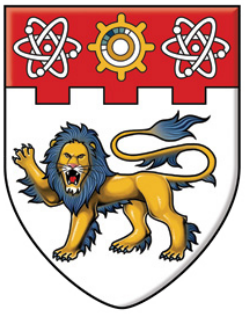


NANYANG
TECHNOLOGICAL
UNIVERSITY

UNDERSTANDING AND PROFILING A CNN APPLICATION ON DIFFERENT PLATFORMS USING OPENCL

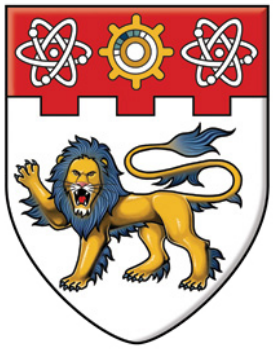
by

Shuvam Nandi

U1322990B

School of Computer Science and Engineering

27th March, 2017



NANYANG TECHNOLOGICAL UNIVERSITY

SCE16-0101
Understanding and Profiling a Convolutional Neural Network Application on Different Computing Platforms using OpenCL

by
Shuvam Nandi

Submitted in Partial Fulfillment of the Requirements for the Degree of Bachelor of Computer Engineering of the Nanyang Technological University

School of Computer Science and Engineering

27th March, 2017

Abstract

The decline of Moore's law has led to a fundamental shift in the design of micro-processor architectures. Devices with parallel processing architectures such as GPUs, FPGAs and DSPs initially used specifically for dedicated tasks are now gaining popularity as accelerators for more general-purpose computations. Performance is exploited in these devices by massively parallelising tasks across various compute units. CUDA and OpenCL are two application programming interface (API) models used to program parallel devices. The long-term objective this project seeks to achieve is the design of hypothetical network of multiple processors, capable of running applications in parallel.

OpenCL is used to facilitate comparison of performance being a cross-compatible framework across multiple heterogeneous platforms. Initially, this report examines the performance of numerous computing devices. A simple matrix multiplication kernel was executed with different mappings of the kernel onto the devices. This was followed by profiling a complex application recognising handwritten digits from the MNIST database. Performance in terms of GOPS was computed from the execution timings obtained and by analysing the number of computations performed in the application.

The second half of this project investigates free ISAs for implementing a processor as the core unit of the hypothetical engine. RISC-V is picked and studied as it provides several extensions to its base integer instruction set, thereby supporting computationally intensive tasks. An existing processor implementation is examined, followed by developing a new implementation based on RV32IM.

Acknowledgment

First and foremost, I would like to offer my sincerest gratitude to my supervisor, Assoc Prof Douglas Maskell, for ensuring that this project was a valuable and enriching experience in my final year.

I would like to extend my gratitude and appreciation to Abhishek Jain for his constant guidance and support in the duration of this project. His constructive feedback, suggestions and friendly nature have always been motivational and inspired me to work towards my objectives.

I am grateful to Prashant Ravi for his professional guidance, continuous support, and teachings in the beginning, which were essential for me to understand important concepts and topics driving the project. I am also thankful to Swarna Jayaraman for her inputs and feedback on several version of this project.

Thanks to Mr. Jeremiah Chua in Hardware and Embedded Systems Lab (HESL) for his technical support and the facilities.

I would also like to extend my thanks to my examiner, Dr. Sharad Sinha, for taking out time to evaluate my project.

The list of acknowledgements will not remain complete without mentioning the encouragement and motivation given by my family and friends throughout the year.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The demand for high performance computing has led to a shift from single-core to multi-core processor architectures. Moore's law predicted that the number of transistors per square inch on integrated circuits will double every year, since transistors were invented. This seemingly endless size reduction and density increment of transistors has positively impacted the growth in the number of cores, while causing a negative impact of exponential surges in power consumed per square unit of transistors [8]. This has resulted in a fundamental shift in microprocessor design from frequency scaling to increased number of cores, leading to emergence of many-core architectures. Such designs have proven to enhance performance without the cost of greater power consumption [9].

Typically, parallel devices like Graphics Processing Unit (GPU) are used for processing large graphics data sets with extremely fast performance. However, the usage of GPUs and Field Programmable Gate Array (FPGA) for accelerating complex tasks has been becoming more and more common these days [10] [11]. Thus, multi-core heterogeneous co-processors are being used

widely with the promise of greater performance and power efficiency gains [12].

The goal of the project is to design a processor capable of delivering high computational performance. This would be achieved by connecting several of these processors in a many-core architecture network. Such a powerful hypothetical engine is a distant goal in the future, with an expectation of performance in the range of Peta OPS (10^{15} Operations per Second). The Instruction Set Architecture (ISA) based on which the processor will be designed is RISC-V, a new ISA initially designed with the purpose to support computer architecture research and education [13]. It has now set to become an industry wide implementation standard, with several processors already running this ISA (discussed in Chapter 3).

GPUs and CPUs have been selected to study performance in running a complex real world application in this project. GPUs offer high data-parallel processing throughputs and naturally map the convolution-rich nature of deep learning code to floating-point ALUs [14]. OpenCL is an open-source framework supporting GPU programming. It can be used to program devices ranging from CPUs, GPUs, FPGAs, and other devices from different vendors. Therefore, such a framework is opted for supporting programs to introduce cross-compatibility across various heterogeneous parallel computing platforms.

1.2 Project Scope

The project covers the following areas:

1. Understanding the OpenCL programming model to execute programs on heterogeneous devices

2. Familiarisation with convolutional neural networks and techniques used in solving learning problems

3. Comparison of performance of an application solving a real-world problem using OpenCL and profiling on various platforms

4. Understanding the need of an open-source ISA like RISC-V and studying a processor following this architecture

5. Design and implement a new processor with RISC-V foundation

1.3 Organization

The report consists of the following chapters: **Chapter 2** presents background software knowledge and hardware requirements needed for this project. **Chapter 3** 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 4** 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 5** 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 6** 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 7** and discuss work to be done in the future.

Chapter 2

Background

Basics of certain programming knowledge and possessing concepts along with the ability to use some softwares, tools and devices is a necessity to handle the tasks in this project. OpenCL has been used extensively throughout the length of this project, which requires thorough understanding of how parallel programs are executed on heterogeneous devices. The notion of host and device must be clear in order to conceptualise the execution of programs on the device.

A variety of co-processors like GPUs, FPGAs are investigated for performance metrics in this project. Therefore, familiarity with these computing devices is an added benefit. A high-level understanding of neural networks proved favourable while running profiling experiments on a CNN application. In order to evaluate performance, metrics such as Operations per second must be known.

The project also discusses the details of RISC-V ISA, requiring the foundational knowledge of computer organisation and architecture. This is succeeded by implementing a new processor based on the ISA using Verilog. In order to evaluate the functional correctness of the implementation, compre-

hension of Register Transfer Level (RTL) simulation is a requisite. Icarus Verilog is used to run the testbench on Linux while ModelSim SE is used on Windows platform.

GNU utilities are used in this project for compiling programs written in C and OpenCL. All work is done on a PC running Ubuntu 14.04 and Windows 10 in dual-boot mode.

Chapter 3

Literature Review

OpenCL implementation Portable Computing Language (POCL) was studied as a part of the related work. It is portable and performance portable by virtue of a kernel compiler that can utilize the data parallelism of the program on different hardware architectures. Compiler transformations which produce work-group functions with multiple work items which can be parallelized are made by integrating the OpenCL implementation with LLVM compiler infrastructure. This newly implemented feature allows the kernels to be mapped to the parallel resources available on the different computing platforms [15]. The experiments conducted by Pekka, J. et al on multiple hardware platforms showed that POCL could successfully port OpenCL applications efficiently.

One main part of this project constitutes studying an OpenCL application based on Convolutional Neural Networks which recognizes digits from MNIST database. L. Yann et al talks about how multilayer neural networks trained with backpropagation algorithm make an efficient Gradient-Based learning technique. This was incorporated in his model of CNN, known as the LeNet-5 architecture. The authors state two main modules to split the neural networks system into for recognizing individual patterns. The first

module is a feature extractor which changes the input patterns into low-dimensional vectors or short strings of symbols that allow easy comparison and are invariant to the distortions in the input patterns. The second module is a trainable classifier which is general-purpose [16]. L. Yann et al performs handwritten character recognition using several learning techniques on the benchmark dataset MNIST for handwritten digit recognition. Such an intricate convolutional neural network requires a large amount of computations, thereby requiring computational power for fast classifications.

Instruction Set Architecture (ISA) refers to the set of instructions which a processor understands to execute an instruction. It is an interface between a computer's software and hardware enabling high language code to execute on the machine. Different architectures use different bit sizes, number of operands for specific instruction types and endianness, which affect their performance.

Complex Instruction Set Computers (CISC) architecture, a predecessor of RISC, aimed to complete a task in as few lines of assembly code as possible. This, however, requires processor hardware capable of understanding complex instructions. RISC emphasizes on software and uses only simple instructions that can be executed in a single clock cycle. RISC architecture also reduces the load on hardware space, allowing more general purpose registers to be used [17]. Intel x86 has retained CISC architecture due to other technological advancements, but is still considered too complex for simple projects.

RISC-V was developed with the need to meet the case for a free and open source ISA. For over 30 years, no other ISA has been a successful stack ISA including CISC. In this regard, RISC-V is better than its predecessors by removing unnecessary features like shift and keeping necessary load/store byte

capabilities [18].

Several implementations of processors based on modified RISC-V architecture are present. Few of them are studied in this section. Berkeley Out of Order Machine (BOOM) is a synthesizable, superscalar and out-of-order RISC-V core written in Chisel, the hardware construction language [19]. Chisel stands for Constructing Hardware in a Scala Embedded Language. It supports a multi-core processor system by replacing the in-order Rocket core with the out-of-order BOOM core. It provides support for floating point, atomics and page-based virtual memory.

PULPino is a Parallel Ultra Low Power Processor with a small single-core RISC-V System on Chip (SoC). It is a small part of PULP which is a huge energy efficient many-core SoC with multiple software framework support. PULPino focuses on simplicity by removing caches, memory hierarchy and DMA (Direct Memory Access). As seen from Figure 3.1, the processor has a single cycle access being directly connected to the instruction and data RAM. It consists an Advanced Peripheral Bus to allow easy addition of peripherals to the core. The Boot ROM loads the program from SPI Flash.

Sources related to existing implemented accelerators were studied to understand what is lacking in the current scenario. Hugo van der Wijst explains in his Master thesis the increasing need for better performing accelerators [20]. He describes in detail his work on ρ-VEX processor to create an accelerator, including the design considerations, implementation, experiments and results. As part of the implementation, a PCI Express communication is set up between the FPGA and the accelerator.

The thesis continues to explain how ρ-VEX can be used as an accelerator in an OpenCL runtime environment by computing various performance

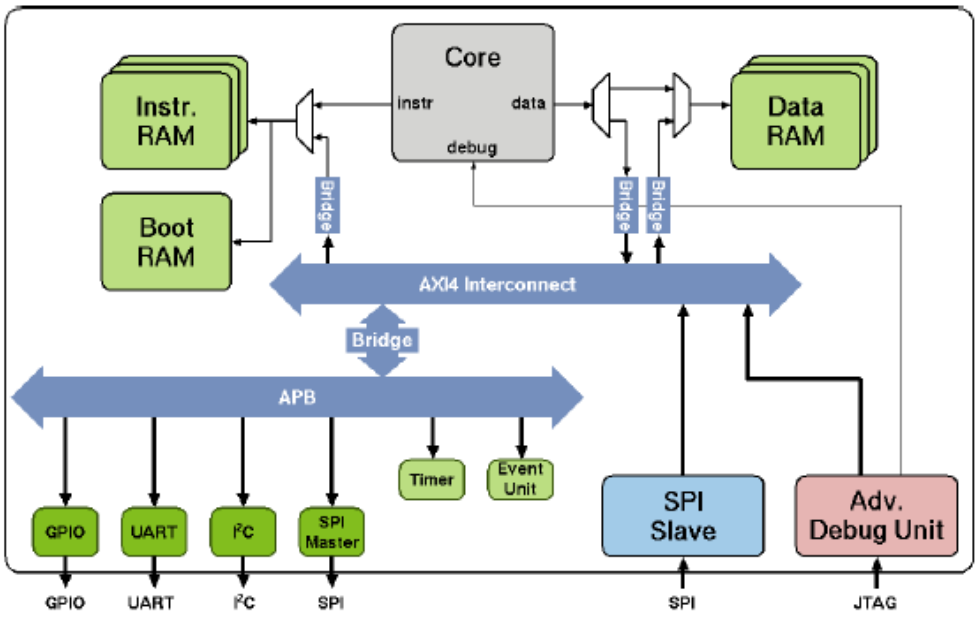


Figure 3.1: Architecture of PULPino single-core SoC [1]

metrics through experiments. Open64 compiler is found to perform better than LLVM compiler. Open source implementation POCL was chosen as the OpenCL runtime for the new accelerator, with a device-layer added to support ρ-VEX based accelerator. The execution time was improved by following techniques like reducing Instruction Count, Clock Cycles Per Instruction (CPI), memory stall cycles and clock cycle time. The parallel execution and data throughput was increased as well, to influence the execution time [20].

This project is different from the previous work, in the sense, it aims to develop a hypothetical engine with multiple CPUs, GPUs, and other co-processors acting as accelerators individually and combined in an overlay manner to provide a huge performance boost. Each core in the overlay would run RISC-V ISA and be capable of supporting OpenCL framework. This engine is different from the previous work in related areas, providing a new mega-accelerator platform to the computing world.

Chapter 4

The OpenCL Programming Model

OpenCL is a framework for writing application programs that run across heterogeneous platforms, comprising GPU, Central Processing Unit (CPU), Digital Signal Processor (DSP), FPGA, and other processors or hardware accelerators [21]. A framework suited for parallel programming, it has been standardized by the Khronos group which includes companies like AMD, Apple, Intel, NVIDIA, Xilinx, etc.[22].

4.1 Introduction

OpenCL is the first open programming standard for general-purpose computations on heterogeneous systems, allowing programmers to target their source code on multi-core CPUs, GPUs, and other devices. It specifies a programming language (centered on the C99 standard) for programming these devices and provides Application Programming Interfaces (APIs), to control the platform and execute programs on heterogeneous computing devices. OpenCL is vendor independent, i.e. not specialized for any device. Therefore, OpenCL is a powerful way to write parallel programs on a wide range

of devices.

