UNIVERSITAT POLITÈCNICA DE VALÈNCIA

# FINAL PROJECT OF MASTER

**TITLE: Implementation of resilient algorithm Backpropagation on ARM-FPGA through OpenCL**

**QUALIFICATIONS: Master of Electronics Systems Engineering**

**AUTHOR: Messelka Mohamed**

**SUPERVISORE: Rafeal Gadea Girones**

## Dedicated to:

*I would to thank all who helps me to finish my master study at UPV, my Parents, the EMMAG team, supervisor Rafeal Gadea Girones, all my Professors and classmates in UPV, all my professors and classmates in UMBB-IGEE, the international office in UPV, laboratory technicians and all family members of UPV.*

*Thank you all*

# Contents

**Introduction……………………………………………………………….……01**

**Chapter 1. Conceptual Foundations of OpenCL…………………….…03**

**1.1. Platform Model……………………………………………………………03**

**1.2. Execution Model…………………………………………………………04**

1.2.1. How a Kernel Executes on an OpenCL Device…………….04

1.2.2. Context……………………………………………………………06

1.2.3. Command-Queues……………………………………….………07

**1.3. Memory Model……………………………………………………………08**

**1.4. Programming Model……………………………………………………09**

**1.5. The OpenCL Framework………………………………………………10**

1.5.1. Platform API……………………………………………………10

1.5.2. Run Time API…………………………………………………..10

1.5.3. Kernel Programming Language…………………………….11

**1.6 OpenCL Overview…………………………………………………………14**

**Chapter 2. Conceptual Foundations of OpenCL with Altera SDK…………………………………………………………………………..15**

**2.1. The OpenCL Stander on FPGA…………………………………………..15**

**2.2. FPGA OpenCL Architecture……………………………………………18**

**2.3. OpenCL-to-FPGA framework…………………………………………..19**

**2.4. Kernel Compiler…………………………………………………………..20**

**2.5. Memory Organization……………………………………………………..24**

2.5.1. Optimize Global Memory Accesses……………………….…25

2.5.2 Contiguous Memory Accesses……………………………….…26

2.5.3 Constant Cache Memory…………………………………………27

**2.6. Strategies for Improving NDRange Kernel Data Processing Efficiency……………………………………………………………………………..28**

**2.7 Optimize Floating-Point Operations……………………………………32**

2.7.1. Floating-Point versus Fixed-Point Representations…….34

**Chapter 3. Matrix-multiplication (Study an example)……………35**

**3.1. Tiling in the local memory…………………………………………………….35**
**3.2. Host Code……………………………………………………………………………36**
**3.3. Kernel code…………………………………………………………………………37**

**Chapter 4. Application of matrix-multiplication kernel in Backpropagation……………………………………………………………..38**

**4.1. The time execution……………………………………………………………38**
4.1.1 $T_{cw}$……………………………………………………………………………38
4.1.2 $T_{uw}$……………………………………………………………………………40
**4.2. Original algorithm……………………………………………………………41**
**4.3 Final algorithm…………………………………………………………………42**

**Chapter 5. Ability to quickly integrate IPs in OpenCL and Results………………………………………………………………………………45**

**5.1. OpenCL Library…………………………………………………………………45**
5.1.1. RTL Modules and the OpenCL Pipline……………………….46
5.1.2. Packaging an OpenCL Helper Function File………………48
5.1.3. Packaging an RTL Component for an OpenCL Library…49
5.1.4. Restrictions and Limitations in RTL Support………………49
5.1.5. Verifying the RTL Modules………………………………………50
5.1.6 .Packaging Multiple Object Files into a Library File……..50
5.1.7. Specifying an OpenCL Library when Compiling…………51
5.1.8 OpenCL Library Command-Line Options…………………….51

**5.2. Function Tangent hyperbolic 'tanh'…………………………………52**
**5.3. Results…………………………………………………………………………….53**
5.3.1. Comparison……………………………………………………………57
5.3.1. Comments………………………………………………………………68

**Conclusion…………………………………………………………………………69**

**References…………………………………………………………………………68**

# INDEX OF FIGURES

Figure 1.1 OpenCL Platform……………………………………………………………03
Figure 1.2 An example of how the global IDs, local IDs, and work-group indices are related for a two-dimensional NDRange………………………….....06
Figure 1.3 The Memory Model in OpenCL…………………………………………..09
Figure 1.6 general block diagram of OpenCL………………………………………..14
Figure 02 Trend of Programmable and Parallel Technology……………………15
Figure 2.1.1 Overview of OpenCL……………………………………………………16
Figure 2.1.2 Example of OpenCL Implementation on an FPGA………………..16
Figure 2.1.3 Pipelined information…………………………………………………..17
Figure 2.2 OpenCL system implementation…………………………………………..18
Figure 2.3 OpenCL-to-FPGA framework……………………………………………..19
Figure 2.4.1 basic blocks……………………………………………………………….21
Figure 2.4.2 basic block module………………………………………………………22
Figure 2.5.1 Global Memory Partitions……………………………………………...25
Figure 2.5.2 Contiguous Memory Access…………………………………………...27
Figure 2.6.1 Compute Unit Replication versus Kernel SIMD Vectorization……………………………………………………………………………………31
Figure 2.6.2 Combination of Compute Unit Replication and Kernel SIMD Vectorization……………………………………………………………………………………..31
Figure 2.7.1 Default Floating-Point Implementation……………………………32
Figure 2.7.2 Balanced Tree Floating-Point Implementation……………………32
Figure 03 matrix-multiplication by sub-block………………………………………36
Figure 4.1.1 Multiplayer perceptron…………………………………………………39
Figure 4.2 Resilient Backpropagation original……………………………………..42
Figure 4.3 Resilient Backpropagation…………………………………………………43
Figure 5.1 Overview of Altera SDK for OpenCL's Library Support…………..45
Figure 5.1.1 Parallel Execution Model of AOCL Pipeline Stages………………46
Figure 5.1.2 Integration of an RTL Module into an AOCL Pipeline…………..47
Figure 5.2 The structure of the function tangent hyperbolic…………………..52

# INDEX OF Graphs

Graph 5.1 Comparison Kernel VS ARM CPU, BZ=4 WI=4, numPatterns_max is Variable……………………………………………………………………………………………59
Graph 5.2 Comparison Kernel VS ARM CPU, BZ=4 WI=4, numHidden1 is Variable……………………………………………………………………………………………59
Graph 5.3 Comparison Kernel VS ARM CPU, BZ=4 WI=4, numHidden2 is Variable……………………………………………………………………………………………60
Graph 5.4 Comparison Kernel VS ARM CPU, BZ=8 WI=8, numPatterns_max is Variable……………………………………………………………...60
Graph 5.5 Comparison Kernel VS ARM CPU, BZ=8 WI=8, numHidden1 is Variable……………………………………………………………………………………………61
Graph 5.6 Comparison Kernel VS ARM CPU, BZ=8 WI=8, numHidden2 is Variable……………………………………………………………………………………………61
Graph 5.7 Comparison Kernel VS ARM CPU, BZ=16 WI=4, numPatterns_max is Variable…………………………………………………………….62
Graph 5.8 Comparison Kernel VS ARM CPU, BZ=16 WI=4, numHidden1 is Variable……………………………………………………………………………………………62
Graph 5.9 Comparison Kernel VS ARM CPU, BZ=16 WI=4, numHidden1 is Variable……………………………………………………………………………………………63
Graph 5.10 Comparison Kernel with 'tanh' VS ARM CPU, BZ=4 WI=4, numPatterns_max is Variable……………………………………………………………63
Graph 5.11 Comparison Kernel with 'tanh' VS ARM CPU, BZ=4 WI=4, numHidden1 is Variable……………………………………………………………………..64
Graph 5.12 Comparison Kernel with 'tanh' VS ARM CPU, BZ=4 WI=4, numHidden2 is Variable……………………………………………………………………..64
Graph 5.13 Comparison Kernel with 'tanh' VS ARM CPU, BZ=8 WI=8, numPatterns_max is Variable……………………………………………………………65
Graph 5.14 Comparison Kernel with 'tanh' VS ARM CPU, BZ=8 WI=8, numHidden1 is Variable……………………………………………………………………65
Graph 5.15 Comparison Kernel with 'tanh' VS ARM CPU, BZ=8 WI=8, numHidden2 is Variable……………………………………………………………………66
Graph 5.16 Comparison Kernel with 'tanh' VS ARM CPU, BZ=16 WI=4, numPatterns_max is Variable……………………………………………………………66
Graph 5.17 Comparison Kernel with 'tanh' VS ARM CPU, BZ=16 WI=4, numHidden1 is Variable……………………………………………………………………67
Graph 5.18 Comparison Kernel with 'tanh' VS ARM CPU, BZ=16 WI=4, numHidden2 is Variable……………………………………………………………………67

# INDEX OF tables

Table 1.3 Memory Region - Allocation and Memory Access Capabilities…08
Table 5.1 Sample Result of ARM CPU…………………………………………………..54
Table 5.2 Sample Result of Kernel, BZ=4 and WI=4……………………………..….54
Table 5.3 Sample Result of Kernel with 'Tanh', BZ=4 and WI=4………….…..54
Table 5.4 Sample Result of Kernel, BZ=8 and WI=8………………………………..55
Table 5.5 Sample Result of Kernel with 'Tanh', BZ=8 and WI=8……………..55
Table 5.6 Sample Result of Kernel with, BZ=16 and WI=4……………………….56
Table 5.7 Sample Result of Kernel with 'Tanh', BZ=16 and WI=4…………….56
Table 5.8 Hardware Comparision, Just a Kernel……………………………………..57
Table 5.9 Hardware Comparison. Kernel with 'tanh'………………………………58

# Introduction

In our project we will introduce a new programming language for hardware which is OpenCL, OpenCL is an industry standard framework for programming computers composed of a combination of CPUs, GPUs, and other processors. These so-called heterogeneous systems have become an important class of platforms, and OpenCL is the first industry standard that directly addresses their needs. First released in December of 2008 with early products available in the fall of 2009, OpenCL is a relatively new technology.

*Justification*

The OpenCL with Altera SDK has many advantages for hardware programing in our life, it is good industry challenges such as power efficient acceleration, FPGA lifecycle over 15 years, allows for streaming IO channels and kernel channels and shared virtual memory, also OpenCL flow abstracts away FPGA hardware flow bringing the FPGA to low level software programmers and the OpenCL optimization doesn't require a board. It fits all markets like medical, military, automotive, industrial and broadcast.

*Objectives*

The aim goal of our project are:

- Know and understand the main construction and structure of OpenCL.
- Introduce for the first time the flow of design of OpenCL with FPGAs.
- We will do a study about the algorithm Matrix-multiplication on deferent hardware configurations by using OpenCL programing and we see the faster case, we will improve the performance by using the Kernel function in FPGA with embedded ARM CPU, first we will run our algorithm on CPU ARM and this a simple way, than we add kernel and we run our algorithm on that kernel.
- We demonstrate that the use of HDL IP (RTL code) can improve the results of standard kernel, our experiment consists a introduce the nonlinear function IP (tanh) in a generic kernel (matrix-

multiplication), we add new technic it called Hyperbolic Tangent 'tanh' to our kernel and we run our algorithm on it.

- We represent the results of application of this kernel (matrix-multiplication) in a very important algorithm in training of neural networks: Backpropagation algorithm.

*Structure of TFM*

In our thesis we will give a description of new programing OpenCL in first chapter as general and in the second chapter the OpenCL with Altera SDK, so we talk about the structure of OpenCL. Than we will study an example of implementation and optimization of the matrix-multiplication kernel cod also the host code that execute in the kernel in chapter three, in chapter four we will talk about the application of matrix-multiplication in Backpropagation algorithm, and finally we describe of 'tanh' IP and present the result that we collect in our study.

# Chapter 1. Conceptual Foundations of OpenCL

We can define OpenCL by the following models:

- **Platform model**: a high-level description of the heterogeneous system
- **Execution model**: an abstract representation of how streams of instructions execute on the heterogeneous platform
- **Memory model**: the collection of memory regions within OpenCL and how they interact during an OpenCL computation
- **Programming models**: the high-level abstractions a programmer uses when designing algorithms to implement an application

## 1.1 Platform Model

An OpenCL platform always includes a single host. The host interacts with the environment external to the OpenCL program, including I/O or interaction with a program's user. This model is shown in Figure below:

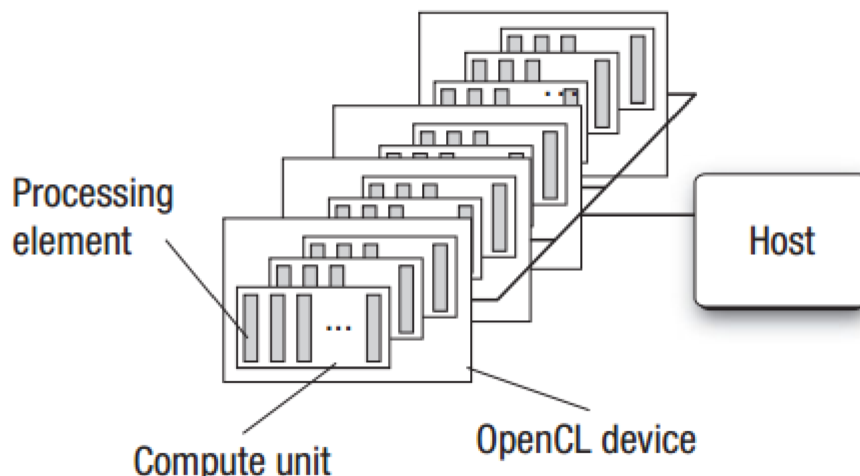



***Figure 1.1*** *OpenCL Platform*

The host is connected to one or more OpenCL devices. A device can be a CPU, a GPU, a DSP, or any other processor provided by the hardware and supported by the OpenCL vendor. The OpenCL devices are further divided into compute units which are further divided into one or more processing elements.

## 1.2 Execution Model

An OpenCL application consists of two distinct parts: the host program and a collection of one or more kernels. The host program runs on the host. The kernels execute on the OpenCL devices. Kernels are typically simple functions that transform input memory objects into output memory objects.

OpenCL defines two types of kernels:

- **OpenCL kernels:** functions written with the OpenCL C programming language and compiled with the OpenCL compiler
- **Native kernels:** functions created outside of OpenCL and accessed within OpenCL through a function pointer. These functions could be, for example, functions defined in the host source code or exported from a specialized library. And it is an optional functionality within OpenCL.

### 1.2.1 How a Kernel Executes on an OpenCL Device

Execution of an OpenCL program occurs in two parts: kernels that execute on one or more OpenCL devices and a host program that executes on the host. A kernel is defined on the host. The host program issues a command that submits the kernel for execution on an OpenCL device. When this command is issued by the host, the OpenCL runtime system creates an integer index space. An instance of the kernel executes for each point in this index space. We call each instance of an executing kernel a work-item, which is identified by its coordinates in the index space. These coordinates are the global ID for the work-item.

The command that submits a kernel for execution, therefore, creates a collection of work-items. Work-items are organized into work-groups. Work-groups are the same size in corresponding dimensions, and this size evenly divides the global size in each dimension. The work-items in a given work-group execute concurrently on the processing elements of a single compute unit.

The index space spans an N-dimensioned range of values and thus is called an NDRange (N can be 1, 2 or 3). Each work-item's global and local ID is an N-dimensional tuple. Work-groups are assigned IDs using a similar approach to that used for work-items.

In figure 2.1 shown a 2D NDRange, each small square is a work-item. Let's us define that uppercase letter $G_{x/y}$ is the size of the index space in each dimension and lowercase letter $g_{x/y}$ is the global ID of a work-item in each dimension, and uppercase letter $W_{x/y}$ is the number of work-groups in each dimension and lowercase letter $w_{x/y}$ the work-group ID.

OpenCL requires that the number of work-groups in each dimension evenly divide the size of the NDRange index space in each dimension. We will refer to this index space inside a work-group as the local index space. The size of our local index space in each dimension (x and y) is indicated with an uppercase L and the local ID inside a work-group uses a lowercase l.

So we get this results:

- $L_{x/y} = G_{x/y} / W_{x/y}$
- $g_{x/y} = w_{x/y} * L_{x/y} + l_{x/y}$ (the index space starts with a zero in each dimension)

From the figure 2.1 and we use the default offset of zero in each dimension. The shaded block has a global ID of $(g_x, g_y) = (6, 5)$ and a work-group plus local ID of $(w_x, w_y) = (1, 1)$ and $(l_x, l_y) =(2, 1)$.

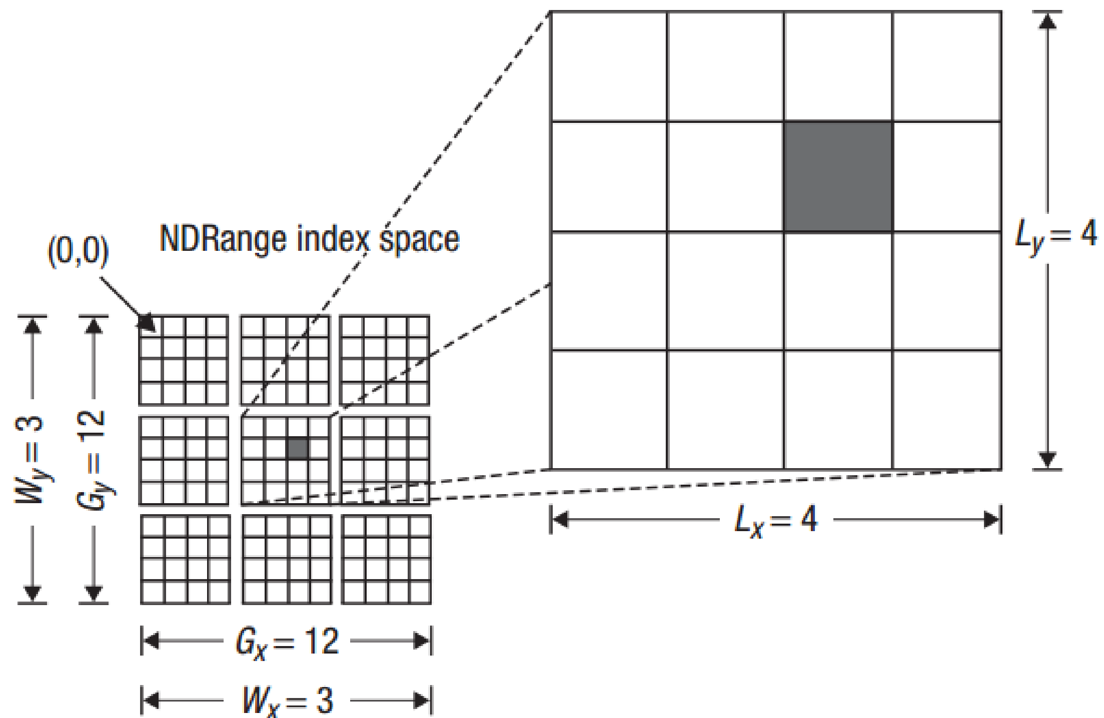



***Figure 1.2*** *An example of how the global IDs, local IDs, and work-group indices are related for a two-dimensional NDRange*

## 1.2.2 Context

The computational work of an OpenCL application takes place on the OpenCL devices. The host defines a context for the execution of the kernels. The context includes the following resources:

- Devices: the collection of OpenCL devices to be used by the host
- Kernels: the OpenCL functions that run on OpenCL devices
- Program objects: the program source code and executables that implement the kernels
- Memory objects: a set of objects in memory that are visible to OpenCL devices and contain values that can be operated on by instances of a kernel

The context is created and manipulated by the host using functions from the OpenCL API. The context included one or more program objects that contain the code for the kernels.

### 1.2.3 Command-Queues

The interaction between the host and the OpenCL devices occurs through commands posted by the host to the command-queue. A command-queue is created by the host and attached to a single OpenCL device. OpenCL supports three types of commands:

• Kernel execution commands execute a kernel on the processing elements of an OpenCL device.

• Memory commands transfer data between the host and different memory objects, move data between memory objects, or map and unmap memory objects from the host address space.

• Synchronization commands put constraints on the order in which commands execute.

In a typical host program, the programmer defines the context and the command-queues, defines memory and program objects, and builds any data structures needed on the host to support the application. When multiple kernels are submitted to the queue, they may need to interact.

Commands within a single queue execute relative to each other in one of two modes:

• **In-order execution:** a prior command on the queue completes before the following command begins.

• **Out-of-order execution:** Commands are issued in order but do not wait to complete before the following commands execute.

All OpenCL platforms support the in-order mode, but the out-of-order mode is optional. And it is possible to associate multiple queues with a single context for any of the OpenCL devices within that context.

## 1.3 Memory Model

OpenCL defines two types of memory objects: **buffer objects** and **image objects**. A buffer object, is just a contiguous block of memory made available to the kernels. A programmer can map data structures onto this buffer and access the buffer through pointers. Image objects are restricted to holding images. An image storage format may be optimized to the needs of a specific OpenCL device.

The OpenCL memory model defines five distinct memory regions:

• **Host memory:** This memory region is visible only to the host.

• **Global memory:** This memory region permits read/write access to all work-items in all work-groups.

• **Constant memory:** This memory region of global memory remains constant during the execution of a kernel.

• **Local memory:** This memory region is local to a work-group.

• **Private memory:** This region of memory is private to a work-item.

Table describes whether the kernel or the host can allocate from a memory region, and the type of access allowed.

| | Global | Constant | Local | Private |
|---|---|---|---|---|
| **Host** | Dynamic allocation<br><br>Read / Write access | Dynamic allocation<br><br>Read / Write access | Dynamic allocation<br><br>No access | No allocation<br><br>No access |
| **Kernel** | No allocation<br><br>Read / Write access | Static allocation<br><br>Read-only access | Static allocation<br><br>Read / Write access | Static allocation<br><br>Read / Write access |

***Table 1.3*** *Memory Region - Allocation and Memory Access Capabilites*

The memory regions and how they relate to the platform model are described in figure below.

