NANYANG TECHNOLOGICAL UNIVERSITY

ANALYSIS AND ACCELERATION OF COMPUTE-INTENSIVE APPLICATIONS ON HETEROGENEOUS COMPUTING PLATFORMS

by

SWARNA KAMAKSHI JAYARAMAN
(G1601351B)

A Dissertation Submitted in partial fulfillment of
the requirements for the degree of
Master of Science in Embedded Systems

Supervised by

Assoc. Prof. Douglas L. Maskell

July 2017

Abstract

With the advancements in technology, parallel processing platforms such as graphics processing units (GPUs), massively parallel processor arrays (MPPAs) and multi-core processors are gaining popularity for accelerated execution of compute-intensive applications. However the performance gains achieved by adding more cores inside a computing platform come at the cost of rapidly scaling complexities to the inter-core communication, memory coherency and, most importantly, the power consumption. Increasing the operating frequency or the number of cores does not yield the performance desired for the current complex, compute-intensive applications. This calls for heterogeneous architectures that achieve the desired performance by integrating specialized processing abilities required for specific tasks, after extensive analysis of an application and its performance demands. For example, the performance demand for first convolution layer in MNIST digit classifier is 300 million multiply-accumulate (MAC) operations per second for sub-millisecond latency, which can go up to 300 billion MAC/s for sub-microsecond latency.

This report intends to investigate two compute-intensive applications, namely MNIST Digit classifier using Convolutional Neural Network (CNN), and Fully Homomorphic Encryption (FHE). For MNIST-CNN, we observe the performance of 24 million MAC/s on Intel Core i3 running at 2.3 GHz using a single-threaded C++ code which can go up to 2 billion MAC/s using OpenCL on the same platform. To achieve sub-microsecond latency, 20 convolution engines (running at 600 MHz) can be used where each engine is able to produce one pixel by performing 25 MAC operations every clock cycle. The peak performance of such a hypothetical engine is 300 billion MAC/s. A similar analysis is presented for the second application (FHE). Apart from the theoretical analysis, experiments pertaining to architectural and algorithmic explorations are also carried out for these two applications.

Acknowledgment

Firstly, I would like to extend my deepest gratitude to my supervisor, Assoc Prof Dr. Douglas Leslie Maskell for giving me an opportunity to work on my area of interest. His deep insight in the field, enthusiastic support and invaluable suggestions helped me progress through my project work.

I am also grateful for the effective knowledge sharing sessions I have had with my mentor, Dr. Abhishek Kumar Jain. His patient reviews, constructive feedback, constant encouragement and timely help steered me in the right direction and helped finish my thesis on time.

I am greatly indebted to Mr. Gopalakrishna Hegde, a former research assistant at NTU, for his inputs and assistance during the first phase of the project. Special thanks to Mr. Prashant Ravi, a former M.Sc. student under Prof Douglas, who helped me understand the rudiments of the project field during the project exploration phase, gave warm-up exercises for me to get a hang of the work ahead and eased the tool setup.

My sincere thanks to Mr. Jeremiah Chua from the Hardware and Embedded Systems Lab (HESL) for the lab facilities and technical support.

Last but not the least, I would like to thank my family for their prayers and support in my pursuit of higher education.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Contribution . . . 4
- 1.3 Organization . . . 4

2 Background **6**
- 2.1 What is Hardware Acceleration? . . . 6
- 2.2 Heterogeneous Platforms . . . 7
  - 2.2.1 Intel Platform with CPU and GPU . . . 7
  - 2.2.2 Avnet Zedboard with Xilinx Zynq 7000 . . . 7
- 2.3 Programming Models for Hardware Acceleration . . . 8
  - 2.3.1 GPGPUs . . . 8
    - 2.3.1.1 Device Model . . . 9
    - 2.3.1.2 Memory Model . . . 9
    - 2.3.1.3 Execution Model . . . 10
  - 2.3.2 Field Programmable Gate Arrays . . . 12
    - 2.3.2.1 High-Level Synthesis . . . 12
    - 2.3.2.2 OpenCL . . . 14

3 Hardware Acceleration of Convolutional Neural Networks **17**
- 3.1 Deep Learning using Convolutional Neural Networks . . . 18
  - 3.1.1 MNIST Digit Recognition using Lenet-5 ConvNet . . . 20
    - 3.1.1.1 Convolution Layer [1] . . . 21
    - 3.1.1.2 MaxPool Layer . . . 24
    - 3.1.1.3 Inner Product Layer . . . 25
    - 3.1.1.4 ReLU Layer . . . 26
    - 3.1.1.5 Softmax Layer . . . 26
    - 3.1.1.6 Modified Hyperparameters for MNIST Dataset 27

3.1.2 Experiments with C++ Code . . . 27
3.1.2.1 Prerequisites . . . 27
3.1.2.2 Existing Code Flow Description . . . 29
3.1.2.3 Improvements . . . 35
3.1.3 Experiments with OpenCL Code . . . 36
3.1.3.1 Pre-requisites . . . 36
3.1.3.2 Existing Code Flow Description . . . 39
3.1.3.3 Modifications to remove Platform Dependencies 42
3.1.3.4 Compiling and executing the code . . . 44
3.1.4 Comparative Study of Results . . . 46

4 Hardware Acceleration of Fully Homomorphic Encryption 51
4.1 Fully Homomorphic Encryption Scheme . . . 51
4.1.1 Background . . . 52
4.1.2 Existing code flow . . . 55
4.1.3 Hot-Spots for Hardware Acceleration . . . 56
4.1.3.1 Dimensionality Analysis . . . 57
4.1.4 Results . . . 59
4.1.4.1 Hand-optimized non-recursive 1-D FFT . . . 61
4.1.4.2 MachSuite FFT . . . 62
4.1.4.3 Cooley Tuckey Radix-2 DIF FFT . . . 64
4.1.4.4 Comparison of Execution Times . . . 66

5 Conclusion and Future Work 68
5.1 Conclusion . . . 68
5.2 Future Work . . . 69

Appendix A CNN 71

Appendix B FHEW 75

List of Figures

1.1 The Productivity Gap [2] 2

2.1 The OpenCL Platform Model [3] 9
2.2 The Device Model of OpenCL [4] 10
2.3 An Illustration of Data-parallel execution [4] 11
2.4 Performance vs. Design Time with RTL Design [5] 12
2.5 Performance vs. Design Time with HLS Compiler [5] 13
2.6 Pipeline Parallelism achieved by OpenCL-to-FPGA compiler [6] 15

3.1 Types of Machine Learning Algorithms [7] 18
3.2 Formula to determine Classifier Efficiency [8] 19
3.3 Learning differences - Traditional vs. Deep Learning [9] 20
3.4 Original Lenet-5 ConvNet Architecture; Each plane represents a feature map in which weights are shared. [8] 21
3.5 Convolution Layer [10] 22
3.6 Preserving Input dimensions using Zero Padding [10] 23
3.7 3-dimensional volume of Neurons [10] 24
3.8 Max Pooling [1] 24
3.9 Inner Product Layer [1] 25
3.10 ReLU Function in Neural Networks [11] 26
3.11 Softmax Function [12] 26
3.12 Lenet-5 CNN Architecture for MNIST Dataset with modified hyperparameters [13] 27
3.13 Software Code Flow [13] 30
3.14 OpenCL Device Query code Output [14] 38
3.15 OpenCL code flow for Sample (single image recognition) mode [15] 40
3.16 Kernel Compilation Modes [16] 41

3.17 Default (bus-interleaved) vs. Manual Global Memory Partitioning [17] 42
3.18 Linking libOpenCL Library to Makefile 44

4.1 FHE: Problem Statement [18] 52
4.2 Homomorphic Encryption Scheme : Example [19] 54
4.3 Homomorphic Decryption: Example [20] 55
4.4 Cycle of simple NAND operation [21] 55
4.5 Precision Loss with single-precision float 60
4.6 Area vs. Latency plot for Radix-2 DIF FFT 65
4.7 Post-implementation resource utilization summary for Radix-2 DIF FFT 65

List of Tables

3.1 Hyperparameters for Lenet-5 CNN described in MNIST/Lenet-5 ConvNet Benchmark code[13] . . . 28
3.2 Analysis of Application Hot-spots for Acceleration [22] . . . 32
3.3 Inferences from Table 3.2 for Convolution Layer 1 running on CPU . . . 33
3.4 Inferences from Table 3.2 for Convolution Layer 2 running on CPU . . . 35
3.5 Test Device Specifications . . . 47
3.6 Comparison of kernel runtime in various OpenCL Devices . . . 48
3.7 Inferences from execution times of OpenCL devices running offloaded Convolution Layer 1 code (Calculations similar to Tables 3.3 and 3.4) . . . 49

4.1 Analysis of Software Bottlenecks . . . 57
4.2 Dimensionality Analysis (Section 6.2, [21]) . . . 58
4.3 Software Computation Time . . . 58
4.4 Design Space Exploration for non-recursive 1-D FFT . . . 62
4.5 Pre-implementation Design space exploration for Machsuite FFT . . . 63
4.6 Post-implementation execution time for MachSuite benchmark FFT . . . 63
4.7 Pre-implementation Design Space Exploration for Radix 2 DIF-FFT . . . 64
4.8 Post-implementation execution time for Radix-2 DIF FFT . . . 66

A.1 Removal of Altera device-specific Macros . . . 71
A.2 Loading kernel from Source . . . 72
A.3 Initialization of OpenCL Objects for Altera FPGA (Taken from Altera Design Examples) . . . 73

A.4 Initialization of OpenCL Objects for a GPU Device 74

B.1 Non-recursive 1-D FFT 75
B.2 Radix-2 DIF FFT 76

Chapter 1

Introduction

1.1 Motivation

Moore's legendary law: "The number of transistors and resistors on a chip doubles every 18 months," which predicts the pace of technology scaling is largely mistranslated to imply CPU performance. Although conventional scaling techniques have challenged the tacit promises of performance put forth by Moore, Intel - co-founded by Moore himself - has found novel ways to steadily stride along his prognosis, an achievement that can be attributed to a legion of engineers. However, as we approached infinitesimally small-sized transistors, we chalked up paltry performance gains and came to terms with the fact that purely upgrading hardware generations is not the solution.

The Figure 1.1 illustrates the gap between the computational demand with increasing complexity and the actual productivity. Increasing the operating frequency or the number of cores does not yield the performance desired from the current complex, compute-intensive applications.

Heterogeneous system architectures exploit multiple processor types, to realize the best of both data-parallel and task-parallel (sequential) applications

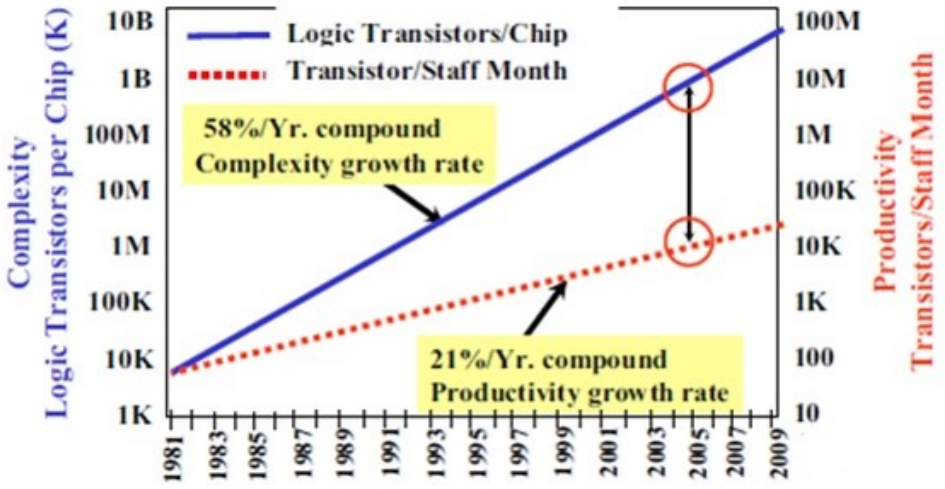


Figure 1.1: The Productivity Gap [2]

[23]. To realize the full potential of such systems, the system designers should scrupulously integrate the diverse compute elements available in the platform and allow them to work together seamlessly. While conventional accelerators limit productivity by demanding high skills in hardware engineering, heterogeneous architectures having GPPs and specialized co-processors offer software-like programmability, enhanced application portability and consequently, improved productivity. Some of the popular heterogeneous computing platforms include network processors such as Intel IXP, Embedded systems such as TI OMAP, NVIDIA Tegra and Apple A9, Reconfigurable devices such as Xilinx FPGAs (Virtex, Kintex, etc.) in Zynq platform containing dual core ARM Cortex A9 Processors, General purpose processors such as IBM Cell and ARM big.LITTLE CPU architectures [24].

These devices guarantee a power-aware design with increased data parallelism and throughput. However, we should also be mindful of the challenges they pose such as different instruction set architectures for different compute elements, varying cache structure and coherence protocols, varying mem-

ory access patterns (uniform or non-uniform) and interface types [24], etc. Opaque programming paradigms (like POSIX standard for threads), the differences in underlying micro-architecture and abstractions associated with high-level language programming can impede performance predictions and sometimes, increase power consumption. Designers should explicitly handle thread synchronization and shared variable protection in multi-threaded applications and partition the application suitably between various computing elements. Example: Running a sequential task on FPGA leads to underutilization of resources and slows down the performance. Similarly, performing SIMD operations on a CPU would be a bad choice. Design decisions generally involve domain expertise and design space exploration, which is a quantitative approach to recognizing the design variables with the most beneficial effect on the system's performance goals.

To mitigate the challenges listed above, we need to establish a standard programming model that is portable across devices and capable of delivering the desired performance. For simple applications, design decisions are straightforward. This prompts us to explore some complex, compute-intensive applications and study the impact of various design decisions on their performance. This report deals with the study of two such compute-intensive applications, namely MNIST Digit classifier using Convolutional Neural Network (CNN), and Fully Homomorphic Encryption (FHE), their efficient partitioning between hardware and software and gauging of various design points to achieve a specific optimization goal, i.e. low latency. For example, the first convolutional layer of the MNIST-CNN having a computation demand of 300K multiply-accumulate operations can be processed at 24M MACs/s with single-threaded flow on Intel Core i3 platform, whereas the same platform can process upto 2B MACs/s with multi-threaded OpenCL execution. To achieve sub-microsecond latency, 20 convolution engines (running at 600 MHz) can be used where each engine is able to produce one

pixel by performing 25 MAC operations every clock cycle. The peak performance of such a hypothetical engine is 300 billion MAC/sec. In a broader sense, a mere 14 frames/s with sequential flow can be boosted to about 826 frames/s on the same CPU while a cross-platform performance of as high as 3105 frames/s can be accomplished with parallel thread executions. We also present similar analysis for second application (FHE). Here, we focus mainly on the algorithmic design exploration for a specific architecture using high-level synthesis tools.

1.2 Contribution

The primary focus areas of this thesis can be summarized as follows:

- Identification and understanding of two applications with high computational complexity which show potential for hardware acceleration.
- Understanding of programming models best suited for the parallelization of the identified complex applications.
- Profiling various parts of the applications to isolate the critical paths that need improvement.
- Performing architectural exploration and suitably partitioning the application between various compute elements available in the platform.

1.3 Organization

The report consists of the following chapters: **Chapter 2** presents background knowledge, software and hardware requirements needed for this thesis. **Chapter 3** delves into the C++ and OpenCL implementation of MNIST Digit Recognition program and studies the runtime benefits with parallel execution. **Chapter 4** explains another complex encryption algorithm imple-

mented in software that exhibits potential for hardware acceleration. **Chapter 5** draws conclusion to the contents perused in this thesis and throws light on potential direction for future research.

Chapter 2

Background

2.1 What is Hardware Acceleration?

Migration of some applications running on a general purpose CPU, to custom hardware acceleration engines, to resolve inherent bottlenecks of the system and improve system performance is referred to as hardware acceleration [25]. Such specialized accelerators intend to improve portions of the code that incur significant performance overheads such as:

- Mathematically rigorous functions with more data dependence and reduced control dependence among operations,
- Repeated routines on different data sets,
- Other parallelizable tasks, etc.

Some common real-world scenarios demanding the computation bandwidth of hardware accelerators are Audio Codec applications, high-speed Video Streaming, Network protocols, Cryptanalysis, Data mining, Natural Language Processing, Computer Vision, etc. [26] The goal is to accomplish a faster execution time in hardware than in software. The hardware execution time includes the actual computation time by the accelerator as well as the communication overheads associated with reading and writing back the data.

2.2 Heterogeneous Platforms

Heterogeneous computing platform constitutes different kinds of processors on the same silicon die. Commonly found constituents of an embedded system platform are a general-purpose processor (CPU) and a few specialized co-processors designed for a specific purpose. Examples of co-processors are Digital Signal Processors, which provide Instruction Level parallelism with VLIW, SIMD and superscalar capabilities, GPGPUs and FPGAs. The heterogeneous devices that were used for this project are listed below:

2.2.1 Intel Platform with CPU and GPU

This platform has been chosen to demonstrate code portability, analyze whether the given application is control-bound or compute-bound, evaluate the application runtime in CPU and GPU, and estimate the improvement in latency. The **Intel SDK for OpenCL**[27] is available for both Windows and Linux Operating Systems and offers packages to run applications on Intel CPU and GPU. Also, the **OpenCL Runtime Environment** (RTE) [28] provides drivers and library packages required to test applications while they are running. The installation of these packages will be discussed in detail in Section 3.1.3.1.

2.2.2 Avnet Zedboard with Xilinx Zynq 7000

This platform comprises of a Processing System with dual-core ARM Cortex A9, running at 667 MHz with NEON SIMD engine and Floating Point Unit, and a Programmable Logic with Artix-7 FPGA. The processing system and programmable logic are connected via AXI Interface. Zedboard has found its place in different market segments, be it Automotive, Consumer Electronics, software-defined Radio applications [29], Aerospace and Defense, Medical diagnostics and Imaging, Wired and Wireless communication, Control and

Bridging applications [30]. Owing to its versatility, this platform has been chosen to conduct experiments on the complex applications at hand.

