ADDIS ABABA UNIVERSITY

ADDIS ABABA INSTITUTE OF TECHNOLOGY

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

Acceleration of Convolutional Neural Network Training using Field Programmable Gate Arrays

---

By:

Guta Tesema Tufa

Advisor:

Dr. Fitsum Assamnew Andargie

A thesis submitted in partial fulfillment of the requirements for the degree of ***Master of Science*** in **Computer Engineering**

January, 2022

Addis Ababa, Ethiopia

ADDIS ABABA UNIVERSITY

ADDIS ABABA INSTITUTE OF TECHNOLOGY

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

Acceleration of Deep Neural Network Training using Field Programmable Gate Arrays

By: Guta Tesema Tufa

Dr. Bisrat Derebssa	
Dean, SECE, AAiT	Signature
Dr. Fitsum Assamnew	
Thesis Advisor	Signature
Dr. Eng. Getachew Alemu	
External Examiner	Signature
Dr. Bisrat Derebssa	
Internal Examiner	Signature

Declaration

I hereby announce that I am the sole author of this thesis and that I have not used any references other than those mentioned as reference in the bibliography. I also announce that I have not applied this manuscript in order to receive a degree at any other university.

Guta Tesema

Acknowledgment

Glory to GOD, the Almighty, for his blessings to allow me complete this thesis successfully.

I would like to express my sincere gratitude to my thesis supervisor Dr. Fitsum Assamnew, for his invaluable guidance throughout this thesis. I was deeply inspired by his dynamism, vision, sincerity and motivation. He taught me the procedures to conduct the research and to present the works of the research as clearly as possible. I want to thank him, too, for his friendship, his empathy and his wonderful sense of humor.

I am most thankful to my father, Tesema Tufa, and my mother, Ayantu Meskelu, for their devotion, prayers, support and sacrifices for my education. I am very thankful to my wife Dhangaa Fatansaa for her love, understanding, prayers and continuing support to complete this thesis work.

Finally, I must express my very profound gratitude to my brothers, sisters, and friends for providing me with unfailing support and continuous assistance throughout my years of study, through the research and writing process of this thesis. This success might not have been possible without them Thank you.

Abstract

Convolutional neural networks (CNN) training often necessitates a considerable amount of computational resources. In recent years, several studies have proposed CNN inference and training accelerators, which the FPGAs have previously demonstrated good performance and energy efficiency. To speed processing, the CNN requires additional computational resources such as memory bandwidth, a FPGA plantform resource usage, time, and power consumption. As well as training the CNN needs large datasets and computational power, and they are constrained by the requirement for improved hardware acceleration to support scalability beyond existing data and model sizes. In this study, we propose a procedure for energy efficient CNN training in collaboration with an FPGA-based accelerator. We employed optimizations such as quantization, which is a common model compression technique, to speed up the CNN training process. Additionally, a gradient accumulation buffer is used to ensure maximum operating efficiency while maintaining gradient descent of the learning algorithm.

Subsequently, to validate our design, we implemented the AlexNet and VGG16 models on an FPGA board and a laptop CPU and GPU. Consequently, our designs achieve 203.75 GOPS on Terasic DE1-SoC with the AlexNet model and 196.50 GOPS with the VGG16 model on Terasic DE-SoC. This, as far as we know, outperforms existing FPGA-based accelerators. Compared to the CPU and GPU, our design is 22.613X and 3.709X more energy efficient respectively.

Key-Word: FPGA, CNN, Kernel, Terasic DE-SoC, Energy efficiency, Quantization

Contents

List of Figures

List of Tables

List of Abbreviations

ALMs	Adaptive Logic Modules
ALUT	Adaptive look-up Table
AOC	Intel Offline Compiler
API	Application program interface
ASIC	Application Specific Integrated Circuit
BSP	Board Support Package
CNN	Convolutional Neural Network
CU	Compute Unit
CPU	Central Processing Unit
DMA	Direct Memory Access
DSP	Digital Signal Processor
FC	Fully-Connected
FFT	Fast Fourier Transform
FLOPS	Floating Point Operations Per Second
FP	Float-Point
FPGA	Field Programmable Gate Array
GCC	GNU Compiler Collection
GOPS	Giga oprations per second
GPU	Graphic Processing Units
GPGPU	General Processor Graphic Processing Units
HDL	Hardware Description Languages
HLS	High-level S ynthesis

HMC	Hybrid Memory Cube
HPC	High Performance Computing
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
K40	Kepler 40
LEs	Logic Elements
MAC	Multiply-accumulate
LRN	Local Response Normalization
OpenCL	Open Computing Language
ReLU	Rectified Linear Unit
RTL	Register transfer level
SDK	Software Development Kit
SDRAM	Synchronous Dynamic Random Access Memory
SIMD	Single Instruction Multiple Data
SPMD	Single Program Multiple Data
SoC	System on Chip
SRAM	Static Random Access Memory
TDP	Thermal Design Power

Dedicated to my parents Tesema Tufa and Ayantu Meskelu as well as to my wife Dhanga Fatansaa.

Chapter 1

Introduction

1.1 Background

In recent years, Deep neural network have shown their vital role in solving many real world problems. The DNN, notably the convolutional neural network, is at the root of this renaissance. The convolution neural network was shown to be a useful tool for a variety of functions, including image classification [1], image recognition [2], and object detection[3]. A CNN involves a massive number of computations that it could profit from acceleration using GPUs and FPGAs [4, 5]. CNN hardware implementations are constrained by a memory bottleneck that need numerous convolutions and fully-connected layers, which necessitate a considerable amount of communication for parallel processing[6].

A variety of accelerators, including graphics processing units (GPUs), Field Programmable Gate Arrays, and application specific integrated circuits were used to increase the efficiency of CNNs[7, 8, 9]. Among these accelerators, GPUs are the most commonly employed to enhance throughput, memory bandwidth [8], both in the training and inference processing of CNN, however, they use high power[10, 6, 1]. An alternatively, Field-programmable gate arrays (FPGAs) are a n option for neural network deployment since computing, logic, and memory resources may be merged into a single device. Based on FPGAs (field-programmable gate arrays), CNN accelerators provide significant benefits because of their reduced power consumption, high throughput, and design flexibility [11]. FPGAs also provide high parallelism and exploit the features of neural network processing [12]. However, CNN on FPGA has a number of challenges such as requirements of memory storage, external memory bandwidth, and computational resource limitations. The previous hardware accelerators for CNN have used different kernel for convolution and fully connected layers, which affect the FPGAs resource utilization [13, 5].

Intel's programmable solutions division has created a scalable convolutional neural

network reference architecture for deep learning systems based on the OpenCL programming language. The OpenCL-based design tool is used to effectively accomplish the required accelerator design. This allows us to reuse the current code for Graphics Processing Units (GPUs) in FPGAs using OpenCL-based high-level synthesis tools [14, 6]. Developers may program the FPGAs in high-level languages like as C / C++ using high-level synthesis (HLS), which speeds up the development process. HLS techniques, provide a developer with an extremely simple programming model as FPGA.[12].

1.2 Problem Statement

Many researches have been done on the deep neural network accelerator based FPGA. Unfortunately, most of these researches are specific to increase the CNN model optimization[15, 16, 17], space exploration[18, 19], computational workload of a CNN inference[20], memory accesses[21], algorithmic optimizations[22, 23, 24, 25], data path optimizations[26, 27] and HLS based design with OpenCL[6, 5, 8] and Vivado HLS [28, 29, 8, 30].

CNNs are mostly solved using methods based on matrix-multiplication, this somehow requires the movement of huge volumes of data between computing units and external memory[5]. To speed up processing, the CNN requires more computational resources. Nonetheless, when processing CNNs a memory bandwidth is often the bottleneck. Because of the high memory requirements of the fully connected layers (FC) layer sections as well as the execution might be limited. A large number of weights which are held by these layers are considered for the large number of memory reads. If any of these accesses are to external memory, for instance dynamic random access memory, throughput and energy power usage would be significantly impacted, because dynamic random access memory accesses have far more latency and energy consumption than the compute itself [13]. However, memory storage, external memory bandwidth, and computing resource limits provide a number of challenges for CNN on FPGA.

In addition to this problem, RTL implementation requires tedious design processing cost. However, HLS tool such as OpenCL reduce development cost [12].

This work is to make a simpler design of accelerator of CNN for image classification. Hence, the following research questions will be answered by this report.

- Can we use high-level synthesis tools to design a parameterized CNN accelerator?
- Using the Intel FPGA SDK OpenCL to accelerate CNN, can it outperform other platforms?

1.3 Objective

The main goal of this research is to develop an OpenCL-based accelerator for CNN training on FPGA.

1.3.1 Specific objective

- To built a naive CNN model on FPGA using OpenCL.
- To optimize using technique such as model quantization, design space exploration method (CUs and SIMDs).
- Evaluate the performance of our design's to select the best design.
- To compare our system's outcomes and performance with that of another platform.

1.4 Contribution

The following are the contributions of this work:

- To perform both convolution and FC layers, we use a single kernel, which helps to improve the efficiency of hardware accelerator resource consumption in FPGAs.
- Loop parallelization and Single Instruction Multiple Data (SIMD) have been applied.
- To get maximum throughput we use design space exploration method that leverages resource usage and throughput.
- We compare our design to the most recent implementations.

1.5 Scope

The aim of this research is to develop CNN accelerator on FPGAs system. The scope of study is limited to image classification using CNN and implementing on FPGAs platform.

1.6 Methodology

In this section explain the literature review, design flow of Intel FPGA SDK for OpenCL, dataset and results evaluation metric.

1.6.1 Literature Review

Since accelerating convolutional neural networks on FPGA are a popular research topic using the Intel OpenCL SDK, a thorough review of the literature will be undertaken to analyze existing research effort and get a thorough knowledge of future research prospects. We begin by examining the different approaches for overcoming these problems, such as precision quantization, OpenCL design for Intel FPGA SDK for OpenCL and architecture, to ensure that synthesized hardware performs optimally.

1.6.2 Design Flow of Intel FPGA SDK for OpenCL

The emulator kernels program (.cl) is generated by the Intel offline Compiler (AOC) at a first phase of the OpenCL design for the Intel FPGA SDK. The emulator, which can operate on both an x86 and an x64 host, can quickly check for syntax errors and functional correctness. AOC also creates an optimization report, while building the emulator, which contains information on memory transfers and the execution initiation interval. The kernel code is then fully built with AOC, allowing the OpenCL code to be synthesized immediately to the Verilog. The Intel FPGA SDK for OpenCL tools accelerates development by automatically handling the interaction between memory regions. Depending on the program, full compilation might take hours. Finally, the GNU Compiler Collection (GCC) compiles and runs the host software and the FPGA executable on the system. If the kernel's performance or resource use does not match the criteria, it must be verified and built again.

1.6.3 Dataset

ImageNet is a database of over 15 million annotated large images grouped into approximately 22K categories. These images were collected from the internet and identified with people using Amazon's Mechanical Turk crowdsourcing service. Since 2010, the Pascal Visual Object Challenge has included an annual contest known as the ILSVRC. ILSVRC use a portion of ImageNet, with approximately 1000 images for each of the 1000 classes. There are approximately 1.2 million train images, 50K validation photos, and 150K test photos in total[1]. The AlexNet Caffe model, which contains 61 million parameters as well it has prediction accuracy for the top-1 is 57.2 %, while the overall accuracy for the top-5 is 80.3 %. And while VGG-16 has 138 million parameters and considerably more convolutional layers, it only has three fully connected layers at the moment.[31].

1.6.4 Results and Evaluation

We employed frequently used evaluation metrics to assess the effectiveness of our approach, such as accuracy, throughput, energy efficiency, and resource consumption. Finally, we contrast, our approaches to earlier research.

1.7 Organization

The remains of this work are structured as follows:

In Chapter 2, provides a basic description of FPGA followed by introduction of the FPGA design process known as High-level Synthesis. Then, we examine the fundamentals of the design tools needed in this research, such as the OpenCL platform and the Intel FPGA SDK for OpenCL. Finally, we will go through convolutional neural networks and backpropagation.

In chapter 3, Provides related work reviews. It is divided into three sections in which we describe the most current research in CNNs with GPUs, ASICs as well as FPGAs.

In Chapter 4, we discussed the proposed methods used in this paper. It first discusses the OpenCL design process, and then hardware accelerator architecture design,

performance, data quantization, memory communication, parallelism, and data reuse in the convolutional layer. Finally, we discussed converting fully connected layer to convolution layers.

In chapter 5, we discussed about the results. We have discussed about the experimental setup, which followed by performance analysis of the proposed design that include accuracy comparison, performance, and resource utilization. Finally, we discuss the power measurement and comparison with previous works CNN and other HPC platform design. In chapter 6, we provide conclusions and recommendations for future work.

Chapter 2

Background

This chapter explains the basic theoretical basis for solving image classification problems. As such, we explain how hardware accelerators are used for image classification by first giving brief description of the hardware platforms and convolution neural network.

2.1 Architecture of FPGA

FPGAs (Field Programmable Gate Arrays) was first used nearly two and a half decades ago. FPGAs are semiconductor devices that are built around a grid of configurable logic blocks (CLBs) interlinked via programmable interconnects. The FPGAs are programmable devices that offer a versatile platform for developing unique hardware capabilities at a reduced development cost. [32].

The modern FPGA has two main parts: programmable logic blocks (ALMs) and logic components. [12]. Figure 2.1 shows that the FPGA has a different configurable logic block (CLB) as well as input and output ports. The configurable logic block (CLB) is the basic repeating logic resource on an FPGA, which contains smaller components, such as flip-flops, look-up tables (LUTs), and multiplexers. The FPGA resources that allow connecting the FPGA target to other devices are the input and output (I/O). Input and output are to change analog or digital signals to or from a digital value so that we can process the signals using an FPGA target. The FPGAs logic capacity has been greatly increased because to advancements in process technology, making them a feasible implementation option for bigger and more sophisticated designs. Generally, the FPGA nature of logic and resource usage has an affects the FPGA device's space, speed, and power efficiency. [33].

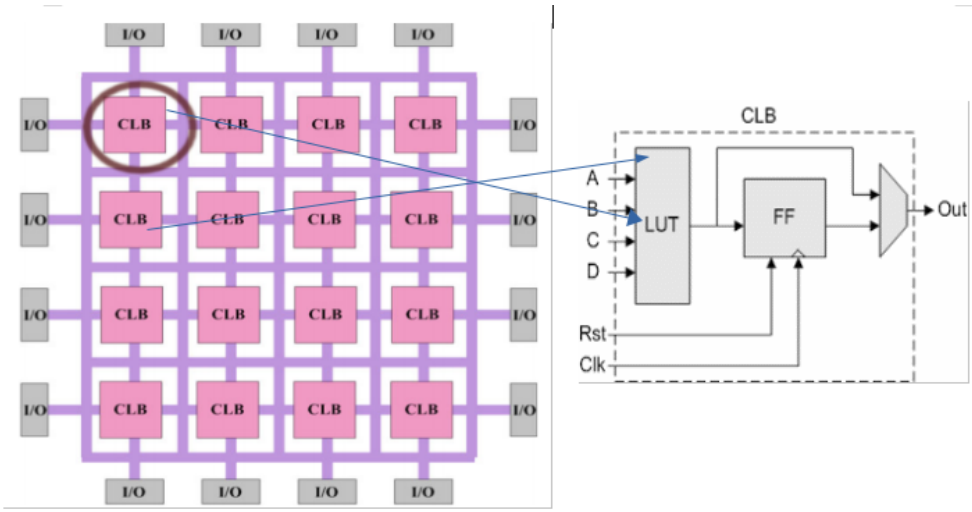

Figure 2.1: Overview of FPGA architecture, taken from [33]

2.2 Overview High Level Synthesis

The high-level synthesis (HLS) is now a methods for generating optimized hardware from high-level language standards such as C/C++ as well as system C. High-level synthesis tools enable developers to specify target system functionality using a software, allowing them to invest on hardware benefits without acquiring hardware knowledge and experience.[34]. There are currently a number of commercial and academic HLS Computer aided tools available. Throughout this study, the Intel SDK for OpenCL is often used to target Intel FPGAs. In figure 2.2, the high-level synthesis is summarized.

Figure 2.2: The High-Level Synthesis Tools, adopted from [34]

Compiler	Input	Output	Owner	license
CHC	C subset	VDHL/Verilog	Altium	Commercial
Ctos	C/C++	VDHL/Verilog	Cadence	Commercial
Synphony C	C/C++	DHL/Verilog System C	Synopsys	Commercial
LegUP	C	Verilog	U. Toronto	Academic
Bambu	C	Verilog	PolMi	Academic
Intel FPGA SDK for OpenCl	C/C++ with OpenCL	Verilog	Intel	Commercial
Vivao HLS	C/C++ System C	DHL/Verilog System C	Xilinx	Commercial
CoDeveloper	Impulse-C	Verilog	mpulse-Accelerated	Commercial

2.3 Overview of OpenCL

OpenCL is that is a device agnostic programming interface that allows parallel programs to be written easily. This allows OpenCL program to be portable across multiple computing platforms such as CPUs, GPUs, and FPGAs. However, performance may not be portable[35]. OpenCL is a library that can be used with any C/C++ compiler, which makes it self-governing of additional tools. The kernel compiler originates with the OpenCL application provided by the vendor. Universal running of a kernel written in OpenCL, like traditional CPUs, Intel Coprocessors for Xeon Phi, GPGPUs, some FPGAs, and even mobile device[36]. There are four frameworks for the OpenCL architecture that will be discussed below.