The framework includes a language, API, libraries and a runtime system to support software development. The architecture of OpenCL can be described using a hierarchy of models [2]:

- Platform Model
- Memory Model
- Execution Model
- Programming Model

4.1.1 Platform Model

In this model, a host device is connected to one or more OpenCL devices, which is divided into one or more compute units (CUs), further divided into one or more processing elements (PEs). All computations on a device are executed on the PEs [23].

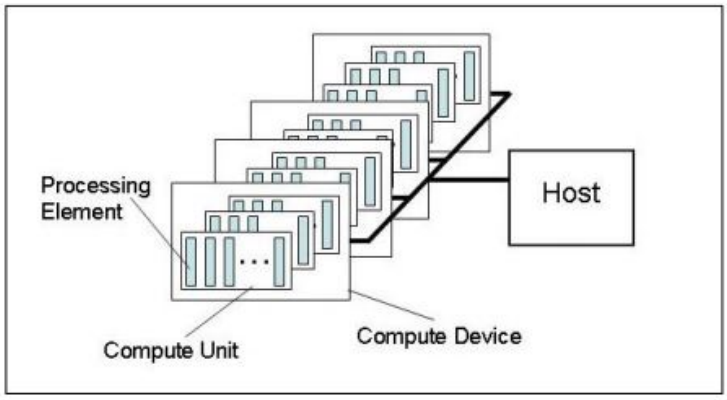


Figure 4.1: Platform Model in OpenCL

The host device holds the OpenCL application running, from where the commands are submitted to the device, to be executed on the PEs. The

PEs execute a single set of instructions as Single Instruction Multiple Data (SIMD) units in parallel.

4.1.2 Memory Model

Memory directly available to kernels executing on OpenCL devices is known as device memory. The device memory consists of the four distinct memory regions as follows:

- **Global Memory**: This memory region allows read/write access to each work item in all the work groups executing on any device inside a context. It is shared with all processing elements, and is also available to the host. The host utilises this memory to copy data to/from the device.
- **Constant Memory**: This memory region contains content that stays constant throughout the kernel execution. It is also shared between all processing elements, but it is read-only. It provides an efficient way to share data with all processing elements.
- **Local Memory**: A region of memory local to a work group. This region can be used to allocate variables common to all work items within a work group. Each CU has its own local memory, only shared with the processing elements within the compute unit, which cannot be accessed by other compute units.
- **Private Memory**: A region of memory private to a work item. Variables declared in one work item's private memory are not visible to another work item.

Global memory is the only memory which is persistent between kernel calls. Constant, local, and private memories are temporary spaces, which get reset after every kernel call.

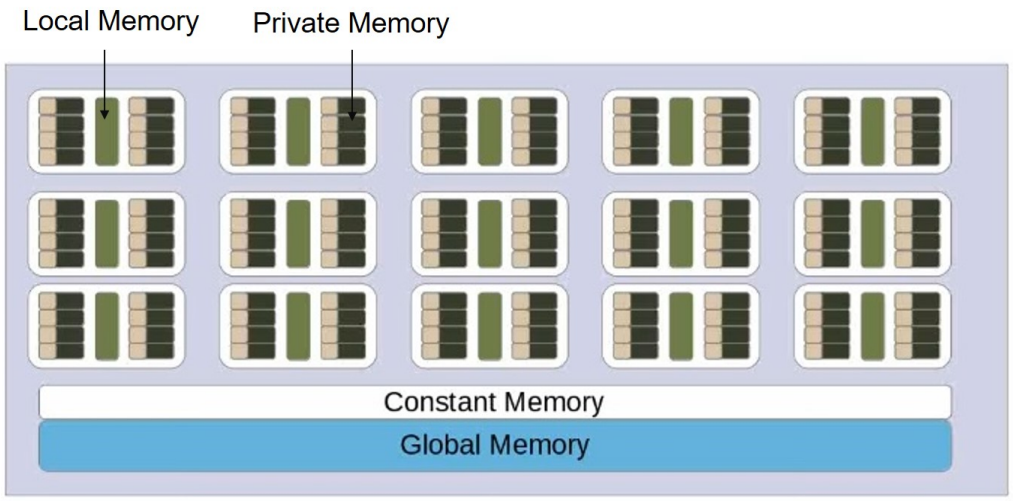


Figure 4.2: Memory Model in OpenCL [2]

4.1.3 Execution Model

The core of the OpenCL execution model is based on how kernels execute on the device. The execution of an OpenCL program happens in two parts: kernels that execute on the OpenCL devices, and a host that runs on the host device. The host is responsible for executing kernel functions on the device [23]. These are ordinary functions with special signatures. The kernel call comprises two parts: an ordinary argument list, and external execution parameters that control the parallelism.

The host coordinates the execution of the kernels on the OpenCL device.

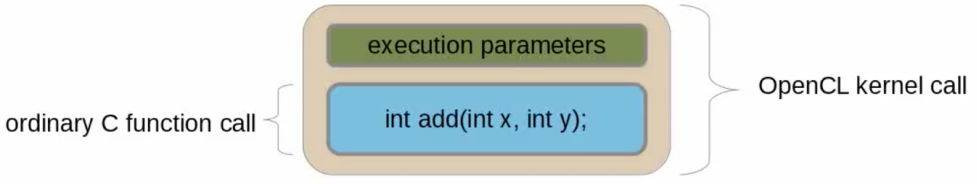


Figure 4.3: C and OpenCL Function Calls [2]

It does not get involved in the computations itself, only providing execution parameters to launch the kernel and arguments required by the kernel for its computations.

4.1.3.1 Execution Strategy

The kernel function gets called repeatedly with an identical argument list for all invocations, based on the execution parameters specified by the host while launching the kernel.

There exists an index space provided by an N-dimensional index space, called an NDRange. It can be one, two or three dimensional. Each function call can access its index, a way for it to be identified. A work item is a kernel invocation for a particular index, and the global id is a unique identity for every work item belonging to the index space. Every work item is provided with a global ID. The global work size is the number of work items in every dimension. The work dimension is the dimension of the index space.

The Processing Element (PE)s execute the instructions according to the Device Model. Thus, the work items need to be mapped to PEs for execution. Multiple work items can be assigned to each PE.

Every Compute Unit (CU) has a dedicated local memory which is shared by all PEs within. Local memory provides huge advantages in performance. Thus, work items must be mapped to the PEs in such a way that a group of them can access the local memory of a CU for effective execution. The partitioning of the global work size into smaller pieces leads to the formation of work groups. The work groups provide a more coarse-grained decomposition of the index space. Work groups are assigned a unique work group ID with the same dimension as the index space used for the work items. Work items are assigned a unique local ID within a work group so that a single

work item can be uniquely identified by its global ID or by a combination of its local ID and work group ID [23].

Work groups execute on CUs and share their local memory. All the work items in the work group share local memory and are mapped to PEs within a CU. These work items are capable of identifying which work group they are parts of, their work group ID, size of work groups, global IDs and global work size.

The work group size has a device-specific physical meaning. The maximum work group size is a device characteristic. This can be determined by querying the device. Also, it is an integer value; thus, n-dimensional work groups have to be handled in a special way. Work groups are launched by the host device.

The device dependent work group size is a scalar, but work groups can have multiple dimensions. For example, if the maximum work group size is 32, work groups could be launched in 3 dimensions like (8, 2, 2). The following must be true in order to launch the work groups successfully:

$$\textbf{Work group size} = (\textbf{W1}, \textbf{W2}, \textbf{W3}, ..., \textbf{Wk}) \tag{4.1}$$

$$(\textbf{W1} * \textbf{W2} * \textbf{W3} * ... * \textbf{Wk}) <= \textbf{Max Work group size} \tag{4.2}$$

The figure below shows an example of how the local size, number of work groups and global size vary for different dimensions of `NDRange`:

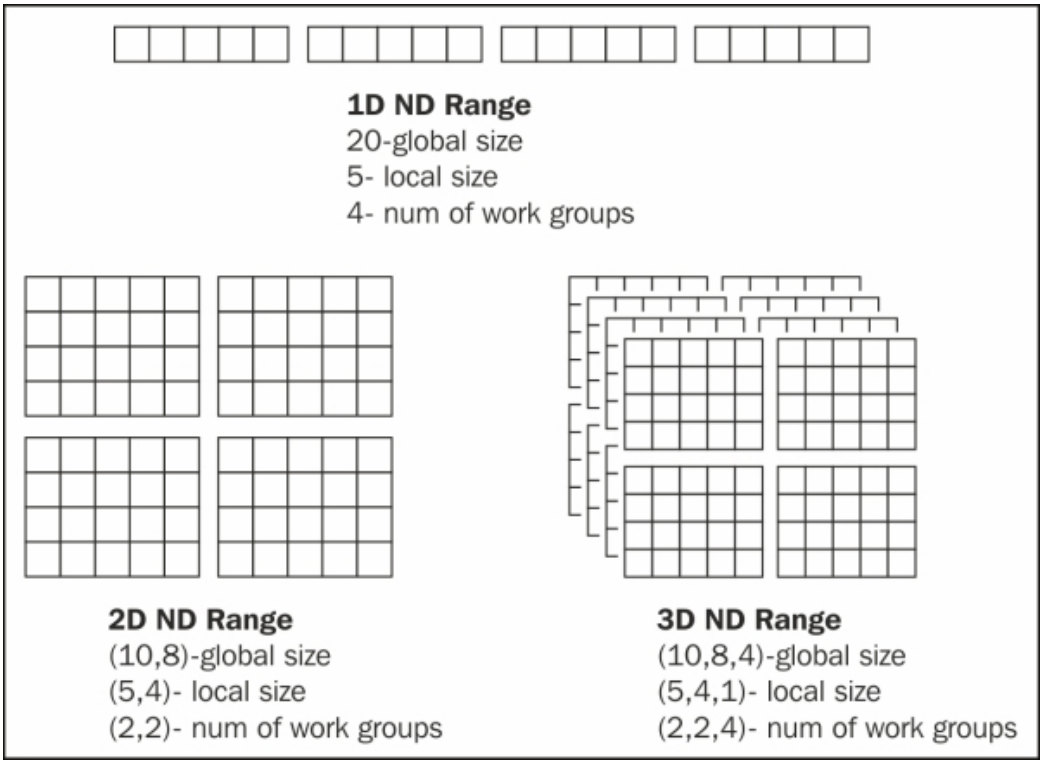


Figure 4.4: NDRange multi-dimensional index space example

4.1.3.2 Host API

The host must call certain APIs in order to execute the kernels, which take care of the following:

- Platform
  A platform is an implementation of OpenCL. These are the drivers for specific devices which support the execution of OpenCL kernels. Platforms enable supported devices to be exposed to the programmer. Platform is used to discover devices available.

- Context
  A context is created by the host for each platform on which the kernel executes. Thus, a context cannot have multiple platforms. A context

acts as a container for devices and memory. Most kernel operations are related to a context.

- Program
  A program is a collection of kernels. A program is loaded by the host device and kernels are extracted from the program to be invoked. The host could either directly compile the OpenCL C code or load a binary representation.

- Asynchronous Device Calls
  The host manages devices asynchronously. Host issues commands to the device, telling the device to perform some work. The host waits for the commands to complete, meaning that the device has completed the action. Commands can have dependencies on other commands. OpenCL commands are issued by `clEnqueue*` calls. A `cl_event` returned by `clEnqueue*` calls is used for dependencies.

4.1.4 Programming Model

The OpenCL execution model supports data parallel and task parallel programming models. In OpenCL, the primary model is data parallel.

4.1.4.1 Data Parallel

Data parallel programming model (equivalent to SIMD) defines computations in terms of a sequence of instructions applied to multiple elements of a dataset. In a strict data parallel model, a one-to-one mapping exists between a work item and an element in the dataset over which a kernel can be executed in parallel. However, a strict one-to-one mapping is not required in OpenCL.

A hierarchical data programming model exists in OpenCL, where the hierarchical subdivision can be achieved either explicitly or implicitly. In the explicit model, the total number of work items to execute in parallel and how work items are divided among work groups must be defined by the programmer. On the other hand, the implicit model only requires the total number of work items to execute in parallel to be defined, while the division into work groups is taken care by the OpenCL implementation.

4.1.4.2 Task Parallel

Task parallel programming model in OpenCL defines a model in which a single instance of a kernel is executed independent of any index space. It is equivalent to executing a kernel on a compute unit with a work group containing a single work item. It is the simultaneous execution of many different tasks across the same or different memory objects.

4.2 Experiments

A few simple experiments were conducted on different devices – an AMD GPU and an NVIDIA GPU, to see how performance varied with different execution parameters. A small kernel which performs 2-dimensional matrix multiplication was written in OpenCL and was launched with different global sizes and work group sizes on the specific platform to see how execution timings varied.

The host device selects the platform and the device, creates the context and command queue, initializes the matrices, creates memory buffers in the device, and transfers the data to the device memory. The two input matrices and one output matrix are passed to the kernel function as arguments, on which the matrix multiplication was performed in parallel. The execution is controlled by the `clEnqueueNDRangeKernel` parameters, which define the

work group size by specifying the local size and global size.

$$\textbf{Work group size} = \textbf{Local size} \tag{4.3}$$

The number of work groups can be found out as follows:

$$Number\,of\,work\,groups = \frac{Global\,Size}{Local\,Size}$$

The work group size must be perfectly divisible by the specified global work size, failing which the kernel would not be launched and an error would be thrown. A work group gets mapped to a compute unit for execution.

The global size, or index space depends on the total number of computations required to be performed. In matrix multiplication, the global size is two dimensional, corresponding to the dimensions of the matrices being multiplied. For example, if two 4 x 4 matrices are to be multiplied, the global size would be two-dimensional (4, 4).

The resultant matrix was stored in a buffer and transferred back to the host asynchronously. The execution time was calculated and the result was compared for correctness by multiplying matrices sequentially on the host.

In the experiments conducted, the sizes of the matrices *(or global size)* being multiplied was increased from 128 to 2048, and for each global size, the work group size was changed from 2 to 128. The objective of these experiments was to notice the effect of changing the mapping of work groups onto the compute units of the device on the kernel execution timings.

