CAPITAL UNIVERSITY OF SCIENCE AND TECHNOLOGY, ISLAMABAD

Analysis of OpenCl Application on CPU and GPU

by

Muhammad Nadeem Nadir

A thesis submitted in partial fulfillment for the degree of Master of Science

in the

Faculty of Computing
Department of Computer Science

2021

My dissertation work is devoted to My Family, My Teachers and My Friends. I have a special feeling of gratitude for My beloved ammi, abu, brothers and sisters. Special thanks to my supervisor whose uncountable confidence enabled me to reach this milestone.I would also like to dedicate my work to my wife whom I have not yet married and to those of my children who have not yet been born.

CERTIFICATE OF APPROVAL

Analysis of OpenCl Application on CPU and GPU

by

Muhammad Nadeem Nadir

(MCS183008)

THESIS EXAMINING COMMITTEE

S. No.	Examiner	Name	Organization
(a)	External Examiner	Dr. Muhammad Aleem	FAST-NUCES, Islamabad
(b)	Internal Examiner	Dr. Nadeem Anjum	CUST, Islamabad
(c)	Supervisor	Dr. Muhammad Siraj	CUST, Islamabad

Dr. Muhammad Siraj
Thesis Supervisor
May, 2021

Dr. Nayyer Masood	Dr. Muhammad Abdul Qadir
Head	Dean
Dept. of Computer Science	Faculty of Computing
May, 2021	May, 2021

Author's Declaration

I, **Muhammad Nadeem Nadir** hereby state that my MS thesis titled "**Analysis of OpenCl Application on CPU and GPU**" is my own work and has not been submitted previously by me for taking any degree from Capital University of Science and Technology, Islamabad or anywhere else in the country/abroad.

At any time if my statement is found to be incorrect even after my graduation, the University has the right to withdraw my MS Degree.

(Muhammad Nadeem Nadir)

Registration No: MCS183008

Plagiarism Undertaking

I solemnly declare that research work presented in this thesis titled "**Analysis of OpenCl Application on CPU and GPU**" is solely my research work with no significant contribution from any other person. Small contribution/help wherever taken has been duly acknowledged and that complete thesis has been written by me.

I understand the zero tolerance policy of the HEC and Capital University of Science and Technology towards plagiarism. Therefore, I as an author of the above titled thesis declare that no portion of my thesis has been plagiarized and any material used as reference is properly referred/cited.

I undertake that if I am found guilty of any formal plagiarism in the above titled thesis even after award of MS Degree, the University reserves the right to withdraw/revoke my MS degree and that HEC and the University have the right to publish my name on the HEC/University website on which names of students are placed who submitted plagiarized work.

(Muhammad Nadeem Nadir)
Registration No: MCS183008

Acknowledgement

All praise is to ALLAH (S.W.T). The creator and Sustainer of all seen and unseen worlds. First and foremost, I would like to express my gratitude to ALLAH Almighty for his countless blessings upon me to complete this work. Secondly, I would like to express my sincerest thanks to my supervisor **Dr. Muhammad Siraj** and **Dr. Maryam Abdul Ghafoor** for his valuable guidance and assistance. I am sincerely grateful for his support, encouragement, and technical advice in the research area. He has taught me, both consciously and unconsciously, many techniques and strategies during my research. I pray for his wellbeing and success in the future.

I am highly indebted to my parents and my family, for their support, assistance and encouragement throughout the completion of my degree. They form the most important part of my life after ALLAH (S.W.T). They are the sole source of my being in this world. No words can ever be sufficient for the gratitude I have for my parents and for my family. A special thanks to my sister for their support and encouragement during my studies. I would also like to extend my gratitude to **Dr. Yasir Nouman Khalid**, assistant professor at HITEC University and **Mr. Muhammad Aif** a Ph.D. Scholar at CUST and **Mr. Muhammad Usman**, Lecturer Fast University and a generous friend for his valuable support and guidance.

Finally, I am very thankfull to **Mr. Asad Hayyat** and **Mr. Baber Naeem** MS students at CUST, with whom I completed this thesis.

I pray to ALLAH (S.W.T) that may He bestow me with true success in all fields in both worlds and shower His blessings upon me for the betterment of all Muslims and the whole Mankind.

(Muhammad Nadeem Nadir)

Abstract

The idea of accelerating an application's execution speed by running it on the GPU is GPU acceleration. For their applications, researchers and developers have always tried to attain greater speed and GPU acceleration is a very common way of doing so. For highly graphical applications using powerful dedicated GPUs, this has been achieved for a long time. Researchers, however, have been increasingly interested in using GPU acceleration for regular applications. Furthermore, it has been observed in the literature, even though the application is well suited for parallelism it is not guaranteed to run faster on the GPU. Therefore the purpose of this thesis is to examine the performance of OpenCL applications by executing each application on different architecture to find out which application is GPU suitable and which one is CPU suitable. After executing the OpenCL application on CPU-GPU with different input sizes compare the execution time of both architecture and find out which software feature affecting the performance. Some application like 2DCONV, 3DCONV, and FDTD-2D execution is fast on GPU as compare to CPU for all input size in the same way ATAX application execution is fast on CPU than GPU these applications. There are some applications like 2MM, 3MM, and GEMM which execute fast on GPU for small and large input size but some middle input size CPU perform better than GPU, in the same way, some applications like BICG, GESUMV, MVT, Covariance, and Correlation which executes fast on GPU for small input size but when input size increase CPU become fast as compared to GPU.

Contents

List of Figures

List of Tables

Abbreviations

CPU	Central Processing Unit
GPU	Graphics Processing Unit
GPGPU	General Purpose Graphics Processing Unit
ILP	Instruction Level Parallelism
MIMD	Multiple Instructions Multiple Data
OpenCL	Open Compute Language
SIMD	Single Instruction Multiple Data
TLP	Thread Level Parallelsim

Chapter 1

Introduction

Engineers have often sought to create computers fast to solve more complicated problems the way to do this has traditionally been to increase the central processing unit (CPU) clock speed. Due to the power wall, instruction-level parallelism (running separate instructions in parallel), and memory wall, however, the clock speed has more or less stopped increasing. The power wall means that the frequency of the processor clock can't be increased because it produces too much heat. The "ILP wall" means that, due to control and data dependency, you can't execute more than 3-4 parallel instructions on the same processor. The "memory wall" suggests that the memory access speed is lagging enough to impose a processor speed limit [1].

As a result, the focus has changed from increasing the speed of single-core to using parallelization through programming. Parallelization enables a multi-core processor unit to execute several tasks in parallel, ensuring quick execution. Now a days in most computers these days have some type of graphics processing units (GPU). In computer GPUs (Graphics processing units) were developed.

In the last 20 years or so, GPUs have become popular. For high-performance general-purpose computing, like Processing of video, images medical, video, and processing images.Until now, however, GPUs combined with high performance and low costs, have been used to improve computational programmability.

It has been observed in recent years that the performance capabilities of the GPU have, in some cases, overcome those of the CPU. Which in turn, inspired the creation of the graphical processing unit for general purpose (GPGPU) [2]. This has not only led to the use of the GPU in graphical applications but also led to the use of GPU for other applications like scientific calculations, linear algebra, data mining, and convolutions. These trends have led to a boom in the design of graphical computing. Now new product lines were introduced by manufacturers specifically for scientific measurements.

Now, after these improvements, one is always concerned to see if it is feasible to use the GPU for calculations of such tasks, to achieve some improved performance. Job like processing of an image, medical image, physical modeling, and linear programming, as well as most applications, are well parallelized based on GPU [3]. Nevertheless, because of the SIMD (Signal Instruction Multiple Data) architectures, task-level parallelism is best used in Graphics Processing Unit [4] and MIMD (Multiple Instruction Multiple Data) architectures, task-level parallelism is best used in general-purpose processors [5].

This work select benchmark programs from a widely used benchmark set to achieve a more general comparison of the GPU and the CPU. To rule out the probability that the outcome depends on the author's programming abilities,rather than the hardware, the benchmark application is often used. It is important to review their findings and code to figure out whether or not this particular GPU can be used for GPU acceleration. In this way a developer can choose a dedicated GPU to speed up their application.

Also when developing OpenCL programs for dedicated GPUs, it can offer realistic lessons for what a developer needs to remember. This work going to use Polybench suits used in [6].

Which consists of multiple algorithms with diffrent nature some of applications belong to image processing , linear algebra and data mining. In this work going to run those algorithms on CPU and GPU. Having a better picture of what's behind it, In this effects, performance analyzes the result to find, when GPU implementation has a faster execution time than CPU execution time, and similarly if CPU execution faster than GPU,what is the reason behind this speedup and what type of parameters it focuses on i.e. The number of dimension in data, loop unrolling the size of the problem, number of cores, branch prediction.

1.1 Purpose

With technological advances, GPUs are largely used in addition to the traditional CPUs. We can maximize throughput through better utilization of both architectures. In this work we will analyze the characteristics of the program as well as the architecture on which it is being run to find the factors which contribute to increasing in GPU and/or CPU utilization.
In this scenario, a performance benefit is defined as: A speedup of execution time can be found when running the application on the GPU relative to how it had been run on the CPU. This will also help the programmer in designing applications suitable for GPU and CPU.