2.3.1 Platform Model

The host is normally a central processing unit that is in charge of managing machines during runtime. Any device that runs on the OpenCL platform, the system is made up of more compute units that are further divided to processing elements.[36]. The actual program is carried out in processing units. Generally speaking, the OpenCL framework model defines the host functions, devices and provides for devices an abstract hardware model[37].

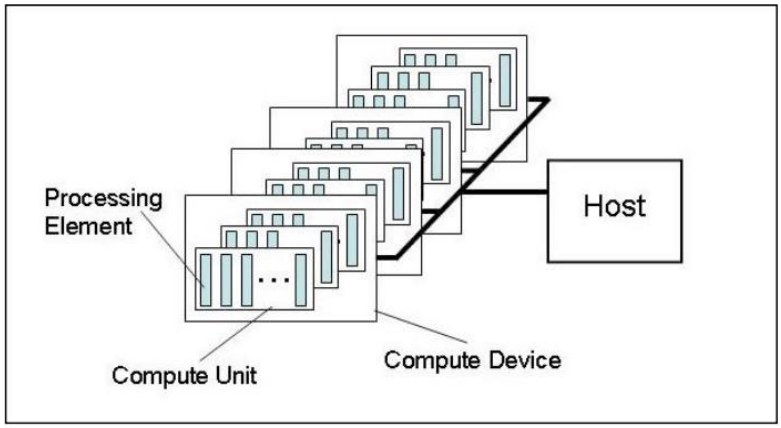

Figure 2.3: Platform Model, taken from [36, 37]

2.3.2 Execution Model

The OpenCL program is divided into two parts: host and kernel code. Host section is a particular C/C++ code with any application programming interface (API) for handling command queues in a context, program objects, memory objects, and kernels [38]. The kernel code hold the essential computational portion that runs on the devices. Until a host may order the execution of a kernel on a device, it is important to configure a context on the host that allows it to transfer commands and data to the device[39].

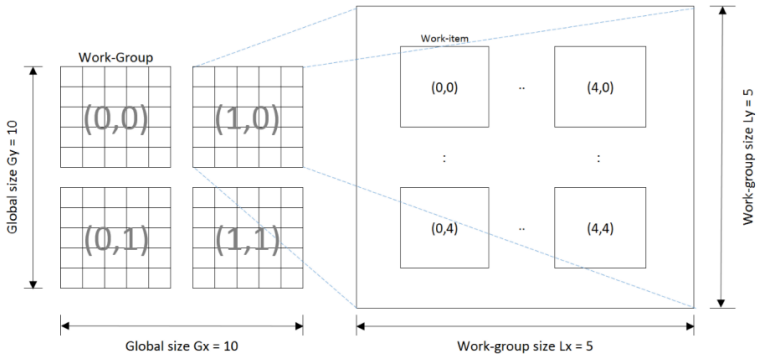

Figure 2.4: The 2D Range Execution Model , taken from [38]

Context The context is developed for one or more devices as well as contains all of the information required for targeted device. A context is an abstract container in OpenCL where on the host, it occurs. A context coordinates host device interface processes, It manages the memory objects obtainable to devices and track the progress of device. The kernels and programs that are generated for each device.

Program Objects The program contains the kernel binary implementation, which provides as a dynamic library for various kernels all through the running time.

Memory Objects This will be used to send data between the host as well as the devices. In the memory details, the memory model would be described.

Command Queue A command queue is required to support command execution within the host. There are three types of commands: Memory commands are used to

transfer memory, kernel commands are used to start it up kernels, and synchronization commands are used to manually allocate synchronization points in host code.

Work Items and Work Groups Several more kernel occurrences have occurred concurrently, with each instance referred to as a work-item. Work items are arranged in a three-dimensional area with different dimensions. Their index, which consists of global IDs that are unique across the index space, identifies them within every dimension.

Events In OpenCL, activities have two key roles: Representing dependencies and Providing a profiling mechanism any operation that enqueues an order into a command[39]. That is, any API call that starts with clEnqueue creates an occurrence in the command queue.

2.3.3 A Model of Memory

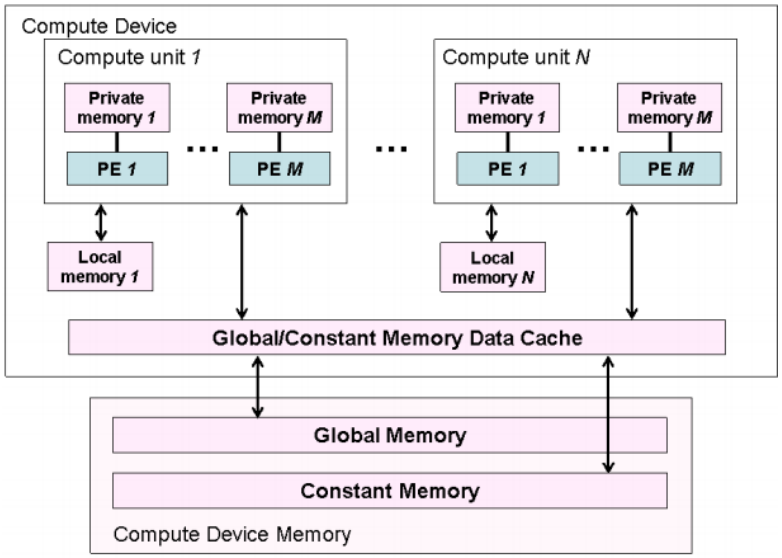

Figure 2.5: OpenCL Memory Model, taken from[39, 37]

The OpenCL framework supports four different memory categories.

Global Memory A memory area visible as well as available to a host as well as all work-items inside devices for read or write operations.

Constant Memory Constant memory is a different memory space in OpenCL. Which, because it may be physically different memory, have different pointer sizes, or use physical addresses that overlap with other spaces and cannot be cast to a different memory space. Global Common Memory refers to the fact that constant data is copied onto cache prior to launching the kernel.

Local Memory The region of system memory that is only accessible to the work item with the same workgroup device is referred to as local memory. This memory space enables work items within the same compute unit to perform read and write operations. This memory level is typically used to store and transfer data that should be shared among multiple work items.

Private Memory The memory area is only accessible to one work-item and is not accessible to another.

2.3.4 Programming Model

The OpenCL programming model is based on the generation of complex work from data parallel execution access points[31]. A evaluation is defined in terms of a set of instructions which implements at every point in an N-dimensional index area inside a given data parallel implementation, also recognized like a kernel launch[40].

2.4 Intel FPGA SDK for OpenCL

A high-level abstraction for FPGA programming is provided by the Intel OpenCL SDK as one of the HLS tools. A concurrent program is built to von Neumann fixed structure as shown in a series of instructions for hardware acceleration. That each computation generally requires the retrieval of instructions as well as the moving of data between register data and also the memory.[36]. The Intel OpenCL SDK solution, on the other hand, provides a highly effective solution. Inside this model, the platform resources are customized to the algorithm being run[36].

Global memory is arranged as external memory in the FPGA device for the memory

system in the Intel OpenCL SDK, which could be DDR3 synchronous dynamic random access memory as well as other memory. [39]. Table 2.1 summarizes the OpenCL memory model for FPGA.

Table 2.1: FPGA and OpenCL memory

OpenCL Memory	FPGA memory
Global	External
Constant	Cache C
Local memory (C)	On- Chip memory
Private	On- Chip register

2.5 Convolutional Neural Networks

CNN is a type of deep neural network that is very useful for classification. It takes an input and predicts a class tag for it. CNN typically consists of many layers, such as convolutional layers, ReLU layers, pooling layers, normalization layers, and fully connected layers. So every layer will have its own input and output, with the input mapped to either a linear or nonlinear transformation of the output. Below are listed a descriptions of the individual layers.

2.5.1 Convolutional layer

The convolution layer parameters are made up of a series of learnable filters. Each filter has a small spatial footprint, but extending to the maximum depth of the input volume. CNN's most important layer is the convolutional layer. It's being used to retrieve the characteristics of the input image or the upper layer's extracted features data [38]. The procedure is a three dimensional convolution calculation based on input data and a huge variety convolution kernels, as well as the convolution operation is essentially a three dimensional multiply accumulate operation that could be described mathematically.

$$y_{out}(f_o, y, x) = \sum_{i_y=0}^{i-1} \sum_{i_x=0}^{i-1} w_l(f_o, i_y, i_x) \times y_i(f_i, y+i_y, x+i_x) + b_i \tag{2.1}$$

In which $y_i(f_i, y, x)$ as well as $y_{out}(f_o, y, x)$ referring neurons as in input extracted feature f_i but also extracted feature f_o, respectively. $W_l(f_o, f_i, y, x)$ demonstrates the weights in the l^{th} layer which is combined with f_i, as well as b_i would be a bias. The convolution filters are $i \times i$ in length.

2.5.2 Rectified Linear Unit Layer

The Rectified Linear Unit (ReLU), a recently proposed activation function in CNN, could be used by threshold holding a matrix at least zero. Which would be known to have a lower computational complexity as well as converging faster throughout training. Whereas the Sigmoid or tanh (x) activation functions necessitate a time-consuming mathematical operation [38]. In the convolutional neural network framework, the ReLU has become very popular in recent years. The mathematical statement of rectified linear unit layer as follows:

$$f(x) = max(0, x) \tag{2.2}$$

2.5.3 Pooling Layer

The pool layer, as well known as down sample layer, reduces extracted feature redundancy as well a network computational cost by minimizing extracted feature dimensions but rather effectively prevents over-fitting. Pooling is among the common operators inside a convolutional neural network. Convolved extracted feature are compressed in a pooling layer by a dataset obtained near the area feature values [41]. Because images the have regional property, this operation is possible. Pooling is spatial size of feature values to reduced the large computation for layers after the pooling layer. The pooling operator's most common options include max pooling as well as average pooling. The term "max-pooling" refers to the following:

$$O(i, j) = max[O(i_o, j_o) : i \leq i_o < i + p, j \leq j_o < j + p] \tag{2.3}$$

In which p is the operator's length and (i, j) is the vertical and horizontal index.

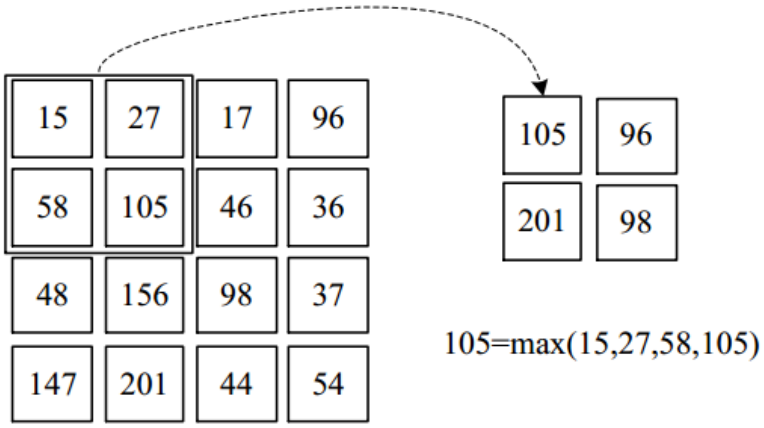

Figure 2.6: Max-pool, taken form [38]

2.5.4 Fully-connected Layer

The Fully-connected layer is the classical component of a feed-forward neural network, wherein every element inside the max pooling is linked to every component in the output nodes. The extracted features of the convolutional as well as max pooling require the input image's distributed high-level attributes. The fully connected layers were outlined to add such extracted features in order to categorize the input into several classes. The forward throw of the l^{th} FC layer is calculated as follows:

$$O^{l+1} = f(b^l + f^l + w^l) \tag{2.4}$$

Where the O^{l+1} output at $l+1$ layers, f activation function, b^l bias for l^{th} layers, f^l feature map in l th layers, and W^l weights at the l^{th} layers.

By adjusting the filter size of the convolution controller, an FC layer could be easily translated to the convolutional layer, which would be especially useful in practice.

2.5.5 Normalization Layer

Normalization has now become critical for deep learning models, which compensate for the infinite nature of certain activation functions. The output layers are still not limited within such a limited range through these activation functions, but it can rise as high as the training allows. Normalization is used until activation function to prevent unbounded activation from increasing the output layer values. Local Response Nor-

malization (LRN) had first been introduced throughout the AlexNet architecture with ReLU as with the activation function [38].With the exception of the previous paragraph explanation, the use of LRN has been improving to improve lateral inhibition[8] . It is indeed a concept in neurobiology that refers to a neuron's ability to reduce the activity of its neighbors [1].

The goal of all this lateral inhibition throughout DNNs would be to improve local comparison so that maximum image pixels were being used as excitation for the next layers locally. The LRN would be a non-trainable layer which normalizes pixel values within such a local neighborhood in an exrtacted features. The following formula is used to perform normalization in the depth dimension for each (x,y) location.

$$b^i_{x,y} = a^i_{x,y}/(k+\alpha \sum_{j=max(0,i-n/2)}^{min(N-1,i+n/2)} (a^i_{x,y})^2)^\beta \tag{2.5}$$

This equation is taken form [1].

For which i represents the output of filter i, $a(x,y)$ as well as $b(x,y)$ represent the image pixels at (x,y) location during normalization respectively, and N represents the total number of channels. The variables (k,α,β,n) were also hyper parameters. K has been used to eliminate singularities (divide by zero), $alpha$ is a normalization constant, as well as β is a characterized constant. The constant n can be used to describe the neighborhood size, or even how many sequential image pixels must be recognized even as normalizing.

2.6 Backpropagation

For back propagation has performed two updates that are for the weights and the deltas[42]. We are looking to compute $\frac{\partial E}{\partial w^l_{m,n}}$Which can be translated as the measurement of how a single pixel alters $w_{m,n}$ in the weight affects the loss function E. During forward propagation, the convolution operation ensures that the pixel $w_{m,n}$ in the weight, between and element of the weight and the input feature map element that it overlaps, a contribution is made in all the products[41]. Convolution between the input feature map of

dimension $H \times W$ and the weight of dimension $k_1 \times k_2$ produces an output feature map of size $(H - k_1 + 1)$ by $(W - k_2 + 1)$. By applying the chain rule in the following way, the gradient component for the individual weights can be obtained[9].

$$\frac{\partial E}{\partial w^l_{m,n}} = \sum_{i=0}^{H-k_1} \sum_{j=0}^{W-k_2} \delta^l_{i,j} \frac{\partial x^l_{i,j}}{\partial w^l_{m,n}} \tag{2.6}$$

The summations represents a collection of all the gradients $\partial^l_{i,j}$ coming from all the outputs in layer l.

Chapter 3

Related work

Convolutional neural networks have been used in a variety of applications over the last decade, which would include object detection, image classification, natural language processing, and others. We examine convolutional neural network implementations on various hardware accelerators such as GPUs, ASICs, and FPGAs..

3.1 Convolutional Neural Networks on GPUs

Due to an increased computation inquire, the use of many core platform has recently become hot for the high-performance computing region. Convolutional neural networks approach was among the most powerful machine learning for a wide range of important real-world problems. CNNs require a considerable amount of data processing both for training and forward classification.

Convolution has recently been recognized as among the most computationally intensive operations in deep networks [1]. When trying to compare to certain other systems, which have a number of core CPU-based methods, have demonstrated inadequacy for the memory block issue and low parallelism [8]. The multiprocessor CPU platforms find it difficult to provide sufficient computation capacity [1, 8]. Recent advancements in CPU design, on the other hand, show a significant improvement in computing capabilities, narrowing the performance gap between CPUs and GPUs[43]. With their high throughput and bandwidth, many core GPUs are some of the most commonly used to enhance both the training and classification processes of the convolution neural network [4]. It is developed for large throughput, as the latest GPUs could execute hundred of hundred of floating-point operations per second (FLOPS)[12]. Because of all these benefits, many researchers have focused on accelerating GPU based convolutional neural network implementations [12]. Nvidia's huge Kepler 40 (K40) GPU accelerator could compute and perform around four thousand floating point multiple as well as add operations per clock cycle while also that upto 288 GOPS of memory bandwidth [44].

The work by Alex Krizhevsky et al. [1] accelerated the computational process by using a workload partitioning technique with an emphasis on communication on multi-GPUs. And won the ImageNet contest with the high-resolution image and reduce the error rate form 26.2% to 15.3%. M. Mathieu et al. [9] used GTX Titan GPU for simple algorithm which speeds up training and inference by a significant factor. They improved single-node GPU performance through substituting convolution with the fast fourier transform (FFT).

Han et al. [45] recently proposed the compressed pipeline architecture have used pruning as well as weight sharing methods, which were implemented and benchmarked on CPU, GPU, and mobile GPU. Their results show that compressed networks have $3X$ to $4X$ layerwise speedup and $7X$ energy efficiency. The work by Jeng-Hau Lin et al. [4] They proposed the binary convolution neural network of distinguishable filters on binaries that uses singular value decomposition (SVD) on binary convolution neural network kernels to decrease computational as well as storage complex nature. They reduced compilation time through 31.3 % as well as reduced memory consumption by 17 % to BCNN, only with slight accuracy ultimate sacrifice.

In general, GPUs use a significant some aggregate of power, which would be a critical vital throughout hardware acceleration valuation metric in modern GPU. [10]. Those certain accelerators have been investigated to implement convolutional neural network as a growing range of applicants require low power solution.

3.2 The Convolutional Neural Networks on ASICs

ASIC is an abbreviation for the application specific integrated circuit. ASICs, even as the name implies, are applications that are only associated with one thing. They are dedicated to a single purpose and perform the same function throughout their operating period. ASIC seems to be a custom architecture designed for a specific use, though low cost and high throughput, but still it takes a lot of time to develop.