2.3 Programming Models for Hardware Acceleration

The various programming models that have been explored in this thesis are discussed below. The models have been chosen with the view to reducing the burden on the engineers to learn coding at lower levels of abstraction while also achieving unparalleled performance.

2.3.1 GPGPUs

The first half of this thesis delves into the use of GPGPUs for applications other than their conventional role in computer graphics. The most commonly used programming languages for GPU programming are Open Computing Language (OpenCL) and CUDA. It is interesting to note that CUDA implementations currently support only one vendor, NVIDIA Corporation while OpenCL supports the vendors AMD, Intel, Altera, NVIDIA and Apple.

While OpenCL is open-source, CUDA is proprietary. After a basic run-through of the features of both frameworks such as code portability and flexibility, OpenCL programming model was chosen to carry out the acceleration experiments. The prime focus of this thesis is on OpenCL C APIs, which are maintained by the Khronos group [31]. The OpenCL architecture is composed of a Host which dispatches commands to the devices. The host CPU offsets loads to the devices and the devices execute these workloads for the host.

There are three popular OpenCL Models [4], which shall be discussed briefly to aid the understanding of internals in OpenCL Programming.

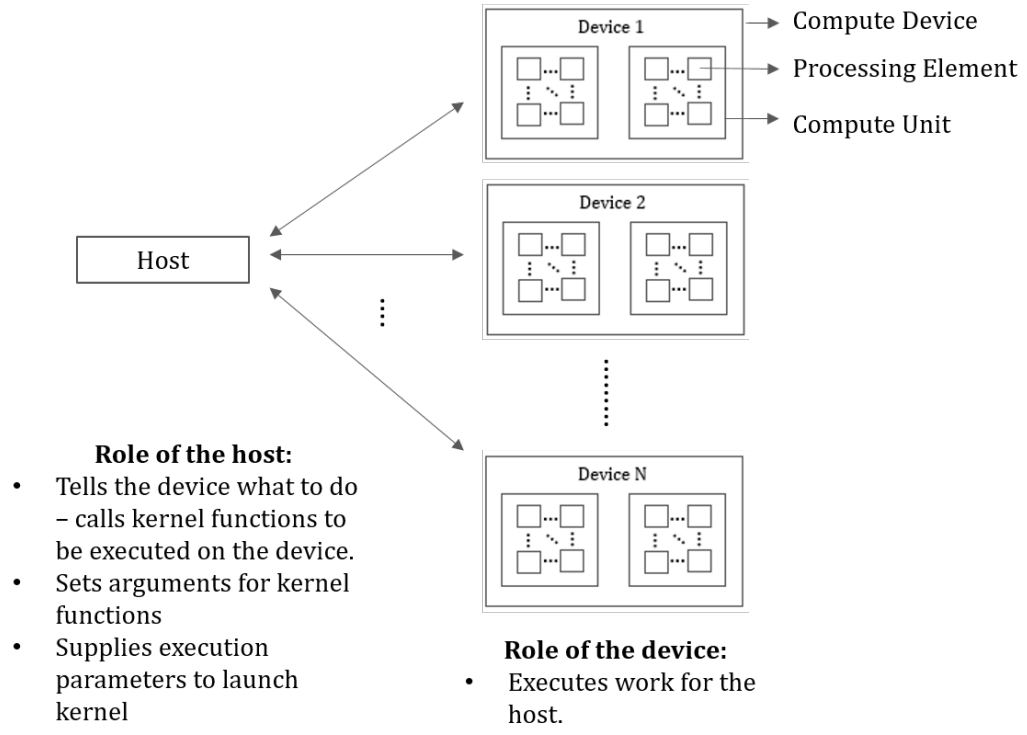


Figure 2.1: The OpenCL Platform Model [3]

2.3.1.1 Device Model

It is an abstract view of various components in a Compute device. Each device consists of various Compute Units, and each of those units are composed of several Processing Elements (PEs). Hence, Compute units can be viewed as containers of very simple processors (PEs).

2.3.1.2 Memory Model

It defines the memory hierarchy inside an OpenCL device.

- **Global Memory:** Persistent storage accessible by all Processing Elements (PEs) and the host.

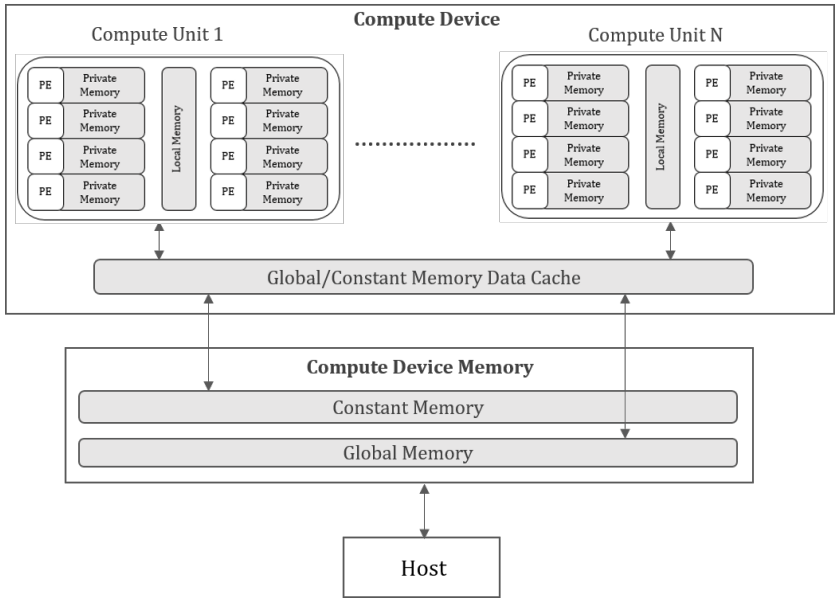


Figure 2.2: The Device Model of OpenCL [4]

- **Constant Memory:** Non-persistent, Read-Only Memory shared among all Processing Elements.
- **Local Memory:** Shared by all PEs in one Compute Unit and not available to PEs from other compute units. Each Compute Unit has its own local memory.
- **Private Memory:** Non-persistent memory accessible by a single Processing Element.

2.3.1.3 Execution Model

OpenCL **kernels** are ordinary functions with special signatures written in OpenCL C, which run on each Processing Element. For data-parallel applications where the same function is invoked several times, the kernels execute concurrently on different PEs over a pre-defined N-dimensional index space [32].

A **work item** is an independent element of execution. It can also be interpreted as the invocation of the kernel for a specific index "i". The **global work size** defines the number of work items per work dimension (dimension of the index space).

The host describes an N-dimensional computational load where each index point is represented by a work item. The work items are grouped into **work groups** by the host and each of these work groups execute in parallel within the compute unit. The work group size is device-dependent and can be found by querying the device using OpenCL APIs. Each Compute Unit has its own work-group(s) and each work item in the work group is executed by a single processing element.

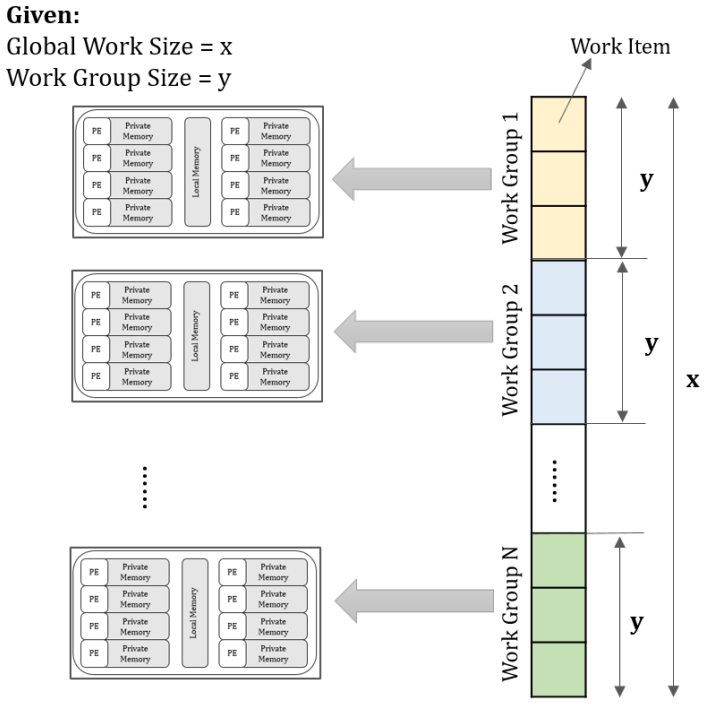


Figure 2.3: An Illustration of Data-parallel execution [4]

2.3.2 Field Programmable Gate Arrays

2.3.2.1 High-Level Synthesis

Until recently, we were directing our attention to programming in specialized processors using high-level languages such as C and C++. With growing computational demand, a sudden shift in focus to FPGAs necessitated the hardware programming knowledge among software engineers.

The Figure 2.4 depicts implementation time for various programming models and we notice that RTL design, although the most beneficial in terms of performance compared to standard and specialized processors, demands the highest development time, beyond the acceptable software development time, to capture the market. This can be attributed to the increased concretization in the design at lower-levels and the deficit of hardware programming experience and expertise. To relieve the engineers of this burden and improve the

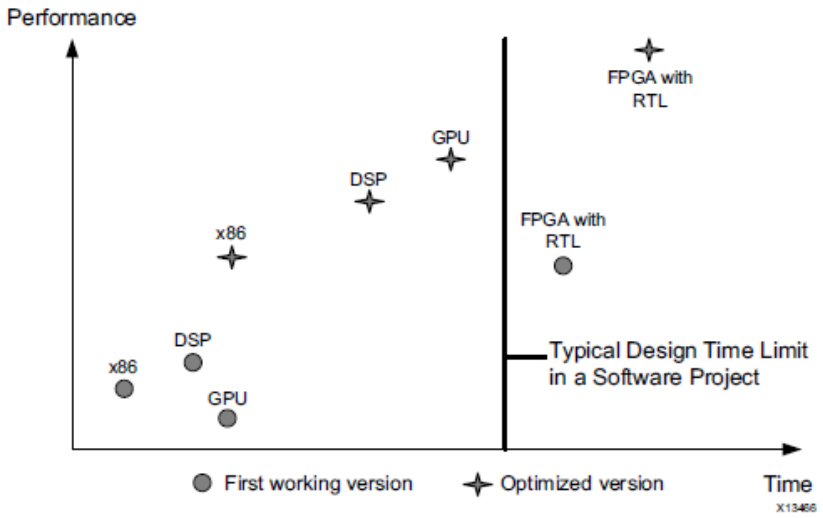


Figure 2.4: Performance vs. Design Time with RTL Design [5]

time-to-market, High-Level Synthesis tools which eliminate the differences in

programming models of processors and FPGAs have been introduced. HLS tools translate a C/C++ specification into an equivalent RTL description. The Figure 2.5 illustrates the performance peaks that can be accomplished with High-Level Synthesis, in comparison to standard processors and GPUs. It is only fair that we acknowledge the fact that RTL code automatically gen-

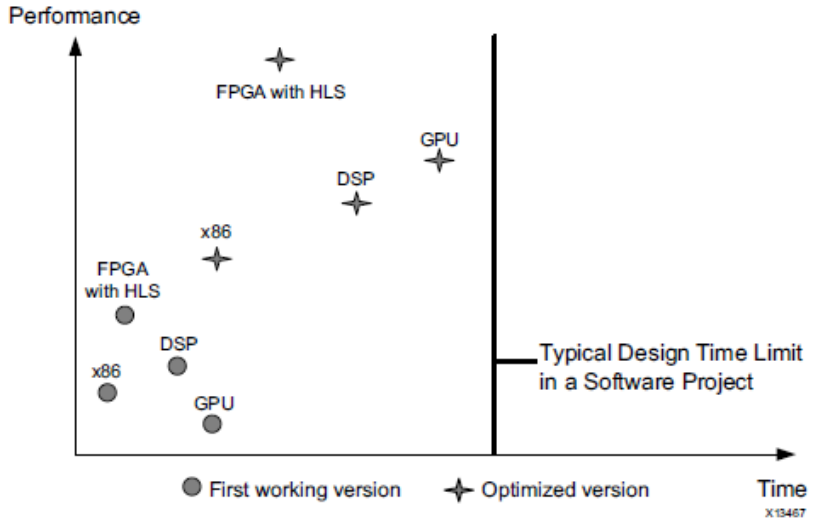


Figure 2.5: Performance vs. Design Time with HLS Compiler [5]

erated by HLS tools may not be the most optimal implementation. It may not fully exploit the parallelism offered by the underlying hardware, unlike the design with HDL languages. However, it meets the time limit specified for software development in many cases and hence proves very useful in that respect.

Pointers are supported in HLS when they can be completely described at compile-time, without any need for runtime intelligence. FPGA-based designs using HLS demand the data and size of memory blocks to be deterministic at compile-time. This static memory allocation facilitates realization of an algorithm's memory as a register, FIFO or Block RAM [5].

Register-based memory implementation is the fastest as a register is an independent entity, which doesn't require any addressing logic. FIFOs are used to transfer data between loops and functions. It is a queue with a single entry and exit point. FPGAs have dedicated Random-access memory blocks called Block RAMs which retain values for as long as the system is powered on. Block RAMs support parallel access of two different memory locations.

HLS tools provide easy testing of functional correctness in both C and RTL implementations and offer numerous optimization directives, which when aptly used, help accomplish multi-objective optimizations.

2.3.2.2 OpenCL

OpenCL standard facilitates implementing parallel algorithms at higher levels of abstraction on FPGAs as opposed to traditional low-level programming using Hardware Description Languages such as VHDL and Verilog [33]. The drawbacks of High-level Synthesis tools in this respect is that they take in a sequential C description and try to extract thread-level parallelism out of it. Failure to gain the maximum parallelism beats the purpose of using an FPGA. Thus, OpenCL standard allows spawning of threads and annotating them with explicit constructs that describe parallelism and memory access hierarchy (execution parameters discussed in Figure 2.1).

Unlike the CPU-GPU platform where concurrent threads are run on different cores, kernels are translated to equivalent dedicated circuits which implement each function in the hardware. These circuits are wired appropriately to simulate the dataflow in the kernel [6]. The final circuit implemented on FPGAs is heavily pipelined and exhibits multi-threading capabilities, offering a final design with pipelined parallelism.

In conventional RTL design, the designers should handle cycle-wise hardware

descriptions, create data paths, create FSMs for control flow, manage timing constraints and integrate low-level IP cores to the design, all by themselves. OpenCL Compiler automates these steps and helps shift the focus to refining the algorithm rather that detailing the hardware design. OpenCL being a cross-platform standard can be easily carried forward to different FPGA generations with little design effort, while the benefits of improved capabilities and performance remain intact [6]. Figure 2.6 depicts the pipelined execution

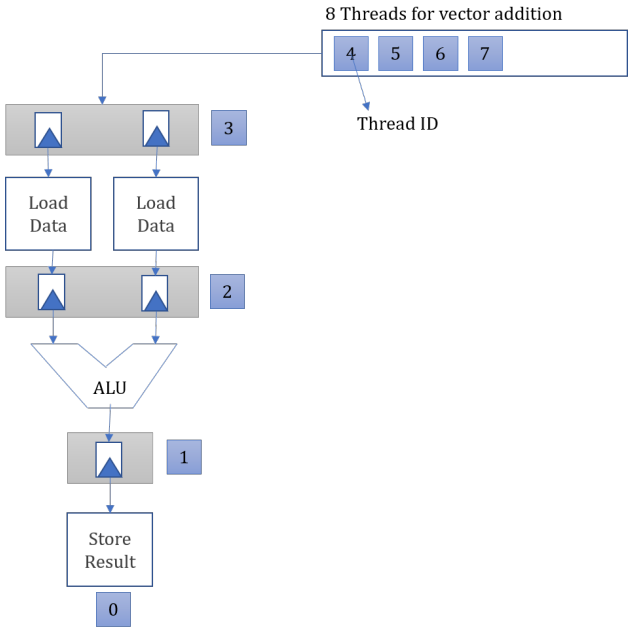


Figure 2.6: Pipeline Parallelism achieved by OpenCL-to-FPGA compiler [6]

of the 8 threads by the generated circuit. Assuming there are three pipeline stages, at cycle 3, we observe that thread 0 stores the computed result, thread 1 computes the sum for a new set of data values, thread 2 copies the values read from memory while thread 3 reads data from the memory. Thus, at any point during the execution, all pipeline stages manipulate a different thread and all stages of the pipeline are active, until the processing of all threads are complete.

Some FPGA vendors like Xilinx and Altera offer OpenCL SDKs for FPGAs. We are not using the Altera Toolchain for our experiments, but the benchmark code taken for test relies on some platform-independent C++ headers (aocl_utils) and Quartus II Emulator that are available with this SDK. Hence, this thesis shall make use of Altera OpenCL (AOCL) SDK to analyze, modify the code and study the results.

Chapter 3

Hardware Acceleration of Convolutional Neural Networks

Deep Learning is an avant-garde approach to imparting knowledge to the machines to achieve the ultimate goal of artificial intelligence without explicit coding, and bridge the current gap between technology integration and expertise. It is of interest in several domains[34], such as:

- Self-driving cars, Automated flight control, Handwriting and Voice recognition software, which are real-time and cannot be programmed by hand or require intense effort doing so manually.
- Database Mining.
- Applications with Product Recommendations in e-commerce websites such as Amazon and Netflix, which are essentially self-customizing.
- Understanding of the human genome.
- Anti-Spam filters and Intelligent Search bars in browsers.

Claims have been made that off-the-shelf accelerators in the embedded platforms offer an edge over CPU-based systems in deep learning computations[35].

We seek to validate the efficiency of deep-learning methods on heterogeneous architectures with a simple Lenet-5 Model of MNIST Dataset classifier.