1.2 Scope

In this thesis, different nature of applications are executed on different CPU-GPU architectures to analyze the performance of different architecture on the behalf of a software feature.
In comparison, the cores have access to advanced operations such as branch prediction and are quicker and more stable relative to GPU cores in general. This gives further understanding of applications that are suitable for CPU or GPU and what needs to be considered when writing an OpenCL application for a specific architecture.

1.3 Delimitations

The testing of Pollybench suit application implementations will be addressed in this thesis. This implies that there would not be any implementation from scratch. Also, to compare the effects of two different architectures on the performance of application, the study of the execution times will be carried out.The specific concentration is on the time of execution, not on power usage. We are not discusing about the power consumption.

1.4 Problem Statement

it has been observed in the literature [7] [43] [46], even though the application is well suited for parallelism it is not guaranteed to run faster on the GPU. It means execution time of an application depends on the other factors as well and how can you determine whether particular architecture is suitable or not for general purpose application therefor, OpenCL applications analysis is required to find out such factors which can affect the performance of application. Therefore, an in-depth analysis of OpenCL applications is done, to identify which software feature affect the performance of the application on CPU-GPU architecture.

1.5 Research Question

Based on the research gap stated in the previous section, the following research questions were formulated in this thesis:
RQ1: What is the better architecture (CPU-GPU) in term of performance for these applications ?
RQ2: What are the most important factors which affect the execution time of different OpenCL applications on CPU vs GPU ?

Chapter 2

Backgroud

All theories concerning platforms, OpenCL, parallel algorithms, architectures and features are discussed in this chapter.

2.1 Parallel Computing

The computational method of executing many instructions in parallel is parallel computation. This is done by concurrently running several threads on several cores, rather than running it on one core sequentially as informal programming. Unlike sequential programming, where the Von Neumann model is the dominant programming model, there are two different architectures for parallel programming [8].

2.2 Distributed Memory and Shared Memory

Multiple processors sharing the same memory and a global clock controlling all memory and processors are part of shared memory architecture.Distributed memory systems are systems with their memory consisting of several processors. Using a message-passing protocol over an interconnection network, these processors communicate with each other [9]. There are the distributed and shared architectures of memory, as well as a variety of various distributed and shared structures that are ideally suited to various problems. In large clusters, the distributed memory architecture is mostly used when you have multiple processor nodes running together over an interconnection network.

2.3 The Data Parallel Computing Model

Although there are many types of parallel computing models our focus is on data-parallel. This is because it is one of the two OpenCL-supported ones that are suitable for GPU computing [10].
With SIMD execution, data parallelism operates very well because executing the same operational portion wisely on large collections of independent data revolves around achieving parallelism. The activities may be performed in parallel because the data is independent of one another. All programs evaluated in this thesis make use of parallel algorithm.

2.4 SIMD

Computers with multiple processing elements that perform the same operation simultaneously on multiple data points are defined as Signal Instruction Multiple Data (SIMD)[11]. SIMD architectures are made up of a variety of processors that run in a synchronous manner, with each processor executing the same instruction on a different data at the same time.

2.5 MIMD

Computers with multiple processing elements that perform the different operations simultaneously on multiple data points are defined as Multiple Instruction Multiple Data (MIMD)[11]. This is the most common and efficient architecture, and most practical implementations include MIMD machines. When solving a single problem, all processors can be executing different subproblems on different data.

2.6 CPU Architecure

CPUs are built for general application i.e, A CPU must be able to manage both parallel and sequential programs and whatever problem is given to it in general. Therefore, CPUs depends on MIMD execution, which ensures that the separate cores are issued with different operations. This is done so that multiple operations on different data can be performed by the various cores of multi-core CPUs.
As a consequence, a core can run a whole program on its own, or the core can break the program between itself and split the workload. In comparison, the cores have access to advanced operations such as branch prediction and are quicker and more stable relative to GPU cores in general [12][13]. In this work Intel i7-4700 CPU is used. The next section explains the Intel i7-4700 CPU in detail.

2.7 Intel i7-6700 CPU

One of Intel's sixth-generation (Haswell) i7 processors is the Intel i7-6700. It has four cores, eight (two per core) threads and a 3.40 GHz base clock frequency. If required, the Intel turbo boost feature can also accelerate one of the cores to 4 GHz. Also, it has 3 levels of cache where each core is separated from the first (256kb) and second (1 megabyte), while the third (8 megabytes) is shared between the cores34.1GB/s. Max Memory bandwidth is the maximum rate at which the processor can read data from or store in a semiconductor memory.
The memory system also facilitates transactional memory as well as gather instructions. The transactional synchronization extension allows transactional memory to be supported. In practice, it ensures that operations are secured by an atomic lock.
This means that several threads will run them at the same time as long as they don't perform overlapping operations on the files [14].

2.8 GPU

In comparison to CPUs, modern GPUs are constructed with the expectation that incredibly parallel tasks are assigned to them and that they focus on high throughput. This indicates GPU is not powerful like CPU, but for speedup of several parallel issues, GPU can be used [15]. Execution of SIMD, a very large number of small and basic execution cores and abundant use of multithreading hardware [16]. .

- You execute the same operation with SIMD execution on multiple cores that control different data. You can effectively execute the same procedure on huge collections of data in this manner [16].
- In GPUs, instead of CPUs, more simple cores are used, but GPU cores are a little complicated. The only feature that these small cores compromise is the probability of execution out of sequence and branch estimation. But for the chip, the spatial savings result in more cores that exceed these compromises. This works well for SIMD, too [16].
- Hardware multithreading allows the decomposition of extremely simultaneous computations into several serial tasks equal to an even greater degree of parallelism [16].

2.9 Discrete Graphics Processing Unit

The Intel i7-6700 is a CPU with a discrete GPU, as mentioned above. Discrete graphics belongs to a graphics device that is independent from the processor. Discrete GPU has its physical memory as a separate computing unit from the CPU.

The PCI-Express bus is used to transmit data between CPU and GPU memory. [17]. NVIDIA Geforce GT 740 discrete GPU is used in this thesis beside Integrated GPU becuse discrete GPU has it own memory. It has 384 cores and 28.8 GB/s memory bandwidth [18].

2.10 OpenCL

Software developers should consider portability when developing their applications because of the diversity of architectures. For any new architecture, both rewriting an application and retaining multiple branches of a code are resource-intensive functions. Some programming languages and frameworks to prevent rewriting codes e.g. OpenCL [19], OpenMP [20], OpenACC [21], Kokkos [22], etc. allow different platforms to run the same source code.
The Open Computing Language (OpenCL) [19] is an open Khronos Group standard [23]. It aims to provide heterogeneous architectures with a parallel computing framework. The majority of available CPUs, GPUs, or accelerators on the market have an OpenCL implementation that works. This helps an application on a given computer to harness the computing power of the multiple enabled devices. OpenCL offers both an Application Programming Interface (API) and a programming language to achieve this purpose.
To communicate with the OpenCL runtime model, the OpenCL and the language OpenCL C is used to program kernels that map to the OpenCL system architecture model[19]. OpenCL helps to run the same program on two different architectures. This is generally not feasible since the architectures of a GPU and a CPU are inherently incompatible. Using the MIMD model, to begin with, CPUs while using SIMD for GPUs. Also, there are not only variations between CPUs and GPUs, but there can also be significant differences between different GPU models in the way parallel code is written.
That implies that the code can vary from model to model. OpenCL fixes this problem [24]. In this thesis concentrates on analyzing the execution of the OpenCL program on two different architectures.

2.10.1 Platform Model

OpenCL is an open platform specifically for parallel programming that produces highly portable code, this means that different platforms use the same code. Used by OpenCL, the model consists of a host and many devices. The host maps to the main program control core, For example, if the program is running on a processor,

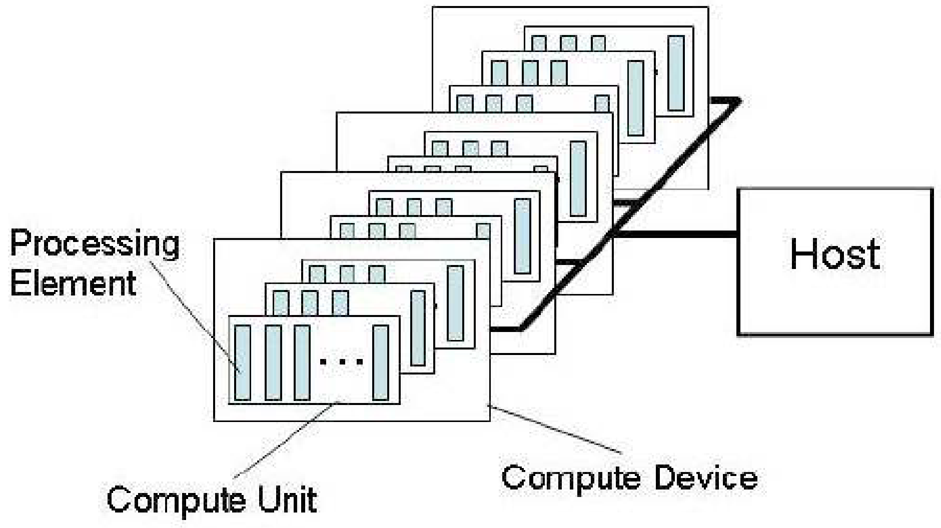


FIGURE 2.1: Platform model.one host plus one or more compute devices each with one or more compute units. [10].

it maps to the core that begins and merges the threads, and it maps to the system CPU if it is running on a GPU. As a consequence, the modules will either map to the rest of the cores on the CPU or to the cores of the GPU [10][25]. The Platform model for OpenCL is defined in Figure 2.1. The model consists of a host to which one or more OpenCL devices are attached.

