Reducing Computational Complexities Using GPU and FPGA

M.Tech Thesis

By

HARSHIT VERMA

2002102019

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY INDORE

June 2022

Reducing Computational Complexities Using GPU and FPGA

A THESIS

Submitted in fulfillment of the
requirements for the award of the degree
of
Master of Technology

By
HARSHIT VERMA

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY INDORE

June 2022

INDIAN INSTITUTE OF TECHNOLOGY INDORE

CANDIDATE'S DECLARATION

I hereby certify that the work which is being presented in the thesis entitled **Reducing Computational Complexities Using GPU and FPGA** in the partial fulfillment of the requirements for the award of the degree of **MASTER OF TECHNOLOGY** and submitted in the **DEPARTMENT OF ELECTRICAL ENGINEERING, Indian Institute of Technology Indore**, is an authentic record of my own work carried out during the time period from August 2020 to June 2022 under the supervision of **Dr. Srivathsan Vasudevan, Associate Professor, Department of Electrical Engineering, Indian Institute of Technology Indore**, and **Dr. Satya S. Bulusu, Associate Professor, Department of Chemistry, Indian Institute of Technology Indore**.

The matter presented in this thesis has not been submitted by me for the award of any other degree at this or any other institute.

31/05/2022
Signature of the student with date
HARSHIT VERMA

This is to certify that the above statement made by the candidate is correct to the best of our knowledge.

31/5/22
Signature of the Supervisor of M.Tech. thesis (with date)
Dr. Srivathsan Vasudevan

31/05/2022
Signature of the Supervisor of M.Tech. thesis (with date)
Dr. Satya S. Bulusu

Mr. Harshit Verma has successfully given his M.Tech. Oral Examination held on **7th June 2022.**

07/06/2022
Signature(s) of Supervisor(s) of M.Tech. thesis
Date: 7/6/22

Convener, DPGC
Date: 07/06/2022

Signature of PSPC Member #1
Date: 07/06/2022

Signature of PSPC Member #2
Date: 09/06/2022

ACKNOWLEDGEMENT

I wish to thank my supervisors, Dr. Srivathsan Vasudevan, Associate Professor, IIT Indore, and Dr. Satya S. Bulusu, Associate Professor, IIT Indore for their patience, guidance, and support. I have benefited greatly from their wealth of knowledge. I am extremely grateful that they took me on as a student and continued to have faith in me throughout the year.

I sincerely thank my PSPC members, Prof. Santosh Kumar Vishvakarma, and Dr. Devendra Deshmukh, for their useful suggestions and valuable remarks during M.Tech project work. Their encouraging words and thoughtful, detailed feedback have been very important to me. I also acknowledge IIT Indore for providing me the necessary infrastructure and research facilities for the project work. I am thankful to all the faculty members of the Discipline of Electrical Engineering for their kind support during my M.Tech work.

Besides, I would not forget Mrs. Kritika Bhardwaj for investing all her valuable time with me. Finally, I owe a lot to my family, friends, and classmates for always encouraging and supporting me to stay motivated and focused on completing the project.

HARSHIT VERMA

Dedicated to my family

Abstract

REDUCING COMPUTATIONAL COMPLEXITIES USING GPU AND FPGA

Computationally intensive applications such as weather forecasting, computational biology, Molecular Dynamics etc. requires a lot of time to execute a task, because usually these tasks are sequentially executed. Because the iterative nature of these algorithms on a single CPU can take up a lot of computation time for relatively basic simulations, there's a lot of pressure to find ways to improve efficiency. As a result of present processor technology's clock restrictions, improved computer performance is becoming increasingly reliant on parallelism.

This project deals with exploring the different ways in which parallelism can be inherited for calculations of Interatomic potentials of gold nanoparticle. Phase 1 of the research focuses on computing on GPUs, which can drastically speed up computer applications by leveraging GPU capabilities. The sequential component of the workload in GPU-accelerated applications operates on the CPU, which is tuned for single-threaded speed, while the compute-intensive portion of the application runs in parallel on thousands of GPU cores. A portion of complex algorithm is thereby studied and parallelized and subsequent comparative analysis is done to observe whether the results are obtained accordingly or not.

Phase 2 of the project deals with the computations being carried out on FPGA and establishing communication between PC and FPGA using PCIe protocol. Input data from host PC is first sent to FPGA where various calculations such as force, energy values is being done and these values are then sent back from FPGA to host PC and this process is done iteratively by defining the number of steps. Finally, the total time taken in these steps is calculated. The primary goal of our work is to reduce the computational time taken by sequential C code of complex algorithm with the help of Parallelism using GPU and FPGA.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

API	*Application Programming Interface*
ASIC	*Application Specific Integrated Circuit*
AXI	*Advanced Extensible Interface*
CAD	*Computer Aided Design*
CPI	*Cycles Per Instruction*
CPU	*Central Processing Unit*
CUDA	*Compute Unified Device Architecture*
DDR	*Double Data Rate*
DMA	*Direct Memory Access*
FPGAs	*Field Programmable Gate Arrays*
GPGPU	*General Purpose Computing on Graphics Processing Unit*
GPUs	*Graphics Processing Units*
GUI	*Graphical User Interface*
H2C/C2H	*Host to Card/Card to Host*
HDL	*Hardware Description Language*
HLS	*High Level Synthesis*
IP	*Intellectual Property*
MD	*Molecular Dynamics*
PCIe	*Peripheral Component Interconnect Express*
PGA	*Professional Graphics Adapter*
RTL	*Register Transfer Level*
SFUs	*Special Function Units*
SGI	*Silicon Graphics Inc.*
SIMD	*Single Instruction Multiple Data*
SIMT	*Single Instruction Multiple Thread*
SM	*Streaming Multiprocessor*
SP	*Scalar Processor*

Chapter 1

Introduction

1.1 Background

The size and volume of data that enterprises must cope with is expanding quickly as technology advances. As a result, a high-performance computing architecture is required. High performance computing is a rapidly growing field in computer and electronics engineering. These high-performance computing algorithms can be run in parallel on a variety of processing devices and then combined to provide the desired output owing to parallelism. Parallel algorithms are extremely effective for processing large amounts of data in a short amount of time. An algorithm's analysis allows us to assess whether or not the algorithm is useful. In general, an algorithm is evaluated based on how long it takes to execute (Time Complexity) and how much space it takes up (Space Complexity). Storage space is no longer an issue because sophisticated memory devices are now available at cheap prices. Thereby, the main focus here is on reducing the computation time.

1.2 Motivation

Due to increased power consumption, frequency scaling came to an end in 2004. Moore’s law is still in effect and is facilitating parallel scaling [1]. This has caused a shift to parallel multi-core computing to increase performance. There are a wide variety of technologies that can be used for acceleration which have dedicated logic for specific workloads. These include Graphics Processing Units (GPUs) [2] and Field-Programmable Gate Arrays (FPGAs) [3]. Reconfigurable and parallel computing is a way to perform high performance computing. Suppose we want to perform 1 million tasks, CPU will do this work in a sequential manner and take much time. But, GPU utilizing its thousands of cores at a time takes much lesser time. Furthermore, because we can incorporate numerous parallel execution units into an FPGA, we may distribute data across the parallel execution units and

acquire the desired computational results in much less time if we have a big volume of data to flow through the same algorithm.

1.3 Objective

The main objective of this project is to reduce the computation time taken by executing complex C algorithm based on interatomic potential of gold nanoparticle. Parameters such as Forces, Energy, and Distances are being calculated by this algorithm. We will show how GPU (Graphics Processing Unit) & FPGA (Field Programmable Gate Array) can be used efficiently in High performance computing applications. For calculating force values, we need both radial and angular description of that particle [4]. The major computation time of force calculation is taken by calculating angular description. Thereby, a portion of this algorithm till calculation of angular descriptor is studied and parallelized using GPU. Subsequently, these results are then compared with their CPU counterpart results.

In this project, it is also proposed to use FPGA. The application of directives (to reduce latency) [5] is possible with the help of Vivado High Level Synthesis (HLS). Vivado HLS is software provided by Xilinx which converts high level language(C, C++) to Hardware Description Language (HDL). With the help of Amdahl's law, speed up is calculated with and without application of directives.

In the present work, Xilinx VIVADO (2017.2) software is used and HLS IP for calculating force and energy values is exported to VIVADO for completion of block design which will be dumped on to FPGA board (Kintex KC 705). Then, communication is established between host PC and FPGA board using PCIe (peripheral component interconnect express) protocol. Input values from PC are being sent to FPGA which calculates the force, energy values and send back these values to PC. Finally, total time taken in the entire process is calculated.

1.4 Organization of the thesis

This section describes the organization of thesis. The thesis is sub- divided into five chapters and presented as follows:

Chapter 1 provides the background, motivation, and objective of the project.

Chapter 2 contains the introduction to GPU, architecture of GPU, FPGA, its history and details about the FPGA board used in the project.

Chapter 3 describes the methodology used to implement the project work.

Chapter 4 provides the results obtained and its related discussion.

Chapter 5 presents the project conclusion and scope for future work.

(This page is left blank intentionally)

Chapter 2

Literature Survey

2.1 Parallel Computation and Graphics Processing Unit

In general, computing speed is often gauged by the number of clock pulses needed to complete an instruction, the so-called "cycles-per instruction" (CPI). Although a variety of techniques are used to achieve instruction-level and data-level parallelism, and hence improve CPI, until the turn of the century the most popular means among commodity microprocessors of improving computing speed was to increase the clock frequency. Due to heat dissipation and energy consumption issues, however, this strategy eventually became untenable. The number of tasks that could be processed in each clock period within a practical power budget stopped improving at its previous rate. Processor developers responded to this circumstance by designing units with multiple processors or cores per die and thus offered the possibility of executing instructions in parallel. IBM's POWER4 [6], released in 2002, is the first commercial processor to explore this. Today, two different families of processors are marketed: multi core and many cores. In a multi core approach, a few cores (typically two to ten at present) are integrated into a single microprocessor chip with the intention of speeding up the execution of the programs traditionally run on commodity machines including personal computers. In a many core approach, several hundred & thousands cores (each with a limited computational ability and lower power needs) are oriented in such a way that maximizes the throughput of traditionally parallel problems. The GPU falls into the many core category and has been adopted as the parallel platform of focus in this thesis.

When it was first conceived, the Graphics Processing Unit (GPU) was designed for real-time graphics, as we all know. A modern GPU, on the other hand, is not only a strong graphics processor, but also a general-purpose computing processor that focuses on parallel processing and high data bandwidth. Since the

clear slowing of CPU speed over the last decade, more people are turning to GPU for general-purpose computing, and it has become a popular topic since 2011. GPU became a very promising rival in high performance computing with rapid increases in both computation power and programmability. GPUs are multi-core processors that can run a large number of threads. SIMD processors are energy efficient because they only fetch, decode, and dispatch one instruction per a vector of processing elements [7]. A GPU's multiple threads keep it versatile, or at least flexible enough to handle a wide range of calculations. GPGPU stands for "General Purpose Computing on Graphics Processing Units."

Graphics processors began as fixed-function display controllers, offloading the rendering of a computer screen from a general-purpose CPU. Originally graphics chips could only draw lines, arcs, and geometrical shapes like circles, rectangles, as well as character bitmaps in 2D. Many rendering stages, particularly those involving computations in floating point, were executed on a CPU initially. At later stage the graphics processors could also render 3D images, which are used mainly in computer games. With technological advancements, graphics chips became capable of performing many steps of the rendering process on their own. The NVIDIA GeForce 256 was released around the beginning of the century, and it was the first graphics processor to be named a GPU. It could perform all of the geometry calculations independently; eliminating the need for the host CPU's floating point computations.

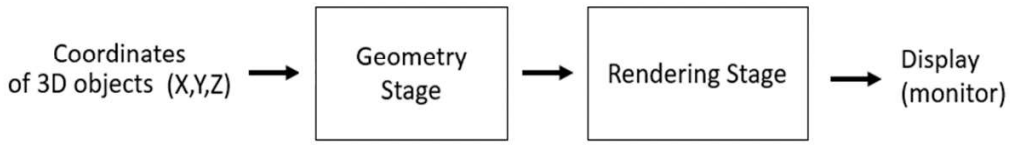


Fig. 2.1: Simplified 2 stage graphics pipeline

Fixed processing pipelines were used on the early GPUs. Each stage of the pipeline had its own set of functions, which were implemented in specialized hardware. The graphics pipeline is a diagram that depicts the flow of graphics data via multiple stages. It's basically a method of translating coordinates from the

application programmer's preferred format to the display hardware's preferred format. The pipeline converts the 3D data in the form of 20-triangle models or shapes, as well as 3D coordinates (x, y, z), into pixels that are displayed on the monitor's 2D space, representing the objects' surface, are stored at the top of the pipeline. The concept of a 3D graphics pipeline can be represented in a numerous ways, some of which are quite complicated while others are quite simple. In its most basic form, the entire procedure can be divided into two steps. In the first step, the geometry processing stage converts 3D object coordinates to 2D window coordinates. The rendering step, which represents the surface of the object represented in Figure 2.1 by filling the pixels' area between the 2D, coordinates with pixels.