4.2.1 Results

The following tables show the average timing results on AMD Radeon HD 8730M and NVIDIA GeForce 405.

Table 4.1: Execution time (in µs) of matrix multiplication on AMD Radeon HD 8730M GPU

2-D Matrix Multiplication: Timings on AMD GPU					
Global / **Local**	**128**	**256**	**512**	**1024**	**2048**
Not Specified	6121.25	48282.5	401073.58	3270198.86	16901530.39
2	8400.23	67250.57	530250.24	1082663.32	5365565.08
4	2181.05	16945.09	142875.45	298480.89	1703353.98
8	3359.17	25594.24	206148.51	265741.50	1362683.18
16	6123.93	48253.26	385895.55	291104.51	1243956.77
32	*Error*	*Error*	*Error*	*Error*	*Error*
64	*Error*	*Error*	*Error*	*Error*	*Error*
128	*Error*	*Error*	*Error*	*Error*	*Error*

Table 4.2: Execution time (in µs) of matrix multiplication on NVIDIA GeForce 405 GPU

2-D Matrix Multiplication: Timings on NVIDIA GPU					
Global / **Local**	**128**	**256**	**512**	**1024**	**2048**
Not Specified	6138.05	48312.10	401356.14	5773783.80	*Error*
2	10721.54	58040.95	621084.80	4527345.84	*Error*
4	3488.40	18654.32	232133.67	1776133.69	*Error*
8	2969.30	17499.01	229669.94	1729808.40	*Error*
16	8382.89	34673.16	486371.19	3369363.59	*Error*
32	*Error*	*Error*	*Error*	*Error*	*Error*
64	*Error*	*Error*	*Error*	*Error*	*Error*
128	*Error*	*Error*	*Error*	*Error*	*Error*

4.2.2 Discussion

Table 4.1 and Table 4.2 show the execution time (in µs) taken by the two-dimensional matrix multiplication kernel in OpenCL on AMD and NVIDIA GPUs, for different local sizes. The matrix sizes were increased from 128 x 128 to 2048 x 2048 and the execution times were measured and noted.

The results can be analysed better by comparing the performance of both devices in Giga Operations per Second (GOPS) for different global sizes, calculated in Table 4.3. It has been computed by dividing the number of operations by the best performing OpenCL execution timings among the different local sizes, specified in Tables 4.1 and 4.2. The performance in GOPS for AMD Radeon 8730M and NVIDIA GeForce 405 are for local sizes 4 and 8, respectively.

Table 4.3: Performance in GOPS for AMD Radeon 8730M and NVIDIA GeForce 405 for different global sizes

2-D Matrix Multiplication: Performance in GOPS		
Device / **Size**	**AMD**	**NVIDIA**
128	5.78	4.25
256	5.95	5.76
512	5.64	3.51
1024	21.59	3.73
2048	30.26	*Not Applicable*

As seen in Figure 4.5, the number of operations performed during the execution is equal to `6 * GLOBAL_SIZE`^{3} `+ 2 * GLOBAL_SIZE`^{2} where the `GLOBAL_SIZE` varies from 128 to 4096 as exponents of 2. This program has a complexity of O(n^3) which largely increases the number of operations.

```
for(i = 0; i < GLOBAL_SIZE; i++)
{
    for(j = 0; j < GLOBAL_SIZE; j++)
    {
        sum = 0;
        for(k = 0; k < GLOBAL_SIZE; k++)
            sum += data1[i*GLOBAL_SIZE + k] * data2[k*GLOBAL_SIZE  + j];
        results[i*GLOBAL_SIZE + j] = sum;
    }
}
```

Figure 4.5: Code for matrix multiplication in 2D

As observed in the results, the AMD Radeon 8730M overpowers the NVIDIA GeForce 405 if their best execution times are compared for different global sizes. Both the devices follow a similar trend for maximum performance (or least execution time). For both the devices, there exists a trend of having a least execution time at a particular work group size, beyond which the increasing the local work size leads to higher execution times (lower performance). In the case of AMD Radeon 8730M, the best performance is achieved when the work group size is 4, while for NVIDIA GeForce 405 it is 8. The best performing work group size values are device specific, as seen from the results.

Upon further increasing the work group size, it was noticed that the execution of OpenCL kernel failed due to error in launching it. This work group size was 32 for both AMD Radeon 8730M and NVIDIA GeForce 405. This conclusion follows the discussion in Section 4.1.3.1, which states that maximum work group size is a characteristic of the device. Thus, we can conclude that the maximum work group size for both GPUs is 16.

Chapter 5

Convolutional Neural Networks

Convolutional Neutral Network (CNN) is a classification of multi-layered neural networks [24]. CNNs have excellent performance in areas such as image recognition and classification. CNNs are at the core of state-of-the-art approaches to a variety of computer vision tasks, including image classification [25] and object detection [26]. LeNet-5 is one such model designed for handwritten and machine-printed character and digit recognition.

5.1 Introduction

5.1.1 Neural Networks

An Artificial Neural Network, simply put as Neural Network, is a computational model motivated by the manner in which neural networks in the human brain and nervous system process information [27]. The field of Neural Networks was primarily inspired by the objective of modeling biological neural systems, but has since diverged and become a part of engineering and achieving great results in machine learning [3].

The fundamental building block of a neural network is the neuron (or node).

The following figures shows a biological neuron and its computational model are related:

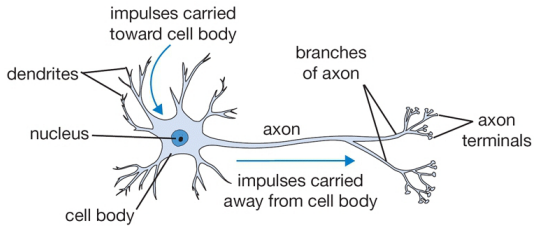


Figure 5.1: Representation of a biological neuron[3]

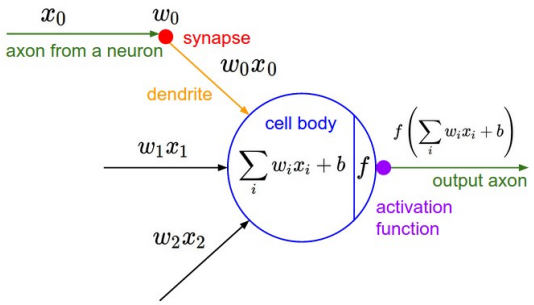


Figure 5.2: Computational model of a neuron [3]

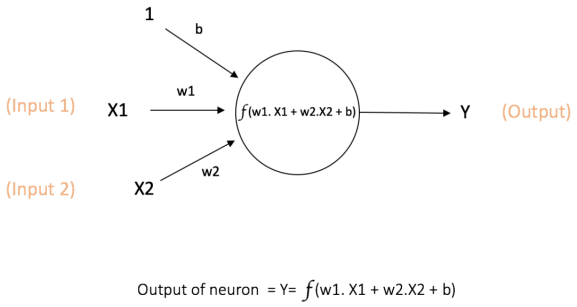


Figure 5.3: Computation in a Neuron[4]

In this model, a node gets inputs from other node or an external source, and calculates an output. Inputs have associated weights assigned based on

their relevance to other input nodes. The node applies a function on the weighted sum of inputs, as shown in Figure 5.3.

The neuron receives inputs x1 and x2 with weights w1 and w2. Each neuron also has another input 1, with a weight **bias** associated with it. The bias value provides each node a trainable constant value.

The function applied to the sum of weighted inputs, known as Activation Function, is a non-linear function introduced to force neurons to learn non-linear representations, as most of real world data is non-linear. Following are some common activation functions used:

- Sigmoid: Takes a real-valued input and compresses it within the range 0 to 1

$$\sigma(\mathbf{x}) = \mathbf{1}/(\mathbf{1} + \mathbf{exp}(-\mathbf{x})) \tag{5.1}$$

- tanh: Takes a real-valued input and squeezes it in the range -1 to 1

$$\mathbf{tanh}(\mathbf{x}) = \mathbf{2}\sigma(\mathbf{2x}) - \mathbf{1} \tag{5.2}$$

- ReLU: Takes a real-valued input and thresholds it at zero (replaces negative values with zero) f(x) = max(0, x)

$$\mathbf{f}(\mathbf{x}) = \mathbf{max}(\mathbf{0}, \mathbf{x}) \tag{5.3}$$

5.1.1.1 Feedforward Neural Network

One of the first and simplest neural networks devised was the feedforward neural network. There are multiple neurons placed in layers, with edges connecting them. Each of these edges are tagged a weight. Figure 5.4 shows an example of a feedforward neural network.

Information only moves in the forward direction in a feedforward neural

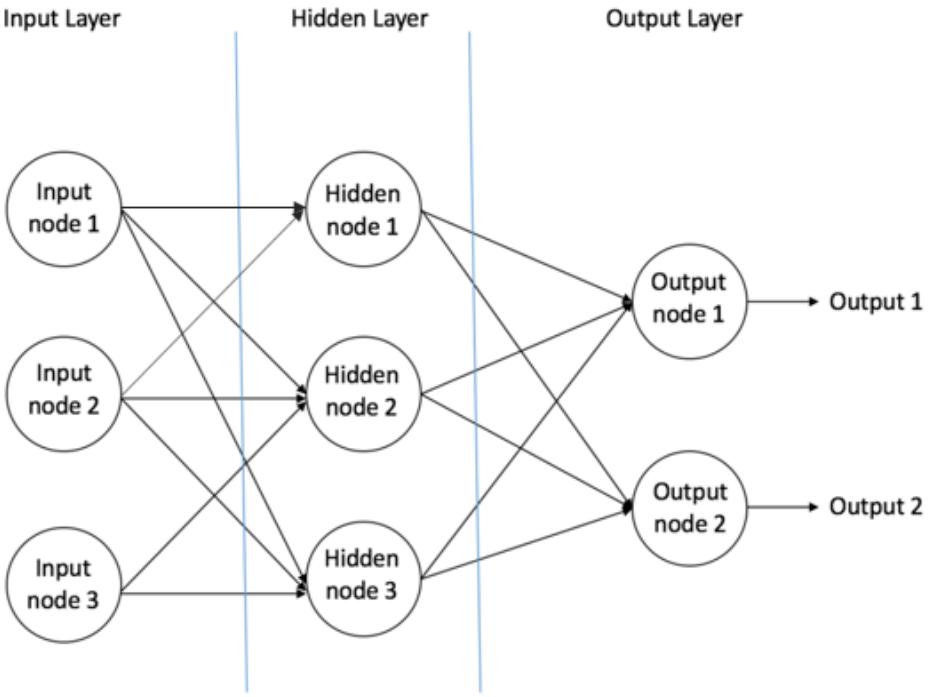


Figure 5.4: Feedforward Neural Network [4]

network. Input nodes receive signals from the outside world and pass on the information to the hidden nodes, which perform computations and forward the information to output nodes. The output nodes carry out some calculations and send the result to the outside world. There can only be one input and output layer of nodes, but multiple hidden layers of nodes in a feedforward neural network.

They are of two types, **Single Layer Perceptron** and **Multi Layer Perceptron** (MLP). The former does not consist of any hidden nodes in the middle, while the the latter has at least one hidden layer. MLPs are very useful in several practical learning applications.

5.1.1.2 Training

For any neural network to accurately learn relationships between inputs and outputs and generate correct predictions, training is required. For an MLP to learn, back propagation algorithm is used.

Back-propagation is one of the many ways to train a MLP. It is a supervised learning algorithm which learns from labeled training datasets. Back-propagation algorithm trains the MLP by correcting its output when a mistake is made. Its objective is to assign correct weights to the connections between nodes in different layers, so as to generate accurate outputs.

Initially, all edge weights are assigned random values. For every input in the training dataset, the MLP is activated and its output is noticed and checked against the desired output from the labeled data. The error is then "propagated" back to the previous layer. This error is computed and the weights are updated. This process repeats until the output error is less than a pre-specified threshold.

Once the algorithm finishes, a "learned" neural network is created which can work with new inputs. It will have learned from millions of example inputs (labeled data) and from mistakes made in predicting outputs (error propagation).

5.1.1.3 Visualisation

Andy Harley developed a two-dimensional and three-dimensional representation of an MLP trained to recognise the handwritten digits from the Mixed National Institute of Standards and Technology (MNIST) database [5]. The following figure shows how a MLP is visualised in 3D.

This MLP network has 784 nodes on the bottom layer (corresponding to

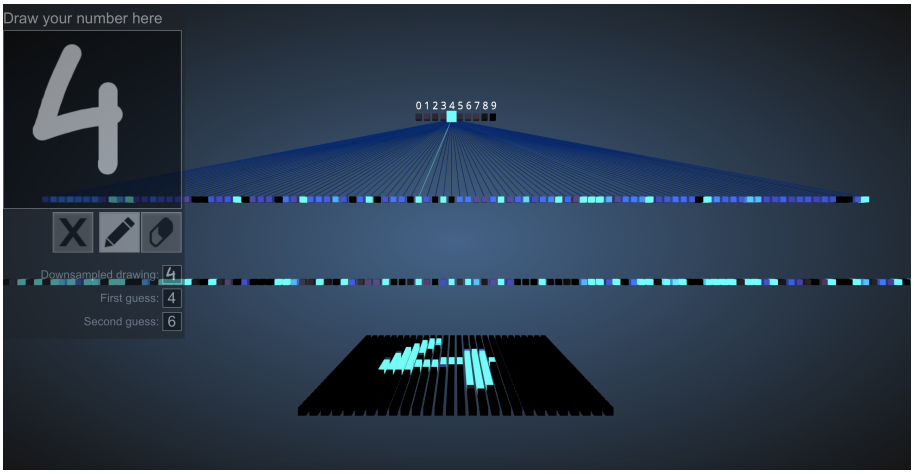


Figure 5.5: 3D Visualisation of a fully connected MLP [5]

pixels), 300 nodes in the first hidden layer, 100 nodes in the second hidden layer, and 10 nodes in the output layer (corresponding to the 10 digits)[5].

A brighter colour represents a node which has a higher output value than others. In the input layer, the bright nodes are the ones receiving higher numerical pixel values as input. In the output layer, the only bright node corresponds to the digit 4, indicating that the MLP correctly classified the input digit as 4.