3.1 Deep Learning using Convolutional Neural Networks

The Figure 3.1 shows the most common types of learning algorithms. The choice of the algorithm depends on the problem we intend to solve.

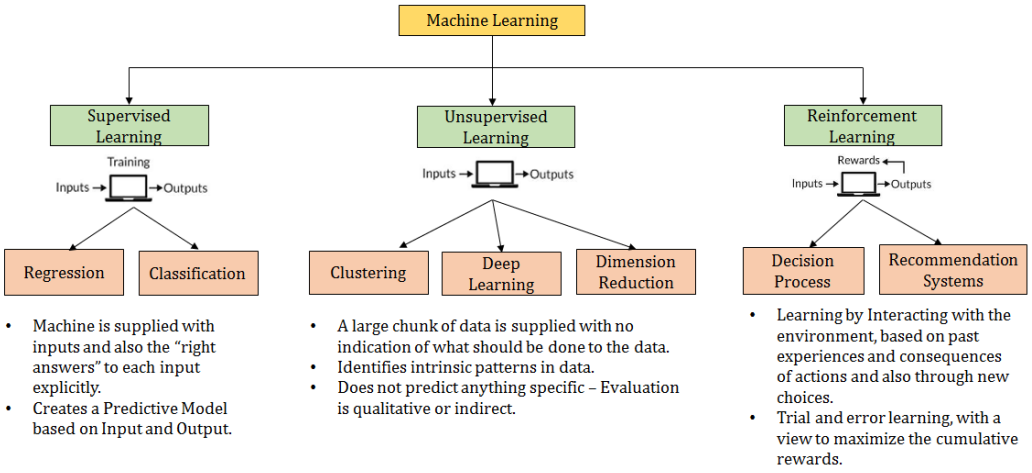


Figure 3.1: Types of Machine Learning Algorithms [7]

Various experiments have substantiated the claim that Convolutional Neural Networks (also called ConvNets or CNNs) outperform other gradient-based learning techniques in handling variable input dimensions in the 2-dimensional space [8]. Multilayer ConvNets with back-propagation can be exploited to build a strong decision layer capable of classifying data of high dimensionality, with minimal processing.

Any character recognition system is comprised of the following two parts:
1. **Feature Extractor** –

It transforms the input into low-dimensional feature vectors which comprise of only the relevant information of interest from the huge input data [36]. The chosen features are essentially invariant to the transformations and distortions that are applied to the input.
Feature extraction attempts to reduce the complexity that stems from high input dimensionality, by downsizing the data while still accomplishing reasonable accuracy in the description of data [36]. Feature extractors are application-specific.

2. **Classifier** –
It is a trainable general-purpose entity which analyzes the data and categorizes the feature vectors appropriately into classes. The accuracy of a classifier is predominantly decided by the features selected in the feature extraction process.
The efficiency of a classifier is determined not just by the correctness in categorizing a given set of test input samples but also the error rate.

$$E_{test} - E_{train} = k\left(\frac{h}{P}\right)^{\alpha},$$

Where,

E_{test} – *Expected error rate on the test set*
E_{train} – *Error rate on the training set*
k – *constant*
h – *measure of effective capacity or complexity of the system*
P – *Number of training samples*
α – *a number between* 0.5 *and* 1

Figure 3.2: Formula to determine Classifier Efficiency [8]

Studies have revealed the relationship between expected error rate on test set and error rate on training set as shown in Figure 3.2. The difference between these two error values decreases as the number of training samples increases. Also, if the complexity of the system "h" increases, training error decreases. Hence, we infer that the system becomes more robust with more training.

The traditional machine learning approach involves handcrafting features of interest, which can take painstaking amount of time and effort, coupled with domain expertise. Feature engineering in Deep nets is automatic and more accurate in comparison to conventional methods [9].

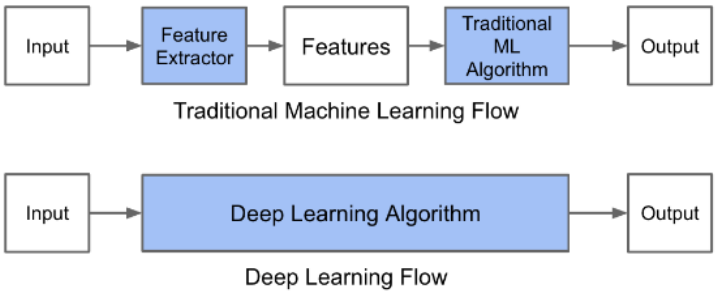


Figure 3.3: Learning differences - Traditional vs. Deep Learning [9]

Owing to high computational complexity, CNN usage is restricted, especially in portable devices [37].

3.1.1 MNIST Digit Recognition using Lenet-5 ConvNet

The Lenet-5 Architecture for handwritten digit recognition was first conceived by LeCun et al. in 1998. The MNIST(Modified National Institute of Standards and Technology) database consisting of 60000 training samples and 10000 test inputs available for download in [38] was used for the experiments discussed in the paper [8]. This paper proved the general consensus 3.3 that ConvNets eliminate the need for hand-made feature extractors and are the most efficient. Today, Artificial Intelligence is a buzzword and almost all AI related applications are leveraging ConvNets to achieve the best performance with low runtime complexities. Figure 3.4 shows the original Lenet-5 architecture described in [8]. It is important to understand the purpose of

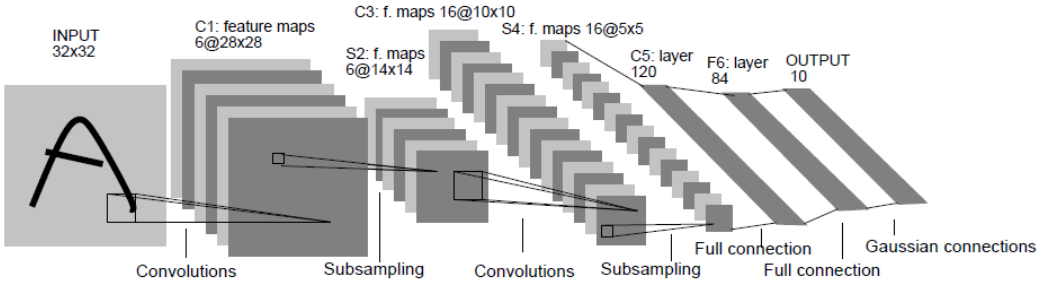


Figure 3.4: Original Lenet-5 ConvNet Architecture; Each plane represents a feature map in which weights are shared. [8]

various layers of the Lenet-5 ConvNet architecture to study their runtime in the application. The following subsections shall describe the layers in detail.

3.1.1.1 Convolution Layer [1]

Convolution Layer is the core of a ConvNet. Consider an input volume of height H_i, width W_i and depth D_i. The depth indicates the color channels, i.e. the third dimension of input volume which can be activated. A filter of dimension F $\times$ F is slid over the input image spatially to evaluate dot products between the input image volume and the filter, thus generating 2-dimensional activation maps. The filter spans through the depth of the input image.

Activation map is a visualization of which portions of the input volume are responding to the filter. For example, if the filter is intended to filter out vertical lines, activation map is representative of filter activations on the image. i.e. it contains all portions of the image which are likely to have vertical lines. Usually, several filters, also called kernels are convolved with the input image, resulting in several activation maps stacked in the depth dimension. In a ConvNet, there are several convolution layers and intuitively, they build up an entire feature hierarchy.
Each stage builds up very specific features which filters in the subsequent

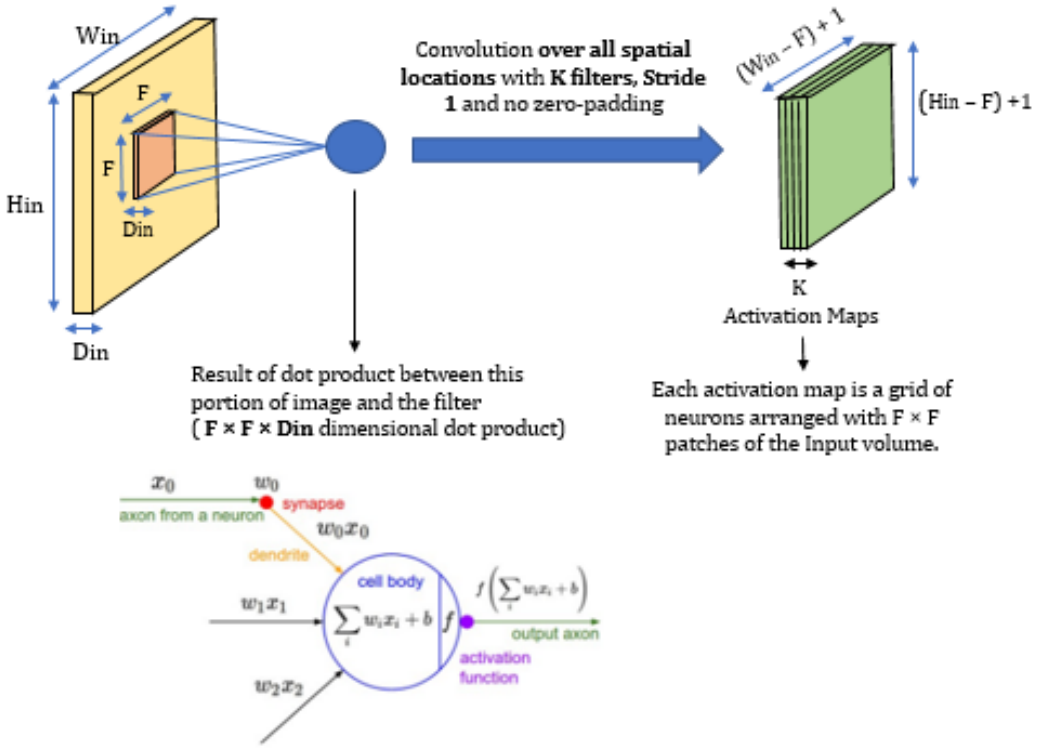


Figure 3.5: Convolution Layer [10]

stages will be excited about. i.e. piece by piece, we create 3-D volumes of higher levels of abstraction than the previous stage[39].

Generalization of Concepts:

Required Hyperparameters:

Number of filters, K

Spatial Extent of the filter, F

Stride, S

Quantum of Zero padding, P (Figure 3.6)

Input Dimensions: $W_1 \times H_1 \times D_1$

Output Dimensions: $W_2 \times H_2 \times D_2$,

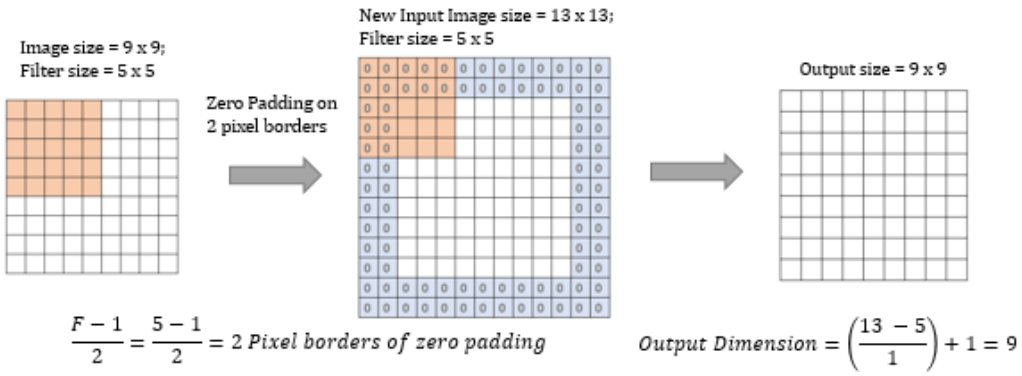


Figure 3.6: Preserving Input dimensions using Zero Padding [10]

Where
D_2 = K
W_2 = ((W_1 – F +2P)/S) + 1
H_2 = ((H_1 – F +2P)/S) + 1

Each filter has an associated bias. The value 1 is added in the above formulae to account for that bias. Stride is the distance by which the filter is slid around the input volume.
Hence, total number of parameters introduced in the neural network is given by (F . F. D_1) × K weights and K biases. For computational convenience, K is usually set as powers of 2. Some libraries branch into special routines when encountering powers of 2, and these routines are highly optimized and efficient for computations in a vectorized form [10].

The output of a filter covering a particular region of the input x can be interpreted to be a neuron fixed in space, which computes w^Tx + b. The connections of the neuron are localized and this connectivity expands up to the receptive field of the neuron, given by the filter size F × F. An activation map can be perceived as a grid of neurons with shared weights and representing the dot products of each F × F patch of the input volume. As there

can be multiple filters in a single convolution layer, the resultant output is a 3-D volume of neurons, as illustrated in Figure 3.7. This 3-D volume has shared parameters spatially (H×W – within the same depth slice) but across depth, the parameters are different. The neurons illustrated in the Figure 3.7 are all acting on the same input patch but with different weights.

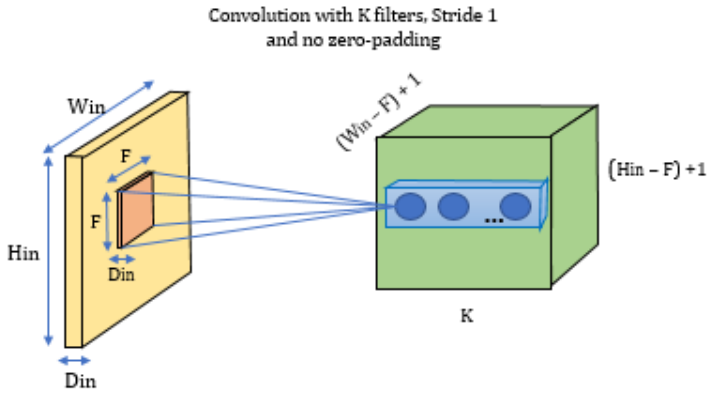


Figure 3.7: 3-dimensional volume of Neurons [10]

3.1.1.2 MaxPool Layer

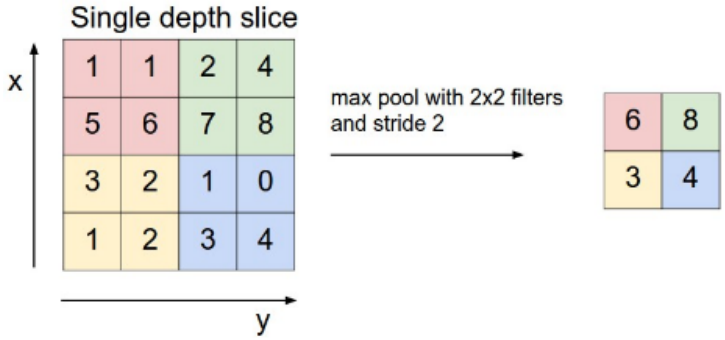


Figure 3.8: Max Pooling [1]

Down-sampling layer which operates independently on all activation maps.

Required Hyperparameters:

Spatial Extent of the filter, F

Stride, S

Input Dimensions: $W_1 \times H_1 \times D_1$

Output Dimensions: $W_2 \times H_2 \times D_2$

Where

$W_2 = ((W_1\text{-F})/\text{S})+1$

$H_2 = ((H_1\text{-F})/\text{S})+1$

$D_2 = D_1$

Example of Maxpool operation with filter size 2×2 and Stride 2 is illustrated in Figure 3.8.

3.1.1.3 Inner Product Layer

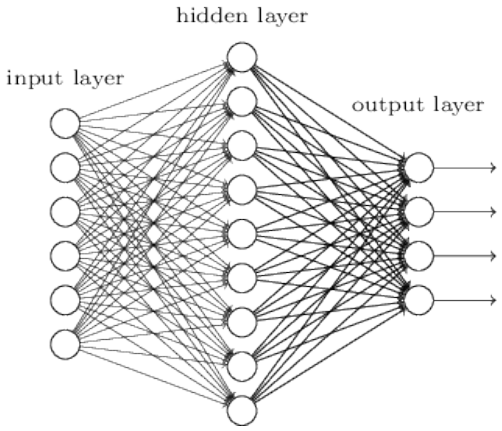


Figure 3.9: Inner Product Layer [1]

It is also called the fully connected layer as the neurons of this layer are pairwise fully connected with the neurons of the previous (input) layer. The neurons within the same layer do not share connections.

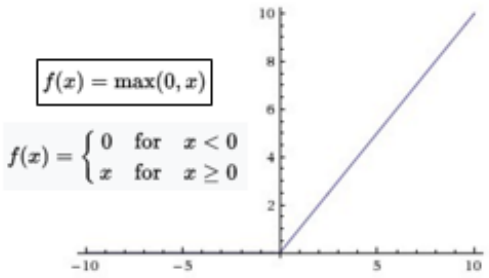


Figure 3.10: ReLU Function in Neural Networks [11]

3.1.1.4 ReLU Layer

Rectified Linear Unit [40] is a non-linear activation function described by Figure 3.10, commonly used in neural networks for the purpose of thresholding after convolution. ReLU is faster compared to other activation functions such as sigmoid and tanh units as it does not involve any normalization or exponential calculation, unlike its counterparts.

3.1.1.5 Softmax Layer

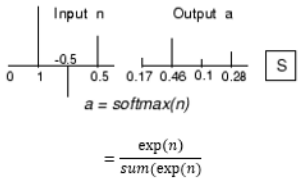


Figure 3.11: Softmax Function [12]

MNIST Digit Classifier has ten class labels for the ten digits 0 to 9, which are mutually exclusive. An ideal classifier should assign a probability of 1 to one of the ten possible nodes at the output and assign 0 probability to others. Due to difficulty in realizing this, we use Softmax function usually in the last layer of the ConvNet, which increases the probability of the maximum value

from the previous stage in such a way that sum of the output probabilities of the 10 classes is 1 [41].

