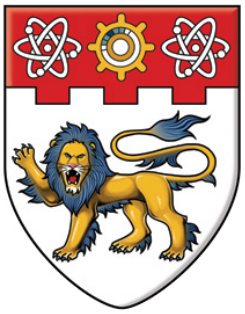


NANYANG TECHNOLOGICAL UNIVERSITY

ANALYSIS AND ACCELERATION OF COMPUTE-INTENSIVE APPLICATIONS ON HETEROGENEOUS PLATFORMS

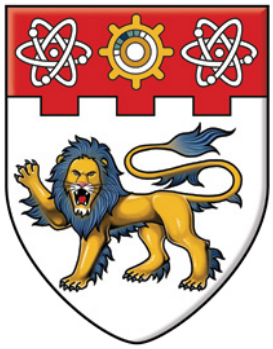
by

Swarna Kamakshi Jayaraman

G1601351B

School of Computer Science and Engineering

May 2017



NANYANG TECHNOLOGICAL UNIVERSITY

Analysis and Acceleration of Compute-Intensive Applications on Heterogeneous Platforms

by

Swarna Kamakshi Jayaraman

Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Embedded Systems

Supervised by

Assoc. Prof. Douglas L. Maskell

May 2017

Abstract

After nearly 40 years of pondering over CPU power and performance, Dr. David Patterson from Berkeley put forth the Three walls namely, the Power Wall, ILP (Instruction Level Parallelism) Wall and the Memory Wall, which marked the end of single-core computing systems[1]. While ILP Wall presented the diminishing returns from aggressive pipelining in superscalars, Power Wall revealed an exponential increase in power consumption with operating frequency. Memory wall projected the gap between compute bandwidth and memory bandwidth, which could not be bridged with increasing cache sizes and optimizations. With the emergence of multiple identical cores (Multicores of the sub-micron era) on a single chip came a new set of challenges, such as scalability, parallel software availability and power limit [2].

Recent research efforts have unveiled the strengths of heterogeneous computing platforms which are more than just differences in Instruction Set Architectures. Heterogeneous systems have gradually evolved with time, from multiple types of processors on a single chip (GPUs, NICs, FPGAs) to previously discrete components integrated onto a single System-On-Chip. There are several aspects to be addressed to harness the full potential of GPUs and FPGAs in mainstream computing applications. Hardware accelerators provide greater efficiency in realizing a design compared to software counterparts, with improved power economics. With growing complexity of the architectures, there is a pressing need for engineers to come up with better design decisions and partition the application suitably between hardware and software, to avoid any performance bottlenecks.

This report deals with identification of such compute-intensive applications which can be offloaded to hardware accelerators, efficient partitioning of application between hardware and software and gauging of the various

design points to achieve specific optimization goals.

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've had with my mentor, Mr.Abhishek Kumar Jain, a PhD student under Dr.Douglas Maskell. 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

List of Figures

List of Tables

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.

This calls for multi-core systems that achieve the desired performance by integrating specialized processing abilities required for specific tasks. Het-

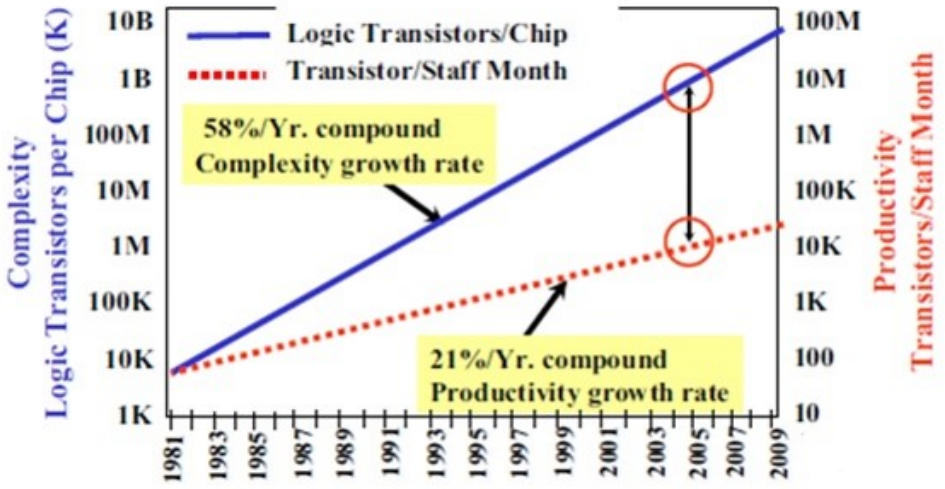


Figure 1.1: The Productivity Gap [4]

erogeneous system architectures exploit multiple processor types, to realize the best of both data-parallel and task-parallel (sequential) applications [21].

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 [22].

These devices guarantee a power-aware design with increased data paral-

lelism 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 memory access patterns (uniform or non-uniform) and interface types, etc. [22] 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 synchronizations 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 such as vector/ array processing 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 intends to investigate two such applications, namely MNIST Digit classifier using Convolutional Neural Network, and Fully Homomorphic Encryption scheme, and optimize them to achieve the best-case acceleration on a heterogeneous platform.

1.2 Contribution

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

- Identification and understanding of two applications of 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 improvements.
- Performing architectural exploration and suitably partitioning the application between various compute elements available in the platform.
- Running and profiling the applications to study the performance improvements with the modified design.

1.3 Organization

The report consists of the following chapters: **Chapter 2** presents background software knowledge and hardware requirements needed for this project. **Chapter ??** studies an implementation of OpenCL platform and its features. Few details of the LeNet-5 architecture of Convolutional Neural Networks are also explained. The chapter continues to discuss previously used ISAs and the need for a new ISA which was met by RISC-V. Implementations of processors based on this architecture are also seen here. **Chapter ??** explains in detail how OpenCL capitalises on devices with parallel programming capabilities. A few simple experiments are conducted to demonstrate how optimum results can be achieved. **Chapter ??** throws light into the field of neural networks, exploring few models of neural networks. The work of Lecun, Y. in the field of character recognition using Convolutional Neural Network (CNN) and Deep Learning is discussed in detail. An existing application is analysed and profiled on multiple devices. **Chapter ??** shows how an open-source architecture would benefit the evolution of technology. RISC-V is studied as

the architecture to be used for the development of a hypothetical accelerator consisting of a network of individual processors. The implementation of a new processor based on this ISA is discussed and explained in detail. We thereby conclude in **Chapter ??** and discuss work to be done in the future.

