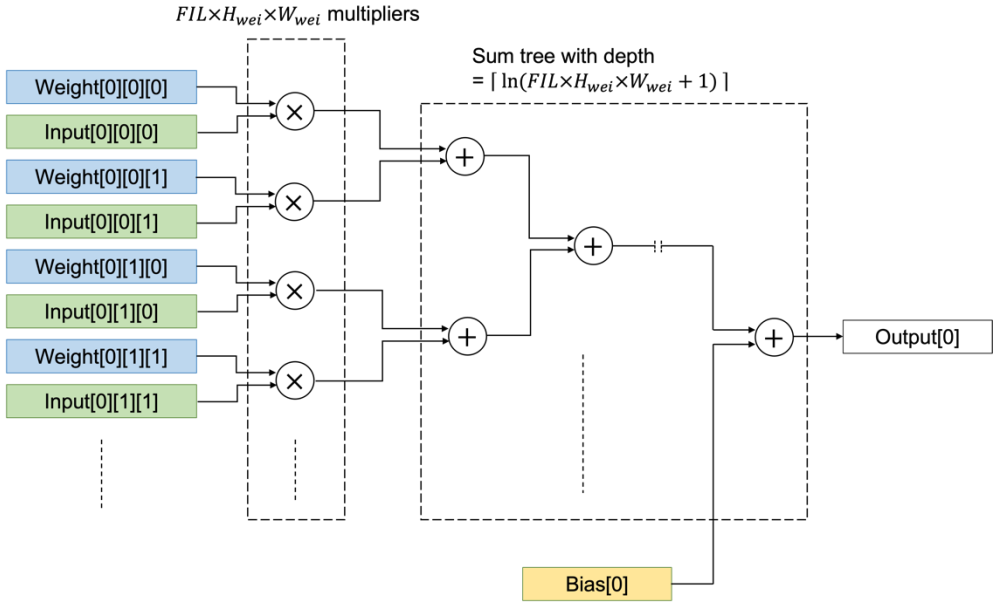

Figure 7. Conv layer hardware mapping

3. Resource sharing within the computation engine:

   According to [2] and [17], adders and multipliers mainly consume digital signal processing (DSP) and look-up table (LUT) resources. Since all layers in ResNet18 mainly utilize adders and multipliers, the consumption of DSP and LUT resources can be estimated using the results in [4]. As float-32 data type is used in the current implementation, an adder takes up 2 DSPs and 214 LUTs, while a multiplier takes up 3 DSPs and 135 LUTs.

   The adders needed in the computation engine is:

$$N_{adder_{conv}} = 2^{\lceil \ln(FIL \times H_{wei} \times W_{wei} + 1) \rceil - 1}$$

   The multipliers needed in the computation engine is :

$$N_{mul_{conv}} = FIL \times H_{wei} \times W_{wei}$$

   Therefore, the estimated DSP and LUT resources consumed by the convolution layer is

$$N_{DSP} = N_{adder_{conv}} \times 2 + N_{mul_{conv}} \times 3$$

$$N_{LUT} = N_{adder_{conv}} \times 214 + N_{mul_{conv}} \times 135$$

Part 5. Batch Normalization Layer Optimization

Batch normalization (BN) can greatly reduce training steps in neural networks, and thus are included in many machine learning modules since its introduction. According to [20], BN "normalizes" batches of data through the following equation:

$$\hat{x} = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}}$$

with x being the input element, $E[x]$ being the running mean, $Var[x]$ being the running variance, and ϵ being an epson offset. $E[x]$ and $Var[x]$ are extracted during the training process with the following two equations:

$$E[x] = momentum \times E[x] + (1 - momentum) \times E[x]_{original}$$

$$Var[x] = momentum \times Var[x] + (1 - momentum) \times Var[x]_{original}.$$

The momentum in our project is set to 0.9, and the details of these calculations can also be found in [20]. Furthermore, batch normalization adds a gamma and a beta parameter to $\hat{x}$, and the final calculation is shown below:

$$\widehat{out} = \widehat{gamma} \times \hat{x} + \widehat{beta}.$$

For all BN layers in our accelerator, arrays of $N \times FIL \times H \times W$ shape are read as input. Normalization are performed on the second dimension, i.e. FIL dimension, instead of the N dimension. All four parameter buffers, E, Var, $\widehat{gamma}$, and $\widehat{beta}$ respectively, are concatenated into one BN weight bin file during the preprocess training. The BN weight bin file is stored on off-chip DRAM in consecutive memory space, each with an offset of FIL as normalization are performed on the FIL dimension.

When the accelerator captures the control signal to execute a BN layer task, it initiates the data scatter modules to read the weights and inputs from their respective addresses in DRAM. The BN weight buffer is block partitioned to 4 consecutive equal arrays, corresponding to $\widehat{gamma}$, $\widehat{beta}$, E, and Var sub-buffers. As a result, only one clock cycle is required to load a single data point from all four buffers, thus then kickstarts the BN calculations. The gamma, beta, running mean, and running variance buffers are all then partitioned by cyclic method with a factor of 4. In this case, every sub-buffer contains 4 independent r/w ports, allowing 4 $\widehat{out}$ data to be calculated in one operation with the equations above. Adding in pipelining, the BN accelerator subpart is capable of outputting 4 independent $\widehat{out}$ data, reaching an initiation interval of 1 regardless of the latency. A simple overview of the pipelined timing is shown in figure 8 below.

