4-14-2017

Acceleration of Deep Learning on FPGA

Huyuan Li
University of Windsor

Acceleration of Deep Learning on FPGA

by

Huyuan Li

A Thesis

Submitted to the Faculty of Graduate Studies

through the Department of Electrical and Computer Engineering

in Partial Fulfillment of the Requirements for

the Degree of Master of Applied Science

at the University of Windsor

Windsor, Ontario, Canada

2017

Acceleration of Deep Learning on FPGA

by

Huyuan Li

APPROVED BY:

Feb 14, 2017

AUTHOR'S DECLARATION OF ORIGINALITY

ABSTRACT

In recent years, deep convolutional neural networks (ConvNet) have shown their popularity in various real world applications. To provide more accurate results, the state-of-the-art ConvNet requires millions of parameters and billions of operations to process a single image, which represents a computational challenge for general purpose processors. As a result, hardware accelerators such as Graphic Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs), have been adopted to improve the performance of ConvNet. However, GPU-based solution consumes a considerable amount of power and a traditional RTL design on FPGA requires tedious development that is very time-consuming.

In this work, we propose a scalable and parameterized end-to-end ConvNet design using Intel FPGA SDK for OpenCL. To validate the design, we implement VGG 16 model on two different FPGA boards. Consequently, our designs achieve 306.41 GOPS on Intel Stratix A7 and 318.94 GOPS on Intel Arria 10 GX 10AX115. To the best of our knowledge, this outperforms previous FPGA-based accelerators. Compared to the CPU (Intel Xeon E5-2620) and a mid-range GPU (Nvidia K40), our design is 24.3X and 1.7X more energy efficient respectively.

DEDICATION

To God, my parents and my wife, for all their love, kindness and support.

ACKNOWLEDGEMENTS

First of all, I would like to thank my supervisor Dr. Mohammed Khalid, for his patient guidance, encouragement, and advice during my research and study at the University of Windsor. His willingness to offer me so much of his time and intellect is the main reason of this thesis completion. It was a real privilege and an honor for me to share of his exceptional knowledge and his deep insights.

I would like to appreciate my thesis committee members Dr. Tirupati Bolisetti and Dr. Huapeng Wu for generously offering their time, encouragement and perceptive comments.

I would like to acknowledge Ryan Tang for his technical assistance and suggestions on selecting research topics. I would also like to thank Mohamed Abdinur and Mohammad Momen for the interesting discussion on the research problems.

I would also express my appreciation to my colleagues and friends, Chunyu Mao, Akilan Thangarajah, Liyuan Liu, Pengzhao Song, Xuhui Zhong, Brian & Jennifer Moffatt, David & Cristina Lahoud and Sam & Sue Mannina for their generous support and help.

A special gratitude goes to my beloved family for their support. I thank my parents for their wholehearted love, financial support, and encouragement. I appreciate my wife for her endless assistance and inspiration in the work and life. Without them, this thesis would not be completed.

Finally, I would like to extend my sincere thanks to my dear God, who let me through all the difficulties with His unconditional love for me.

Huyuan Li

TABLE OF CONTENTS

AUTHOR'S DECLARATION OF ORIGINALITY iii

ABSTRACT iv

DEDICATION v

ACKNOWLEDGEMENTS vi

LIST OF TABLES x

LIST OF FIGURES xi

LIST OF ACRONYMS xiii

1 Introduction **1**

1.1 Motivation 1

1.2 Objectives 2

1.3 Contributions 2

1.4 Organization 3

2 Background and Related Work **5**

2.1 FPGA Architecture 5

2.2 High-Level Synthesis 6

2.3 Overview of OpenCL 7

2.3.1 Platform Model 8

2.3.2 Execution Model 8

2.3.3 Memory Model 10

2.3.4 Programming Model 11

2.4 Intel FPGA SDK for OpenCL 12

2.5 Supervised Machine Learning 13

2.5.1 Feedforward Neural Networks 14

2.5.2 Convolutional Neural Networks 16

2.6 Related Work 20

2.6.1 Convolutional Neural Networks on GPUs 21

2.6.2 Convolutional Neural Networks on ASIC 21

2.6.3 Convolutional Neural Networks on FPGAs 22

2.7 Summary 23

3 Design Flow and Optimization for Intel FPGA SDK for OpenCL 24

3.1 OpenCL Design flow 24

3.2 Optimization Strategies in OpenCL for FPGA 26

3.2.1 Parallelism Scheme 26

3.2.2 Throughput-oriented Optimizations 29

3.2.3 Communication Optimization 30

4 Methodology for Creating Optimized OpenCL Specification for ConvNet Algorithm 32

4.1 Analysis of Computational Complexity and Space Complexity 32

4.2 Data Quantization 34

4.3 Data Arrangement and Pre-processing 36

4.4 Converting FC Layers to Convolutional Layers 37

4.5 Optimized FPGA Hardware Synthesis from OpenCL Specification . . 38

4.5.1 Parallelism in ConvNet 38

4.5.2 Data Reuse in ConvNet 40

4.5.3 Kernel Design 40

4.6 Summary 45

5 Results **46**

5.1 Experimental Setup 46

5.2 Performance Analysis of the Proposed Design 48

5.2.1 The Accuracy Comparison 48

5.2.2 The Resource Utilization and Performance 49

5.2.3 Power Measurement 57

5.3 Comparison With Previous Research ConvNet 57

5.3.1 Performance Comparison with FPGA-based Design 58

5.3.2 Performance Comparison with Other HPC Platform-based Design 58

6 Conslusion **61**

6.1 Future Research 62

REFERENCES **63**

VITA AUCTORIS **70**

LIST OF TABLES

2.1 Overview of High-Level Synthesis Tools 7

2.2 OpenCL memory model for FPGAs 13

4.1 The Complexity of VGG 16 Model 35

5.1 The Comparison of FPGA Accelerators 47

5.2 The Accuracy Comparison 48

5.3 Performance Model on Throughput-oriented Configuration 56

5.4 Power Consumption of Accelerators 57

5.5 The Comparison with Previous FPGA Works 59

5.6 The Comparison with Other Platforms 60

LIST OF FIGURES

2.1 Intel Arria 10 Architecture 6

2.2 OpenCL Platform Model 8

2.3 OpenCL Execution Model in 2D Range 10

2.4 OpenCL Memory Model 11

2.5 Feedforward Neural Networks 14

2.6 The Comparison between Three Popular Activation Functions 15

2.7 The Example of Convolutional Neural Network for Image Classification 16

2.8 The Example of Pooling Layer 19

3.1 Intel FPGA Design Flow 25

3.2 Pictorial Depiction of Accelerated System 26

3.3 The Comparison between Data Parallelism and Loop Parallelism . . . 28

4.1 The Data Layout Comparison Between ”NHW” and ”HWN”. 37

4.2 Task Parallelism in Proposed Design 40

4.3 Decomposition of 3D Convolution 41

4.4 The Block Diagram of Proposed System 45

5.1 ALUTs Utilization on Stratix A7 with Different Groups of N_{CU} and N_{SIMD} 50

5.2 LEs Utilization on Stratix A7 with Different Groups of N_{CU} and N_{SIMD} 50

5.3 Register Utilization on Stratix A7 with Different Groups of N_{CU} and N_{SIMD} 51

5.4 M20K Utilization on Stratix A7 with Different Groups of N_{CU} and N_{SIMD} 51

5.5 DSP Utilization on Stratix A7 with Different Groups of N_{CU} and N_{SIMD} 52

5.6 Throughput on Stratix A7 with Different Groups of N_{CU} and N_{SIMD} 52

5.7 ALUTs Utilization on Arria 10 with Different Groups of N_{CU} and N_{SIMD} 53

5.8 LEs Utilization on Arria 10 with Different Groups of N_{CU} and N_{SIMD} 54

5.9 Register Utilization on Arria 10 with Different Groups of N_{CU} and N_{SIMD} 54
5.10 M20K Utilization on Arria 10 with Different Groups of N_{CU} and N_{SIMD} 55
5.11 DSP Utilization on Arria 10 with Different Groups of N_{CU} and N_{SIMD} 55
5.12 Throughput on Arria 10 with Different Groups of N_{CU} and N_{SIMD} . 56

LIST OF ACRONYMS

ALMs	Adaptive Logic Modules
AOC	Intel Offline Compiler
API	Application program interface
ASIC	Application Specific Integrated Circuits
BSP	Board Support Package
CAD	Computer Aided Design
ConvNet	Convolutional Neural Networks
CU	Compute Unit
CUDA	Compute Unified Device Architecture
FC	Fully-Connected
FFT	fast Fourier Transform
FIFO	First-In, First-Out
FLOPS	Floating Point Operations Per Second
FPGA	Field Programmable Gate Arrays
GCC	GNU Compiler Collection
GOPS	Giga oprations per sec
GPU	Graphic Processing Units
HDL	Hardware Description Languages
HLS	High-level Synthesis

HMC	Hybrid Memory Cube
HPC	High Performance Computing
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
K40	Kepler 40
LEs	Logic Elements
LOC	Lines of Code
LRN	Local Response Normalization
MLPs	Multilayer Perceptrons
OpenCL	Open Computing Language
ReLU	Rectified Linear Unit
RTL	Register transfer level
SDRAM	Synchronous Dynamic Random Access Memory
SIMD	Single Instruction Multiple Data
SPMD	Single Program Multiple Data
SRAM	Static Random Access Memory
TDP	Thermal Design Power
W	Watt

1 Introduction

1.1 Motivation

In recent years, artificial intelligence and deep learning have shown their utility and effectiveness in solving many real world computation intensive problems. At the center of this resurgence is the artificial neural network, more specifically the convolutional neural network (ConvNet) [1]. The ConvNet has been demonstrated as an effective method for various applications including image [2] and video classification [3], document processing [4] and speech recognition [5]. To provide more accurate results, the state-of-the-art ConvNet requires millions of parameters and billions of operations to process a single image, which represents a computational challenge for general purpose processors. As a result, hardware accelerators such as Graphic Processing Units (GPU) [6][7][8], Field Programmable Gate Arrays (FPGA) [9][10], and Application Specific Integrated Circuits (ASIC) [11][12], have been utilized to improve the throughput of the ConvNet.

Among these accelerators, GPUs are the most widely used to improve both training and classification process of ConvNet, thanks to their high throughput and memory bandwidth. However, GPUs consume a considerable amount of power which is another important evaluation metric in the modern digital systems. ASIC design, on the other hand, has achieved high throughput with low power consumption by assigning dedicated resources and customizing memory hierarchy. But the development time and cost is significantly high compared to other solutions. As an alternative, FPGA-based accelerators provide high throughput, low power consumption, and reconfigurability at a reasonable price.

Three [9][10][13] of FPGA-based ConvNet designs have proven that it is feasible to implement end-to-end inference of deep ConvNet. Two [10][13] of these designs used the traditional Register transfer level (RTL) implementation, which requires tedious design and debugging process resulting in longer time-to-market. The introduction of high-level synthesis (HLS) enables developers to program FPGAs using high-level language such as C and C++ to accelerate the design process. While HLS tools provide developers easy-to-use programming model for FPGA, being able to fully utilize the fine-grained architecture to achieve peak performance presents challenges. In[14], the author showed several optimization skills on FPGA which leads up to 20X speedup compared with baseline design in N-body application. Although Suda et al. [9] have shown promising results on ConvNet by using HLS method, there is still more parallelism can be achieved.

1.2 Objectives

Motivated by the issues mentioned in the section above, the objective of this thesis is to answer two main question as follows:

- How to optimize HLS design when targeting Intel FPGAs?
- Can FPGAs outperform other HPC platforms on deep ConvNet by using Intel FPGA SDK for OpenCL?

1.3 Contributions

The contributions of this work are as follows:

- Due to the different architecture from other parallel accelerators, optimization strategy on FPGA is unique which is an open problem. In this work, we present a systematic tuning scheme for HLS tools targeting Intel FPGAs.
- We present complexity analysis on deep ConvNet based upon VGG 16 model which requires over 30 billion operations using 138 million parameters.

- We propose a scalable and parameterized ConvNet design using Intel FPGA SDK for OpenCL targeting Intel FPGA. As a result, our architecture achieved overall throughput of 306.41 GOPS on Intel Stratix A7 and 318.94 GOPS on Intel Arria 10 GX 10AX115 when implementing VGG 16 model.
- We compare our results with recent FPGA-based works and designs from other HPC platforms. To the best of our knowledge, this is the best result reported on FPGA accelerators on ConvNet. Compared to the CPU (Intel Xeon E5-2620) and a mid-range GPU (Nvidia K40), our design is 24.3X and 1.7X more energy efficient respectively.

1.4 Organization

The rest of this thesis is organized as follows:

Chapter 2 provides background information on FPGAs, High-level Synthesis, OpenCL, and supervised machine learning and related work on Convolutional Neural Networks implementation. We first provide a brief overview of FPGA architecture followed by an introduction to FPGA design method called High-level Synthesis. Then we give a basic knowledge of design tools used in this work, OpenCL framework and Intel FPGA SDK for OpenCL. We also provide the background on supervised machine learning, including two typical widely used neural network architectures: Feedforward Neural Networks and Convolutional Neural Networks. Finally, we discusses related work by reviewing state-of-the-art literature. It consists of three sections, where we present the most recent progress of Convolutional Neural Networks on GPUs, ASIC, and FPGAs.

In Chapter 3, we discuss the design flow and optimization schemes of the design tool used in this work. We begin by a discussion on OpenCL design flow, followed by optimization strategies on OpenCL design for FPGA, including parallelism optimization, throughput optimization, and communication optimization.

In Chapter 4, we present the methodology used in this work. It starts with analyzing computational complexity and space complexity of the architecture. It also provides the discussion on data quantization and data arrangement to improve the performance. Then, we describe detailed hardware design of the proposed Convolutional Neural Network architecture.

Chapter 5 provides the testing and evaluation results of the proposed system. It begins with the summary of experimental setup including software and hardware information. Then the performance of the proposed design is analyzed in terms of accuracy, resource utilization, throughput, and power consumption. Finally, we compare our design with contemporary FPGA-based implementation as well as the work on other HPC platforms.

Chapter 6 summarizes the results obtained by this research and gives suggestions on future work.

2 Background and Related Work

Recently, the utilization of many-core architecture is popular in the HPC area to meet ever-increasing computation demand. Compared to other systems, general purpose GPU is widely chosen, due to the programming simplicity as well as the combination of instruction and data parallelism[15]. In [16], the author shows that to develop faster and more energy-efficient architectures, low level architecture like memory organization and interconnect topology needs to meet algorithmic requirements. Also, it has been estimated that half the lifetime cost of HPC platforms is devoted to electrical power consumption[17]. For these reasons, FPGAs will be favorable in the HPC domain, as they provide reconfigurable hardware resources and low power consumption. The following sections introduce FPGA architecture, the Intel FPGA SDK for OpenCL and the accelerator used in this work.

2.1 FPGA Architecture

A Field Programmable Gate Array (FPGA) is a large integrated circuit that can be used to create custom logic functions and perform specific tasks as a digital circuit. While ASIC outperforms FPGA in terms of throughput and power consumption[18], FPGA development is much more cost effective and fast.

The modern FPGA consists of two parts: fine-grained programmable logic blocks including adaptive logic modules (ALMs) and coarse-grained functional logic components such as memory blocks, DSP blocks, communication blocks and soft-core processor. Xilinx and Altera (Acquired by Intel) are the current FPGA market leaders, who control over 70% of the market[19]. The layout of different hardware resources on Intel Arria 10 FPGA is shown in the Fig. 2.1.

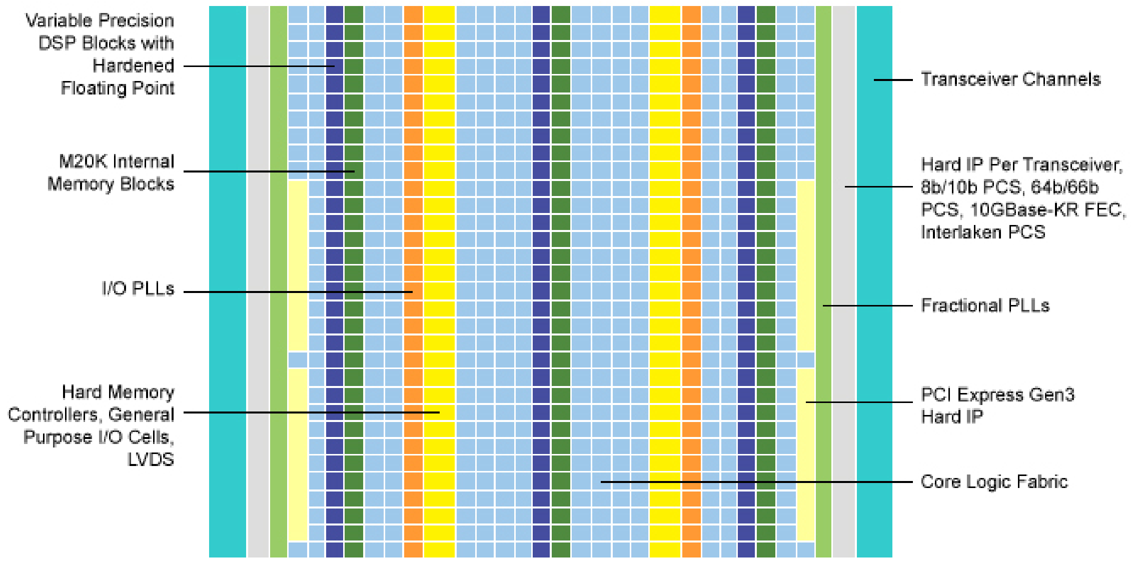
Variable Precision DSP Blocks with Hardened Floating Point
M20K Internal Memory Blocks
I/O PLLs
Hard Memory Controllers, General Purpose I/O Cells, LVDS
Transceiver Channels
Hard IP Per Transceiver, 8b/10b PCS, 64b/66b PCS, 10GBase-KR FEC, Interlaken PCS
Fractional PLLs
PCI Express Gen3 Hard IP
Core Logic Fabric

Fig. 2.1: Intel Arria 10 Architecture, taken from [20]

The FPGA implementation conventionally needs to describe hardware at Register transfer level (RTL) or even at the gate level using Hardware Description Languages (HDL) such as Verilog and VHDL. These HDL models can be synthesized and placed and routed on the target FPGA board by Computer Aided Design (CAD) tools. Unlike traditional software programming language C/C++ or Java, HDL design requires extensive hardware knowledge and tedious debugging process. These challenges have caused engineers to embrace CPU and GPU based solutions over FPGA implementation.

2.2 High-Level Synthesis

High-level synthesis (HLS) is a methodology that provides optimized hardware synthesis from high-level programming language specifications such as C/C++ and System C. HLS tools allow designers to use a software program to specify the target system functionality, enabling them to exploit hardware advantages without building up hardware expertise. Several commercial and academic HLS CAD tools are currently available. Table 2.1 lists some of these HLS tools. The Intel SDK for OpenCL is used in this research for targeting Intel FPGAs.