Chapter 2

Background

2.1 What is Hardware Acceleration?

Migration of some applications originally implemented in software 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 [23]. 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. [24] 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 heterogeneous platform has been chosen to demonstrate code portability across different compute elements, evaluate the runtime of an application in CPU and GPU, analyze whether the given application is control-bound or compute-bound, and estimate the percentage improvement in latency. The **Intel SDK for OpenCL**[25] is available for both Windows and Linux Operating Systems and offers packages to run applications on Intel CPU and Graphics Processing Unit. Also, the **OpenCL Runtime Environment** (RTE) [26] provides drivers and library packages required to test applications while they are running. The installation of these packages is easy and will be discussed in detail in Section¡TBD¿. The CPU used for this thesis is ¡¿(¡¿cores) running at a clock frequency of ¡¿ and the GPU is ¡¿ (¡¿ cores) running at a frequency of ¡¿.

2.2.2 Avnet Zedboard with Xilinx Zynq 7000 All-programmable SoC

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 [27], Aerospace and Defense, Medical diagnostics and Imaging, Wired and Wireless communication, Control and Bridging applications [28]. 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 [29]. 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.

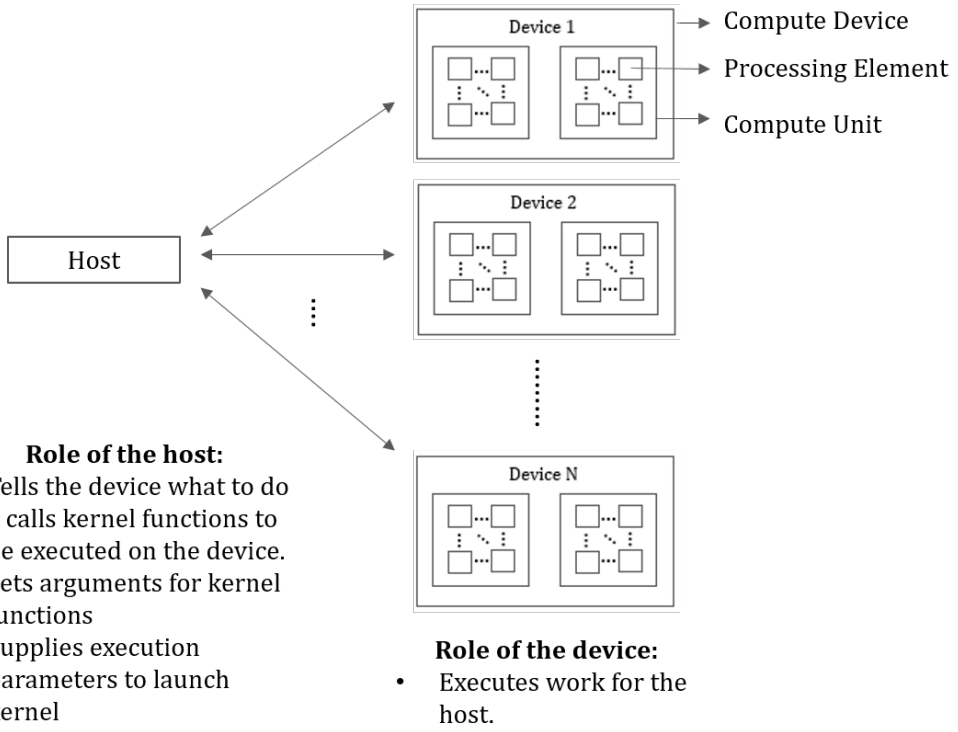


Figure 2.1: The OpenCL Platform Model [5]

There are three popular OpenCL Models [6], which shall be discussed briefly to aid the understanding of internals in OpenCL Programming.

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 compute units are composed of several Processing Elements (PEs). Hence, Compute units can be viewed as containers of very simple processors (PEs).

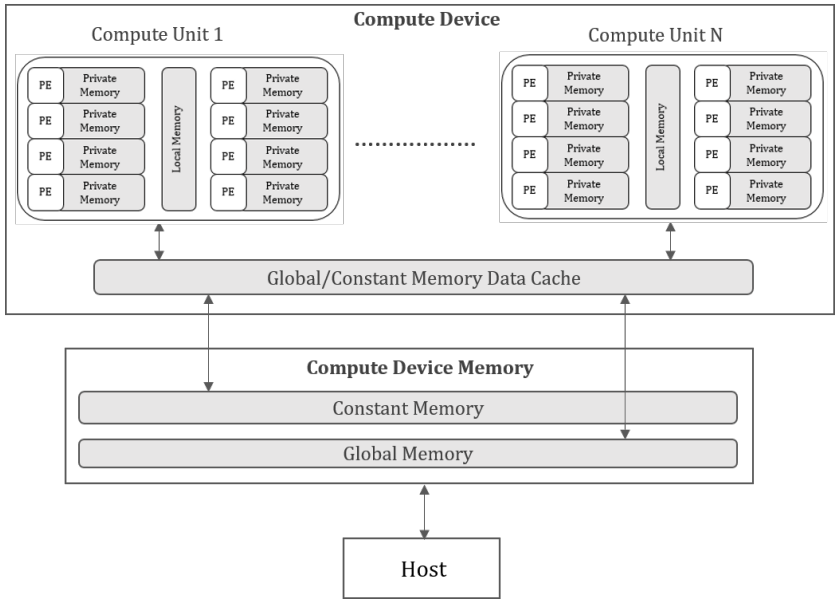


Figure 2.2: The Device Model of OpenCL [6]

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.
- **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 in parallel on different PEs over a pre-defined N-dimensional index space [30].

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.

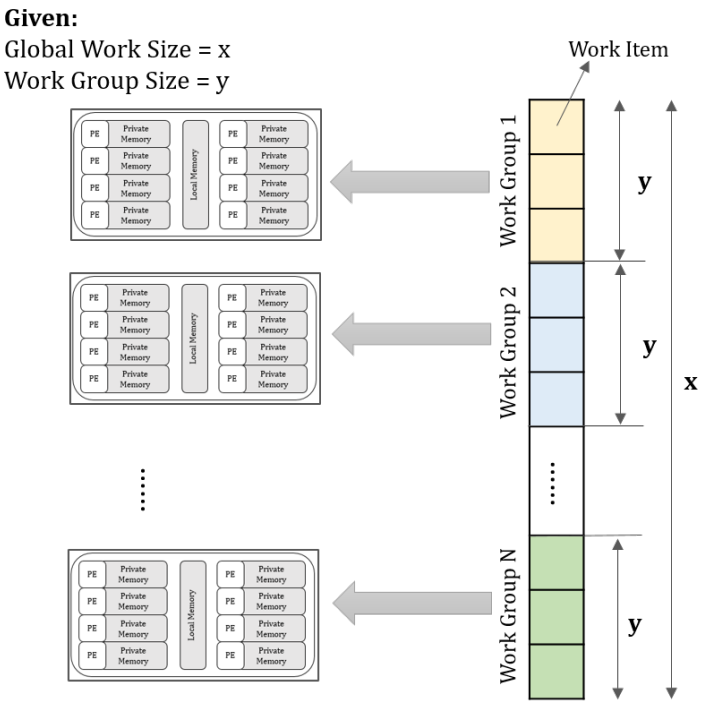


Figure 2.3: An Illustration of Data-parallel execution [6]