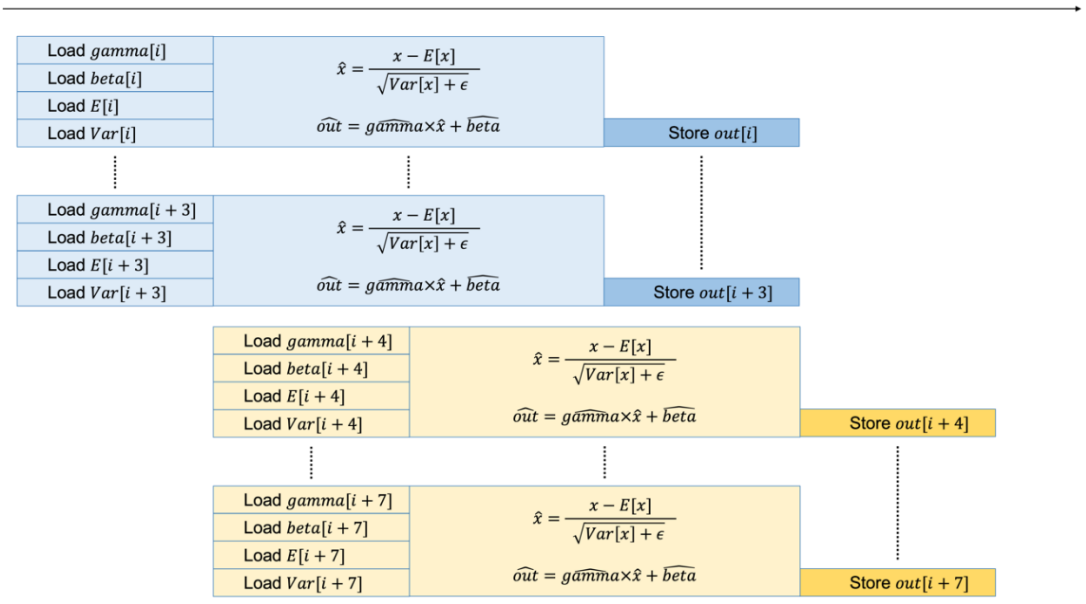

Figure 8. Batch normalization timing overview

Part 6. ReLU Layer Optimization

ResNet18 uses ReLU functions as the activation layer. ReLU function is a simple non-linear function that activates only positive neurons. It makes computation far more efficient during the training process as only certain neurons are contributive. The equation of ReLU function is provided below.

$$f(x) = \begin{cases} x, & x > 0 \\ 0, & otherwise \end{cases}$$

ReLU layer is pipelined to efficiently address the assigned task. Figure 9 shows the timing of ReLU layer.

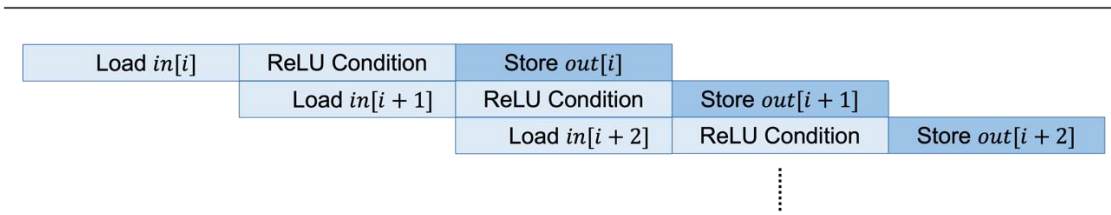

Figure 9. ReLU layer timing overview

Part 7. Residual Layer Optimization

The residual layer module is specially designed for the $y = F(x) + x$ operation mentioned in section **2.1.1**. According to the construction of ResNet18, residual operation is performed once in every residual block, summing up a previously input x with the calculated $F(x)$. It reads from a pre-stored residual buffer and the input buffer, perform summation, and write the result back to the respective output address on DRAM. The data of the residual buffer is stored in previous relu layers which is controlled by the CPU. The optimization is similar to batch normalization layer, as we partitioned both the input buffer and the residual buffer in cyclic method. Therefore eight parallel adders can perform calculation task simultaneously in one clock cycle. The timing overview of the residual layer is shown in figure 10.