Table 2.1: Overview of High-Level Synthesis Tools, adopted from [21]

Compiler	Input	Output	Owner	License
CHC	C subset	VHDL/Verilog	Altium	Commercial
CtoS	System C	VHDL/Verilog	Cadence	Commercial
Synphony C	C/C++	VHDL/Verilog System C	Synopsys	Commercial
LegUP	C	Verilog	U. Toronto	Academic
Bambu	C	Verilog	PoliMi	Academic
Intel FPGA SDK for OpenCL	C/C++ with OpenCL	Verilog	Intel	Commercial
Vivado HLS	C/C++ System C	VHDL/Verilog System C	Xilinx	Commercial

2.3 Overview of OpenCL

Open Computing Language (OpenCL) is an industry standard framework for programming heterogeneous parallel systems, which consists of one or more CPUs, GPUs, DSPs, and FPGAs. OpenCL specifies a programming standard based on C99 and a set of Application Program Interface (API). Compared to Compute Unified Device Architecture (CUDA), OpenCL provides functional portability for different devices. OpenCL is an open source maintained by Khronos Group and supported by a variety of companies including Intel, AMD, Apple, ARM Holdings, Creative Technology, IBM, Imagination Technologies, Nvidia, Qualcomm, Samsung, Vivante, Xilinx, and ZiiLABS[22].

OpenCL framework has four models that will be discussed below.

2.3.1 Platform Model

A single *platform* consists of a *host* and one or more *devices* as shown in Fig. 2.2. The host is usually a CPU which is responsible for runtime control over devices. Any device under the OpenCL platform is a combination of one or more *compute units*, which are broken down into *processing units*. The actual computing takes place on processing units.

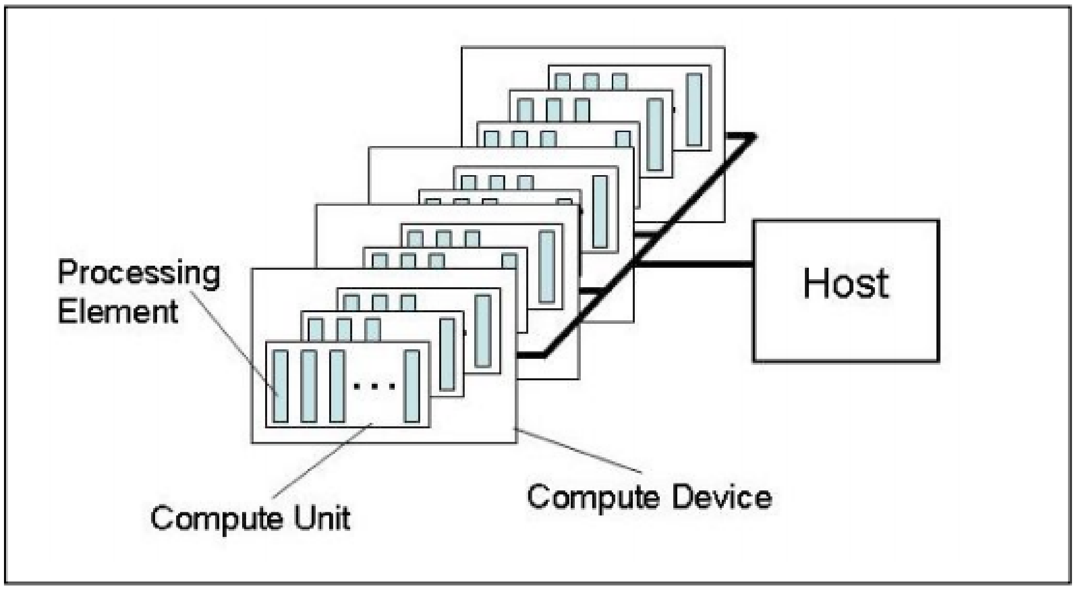

Fig. 2.2: OpenCL Platform Model, taken from [23]

2.3.2 Execution Model

OpenCL program also needs two parts: host code and kernel code. Host code is a general C/C++ code with some API for managing the *program objects*, *memory objects* and the *kernels* in a *context* through *command queues*. Kernel codes contain the core computational part of the application which executes on the devices.

- ***Context***:

  The context is created for one or more devices, containing all necessary information for targeting devices.

- ***Program Objects***:

  The program is including the binary implementation of the kernels, providing a dynamic library for multiple kernels during runtime.

- ***Memory Objects***:

  Memory objects are input and output data for kernels, which are used to transfer data between the host and the devices. The details of memory will be illustrated in the following section.

- ***Command Queue***:

  In order to maintain the execution of commands within the host, command queue is needed. There are three kinds of commands: Memory commands for transferring memory, Kernel commands for launching kernels and Synchronization commands for assigning manual synchronization point in the host codes.

- ***Work Items and Work Groups***:

  Many instances of the kernel are executed in parallel, and each instance is called *work-item*. Work-items are grouped in a multi-dimensional space, which can be one, two or three dimensions. In each dimension, they are recognized by their index, namely *global IDs* that is unique throughout the index space. Each work-item execute the same code on different data. Work-items are organized in another multi-dimensional collection called *work-group*. Each work-group is assigned a *group ID*, and within it, the work-item has a *local ID*.

  Fig. 2.3 illustrates how the work-items are mapped in a 2D range space of index. In the example, we have 10x10 work-items which are evenly divided into four work-groups. The work-group with the index of (0,1) is shown on the right, which consists of 5x5 work-items.

  OpenCL ensures that all work-items in a work-group execute concurrently. However, work-items from different work-groups cannot be guaranteed to run at the same time.

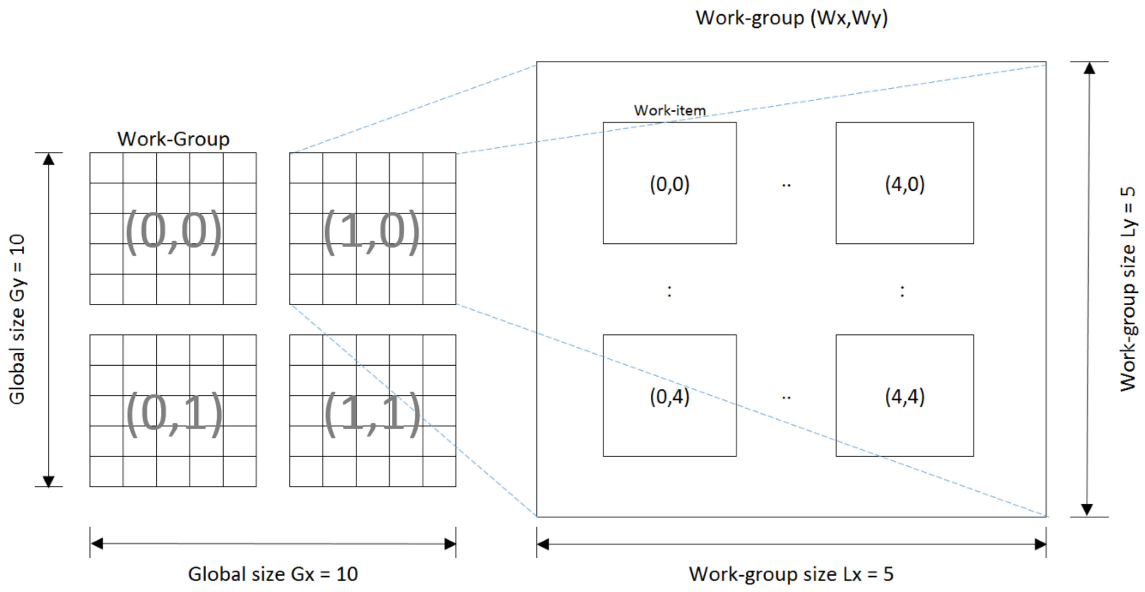

Fig. 2.3: OpenCL Execution Model in 2D Range

2.3.3 Memory Model

The four memory types that are available in the OpenCL framework are illustrated in Fig. 2.4 and are as follows:

- **Global Memory**:

  A memory region that is visible and accessible to the host and all work-items in the devices for read/write purpose.

- **Constant Memory**:

  This subset of memory is a read-only global memory that stores constant data during a kernel execution. The constant memory access is faster than common global memory, as constant data is copied onto on-chip cache before launching the kernel.

- **Local Memory**:

Local memory is accessible for work-items within the same work-group, which enables work-items to communicate with each other within the work-group.

- **Private Memory**:

  The memory region is only visible for a single work-item.

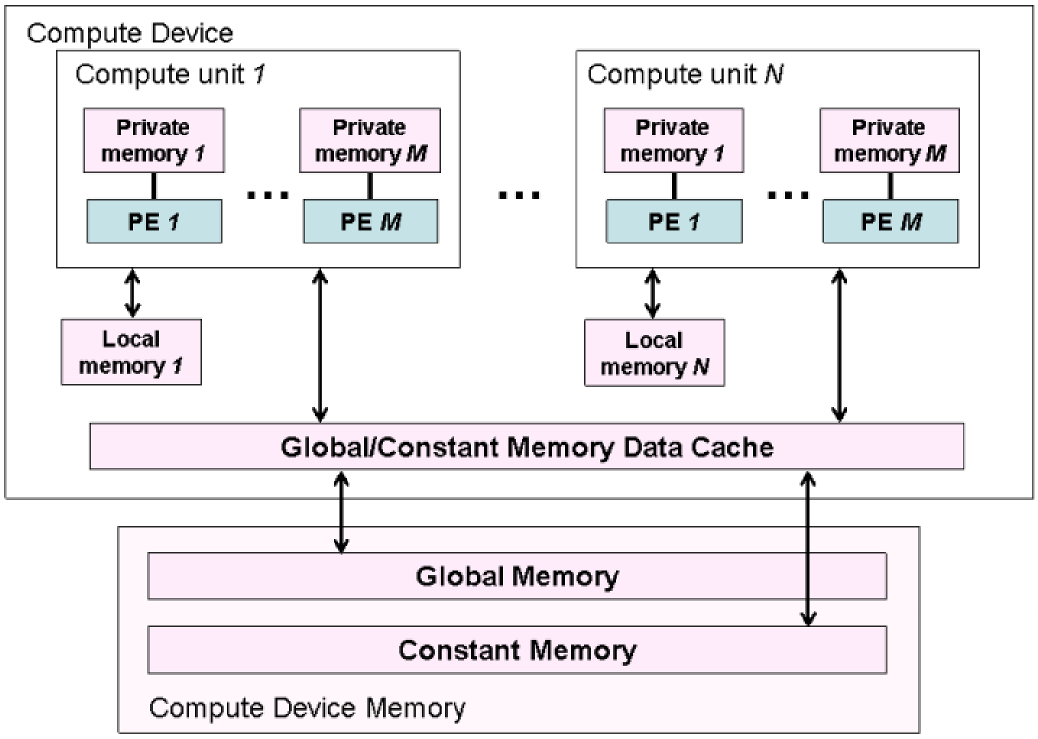

Fig. 2.4: OpenCL Memory Model, taken from [23]

2.3.4 Programming Model

There are two kinds of parallel programming models in the OpenCL: *data parallelism* and *task parallelism*. In data parallel model, each work-item processes different elements of a data set concurrently according to its global ID. Such data parallelism can be further classified as Single Program Multiple Data(SPMD) and Single Instruction Multiple Data(SIMD). In a task parallelism, a large number of kernels with a single work-item execute at the same time. In GPU programming, data parallelism is preferable over task parallelism due to its fixed architecture, while in FPGA design, both models can lead to high throughput.

2.4 Intel FPGA SDK for OpenCL

As one of HLS tools, Intel OpenCL SDK provides a high-level abstraction for FPGA programming. For CPUs or GPUs, a parallel program is compiled to fit von Neumann fixed architecture according to a sequence of instructions. Every calculation requires fetching instructions and transferring data between register files and memory system, which leads to inefficiency. Intel OpenCL SDK solution, in contrast, generates a highly customizable architecture. In this paradigm hardware resources are tailored to the algorithm being executed. Thus customized hardware can perform faster and more power-efficiently than von Neumann processors [24].

For the memory system in Intel OpenCL SDK, global memory is mapped to external memory in the FPGA system, which can be DDR3 synchronous dynamic random access memory (SDRAM), DDR2 SDRAM, DDR SDRAM, and QDR II static random access memory (SRAM) [25]. This types of memory have large capacity but long latency. Constant memory, a special case of global memory, is loaded into cache during the runtime. Local memory resides in embedded region of FPGA, providing much lower latency and higher bandwidth than global memory. This memory region split into N logical banks to handle N request per clock cycle. Lastly, Intel OpenCL SDK solution assigns private memory using FPGA on-chip registers. OpenCL memory model for FPGAs is summarized in Table 2.2

Table 2.2: OpenCL memory model for FPGAs

OpenCL Memory	FPGA memory	DE5a-Net
Global	External	2 x 4GB DDR3-SODIMM
Constant	Cache C	16kB DDR3(default)
Local C	On-Chip memory	M20K
Private	On-Chip register	register

The Intel OpenCL SDK supports the OpenCL 1.0 specification with some flexible requirements and advanced features. As an example of these extensions, we can point to the advantage of using I/O channels and kernel channels, which appeared in OpenCL 2.0. Intel's channel extension allows the transfer of data between work-items in the same kernel or different kernels by means of a first-in, first-out (FIFO) buffer defined with a channel ID and depth. This makes it possible to pass data between kernels without additional synchronization and without host interaction [26]. Additionally, a work-item will stall if it attempts to read from an empty buffer or write to a full channel, and thus channels may also be used as synchronization points between two work-times. [27].

2.5 Supervised Machine Learning

Machine learning algorithms can be widely classified as *unsupervised* or *supervised* by learning experience during a training process. Unsupervised learning algorithms experience a dataset containing many features, then learn useful properties of the structure of the dataset. Using these properties, one can achieve probability distribution or divide the dataset into clusters. On the other hand, supervised learning algorithms experience a dataset containing the feature, but each example is also associated with a *label* or *target* [28]. For example, large data set of images of animals with labels go through training and generates internal adjustable parameters called *weights*. After learning process, animal images can be categorized as a certain label, such as dog or cat, using these parameters. The key difference between these two learning schemes is the availability of desired output, a *target*. Supervised learning is the most common form of machine learning that is including linear regression, classification, and structured output problem.

2.5.1 Feedforward Neural Networks

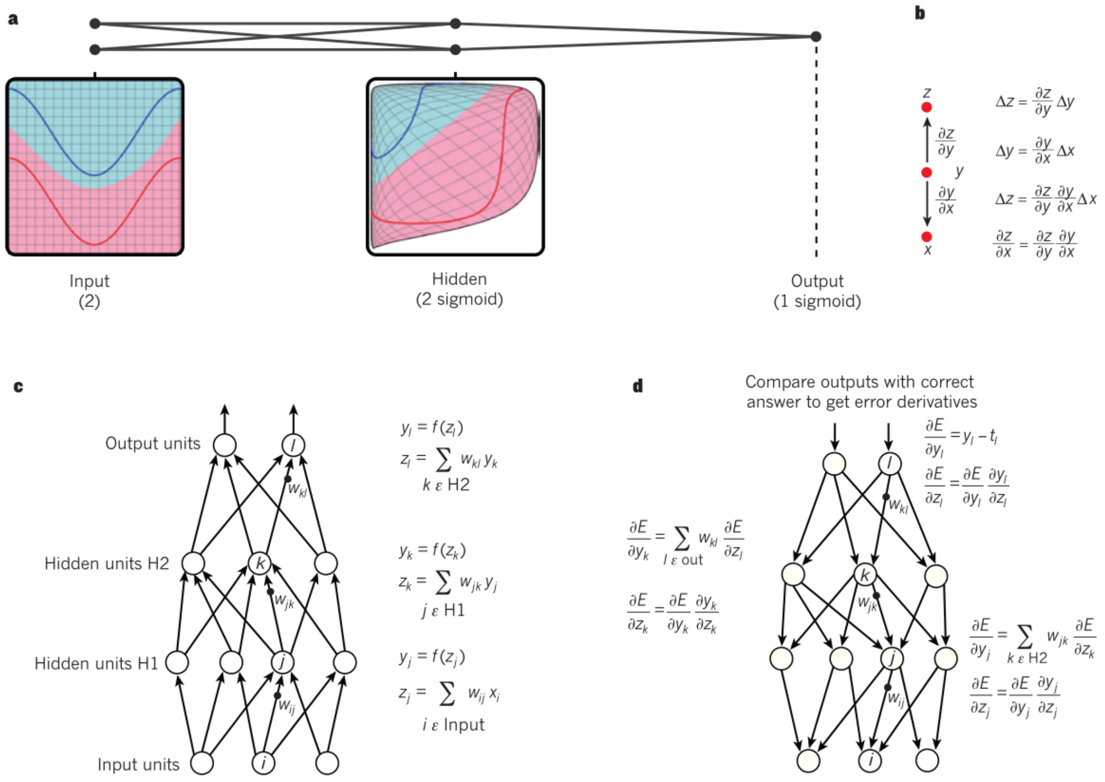
a
Input
(2)
Hidden
(2 sigmoid)
Output
(1 sigmoid)
b
$\Delta z = \frac{\partial z}{\partial y}\Delta y$
$\Delta y = \frac{\partial y}{\partial x}\Delta x$
$\Delta z = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\Delta x$
$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$
c
Output units
Hidden units H2
Hidden units H1
Input units
$y_l = f(z_l)$
$z_l = \sum_{k \in H2} w_{kl} y_k$
$y_k = f(z_k)$
$z_k = \sum_{j \in H1} w_{jk} y_j$
$y_j = f(z_j)$
$z_j = \sum_{i \in Input} w_{ij} x_i$
d
Compare outputs with correct answer to get error derivatives
$\frac{\partial E}{\partial y_l} = y_l - t_l$
$\frac{\partial E}{\partial z_l} = \frac{\partial E}{\partial y_l}\frac{\partial y_l}{\partial z_l}$
$\frac{\partial E}{\partial y_k} = \sum_{l \in out} w_{kl}\frac{\partial E}{\partial z_l}$
$\frac{\partial E}{\partial z_k} = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial z_k}$
$\frac{\partial E}{\partial y_j} = \sum_{k \in H2} w_{jk}\frac{\partial E}{\partial z_k}$
$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial z_j}$

Fig. 2.5: Feedforward Neural Networks, taken from [29]

Feedforward neural networks, or multilayer perceptrons (MLPs), are the essential deep learning models. The goal of a feedforward network is to approximate some measurable function f^*. For a classifier, $y = f^*(x)$ maps an *input unit* x to a category *output unit* y. A feedforward network defines a mapping $y = f(x; \theta)$, and learns the value of the θ that result in the best function approximation [28]. In the neural network terminology, the *layer* is used to express a collection of *units*. The middle layer between input and output layer is called *hidden layer*. The other two important term, *weights* and *bias* are introduced to express the importance of the respective the inputs to the output.