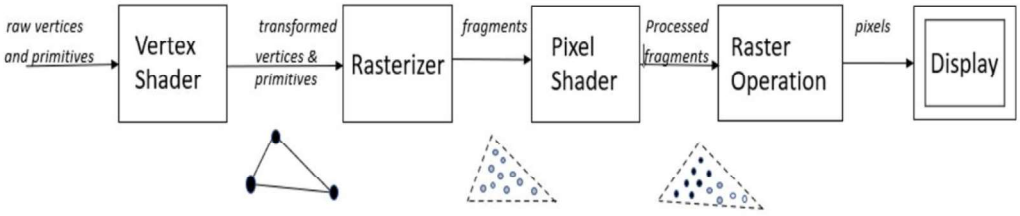


Fig. 2.2: Graphics Rendering Pipeline

Graphics rendering pipeline [8] which is shown in Figure 2.2 consists of the following stages:

• Model Transformations: Converting the coordinates of objects to a common coordinate system is a common step in such a pipeline (e.g. scaling, rotation and translation).

• Lighting: The color of each triangle is calculated based on the lights in the scene. This stage was previously done with Phong shading.

• Camera Simulation: Each colored triangle is projected onto the film plane of the virtual camera.

• Rasterization: It is the process of converting triangles to pixels and includes clipping to the edges of the screen. The color of each pixel is calculated by interpolating the vertices of a triangle.

• Texturing: If a texture is used for increased realism, it should be mapped to pixels. In the rasterization process, texture coordinates are calculated.

2.2 The Modern GPU

The early 1980s are often regarded as the beginning of the current era of computer graphics. The first video cards for the PC were manufactured by IBM in 1981. The monochrome display adaptor could only display monochrome text and couldn't address a single pixel; therefore the screen had to be changed in a 9 × 14 pixel region. Despite the limitations associated with these cards and only non-colored text being allowed, given the restrictions of the original PC, they were an acceptable answer at the time. Although video cards improved with higher resolutions, colour depths, and the ability to change individual pixels (known as "All Point Addressable"), they were still constrained because they were dependent on the main system CPU to conduct all of the calculations. These new audiovisual capabilities put extra burden on the CPU. The first processor-based video card for the PC was released by IBM in 1984. The Professional Graphics Adapter (PGA) had its own Intel 8088 microprocessor onboard, which handled all video-related operations and freed up the main CPU to focus on other things. This was a pivotal moment in the history of the graphics processing unit (GPU), as it introduced the idea of using a separate processor for graphics computing system computations. In an XT or AT machine, the PGA video card required three card slots, as well as a specific professional graphics display, and cost more than $4000 [9]. All of this contributed to the PGA card's short lifespan; nonetheless, version 17 which was capable of 3D animation with 60 frames per second and was aimed at high-end corporate markets for engineering and scientific purposes rather than end users. Larger system needs and higher processing power were necessary for the graphics computer system family to transition and occupy a more significant place after the release of VisiOn, the first GUI for the PC, in 1983. Silicon Graphics Inc. (SGI)

was a nascent computer graphics hardware business that concentrated its efforts on developing the fastest computer graphics machines available. They were also developing industry standards that serve as the framework for today's computer graphics hardware and software, in addition to their technologies. The platform-independent graphics API OpenGL [9] is one of today's software standards. OpenGL was first launched by SGI in 1989 and has since become the most extensively used and supported 2D and 3D API in the industry. The concept of a graphics pipeline was also pioneered by SGI and is now a fundamental part of the design of graphics hardware (which will be discussed later). This design approach is quite comparable across all major GPU manufacturers, such as NVIDIA [10], ATI[11], Matrox[12], and others, and has been a main driving force behind GPU technology's rapid expansion. The development of three-dimensional PC games such as Quake III, Doom, Ultimate Tournament, flight simulators, and others has been the most recent motivation behind the GPU design and implementation boom. These games have prompted the adoption of high-performance GPUs in PCs, and as GPUs have become more powerful, new 3D applications have emerged. Some applications, like geophysical visualization software for oil companies [13], are supercomputer-class applications that have been ported to the PC. 3D games accomplished for the PC the same task as CAD has done for high-end graphics systems. Despite both PC graphics and high-end industrial graphics achieving significant technological advances, it is very important to realize that they serve different purposes. Offline rendering systems, such as those used in CAD programmes, prioritize accuracy over frame rate, but real-time rendering systems, such as gaming engines and simulators, prioritize frame rates for smooth and fluid animations, even if it means sacrificing geometry and texture detail. GPUs have become far more complicated than a general-purpose CPU as technology has progressed. With 222 million transistors, NVIDIA's latest GeForce 6 Series GPUs have more than doubled the transistor count of a Pentium 4. According to Moore's Law, the density of transistors on a given die size doubles every 12 months, however, this has now dropped to every 18 months, in fact, the performance of processor doubles every 18 months. GPU makers have discovered that by

increasing the transistor count two fold on a given chip size every 6 months, transistors have outperformed Moore's Law principles in the last 8 to 10 years. With this rate of development and performance possibilities, it's quite easy to see why a deeper look at the GPU for applications other than graphics is required. Multiple processes can run over the same single threaded pipeline on a general-purpose CPU with a single threaded processing architecture, which accesses its specialized data over a single memory interface. Stream processing, on the other hand, is the architecture of GPUs. This design is much better at dealing with huge data streams, which are common in graphics applications. A GPU can have thousands of stream processors, each of which is connected to another stream processor through an exceptionally fast dedicated pipeline. Unlike the CPU, the stream processors are interconnected through a specialized pipeline, therefore there are no conflicts or delays. Every transistor on the device is active at all times in the stream processing model. This raises the question of whether the GPU will eventually replace the CPU as the computer's primary processor.

2.3 Architecture of GPU

i. The future of high throughput computing is streaming processor.
ii. A Architecture need to be build around the unified scalar stream processing cores.
iii. GeForce 8800 GTX (G80) was the first GPU architecture built with this new paradigm.

In our project, we have used NVIDIA TESLA K80 GPU device. It is based on Kepler architecture as shown in fig.2.3. GK210 is the graphics processor used.

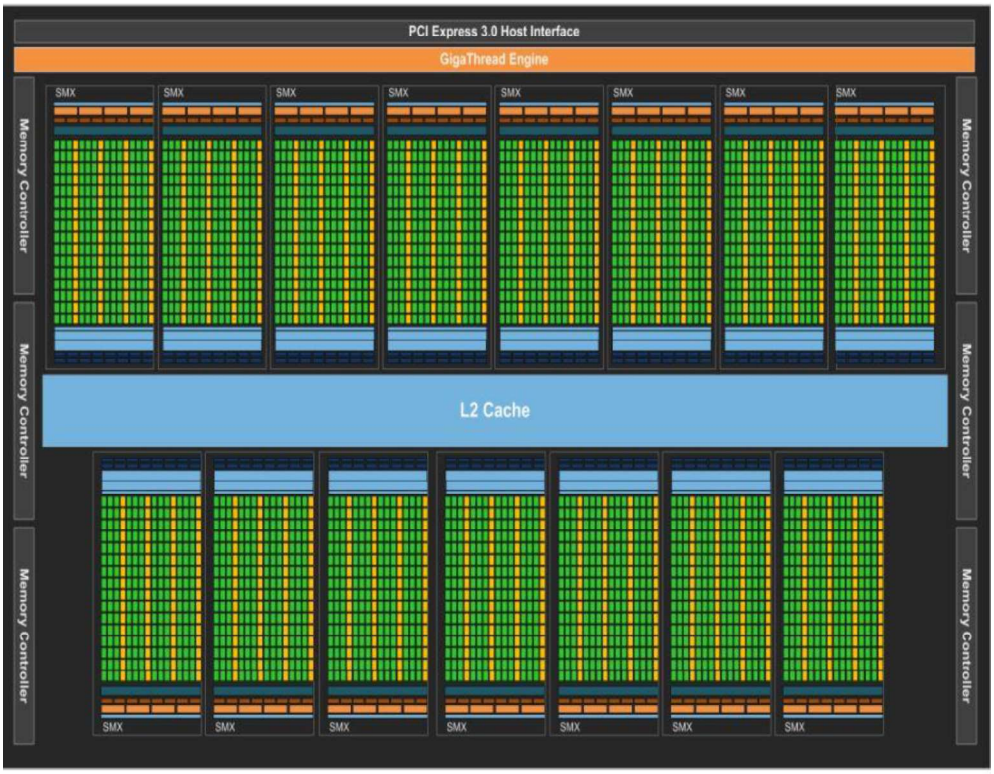


Fig. 2.3: GPU Architecture

2.3.1 Inside a Stream Multiprocessor(SM)

The streaming multiprocessors (SMs) illustrated in fig.2.4, which are a component of the GPU, run our CUDA kernels. Each SM has the following :

- Thousands of registers that can be divided into threads of execution
    - Several caches:
    - Contexts switching between threads and issue commands can be made fast using warps scheduler to ready-to-run warps.
    - Shared memory allows threads to exchange data quickly. Constant cache for quick readings from constant memory broadcast.
    - Bandwidth from texture memory is aggregated using texture cache.
    - Latency to local or global memory is reduced using L1 cache.
- Integer and floating-point operations execution cores:
    - Double-precision floating point operations

- Integer and single-precision floating point operations
- Single-precision floating-point transcendental functions are represented using Special Function Units (SFUs).

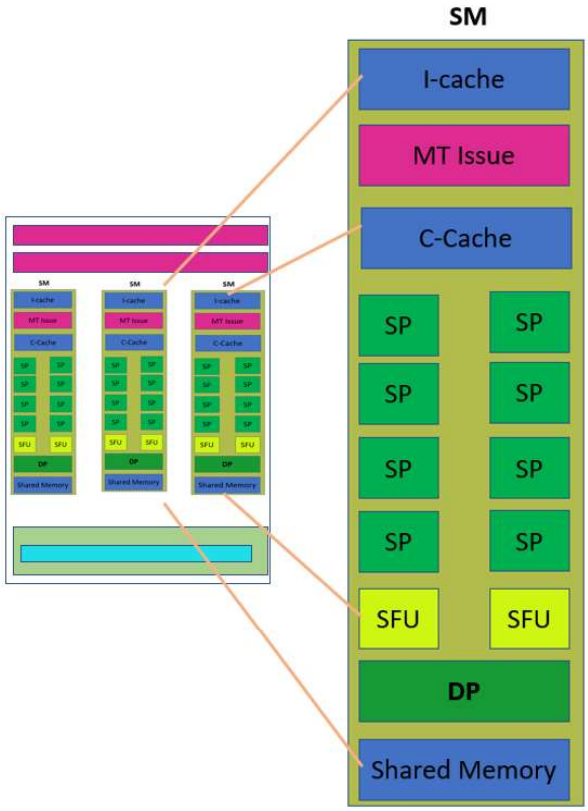


Fig. 2.4: Inside a Streaming Multiprocessor (SM)

2.4 FPGA

FPGAs are prefabricated electronically programmable silicon devices that may build practically any form of digital circuit or system [14]. Compared to fixed-function ASIC technologies like conventional cells, they have a number of major advantages: FPGAs are relatively cheap in cost and can be programmed in less than a second (and can often be changed if a mistake is made). It can take months to build ASICs and its production can cost hundreds or millions of dollars. The flexibility of an FPGA comes at a cost in terms of area, latency, and power consumption: an FPGA usually takes more area than ASIC requires, but it has

much lesser speed performance about 4 times compare to ASIC's, and uses approximately 10 times the dynamic power. These drawbacks stem mostly from the configurable routing fabric of an FPGA, which sacrifices size, speed, and power in exchange for immediate fabrication.

2.4.1 Why FPGAs?

Because of their quick turnaround and low volume cost, FPGAs are an appealing solution for digital system implementation. FPGAs are the only cost-effective way to take advantage of Moore's law's scalability and performance for small businesses or small entities within huge firms. The ASIC design is more difficult and costly because of the challenges faced in deep submicron techniques as Moore's law advances.

The challenges provided by state-of-the-art deep submicron techniques make ASIC design more difficult and costly as Moore's law advances. The investment required to create an effective ASIC consists of three primary components in terms of both time and money:

(1) Extremely expensive cutting-edge ASIC CAD tools for timing analysis, extraction, routing, placement, synthesis, power analysis and simulation.