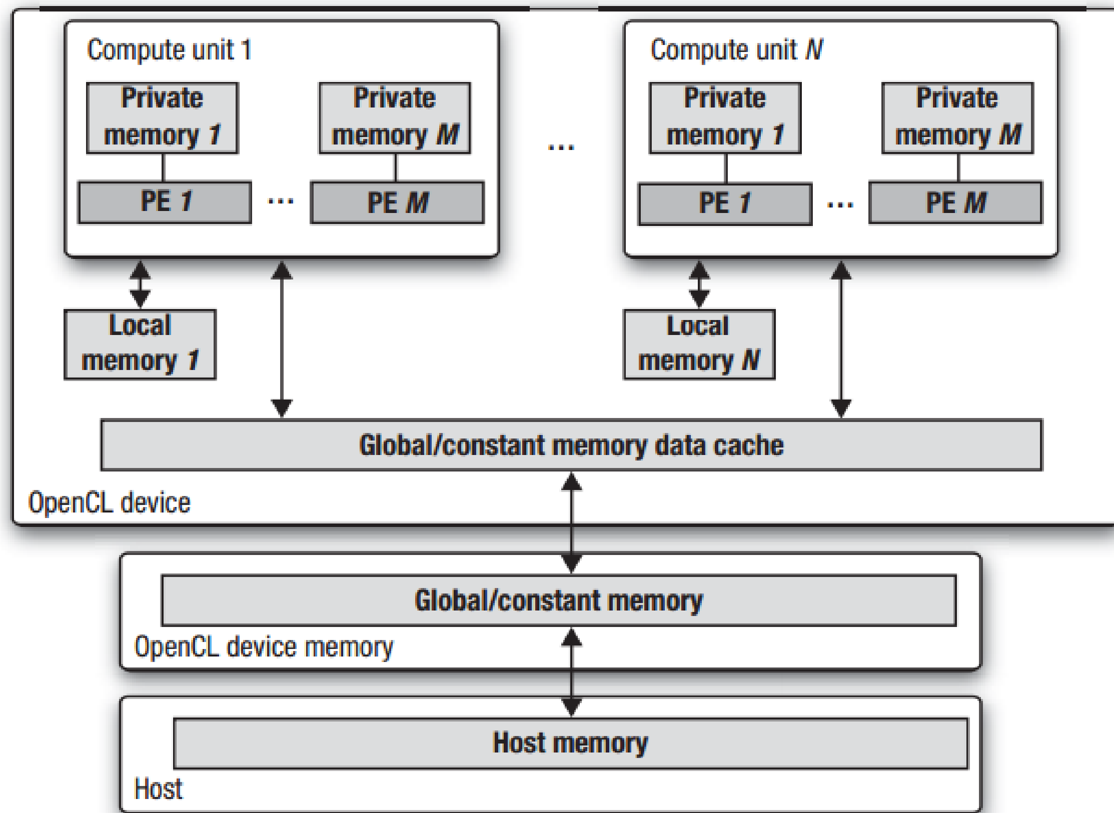



***Figure 1.3*** *The Memory Model in OpenCL*

## 1.4 Programming Models

Programming Models are intimately connected to how programmers reason about their algorithms. OpenCL was defined with two different programming models in mind: task parallelism and data parallelism.

- **Data-Parallel Programming Model:** A data parallel programming model defines a computation in terms of a sequence of instructions applied to multiple elements of a memory object. The index space associated with the OpenCL execution model defines the work-items and how the data maps onto the work-items.

- **Task-Parallel Programming Model**: The OpenCL task parallel programming model defines a model in which a single instance of a kernel is executed independent of any index space.

- **Other Programming Models**

A programmer is free to combine OpenCL's programming models to create a range of hybrid programming models.

## 1.5 The OpenCL Framework

The OpenCL framework is divided into the following components:

• **OpenCL platform API:** The platform API defines functions used by the host program to discover OpenCL devices and their capabilities

• **OpenCL runtime API:** This API manipulates the context to create command-queues and other operations that occur at runtime.

• **The OpenCL programming language:** This is the programming language used to write the code for kernels.

### 1.5.1 Platform API

The term *platform* has a very specific meaning in OpenCL. It refers to a particular combination of the host, the OpenCL devices, and the OpenCL framework. Multiple OpenCL platforms can exist on a single heterogeneous computer at one time.

### 1.5.2 Runtime API

The tasks of the runtime API are:

- To set up the command-queues.
- When the command-queues in place, the runtime API is used to define memory objects.
- Managed by the runtime API is to create the program objects used to build the dynamic libraries from which kernels are defined. The program objects, the compiler to compile them, the definition of the kernels.
- Issues the commands that interact with the command-queue.
- Synchronization points for managing data sharing and to enforce constraints on the execution of kernels

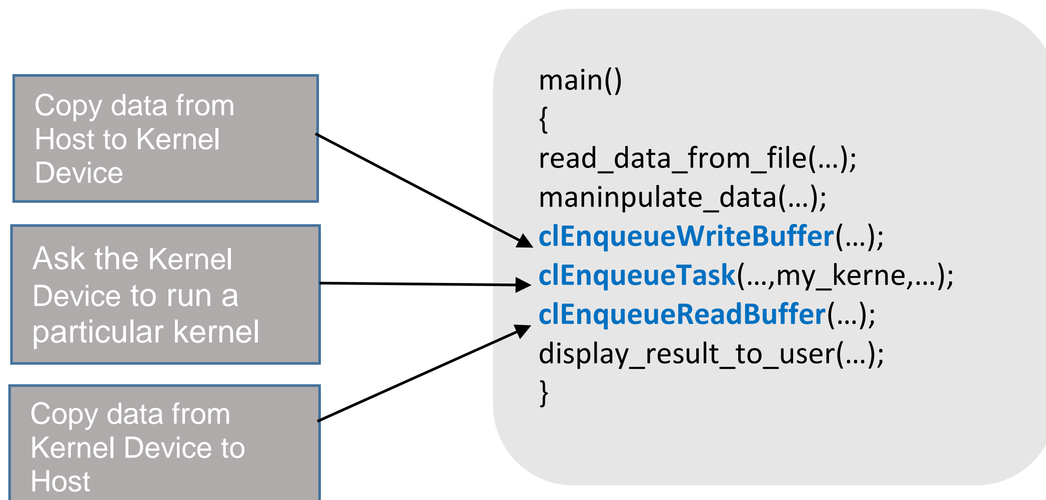
### 1.5.3 Kernel Programming Language

OpenCL consists two program, the OpenCL Host Program and the OpenCL Kernel, The host program is very important, but it is the kernels that do the real work in OpenCL.

❖ **The OpenCL Host Program**

Pure software written in standard 'C'. Communicates with the Kernel Device via a set of library routines which abstract the communication between the host processor and the kernels.

*Example:*




❖ **The OpenCL Kernel**

The kernel programming language in OpenCL is called the OpenCL C programming language. It is derived from the ISO C99 language.

Language Features Added like Work-items and work-groups, Vector types, Synchronization and Address space qualifiers. Also includes a large set of built-in functions like Image manipulation, Work-item manipulation and Math functions.

The OpenCL working group has already approved many extensions to the OpenCL specification:

—Double precision floating-point types
—Built-in functions to support doubles
—Atomic functions
—Byte-addressable stores (write to pointers to types < 32-bits)
—3D Image writes
—Built-in functions to support half types

OpenCL Language Restrictions:

- Pointers to functions are not allowed
- Pointers to pointers allowed within a kernel, but not as an argument
- Bit-fields are not supported
- Variable-length arrays and structures are not supported
- Recursion is not supported
- Writes to a pointer to a type less than 32 bits are not supported
- Double types are not supported, but reserved
- 3D Image writes are not supported

### ❖ The BIG idea behind OpenCL

Replace loops with functions (a kernel) executing at each point in a problem domain, define N-dimensional computation domain and execute a *kernel* at each point in computation domain.

The following example, we have two codes, the left code with traditional loop as a function in C and the right code with OpenCL C kernel. We can see the different in the declaration of function and the variables, and the most important one is the loop.

```
Void
trad_mul(int n,
         const float *a,
         const float *b,
         float *c)
{
   int i;
   for (i=0; i<n; i++)
      c[i] = a[i] * b[i];
}
```

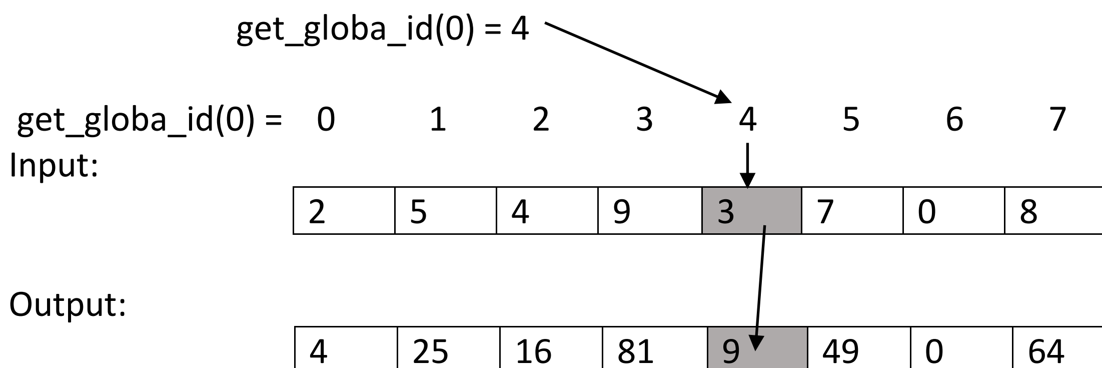
```
__kernel void
dp_mul(__global const float *a,
       __global const float *b,
       __global  float *c)
{
int id = get_global_id(0);
   c[id] = a[id] * b[id];}

 // execute over n"work items"
```

Traditional loop as a function in C OpenCL C kernel

*Example:*

We will give an example for a kernel function, the Output is square of Input, so we have the following code:

```
_kerner void square(_global float* input, _global float* output)
{
        int i = get_globa_id(0) ;
        output[i]  = input[i] * input[i] ;
}
```



## 1.6 Opencl Overview

In figure we can see general block diagram of OpenCL. First, the host program that defines the context. The context contains two OpenCL devices, a CPU and a GPU. Next we define the command-queues. In this case we have two queues, an in-order command-queue for the GPU and an out-of-order command-queue for the CPU. The host program then defines a program object that is compiled to generate kernels for both OpenCL devices (the CPU and the GPU). Next the host program defines any memory objects required by the program and maps them onto the arguments of the kernels. Finally, the host program enquires commands to the command-queues to execute the kernels.

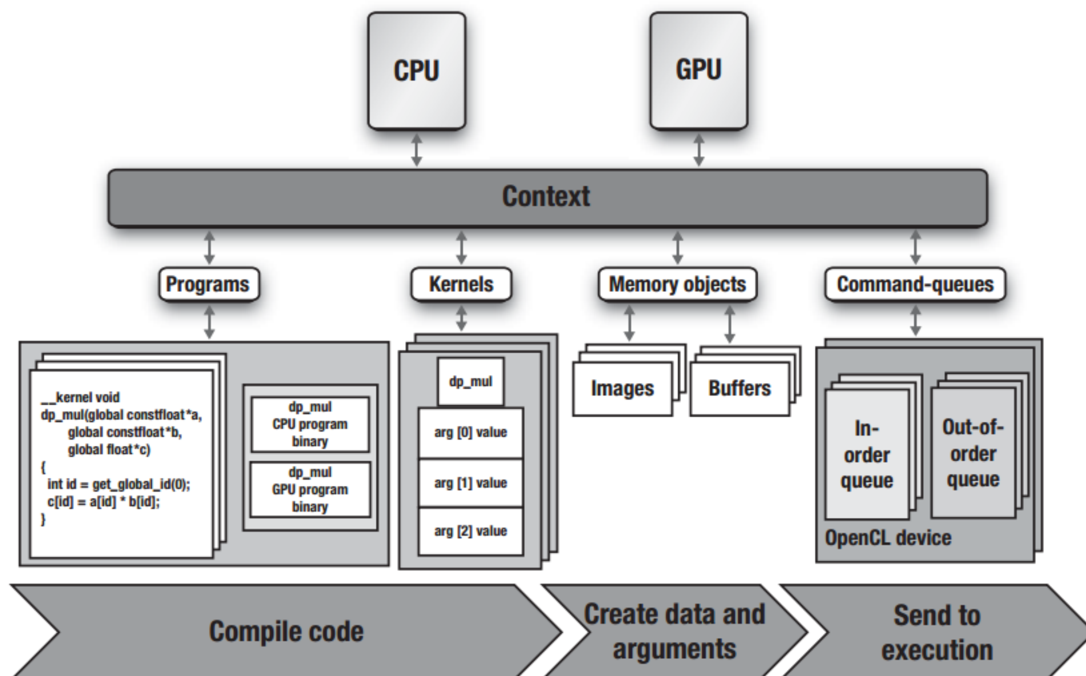



***Figure 1.6*** *general block diagram of OpenCL*

# Chapter 2. Conceptual Foundations of OpenCL with Altera SDK

Utilizing the OpenCL standard on an FPGA may offer significantly higher performance than other hardware architectures such as CPU, GPU and DSP. Moreover, an FPGA-based heterogeneous system (CPU + FPGA) using the OpenCL standard.

In Figure 02 we see the programmable technologies, is represented by CPU, DSP, Multicores and Arrays.

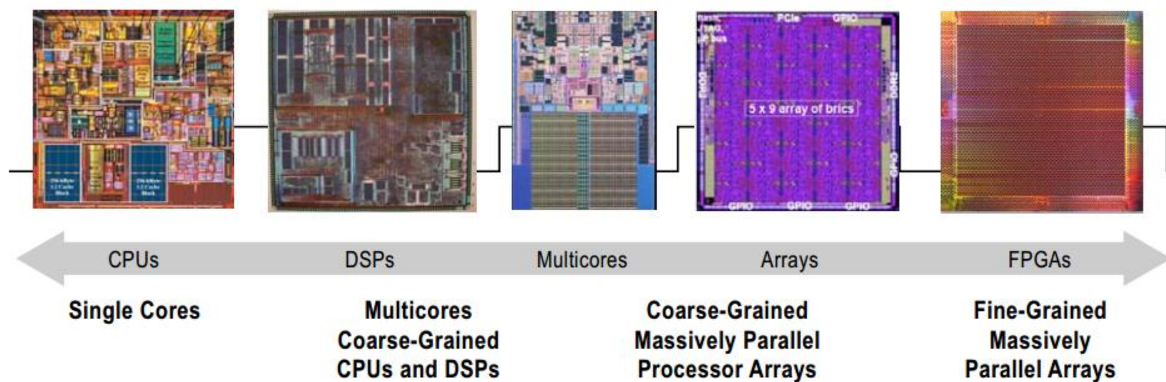



***Figure 02*** *Trend of Programmable and Parallel Technology*

## 2.1 The openCL stander on FPGA

As we mention on chapter 1, the OpenCL application has two part, one part is OpenCL host program, is a pure software routine written in standard C/C++ that runs on any sort of microprocessor. That processor may be, an embedded soft processor in an FPGA, a hard ARM processor, or an external x86 processor. And second part is OpenCL Kernels program is written in standard C that make our kernel circuit on FPGA.

In Figure 2.1.1 we provide an overview.

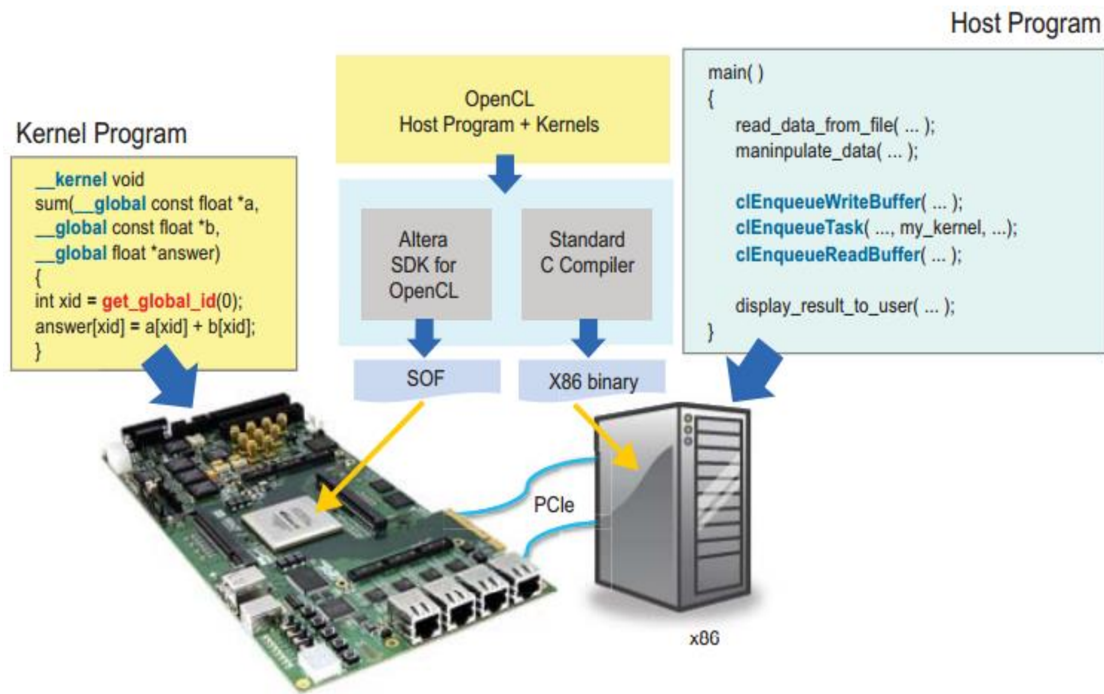



***Figure 2.1.1*** *overview of OpenCL*

In figure 2.1.2 we have an example shown that circuit architecture to perform the vector addition of two arrays, a and b.

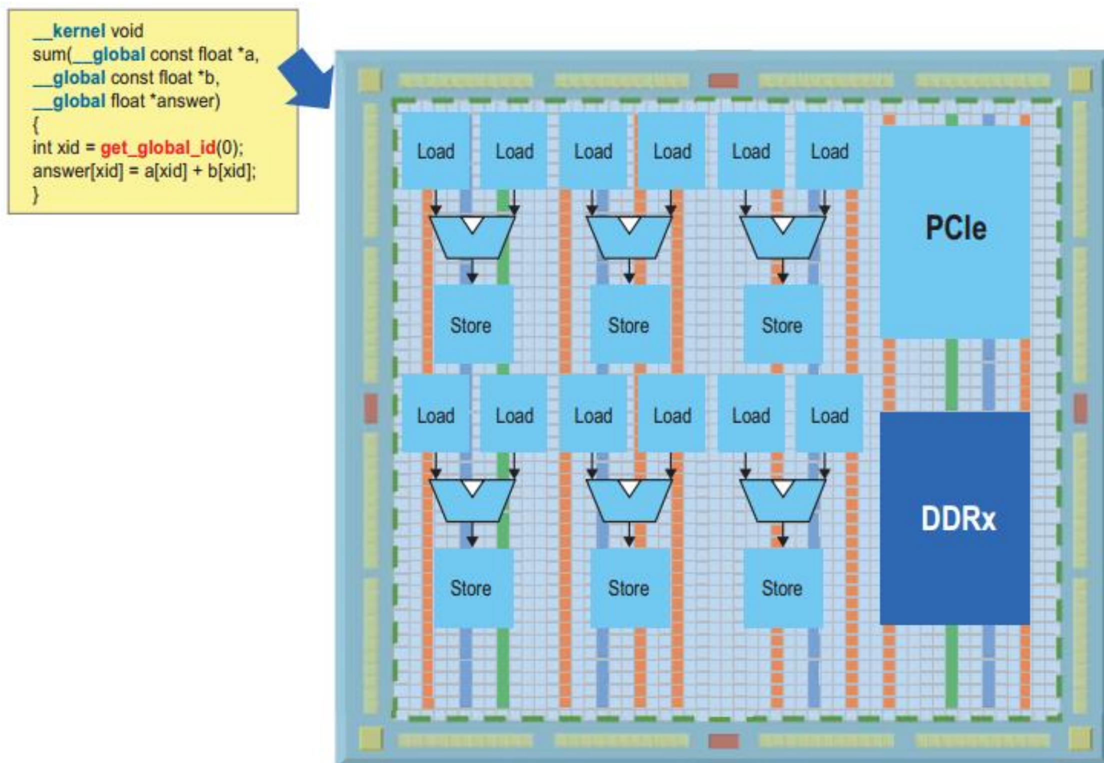



***Figure 2.1.2*** *Example of OpenCL Implementation on an FPGA*

OpenCL Compiler translates an OpenCL kernel to hardware by creating a circuit that implements each operation. These circuits are wired together to mimic the flow of data in the kernel. For example in victor addition, The loads from arrays A and B are converted into load units, which are small circuits responsible for issuing addresses to external memory and processing the returned data. The two returned values are fed directly into an adder unit responsible for calculating the floating-point addition of these two values. Finally, the result of the adder is wired directly to a store unit that writes the sum back to external memory.

The most important concept behind the OpenCL-to-FPGA compiler is the notion of pipeline parallelism. For simplicity, in figure 2.1.3 we have a circuit has three pipeline stages for the kernel. On the first clock cycle, thread 0 is clocked into the two load units. This indicates that they should begin fetching the first elements of data from arrays A and B. On the second clock cycle, thread 1 is clocked in at the same time that thread 0 has completed its read from memory and stored the results in the registers following the load units. On cycle 3, thread 2 is clocked in, thread 1 captures its returned data, and thread 0 stores the sum of the two values that it loaded. It is evident that in the steady state, all parts of the pipeline are active, with each stage processing a different thread.

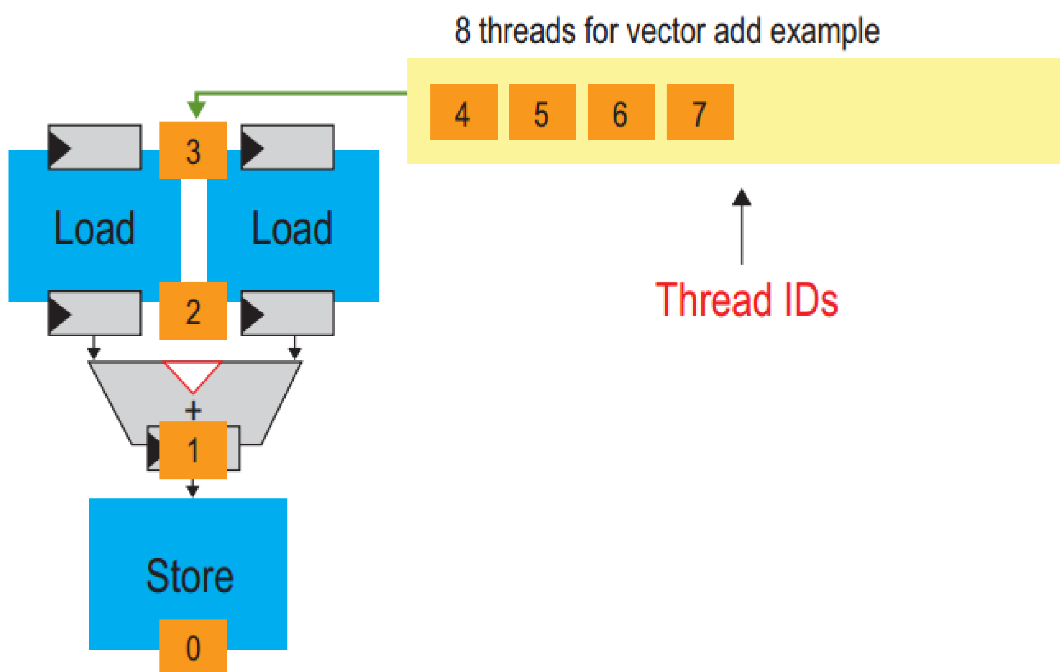



***Figure 2.1.3*** *pipelined information*

## 2.2 FPGA OpenCL Architecture

The figure 2.2 represents the high level of a complete OpenCL system containing multiple kernel pipelines and circuitry connecting these pipelines to off-chip data interfaces. In addition to the kernel pipeline, Altera's OpenCL compiler creates interfaces to external and internal memory. The load and store units for each pipeline are connected to external memory via a global interconnect structure that arbitrates multiple requests to a group of DDR DIMMs. Similarly, OpenCL local memory accesses are connected through a specialized interconnect structure to on-chip M9K RAMs. These specialized interconnect structures are designed to ensure high operating frequency and efficient organization of requests to memory.