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

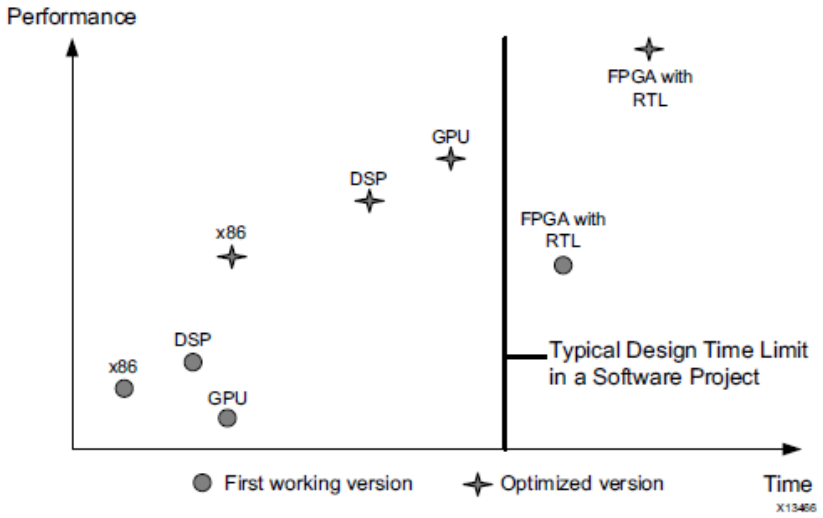


Figure 2.4: Performance vs. Design Time with RTL Design [7]

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 generated 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 determin-

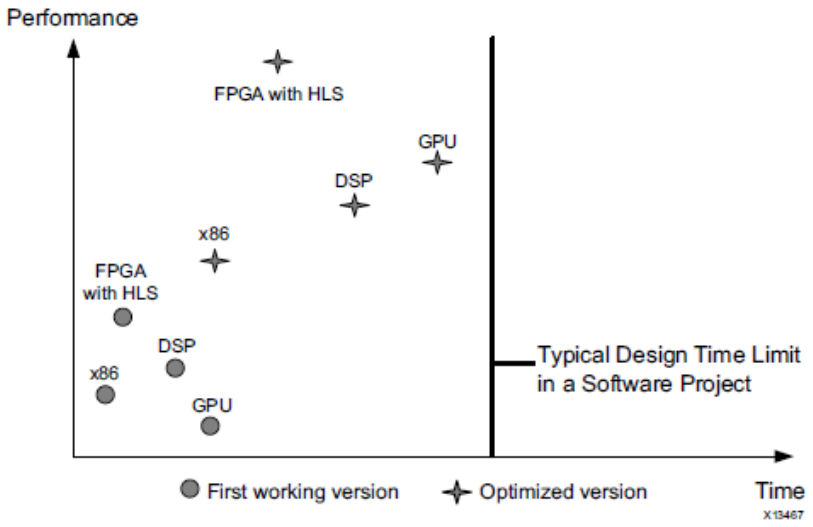


Figure 2.5: Performance vs. Design Time with HLS Compiler [7]

istic at compile-time. This static memory allocation facilitates realization of an algorithm's memory as a register, FIFO or Block RAM [7].

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 several performance goals.

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 [31]. 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 [8]. 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 [8]. Figure 2.6 depicts the pipelined execution 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

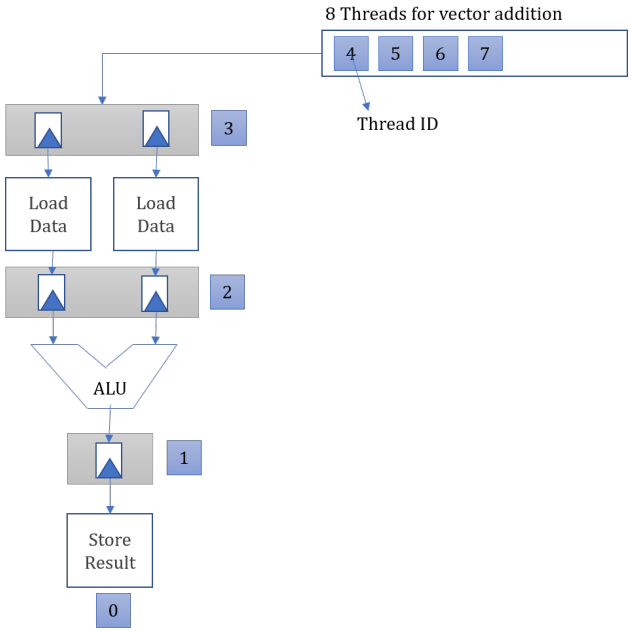


Figure 2.6: Pipeline Parallelism achieved by OpenCL-to-FPGA compiler [8]

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 and includes that are available with this SDK (Example: aocl_utils.h). Hence, this thesis shall make use of Altera OpenCL (AOCL) SDK to analyze and modify the code and study the results.

Chapter 3

Literature Review

Chapter 4

Hardware Acceleration using Graphics Processing Unit

It is no secret that many technical giants such as NVIDIA, IBM, Google and Fujitsu have recently invested a lot to train engineers and carry out research in the field of Artificial Intelligence, to harness automated solutions that will bridge the current gap between technology integration and expertise. Deep Learning is an avant-garde approach to imparting knowledge to the machines to achieve the ultimate goal of artificial intelligence, without explicit coding. It is of interest in several domains[32], 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[33]. We seek to validate the efficiency of deep-learning methods on heterogeneous architectures with a simple Lenet-5 Model of MNIST Dataset classifier.

4.1 Deep Learning using Convolutional Neural Networks

The Figure 4.1 shows the most common types of learning algorithms. The choice of the algorithm depends on the problem we intend to solve.

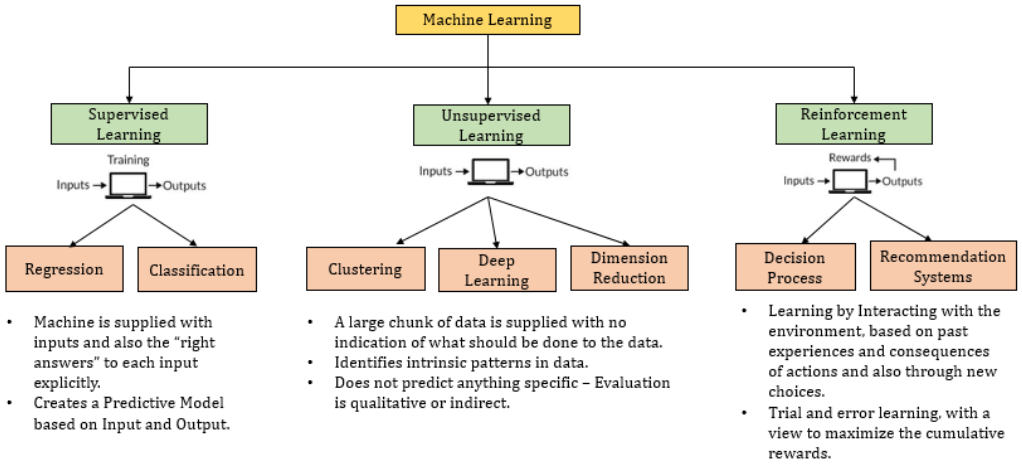


Figure 4.1: Types of Machine Learning Algorithms [9]

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 [10]. 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 [34]. 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 [34]. 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 4.2: Formula to determine Classifier Efficiency [10]

Studies have revealed the relationship between expected error rate on test set and error rate on training set as shown in Figure 4.2. 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 can 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 [11].