(2) A mask required for a fully assembled device costs millions of dollars. This cost can be lowered by sharing the costs incurred in prototyping or by using a structured technique so that it needs lesser masks than before for manufacturing of ASICs.

(3) Enormous cost required for an engineering team who will be working on a large ASIC over several years. (An FPGA design team would incur a similar expense, although it would be less.) Most digital design projects begin with FPGA implementation because of the high expenses and the necessity for a proportionally better return on investment.

2.4.2 Overview

FPGAs are made up of a network of programmable logic blocks of various types (fig.2.5) surrounded by a routing fabric which is programmable. It allows the interconnections of blocks to be programmable. Programmable input/output (I/O) blocks surround the array, allowing the device to interface with the outside environment. After silicon manufacture, the "programmable" phrase in FPGA refers to the ability to programme a function into the chip. A mechanism called programming technology, can induce a change in the behavior after fabrication of a pre-fabricated chip in the "field" where this customization is enabled by the design of system users. The programming technology for the first programmable logic circuits was extremely small fuses. The next section on the history of programmable logic briefly describes these devices.

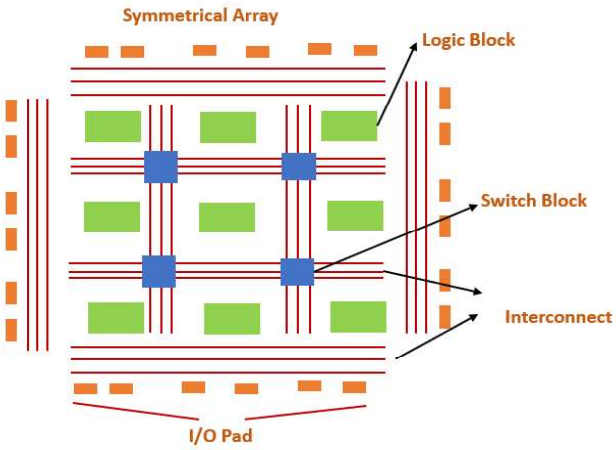


Fig. 2.5: Basic FPGA structure

2.4.3 History

The foundations of the current Field-Programmable Gate Array can be traced back in the early 1960s to the creation of the integrated circuit. In the early days of programmable electronics, regularity in the architecture and flexible functionality were used. Cellular arrays [15] which consists of a 2-D array made of

simple logic cells that interacted in a fixed, point-to-point manner. The Maitra cascade [16] was one of the first arrays to include logic cells that could be programmed to perform a range of two-input logic functions via metalization during manufacture. With the introduction of "cutpoint" cellular arrays in the mid-1960s, field-programmability, or the ability to change a chip's logic function after it was created, was realised. The function of each logic cell in the array can be determined by setting programmable fuses inspite of the fixed connections between the array's parts. Using programming currents or photo-conductive exposure in the field, these fuses could be programmed. Because of this field-customization, array production becomes easier and it also allows for a wider range of applications. A novel technique to implement logic functions was presented in the 1970s with a series of read-only memory (ROM)-based programmable devices. Despite the fact that mask-programmable ROMs and fuse-programmable ROMs (PROMs) may implement any N-input logic function with N address inputs, area efficiency becomes an issue for everyone due to the exponential dependence of area on the value of N. The earliest programmable logic arrays (PLAs) improved on this by adopting two-level AND–OR logic planes that closely matched the structure of common logic functions while being substantially more space-efficient (each plane in a wired-AND or wired-OR configuration, along with inverters, can build any AND or OR logic term). Figure 2.7 is an example PLA. With the understanding that a programmable AND plane followed by a fixed OR plane gave considerable flexibility (fig2.6) announced by Monolithic Memories Incorporated (MMI) [17] in 1977. in PAL devices, these architectures evolved even farther. These devices had configurable combinational logic that provided fixed sequential logic in the form of D-type flip–flop macrocells.

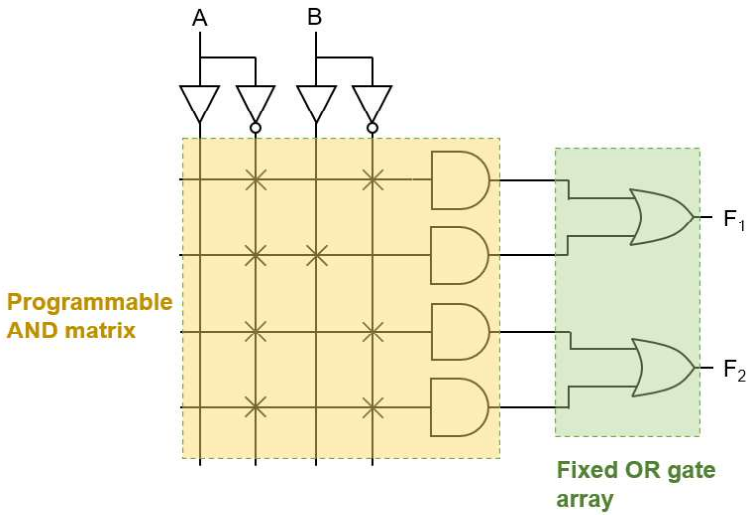


Fig. 2.6: PAL (Programmable Array Logic)

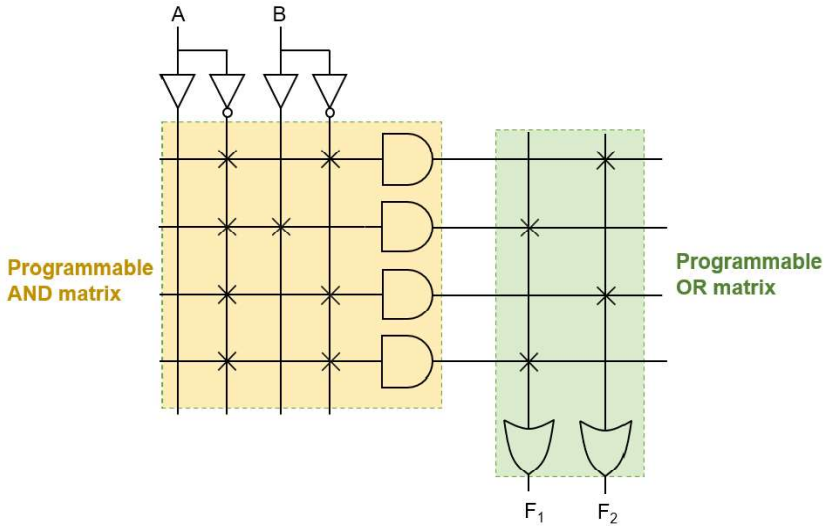


Fig. 2.7: PLA (Programmable Logic Array)

Wahlstrom [18] proposed the SRAM-based FPGA which is the first static memory-based FPGA in 1967. Using a stream of configuration bits, both logic and connectivity configuration is allowed in this architecture. Unlike its modern cellular array equivalents, each logic cell could have both wide-input logic functions and storage elements. In addition to this, to allow for a number of circuit topologies to be implemented, the connections of programmable inter-cell could be easily altered. Although static memory provides the maximum flexibility in terms of

device programmability, it does so at the expense of a significant increase in space per programmable switch when compared to ROM solutions. This difficulty most likely delayed the commercialization of programmable devices based on static memory until the mid-1980s, when the cost per transistor had dropped significantly. Xilinx released the first modern-era FPGA in 1984 [19]. It included the now-ubiquitous Configurable Logic Blocks. FPGAs have expanded in complexity dramatically since the first one, which had 64 logic blocks and 58 outputs and inputs.

2.5 Kintex KC 705 Evaluation Board

The Kintex7 FPGA KC705 evaluation board provides platform for creating and examining hardware designs called hardware platform for the Kintex-7 XC7K325T-2FFG900C FPGA [20]. This board includes DDR3 memory, an 8-lane PCI Express interface, general purpose I/O, and a UART interface, which are all standard characteristics in embedded computing systems. The board utilized in this project is shown in fig.2.8. The following are some of the board's features:

- Kintex-7 XC7K325T-2FFG900C FPGA
- 1 GB DDR3 memory SODIMM
- Clock generation
    - Fixed 200 MHz LVDS oscillator (differential)
    - Inter-integrated circuit (I2C) programmable LVDS oscillator (differential)
    - SMA connectors (differential)
    - SMA connectors for GTX transceiver clocking
- PCI Express endpoint connectivity
    - Gen1 8-lane (x8)

- Gen2 8-lane (x8)

- USB-to-UART bridge
- Status LEDs
    - Ethernet status
    - Power good
    - FPGA INIT
    - FPGA DONE
- User I/O
    - USER LEDs (eight GPIO)
    - User pushbuttons (five directional)

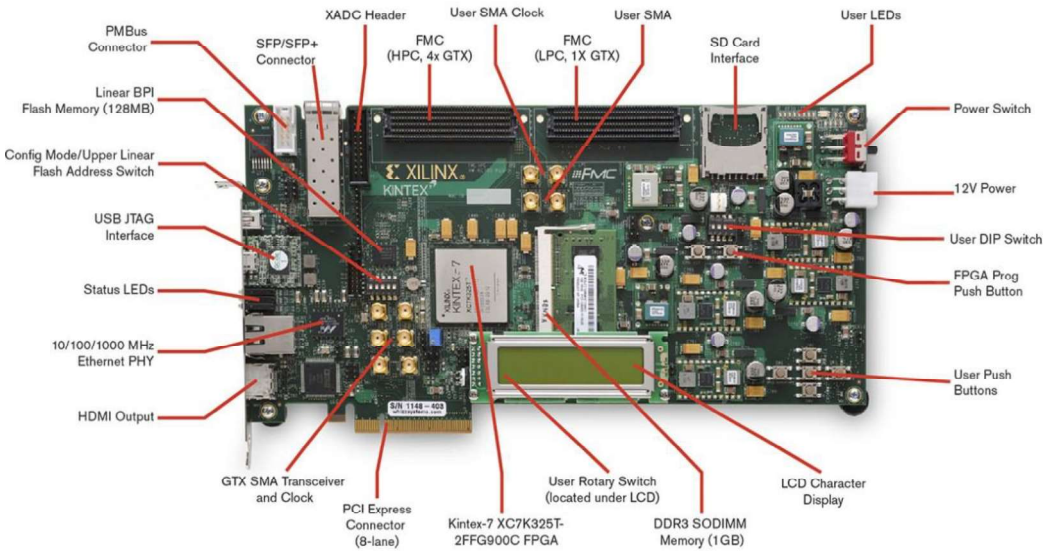


Fig. 2.8: The Kintex KC 705 Evaluation Board

Chapter 3

Methodology

3.1 Problem Formulation

MD (molecular dynamics) describes physical events with a significant parallel component, in which particles (atoms) individually assess the cumulative forces exerted on them by their surroundings based on cutoff distance. MD calculations are usually performed in serial, with processing loops iterating through all particles or particle-pairs at each time step. Most of these explicit loops are no longer required due to the availability of large-scale data parallelism. The overall force computation for each particle, for example, can be done totally independently and in parallel, but the fundamental addition to determine total energy is unavoidably serial, necessitating the use of efficient, but more complicated, parallel reduction algorithms. We require a radial and angular description of the i^{th} particle to determine the force operating on it owing to other particles in a nanoparticle.

An angular description of an i^{th} particle surrounded by other particles [4] can be given by the equation 3.1 as follows:

$$C_{lm} = \sum_{i \neq j}^{n\ atoms} e^{-\epsilon r_{i,j}} * f_c(r_{i,j}) * Y_{lm}^*(\theta, \emptyset) \tag{3.1}$$

where, C_{lm} is angular descriptor

$f_c(r_{i,j})$ is cut-off function

$Y_{lm}^*(\theta, \emptyset)$ is spherical harmonics given by equation 3.2 as follows:

$$Y_{lm}^*(\theta, \emptyset) = N * e^{im\emptyset} * P_{lm}(cos\theta) \tag{3.2}$$

where, $P_{lm}(cos\theta)$ is legendre polynomial

3.2 Algorithmic flow

We want to calculate the angular description values (C_{lm}), and the algorithmic flow (as shown in fig.3.1) to do so is depicted in the diagram below. The entire computation is already done in C code, which runs on a CPU and is executed in a sequential manner.