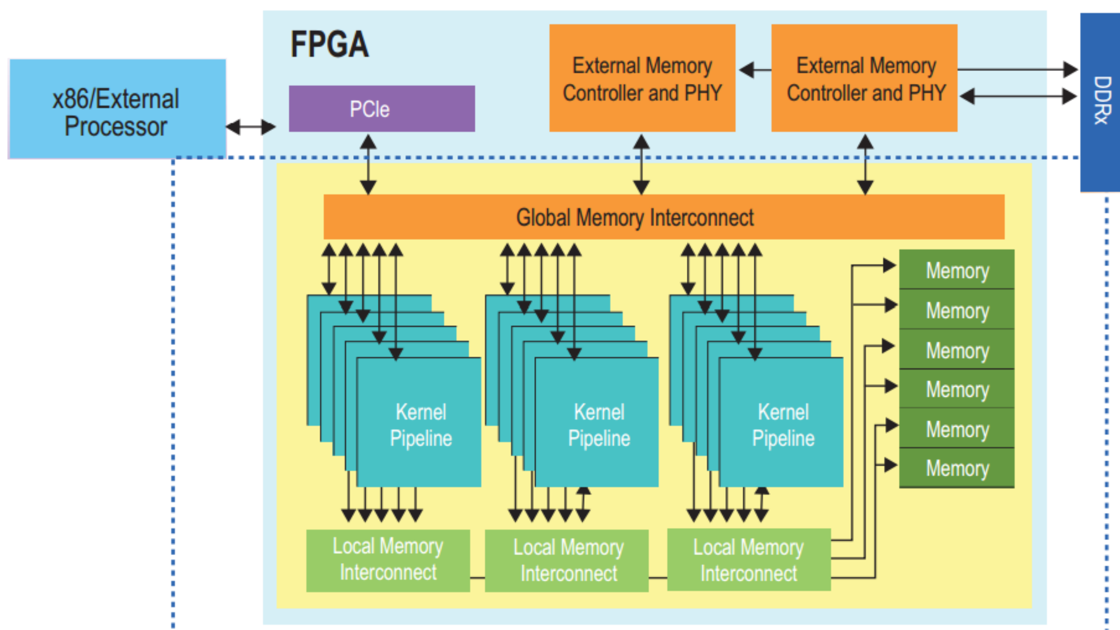



***Figure 2.2*** *OpenCL system implementation*

## 2.3 OpenCL-to-FPGA framework

We define the OpenCL-to-FPGA framework in the following schematic:

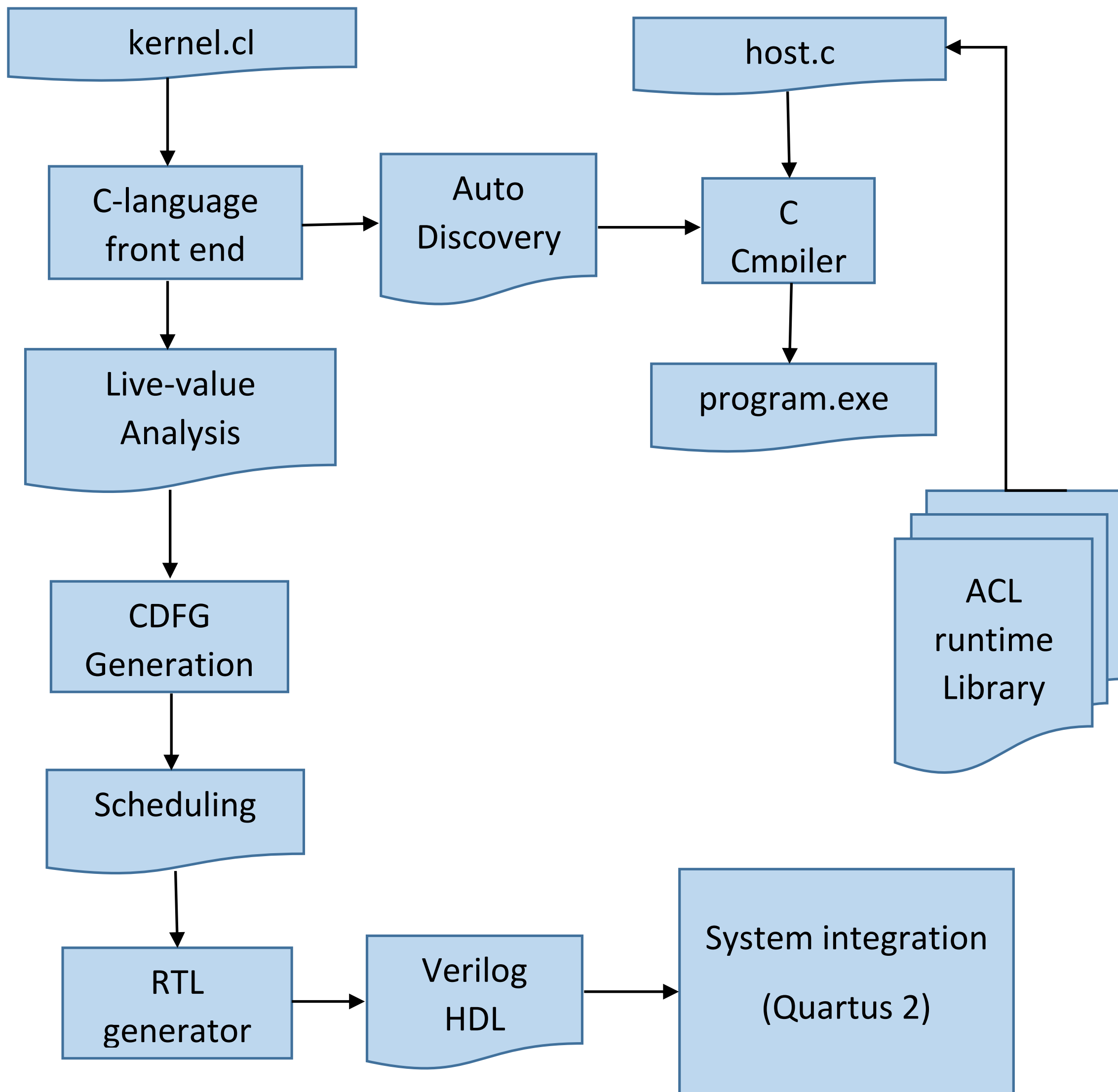



***Figure 2.3*** *OpenCL-to-FPGA framework*

The schematic presents the flow of our compilation framework, based on an LLVM compiler infrastructure. The input is an OpenCL application comprising a set of kernels (.cl files) and a host program (.c file). The kernels are compiled into a hardware circuit, starting with a C-language parser that produces an intermediate representation for each kernel. The intermediate representation (LLVM IR) is in the form of instructions and dependencies

between them. This representation is then optimized to target an FPGA platform. An optimized LLVM IR is then converted into a Control-Data Flow Graph (CDFG), which can be optimized to improve area and performance of the system, prior to RTL generation that produces Verilog HDL for a kernel.

For the host program, we compile it using a C/C++ compiler. There are two elements in the compilation of the host program. One is the Altera OpenCL (ACL) Host Library, which implements OpenCL function calls that allow the host program to exchange information with kernels on an FPGA. The second is the Auto-Discovery module which allows a host program to detect the types of kernels on an FPGA.

## 2.4 Kernel Compiler

To compile OpenCL kernels into a hardware circuit, we extended the LLVM Open-Source compiler to target an FPGA platform as shown in schematic. The LLVM compiler represents a program as a sequence of instructions, such as load, add, subtract, store. A group of instructions in a contiguous sequence constitutes a basic block. At the end of a basic block there is always a terminal instruction that either ends the program or redirects execution to another basic block. The compiler uses this representation to create a hardware implementation of each basic block, which are then put together to form the complete kernel circuit.

There are six basic group for kernel compiler, and they are:

- **C-Language Front-End:**

The first step in the conversion of a high-level description to a hardware circuit is to produce an intermediate representation (IR). To illustrate the IR, consider a program in following example kernel code :

```
__kernel void triangle(_global int *x, _global int *y) {
        int i, t = get global id(0), sum=0;
        for (i=0; i < t; i++) sum += x[i];
        y[id] = sum;
}
```

In this example, each thread reads its ID using the *get_global_id(0)* function and stores it in variable t. It then sums up all elements of array x beginning at the first and ending at t-1. Finally, the result is stored in array y. C-Language front-end parses a kernel description and creates an LLVM Intermediate Representation (IR), which is based on static single assignment. It comprises basic blocks connected by control-flow edges as shown in Figure2.4.1. The first basic block, Entry, performs initialization for the kernel and ends with a branch instruction that decides if a thread should bypass the loop. The second basic block represents the loop body and the last basic block stores the result to memory. To determine the data each basic block consumes and produces, we perform Live Variable Analysis.

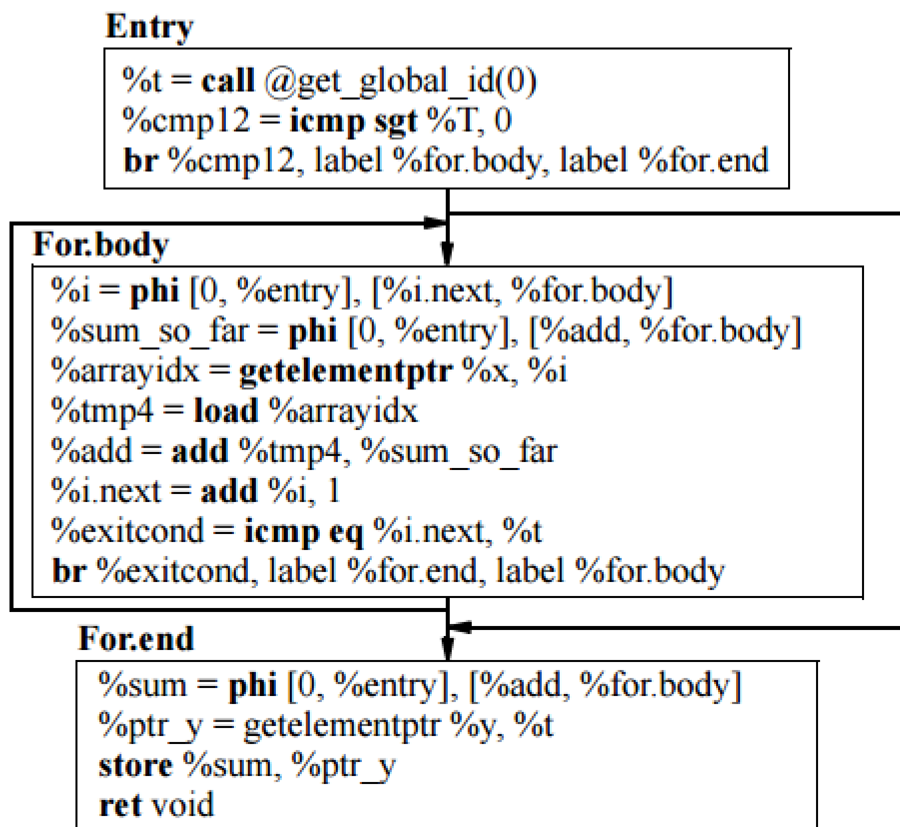



***Figure 2.4.1** basic blocks*

- **Live Variable Analysis**

Live Variable Analysis identifies variables consumed and produced by each basic block. In our example, the Entry basic block contains only kernel arguments as input variables (x, y). At the output of the basic block, variables sum, t and i are also created. This tells us that each thread produces these values when it completes execution in this basic block.

The For.body basic block includes all kernel arguments as well as the three arguments produced by the first basic block. It then produces y, t, i.next and add as output live variables. Notice that i.next and add effectively replace i and sum when the basic block loops back to itself, allowing the loop to function correctly. Finally, the last basic block has input live variables y, t and add, while no variables are live after the return instruction.

- **CDFG Generation**

Once each basic block is analyzed, we create a Control-Data Flow Graph (CDFG) to represent the operations inside it. Each basic block module takes inputs either from kernel arguments or another basic block, based on the results of Live Variable Analysis. Each basic block then processes the data according to the instructions contained within it and produces output that can be read by other basic blocks.

A basic block module, shown in Figure 2.4.2, consists of three types of nodes. The first node is the merge node, which is responsible for aggregating data from previously executed basic blocks. This ensures that for each thread, its id as well as all other live variables are valid when the execution of the basic block begins.

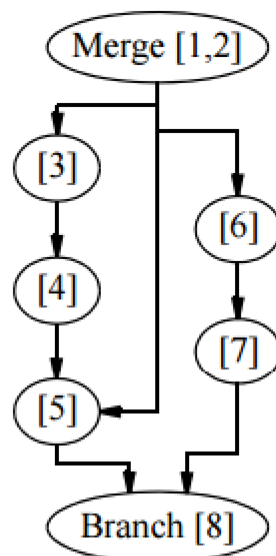



***Figure 2.4.2*** *basic block module*

Operational nodes represent instructions that a basic block needs to execute, such as load, store, or add. They are linked by edges to other nodes to show where their inputs come from and where their outputs are used. Each operational node can be independently stalled when the successor node is unable to accept data, or not all inputs of the successor node are ready.

The last node in a basic block module is a branch node. It decides which of the successor basic blocks a thread should proceed to.

- **Loop Handling**

Loops are handled at a basic block level. A simple example of a loop is a basic block whose output is also an input to it, such as shown in Figure 2.4.1. The loop itself presents a problem in that it is entirely possible that a loop can stall. To remedy the problem, we insert an additional stage of registers into the merge node, that allows the pipeline to have an additional spot into which data can be placed.

When loops comprise many basic blocks, it is possible that stalling can occur when loop-back paths are unbalanced. In such cases, we instantiate a loop limiter that allows only specific number of threads to enter the loop. The number of threads is equal to the length of the shortest path in a loop.

- **Scheduling**

Once each basic block is represented as a CDFG, scheduling is used to determine the clock cycles in which each operation is performed. This is important because not all instructions require the same number of clock cycles to complete. For example, an AND operation may be purely combinational, but a floating point addition may take eight cycles. In some cases, it may be necessary to insert pipeline balancing registers into the circuit because one execution path is longer than another.

To solve the scheduling problem we apply SDC(system of difference constraints ) scheduling algorithm. The SDC scheduler uses a system of linear equations to schedule operations, while minimizing a cost function. In the context of scheduling, each equation represents a clock cycle relationship between connected operations. For example, in implementing an equation $f = a * b + c * d$, the scheduler has ensure that both multiplications occur before addition. A secondary objective is the reduction of area, and in particular the amount of pipeline balancing

registers required. To minimize the impact on area, we minimize a cost function that reduces the number of bits required by the pipeline.

- **Hardware Generation**

To generate a hardware circuit for a kernel we build it out of basic block modules. To achieve high performance, we implement each module as a pipelined circuit, rather than a finite state machine with datapath (FSMD). This is because a potentially large number of threads need to execute using a kernel hardware, and their computation is largely independent. Hence, the kernel hardware should be able to execute many threads at once, rather than one at a time.

In a pipelined circuit, a new thread begins execution at each clock cycle. Thus, a basic block with pipeline depth of 100 executes 100 threads simultaneously. This is similar to replicating an FSMD circuit 100 times, except that subsequent threads execute different operations. Once each basic block is implemented, we put the basic blocks together by linking the stall, valid and data signals as specified by the control edge. We then generate a wrapper around a kernel to provide a standard interface to the rest of the system.

## 2.5 Memory Organization

OpenCL defines three types of memory spaces: global, local and private. The global memory space is designated for access by all threads. Read and write operations to this memory can be performed by any thread. The global memory space resides in off-chip DRRx memory. It has large capacity allowing us to store data, but long access latency. Accesses to this memory are coalesced when possible to increase memory throughput.

Local memory is used by work groups to enable synchronized exchange of data. To synchronize threads within a workgroup, barriers/memory fences are used to force threads to wait on one another before proceeding further. This allows complex algorithms that require collaboration of threads to be implemented. Local memory is implemented using on-chip memory. It has short latency and multiple ports, allowing the kernel to access it efficiently. To do this we create a shared memory space for each load and store unit to any local memory.

Private memory is implemented with registers that store the data on a per-thread basis, and are pipelined through the kernel to ensure that each thread keeps the data it requires as it proceeds with the execution through the kernel.

### 2.5.1 Optimize Global Memory Accesses

The Altera Offline Compiler (AOC) uses SDRAM as global memory. By default, the AOC configures global memory in a burst-interleaved configuration. The AOC interleaves global memory across each of the external memory banks, the default burst-interleaved configuration leads to the best load balancing between the memory banks.

The figure below illustrates the differences in memory mapping patterns between burst-interleaved and non-interleaved memory partitions.

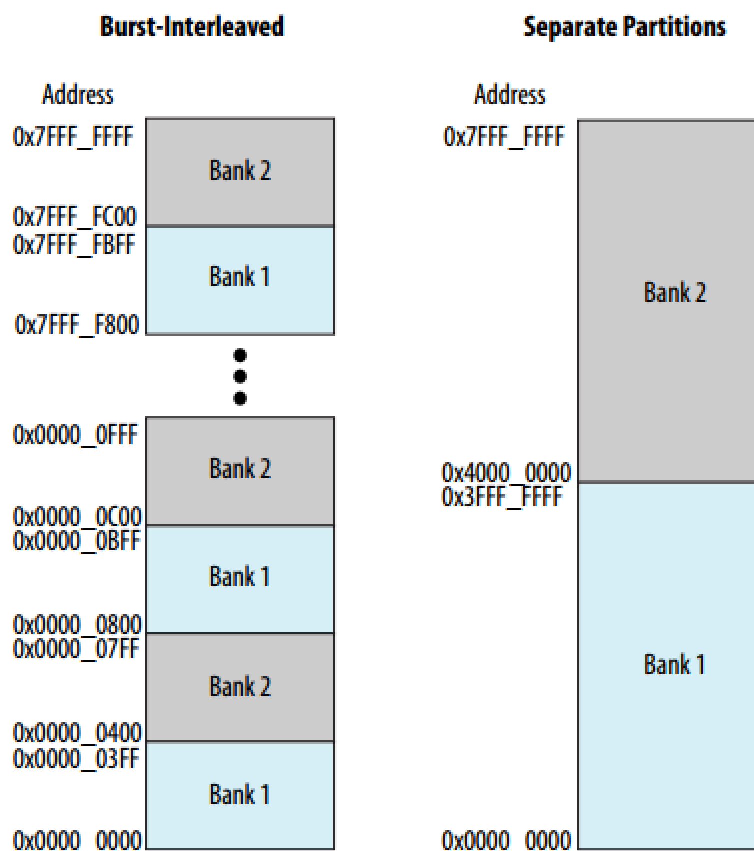



***Figure 2.5.1*** *Global Memory Partitions*

### 2.5.2 Contiguous Memory Accesses

Contiguous memory access optimizations analyze statically the access patterns of global load and store operations in a kernel. For sequential load or store operations that occur for the entire kernel invocation, the Altera Offline Compiler (AOC) directs the kernel to access consecutive locations in global memory. Consider the following code example:

```
__kernel void sum ( __global const float * restrict a,
                    __global const float * restrict b,
                    __global float * restrict c )
{
                    size_t gid = get_global_id(0);
                    c[gid] = a[gid] + b[gid];
}
```

The load operation from array a uses an index that is a direct function of the work-item global ID. By basing the array index on the work-item global ID, the AOC can direct contiguous load operations. These load operations retrieve the data sequentially from the input array, and sends the read data to the pipeline as required. Contiguous store operations then store elements of the result that exits the computation pipeline in sequential locations within global memory.

The following figure illustrates an example of the contiguous memory access optimization:

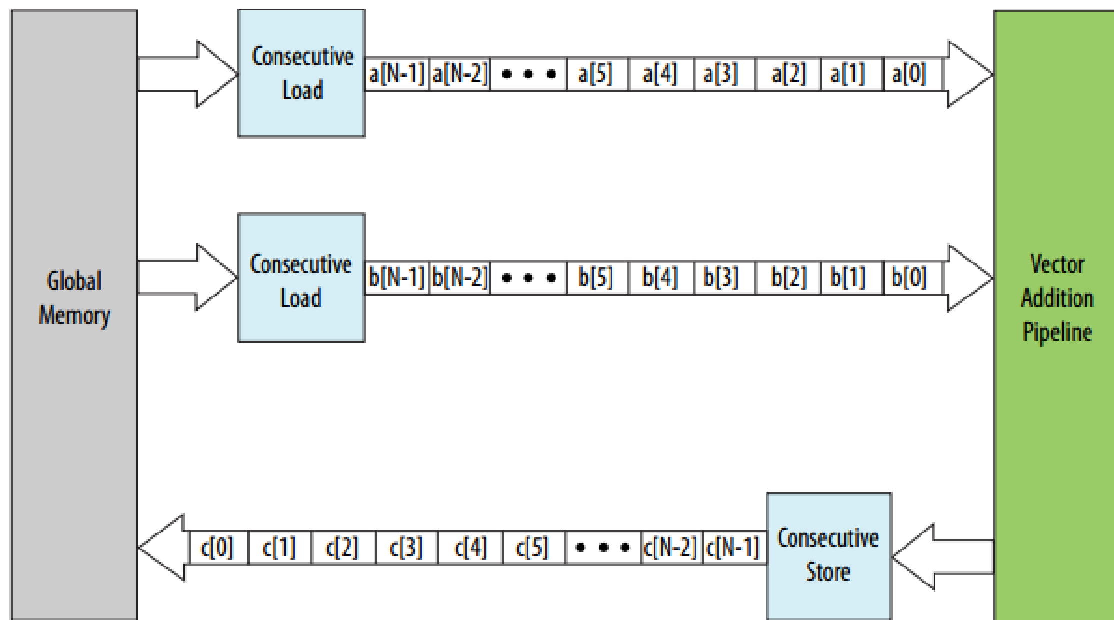



***Figure 2.5.2*** *Contiguous Memory Access*

Contiguous load and store operations improve memory access efficiency because they lead to increased access speeds and reduced hardware resource needs. The data travels in and out of the computational portion of the pipeline concurrently, allowing overlaps between computation and memory accesses. If possible, use work-item IDs that index consecutive memory locations for load and store operations that access global memory. Sequential accesses to global memory increase memory efficiency because they provide an ideal access pattern.