The DaDianNao supercomputer was developed for machine learning, and therefore by utilizing on-chip memory, they were able to achieve a significant speedup while

consuming less power than the GPU.[46]. The work by Haitong Li et al. [47] the design space exploration provides a comprehensive trade off interpretation for practical mobile DNN accelerator design features, resulting in a landscape energy area efficiency for various on-chip memories. Their research demonstrates that an accelerator based on MRAM can save up to 4.68x energy (57 % space overhead) and a 3D VRRAM-based design can save up to 2.22x energy (33 % space reduction).

Finally, EIE by Han et al. [48] by pruning the redundant connections, we were able to take advantage of deep compression schemes. Their analysis demonstrates 102 GGOPS processing power on the a compressed system of 600 MW power consumption, which would be 3,400x as well as 2.9x more effective than even a GPU as well as DaDianNao respectively. ASIC solutions, in general, aim to provide preferable performance in terms of energy usage as well as throughput. However, there are some disadvantages to this approach,such as an insufficiency of ability to adapt, high development expenses, as well as lengthy processing times, prevent their wide acquisition.

3.3 Convolutional Neural Networks on FPGA

Recent FPGAs had also provided a significant design space for a convolutional neural network due to an increase throughout FPGA fabric density as well as reducing transistor scale.

The work by Tapiador et al. [6] implemented a depth-wise separable convolution with a high rate of resources. Also significantly increases bandwidth as well as accomplishes a complete pipeline through parameter tuning and through a streaming data interface and the on ping-pong. The work by Kaiyoua et al. [49] CNN models and CNN-based implementations have been distinguished. The requirements for memory, computation, and system reliability for mapping CNN on embedded FPGAs were summarized. Requirement analysis, they proposed Angel-Eye, which is a programmable as well as configurable CNN hardware accelerator combined with quantization method, compilation tool, as well as then a data quantization technique. The compilation tool converts a specific CNN model into the hardware configuration. They were tested on

the Zynq XC7Z045 platform and outperformed previous FPGAs work by 6*x* and 5*x* efficiency respectively. The work by Naveen Suda et al. [5] by optimizing FPGA throughput on large scale CNN with 3-D convolution as matrix multiplication. Their work demonstrated that ImageNet classification on the P395-D8 board can achieve a peak performance of 136.5 GOPS for convolution operations and 117.8 GOPS for the entire VGG network.

Generally, FPGA based CNN acceleration has big advantages due to its low power consumption, deliver high throughput, design flexibility, and require less development costs.

Table 3.1: Summary on convolutional neural networks on FPGAs

Researches	Approach	Methods	Models	Data precision	Boards	GOPS
[13] ALAMO: FPGA acceleration of deep learning algorithms with a modularized RTL compiler.	CNN	RTL	ALEXNET, NiN CNN	8-bit	Stratix-VGA7	114.5 117.3
[5] Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks	CNN	Intel OpenCl	VGG	16-bit	Stratix-V GSD8	117.8
[38] An FPGA-Based CNN Accelerator Integrating Depthwise Separable Convolution.	CNN	Vivado HLS	MobileNet	32 FP	Xilinx- ZYNQZ-7100	17.11
[50] Going Deeper with Embedded FPGA Platform for Convolutional Neural Network.	CNN	Vivado HLS	VGG16-SVD, VGG-16	8/4-bit	Xilinx Zynq ZC706	187.8-137.0
[49] Angle-Eye:A complete Design flow for mapping CNN onto embedded FPGA.	CNN	RTL	VGG16	16-bit	Zynqxc-7z045	137

[29]Optimized Compression for Implementing Convolutional Neural Networks on FPGA	CNN	Xilinx HLS	AlexNet	32 bit FP	Virtex 7 VX485T	17.67
[18] Scalable and modularized rtl compilation of convolutional neural networks onto fpga,	CNN	RTL	AlexNet	8-16 bit fixed	Stratix-V GXA7	114.5

Chapter 4

Proposed Approach

In this chapter, we talk through the proposed methodology for implementing a convolutional neural network on to an FPGA using the Intel OpenCL SDK. We begin by exploring the different techniques for easing the approach, such as OpenCL desig for Intel FPGA SDK, design architecture, and optimization for performance in order for an integrated hardware to provide the best possible performance and resource utilization.

4.1 Intel FPGA SDK for OpenCL

The Intel FPGA SDK for OpenCL allows developers to use heterogeneous platforms such as Intel CPUs and FPGAs to accelerate their applications. Also, fully utilize FPGAs' unique capabilities to deliver high-performance acceleration with low latency and low power consumption. It increases resource utilization and decreases local memory usage.

4.2 OpenCL Design Outflow on FPGA

The emulator kernel program is generated by the Intel offline Compiler (AOC) at a first phase of the OpenCL design for the Intel FPGA SDK for OpenCL. The emulator, which can operate on both an x86 and an x64 host, can quickly check for syntax errors and functional correctness.

An Intel offline compiler also creates an optimization report, while building the emulator, which contains information on memory transfers and the execution initiation interval and resource utilization. The Intel FPGA SDK for OpenCL tools automatically handling the interaction between memory regions. The kernel program is then fully compiled with Intel offline compiler, which allows the OpenCL code to be synthesized immediately to the Verilog. Depending on a device program, full compilation may take hours during the binary file generation process.

Finally, the GNU Compiler Collection (GCC) compile the host and devices portion

code in order to generate a hardware configuration binary file, which is an executable on the FPGA platform. If the kernel's performance or resource use does not match the criteria, it must be verified and built again. Generally, figure 4.1 shows the overall implementation process of the Intel FPGA for OpenCl design outflow accelerating for the FPGA platform.

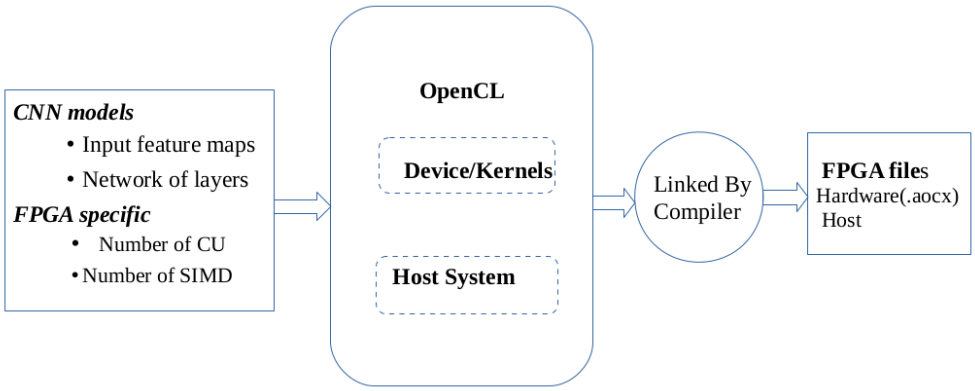

Figure 4.1: OpenCL Design Outflow

4.3 Hardware Accelerator Architecture

In this section, we would go over the architecture in general, including convolution, input, max-pooling, backward and output kernels.

4.3.1 Proposed System

The figure 4.2 shows that the overall system design flow as well as both host and device system section of the OpenCL kernels created with the Intel FPGA SDK for the OpenCL enhanced version channel. The hardware accelerators design has five kernels, such as forward convolution, backward convolution, pooling, input, and output. The input and output kernels have been used to transfer extracted features as well as weight from and to the main memory, which brings some kernels with high - throughput of data. The convolution kernel is designed to speed up the most computations in CNNs, which typically include the convolution operation and the FC layer.[7].

The max-pool perform to dwindle the dimension of weight and undersampling operations specifically on the output data stream of the convolutional layer. The cascaded

kernels shape a channel,which can operate the essential CNN operations without the require of putting away interlayer information backmost to external memory. So, every convolution channel has a computing unit, and the kernel is made up of many computing units to do parallel convolution[9]. The input and output kernels are a most vital kernel which utilized for a data movement and a kernel that are outlined to bring or store information from or to a main memory for the computing path. The input kernels begin with such a global work-items in convolution configuration. Whereas the output kernel is operating in an NDRange unveiling with global work-items. To enable concurrent work group processing, the work-items have been organized up into multiple running in parallel work groups, also with a local work group length of (i, i). The convolution filters size is 3×3, which it minimizes computational costs and weight sharing that to lower back-propagation weights. As well as the number of pixels shifted over the input matrix is referred to as the stride and we use the stride size is 2×2.to modifies the amount of movement over the image

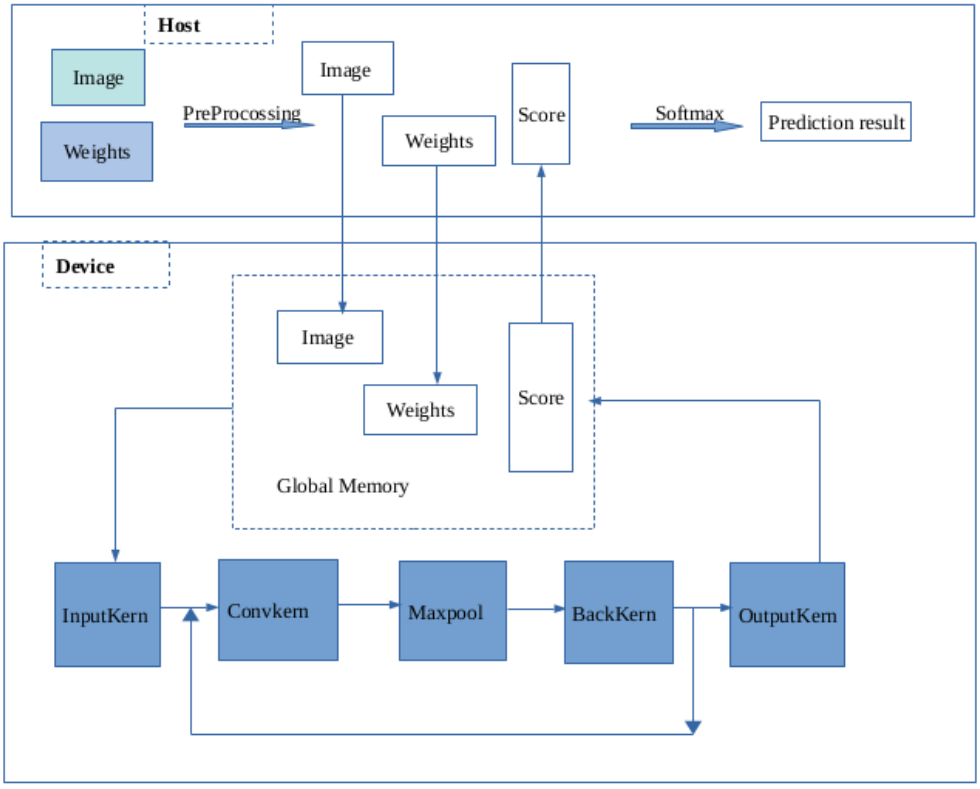

Figure 4.2: Block Diagram of Proposed System

4.3.2 Forward Convolution Kernel

The forward convolution kernel performs a convolution operation. At each position, the multiplication between each element of the kernel and the input feature map element, it overlay is computed and the results summed up to obtain the output at that current location. The convolution operation is essentially a three dimensional multiply accumulate (MAC) operation, which can be defined as.

$$O_{out}(fe_o, y, x) = \sum_{fe_i=1}^{C_l} \sum_{i_y=0}^{i-1} \sum_{i_x=0}^{i-1} w_l(fe_o, i_y, i_x) \times y_i(fe_i, y+i_y, x+i_x) + b_i \tag{4.1}$$

In which $y_i(fe_i, y, x)$ as well as $y_{out}(fe_o, y, x)$ referring neurons as in input extracted feature f_i but also extracted feature fe_o, respectively. $W_l(fe_o, fe_i, y, x)$ demonstrates the weights in the l^{th} layer which is combined with fe_i, as well as b_i would be a bias.

Algorithm 1 Algorithm for convolution kernel

```
Get global as well as local index
Initializing the channel for data, weight, bias out and convolution input
Initializing the accumulate, round bit and sum of bias
Initializing convolution inner count
convolution loop count iterations generate NUM output pixels
for for each elements of k output NUM multiple with the conv_loop_cnt do
    if (conv_cnt == 0) then
        Iniatializing the deep register which store th copy of partial result
        Add the result from all channels output and accumulate the last copy
        Shift the registers to backwards
    end if
    if (conv_cnt == conv_loop_cnt-1) then
        The convolution loop is finished and NUM convolution generated
        Then the bias and rounding operation is performed.
        #pragm unroll
        for each element l in NUM do
            Accumulate all the partial result
            Round and truncate the result to the output precision
            1st, perform the sign extension and in the 1st step rounding before sum with bias
            2nd, deal with overflow and underflow case and rounding after sum with bias
        end for
    end if
end for
```

4.3.3 Input kernel

The input kernel is used for reading input extracted features and relates filters from memory, along with feeding weight into the local buffer and obtaining extracted features and caching them in the local buffer. Because an input extracted feature is recycled by numerous different filters, the input array is cached in local memory for access during a data processing as well as to reduce the access of the global memery.

Algorithm 2 algorithm for Input kernel.

```
1: Get global and local index of work-item
2: Calculate the location for input features also filters by index
3: Bring input features into the cache memory
4: Bring filter into the local memory
5: #progma unroll
6: for each component i input feature and filter do
7:     Load weight into weight buffer
8:     Fetch the weight and bias by fetcher
9: end for
10:
```

4.3.4 Max-Pooling kernel

The max-pool layer, as well known as down sample layer, reduces extracted feature redundancy as well a network computational cost by minimizing extracted feature dimensions but rather effectively prevents over-fitting. Pooling is among the common operators in- side a convolutional neural network. The pooling layer reduce the convolutional outcomes while using the average or maximum value of elements in an area that is dependent on subsequent iterations. The shift register is outlined for buffering the pooling data, similar to such convolutional layer. Then, depending on the pooling method, accumulating operations are performed on the shift register.

4.3.5 Output Kernel

This kernel used to read initial the feed forward result from the accumulation buffer and write to a global memory. And then the output to a local buffer, then extract the data from the buffer and copy it back to DDR. This work makes use of batch processing to reduce the time it takes for filters to be reused in FC layers. As a result, in the FC layer

output kernel, N batch sets of results must be collected and written. It processes one set of results for the additional layer. The kernel is constructed in a NDRange manner, executing with work-items in parallel, so the output processing is entirely independent.

4.3.6 Backward Convolution Kernel

This kernel read the result from the max-pooling buffer channel as well as performs two functions: error δ calculation and partial derivatives ΔW and ΔE calculation. To calculating the error and partial derivatives, we require two input buffers for both δ^l and δ^{l-1} in order to store the data for temporary. Convolution between the input feature map of dimension $H \times W$ and the weight of dimension $k_1 \times k_2$ produces an output feature map of size $(H - k_1 + 1)$ by $(W - k_2 + 1)$. By applying the chain rule in the following way, the gradient component for the individual weights can be obtained[9].

$$\frac{\partial E}{\partial w^l_{m,n}} = \sum_{i=0}^{H-k_1} \sum_{j=0}^{W-k_2} \delta^l_{i,j} \frac{\partial x^l_{i,j}}{\partial w^l_{m,n}} \tag{4.2}$$

4.4 Optimizations for Performance

In this section, we will discuss performance optimization techniques such as throughput maximizing, quantization, memory communication, parallelism in convolution neural networks, and converting fully connected layer to convolution layer.

4.4.1 Throughput

To keep moving forward the accelerator's throughput the SIMD, and computing units were used. The input kernel retrieves the SIMD and sends it to numerous computing units in the convolution. By adjusting the value of the SIMD as well as the number of computing units that is deployed design could obtain scalable performance and hardware costs without requiring changes to the kernel code.

- **Computing Unit:**

  The replication of full data paths is the total number of computing units. This necessitates the creation of multiple copies of the various transmission lines to

increase the throughput. Even so, multiple computing units could not always improve throughput linearly since all computing units communicate over the global memory bandwidth. This causes memory access contention among computing units.

- **Single instruction multiple data (SIMD):**

  This is used to increase data processing performance of an OpenCL kernel by enabling work -items in same work-group. The work group size that could be defined by the compiler and used the local work size argument to clEnqueueNDRangeKernel. The workgroup length that can be allowed to pass to clEnqueueNDRangeKernel as such local work length argument. The above enables the compiler to adequate enhance the generated kernel code.

- **Loop Unrolling:**

  The several loop iterations in the device code it could have an impact on the kernel performance. The Loop unrolling method could assign the most hardware resources and minimize or even eliminate the loop queue, that is increase the throughput a linear manner . This approach as well supports memory coalescing, that also reduces memory transaction cost.

4.4.2 Quantization

Quantization is the process of estimating a neural network that uses floating-point bit with a neural network that uses fixed-point numbers. Converting from a floating-point value to a fixed-point value it involves the overall number of bits is calculated as the sum of IL and FL as well as the fixed-point number has an exactness of 2^{-FL} , as well as the scope could be described this way: $[-2^{IL-1}$, $2^{IL-1} - 2^{-FL}]$ [51]. Where IL seems to be the total number of integer bits and FL has been the total number of fractional bits, would be a fixed-point number structure.

The fixed point is the hardware-friendly as well as minimize the compute resources in FPGAs [52]. This even decrease the chip's memory usage and cost effective and much more precise hardware kernels [32]. Several more latest FPGA works on con-

volutional neural networks had also centred to use the fixed-point representation of the extremely narrow bit-width, which now has accuracy reduction [51, 45, 53]. But nevertheless, low bit reduction based designs demonstrate exceptional performance and energy efficiency, this indicating that extremely low bit-width is an useful solution for higher efficiency design [51, 54, 55]. In overall, quantization is the most significant element in accelerating huge CNNs on the FPGA platform.