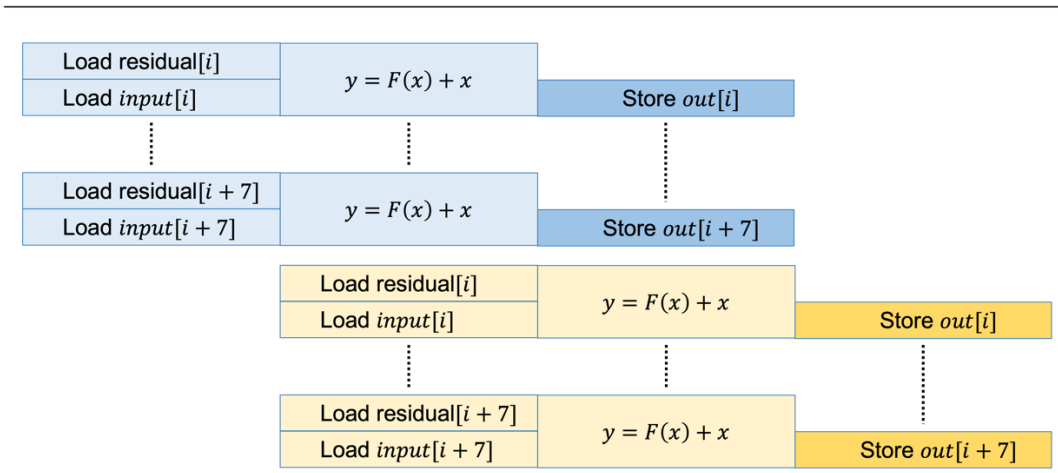

Figure 10. ReLU layer timing overview

Part 8. OpenCL and CPU

To deploy and run an FPGA application, a typical way is to control the FPGA device with a host program. This type of computation is refered to as heterogeneous computing, where tasks are distributed among two or more computing devices based on their respective strengths and weaknesses. In our ResNet18 application, the host program runs on a CPU or a microcontroller and is responsible for data pre-processing, post-processing, and control signals. After device initialization, the CPU deploys the xcl bin file to be programmed on the FGPA device. It then immdediately allocates device global memory and transfers all required data from host memory to FPGA DRAM. Once data transfer is finished, the CPU will send control signals to the FPGA to start the ResNet18 inferencing. The whole procress is supervised under the CPU as it will continue to monitor the status until the end. When the last layer is computed, the results are transferred back to the designated host memory from the FPGA. As mentioned in **part 1**, our CPU will continue to on to finish the global average layer and the full connected layer. We then calculate the final scores and test the accuracy. A queue is setup to for the CPU to place commands and be captured by the FPGA. This avoids bottlenecking the host process while the device is performing computation intensive tasks. A detailed host and device relationship is provided in figure 11.

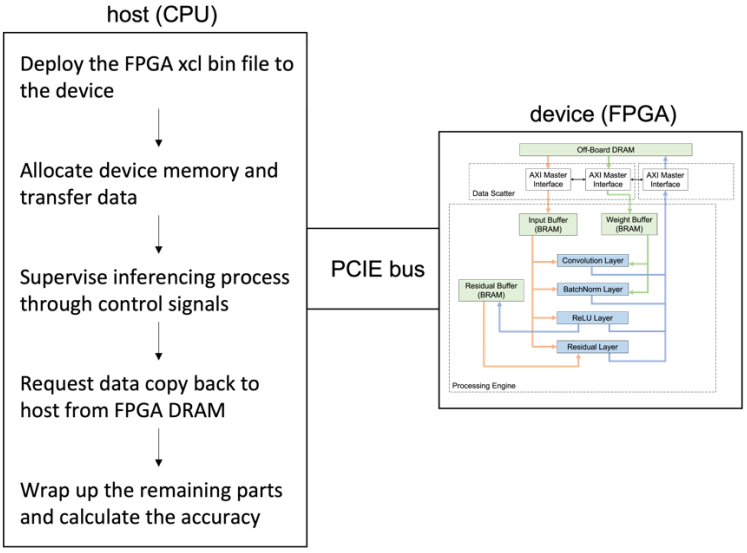

Figure 11. Relationship between CPU and FPGA

Table 8. CPU and FPGA timing

	Bin file and data transfer	ResNet18 Calculation
Time	0.0618 s	1.0592 s

Part 9. Deploying on AWS FPGA

In this thesis, we target AWS FPGA instances as our main device to run our ResNet18 accelerator [21]. Unlike running on local host and device, AWS FPGA requires bitstreams to be transformed into specific aws xclbin files. Pre-built Xilinx FPGA Amazon-Machine-Images (AMI) can greatly enhance development speed as Vitis installations and FPGA communications are guaranteed. Both Vitis and Vitis HLS tools are provided in all FPGA AMIs. In our design workflow, we utilize Vitis HLS software and hardware emulations, and switch to Vitis to build the hardware and generate the bitstream. Furthermore, AWS provides a special tool to transform FPGA bitstream to Amazon-FPGA-Images (AFI). AFIs are specially designed to run on AWS FPGA instances. In addition, to save costs, cheaper AWS EC2

instances that are not actually connected to FPGAs are allocated for all emulations and bitstream generation. Only when all programming is done and ready for deployment will we swarm a true FPGA instance with the built AFI. Executables and awsxclbin files are transferred between difference virtual machines through a shared AWS S3 storage.

*In the following section, we divide our whole accelerator design into smaller tasks. Our main focus is to implement all optimizations and work from **Part 1** to **Part 9** through all tasks. Only brief descriptions are given while technical details will be referred to the above.*

Task 1

Aim: Build ResNet18 with Pytorch to for training and extracting parameters.

Expected Outcome: A ResNet18 model that can successfully classify Cifar-10 [22] image dataset.

Work Description: The Pytorch version of ResNet18 is built to enable correctness checks for the FPGA ResNet18 accelerator. In addition, the Pytorch version allows GPUs to train the model, and the trained parameters act as the input, weight, and bias data of the FPGA ResNet inferencing accelerator.