The Fig. 2.5 *a* provides a simple Feedforward neural network example with only two input units, two hidden layers, and one output. Each hidden layer is made up of a set of hidden units, where each unit is *fully-connected* to all units in the previous layer. Part *b* shows how small changes in the input unit effect in the output unit according to the chain rule of derivatives [29]. In the *c* and *d* the forward pass and the backward propagation are illustrated. At each layer, as shown in *c*, a weighted sum of input units combined with bias adjustment is followed by an activation function $f(dot)$ which introduces non-linearity into the system. There are three popular non-linear function, shown in Fig. 2.6, used in the neural network: **Sigmoid**, **tanh(x)** and **Rectified Linear Unit (ReLU)** that is discussed in the following section. During the backward training process, the cost function of the error between target and output unit is first computed. Then adjustment of weight is calculated using gradient descent method.

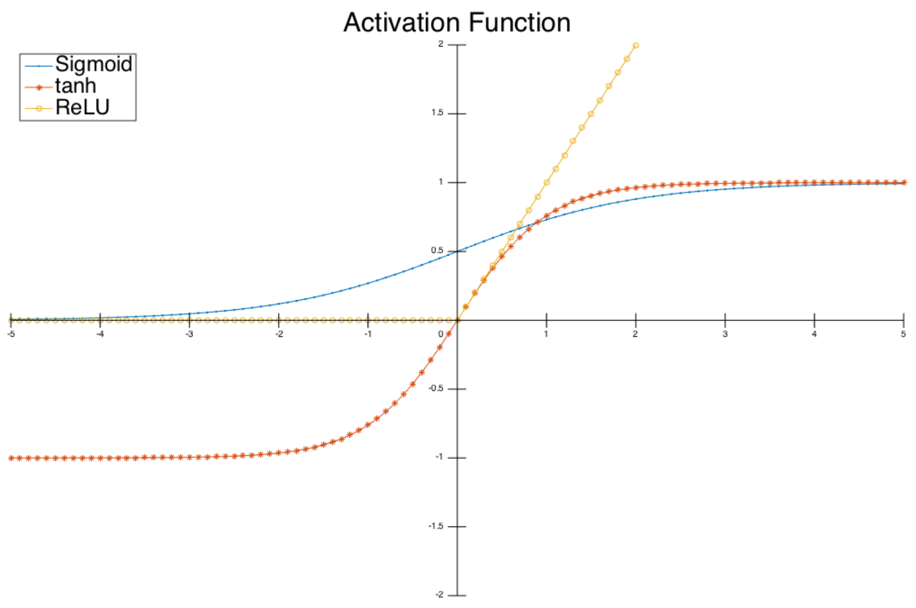

Fig. 2.6: The Comparison between Three Popular Activation Functions

2.5.2 Convolutional Neural Networks

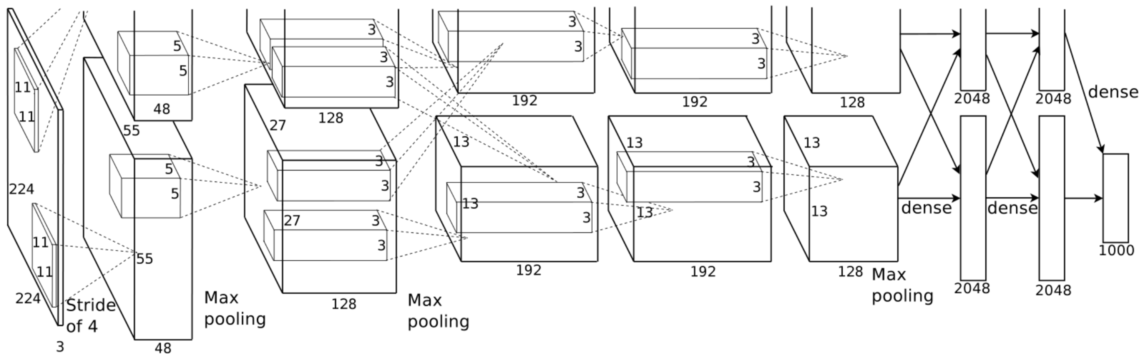

Fig. 2.7: The Example of Convolutional Neural Network for Image Classification, taken from [2]

Feedforward neural networks are able to handle the problem with relatively small-sized input. However, such a network architecture does not take into account the spatial structure of the large dataset like images, as a fully-connected structure is scalable. For example, an RGB image of size 500x500 would need 500*500*3=750,000 weights which are not manageable in ordinary neural networks. Convolutional neural networks[30], also known as CNNs or ConvNets, resolved these issues, which are specialized kind of neural network for processing data that has a know, grid-like topology [28]. ConvNets is a sequence of distinct layers. Five main types of layers to build ConvNet architecture are: **Convolutional Layer**, **ReLU Layer**, **Pooling Layer**, **Fully-Connected Layer** and **Normalization Layer**. Convolutional layer takes an image as an input, and compute regional weighted sum operation resulting in a matrix called *feature map*. This feature map is fed into ReLU layer to apply a $max(x, 0)$ function and then forwarded to Pooling layer to perform down sampling. Fully-Connected (FC) layer operates same as the regular neural network but with a smaller dataset, resulting in the final class score. In typical ConvNet architecture such as AlexNet[2], two or three stages of Convolutional, ReLU, and Pooling layers are stacked followed by two or three FC layers. The details of the individual layers are described below.

- **Convolutional Layer**:

  The Convolutional layer is the core functional part in the ConvNet that is responsible for more than 90% computation [31]. The convolution operation in the discrete system can be expressed as follows:

  $$s(t) = (x * f)(t) = \sum_{\alpha=-\infty}^{\infty} x(\alpha) f(t - \alpha) \tag{1}$$

  where $x(t)$ and $f(\alpha)$ are two input functions with discrete valuable t. In ConvNet terminology, two input function is referred as the input and weight function, and output is referred as the feature map. The convolution is often used in 2D space. For example, a monochrome image I can be convolved with a 2D filter function f, which is expressed as:

  $$s(i, j) = (I * f)(i, j) = \sum_{m=-\infty}^{\infty} \sum_{n-\infty}^{\infty} I(m, n) f(i - m, j - n). \tag{2}$$

  where variables (i, j) and (m, n) are the index over the horizontal and vertical axis. Many machine learning libraries implement **cross-correlation** but call it convolution [28]. The cross-correlation is the same as convolution without flipping the weight function:

  $$s(i, j) = (I * f)(i, j) = \sum_{m=-\infty}^{\infty} \sum_{n-\infty}^{\infty} I(i + m, j + n) f(m, n). \tag{3}$$

  Through the particular convolutional filter, features, such as line, curve, in the local field of the image can be extracted. In the common ConvNet architecture, multiple filters are embedded in a single convolutional layer to learn different features in the image. At each layer, the output of convolutional layer is:

$$S(i,j) = \sigma(b + (I * f)(i,j)) = \sigma(b + \sum_{m=-\infty}^{\infty} \sum_{n-\infty}^{\infty} I(i+m, j+n) f(m,n)). \quad (4)$$

Here, σ is the activation function mentioned in the common neural network and b is the bias value. The shared weight array f is normally smaller than 5x5. These shared parameters along with local connectivity make ConvNet more computationally efficient than Feedforward neural network.

- **ReLU Layer**:

  Rectified Linear Unit($ReLU$) can be implemented by thresholding a matrix at zero, while $Sigmoid$ or $tanh(x)$ activation functions involve expensive arithmetic operations. Additionally, $ReLU$ has a non-saturation form that accelerates the convergence of stochastic gradient descent. $ReLU$ has become very popular in the last few years in ConvNet architecture. The equation of $ReLU$ is very simple as follows:

$$f(x) = max(0, x). \quad (5)$$

- **Pooling Layer**:

  One of the typical operators in a ConvNet is Pooling. In a Pooling layer, convolved feature maps are condensed by a statistical summary of the nearby feature values. This operation is feasible due to the fact that images have the regional property. After pooling operation, the spatial size of feature values is reduced, resulting less computational tasks to perform in the flowing layer. Common choices of the pooling operator include $max - pooling$ and $average - pooling$. The max-pooling is defined as:

$$S(i,j) = max\{S(i',j') : i \leqslant i' < i+p, j \leqslant j' < j+p\}. \tag{6}$$

where p is the window size of the operator. Fig. 2.8 shows an example of max-pooling operation on a feature map with a 2x2 window.

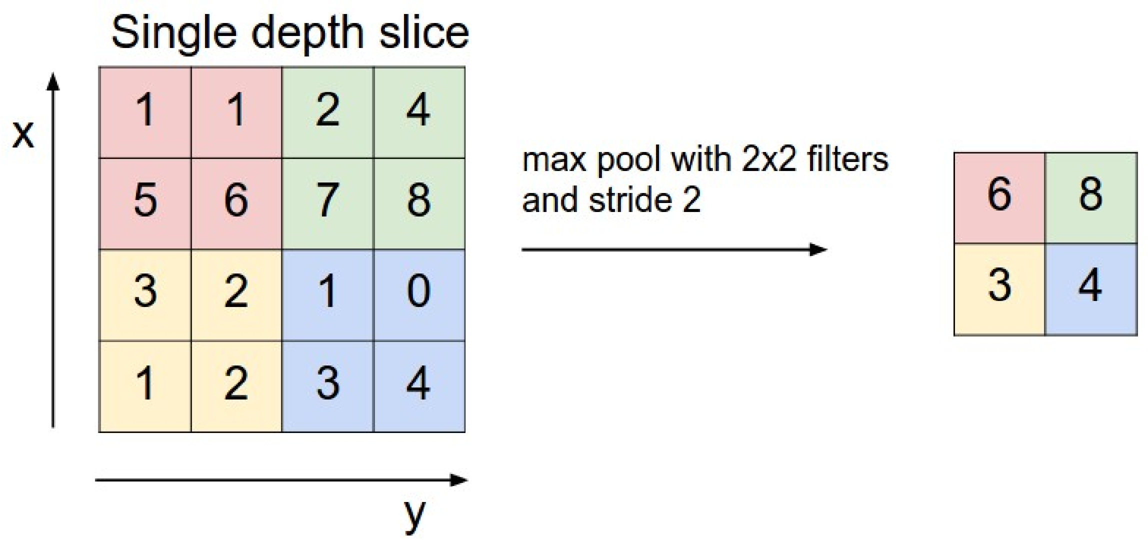

Fig. 2.8: The Example of Pooling Layer [32]

- **FC Layer**:

  The FC layer is the traditional component in the Feedforward neural network, in which every unit from the pooling layer connects to every unit of the output layer. The feature maps from the convolutional and pooling layers represent distributed high-level attribute of the input image. The FC layers are designed to combine these extracted features to classify the input image to various classes. The forward pass of the lth FC layer is computed as:

$$S^{l+1} = \sigma(b^l + F^l \times W^l). \tag{7}$$

FC layer can be easily converted to Convolutional layer, by adjusting filter size of the convolution operator, which is particularly useful in practice.

- **Normalization Layer**:

  There is another type of layers in ConvNet, called Normalization layer that is implemented to accelerate training process. In this work, we will focus Local Response Normalization (LRN), which is used in the AlexNet. The core idea behind this layer is to encourage some largely activated unit and form a local maximum. The formula of LRN is as follows:

$$y_{i,j}^{k'} = \frac{x_{i,j}^{k'}}{\left(N + \alpha \sum_{k \in G(k')} \left(x_{i,j}^{k}\right)^2\right)^{\beta}} \tag{8}$$

  where $x_{i,j}^{k'}$ represents activated feature unit from the kth convolutional filter at the position(i,j),

  $y_{i,j}^{k'}$ represents the output of LRN layer from the kth convolutional filter at the position(i,j),

  $G(k') = [k' - \lfloor \frac{n}{2} \rfloor, k' + \lceil \frac{n}{2} \rfloor]$ is a group of $n/2$ consecutive neighbor of kth element in the feature map,

  N, α and β are the hyper-parameters. In the AlexNet they are assigned as $N = 2$, $\alpha = 10e - 4$ and $\beta = 0.75$.

2.6 Related Work

For past decade, ConvNet has been applied to numerous applications including image classification, natural language processing, recommender system, etc. In this chapter, we review ConvNet implementations on the different hardware accelerators including GPUs, ASICs, and FPGAs.

2.6.1 Convolutional Neural Networks on GPUs

GPUs are designed for high throughput and modern GPUs can achieve thousands of floating point operations per second (FLOPS). As a workstation GPU, Kepler 40 (K40) GPU accelerator from Nvidia is able to compute more than four thousands of single precision floating point multiply and add each clock cycle and provide 288 Gigabyte per second memory bandwidth [33]. Due to these advantages, many researches have focused on accelerating GPU-based implementation of ConvNet. Coates et al. [34] and Krizhevsky [35] show an efficient workload partitioning with on-device communication on multi-GPUs to accelerate the computational process. To improve single-node performance on GPU, Mathieu et al. [36] replaced convolution by fast Fourier transform (FFT) and Denton et al. [37] utilized clustered filters and low-rank approximation. Some other efforts can be found on convolution layer parallelization by Ciresan et al. [38] and basic layer vectorization by Ren and Xu [39]. Recently, Han et al. [6] proposed compressed pipeline architecture using pruning and weight sharing, resulting up to 4X speedup and 7X energy efficiency. There are also various ConvNet framework and libraries for different layers targeting GPU, such as Caffe[8] *cuDNN*[7] and *cuda-convnet*[2].

However, high-end GPUs consume significant amount of energy. For example, the Thermal Design Power (TDP) of the Nvidia K40 is 235 Watt(W). As an increasing number of applications require low power solution, other accelerators have been explored to implement ConvNet.

2.6.2 Convolutional Neural Networks on ASIC

ASIC is a custom architecture for a particular use with lowest energy consumption and high throughput, while it requires long development time. In 2014, a machine learning supercomputer called DaDianNao was designed by Chen et al. [40]. They achieved a speedup of 450X over a GPU, and reduce power consumption by 150X, by using on-chip memory. Chen et al. [11] proposed an energy efficient custom accelerator on ConvNet. In this design, they presented a new data-flow to maximally reuse data and minimize data movement, resulting in more than 1.4X energy efficient on convolutional layer compared to GPU implementation. Finally, EIE by Han et al. [12] exploited deep compression schemes by pruning the redundant connections. Their work showed a processing power of 102 Giga operations per sec (GOPS) on a compressed network with a power dissipation of 600 mW, which is 3,400X and 2.9X energy efficient than a GPU and DaDianNao respectively. Although ASIC-based solutions provide desirable performance in terms of power consumption and throughput, the drawbacks of these solutions such as lack of flexibility, high development cost and long turnaround time hinder its adoption.

2.6.3 Convolutional Neural Networks on FPGAs

With an increase in the density of FPGA fabric and decreasing transistor size, recent FPGAs provided a large design space for ConvNet. Zhang et al. [41] proposed a convolutional layer implementation by exploring a optimization space on computational resources and memory access patterns. They achieved 61.6 GFLOPS on convolution layers of AlexNet at a power consumption of 18.6 W, by targeting a Xilinx Virtex 7 485T FPGA and using Vivado HLS tools. This result outperformed most of the previous work on FPGAs. A later implementation by Suda et al. [9] showed a throughput-optimized design with Intel OpenCL solution. Their 8-16 bit fixed point work achieved 72.4 GOPS and 117.8 GOPS on AlexNet and VGG [42] model running on an Intel Stratix-V GSD8 FPGA chip. Traditional FPGA design using HDL also contributed good results. In [10], Qiu et al. proposed a dynamic-precision data quantization flow to reduce bandwidth requirements, performing a throughput of 137 GOPS on the VGG16-SVD model. Similarly, an 8-10 bit fixed RTL design by Ma et al. [13] presented 114.5 GOPS on AlexNet with Altera Stratix-V GXA7. The FGPA based Caffe framework is implemented by DiCecco et al. [43] provides single precision implementation of ConvNet at the performance of 45.8 GFLOPS on AlexNet and 55 GFLOPS on VGG model.

2.7 Summary

In this chapter, we describe preliminary background on FPGA Architecture, High-level Synthesis, OpenCL framework, Intel FPGA SDK for OpenCL tool, and supervised machine learning. Then we reviewed state-of-art implementations of ConvNet on various hardware including GPUs, ASIC, and FPGAs. GPU implementations are optimized for high throughput but consume significant energy. On the other hand, ASIC implementation presents more energy efficiency while its development time is significant. Finally, FPGAs provide a balance point between GPUs and ASIC, by offering relatively high throughput and short design processing time. While FPGA-based implementation has already presented comparable energy efficiency to GPUs, there is still a performance improvement can be achieved. In this work, we fully utilize several parallelism schemes and optimization strategies to improve throughput on end-to-end ConvNet architecture.

3 Design Flow and Optimization for Intel FPGA SDK for OpenCL

In this chapter, we discuss detailed design flow and optimization strategies for Intel FPGA SDK for OpenCL.

3.1 OpenCL Design flow

During the initial stage, kernel program (.cl) is compiled by Intel Offline Compiler (AOC) to generate an emulated kernel. The emulator that runs on x86 based host is able to check for syntax errors and functional correctness in a short time. When generating the emulator, AOC also provides an optimization report to provide information about memory transaction, initiation interval of pipeline execution. Using this feedback, the designer can optimize the kernel. At the next stage, kernel program is fully compiled with AOC to synthesize the OpenCL code directly to an RTL design in Verilog. The tool simplifies the development process by automatically handling interactions between different memory region and pipeline depth. Full compilation takes 4-6 hours to finish depending on the application. Finally, the host program along with the FPGA executable file are compiled by the GNU Compiler Collection (GCC) and run on the system. If performance or resource usage fails to meet the requirement, the kernel needs to be further optimized and compiled. The design flow is shown in Fig. 3.1.