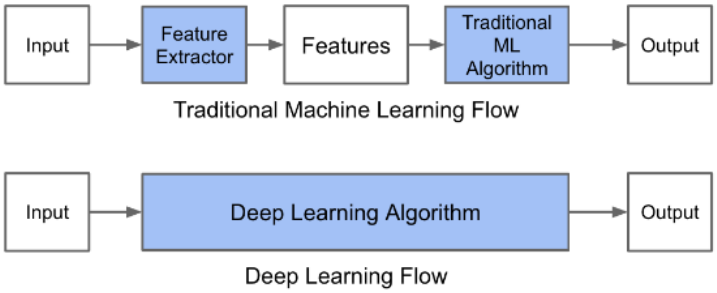


Figure 4.3: Learning differences - Traditional vs. Deep Learning [11]

Owing to high computational complexity, CNN usage is restricted, especially in portable devices [35].

4.1.1 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 [36] was used for the experiments discussed in the paper [10]. This paper proved the general consensus 4.3 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 perfor-

mance with low runtime complexities. Figure 4.4 shows the original Lenet-5

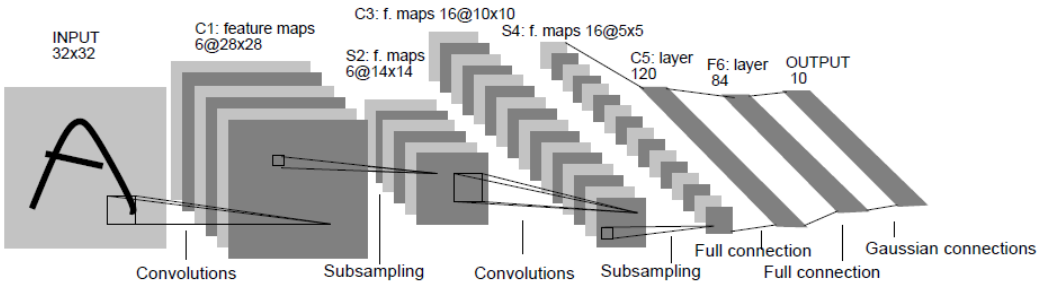


Figure 4.4: Original Lenet-5 ConvNet Architecture; Each plane represents a feature map in which weights are shared. [10]

architecture described in [10]. It is important to understand the purpose of various layers of the Lenet-5 ConvNet architecture to study their runtime in the application. The following subsections shall describe the layers in detail.

4.1.1.1 Convolution Layer [3]

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

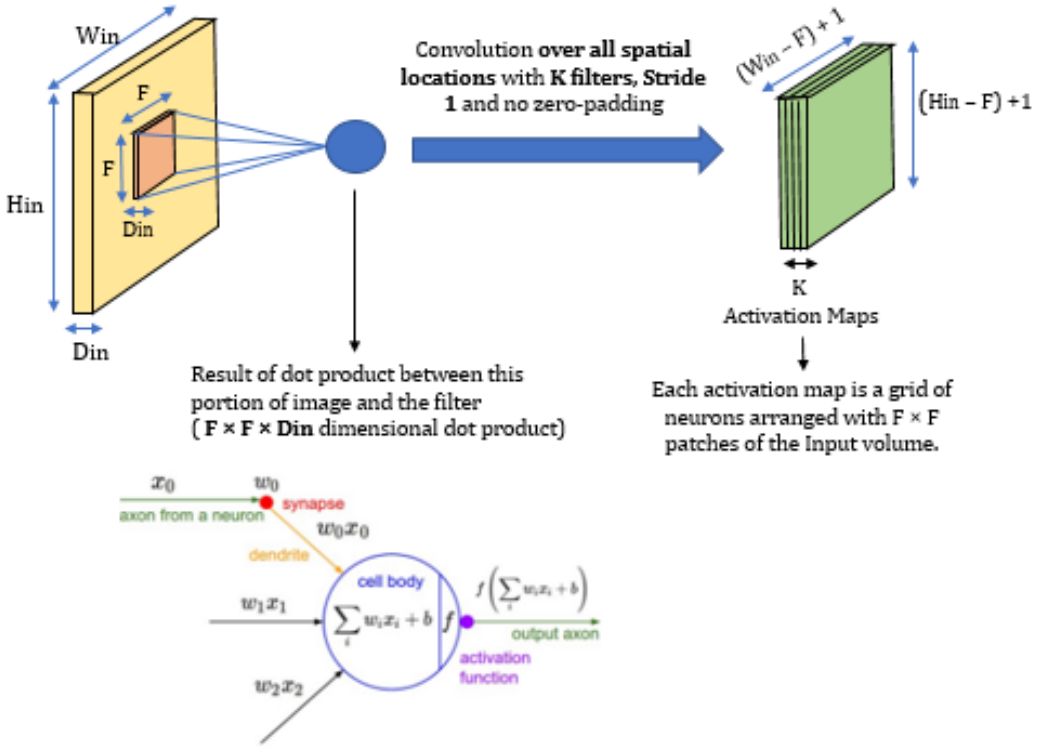


Figure 4.5: Convolution Layer [12]

up an entire feature hierarchy.

Each stage builds up very specific features which filters in the subsequent stages will be excited about. i.e. piece by piece, we create 3-D volumes of higher levels of abstraction than the previous stage[37].