We can execute our kernel on an FPGA board that includes multiple global memory types, such as DDR, quad data rate (QDR), and on-chip RAMs.

### 2.5.3 Constant Cache Memory

Constant memory resides in global memory, but the kernel loads it into an on-chip cache shared by all work-groups at runtime. For example, if we have read-only data that all work-groups use, and the data size of the constant buffer fits into the constant cache, allocate the data to the constant memory. The constant cache is most appropriate for high-bandwidth table lookups that are constant across several invocations of a kernel. The constant cache is optimized for high cache hit performance.

## 2.6 Strategies for Improving NDRange Kernel Data Processing Efficiency

Consider the following kernel code:

```
__kernel void sum ( __global const float * restrict a,
                    __global const float * restrict b,
                    __global float * restrict answer )
{
                    size_t gid = get_global_id(0);
                    answer[gid] = a[gid] + b[gid];
}
```

This kernel adds arrays a and b, one element at a time. Each work-item is responsible for adding two elements, one from each array, and storing the sum into the array *answer*. Without optimization, the kernel performs one addition per work-item.

To maximize the performance of our OpenCL kernel, consider implementing the applicable optimization techniques to improve data processing efficiency.

- **Specify a Maximum Work-Group Size or a Required Work-Group Size**

Specify the *max_work_group_size* or *reqd_work_group_size* attribute for our kernels whenever possible. These attributes allow the Altera Offline Compiler (AOC) to perform aggressive optimizations to match the kernel to hardware resources without any excess logic.

For example, the code fragment below assigns a fixed work-group size of 64 work-items to a kernel:

```
__attribute__((reqd_work_group_size(64,1,1)))
__kernel void sum ( __global const float * restrict a,
                    __global const float * restrict b,
                    __global float * restrict answer )
{
                    size_t gid = get_global_id(0);
                    answer[gid] = a[gid] + b[gid];
}
```

## ➢ Kernel Vectorization

To achieve higher throughput, we can vectorize our kernel. Kernel vectorization allows multiple work-items to execute in a single instruction multiple data (SIMD) fashion.

Include the *num_simd_work_items* attribute in our kernel code to direct the AOC to perform more additions per work-item without modifying the body of the kernel. The following code fragment applies a vectorization factor of four to the original kernel code:

```
__attribute__((num_simd_work_items(4)))
__attribute__((reqd_work_group_size(64,1,1)))
__kernel void sum ( __global const float * restrict a,
                    __global const float * restrict b,
                    __global float * restrict answer )
{
                   size_t gid = get_global_id(0);
                    answer[gid] = a[gid] + b[gid];

}
```

## ➢ Static Memory Coalescing

Static memory coalescing is an Altera Offline Compiler (AOC) optimization step that attempts to reduce the number of times a kernel accesses non-private memory.

In previous code. The OpenCL kernel performs four load operations that access consecutive locations in memory. Instead of performing four memory accesses to competing locations, the AOC coalesces the four loads into a single wider vector load. This optimization reduces the number of accesses to a memory system and potentially leads to better memory access patterns. Although the AOC performs static memory coalescing automatically when it vectorizes the kernel.

## ➢ Multiple Compute Units

To achieve higher throughput, the Altera Offline Compiler (AOC) can generate multiple compute units for each kernel. The AOC implements each

compute unit as a unique pipeline. Generally, each kernel compute unit can execute multiple work-groups simultaneously.

To increase overall kernel throughput, the hardware scheduler in the FPGA dispatches work-groups to additional available compute units. A compute unit is available for work-group assignments as long as it has not reached its full capacity.

Assume each work-group takes the same amount of time to complete its execution. If the AOC implements two compute units, each compute unit executes half of the work-groups. Because the hardware scheduler dispatches the work-groups, you do not need to manage this process in your own code.

The AOC does not automatically determine the optimal number of compute units for a kernel. To increase the number of compute units for our kernel implementation, we must specify the number of compute units that the AOC should create using the *num_compute_units* attribute, as shown in the code sample below.

```
__attribute__((num_compute_units(2)))
__kernel void sum ( __global const float * restrict a,
                    __global const float * restrict b,
                    __global float * restrict answer )
{
                   size_t gid = get_global_id(0);
                    answer[gid] = a[gid] + b[gid];

}
```

- **Combination of Compute Unit Replication and Kernel SIMD Vectorization**

If our replicated or vectorized OpenCL kernel does not fit in the FPGA, we can modify the kernel by both replicating the compute unit and vectorizing the kernel. Include the *num_compute_units* attribute to modify the number

of compute units for the kernel, and include the *num_simd_work_items* attribute to take advantage of kernel vectorization.

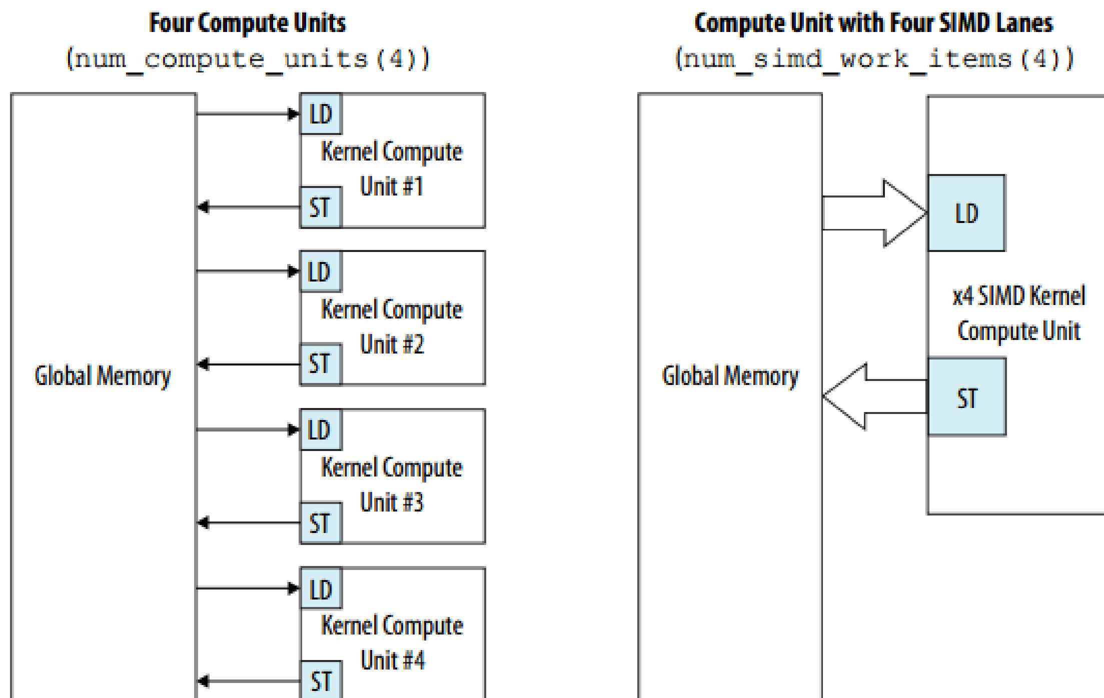



***Figure 2.6.1*** *Compute Unit Replication versus Kernel SIMD Vectorization*

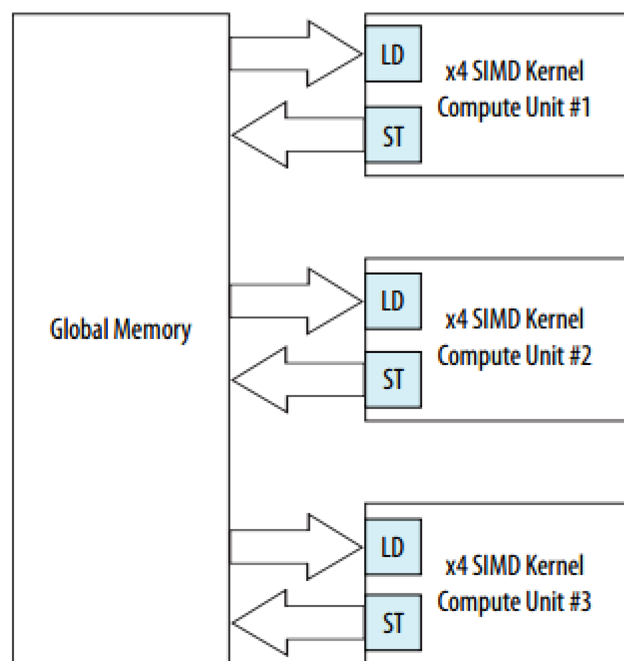



***Figure 2.6.2*** *Combination of Compute Unit Replication and Kernel SIMD Vectorization*

## 2.7 Optimize Floating-Point Operations

For floating-point operations, we can manually direct the Altera Offline Compiler (AOC) to perform optimizations that create more efficient pipeline structures in hardware and reduce the overall hardware usage. These optimizations can cause small differences in floating-point results.

- **Tree Balancing**

We have this equation:

```
result = (((A * B) + C) + (D * E)) + (F * G);
```

By default, the AOC creates an implementation that resembles a long vine for such computations:

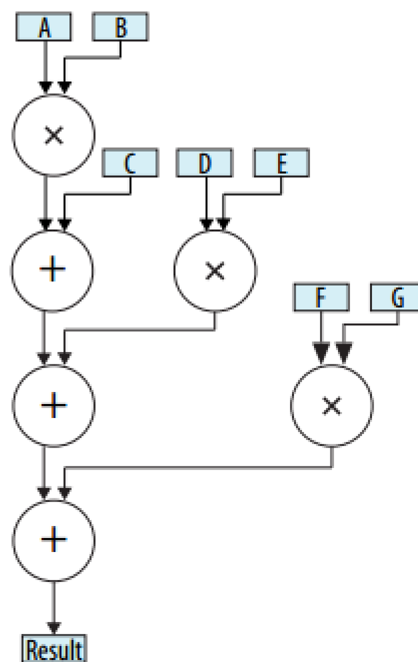



***Figure 2.7.1*** *Default Floating-Point Implementation*

Long, unbalanced operations lead to more expensive hardware. A more efficient hardware implementa- tion is a balanced tree, as shown below:

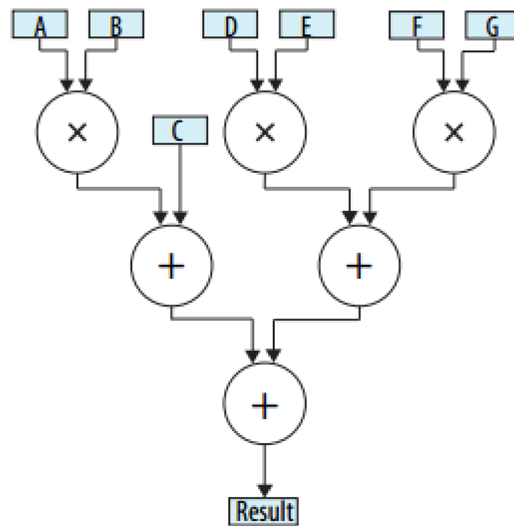



***Figure 2.7.2*** *Balanced Tree Floating-Point Implementation*

In a balanced tree implementation, the AOC converts the long vine of floating-point adders into a tree pipeline structure. The AOC does not perform tree balancing of floating-point operations automatically because the outcomes of the floating-point operations might differ. As a result, this optimization is inconsistent with the IEEE Standard 754-2008.

If we want the AOC to optimize floating-point operations using balanced trees and your program can tolerate small differences in floating-point results, include the *--fp-relaxed* option in the aoc command.

- **Rounding Operations**

The balanced tree implementation of a floating-point operation includes multiple rounding operations. These rounding operations can require a significant amount of hardware resources in some applications. The AOC does not reduce the number of rounding operations automatically because doing so violates the results required by IEEE Standard 754-2008.

We can reduce the amount of hardware necessary to implement floating-point operations with the -- *fpc* option of the aoc command.

### 2.7.1 Floating-Point versus Fixed-Point Representations

An FPGA contains a substantial amount of logic for implementing floating-point operations. However, we can increase the amount of hardware resources available by using a fixed-point representation of the data whenever possible. The hardware necessary to implement a fixed-point operation is typically smaller than the equivalent floating-point operation. As a result, you can fit more fixed-point operations into an FPGA than the floating-point equivalent.

The OpenCL standard does not support fixed-point representation; we must implement fixed-point representations using integer data types. Hardware developers commonly achieve hardware savings by using fixed-point data representations and only retain a data resolution required for performing calculations. We must use an 8, 16, 32, or 64-bit scalar data type because the OpenCL standard supports only these data resolutions

# Chapter 3. Matrix-multiplication (Study an example)

According to the definition of BLAS libraries, we define the computation of Matrix-multiplication as follow: the single-precision general matrix-multiplication (SGEMM) computes the following form:

$$C = alpha * A * B + beta * C$$

Where: A is a K by M input matrix, B is an N by K input matrix, C is the M by N output matrix, and alpha and beta are scalar constants. For simplicity, we assume the common case where alpha is equal to 1 and beta is equal to zero, we get:

$$C = A * B$$

There is a condition for the matrix sizes must be equal or multiples of OpenCL workgroups.

There are many ways for implementations of single-precision matrix-multiplication, we will study ‘’ Tiling in the local memory’’.

## 3.1 Tiling in the local memory

The main reason the naive implementation doesn't perform so well is because we are accessing the kernel device's off-chip memory way too much. We have M*N*K multiplications and additions, we need M*N*K*2 loads and M*N stores. To avoid that we compute by sub-block, so we divide the matrixes to sub-block and work with them, in figure 03 we multiply sub-A with sub-B and we get sub-C and this save time of accessing to kernel device's off-chip memory.

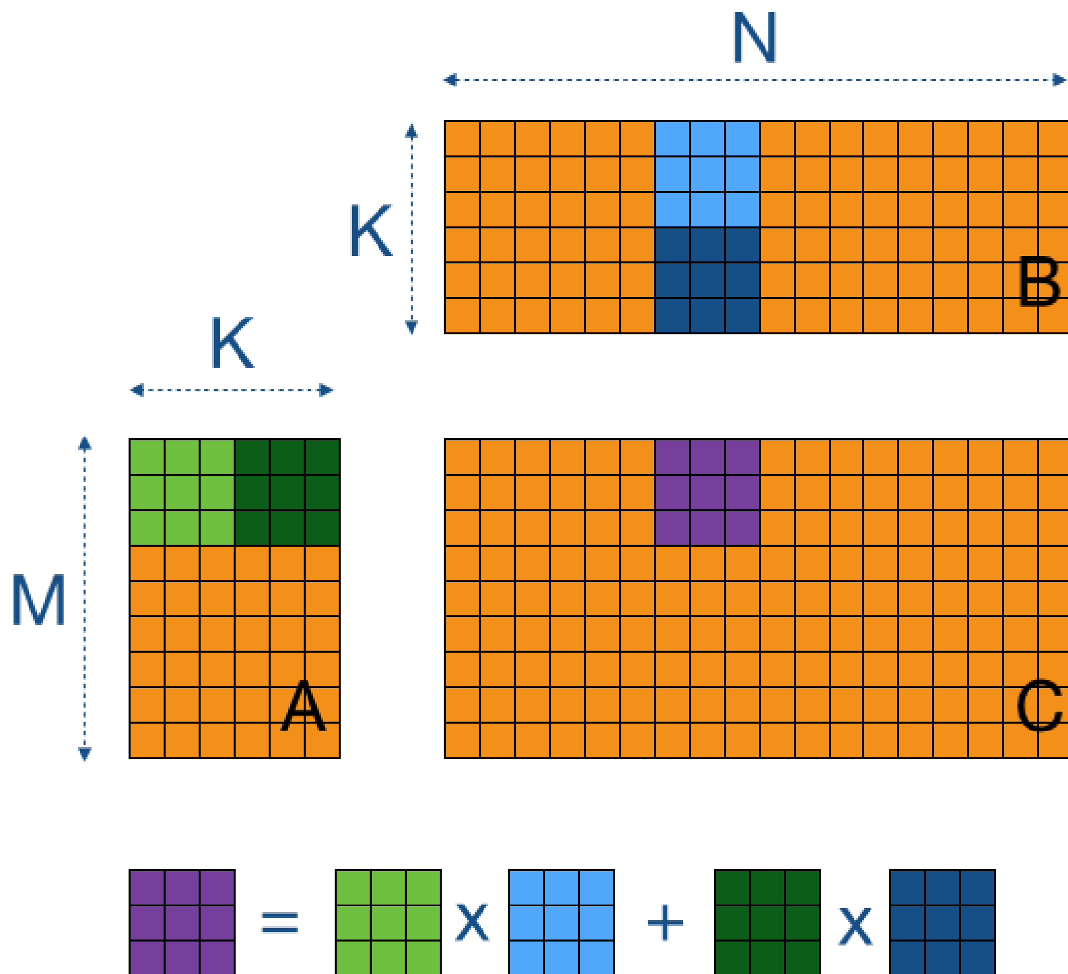



***Figure 03*** *matrix-multiplication by sub-block*

## 3.2 Host Code

The version of SGEMM can be implemented in plain C using 3 nested loops, we see code below

```
1. for (int m=0; m<M; m++) {
2.     for (int n=0; n<N; n++) {
3.         float acc = 0.0f;
4.         for (int k=0; k<K; k++) {
5.             acc += A[k*M + m] * B[n*K + k];
6.         }
7.         C[n*M + m] = acc;
8.     }
9. }
```

## 3.3 Kernel code

The following code is an implementation of tiling:

```
1. // Tiled and coalesced version
2. __kernel void myGEMM2(const int M, const int N, const int K,
3.                       const __global float* A,
4.                       const __global float* B,
5.                       __global float* C) {
6.
7.     // Thread identifiers
8.     const int row = get_local_id(0); // Local row ID (max: TS)
9.     const int col = get_local_id(1); // Local col ID (max: TS)
10.    const int globalRow = TS*get_group_id(0) + row; // Row ID of C (0..M)
11.    const int globalCol = TS*get_group_id(1) + col; // Col ID of C (0..N)
12.
13.    // Local memory to fit a tile of TS*TS elements of A and B
14.    __local float Asub[TS][TS];
15.    __local float Bsub[TS][TS];
16.
17.    // Initialise the accumulation register
18.    float acc = 0.0f;
19.
20.    // Loop over all tiles
21.    const int numTiles = K/TS;
22.    for (int t=0; t<numTiles; t++) {
23.
24.        // Load one tile of A and B into local memory
25.        const int tiledRow = TS*t + row;
26.        const int tiledCol = TS*t + col;
27.        Asub[col][row] = A[tiledCol*M + globalRow];
28.        Bsub[col][row] = B[globalCol*K + tiledRow];
29.
30.        // Synchronise to make sure the tile is loaded
31.        barrier(CLK_LOCAL_MEM_FENCE);
32.
33.        // Perform the computation for a single tile
34.        for (int k=0; k<TS; k++) {
35.            acc += Asub[k][row] * Bsub[col][k];
36.        }
37.
38.        // Synchronise before loading the next tile
39.        barrier(CLK_LOCAL_MEM_FENCE);
40.    }
41.
42.    // Store the final result in C
43.    C[globalCol*M + globalRow] = acc;
44. }
```

We see that the original accumulation loop over K has been split into two new loops: first one over all K/TS tiles and second one over all TS elements within a tile. We can identify two parts which are separated with synchronization barriers: first loading from off-chip memory to local memory, and second computation based on local memory data.

# Chapter 4. Application of matrix-multiplication kernel in Backpropagation

In this chapter we will talk about the algorithm of Backpropagation that we used in our study.

## 4.1 The time execution

The following equation defines the time execution of the algorithm Backpropagation.

$$T_{total} = T_{ini} + E \cdot (T_{CW} + T_{uw}) + T_{test} \qquad (1)$$

Where:

$T_{ini}$ : is the time of initialization.
$E$ : is the number of iterations of a loop.
$T_{cw}$ : is the calculus of partial derivative of the error with respect to each weight.
$T_{Uw}$: is the update of the weights.
$T_{test}$ : is the time of obtaining of calculus with test set.