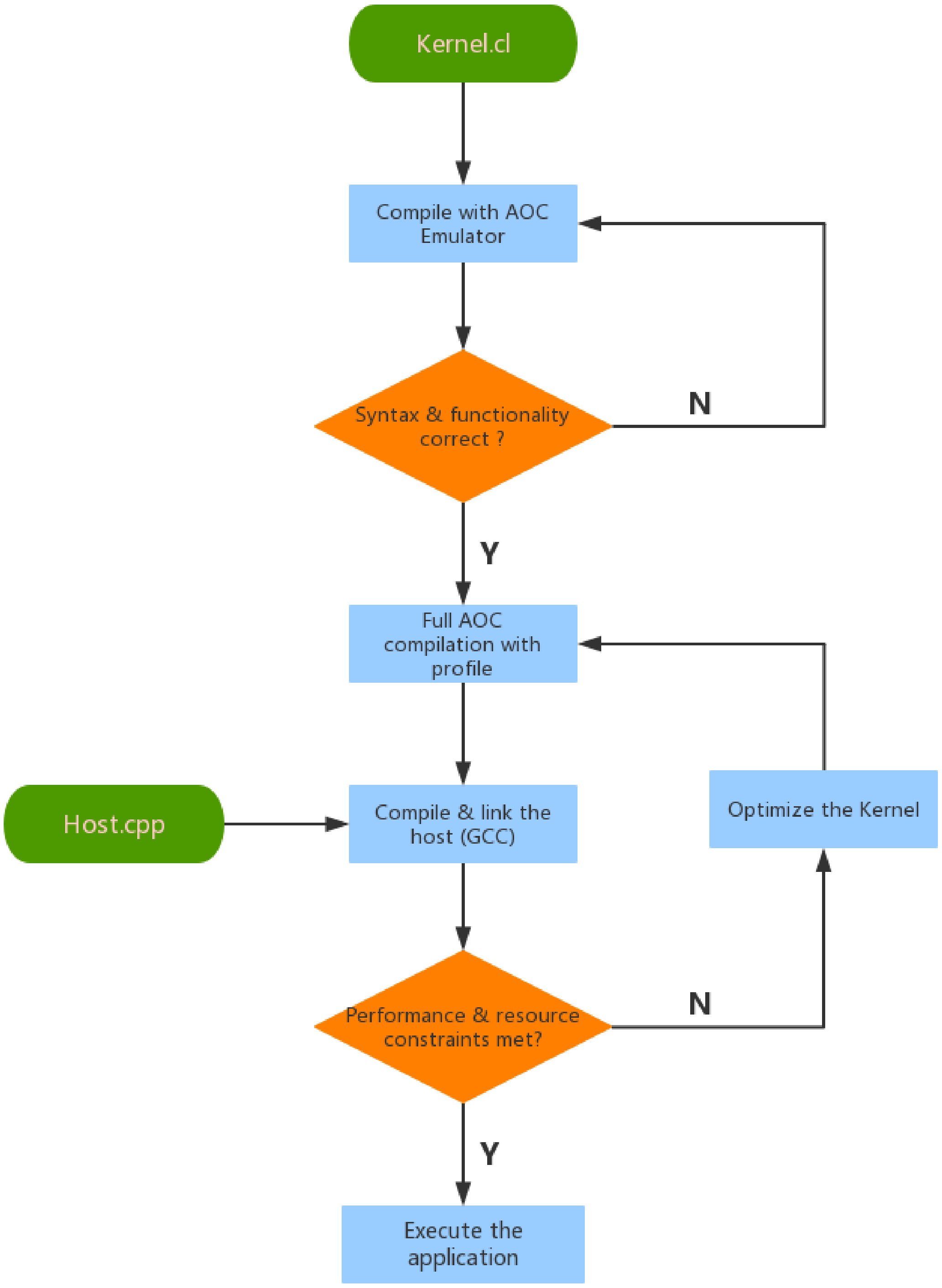
Kernel.cl
Compile with AOC Emulator
Syntax & functionality correct ?
N
Y
Full AOC compilation with profile
Host.cpp
Compile & link the host (GCC)
Optimize the Kernel
Performance & resource constraints met?
N
Y
Execute the application

Fig. 3.1: Intel FPGA Design Flow

3.2 Optimization Strategies in OpenCL for FPGA

The most important rule in high-performance computing is to make the computationally intensive part fast. In FPGA accelerator system, this goal can be largely divided into two parts as shown in Fig. 3.2: reducing the computational time spent on targeting tasks and decreasing communication delay between host and FPGA. In the serial system, the host prepares data using T_{host} and executes computational functions using $T_{un-affectedfunc}$. However, in the parallel system, the accelerator device is utilized to reduce the computational time to $T_{affectedfunc}$, and communication overhead T_{comm} between the host and the device is introduced to the total execution time.

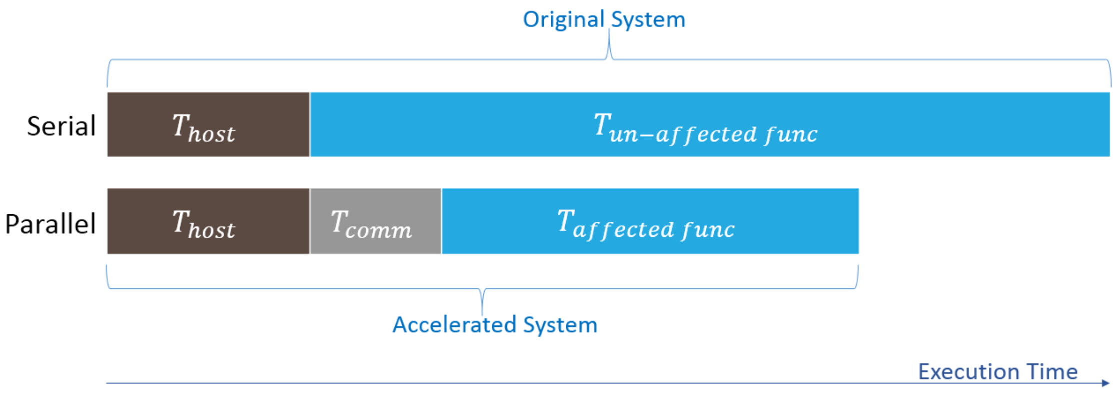

Fig. 3.2: Pictorial Depiction of Accelerated System

3.2.1 Parallelism Scheme

There are three different kinds of parallelism in Intel FPGA SDK for OpenCL: **Data Parallelism**, **Loop Parallelism**, and **Task Parallelism**. Data parallelism and Loop parallelism are forms of parallelism performed within computation tasks (kernels), while Task parallelism is a scheme to manage and execute these tasks to obtain maximum possible parallelism in the system.

- **Data Parallelism**:

  In Data parallelism mode, the kernel is executed in a Single Program Multiple

Data (SPMD) style across a 1D, 2D or 3D grid of work-items. Similar to GPU solution, all work-items are grouped into work-groups and each work-group executes the same function. The work-group size is the number of work-items in a work-group, which is an important design parameter that impacts the kernel performance. It needs to be tuned to utilize the hardware resources and maintain work-group level parallelization. Data parallelism is best suited for dealing with loops where there are no dependencies between instructions.

- **Loop Parallelism**:

  Loop parallelism is implemented by launching kernels with a single work-time which is defined as a *Task* in OpenCL. In GPU programming, a single-work item kernel is designed to resolve data dependent section in the loop, while it keeps other processing elements idle that hinders overall performance. However, FPGA provides flexibility to extract parallelism between each loop iteration to resolve dependencies. Compiler pipelines each iteration of the loop by launching the next iteration as soon as loop dependencies have been resolved. Although compiler handles pipeline structure and scheduling, the designer can improve pipeline performance by removing, relaxing and simplifying loop-carried dependency.

  The pictorial comparison of loop parallelism and data parallelism is presented in Fig. 3.3 . In this simple example, we have six work-items labeled with 1 to 6 executing a kernel with five stages (A-E). In the data parallelism scheme, the system handles three work-items at a time and finishes the work using the 10 clock cycles. In the loop parallelism, five stages are executed in pipeline fashion within the kernel, resulting in finishing the work using the same amount of the time. However, the throughput in the system utilizing loop parallelism is greater than the one with data parallelism. If we proceed the system with additional work-items, loop parallelism would complete the each task in one clock cycle, while data parallelism would still need five clock cycles to finish three tasks.

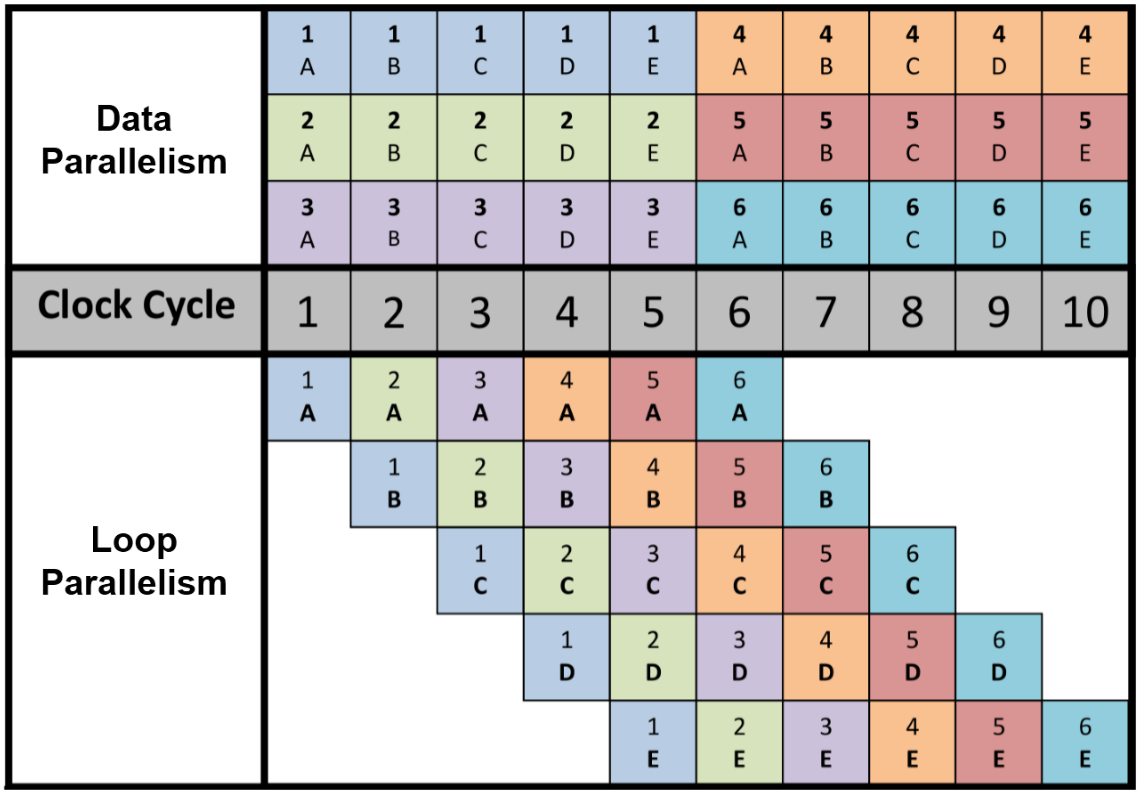

Fig. 3.3: The Comparison between Data Parallelism and Loop Parallelism, adopted from [25]

A local variable can be introduced to store intermediate results. This procedure can decouple the complex computations in the loop, which can remove loop carried dependencies. Relaxing loop-carried dependency is done by increasing the dependence distance, the number of iterations between generation and use of a variable. Inferring shift registers into single thread kernel that provides temporal locality is implemented for this purpose. Simplifying dependency means that expensive functions need to be avoided when computing loop-carried variables. From the memory perspective, transferring dependency from global memory to local memory is another strategy to reduce initiation interval of the stalled pipeline.

- **Task Parallelism**:

  Data parallelism and loop parallelism optimize the computational architecture in a kernel, while task parallelism manages these kernels into a pipeline running fashion through command queues. Intel solution supports concurrent kernel execution driven by multiple queues. These executions are asynchronously enqueued which requires explicit synchronization points. Task parallelization is extremely useful for a big application that can be divided into separate kernels. These kernels are enqueued on different queues and communicate through synchronization methods like *clFinish* and *channel*.

3.2.2 Throughput-oriented Optimizations

Another method for improving throughput on FPGAs is to create multiple hardware components for a specific computational part. Three hardware replication methods are available in Intel FPGA SDK, **Kernel Vectorization**, **Loop Unrolling** and **Computer Unit Replication**.

- **Kernel Vectorization**:

  This optimization converts read, write and arithmetic operations from scalar fashion into a Single Instruction Multiple Data (SIMD) mode. The compiler will replicate the kernel data path according to the number of vectorization, which can reduce the number of memory access. In addition, the vectorized input and output can ensure contiguous memory access pattern that improves memory usage efficiency.

- **Loop Unrolling**:

  A large number of loop iterations in the kernel can hinder performance. Loop unrolling technique can allocate more hardware resources to reduce or even remove the loop counter, which improves throughput linearly. This method also helps memory coalescing to reduce memory transaction time.

- **Computer Unit Replication**:

  If hardware resource is still sufficient after above two optimization strategies, multiple Compute Units (CU) can be generated for each kernel that means create multiple copies of the separate pipelines. However, multiple CUs cannot always linearly improve the throughput as all CUs share the global memory bandwidth that brings memory access contention among the CUs. Additionally, CU replication also can lower the operating frequency.

3.2.3 Communication Optimization

Since many applications are bounded by memory bandwidth, efficient memory access for decreasing communication overhead is the other optimization topic to pay attention.

- **Memory Alignment**:

  The memory allocation on the host side needs to be at least 64-byte aligned. This enables DMA transfer on host-FPGA communication which significantly improves transfer efficiency. The allocation can be implemented using *posix_memalign* function in Linux supported by GCC or *_aligned_malloc* function in Windows supported by Microsoft.

- **Local Memory Caching**:

  As introduced in Chapter 2, OpenCL defines a memory model with four region: global memory, constant memory, local memory and private memory. The local memory, that implemented in on-chip RAM block, has much lower latency and higher bandwidth than global memory. Thus the local memory allows to cache global memory that requires repeated accesses before computation. In the data parallelism mode, these cached local memory is visible to all work-items within the same work-group. Use of local memory can improve kernel performance by reducing global memory access.

- **Coalescing Memory Access**:

  As discussed in the previous section, coalescing memory access can reduce the number of memory access and improve memory efficiency. This is important when reading from and writing to the global memory which is large but slow.

- **Channels**:

  In the typical GPU application, data movement between different kernels requires global memory accesses that have high latency and limited bandwidth. This latency, that increases linearly with additional kernels, makes the pipeline stall. To address this problem, Intel has made available the vendor extension called *channel* to use FIFOs for data transferring between kernels. With channels, task parallelism can launch the *consumer* kernel as soon as the intermediate results are available in the FIFO fed by the *producer* kernel. The channels serve as a synchronization point between the *consumer* kernel and the *producer* kernel. One of the current restrictions of using channels is that we are not allowed to use automatic vectorization (*num_simd_work_items*). However, this can be resolved by manual vectorization by using structured data set or vectorized data type, such as *float8* and *int16*.

[44] describes some other optimization tips such as using compiler option for global memory partitioning and floating point arithmetic, which might be helpful in providing better performance for certain applications.

4 Methodology for Creating Optimized OpenCL Specification for ConvNet Algorithm

In this chapter, we discuss the methodology of an efficient implementation of ConvNet on FPGA by using Intel OpenCL SDK. We begin by analyzing the complexity of ConvNet to find the bottleneck of the architecture. Then we discuss various techniques to alleviate these challenges, including precision quantization, data re-arrangement, and pipeline architecture, so that synthesized hardware provides the best possible performance

4.1 Analysis of Computational Complexity and Space Complexity

State-of-the-art ConvNet shows a trend to go deeper to increase classification accuracy. In ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2012, AlexNet [2] won image classification task by achieving 84.7% top-5 accuracy [45]. The parameter size of AlexNet is around 250 MB that spread in 8 layers. The model requires a total of 1.45 billion operations, including convolution, activation, pooling, and element-wise product calculation. In 2014, VGG [42] achieved 92.6% top-5 accuracy and won the second place in the same task. VGG architecture needs around 500 MB of weight parameters and 30.9 billion operations. In this work, we will focus on VGG 16 model that has more deeper architecture and better results than AlexNet. The idea of this work can migrate to other models, like GoogLeNet [46] or SqueezeNet [47].

The computational complexity of a layer is the sum of each operator. In a convolution layer, each input feature map convolves with a $F_W \times F_H$ filter, resulting in a $W_{cout} \times H_{cout}$ output feature map. The number of input and output is N_{in}, N_{cout} respectively. Each element needs two operations for multiplication and addition, the complexity of convolution layer is:

$$C_{Conv} = 2 \times N_{in} \times F_W \times F_H \times W_{cout} \times H_{cout} \times N_{cout}. \tag{9}$$

For ReLu activation layer, only one operation(comparison) is required for each unit in the convolved result:

$$C_{ReLU} = N_{cout}. \tag{10}$$

In the pooling layer, input feature maps are down-sampled through a $P_W \times P_H$ filter, generating N_{pout} of $W_{pout} \times H_{pout}$ outputs. Then computation complexity of pool layer is:

$$C_{Pool} = P_W \times P_H \times W_{pout} \times H_{pout} \times N_{pout}. \tag{11}$$

Finally, in the FC layer, N_{FCin} of input features multiply and add with N_{FCout} of weight parameters:

$$C_{FC} = 2 \times N_{FCin} \times N_{FCout}. \tag{12}$$

The memory space requirement is described with space complexity. The main parameter in the ConvNet is the weight which is used in the convolutional and FC layer. The number of weight in the convolutional layer can be expressed as:

$$S_{Conv} = w^2 \times N_{cin} \times N_{cout}. \tag{13}$$

Similarly, the space requirement of FC layer is:

$$S_{Conv} = N_{FCin} \times N_{FCout}. \tag{14}$$

The detailed configuration, computational complexity and space complexity of VGG 16 model are calculated and shown in Table 4.1. In this model, the classification on one image requires around 138 million weight parameters and 30.9 billion operations. Table 4.1 also shows that 99.2% of computation is done in the Stage 1-5 using only 10.63% of weights. On the other hand, FC layer in Stage 6-8 is responsible for 0.8% computational task but using 89.37% of the network parameters. From the table, we can summarize that convolutional layers are computationally intensive, while FC layers are memory-centric. In the rest of chapter, we will discuss how to offer a large enough computational throughput in the Stage 1-5 and how to efficiently use memory bandwidth to keep the processing units busy in the Stage 6-8.

4.2 Data Quantization

In general, neural network implementation including ConvNet uses 32-bit floating point. However, the scenario has been changed. Vanhoucke et al. [48] built reduced precision speech recognition, which is based on the typical neural network model, with a 10X speedup and no cost in accuracy. Recently, many of the FPGA works on ConvNet have been focused on using the fixed-point representation with less than 1% accuracy degradation on AlexNet and VGG 16 model [9][10][13].

A fixed-point number format is: [IL.FL], where IL is the number of integer bits, and FL is the number of fractional bits. The sum of IL and FL gives the total number of bits in WL. The precision of the fixed-point number is 2^{-FL}, and the range can be defined as $\left[-2^{IL-1}, 2^{IL-1} - 2^{-FL}\right]$. For example, the range and precision of 16-bit fixed-point number with [8.8] are $2^{-}8$ and $[-128, 128 - 2^{-8}]$ respectively.

Fixed-point arithmetic is hardware-friendly and saves more logic resources on FPGA to enable more parallelism. It also reduces memory footprint on the chip and reduces bandwidth requirement.