Generalization of Concepts:

Required Hyperparameters:

Number of filters, K

Spatial Extent of the filter, F

Stride, S

Quantum of Zero padding, P (Figure 4.6)

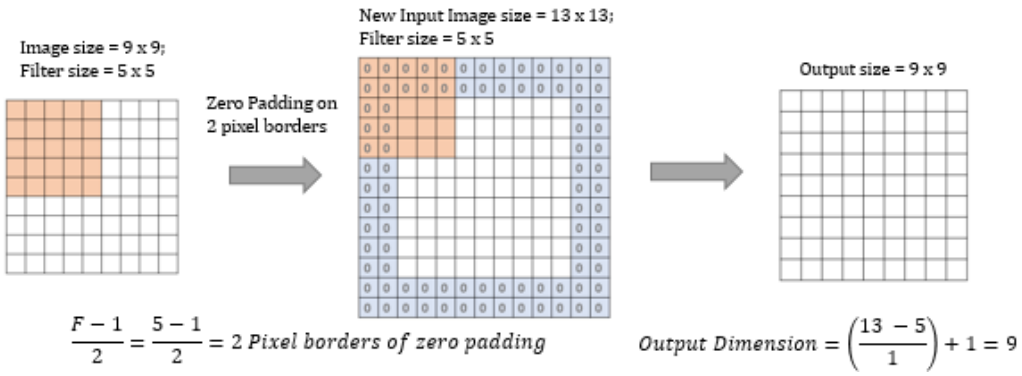


Figure 4.6: Preserving Input dimensions using Zero Padding [12]

Input Dimensions: $W_1 \times H_1 \times D_1$
Output Dimensions: $W_2 \times H_2 \times D_2$,

Where

$D_2 = \mathrm{K}$

$W_2 = ((W_1 - \mathrm{F} + 2\mathrm{P})/\mathrm{S}) + 1$

$H_2 = ((H_1 - \mathrm{F} + 2\mathrm{P})/\mathrm{S}) + 1$

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) $\times$ 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 [12].

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 $\times$ 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 4.7. 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 4.7 are all acting on the same input patch but with different weights.

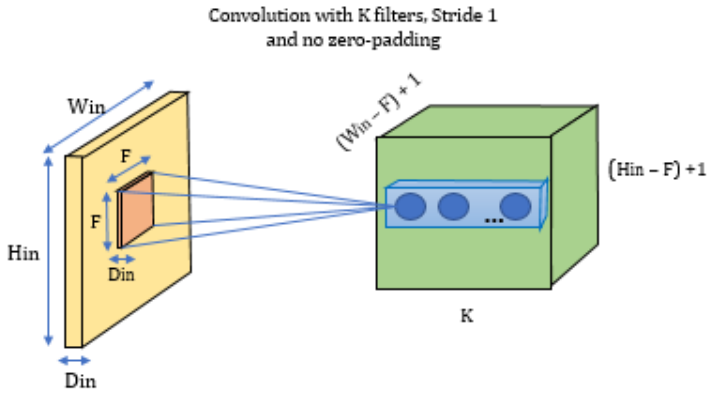


Figure 4.7: 3-dimensional volume of Neurons [12]

4.1.1.2 MaxPool Layer:

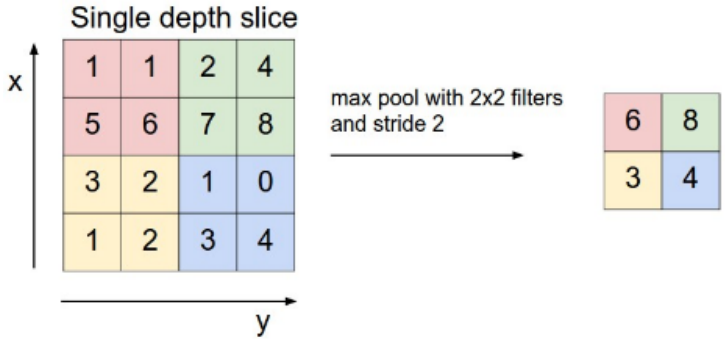


Figure 4.8: Max Pooling [3]

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 4.8.

4.1.1.3 Inner Product Layer

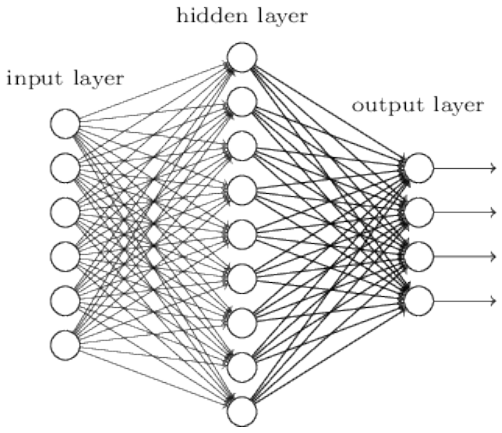


Figure 4.9: Inner Product Layer [3]

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.

4.1.1.4 ReLU Layer

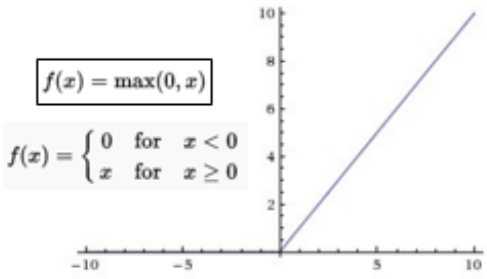


Figure 4.10: ReLU Function in Neural Networks [13]

Rectified Linear Unit [38] is a non-linear activation function described by Figure 4.10, 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.

4.1.1.5 Softmax Layer

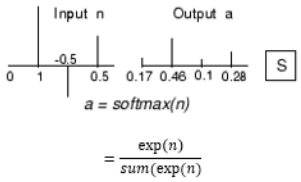


Figure 4.11: Softmax Function [14]

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 [39].

4.1.1.6 Modified Hyperparameters for MNIST Dataset

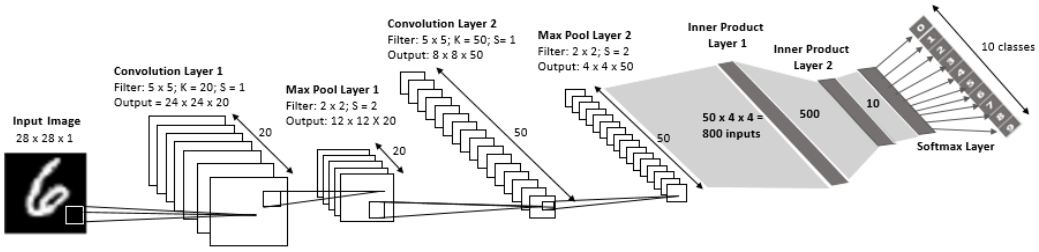


Figure 4.12: Lenet-5 CNN Architecture for MNIST Dataset with modified hyperparameters [15]

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 [10]. 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 4.4 uses 32×32 images while the benchmark code [15] that will be used for our experiments uses 28x28 images. Table 4.1 defines the hyper-parameters for the different layers of the MNIST/Lenet-5 CNN benchmark application[15].

4.1.2 Experiments with C++ Code

4.1.2.1 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:

```
1 $ sudo apt-get install papi-tools
```

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 4.1: Hyperparameters for Lenet-5 CNN described in MNIST/Lenet-5 ConvNet Benchmark code[15]

Download PAPI files from the official PAPI Website [40].

```
$ 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
```

4.1.2.2 Existing Code Flow Description

Figure 4.13 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 the pre-defined image label and calculate the prediction accuracy.