### 4.1.1 $T_{cw}$

The time dedicated to calculus of partial derivative of the error with respect to each weight is usually broken down into three phases to be performed iteratively for each pattern. If we had $N_p$ patterns it would be

$$T_{cw} = N_p \cdot (T_f + T_{b1} + T_{b2}) \qquad (2)$$

The three steps involved in this time taking one pattern $m$ at a time, having $L$ layers of neurons and $N_l$ neurons in each layer, and calculating the partial derivative of the error with respect to each weight are as follows:

- Forward step. Apply the pattern $y_i^K$ to the input layer and propagate the signal forward through the network until the final outputs $y_i^L$ have been calculated for each $i$(index of neuron) and $l$(index of layer)

➢

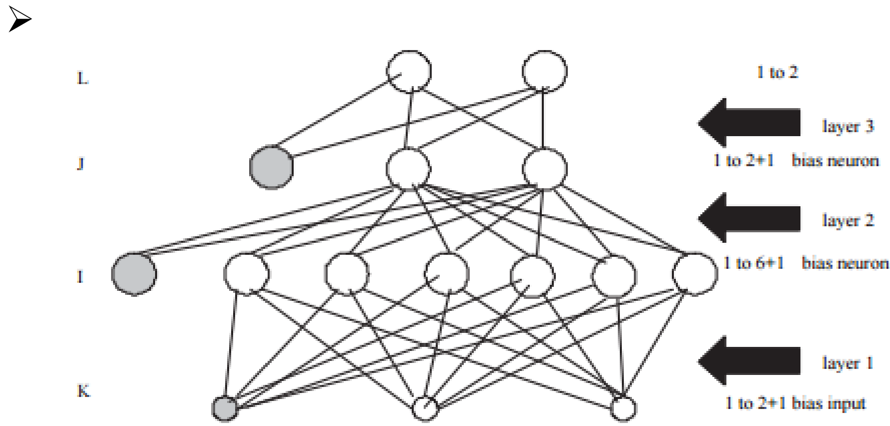



*Figure 4.1.1: Multiplayer Perceptron*

$$u_i^l = \sum_{j=0}^{N_{l-1}} w_{ij}^l y_j^{l-1} \quad (3)$$

$$y_i^l = f(u_i^l) \quad (4)$$

$$y_i'^l = f'(u_i^l) \quad (5)$$

$$1 \le i \le N_l, \qquad 1 \le l \le L$$

where *y* is the activation, *w* the weights and *f* the non-linear function

➢ Backward step 1. Compute the δ's for the output layer $L$ and compute the δ's for the preceding layers by propagating the errors backwards using

$$\delta_j^L = f'(u_j^L)(t_i - y_i) \quad (6)$$

$$\theta_i^{l-1} = \sum_{i=1}^{N_l} w_{ij} \delta_i^l \quad (7)$$

$$\delta_j^{l-1} = f'(u_j^{l-1})\ \theta_i^{l-1} \quad (8)$$

$$1 \le i \le N_l, \qquad 1 \le l \le L$$

where $\delta$ are the error terms, $t$ the targets and $f'$ the derivative function of $f$

➢ Backward step 2. Meanwhile we obtain the delta errors en backward step 1 we can obtain the accumulation of partial derivative of the error with respect to each weight using

$$e = 1/2 \sum_{i=1}^{N_l} (t_i - y_i^L)^2 \quad (9)$$

$$m \frac{\partial e}{\partial w_{ij}^l} = \delta_i^l y_j^{l-1} \quad (10)$$

$${}^{m}cw_{ij}^l = {}^{m-1}cw_{ij}^l + m \frac{\partial e}{\partial w_{ij}^l} \quad (11)$$

$$1 \le i \le N_l, \qquad 1 \le l \le L$$

where *cw* is the accumulation of the direction of the gradient of the error . When we finish with all the patterns, this accumulation will be

$$cw_{ij}^{l} = \frac{\partial E}{\partial w_{ij}^{l}} \tag{12}$$

$$E = \frac{1}{2}\sum_{m=1}^{N_p}\sum_{i=1}^{N_l}(t_i - y_i^L)^2 \tag{13}$$

$$1 \leq l \leq L$$

### 4.1.2 $T_{uw}$

The time dedicated to update the weights of the ANN is dedicated to actualize all the weights of the ANN when each EPOCH has finished

$$w_{ij}^{(t+1)} = w_{ji}^{(t)} + \Delta w_{ij}^{(t)} \tag{14}$$

$$\Delta w_{ij}^{(t)} = \begin{cases} -\Delta w_{ij}^{(t-1)}, if \left(\frac{\partial E}{\partial w_{ij}}\right)^{(t-1)} * \left(\frac{\partial E}{\partial w_{ij}}\right)^{(t)} < 0 \\ -\Delta_{ij}^{(t)}, elsif \left(\frac{\partial E}{\partial w_{ij}}\right)^{(t)} > 0 \\ -\Delta_{ij}^{(t)}, elsif \left(\frac{\partial E}{\partial w_{ij}}\right)^{(t)} < 0 \\ 0,\ else \end{cases} \tag{15}$$

$$\Delta_{ij}^{(t)} = \begin{cases} \eta^{+} * \Delta_{ij}^{(t-1)}, if \left(\frac{\partial E}{\partial w_{ij}}\right)^{(t-1)} * \left(\frac{\partial E}{\partial w_{ij}}\right)^{(t)} > 0 \\ \eta^{-} * \Delta_{ij}^{(t-1)}, if \left(\frac{\partial E}{\partial w_{ij}}\right)^{(t-1)} * \left(\frac{\partial E}{\partial w_{ij}}\right)^{(t)} < 0 \\ \Delta_{ij}^{(t-1)},\ else \\ where\ 0 < \eta^{-} < 1 < \eta^{+} \end{cases} \tag{16}$$

If we combine the Equations 14, 15 and 16 we consider the following pseudo code

$$
\begin{aligned}
& \textit{for each } w_{ij} \textit{ do } \{ \\
& \quad \textit{if } (\partial E/\partial w_{ij})^{(t-1)} . (\partial E/\partial w_{ij})^{(t)} > 0 \textit{ then } \{ \\
& \qquad \Delta_{ij}^{(t)} := \min(\Delta_{ij}^{(t-1)} . \eta^{+}, \Delta_{max}) \\
& \qquad \Delta w_{ij}^{(t)} := -\mathrm{sign}((\partial E/\partial w_{ij})^{(t)}) . \Delta_{ij}^{(t)} \\
& \qquad w_{ij}^{(t+1))} := w_{ij}^{(t)} + \Delta w_{ij}^{(t)} \quad \} \\
& \quad \textit{elsif } (\partial E/\partial w_{ij})^{(t-1)} . (\partial E/\partial w_{ij})^{(t)} < 0 \textit{ then } \{ \\
& \qquad \Delta_{ij}^{(t)} := \max(\Delta_{ij}^{(t-1)} . \eta^{-}, \Delta_{min}) \\
& \qquad w_{ij}^{(t+1))} := w_{ij}^{(t)} - \Delta w_{ij}^{(t-1)} \\
& \qquad (\partial E/\partial w_{ij})^{(t)} := 0 \quad \} \\
& \quad \textit{elsif } (\partial E/\partial w_{ij})^{(t-1)} . (\partial E/\partial w_{ij})^{(t)} = 0 \textit{ then } \{ \\
& \qquad \Delta w_{ij}^{(t)} := -\mathrm{sign}((\partial E/\partial w_{ij})^{(t)}) . \Delta_{ij}^{(t)} \\
& \qquad w_{ij}^{(t+1))} := w_{ij}^{(t)} + \Delta w_{ij}^{(t)} \quad \} \\
& \}
\end{aligned}
\tag{17}
$$

### 4.2 Original algorithm

We can perform the equation 01 by implementing this algorithm with OpenCL, our main candidates for the kernels are the tasks called in lines 1,2,3,4 and 5 (see Figure 4.2). The tasks called in lines 1, 2 and 3 are implementing the Equation 3, and the tasks called in lines 4 and 5 are implementing the Equation 7. Each kernel would have a similar structure based on a matrix vector multiplication, and how a path dependence between them, the better solution exist is implement a unique kernel and reuse 5 times in each iteration. Therefore we would use the same kernel in $5 \cdot Np \cdot E$ times in our algorithm. .

**input** : A training set (Inputs_train,Targets_train) and a test set (Inputs_tets,Targets_test)
**output:** Number of errors in test set

Initweights $({}^{0}w^{I},{}^{0}w^{J},{}^{0}w^{L}) \leftarrow 0$;
**for** $i \leftarrow 1$ **to** $E$ **do**
  Initcweights $({}^{0}cw^{I},{}^{0}cw^{J},{}^{0}cw^{L}) \leftarrow 0$;
Pattern  **for** $j \leftarrow 1$ **to** $N_p$ **do**
1    $u^{I} \leftarrow$ Forward_layer1$({}^{i-1}w^{I}, Inputs_train)$;
    ${}^{j}y^{I} \leftarrow$ Funct_nolineal$(u^{I})$;
    ${}^{j}y'^{I} \leftarrow$ der_Funct_nolineal$(u^{I})$;
2    $u^{J} \leftarrow$ Forward_layer2$({}^{i-1}w^{J}, {}^{j}y^{I})$;
    ${}^{j}y^{J} \leftarrow$ Funct_nolineal$(u^{J})$;
    ${}^{j}y'^{J} \leftarrow$ der_Funct_nolineal$(u^{J})$;
3    $u^{L} \leftarrow$ Forward_layer3$({}^{i-1}w^{L}, {}^{j}y^{J})$;
    ${}^{j}y^{L} \leftarrow$ Funct_nolineal$(u^{L})$;
    ${}^{j}y'^{L} \leftarrow$ der_Funct_nolineal$(u^{L})$;
    ${}^{j}\delta^{L} \leftarrow$ Backward1_layer3$(Targets_train, {}^{j}y^{L}, {}^{j}y'^{L})$;
    ${}^{j}cw^{L} \leftarrow$ Backward2_layer3$({}^{j}\delta^{L}, {}^{j}y^{J}, {}^{j-1}cw^{L})$;
4    ${}^{j}\theta^{J} \leftarrow$ Backward1_layer2$({}^{i-1}w^{L}, {}^{j}\delta^{L})$;
    ${}^{j}\delta^{J} \leftarrow {}^{j}\theta^{J} \cdot {}^{j}y'^{J}$;
    ${}^{j}cw^{J} \leftarrow$ Backward2_layer2$({}^{j}\delta^{J}, {}^{j}y^{I}, {}^{j-1}cw^{J})$;
5    ${}^{j}\theta^{I} \leftarrow$ Backward1_layer1$({}^{i-1}w^{J}, {}^{j}\delta^{J})$;
    ${}^{j}\delta^{I} \leftarrow {}^{j}\theta^{I} \cdot {}^{j}y'^{I}$;
    ${}^{j}cw^{I} \leftarrow$ Backward2_layer1$({}^{j}\delta^{I}, Inputs_train, {}^{j-1}cw^{I})$;
  ${}^{i}w^{L} \leftarrow$ Update_layer3$({}^{i-1}w^{L}, {}^{N_p}cw^{L})$;
  ${}^{i}w^{J} \leftarrow$ Update_layer2$({}^{i-1}w^{J}, {}^{N_p}cw^{J})$;
  ${}^{i}w^{I} \leftarrow$ Update_layer1$({}^{i-1}w^{I}, {}^{N_p}cw^{I})$;
Fitness $\leftarrow$ TestNeuralNetwork$(Inputs_test, Targets_test, {}^{E}w^{I}, {}^{E}w^{J}, {}^{E}w^{L})$;

***Figure 4.2*** *Resilient Backpropagation original*

## 4.3 Final algorithm

We can perform the equation 01 by implementing this algorithm with OpenCL, our main candidates for the kernels are the lines 1, 2, 3, 4, 5, 6, 7 and 8 of algorithm 2(see Figure 4.3). Each kernel would have a similar structure based on a matrix matrix-multiplication, and how a path dependence between them, the better solution exist is implement a unique kernel and reuse 7 times in each iteration. Therefore we would use the same kernel in $7 \cdot E$ times in our algorithm. In equation 18 we can see the matrix matrix-multiplications of forward phase, in equation 19 we can see the matrix matrix-multiplications of backward1 phase and in equation 20 we can observe the matrix matrix-multiplications of backward2 phase. In each matrix matrix-multiplication the size of the matrices must be modified and of course our OpenCL implementation must permit to change this parameters. This algorithm use Matrix-multiplication with tiling that we had it in chapter 3.

```
input  : A training set (Inputs_train,Targets_train) and a test set (Inputs_tets,Targets_test)
output: Number of errors in test set
Initweights (⁰w^I,⁰w^J,⁰w^L)← 0;
for i ← 1 to E do
    Initcweights (⁰cw^I,⁰cw^J,⁰cw^L)← 0;
1   u^I ← Forward_layer1(^{i-1}w^I,Inputs_train);
    y^I ← Funct_nolineal(u^I);
    y'^I ← der_Funct_nolineal(u^I);
2   u^J ← Forward_layer2(^{i-1}w^J,y^I);
    y^J ← Funct_nolineal(u^J);
    y'^J ← der_Funct_nolineal(u^J);
3   u^L ← Forward_layer3(^{i-1}w^L,y^J);
    y^L ← Funct_nolineal(u^L);
    y'^L ← der_Funct_nolineal(u^L);
    δ^L ← Backward1_layer3(Targets_train,y^L,y'^L);
4   cw^L ← Backward2_layer3(δ^L,y^J,cw^L);
5   ^jθ^J ← Backward1_layer2(^{i-1}w^L,δ^L);
    δ^J ← ^jθ^J · y'^J;
6   cw^J ← Backward2_layer2(δ^J,y^I,cw^J);
7   ^jθ^I ← Backward1_layer1(^{i-1}w^J,δ^J);
    δ^I ← ^jθ^I · y'^I;
8   cw^I ← Backward2_layer1(δ^I,Inputs_train,cw^I);
    ^iw^L ← Update_layer3(^{i-1}w^L,cw^L);
    ^iw^J ← Update_layer2(^{i-1}w^J,cw^J);
    ^iw^I ← Update_layer1(^{i-1}w^I,cw^I);
Fitness← TestNeuralNetwork(Inputs_test,Targets_test,^Ew^I,^Ew^J,^Ew^L);
```

***Figure 4.3** Resilient Backpropagation*

$$\left\{\begin{array}{c}
\begin{pmatrix} Input_{11} & \cdots & Input_{1n} \\ \vdots & \ddots & \vdots \\ Input_{m1} & \dots & Input_{mn} \end{pmatrix}\begin{pmatrix} w_{11} & \cdots & w_{1k} \\ \vdots & \ddots & \vdots \\ w_{n1} & \dots & w_{nk} \end{pmatrix} = \begin{pmatrix} u_{11} & \cdots & u_{1k} \\ \vdots & \ddots & \vdots \\ u_{m1} & \dots & u_{mk} \end{pmatrix} \\
\rightarrow \left\{\begin{matrix} m = numPattern \\ n = numInputs \\ k = numHidden1 \end{matrix}\right\} \rightarrow forward - layer1 \\
\begin{pmatrix} y_{11} & \cdots & y_{1n} \\ \vdots & \ddots & \vdots \\ y_{m1} & \dots & y_{mn} \end{pmatrix}\begin{pmatrix} w_{11} & \cdots & w_{1k} \\ \vdots & \ddots & \vdots \\ w_{n1} & \dots & w_{nk} \end{pmatrix} = \begin{pmatrix} u_{11} & \cdots & u_{1k} \\ \vdots & \ddots & \vdots \\ u_{m1} & \dots & u_{mk} \end{pmatrix} \\
\rightarrow \left\{\begin{matrix} m = numPattern \\ n = numHidden1 \\ k = numHidden2 \end{matrix}\right\} \rightarrow forward - layer2 \\
\begin{pmatrix} y_{11} & \cdots & y_{1n} \\ \vdots & \ddots & \vdots \\ y_{m1} & \dots & y_{mn} \end{pmatrix}\begin{pmatrix} w_{11} & \cdots & w_{1k} \\ \vdots & \ddots & \vdots \\ w_{n1} & \dots & w_{nk} \end{pmatrix} = \begin{pmatrix} u_{11} & \cdots & u_{1k} \\ \vdots & \ddots & \vdots \\ u_{m1} & \dots & u_{mk} \end{pmatrix} \\
\rightarrow \left\{\begin{matrix} m = numPattern \\ n = numHidden2 \\ k = numOutputs \end{matrix}\right\} \rightarrow forward - layer3
\end{array}\right\} \rightarrow forward\ phas$$

*(18)*

$$\left.\begin{cases} \begin{pmatrix} \delta_{11} \cdots \delta_{1n} \\ \vdots \ \ddots \ \vdots \\ \delta_{m1} \cdots \delta_{mn} \end{pmatrix} \begin{pmatrix} w_{11} \cdots w_{1k} \\ \vdots \ \ddots \ \vdots \\ w_{n1} \cdots w_{nk} \end{pmatrix} = \begin{pmatrix} \theta_{11} \cdots \theta_{1k} \\ \vdots \ \ddots \ \vdots \\ \theta_{m1} \cdots \theta_{mk} \end{pmatrix} \\ \rightarrow \begin{Bmatrix} m = numPattern \\ n = numOutputs \\ k = numHidden2 \end{Bmatrix} \rightarrow backward1 - layer2 \\ \begin{pmatrix} \delta_{11} \cdots \delta_{1n} \\ \vdots \ \ddots \ \vdots \\ \delta_{m1} \cdots \delta_{mn} \end{pmatrix} \begin{pmatrix} w_{11} \cdots w_{1k} \\ \vdots \ \ddots \ \vdots \\ w_{n1} \cdots w_{nk} \end{pmatrix} = \begin{pmatrix} \theta_{11} \cdots \theta_{1k} \\ \vdots \ \ddots \ \vdots \\ \theta_{m1} \cdots \theta_{mk} \end{pmatrix} \\ \rightarrow \begin{Bmatrix} m = numPattern \\ n = numHidden2 \\ k = numOutputs \end{Bmatrix} \rightarrow backward1 - layer1 \end{cases}\right\} \rightarrow backward1\ fase$$

*(19)*

$$\left.\begin{cases} \begin{pmatrix} \delta_{11} \cdots \delta_{1n} \\ \vdots \ \ddots \ \vdots \\ \delta_{m1} \cdots \delta_{mn} \end{pmatrix} \begin{pmatrix} y_{11} \cdots y_{1k} \\ \vdots \ \ddots \ \vdots \\ y_{n1} \cdots y_{nk} \end{pmatrix} = \begin{pmatrix} cw_{11} \cdots cw_{1k} \\ \vdots \ \ddots \ \vdots \\ cw_{m1} \cdots cw_{mk} \end{pmatrix} \\ \rightarrow \begin{Bmatrix} m = numPattern \\ n = numOutputs \\ k = numHidden2 \end{Bmatrix} \rightarrow backward2 - layer3 \\ \begin{pmatrix} \delta_{11} \cdots \delta_{1n} \\ \vdots \ \ddots \ \vdots \\ \delta_{m1} \cdots \delta_{mn} \end{pmatrix} \begin{pmatrix} y_{11} \cdots y_{1k} \\ \vdots \ \ddots \ \vdots \\ y_{n1} \cdots y_{nk} \end{pmatrix} = \begin{pmatrix} cw_{11} \cdots cw_{1k} \\ \vdots \ \ddots \ \vdots \\ cw_{m1} \cdots cw_{mk} \end{pmatrix} \\ \rightarrow \begin{Bmatrix} m = numHidden2 \\ n = numPattern \\ k = numHidden1 \end{Bmatrix} \rightarrow backward2 - layer2 \\ \begin{pmatrix} \delta_{11} \cdots \delta_{1n} \\ \vdots \ \ddots \ \vdots \\ \delta_{m1} \cdots \delta_{mn} \end{pmatrix} \begin{pmatrix} y_{11} \cdots y_{1k} \\ \vdots \ \ddots \ \vdots \\ y_{n1} \cdots y_{nk} \end{pmatrix} = \begin{pmatrix} cw_{11} \cdots cw_{1k} \\ \vdots \ \ddots \ \vdots \\ cw_{m1} \cdots cw_{mk} \end{pmatrix} \\ \rightarrow \begin{Bmatrix} m = numHidden1 \\ n = numPattern \\ k = numIntputs \end{Bmatrix} \rightarrow backward2 - layer1 \end{cases}\right\} \rightarrow backward2\ fase$$

*(20)*

# Chapter 5. Ability to quickly integrate IPs in OpenCL and Results

In this section we will see one advanced features for OpenCL Altera SDK how we can write our own code HDL inside OpenCL kernel and create its library. And we present our result with graphs.

## 5.1 OpenCL Library

The Altera SDK for OpenCL provides advanced features we can create our OpenCL library, OpenCL library is a single file that contains multiple functions and we can create it in OpenCL or RTL.

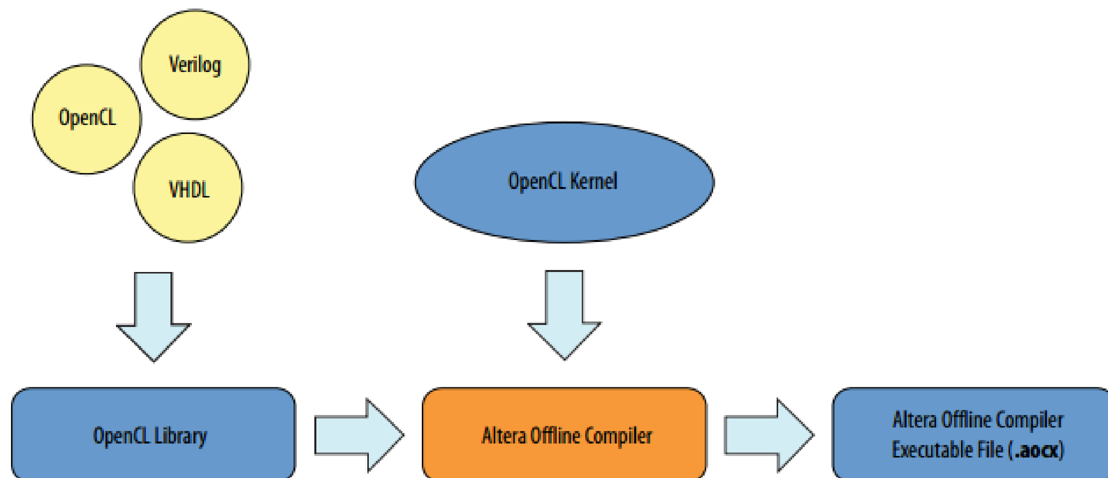



***Figure 5.1*** *Overview of Altera SDK for OpenCL's Library Support*

To create an OpenCL library we need the following files:

- ***RTL Components***

  - **RTL source file:** Verilog, VHDL.
  - **eXtensible Markup Language File (.xml):** The Altera Offline Compiler uses these properties to integrate the RTL component into the OpenCL pipeline.
  - **Header file (.h):** A C-style header file that declares the signatures of function(s) that are implement by the RTL component.

- **OpenCL emulation model file (.cl):** Provides C model for the RTL component that is used only for emulation.

- ***OpenCL Functions***

  - **OpenCL source files (.cl):** Contains definitions of the OpenCL functions.
  - **Header file (.h):** A C-style header file that declares the signatures of function(s) that are defined in the OpenCL source files.

### 5.1.1 RTL Modules and the OpenCL Pipeline

In the following figure represent the architecture of an AOCL pipeline

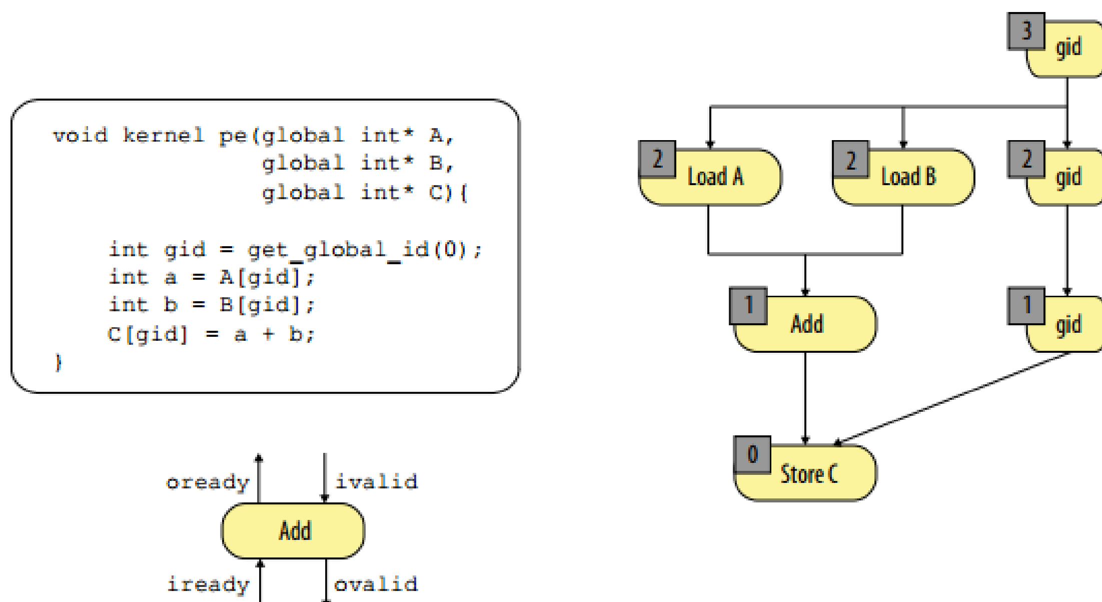



***Figure 5.1.1** Parallel Execution Model of AOCL Pipeline Stages*

In the figure 5.1.1 the operations on the right represent the AOCL pipeline implementation of the OpenCL kernel code on the left. At each stage, the AOC executes all operations in parallel by the thread existing at that stage.