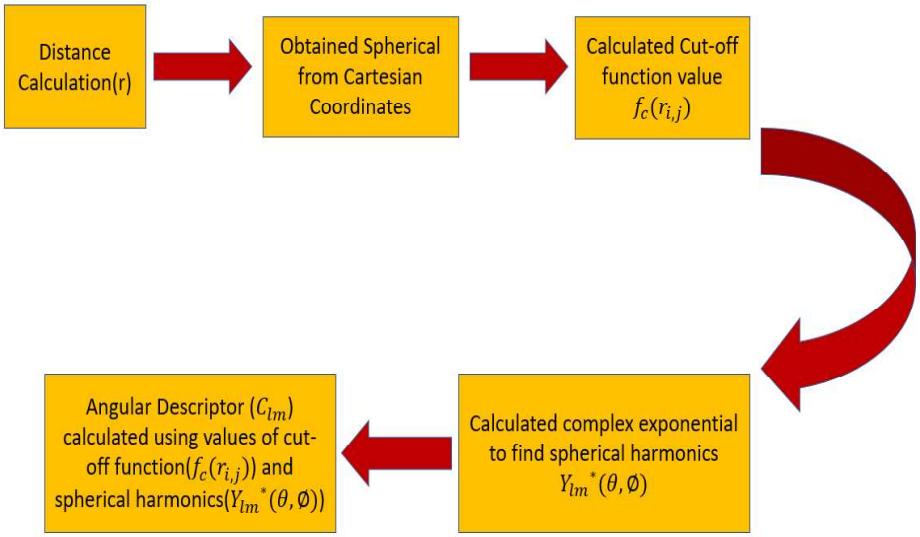


Fig. 3.1: Algorithmic flow of our code

3.3 CPU vs GPU

The main distinction between CPU and GPU architecture is that a CPU is designed to do a wide range of tasks fast (as seen by CPU clock speed), but it has a limit on the number of processes that can run at the same time(as can be seen by comparing number of cores in fig.3.2) [21]. A GPU is a computer processor that is capable of rendering high-resolution graphics and video in real time. GPUs are commonly utilised for general purpose applications such as scientific computation and machine learning because they can conduct simultaneous operations on several sets of data. GPUs provide remarkable parallelism by allowing thousands of CPU

cores to run at the same time. Each core is focused on running operations as efficiently as possible. GPUs can handle data several orders of magnitude faster than CPUs due to their massive parallelism. GPUs, on the other hand, are not as adaptive as CPUs. CPUs, unlike GPUs, have wide and comprehensive instruction sets that handle all of a computer's input and output. In a server, there might be 24 to 48 very fast CPU cores. Connecting 4 to 8 GPUs to the same server can add 40,000 cores to the system. Individual CPU cores are quicker (as measured by clock speed) and smarter (as assessed by available instruction sets), but the sheer number of GPU cores and vast amounts of parallelism they enable more than compensate for the single-core clock speed disparity and restricted instruction sets. Jobs that demand a lot of repetition and parallel processing are best suited for GPUs.

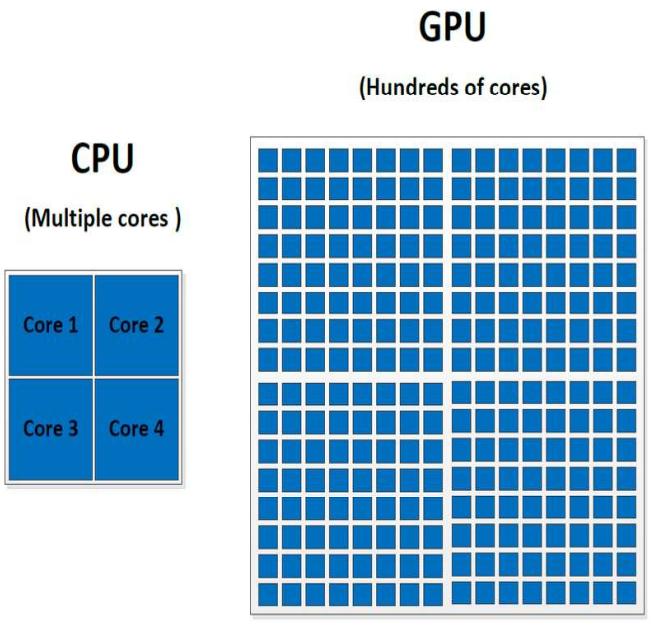


Fig. 3.2: Comparison of CPU and GPU with respect to cores

3.4 CUDA Programming

CUDA is a parallel computing platform and API developed by the company. It's an Nvidia parallel computing design that uses the GPU's capabilities to boost processing performance dramatically. As a best practice, accelerated computing is replacing CPU-only computing. Comprised of both CPUs and GPUs,

accelerated systems are also known as heterogeneous systems. Accelerated systems run CPU applications, which then launch operations that take advantage of GPUs' tremendous parallelism. CUDA programming includes running code on two platforms at the same time: a host system with one or more CPUs and one or more NVIDIA CUDA-enabled GPU devices. While NVIDIA GPUs are most often associated with gaming and graphical aspects, they can also simultaneously execute thousands of lightweight threads. They're well-suited to computations that gain from parallel processing as a result of this.

3.4.1 Host and Device

In the realm of CUDA parallel programming, our CPU is known as the Host, and our GPU is known as the Device.(fig.3.3)

(a) (b)

Fig. 3.3: (a) Host and (b) GPU Device

The threading model and different physical memories are the main differences between them. On host systems, execution pipelines can only handle a certain number of simultaneous threads. Today's servers can only operate 24 threads at a time due to their four hex-core processors (or 48 if the CPUs support Hyper Threading). These threads are executed in a group termed as warp of threads having 32 threads at a time. On GPUs with 16 multiprocessors, modern NVIDIA GPUs can support up to 1536 active threads per multiprocessor, resulting in about 24,000 simultaneously active threads.

3.4.2 Porting to CUDA

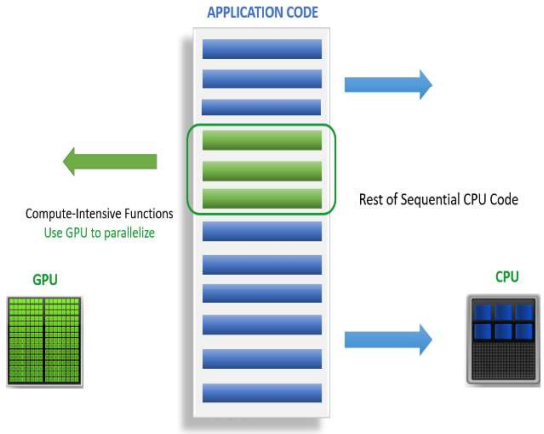
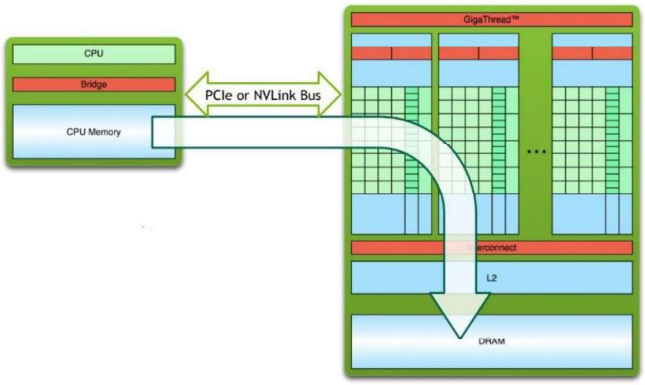


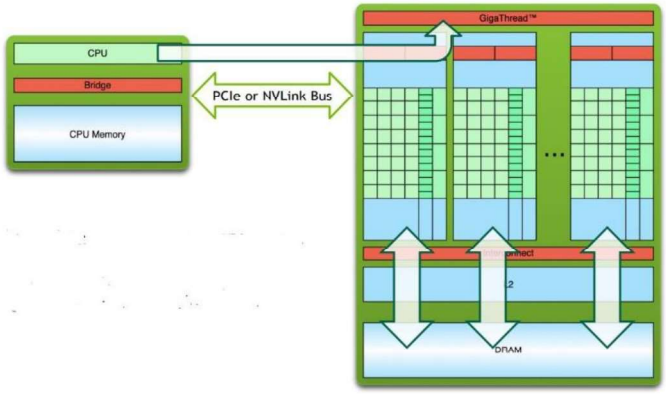
Fig. 3.4: Distribution of code execution between CPU and GPU

Any application code can run heterogeneously on CPU and GPU, as shown in the fig.3.4 [23]. A portion of the code that is compute heavy will run in parallel on the GPU, while the rest will run sequentially on the CPU.

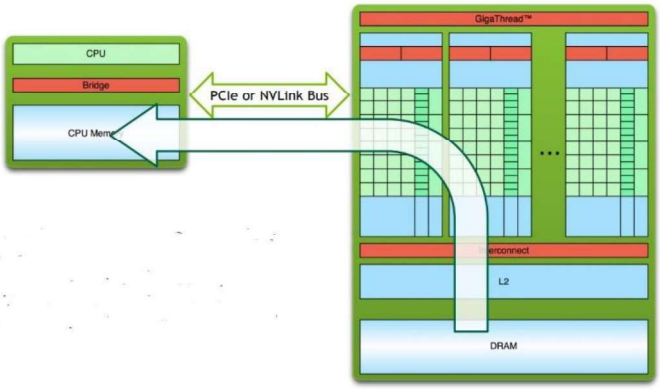
3.4.3 Simple Processing Flow



(a)



(b)



(c)

Fig. 3.5: (a) Copy input data from CPU memory to GPU memory, (b) Load and run a GPU program, caching data on the chip for better performance, (c) Copy results from GPU memory to CPU memory.

3.4.4 Parallel Programming in CUDA C

GPU computing is all about massive parallelism. It can be understood with the help of an interesting example of vector addition.

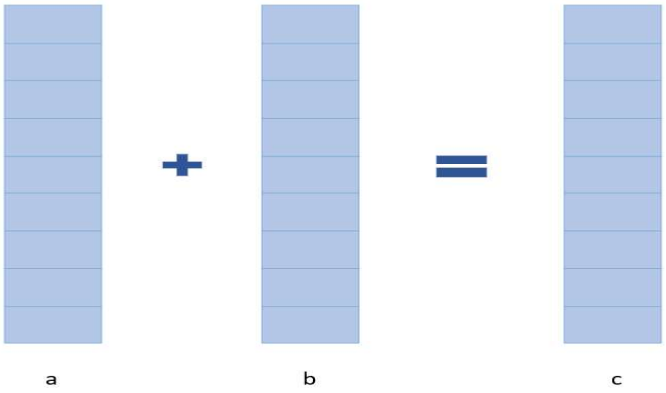


Fig. 3.6: Representation of Vector addition in parallel

3.4.4.1 GPU Kernels: Device Code

```
__global__ void mykernel (void) {

}
```

- CUDA C++ keyword __global__ indicates a function that:
  - The following function will run on the GPU, and can be invoked globally, which in this context means either by the CPU, or, by the GPU
  - It is required that functions defined with the __global__ keyword return type void
- nvcc breaks down source code into host and device components
  - Device functions (e.g. mykernel()) processed by NVIDIA compiler
  - Host functions (e.g. main()) processed by standard host compiler

```
mykernel<<<1,1>>>();
```

- When a function is called to run on the GPU, it is usually referred to as a kernel.
- Before handing the kernel any expected arguments, we must give an execution configuration by using the<<<... >>> syntax.

- At a high level, execution configuration allows programmers to define the thread hierarchy for a kernel launch, which includes the number of thread groupings (called blocks) and the number of threads to execute in each block. In the above example, the kernel is launching with 1 block of threads (the first execution configuration argument) which contains 1 thread (the second configuration argument).

cudaDeviceSynchronize();

• Unlike a lot of C/C++ code, launching kernels is asynchronous: the CPU code will continue to run while the kernel is being launched.

• A call to cudaDeviceSynchronize, a CUDA runtime function, will cause the host (CPU) code to wait until the device (GPU) code completes before resuming CPU execution [23].

3.4.4.2 CUDA Thread Organization

There are three layers in the CUDA thread hierarchy. The basic parallel unit is warp, which also defines the hardware memory bandwidth of the GPU device. Each warp has a total of 32 threads, which are split into two half warps and scheduled to the hardware scheduler's execution queue during execution. The CUDA language extension does not provide a clear description for controlling warp behaviour flow from the standpoint of a programmer. Thus, at the bottom of the programmable CUDA thread hierarchy, the thread blocks are visible. Because the existing architecture only enables one grid on device at a time when the kernel is launched from the host, grid is a collection of thread blocks that may be considered as a device abbreviation. The kernel function is executed by all threads in a grid. The size and dimension of thread blocks can be determined by the programmer using built-in variables within hard limits that vary from product to product, which is extremely flexible in reality. Threads can use pre-defined thread indices to distinguish themselves from one another.

For block and grid dimensions as below dim3 **block**(4, 1, 1) & dim3 **grid**(8, 1, 1) can be visualized as in fig 3.7 and dim3 **block**(8, 2, 1) & dim3 **grid**(2, 2, 1) can be visualized as shown in fig 3.8.