3.1.1.6 Modified Hyperparameters for MNIST Dataset

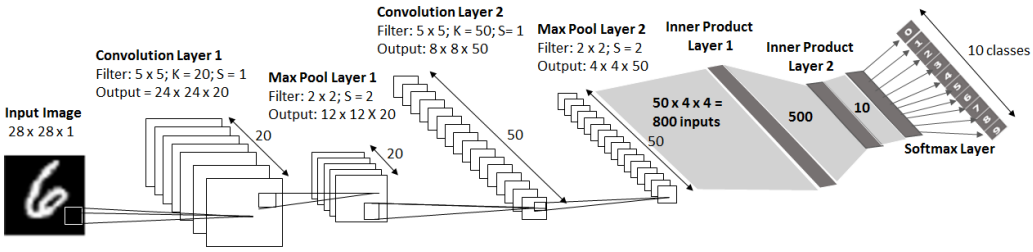


Figure 3.12: Lenet-5 CNN Architecture for MNIST Dataset with modified hyperparameters [13]

Different versions of MNIST Datasets have been introduced over the years. The first version had images centred within the 28×28 region and it was extended to 32×32 images by adding extra background pixels [8]. In the later versions of the database, images were normalized in size to fit a 20×20 field, forming the centre of mass of the resultant 28×28 image. The Architecture illustrated in the Figure 3.4 uses 32×32 images while the benchmark code [13] that will be used for our experiments uses 28x28 images. Table 3.1 defines the hyper-parameters for the different layers of the MNIST/Lenet-5 CNN benchmark application[13].

3.1.2 Experiments with C++ Code

3.1.2.1 Prerequisites

Performance Application Programming Interface (also called PAPI) offers interfaces to hardware performance counters in the underlying platform. These counters count the number of occurrences of a specific event or signal related to the functioning of the processor. This library is used to benchmark the test application and can be installed as follows:

Layers	**Input Dimensions**	**Hyper-parameters**	**Output Dimensions**
Convolution Layer 1	$W_1 \times H_1 \times D_1$= 28×28×1	F = 5, S = 1, K = 20, P = 0	W_2 = ((28-5)/1)+1 = 24 H_2 = ((28-5)/1)+1 = 24 W_2 = 20
MaxPool Layer 1	$W_1 \times H_1 \times D_1$= 24×24×20	F = 2, S = 2	W_2 = ((24-2)/2)+1 = 12 H_2 = ((24-2)/2)+1 = 12 W_2 = 20
Convolution Layer 2	$W_1 \times H_1 \times D_1$= 12×12×20	F = 5, S = 1, K = 50, P = 0	W_2 = ((12-5)/1)+1 = 8 H_2 = ((12-5)/1)+1 = 8 W_2 = 50
MaxPool Layer 2	$W_1 \times H_1 \times D_1$= 8×8×50	F = 2, S = 2	W_2 = ((8-2)/2)+1 = 4 H_2 = ((8-2)/2)+1 = 4 W_2 = 50
Inner Product Layer 1	(4×4×50=800) $W_1 \times H_1 \times D_1$= 1×1×800 (Vector of matrices)	Number of Outputs = 500 (defined in lenet5Model.h)	500 (Vector of float values)
ReLU Layer	500	-	500
Inner Product Layer 2	500	Number of Outputs = 10 (defined in lenet5Model.h)	10
Softmax Layer	10	-	10

Table 3.1: Hyperparameters for Lenet-5 CNN described in MNIST/Lenet-5 ConvNet Benchmark code[13]

```
$ sudo apt-get install papi-tools
```

Download PAPI files from the official PAPI Website [42].

```
$ wget http://icl.cs.utk.edu/projects/papi/downloads/
   papi-5.5.0.tar.gz
```

Extract the tar file and open the directory:

```
$ tar -zxvf papi-5.5.0.tar.gz
$ cd papi-5.5.0
```

Follow the steps specified in the file INSTALL.txt inside the PAPI directory. As the Makefile is not already available, we create the Makefile using the command:

```
$ sudo ./configure
```

After the creation of Makefile, compile and link the library using the command (spawn as many parallel threads as is supported by the number of CPUs in the system):

```
$ sudo make -j24
```

To check for errors, perform a simple test:

```
$ sudo make test -j24
```

To run all the available test programs:

```
$ sudo make fulltest -j24
```

Navigate to the directory when the benchmark code using PAPI is located and link the code to PAPI library by setting the following environment variable:

```
$ export LD_LIBRARY_PATH=/usr/local/lib
```

3.1.2.2 Existing Code Flow Description

Figure 3.13 shows the code flow in software. The model is pre-trained using Caffe framework and the weights and biases are stored in the file lenet5_model.cpp for use in the main application.
There are two Application modes, namely *Sample* and *Test*. The *Sample* mode is used when a MNIST single image has to be identified. The *Test* mode is to test the full MNIST dataset, compare the predicted digit against

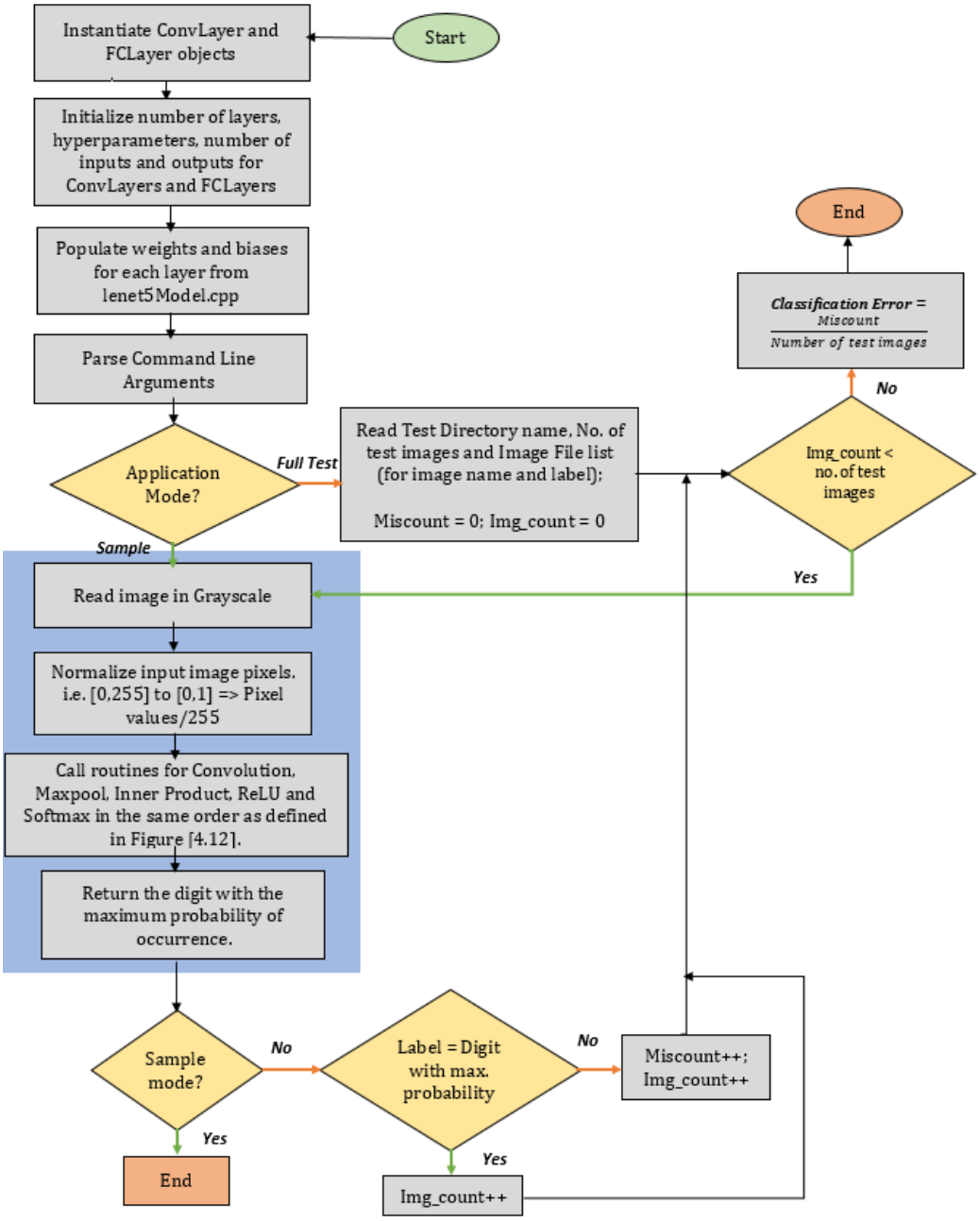


Figure 3.13: Software Code Flow [13]

the pre-defined image label and calculate the prediction accuracy.

Compilation Steps [13]

To compile the code:

```
$ make all
```

To test a sample image:

```
$ ./lenet_app -m sample -i <image_path>
```

Example:

```
$ ./lenet_app -m sample -i ../imgs/mnist_test_img_0.pgm
```

To test the full MNIST Dataset:

```
$ ./lenet_app -m test -f <image_list_file> -d <image_dir
   > [-n <no_images_to_test>]
```

Example:

```
$ ./lenet_app -m test -f ../imgs/mnist_test_img_list.csv
    -d ../imgs/mnist-testset
```

The .csv image list file contains all MNIST handwritten images sized 28x28, along with their labels. These labels help calculate the prediction accuracy and error probability of the digit classifier.

Acceleration Hot-spots

In order to identify the acceleration hot-spots, each operation of the ConvNet was profiled and the results were observed as illustrated in Table 3.2. The total application runtime (execution of all 8 layers) is around 77269 μs = 0.077 seconds. The API ***PAPI_get_virt_usec()*** is used to get the timestamp in microseconds. To use this API, header file "papi.h" has to be included in the source file.

In Table 3.1, the output volume of each Layer of ConvNet is computed based on the concepts discussed in Section 3.1.1.1, given the input dimensions and hyperparameters. Each layer is studied carefully to gather the computational demand of the application.

Table 3.2: Analysis of Application Hot-spots for Acceleration [22]

Layers	Computational Complexity	Execution Time (ms)	
		Intel Core i3-2350M CPU @ 2.30GHz	ZedBoard CPU (2 ARM Cortex A9 Cores)
Convolution 1	300020	12.308	66.577
MaxPool 1	2880	3.382	14.248
Convolution 2	1689000	48.164	359.519
MaxPool 2	800	0.559	4.166
Inner Product 1	400500	5.522	47.924
ReLU	500	0.012	0.073
Inner Product 2	5010	0.073	0.609
Softmax	20	0.009	0.110

For example, in the first convolution layer, filtering operation requires 288K ($24 \times 24 \times 5 \times 5 \times 1 \times 20$) MAC operations. Feature map accumulation along the depth dimension demands another 500 ($20 \times 1 \times 5 \times 5$) accumulation operations. Addition of bias to each filter output involves 11520 ($20 \times 1 \times 24 \times 24$) accumulation operations. Hence, the total computational demand of the first convolution layer is estimated to be around 300K MAC operations.

In the MaxPool layer, the maximum value is identified in each pooling window region spanning to 2×2. Hence, 20 channels of 24×24 inputs demand a total of $20 \times 12 \times 12 = 2880$ findMax operations in the first Maxpool layer. The Inner Product Layer's computational demand is dictated by the input-output dimensions. Hence, the first IP Layer involves 400K MAC operations

(500 × 800) and 500 accumulations for addition of filter bias. Complexity of the ReLU and Softmax layers are dependent on the input size.

The convolution layer involves about 83% of the required arithmetic operations in the ConvNet framework. Following this, the fully connected layers are the next most resource-intensive layers.

Table 3.2 illustrates the computational complexity of all layers in the Lenet-5 ConvNet (Figure 3.12) and their execution time in Intel Core i3-2350M CPU and Zedboard processing system containing 2 ARM Cortex A9 Cores.

Table 3.3: Inferences from Table 3.2 for Convolution Layer 1 running on CPU

	Test Devices	
	Intel Core i3-2350M CPU	ZedBoard ARM
Time taken for 1 MAC	$\frac{12.308ms}{300020}$ = 41 ns	$\frac{66.577ms}{300020}$ = 221.9 ns
MACs/second	$\frac{1}{41ns}$ = 24,390,243.9 ≈ 24M MACs/s	$\frac{1}{221.9ns}$ = 4,506,534.47 ≈ 4M MACs/s
Expected speedup with hypothetical platform (20 processors @ 100 MHz, 1MAC/cycle each)	$\frac{12.308ms}{150\mu s}$ = 82.05	$\frac{66.577ms}{150\mu s}$ = 443.84

We observe from the table 3.2 that for the first convolution layer, the execution time taken by different CPUs to compute 300,020 MAC operations gives an estimate on the number of MACs/second. Using this estimates, we assess the number of MAC operations per second as shown in Table 3.3.

In the software implementation, the input image is loaded from external memory. These load-stores in external memory can create a lot of data movement overheads. Hypothetically, these can be overcome by storing the images on-chip, avoiding unnecessary load-stores in external memory. Consider a hypothetical platform with 20 RISC processors, each capable of performing 1 MAC every cycle and running at 100 MHz.

(i) Time period of 1 cycle = $\frac{1}{100MHz}$ = 10 ns

(ii) Number of MACs computed by 20 RISC processors in one cycle = 20

(iii) Number of cycles required to compute 300020 MACs in Convolution Layer 1 by 20 RISC processors = 15001 Cycles

(iv) Time taken by 20 RISC processors to compute 300020 MACs in Convolution Layer 1 = 15001 $\times$ 10 ns = 150 us

Table 3.4 illustrates inferences for convolution layer 2 from its execution time on different CPUs. Each 12 $\times$ 12 input image involves 1689 MACs (Filtering: (8 $\times$ 8 $\times$ 5 $\times$ 5) + Feature Map Accumulation: (5 $\times$ 5) + Bias Addition: (8 $\times$ 8)).

(i) Number of cycles required to compute 1689000 MACs in Convolution Layer 2 by 20 RISC processors = 84450 Cycles

(ii) Time taken by 20 RISC processors to compute 1689000 MACs in Convolution Layer 2 = 84450 $\times$ 10 ns = 844.5 us

The time taken for classification of one image is 70.029 ms in Intel Core i3 platform and 493.226 ms in Zedboard. Hence, the number of images that can be classified in one second is given by:

1. Intel Core i3-2350M: $\frac{1}{70.029ms} \approx$ 14 frames/second
2. Zedboard ARM Core: $\frac{1}{493.226ms} \approx$ 2 frames/second

In real-time applications, this speed is quite less and hence, we explore possibilities to improve the latency metric.

Table 3.4: Inferences from Table 3.2 for Convolution Layer 2 running on CPU

	Test Devices	
	Intel Core i3-2350M CPU	ZedBoard ARM
Time taken for 1 MAC	$\frac{48.164ms}{1689000}$ = 28.5 ns	$\frac{359.519ms}{1689000}$ = 212.8 ns
MACs/second	$\frac{1}{28.5ns}$ = 35,087,719.3 ≈ 35M MACs/s	$\frac{1}{212.8ns}$ = 4,699,248.12 ≈ 4M MACs/s
Expected speedup with hypothetical platform (20 processors @ 100 MHz, 1MAC/cycle each)	$\frac{48.164ms}{844.5\mu s}$ = 57.03	$\frac{359.519ms}{844.5\mu s}$ = 425.7

3.1.2.3 Improvements

One approach to minimizing data transfer to off-chip memory is by using reduced bit-width fixed point numbers, realizable by using open-source fixed point arithmetic libraries like LibFi [43]. This approach is very straightforward and promises speedup, reduced area and consequently reduced energy consumption. However, the specifics of this approach are beyond the scope of this thesis.

We intend to port the various layers of the ConvNet into fine-grained Graphics Processing Units which exhibit a high degree of data-parallelism. This requires some understanding of the OpenCL device models discussed in Section 2.3.1 Each platform comes with a ready-to-use library which may pose optimization challenges, especially when designing larger applications. Yet another challenge is mapping, owing to differences in on-chip memory, kinds

of parallelism that a particular accelerator can support and communication bandwidth.

3.1.3 Experiments with OpenCL Code

3.1.3.1 Pre-requisites

OpenCL Setup in Ubuntu 14.04

The following are required to run OpenCL Applications on the system:

- Drivers to support OpenCL - Already available in current GPUs
- OpenCL Headers
- Vendor-specific libraries (specific to Intel, NVIDIA, AMD, etc.)
- Installable client driver (.icd)
- libOpenCL.so

1. Installing OpenCL Headers [44]:
Navigate to the path */usr/include* and create a directory named CL.

```
$ sudo apt-get install opencl-headers
```

2. Installing vendor-specific libraries
As Intel CPU is used for our experiments, the following packages are to be installed:

- OpenCL™ Runtime 16.1 for Intel Core™ and Intel Xeon Processors for Ubuntu (64-bit) [28]
- Intel SDK for OpenCL™ Applications [27]

After navigating to the respective installation directories, the command:

```
$ sudo ./install.sh
```

is used to initiate installation.

Dependencies:

mono-devel package (Installation steps summarized in [45]).
Other missing packages are usually prompted during installation and can be installed using the command:

```
$ sudo apt-get install <package_name>
```

Extract the SDK tarball and navigate to the extracted directory:

```
$ tar -xzvf intel_sdk_for_opencl_2016_ubuntu_6.3.0.1904
   _x64.tgz
$ cd intel_sdk_for_opencl_2016_ubuntu_6.3.0.1904_x64
```

The rpm directory contains many default packages for RedHat Linux with **.rpm** extension. They need to be converted to **.deb**(Debian) files to be installed in Ubuntu. To handle .rpm files, **libnuma** package is required:

```
$ sudo apt-get install -y rpm alien libnuma1
```

To **convert rpm format to deb** format and install the Debian packages:

```
$ alien *.rpm
$ dpkg -i *.deb
```

3. Installing the Intel OpenCL ICD Loader

```
$ sudo ln -s /opt/intel/opencl-1.2-5.2.0.10002/etc/
   intel64.icd /etc/OpenCL/vendors/intel64.icd
```

4. Installing a symbolic link to libOpenCL.so

```
$ sudo ln -s /opt/intel/opencl-1.2-5.2.0.10002/lib64/
   libOpenCL.so /usr/lib/libOpenCL.so
$ sudo ldconfig
```

To check if OpenCL applications run properly, clone the GitHub repository from the link [14] and run the Device Query program as follows:

```
$ cd OPENCL_EXAMPLES_ZEDBOARD/devquery
$ gcc devquery.c -lOpenCL
```

The output should be the available devices in the system (CPU, GPU) as shown in Figure 3.14.