Task 2

Aim: Build a workable prototype with abstracted modules according to the above three steps. Data input will be a streamline input just like the traditional method; Convolutional operations will be done also by simply multiplying all data inputs and adding them together; resource sharing will not be a large consideration in this task.

Expected Outcome: A workable convolutional layer that can get input and deliver output correctly.

In this task, simple loops will be used in HLS to build a prototype for futher evaluation and improvement. No resource reuse is considered here, as this version mimics the primitive python version and therefore are fragile to large input data.

Work Description: A preliminary ResNet versions is built in Vitis HLS to meet this task goal. Minimum optimization is done in this version. Details and results are illustrated as ResNet_v1 in table 9 and table 10. The source code and the pre-built aws xcl bin file are both uploaded to the below github link: https://github.com/chengyih001/ResNet18_fpga_accelerator.

Table 9. Timing Performance

Clock	Target	ResNet_v1
ap_clk	10 ns	13.542 ns

Table 10. Resource Utilization

Resource	Available	ResNet_v1
BRAM	4320	2143 (50%)
DSP	2280	365 (16%)
FIFO	788160	54567 (7%)
LUT	394080	65060 (17%)
URAM	320	0 (%)

Task 3

Aim: Implement the prototyped FPGA accelerator in **2.1.2**.

Expected Outcome: The FPGA accelerator can successfully perform inferencing of ResNet18 model.

Work Description: In this task, a simple accelerator is constructed in order to compare it with other implementations for further improvement. Different from ResNet_v1, this version focuses on resource reuitilization as shown in the hardware design of figure 3. The FPGA accelerator is no longer a black box to the CPU as the CPU is required to monitor the whole process with control signals. Details and results are illustrated as ResNet_v2 in table 11 and table 12. The source code and the pre-built aws xcl bin file are both uploaded to the below github link: https://github.com/chengyih001/ResNet18_fpga_accelerator.

Table 11. Timing Performance

Clock	Target	ResNet_v2
ap_clk	5 ns	7.922 ns

Table 12. Resource Utilization

Resource	Available	ResNet_v2
BRAM	4320	2057 (48%)
DSP	2280	385 (17%)
FIFO	788160	117805 (15%)
LUT	394080	158409 (40%)
URAM	320	0 (0%)

Task 4

Aim: Implement the host side of the ResNet18 accelerator system.

Expected Outcome: A host file running on CPU that correctly interacts with the FPGA device.

Work Description: As mentioned in ***Part 8***, the FPGA accelerator requires a CPU to provide control signals that refer to the layer to be executed, the address of the weight data in device global memory etc. This task implements the host side file with OpenCL as described in ***Part 8***.

Task 5

Aim: Optimization of the data input part for ResNet_v2 will be done in this task. We mainly focus on implementing the double buffer algorithm of ***Part 3***.

Expected Outcome: Implement double buffering and burst AXI read and writes for the data scatter and data gather models.

Work Description: This task is mainly to optimize input and weight data access. A second ResNet version is built to fulfill this task. The main job that is accomplished now is the double buffering to reduce latency as in ***Part 3***. The source code and the pre-built aws xcl bin file are both uploaded to the below github link: https://github.com/chengyih001/ResNet18_fpga_accelerator.

Task 6

Aim: Generalize the convolution layer module so that the same computation resources can be repeatedly used for different layers. Also, enable it to scale up or scale down depending on layer input and output.

Expected Outcome: Abstract the convolution layer modules in **Task 3** and a lower resource utilization rate is expected.

Work Description: To reuse computation and BRAM resources in between layers, all layers have been constructed to adapt to different input parameters, i.e. $N \times FIL \times H \times W$. The assigned BRAMs to each layer component is set to be the largest amount needed for each layer in between the whole network. This way, BRAMs can be reutilized after the data it holds is no longer in need. Same goes as to the computation resources, as the multipliers and additions are set to be the maximum amount needed in the whole network.

Task 7

Aim: Optimization of the convolution layers for ResNet_v2 are done in this part. We mainly focus on implementing the input reuse algorithm of ***Part 4***.

Expected Outcome: A much more efficient convolutional layer operation that provides the same result as of in ResNet_v1.

Work Description: As stated in the main goals of this objective, most of the computation jobs are in ResNet18 lies in the convolution layers. Therefore, optimizations to convolution layers greatly effects the efficiency of the accelerator. General parallelism techniques such as loop unrolling and pipelining are utilized along with the input reuse algorithm that rearranges the summation sequence. In addition, the sum tree allows a $O(ln)$ time to finish off the convolution task. Details and results of this version is illustrated as ResNet_v2_1 in table 13 and table 14. The source code and the pre-built aws xcl bin file are both uploaded to the below github link: https://github.com/chengyih001/ResNet18_fpga_accelerator.

Table 13. Timing Performance

Clock	Target	ResNet_v2_1
ap_clk	10 ns	7.284 ns

Table 14. Resource Utilization

Resource	Available	ResNet_v2_1
BRAM	4320	1304 (30%)
DSP	2280	54 (2%)
FIFO	788160	5931 (1%)
LUT	394080	7109 (2%)
URAM	320	0 (0%)

Task 8

Aim: Implement optimizations for the batch normalization module of ResNet_v2. We mainly focus on implementing the optimization methods mentioned in ***Part 5***.