- **Integration of an RTL Module into the AOCL Pipeline**

In the figure 5.1.2 we see how the AOC integrates the RTL module myMod within the library into the AOCL pipeline. The RTL module has a balanced latency where the threads of the RTL module match the number of pipeline stages.

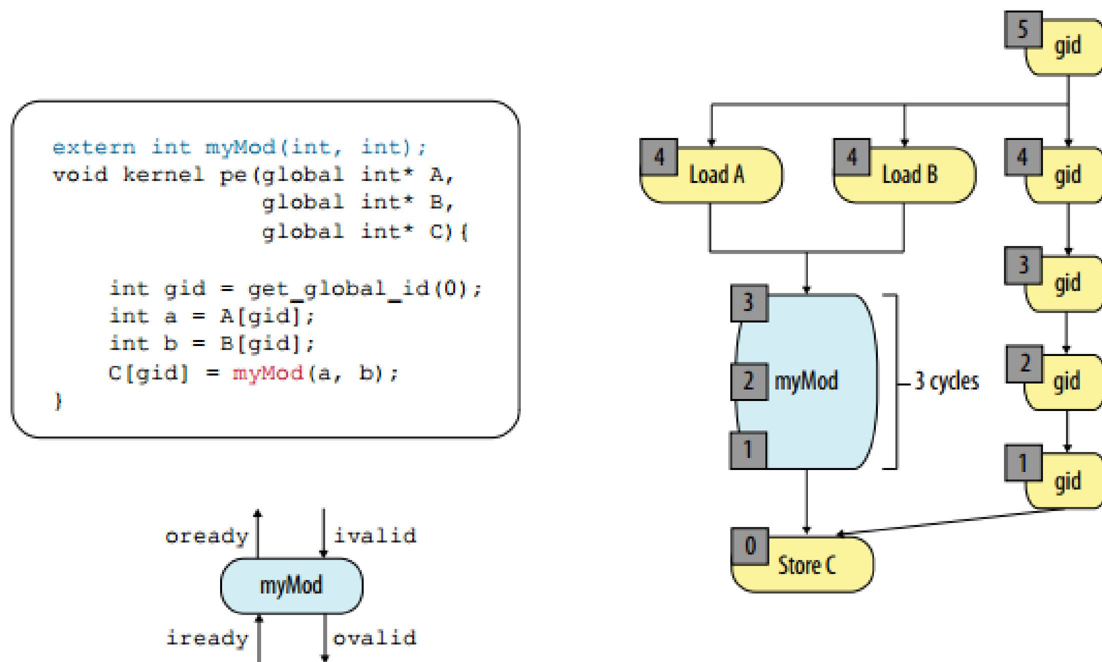



***Figure 5.1.2*** *Integration of an RTL Module into an AOCL Pipeline*

### ➢ RTL Reset and Clock Signals

Resets and clocks of RTL modules are connected to the same clock and reset drivers as the rest of the OpenCL pipeline.

The following steps outline the process of setting the kernel clock frequency:
1. The Quartus Prime software's Fitter applies an aggressive constraint on the kernel clock.
2. The Quartus Prime software's TimeQuest Timing Analyzer performs static timing analysis to determine the frequency that the Fitter actually achieves.
3. The phase-locked loop (PLL) that drives the kernel clock sets the frequency determined in Step 2 to be the kernel clock frequency.

### ➢ XML Syntax of an RTL Module

Elements and Attributes in the XML Specification File:

RTL_SPEC: Top-level element in the XML specification file. There can only be one such top-level element in the file.

FUNCTION: defines the OpenCL function that the RTL module implements. We can have multiple function.
ATTRIBUTES: containing other XML elements that describe various characteristics (for example, latency) of the RTL module.
INTERFACE: containing other XML elements that describe the RTL module's interface.
C_MODEL: specifying one or more files that implement OpenCL C model for the function.
REQUIREMENTS: specifying one or more RTL resource files (that is, .v, .vhd…..)

- **Order of Threads Entering an RTL Module**

That threads entering an RTL module not follow a defined order.

- **OpenCL C Model of an RTL Module**

Each RTL module within an OpenCL library must have an OpenCL C model, if we decide not to emulate our OpenCL system, we have to create an empty function with a name that matches the function name you specified in the XML specification file.

- **Potential Incompatibility between RTL Modules and Partial Reconfiguration**

If a library user then uses the library's RTL module inside a PR region, the module might not function correctly after PR. To ensure that the RTL modules function correctly on a device that uses PR:
1-The RTL modules do not use memory logic array blocks (MLABs) with initialized content.
2-The RTL modules do not make any assumptions regarding the power-up values of any logic.

### 5.1.2 Packaging an OpenCL Helper Function File

Before creating an OpenCL library file, package each OpenCL source file with helper functions into a **.aoco** file. Unlike RTL modules, we do not need to create an XML specification file.

To package an OpenCL source file into a .**aoco** file, invoke the following command:
*aoc -c -shared <OpenCL_source_file_name>.cl –o <OpenCL_object_file_name>.aoco*

where the -shared AOC command option instructs the AOC to create a **.aoco** file that is suitable for inclusion into an OpenCL library.

### 5.1.3 Packaging an RTL Component for an OpenCL Library

To package an RTL component into a **.aoco** file, invoke the following command:
*aoc -c <RTL component description file name>.xml -o <RTL object file name>.aoco*

### 5.1.4 Restrictions and Limitations in RTL Support

The Altera SDK for OpenCL supports the use of RTL modules in an OpenCL library with the following restrictions:

1- An RTL module must contain one Avalon-ST interface. In particular, a single ready or valid logic must control all the inputs.
2- The RTL module must work correctly with exactly one clock, regardless of clock frequency.
3- Data input and output sizes must match valid OpenCL data types, from 8 bits for *char* to 1024 bits for *long16*.
4- RTL modules cannot connect to external I/O signals. All input and output signals must come from an OpenCL kernel.
5- An RTL module must have a clock port, a resetn port, and Avalon-ST input and output ports (that is, ivalid, ovalid, iready, oready).
6- RTL modules that communicate with external memory must have Avalon Memory-Mapped (AvalonMM) port parameters that match the corresponding Custom Platform parameters.
7- RTL modules that communicate with external memory cannot burst across the burst boundary also cannot make requests every clock cycle and stall the hardware by monopolizing the arbitration logic.
8- RTL modules cannot act as stand-alone OpenCL kernels.
9- Every function call that corresponds to RTL module instantiation is completely independent of other instantiations. There is no hardware sharing.
10- Do not incorporate kernel code (that is, functions marked as kernel) into a **.aoclib** library file.
11- An RTL component must receive all its inputs at the same time.

12- AOCL does not support I/O RTL modules.
13- You can only set RTL module parameters in the *<RTL module description file name>.xml* specification file, not the OpenCL kernel source file.

- AOCL's RTL module support for the library feature has the following limitations:

1- We can only pass data inputs to an RTL module by value via the OpenCL kernel code.
2- We cannot include the *-g* AOC command option when compiling kernels that use libraries.
3- Names of RTL module source files cannot conflict with the file names of AOC IP.
4- AOCL does not support **.qip** files. We must manually parse nested .**qip** files to create a flat list of RTL files.
5- It is very difficult to debug an RTL module that works correctly on its own but works incorrectly as part of an OpenCL kernel.
6- All AOC area estimation tools assume that RTL module area is 0.
7- RTL modules cannot access a 2x clock that is in-phase with the kernel clock and at twice the kernel clock frequency.

### 5.1.5 Verifying the RTL Modules

We verify each RTL module using standard hardware verification methods and we can also modify one of Altera's OpenCL library design examples to test our RTL modules inside the overall OpenCL system.

### 5.1.6 Packaging Multiple Object Files into a Library File

To package multiple object files into a single library file, invoke the following command:

*aocl library create -o <library file name>.aoclib <object file 1>.aoco [<object file 2>.aoco ... <object file N>.aoco]*

The *aocl* library utility command creates a *<library file name>.aoclib* library file, which includes the .**aoco** object files we specify in the command.
A library file may contain both RTL-based object files and OpenCL-based object files.

### 5.1.7 Specifying an OpenCL Library when Compiling an OpenCL Kernel

We can specify an OpenCL library to the AOC by invoke the following command:

*aoc –l <library_file_name>.aoclib [-L <library directory>] <kernel file*

*name>.cl*

Where the command *-l <library_file_name>.aoclib* specifies the library file name, and the command -L <library directory> specifies the directory containing the library files.

### 5.1.8 OpenCL Library Command-Line Options

We can invoke the following commands to perform OpenCL library-related tasks:

1-Library-Related AOC Command Options

*’’-shared’’:* In conjunction with the -c command option, compiles an OpenCL source file into an object file (.aoco)
*’’ -I <library_directory>’’*: Adds <library directory> to the header file search path.
*’’ -L <library directory>’’:* Adds <library directory> to the OpenCL library search path.
*’’-l <library_file_name>.aoclib’’:* Specifies the OpenCL library file (**<library_file_name>.aoclib**).
*’’--library-debug’’:* Generates debug output that relates to libraries.

2- AOCL Library Utility (aocl library) Command Options

’’*hdl-comp-pkg <XML_specification_ file>.xml*’’ or ’’*aoc -c <XML_specification_file>.xml.*’’: Packages a single HDL component into a .**aoco** file that we then include into a library.
*’’ -c <XML_specification_file>.xml’’:* Same function as *hdl-comp-pkg <XML_specification_file>.xml*.
*’’create’’:* Creates a library file from the .**aoco** files.
*’’list <library_name>’’:* Lists all the RTL components in the library.
*’’help’’:* Prints the list of AOCL library utility options and their descriptions on screen.

## 5.2 Function tangent hyperbolic ‘tanh’

The function Tangent hyperbolic is a no linear function for the hardware accelerate, we had wrote this function inside OpenCL Kernel and we get better result than the kernel alone, the function work on fixed point so it must create two converter: Floating point to fixed point and fixed point to floating point, the following photo shows the structure of the function tangent hyperbolic.

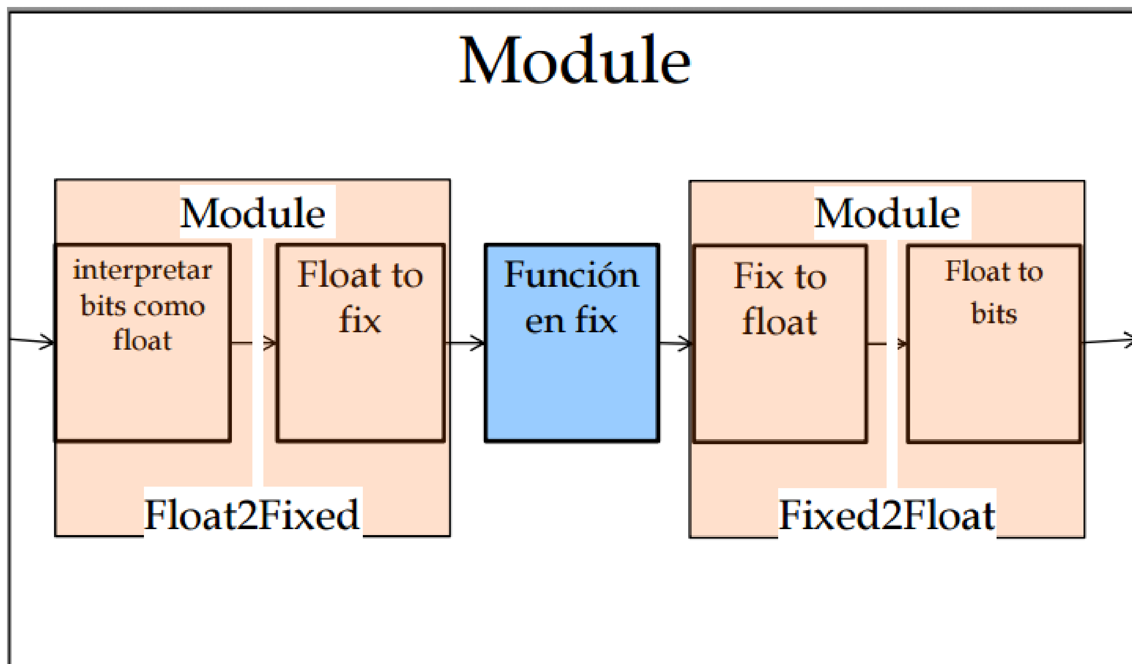



***Figure 5.2*** *The structure of the function tangent hyperbolic*

## 5.3 Results

In this part we will present the results on our study of Backprobagation algorithm with acceleration kernel on FPGA Cyclon 5 DE1-Soc with ARM CPU embedded, we have three sets of results depends on Block Size and Work items and they are

1- Block Size = 4 and Work Items =4
2- Block Size = 8 and Work Items =8
3- Block Size = 16 and Work Items =4

We start our study by applying our algorithm on ARM CPU without kernel and then we apply the algorithm on the Kernel in deferent parameter of Block Size and Work Items, we have three sets as mention before and in each set we apply the algorithm on kernel acceleration Matrix-multiplication than we apply it another time on kernel acceleration with Hyperbolic Tangent 'tanh'.

Our algorithm have 5 parameters, two fixe are `numInputs_MAX` and `numOutputs_MAX`, three variable are `numHidden1_MAX`, `numHidden2_MAX`, `numPatterns_MAX`, we vary the parameters `numHidden1_MAX` and `numHidden2_MAX` from 4 to 64, the `numPatterns_MAX` from 256 to 65536. And we have four results, CPU time, Forward time and backward time, we do all possibility by varying the parameter.

For each set of Work size and Work Items we have the following parameters:

```
numHidden1_MAX ={4, 8, 16, 32, 48, 64}

numHidden2_MAX ={4, 8, 16, 32, 48, 64}

numPatterns_MAX ={256, 1024, 4096, 16384, 65536}
```

From that parameter we have 6*6*5 =180 study cases so we three sets of Work Size and Work Items so in total we have 180*3 =540 study cases.

The first subset of our results is applying our algorithm Backpropagation on ARM CPU in FPGA, we got slow result and we give a sample about the results in the following table.

| numPatterns_MAX | 256 | 1024 | 4096 | 16384 | 65536 |
|---|---|---|---|---|---|
| numHidden1 | 8 | 32 | 32 | 64 | 64 |
| numHidden2 | 4 | 16 | 48 | 48 | 64 |
| CPU time | 0,98 | 16,58 | 199,46 | 1970,38 | 9257,73 |
| Forward time | 0,41 | 6,2 | 36,83 | 222,4 | 1005,28 |
| Backward time | 0,37 | 10,23 | 162,44 | 1747,68 | 8251,92 |

***Table 5.1*** *Sample Result of ARM CPU*

**First set (Block size = 4 and Work Items =4)**

The first subset we applying our algorithm backprobagation on Kernel Matrix-multiplication and we got better result than the previous subset.

A sample :

| numPatterns_MAX | 256 | 1024 | 4096 | 16384 | 65536 |
|---|---|---|---|---|---|
| numHidden1 | 16 | 8 | 48 | 32 | 64 |
| numHidden2 | 4 | 16 | 48 | 64 | 64 |
| CPU time | 1,87 | 5,87 | 51,24 | 210,68 | 1048,06 |
| Forward time | 1,05 | 3,55 | 35,51 | 140,69 | 717,26 |
| Backward time | 0,67 | 2,14 | 15,52 | 69,72 | 330,18 |

***Table 5.2*** *Sample Result of Kernel, BZ=4 and WI=4*

The second subset we applied our algorithm on the kenel Hyperbolic Tangent. We give a sample one in the following table:

| numPatterns_MAX | 256 | 1024 | 4096 | 16384 | 65536 |
|---|---|---|---|---|---|
| numHidden1 | 4 | 16 | 16 | 48 | 64 |
| numHidden2 | 8 | 16 | 32 | 8 | 64 |
| CPU time | 1,33 | 4,1 | 17,07 | 73,4 | 485,97 |
| Forward time | 0,46 | 1,56 | 6,44 | 25,96 | 160,41 |
| Backward time | 0,72 | 2,37 | 10,44 | 47,19 | 324,99 |

***Table 5.3*** *Sample Result of Kernel with 'Tanh', BZ=4 and WI=4*

### The second set (Block size = 8 and Work Items =8)

The first subset we applying our algorithm backpropagation on Kernel Matrix-multiplication.

A sample:

| numPatterns_MAX | 256 | 1024 | 4096 | 16384 | 65536 |
|---|---|---|---|---|---|
| numHidden1 | 4 | 8 | 32 | 48 | 64 |
| numHidden2 | 4 | 16 | 8 | 16 | 64 |
| CPU time | 1,77 | 6,09 | 29,39 | 154,51 | 1048,12 |
| Forward time | 0,78 | 3,5 | 18,95 | 102,24 | 718,53 |
| Backward time | 0,82 | 2,41 | 10,25 | 51,98 | 328,98 |

***Table 5.4*** *Sample Result of Kernel, BZ=8 and WI=8*

The second subset we applied our algorithm on the kernel Hyperbolic Tangent. We give a sample one in the following table:

| numPatterns_MAX | 256 | 1024 | 4096 | 16384 | 65536 |
|---|---|---|---|---|---|
| numHidden1 | 16 | 32 | 16 | 64 | 64 |
| numHidden2 | 8 | 32 | 48 | 8 | 64 |
| CPU time | 1,58 | 5,65 | 20,55 | 89,63 | 497,24 |
| Forward time | 0,54 | 2,17 | 7,32 | 31,03 | 164,74 |
| Backward time | 0,86 | 3,3 | 13,02 | 58,29 | 331,96 |

***Table 5.5*** *Sample Result of Kernel with 'Tanh', BZ=8 and WI=8*

**The third set (Block size = 16 and Work Items =4)**

The first subset we applying our algorithm backpropagation on Kernel Matrix-multiplication,

A sample:

| numPatterns_MAX | 256 | 1024 | 4096 | 16384 | 65536 |
|---|---|---|---|---|---|
| numHidden1 | 8 | 8 | 32 | 48 | 64 |
| numHidden2 | 4 | 8 | 48 | 64 | 64 |
| CPU time | 2,28 | 7,19 | 46,27 | 237,66 | 1039,09 |
| Forward time | 1,09 | 3,68 | 31,17 | 158,38 | 700,24 |
| Backward time | 1,02 | 3,33 | 14,89 | 79,01 | 338,26 |

***Table 5.6*** *Sample Result of Kernel with, BZ=16 and WI=4*

The second subset we applied our algorithm on the kernel Hyperbolic Tangent, we give a sample one in the following table:

| numPatterns_MAX | 256 | 1024 | 4096 | 16384 | 65536 |
|---|---|---|---|---|---|
| numHidden1 | 4 | 4 | 16 | 48 | 16 |
| numHidden2 | 4 | 16 | 32 | 48 | 16 |
| CPU time | 1,95 | 5,37 | 19,77 | 110,28 | 511,95 |
| Forward time | 0,71 | 1,99 | 7,24 | 36,56 | 167,24 |
| Backward time | 1,07 | 3,22 | 12,32 | 73,36 | 344,13 |

***Table 5.7*** *Sample Result of Kernel with 'Tanh', BZ=16 and WI=4*

## 5.3.1 Comparison

This part we will compare our result which case is better, so we have three hardware which are ARM CPU without kernel, kernel acceleration Matrix-multiplication and kernel acceleration Matrix-multiplication with Hyperbolic Tangent 'tanh'.