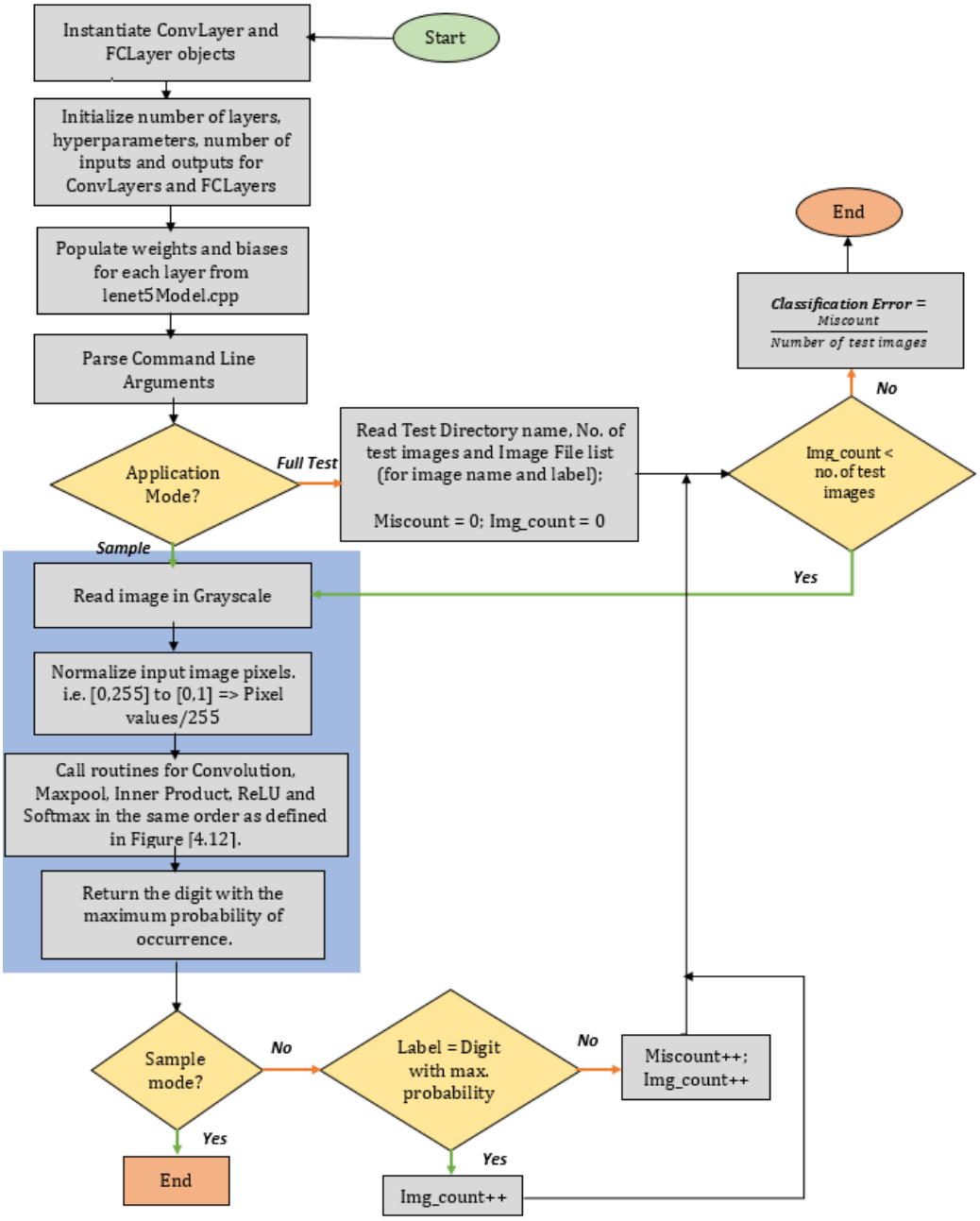


Figure 4.13: Software Code Flow [15]

Compilation Steps [15]

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 4.2. 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. The convolution layer involves about 86% ((12308+48164)/70029) of the required arithmetic operations in the ConvNets framework. Following this, the fully connected layers are the next most resource-intensive layers.

4.1.2.3 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

Table 4.2: Analysis of Application Hot-spots for Acceleration

Layers	Computation Time (usec)
Convolution 1	12308
MaxPool 1	3382
Convolution 2	48164
MaxPool 2	559
Inner Product 1	5522
ReLU	12
Inner Product 2	73
Softmax	9

point arithmetic libraries like LibFi [41]. 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 Graphics Processing Unit for concurrent execution and significant speedup. 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. We seek to accelerate the layers of the ConvNet by using fine-grained GPUs which exhibit a high degree of data-parallelism.

4.1.3 Experiments with OpenCL Code

4.1.3.1 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 [42]:
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) [26]
- Intel SDK for OpenCL™ Applications [25]

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 [43]).
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 [16] 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 4.14.

AOCL SDK and Quartus Installation Steps

The FPGA Implementation of MNIST digit recognition [17] uses Altera OpenCL (AOCL) SDK (also called Intel FPGA SDK) and Quartus Software for high-level synthesis and execution. Although our experiments are

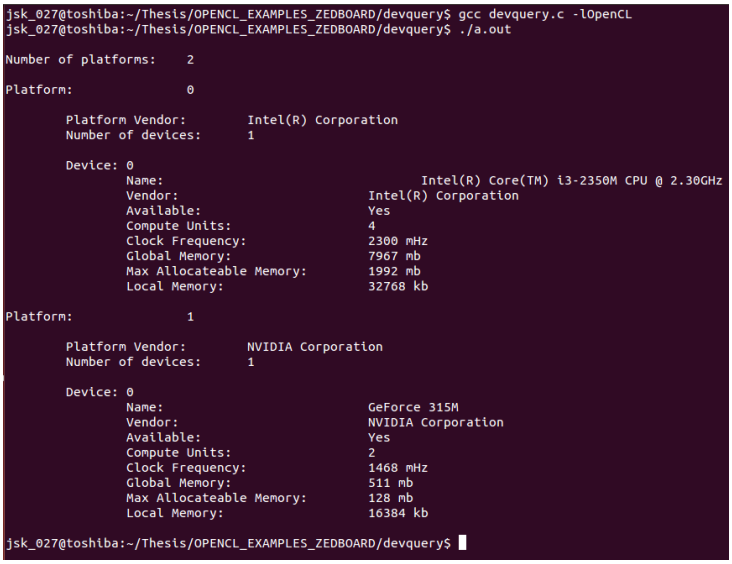


Figure 4.14: OpenCL Device Query code Output [16]