Expected Outcome: A faster batch normalization layer constructed as mentioned in ***Part 5***.

Work Description: Refer to ***Part 5*** for the detail optimizations of the batch normalization module.

Task 9

Aim: Implement optimizations for the ReLU module of ResNet_v2. We mainly focus on implementing the optimization methods mentioned in ***Part 6***.

Expected Outcome: A faster ReLU layer constructed as mentioned in ***Part 6***.

Work Description: Refer to ***Part 6*** for the detail optimizations of the ReLU module.

Task 10

Aim: Implement optimizations for the residual module of ResNet_v2. We mainly focus on implementing the optimization methods mentioned in ***Part 7***.

Expected Outcome: A faster ReLU layer constructed as mentioned in ***Part 7***.

Work Description: Refer to ***Part 7*** for the detail optimizations of the residual module.

2.2.2.1 Evaluation of Objective 2.2.1 and Objective 2.2.2

Expected Outcome: A fully optimized FPGA-based ResNet18 accelerator to deploy on FPGA devices.

Objective 1: Design an algorithm to optimize convolution layer computation on FPGA which can increase acceleration speed for 5 times compared to for loop implementation.

Objective 2: To deploy ResNet on Zynq FPGA boards that results in an overall 25-30 times acceleration speed for the total training process when compared to implementing on 4 core ARM processors.

Actual Outcome:
All optimization methods and work stated from ***Part 1*** to ***Part 9*** are implemented into the final ResNet18 accelerator. The accelerator has been successfully deployed and tested on AWS FPGA instances instead of on Zynq FPGA boards. The implementation of each module are dynamic, as every parameter can be adjusted during runtime by the host device as long as it meets the memory constraints. This allows the highest flexibility for the accelerator to be deployed under different circumstances or even other deep learning architectures.

The input reuse algorithm stated in ***Part 4*** showed a 12 times performance increase over traditional loop methods, which greatly exceeds our goal of a 5 time speedup. This is mainly due to the reduction in redundant memory access by rescheduling the multiplication operations in the convolution layer. Arrays are carefully partitioned to meet the required parallelism goals and to maximize computation resources. As shown in the results of table 4, we consider *Objective 1* as successful.

The FPGA accelerator is run on AWS FPGA instances rather the Zynq boards stated in objective 2. The main difference is that Zynq boards are SoC FPGAs that has an embedded CPU within the FPGA, while typical FPGA devices need an external CPU to communicate with it through the PCIE bus. However, this does not affect the final evaluation as the ResNet18 accelerator is still successfully deployed on an FPGA. The CPU model is not tested on ARM processors as we could not land one of them. The initial plan is to directly run the CPU version on Zynq board's ARM Cortex a53 4-core processor, but since Zynq boards are not available to us, we are unable to benchmark the CPU version on it. As a result, the ResNet18 accelerators are tested on the CPUs of our personal computer. Performance comparisons are provided in table 15. AMD Ryzen9 is a very high end laptop CPU in the market, listed with a 23160 score on cpubenchmark [23]. On the other hand, ARM Cortex a53 4-core CPU obtained only a 557 mark, significantly lower compared to AMD Ryzen9 as it is more common in embedded systems. Despite the strong competitions from the laptop CPUs, our FPGA ResNet18 accelerator still managed to out perform AMD Ryzen9 by around 5.4 times. Also, compared to results of AWS t2.micro virtual machine, a performance improvement of around 6 times can be observed. Given the significantly stronger computation powers of computer CPUs when compared to ARM Cortex a53, we therefore still consider *Objective 2* as successful.

Table 15. Accelerator performance on difference platforms ($N = 5$)

CPU	FPGA	Primitive Version	Final Version
AMD Ryzen9	-	5.684 sec.	5.672 sec.
AWS t2.micro	-	6.307 sec.	6.323 sec.
AWS f1.xlarge	XCVU9p	12.798 sec.	1.06 sec.

Designing from a hardware perspective posed a challenge for us, as same software emulations could result in different hardware results. We read many research papers and FPGA documentations to familiarize ourselves with different design choices and implementations. In the end, most of the challenges regarding implementations and coding were resolved and we were able to built the accelerator as planned.

The biggest obstacle, however, lies in the deployment of our design onto the FPGA device. A local VCU108 FPGA provided by HKUST reconfigurable system lab was utilized at first, but it quickly became unavailable as lab sessions were very limited due to the COVID-19 situation in Hong Kong. As a result, we chose to switch to AWS cloud FPGA in a relatively early stage of this thesis. The concept of FPGA on cloud is relatively new, and documentations are mostly primitive and incomplete. When facing synthesis and implementation errors, there are rarely any relative information online usually. Moreover, the tools provided by AWS and Xilinx in the pre-built AFI images are still buggy, and sometimes generate unseen errors before. We spend quite a portion of our time understanding and debugging the tools for it to work. Despite the challenges, we still managed to successfully deploy our ResNet18 accelerator on the AWS FPGAs.

2.2.3 Testing of Accelerator on Existing Image Datasets

Objective: Test the ResNet model on existing datasets and reach accuracies as existing research. (For ResNet18 on CIFAR-10, the accuracy is around 93-95%.)