4.4.3 Memory Communication

Because several developments are limited by memory bandwidth, the other option is to use efficient memory access to reduce communication cost. Subject of optimization to paying attention.

- **Memory Alignment:**

  Here on host side, memory is allocated would have to be at least 64-byte aligned. This significantly improves the transmission efficiency of DMA transmitting on the host-FPGA communication. The allocation can be executed in Linux using the POSIX mem-align function, which is supported by GCC, or in Windows use that aligned malloc function, which is hold by Microsoft.

- **The Local Memory Caching :**

  Global memory, constant memory, local memory, and private memory are the four areas of the OpenCL memory model. Local memory, which would be executed in the on-chip Random access memory block, does have significantly decreased latency and high bandwidth than main memory. As a result, we can cache global memory which it requires multiple accesses previous to computation using local memory. Those certain cached local memories have been viewable to everyone, work-items in a same work group when data parallelism is enabled. By minimizing the memory access, the use of local memory would improve kernel performance.

4.4.4 Parallelism in Convolutional Neural Network

Those operations, which include reading, convolving, pooling, as well as writing, are separate of the extracted features. As a result, the entire extracted feature output could be vectorized into a N dimension, to every section extracted on a separate data channel. It could be implemented through an OpenCL computing unit, significantly increasing the proposed design throughput [10]. Furthermore, every extracted feature unit convolution layer consists of stage element wise multiplication of the input extracted feature as well as filters, followed by the accumulation of the product of these operations [38]. In the first process, multiplication is completely independent and could be performed using a data parallelism technique.

4.4.5 Changing FC Layers to Convolution Layers

The fully connected layers has to be converted to an equivalent convolutional layers. Therefore, to convert a fully connected layers o convolutional layer set the filter size to be exactly the size of of the input volume. It should be noted, difference between the fully connected and convolution layer would be that the neurons in the convolution layer are only connected to a local region at the input, and that many of the neurons in the convolution layer volume share parameters. The fully connected layer (FC) operates on a flattened input, with each input connected to all neurons. Dot products, on the other hand, are always computed by neurons in both layers, so their functional form is similar.

Chapter 5

Results

The evaluation of the proposed approach are described in this chapter. The hardware and software that were used in the analysis. The accuracy, resource consumption, efficiency, and energy consumption of a system are then examined for various design requirements in the CNN model. We then compare our design to other current researches on an FPGA based convolution neural network.

5.1 Experimental Setup

The Terasic DE1-SoC Development Kit (DK) of FPGA board is used to implement the experiments. DE1 SoC would be a powerful hardware design platform based on Intel System-On-Chip (SoC) FPGA. The DE1 SoC board uses several features which enable designer to complete a broad range of designing circuits projects.

The terasic DE1 SoC board has M10K-10-kbit memory blocks including soft error correction code (ECC), as well as a 400 MHz/800 Mbps interface of an external memory and 64MB of the SDRAM, 1GB (2×256M×16) of DDR3, and micro SD card port on Hard Processor System (HPS) memory [56]. The Intel cyclone V SoC 5CSEMA5F31C6 has 85K programmable logic elements, 4,450 Kbits of memory embedded , 6 fractional phase locked loops (PLLs), dual-core ARM Cortex-A9 (HPS), and 2 memory controllers based on TSMC's 28-nm low power (28LP) process technology. Figure 5.1 illustrates the architecture of a DE1 SoC, which includes two USB 2.0 Host ports (ULPI interface with USB type A connector)[56]. As communication ports, connectors, displays, switches,buttons, indicators, audio and video inputs, G-Sensor on HPS and UART to USB (USB Mini-B connector), 10/100/1000 Ethernet, PS/2 mouse/keyboard, IR emitter/receiver, and I2C multiplexer are used. The accelerator boards communicate with the host through the use of an 8-lane PCI express link. Additionally, the other platform such as (Intel® CoreTM i5-4300) CPU, and (AMD Radeon (TM) R5 M330) GPU are used.

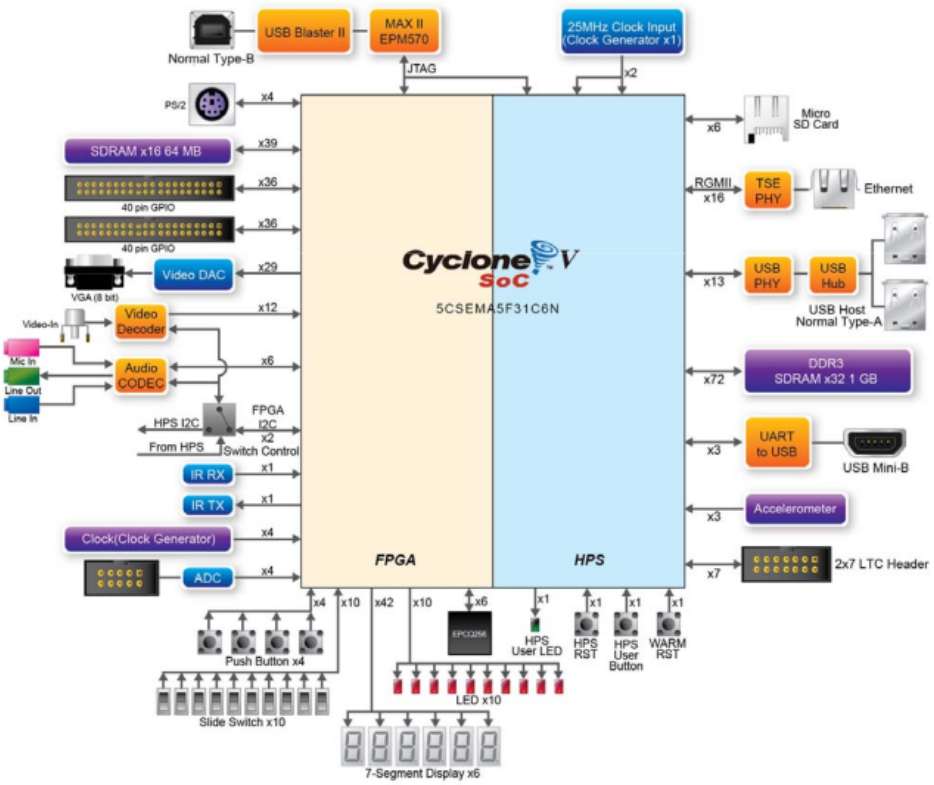

Figure 5.1: Block diagram of DE1-SoC[56]

We use Intel SDK for OpenCL intelFPGA_Standard_18.1.0 build 625. The Intel FPGA SDK for OpenCL Standard Version includes programs, drivers, Development kit library resources as well as files, and much more.. The Intel SDK for OpenCL the Standard_18.1.0 has logic components such like offline Compiler translates, set of commands, host runtime provides the OpenCL host and runtime API for the OpenCL host code. We used the Board Support Package (BSP) 18.1 version for de1soc board BSP from Terasic. And Intel SDK for OpenCL the intelFPGA_Standard_18.1.0 with 625 building are used.

5.2 The Proposed Design's Performance Analysis

In this section we evaluate the performance of our proposed system with the different design specifications chosen in Chapter 4. The objective of this exercise is to learn the resource utilization and performance figures for combinations of design specifications. We employ two well know CNN models such as AlexNet which has 5 layers with a combination max-pool as well as 3 fully connected layers also 60 million learnable pa-

rameters. And also VGG16 which, has sixteen layers with a combination of max-poolin layers as well as 138 learnable parameters and possible combinations of convolutional neural network design specifications. We measure the Top-1 and Top-5 accuracy of the two above-mentioned CNN models. The Top-1 accuracy is the standard accuracy, which a model predicts the one with the highest probability that should be the correct response. While the Top-5 accuracy implies that a model predicts the top five likely outcomes, which match the expected.

5.2.1 Comparison of Accuracy

We advanced to evaluate the accuracy of our design on the ImageNet ILSVRC-2012 data-set, where it contains up to 1.2 million training and 50k validation instances. An AlexNet Caffe model, that has 61 million parameters as well as a top-1 accuracy of 57.2 % and a maximum classification of top-5 accuracy of 80.3 %, has been used as a reference model. On the same ILSVRC-2012 data-set, we furthermore examined a larger, more latest network, VGG-16 [31]. The VGG-16 does have 138 million parameters and much more convolutional layers, but still only three fully-connected layers at the moment [12]. The accuracy of our work was assessed with executing our models on 728K training and 50K validation samples from the ImageNet 2012 data set. The figure below demonstrates the top-1 as well as top-5 accuracy obtained from of the Caffe and FPGA.

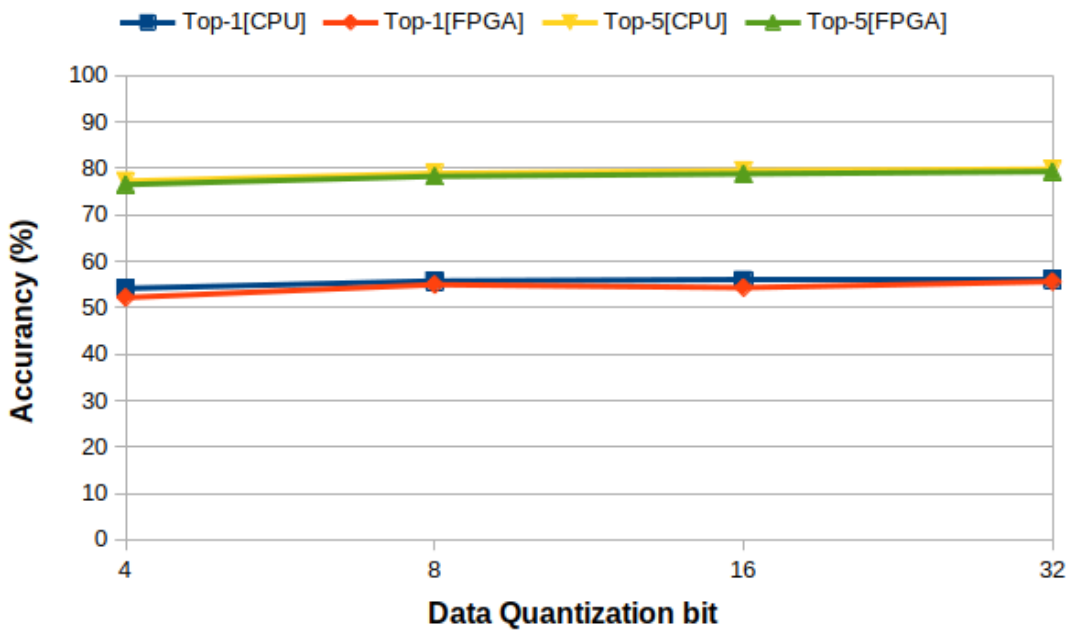

Figure 5.2: The Accuracy Comparison for AlexNet model

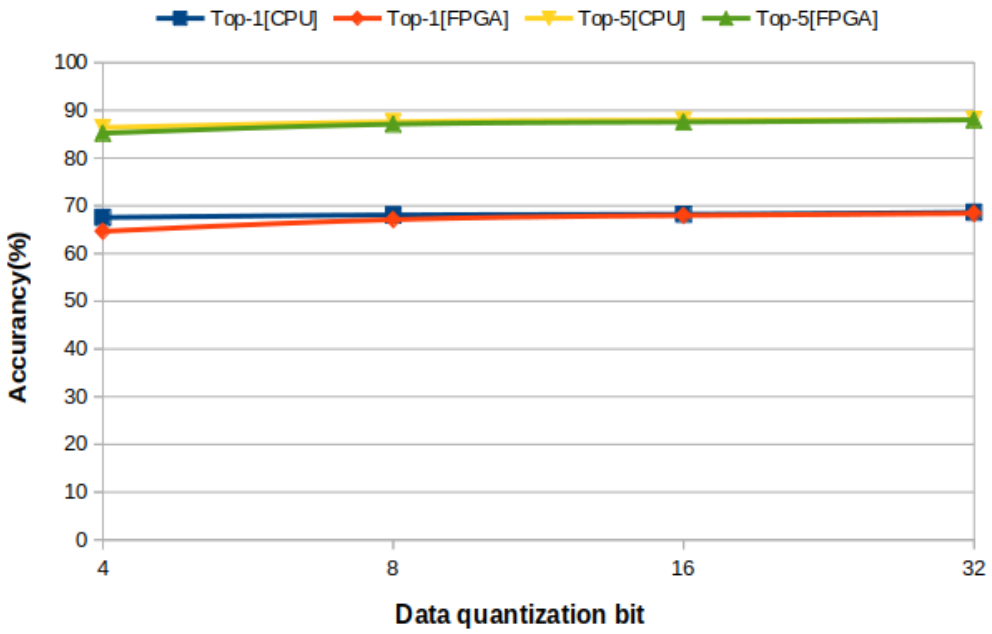

Figure 5.3: The Accuracy Comparison for VGG16 model

Our experiments have shown that with lower precision, training can be performed with little loss or no loss in result accuracy. The figure above demonstrates the accuracy of various quantization compression rates. As has been shown, the network's accuracy starts to decline considerably while compressing below 8 bit data quantization of the its base accuracy.

The difference between the Caffe tool using AlexNet model with 32 bit floating point and the 32 floating point FPGA design on top-1 and top-5 accuracies are 0.5% and 0.59% respectively. The difference between both a 16 bit fixed point Caffe tool and FPGA design on top-1 accuracy are 0.59% and top-5 accuracy, are 0.9% accuracy loss compared to the reference design. The accuracy difference between 8 bit Cffe and FPGGA implementation design on top-1 and top-5 accuracies are 0.77%a and 0.5% respectively. The accuracy difference between 4 bit Caffe tool and FPGGA implementation design on top-1 and top-5 accuracies are 2.05%a and 1.19% respectively. Therefore, the accuracy of our implementation is excellent.

5.3 Computation Throughput and Energy Efficiency

In this subsection, we will discuss the computation throughput as well as the energy efficiency of our system. Figure 5.4, figure 5.5 and figure 5.6 depict throughput for CPU, FPGA and GPU respectively, while figure 5.7. figure 5.8 and figure 5.9 depict energy efficiency for CPU, FPGA and GPU respectively.

5.3.1 Computation Throughput

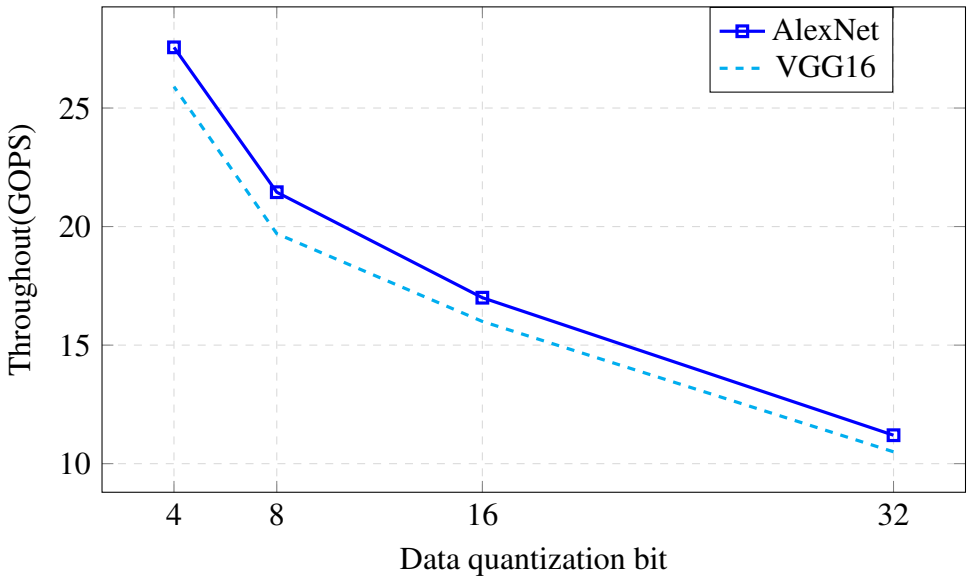

Figure 5.4: Throughout with different data quantization bit with Caffe[CPU]

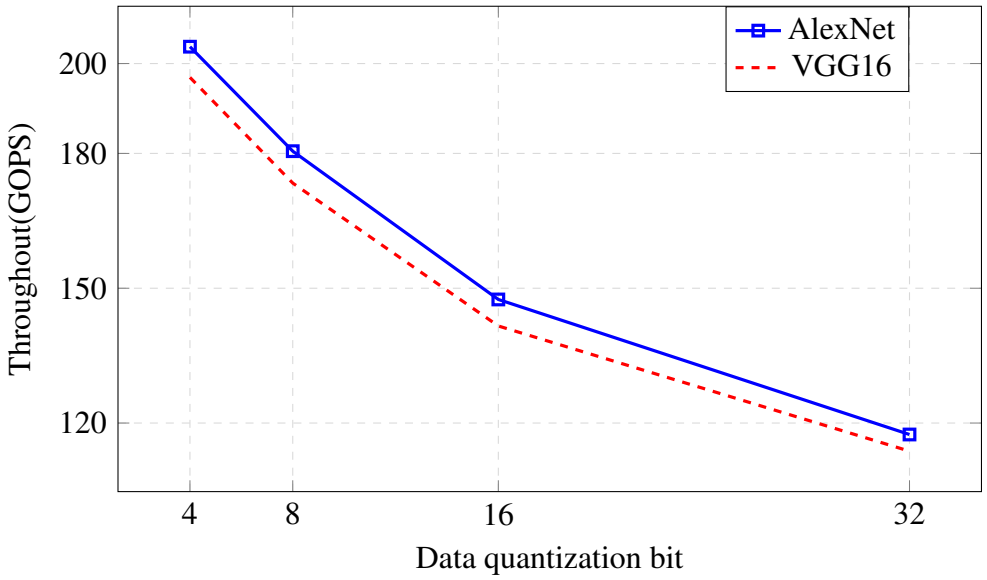

Figure 5.5: Throughput with different data quantization bit on FPGA

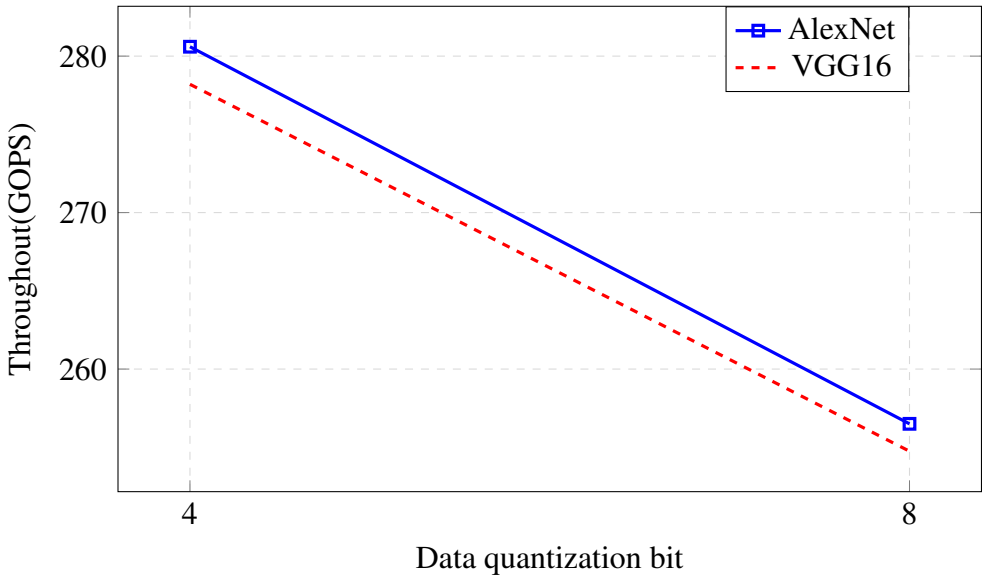

Figure 5.6: Throughput with different data quantization bit on GPU

Our experiments have shown that with low-bit-width quantization, we can achieve a high throughput in results. The low-bit-width quantization techniques have significant benefits. because it allows for high memory cache to be used as well as removes memory constraints in deep learning methods. This enables faster data movements and more efficient computation of the throughput in hardware acceleration. And it enables the device to do more operations per second, significantly speeding up workloads. Because

of these advantages, low-bit-width implementations are likely to become common in training and inference, particularly for convolutional neural networks.