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 [44]. The installation steps of AOCL and Quartus from the extracted tarball are detailed in [45]. 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
4 $ 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)
```

These steps are sufficient to run the modified OpenCL code for GPU, without dependencies on Altera platform.

4.1.3.2 Existing Code Flow Description

The OpenCL implementation of MNIST/Lenet-5 architecture available in the repository [17] 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 4.15 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 the platform be available, the next step is to query all OpenCL devices in

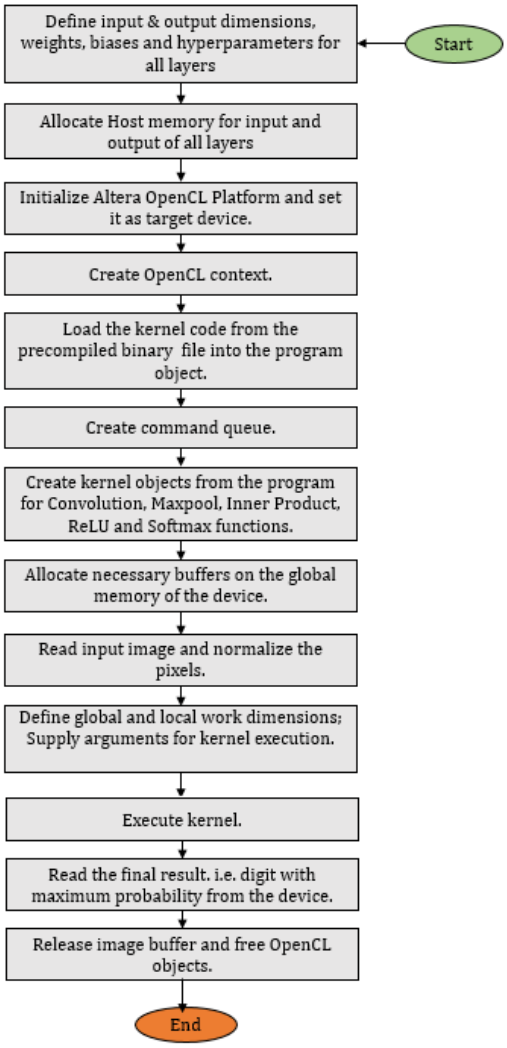


Figure 4.15: OpenCL code flow for Sample (single image recognition) mode [17]

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 [18]. **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 embedded systems which serve real-time 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 [17] is spe-

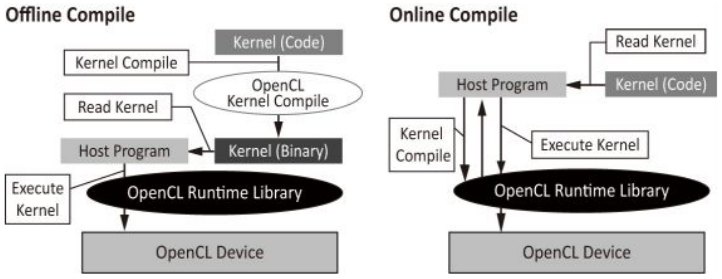


Figure 4.16: Kernel Compilation Modes [18]

cific to Altera devices and hence uses 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, all layer operations

are offloaded to the GPUs. Hence, kernel objects are created for Convolution, Maxpooling, Inner Product and Activation Layers (ReLU and Softmax). In order t execute the kernel calls on the OpenCL device, memory has to allocated, which is enough to support the weights, biases and IO dimensions for all 8 layers of the Lenet-5 Model.

The input image pixels are read and normalized. The kernel code is executed on the decide after the kernel arguments are supplied to all layers. The final result, i.e. the digit with maximum likelihood is read from the device, buffers and memory objects freed.

4.1.3.3 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 [19]. It uses the device SDRAM

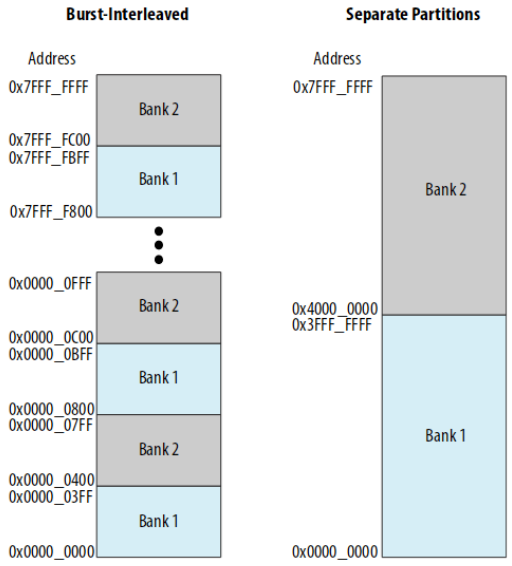


Figure 4.17: Default (bus-interleaved) vs. Manual Global Memory Partitioning [19]

as global memory and by default, stores the data in a burst-interleaved fashion 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 types, data cannot be impartially interleaved to these banks. AOC directs the kernels to use up contiguous memory locations in the global memory when load and store operations occurring sequentially during kernel execution.

The code [17] uses optimized global memory access using memory banks instead of default allocation of data in the global memory in burst fashion. Here, the data has been stored in Bank 1 while the weights and biases are stored in Bank 2 for efficient global memory access. However, in case of GPUs, memory banks 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 [46]. 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 4.1: 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 4.2: 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 [47].

Kernel Loading Mechanism

On-the-fly kernel loading, i.e. Online compilation method explained in subsection 4.1.3.2 is employed. The existing and modified code changes are depicted clearly in A.3, A.4 and A.2.

Changes to Makefile

As the test platform contains an Intel CPU and NVIDIA GPU, the Intel OpenCL Library (which is a part of the Intel FPGA SDK for OpenCL) is required. *libOpenCL.so* Shared Library is linked in the Makefile as shown in Figure 4.18.

After these changes, the resultant code is free of any dependencies with Altera platform and Intel FPGA SDK. The code can be compiled and run on any OpenCL device.

4.1.3.4 Compiling and executing the code

After the changes listed above are done, there are two modes in which the application can be run as follows:

To test a sample image:

```
$ make run
```

To test the full MNIST Dataset:

```
$ make test
```

```
# 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 4.18: Linking libOpenCL Library to Makefile