5.1.2 Convolutional Neural Networks

CNNs are quite similar to regular neural networks – they are composed of neurons possessing learnable weights and biases; neurons get inputs, perform dot products and follow it up with non-linearity (optionally). The network expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other. There is a loss function on the last, fully-connected layer which generates the final output. The difference exists for the type of input CNN architectures explicitly assume, which are images, allowing certain properties to be encoded into the architecture. These then

make the forward function more efficient to implement and hugely decrease the number of parameters in the network [28].

5.1.2.1 Architecture

CNNs take benefit of the fact that inputs accepted are images, and constrain the architecture in a more sensible way. Unlike a regular Neural Network, the layers of a CNN consist of neurons arranged in 3 dimensions. Neurons in a layer are only be connected to a small region of the layer before it, instead of all the neurons in a fully-connected way. Every layer transforms a 3D input to 3D output using some differentiable function. Eventually, the final output layer is one-dimensional, as the full image gets reduced into a single vector of scores by the end of all processing.

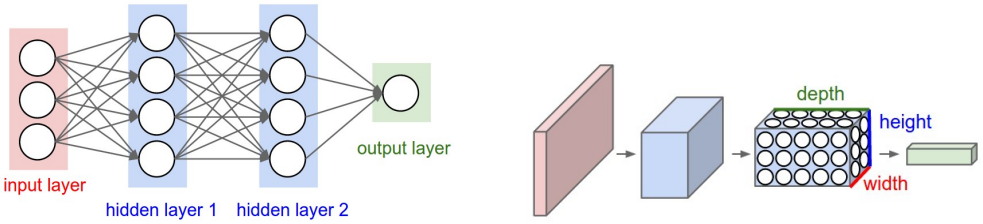


Figure 5.6: Visualisation of a Feedforward neural network *(left)* and a CNN *(right)*

For accuracy in predictions made by neural networks, a lot depends on how the layers are defined by the architecture. In this case, the LeNet-5 architecture is studied.

LeNet-5 Architecture

This was one of the very first convolutional neural network, conceptualised in 1994, which propelled the field of Deep Learning. Yann LeCun, a French scientist was the person behind this pioneering work, achieved as a result of several successful iterations since 1988 [29]. Several new architectures have

been proposed in the recent years which have improved features over LeNet, but they are all based upon the foundation of Lenet.

The network receives an image as an input and assigns probabilities to various output possibilities. The one with the highest probability is the classified character or digit. Following are the steps involved in the classification process:

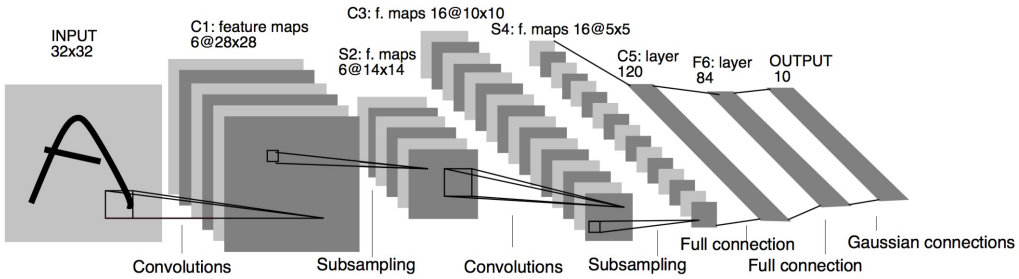


Figure 5.7: Lenet-5 architecture

The operations forming the building blocks of every convolutional neural network are as follows:

- Convolution
- Pooling or Down-Sampling
- Activation Function for Non Linearity (ReLu)
- Classification (Fully Connected Layer)

Convolution Layer

Channel is a traditional term used to refer to the coloured component of an image. A coloured image has three channels – red, green and blue, and could be imagined as three 2D-matrices stacked over each other (one for each color).

On the other hand, grayscale images have just one channel. The value of each pixel in the matrix will range from 0 to 255 – zero indicating black and 255 indicating white. Convolution layer extracts key features from the input image using convolution operations. It preserves the spatial relationship between pixels by learning image features using small squares of input data.

The image is convolved with multiple filters, resulting in the formation of different feature maps. Essentially, each filer is a 2D matrix (for a grayscale image), based on the value of whose individual elements, feature maps vary.

The orange matrix is slided over the image matrix, 1 pixel at a time (also called 'stride'), and for each position, an element wise multiplication is computed (between the two matrices). The multiplication outputs are added to get the final integer which makes one element of the output matrix at the corresponding position. The 3×3 filter only observes one part of the input image in each stride.

For example, if there is a 5x5 image convolved with a 3x3 filter, a feature map will be generated based on the convolution result. For a different filter matrix, another feature map would have been generated.

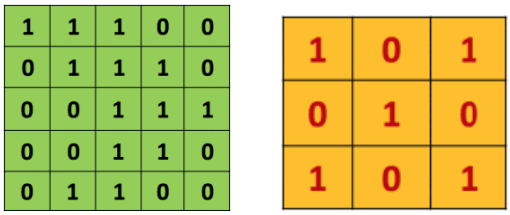


Figure 5.8: Convolution operation in action - a 5x5 Image and a 3x3 filter

In the practical world, a CNN learns the values or weights of these filters itself during the training process. The greater the number of filters, the more image features are extracted and the better the network becomes at

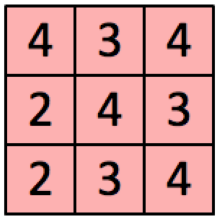


Figure 5.9: Result of convoluting a 5x5 Image against a 3x3 filter

recognizing patterns in images, not seen previously.

Pooling Layer

Pooling or down-sampling reduces the dimensions of each feature map but retains the most important details. It could be done using different mathematical operations: addition, average or maximum.

In case of Max Pooling, a spatial neighborhood is defined (for example, a 2×2 window) and the largest element is taken from the rectified feature map within that window. The dimensionality of the 2-dimensional window (2x2)

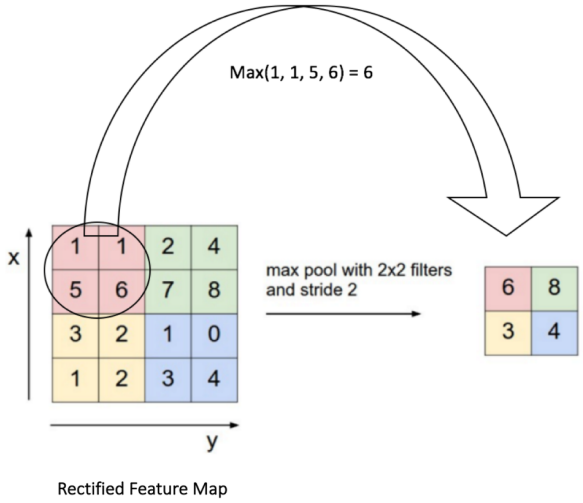


Figure 5.10: Demonstration of Max Pooling

is also called as stride. The window is slided by the stride value, and the maximum value from that region is taken in the reduced feature map. This is how the dimensionality of the feature map decreases. Figure 5.9 demonstrates how pooling reduces the size of the feature map for a sample image.

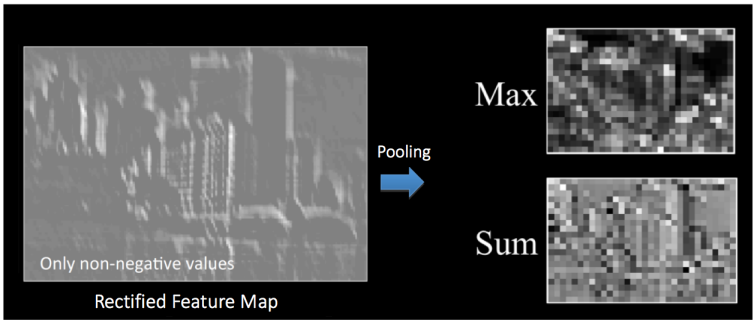


Figure 5.11: Effect of Max Pooling layer on a sample image

Non-Linearity Layer (ReLU)

This is an additional layer utilised to introduce non-linearity in the image feature maps. ReLU stands for Rectified Linear Unit, which is a non-linear function.

$$\textbf{Output} = \textbf{max}(\textbf{0}, \textbf{input}) \tag{5.4}$$

This operation is done on every single element of the image or its feature map(s), replacing every negative value with a zero. The purpose of ReLU is to introduce non-linearity in our CNN, since most of the real-world data the network would be classifying would be non-linear. Other non-linear functions such as **tanh** or **sigmoid** could also be used as an alternative. Figure 5.11 shows the effect of applying non-linearity on an image.

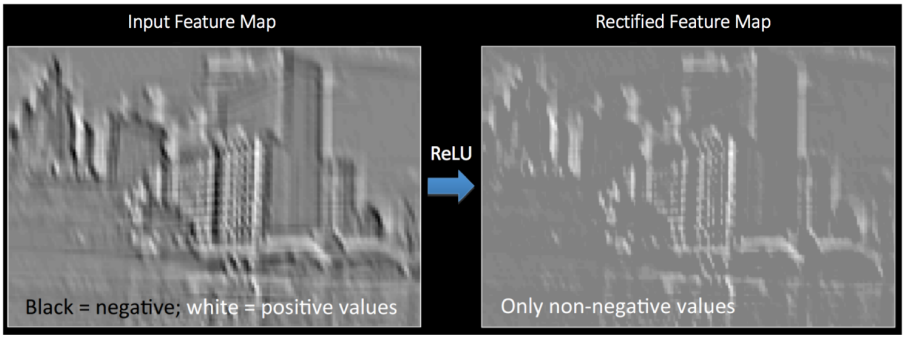


Figure 5.12: Effect of ReLU layer on a sample image

Fully Connected Layer

The fully connected layer is a classic Multi Layer Perceptron (MLP) using a softmax activation function in the output layer. Each node from the previous layer is connected to each node in the the next layer, hence the term "fully connected".

The output from the convolutional and pooling layers represent high-level features of the input image. The purpose of this fully connected layer is to use these features for classifying the input image into various categories, depending on the training data.

The sum of output probabilities from the Fully Connected Layer is 1, ensured by using **softmax** as the activation function in the output layer. The **softmax** function takes a vector of arbitrary real-valued scores and squashes it to a vector of values between zero and one that sum to one.

Training

The training of a CNN follows the same methods as those for a regular neural network. All filters and weights are assigned with random values. The network then accepts an image as its input and performs the convolution,

pooling, and non-linearity operations, calculating its output probability in the end. This prediction may not be accurate, since weights were assigned randomly.

The total error is computed at the output layer. Afterwards, back propagation is used to calculate the gradients of the error with respect to all weights in the network and gradient descent is used to update the values of all weights to minimise the output error. The weights are updated in proportion to their contribution to the total error. These steps are repeated across all images present in the dataset to train the CNN completely, which produces a "learned" network.

5.1.2.2 Visualisation

Along with visualising MLP, Andy Harley developed a visualisation tool for a CNN trained to recognise handwritten digits from the MNIST database [5].

Figure 5.13: 2D Visualisation of a CNN recognising MNIST handwritten digits [5]

The above figure shows how a network can be visualised in 2D. This CNN has 1024 nodes on the bottom layer (corresponding to pixels), six 5x5 (stride 1) convolutional filters in the first hidden layer, followed by sixteen 5x5 (stride 1) convolutional filters in the second hidden layer, then three fully-connected layers, with 120 nodes in the first, 100 nodes in the second, and 10 nodes in the third. The convolutional layers are each followed by a downsampling layer that does 2x2 max pooling (with stride 2)[5].

5.2 MNIST Database

The MNIST database is an enormous database of handwritten digits which is frequently used as a training dataset for different image processing procedures [30]. It is widely used for training and testing in the field of machine learning.

The MNIST database consists of 60,000 training and testing images [31]. Samples from National Institute of Standards and Technology's (NIST) original training and testing dataset were remixed to generate the MNIST database. The digits have been size-normalized and centered in a fixed-size image of 20x20 while preserving their aspect ratio [32]. The resulting images contain grey levels because of anti-aliasing methods used by the normalization algorithm. The images were centred in a 28x28 image by calculating the centre of mass of the pixels and rendering the image to place this point at the centre of the 28x28 field.

The database is a good starting point for individuals who desire to learn machine learning techniques and pattern recognition methods on real-world data while spending marginal efforts on preprocessing and formatting. When one learns how to program, there's a tradition to print "Hello World" in his first program. Similar to how programming has Hello World, machine learning has MNIST.

A number of scientific papers have been written to attempt achieving the lowest error rate in predicting the digits – one paper achieves a rate as low as 0.23 percent using a hierarchical system of convolutional neural networks.

5.3 Experiments

An existing real word application was obtained and executed to further gauge the performance capabilities of OpenCL. An MNIST handwritten digits prediction application written by Gopala Krishna Hegde based on LeNet-5 convolutional neural networks was cloned from GitHub for this purpose [33]. The implementation has been done in C++ as well as OpenCL.

5.3.1 Analysis of Application Code

After understanding how the LeNet-5 architecture works, the application was studied and the sizes and computations of the various layers within the CNN were determined. In each layer, addition, subtraction, multiplication, division and maximum operations were counted to calculate the total number of computations.