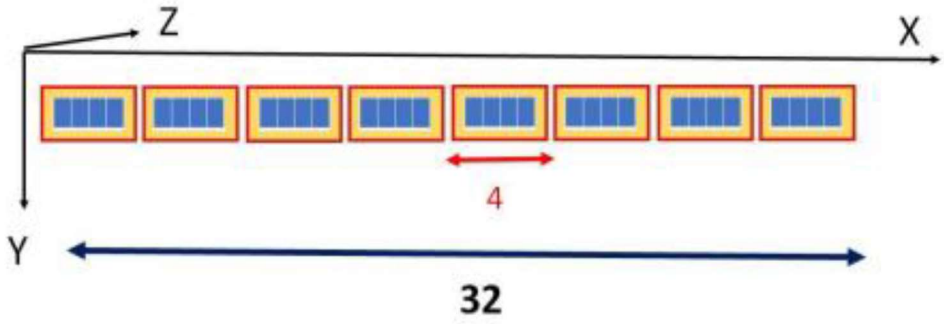


Fig. 3.7: Representation of Block and Grid in a single dimension

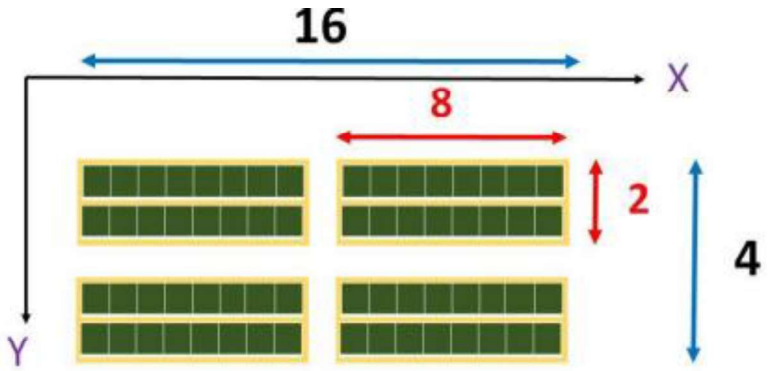
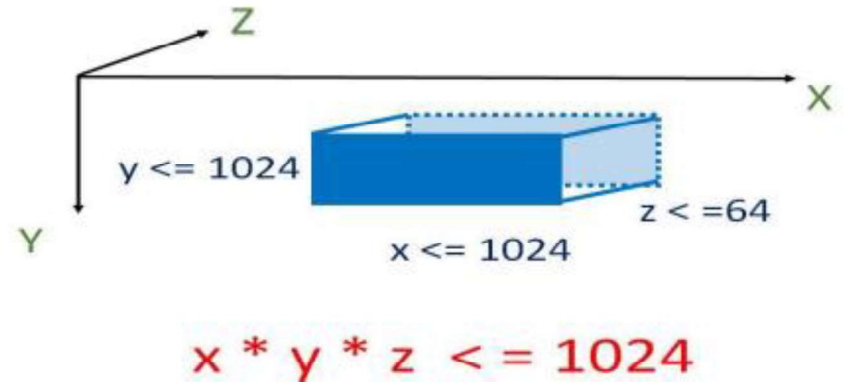
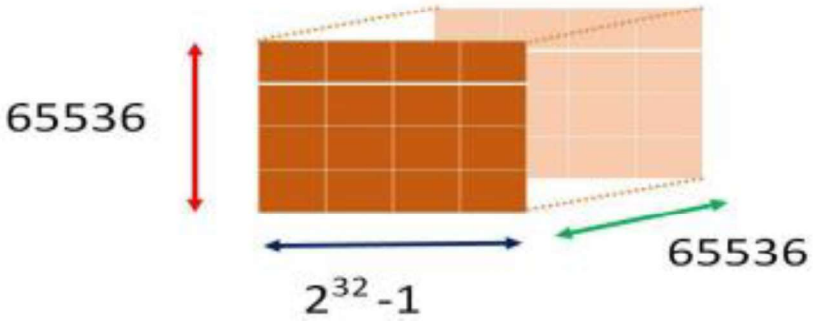


Fig. 3.8: Representation of Block and Grid in multi-dimension



(a)



(b)

Fig. 3.9: Limitation for number of (a) threads in a block in each dimension, (b) blocks in a grid

There are certain limitations on determining the block size. As these block and thread variables are three dimensional of type dim3, threads in a block can be arranged in a 3D manner. Fig 3.9(a) shows the maximum number of threads that can be arranged in each dimension of a threadblock whereas fig 3.9(b) shows the maximum number of blocks that can be arranged in each dimension of a grid. Some key terms can be defined as follows:

(a) ThreadIdx - CUDA runtime uniquely initialized threadIdx variable (fig.3.10) for each thread depending on the coordinates of the belonging thread in the block(Table 3.1). ThreadIdx is a dim3 type variable.

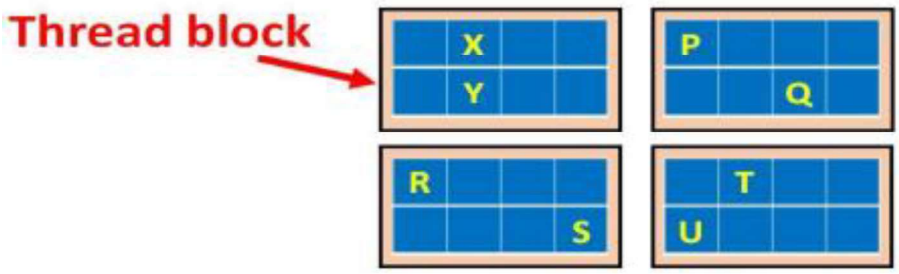


Fig. 3.10: Illustration of thread index assigned to various threads

Thread	X	Y	P	Q	R	S	T	U
ThreadIdx.x	1	1	0	2	0	3	1	0
ThreadIdx.y	0	1	0	1	0	1	0	1

Table. 3.1: Values of thread index assigned to various threads

(b) BlockIdx - CUDA runtime uniquely initialized blockIdx variable (fig.3.11) for each thread depending on the coordinates of the belonging thread block in the grid(Table 3.2). blockidx is dim3 type variable.

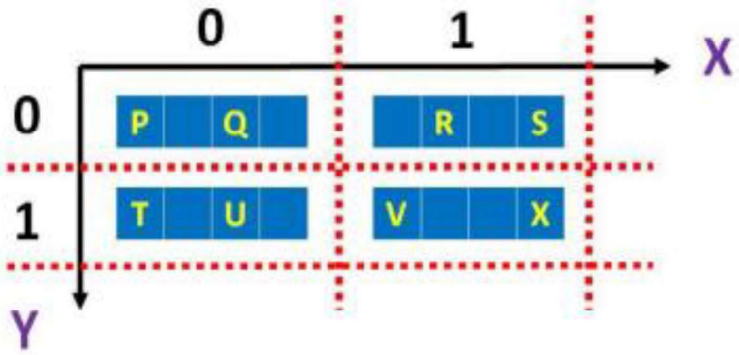


Fig. 3.11: Illustration of block index assigned to various threads

Thread	P	Q	R	S	T	U	V	X
BlockIdx.x	0	0	1	1	0	0	1	1
BlockIdx.y	0	0	0	0	1	1	1	1

Table. 3.2: Values of Block index assigned to various threads

(c) blockDim - blockDim variable consists number of threads in each dimension of a thread block (fig.3.12). Each blue square represent a thread and group of blue squares in a yellow border represents a block.

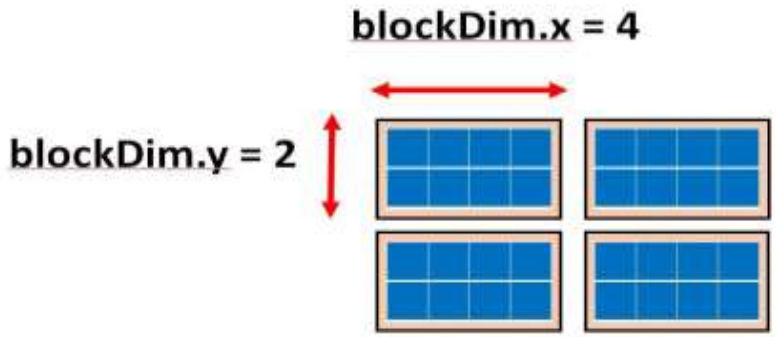


Fig. 3.12: Illustration of dimension of a block based on number of threads

(d) GridDim - GridDim variable consists of number of thread blocks in each dimension of a grid.

3.4.4.2.1 Unique index calculation for a thread in a grid

There are various blocks in a grid each having same number of threads. For doing any calculation, we need to access these threads so that they are addressed uniquely as shown in the fig.3.13.

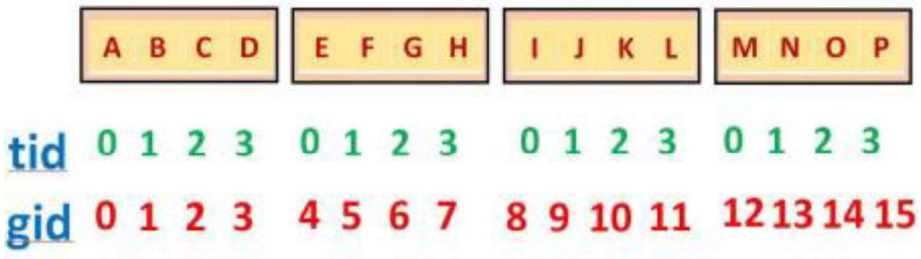


Fig. 3.13: Representation of local and global thread Id

Where tid is thread id and gid is global id.

gid = tid + offset

gid = tid + blockIdx.x*blockDim.x

Example: For thread in second thread block, blockIdx.x = 1 & blockDim.x =4

Offset = blockIdx.x * blockDim.x = 1*4 =4

3.4.4.2.2 Execution Model

- Scalar processors are used to execute threads.
- Thread blocks are not migrated and are executed on multiprocessors.
- On a single multiprocessor, multiple concurrent thread blocks can exist, but multiprocessor resources are limited by shared memory and register file constraints.
- A kernel is launched as a series of thread blocks in a grid.

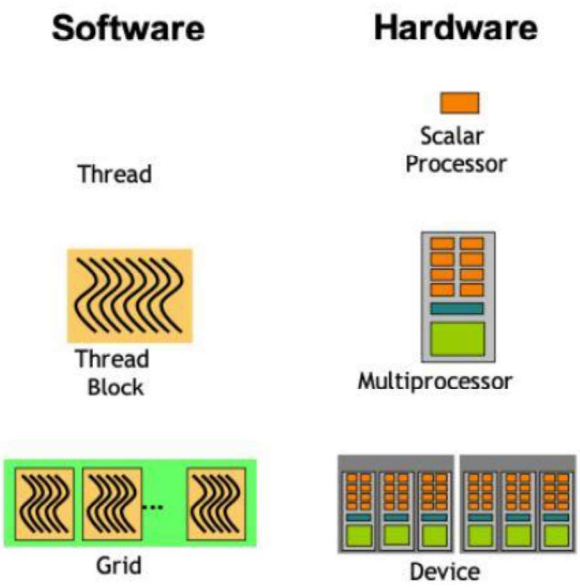


Fig.3.14: Hardware perspective of CUDA terminologies while execution [24]

- Thread blocks are divided into smaller units called warps each having 32 consecutive threads.
- Suppose number of Streaming multiprocessors(SMs) are 13 and total number of cores per SM is 128 and the Block size we took is 512.
    - Number of warps per block = Block size/ Warp size = 512/32 = 16
    - 4 warps can execute parallelly in single SM.
- Warps can be defined as the basic unit of execution in a SM. Once a thread block is scheduled to an SM, threads in the thread block are further partitioned into warps.
- And all threads in a warp are executed in Single Instruction Multiple Thread (SIMT) fashion.

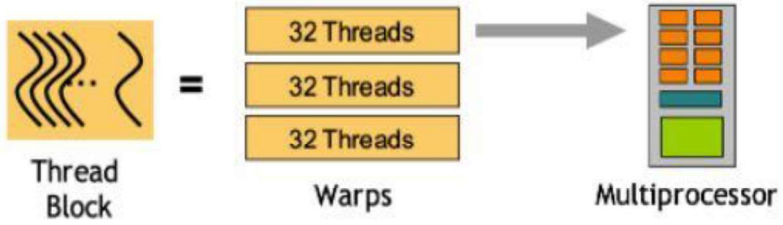


Fig.3.15: Group of threads in a warp before executing on multiprocessor

3.4.4.3 Running Code in Parallel

As we were discussing about parallelization of vector addition, suppose we change the execution parameter from 1 to N. Consequently, instead of executing add() once, it is executed N times in parallel.

```
add<<< 1, 1 >>>();
```

⇩

```
add<<< N, 1 >>>();
```

Each parallel invocation of add() is referred to as a block. The set of all blocks is referred to as a grid. Each invocation can refer to its block index using blockIdx.x .

```
__global__ void add(int *a, int *b, int *c) {
c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

By using blockIdx.x to index into the array, each block handles a different index. Built-in variables like blockIdx.x are zero-indexed (C/C++ style), 0,..,N-1, where N is from the kernel execution configuration indicated at the kernel launch.