5.3.2 Computation Efficiency

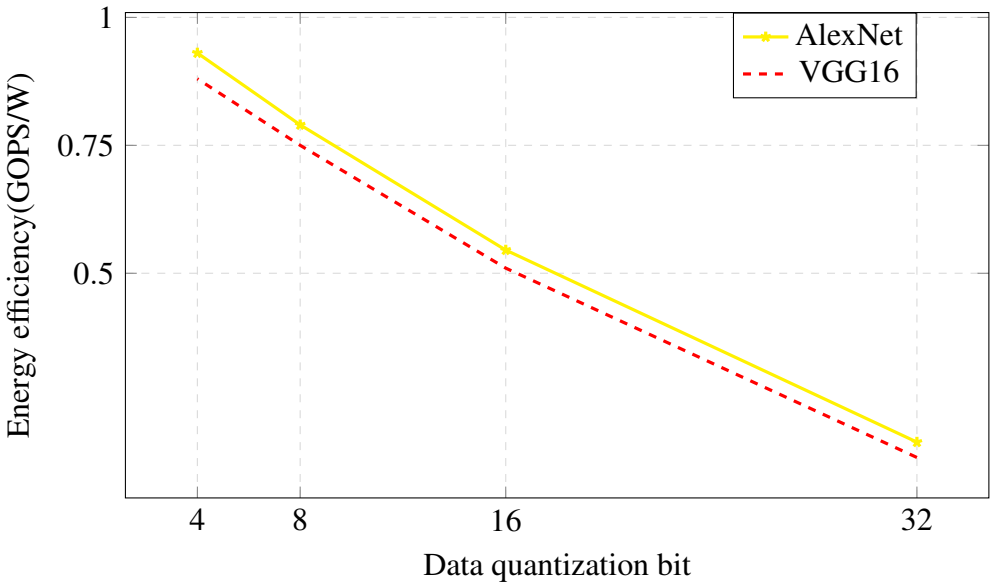

Figure 5.7: Energy efficiency with different data quantization bit with Caffe[CPU]

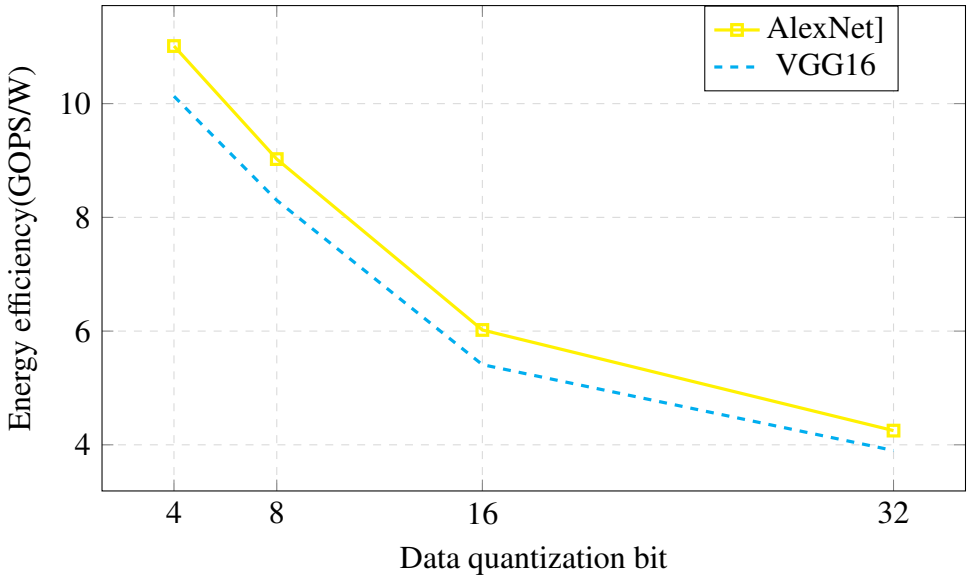

Figure 5.8: Energy efficiency with different data quantization bit on FPGA

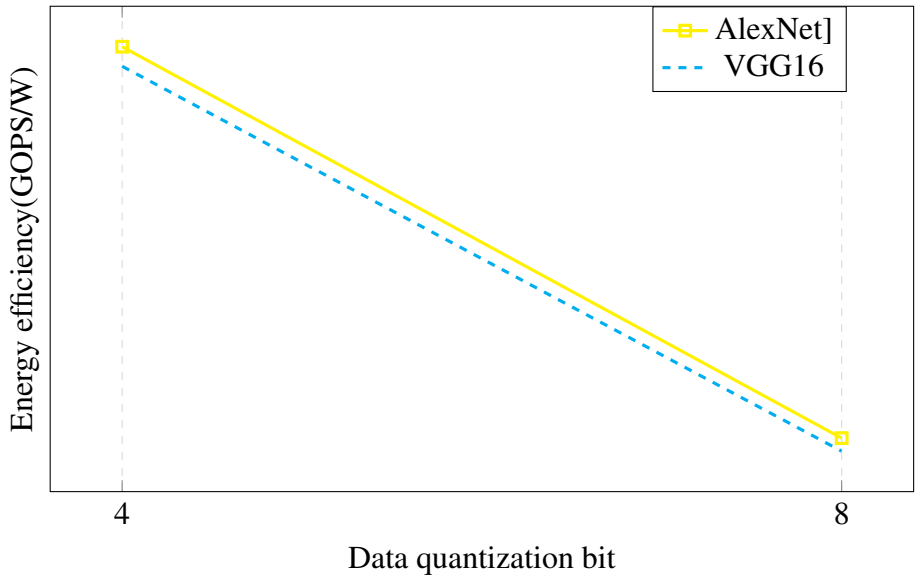

Figure 5.9: Energy efficiency with different data quantization bit on GPU

Aside from improving performance, low-bit-width quantization technique in neural networks improve power efficiency. As we have discussed in the subsection of computation throughput, it reduces memory access costs by enabling high memory cache usage and increasing compute efficiency. Using low-bit quantization can reduce power consumption and save significant energy. Low-bit-width quantization uses less energy and enhances compute efficiency, resulting in lower power consumption.

Generly, among all the data quantization bit as observed from the above figures, the low bit width based designs demonstrate exceptionally good speed and energy efficiency. this indicates which extremely low bit-width is a likely solution for high performance. However, the extremely low bit width that has accuracy reduction.

5.3.3 Resource Utilization

The following table 5.1, show that the trained CNN on FPGA resource usage. During training, CNN on FPGA consumes huge computational resources. We trained our models on the de1_SoC board before changing any parameters, and the resource usage is illustrated in the below figure.

Data quantization	ALUTs	DSP	FFS	M10K
32 bit	139913	85	172677	497
16 bit	108313	76	144798	422
8 bit	88712	64	126918	346
4 bit	74511	52	91158	187

Table 5.1: Resource Usage CNN Training

Our design, as stated in Chapter 4, includes two major variables: the number of computing units and the number of SIMD. The replication of full data paths is the total number of computing units, that also controls the balance among both resource usage. Additionally, a computing unit may be composed one or more processing elements depending on our design choice. Having more processing elements per compute unit can significantly raise the data processing speed. While the numbers of SIMD processing elements is a design choice that also allows for contiguous memory retrieval which can enhance memory utilization efficiency. In the design, we were using a static configuration number of an SIMD and computing units, which allowed us to work with restricted resources of FPGA broad. We investigate how well the number of computing units and SIMD impact the De1 Soc-based board's resource utilization. In order to achieve the maximum resource utilization of our design, we configured the SMID as fixed as well as varied the number of computing units.

Figures 5.10 to 5.13 show the resource usage, such as the amount of ALUTs, FFs, M10K blocks, as well as DSP blocks for various parameter configurations. similar studies with the VGG16 CNN model have been carried out. The consequences of resource use are depicted in Figures 5.14 to 5.17.