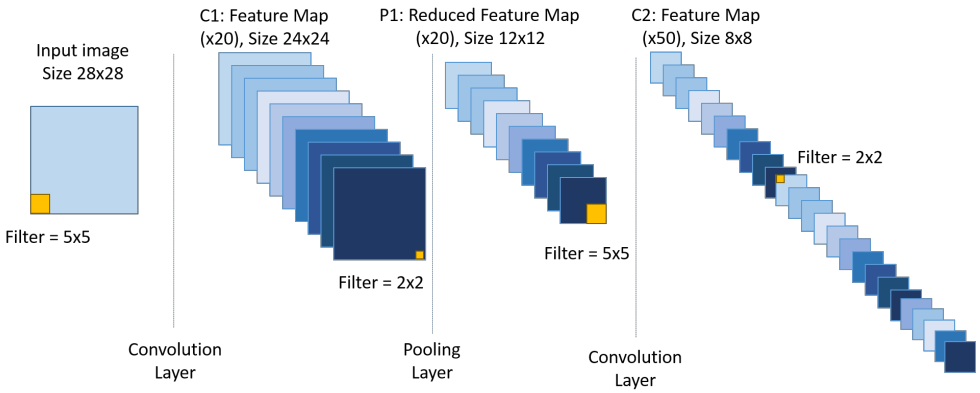


Figure 5.14: Representation of the classification process - *Part 1*

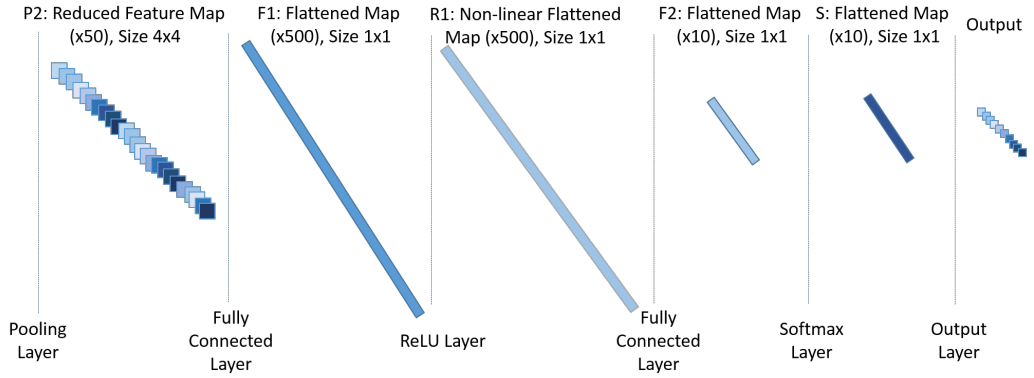


Figure 5.15: Representation of the classification process - *Part 2*

The layers in the network can be represented as shown in Figures 5.14 and 5.14. Table 5.1 indicates various layers from the LeNet-5 architecture used in this application. Each layer has an input size, output size and number of computations associated with it.

Table 5.1: Characterisitics of LeNet-5 layers in application using CNN to recognise hand-written digits

LeNet-5 CNN Application: Characterisitics layers					
Layer	**Inputs**	**Outputs**	**Input Size**	**Output Size**	**Computations**
Convolution 1	1	20	28x28	24x24	1324800
Pooling 1	20	20	24x24	12x12	80640
Convolution 2	20	50	12x12	8x8	6812800
Pooling 2	50	50	8x8	4x4	22400
Inner Product 1	800	500	1x1	1x1	1600500
ReLu	500	500	1x1	1x1	500
Inner Product 2	500	10	1x1	1x1	20010
Softmax	10	10	1	1	22

From Table 5.1, it can be calculated that the total number of computations performed by all the layers is `9,861,672`. This is the number of computations required for a single image to be classified and recognised as a digit.

5.3.2 Results

The MNIST handwritten digit recognition application was executed in parallel on different devices - two CPUs and one GPU, and average timings were calculated for different parts in the execution of kernels. Also, the sequential variant of the code written in C++ was executed on the two CPUs, and timings were noted.

The following results were obtained on Intel(R) Core(TM) i3-2350M, Intel(R) Xeon(R) E5-1650 CPU, and NVIDIA Quadro 600 GPU.

Table 5.2: Execution time (in µs) for different operations in MNIST Application on different devices

LeNet-5 CNN Application: OpenCL Execution Times			
Device	**i3-2350M**	**E5-1650**	**Quadro 600**
Initialize Application	1.6	1.3	1.5
Allocate Host Memory	17.2	16.9	16.5
Initialize Device	81058.2	78408.9	36985.2
Build Kernel	152719.7	130634.2	16914.6
Allocate Device Memory	1312.1	1025.2	1364.7
OpenCL Kernel Execution	1223.8	397.7	3903.5
Sequential Execution	85523	33892	*Not Applicable*

There is a total of eight kernels executing on the device, launched by the host one after the other. Each kernel corresponds to a layer, as show in Figures 5.14 and 5.15. The convolutional and pooling kernels are launched with a 3D global size corresponding to the number of outputs and their sizes, while the ReLu, fully connected and Softmax layers are launched with a 1D global size corresponding to the number of outputs of the layer (mentioned in Table 5.1.

5.3.3 Performance Analysis

The results above in Table 5.2 show the time taken on three different processors for initializing the application, allocating memory in the host, initializing the device on which the kernels will execute, building the kernels, allocating memory on the device, and the execution of all kernels. The sequential execution timing indicates the time taken by the CPU to run the C++ code for digit recognition.

Performance in GOPS

Based on the total number of computations found out in Section 5.3.1 and execution times in Table 5.2, the performance of the devices was calculated in Giga Operations Per Second (GOPS). The following formula was used to derive this value:

$$Performance\,in\,GOPS = \frac{Number\,of\,computations}{Execution\,time\,in\,ns}$$

Discussion

Comparing the timing values for the CPUs from Tables and 5.2, the extent of parallelism introduced by OpenCL can be evidently noticed. There is a speedup of 69 times for Intel(R) Core(TM) i3-2350M and 85 times for In-

Table 5.3: Execution time (in µs) for different operations in MNIST Application on different devices

LeNet-5 CNN Application: Performance in GOPS			
Device	**i3-2350M**	**E5-1650**	**Quadro 600**
Sequential Execution	0.115	0.291	*Not Applicable*
OpenCL Kernel Execution	8.06	24.79	2.53

tel(R) Xeon(R) E5-1650.

As evident from Table 5.3, the performance of Intel(R) Xeon(R) E5-1650 CPU running OpenCL kernels is the best at 24.79 GOPS, followed by Intel(R) Core(TM) i3-2350M at 8.06 GOPS and then by NVIDIA Quadro 600 GPU at 2.53 GOPS. This can be attributed to the device characteristics, which were queried using provided APIs by OpenCL seen as follows:

```
Number of platforms:    2

Platform:               0

        Platform Vendor:        NVIDIA Corporation
        Number of devices:      1

        Device: 0
                Name:                           Quadro 600
                Vendor:                         NVIDIA Corporation
                Available:                      Yes
                Compute Units:                  2
                Clock Frequency:                1280 mHz
                Global Memory:                  1023 mb
                Max Allocateable Memory:        256 mb
                Local Memory:                   49152 kb

Platform:               1

        Platform Vendor:        Intel(R) Corporation
        Number of devices:      1

        Device: 0
                Name:                                   Intel(R) Xeon(R) CPU E5-1650 0 @ 3.20GHz
                Vendor:                         Intel(R) Corporation
                Available:                      Yes
                Compute Units:                  12
                Clock Frequency:                3200 mHz
                Global Memory:                  15972 mb
                Max Allocateable Memory:        3993 mb
                Local Memory:                   32768 kb
```

Figure 5.16: Device characterisitcs of Intel Xeon E5-1650 CPU and NVIDIA Quadro 600 CPU

Because of the higher number of compute units (12) for the CPU, OpenCL brings about parrallelisation of the code to a much greater extent, as compared to the GPU (2). Furthermore, the clock frequency at which the CPU can execute at (3.2 GHz) is higher than that of the GPU (1.28 GHz). Thus, this CPU shows better performance relative to the GPU in executing OpenCL kernels.

Chapter 6

RISC-V

6.1 Introduction

The Central Processing Unit (CPU) is one of the most significant parts of any computing device, as it is the brain of the computer [34]. It performs arithmetic and logical operations and moves data from one place to another. This is done by the processor according to ISA it is based on. An ISA is a well-defined interface linking computer software to the hardware its running on, enabling the independent development of the two computing realms. It is strictly an interface specification, not an implementation.

In today's age, technology has been able to evolve much more because of the existence of open standards and open source software – Linux as an open source operating system for example. This builds the case for an ISA which is not proprietary and is powerful enough to drive custom processing chips.

Historically, ISAs have been proprietary for business reasons, but there is no good technical reason for the absence or deficiency of open source ISAs. Having a free and open source ISA specification would hugely benefit the industry, just like how open source software has added great value. It would

directly lead to a rise in innovation via free-market competition, with several designers coming with various implementations, which comprise both the open source and proprietary ISA versions. Moreover, an open ISA would lead to processors becoming affordable for a greater number of devices, helping in expanding the Internet of Things (IoTs) [18].

6.1.1 About

RISC-V is an ISA originally designed to support computer architecture research and education, now reaching the stage to become a standard open architecture for industry implementations, under the governance of the RISC-V foundation. It was developed in the Computer Science Division of the EECS department at University of California, Berkeley [13]. Its development was inspired by ARM's IP restrictions together with the lack of 64-bit addresses.

RISC-V has learned from various mistakes in different ISAs, making its design superior to other ISAs incorporating a better mix of capabilities [18]. It provides a small core instruction set that compilers and operating systems can depend on, as well as optional standard extensions. It has a compact instruction set encoding – smaller code is desirable as it reduces the memory requirement. It offers single, double and quadruple precision floating point capabilities, in addition to being 32, 64 and 128-bit addressable.

RISC-V was defined with a goal to be completely open to academic researchers and industries. Designed with small, fast and low-power real-word implementations in mind, it does not over-architect for a particular style of microarchitecture. It is a real ISA suited for direct native hardware implementation. It provides support for the revised 2008 IEEE-754 floating-point standard [35]. The widespread usage of RISC-V would allow developers to target an open hardware, while making commercial processor designers compete on implementation superiority.

6.1.2 ISA Specifications

RISC-V is specified as a base integer ISA, which is a must in any implementation, with additional extensions to the base ISA. Based on the width of the integer registers, the base instruction set has two variants, RV32I and RV64I. There is scope for a future variant RV128I for 128-bit address support [7]. It provides support for extensive customization and specialization, along with extension beyond the base instruction set, but doesn't allow redefining these base integer instructions.

There are four core instruction formats defined in the base instruction set, as shown follows:

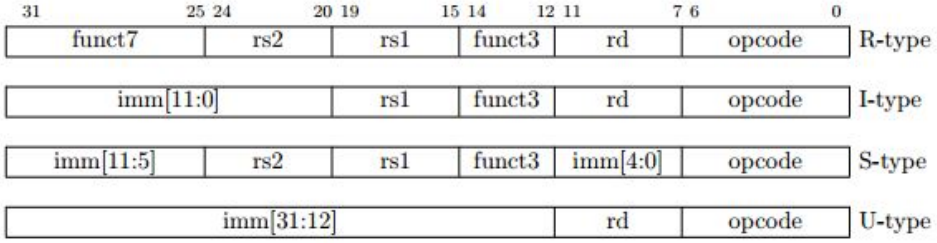


Figure 6.1: Base instruction formats in RISC-V [6]

Furthermore, there are two variations of the instruction formats (SB/UJ) based on how immediate values are taken care of. A few sets of standard extensions have been pre-defined to cater to general software development. These comprise integer multiplication/division (extension "M"), atomic operations (extension "A"), single-precision floating-point (extension "F"), and double-precision floating-point (extension "D"). A combination of these 4 standard extensions ("IMAFD") is extension "G", which results in a general-purpose scalar instruction set. RV32G and RV64G are currently set as default targets of the RISC-V compiler toolchains. The base set and extensions have been kept separated to have a simple instruction set.

The base integer instruction set for the 32-bit and 64-bit variants (RV32I/ RV64I) comprises 40 instructions, while the general-purpose instruction set (RV32G/RV64G) supports 122 instructions. There are 31 general purpose registers x1-x31 capable of storing integer values, register x0 being hardwired to the constant value 0. The register width is 32 bits for RV32I and 64 bits for RV64I. RISC-V is based on the load-store architecture, where arithmetic operations only function on the data stored the registers and only the load and store operations access memory. The user address space is byte-addressed and little-endian.

There also exists a reduced version of the base integer instruction set (RV32I), designed specifically for embedded systems (denoted by "E"). The major difference is that the number of registers in RV32E is reduced to 16 from 31 in RV32I. A draft proposal for the RISC-V standard compressed instruction set has been made which reduces static and dynamic code size by adding short 16-bit instruction encodings for conventional operations. Denoted by the extension "C", it could be added to any of the base ISA variants (RV32I, RV64I, and RV128I).

6.2 Software Tools and Setup Required

6.2.1 Software Tools

Several software tools and libraries are required to be downloaded and setup before any development on RISC-V can begin. A meta-repository is available on GitHub, with Git submodules containing every stable component of the RISC-V software toolchain [6]. The image below shows the software stack of riscv-tools:

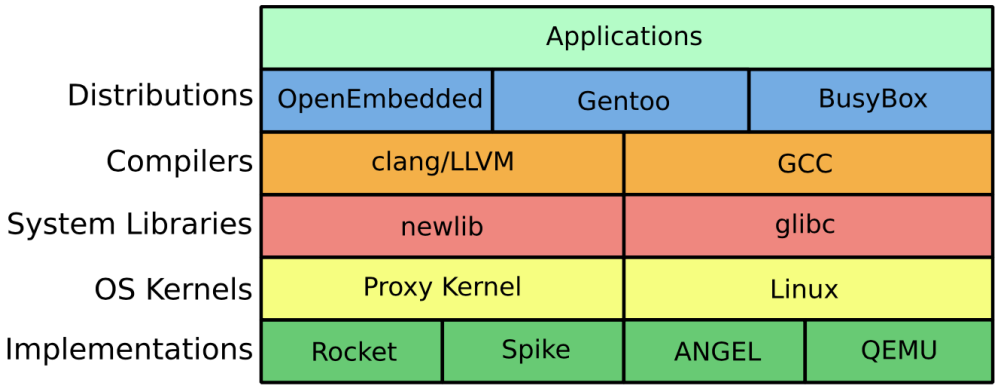


Figure 6.2: Software stack for RISC-V tools [6]

6.2.1.1 GNU Toolchain

The RISC-V GNU Toolchain is a RISC-V C and C++ cross compiler utility. It contents include binutils, gcc, newlib, glibc, and Linux UAPI headers. It supports two build modes: a generic ELD/Newlib toolchain and a more sophisticated Linux-ELF/glibc toolchain. The riscv-unknown-elf command is used to compile a program while riscv64-unknown-elf-gcc is used to assemble and link with gcc/binutils.