2.10.2 Memory Model

The memory model of OpenCL consists of two primary types of memory, host memory and device memory. Not unexpectedly, the host memory is the memory accessed by the host, and the device memory is the memory accessed by the devices. In contrast, four memory regions of the device memory consist of were based on what type of data it is, the kernels will distribute data, and which kernels need access to it.

In all workgroups, two global memory regions can be accessed by anyone. The first one is global memory. In this memory area provides access to all work-items in all work-groups to read/write and the second one is constant memory in this type of memory data remains stable throughout execution time. Device can read data from it but the host can allocate data in constant memory. There is a memory area that is an independent region of each kernel that is a private memory. Finally,

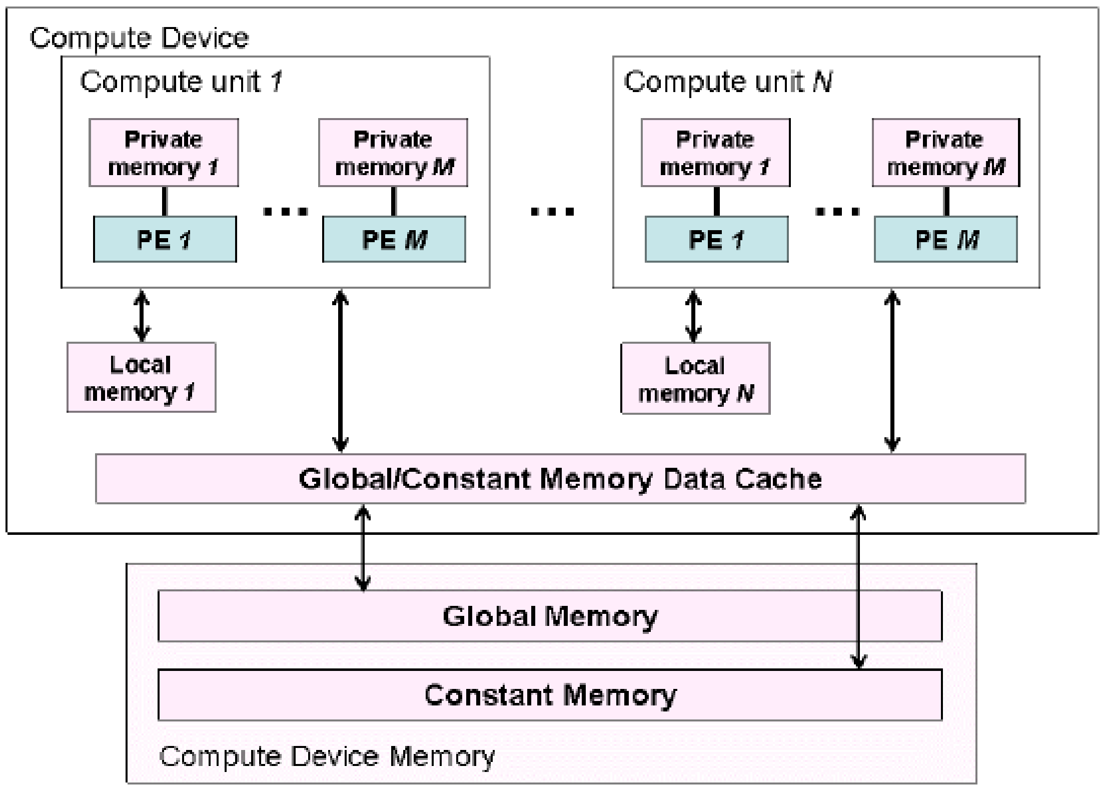


FIGURE 2.2: Conceptual OpenCL device architecture with processing elements (PE), compute units and devices. [10].

there is a local memory that acts as a shared memory for devices in the same workgroup, ensuring it can be accessed by all devices in the same workgroup [10] [25]. The memory regions and how they relate to the platform model are described in 2.2

2.10.3 Execution Model

Two entities are mapped to the host and the devices in the platform in the execution model. There are kernels and a host application. The kernels are run on the devices.

Either the kernels are mapped one to one, or one to many, this mean kernel program can run on one device and can run on many devices. However, only one host program can run simultaneously on the host.

The host program is the only program that does everything, such as memory control and synchronization, while the computing work is done by the kernels. This work is carried out in so-called work-items most generally known as program

threads, and in so-called work-groups, they work together.
In a context consisting of the devices it executes on, the kernels are specified, OpenCL functions, the actual implementation of the kernel, as well as the memory needed for the variables on which it runs.
Every kernel context is handled via a command queue by the host program that can be loaded from the OpenCL API with functions. The host launches new kernels via the command-queue. For example, it transfers data between host and device memory and handles synchronization between various kernels.
Also, each kernel may enqueue commands to a particular command queue for the system on which it is running. New child kernels can be initiated from this command-queue [10] [25].

2.10.4 OpenCL for CPUs

Although OpenCL has been developed for portability between various architectures, GPU architectures are the key target and what it has mainly been used for. As a consequence, the performance is not as portable, although the code can run on both CPUs and GPUs [26]. Any of the particular optimizations in OpenCL for a GPU architecture may potentially have the reverse effect on a CPU.
First, the OpenCL work-items are very small, which suits the small GPU execution units, However, it's something of a mismatch to map these basic tasks to a complex CPU core.
This mismatch results in the cache use of the CPU being limited, That only a small part of the whole issue is processed by the work-item running on one hardware thread. Secondly, inside the CPU cores, the SIMD units require vectorized data that you don't typically need in GPU code.
This means that the processing capacity of any of the cores would not be used. Also, it would result in unnecessary data transfer between host and device if the default data transfer model is used on GPUs. Of course, OpenCL code can also be designed for CPUs if you run it on a GPU, which can harm performance [10].
In this work Opencl application are used for experiment to find out the execution time of two different divice.

2.11 Dimensions and Work-items

An index space is defined when a kernel is submitted for execution by the host. For each point in this index space, an instance of the kernel runs. This instance of a kernel is called a work-item and is defined by its point in the index space, which provides the work-item with a global ID. Each work-item executes the same code, but the specific execution pathway will vary by work-item across the code and the data operated on. Work-items are organized into working groups. The work-groups include the index space for a more coarse-grained decomposition. A special work-group ID with the same dimensional space as the index space used by work-items is assigned to the work-groups. Work-items within a work-group are given a specific local ID such that a particular work-item can be identified uniquely by its global ID or by a combination of its local ID and work-group ID. The work-items operate simultaneously on the processing elements of a single computing unit in a given work-group. The index space that OpenCL supports is called NDRange. An NDRange is an index space of the N-dimensional, where N is one, two, or three. An NDRange is specified by an integer array of length N indicating the degree of each dimension's index space. global ID and local ID are N-dimensional tuples of each work item [10].

For GPUs, in which groups of processing cores function in a SIMD fashion, data-parallel tasks are appropriate. A data-parallel task in OpenCL is represented as a kernel representing the processing of a single work item.

A user-specified number of work-items are launched to run in parallel during program execution. In a multidimensional grid, certain work-items are ordered and Work-item subsets are combined to form workgroups, that enables the interaction of work-items [27]. The difference in performance arises from the various architectural features that occur between CPUs and GPUs.

A scalar processor (SP) or one single SIMD lane processes a single work-item on GPUs. As is well known, GPUs are designed to accommodate a large number of threads running simultaneously, and high thread-level parallelism (TLP) is important for high performance [28–32]. The number of dimensions necessary for the ID of a work-item typically equals the number of indices you can use to access an array variable that contains the data of the work-item. Suppose, for instance,

that input data is stored in an array called point. If you usually use point[x][y] to access the data, the number of dimensions is two. If you are using point[x][y][z] to access the details, the number of dimensions is three [10]. One function only needs to be known: clEnqueueNDRangeKernel. This is one of the most important features of the OpenCL clEnqueueNDRangeKernel is used to control how the kernel operates. Execution is divided among the computing resources of the system. in OpenCL clEnqueueNDRangeKernel signature is
clEnqueueNDRangeKernel(cl_command_queue queue, cl_kernel kernel, cl_uint work_dims, const size_t *global_work_offset, const size_t global_work_size, const size_t *local_work_size, cl_uint num_events, const cl_event *wait_list, cl_event *event)
In this function, the second argument is work_dims which represent the number of data dimensions and in this work dimension of data has a strong impact on our results impact of dimension is discussed in the result and analysis section.