```
#define K 524
int main(void) {
```

```
int *p, *q, *r;                    // host instances of p, q, r
int *dev_p, *dev_q, *dev_r;        // device instances of p, q, r
int size = K * sizeof(int);
// Allocate space for device instances of p, q, r
cudaMalloc((void **)&dev_p, size);
cudaMalloc((void **)&dev_q, size);
cudaMalloc((void **)&dev_r, size);
// Allocate the space for host instances of p, q, r
p = (int *)malloc(size);
q = (int *)malloc(size);
r = (int *)malloc(size);
// Inputs are copied to device for computation
cudaMemcpy(dev_p, p, size, cudaMemcpyHostToDevice);
cudaMemcpy(dev_q, q, size, cudaMemcpyHostToDevice);
// Launch add() kernel on GPU with K blocks
add<<>>(dev_p, dev_q, dev_r);
// Result is copied back to host
cudaMemcpy(r, dev_r, size, cudaMemcpyDeviceToHost);
// Free the occupied memory
 free(p); free(q); free(r);
 cudaFree(dev_p); cudaFree(dev_q); cudaFree(dev_r);
```

```
return 0;

}
```

3.5 GPU Memory Hierarchy

The CUDA programming approach implies that each device manages its own memory area on GPU DRAM. CUDA offers three forms of memory that programmers can use: global (device) memory, shared memory, and registers, as shown in Figure 3.4. Each thread in a thread block has 32-bit registers that it can read and write. Each thread block has shared memory that is visible to all threads within it and has the same lifetime as the block. On-chip memories include registers and shared memory. Variables in these kinds can be accessed in a highly parallel manner at very high speeds. By sharing input data and intermediate outcomes of processed data, shared memory allows threads to collaborate more effectively. Appropriate accessing may allow shared memory to operate at the

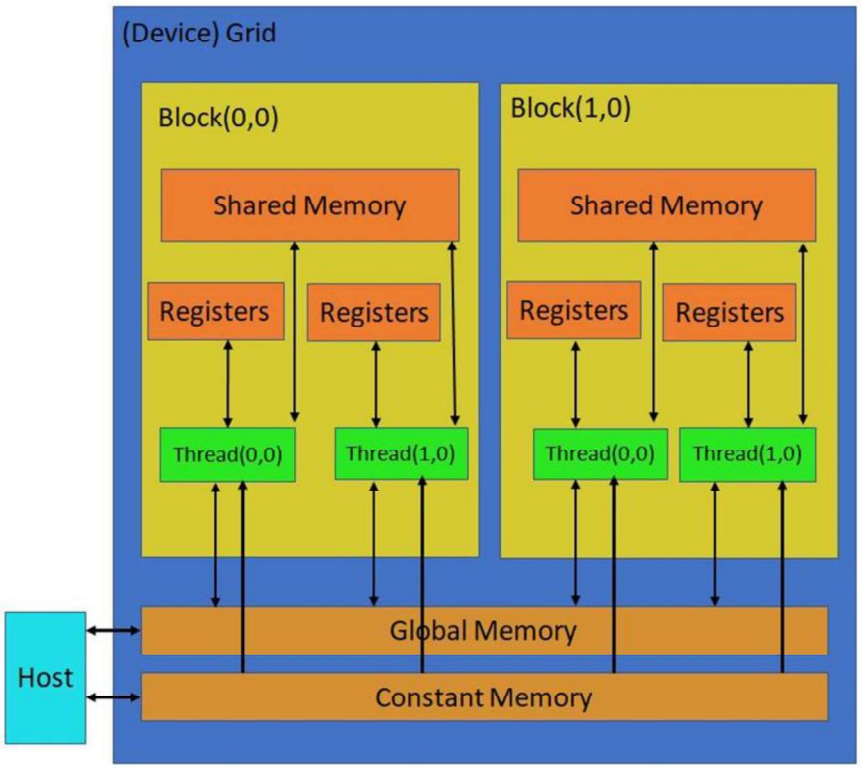


Fig. 3.16: Organization of memory inside a GPU

same speed as registers. The global memory is shared by all threads. To access data in device memory, many hundred clock cycles are required. Constant and texture memory are stored off-chip as part of device memory. Constant memory, on the other hand, is read-only and cached, and it provides low-latency, high-bandwidth memory access by device, with all threads accessing the same address at the same time [25].

3.6 Basic Architecture of work done on FPGA

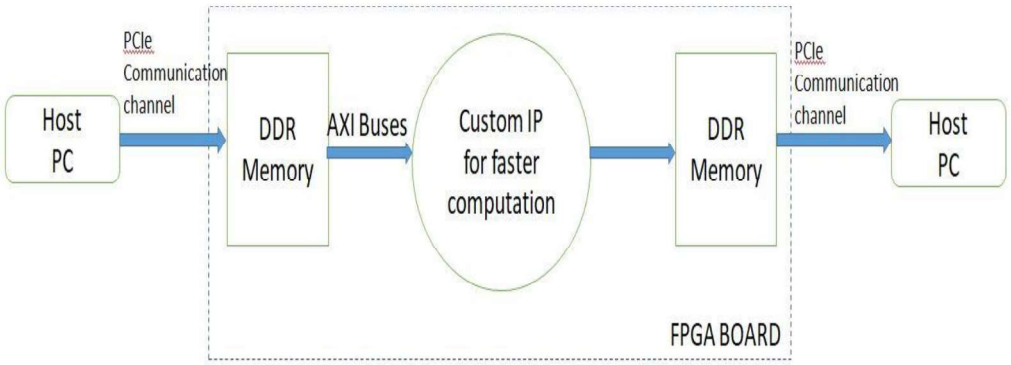


Fig. 3.17: Flow of the work done on FPGA

As shown in the fig.3.17, input data from host PC is first sent to FPGA. We have used PCIe (Peripheral component interconnect express) protocol so that proper communication channel is established between host PC and FPGA endpoint.

The architecture of the work done using FPGA can be understood with the help of above block diagram. We need to understand how the data should be sent from host PC to FPGA, perform the required computations and the results to go back to the host PC. A Vivado Design Suite subsystem for PCI Express endpoint-initiated Direct Memory Access (DMA) data transfers is demonstrated in this project. The offered subsystems target the following devices to initiate data transfers between DDR3 memory and an externally attached PCI Express Root Complex Kintex -7 KC705 device (in our project). This design shows how to leverage the AXI Memory Mapped to PCI Express IP to execute high-throughput data transmission across a PCI Express link (see diagram below). A Scatter Gather

capable DMA engine is used in conjunction with the PCI Express IP to achieve this. The DMA engine enables the FPGA to handle data transmission over the PCI Express link, allowing it to boost throughput while lowering processor use on the Root Complex side of the link [26].

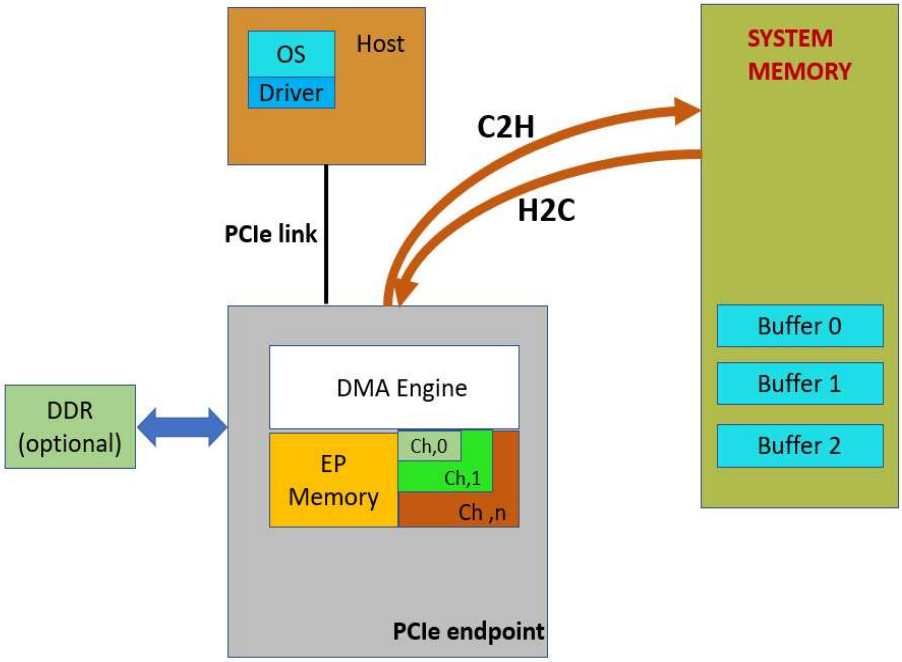


Fig.3.18: Transfer between host and card using PCIe communication

3.7 Communication protocol - PCIe

PCI Express is a high-speed, general-purpose I/O connection standard designed for a wide range of computing and communication devices. The usage model, load-store architecture, and software interfaces of PCI are all preserved, but the parallel bus implementation is substituted by a highly scalable, entirely serial interface. PCIe has a bidirectional connection that allows it to send and receive data at the same time [27]. Because each interface contains a simplex transmit and receive path, the model deployed is referred to as a dual-simplex connection, as shown in the diagram below. The communication path between two devices is functionally full duplex because traffic can flow in both directions at the same time,

but the spec uses the term dual-simplex to describe the actual communication channel.

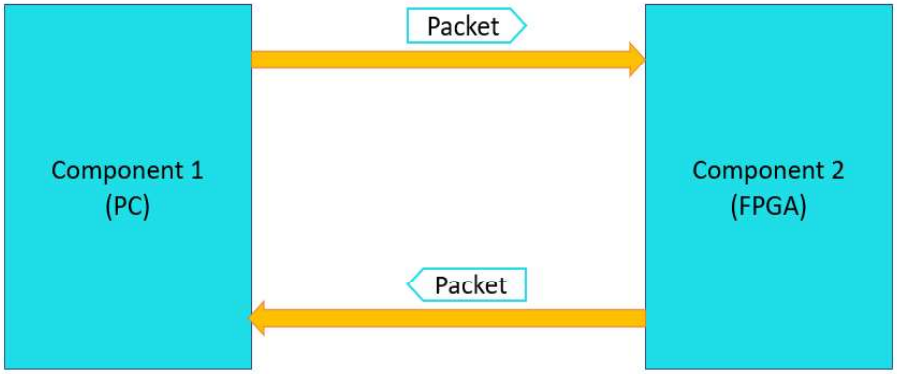


Fig. 3.19: Bidirectional connection between the two components using PCIe

This path between the devices is referred to as a Link, and it is formed by the combination of one or more transmit and receive pairs. A Lane is one of these pairs, and a Link can be made up of 1, 2, 4, 8, 16, or 32 Lanes, depending on the standard. The Link width is the number of lanes and is expressed as x1, x2, x4, x16, and x32. The trade-off between the number of lanes to utilize in a particular architecture is simple: more lanes increase the Link's bandwidth while also increasing its cost, space need, and power consumption.

3.7.1 Definition of Bus, Device and Function

Every PCIe function is individually recognized by the Device it sits in and the Bus to which it connects, just as PCI. A 'BDF' is the popular name for this unique identification [27]. Every Bus, Device, and Function (BDF) inside a given topology is detected by configuration software.

3.7.2 PCIe Buses

Configuration software can allocate up to 256 Bus Numbers. The Root Complex is usually allocated Bus 0, which is the first bus number by hardware. Bus 0 is made up of a virtual PCI bus with integrated endpoints and virtual PCI-to-PCI

Bridges (P2P) with Device and Function numbers hard-coded. Each P2P bridge establishes a new bus that can be used to connect more PCIe devices. A unique bus number must be assigned to each bus [27]. The procedure of allocating bus numbers starts with the configuration programme looking for bridges that start with Bus 0, Device 0, Function 0. When a bridge is discovered, software provides a bus number to the new bus that is unique and greater than the bus number on which the bridge is located.

3.7.3 PCIe Devices

PCIe allows up to 32 device attachments on a single PCI bus, however due to its point-to-point structure, only one device can be directly attached to a PCIe connection, and that device is always Device 0 [27]. Virtual PCI buses on Root Complexes and Switches allow many Devices to be joined to the bus.

3.7.4 PCIe Functions

Every device has functions built in. Each Function does have its own configurable address space, which is used to set up the Function's resources. Each Function has its own dedicated block of configuration address space defined by PCI. Software can detect the existence of a Function, configure it for regular operation, and verify its status using registers mapped into the configuration space. Because it was initially built for PCI, the 256 bytes of PCI-compatible configuration space has that designation. This space's configuration header takes up the first 16 dwords (64 bytes) (Header Type 0 or Header Type 1). Except for the bridge functions, which employ Type 1 headers, Type 0 headers are required for all functions [27]. Optional registers, such as PCI capability structures, are used with the remaining 48 dwords.

3.8 Implemented block design

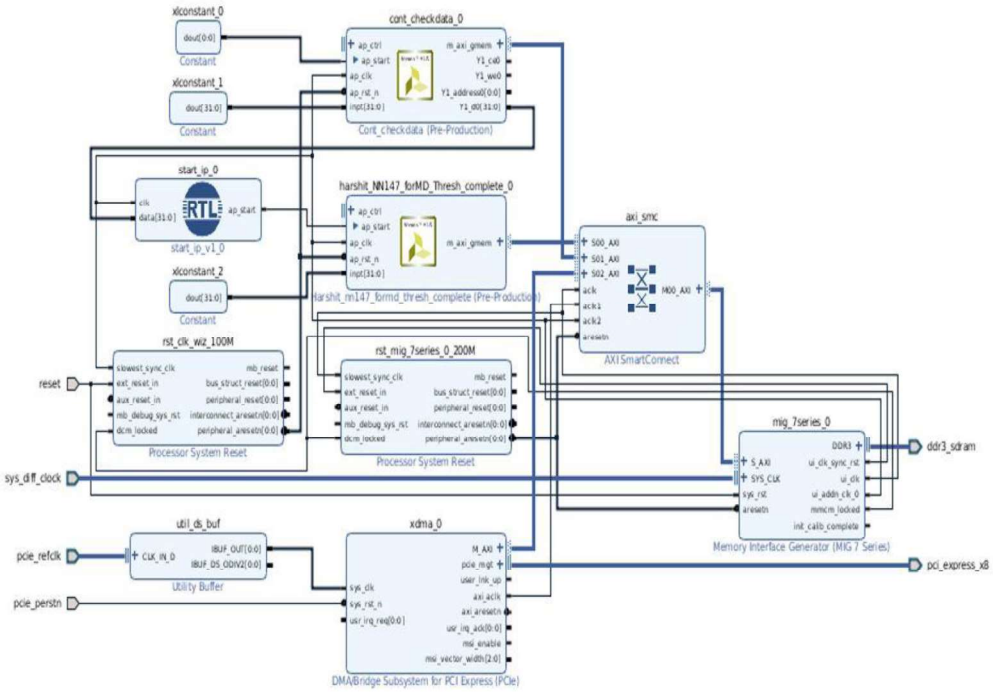


Figure 3.20: Block design implemented on Xilinx Vivado

Above block design is implemented with the help of some key IPs:

3.8.1 DMA/Bridge Subsytem for PCI Express (PCIe)

For using the PCI Express protocol, the Xilinx's DMA for PCI Express (PCIe) IP [28] delivers a high-performance DMA. There are two possible interfaces available, one is AXI4 Memory Mapped and other one is AXI4-Stream.

Features:

- Supports 64, 128, 256, 512-bit datapath
- 64-bit source, destination, and descriptor addresses
- Up to 4 (H2C/Read) data channels
- Up to 4 (C2H/Write) data channels
    - Single AXI4 memory mapped (MM) user interface

- AXI4-Stream user interface

The PCI Express DMA/Bridge Subsystem allows data to be moved between the memory of our PC and the DMA used. This is accomplished by using descriptors. These descriptors consists of information about the source address which is the address from where data is to be taken, destination address which is the address where data needs to be sent, and size of data to be transferred. Host to Card/System (H2C) and Card/System to Host (C2H) transfers are examples of direct memory transfers. The DMA can be set up to have a single AXI4 Master interface shared by all channels or a separate AXI4-Stream interface for each channel. Memory transfers are described per-channel in descriptor linked lists, which the DMA retrieves and processes from host memory. Interrupts are used to notify events such as descriptor completion and errors.

The transactions on which bus are determined by the channel type chosen.

- A Host-to-Card (H2C) channel sends read requests to PCIe and returns data, or sends a write request to the user application.
- A Card-to-Host (C2H) channel, on the other end either waits for data on the user side or generates a read request on the user side and then generates a write request containing the data received to PCIe.

3.8.2 Custom HLS IPs

The algorithms used are becoming complicated day by day. Vivado High-Level Synthesis is available as an upgradation in the HLx edition. With the advent of Vivado HLS, it has become easy to create an IP with complex code, as it allows users to use C, C++ and System C languages and directly create configurations for the programmable device without the need to develop any RTL manually. Vivado HLS creates a similar system and design architects, providing a faster IP creation. Our entire algorithm to calculate Force values and Energy is based on C/C++ language. By using HLS software, we can create an IP of this algorithm and later export it to the Vivado HLx software where this IP can be used inside the Block

Design. This IP will receive the input coordinate values and subsequently calculates the Energy and Force values based on the algorithm written inside it.

3.9 Functionality

After installing the drivers successfully, the following user-accessible devices make up the XDMA driver. As a reference, the driver is given. We can edit the driver to include particular needs, based on their requirements [26].

- xdma0_control : for accessing Xilinx DMA registers
- xdma0_h2c_0/1/2/3, xdma0_c2h_0/1/2/3 : All the 4 channels can be accessed corresponding to the number written after h2c/c2h.

There are some script files used to perform various tasks:

- run_test.sh : Basic transfer is done by this script file(determines the transfer size)
    - After loading driver, checks if the design is AXI-MM or AXI_ST and the number of channels enabled.
    - Basic transfer can be performed to all enabled channels.
- load_driver.sh: loads driver
- dma_memory_mapped_test.sh
    - Write to enabled H2C channels.
    - There are 4 address offset values present in this file which should be given appropriate values:
        - **First addrOffset(2151750108)**: Set address offset1 for writing zero(zero.bin file) at energy location. Whenever this zero is replaced by actual value of energy, computer understands that calculation is over and now it has to receive the values of energy and forces from FPGA board.
        - **Second addrOffset(2150629376)**: This is to set address location of DDR memory where we want to store one extra value & XYZ values (inptfile1.bin).

- **Third addrOffset(2151750108)**: This is again to set the last location of DDR memory, where energy will get stored. We will continuously read the value of DDR memory location set by this addrOffset and compare it with zero(by comparing krit_out0 and zero.bin file) and once read value is not equal to zero, computer gets to know that energy and force calculation is over.
- **Fourth addrOffset(2151748344)**: Set address offset4 to read the force and energy values.

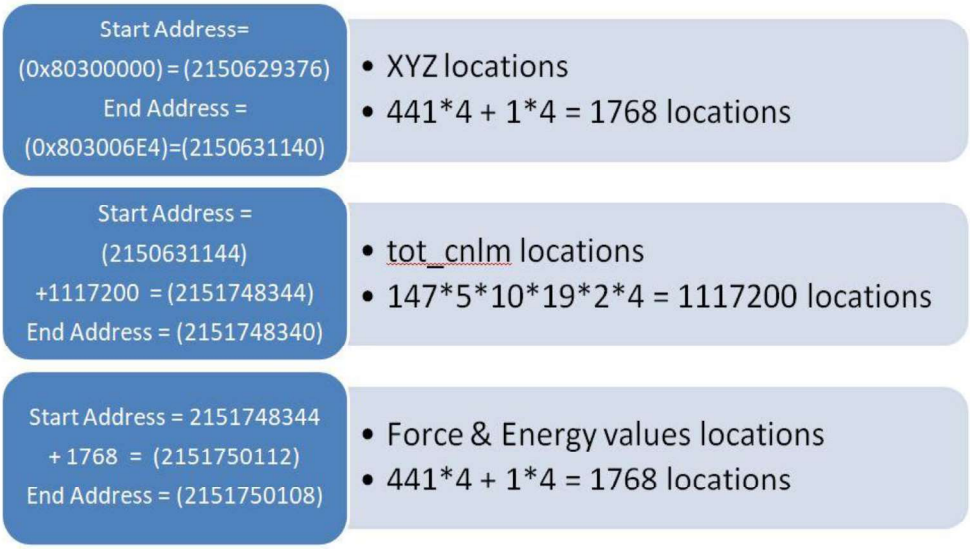


Fig. 3.21: Memory Mapping for storing data in DDR3 memory

After doing all the above mentioned steps in host PC, input values or coordinates of the atoms positioned in 3D space are being sent from host PC to FPGA endpoint. On the other side, we have implemented the block design on Vivado HLx software and burn the binary file (after generating bitstream) on the FPGA board. The entry point of these input values on FPGA is the DMA/Bridge Subsytem for PCIe IP. The data is moved with the help of DMA engine which uses descriptors and allocates buffer space in system memory to do so. The descriptor

formats have various contents which can be determined by a number of factors, including the DMA engine's user interface. A C2H transaction in an AXI Memory Mapped interface has an AXI address as the source address and a PCIe address as the destination address. The source address for an H2C transaction is a PCIe address, and the destination address is an AXI address. The PCI Express DMA/Bridge Subsystem uses a linked list of descriptors to identify the DMA transfers' source, destination, and length. The driver generates and stores descriptor lists in the host memory. The driver sets up the DMA channel with a few control registers so that it can start fetching descriptor lists and performing DMA operations. These descriptors describe the data transfers with memory (DDR3 in this project) that the DMA/Bridge Subsystem for PCIe should complete. Every channel has its own set of descriptors. The hardware registers are set with address of channel's descriptor list by the driver installed. The descriptor channel starts fetching descriptors from the initial address once it is enabled. It then fetches from the Next address field of the previously fetched descriptor.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "hls_math.h"

void harshit_27jan(volatile int *bram)
{
#pragma HLS INTERFACE m_axi depth=20 port=bram offset=direct
#pragma HLS INTERFACE ap_none port=return
	int x[10];

	for(int i=0;i<10;i++){
		x[i] = i;
	}
	memcpy((int*)(bram),x,10*sizeof(int));
}
```