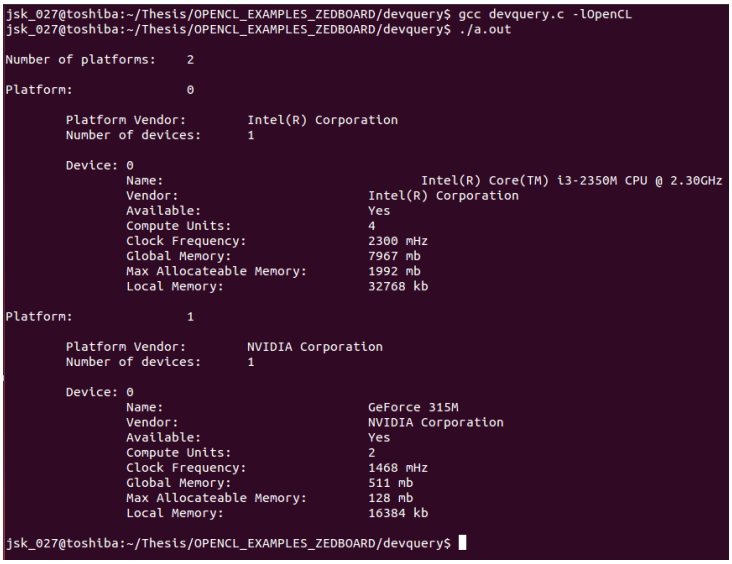


Figure 3.14: OpenCL Device Query code Output [14]

AOCL SDK and Quartus Installation Steps

The FPGA Implementation of MNIST digit recognition [15] uses Altera OpenCL (AOCL) SDK (also called Intel FPGA SDK) and Quartus Software for high-level synthesis and execution. Although our experiments are not based on the Altera Platform, we may use this SDK to use some OpenCL Libraries which are independent of the hardware.

Intel FPGA SDK for OpenCL™ can be downloaded from [46]. The installation steps of AOCL and Quartus from the extracted tarball are detailed in [47]. Following the installation, the environment variable *$ALTERAOCLSDKROOT* is by default set to point to the path where the software was installed. A few more environment variables have to be set to inform the software of the FPGA Board in use and the runtime of the host. If the software was installed in the path, say */home/intelFPGA_pro/17.0/hld/*, then *echo $ALTERAOCLSDKROOT* returns the same path where software was installed.

```
1 $ export PATH=$ALTERAOCLSDKROOT/bin:$PATH
2 $ export AOCL_BOARD_PACKAGE_ROOT=/home/intelFPGA_pro
    /17.0/hld/board/s5_ref
3 $ export QUARTUS_ROOTDIR=/home/intelFPGA_pro/17.0/
    quartus/bin
```

```
$ export LD_LIBRARY_PATH=$ALTERAOCLSDKROOT/host/linux64/
   lib:$AOCL_BOARD_PACKAGE_ROOT/linux64/lib:/usr/local/
   lib:$LD_LIBRARY_PATH
$ source $ALTERAOCLSDKROOT/init_opencl.sh
```

$AOCL_BOARD_PACKAGE_ROOT has to refer to the path of the FPGA Board in use. *s5_ref* is a reference platform available with the SDK files. When using a specific platform, the corresponding platform files are downloaded and the path of the files is used as Board Package Root.

The Altera.icd is copied from *$ALTERAOCLSDKROOT* to */etc/OpenCL/vendors* and the host application is linked to the ICD Loader using the following lines in the Makefile of the host.

```
AOCL_LDFLAGS=$(shell aocl ldflags)
AOCL_LDLIBS=$(shell aocl ldlibs)

host_prog : host_prog.o
g++ -o host_prog host_prog.o $(AOCL_LDFLAGS) -lOpenCL $(
   AOCL_LDLIBS)
```

3.1.3.2 Existing Code Flow Description

The OpenCL implementation of MNIST/Lenet-5 architecture available in the repository [15] is specific to Altera FPGA devices. In order to make this implementation generic and executable on CPU and GPU, the existing code flow has been examined. Figure 3.15 shows the sequence of steps that are done when the a sample image has to be identified.

The first step is the initialization of parameters for all layers in the CNN. This is followed by allocation of buffers necessary for storing inputs and outputs of all layers on the global memory of the device, which is also accessible by the host. The function *findPlatform()* searches for relevant strings such as Intel FPGA SDK for OpenCL, Altera SDK, etc. When a match-word "Altera" is given as argument to this function, it looks for an Altera platform. Should

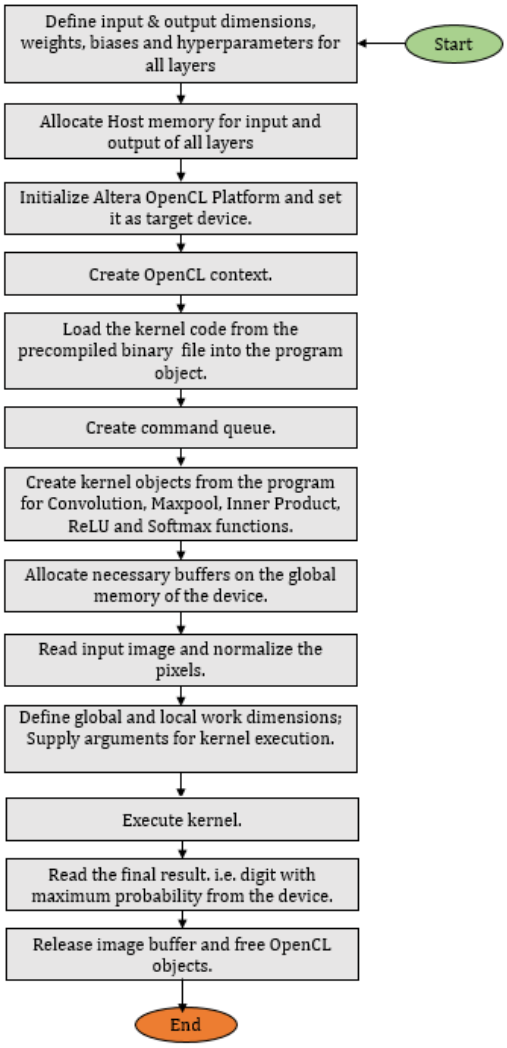


Figure 3.15: OpenCL code flow for Sample (single image recognition) mode [15]

the platform be available, the next step is to query all OpenCL devices in this platform and set one of them as the target device.

The OpenCL Runtime Environment requires a **context** to manage memory, program, command issue, kernels, and program execution on the device

for which the context is defined. Following the context creation, the source code to be ported to GPU is read into a program object.

There are two ways to compile a kernel [16]. **Online compilation** involves reading of the kernel source code by the host and building of the source code at runtime, by the OpenCL Runtime library. For this, the API *clCreateProgramWithSource()* is used, followed by the API *clBuildProgram()*. This method is not recommended for real-time embedded applications. If the kernel is pre-compiled using an OpenCL compiler, the kernel binary is already available and is directly read by the host program, skipping the runtime compilation. This is called **Offline compilation** and requires only one OpenCL function *clCreateProgramWithBinary()*. Although this saves the time to compile the kernel source during runtime, it is platform-specific. If the same kernel code is to be offloaded to other platforms, then a different set of binaries should be generated. Inclusion of multiple kernel binaries increases the size of the executable. The reference code [15] is specific to

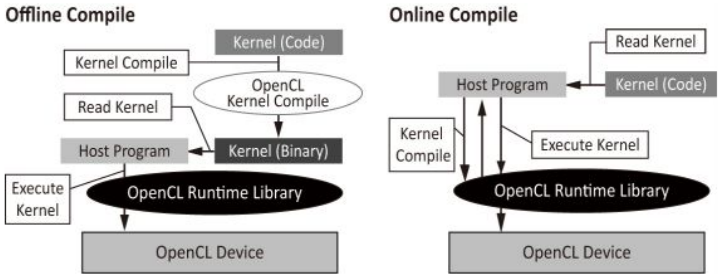


Figure 3.16: Kernel Compilation Modes [16]

Altera devices and uses an offline compilation flow, due to the availability of pre-compiled binary.

Next, a **command queue** is created which instructs which command has to be executed in which device of the group of devices in a particular context. It also dictates whether the execution should occur in-order or out-of-order.

Because the intention is to accelerate the entire ConvNet, kernel objects are created for Convolution, Maxpooling, Inner Product and Activation Layers (ReLU and Softmax). Enough memory has to allocated on the OpenCL device to support the weights, biases, I/Os and execution of kernel calls for all 8 layers of the Lenet-5 Model.

The input image pixels are read and normalized. The kernel code is executed on the device and the final result, i.e. the digit with maximum likelihood is read from the device. Finally, the buffers and memory objects freed.

3.1.3.3 Modifications to remove Platform Dependencies

Allocation of Buffers on the Device Memory:
For Altera FPGAs, the Altera Offline Compiler (AOC) is responsible for generation of logic to support memory accesses [17]. It uses the device SDRAM

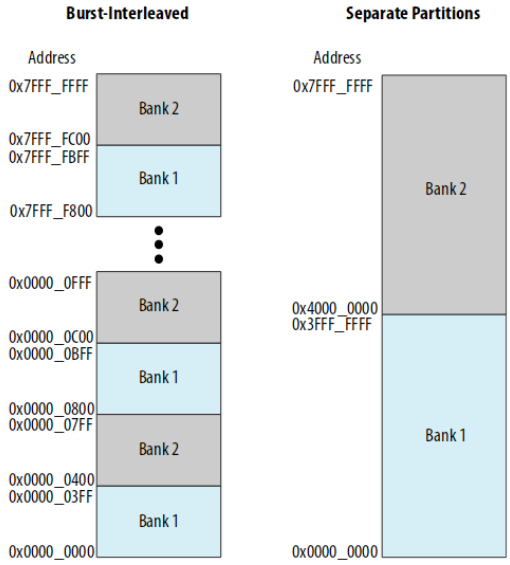


Figure 3.17: Default (bus-interleaved) vs. Manual Global Memory Partitioning [17]

as global memory and by default, stores the data in a burst-interleaved fash-

ion across various external memory banks. Although this offers uniform load distribution and better balance between the banks, manual partitioning of the data may come in handy for certain applications. For example, when the memory banks support different data-types, data cannot be impartially interleaved to these banks.

The MNIST-Altera code [15] accesses global memory using optimized memory banks instead of default burst data allocation in the global memory. For efficient global memory access, the weights and biases are stored in Bank 2 while the data is stored in Bank 1. However, the memory banks in the GPU context refer to partitioning of shared memory into equal blocks which can be accessed simultaneously. Bank conflicts due to certain access patterns can slow down the GPU performance [48]. Hence, the first step to removing platform dependencies is removal of flags CL_MEM_BANK_1_ALTERA and CL_MEM_BANK_2_ALTERA which characterize Altera memory banks (Refer A.1).

Listing 3.1: Header files for Altera FPGA

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <iomanip>
#include <fstream>
#include <unistd.h>
#include <math.h>
#include "CL/opencl.h"
#include "AOCLUtils/aocl_utils.h
#include "cnn_structs.h"
#include "pgm.h"
#include "lenet5_model.h"

using namespace aocl_utils;
using namespace std;
```

Listing 3.2: Header files for GPU

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <iomanip>
#include <fstream>
#include <unistd.h>
#include <math.h>
#include <CL/cl.h>
#include <CL/cl_ext.h>
#include "cnn_structs.h"
#include "pgm.h"
#include "lenet5_model.h"

using namespace std;
```

Usage of Generic OpenCL headers

Although AOCL Utility is a platform independent C++ header file, it has been replaced with standard OpenCL headers for the sake of generality. All

APIs in the scope of AOCL Utility namespace are substituted by general OpenCL APIs described in [49].

Kernel Loading Mechanism

On-the-fly kernel loading, i.e. Online compilation method explained in subsection 3.1.3.2 is employed. The existing and modified code changes are depicted clearly in A.2, A.3 and A.4.

Changes to Makefile

Altera platform specific libraries are unlinked. *libOpenCL.so* Shared Library is linked in the Makefile as shown in Figure 3.18. The resultant code is free from any dependencies with Altera platform and hence, can be compiled and run on any OpenCL device.

```
# Compiler
CXX := g++
# Target
TARGET := host
TARGET_DIR := bin
# Directories
INC_DIRS := host/inc common/inc
LIB_DIRS :=
# Files
INCS := $(wildcard host/inc/*.h)
SRCS := $(wildcard host/src/*.cpp common/src/AOCLUtils/*.cpp)
LIBS := rt\
        OpenCL
# Make it all!
all : $(TARGET_DIR)/$(TARGET)
# Host executable target.
$(TARGET_DIR)/$(TARGET) : Makefile $(SRCS) $(INCS) $(TARGET_DIR)
        $(ECHO)$(CXX) $(CPPFLAGS) $(CXXFLAGS) -fPIC $(foreach D,$(INC_DIRS),-I$D) \
                        $(SRCS)\
                        $(foreach D,$(LIB_DIRS),-L$D) \
                        $(foreach L,$(LIBS),-l$L) \
                        -o $(TARGET_DIR)/$(TARGET)
$(TARGET_DIR) :
        $(ECHO)mkdir $(TARGET_DIR)

run: $(TARGET_DIR)/$(TARGET)
        $(TARGET_DIR)/$(TARGET) -mode=sample \
                -img=mnist_test_img_0.pgm
```

Figure 3.18: Linking libOpenCL Library to Makefile

3.1.3.4 Compiling and executing the code

To test a sample image:

```
$ make run
```

To test the full MNIST Dataset:

```
$ make test
```

The path of the test images (sample and full dataset) is also supplied to the host through the Makefile. Hence, the path has to be suitably modified to point to the test images in the local machine.

The kernels can be executed either in CPU or GPU devices which support OpenCL.

Listing 3.3: CPU or GPU Device Selection

```
int gpu = 1;
for(unsigned i = 0;i < dev_cnt; i++){
    err = clGetDeviceIDs(platform_ids[i], gpu ?
        CL_DEVICE_TYPE_GPU : CL_DEVICE_TYPE_CPU, 1, &
        target_device, NULL);
    if(err == CL_SUCCESS){
        break;
    }
}
```

When the integer variable *'gpu'* is set to 1, the GPU device is selected and when it is set to 0, the CPU device is selected.

Benchmarking Kernel Execution Time

- Profiling should be enabled during the creation of command queue as follows:

```
queue = clCreateCommandQueue(context, target_device,
    CL_QUEUE_PROFILING_ENABLE, &status);
checkError(status, "Failed to create command queue");
```

- An event is associated with the kernel during its launch as follows:

```
status = clEnqueueNDRangeKernel(queue, kernel[0], 3, NULL, global_work_size,
    NULL, 0, NULL, &kernel_event[0]);
checkError(status, "Failed to launch conv1 kernel");
```

- Kernel execution has to be completed and also all enqueued tasks in the command queue should finish.

```
clWaitForEvent(1, &kernel_event[0]);
clFinish(queue);
```

- The following APIs can be used to estimate the kernel execution time:

```
cl_ulong start_time, end_time;
double total_time;
clGetEventProfilingInfo(kernel_event[0], CL_PROFILING_COMMAND_START,
    sizeof(start_time), &start_time, NULL);
clGetEventProfilingInfo(kernel_event[0], CL_PROFILING_COMMAND_END,
    sizeof(end_time), &end_time, NULL);
total_time = end_time−start_time;
printf("Kernel Execution Time is: %0.3f \n",total_time/1000000.0);
```

3.1.4 Comparative Study of Results

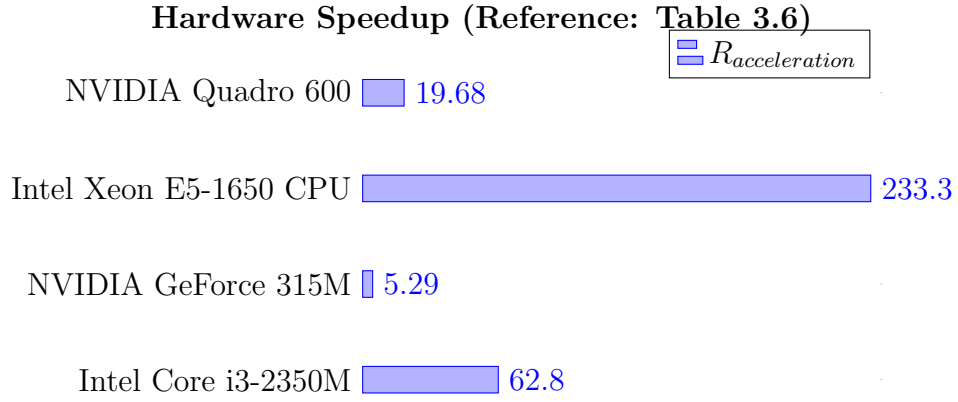
A simple device query program is run on each of the test devices to query attributes such as memory, number of compute units, etc. The test device specifications are tabulated for comparison in Table 3.5.