The path of the test images (sample and full dataset) is also supplied to the host using 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 4.3: 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);
```

4.1.4 Comparative Study of Results

Test Devices

1. **Intel OpenCL** from Intel(R) Corporation
   OpenCL Version: OpenCL 1.2 LINUX
   Compute Units: 4

2. **NVIDIA CUDA** from NVIDIA Corporation
   OpenCL Version: OpenCL 1.1 CUDA 4.2.1
   Compute Units:2

Table 4.3: Comparison of kernel runtime in various OpenCL Devices

	Kernel Execution Time (ms)	
	Intel Core i3-2350M CPU @ 2.30GHz (4 CUs)	NVIDIA GeForce 315M (2 CUs)
Convolution 1	0.216	0.707
MaxPool 1	0.046	0.166
Convolution 2	0.716	11.332
Maxpool 2	0.026	0.371
Inner Product 1	0.187	1.651
ReLU	0.011	0.012
Inner Product 2	0.010	0.287
Softmax	0.014	0.012

Chapter 5

Hardware Acceleration using FPGA

5.1 Fully Homomorphic Encryption Scheme

Cyber-security is a topic of great importance today owing to increasing instances of hacking, ransomeware attacks and identity thefts. Encryption forms a crucial part of all security schemes. Recently, a lot of interest has sprung in the area of Homomorphic encryption schemes which allow computations in the cloud without exposing the data. The Figure 5.1 illustrates the problem statement that we seek to address through this scheme. The idea of the fully homomorphic encryption is to delegate data-processing (i.e. Arithmetic and Logical Operations on data) to the cloud without giving away the original data. Fully homomorphic encryption scheme ϵ involves 4 key steps:

- $KeyGen_{\epsilon}(\lambda)$**:** It involves generation of random odd integer p, which is P-bits long as the secret key. The security parameter Λ dictates the bit-length of the key.

  – Symmetric Encryption:
  The same secret key is used for both Encryption and Decryption.

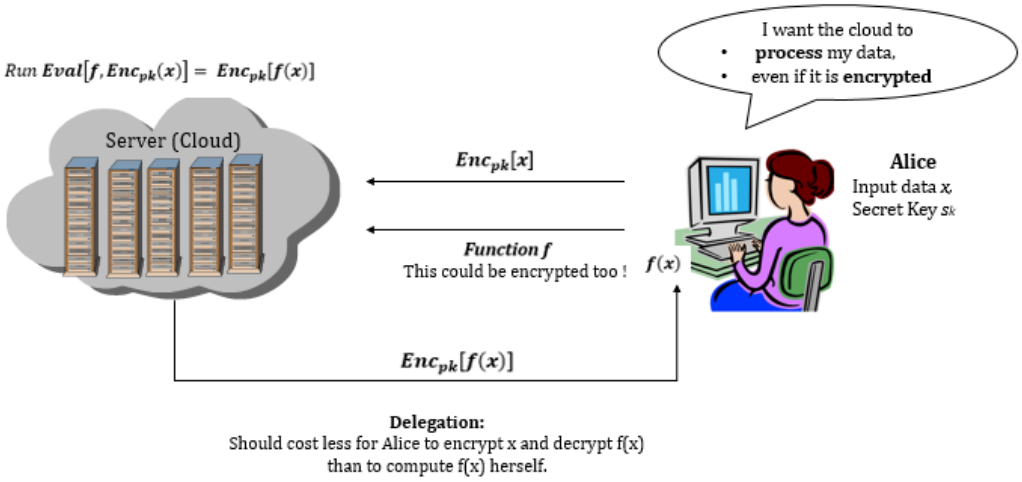


Figure 5.1: FHE: Problem Statement [20]

– Asymmetric Encryption:
It is a public key encryption scheme and comprises of a public encryption key (p_k) and a secret decryption key (s_k).

- $Encrypt_\epsilon(\boldsymbol{p,m})$**:**
Required Parameters:
Given the security parameter λ,
N = λ; P = λ^2; Q = λ^5
Scheme:
Consider a bit $m \in 0,1$.
To encrypt this bit, 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. $c \leftarrow m \ mod \ 2 + pq$

- $Decrypt_\epsilon(\boldsymbol{p,c})$**:**
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 as is observed in [48].

Decryption necessitates a *compact ciphertext requirement.* i.e. Any two ciphertexts c_1 and c_2 outputted from the encryption scheme should be of the same size in order to maintain a constant complexity of decryption[48]. Size of the ciphertext and the time taken to decrypt should be independent of the complexity of function *f*, delegated to the cloud to compute.

- $Evaluate_\epsilon(p_k$**,** ***f*****,** c_1**,** c_2**, ...** c_t**):**
  Evaluation is associated to a set of permissible functions F_ϵ. For any function f in F_ϵ and ciphertexts c_1, c_2... c_t, where $c_i \leftarrow Encrypt_\epsilon(p_k, m_i)$, the following 2 steps are performed:
  $c \leftarrow Encrypt_\epsilon[\, f\,(c_1,\, c_2,\, \ldots\, c_t)]$
  $Decrypt_\epsilon(\mathrm{c}\ ,\, s_k) = f\,(m_1,\, m_2,\, \ldots\, m_t)$

This scheme guarantees one-wayness. i.e. given the ciphertext c, it should be very hard to output the message m under public key p_k. i.e. The probability of success should be less than $\frac{1}{\lambda^k}$, for any constant k. It also provides semantic security against chosen plain-text attacks as it is probabilistic. There can be many ciphertexts which encrypt a given message and $Encrypt_\epsilon()$ chooses one at random.

Example

The homomorphic operations are defined as follos:
$Add_\epsilon(c_1,c_2) \rightarrow c_1 + c_2$
$Sub_\epsilon(c_1,c_2) \rightarrow c_1$ - c_2
$Mult_\epsilon(c_1,c_2) \rightarrow c_1\ .\ c_2$
Consider multiplication of 2 ciphertexts c_1 and c_2. $\Rightarrow$ c = $c_1\ .\ c_2$
c_i's noise is defined by m_i' = m_i (mod 2), where m_i and m_i' have the same

parity.
c = m_1'.m_2' + pq' for some q'.

Concept of Bootstrapping

5.1.1 Existing code flow

5.1.2 Hot-Spots for Hardware Acceleration

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 5.1. The accumulator in turn calls the 2048-point FFT and IFFT routines several times. The tabu-

Table 5.1: 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

lated 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 [49] 5.1.

This is because the circuit under test is defined by: (**a** NAND **b**) NAND (**c** NAND **d**).

The table[] illustrates the average execution time of a single FFT and IFFT, taken across –readings in a quad-core CPU. Since maximum calls are made to FFT and IFFT functions, porting these portions to hardware might facilitate parallel execution as opposed to sequential flow in software and speed it up further. Embedded devices are usually memory and power constrained. In the latest patch of FHEW Library [49], the FFTs are computed on double-precision floating point values which are 64 bits long. 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. These experiments shall also be covered in the forthcoming sections of this thesis.

5.1.3 Hardware Results

5.1.4 Comparative Analysis of results from software and 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

Bibliography

[1] Russell Fish. The future of computers - part 1: Multicore and the memory wall. http://www.edn.com/design/systems-design/4368705/The-future-of-computers--Part-1-Multicore-and-the-Memory-Wall, 2011.

[2] Chuck Moore. Data processing in exascale-class computer systems. In *The Salishan Conference on High Speed Computing*, 2011.

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

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

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

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

[7] 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.

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

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

[10] 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.

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

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

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

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

[15] 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.

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

[17] 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.

[18] 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.

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

[20] 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.

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

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

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

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

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

[26] 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.

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

[28] 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.

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

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

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

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

[33] 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.

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

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

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

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

[38] 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.

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

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

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

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

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

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

[45] 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.

[46] 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.

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

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

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