First we do a comparison about the hardware:

| | ARM-CPU | BZ=4, WI=4 | BZ=8, WI=8 | BZ=16, WI=4 |
|---|---|---|---|---|
| **Logic utilization (in ALMs)** | | 14,328 / 32,070 ( 45 % ) | 26,978 / 32,070(84% ) | 25,536 / 32,070 ( 80 % ) |
| **Total registers** | | 27659 | 56719 | 49801 |
| **Total pins** | | 103 / 457 ( 23 % ) | 103 / 457 ( 23 % ) | 103 / 457 ( 23 % ) |
| **Total block memory bits** | | 783,992 / 4,065,280(19%) | 961,720 / 4,065,280(24%) | 1,327,024 / 4,065,280(33% ) |
| **Total DSP Blocks** | | 26 / 87 ( 30 % ) | 74 / 87 ( 85 % ) | 74 / 87 ( 85 % ) |
| **Total PLLs** | | 2 / 6 ( 33 % ) | 2 / 6 ( 33 % ) | 2 / 6 ( 33 % ) |
| **Total DLLs** | | 1 / 4 ( 25 % ) | 1 / 4 ( 25 % ) | 1 / 4 ( 25 % ) |
| **Total FPGA Thermal Power Dissipation** | | 1131.78 mW | 1637.59 mW | 1588.03 mW |
| **Core Dynamic Thermal Power Dissipation** | | 650.91 mW | 1152.11 mW | 1102.71 mW |
| **Core Static Thermal Power Dissipation** | | 433.79 mW | 438.40 mW | 438.24 mW |
| **I/O Thermal Power Dissipation** | | 47.08 mW | 47.08 mW | 47.08 mW |
| **HPS Dynamic (Dual core) Power** | 1392.92 mW | 1392.92 mW | 1392.92 mW | 1392.92 mW |
| **HPS Dynamic (Single core) Power** | 1247.25 mW | 1247.25 mW | 1247.25 mW | 1247.25 mW |
| **Total FPGA and HPS Power** | | 2524.70 mW | 3030.51 mW | 2980.95 mW |

***Table 5.8*** *Hardware Comparison, Just a Kernel*

| | ARM-CPU | BZ=4, WI=4 | BZ=8, WI=8 | BZ=16, WI=4 |
|---|---|---|---|---|
| **Logic utilization (in ALMs)** | | 15,047 / 32,070 ( 47 % ) | 28,327 /32,070 (88%) | 26,241/32,070 ( 82 % ) |
| **Total registers** | | 28690 | 58763 | 50914 |
| **Total pins** | | 103 / 457 ( 23 % ) | 103 / 457 ( 23 % ) | 103 / 457 ( 23 % ) |
| **Total block memory bits** | | 783,992 / 4,065,280 (19%) | 961,720 / 4,065,280(24%) | 1,327,024 / 4,065,280 ( 33 % ) |
| **Total DSP Blocks** | | 26/87 (30 %) | 74 / 87 ( 85 % ) | 74 / 87 ( 85 % ) |
| **Total PLLs** | | 2 / 6 ( 33 % ) | 2 / 6 ( 33 % ) | 2 / 6 ( 33 % ) |
| **Total DLLs** | | 1 / 4 ( 25 % ) | 1 / 4 ( 25 % ) | 1 / 4 ( 25 % ) |
| **Total FPGA Thermal Power Dissipation** | | 1146.95 mW | 1660.86 mW | 1602.18 mW |
| **Core Dynamic Thermal Power Dissipation** | | 665.98 mW | 1175.21 mW | 1116.76 mW |
| **Core Static Thermal Power Dissipation** | | 433.90 mW | 438.57 mW | 438.34 mW |
| **I/O Thermal Power Dissipation** | | 47.08 mW | 47.08 mW | 47.08 mW |
| **HPS Dynamic (Dual core) Power** | 1392.92 mW | 1392.92 mW | 1392.92 mW | 1392.92 mW |
| **HPS Dynamic (Single core) Power** | 1247.25 mW | 1247.25 mW | 1247.25 mW | 1247.25 mW |
| **Total FPGA and HPS Power** | | 2539.87 mW | 3053.78 mW | 2995.10 mW |

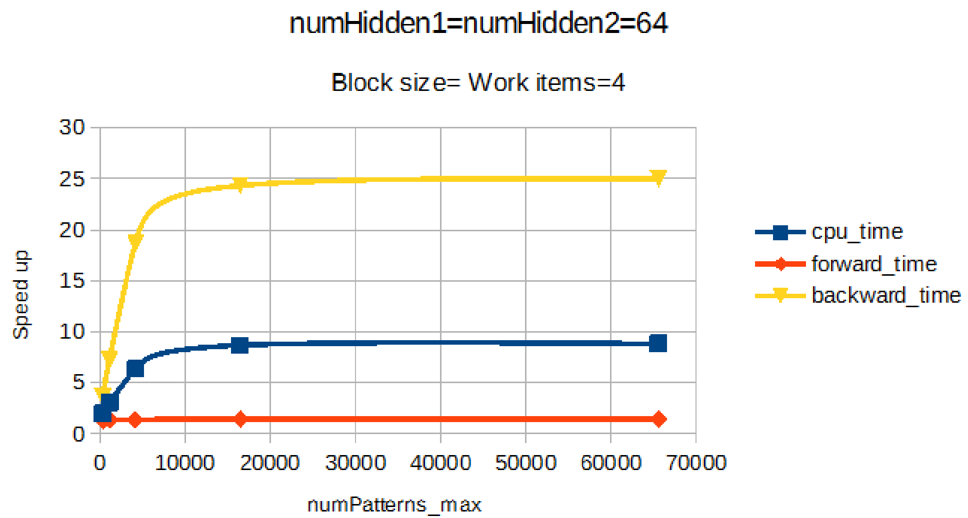
***Table 5.9*** *Hardware Comparison, Kernel with ‘tanh’*

We tested our Kernel in many case and we get results then we present this result in graph to compare it with the results on ARM CPU, for every case we try to fix one parameter and varying others, we have three parameter numHidden1, numHidden2 and numPatterns. We choose the best case to draw the graphs.

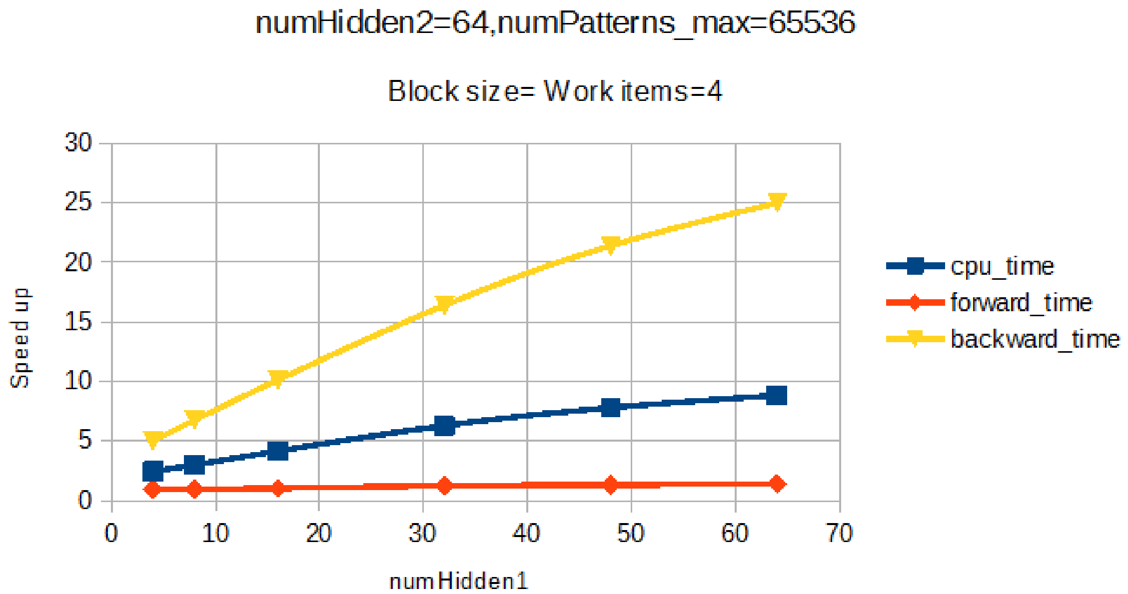
So let see what we got:

## ❖ Kernel acceleration Matrix-multiplication vs ARM CPU

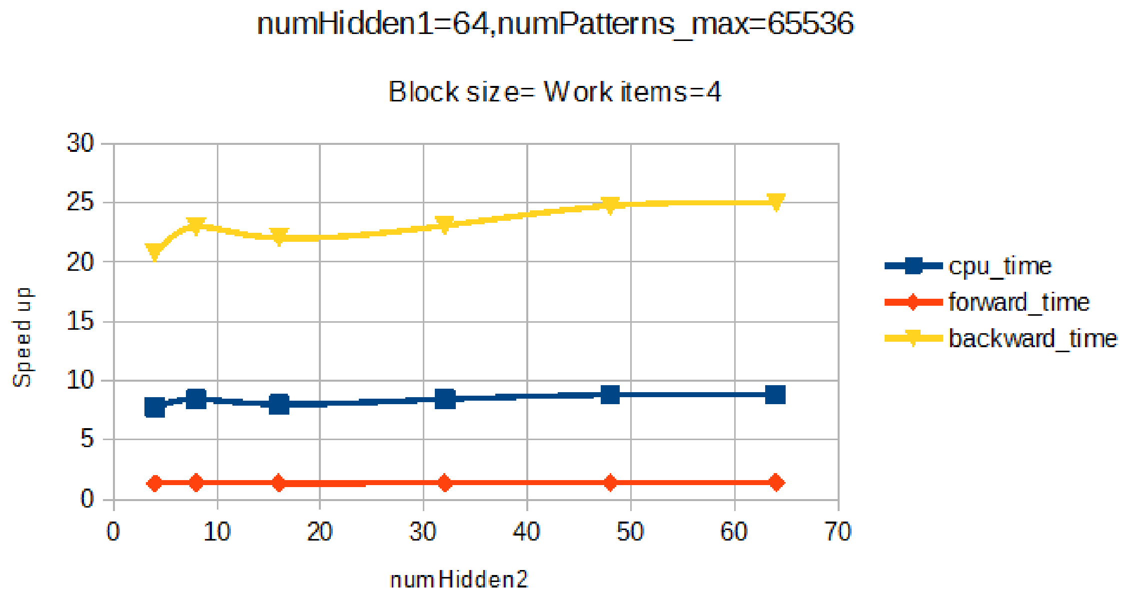
➢ First set (Block size = 4 and Work Items =4)




***Graph 5.1*** *Comparison Kernel VS ARM CPU, BZ=4 WI=4, numPatterns_max is Variable*

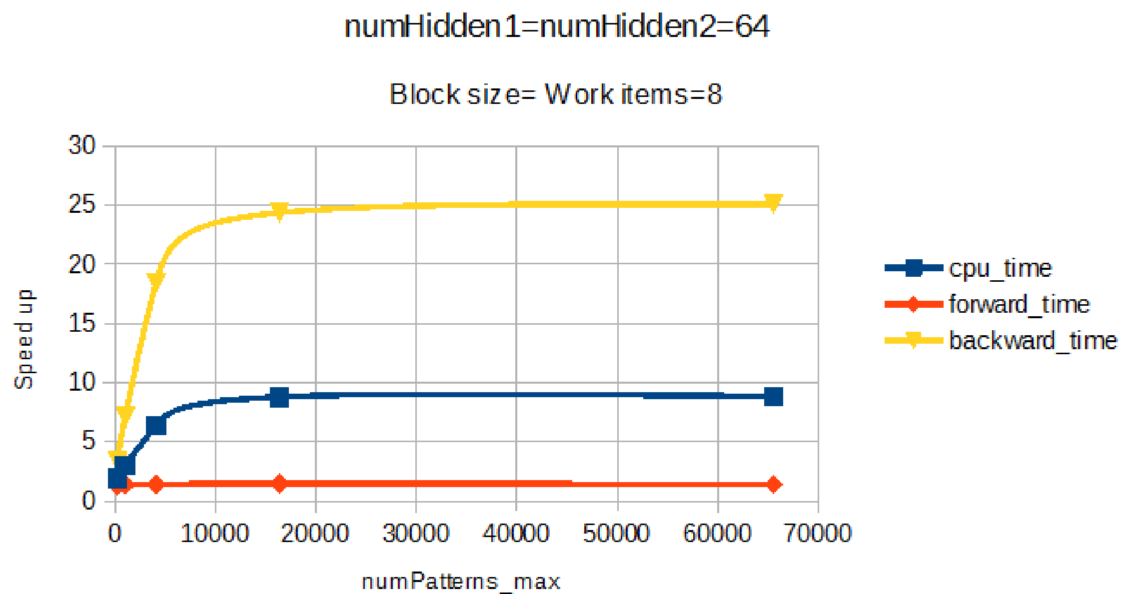



***Graph 5.2*** *Comparison Kernel VS ARM CPU, BZ=4 WI=4, numHidden1 is Variable*

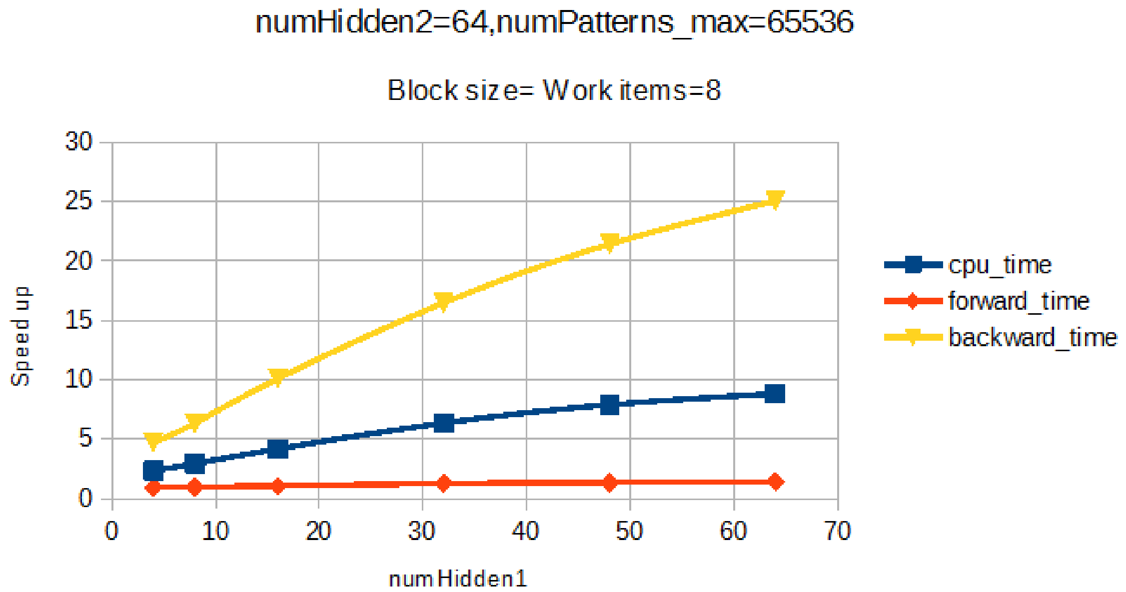



***Graph 5.3*** *Comparison Kernel VS ARM CPU, BZ=4 WI=4, numHidden2 is Variable*

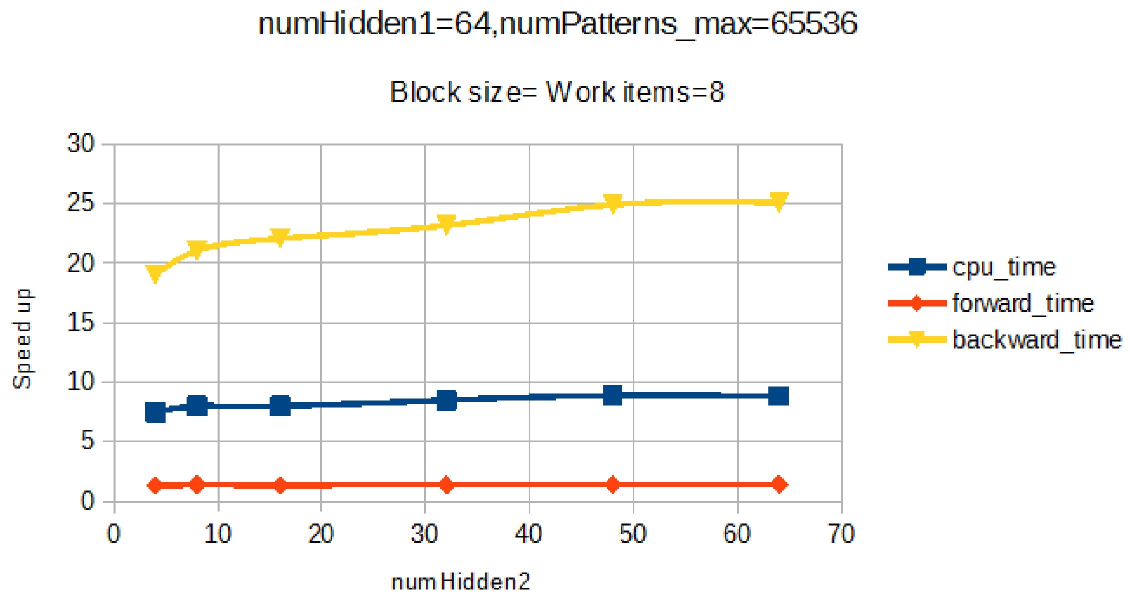
- Second set (Block size = 8 and Work Items =8)




***Graph 5.4*** *Comparison Kernel VS ARM CPU, BZ=8 WI=8, numPatterns_max is Variable*

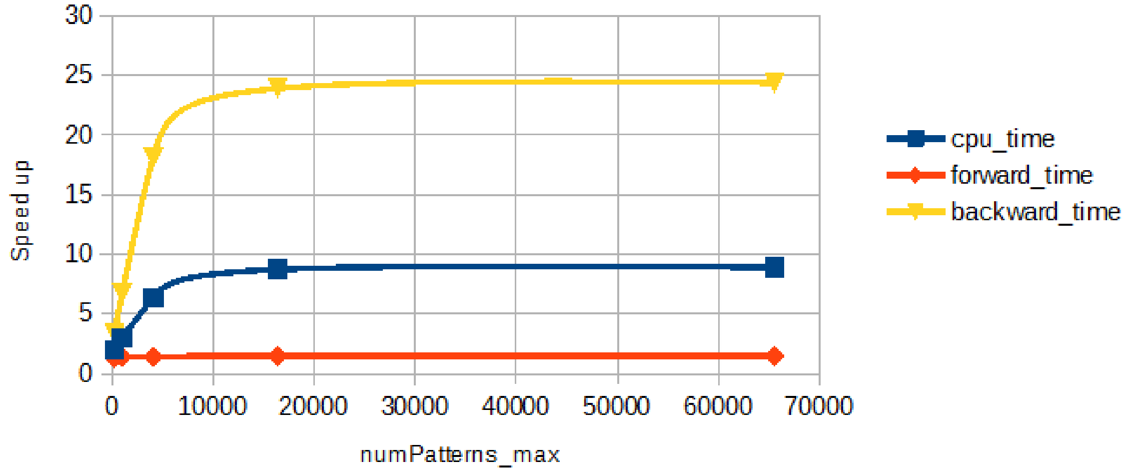



***Graph 5.5*** *Comparison Kernel VS ARM CPU, BZ=8 WI=8, numHidden1 is Variable*

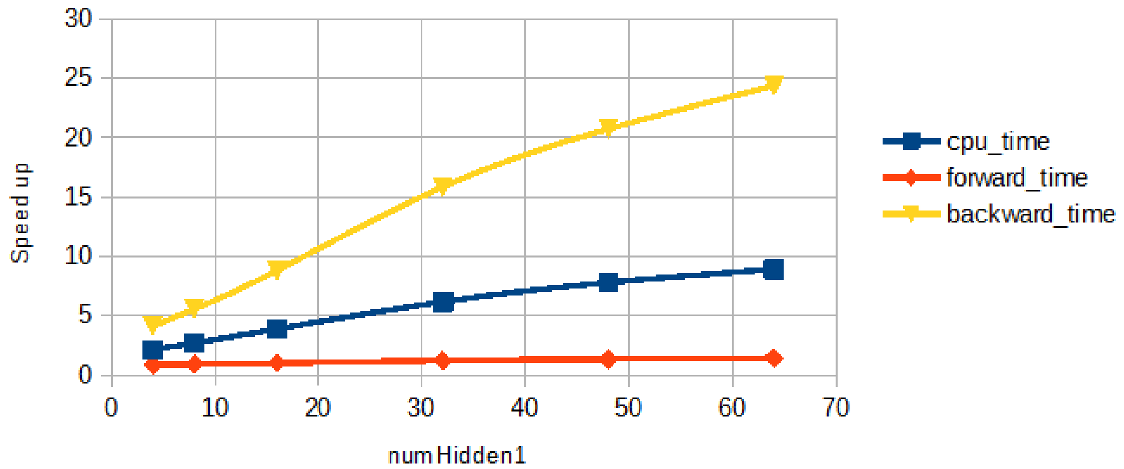



***Graph 5.6*** *Comparison Kernel VS ARM CPU, BZ=8 WI=8, numHidden2 is Variable*

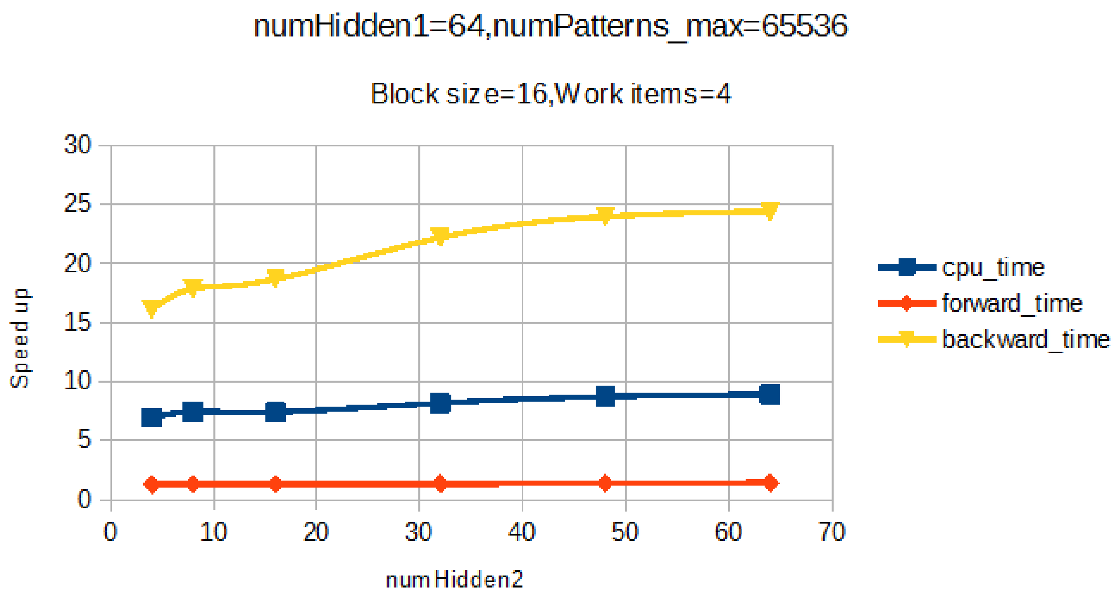
- Third set (Block size = 16 and Work Items =4)