Experimental results provided by Qiu et al. [10] show that 8/4 bit fixed-point

Table 4.1: The Complexity of VGG 16 Model

Stage	Name	Input Nin	Input W	Input H	Stride	Operator W	Operator H	Output Nout	Output W	Output H	#Operations	#Weights
Stage 1	Conv/ReLU	3	224	224	1	3	3	64	224	224	176,619,520	1,728
	Conv/ReLU	64	224	224	1	3	3	64	224	224	3,702,587,392	36,864
	Pooling	64	224	224	2	2	2	64	112	112	3,211,264	0
Stage 2	Conv/ReLU	64	112	112	1	3	3	128	112	112	1,851,293,696	73,728
	Conv/ReLU	128	112	112	1	3	3	128	112	112	3,700,981,760	147,456
	Pooling	128	112	112	2	2	2	128	56	56	1,605,632	0
Stage 3	Conv/ReLU	128	56	56	1	3	3	256	56	56	1,850,490,880	294,912
	Conv/ReLU	256	56	56	1	3	3	256	56	56	3,700,178,944	589,824
	Conv/ReLU	256	56	56	1	3	3	256	56	56	3,700,178,944	589,824
	Pooling	256	56	56	2	2	2	256	28	28	802,816	0
Stage 4	Conv/ReLU	256	28	28	1	3	3	512	28	28	1,850,089,472	1,179,648
	Conv/ReLU	512	28	28	1	3	3	512	28	28	3,699,777,536	2,359,296
	Conv/ReLU	512	28	28	1	3	3	512	28	28	3,699,777,536	2,359,296
	Pooling	512	28	28	2	2	2	512	14	14	401,408	0
Stage 5	Conv/ReLU	512	14	14	1	3	3	512	14	14	924,944,384	2,359,296
	Conv/ReLU	512	14	14	1	3	3	512	14	14	924,944,384	2,359,296
	Conv/ReLU	512	14	14	1	3	3	512	14	14	924,944,384	2,359,296
	Pooling	512	14	14	2	2	2	512	7	7	100,352	0
Stage 6	FC/ReLU	25088						4096			205,524,992	102,760,448
Stage 7	FC/ReLU	4096						4096			33,558,528	16,777,216
Stage 8	FC	4096						1000			8,192,000	4,096,000
Stage 1-5											30,712,930,304 (99.20%)	14,710,464 (10.63%)
Stage 6-8											247275520 (0.80%)	123633664 (89.37%)
Total											30,960,205,825	138,344,128

design with dynamic-precision produces less than 2% accuracy loss. However, we will use 16-bit fixed-point number representation with static configuration in the fixed-point implementation to design scalable and more accurate hardware kernels.

4.3 Data Arrangement and Pre-processing

Given a ConvNet, the task of pre-processor is to decompose and re-organize input data including images and weight matrix. The goal of this processing is to increase communication bandwidth by maximizing data reuse and improving memory access efficiency. As shown in section 4.1, convolution is the most computationally important tasks involving 99% of the operations. Using 2D convolution defined in Equation 4, the 3D convolution utilized in this work can be extended as:

$$Out[N_{cout}][H_{in}][W_{in}] = \sum_{N_{in}=0}^{N} \sum_{f_h=0}^{F_H} \sum_{f_w=0}^{F_W} In[N_{in}][H_{in}+f_h][W_{in}+f_w] * weight[N_{cout}][N_{in}][f_h][f_w]. \tag{15}$$

where N_{in} is the number of input feature maps, H_{in} and W_{in} are the height and width of a single input feature map, and F_H and F_W are the size of a convolutional window. To differentiate data arrangement in the 3D arrays, we use the following notations in this work:

- N, the number of feature maps
- H, the vertical size of a feature map
- W, the horizontal size of a feature map

With the notation above, the data layout in the Equation 15 can be pictorially represented in NHW fashion, as illustrated in Fig. 4.1 the data is stored consecutively along the lowest dimension W. Similarly, data in a stride of W is stored consecutively along the H dimension, and a stride of $W * H$ is stored along the N dimension.

From Table 4.1, we can find that the width and height of each feature map vary from 224 to 12 in the different layers. When W or H is the lowest dimension, the great common divisor of that dimension is 4 which is too small to provide coalesced memory access pattern. However, another dimension N is a multiple of 16 in the all convolutional and FC layers, except the first layer when loading the original image. Therefore, using N as the lowest dimension with zero padding in the fist layer is favorable for memory coalescing thus increasing bandwidth efficiency. Due to the fact that dimensions W and H have the same value for all layers, the sequence of these two dimensions does not affect performance. As a conclusion, we use HWN data layout in the accelerator, and this transformation is done on the host side that is shown in Fig. 4.1.

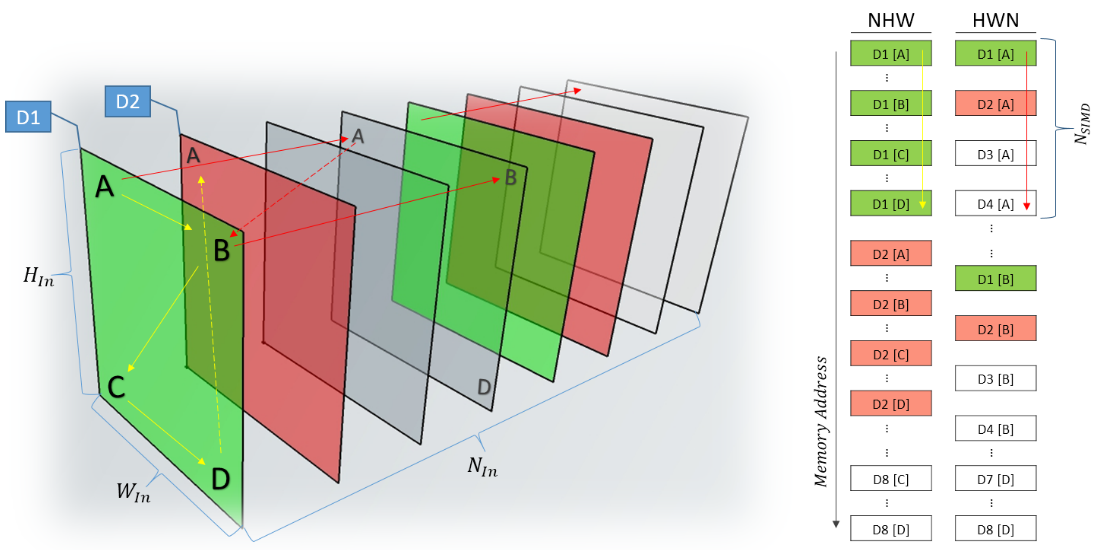

Fig. 4.1: The Data Layout Comparison Between "NHW" and "HWN".

4.4 Converting FC Layers to Convolutional Layers

As discussed in [32], it is worth nothing that the functional form of FC layers and convolutional layers is identical, and involves multiply and add. The difference is that in convolutional layers the filter weights are locally connected to a certain region of the input, while in FC layer the weights are connected to all units of the input. Therefore, it is possible to convert from FC layers to convolutional layers by setting filter size same as the input size. In the FC layer from Stage 6 in the VGG 16, for example, an input feature with the size of $7 \times 7 \times 512$ can be convolved with 4096 of filters with the size of $F_W = 7, F_H = 7$, stride size of 1, generating the output with the size of $1 \times 1 \times 4096$. In the same fashion, the other FC layers also can be easily converted as convolutional layers. This way we can utilize the same computational engine for both layers, thus saving the hardware resources for more parallelism.

4.5 Optimized FPGA Hardware Synthesis from OpenCL Specification

This section describes Optimized FPGA Hardware Synthesis from OpenCL Specification based on the optimization strategies mentioned in Chapter 3. First, we look into the available parallelism in the ConvNet that we can exploit in the design. Then the detailed implementation of parallelism in the OpenCL specification is described.

4.5.1 Parallelism in ConvNet

To obtain the highest throughput in the computation-intensive part of ConvNet, all of the possible parallelism should be properly exploited.

- **Data Parallelism in ConvNet**:

  For different output feature maps, all processing, including reading, convolving, pooling and writing back is data independent. Thus the whole output feature maps can be partitioned along the N dimension and each part is processed on a separate data path. This can be implemented by CU replication in OpenCL which could drastically improve the throughput of the proposed system. Also, each convolution operation for an output feature unit consists of phase: element-wise *multiplication* between input feature maps and filters and *accumulation* of these products. The fist phase, multiplication is fully independent which can execute in data parallelism fashion.

- **Loop Parallelism in ConvNet**:

  As mentioned above, accumulated summation is dependent on neighbor units. Similarly, pooling operation requires communication, either comparison or summation, between neighbor units. At the same time, the summation and the pooling need considerable amount of iterations to get the final output. These operations fit well in the loop parallelism implemented by Intel OpenCL compiler, which schedules each iteration into a pipeline.

- **Task Parallelism in ConvNet**:

  Both data parallelism and loop parallelism increase the performance of the task, while task parallelism can boost the throughput in the system level. Each layer of ConvNet consists of several different computational tasks that run with a producer/consumer relationship. These tasks can be mapped into separate kernels and executed on multiple command queues. The proposed tasks utilize channels as a data communication method as well as synchronization points. As a result, all tasks execute in a pipeline mode, resulting in increasing hardware utilization. The task parallelism in this work is shown in Fig. 4.2.

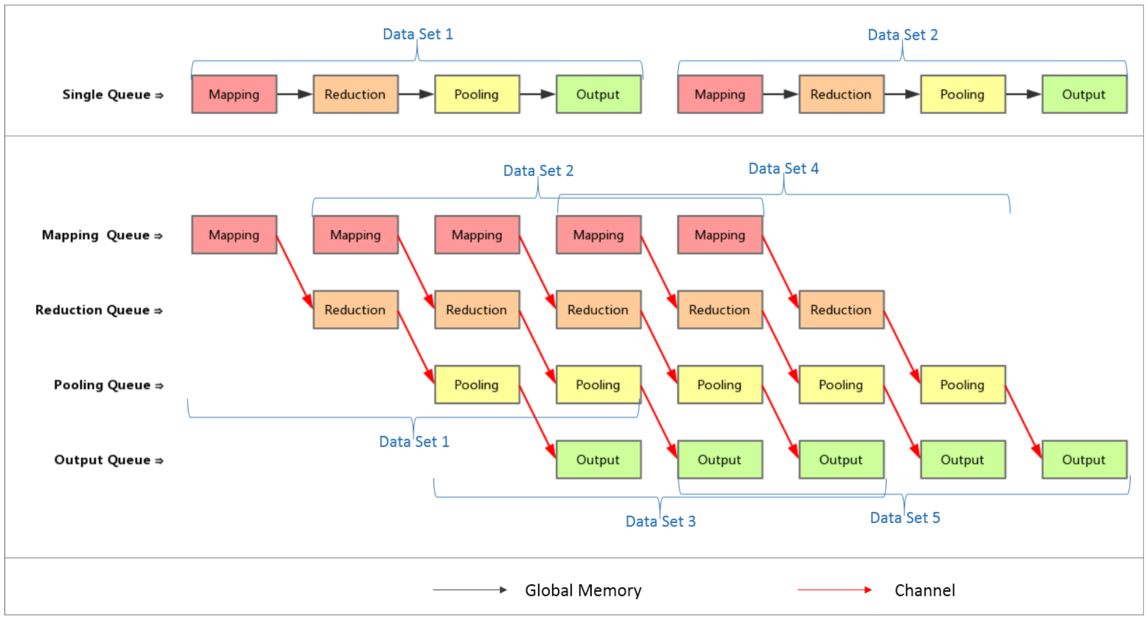

Fig. 4.2: Task Parallelism in Proposed Design

4.5.2 Data Reuse in ConvNet

To improve bandwidth utilization, data reuse can be achieved in two parts of ConvNet. One is from the convolutional part, where N_{cout} of different filters share a single input feature map. Thus input features from each layer can be cached into local memory to enable data locality and save memory bandwidth. The other data reuse could happen in the FC layer. Batch processing of multiple images allows data reuse of filters in the FC layers so as to decrease memory access time.

4.5.3 Kernel Design

In the context of the optimized implementation of ConvNet based on the parallelism analysis and data reuse pattern, The proposed design consists of N_{cu} of CUs, each of which is made up of four OpenCL kernels:

1. **Mapping kernel**:

   The *Mapping kernel* is designed as a multi work-item (NDRange) kernel that is responsible for reading input feature maps and corresponding filters from the memory as well as computing the dot product of loading features and filters. As an input feature map is reused by multiple filters N_{cout} times, a local memory is designed to cache the input array to allow fast access during the computation and reduce the number of global memory access. Due to the fact that the data layout in the feature maps and filters are in HWN mode, kernel vectorization can be achieved along the N dimension. By vectorization, scalar operations are translated into SIMD operations to achieve higher throughput. Vectorization also ensures coalesced memory access patterns which decreased the number of memory access and improve memory bandwidth efficiency.

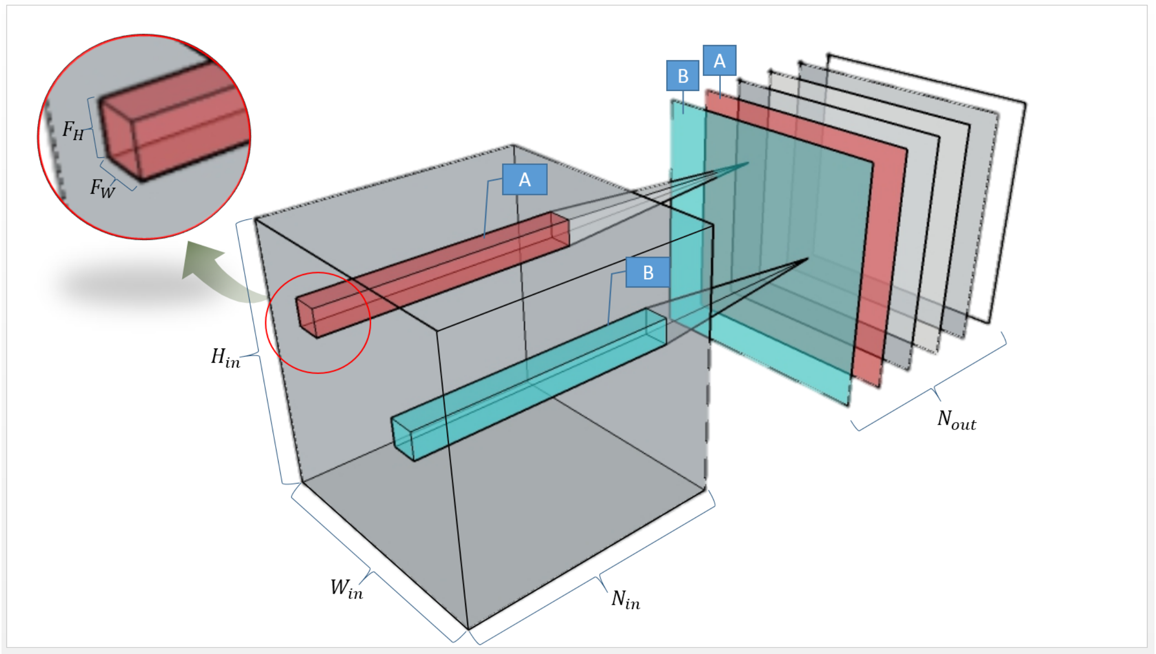

Fig. 4.3: Decomposition of 3D Convolution

The kernel consists of $W_{cout} \times H_{cout} \times (N_{cout}/N_{CU})$ work-groups, each of which operates on each output unit in one partition. In a work-group, there are $F_w \times F_H \times (N_{in}/N_{SIMD)}$ work-items, each of which executes on an input vector and a weight vector. The pseudo-code for *Mapping kernel* is shown in Algorithm 4.1. It can be summarized as follows:

Algorithm 4.1 Pseudo-code for Mapping kernel.

```
1: Get global and local index of work-item
2: Compute address locations for input features and filters using index
3: Fetch input feature units to a vector $in[N_{SIMD}]$ in local memory
4: Fetch filter units to a vector $filter[N_{SIMD}]$ in local memory
5: #progma unroll
6: for each element $i$ in both vectors do
7:     $Partial\_Sum \leftarrow Partial\_Sum + in[i] \times filter[i]$.
8: end for
9: Write $Partial\_Sum$ into $channels$
```

(a) Calculate the index of input feature maps and fetch corresponding units into local memory.

(b) Calculate the index of filters and fetch corresponding filter values into local memory.

(c) Perform element-wise multiplication and partial summation within a vector.

(d) Write the partial sum into the channels

2. **Reduction kernel**:

The *Reduction kernel* is implemented as a single work-item kernel which computes the convolution result of the output feature maps and performs ReLU function on each result. It iterates over all units in the output feature maps. In each iteration, the kernel calculates the convolution result of each unit in the output feature maps, by traversing along N dimension and reducing partial sums to final sum. During this traversing, a shift register is inferred to alleviate loop-carried dependency of partial sums from different sources. The size of the shift register needs to be larger than the delay between the successive iteration. Then reduced results go through ReLU filter, discarding all negative results. Finally, the filtered results are fed into another channel to next stage. The outer loop iterates $W_{cout} \times H_{cout} \times (N_{cout}/N_{CU})$ times, while inner loop traverses

$F_w \times F_H \times (N_{in}/N_{SIMD)}$ times. After successful synthesis, the compiler run the kernel in a pipeline fashion, launching the next iteration as soon as the dependency on previous iterations is resolved. The pseudo-code of *Reduction kernel* is shown in Algorithm 4.2.

Algorithm 4.2 Pseudo-code for Reduction kernel.

```
1:  for each output element do
2:      Initialize a register $Sum\_REG$ for reduction result
3:      Initialize shift register $SR[DEPTH]$
4:      for each filter vector do
5:          Read $Partial\_Sum$ from $channels$
6:          $SR[DEPTH-1] \leftarrow SR[0] + PartialSum$
7:          #progma unroll
8:          for $i = 0$ to $DEPTH - 1$ do
9:              $SR[i] = SR[i+1]$
10:         end for
11:     end for
12:     #progma unroll
13:     for $i = 0$ to $DEPTH - 1$ do
14:         $Sum\_REG \leftarrow Sum\_REG + SR[i]$
15:     end for
16:     if ReLU filter is on then
17:         if $Sum\_REG > 0$ then
18:             $Sum\_REG \leftarrow Sum\_REG$
19:         else
20:             $Sum\_REG \leftarrow 0$
21:         end if
22:     end if
23:     Write $Sum\_REG$ into $channels$
24: end for
```

3. **Pooling kernel**:

   Pooling layer down-samples the convolutional results using the average or maximum value of units in a region which has the dependency between successive iterations. Thus we implement the single work-item kernel for this layer. Similar to the convolutional layer, a shift register with the depth of $(P_W - 1) \times W_{cout} + (P_H - 1)$ is designed for buffering the pooling data. Then pooling operations run on the shift register according to pooling mode. For average pooling, the division operation is replaced by multiplying the inverse of the pooling window size. The outer loop in the pooling layer iterates $W_{cout} \times H_{cout} \times (N_{cout}/N_{CU})$ times.