2.12 Loop Unrolling

Loop unrolling is the method of reducing the power of the loop by rising the body size loop. This mechanism can be seen as repetitive individual statements re-writing the loop body. In many OpenCL/CUDA programs, loop unrolling is used [33] [34]. The key advantages obtained from loop unrolling are: Due to fewer comparisons and branch behaviors with the same amount of work performed, the decreased complex instruction count.
The compiler's scheduler will use these instructions to increase Instruction Level Parallelism (ILP) and hide pipeline and memory access latencies to improve scheduling opportunities due to the availability of additional independent instructions. Opportunities to exploit the position of the register and memory hierarchy as external loops are unrolled and internal loops are fused [34]. loop unrolling is a helpful optimization for GPGPU programs. [35] To parallelize the process of image convolution with CUDA [35] suggested the strategy of loop unrolling. This method requires lots of iterations, but a few computations were used in each iteration. The unrolling of the loop thus increased the calculations per iteration and

decreased the overall number of iterations [36].
The technique of loop interchange will improve the efficiency of a parallel code. If no dependencies exist in the body of two nested loops, nested loops may be interchanged. [37] To simplify the parallel code, the loop interchange technique [37]. On several tasks, like the Laplace filter to a 256*256-pixel image, they then evaluated the proposed parallel code. In this work loop unrolling has a strong impact on our result. Impact of loop unrolling is discussed in the result and analysis section.

2.13 Parallel Algorithms and Data parallel Algorithms

Parallel algorithms are algorithms programmed to execute various instructions in one clock cycle. However, the algorithm in question has to include many parallelization possibilities to achieve output from parallelizing an algorithm [38]. In other words, to be able to break the problem into sections, the algorithm can manage a very broad problem and have a suitable division such that as many as possible can be run in parallel.
Algorithms that are specifically appropriate for this are, for example, convolution and matrix operations. In this work, eleven different parallel data algorithms from Pollybenchmark suit [6] were used to compare CPU and GPU performance.

2.14 PolyBench Kernels

Polybench is a set of computation kernels used to test the performance of compilers and related applications, such as matrix multiplication, 2D or 3D convolution, or linear equation solver.

Mentioned application is consider as a core of many applications for high-performance like image processing [39]. In this work, we will therefore evaluate the poly bench suit [6]. The next session explains some details about each application of the polybench suit.

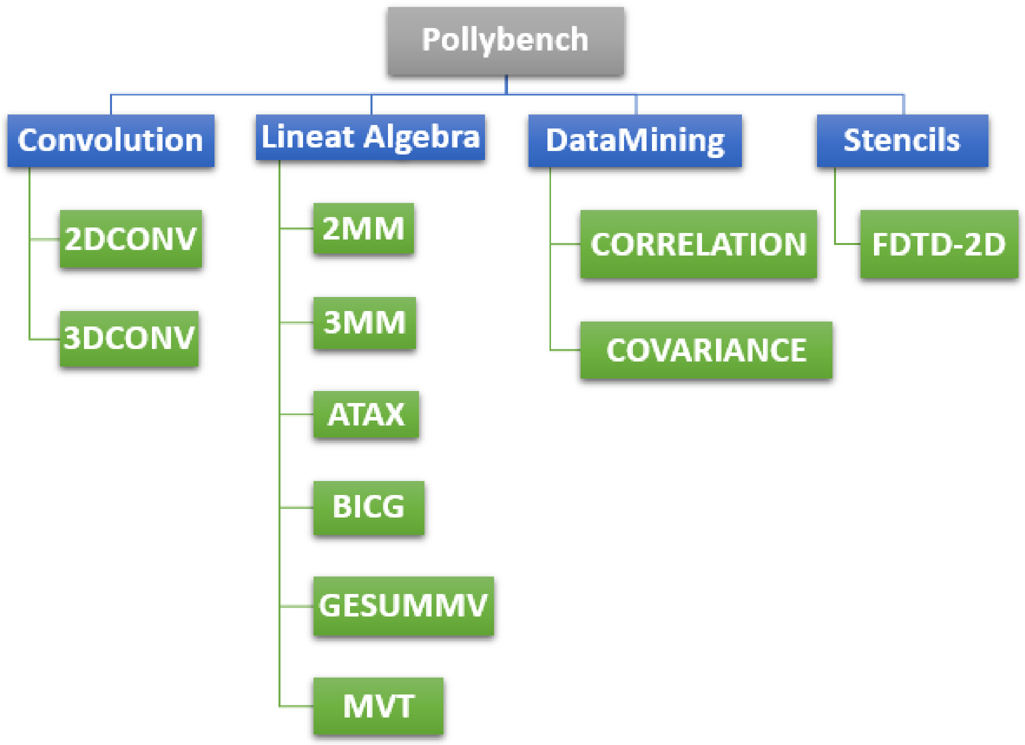


FIGURE 2.3: Application in Polybench suit that are tested in this work.

2.14.1 2MM

Two Matrix Multiplication (2MM) is one of the kernels of linear algebra that consists of two multiplications of matrixes [40].

In most the papers use this application for comparison of two architecture. In this application we give simple floating value as a input size. When we input to that application floating value in two 2 x 2 matrix beacuse it is 2mm application in which two matrix are multiped and find the performance of two device.

The following are provided as input:

$\alpha\beta$

A is a matrix of P x Q

B is a matrix of Q x R

C is a matrix of R x S

D is a matrix of S x T

Gives as output the following:

E P x S matrix, Where E = αABC + βD

2.14.2 3MM

Three Matrix3 Multiplication (3MM) is one of the kernels of linear algebra [40] that consists of three multiplications of matrixes and includes G= (A*B)*(C*D). that take the following as a input:

A is a matrix of P x Q
B is a matrix of Q x R
C is a matrix of R x S
D is a matrix of S x T
Gives as output the following: G P x T matrix, Where G = (A.B). (C.F).

2.14.3 ATAX

ATAX is one of the kernels of linear algebra [39] that computes A^T time Ax that takes the following input:

A is a matrix of M x N
x is the vector of N length
Gives as output the following: Y vector of length N, where y = A^T(Ax).

2.14.4 BICG

BiCGSTAB's Kernel (BiConjugate Gradient STABilized method). BICG is one of the kernels of linear algebra [40].
It takes the following input:
A is a matrix of N x M
p is the vector of M length
r is the vector of N length

Gives as output the following:

q is the vector of N length Where q = A_p s is the vector of M length Where = A^T r.

2.14.5 Gesummv

Gesummv is one of the kernels of linear algebra [40]. It takes the following input:

α β scalars

A, B is a matrix of N x N

X is the vector of N length

Gives as output the following: y is the vector of N length, Where y = αAx + βBx.

2.14.6 MVT

Matrix vector multiplication with a separate matrix-vector multiplication, but with a matrix transposed [40].

It takes the following input:

A is a matrix of N x N

y1, y2 is the vector of N length

Gives as output the following:

x1 is the vector of N length , Where x1 = x1 + Ay1

x2 is the vector of N length , Where x2 = x2 + A^Ty2.

2.14.7 Covariance

Compute the covariance, a mathematical measure that indicates how two variables are linearly connected [40].

It takes the following input:

Data: N x M matrix representing.

N data points, each with M attributes.

Gives as output the following

Cov: M x M matrix where the covariance between i and j is the i, j-th element.

2.14.8 Correlation

Correlation measures the coefficient correlation (Pearson's), which is normalized covariance[40].

It takes the following input:

Data: N x M matrix representing N data points, each with M attributes.

Gives as output the following

Cov: M x M matrix where the correlation coefficient between i and j is the i j-th element.

Chapter 3

Literature Review

GPU acceleration has been studied by many scientists since the rise of multi-core computers. V.W.Lee et al [7] examines how much performance GPU acceleration on an NVidia GTX280 GPU achieves compared to running it normally on an Intel Core i7-960 Processor. The comparison was conducted on the two processors by running 14 different throughput computing kernels. To capture the core computing and memory characteristics defined by the PARSEC and PARBOIL benchmark suite apps, the kernels are designed. Also, their performance study reviews the execution time of the various kernels was bound by and finally explains how to modify the code based on which platform you use. Their outcome indicates that although GPU acceleration is equivalent to performance improvement, it is not usually specified in the order of magnitude. The performance increase is about 2.5 time faster execution time on the GPU, according to their comparisons between the GTX 280 and i7-960. They figured that depending on which CPUs and GPUs were used and what kind of optimizations were used in the code.
M.Daga et al [41] analyses the accelerated AMD Fusion processing unit, a CPU with an integrated GPU. They compare this APU architecture more specifically to the more common of a discrete CPU combined with a discrete GPU and why APUs can beat the performance of discrete GPUs wisely. They also state that the main advantage of using an APU architecture is that you can get around the PCIe

performance bottleneck, as in many of the other works referenced here. Nonetheless, they also state that you can not necessarily get better results. The size of data needs to be very high to benefit performance by not having to submit data over the PCIe. There are no major efficiency gains for small data sets Moreover to model real-life workloads, they tested the architecture with four benchmark applications (MD, FFT, Search and Reduction) from the SHOC benchmark suite. The AMD Zacate APU, the AMD Radeon HD 5870 GPU, and the AMD Radeon HD 5450 were used to perform the test. The 5870 is a high-performance GPU, while the 5450 is an APU variant that is more or less discrete. MD runs fast on the 5870, but on the APU it runs faster than on the 5450. However, FFT on the APU is the slowest because it relies too much on the processing and memory which is very slow compare to the other two. However, given a big enough problem size, both scan and reduction ran fastest on the APU.