6.2.1.2 Front End Server

RISC-V front end server library facilitates communication between the host machine and the RISC-V target device on the Host-Target Interface (HTIF). It also provides a virtualized console and disk device [36].

6.2.1.3 Proxy Kernel

RISC-V proxy kernel is responsible for servicing system calls generated by code built and linked with the RISC-V Newlib port [36]. It handles system calls like open, close and printf. Abbreviated as `pk` (or `riscv-pk`), it is a lightweight application execution environment that can host statically-linked

RISC-V ELF binaries. It is designed to support tethered RISC-V implementations with limited I/O capability and thus handles I/O-related system calls by proxying them to a host computer [37].

6.2.1.4 ISA Simulator

RISC-V ISA simulator consists of a functional simulator known as Spike [36]. It implements a functional model of one or more RISC-V processors [38]. Since there is no proper operating system, it executes the generated binary running on top of the proxy kernel.

Spike takes the path of the binary to run as its argument, which is `pk`, located at `$RISCV/riscv-elf/bin/pk` and finds it automatically. The name of the program to be run is the argument for `riscv-pk`, which then executes the program. An interactive debug mode can be invoked in Spike with the -d command line flag or `SIGINT`, which enables single-stepping through instructions, setting break point conditions, printing register and memory contents, etc.

6.2.1.5 Opcodes

RISC-V Opcodes is a subcomponent of riscv-tools which enumerates all standard RISC-V instruction opcodes executable by the simulator, and control and status registers (CSRs). It also has a script to convert them into different formats (C, Scala, LaTex) [39].

6.2.2 Setup Required

There are many possible combinations to pick for the different layers of the stack shown in Figure 4, but there exist 2 of the most common workflows for RISC-V software development [6]. These are as follows:

- **Spike + pk**
  The use case for following this workflow are embedded or single applications.
  - Distributions: None
  - Compilers: clang/LLVM or GCC
  - System Libraries: newlib
  - OS Kernels: Proxy Kernel (pk)
  - Implementations: Spike
- **QEMU + Linux**
  The use case for following this workflow is Simple POSIX environment.
  - Distributions: BusyBox
  - Compilers: GCC
  - System Libraries: glibc
  - OS Kernels: Linux
  - Implementations: QEMU

In this case, the **Spike + pk** workflow was followed, the aim being familiarization with the RISC-V development phase. The selected workflow was easier to set up and took lesser time. This requires the components riscv-gnu-toolchain, riscv-fesvr, riscv-isa-sim, and riscv-pk.

The binaries generated against newlib system library will not be running on a full-blown operating system, but will require access to few crucial system calls [36]. The setup will thus require the installation of riscv-newlib. Newlib is a C library intended for embedded systems. It has the edge of not being unnecessarily complicated over Glibc, as well as having sufficient support. Also, its porting process is much simpler than that of Glibc because it only requires a few stubs of glue code. These stubs of code include

the system calls that are supposed to call into the operating system you're running on.

6.2.2.1 Toolchain

The following steps were followed to obtain and compile the RISC-V toolchain sources for the above-mentioned workflow:

1. To build GCC, several other Ubuntu packages like `flex`, `bison`, `autotools`, `libmpc`, `libmpfr`, and `libgmp` are required. Obtain the required Ubuntu packages required for installation with this command:
   sudo apt-get install autoconf automake autotools-dev curl libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc

2. Set the environment variables `TOP` as the directory where the tools will be installed, i.e. the current working directory:
   export $TOP = $(pwd)

3. Clone riscv-tools repository from GitHub:
   git clone https://github.com/riscv/riscv-tools.git

4. Change into the `riscv-tools` directory:
   cd $TOP/riscv-tools

5. Instruct Git to update its submodules:
   git submodule update —-init –recursive

6. Set the `RISCV` environment variable to point to the path where new tools will be installed:
   export $RISCV = $TOP/riscv

7. The `PATH` environment variable also needs to be set to point to the directory specified by RISCV:
   export $PATH = $PATH:$RISCV/bin

8. After everything has been set up, run the `build` script:
   ./build.sh

A successful build marks the end of the toolchain installation process.

6.2.2.2 Testing

After the installation procedure is finished, testing was done to make sure that there were no issues with the set up.

1. Change to the directory of toolchain installation:
   cd $TOP

2. Write a simple "Hello World" program in a C file

3. Build your program using the `riscv64-unknown-elf-gcc` command:
   riscv64-unknown-elf-gcc -o hello hello.c

4. Run the binary on the ISA Simulator (Spike):
   spike pk hello

The output generated should be "Hello world!", as shown in Figure 6.3

Figure 6.3: Hello World program execution on Spike

6.3 PicoRV32

To understand how RISC-V is implemented in real, a processor implementation PicoRV32 was studied and analyzed. PicoRV32 is a processor core implementing the RV32IMC instruction set. It can be configured as RV32E, RV32I, RV32M, RV32C, or RV32IMC, and optionally contains a built-in interrupt controller. It is available free and is open sourced under the ISC license [40].

6.3.1 About

PicoRV32 is a small sized processor which can operate at high frequencies (250-450 MHz on 7-Series Xilinx FPGAs). The core exists in two variants, `picorv32` and `picorv32_axi`. The former providing a simple native memory interface, easy to use in simple environments, while the latter providing an AXI-4 Lite Master interface that can be easily integrated with existing systems already using the AXI standard. It provides a selectable native memory interface, optional IRQ support and optional co-processor interface. This CPU core is designed to be added as an auxiliary processor in FPGA designs and ASICs. It could be easily integrated with most existing designs without exceeding clock domains.

6.3.2 Setup Required

PicoRV32 is based on the 32-bit version of RISC-V ISA, hence requires a setup of the complete toolchain targeting a pure RV32I CPU.

6.3.2.1 Toolchain

The following steps will download the toolchain required for programs to be compiled on PicoRV32 processor:

1. To build GCC, several other Ubuntu packages like `flex`, `bison`, `autotools`, `libmpc`, `libmpfr`, and `libgmp` are required. Obtain the required Ubuntu packages required for installation with this command:
   sudo apt-get install autoconf automake autotools-dev curl libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc

2. Create a new directory `/opt/riscv32i` to store the toolchain once installed:
   sudo mkdir /opt/riscv32i

3. Change permissions of the directory created in the previous step:
   sudo chown $USER /opt/riscv32i

4. Obtain the toolchain from the riscv-tools GitHub repository of RISC-V foundation and store it in the directory `riscv-gnu-toolchain-rv32i`:
   git clone https://github.com/riscv/riscv-gnu-toolchain riscv-gnu-toolchain-rv32i

5. Change into this directory:
   cd riscv-gnu-toolchain-rv32i

6. Checkout the specific commit version:
   git checkout 914224e

7. Update submodules within the repository:
   git submodule update –init –recursive

8. Create a new directory `build`:
   mkdir build

9. Change into this directory:
   cd build

10. Run the `configure` script, which specifies the instruction set architecture as RV32I, and the directory to install the toolchain as /opt/riscv32i:
    ../configure –with-arch=rv32i –prefix=/opt/riscv32i

11. Run the `make` script to build the toolchain:
    make -j$(nproc)

6.3.2.2 PicoRV32 Source Code

Now that the pure RV32I toolchain is downloaded, the CPU implementation can be downloaded and tested.

1. Clone PicoRV32 repository from GitHub:
   git clone https://github.com/cliffordwolf/picorv32.git

2. Run the `make` script to build the toolchain:
   cd ~/picorv32

3. Run the `make` script in the PicoRV32 source directory to download RV32IMC toolchains to be able to run all tests:
   make -j$(nproc) build-riscv32imc-tools - RV32IMC

6.3.2.3 Icarus Verilog

In order to test the working of the downloaded toolchain, the test bench provided by PicoRV32 must be run. This requires a software called *Icarus Verilog*, which needs to be installed. Icarus Verilog is a Verilog simulation and synthesis tool. It operates as a compiler, compiling source code written in Verilog (IEEE-1364) into some target format. For batch simulation, the compiler can generate an intermediate form called vvp assembly. This intermediate form is executed by the vvp command. For synthesis, the compiler generates netlists in the desired format [41]. The following steps will download and install Icarus Verilog:

1. Clone the GitHub repository:
   git clone https://github.com/steveicarus/iverilog.git

2. Change into the `iverilog` directory:
   cd iverilog

3. Update to the latest code in master branch:
   git pull origin master

4. Install `autoconf` and `gperf` packages installed for the configuration script to work (skip if step already done):
   sudo apt-get install autoconf gperf

5. Run `autoconf.sh` script:
   sh autoconf.sh

6. Run `configure` script:
   ./configure

7. Run `make` command with sudo access:
   sudo make

8. Run `make` install:
   make install

6.3.3 Execution of Simple C Programs

After the installation of all tools required for using PicoRV32 was completed, the testbench provided was run by executing the make command. The following output was obtained: This demonstrates that the installation was done

```
Cycle counter ......... 281843
Instruction counter .... 63691
CPI: 4.42
DONE
Number of fast external IRQs counted: 35
Number of slow external IRQs counted: 4
Number of timer IRQs counted: 22
TRAP after 322543 clock cycles
ALL TESTS PASSED.
```

Figure 6.4: Successful execution of PicRV32 testbench

successfully. The output is produced by different C files which are executed on the PicoRV32 CPU. The functions written in the C files are called in an assembly file *start.S*, located in `picorv32/firmware` directory. Along with C function calls, the assembly file also contains various tests in assembly files which get executed upon running make command.

Upon further analysis of the compilation and execution flow, it was observed that the Icarus Verilog compiler compiles the *testbench.v* and *picorv32.v* files and generates an intermediate `vvp` assembly file, *testbench.vvp*. This is followed by the compilation of *start.S* assembly file, the C codes and test assembly files into their binaries using `riscv32-unknown-elf-gcc` command. This is succeeded by the generation of an executable file *firmware.elf*, which is translated into a *firmware.bin* file with `riscv32-unknown-elf-objcopy` utility. In order to run the machine code on the processor, a script makehex.py in the firmware directory was used to convert the *firmware.elf* file into *firmware.hex* file. The *firmware.hex* file contains instructions as defined by the RISC-V specification in the 32-bit format (represented as 8 digit hexadecimal numbers), which are passed to PicoRV32. The `vvp` command of Icarus Verilog runs the previously generated *testbench.vvp* file and executes the machine code on PicoRV32. The whole flow is shown as an activity diagram in Figure 6.5.

Observing the testbench output provided deep insights into how the execution maps from the high-level language C code to the low-level machine code. To understand how the processor executes these instructions, the Verilog file *picorv32.v* was studied and understood. The instructions are loaded from the firmware.hex file in the testbench, which is stored in a 1024-bit wide register.

```
reg [1023:0] firmware_file;
initial begin
```

```
end
if(!$value$plusargs("firmware=%s", firmware_file))
        firmware_file = "firmware/firmware.hex";
$readmemh(firmware_file, mem.memory);
```

The above code snippet shows the loading of the instructions into a 32-bit wide, 16384 lines long memory in an `axi4_memory` instance. The instance is declared as follows:

```
reg [31:0]      memory [0:64*1024/4-1];
```

This memory is the source of instructions for the processor, which lacks a default instruction memory. As mentioned in 5.3.1, PicoRV32 is available in two variants. In this case, the `axi4_memory` is interfaced to the `picorv32_axi` version of PicoRV32 core, using the `picorv32_axi_adapter`.

Along with being the instruction source, this `axi4_memory` memory instance acts as the data memory as well. This implies that PicoRV32 follows a **von Neumann** architecture with a unified memory for storing data and instructions. The assembly file *start.S* is used to allocate specific parts of the memory instance to be used as data memory. This brings complexity in the form of management of different sections of the same memory for instruction and data storage purposes.

On an average, PicoRV32 was observed to have a Cycles Per Instruction (CPI) value of 4.8. The execution of some specific instructions increased the CPI.

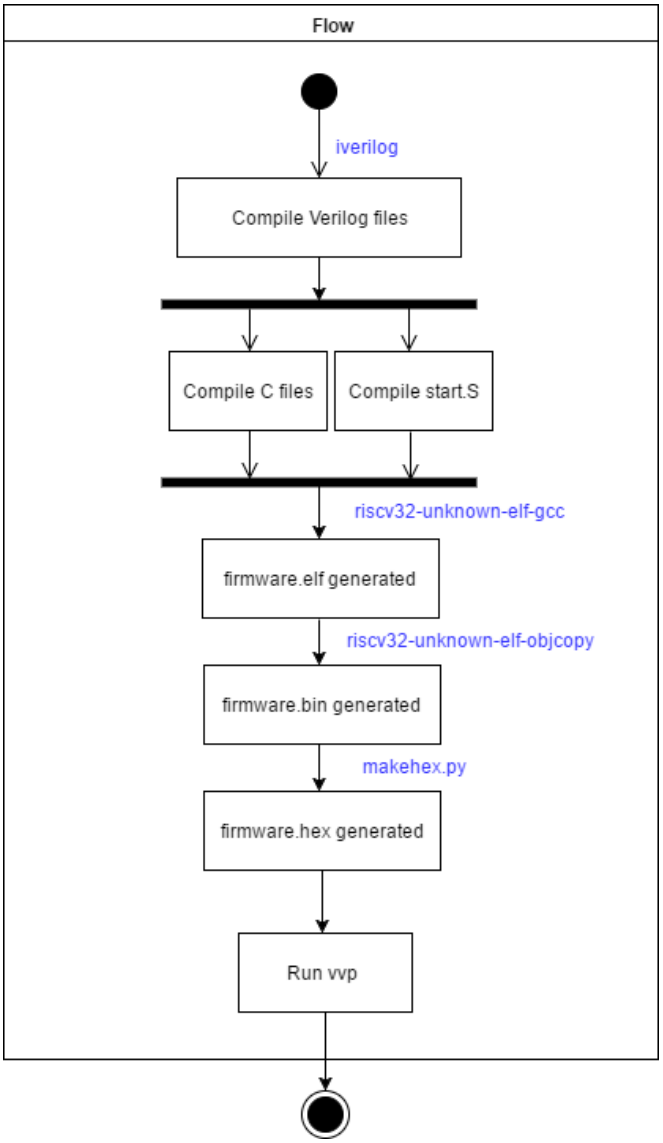


Figure 6.5: Compilation flow before execution on PicRV32

6.4 Implementation of RISC-V Processor

After understanding the working of PicoRV32 from a high-level perspective, the next step is to design and implement a small processor based on the

RISC-V 32-bit base integer instruction set. PicoRV32 has several features, like custom instructions for IRQ handling, a unified memory for data and instructions, which make the processor complicated and heavier.