4. **Output kernel**:

   Output kernel reads pooling results from the pooling channel and writes them back to the global memory. The batch processing is exploited for this work to improve filters reuse time in FC layers. Hence, in the FC layer output kernel needs to collect and write N_{batch} sets of results. For other layers, it processes one set of result. Since the output processing is fully independent, the kernel is designed in an NDRange fashion, running with $P_W \times P_H \times (N_{pout}/N_{CU)}$ work-items in parallel.

The four implemented kernels execute on the separate command queues to exploit task parallelism. The block diagram of the proposed system with $N_{CU} = 4$ is shown in Fig. 4.4.

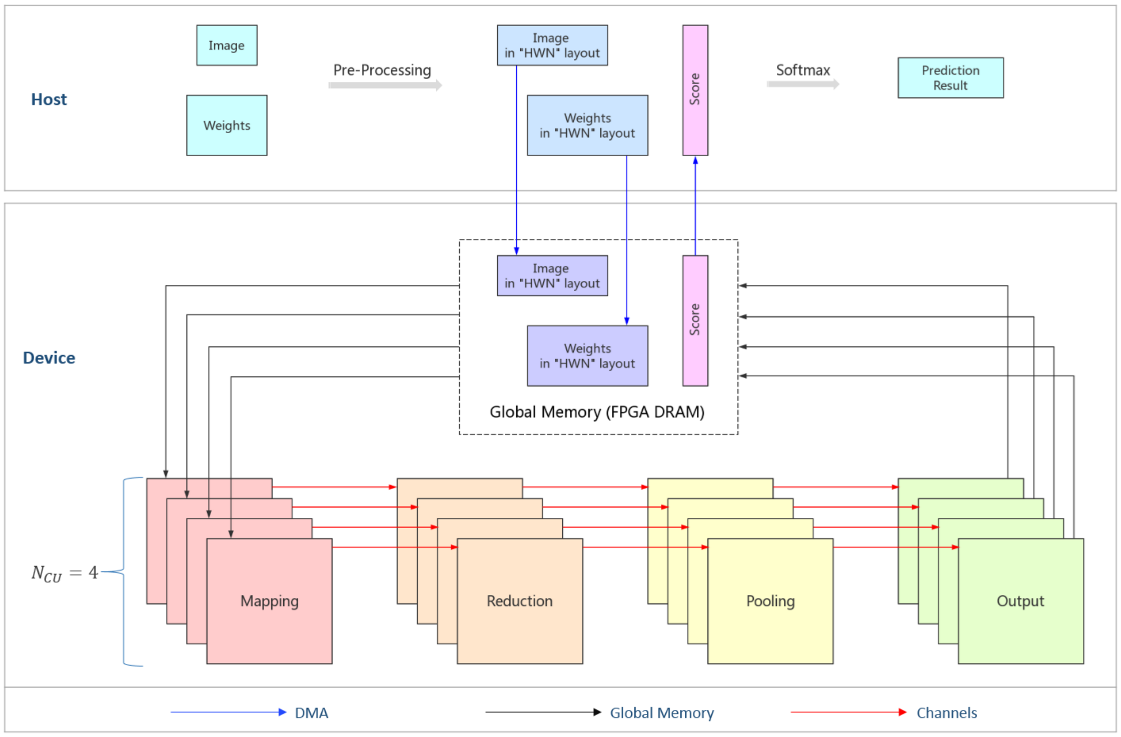

Fig. 4.4: The Block Diagram of Proposed System

4.6 Summary

In this chapter, we first quantitatively analyzed the computational and space complexity of ConvNet using VGG 16 model, and the result showed that there are two main implementation challenges including computation intensive convolutional layers and memory bandwidth bounded FC layers. To alleviate these challenges, we utilized data rearrangement, data quantization, and batch processing strategies on the host side. To fully use limited FPGA resources, we created universal and scalable OpenCL kernels for both convolutional and FC layers that embraces three parallelism schemes discussed in Chapter 3. As a result, our host C++ program has 1400 lines of code (LOC), and four OpenCL kernels that execute on FPGA have 400 LOC in total.

5 Results

In this chapter, the evaluation results of the proposed system are described. First, we introduce the hardware and software used in the evaluation. Then, we analyze the accuracy, resource utilization, performance and power consumption of the system for different values of design parameters in ConvNet. After that, we compare our system with the other state-of-the-art FPGA-based ConvNet system. Finally, the comparison between our design and ConvNet implementation on other HPC systems is provided.

5.1 Experimental Setup

The ConvNet model used in the evaluation is VGG 16 and the testing images and pre-trained model are obtained from Caffe deep learning framework [8]. The proposed implementation runs on two FPGA boards with different hardware resources that shown in Table 5.1. The first platform we used is Nallatech P385 board [49] that contains an Intel Stratix V 5SGA7 with 2 x 4GB of 1600 MHz DDR3 memory. The FPGA is from 28 nm node which consists of 622K logic elements (LEs), 234,720 ALMs, 938,880 registers, 2,560 M20K blocks and 256 DSP blocks. An ALM consists of 6 or 8 input Adaptive LUT and other basic components such as adders and registers. The M20K refers to a 20 Kb sized on-chip memory block, which is the basic building block of the local memory. The DSP is a hardwired block supporting variable-precision arithmetic operations. The second platform used in this work is Terasic DE5a-Net board [50] with Intel Arria 10 GX FPGA and 2 x 4GB of DDR3 memory. The Arria 10 GX 10AX115 FPGA, based on TSMC's 20 nm process technology, has 1,150K LEs, 427,200 ALMs, 1,708,800 registers, 2,713 M20K blocks and 1,518 DSP blocks. These DSP blocks in Arria 10 are designed for optimized implementation of IEEE standard single-precision floating point arithmetic. Both accelerator boards are connected to the host via 8-lane PCI Express interface.

We use Intel SDK for OpenCL v15.1 and Nallatech OpenCL Board Support Package (BSP) R001.003.0001 for P385 board. For the DE5a-Net board, the version 1.0.0 BSP from Terasic and Intel SDK for OpenCL v16.0 are used. At the time of writing the thesis, the most recent release of Intel FPGA compiler is v16.1, which could improve Arria 10 series FPGA compilation results. However, the available BSPs for both boards still do not support v16.1.

The Intel Xeon E5-2620 v3 (6 cores with 15MB Cache) running at 2.4 GHz is used for CPU comparison as well as for running the host code of FPGA accelerators. The compiler used to compile the host application is GCC 4.4.7. The CPU reference design is from Caffe framework, running with Intel MKL, which provides optimized linear algebra library. In the CPU implementation, we use single precision IEEE floating point representation, while both single precision IEEE floating point and 16 bit fixed point representation are tested on FPGA implementation. All experiments were run on Red Hat 6.7 workstation.To keep the comparison fair, we evaluate our FPGA design against the GPU-based implementation from [10], where an NVDIA K40 GPU (2880 CUDA cores and 12GB GDDR5 memory) was used to execute VGG 16-SVD network using Caffe Model.

Table 5.1: The Comparison of FPGA Accelerators

Board	P385-A7	DE5a-Net
FPGA	Stratix V 5SGA7	Arria 10 GX 10AX115
Technology	28 nm	20 nm
LEs	622K	1,150K
ALMs	234,720	427,200
Registers	938,880	1,708,800
M20K Blocks	2,560	2,713
DSP	256	1,518
Global Memory	2 x 4GB DDR3	2 x 4GB DDR3

5.2 Performance Analysis of the Proposed Design

In this section, we first measure the accuracy of both implementation with floating point and fixed point representation. Then we explore the design space of both implementations using two FPGA boards and different combination of ConvNet design parameters. The objective is to obtain resource utilization and performance numbers for different combinations of parameters.

5.2.1 The Accuracy Comparison

The accuracy of our implementation was tested by running our models using pre-trained weights from the Caffe tool on first 50K images of ImageNet 2012 validation data set. The top-1 and top-5 accuracies from the Caffe design, single-precision floating point and 16 bit fixed point of FPGA implementation are tabulated in Table 5.2.

Table 5.2: The Accuracy Comparison

Design	Caffe	FPGA Design(Single)	FPGA Design(Fixed)
Data Precision	32 bit	32 bit	16 bit
Top-1 Accuracy	68.15%	68.08%	67.98%
Top-5 Accuracy	88.09%	88.00%	87.75%

The difference between the Caffe tool and single floating point FPGA design on top-1 and top-5 accuracies are 0.07% and 0.09% respectively. The rounded 16 bit fixed point FPGA design achieves 67.98% top-1 accuracy and 87.75% top-5 accuracy, resulting in 0.17% and 0.34% accuracy loss compared to the reference design. Therefore, the accuracy of our implementation is excellent.

5.2.2 The Resource Utilization and Performance

As mentioned in Chapter 4, our design has two main parameters, namely: N_{CU} and N_{SIMD}. The N_{CU} defines the replication number of full data path, which controls the balance between resource usage and throughput. The N_{SIMD} describes vectorization length of the design, which not only improves throughput at the expense of hardware resources but also memory coalescing to increase bandwidth utilization. N_{batch} is another parameter that controls the number of images collected by FC layer to share the weights. We use static configuration on $N_{batch} = 32$ in the design, which is the largest number we could use under limited on-board resource constraints.

We first look into how N_{CU} and N_{SIMD} affect the resource utilization and throughput of the Nallatech A7-based board. The resource usage including the number of ALUTs, LEs, registers, M20K blocks, and DSP blocks for different combinations of parameters are illustrated in Figure 5.1 to 5.5. The corresponding throughput achieved by the same configuration is shown in Fig. 5.6. The design with floating point representation is on the left side, while the design with fixed point data shows on the right side. Due to the limited resources on A7 FPGA, the combination of $N_{CU} = 32$ and $N_{SIMD} = 8$ or 16 does not lead to successful synthesis on floating point version. The similar compilation failure can be found when $N_{CU} > 32$ and $N_{SIMD} > 32$.

The increasing trend is shown by increase in resource usage when two parameters grow. Also, N_{CU} has more impact on resource utilization compared to N_{SIMD}, due to many full data path replication in the system. From the throughput perspective, it is easy to see linear improvement when these parameters increase. The comparison between floating point design and fixed point design also can be summarized. Floating point based design requires more resources especially DSP blocks than fixed point implementation. However, the fixed point design outperform floating point one for all configurations.

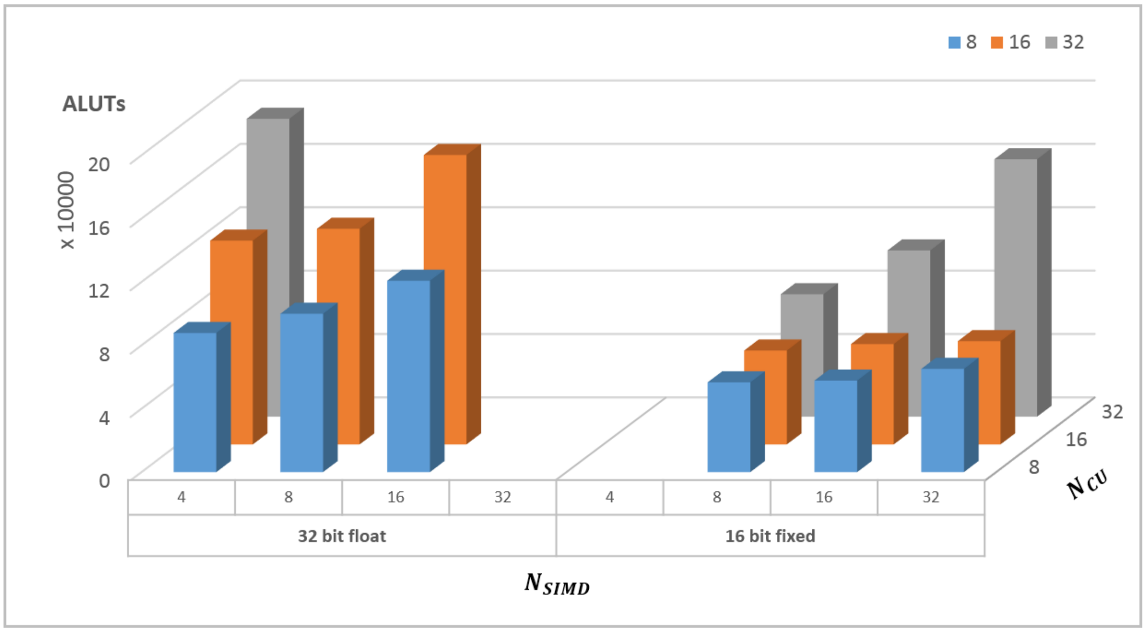

Fig. 5.1: ALUTs Utilization on Stratix A7 with Different Groups of N_{CU} and N_{SIMD}

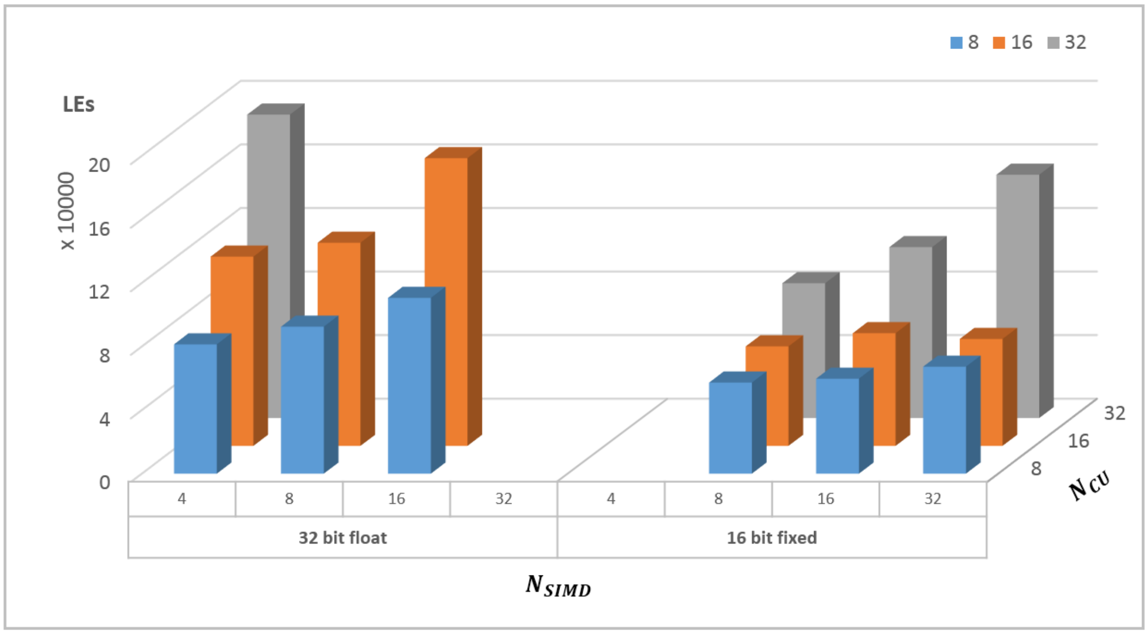

Fig. 5.2: LEs Utilization on Stratix A7 with Different Groups of N_{CU} and N_{SIMD}

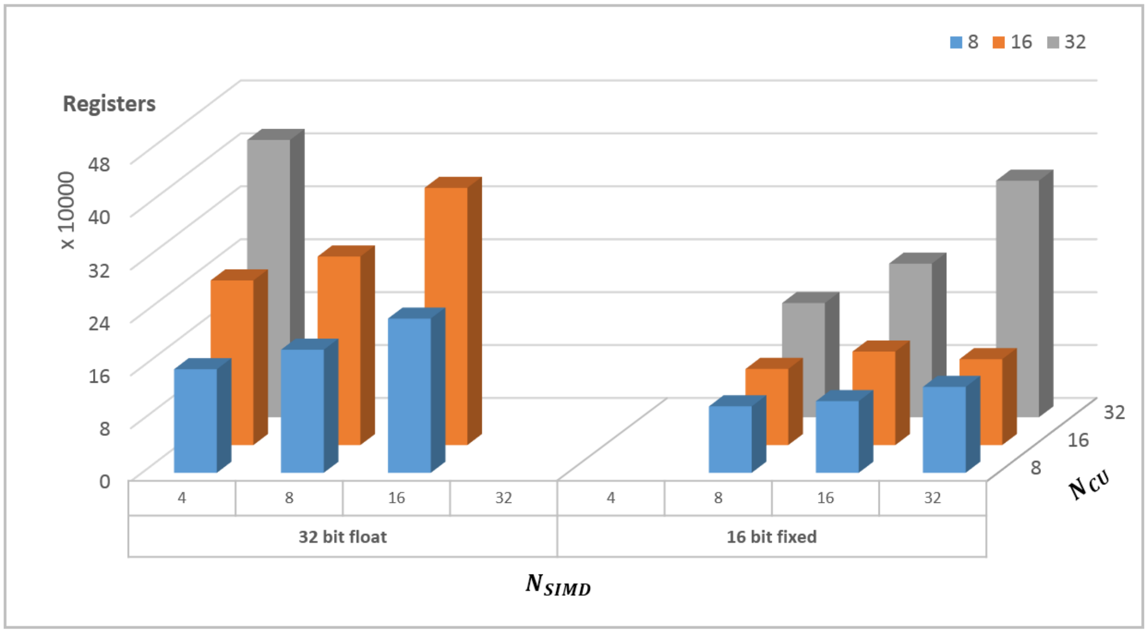

Fig. 5.3: Register Utilization on Stratix A7 with Different Groups of N_{CU} and N_{SIMD}

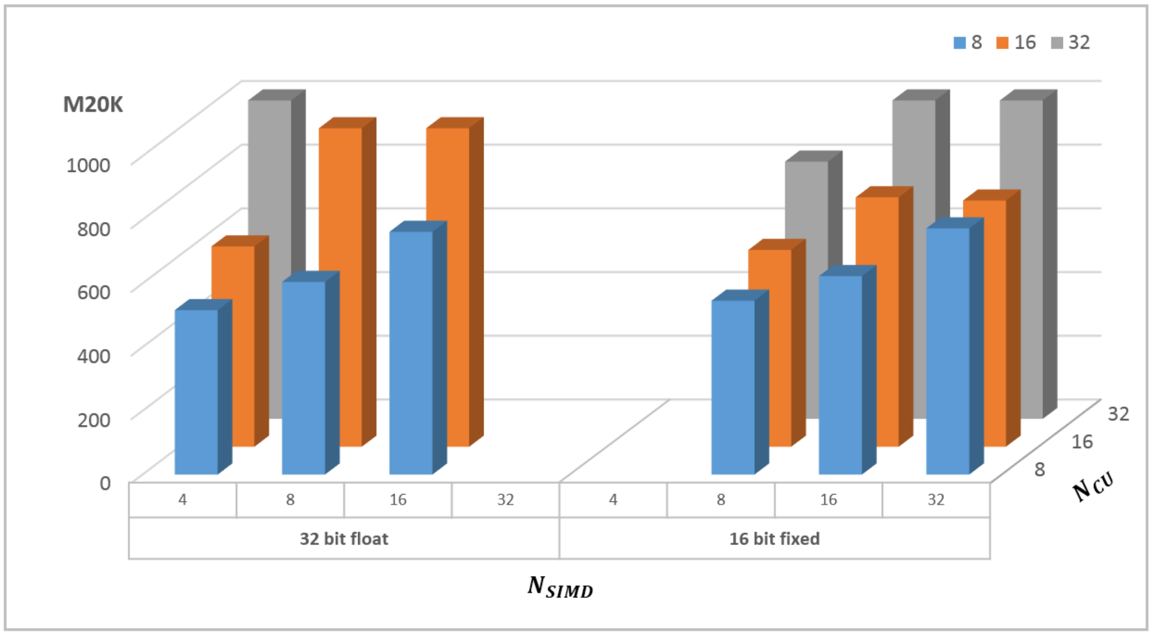

Fig. 5.4: M20K Utilization on Stratix A7 with Different Groups of N_{CU} and N_{SIMD}