S.Azmat et al [42] analyses if the multi-modal mean (MMM) algorithm can be accelerated compared to running it on a single core Atom-330 CPU with Nvidia's low-powered ION GPU. MMM is an image processing algorithm that segments the background from the foreground and holds the background pixel values in running a program. Depending on the sort of context it represents, pixels have up to 4 modes and each mode contains different means of all the color components (i.e.RGB) of a pixel. All the optimizations performed on the CUDA platform are still comprehensive. Their finding shows that 6X speedups have been reached. Compared to (100fps) on a low-power integrated GPU platform, of similar power specifications to a CPU platform.

S.Kim et al [43] analyses the Intel HD Graphics 4600 GPU is used to speed up MapReduce tasks in the data center cluster system of Apache Hadoop and compared them to an equivalent CPU implementation. On a 4-node cluster and a 1-node cluster, the experiments were conducted and they used the HiBench benchmark suite to analyze. The metrics of performance they used were performed in time, consumption of power, and IO Overhead isolation. They simply measured the execution times and compared them to measure the efficiency with time. They ran the experiments several times to eliminate differences and used an average Since MapReduce is very IO-dependent, the tests they conducted separated the performance impact of the IO.

In actual comparison, they calculated how easily they could send data isolated from all other components to the map task. Finally, the power usage of all 4 nodes together on the 4-node cluster was calculated and the power consumption of the CPU and integrated GPU on the 1-node cluster was calculated. They concluded that the MapReduce task was transformed from a Compute bound kernel to an IO-bound kernel on the integrated GPU and that the GPU got a significant speed-up over the CPU.

E.Ching et al [44] authors using integrated GPUs in database processing for data bandwidth-dependent jobs, such as queries, instead of discrete GPUs. They used the Nvidia GTX 780 discrete GPU and the Intel HD 4600 integrated GPU in their tests. The memory interconnection speeds, the cache design, and the computing power are the key architectural differences. Cache design and interconnection speed in the favor of integrated GPU beside the discrete GPU has access to a bigger cache. As far as computing power is concerned, it is a little more complex, since the discreet GPU outperforms the incorporated nine times solely in floating-point operations. Nevertheless, the integrated one has many more data lanes on which instructions can be run, resulting in a larger amount of parallelism that favors many small jobs. The integrated GPU outperforms the discrete in this kind of bandwidth dependent workloads because of the faster memory links and the extra parallelism. In addition to running the database queries on the i7-4770k CPU, they also compared the integrated GPU. The CPU outperformed the GPU by a small amount (1-2 ms) at pure execution speed, but when it was normalized to power consumption, the GPU outperformed the CPU by a significant amount.

Another work introducing BLAS and contrasting the various architectural styles of CPU and GPU attributes. The various subprograms in BLAS were tested by detailed experiments and suggest a selection method for the processor that can lead the optimization of the main computational activity and author analyze the effectiveness of various BLAS activities on various platforms. Xeon E5-2620v3 uses the Haswell core which supports two CPUs at the same time, while the Kaby Lake core is used for Core i7 7700 and Core i5 7500. The Kepler architecture is Tesla K40c, while the Pascal architecture is GTX1080 Ti and GTX1070. The experiment shows that the result of the Core i7 7700 is higher than that of the Xeon E5 2620v2, but the three CPUs have poorer performance than that of the

GPU. When we execute GEMM and GEMV as the matrix dimension increases, the processing time increases on the CPU and GPU. GPU tests are faster than CPU experiments and in TSRV and if certain serial data dependencies remain, but since the sum operation can be done in parallel on the GPU, the impact of both the GPU is still greater than the CPU [45].

Z.Huang et al [46] authors work on matrix multiplication in machine learning, which is a popular and time-consuming computing process is implemented on various data scales and methods of development analyze the relationship between the efficiency of GPU computing with matrix scale and methods of growth. Two architecture used for the experiment first one was intel Xeon E5_2640 CPU and NVIDIA Tesla P100 GPU, the experimental studies show that in small-scale data estimation, the GPU output is not much improved relative to the central processing unit. In the estimation of the small-scale matrix, the GPU degree of parallelism is not massive, because many computer cores are not completely used and the performance difference compared to multi-core CPUs is not high but GPU is fully utilized when the matrix size increased.

V.Saahithyan et al [47] analyses the performance of Different fundamental image processing algorithms on the GPU and CPU. For testing, different images with a variety of dimensions were used. The findings reveal that the GPU's usability for problems with image processing is strongly dependent on the nature and size of the problem. As the matrix size gets greater and bigger GPU wins, the CPU continues to generate better performance up to a certain matrix size. In W.Thomas et al [48] defined an application as GPU or CPU suitable by analyzing factors, whether they are hardware components or software constructs, on which faster execution of program depends. In this study author thoroughly discusses the evolutionary journey of GPUs. Parameters taken into account for performance comparison are throughput and latency So, written with the Compute Unified System Architecture (CUDA) C language, based on the execution time of a GPU and processor for a specified task.With a change in workload size, the two parameters are measured. As the job size is raised, as the GPU reaches 100 % occupancy, the GPU is found to be around 51% faster than the multithreaded CPU. GPU's throughput is observed to be 2.1 times greater than that of the Large Job Size CPU. GeForce GT630M from NVIDIA with Intel's i-5 3210M 3rd generation CPU is used for the experiment.

Heterogeneous architecture has multi-core CPUs as well as many-core GPUs and performs parallel execution [49]. Task scheduling in heterogeneous architecture is a challenging job. An OpenCL framework is designed to perform task execution on heterogeneous architecture. Many features affect the scheduling of a task. Tasks like out-of-order execution, branch prediction, etc. are suitable for CPU while parallel execution of tasks is more suitable on GPU. The main theme of this research is to map OpenCL applications based on process capability and application/device suitability and is achieved through a machine learning classifier that predicts the computational compatibility of processors. LLVM based analyzer is used for feature extraction and tree-based method for classifier selection.

Resource Aware Load Balancer for Heterogeneous Cluster [50] RALB-HC is a supervised machine learning-based approach that distributes the workload in multi-node heterogeneous computing environments based on the computing capabilities of resources and needs of applications computing. The model considers the device suitability, the expected speedup, and the load balancing for job mapping in heterogeneous environments.

The RALB-HC technique works in 2 phases: 1) Mapping of jobs is based on available resources. 2) Load balancing for a higher resource utilization ratio. It also automates the decision about jobs for a specific computing device. Synthetic and Google-like workloads are produced using AMD, Polybench benchmark, etc. for testing the performance. RALB-HC reduces the execution time, increases the utilization of resources, and improves the throughput.

Troodon [51] schedules the given task in a heterogeneous system in a load-balanced manner considering job requirements, device suitability, and also performance predicted on a processor.

CPU suitable and GPU suitable jobs are combined in a pool of jobs based on the suitability of the device and sorted on the base of predicted speedup. Load balancing mechanism is gained on the basis of job processing requirements and device computation capabilities. Lower execution time maximum throughput, higher device utilization is achieved through device suitability. and mapping of jobs in a load-balancing manner. Users allocate OpenCL programs, and the computational assessment module checks the computational requirement through computational complexity and data size. The Kernel code features extractor extracts the Code features from the OpenCL job and provides a device suitability classifier to classify and label according to device suitability. The OpenCL programs along with the code feature extracted and input data size are provided to the speedup predictor component to predict speedup concerning other devices. The application then sorts the CPU-GPU job pool on the basis of device suitability where CPU suitable jobs pool is arranged in descending order on the basis of speedup and GPU suitable jobs pool is arranged in ascending order. After sorting, both job pools are combined for scheduling. E-OSched maps the jobs to CPU-GPU jobs. The top jobs from the job pool are mapped to CPU while jobs at the bottom are mapped to GPU.

TABLE 3.1: Critical Analysis of Literature Review.