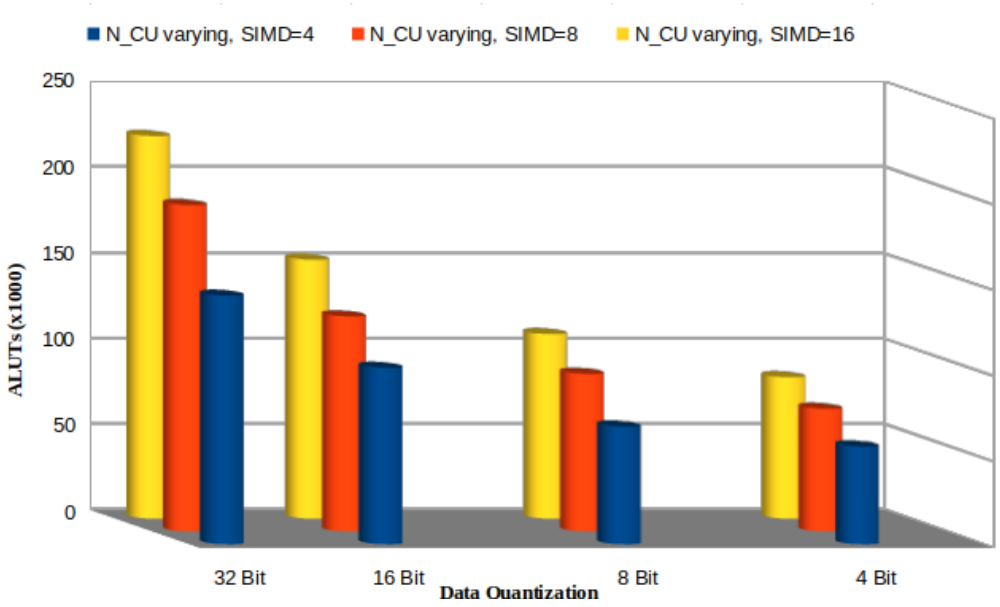

Figure 5.10: ALUTs Utilization on DE1_SoC board for AlexNet

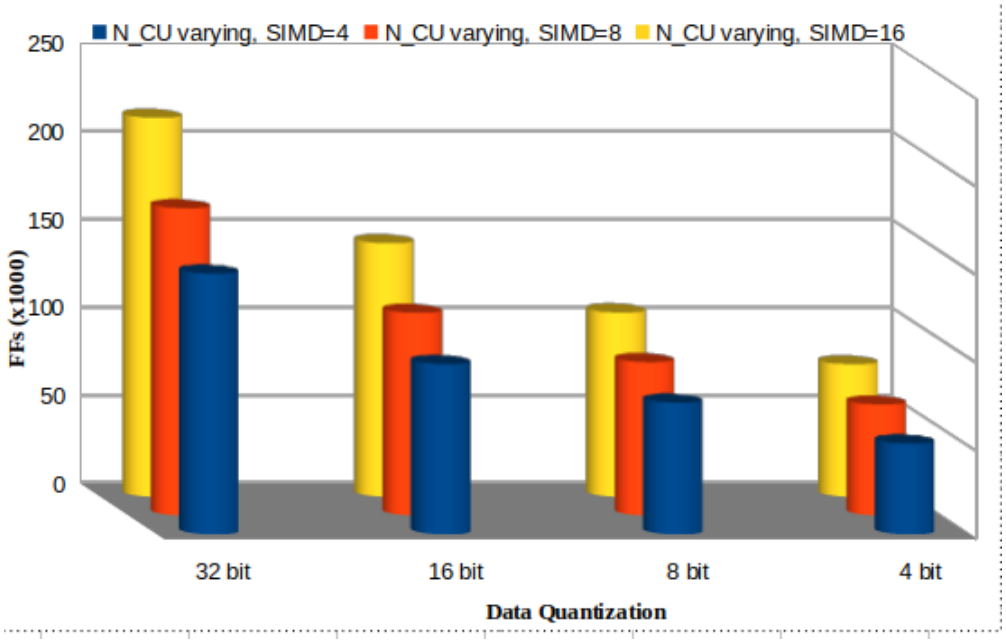

Figure 5.11: FFs Utilization on DE1_SoC board for AlexNet

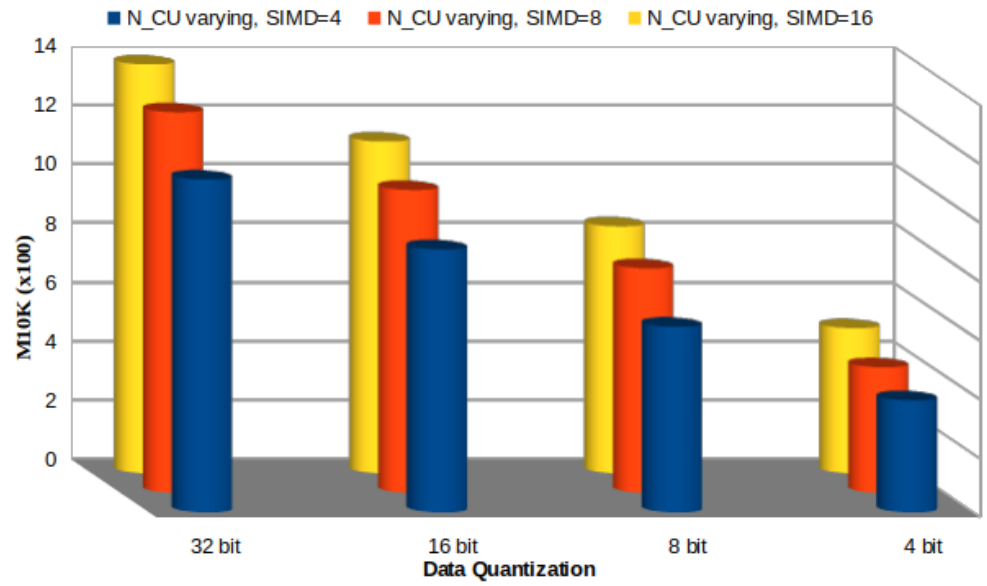

Figure 5.12: M10K Utilization on DE1_SoC board for AlexNet

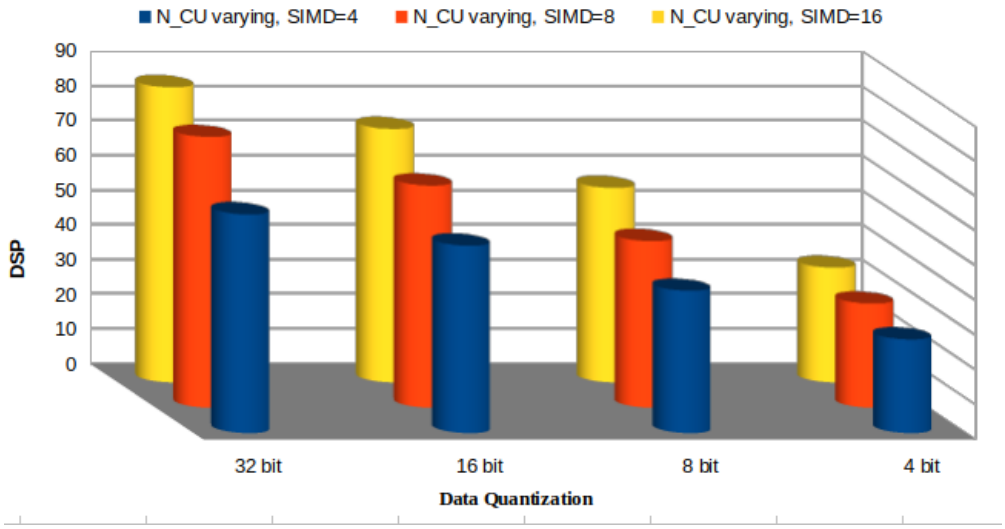

Figure 5.13: DSP Utilization on DE1_SoC board for AlexNet

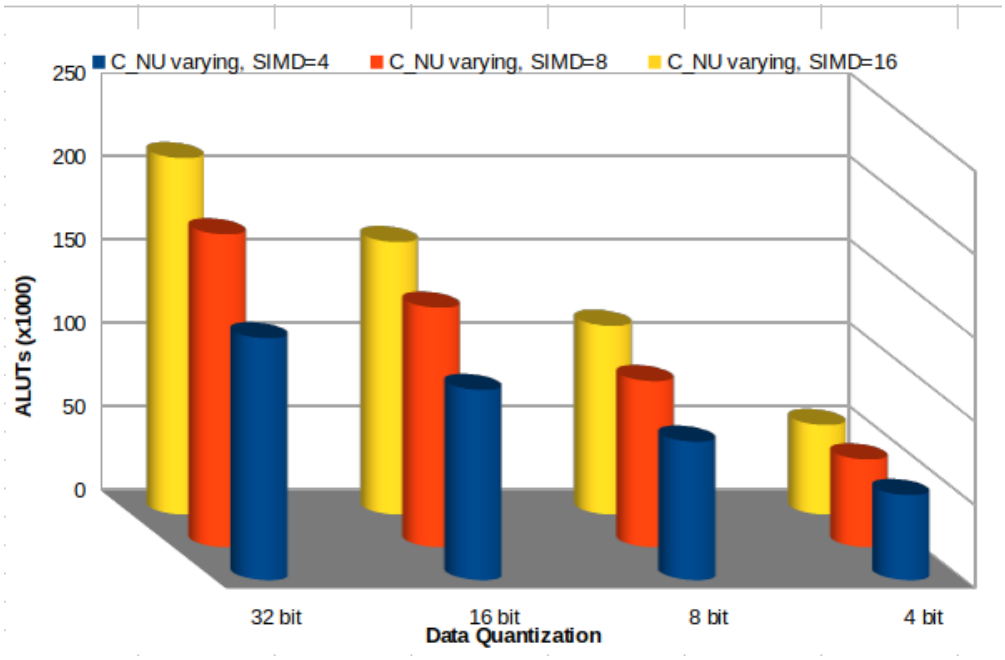

Figure 5.14: ALUTs Utilization on DE1_SoC board for VGG16 model

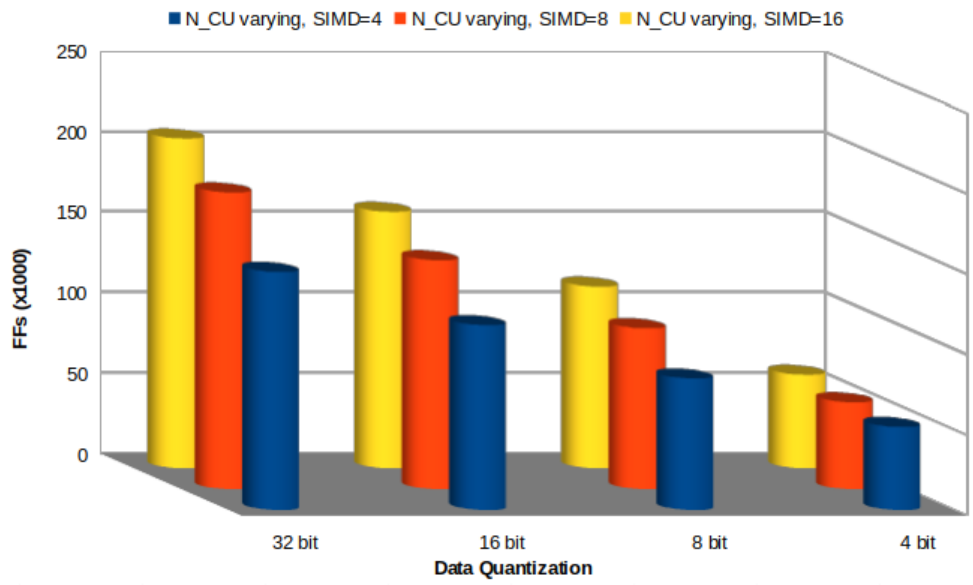

Figure 5.15: FFs Utilization on DE1_SoC board for VGG16 model

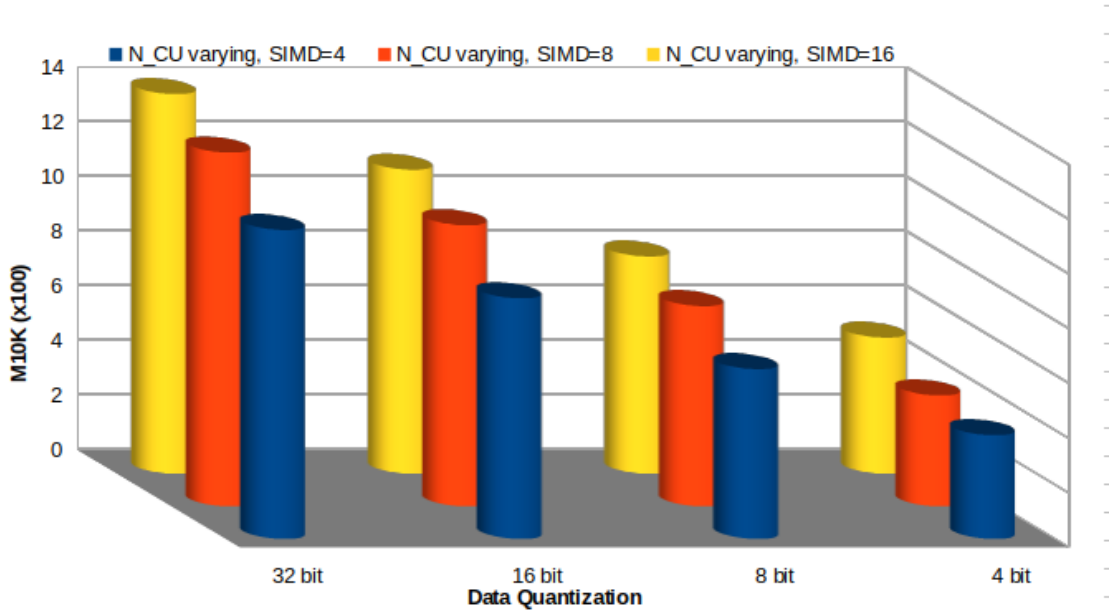

Figure 5.16: M10K Utilization on DE1_SoC board for VGG16 model

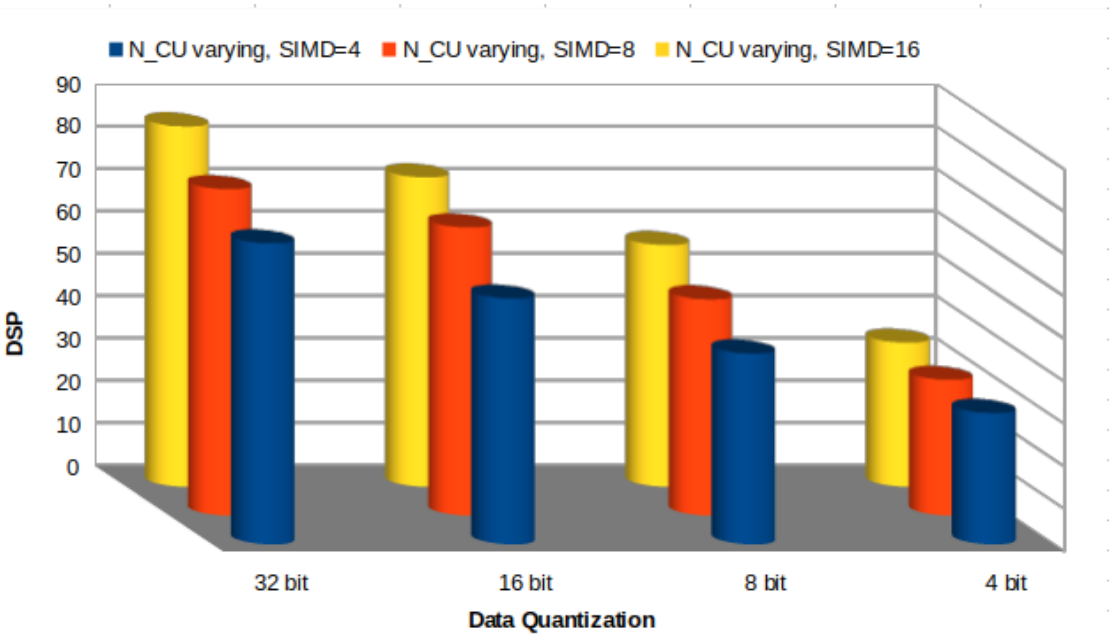

Figure 5.17: DSP Utilization on DE1_SoC board for VGG16 Model

Generally, When both parameters grow, there is also a growth in resource usage. while the number of computing units does have a greater effect on resource utilization than that of the number of SIMD. Whenever these two parameters are increased up-to sixteen, it is simple to see that a resource usage is increase. When the configuration of computing is equal to sixteen and SMID sixteen, the compilation is successful compiled both for fixed point quantization and 32 bit floating point. Once, when the computing unit is more than sixteen, and also when the SIMD number is greater than sixteen, a similar compilation failure occurs.

5.4 Power Measurement

The power consumption is an important element in hardware accelerator performance. The power drain on one of the devices tells us how hard it is working and how power-intensive the design would be. This is especially essential for evaluating deep learning applications for hardware accelerators, where power consumption is a major consideration. We measure performance and power consumption by using the Perf performance analysis tool for Linux. The idle CPU-only system absorbs 50.70 W before the FPGA accelerator board is installed on the system. When using Caffe tools to run AlexNet and VGG16 models, the average power utilization starts to rise to 109.2 W. Whenever a DE1 SoC-based FPGA board is properly configured, the idle power consumption rises to 63.40 W. Throughout CNN kernel implementation, the overall power usage of the hardware acceleration rises to 78.2 W by averages. Thereby, a power use for running a CNN framework on the a DE1 SoC-based board is (78.20-50.70) = 27.5W. Table 5.2 summarizes the power consumption results.

Table 5.2: Consumption of Power

Platform	CPU	CPU as well as DE1_SoC Board
System Idel Power (W)	50.70	63.40
System Execution Power(W)	109.2	78.20
Accelerator Power(W)	58.5	27.5

5.5 Compared to Previous Works

In this subsection, we would first compare our implementations with previous FPGA research. The following is a comparison with similar designs focused on other high performance computing platforms, such as CPUs and also GPUs.

5.5.1 Compared to an FPGA-based design

Table 5.3 contrasts the proposed models' efficiency to that of a number of other recent FPGA-based CNN design features. To determine the throughput, divide the total floating-point numbers or fixed point operations through the entire execution time, and then use GOP/S as a unit for floating-point as well as fixed-point operations in our implementation design. Zhang et al. [29] implemented a convolution layer which obtained 61.62 GOPS again for single precision floating point design. Similarly, the work by Yufei Ma et al. [13] reported 134.1 GOPS and 117.3 GOPS on a convolution layer for AlexNet and NIN model respectively, while they achieved the overall performance 114.5 GOPS and 117.1 GOPS for AlexNet and NiN model respectively. Our throughput from 8 bit fixed point on DE1_SoC for AlexNet model 180.50 GOPS and also for VGG16 model 196.50 GOPS on DE1_SoC. Our work gained 1.53x more throughput over the work by Naveen Suda et al. [5] with only using 85 DSP blocks. Furthermore, Our design outperforms the RTL design in [13] by 1.576x on the different boards, demonstrating that OpenCL-based designs can compete with RTL designs. When compared to other designs, our DE1 SoC design has had the highest throughput, and there is still room for improvement.

Table 5.3: Compared of Previous FPGA Works

Metrics	FPGA	Method	Model	Precision	DSP Usage	Conv (GOPS)	Overall (GOPS)
[13](a)	Stratix-VGA7	RTL	AlexNet	8-bit fixed	193	134.1	114.5
[13](b)	Stratix-VGA7	RTL	NiN	8-bit fixed	249	117.3	117.3
[29]	Virtex 7 VX485T	Xilinx HLS	AlexNet	32 bit float	2,240	61.62	N/A
[5](a)	Stratix-V GSD8	Intel OpenCL	VGG	8-16 bit fixed	N/A	136.5	117.8
[5](b)	Stratix-V GXA7	Intel OpenCL	VGG	8-16 bit fixed	N/A	N/A	117.8
[49]	Zynq XC7Z045	RTL	VGG16	16-bit fixed	780	187.80	136.97
[18]	Stratix-V GXA7	RTL	AlexNet	8-16 bit fixed	256	134.1	114.5
This work (A)	DE1_SoC	Intel OpenCL	AlexNet	8-bit fixed	70	198.50	180.5
This work (B)	DE1_SoC	Intel OpenCL	VGG16	8-bit fixed	72	190.60	173.40
This work (a)	DE1_SoC	Intel OpenCL	AlexNet	4-bit fixed	57	213.9	203.75
This work (b)	DE1_SoC	Intel OpenCL	VGG16	4-bit fixed	61	225.40	196.50

5.5.2 Compared to Other HPC Platform Designs

Table 5.3 shows the performance contrast among both our FPGA implementations as well as the work on CPU and GPU. To implement our designs on the CPU as well GPU, we have choose a fixed point design on two CNN models.

We as well introduce energy consumption as both a measure for evaluation, which would be the ratio of throughput to power consumption (GOPS/Watt). In terms of

throughput, the GPU is the best alternative, preceded by one FPGA design. Power usage, on either hand, is an important measure to take into account in modern digital design. The GPU absorbs 3.709X so much energy than that of the FPGA, and the FPGA is 22.613x more efficient than the CPU.

Table 5.4: Compare with Other devices

Platform	CPU	GPU	FPGA
	Intel® Core™ i5-4300	AMD Radeon (TM) R5 M330	DE1_SoC
Technology	22 nm	28 nm	28 nm
Power(Watt)	58.5	94.50	18.5
Throughput (GOPS)	28.50	280.60	203.75
Energy Efficiency (GOPS/W)	0.487	2.969	**11.013**

Our research found that FPGAs outperform CPUs in terms of computation because they support shift registers, which enable for a reducing required on-chip memory when compared to CPUs. Furthermore, the flexibility provided by FPGAs enables us to efficiently allocate on-chip memory between different parallel blocks. Based on our findings, FPGAs can outperform CPUs in terms of performance and power efficiency, while GPUs can outperform FPGAs in terms of energy efficiency and limiting improvement to only power efficiency in most cases. We stated that it is due to a difference in computing performance as well as the external memory bandwidth between FPGAs as well as GPUs. In general, our design's significant concentration of optimization and flexibility in terms of performance parameter tuning enabled us to achieve maximum FPGA resource usage at increasing the operation.

Chapter 6

Conclusion and Recommendation

6.1 Conclusion

In this work, we show a training and classification of a deep neural network that use the Intel, FPGA OpenCL SDK. To determine the best design requirements to speed up the CNN model for training while using constrained FPGA resources, we proposed a design space exploration methodology for energy efficiencies and resource utilization. We implemented CNN models such as AlexNet and VGG on the DE1 SoC FPGA board using the proposed approach, and we gained higher performance when compared to earlier work. As we compared with the other platform, the CNN training on FPGA consume less power consumption and training time. Our findings indicated that FPGAs could obtain greater power or energy efficiency than GPUs, which typically restrict improvement only to power efficiency. We noted that it is mainly due to the huge difference in maximum compute performance as well as the external memory bandwidth between FPGAs and GPUs.

Generally, our designs with 4 fixed point bits achieved 203.75 GOPS for the AlexNet model and 196.50 GOPS for the VGG16 model on the Terasic DE1-SoC broad, in addition to being 22.613X and 3.709X more energy efficient than the CPU and GPU, respectively. As stated in Chapter 1, the purpose of this research paper would be to answer two major questions:

- Can we use high-level synthesis tools to design a parameterized CNN accelerator?
- Using the Intel FPGA SDK OpenCL to accelerate CNN, can it outperform other platforms?

As conclusions, we have answered our first question by having the capabilities to allow choosing the two CNN models, SIMD, and parallel computing.We responded to the second question by comparing our work to recently proposed work of CNN on Hardware implementation, which is more power saving than CPU and GPU.

6.2 Recommendation

This research paper identifies the following direction as future work.

1. This work paper employs space exploration methods that are specifically designed to maximize resource usage and performance. We intend to investigate sparse convolution or Winograd transformation algorithms in future research to minimize the number of computations needed mostly by convolution layer and enhance the performance of this work.

2. This work paper, we have used the same kernel for both fully connected and convolution layers to minimize the number of memory access and to ensure a contiguous memory access pattern. In future work, to release the load on the bandwidth of the external memory , we plan to integrate a slide window technique based on data buffer with vectorization of data input and output.

3. This work was used to implement the experiments on an FPGA board using the Terasic DE1-SoC Development Kit (DK). In future work, we plan to implement different FPGA platforms such as Stratix-V GXA7, Arria-10 GX1150 and Cyclone 10 GX.

Bibliography

[1] K. et al, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[3] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Proceedings of the ieee conference on computer vision and pattern recognition," 2015.