Goal: Achieve a faster runtime in hardware compared to software. The ratio $R_{acceleration} = \frac{t_{sw}}{t_{hw}} > 1$

Table 3.5: Test Device Specifications

Attributes	Test Devices			
	Intel Core i3-2350M CPU	NVIDIA GeForce 315M	Intel Xeon CPU E5-1650	NVIDIA Quadro 600
OpenCL Version	1.2	1.1	2.1	1.1
Compute Units	4	2	12	2
Clock Frequency (MHz)	2300	1468	3200	1280
Global Memory (MB)	7967	511	15971	1023
Maximum Allocateable Memory (MB)	1992	128	3993	256
Local Memory (KB)	32768	16384	32768	49152

Hardware Speedup (Reference: Table 3.6)



The total time required to classify one input image is 1.21ms in Intel Core i3-2350M, 14.526ms in NVIDIA GeForce 315M, 3.905ms in NVIDIA Quadro

Table 3.6: Comparison of kernel runtime in various OpenCL Devices

	Kernel Execution Time (ms)			
	Intel Core i3-2350M CPU	NVIDIA GeForce 315M	Intel Xeon CPU E5-1650	NVIDIA Quadro 600
Convolution 1	0.216	0.707	0.088	0.143
MaxPool 1	0.046	0.166	0.022	0.029
Convolution 2	0.716	11.332	0.146	1.846
Maxpool 2	0.026	0.371	0.014	0.026
Inner Product 1	0.187	1.651	0.039	1.747
ReLU	0.011	0.012	0.007	0.005
Inner Product 2	0.010	0.287	0.006	0.109
Softmax	0.014	0.012	0.008	0.007

600 and 322 us in Intel Xeon platform. This implies that in one second, the number of images that can be processed by each of these heterogeneous platforms is given by:

1. Intel Core i3-2350M: $\frac{1}{1.21ms} \approx 826$ frames/second
2. NVIDIA GeForce 315M: $\frac{1}{14.526ms} \approx 68$ frames/second
3. Intel Xeon CPU E5-1650: $\frac{1}{0.322ms} \approx 3105$ frames/second
4. NVIDIA Quadro 600: $\frac{1}{3.905ms} \approx 256$ frames/second

In summary, it can be observed from Table 3.3 that when running single-threaded C++ code for convolution layer 1, the hypothetical platform is approximately 82 times faster than Intel core-i3 and 440 times faster than Zedboard ARM. Table 3.4 estimates reveal that the time required for the hypothetical platform to get all outputs of Convolution Layer 2 is 57 times

Table 3.7: Inferences from execution times of OpenCL devices running offloaded Convolution Layer 1 code (Calculations similar to Tables 3.3 and 3.4)

	Test Devices			
	Intel Core i3-2350M	NVIDIA GeForce 315M	Intel Xeon CPU E5-1650	NVIDIA Quadro 600
Time taken for 1 MAC	0.7 ns	2.35 ns	0.29 ns	0.47 ns
MACs/second (approx.)	1 B	400 M	3 B	2 B
Gained Speedup compared to Intel Core i-3 running single-threaded C++ code	56.98	17.4	139.86	86.069
Gained Speedup compared to Zedboard ARM running single-threaded C++ code	308.2	94	756.5	465.57
Expected speedup with hypothetical platform (20 processors @ 100 MHz, 1 MAC/cycle each)	1.44	4.7	0.58	0.95

faster than Intel core-i3 platform and 426 times faster than Zedboard ARM for convolution layer 2.

A close observation of Tables 3.5 and 3.6 shows that with more memory to work on and more compute units, more threads can be concurrently executed and hence, more speedup can be achieved.

When the same CNN code is offloaded to an OpenCL device capable of executing multiple threads in parallel, the gain in latency is enormous, almost comparable to our hypothetical platform. For example, the same convolution

algorithm running on the same platform, Intel Core i-3 CPU offers different latencies depending on whether it is a sequential execution or multi-threaded concurrent execution on several compute units. The OpenCL code is approximately 57 times faster than its single-threaded C++ counterpart (Reference: Table 3.7). Executing work items on several low-power compute units not only improves latency but also allows CPU to perform other critical tasks. These heterogeneous architectures allow complex applications to be made more practical and hence drive today's technology roadmap.

Chapter 4

Hardware Acceleration of Fully Homomorphic Encryption

4.1 Fully Homomorphic Encryption Scheme

Fully Homomorphic Encryption scheme allows computation of arithmetic or logical functions on encrypted data, without decrypting them. It was first conceptualized and realized by Craig Gentry in 2009 [19]. Several improvements have been made since then to improve the security of the initial scheme [50][51][52] and to make the number of homomorphic operations asymptotically large, by reducing the noise in ciphertexts. *Bootstrapping* is a novel method introduced by Gentry to reduce noise in ciphertexts to acceptable levels, by homomorphically evaluating the decryption function using the encrypted secret key. However, Bootstrapping is a costly operation [21] and takes around 0.69 seconds in software. This brings in some motivation for hardware acceleration, to achieve a practical performance.

To offload the FHE operations to hardware, it is important to have some mathematical awareness and understanding of the various steps involved in FHE. The paper [21] introduces the library, FHEW [53] which performs a

simple Bootstrapped NAND operation exhibiting lower noise levels compared to previous FHE schemes discussed in [52][51]. This method is not restricted to NAND operation but can be extended to various other arithmetic and logical computations [21].
The Figure 4.1 illustrates the problem statement that we seek to address through this scheme. One practical application of this scheme is to delegate data-processing to the cloud without giving away the original data.

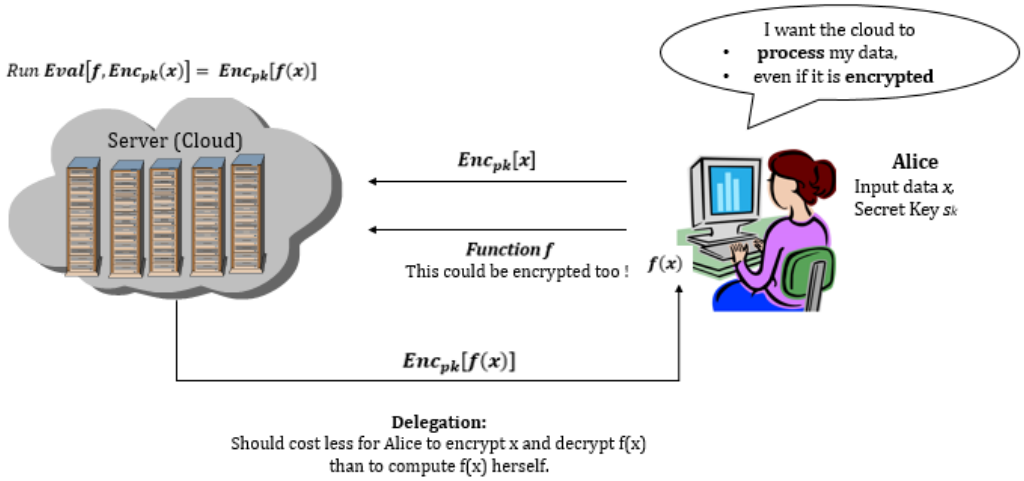


Figure 4.1: FHE: Problem Statement [18]

4.1.1 Background

Homomorphism

Given two groups, G and H, Homomorphism from G to H can be defined as a function *f:* $G \rightarrow H$, such that:

$f(g_1 * g_2) = f(g_1) * f(g_2)$,

g_1, g_2 : Elements in G,

$*$: Operation in G,

$*$: Operation in H

Steps in FHE

Fully homomorphic encryption scheme ϵ has 4 key steps:

1. $KeyGen_{\epsilon}(\lambda)$

It involves generation of random secret key, which is an odd integer p, P-bits long. The security parameter λ dictates the bit-length of the key.

- Symmetric Encryption:
  Encryption and Decryption are performed using the same secret key.
- Asymmetric Encryption:
  Encryption is done using a public key (p_k) and Decryption using a secret key (s_k).

2. $Encrypt_{\epsilon}(p,m)$

Given the security parameter λ,

N = λ; P = λ^2; Q = λ^5

Scheme [19]:

To encrypt a bit $m \in \{0,1\}$, set $m' = m \ mod \ 2$, a random N-bit number.

Output ciphertext: $c \leftarrow m' + pq$,

where q is a random Q-bit integer.

i.e. $c \leftarrow m \ mod \ 2 + pq$

3. $Decrypt_{\epsilon}(p,c)$

Output: $c' \ mod \ 2$,

where $c' = c \ mod \ p$ is an integer in the range $(-p/2, \ p/2)$ and p divides c-c'. $c - c' = c - c \ mod \ p = c \ (1 - mod \ p)$ is divisible by p. Hence, the ciphertexts of ϵ are near-multiples of p.

To maintain a constant complexity for decryption, any two ciphertexts c_1 and c_2 outputted from the encryption scheme should be of the same size [19]. Size of the ciphertext and the time taken to decrypt should be independent of the complexity of function f, delegated to the cloud.

4. $Evaluate_{\epsilon}(p_k, \ f, \ c_1, \ c_2, \ \ldots \ c_t)$

For any function f in a set of permissible functions F_ϵ, and ciphertexts c_1, $c_2 \ldots\ c_t$, where $c_i \leftarrow Encrypt_\epsilon(p_k,\ m_i)$, the following 2 steps are performed:

$$c \leftarrow Encrypt_\epsilon[\ f\ (c_1,\ c_2,\ \ldots\ c_t)]$$

$$Decrypt_\epsilon(\mathrm{c}\ ,\ s_k) = f\ (m_1,\ m_2,\ \ldots\ m_t)$$

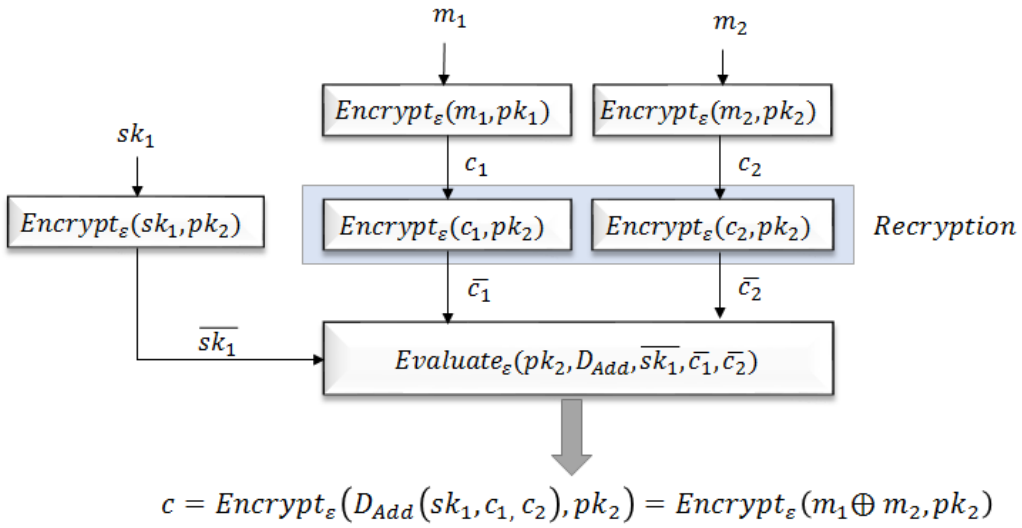


Figure 4.2: Homomorphic Encryption Scheme : Example [19]

This scheme guarantees one-wayness and semantic security against chosen plain-text attacks, as it is probabilistic [19]. Figure 4.2 shows 2 messages m_1 and m_2 encrypted using public key pk_1 whose associated secret key is sk_1. $Evaluate_\epsilon$ takes in the resultant ciphertexts c_1, c_2 and secret key sk_1 encrypted under another key pk_2 and outputs c which is an encryption of D_ϵ result under pk_2.

Concept of Bootstrapping

Decryption reduces the noise. However, decrypting the data in remote server can compromise security. So, homomorphic decryption described in Figure 4.2 is used to reduce the noise level, so that the result is decipherable at the receiver.

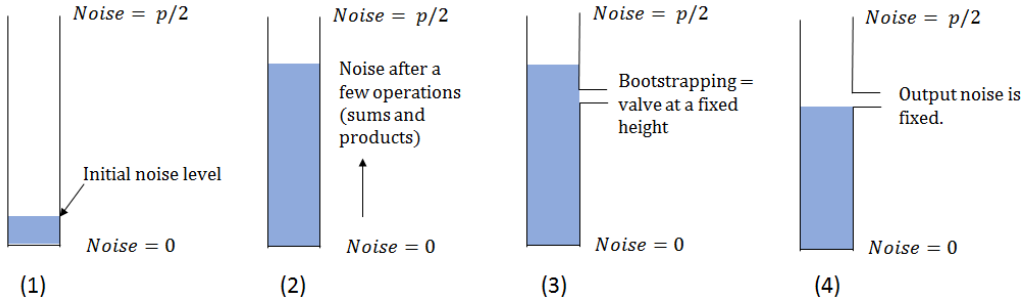


Figure 4.3: Homomorphic Decryption: Example [20]

4.1.2 Existing code flow

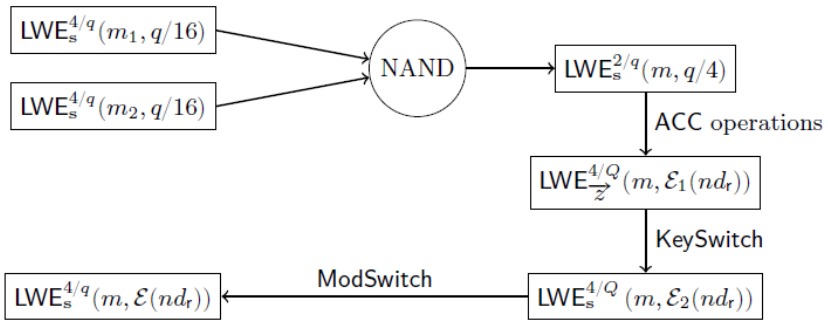


Figure 4.4: Cycle of simple NAND operation [21]

The FHEW Library [53] uses the ring lattice for encryption and bootstrapping, to reduce the computation time to quasi-linear complexity (Using FFT) as opposed to the quadratic complexity of previous methods [21].

The *Learning With Errors* (LWE) encryption scheme has been used to generate ciphertexts, with a message modulus of 4 and error bound of q/16. The specifics of this scheme have been described in great detail in [21] for further reading, and shall not be covered in this thesis. A random evaluation key, constituting a bootstrapping key and switching key is generated for a given LWE secret key. Both encryption and recryption (Figure 4.2) use the

LWE scheme but with different keys. Two ciphertexts LWE-encrypted using a n-dimensional secret key (n = 500) are fed to the Homomorphic NAND block:

HomNAND: $LWE_s^{\frac{4}{q}}(m_0$, q/16) x $LWE_s^{\frac{4}{q}}(m_1$, q/16)$\rightarrow LWE_s^{\frac{2}{q}}(m_0 \bar{\wedge} m_1$, q/4)

where LHS denotes ciphertext inputs encrypting binary messages, $m_0, m_1 \in \{0,1\}$ with error limit as q/16. RHS is the HomNAND output which is an encryption of the logical NAND of the two input messages = 1 - $m_0.m_1 = m_0 \bar{\wedge} m_1$, with an error bound of q/4. The switching key facilitates conversion of LWE encryption with one secret key to another LWE encryption using a different secret key. Modulus switching helps switch the LWE ciphertexts from one modulus (Q) to another (q).
The most efficient Bootstrapping method proposed by the author is the one using FFTs (Section 5.3, [21]). The Fastest Fourier Transform in the West (FFTW) Library [54] is an open-source benchmark library for optimized software FFT computations. This library has been used for performing FFTs and inverse FFTs in the FHE algorithm, with double-precision floating point accuracy.

4.1.3 Hot-Spots for Hardware Acceleration

One major difference in FPGA programming, be it RTL or HLS, is the bit-width precision. Register sizes are deterministic at compile time. Libraries such as $< ap\_cint.h >$ and $< ap\_int.h >$ in Vivado HLS facilitate specification of bit-accurate variables. We notice that FFTW Library required by FHEW library is compiled for double-precision floating point accuracy which is 64-bits long by default. Usually, Embedded devices are memory and power constrained. Hence, exploring the accuracy of results with varying bit-widths is another interesting aspect to investigate, for a holistic analysis of hardware acceleration using FPGAs.

Software Profiling results have confirmed that the Homomorphic NAND operation takes up the maximum execution time. Each HomNAND operation makes several calls to Accumulator as illustrated in Table 4.1. The accumulator in turn calls the 2048-point FFT and IFFT routines several times. The

Table 4.1: Analysis of Software Bottlenecks

HomNAND Test Count	Function	Number of function calls
0	FFT	396002
	Inverse FFT	132000
	Homomorphic NAND	0
	Add to Accumulator	0
1	FFT	499430
	Inverse FFT	166470
	Homomorphic NAND	3
	Add to Accumulator	2872

tabulated values are averages obtained from 5 runs of the application in each of the two cases, HomNAND Test for 0 and 1 rounds. The number of function calls is not deterministic but usually around a certain range, due to the random nature of input, secret and bootstrapping keys. Each HomNAND Test involves 3 HomNAND function calls in the implemented FHE design [53] 4.1. This is because the circuit under test is defined by: (**a** NAND **b**) NAND (**c** NAND **d**).

4.1.3.1 Dimensionality Analysis

Table 4.2 shows the dimensions of the inputs, intermediate values, secret keys and output in the encryption scheme. The bootstrapping and switching keys used in AddToAccumulator block of HomNAND operation are of huge sizes and hence, replicating multiple AddToAccumulator blocks in the

Table 4.2: Dimensionality Analysis (Section 6.2, [21])

Parameter	**Size**
LWE Secret Key	Array of size 500
Evaluation Key	BootstrappingKey[500][23][2] $\approx$ 1032 MBytes
	SwitchingKey[1024][25][7] $\approx$ 314 MBytes
HomNAND Inputs	1-bit
HomNAND Output Cipher (a,b);	a[500] - coefficient vector over a cyclotomic ring R in Z[X]/X^N+1;
	b = **a.s** + e, where e is the error.
Decrypt Output	32 bits

hardware could be costly, without prior optimizations. Table 4.1 shows that each Homomorphic NAND test involves around $\frac{499430-396002}{3} = 34476$ FFT operations and the author confirms in Section 6.2, [21] that as high as 48000 FFTs are performed per NAND gate. The Table 4.3 illustrates the average

Table 4.3: Software Computation Time

Operation	**Computation Time (μs)**
FFTW FFT	30.892
FFTW IFFT	23.93

execution time of a single FFT and IFFT, taken across 500 readings in a quad-core CPU. Since the speed of the algorithm depends on the speed of FFTs, porting this portion to hardware helps achieve a faster execution. We observe that FFTW is highly efficient in terms of runtime in software. However, it is a huge library and not hardware-friendly.