***Graph 5.7*** *Comparison Kernel VS ARM CPU, BZ=16 WI=4, numPatterns_max is Variable*



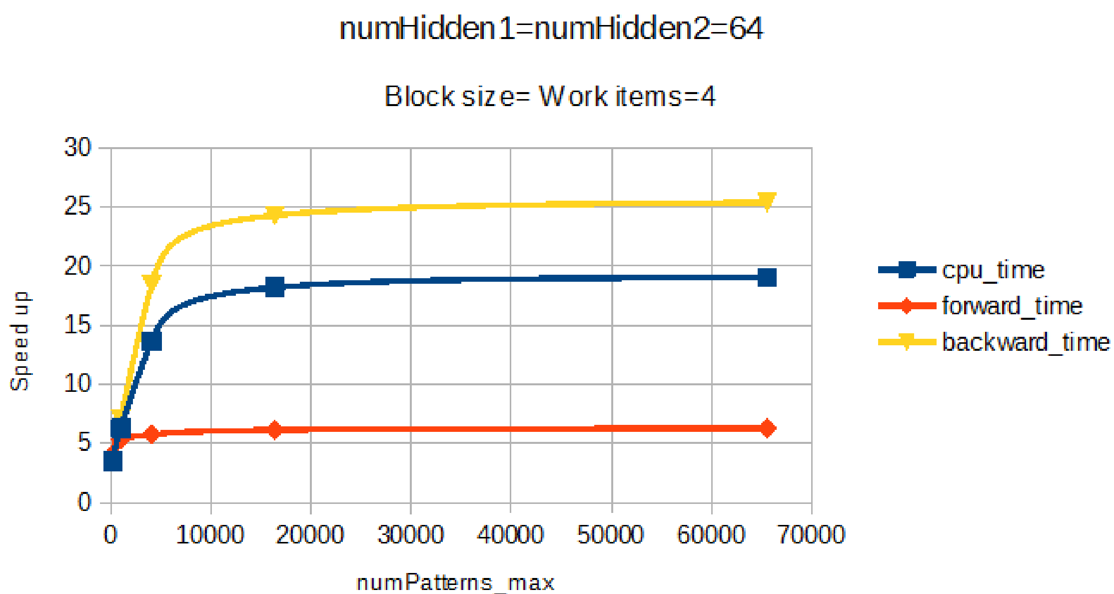

***Graph 5.8*** *Comparison Kernel VS ARM CPU, BZ=16 WI=4, numHidden1 is Variable*



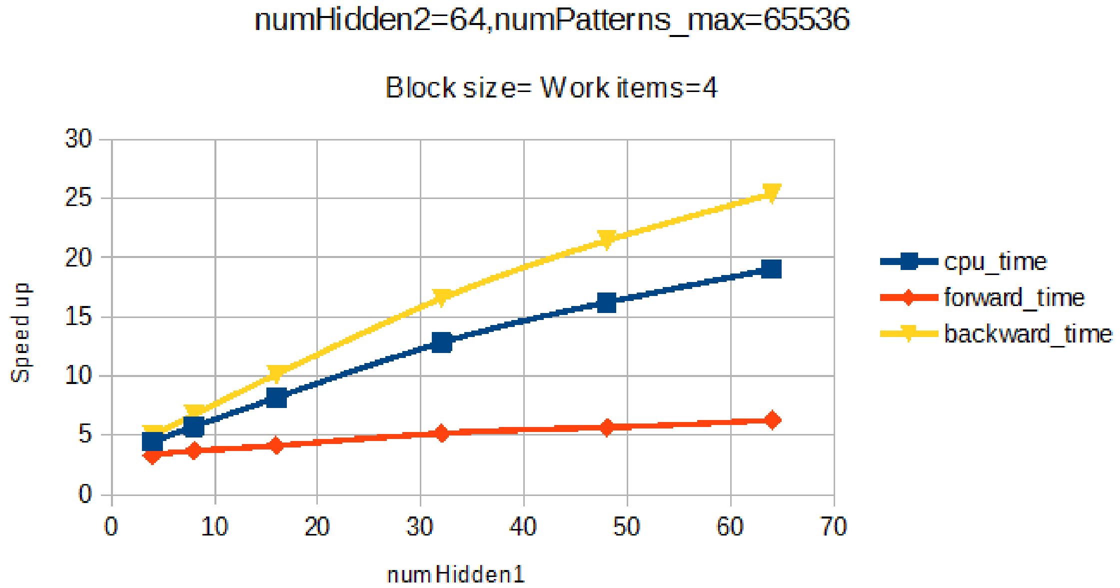

***Graph 5.9*** *Comparison Kernel VS ARM CPU, BZ=16 WI=4, numHidden1 is Variable*

- **kernel acceleration Matrix-multiplication with Tanh vs ARM CPU**

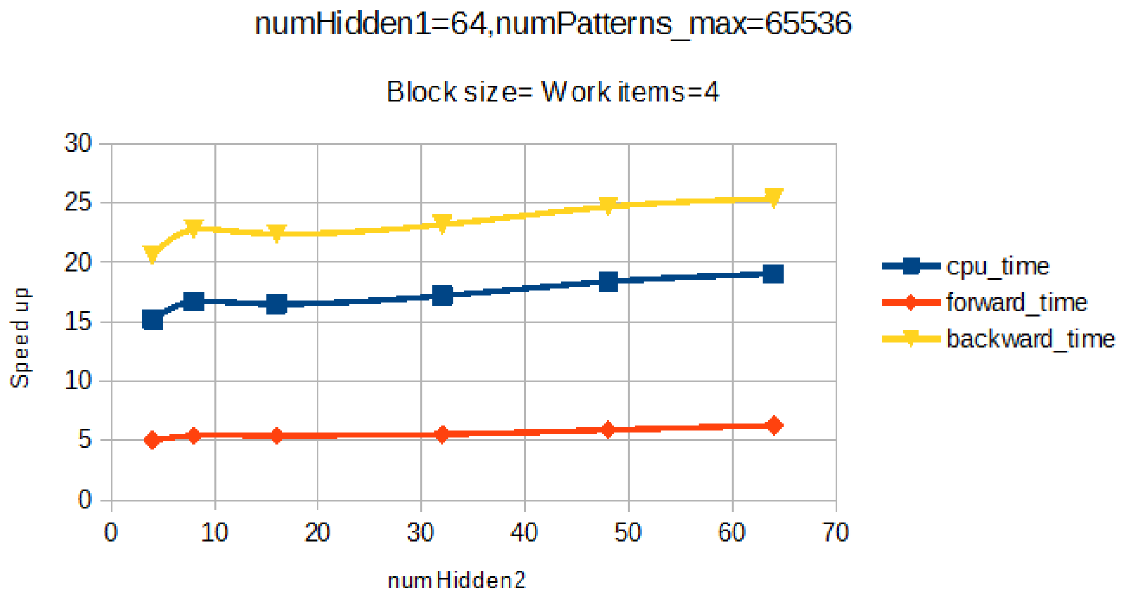
- First set (Block size = 4 and Work Items =4)




***Graph 5.10*** *Comparison Kernel with 'tanh' VS ARM CPU, BZ=4 WI=4, numPatterns_max is Variable*

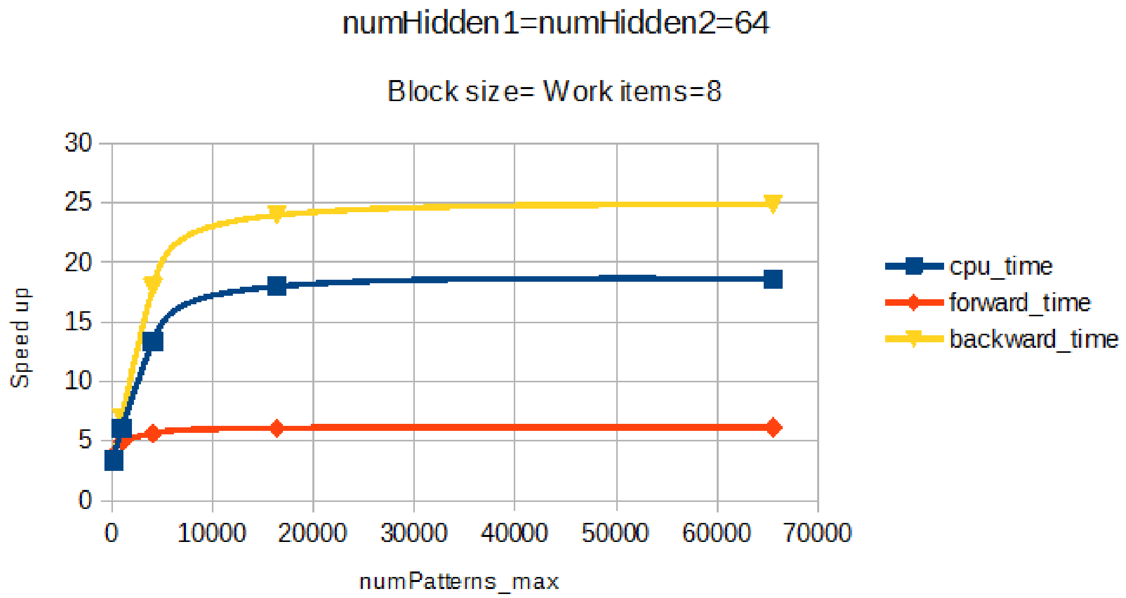



***Graph 5.11*** *Comparison Kernel with 'tanh' VS ARM CPU, BZ=4 WI=4, numHidden1 is Variable*

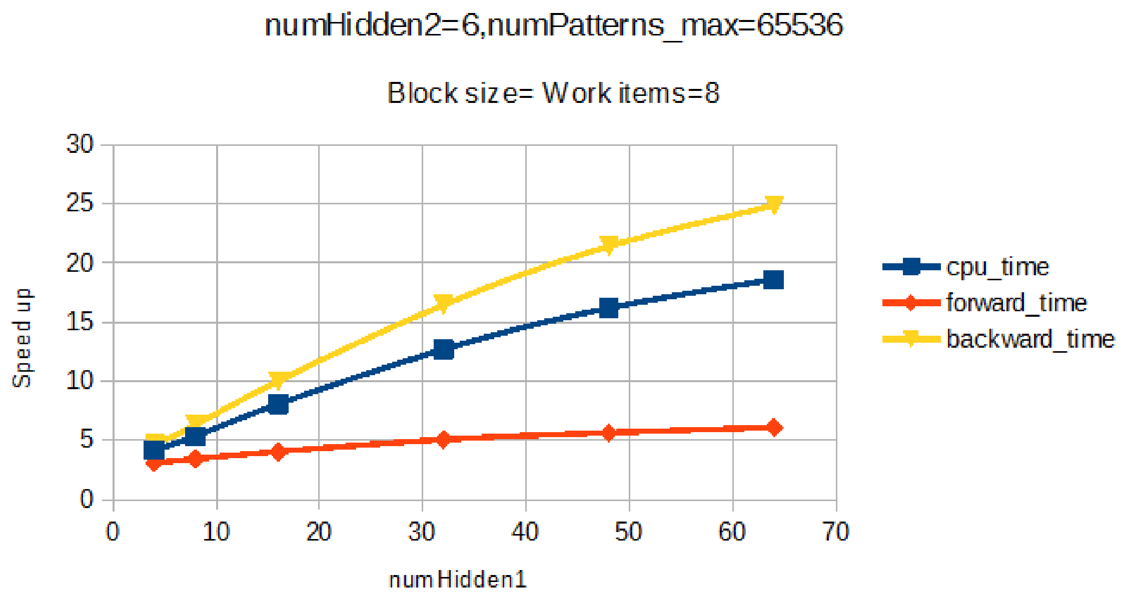



***Graph 5.12*** *Comparison Kernel with 'tanh' VS ARM CPU, BZ=4 WI=4, numHidden2 is Variable*

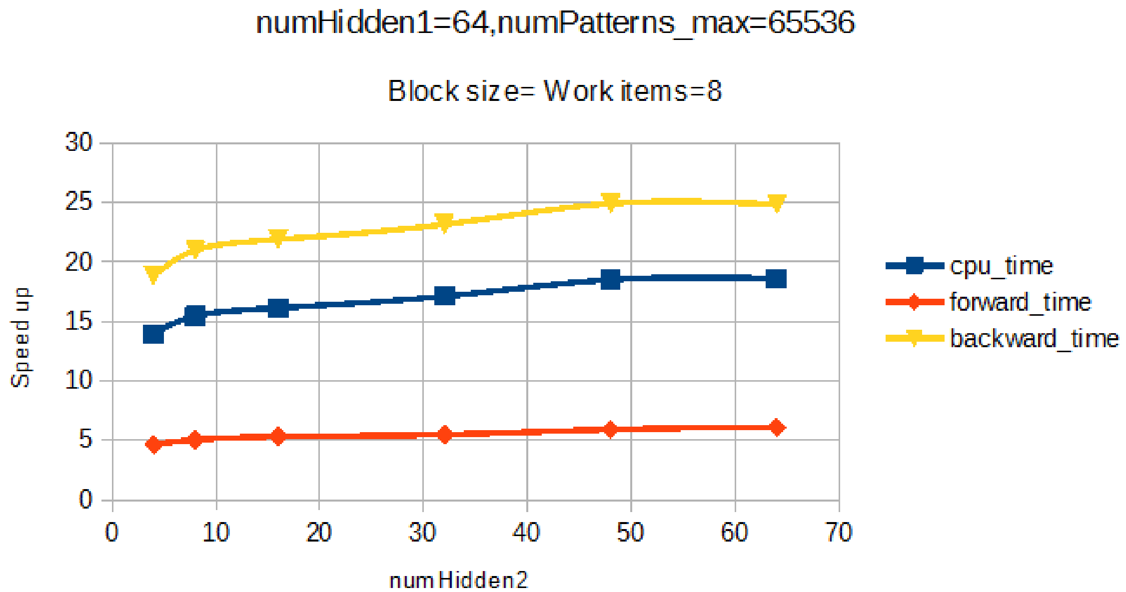
➢ Second set (Block size = 8 and Work Items =8)




***Graph 5.13*** *Comparison Kernel with 'tanh' VS ARM CPU, BZ=8 WI=8, numPatterns_max is Variable*

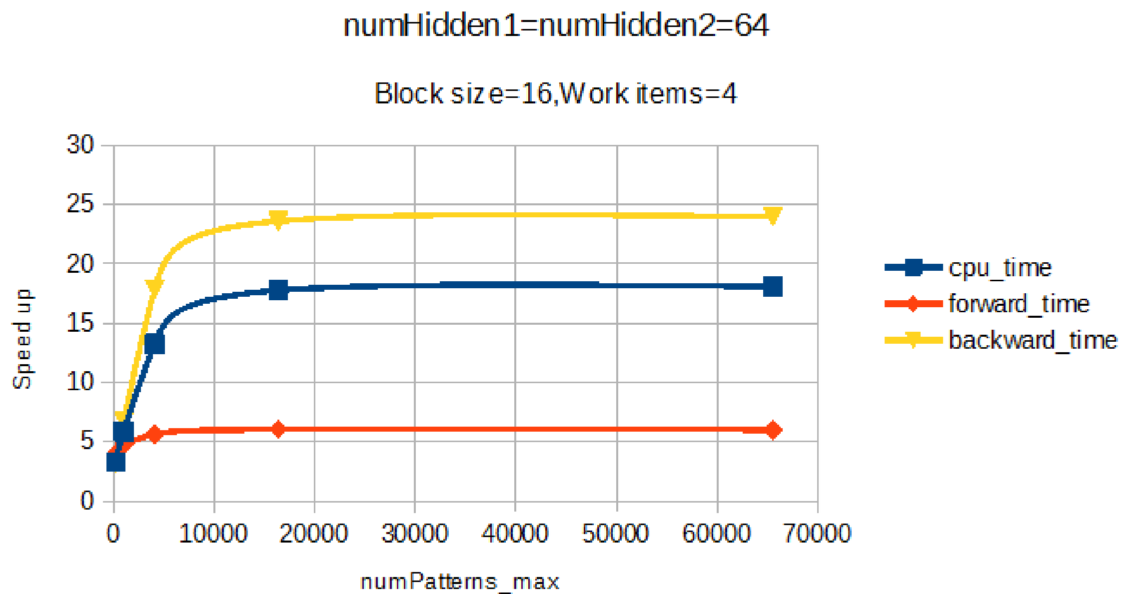



***Graph 5.14*** *Comparison Kernel with 'tanh' VS ARM CPU, BZ=8 WI=8, numHidden1 is Variable*

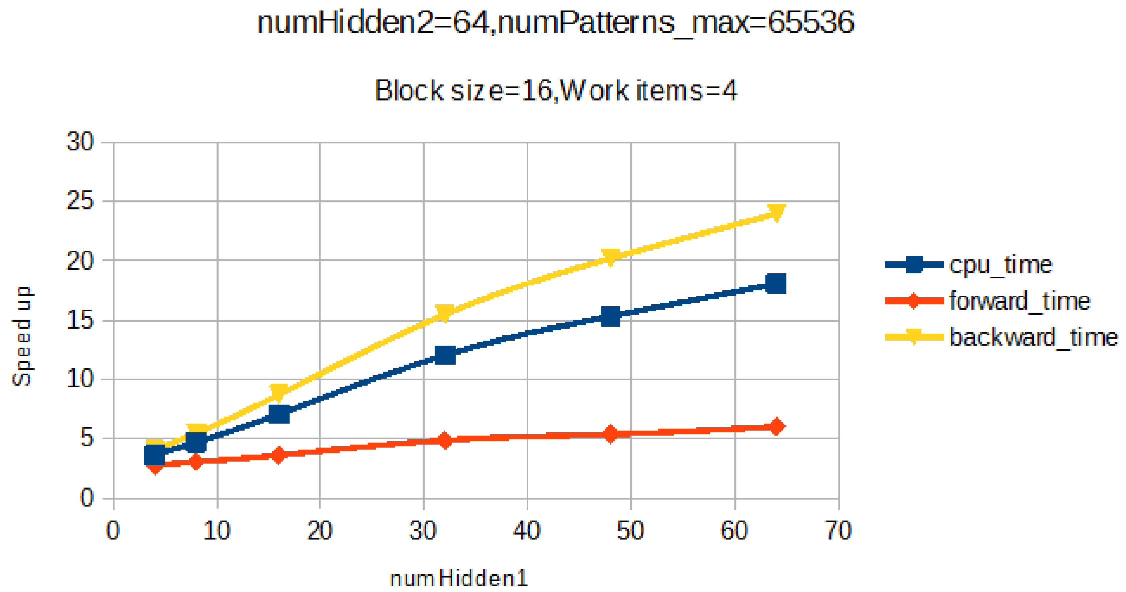



***Graph 5.15*** *Comparison Kernel with 'tanh' VS ARM CPU, BZ=8 WI=8, numHidden2 is Variable*

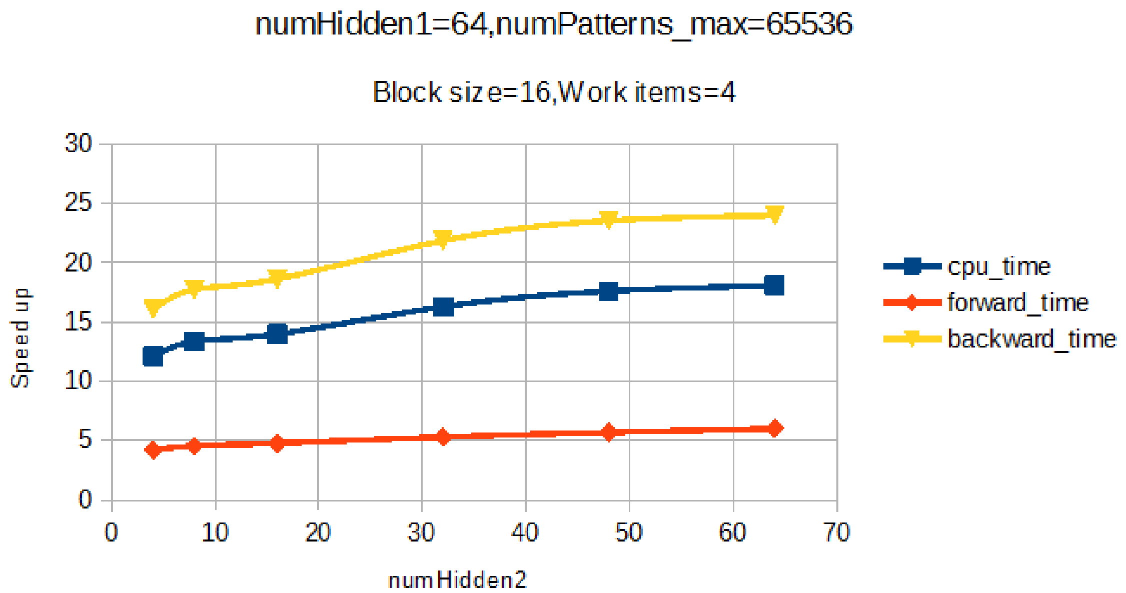
- Third set (Block size = 16 and Work Items =4)




***Graph 5.16*** *Comparison Kernel with 'tanh' VS ARM CPU, BZ=16 WI=4, numPatterns_max is Variable*




***Graph 5.17*** *Comparison Kernel with 'tanh' VS ARM CPU, BZ=16 WI=4, numHidden1 is Variable*




***Graph 5.18*** *Comparison Kernel with 'tanh' VS ARM CPU, BZ=16 WI=4, numHidden2 is Variable*

### 5.3.2 Comments

As we see in the graphs, when we varying the parameter numPatterns_max at first increase so quickly and in one point the acceleration will stop and this point around 20000, when we varying the parameter numHidden1 we see the acceleration get increase when we increase numHidden1, and for the last parameter numHidden2 we see its effect not that much as others parameters just increase little bit each value.

For the best case, the best hardware is kernel acceleration Matrix-multiplication with Tanh, it is the faster and for Work Size and Work Items almost are the same but we can see the little different, for Work Size = 4 and Work Items = 4 is the best one and we have the parameters numPatterns_max=65536 , numHidden64= and numHidden2=64 is the best case and we got 9257.73/485.97= 19.05 times faster for CPU time, 1005.28/160.41= 6.27 times faster for Forward time and 8251.92/324.99=25.39 times faster for Backward time.

# Conclusion

We started our thesis with general description of OpenCL and we spoke about the conceptual of OpenCL such as the general structure, execution model, memory model and platform model. In the seconde chapter we described in detail the use of Opencl with Altera SDK, also here we spoke about the FPGA OpenCL architecture like in previous chapter just we specified OpenCL with FPGA.

In chapter three we had an example about the implementation and optimization of the kernel OpenCL code for acceleration of Matrix-multiplication with the host code to execute in the kernel, in the chapter four we talked about the acceleration of algorithm Backprobagatin that use the kernel of matrix-multiplication, and finally we presented the result that we got in our study with take some graph to see more clear about the changing of the results.

We applied the Algorithm of matrix-multiplication in our project on three deferent hardware which are ARM CPU, Kernel acceleration and Kernel acceleration with Hyperbolic Tangent, and with deferent parameters. We get result faster and better in kernel acceleration and especially Kernel acceleration with Hyperbolic Tangent where we get our best result with the parameter numPatterns_max = 65536 , numHidden1 = 64, numHidden2 = 64, Work Size = 4 and Work Items = 4, in that case we get 19.05 times faster for CPU time and that result considered as a good speed up.

# REFERENCES


[1] Aaftab Munshi, Benedict R. Gaster, Timothy G. Mattson, James Fung, Dan Ginsburg. OpenCL Programming Guide 2011.
http://www.it-ebooks.info/book/1602/
[access 02/2016]

[2] Aaftab Munshi. The OpenCL Specification version: 1.0, 12/08/2008.
https://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf
[access 02/2016]

[3] Cedric Nugteren. Tutorial: OpenCL SGEMM tuning for Kepler 2014.
http://www.cedricnugteren.nl/tutorial.php?page=1
[access 02/2016]

[4] Altera Corporation. Implementing FPGA Design with the OpenCL Standard November 2013.
https://www.altera.com/en_US/pdfs/literature/wp/wp-01173-opencl.pdf
[access 03/2016]

[5] Altera Corporation. Altera SDK for OpenCL Best Practices Guide 04/05/2015.
https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl_optimization_guide.pdf
[access 03/2016]

[6] Tomasz S. Czajkowski and others. OpenCL for FPGAs: Prototyping a Compiler 2012.
http://ersaconf.org/ersa12/papers/Brown-opencl-for-fpgas.pdf
[access 03/2016]

[7] Altera Corporation. Altera SDK for OpenCL Best Practices Guide 02/05/2016.
https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf
[access 06/2016]

[8] Acceleration of optimization of neural networks throught on SoC with OpenCL. Rafael Gadea-Gironés, Jorge Fe. 03/2010.