This objective is for testing our implemented model on existing datasets. CIFAR-10 is the main image set that is used as our testing.

Task 1

Aim: Organize and split the dataset into training, validation, and testing with Python.

Expected Outcome: Perform data-preprocessing in order to ease the workload on FPGA and also to retrieve validation and testing data.

In this task, the dataset will be split and some data-preprocessing will be done on the dataset.

Work Description: The Pytorch version constructed in **2.2.2 Task 1** is the baseline model to verify correctness and train the parameters. An additional Python version that mimicks the logic of the ResNet model is constructed as a reference towards different ResNet HLS accelerator implementations. All implementations are shown in github: https://github.com/chengyih001/ResNet18_fpga_accelerator.

Task 2

Aim: Test the FPGA accelerator and the CPU version with the existing dataset.

Expected Outcome: The accuracy should be similar (+-3%) to existing experiments (For ResNet18 on CIFAR-10, the accuracy is around 93-95%.), and the FPGA accelerator should be significantly faster than the CPU version

Work Description: In this task, the whole project will be concluded and tested. The primitive version refers to the FPGA accelerator with no optimizations while the final version refers to the fully optimized accelerator. The PyTorch version is tested as the baseline model of correctness. Details of the correctness testing along with timing performance and resource utilization information are provided in tables 16 to 19.

Table 16. Clock Performance

Clock	Target	Primitive Version	Final Version
ap_clk	3.00 ns	3.163 ns	3.00 ns

Table 17. Accelerator performance on difference platforms ($N = 5$)

CPU	FPGA	Primitive Version	Final Version
AMD Ryzen9	-	5.684 sec.	5.672 sec.
AWS t2.micro	-	6.307 sec.	6.323 sec.
AWS f1.xlarge	XCVU9p	12.798 sec.	0.98 sec.

Table 18. Resource Utilization

Resource	Available	Primitive Version	Final Version
BRAM	4320	767 (17%)	835 (19%)
DSP	2280	67 (2%)	411 (18%)
FIFO	788160	34842 (4%)	79247 (10%)
LUT	394080	29870 (7%)	53659 (13%)
URAM	320	0 (0%)	0 (0%)

Table 19. CiFAR-10 testing results

CiFAR-10	PyTorch	Primitive Version	Final Version
100	90.00%	90.00%	90.00%
1000	89.1%	89.00%	89.00%
5000	88.84%	88.82%	88.82%
10000	88.52%	88.48%	88.48%

2.2.3.1 Evaluation of Objective 2.2.3

Expected Outcome: The correctness of the ResNet18 accelerator is tested and verified.

Objective 3: Test the ResNet model on existing datasets and reach accuracies as existing research. (For ResNet18 on CIFAR-10, the accuracy is around 93-95%.)

Actual Outcome:

Testing are mainly done on CiFAR-10 image dataset [2]. As the results shown in table19, the accuracy of our FPGA accelerator can reach around 88.5%. The 3% to 5% lower in accuracy compared to state-of-the-art ResNet research is expected mainly due to two reasons. First and for most, our pre-trained parameters are not able to reach such high accuracies, and therefore it is nearly impossible for the inferencing on FPGA to do so. This can be verified by the PyTorch accuracy test since the trained PyTorch model on GPU is also only able to reach around 90% correctness. Second is the data accuracy lost when extracting trained parameters from Python to Vitis HLS. The extracted parameters were saved as double type data in bin files to be read into the HLS program. However, due to the limited BRAM resources on FPGAs, we implemented all data with floating points. The narrowing data conversion resulted in a slight difference and may also reduce accuracy during the inferencing of ResNet18 on FPGAs. Overall, the accuracy of the FPGA-based ResNete18 accelerator is still high enough to be considered as a pass in correctness testing. Therefore, as of *Objective 3*, we consider it as acceptable rather than successful.

2.3 Main Objective Evaluation & Discussion

The main objective of this thesis is to implement a fast ResNet18 accelerator on FPGA devices. The ability to run deep learning architectures such as ResNet makes FPGA an ideal platform for testing different high performance computing implementation algorithms.

In this section, the first thing we analyzed was the ResNet18 architecture in **2.1.1**. We briefly discussed the functionalities of ResNet architectures and the constructions based on [1]. **2.1.2.** and **2.1.3.** mainly discuss the the accelerator design from a high level perspective. Hardware system designs and the execution workflow are provided while the detail implementations are postponed to **2.2**.

The whole **2.2** subsection describes the exact optimization methods for each components. A huge amount of time was put in towards miminizing data transfer latency and increasing computing efficiency. Standard techniques such as loop pipelining, unrolling, and array partitioning are carefully considered to maximize resource to computation efficiency. In additional special algorithms are implemented for several components. Data scatter and gather modules are constructed through double buffering method to overlap memory read and write latency between DRAM and BRAM. We implemented the input reuse algorithm, a new algorithm developed from tiling, on the convolution layer components. As shown in table 4, the input reuse algorithm can increase performance by around 12 to 13 times. In addition, the deployment process of the accelerator onto FPGA devices are also discussed. Due to the COVID-19 pandemic, we chose AWS FPGA as the main platform. FPGA on cloud allowed us to test and design of our accelerator without being limited by location or time. Enormous amount of effort was dedicated to familiarize with this deployment methodology as it is currently still in a rather primitive state.