6.4.1 Objective

The objective is to develop a smaller and more efficient processor and analyze whether it could perform better than PicoRV32. The initial implementation intends to be capable of performing only a small set of instructions and compare its performance with PicoRV32. A fully functional processor implementing RISC-V ISA meeting the desired requirements (as mentioned in Section 6.4.2) must be developed before it could be further extended.

ModelSim SE-64 10.5 was used for simulating the RISCVProcessor in Verilog.

6.4.2 Design

6.4.2.1 Requirements

The design of any RISC based processor follows the load-store architecture. This would be a 32-bit variant of the base integer instruction set, meaning that the registers width would be 32 bits. It would support instructions from the base integer instruction set and the multiplication extension (RV32IM). To start off with, the following instructions need to be present in our RISC-V implementation:

- Arithmetic: ADD, ADDI, MUL
- Load/Store: LW, SW
- Jump: JAL
- Branch: BNE

Among the above instructions, ADD, MUL, fall into the category of R-type instructions; ADDI and LW fall into the I-type instruction category, SW is in the S-type instruction category; JAL is in the UJ-type instruction category; BNE uses the SB-type instruction format.

Harvard architecture is selected over von Neumann architecture, as data and instruction memory separation is preferred. Two separate memory systems can perform better than having a single, unified memory for both data and instructions. The various components required by the processor implementation would be the control unit, registers, Arithmetic and Logic Unit (ALU) and instruction memory.

6.4.2.2 Components

The processor would be built using the required components as stated above in 6.4.2.1. This section discusses the functional requirements desired in the components.

1. **Registers**
   Under the RISC-V specification, the base integer instruction set has a set of 32 registers, each 32-bit wide (x1-x31, x0 hardwired to constant 0). The Control Unit decodes instructions and extracts the register address on which the ALU needs to perform specified instructions. Data can be transferred between memory and registers using the load and store instructions. Thus, in a load store architecture, register memory is significant and should be wisely used in order to reduce overheads of memory accesses. It can be declared as shown in the following code snippet:

```
reg [31:0] regdata [0:31];
regdata[0] = 32'h0;  //Constant 0 stored in location 0
```

2. **Decode Unit**

   The decoder determines the meaning of each instruction the processor needs to perform. It fetches the instructions from the instruction memory, and extracts necessary information commanding the CPU what instruction must be performed on what data.

   The decoder extracts the information of register addresses from which data needs to be accessed, as well as the operation required to be done. This is specified by the 7-bit `opcode`, 3-bit `func3` and 7-bit `func7` bits in the 32-bit instruction, as seen in Figure 6.1.

   The design of the decode unit needs to be done according to RISC-V instruction specifications, as seen in Figure 6.6. The `func7` and `func3` bits distinguish several instructions. The `opcode` can be same for various instructions, typically for the same class of instruction (R-type, I-type, etc.).

0000000	rs2	rs1	000	rd	0110011	ADD
imm[11:0]		rs1	000	rd	0010011	ADDI
0000001	rs2	rs1	000	rd	0110011	MUL
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE

Figure 6.6: Formats for RISC-V instructions to be implemented [7]

3. **Control Unit**

   The Control Unit determines the execution of operations on the processor. It instructs the processor's components like memory, ALU and input and output devices to respond to the program instructions. It can be perceived as the brain inside the CPU, making it the brain inside the brain of the computer. It is responsible for controlling the flow of data

in specific ways for different instructions. This is done by signaling various flags which activate or deactivate different components of the CPU.

It consists of two registers, namely the Program Counter (PC) register and the Instruction Register (IR). PC points to the address of the instruction to be executed, while IR contains the instruction being currently interpreted. Instructions are fetched from the instruction memory and decoded in the Control Unit. The data required for the instruction to execute is extracted from the registers or data memory, and passed on to the ALU to perform the operation specified by the instruction opcode.

4. **Memory**
   Memory stores instructions and data. Being a Harvard Architecture processor, there would be a clear separation of instructions and data. A little endian memory system would be followed. The instruction memory stores the instructions generated by the makehex.py script, converting the firmware.bin file to firmware.hex format. The width and length of the instruction memory are 32 bits and 16384 lines, respectively.

5. **Arithmetic and Logic Unit**
   The Arithmetic and Logic Unit of a CPU performs arithmetic and logical operations on operands specified by the instructions. It is the fundamental unit of the CPU of any computing device. After performing an operation, the result is stored back into the register file at the address specified by the instruction.

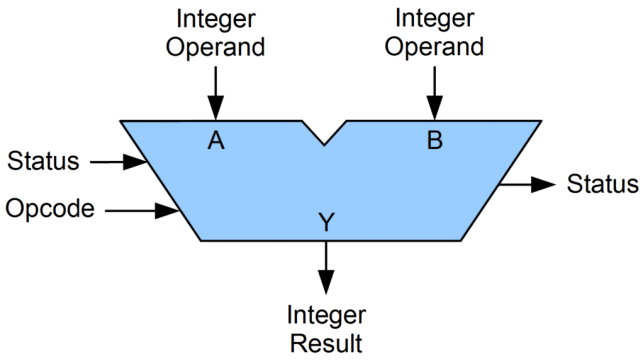


Figure 6.7: Symbolic representation of an ALU

6.4.3 Implementation Details

6.4.3.1 Instruction Execution Flow

The implementation is built using several modules in Verilog. There exist modules `instruction_memory`, `data_memory`, `program_counter`, `decoder`, `control`, `regfile` and `alu`, as mentioned in Section 6.4.2.2 to produce a fully functional processor based on RISC-V ISA. The top-level module connecting all these modules is called `rvproc`. The code for the top level module is listed in Appendix B.

The testbench is initialised with the reset signal enabled and clock signal set with a period of 10 time units. After 20 time units, reset is disabled and the instruction execution commences. The program counter returns the next instruction address in `PC_IN` and current instruction address is `PC_OUT` which is fed to the instruction memory. Module `instruction_memory` initializes a memory of size 16384 with instructions from "firmware.hex" file and returns the instruction at address `PC_OUT`. The `decoder` decodes this instruction based on the opcode, funct3 and funct7 values and returns an instruction code corresponding to the decoded instruction based on the definitions. It also returns register addresses of the source and the destination,

and sing-extended immediate values, based on the instruction.

The `control` module provides the overall path for the instruction being currently executed. It does so by signaling certain flags, instructing the processor to read or write from/to the register file, use memory or ALU output for register write-back, or if a branch instruction is being executed. For an R-type instruction, the source addresses of the registers are made available and `regfile` returns the data stored in the addresses. This data is accepted by the `alu` as inputs to perform the computation as decoded from the instruction. ALU output is written back to the register destination address if flag `AluToReg` is high. Immediate value is used in place of the second source register for ALU computations for I-type instructions.

The processor implementation also includes a data memory. The `data_memory` module is initialized as a memory of size 16384 with dummy data taken from a "data_memory.hex" file. `read_en` and `write_en` flags enable data to be read and written from/to this memory. The address to read or write data in the memory is provided by the ALU from the base address, which is then propagated to `data_memory` for load or store instructions. The data read from the memory is written to the destination register when the flag `AluToReg` is low.

The program counter is updated by incrementing `PC_OUT` by 1 and assigning it to `PC_IN`, which happens every clock cycle. `PC_IN` is updated with the sum of the current program counter and an immediate value for Jump instructions. Flags which allow the `PC_OUT` to be stored in the register are set. For conditional branch instructions, flags are set when certain conditions are met which allow the program counter to be set in the required manner.

6.4.3.2 Testbench Simulation of Instructions

MUL

32-bit Instruction: 0x024100b3
Assembly Representation: MUL R1, R2, R4 Meaning: R1 = R2 x R4

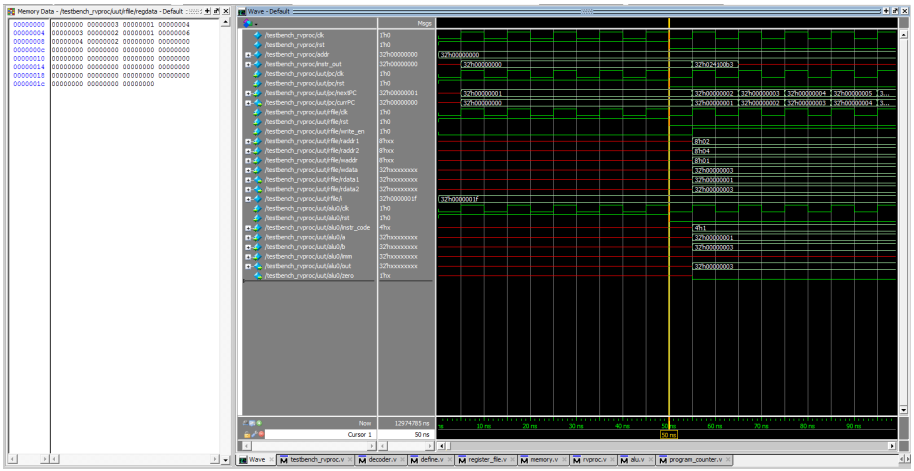


Figure 6.8: Testbench simulation result for MUL

`MUL` is an R-type instruction which multiplies the values stored in 2 source register addresses (R2 and R4 here), and stores it in the destination (R1 in this case) address. The `decoder` extracts register addresses from the instruction and puts them onto raddr1 and raddr2, which is sent to `regfile`. The data stored (1 and 3) is forwarded to the `ALU` as a and b for carrying out multiplication, following which the product aluout is stored into R1 through waddr of the `regfile`. Control signals generated by the `control unit` enable write-back to the register. The output 3 is reflected in R1 as seen in Fig. 6.8.

ADDI

32-bit Instruction: 0x06418393
Assembly Representation: ADDI R7, R3, #100 Meaning: R7 = R3 + 100

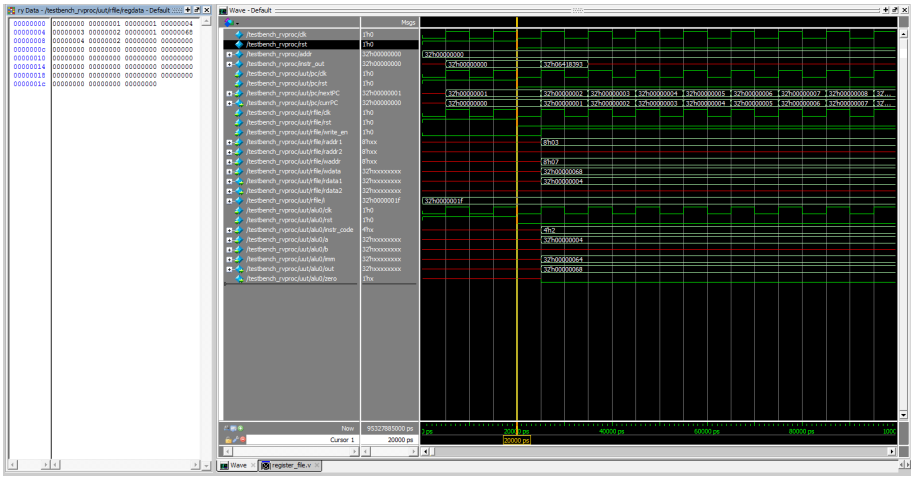


Figure 6.9: Testbench simulation result for ADDI

`ADDI` is an I-type instruction which adds an immediate value to the value stored in a source register addresses (R3 here), and stores it in the destination (R7 here) address. The `decoder` extracts the immediate value (decimal 100 or hexadecimal 64) and the source address from the instruction and puts them onto imm and raddr1, respectively. The immediate value is sign-extended to 32-bits and raddr1 is sent to `regfile` for fetching the data required. The data from R3 (4) and the immediate value is forwarded to the `ALU` as a and imm_ext for carrying out addition, following which the sum aluout is assigned to wdata and stored in address waddr of the `regfile`. Control signals generated by the `control unit` enable write-back to the register. The output 68 is reflected in R7 as seen in Fig. 6.9.

LW

32-bit Instruction: 0x0000a283
Assembly Representation: LW R5, [R1, #0]
Meaning: R5 = D_MEM[[R1] + #0]

`LW` is an I-type instruction which loads a value from the data memory on to a specific address the register file. This is done by specifying a source

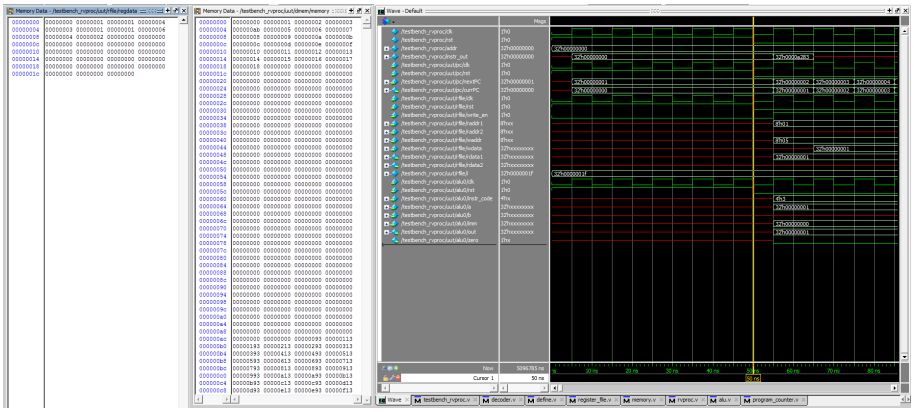


Figure 6.10: Testbench simulation result for LW

address and an address offset. Figure 6.10 shows the data stored in register file, data memory and simulation in three windows. The `decoder` extracts the immediate value (0) and the source address (R1) from the instruction and puts them onto imm and raddr1, respectively. The data at R1 of `regfile` is assigned to rdata1 and the immediate value is extended to 32 bits. These are forwarded to the `ALU` and added, resulting in 1, the memory address (1) to read from. The data read from the data memory (1) is assigned to wdata and written to waddr location of `regfile`, thus 1 is written to R5. Control signals generated by the `control unit` enable memory read from data memory and write-back to the register. The value of R5 is updated to 1 in the next clock cycle, to avoid synchronisation issues, as seen in Figure 6.10.