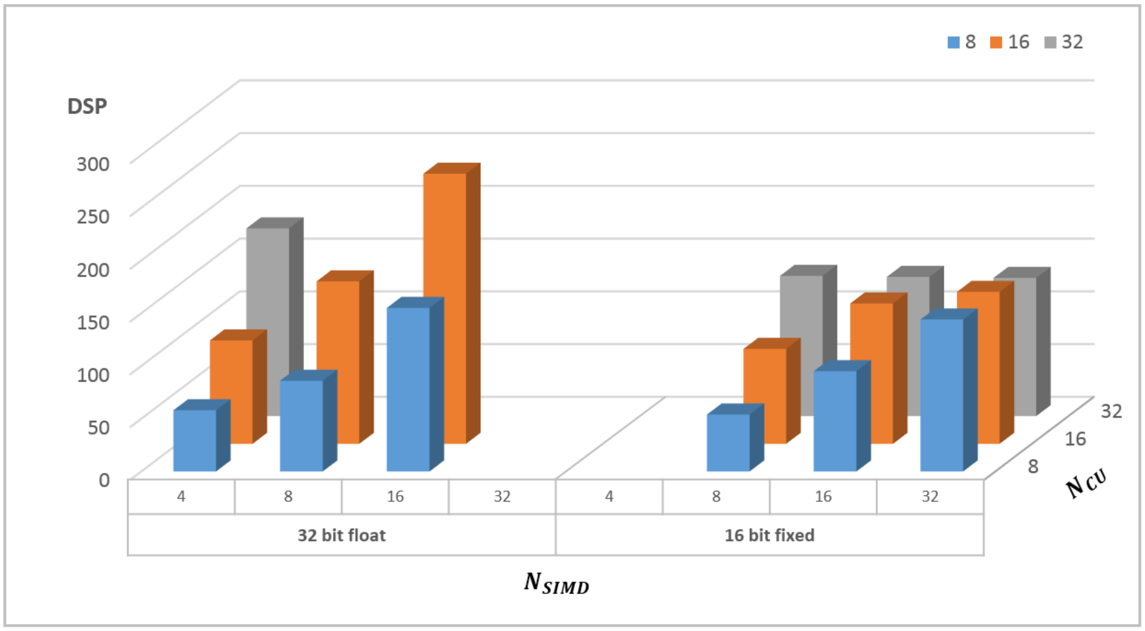

Fig. 5.5: DSP Utilization on Stratix A7 with Different Groups of N_{CU} and N_{SIMD}

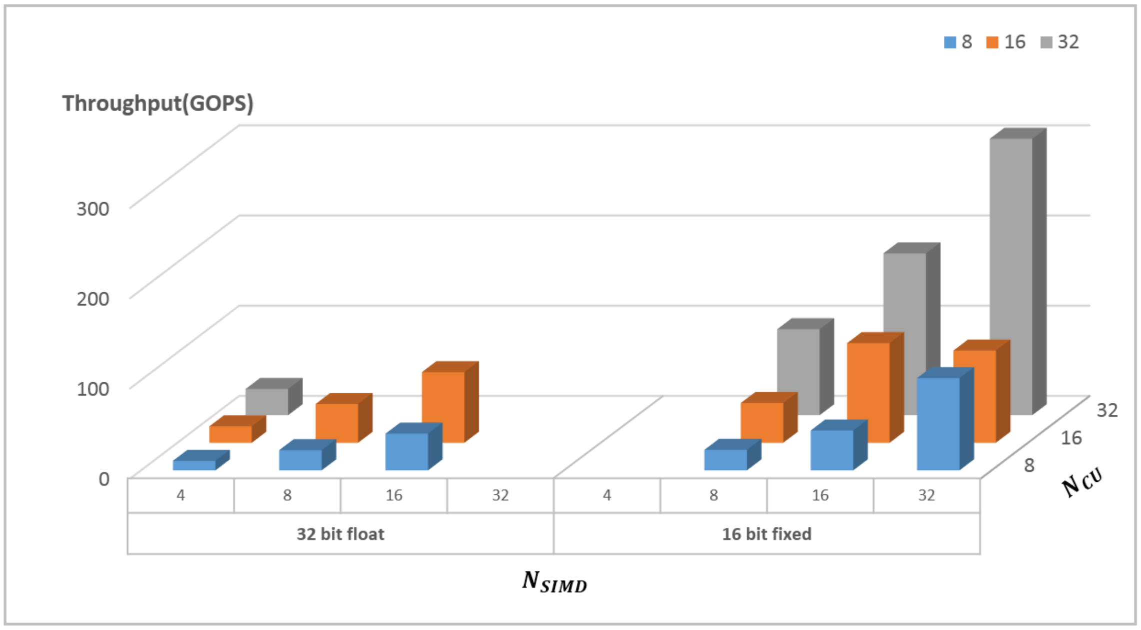

Fig. 5.6: Throughput on Stratix A7 with Different Groups of N_{CU} and N_{SIMD}

Similar experiments were conducted on the Terasic Arria 10-based boards. The resource utilization results are presented in Figure 5.7 to 5.11, and the throughput result is shown in Fig. 5.12. The same trend of A7 is noticed, i.e. throughput increases at the expense of resource usage. The fixed point implementation also achieves higher throughput than floating point design using fewer resources. Compared to A7, Arria 10 has more the DSP blocks which are floating point optimized. Hence, the optimization space is wider than A7, the throughput on floating point representation is nearly 2X greater than the design on A7.

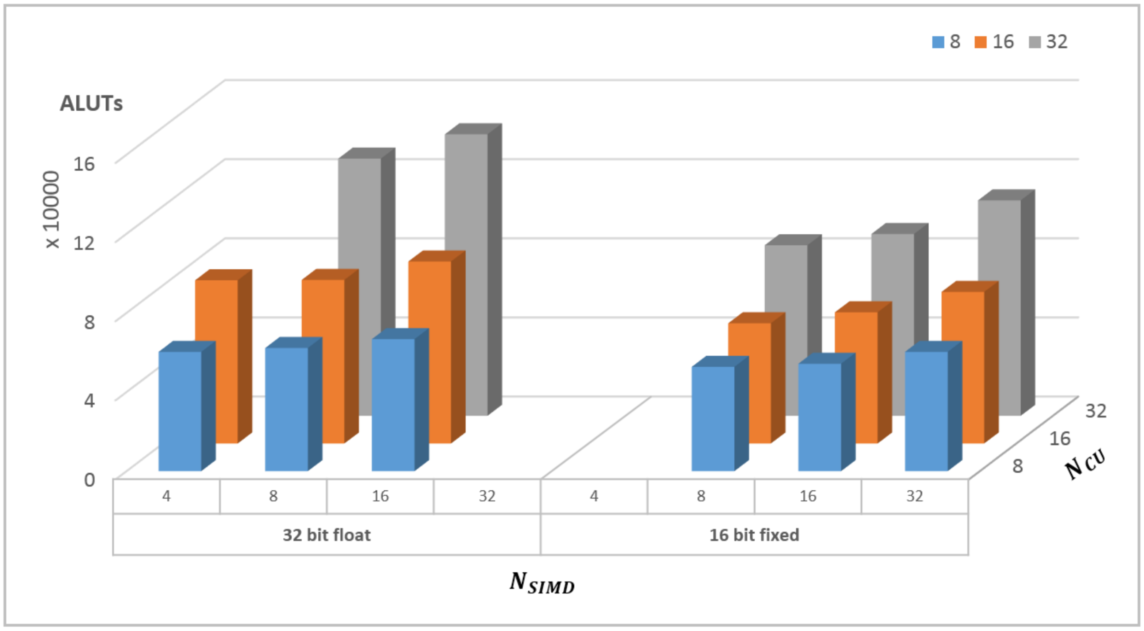

Fig. 5.7: ALUTs Utilization on Arria 10 with Different Groups of N_{CU} and N_{SIMD}

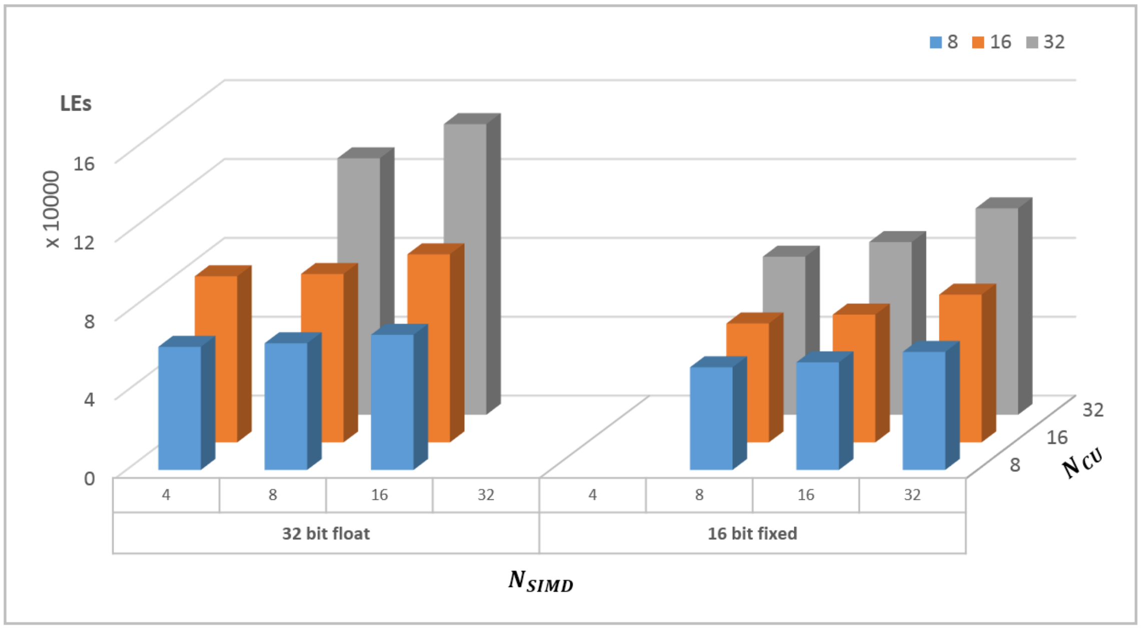

Fig. 5.8: LEs Utilization on Arria 10 with Different Groups of N_{CU} and N_{SIMD}

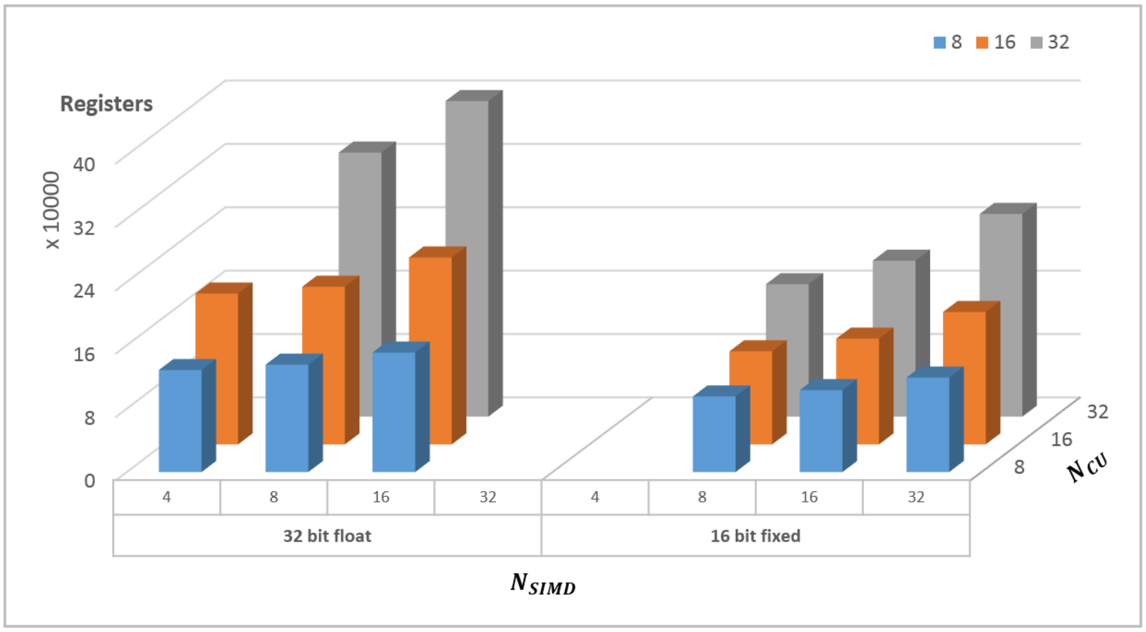

Fig. 5.9: Register Utilization on Arria 10 with Different Groups of N_{CU} and N_{SIMD}

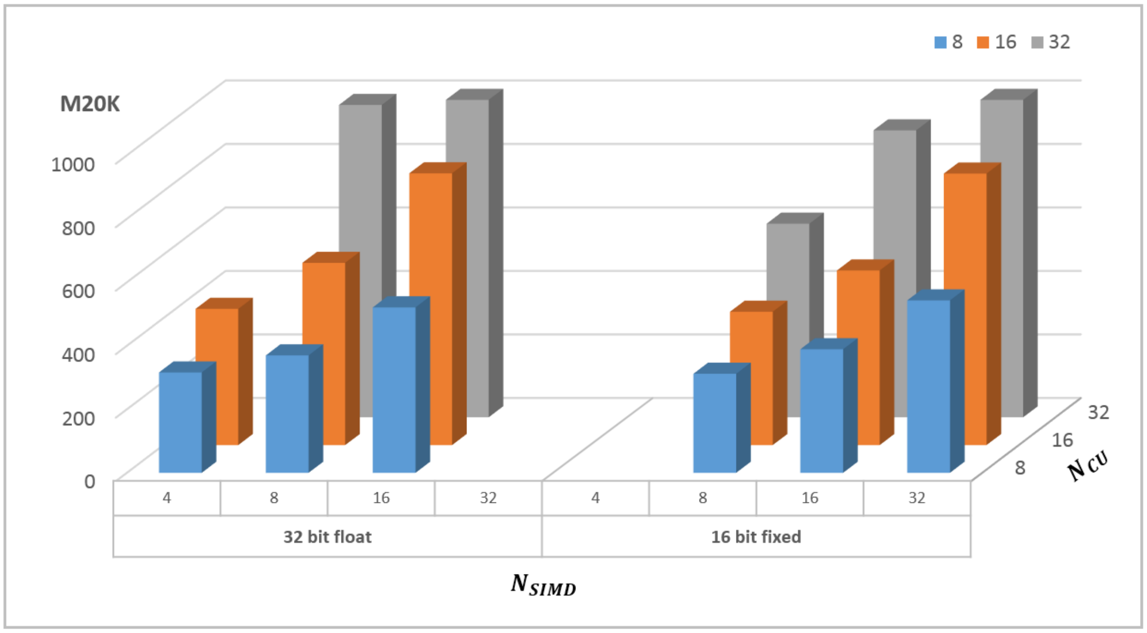

Fig. 5.10: M20K Utilization on Arria 10 with Different Groups of N_{CU} and N_{SIMD}

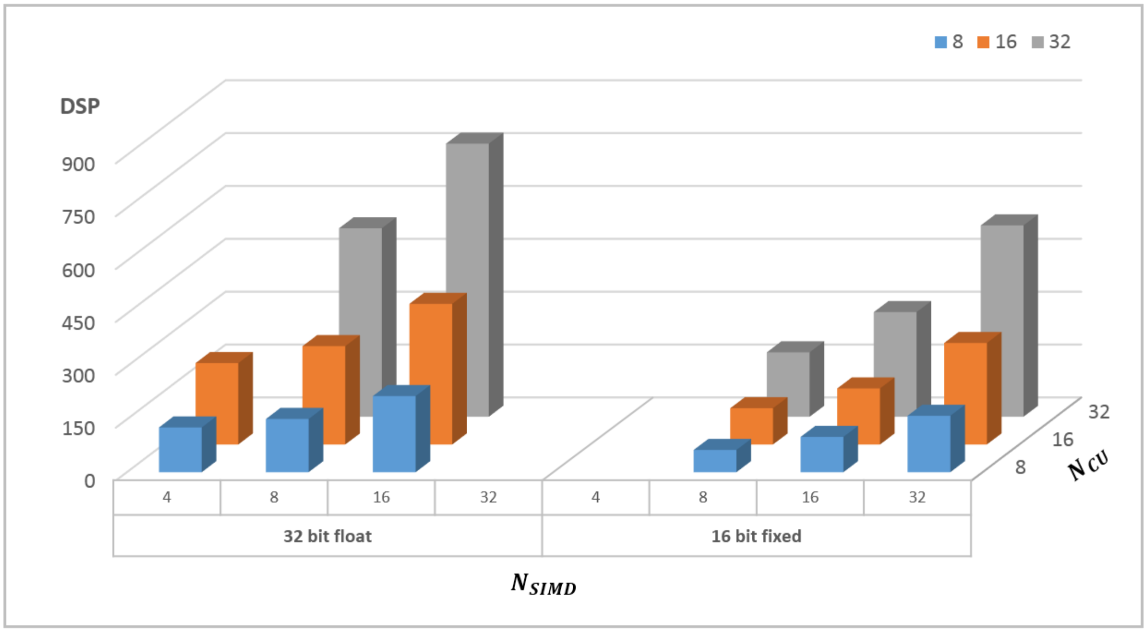

Fig. 5.11: DSP Utilization on Arria 10 with Different Groups of N_{CU} and N_{SIMD}

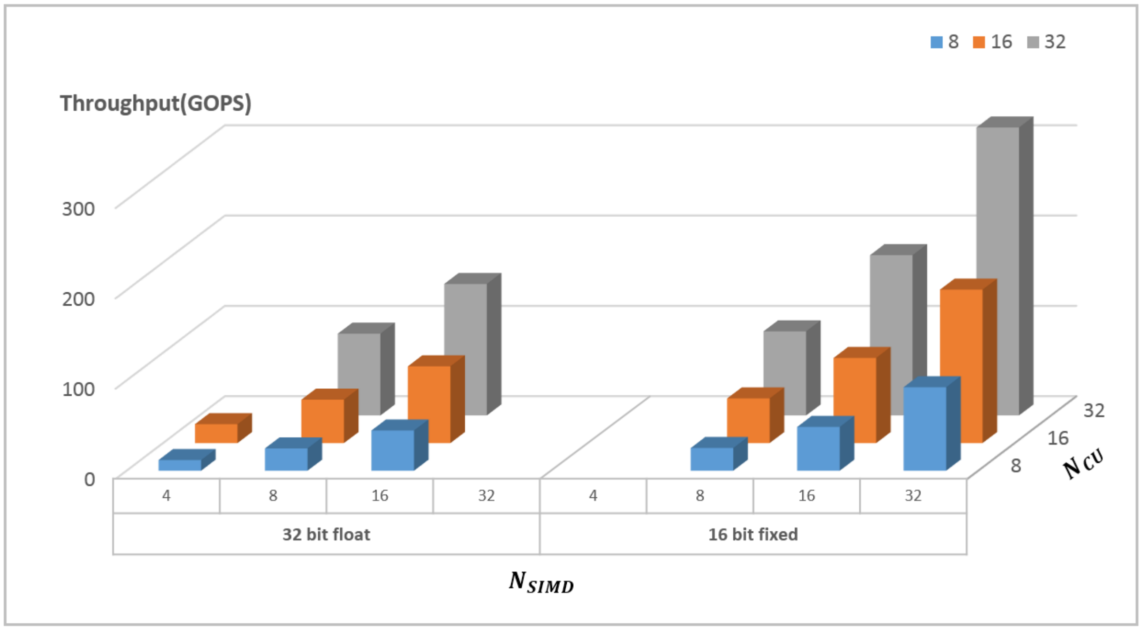

Fig. 5.12: Throughput on Arria 10 with Different Groups of N_{CU} and N_{SIMD}

From the analysis above, the optimal parameter combinations to maximize throughput under board constraints for floating point and fixed point design on both boards are shown in Table 5.3.