From DMA/Bridge Subsystem for PCI Express, this data is stored in DDR3 memory (Memory Interface Generator). When the complete set of data is stored in memory, our HLS IP is turned on by asserting ap_start signal high. This task is

done efficiently by using separate HLS IP, named as cont_checkdata_0. Now, the data should be fed to our custom HLS IP for various calculations such as Force, Energy, Angular description. The data transfer can be done directly by using memcpy() function as shown in the above code for storing 10 integer numbers.

Once HLS IP receives the data, it starts doing calculations and the results are stored back in DDR3 memory at subsequent memory locations. When all the data gets stored in memory, we need to send these output values to the host PC and in this way our whole process gets completed. This process can be performed iteratively according to the number of MD (Molecular Dynamics) steps.

(This page is left blank intentionally)

Chapter 4

Results and Discussion

4.1 Results on GPU

We have executed the sequential C code and parallelized CUDA code on Google Colab platform where we have been provided the access to CPU and GPU having the following hardware specifications (Table 4.1):

CPU	GPU
Intel Xeon	Tesla K80
1 core	2496 cores
Frequency @ 2.3GHz	Frequency @ 562 MHz
12 GB RAM	13 GB RAM

Table. 4.1: Hardware specifications of the CPU and GPU used on Colab

We started with the execution of algorithm used to calculate distances between 147 atoms positioned in 3D space. First we calculated the time taken by sequential C code to calculate the distance values. Subsequently, we wrote the parallel equivalent of that algorithm by using CUDA platform and after measuring the time taken for execution, compared both the results (Table 4.2).

Processing Unit	Computational time (ms)
CPU	5-10
GPU	0.9-1

Table. 4.2: Computational time of CPU and GPU

Firstly, we executed the sequential C code for 10 iterations. It calculates the angular description of atoms, i.e. C_{lm} values and our task was to calculate the time taken by it. All the calculations are done for 147 atoms. Table 4.3 shows the time taken in each iteration.