Last, we evaluated the final version of our ResNet18 FPGA-based accelerator with CiFAR-10 image dataset. Given 5 images, our FPGA accelerator can correctly output the results within 1 second, while it took over 5 to 6 seconds for even high end laptop processors. Resource uitilization are kept at a low level and special focus is given to minimize BRAM usage as it is much more expensive. Compared to earlier ResNet implementations showed in **2.2.2**, BRAM allocation is cut down by over half to an acceptable sub-20 percent. Correctness tests are presented in the end of section **2.2.3** to guarantee the workability of our accelerator. Results are compared to the PyTorch implementation that runs on standard GPUs. The slightly lower accuracies for our FPGA accelerator are expected due to data type narrowing conversions. Well tuned ResNet architectures on CiFAR-10 datasets have reached 93-95% accuracy, which still exists a gap when compared to our FPGA inferencing accelerator. However, the overall accuracy of our design is still around 88.5%, which is an acceptable result to verify the correctness of our implementation.

Overall, we constructed an ResNet18 FPGA accelerator that can calculate 5 CiFAR-10 images under 1 second. Resource allocations have been kept low while achieving such a performance. The accelerator has been successfully deployed on AWS FPGA cloud instances and can reach around 89% classification accuracy. Given the above results and evaluationss, we consider our deployment as successful and achieves the main objectives of this thesis.

SECTION 3—CONCLUSION

This thesis aims to design a fast ResNet18 FPGA-based inferencing accelerator. Various optimization methods are implemented to minimize resource allocation and maximize the parallel computing abilities of FPGAs. Meanwhile, we constructed a new input reuse algorithm to reduce redundant memory read operations and increase parallelism in convolution layers. Experimental results show an overall 10-12X speedup in convolution operations when comparing to conventional for loop implementations. With additional optimizations to memory access and computation tasks, we can achieve a 5-6X overall performance improvement even when compared to very high end processors. Resource utilizations, especially BRAM memory, are carefully monitored and kept low to save area for potential scaleups of the deep learning network. AWS cloud FPGAs are selected as the main deployment of this thesis as it provides ready to go computing powers anytime anywhere. All implementations are constructed to provide largest flexibility for the host controls during

runtime as every parameter can be adjusted to suit different needs. With the integration of FPGAs into CPUs, it is also possible to embed our accelerator into very large scale infrastructures in view of offloading CPU workload.

Improvement can still be made to this accelerator to further enhance its computation powers. With only a slight sacrifice in accuracy, float32 point data types can be narrowed down to 16 bits to boost performance as it is a major cut down in calculations. In addition, pre-trained weights can be fine-tuned to a further extend in order to increase accuracy by another 2-3%. Scalability of this accelerator remains a question, and it will be the main focus in the next phase of the design. It is also expected that future work will be addressed in testing other neural network architectures and to deploy these applications on distributed FPGA systems.

REFERENCES

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR), 2016*.

[2] C. Zhang, P. Li, G. Sun, and Y. Guan, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks, " in *FPGA '15: Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, February 2015.* pp.161-170

[3] G. Wei, Y. Hou, Z. Zhao, Q. Cui, G. Deng, and X. Tao, "Demo: FPGA-Cloud Architecture For CNN, " in *24th Asia-Pacific Conference on Communications (APCC), November 12-14 2018*.

[4] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning" in *arXiv: 1602.07261 [cs.CV],* August 23 2016.

[5] S. Cadambi, A. Majumdar, M. Becchi, S. Chakradhar, and H. P. Graf, "A Programmable Parallel Accelerator for Learning and Classification," in *19th International Conference of Parallel Architectures and Compilation Techniques (PACT), September 11-15 2010.*

[6] Z. Liu, Y. Dou, and J. Jiang, "Throughput-Optimized FPGA Accelerator for Deep Convolutional Neural Networks," in *ACM Transactions on Reconfigurable Technology and Systems, July 2017.* pp. 1-23

[7] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based Learning Applied to Document Recognition," in *Proceedings of the IEEE, November 1998.* pp. 2278-2324

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Communications of the ACM, June 2017.* pp. 84-90

[9] K. Simonyan, and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *ICLR 2015.*

[10] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convollutional Networks for Biomedical Image Segmentation," in *MICCAI, 2015.*

[11] K. Ovtcharov, O. Ruwase, J. Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating Deep Convolutional Neural Networks Using Specialized Hardware," in *Microsoft Research, February 22 2015.*

[12] Y. Ma, Y. Cao, S. Vrudhula, and J. S. Seo, "Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks," in *FPGA '17: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, February 2017.* pp. 45-54.

[13] U. Nath, and S. Kushagra, "Better Together: Resnet-50 accuracy with 13x fewer parameters and at 3x speed," in *arVix: 2006.05624 [cs.LG],* October 10 2020.