SW

32-bit Instruction: 0x00522423
Assembly Representation: SW R5, [R4, #8]
Meaning: D_MEM[[R4] + #8] = R5

`SW` is an S-type instruction storing data from a register to a specified location in the data memory. The `decoder` fetches the instruction and identifies

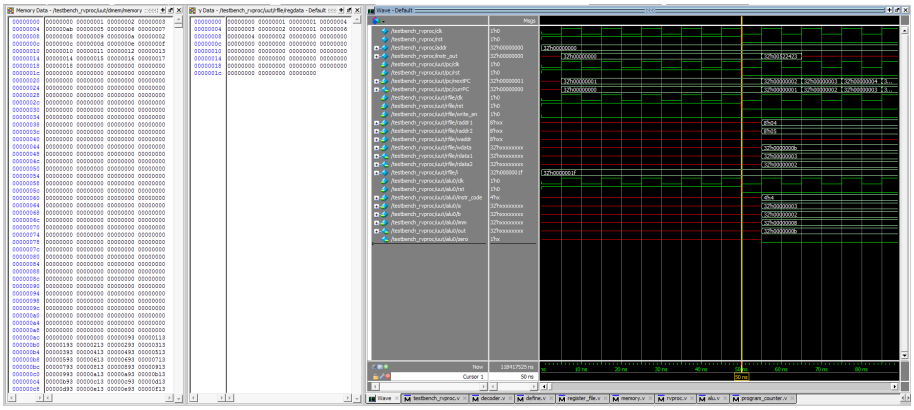


Figure 6.11: Testbench simulation result for SW

raddr1 as R4 (3), raddr2 as R5 (2) and the immediate value is imm (8). The data read from R4 is 3 and R5 is 2, updated in rdata1 and rdata2, respectively. The immediate value (8) is sign extended to 32 bits as imm_ext and forwarded along with rdata1 to the `ALU`. It adds them to compute aluout (b in hex, 11 in decimal), assigned to waddr in data memory for the store operation. rdata2 is passed to the data memory with the write address waddr and location b of data memory is updated with 2, as seen in Figure 6.11. Control signals generated by the `control unit` disable memory read from data memory and write-back to the register and enable writing to the data memory.

Chapter 7

Conclusions and Future Work

This chapter concludes and summarizes this report along with discussing the path to take in the future.

7.1 Conclusions

The first part of this report discussed OpenCL programming model in a detailed manner to explain the results of the experiments conducted thereafter on multiple OpenCL applications. In the beginning, experiments were conducted using a simple matrix multiplication OpenCL kernels with varied data sizes. The performance with respect to execution time and GOPS on different devices and different mappings of computations on each device were profiled. It was concluded that different devices perform the best for particular local sizes chosen for the mapping of kernels onto the device.

This was followed by profiling a real-world application with heavy computational requirements. An existing CNN application following the LeNet-5 architecture was profiled across few accelerating devices, and their performance was inspected. It was observed that the device characteristics effected the amount of parallelisation done by OpenCL, having a direct impact on

the execution times.

The second part of the report shifted focus towards the open source and free RISC-V ISA, explaining its benefits and advantages, its design specifications and installation steps for development. An existing implementation of a RISC-V processor was studied as an example, and its features, installation requirements and steps taken to execute programs were understood.

Following this, an understanding of RISC-V ISA specifications was attained in order to design and develop a new, simpler implementation of a processor with the aim of developing the fundamental unit for the hypothetical engine with huge computational capabilities. RV32IM variant of the ISA was chosen as the target support for this processor.

7.2 Future work

The future work mainly comprises extending the support of instructions on this RISC-V processor. The next step would be to enhance the processor to be able to run C programs. This would be done by using two assembly scripts, for setting up the instruction and data memories to be required for C program execution. The future goal is to emulate the support provided by PicoRV32, with reduced complexity.

Furthermore, the processor needs to be configured such that it could be detected by the OpenCL platform to be used as an accelerator device. A network of such devices could thus be combined to form a dedicated acceleration engine, capable of executing applications with high computational requirements. This substantiates the requirement of having a parallel programming language supported on the processor implementation to divide the tasks among multiple cores of the hypothetical accelerator.

Appendix A

CNN

Table A.1: Model of LeNet-5 Layers

```
#ifndef _LENET5_MODEL_H_
#define _LENET5_MODEL_H_
#include <stdio.h>

#define CONV1_NO_INPUTS 1
#define CONV1_NO_OUTPUTS 20
#define CONV1_FILTER_HEIGHT 5
#define CONV1_FILTER_WIDTH 5

extern const float conv1_weights[CONV1_NO_OUTPUTS][CONV1_NO_INPUTS*CONV1_FILTER_HEIGHT*CONV1_FILTER_WIDTH];
extern const float conv1_bias[CONV1_NO_OUTPUTS];

#define CONV2_NO_INPUTS 20
#define CONV2_NO_OUTPUTS 50
#define CONV2_FILTER_HEIGHT 5
#define CONV2_FILTER_WIDTH 5

extern const float conv2_weights[CONV2_NO_OUTPUTS][CONV2_NO_INPUTS*CONV2_FILTER_HEIGHT*CONV2_FILTER_WIDTH];
extern const float conv2_bias[CONV2_NO_OUTPUTS];

#define IP1_NO_INPUTS 800
#define IP1_NO_OUTPUTS 500

extern const float ip1_weights[IP1_NO_OUTPUTS][IP1_NO_INPUTS];
extern const float ip1_bias[IP1_NO_OUTPUTS];

#define IP2_NO_INPUTS 500
#define IP2_NO_OUTPUTS 10

extern const float ip2_weights[IP2_NO_OUTPUTS][IP2_NO_INPUTS];
extern const float ip2_bias[IP2_NO_OUTPUTS];

#endif // _LENET5_MODEL_H_
```

Appendix B

RISC-V

Table B.1: RISC-V Processor Top Level Unit

```
module rvproc (input clk,
               input rst,
               input [31:0] PCOUT,
               output [31:0] instr_out);

//Program Counter
wire [`ISIZE-1:0] PC_IN;
wire [`ISIZE-1:0] PC_OUT;
wire reg_write_en;
wire imem_read_en;
//To control reading and writing from data mem
wire dmem_read_en, dmem_write_en;
wire aluToReg, branch, jr, jal;
wire [`ASIZE-1:0] r_src_addr1, r_src_addr2, r_dest_addr;
reg [`ISIZE-1:0] addr;
// Outputs
wire [`DSIZE-1:0] data_out;
wire [`DSIZE-1:0] write_data_reg;
wire [`DSIZE-1:0] write_data_mux;
wire [3:0] instr_id;
wire [6:0] opcode;
wire [`DSIZE-1:0] imm_ext;
wire [`DSIZE-1:0] alu_out, r_data1, r_data2;
wire zflag;
assign PC_IN = PC_OUT + 32'b1;
assign write_data_mux = (jal==1) ? PC_OUT : write_data_reg;
assign write_data_reg = (aluToReg==1) ? (alu_out) : (data_out);
//write_data_reg is to be written in Register file
program_counter pc(.clk(clk), .rst(rst), .nextPC(PC_IN), .currPC(PC_OUT));
//PC_OUT is your PC value and PC_IN is your next PC
instruction_memory imem (.clk(clk), .rst(rst), .read_en(1'b1), .r_addr(PC_OUT), .instr_out(instr_out));
data_memory dmem (.clk(clk), .rst(rst), .read_en(dmem_read_en), .write_en(dmem_write_en),
.addr(alu_out), .w_data(r_data2), .data_out(data_out));
decoder dec (.instr(instr_out), .instr_code(instr_id), .rs1(r_src_addr1), .rs2(r_src_addr2),
.imm(imm_ext), .rd(r_dest_addr));
control ctrl (.instr_code(instr_id), .wen(reg_write_en), .mem_read(dmem_read_en), .mem_write(dmem_write_en),
.AluToReg(aluToReg), .branch(branch), .jr(jr), .jal(jal));
regfile rfile(.clk(clk), .rst(rst), .write_en(reg_write_en), .raddr1(r_src_addr1),
.raddr2(r_src_addr2), .waddr(r_dest_addr), .wdata(write_data_reg), .rdata1(r_data1), .rdata2(r_data2));
alu alu0(.clk(clk), .rst(rst), .instr_code(instr_id), .a(r_data1), .b(r_data2), .imm(imm_ext), .out(alu_out), .zero(zflag)
      );

endmodule
```

Bibliography

[1] Andreas Traber. PULPino, a small single-core RISC-V SoC. In *3rd*.

[2] AJ Guillon. Opencl 1.2: High level overview. `https://www.youtube.com/watch?v=8D6yhpiQVVI`, 2013.

[3] Stanford. Neural Networks Part 1. `http://cs231n.github.io/neural-networks-1/`.

[4] Ujjwal Karn. A Quick Introduction to Neural Networks. `https://ujjwalkarn.me/2016/08/09/quick-intro-neural-networks/`, August 2016.

[5] Adam W Harley. An Interactive Node-Link Visualization of Convolutional Neural Networks. In *ISVC*, pages 867–877, 2015.

[6] Albert Ou. RISC-V Software Tools Bootcamp. `https://riscv.org/wp-content/uploads/2015/02/riscv-software-stack-tutorial-hpca2015.pdf`, February 2015.

[7] Andrew Waterman, Yunsup Lee, and Krste Asanović. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1. `https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.pdf`, May 2016.

[8] Nicolas Melot, Janzén, and Christoph Kessler. Mimer and Schedeval: Tools for Comparing Static Schedulers for Streaming Applications on Manycore Architectures. pages 146–155, 2015.

[9] Nicolas Melot, Janzén, and Christoph Kessler. A 48-Core IA-32 Processor in 45 nm CMOS Using On-Die Message-Passing and DVFS for Performance and Power Scaling. pages 173–183, 2010.

[10] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for CPUs: Stream Computing on Graphics Hardware. 2004.

[11] M. Leeser, S. Corid, E. Miller, H. Yu, , and M. Trepanier. Parallel-beam backprojection: An FPGA implementation optimized for medical imaging. volume 39, pages 295–311, 2005.

[12] ACM Computing Surveys. A Survey Of Techniques for Architecting and Managing Asymmetric Multicore Processors. `https://www.academia.edu/18301534/A_Survey_Of_Techniques_for_Architecting_and_Managing_Asymmetric_Multicore_Processors`.

[13] RISC-V Foundation. RISC-V: The Free and Open RISC Instruction Set Architecture. `https://riscv.org`.

[14] Gopalkrishna Hegde and Siddhartha. CaffePresso: An Optimized Library for Deep Learning on Embedded Accelerator-based platforms.

[15] Pekka Jääskeläinen, Carlos Sánchez de La Lama, Erik Schnetter, Kalle Raiskila, Jarmo Takala, and Heikki Berg. pocl: A performance-portable opencl implementation. *International Journal of Parallel Programming*, 43(5):752–785, 2015.

[16] Yann Lecun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-Based Learning Applied to Document Recognition. In *IEEE*, 1998.

[17] Stanford. RISC vs CISC. https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/.

[18] Krste Asanović and David A. Patterson. Instruction sets should be free: The case for risc-v. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html, August 2014.

[19] C. Celio, D. A. Patterson, and K. Asanovic. The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor. *Electrical Engineering and Computer Sciences, University of California*, 2015.

[20] H Van der Wijst. An Accelerator based on the ρ-VEX Processor: an Exploration using OpenCL. Master's thesis, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, Netherlands, 2015.

[21] Wikipedia. OpenCL. https://en.wikipedia.org/wiki/OpenCL.

[22] Ryoji Tsuchiyama, Takashi Nakamura, Takuro Iizuka, Akihiro Asahara, Jeongdo Son, and Satoshi Miki. The OpenCL Programming Book. https://www.fixstars.com/en/opencl/book/OpenCLProgrammingBook/what-is-opencl/, 2012.

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

[24] Yann LeCun. LeNet-5, Convolutional Neural Networks. http://yann.lecun.com/exdb/lenet/.

[25] A. Krizhevsky, I. Sutskever, and G.E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, pages 1106–1114, 2012.

[26] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *CVPR*, 2014.

[27] Ujjwal Karn. An Intuitive Explanation of Convolutional Neural Networks. https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets, August 2016.

[28] Stanford. Convolutional Neural Networks. http://cs231n.github.io/convolutional-networks.

[29] Eugenio Culurciello. Neural Network Architectures. https://culurciello.github.io/tech/2016/06/04/nets.html.

[30] Wikipedia. MNIST Database. https://en.wikipedia.org/wiki/MNIST_database.

[31] Ernst Kussui and Tatiana Baidyk. Improved method of handwritten digit recognition tested on mnist database. 22, 2004.

[32] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The MNIST Database. http://yann.lecun.com/exdb/mnist/.

[33] Gopalkrishna Hegde. Lenet5 opencl application. https://github.com/gplhegde/papaa-opencl/tree/master/opencl-src/mnist, 2012.

[34] XDA Developers. Open source risc-v core designs, why google cares and why they matter. https://www.xda-developers.com/risc-v-cores-and-why-they-matter, January 2016.

[35] ANSI/IEEE Std 754-2008. IEEE Standard for Floating-Point Arithmetic. http://ieeexplore.ieee.org/document/4610935/, August 2008.

[36] RISC-V Foundation. Software Tools. https://riscv.org/software-tools/#linuxman.

[37] RISC-V Foundation. RISC-V Proxy Kernel and Boot Loader. https://github.com/riscv/riscv-pk#about.

[38] Andrew Waterman and Yunsup Lee. RISC-V ISA Simulator. https://riscv.org/software-tools/risc-v-isa-simulator/, June 2011.

[39] RISC-V Foundation. RISC-V Opcodes. https://github.com/riscv/riscv-opcodes.

[40] Clifford Wolf. PicoRV32 - A Size-Optimized RISC-V CPU . https://github.com/cliffordwolf/picorv32.

[41] Stephen Williams. Icarus Verilog. http://iverilog.icarus.com/.