Ref	**Benchmark**	**Feature**	**Result**
[7]	PARSEC, PARBOIL	Hardware Feature	increase is about 2.5time's faster execution time on the GPU, according to their comparisons between the GTX 280 and i7-960
[41]	MD,FFT,Search Reduction	Hardware Feature	MD runs fast on APU, FFT runs fast 5870 and 5450 reduction ran fastest on the APU
[42]	Multi-Model mean (MMM) Algorithm	Nill	CPU 6X seedups have been reached compared to on a low-power integrated GPU platform
[43]	HiBench	Hardware and Software Feature	GPU got a significant speedup over the CPU

[44]	Micro-Benchmarks	Hardware Feature	For small amount of data GPU performance is perfect then GPU but for significant amount GPU perform good compare to CPU
[46]	Matrix Multiplication	Software Feature	For small amount CPU performance is better but when input size increase GPU perform then GPU but for significant amount GPU perform better
[45]	BLAS	Hardware Feature	Performance of i7-7700 is beter than Xeon E5-2620, GEMM, GEMV and TSRV execution is fast on GPUs

In short, speedup on CPU-GPU seems to be achievable from these previous works. However, since the program is well adapted to parallelism, it is not ensured that the GPU can run faster. It seems to rely a lot on the particular program and what it is constrained by ? whether the GPU or the CPU runs the application faster. Applications that are highly computation intensive generally prefer CPUs, as they require higher single thread speeds that are supported by CPUs. Bandwidth-bounded programs, on the other hand, typically tend to support GPUs. Note, though, that this is not always accurate, many software features can affect the CPU-GPU performance. For instance, many researchers use machine learning to predict [49] [50] [51] a suitable architecture (CPU or GPU) for a particular application by extracting various software features (more than twenty software features are used in these studies, like the number of loops, functions and integer value etc). However, in contrast to these studies, we use a different approach, we measure the performance of an application on both CPU and GPU and then investigate the reason behind the better performance for a specific architecture. In this regard, we analyze OpenCL code and find out software features that affect the performance of an application. According to our findings, loop unrolling and data dimension are two important software features that affect the performance of an application. We believe the study of these two software features is our unique contribution since the aforementioned studies of other researchers use a different set of software features. Furthermore, we study the performance of various applications which belong to different domains such as convolution, linear algebra, data mining, and stencils. According to the best of our knowledge, there is no existing work that performed such detailed analysis for all these applications.

Chapter 4

Research Methodology

In this chapter, the selection of applications, implementation and how the analysis was done is presented.The program selection is important when analyzing CPU-GPU performance. If the same type of program runs on different architectures and is only suitable for one of them, then it is impossible to select best device for application that why different type of OpenCL applications are selected. There are two programs in an OpenCL application. The host program is the first, and the kernel program is the second.

If the program is operating on a processor, the host maps to the program's main control center, map the kernel program to reaming core and GPU case map the kernel program to the CPU and kernel program is mapped on GPU by the host program. Until running an OpenCL application, it's necessary to understand the architecture.

In this work, Intel's sixth-generation (Haswell) i7 processor with NVIDIA Geforce GT 740 discrete GPU is used. After selecting the application and underline architecture each application is execute on CPU and GPU with different input sizes using Linux operating system. I compare CPU-GPU performance based on the result after running all programs on CPU-GPU to get a better idea of which applications are suitable for CPU and GPU. The schematic of the proposed method is shown in Fig 4.1.

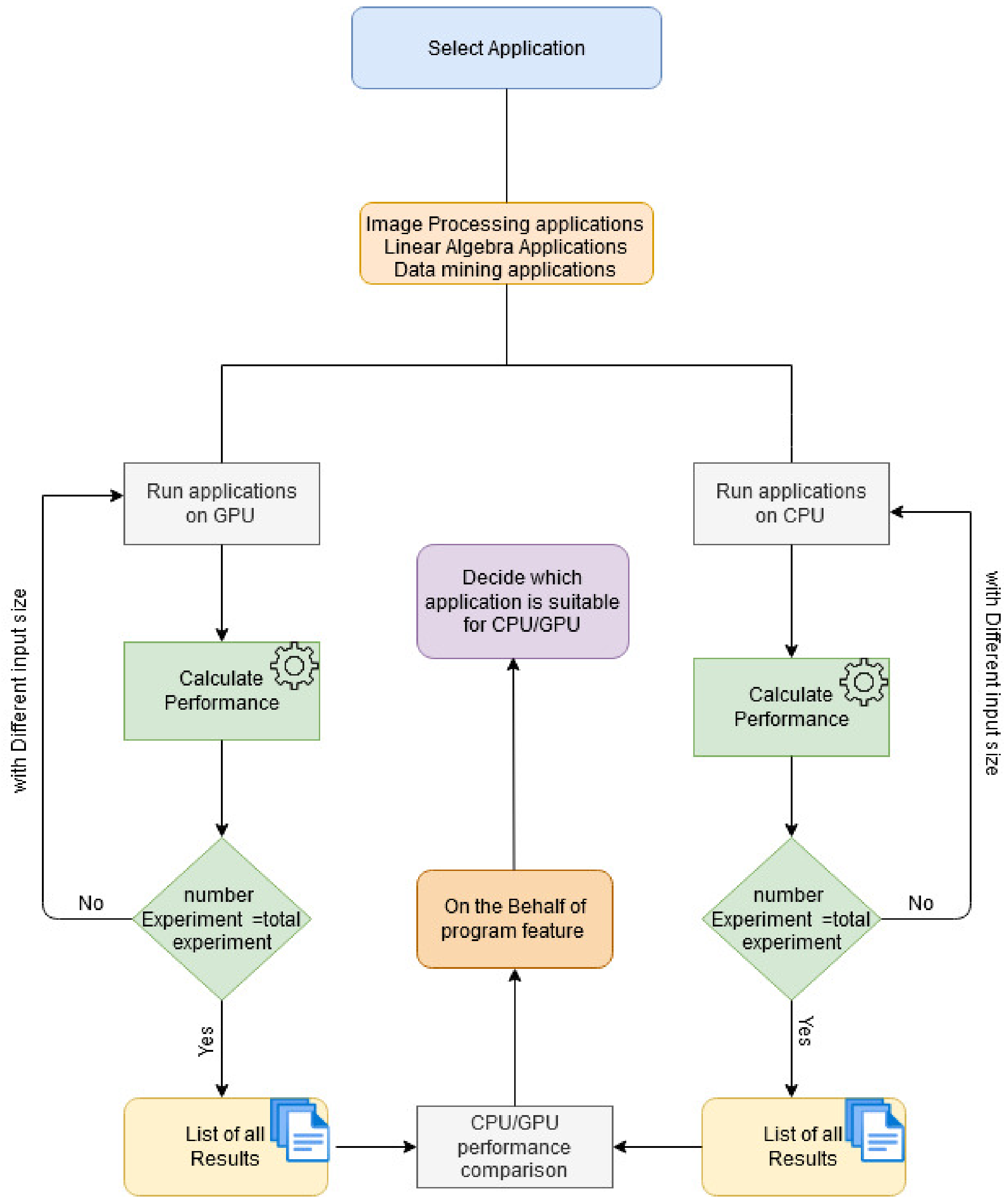

Select Application
Image Processing applications
Linear Algebra Applications
Data mining applications
Run applications on GPU
Run applications on CPU
Decide which application is suitable for CPU/GPU
Calculate Performance
Calculate Performance
with Different input size
with Different input size
number Experiment =total experiment
number Experiment =total experiment
No
No
On the Behalf of program feature
Yes
Yes
List of all Results
CPU/GPU performance comparison
List of all Results

FIGURE 4.1: Methodology Diagram

4.1 Application Selection

To analyze the CPU-GPU performance, the application selection is very important. If the same type of application executes on different architecture and this is only suitable for a single architecture then it is very difficult to decide which architecture is suitable,or if one algorithm that was perfectly fit to the GPU was the one we thought would have the highest chance of running on the GPU faster. This was based on the following considerations and the architectural limitations of

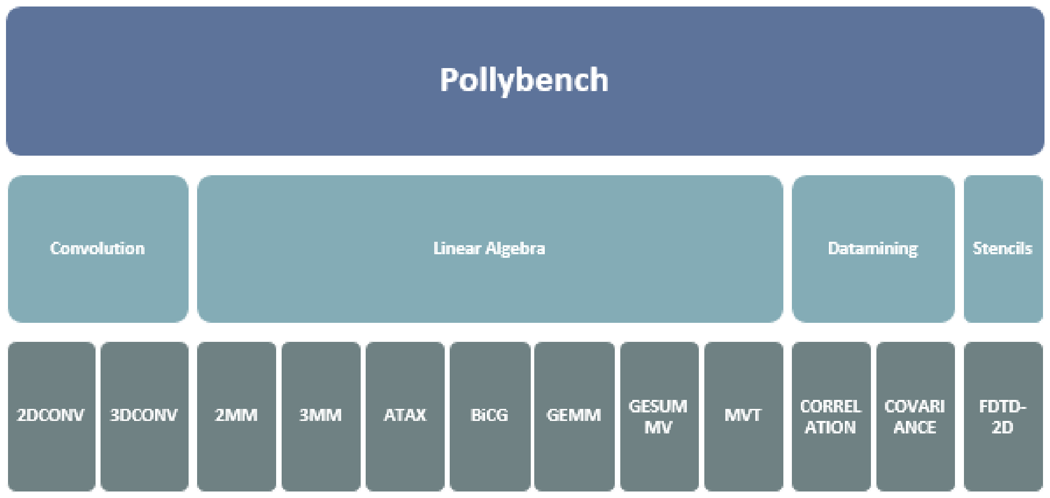


FIGURE 4.2: Classfication of Polybench Application

previous research. The CPU is typically preferred by computer-bound algorithms since they have a much higher clock frequency. Bandwidth-Bounded typically benefits the GPU because the GPU has at least as fast data access as the CPU much of the time (if not faster).
The architecture with the highest cache supports cache-bounded algorithms. That, is why in this work, the different types of applications have been chosen to compare the CPU GPU and performance like convolution, Linear Algebra, and Data mining those widely use applications suitable to find out the performance of two different architecture.in section 2.14 briefly discuss the applications.