[4] J.-H. Lin, T. Xing, R. Zhao, Z. Zhang, M. Srivastava, Z. Tu, and R. K. Gupta, "Binarized convolutional neural networks with separable filters for efficient hardware acceleration," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2017, pp. 27–35.

[5] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016, pp. 16–25.

[6] R. Tapiador, A. Rios-Navarro, A. Linares-Barranco, M. Kim, D. Kadetotad, and J.-s. Seo, "Comprehensive evaluation of opencl-based convolutional neural network accelerators in xilinx and altera fpgas," *arXiv preprint arXiv:1609.09296*, 2016.

[7] W. Ding, Z. Huang, Z. Huang, L. Tian, H. Wang, and S. Feng, "Designing efficient accelerator of depthwise separable convolutional neural network on fpga," *Journal of Systems Architecture*, vol. 97, pp. 278–286, 2019.

[8] J. Zhang and J. Li, "Improving the performance of opencl-based fpga accelerator for convolutional neural network," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 25–34.

[9] M. Mathieu, M. Henaff, and Y. LeCun, "Fast training of convolutional networks through ffts," *arXiv preprint arXiv:1312.5851*, 2013.

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

[11] S. Tatsumi, M. Hariyama, M. Miura, K. Ito, and T. Aoki, "Opencl-based design of an fpga accelerator for phase-based correspondence matching," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. The Steering Committee of The World Congress in Computer Science, Computer . . . , 2015, p. 90.

[12] H. Li, "Acceleration of deep learning on fpga," 2017.

[13] Y. Ma, N. Suda, Y. Cao, S. Vrudhula, and J.-s. Seo, "Alamo: Fpga acceleration of deep learning algorithms with a modularized rtl compiler," *Integration*, vol. 62, pp. 14–23, 2018.

[14] G. et al, "[dl] a survey of fpga-based neural network inference accelerators," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 12, no. 1, pp. 1–26, 2019.

[15] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient inference," *arXiv preprint arXiv:1611.06440*, 2017.

[16] H. Sim and J. Lee, "A new stochastic computing multiplier with application to deep convolutional neural networks," in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[17] P. Gysel, M. Motamedi, and S. Ghiasi, "Hardware-oriented approximation of convolutional neural networks," *arXiv preprint arXiv:1604.03168*, 2016.

[18] Y. Ma, N. Suda, Y. Cao, J.-s. Seo, and S. Vrudhula, "Scalable and modularized rtl compilation of convolutional neural networks onto fpga," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2016, pp. 1–8.

[19] A. Rahman, J. Lee, and K. Choi, "Efficient fpga acceleration of convolutional neural networks using logical-3d compute array," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016, pp. 1393–1398.

[20] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *International conference on artificial neural networks*. Springer, 2014, pp. 281–290.

[21] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE, 2014, pp. 10–14.

[22] R. Girshick, "Fast r-cnn," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1440–1448.

[23] J. H. Ko, B. Mudassar, T. Na, and S. Mukhopadhyay, "Design of an energy-efficient accelerator for training of convolutional neural networks using frequency-domain computation," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2017, pp. 1–6.

[24] R. DiCecco, G. Lacey, J. Vasiljevic, P. Chow, G. Taylor, and S. Areibi, "Caffeinated fpgas: Fpga framework for convolutional neural networks," in *2016 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2016, pp. 265–268.

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

[26] K. Abdelouahab, M. Pelcat, J. Serot, C. Bourrasset, and F. Berry, "Tactics to directly map cnn graphs on embedded fpgas," *IEEE Embedded Systems Letters*, vol. 9, no. 4, pp. 113–116, 2017.

[27] G. Natale, M. Bacis, and M. D. Santambrogio, "On how to design dataflow fpga-based accelerators for convolutional neural networks," in *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2017, pp. 639–644.

[28] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 45–54.

[29] M. Zhang, L. Li, H. Wang, Y. Liu, H. Qin, and W. Zhao, "Optimized compression for implementing convolutional neural networks on fpga," *Electronics*, vol. 8, no. 3, p. 295, 2019.

[30] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 65–74.

[31] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[32] A. Shawahna, S. M. Sait, and A. El-Maleh, "Fpga-based accelerators of deep learning networks for learning and classification: A review," *IEEE Access*, vol. 7, pp. 7823–7859, 2018.

[33] W. Stallings, *Computer organization and architecture: designing for performance*. Pearson Education India, 2003.

[34] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, *et al.*, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, 2015.

[35] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming," *Parallel Computing*, vol. 38, no. 8, pp. 391–407, 2012.

[36] *The path to acceleration: Altera bets on opencl*, Nov 2012 Nov 2012. [Online]. Available: http://www.eejournal.com/archives/articles/20121106-opencl/

[37] A. Fitsum, "Acceleration and energy reduction of object detection on mobile graphics processing unit," Ph.D. dissertation, Addis Ababa University, 2019.

[38] B. Liu, D. Zou, L. Feng, S. Feng, P. Fu, and J. Li, "An fpga-based cnn accelerator integrating depthwise separable convolution," *Electronics*, vol. 8, no. 3, p. 281, 2019.

[39] L. Howes and A. Munshi, "The opencl specfification https://www.khronos.org/registry/opencl/specs/opencl- 1.2.pdf," [Online access 31 Oct 2019] 2013. [Online]. Available: https://www.khronos.org/registry/Opencl/Specs/Opencl-1.2.pdf

[40] H. Kung, B. McDanel, and S. Q. Zhang, "Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 821–834.

[41] Z. Liu, Y. Dou, J. Jiang, Q. Wang, and P. Chow, "An fpga-based processor for training convolutional neural networks," in *2017 International Conference on Field Programmable Technology (ICFPT)*. IEEE, 2017, pp. 207–210.

[42] W. Zhao, H. Fu, W. Luk, T. Yu, S. Wang, B. Feng, Y. Ma, and G. Yang, "F-cnn: An fpga-based framework for training convolutional neural networks," in *2016 IEEE 27Th international conference on application-specific systems, architectures and processors (ASAP)*. IEEE, 2016, pp. 107–114.

[43] Y. Sun, N. B. Agostini, S. Dong, and D. Kaeli, "Summarizing cpu and gpu design trends with product data," *arXiv preprint arXiv:1911.11313*, 2019.

[44] V. NVIDIA, "Tesla k20 gpu accelerator board specification," 2013.

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

[46] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 609–622.

[47] H. Li, M. Bhargav, P. N. Whatmough, and H.-S. P. Wong, "On-chip memory technology design space explorations for mobile deep neural network accelerators," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.

[48] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 243–254.

[49] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang, "Angel-eye: A complete design flow for mapping cnn onto embedded fpga," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 35–47, 2017.

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

[51] L. Jiao, C. Luo, W. Cao, X. Zhou, and L. Wang, "Accelerating low bit-width convolutional neural networks with embedded fpga," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2017, pp. 1–4.

[52] D. Wang, J. An, and K. Xu, "Pipecnn: an opencl-based fpga accelerator for large-scale convolution neuron networks," *arXiv preprint arXiv:1611.02450*, 2016.

[53] A. Prost-Boucle, A. Bourge, F. Pétrot, H. Alemdar, N. Caldwell, and V. Leroy, "Scalable high-performance architecture for convolutional ternary neural networks on fpga," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2017, pp. 1–7.

[54] H. Nakahara, H. Yonekawa, H. Iwamoto, and M. Motomura, "A batch normalization free binarized convolutional deep neural network on an fpga," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 290–290.

[55] H. Nakahara, T. Fujii, and S. Sato, "A fully connected layer elimination for a binarizec convolutional neural network on an fpga," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2017, pp. 1–4.

[56] "Terasic de1soc user manual http://www.ee.ic.ac.uk/pcheung/teaching/ee2_digital/de1-soc_user_manual.pdf," *http://www.ee.ic.ac.uk/pcheung/teaching/ee2_digital/DE1-SoC_User_manual.pdf*, 2014.

Appendix A

kernelvoid Input_Krn(unsigned char datad1, unsigned char datad2, unsigned short datad1d2, unsigned char weightd1, unsigned char weightd2, unsigned short weightd3, unsigned short weightdiv, unsigned char weightd12, unsigned int weightd123, unsigned char conv_x, unsigned char stride, unsigned char padding, unsigned char split, unsigned char grnum_x, unsigned char grnum_y, unsigned char grsize_x, unsigned int grsize_xyz, unsigned char size_x, unsigned char size_y, unsigned int size_xyz, // Data Ports global line_data *restrict bottom, global channel_cv *restrict weights,

global volatile channel_scal *restrict bias

) // Input Data, Weights and Bias line_data data_vec; channel_cv data_ch_vec;

channel_cv weight_ch_vec; channel_scal bias_ch_in; shortdata_offset = 0; unsigned short gpnum_x, gpnum_y, out_idx_z;

unsigned short gpnum_$x_buf, gpnum_y_buf, out_idx_z_buf$;

unsigned char output$_i dx_d 1 = 0, output_idx_d2 = 0$;

unsigne short output_idx_d3=0; unsigned char x, y; unsigned short z; unsigned char gp_item_idx_x=0; unsigned short feature_idx_d1,

feature_idx_d2; unsigned short feature_idx_d3; unsigned int loopbound; unsigned char flag; unsigned char load_weight_flag = 1; char load_feature_flag = 1; local line_data S_buffer[2][BUF_SIZE];

local channel_cv weight_buffer[WEIGHT_BUF_SIZE];

for(z=0; z<weightd3/input_size; z++) for(y=0; y<size_y; y++) for(x=0; x<size_x; x++)

feature_idx_d1 = x; feature_idx_d2 = y; feature_idx_d3 = z; if((feature_idx_d1>=padding && feature_idx_d1<datad1+padding) && (feature_idx_d2>=padding && feature$_i dx_d 2 < datad2 + padding$))

data_vec=

bottom[data_offset$\times$datad1d2 + feature_idx_d3*datad1d2

+ (feature_idx_d2-padding)$\times$

datad1 + (feature_idx_d1-padding)];

else #pragma unroll for(unsigned char j=0; j<input$_s ize; j++$)

data$_vec.data[j] = CZERO; S_buffer[0][z\times$size_y$\times$size_x + y$\times$size_x + x] =

data_vec; x = 0; y = 0; z = 0; if(grnum_x==1 && grnum_y==1) gpnum_x_buf = 0;

gpnum_y_buf = 0; else if(grnum$_x$ ==

1)$gpnum\_x\_buf = 0; gpnum\_y\_buf = 1; elsegpnum\_x\_buf = 1; gpnum\_y\_buf = 0;$

out_idx_z_buf = 0; // reset counters gpnu$_x = 0; gpnum_y = 0; out_idx_z = 0;$

unsigned int conv_x_idx = 0; unsigned int out_idx_xyz = 0; unsigned int

item$_loop_cnt =$

$0; unsignedintLoopBound; unsignedintLastLoopBound; unsignedintTotalLoopBound;$

if(stride>=weightd1 || stride>=weightd2) LoopBound = size_xyz/input_size;

elseLoopBound =

(weightd123*CONV$_G P_S IZE_Y * CONV_G P_S IZE_X / input_s ize); LastLoopBound =$

$size_x >= grsize_x ? (size_x yz/input_s ize) : (grsize_x yz/input_s ize);$

TotalLoopBound =

$((\text{grnum}_x - 1) * LoopBound + LastLoopBound) * grnum_y * weightdiv;$

for(unsigned int total$_cnt = 0; total_cnt < TotalLoopBound; total_cnt++)$

if(item$_loop_cnt == 0)$

if(split==0) data$_offset = 0; elseif(out_idx_zbuf < (weightdiv >> 1))data_offset =$

$0; elsedata_offset = weightd3/input_s ize;$

flag = out$_idx_xyz0x01; load_feature_flag = 1; if(gpnum_x == grnum_x - 1)$

loopbound = LastLoopBound; else loopbound = LoopBound;

// buffer loading operations if(load$_feature_flag == 1)$

feature$_idx_d1 = x + gpnum_xbuf * CONV_G P_S IZE_X * stride; feature_idx_d2 =$

$y + gpnum_ybuf * CONV_G P_S IZE_Y * stride; feature_idx_d3 = z;$

// buffer if((feature$_idx_d1 >= paddingfeature_idx_d1 <$

$datad1 + padding)(feature_idx_d2 >= paddingfeature_idx_d2 < datad2 + padding))$

data$_vec = bottom[data_offset * datad1d2 + feature_idx_d3 * datad1d2 +$

$(feature_idx_d2 - padding) * datad1 + (feature_idx_d1 - padding)]; else$

pragma unroll for(unsigned char vv=0;

vv<input$_size; vv++)data_vec.data[vv] = CZERO;$

$$S_b uffer[(\ flag)0x01][z * size_y * size_x + y * size_x + x] = data_v ec;$$

// loop counters if((z==weightd3/input$_s$ize $-1)(y == size_y - 1)(x == size_x - 1))$

$$z = 0; \text{load}_f eature_f lag = 0; elseif((y == size_y - 1)(x == size_x - 1))z++;$$

$$\text{if((y==size}_y - 1)(x == size_x - 1))y = 0; elseif(x == size_x - 1)y++;$$

$$\text{if(x==size}_x - 1)x = 0; elsex++;$$

// Load weight into weight buffer

$$\text{if(load}_w eight_f lag == 1)weight_c h_v ec = weights[out_i dx_z * weightd123/input_s ize+$$

$$\text{output}_i dx_d 3 * weightd12 + out put_i dx_d 2 * weightd1 + out put_i dx_d 1];$$

$$\text{weight}_c h_v ec = weight_b uffer[out put_i dx_d 3 * weightd2 * weightd1$$

$$+ \text{output}_i dx_d 2 * weightd1 + out put_i dx_d 1];$$

// grouping in x direction $\text{if(gpnum}_x * CONV_G P_S IZE_X + gp_i tem_i dx_x < conv_x)$

$$\text{if(output}_i dx_d 1 == 0 out put_i dx_d 2 == 0 out put_i dx_d 3 ==$$

$$0) if(load_w eight_f lag == 1)bias_c h_i n = bias[out_i dx_z]; write_c hannel_i ntel(bias_c h, bias_c h_i n);$$

// data $\text{data}_v ec = S_b uffer[flag][out put_i dx_d 3 * size_y * size_x + out put_i dx_d 2 * size_x +$

$$(out put_i dx_d 1 + gp_i tem_i dx_x * stride)]; pragmaunroll for(unsignedcharll = 0; ll <$$

$$NUM; ll++)data_c h_v ec.line[ll] = data_v ec; write_c hannel_i ntel(data_c h, data_c h_v ec);$$

// weight and bias fetcher $\text{weight}_c h_v ec = weight_b uffer[out put_i dx_d 3 * weightd2 *$

$$weightd1 + out put_i dx_d 2 * weightd1 + out put_i dx_d 1];$$

$$\text{write}_c hannel_i ntel(weight_c h, weight_c h_v ec);$$

$$\text{else if((output}_i dx_d 3 == weightd3/input_s ize - 1)(out put_i dx_d 2 ==$$

$$weightd2 - 1)(out put_i dx_d 1 == weightd1 - 1))gp_i tem_i dx_x++;$$

$$\text{if((output}_i dx_d 3 == weightd3/input_s ize - 1)(out put_i dx_d 2 ==$$

$$weightd2 - 1)(out put_i dx_d 1 ==$$

$$weightd1 - 1))out put_i dx_d 3 = 0; load_w eight_f lag = 0; elseif((out put_i dx_d 2 ==$$

$$weightd2 - 1)(out put_i dx_d 1 == weightd1 - 1))out put_i dx_d 3++;$$

$$\text{if((output}_i dx_d 2 == weightd2 - 1)(out put_i dx_d 1 == weightd1 - 1))out put_i dx_d 2 =$$

$$0; elseif(out put_i dx_d 1 == weightd1 - 1)out put_i dx_d 2++;$$

$$\text{if(output}_i dx_d 1 == weightd1 - 1)out put_i dx_d 1 = 0; elseout put_i dx_d 1++;$$

$$\text{if(item}_l oop_c nt == loopbound - 1)$$

$$\text{item}_{l}oop_{c}nt = 0; out_{i}dx_{x}yz++;$$

$$\text{if}((\text{out}_{i}dx_{zb}uf == weightdiv-1)(gpnum_{yb}uf == grnum_{y}-1)(gpnum_{xb}uf == grnum_{x}-1))out_{i}dx_{zb}uf = 0; elseif((gpnum_{yb}uf == grnum_{y}-1)(gpnum_{xb}uf == grnum_{x}-1))out_{i}dx_{zb}uf++;$$

$$\text{if}((\text{gpnum}_{yb}uf == grnum_{y}-1)(gpnum_{xb}uf == grnum_{x}-1))gpnum_{yb}uf = 0; elseif(gpnum_{xb}uf == grnum_{x}-1)gpnum_{yb}uf++;$$

$$\text{if}(\text{gpnum}_{xb}uf == grnum_{x}-1)gpnum_{xb}uf = 0; elsegpnum_{xb}uf++;$$