Table 5.3: Performance Model on Throughput-oriented Configuration

FPGA	Stratix A7				Arria 10			
Precision	Floating point		Fixed Point		Floating Point		Fixed Point	
Configuration	N_{SIMD}	N_{CU}	N_{SIMD}	N_{CU}	N_{SIMD}	N_{CU}	N_{SIMD}	N_{CU}
	16	16	32	32	16	32	32	32
ALUTs	182,671(77%)		162,487(69%)		142,344(33%)		108,929(25%)	
LEs	181,026(29%)		153,133(24%)		147,484(12%)		104,722(9%)	
Registers	388,809(41%)		356,798(38%)		398,544(23%)		255,988(15%)	
M20K	1,344(53%)		1,899(74%)		1,615(60%)		1,521(56%)	
DSP	256(100%)		131(51%)		775(51%)		543(36%)	
Throughput (GOPS)	78.24		306.41		145.67		318.94	

5.2.3 Power Measurement

For power measurement, we used *Watts up? PRO* [51] power meter. The device is able to measure the power utilization of the whole system and provides power savings of accelerators more precisely. Before FPGA accelerator board is installed into the system, the idle CPU-only system consumes 105.6 W. When executing VGG 16 models using Caffe tools, the average power utilization increases to 137.5 W.

When Nallatech A7-based FPGA board is installed to the system, the idle power consumption is increased to 126.7 W. During execution of ConvNet kernels, total power utilization of heterogeneous system grows to 130.6 W on average. Thus the power consumption on A7-based board for running ConvNet system is (130.6 − 105.6) = 25.0 W. Similar test is conducted on Terasic Arria 10-based board. The idle power utilization of the system is 127.7 W and the running power is 130.1 W. Hence, the average power consumption for executing ConvNet model on Arria 10-based board is (130.1 − 105.6) = 24.5 W. The results from the power measurement are shown in Table 5.4.

Table 5.4: Power Consumption of Accelerators

System	CPU-only System	CPU with A7-based Board	CPU with Arria 10-based Board
System Idel Power (Watt)	105.6	126.7	127.7
System Execution Power(Watt)	137.5	130.6	130.1
Accelerator Power(Watt)	-	25.0	24.5

5.3 Comparison With Previous Research ConvNet

In this section, we first compare our implementation to the previous FPGA work. Then the comparison to similar designs based on other HPC platforms, such as CPU and GPU, are described.

5.3.1 Performance Comparison with FPGA-based Design

Table 5.5 shows the performance of two proposed on two FPGA boards in comparison to several recent FPGA-based ConvNet designs. To calculate throughput, the number of floating point or fixed point operations is divided by overall classification time, and using GOPS as a unit in both floating point and fixed point design.

For the design based on single precision floating point, Zhang et al. [41] implemented convolution layer that achieved 61.62 GOPS. Similarly, DiCecco et al. [43] reported 55 GOPS on a convolution layer, while they implemented the full system. Our throughput from floating point design on Stratix A7 and Arria 10 is 78.24 GOPS and 145.67 GOPS respectively. Note that our floating point design on A7 achieved 1.65X performance compared to 8-16 fixed point design on the same board [9]. The design on Arria 10 takes advantage of its new DSP block which optimized floating point arithmetic operation.

For the implementation using fixed point representation, our proposed architecture also outperforms all of the previous works, by achieving 306.41 GOPS on A7 and 318.94 GOPS on Arria 10. The speedup obtained on A7 over the best previous design [10] is 2.24X by only using 1/6 number of DSP blocks (131 vs 780). Furthermore, the design shows 2.67X better performance than the RTL design on the same board, which shows that OpenCL based design has potential to compete with RTL design. Our design on Arria 10 presents the highest throughput over other designs, while there is still room to improve.

5.3.2 Performance Comparison with Other HPC Platform-based Design

The performance comparison between our FPGA implementations and the work on CPU and GPU is shown in Table 5.6. We chose fixed point design on two boards to compare with the design on CPU and GPU. We also introduce energy efficiency as an evaluation metric, which is the ratio between throughput and power consumption

Table 5.5: The Comparison with Previous FPGA Works

Metrics	**FPGA**	**Method**	**Model**	**Model Size (GOP)**	**Precision**	**Frequency (MHz)**	**DSP Usage**	**Conv Throughput (GOPS)**	**Overall Throughput (GOPS)**
Zhang et al. [41]	Virtex 7 VX485T	Xilinx HLS	AlexNet	1.33	32 bit float	100	2,240	61.62	-
Suda et al. [9](a)	Stratix-V GSD8	Intel OpenCL	VGG	30.9	8-16 bit fixed	120	-	136.5	117.8
Suda et al. [9](b)	Stratix-V GXA7	Intel OpenCL	VGG	30.9	8-16 bit fixed	-	-	-	47.5
Qiu et al. [10]	Zynq XC7Z045	RTL	VGG-SVD	30.76	16 bit fixed	150	780	187.8	136.97
Ma et al. [13]	Stratix-V GXA7	RTL	AlexNet	1.46	8-16 bit fixed	100	256	134.1	114.5
DiCecco et al. [43]	Virtex 7 XC7VX690T	Xilinx OpenCL	VGG	30.9	32 bit float	200	1,307	55	-
This work(a)	Stratix-V GXA7	Intel OpenCL	VGG	30.9	32 bit float	186	256	-	78.24
This work(b)	Stratix-V GXA7	Intel OpenCL	VGG	30.9	16 bit fixed	207	131	-	306.41
This work(c)	Arria 10AX115	Intel OpenCL	VGG	30.9	32 bit float	198	775	-	145.67
This work(d)	Arria 10AX115	Intel OpenCL	VGG	30.9	16 bit fixed	190	543	-	318.94

(GOPS/Watt). From the throughput point of view, GPU is the best choice, followed by two FPGA designs. However, power consumption is another important metric to consider in the contemporary digital design. GPU consumes 10X more power than FPGA and 1.82X more than CPU. When we consider both performance and power, the design on A7 is 1.72X more energy efficient than GPU design. The design on Arria 10 got similar results that 1.82X more efficient than GPU in term of energy consumption.

Table 5.6: The Comparison with Other Platforms

Platform	**CPU**	**GPU[10]**	**FPGA**	**FPGA**
Device	Intel E5-2620	Nvidia K40	Stratix-V GXA7	Arria 10AX115
Technology	22 nm	28 nm	28 nm	20 nm
Frequency	2.4 GHz	1 GHz	207 MHz	190 MHz
Power(Watt)	137.5	250.0	25.0	24.5
Lattency(ms)	445.66	17.25	100.84	96.88
Throughput (GOPS)	69.33	1783.90	306.41	318.94
Energy Efficiency (GOPS/W)	0.50	7.13	12.26	13.02

6 Conslusion

In this work, we present end-to-end ConvNet implementation using Intel FPGA SDK for OpenCL targeting Intel FPGAs. We first describe the optimization strategies on OpenCL-based FPGA design that utilize massively parallelism and high bandwidth memory access patterns. Then the computational and space complexity is quantitatively analyzed, which shows that convolutional layers are computation intensive while the FC layers are memory bandwidth bounded. We use several techniques, including data rearrangement, data quantization, and batch processing to alleviate imbalanced bandwidth requirement of FC layers. Finally, we propose a scalable and parameterized ConvNet implementation with four OpenCL kernels that leverage data parallelism and loop parallelism: *Mapping kernel*, *Reduction kernel*, *Pooling kernel*, and *Output kernel*. These kernels are executed on multiple streams that utilize task parallelism source in ConvNet. Our experimental results show that the proposed design and implementation outperforms all previous FPGA-based work. Also, our optimized work on two different FPGA boards shows 24.3X and 1.7X better energy-efficiency than the solution on modern CPU and mid-end GPU.

As mentioned in Chapter 1, the objective of this thesis is to answer two main question as follows:

- How to optimize HLS design when targeting Intel FPGAs?
- Can FPGAs outperform other HPC platforms on deep ConvNet by using Intel FPGA SDK for OpenCL?

As a conclusion, we have answered our first question by analyzing optimization techniques in Chapter 3. For the second question, we have answered by presenting optimized deep ConvNet FPGA implementation that is more energy efficient than multi-core CPU and mid-end GPU.

6.1 Future Research

In this research, FPGA with limited resources (Stratix A7) has shown comparable results with the mid-range FPGA (Arria GX 10). The reason is that at the time of writing the thesis, the only initial release version of BSP is available which does not support most recent Intel compiler (AOC v16.1). On that version, the compiler is further optimized for targeting A10 FGPA. It would be interesting to synthesize the design with the newer version of the compiler. Additionally, the current A10-based board uses conventional DRAM technology which has limited bandwidth. Intel 10 series FPGA also support Hybrid Memory Cube (HMC) that will deliver ultra-high memory bandwidth like GPUs. It would be interesting to compile the proposed design on these board to see how much improvement could be achieved. Furthermore, Qiu et al. [10] shows an aggressive data quantization strategy by using 4-8 bit fixed representation with dynamic configuration resulting in less than 1% accuracy loss. It would be interesting to apply this strategy on the proposed architecture to see the results. Finally, in GPU-based solution, Mathieu et al. [36] has shown that replacing convolution by FFT is another direction to have increased performance, which also can be tried on the FPGA.

In this research, we mainly focus on forwarding pass of the ConvNet which is the classification. There is also another computation-intensive part in the ConvNet which is the training process. The training of the recent ConvNet models is widely done by GPUs that takes several hours. It could be interesting to see acceleration of the training process on FPGAs.

REFERENCES

[1] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *IEEE Trans. Commun. Technol.*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[3] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, "Large-scale video classification with convolutional neural networks," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2014.

[4] S. Lai, L. Xu, K. Liu, and J. Zhao, "Recurrent convolutional neural networks for text classification," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, ser. AAAI'15. AAAI Press, 2015, pp. 2267–2273. [Online]. Available: http://dl.acm.org/citation.cfm?id=2886521.2886636

[5] O. Abdel-Hamid, A. Mohamed, H. Jiang, L. Deng, G. Penn, and D. Yu, "Convolutional Neural Networks for Speech Recognition," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 22, no. 10, pp. 1533–1545, Oct. 2014.

[6] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," *ArXiv e-prints*, Oct. 2015.

[7] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, “cuDNN: Efficient Primitives for Deep Learning,” *ArXiv e-prints*, Oct. 2014.

[8] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, “Caffe: Convolutional architecture for fast feature embedding,” *arXiv preprint arXiv:1408.5093*, 2014.

[9] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, “Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks,” in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’16. New York, NY, USA: ACM, 2016, pp. 16–25. [Online]. Available: http://doi.acm.org/10.1145/2847263.2847276

[10] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, “Going deeper with embedded fpga platform for convolutional neural network,” in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’16. New York, NY, USA: ACM, 2016, pp. 26–35. [Online]. Available: http://doi.acm.org/10.1145/2847263.2847265

[11] Y.-H. Chen, J. Emer, and V. Sze, “Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks,” *SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 367–379, Jun. 2016. [Online]. Available: http://doi.acm.org/10.1145/3007787.3001177

[12] 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,” *ArXiv e-prints*, Feb. 2016.

[13] 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)*, Aug 2016, pp. 1–8.

[14] K. Krommydas, A. E. Helal, A. Verma, and W. C. Feng, “Bridging the performance-programmability gap for fpgas via opencl: A case study with opendwarfs,” in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2016, pp. 198–198.

[15] S. Craven and P. Athanas, “Examining the Viability of FPGA Supercomputing,” *EURASIP J. Embedded Syst.*, vol. 2007, no. 1, pp. 13–13, Jan. 2007.

[16] F. A. Escobar, X. Chang, and C. Valderrama, “Suitability Analysis of FPGAs for Heterogeneous Platforms in HPC,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 2, pp. 600–612, Feb 2016.

[17] R. Inta, D. J. Bowman, and S. M. Scott, “The ”Chimera”: An Off-the-shelf CPU/GPGPU/FPGA Hybrid Computing Platform,” *Int. J. Reconfig. Comput.*, vol. 2012, pp. 2:2–2:2, Jan. 2012.

[18] I. Kuon and J. Rose, “Measuring the gap between fpgas and asics,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, Feb 2007.

[19] “Top fpga companies for 2013,” http://sourcetech411.com/2013/04/top-fpga-companies-for-2013/, Apr 2013, [Online; accessed Dec 24, 2016].

[20] “Arria 10 architecture,” https://www.altera.com/products/fpga/arria-series/arria-10/features.html, 2016, [Online; accessed Dec 24, 2016].

[21] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "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, Oct 2016.

[22] "Conformant companies," https://www.khronos.org/conformance/adopters/, 2016, [Online; accessed Dec 27, 2016].

[23] A. Munshi, "The opencl specification 1.2," https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf, 2012, [Online; accessed Jan 2, 2017].

[24] K. Morris, "The path to acceleration: Altera bets on opencl," http://www.eejournal.com/archives/articles/20121106-opencl/, Nov 2012, [Online; accessed Jan 2, 2017].

[25] "Opencl on fpgas for gpu programmers," http://design.altera.com/openclforward, Acceleware Corp., 2014, [Online; accessed Jan 3, 2017].

[26] E. Rucci, C. Garcia, G. Botella, A. E. D. Giusti, M. Naiouf, and M. Prieto-Matias, "OSWALD: OpenCL SmithWaterman on Alteras FPGA for Large Protein Databases," *The International Journal of High Performance Computing Applications*, June,2016.

[27] S. O. Settle, "High-performance dynamic programming on fpgas with opencl," in *Proc. IEEE High Perform. Extreme Comput. Conf.(HPEC)*, 2013, pp. 1–6.

[28] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016, http://www.deeplearningbook.org.

[29] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, no. 521, pp. 436–444, May. 2015.

[30] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, November 1998.

[31] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *Artificial Neural Networks and Machine Learning – ICANN 2014: 24th International Conference on Artificial Neural Networks, Hamburg, Germany, September 15-19, 2014. Proceedings*, 2014, pp. 281–290.

[32] "Cs231n convolutional neural networks for visual recognition," http://cs231n.github.io/convolutional-networks/, Stanford, 2016, [Online; accessed Jan 8, 2017].

[33] "Tesla k40 gpu accelerator board specification," http://www.nvidia.com/content/PDF/kepler/, Nvidia, November 2013, [Online; accessed Jan 17, 2017].

[34] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with cots hpc systems," in *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, S. Dasgupta and D. Mcallester, Eds., vol. 28, no. 3. JMLR Workshop and Conference Proceedings, May 2013, pp. 1337–1345. [Online]. Available: http://jmlr.org/proceedings/papers/v28/coates13.pdf

[35] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *ArXiv e-prints*, Apr. 2014.

[36] M. Mathieu, M. Henaff, and Y. LeCun, "Fast Training of Convolutional Networks through FFTs," *ArXiv e-prints*, Dec. 2013.

[37] E. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, "Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation," *ArXiv e-prints*, Apr. 2014.

[38] D. C. Ciresan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber, "Flexible, high performance convolutional neural networks for image classification," in *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, 2011, pp. 1237–1242. [Online]. Available: http://dx.doi.org/10.5591/978-1-57735-516-8/IJCAI11-210

[39] J. Ren and L. Xu, "On Vectorization of Deep Convolutional Neural Networks for Vision Tasks," *ArXiv e-prints*, Jan. 2015.

[40] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2014, pp. 609–622.

[41] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: ACM, 2015, pp. 161–170. [Online]. Available: http://doi.acm.org/10.1145/2684746.2689060

[42] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *ArXiv e-prints*, Sep. 2014.

[43] R. DiCecco, G. Lacey, J. Vasiljevic, P. Chow, G. Taylor, and S. Areibi, "Caffeinated FPGAs: FPGA Framework For Convolutional Neural Networks," *ArXiv e-prints*, Sep. 2016.

[44] "Intel fpga sdk for opencl best practices guide," https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf, 12 2016, [Online; accessed Jan 2, 2017].

[45] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *ArXiv e-prints*, Sep. 2014.

[46] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going Deeper with Convolutions," *ArXiv e-prints*, Sep. 2014.

[47] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and ¡0.5MB model size," *ArXiv e-prints*, Feb. 2016.

[48] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on cpus," in *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

[49] "Nallatech 385 with stratix v a7 fpga," http://www.nallatech.com/store/pcie-accelerator-cards/385-a7/, 2016, [Online; accessed Jan 31, 2017].

[50] "De5a-net arria 10 fpga development kit," https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=231&No=970#https://www.terasic.com.tw/attachment/archive/970/image/front.jpg, 2016, [Online; accessed Jan 31, 2017].

[51] "Watt's up pro power meter specifications," https://www.wattsupmeters.com/secure/products.php?pn=0&wai=303&spec=4, [Online; accessed Feb 3, 2017].

VITA AUCTORIS

NAME:	Huyuan Li
PLACE OF BIRTH:	Yanji, Jilin, China
YEAR OF BIRTH:	1987
EDUCATION:	Beijing Forestry University, Beijing, China Bachelor of Engineering, College of Technology 2006-2010
	University of Windsor, Windsor ON, Canada Master of Applied Science, Electrical and Computer Engineering 2014-2017