4.2 Underline Architecture

It is important to know about the nature of architecture before executing OpenCL application. In this work, Intel's sixth-generation (Haswell) i7 processor. It has four cores, in which two are physical core and two are logical core. Each core can execute two threads simultaneously and it has 3.40 GHz base clock frequency with NVIDIA Geforce GT 740 discrete GPU it has 384 cores and 28.8 GB/s memory bandwidth is used. NVIDIA Geforce GT 740 is a Discrete GPU that type of GPU has physical memory as a separate computing unit from CPU.

TABLE 4.1: Central processing Unit Details

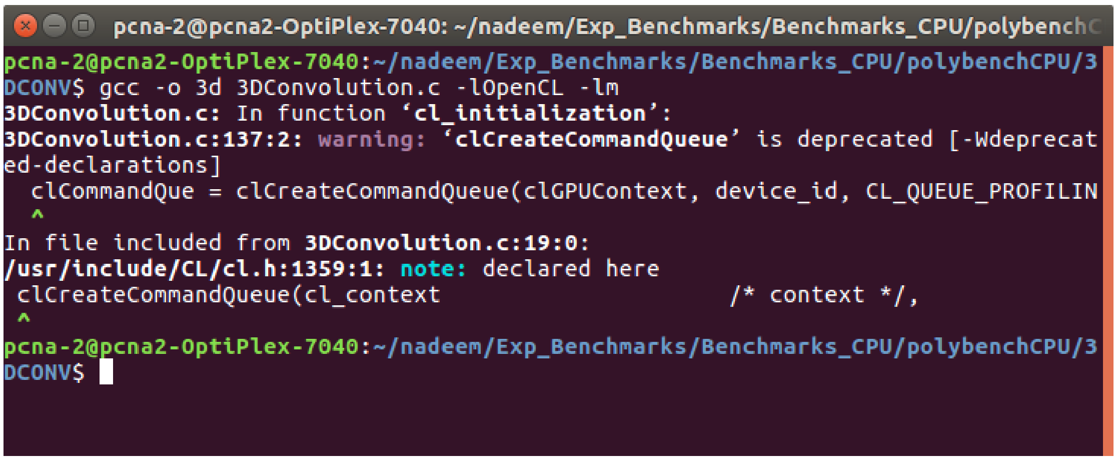
Vendor	intel
Processor	intel i7-6700
Clock Speed	3.40 GHz
Memory Bandwidth	34.1 GB/s
No of Core	4
Threads	8
Cache	8 MB

TABLE 4.2: Graphical Processing Unit Details

Vendor	Nvidia
Graphic Card version	GT 740
Clock Speed	1.8 GHz
Memory Bandwidth	28.8 GB/s
No of Core	384

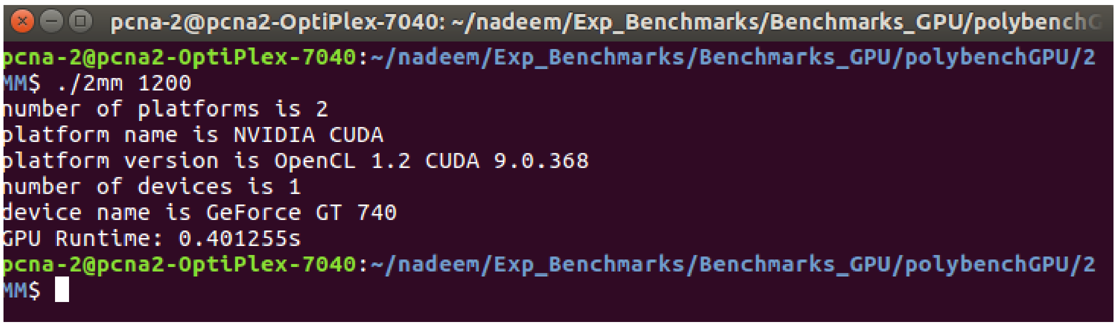
4.3 Job Execution

Although the benchmark programs were written in OpenCL by the authors from scratch, there is no need to change the application. I run each application on CPU as well as GPU using Linux operating system. Every OpenCL application consists of two programs one is host program and second one is kernel program.
For example, if the program is running on a processor, the host maps to main control center of the program, maps to core that starts and merges the threads, and maps to the system CPU if it is running on a GPU. In this work the OpenCL applications are execute on CPU and GPU also. In the CPU case host program is map to a core core and host is map on remaing core of CPU.
In GPU case the host program is map on CPU and the kernel program to GPU. For job execution, first, make the executable file of each program Figure 4.3 shows Linux command by making an executable file of each program after that execute each program on CPU and GPU. Figure 4.4 shows the Linux command to execute application. Every program is executed severally with the different input sizes.



```
pcna-2@pcna2-OptiPlex-7040: ~/nadeem/Exp_Benchmarks/Benchmarks_CPU/polybenchC
pcna-2@pcna2-OptiPlex-7040:~/nadeem/Exp_Benchmarks/Benchmarks_CPU/polybenchCPU/3
DCONV$ gcc -o 3d 3DConvolution.c -lOpenCL -lm
3DConvolution.c: In function ‘cl_initialization’:
3DConvolution.c:137:2: warning: ‘clCreateCommandQueue’ is deprecated [-Wdeprecat
ed-declarations]
  clCommandQue = clCreateCommandQueue(clGPUContext, device_id, CL_QUEUE_PROFILIN
  ^
In file included from 3DConvolution.c:19:0:
/usr/include/CL/cl.h:1359:1: note: declared here
 clCreateCommandQueue(cl_context                     /* context */,
 ^
pcna-2@pcna2-OptiPlex-7040:~/nadeem/Exp_Benchmarks/Benchmarks_CPU/polybenchCPU/3
DCONV$ 
```

FIGURE 4.3: Linux command to make an executable file



```
pcna-2@pcna2-OptiPlex-7040: ~/nadeem/Exp_Benchmarks/Benchmarks_GPU/polybenchC
pcna-2@pcna2-OptiPlex-7040:~/nadeem/Exp_Benchmarks/Benchmarks_GPU/polybenchGPU/2
MM$ ./2mm 1200
number of platforms is 2
platform name is NVIDIA CUDA
platform version is OpenCL 1.2 CUDA 9.0.368
number of devices is 1
device name is GeForce GT 740
GPU Runtime: 0.401255s
pcna-2@pcna2-OptiPlex-7040:~/nadeem/Exp_Benchmarks/Benchmarks_GPU/polybenchGPU/2
MM$ 
```

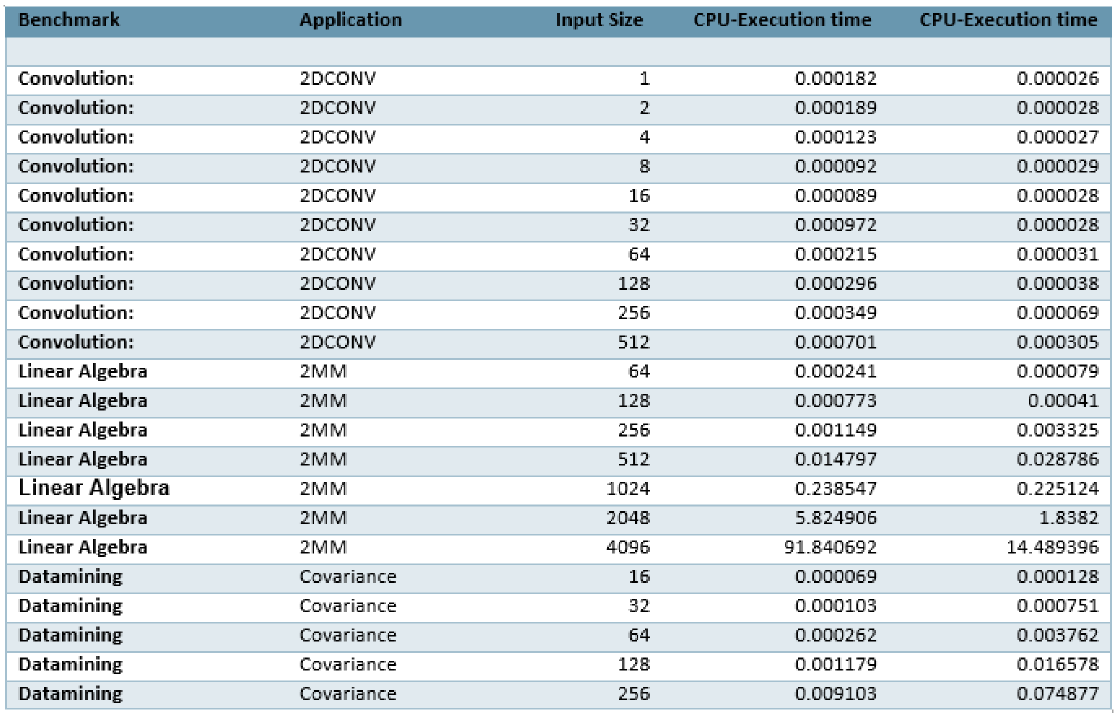
FIGURE 4.4: Linux command to execute the application.