4.1.4 Results

Precision Analysis

To determine whether double-precision floating point FFT is required to preserve the functionality of FHEW, the source code of FHEW was linked to different precision libraries and output analyzed. The configure script that comes with the library source files is used to generate Makefile. By using different compiler flags [55], the FFTW library can be compiled for:

- single precision floating point

```
$ sudo ./configure --enable-float
```

- quadruple-precision floating point (__float128)

```
$ sudo apt-get install libquadmath
$ sudo ./configure --enable-quad-precision
```

- long double

```
$ sudo ./configure --enable-long-double
```

When no option is specified, the default precision is *double*. The below commands install the library with the new precision settings.

```
$ make
$ make install
```

The source code of FHEW is linked to the new libraries: **-lfftw3f** (for float) or **-lfftw3l** (for long double) or **-lfftw3q -lquadmath -lm** (for quad-float) instead of the default library **-lfftw3**.

All lower-case instances of "fftw_" in the FFT function calls and datatypes are replaced by fftwf_ (for float) or fftl_ (for long double) or fftwq_ (for quad-float).
Example: The datatype **fftw_complex** becomes **fftwf_complex** or **fftwl_complex**

or **fftwq_complex** depending on the desired precision.

As our intention is reduction of bit-width, the experiments were based on single-precision and quad-precision float. Upon making the above-mentioned changes, the FHEW functionality was not preserved due to reduction in accuracy. Supporting the analysis, the authors of [21] have stated that double-precision float is barely sufficient to contain the error levels within an acceptable range (Section 6.3, [21]). Figure 4.5 illustrates the loss of functionality upon linking the source code to a lower precision FFT. From the above ex-

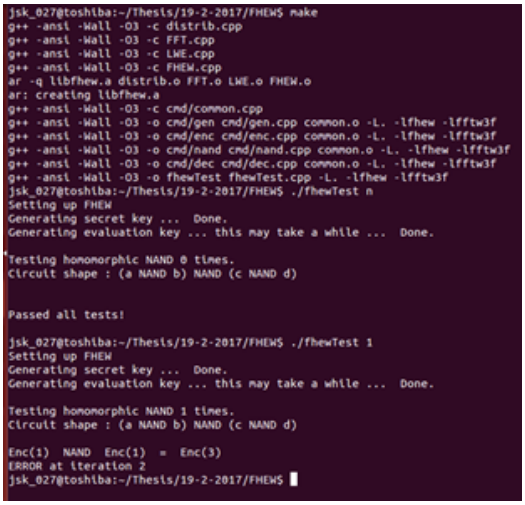


Figure 4.5: Precision Loss with single-precision float

periments, we conclude that of the standard datatypes, **double** is the lowest allowable precision that preserves the homomorphic encryption functionality.

FFTs can also be performed in integer domain using Number Theoretic Transform library (NTL) used in [56]. NTL involves convolution of 2 sequences modulo a prime number. Hence, the choice of modulus with respect to sequence length is restricted [57]. FFTW has proved to be much more optimized and faster than the latter as stated in Section 6.3, [21], which is

why FHEW is an improvement over HELib [18] which uses NTL, in terms of Bootstrapping runtime. As the optimization objective is to improve runtime, FFT function can be hand-optimized for hardware acceleration and the functional correctness verified, by integrating with the existing software implementation.

FFT Offload

Various FFT implementations were explored, intuitively optimized and their runtime performance was studied using High-level synthesis tools. Firstly, the Xilinx FFT IP Core [58] was analyzed and its usage studied. It implements the Cooley Tukey (Radix-4 DIT) FFT and accepts inputs in full-precision fixed point, scaled fixed point and block floating point. Since double-precision inputs are not supported by this hard block, it is not an ideal candidate for our application's acceleration. Following this, two FFT codes were written, one performing simple 1D FFT and another, a simplified Radix-2 Butterfly (Cooley Tukey) DIF FFT implementation of size N, given by the product $N_1.N_2$ where N_1 naive DFTs, each of size N_2 were performed. Thus, these implementations were analyzed and intuitively improved by doing loop and function optimizations, integrated to the FHEW code and the functional correctness was verified.

For these implementations to be synthesizable by the high-level synthesis tool, recursive calls and dynamic memory allocation were avoided and all pointers were made deterministic at compile time. The pros and cons of all three implementations were studied to decide on which best suits our requirement and the code giving the best runtime performance was identified.

4.1.4.1 Hand-optimized non-recursive 1-D FFT

Table 4.4 illustrates the clock period, latency and utilization estimates upon applying different optimization directives to the hand-optimized FFT implementation for hardware. We observe that solution 4 offers the most optimal

area-latency design point. However, in a broader perspective, we get the

Table 4.4: Design Space Exploration for non-recursive 1-D FFT

Target: Avnet Zedboard Evaluation xc7z020clg484-1						
		Solution 1	**Solution 2**	**Solution 3**	**Solution 4**	**Solution 5**
Optimization		*No directives.*	Loop 0: *Pipeline*	Loop 0: *Unroll*	Loop 0 & 2: *Pipeline*	Loop 0: *Pipeline*, Loop 2: *Unroll*
Estimated Clock Period (ns)		9.83	9.83	12.93	11.96	12.19
Latency (cycles)	**Min**	295730178	295705626	297810991	5387290	8048665
	Max	945847298	945822746	947928111	5387290	12492825
Utilization Estimates	**BRAM**	13	13	13	8	13234
	DSP48E	67	73	234	105	31410
	FF	7235	8245	149093	32308	4632256
	LUT	24133	26896	117748	70551	19331620

estimated execution time as 5387290 x 11.96 ns = 64 ms, by calculating the product of estimated clock period and minimum latency. This is an impractical choice, spanning several clock cycles and we want our whole FHEW library to execute in the order of a few milliseconds, .

4.1.4.2 MachSuite FFT

MachSuite is a collection of 19 diverse application benchmarks, which are high-level synthesizable. FFT algorithm which is one of the benchmark applications is explored to decide whether it meets our requirement. We observe that this benchmark implementation is already well-optimized and application of compiler directives offers no gain in latency. Following the co-simulation and RTL export steps for this benchmark code in HLS, Run-time Simulation of resultant IP cores is performed in Vivado Design Suite

Table 4.5: Pre-implementation Design space exploration for Machsuite FFT

Target: Avnet Zedboard Evaluation xc7z020clg484-1			
		Solution 1	**Solution 2**
Optimization		Loop 0: *Trip Count = 10,* Loop 1: *Trip Count = 512*	Loop 0: *Trip Count = 10,* Loop 1: *Trip Count = 512 BRAM to store inputs and twiddle factors*
Estimated Clock Period		8.23	8.23
Latency (cycles)	**Min**	21	21
	Max	117781	122901
Utilization Estimates	**BRAM_18K**	0	0
	DSP48E	56	56
	FF	4285	4286
	LUT	7578	7558

Table 4.6: Post-implementation execution time for MachSuite benchmark FFT

	Solution 1	**Solution 2**
Achievable clock period (ns)	9.87	9.612
Latency (cycles)	104482	109602
Execution Time	1.031 ms	1.053 ms

and the respective execution times are tabulated in Table 4.6. We see that for this algorithm, the best-case latency is obtained without any directive. This is consistent with the notion that the choice of directives is crucial and dependent on the implementation, and mindless application of optimization directives doesn't always guarantee improved performance.

4.1.4.3 Cooley Tuckey Radix-2 DIF FFT

DIF FFT takes inputs in bit-reversed order and produces outputs in natural order. Shift operators are used instead of the costly multipliers. This implementation is in-place to reduce migration costs and less intuitive, but better optimized for hardware.

Table 4.7: Pre-implementation Design Space Exploration for Radix 2 DIF-FFT

Target: Avnet Zedboard Evaluation xc7z020clg484-1					
		Solution 1	**Solution 2**	**Solution 3**	**Solution 4**
Optimization		Loop 1: *Trip count = 512,* Loop 2: *Trip count = 2*	Loop 1: *Trip count = 512,* Loop 2: *Trip count = 2,* Loop 3: *Pipeline*	Loop 1: *Trip count = 512,* Loop 2: *Trip count = 2,* Loop 3: *Unroll (factor 16)*	Loop 1: *Trip count = 512,* Loop 2: *Trip count = 2,* Loop 3: *Unroll*
Estimated Clock Period (ns)		8.23	8.23	8.23	8.23
Latency (cycles)	**Min**	2170	3195	1210	1112
	Max	259194	259195	259194	257112
Utilization Estimates	**BRAM**	0	0	0	0
	DSP48E	56	56	56	56
	FF	4513	4515	7004	131824
	LUT	8532	8532	9911	67505

Appendix B.2 shows that some loop bounds are not statically defined. So, the HLS tool cannot determine the number of iterations of a variable-length loop at runtime and hence, cannot supply the latency estimates. To circumvent this problem, the directive "Loop Trip Count" is applied by manually specifying the maximum number of times this loop will be executed, at

compile time.

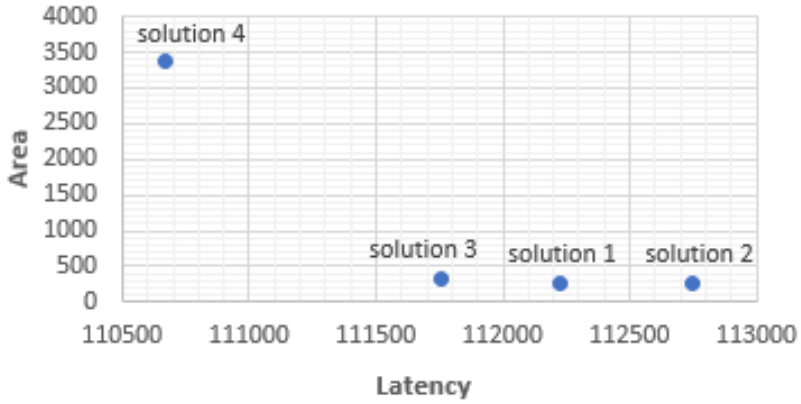
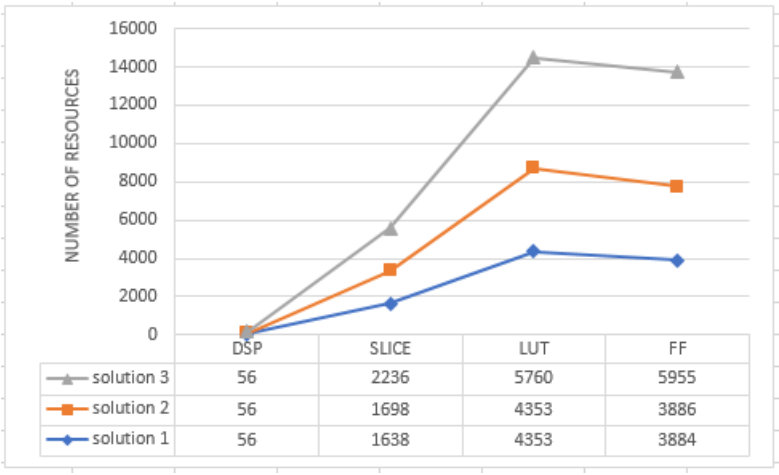


Figure 4.6: Area vs. Latency plot for Radix-2 DIF FFT

From the plot 4.6, we see that solution 4 gives the best case latency. However, this design requires 124% of LUTs and 132% of FFs available in the board. As there are not enough LUTs to do a complete unroll of the loop 3, we choose an unroll factor of 16, which is more practical to realize and also offers a good speed (solution 3). From Figure 4.7, we also observe



	DSP	SLICE	LUT	FF
solution 3	56	2236	5760	5955
solution 2	56	1698	4353	3886
solution 1	56	1638	4353	3884

Figure 4.7: Post-implementation resource utilization summary for Radix-2 DIF FFT

that solution 3 also has a higher resource utilization than solutions 1 and 2. Hence, it can be concluded that solution 3 offers the practical best-case execution time compared to other solutions in Table 4.7.

Table 4.8: Post-implementation execution time for Radix-2 DIF FFT

	Solution 1	**Solution 2**	**Solution 3**	**Solution 4**
Achievable Clock Period (ns)	9.597	9.603	9.445	-
Latency (cycles)	112222	112751	111757	110668
Execution Time	1.076 ms	1.08 ms	1.055 ms	Place design error

4.1.4.4 Comparison of Execution Times

The FFTW library calculates FFT in software in about 30 μs, on an average. However, the execution time could go up based on the nature of inputs to the library. As a first step, FFTW is replaced by a handwritten FFT logic successfully and its functional correctness verified. After replacing, it is observed that a single FFT takes around 0.13 seconds in software, which is very high. An algorithmic exploration is performed for a given Zynq Architecture. The MachSuite benchmark takes about 1.03ms whereas butterfly DIF FFT (solution 3) takes about 1.055ms. Although this is still not comparable to the benchmark performance of FFTW library, a much superior performance is expected once other blocks of AddToAcc are understood and completely ported, allowing for the concurrent execution of several hardware blocks. Using high-speed streaming DMA Transfers from high performance ports of Zynq Processing system to Programmable logic via AXI Interface can help reduce the communication overhead of the system.

The ratio $R_{acceleration} = \frac{t_{sw}}{t_{hw}}$

$= \frac{0.13s}{1.03ms} = 126.2$ times for MachSuite benchmark
$= \frac{0.13s}{1.055ms} = 123.2$ times for Radix-2 butterfly FFT

Alternatively, an architectural exploration can also be performed and higher hardware generations having enough resources can be identified, to allow bigger unroll factors and a much higher throughput. For example, Virtex 7 board is usually faster that the Zynq Zedboard and has more resources giving the freedom to use more suitable compiler optimizations.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This report discussed hardware acceleration using heterogeneous architectures, specifically GPUs and FPGAs by means of threaded programming (OpenCL) and high level synthesis. As the optimization objective was improving execution speed, several compiler and platform-specific optimizations were used to inspect the gain in runtime. This report covered the study of two trending compute-intensive applications, isolation of software hot-spots, design decisions and improved implementation using heterogeneous hardware platforms.

Our experiments with MNIST digit classifier revealed that when the sequential C++ code was translated to parallel threads spawned together for concurrent execution, the speed-up was massive. For example, the CNN code running on Intel Core i3-2350M platform took around 70.029 ms to classify a single image when executing sequentially, while it took only 1.2 ms with multi-threaded OpenCL code. The number of multiply-accumulate operations performed per second increased from 24M MACs/s to 1B MACs/s for convolution layer 1 on the same platform. This layer executed almost 57

times faster with the OpenCL code 3.23.3. Also, a cross-platform speedup of as high as 139 times 3.3 was achieved with the Intel Xeon ES-1650 CPU. This speedup was almost comparable to the gain that can be accomplished using a hypothetical platform with 20 RISC processors running at 100 MHz each and processing 1 MAC/cycle each. A huge dataset comprising of 10000 images was classified in a few microseconds. It also revealed that any OpenCL compliant device, CPU or GPU, can offer acceleration depending on the number of low power cores available in the platform and also the memory access model. Hence, understanding of all the OpenCL models is crucial to schedule the work suitably among different compute units. The study of Fully Homomorphic Encryption scheme revealed that domain expertise is also a key factor to achieve hardware acceleration, in addition to accelerator-awareness. All computations performed in FHE are on ring lattices and hence, offloading the entire Bootstrapping logic onto hardware requires sound mathematical background on lattice-based computations. We noticed a speedup of 126 times with MachSuite benchmark running on hardware compared to the handwritten FFT executing in software. The key challenge in hardware acceleration is ensuring that functionality is preserved upon offloading to an accelerator.

5.2 Future Work

Future works could incorporate:

- **Runtime analysis on coarse-grained and fine-grained Overlay Architectures** with efficient interfacing between host processor, DSP Units and other high-speed vector engines.

- Identifying a novel way to **fit in a single AddToAccumulator block onto a single FPGA**:
  The challenge in doing so is the huge dimensionality involved (Section

4.1.3.1). Computations on ciphertexts take up alot of area and hence, these dimensions have to be intuitively handled.

- **Reuse of the hardware FFT blocks in the new TFHE Library** implemented in April, 2017, to verify the generality of the implementation. As TFHE library already promises bootstrapping speed of less than 0.1 seconds, the gain in hardware for bootstrapping can be studied by porting specific "hot" functions to the hardware.