Iteration Number	**Time Taken(sec)**
Iteration1	1.370
Iteration2	1.377
Iteration3	1.382
Iteration4	1.369
Iteration5	1.343
Iteration6	1.395
Iteration7	1.427
Iteration8	1.389
Iteration9	1.380
Iteration10	1.364

Table. 4.3: Time taken to execute serial C code in each of the 10 iterations

We took the average from these 10 iterations and it came out to be **1.380 seconds.** Secondly, we executed the CUDA C code for the same algorithm on GPU by varying number of threads per block. Table 4.4, 4.5, 4.6 shows the corresponding results:

For threads per block = 128	
Iteration Number	**Time Taken(ms)**
Iteration1	417.74
Iteration2	437.44
Iteration3	390.58
Iteration4	434.36
Iteration5	389.35
Iteration6	411.55
Iteration7	435.73
Iteration8	424.73
Iteration9	422.40
Iteration10	397.48
Average time = 416.13ms	

Table. 4.4: Time taken to execute CUDA C code for 128 threads/Block

For threads per block = 256	
Iteration Number	**Time Taken(ms)**
Iteration1	495.69
Iteration2	470.89
Iteration3	466.67

Iteration4	446.83
Iteration5	488.09
Iteration6	445.43
Iteration7	472.57
Iteration8	475.98
Iteration9	446.88
Iteration10	496.85
Average time = 470.58ms	

Table. 4.5: Time taken to execute CUDA C code for 256 threads/Block

For threads per block = 512	
Iteration Number	**Time Taken(ms)**
Iteration1	458.08
Iteration2	486.43
Iteration3	479.09
Iteration4	461.27
Iteration5	479.77
Iteration6	448.35
Iteration7	486.28
Iteration8	454.44

Iteration9	453.18
Iteration10	466.96
Average time = 467.38ms	

Table. 4.6: Time taken to execute CUDA C code for 512 threads/Block

After observing the results and comparing these values obtained on CPU and GPU by taking the ratio of time taken, we can say that we achieved **2.9 – 3.3 times acceleration** or 2.9 – 3.3 times less execution time in executing the algorithm on GPU as compared to its CPU counterpart.

4.2 Results on FPGA

We first synthesize the C code in the Vivado HLS, and we got to know the performance and utilization estimates sequentially. Later the directives, pipelining and unrolling were applied, and we calculated the estimates again by the synthesis. The results presented here show us the estimates as well as the comparison between the sequential and optimized process.

Figure below shows the latency (clock cycles) of the HLS IP without applying directives.

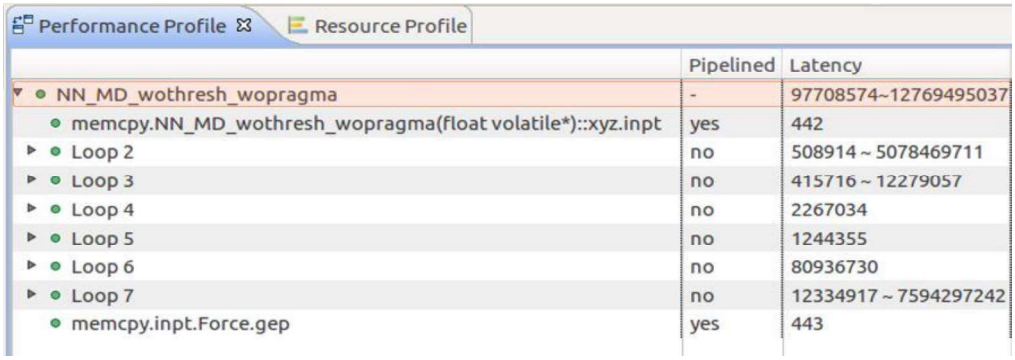


Fig. 4.1: Performance profile showing the latency report without directives

As can be observed, **minimum latency** or the minimum number of clock cycles taken is **97708574**.

Now, we will apply pipeline directive on loop 3, loop 4, loop 5, loop 6. These loops take 84863835 clock cycles. Ratio (r) of code on which pipeline directive is applied can be calculated as:

$$r = \frac{84863835}{97708574} = 0.87$$

We can also call it as $\text{Fraction}_{\text{Enhancement}}$.

Figure below shows the report after applying directives:

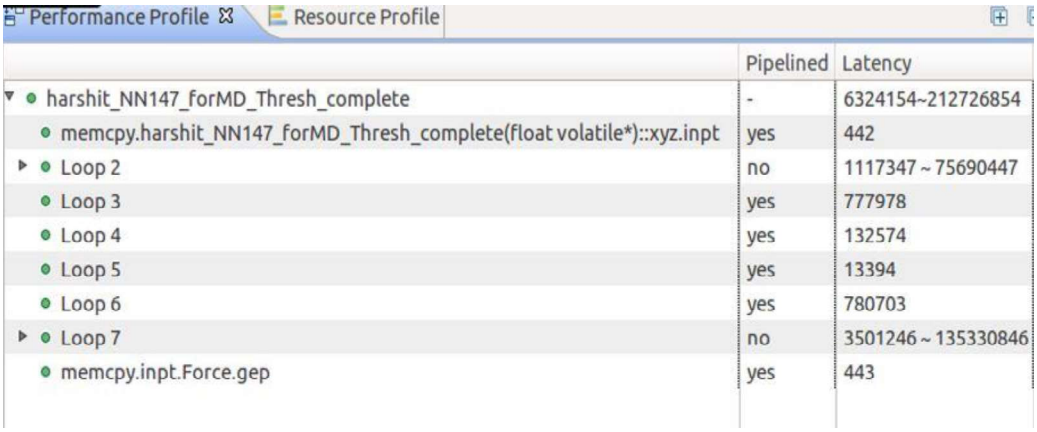


Fig. 4.2: Performance profile showing the latency report with directives

According to Amdahl's law, Speed up can be calculated as:

$$Speed\ up = \frac{1}{1 - Fraction_{Enhancement} + \frac{Fraction_{Enhancement}}{Speed\ up_{Enhancement}}}$$

Speed up for the enhanced fraction can be calculated as:

$$Speed\ up_{Enhancement} = \frac{415716+2267034+1244355+80936730}{777978+132574+13394+780703} = \frac{84863835}{1704649}$$

$$Speed\ up_{Enhancement} = 49.78$$

After substituting the above value, Speed up can be calculated as:

$$Speed\ up = \frac{1}{1-0.87+\frac{0.87}{49.78}}$$

$$Speed\ up = 6.8$$

To run this HPC algorithm on FPGA, the design consists of both hardware and software parts [5]. The software module present in the computer provides the different XYZ coordinates to the FPGA board. The communication between the software module and FPGA is established through PCIe communication. DMA subsystem for PCIe IP available in Xilinx Vivado HLx edition enables host computer to access memory that resides in the FPGA board. Force module would calculate forces and energy and again stores these results in the DDR memory. After storing these values, we need to transfer all the values to host PC. We need a mechanism so that we get to know as and when these calculations are done, these calculated values can be transferred to host PC. For the mentioned mechanism, we are continuously checking a particular memory location described in chapter 3 of the thesis. The software module takes up these forces and calculates the next set of XYZ values which are again fed back to the FPGA fabric. This forms a loop that can run multiple times. These iterations are known as MD steps. In order to get the comparison of performance of MD simulation on FPGA with respect to performance on HPC server, it is important to run the MD code while applying directives as discussed above. Table 4.7 shows the total time taken to run 1, 100 and 500 MD steps and its comparison with the code (already done in past in lab) that is parallelized using MPI on a HPC server.

MD Steps	HPC Server Timings	FPGA Timings
1	4.18 s	2.91 s
100	3.51 min	2.18 min
500	17.43 min	11.49 min

Table. 4.7: Comparison of timings on Server and FPGA

As it can be inferred from table 4.7, time taken to run different MD steps on FPGA is much less as compared to the time taken on HPC server. A performance acceleration of 1.4 - 1.6x is achieved using FPGA in comparison to server timings.

Chapter 5

Conclusions and Future Work

In this project, we have tried to reduce the complexity during calculations, reducing the latency by using devices, the GPU and FPGA board. The use of GPU lets us do the calculations faster, reducing the latency as compared to our sequential process. It has been explained above how we used CUDA, a parallel computing platform developed for harnessing the power of NVIDIA GPUs containing thousands of cores to achieve massive parallelism. We first parallelized the algorithm to calculate the distance between 147 atoms positioned in 3D space. Later, we moved to parallelize the algorithm of angular description calculation which is being used to calculate Energy, Force values.

Thereafter, we used high end FPGA device, Kintex KC 705. The sequential code for Energy, Force calculations is brought to HLS software where we calculated the speed achieved after applying directives. After synthesis, created an IP for the same, exported to Vivado HLx, implemented the block design for its hardware implementation, and transferred the input and output values between host PC and FPGA device using PCIe protocol.

The future work we can apply to this project can be:

- As we have parallelized the algorithm for calculation of angular description of atoms used to calculate Energy and Force values, the work can be extended to parallelize the complete algorithm for calculating Energy and Force values.
- Obviously, the architecture can be reprogrammed according to user needs and the parallelizing techniques can always be improved according to the increasing demand in development models of hardware implemented computation modules.

(This page is left blank intentionally)

References

[1]. Moore, G. E. (1965). Cramming more components onto integrated circuits.

[2]. Nickolls, J., & Dally, W. J. (2010). The GPU computing era. IEEE micro, 30(2), 56-69.

[3]. Ouyang, J., Wu, E., Wang, J., Li, Y., & Xie, H. (2017). XPU: A programmable FPGA accelerator for diverse workloads. 2017 IEEE Hot Chips, 29.

[4]. Jindal, S., & Bulusu, S. S. (2018). A transferable artificial neural network model for atomic forces in nanoparticles. The Journal of Chemical Physics, 149(19), 194101.

[5]. Bulusu, S. S., & Vasudevan, S. (2022). FPGA Accelerator for Machine Learning Interatomic Potential-Based Molecular Dynamics of Gold Nanoparticles. IEEE Access, 10, 40338-40347.

[6]. Tendler, J. M., Dodson, J. S., Fields, J. S., Le, H., & Sinharoy, B. (2002). POWER4 system microarchitecture. IBM Journal of Research and Development, 46(1), 5-25.

[7]. He, Y., Pu, Y., Kleihorst, R., Ye, Z., Abbo, A. A., Londono, S. M., & Corporaal, H. (2010, June). Xetal-Pro: An ultra-low energy and high throughput SIMD processor. In Proceedings of the 47th Design Automation Conference (pp. 543-548).

[8]. Luebke, D., & Humphreys, G. (2007). How gpus work. Computer, 40(2), 96-100.

[9]. Crow, T. S. (2004). Evolution of the graphical processing unit. A professional paper submitted in partial fulfillment of the requirements for the degree of Master of Science with a major in Computer Science, University of Nevada, Reno.

[10]. NVIDIA Corp., NVIDIA Homepage retrieved November 2004 http://www.nvidia.com/page/home

[11]. ATI, ATI Homepage retrieved November 2004 http://www.ati.com/

[12]. Matrox, Matrox Homepage retrieved November 2004 http://www.matrox.com/

[13]. NVIDIA Corp., “3D Graphics Demystified”, retrieved November 2004 http://developer.nvidia.com/object/3d_graphics_demystified.html

[14]. Kuon, I., Tessier, R., & Rose, J. (2008). FPGA architecture: Survey and challenges. Now Publishers Inc.

[15]. Minnick, R. C. (1967). A survey of microcellular research. Journal of the ACM (JACM), 14(2), 203-241.

[16]. Maitra, K. K. (1962). Cascaded switching networks of two-input flexible cells. IRE Transactions on Electronic Computers, (2), 136-143.

[17]. Birkner, J. M., & Chua, H. T. (1978). U.S. Patent No. 4,124,899. Washington, DC: U.S. Patent and Trademark Office.

[18]. Wahlstrom, S. E. (1967). Programmable logic arrays--cheaper by the millions.

[19]. Carter, W. (1986). A user programmable reconfigurable gate array. In Proc. Custom Integrated Circuits Conf., May 1986.

[20]. Kintex, X. (7). FPGA KC705 evaluation kit. Accessed: Aug, 30, 2017.

[21]. Meselhi, M. A., Elsayed, S. M., Essam, D. L., & Sarker, R. A. (2017, December). Fast differential evolution for big optimization. In 2017 11th International Conference on Software, Knowledge, Information Management and Applications (SKIMA) (pp. 1-6). IEEE.

[22]. How To Run CUDA C/C++ on Jupyter notebook in Google Colaboratory (2022) https://www.geeksforgeeks.org/how-to-run-cuda-c-c-on-jupyter-notebook-in-google-colaboratory/

[23]. CUDA TRAINING SERIES (2020) https://www.olcf.ornl.gov/cuda-training-series/https://www.olcf.ornl.gov/wp-content/uploads/2019/12/01-CUDA-C-Basics.pdf

[24]. GPU (2020), https://hackmd.io/@yaohsiaopid/ryHNKkxTr

[25]. CUDA Refresher: The CUDA Programming Model (2020), https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

[26]. DMA Subsystem for PCI Express - Driver and IP Debug Guide (2018) https://support.xilinx.com/s/article/71435?language=en_US

[27]. Jackson, M., Budruk, R., Winkles, J., & Anderson, D. (2012). PCI Express Technology 3.0. Mindshare press.

[28]. DMA Subsystem for PCI Express Product Guide(PG195) (2021) https://docs.xilinx.com/v/u/en-US/pg195-pcie-dma