[14] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning Convolutional Neural Networks for Resource Efficient Inference," in *ICLR 2017.*

[15] A. Rahman, J. Lee, and K. Choi, "Efficient FPGA acceleration of Convolutional Neural Networks using logical-3D compute array," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), March 14-18 2016.*

[16] Y. Ma, M. Kim, Y. Cao. S. Vrudhula, and J. S. Seo, "End-to-end scalable FPGA accelerator for deep residual networks," in *IEEE International Symposium on Circuits and Systems (ISCAS), May 28-31 2017.*

[17] C. Chen, Z. L. Chai, J. Xia, "Design and Implementation of YOLOv2 Accelerator Based on Zynq7000 FPGA Heterogeneous Platform," in *Journal of Frontiers of Computer Science and Technology, 2019.*

[18] Xilinx and AMD, "Vitis High-Level Synthesis User Guide (UG1399)," in *https://docs.xilinx.com/r/en-US/ug1399-vitis-hls, 2021.*

[19] Q. V.Le, J. Ngiam, Z. Chen, D. Chia, P. Koh, A. Ng, "Tiled Convolutional Neural Networks," in *NEURIPS, 2010.*

[20] S. loffe, C. Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,” in *arXiv: 1502.03167 [cs.LG],* March 2 2015.

[21] Amazon Web Services (AWS), “Amazon EC2 F1 Instances: Enable faster FPGA accelerator development and deployment in the cloud,” in *https://aws.amazon.com/ec2/instance-types/f1/.*

[22] A. Krizhevsky, V., Nair, G. Hinton, “The CIFAR-10 dataset,” in *https://www.cs.toronto.edu/~kriz/cifar.html.*

[23] PassMark Software, “CPU Benchmarks,” in *https://www.cpubenchmark.net/*.

APPENDICES

Appendix A. Project Schedule

Table 20. FPGA-based accelerator schedule.

Objective Statements	Task	WK3 13 Sep	WK4 20 Sep	WK5 27 Sep	WK6 4 Oct	WK7 11 Oct	WK8 18 Oct	WK9 25 Oct	WK10 1 Nov	WK11 8 Nov	WK12 15 Nov	WK13 22 Nov	WK 14 29 Nov	WK15 6 Dec	WK16 13 Dec	WK17 20 Dec	WK18 27 Dec
Implementing and Optimizing ResNet architecture on FPGAs																	
	FPGA-based accelerator prototyping																
	CPU version accelerator implementation																
	Computation operation optimization																
	Memory operation optimization																
Accelerator Algorithm and Architecture Improvement																	
	Traditional convolutional algorithm implementation																
	Data input algorithm revision																
	Convolutional algorithm revision																
	Layer module generalization																
Testing of Accelerator on Existing Image Datasets																	
	Data pre-processing																
	Memory module revision																
	FPGA-based accelerator and CPU version testing																

Continued on the next page

Table 20. FPGA-Based Accelerator Schedule (Continued)

Objective Statements	Task	WK18 3 Jan	WK19 10 Jan	WK20 17 Jan	WK21 24 Jan	WK22 31 Jan	WK23 7 Feb	WK24 14 Feb	WK25 21 Feb	WK26 28 Feb	WK27 7 Mar	WK28 14 Mar	WK29 21 Mar	WK30 28 Mar
Implementing and Optimizing ResNet architecture on FPGAs														
	FPGA-based accelerator prototyping													
	CPU version accelerator implementation													
	Computation operation optimization													
	Memory operation optimization													
Accelerator Algorithm and Architecture Improvement														
	Traditional convolutional algorithm implementation													
	Data input algorithm revision													
	Convolutional algorithm revision													
	Layer module generalization													
Testing of Accelerator on Existing Image Datasets														
	Data pre-processing													
	Memory module revision													
	FPGA-based accelerator and CPU version testing													

Appendix B. Budget

Table 21. Expected budget

Items	Cost
Xilinx VCU108 Field Prgrammable Gate Array (FPGA)	HKD 0 (This FPGA is provided by HKUST. The market price of this FPGA is around HKD 58,000.)
AWS FPGA F1 Instances	USD 300. (This cost varies among different virtual machine choices.)
Vivado Design Suite VCU108 License	HKD 0 (This board license is provided by HKUST. The market price of this Design Suite Licence is around HKD 25,000.)
Personal Computer (ROG Strix Scar 17 G733)	HKD 23,500. Linux system Ubuntu LTS 18.04 is installed with no additional fee.
Total	HKD 23,500

*All programs and hardware excluding the personal computer and AWS cloud resources are available from HKUST at no cost.

Appendix C. Meeting Minutes

Meeting 1

Date: 08/09/2021

Time: 5:00 pm

Location: HKUST Campus Rm 2449

Attendees: Yih CHENG and Professor Wei ZHANG

Absent: None

Minutes taken by: Yih CHENG

- The main topic of the thesis is confirmed.
- Several papers was given for reading.
- Fixed float32 input data will be done first, then other may be followed up by dynamic float16 or float8.

Table 22. Action Items from the Previous Meeting

Action Item to be completed	By when	By whom	Status
Proposal report	Sep 15th	Yih CHENG	Done
Literature review	Sep 15th	Yih CHENG	Done
Finish most optimizations	Mar 22nd	Yih CHENG	Done

Table 23. Action Items for Next Meeting

Action Item to be completed	By when	By whom
Finalize proposal report	Sep 15th	Yih CHENG
Start building simple CNN module	Oct 4rd	Yih CHENG
Finish all optimizations	Apr 20th	Yih CHENG

Next Meeting: To be scheduled with professor