Appendix A

CNN

Table A.1: Removal of Altera device-specific Macros

```
void createDeviceBuffer() {
       cl_int status;
       cout << "Allocating buffers on the device memory" << endl;
       // data is allocated in BANK1 and weights are in BANK2 for efficient access.
       d_input_img = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_BANK_1_ALTERA,
              conv1.bot_shape->x * conv1.bot_shape->y * conv1.bot_shape->z * sizeof(DTYPE), NULL, &status);
       conv1.d_input = clCreateBuffer(context, CL_MEM_WRITE_ONLY | CL_MEM_BANK_1_ALTERA,
              (conv1.bot_shape->x+2*conv1.pad) * (conv1.bot_shape->y+2*conv1.pad) * conv1.bot_shape->z * sizeof(DTYPE),
                    NULL, &status);
       conv1.d_output = clCreateBuffer(context, CL_MEM_WRITE_ONLY | CL_MEM_BANK_1_ALTERA,
              conv1.top_shape.x * conv1.top_shape.y * conv1.top_shape.z * sizeof(DTYPE), NULL, &status);

       conv1.d_W = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_BANK_2_ALTERA | CL_MEM_COPY_HOST_PTR,
              conv1.K * conv1.K * conv1.bot_shape->z * conv1.top_shape.z * sizeof(WTYPE), conv1.W, &status);
       conv1.d_b = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_BANK_2_ALTERA | CL_MEM_COPY_HOST_PTR,
              conv1.top_shape.z * sizeof(WTYPE), conv1.b, &status);
       ........
}
```

Table A.2: Loading kernel from Source

```
long LoadOpenCLKernel(char const* path, char **buf)
{
    FILE *fp;
    size_t fsz;
    long off_end;
    int rc;
    /* Open the file */
    fp = fopen(path, "r");
    if( NULL == fp ) {
        return -1L;
    }
    /* Seek to the end of the file */
    rc = fseek(fp, 0L, SEEK_END);
    if( 0 != rc ) {
        return -1L;
    }
    /* Byte offset to the end of the file (size) */
    if( 0 > (off_end = ftell(fp)) ) {
        return -1L;
    }
    fsz = (size_t)off_end;
    /* Allocate a buffer to hold the whole file */
    *buf = (char *) malloc( fsz+1);
    if( NULL == *buf ) {
        return -1L;
    }
    /* Rewind file pointer to start of file */
    rewind(fp);
    /* Slurp file into buffer */
    if( fsz != fread(*buf, 1, fsz, fp) ) {
        free(*buf);
        return -1L;
    }
    /* Close the file */
    if( EOF == fclose(fp) ) {
        free(*buf);
        return -1L;
    }
    /* Make sure the buffer is NUL-terminated, just in case */
    (*buf)[fsz] = '\0';
    /* Return the file size */
    return (long)fsz;
}
```

Table A.3: Initialization of OpenCL Objects for Altera FPGA (Taken from Altera Design Examples)

```
bool init_opencl() {
       cl_int status;
       printf("Initializing OpenCL\n");
       if(!setCwdToExeDir()) {
         return false;
       }
       // Get the OpenCL platform.

       platform = findPlatform("Altera");

       if(platform == NULL) {
        printf("ERROR: Unable to find Altera OpenCL platform.\n");
       return false;
       }
       // Query the available OpenCL device.

       devices.reset(getDevices(platform, CL_DEVICE_TYPE_ALL, &num_devices));

       printf("Platform: %s\n", getPlatformName(platform).c_str());
       printf("Found %d devices in the board. Using only one device for this app\n", num_devices);
       for(unsigned i = 0; i < num_devices; ++i) {
         printf(" %s\n", getDeviceName(devices[i]).c_str());
       }

       target_device = devices[0];

       // Create the context.
       context = clCreateContext(NULL, num_devices, &target_device, &oclContextCallback, NULL, &status);
       checkError(status, "Failed to create context");

       std::string binary_file = getBoardBinaryFile("cnn_kernels", target_device);
       printf("Using AOCX: %s\n", binary_file.c_str());

       program = createProgramFromBinary(context, binary_file.c_str(), &target_device, num_devices);

       // Build the program that was just created.
       status = clBuildProgram(program, 0, NULL, "", NULL, NULL);
       checkError(status, "Failed to build program");

       kernel.reset(num_kernels);

       // Command queue.
       queue = clCreateCommandQueue(context, target_device, CL_QUEUE_PROFILING_ENABLE, &status);
       checkError(status, "Failed to create command queue");

       // Kernel.
       kernel[0] = clCreateKernel(program, "filter3D", &status);
       checkError(status, "Failed to create kernel");
       kernel[1] = clCreateKernel(program, "maxpool3D", &status);
       checkError(status, "Failed to create kernel");
       kernel[2] = clCreateKernel(program, "iplayer", &status);
       checkError(status, "Failed to create kernel");
       kernel[3] = clCreateKernel(program, "relu_layer", &status);
       checkError(status, "Failed to create kernel");
       kernel[4] = clCreateKernel(program, "softmax", &status);
       checkError(status, "Failed to create kernel");

       return true;
}
```

Table A.4: Initialization of OpenCL Objects for a GPU Device

```
bool init_opencl() {
       cl_int status;
       int err;
       cl_platform_id platform_ids[5];
       char *KernelSource;
       long lFileSize;
       cl_uint dev_cnt = 0;

       printf("Initializing OpenCL\n");
       clGetPlatformIDs(0, 0, &dev_cnt);

       clGetPlatformIDs(dev_cnt, platform_ids, NULL);
       int gpu = 1;
       for(unsigned i = 0;i < dev_cnt; i++)
       {
          err = clGetDeviceIDs(platform_ids[i], gpu ? CL_DEVICE_TYPE_GPU:CL_DEVICE_TYPE_CPU, 1, &target_device, NULL);
          if(err == CL_SUCCESS)
          {
               break;
          }
       }
       if (err != CL_SUCCESS)
       {
          printf("Error: Failed to create a device group!\n");
          return EXIT_FAILURE;
       }

       // Create the context.
       context = clCreateContext(0, 1, &target_device, NULL, NULL, &err);
       if (!context)
       {
           printf("Error: Failed to create a compute context!\n");
           return EXIT_FAILURE;
       }

       lFileSize = LoadOpenCLKernel("device/cnn_kernels.cl", &KernelSource);
       if( lFileSize < 0L ) {
              perror("File read failed");
              return 1;
       }

       program = clCreateProgramWithSource(context, 1, (const char **) & KernelSource, NULL, &err);
       if (!program)
       {
              printf("Error: Failed to create compute program!\n");
              return EXIT_FAILURE;
       }

       // Build the program that was just created.
       status = clBuildProgram(program, 0, NULL, "", NULL, NULL);
       checkError(status, "Failed to build program");

       kernel.reset(num_kernels);
       // Command queue.
       queue = clCreateCommandQueue(context, target_device, CL_QUEUE_PROFILING_ENABLE, &status);
       checkError(status, "Failed to create command queue");

       // Kernel.
       kernel[0] = clCreateKernel(program, "filter3D", &status);
       checkError(status, "Failed to create kernel");
       kernel[1] = clCreateKernel(program, "maxpool3D", &status);
       checkError(status, "Failed to create kernel");
       kernel[2] = clCreateKernel(program, "iplayer", &status);
       checkError(status, "Failed to create kernel");
       kernel[3] = clCreateKernel(program, "relu_layer", &status);
       checkError(status, "Failed to create kernel");
       kernel[4] = clCreateKernel(program, "softmax", &status);
       checkError(status, "Failed to create kernel");
       return true;
}
```

Appendix B

FHEW

Table B.1: Non-recursive 1-D FFT

```
void FFT(complex_t input[2*N], complex_t output[N])
{
    int k, n;
    complex_t mul_res, conv_res, add_res;
    // Compute N DFTs of length 1 using naive method
    FFT_label0:for (k = 0; k < N; k++)
    {
       conv_res.re = 1;
       conv_res.im = 0; // -2*PI*n*k2/N2 --> k2 = 0, n = 0 for 1D Convolution
       multiply(input[k], conv_res, &mul_res);
       /* Multiply by the twiddle factors ( e^(-2*pi*j/N * k1*k2)) and transpose
          conv_from_polar(1, -2.0*PI*k1*k2/N, &conv_res); --> k2 = 0 for 1D Convolution */
       multiply(conv_res, mul_res, &mul_res);
       input[k] = mul_res;
    }

    for(k = 0; k < N; k++)
    {
       output[k].re = 0.0;
       output[k].im = 0.0;
       FFT_label2:for(n = 0; n < N; n++)
       {
          std::complex<double> t = std::polar(1.0, -2 * PI * n* k / N);
          conv_res.re = std::real(t);
          conv_res.im = std::imag(t);
          multiply(input[n], conv_res, &mul_res);
          add(output[k], mul_res, &add_res);
          output[k] = add_res;
       }
    }
 }
//------------------Utility Functions-------------------------
 void add(complex_t left, complex_t right, complex_t* result)
 {
    (*result).re = left.re + right.re;
    (*result).im = left.im + right.im;
 }

 void multiply(complex_t left, complex_t right, complex_t* result)
 {
    (*result).re = left.re*right.re - left.im*right.im;
    (*result).im = left.re*right.im + left.im*right.re;
 }
```

Table B.2: Radix-2 DIF FFT

```
void fft(Complex x[2*N])
{
	unsigned int k = N, n;
	double thetaT = 3.14159265358979323846264338328L / N;
	Complex phiT = Complex(cos(thetaT), sin(thetaT)), T;

	fft_label0:for(unsigned int i=0; i < 10; i++)
//because 2^10 = 1024 - After 9 right shifts, k becomes <= 1 => 2^0 = 1
	{
		n = k;
		k >>= 1;
		phiT = phiT * phiT;
		T = 1.0L;
		fft_label1:for (unsigned int l = 0; l < k; l++)
		{
			fft_label2:for (unsigned int a = l; a < N; a += n)
			{
				unsigned int b = a + k;
				Complex t = x[a] - x[b];
				x[a] = x[a] + x[b];
				x[b] = t * T;
			}
			T *= phiT;
		}
	}

	// Decimate
	unsigned int m = (unsigned int)log2(N);
	fft_label3:for (unsigned int a = 0; a < N; a++)
	{
		unsigned int b = a;
		// Reverse bits
		b = (((b & 0xaaaaaaaa) >> 1) | ((b & 0x55555555) << 1));
		b = (((b & 0xcccccccc) >> 2) | ((b & 0x33333333) << 2));
		b = (((b & 0xf0f0f0f0) >> 4) | ((b & 0x0f0f0f0f) << 4));
		b = (((b & 0xff00ff00) >> 8) | ((b & 0x00ff00ff) << 8));
		b = ((b >> 16) | (b << 16)) >> (32 - m);
		if (b > a)
		{
			Complex t = x[a];
			x[a] = x[b];
			x[b] = t;
		}
	}
}
```

Bibliography

[1] Andrej Karpathy. Cs231n convolutional neural networks for visual recognition. `http://cs231n.github.io/convolutional-networks`, 2016.

[2] Industry Association et al. International technology roadmap for semiconductors: 1999 edition. *International Sematech, Austin, Texas*, 1999.

[3] Ofer Rosenberg. "opencl overview. *Khronos Group*, 2011.

[4] A.J.Guillon. Opencl 1.2: High-level overview. `https://www.youtube.com/watch?v=8D6yhpiQVVI&list=PLhqBhHU0mKh9zeei8cdnEe4I5e6ZMNlqy`, 2013.

[5] Xilinx. Introduction to fpga design with vivado high-level synthesis. `https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf`, 2013.

[6] Deshanand Singh. Implementing fpga design with the opencl standard. *Altera whitepaper*, 2011.

[7] Divya Poddar. A beginner's guide to machine learning. `https://upxacademy.com/introduction-machine-learning`, 2016.

[8] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[9] Adil Moujahid. A practical introduction to deep learning with caffe and python. http://adilmoujahid.com/posts/2016/06/introduction-deep-learning-python-caffe, 2016.

[10] Andrej Karpathy. Cs231n winter 2016 lecture 7 convolutional neural networks. https://www.youtube.com/watch?v=AQirPKrAyDg, 2016.

[11] Wikipedia. Rectifier (neural networks). https://en.wikipedia.org/wiki/Rectifier_(neural_networks), 2017.

[12] MathWorks. Softmax transfer function. https://www.mathworks.com/help/nnet/ref/softmax.html?searchHighlight=softmax&s_tid=doc_srchtitle, 2006.

[13] Nachiket Kapre Gopalakrishna Hegde, Nachiappan Ramasamy. Opencl labs for papaa summer school 2016 edition. https://github.com/gplhegde/papaa-opencl/tree/master/cpp-ref, 2016.

[14] Uma Syam Prashant Ravi. Device query in opencl. https://github.com/umaurmi/OPENCL_EXAMPLES_ZEDBOARD.git.

[15] Nachiket Kapre Gopalakrishna Hegde, Nachiappan Ramasamy. Opencl labs for papaa summer school 2016 edition. https://github.com/gplhegde/papaa-opencl/tree/master/opencl-src/mnist-altera, 2016.

[16] Takuro Iizuka Akihiro Asahara Jeongdo Son Satoshi Miki Ryoji Tsuchiyama, Takashi Nakamura. The opencl programming book. https://www.fixstars.com/en/opencl/book/OpenCLProgrammingBook/online-offline-compilation/, 2012.

[17] SDK Altera. for opencl: Best practices guide, altera, 2015.

[18] IBM T.J. Watson Research Center Shai Halevi. Fully homomorphic encryption i : Cryptography boot camp. https://simons.berkeley.edu/talks/shai-halevi-2015-05-18a, 2015.

[19] Craig Gentry. Computing arbitrary functions of encrypted data. *Communications of the ACM*, 53(3):97–105, 2010.

[20] University of Toronto Vinod Vaikuntanathan. Homomorphic encryption: What, why and how? https://www.cs.toronto.edu/~vinodv/Homomorphic-MCSS.pptx, 2017.

[21] Léo Ducas and Daniele Micciancio. Fhew: Bootstrapping homomorphic encryption in less than a second. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 617–640. Springer, 2015.

[22] Alexandr Notchenko, Ermek Kapushev, and Evgeny Burnaev. Sparse 3d convolutional neural networks for large-scale shape retrieval. *arXiv preprint arXiv:1611.09159*, 2016.

[23] AMD. What is heterogeneous system architecture ? http://developer.amd.com/resources/heterogeneous-computing/what-is-heterogeneous-system-architecture-hsa.

[24] Wikipedia. Heterogeneous computing. https://en.wikipedia.org/wiki/Heterogeneous_computing, 2017.

[25] Wikipedia. Hardware acceleration. https://en.wikipedia.org/wiki/Hardware_acceleration, 2017.

[26] Sam Siewert. Soc drawer: Soc design for hardware acceleration, part 1. https://www.ibm.com/developerworks/library/pa-soc8/pa-soc8-pdf.pdf, 2006.

[27] Intel. Intel® sdk for opencl™ applications. https://software.intel.com/en-us/intel-opencl/download.

[28] Intel. Opencl™ runtime 16.1 for intel® core™ and intel® xeon® processors for ubuntu* (64-bit). https://software.intel.com/en-us/articles/opencl-drivers#latest_linux_driver.

[29] Christopher V Dobson. *An Architecture Study on a Xilinx Zynq Cluster with Software Defined Radio Applications*. PhD thesis, Citeseer, 2014.

[30] Xilinx. The first generation of extensible processing platforms: A new level of performance, flexibility and scalability. http://www.uio.no/studier/emner/matnat/ifi/INF3430/h11/undervisningsmateriale/roarsk/Xilinx_ZYNQ_Product_Brief.pdf, 2011.

[31] Khronos OpenCL Working Group et al. The opencl specification. *version*, 1(29):8, 2008.

[32] Jonathan Tompson and Kristofer Schlachter. An introduction to the opencl programming model. *Person Education*, 49, 2012.

[33] Vincent Hindriksen. Why use opencl on fpgas? https://streamcomputing.eu/blog/2014-09-16/use-opencl-fpgas, 2014.

[34] Wikipedia. Machine learning. https://en.wikipedia.org/wiki/Machine_learning, 2017.

[35] Gopalakrishna Hegde, Nachiappan Ramasamy, Nachiket Kapre, et al. Caffepresso: an optimized library for deep learning on embedded accelerator-based platforms. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, page 14. ACM, 2016.

[36] Wikipedia. Feature extraction. https://en.wikipedia.org/wiki/Feature_extraction.

[37] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. Hardware-oriented approximation of convolutional neural networks. *arXiv preprint arXiv:1604.03168*, 2016.

[38] Corinna Cortes Christopher J.C. Burges Yann LeCun, Courant Institute. The mnist database of handwritten digits. http://yann.lecun.com/exdb/mnist/, 2012.

[39] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[40] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[41] Chris McCormick. Deep learning tutorial - softmax regression. http://mccormickml.com/2014/06/13/deep-learning-tutorial-softmax-regression/, 2014.

[42] Phil Mucci and The ICL Group. Current papi software and patches. http://icl.cs.utk.edu/papi/software/index.html, 2016.

[43] gsarkis. Libfi - a fixed-point arithmetic library. https://github.com/gsarkis/libfi, 2013.

[44] Khronos Group. Opencl-headers. https://github.com/KhronosGroup/OpenCL-Headers.git.

[45] Debian. Install mono on linux. http://www.mono-project.com/docs/getting-started/install/linux/.

[46] Intel. Intel fpga sdk for opencl™. http://dl.altera.com/opencl/?edition=pro&download_manager=direct, 2017.

[47] Intel. Intel fpga sdk for opencl™: Getting started guide. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl_getting_started.pdf, 2017.

[48] Nitin Gupta. Shared memory and bank conflicts in cuda. http://cuda-programming.blogspot.sg/2013/02/bank-conflicts-in-shared-memory-in-cuda.html, 2013.

[49] Khronos OpenCL Working Group. *The OpenCL Specification*. 2012.

[50] Craig Gentry and Shai Halevi. Implementing gentry's fully-homomorphic encryption scheme. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 129–148. Springer, 2011.

[51] Craig Gentry, Shai Halevi, and Nigel P Smart. Homomorphic evaluation of the aes circuit. In *Advances in Cryptology–CRYPTO 2012*, pages 850–867. Springer, 2012.

[52] Shai Halevi and Victor Shoup. Algorithms in helib. In *International Cryptology Conference*, pages 554–571. Springer, 2014.

[53] Leo Ducas and Daniele Micciancio. Fhew: A fully homomorphic encryption library, version 1.0. https://github.com/lducas/FHEW.git, 2014.

[54] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[55] Matteo Frigo and Steven G. Johnson. Fftw: Installation on unix. http://www.fftw.org/fftw3_doc/Installation-on-Unix.html#Installation-on-Unix, 2005.

[56] Vaikuntanathan Brakerski, Gentry. Helib. https://github.com/shaih/HElib.git, 2014.

[57] M Bhattacharya, R Creutzburg, and J Astola. Some historical notes on number theoretic transform. In *Proc. 2004 Int. TICS Workshop on Spectral Methods and Multirate Signal Processing*, volume 2004, 2004.

[58] IP LogiCORE. fast fourier transform v8. 0, 2012.