// sending data to the conv kernel $\text{if}((\text{out}_{i}dx_{z} == weightdiv-1)(gpnum_{y} == grnum_{y}-1)(gpnum_{x} == grnum_{x}-1))out_{i}dx_{z} = 0; elseif((gpnum_{y} == grnum_{y}-1)(gpnum_{x} == grnum_{x}-1))out_{i}dx_{z}++; load_{w}eight_{f}lag = 1;$

$$\text{if}((\text{gpnum}_{y} == grnum_{y}-1)(gpnum_{x} == grnum_{x}-1))gpnum_{y} = 0; elseif(gpnum_{x} == grnum_{x}-1)gpnum_{y}++;$$

$$\text{if}(\text{gpnum}_{x} == grnum_{x}-1)gpnum_{x} = 0; elsegpnum_{x}++; elseitem_{l}oop_{c}nt++;$$

$$_{k}ernelvoidConv_{k}rn(unsignedintoutput_{n}um,$$

$$\text{unsigned int conv}_{l}oop_{c}nt, unsignedintcontol,$$

$$\text{unsigned char frac}_{w}, unsignedcharfrac_{d}in,$$

$$\text{unsigned char frac}_{d}out)channel_{c}vmac_{d}ata; channel_{c}vmac_{w}eight;$$

$$\text{channel}_{s}calbias_{c}h_{o}ut; channel_{s}calconv_{c}h_{i}n;$$

$$\text{DHNG bias[NUM]; G}_{T}YPEconv_{o}ut[NUM]; G_{T}YPEline_{a}ccum[NUM];$$

$$\text{G}_{T}YPEaccum[NUM][DEPTH]; G_{T}YPEconv_{s}ign_{e}xten[NUM];$$

$$\text{G}_{T}YPEconv_{w}ith_{r}nd_{b}it[NUM]; G_{T}YPEconv_{s}um_{b}ias[NUM];$$

$$\text{G}_{T}YPEconv_{f}inal[NUM];$$

$$\text{unsigned int conv}_{i}nner_{c}nt = 0;$$

$$\text{for(unsigned int k=0; k<output}_{n}um * conv_{l}oop_{c}nt;$$

$$\text{k++)}$$

$$\text{if(conv}_{i}nner_{c}nt == 0)$$

$$\text{bias}_{c}h_{o}ut = read_{c}hannel_{i}ntel(bias_{c}h);$$

$$\text{pragma unroll for(unsigned char l=0; l<NUM; l++) bias[l] = bias}_{c}h_{o}ut.line[l];$$

$$\text{pragma unroll for(unsigned int p=0; p<DEPTH; p++) accum[l][p] =}$$

$$\text{MASK}_A CCUMCZERO;$$

$$\text{mac}_d ata = read_c hannel_i ntel(data_c h); mac_w eight = read_c hannel_i ntel(weight_c h);$$

pragma unroll for(unsigned char l=0; l<NUM; l++)

$$\text{line}_a ccum[l] = (MASK_A CCUMaccum[l][DEPTH-1]) +$$

$$(MASK_M ULTmac(mac_d ata.line[l], mac_w eight.line[l]));$$

pragma unroll for(unsigned int p=DEPTH-1; p>0; p–) accum[l][p] =

$$\text{MASK}_A CCUMaccum[l][p-1];$$

// update the copy accum[l][0] = $\text{MASK}_A CCUMline_a ccum[l]$;

$$\text{if(conv}_i nner_c nt == conv_l oop_c nt - 1)$$

pragma unroll for(unsigned char l=0; l<NUM; l++) $\text{conv}_o ut[l] = CZERO$;

// accumulate all the partial results pragma unroll for(unsigned i=0; i<DEPTH; i++)

$$\text{conv}_o ut[l] += accum[l][i];$$

$$\text{if(conv}_o ut[l] >= 0) conv_s ign_e xten[l] =$$

$$0x00; elseconv_s ign_e xten[l] = (0xFFFFFFFF >> (frac_w + frac_d in - frac_d out - 1));$$

$$\text{conv}_w ith_r nd_b it[l] = (conv_s ign_e xten[l] | (conv_o ut[l] >>$$

$$(frac_w + frac_d in - frac_d out - 1))) + 0x01;$$

$$\text{if(conv}_w ith_r nd_b it[l] >= 256) conv_s um_b ias[l] = MASK9B0xFF; // =$$

$$255 elseif(conv_w ith_r nd_b it[l] < -256) conv_s um_b ias[l] = MASK9B0x100; // = -256 else$$

$$\text{conv}_s um_b ias[l] = (MASK9Bconv_w ith_r nd_b it[l]) + (bias[l] >>$$

$$(frac_w - frac_d out - 1)) + 0x01; conv_f inal[l] = MASK8B(conv_s um_b ias[l] >>$$

$$0x01); conv_i nner_c nt = 0; elseconv_i nner_c nt++;$$

$$\text{kernel void maxPool(unsigned char conv}_x, unsignedshortconv_x y,$$

$$\text{unsigned char pool}_d 1, unsignedshortpool_d 3, unsignedshortpool_d 12,$$

$$\text{unsigned char pool}_s ize, unsignedcharpool_s tride,$$

$$\text{unsigned char padd}_o ffset, unsignedshortpool_t imes, unsignedshortpool_g roup,$$

$$\text{unsigned short pool}_{yb} ound, unsignedshortloop_b ound,$$

unsigned short

$$\text{load}_d ata_b ound, unsignedshortwrite_b ack_b ound, unsignedcharpoolnum_x,$$

$$\text{unsigned char size}_{x, global volatile channel_s cal * restrict bottom,}$$

$_{g}lobalDHNG*restricttop$

) bool

$\text{pool}_{s}ync = 0; unsignedintbase_{a}ddr; unsignedcharflag; unsignedshortpool_{g}roup_{c}nt;$

unsigned char poolycnt;

unsigned short mloop; unsigned short poolcnt;

unsigned char x; unsigned char y; unsigned char s;

unsigned char cnt; unsigned char fcnt; unsigned char lcnt;

$\text{unsigned short global}_{z}; unsignedshortglobal_{i}ndex_{z}g; unsignedcharglobal_{i}ndex_{z};$

$\text{DHNG shift}_{r}eg[NUM][POOL_{M}AX_{S}IZE]; DHNGtemp_{r}eg0[NUM];$

$\text{DHNG temp}_{r}eg1[NUM]; DHNGtemp_{r}eg2[NUM]; DHNGtemp_{r}eg[NUM];$

$\text{DHNG temp}_{m}ax[NUM];$

$\text{DHNG row}_{r}eg0[NUM][POOL_{G}P_{S}IZE_{X}]$

$_{a}ttribute_{((register));DHNGrow_{r}eg1[NUM][POOL_{G}P_{S}IZE_{X}]_{a}ttribute_{((register));}}$

$\text{DHNG pool}_{f}inal[2][POOL_{G}P_{S}IZE_{X}][NUM];$

$\text{poolycnt = 0; pool}_{g}roup_{c}nt = 0; for(shorti = 0; i < pool_{t}imes; i+$

$+)pool_{s}ync = read_{c}hannel_{i}ntel(poolsyncch); mem_{f}ence(CLK_{C}HANNEL_{M}EM_{F}ENCE);$

// init counters poolcnt = 0; mloop = 0; x = 0; y = 0; s = 0; cnt = 0; fcnt = 0; lcnt = 0;

$\text{for(ushort k=0; k<pool}_{y}bound; k++)flag = poolcnt0x01;$

$\text{base}_{a}ddr = pool_{g}roup_{c}nt * conv_{x}y + pool_{s}tride * conv_{x} * poolycnt + pool_{s}tride *$

$poolcnt * POOL_{G}P_{S}IZE_{X};$

$\text{if((poolcnt < poolnum}_{x})(mloop <$

$\text{load}_{d}ata_{b}ound))if(x > pool_{s}ize - 1)cnt++;$

pragma unroll for(uchar ll=0; ll<NUM; ll++) if(

$\text{(poolcnt*POOL}_{G}P_{S}IZE_{X} * pool_{s}tride + x) < conv_{x})$

$\text{shift}_{r}eg[ll][0] = bottom[base_{a}ddr + y * conv_{x} + x].line[ll];$

$\text{if((x == pool}_{s}ize - 1)||(cnt == pool_{s}tride))temp_{r}eg0[ll] = shift_{r}eg[ll][0];$

$\text{temp}_{r}eg1[ll] = shift_{r}eg[ll][1]; temp_{r}eg2[ll] = shift_{r}eg[ll][2];$

$\text{else temp}_{r}eg0[ll] = CZERO; temp_{r}eg1[ll] = CZERO; temp_{r}eg2[ll] = CZERO;$

$\text{if(pool}_{s}ize == 3)temp_{r}eg[ll] = pool_{m}ax(temp_{r}eg0[ll], temp_{r}eg1[ll]);$

$$\text{temp}_max[ll] = pool_max(temp_reg2[ll], temp_reg[ll]); if(y ==$$

$$0) row_reg0[ll][s] = temp_max[ll]; elseif(y == 1) row_reg1[ll][s] =$$

$$\text{pool}_max(temp_max[ll], row_reg0[ll][s]); elsepool_final[flag][s][ll] =$$

$$\text{pool}_max(temp_max[ll], row_reg1[ll][s]);$$

$$\text{else temp}_max[ll] = pool_max(temp_reg1[ll], temp_reg0[ll]); if(y ==$$

$$0) row_reg0[ll][s] = temp_max[ll]; elsepool_final[flag][s][ll] = pool_max(temp_max[ll], row_reg0[ll][s]);$$

$$\text{pragma unroll for(uchar}$$

$$\text{p=POOL}_MAX_SIZE - 1; p > 0; p--) shift_reg[ll][p] = shift_reg[ll][p-1];$$

$$\text{if((x == pool}_size - 1) || (cnt == pool_stride)) s++;$$

$$\text{if(cnt == pool}_stride) cnt = 0;$$

$$\text{if((y == pool}_size - 1)(x == x - 1)) y = 0; elseif(x == x - 1) y++; if(x == x - 1) x =$$

$$0; elsex++; if(x == 0) s = 0;$$

$$\text{if((poolcnt > 0) (mloop <}$$

$$\text{write}_back_bound)) if(((poolcnt - 1) * POOL_GP_SIZE_X + fcnt) < pool_d1)$$

$$\text{global}_z = pool_groupcnt * NUM + lcnt; global_index_zg =$$

$$(global_z - padd_offset)/input_size; global_index_z = (global_z - padd_offset)$$

$$\text{if((global}_z - padd_offset) < pool_d3 global_z >=$$

$$padd_offset) top[global_index_zg * pool_d12 * input_size + poolycnt * pool_d1 *$$

$$\text{input}_size + ((poolcnt - 1) * POOL_GP_SIZE_X + fcnt) * input_size + global_indexz] =$$

$$pool_final[!flag][fcnt][lcnt];$$

$$\text{if((cnt == POOL}_GP_SIZE_X - 1)(lcnt == NUM - 1)) fcnt = 0; elseif(lcnt ==$$

$$NUM - 1) fcnt++; if(lcnt == NUM - 1) lcnt = 0; elselcnt++;$$

$$\text{if((poolcnt == poolnum}_x)(mloop == loop_bound - 1)) poolcnt = 0; elseif(mloop ==$$

$$loop_bound - 1) poolcnt++; if(mloop == loop_bound - 1) mloop = 0; elsemloop++;$$

$$\text{if((poolcnt == pool}_group - 1)(poolycnt == pool_d1 - 1)) poolcnt =$$

$$0; elseif(poolycnt == pool_d1 - 1) poolcnt++; if(poolycnt ==$$

$$pool_d1 - 1) poolycnt = 0; elsepoolycnt++; kernelvoidBack_krn($$

$$\text{unsigned int output}_num, unsignedintconv_loop_cnt,$$

$$\text{unsigned int contol, unsigned char frac}_w, unsignedcharfrac_din,$$

unsigned char frac$_dout$)$channel_cvmac_data$;$channel_cvmac_weight$;

channel$_scalbias_ch_out$;$channel_scalconv_ch_in$;$DHNGbias[NUM]$;

G$_TYPEconv_out[NUM]$;$G_TYPEline_accum[NUM]$;$G_TYPEaccum[NUM][DEPTH]$;

G$_TYPEconv_sign_exten[NUM]$;$G_TYPEconv_with_rnd_bit[NUM]$;$G_TYPEconv_sum_bias[NUM]$;

G$_TYPEconv_final[NUM]$;

unsigned int conv$_innerc_nt = 0$;

for(unsigned int k=0; k<output$_num * conv_loop_cnt; k++$)

if(conv$_innerc_nt == 0$)

bias$_ch_out = read_channel_intel(bias_ch)$;

pragma unroll for(unsigned char l=0; l<NUM; l++) bias[l] = bias$_ch_out.line[l]$;

pragma unroll for(unsigned int p=0; p<DEPTH; p++) accum[l][p] =

MASK$_ACCUMCZERO$;

mac$_data = read_channel_intel(data_ch)$;$mac_weight = read_channel_intel(weight_ch)$;

pragma unroll for(unsigned char l=0; l<NUM; l++)

line$_accum[l] = (MASK_ACCUMaccum[l][DEPTH-1]) +$

$(MASK_MULTmac(mac_data.line[l], mac_weight.line[l]))$;

pragma unroll for(unsigned int p=DEPTH-1; p>0; p–) accum[l][p] =

MASK$_ACCUMaccum[l][p-1]$;

// update the copy accum[l][0] = MASK$_ACCUMline_accum[l]$;

kernel void Output$_krn($(

unsigned char outd1, unsigned char outd2,

unsigned short outd3, unsigned short outd1batch, unsigned int outd12batch, unsigned

char batch$_indx_d$1,

unsigned char batch$_indx_d$2,

unsigned char padd$_offset, unsignedcharpool_on, unsignedcharpool_size,$

unsigned char pool$_stride$,

global DHNG *restrict top) unsigned char index$_z$;$unsignedshortindex_zgroup$;

unsigned int addr; bool poolonsignal=1; channel$_scaloutput$;

local DHNG buffer[NUM];

unsigned short $\text{out}_i dx_x = 0; unsigned short out_i dx_y = 0; unsigned char l_i dx = 0;$

unsigned short $\text{lnum}_i dx = 0; for(unsigned int i = 0; i <$

$(outd1 * outd2 * (outd3 + 2 * padd_o ffset)); i++) if(l_i dx == 0)$

pragma unroll for(unsigned char ll=0; ll<NUM; ll++) buffer[ll]=output.line[ll];

$\text{if}(\text{pool}_o n ==$

$1) addr = lnum_i dx * outd12batch * NUM + (out_i dx_y + batch_i ndx_d 2 * outd2) * outd1batch * NUM +$

$(\text{out}_i dx_x + batch_i ndx_d 1 * outd1) * NUM + l_i dx; else$

$\text{index}_z = (lnum_i dx * NUM + l_i dx - padd_o ffset)/input_s ize; index_z =$

$(lnum_i dx * NUM + l_i dx - padd_o ffset)$

addr = $\text{index}_z group * outd12batch * input_s ize + (out_i dx_y + batch_i ndx_d 2 * outd2) *$

outd1batch*$\text{input}_s ize + (out_i dx_x + batch_i ndx_d 1 * outd1) * input_s ize + index_z;$

$\text{if}((\text{lnum}_i dx * NUM + l_i dx - padd_o ffset) < outd3 (lnum_i dx * NUM + l_i dx >=$

$padd_o ffset)) top[addr] = buffer[l_i dx];$

$\text{if}(\text{pool}_o n == 1) if((out_i dx_x == outd1 - 1)(out_i dx_y >$

$0)((\text{out}_i dx_y - pool_s ize + 1)\%2 == 0)(l_i dx ==$

$NUM - 1)) write_c hannel_i ntel(poolsyncch,$

poolonsignal);

$\text{if}((\text{lnum}_i dx == (outd3 + 2 * padd_o ffset)/NUM - 1)(out_i dx_y == outd2 -$

$1)(out_i dx_x == outd1 - 1)(l_i dx == NUM - 1)) lnum_i dx = 0; elseif((out_i dx_y ==$

$outd2 - 1)(out_i dx_x == outd1 - 1)(l_i dx == NUM - 1)) lnum_i dx++;$

$\text{if}((\text{out}_i dx_y == outd2 - 1)(out_i dx_x == outd1 - 1)(l_i dx == NUM - 1))$

$\text{out}_i dx_y = 0; elseif((out_i dx_x == outd1 - 1)(l_i dx == NUM - 1)) out_i dx_y++;$

$\text{if}((\text{out}_i dx_x == outd1 - 1)(l_i dx == NUM - 1)) out_i dx_x = 0; elseif(l_i dx ==$

$NUM - 1) out_i dx_x++;$

$\text{if}(\text{l}_i dx == NUM - 1) l_i dx = 0; elsel_i dx++;$

Appendix B

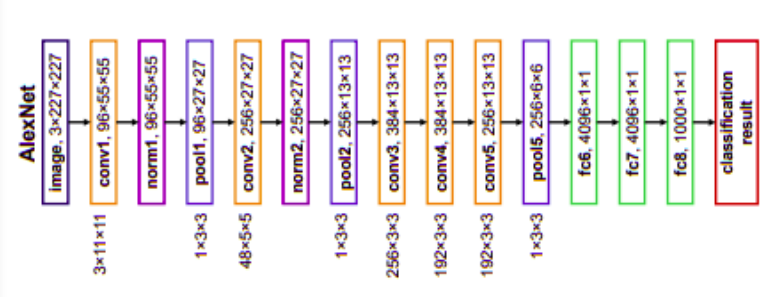

Figure 6.1: AlexNet Hardware configuration to change convolution layer to FC kayer

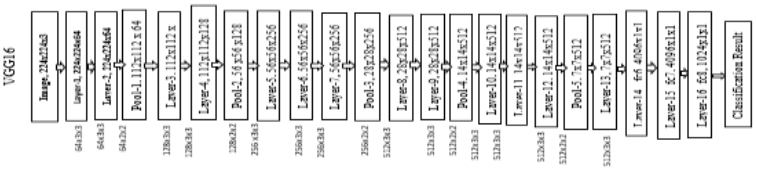

Figure 6.2: VGG-16 Hardware confuguration to change convolution layer to FC kayer