4.4 Result Compute

After executing each application on different architectures result are computed. Result file consists of application name mean the type of application. It belongs to linear algebra, data mining or convolution, etc. Input size represent the input size of the application in my experiment input size is the multiple of two, CPU execution time represent, how much time CPU takes to execute the application with given input size similarly GPU execution time means how much time GPU takes to execute the application with given input size. Figure 4.5 represents the sample of experiment result and complete data is available in appendices.

4.5 Performance Analysis

After executing all applications on CPU-GPU and computing the result, I compare the CPU-GPU performance based on the result.



Benchmark	Application	Input Size	CPU-Execution time	CPU-Execution time
Convolution:	2DCONV	1	0.000182	0.000026
Convolution:	2DCONV	2	0.000189	0.000028
Convolution:	2DCONV	4	0.000123	0.000027
Convolution:	2DCONV	8	0.000092	0.000029
Convolution:	2DCONV	16	0.000089	0.000028
Convolution:	2DCONV	32	0.000972	0.000028
Convolution:	2DCONV	64	0.000215	0.000031
Convolution:	2DCONV	128	0.000296	0.000038
Convolution:	2DCONV	256	0.000349	0.000069
Convolution:	2DCONV	512	0.000701	0.000305
Linear Algebra	2MM	64	0.000241	0.000079
Linear Algebra	2MM	128	0.000773	0.00041
Linear Algebra	2MM	256	0.001149	0.003325
Linear Algebra	2MM	512	0.014797	0.028786
Linear Algebra	2MM	1024	0.238547	0.225124
Linear Algebra	2MM	2048	5.824906	1.8382
Linear Algebra	2MM	4096	91.840692	14.489396
Datamining	Covariance	16	0.000069	0.000128
Datamining	Covariance	32	0.000103	0.000751
Datamining	Covariance	64	0.000262	0.003762
Datamining	Covariance	128	0.001179	0.016578
Datamining	Covariance	256	0.009103	0.074877

Figure 4.5: Experiment Result Sample

That why the research is consists of two parts: firstly, testing which implementation is faster and, secondly, why it is faster? The first component is completely focused on the execution time of the application i.e that architecture takes how much time to execute the application?

The second part of this work is to analyze the result that is in which case the GPU implementation has a faster execution time than CPU exaction time and if what was the reason CPU execution is fast then GPU what reason behind it and what type of software feature effect the performance of CPU and GPU. The next chapter explains the variation in CPU-GPU execution time and explains the software feature which is affecting the performance of CPU-GPU architecture.

Chapter 5

Result and Analysis

This chapter will demonstrate and analyze the performance of CPU Implementation and GPU implementation of Polybench suit. The input data size for each program is multiple of two to evaluate the performance of CPU-GPU implementation. This thesis starts by first understanding kernels that run fast on a single architecture. For example, 2DCONV, 3DCONV and FDTD-2D applications are fast on GPU for every input size and the Atax application is fast on CPU for every input size. The performance effects are evaluated and the features that contribute to the performance of each kernel are established. The performance is measure as CPU execution time and GPU execution against different input sizes of each application. In this work analysis is done on the behalf of software feature first one is loop unrolling and Dimension of data those impact the performance of the application on different architectures. In this chapter, we explain the impact of these features on CPU and GPU performance and also see the kernel and host program of each application.

5.1 3DCONV

3D Convolution kernel program is executed on CPU and GPU with the different input size. But all the time it executes fast on GPU. For a small input size, there

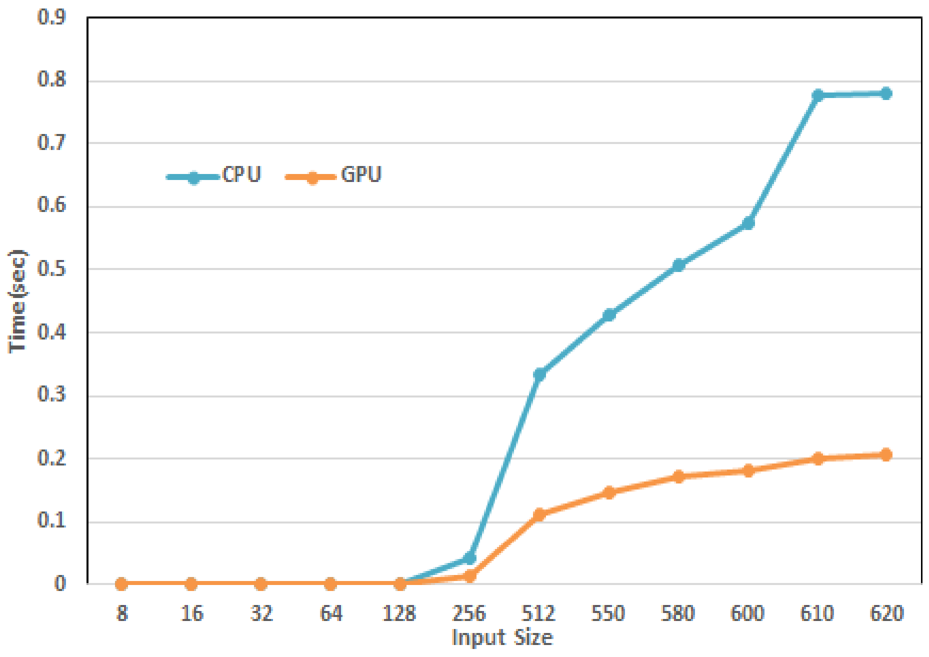


FIGURE 5.1: CPU-GPU execution time for 3DCONV application.

is no big difference between CPU and GPU execution time but when input size increases the performance of GPU increase as compare to CPU performance. It is clearly shown in 5.1 graph when the input size is 620 GPU takes 0.2 sec to execute the 3DCOV application but on the other hand, CPU takes 0.8 sec for execution.Figure 5.1 shows CPU-GPU execution time for the 3DCONV application. To examine why the program was always running faster on the GPU as compared to the CPU, analyze the host program and kernel program of 3DCONV. The analysis demonstrates that the GPU execution was significantly fast by the maximum number of loop unrolling and two Dimension of data in 3DCONV application. All nested loops are unrolled in 3DCONV kernel program.

5.2 2DCONV

2D Convolution kernel program is executed on CPU and GPU with the different input size. but all the time it executes fast on GPU.for a small input size there is no big difference between CPU and GPU execution time but when input size increases the performance of GPU increase as compare to CPU performance. It is clearly shown in 5.2 graph when the input size is 15500 GPU takes 0.168965 sec to

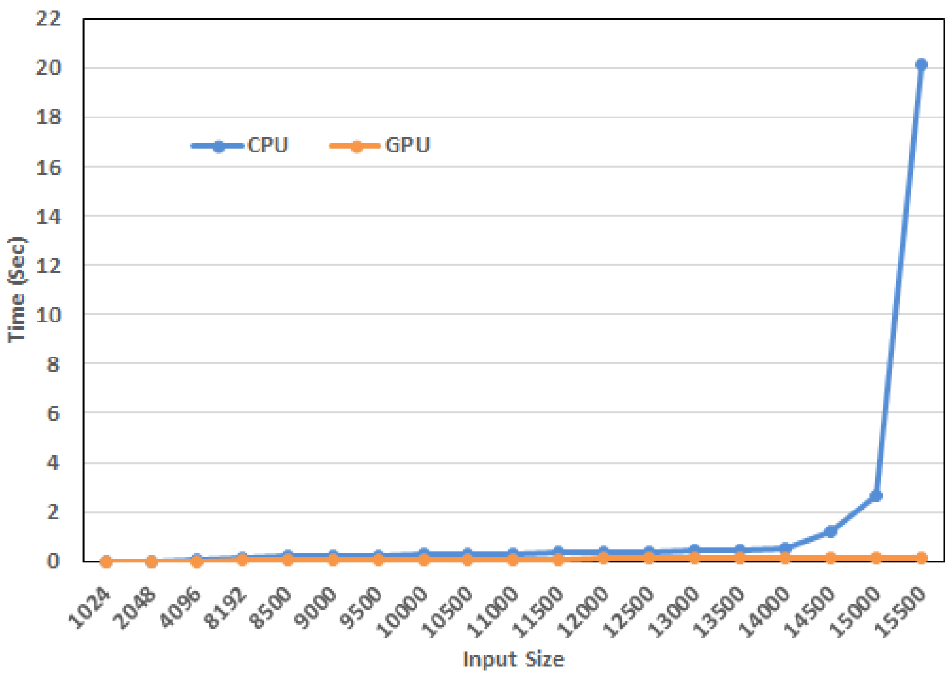


FIGURE 5.2: CPU-GPU execution time for 2CONV application.

execute it but on other hand, CPU takes 20 sec for execution. in 2DCONV input, size is multiple of two and start graph from 1024 for input size 1-512 execution time for CPU and GPU is less than one sec so it is not visible in a graph that why we start it from input size 1024. Figure 5.2 shows the result for 3DCONV implementation. To examine why the program was always running faster on the GPU as compared to the CPU, analyze the host program and kernel program of 2DCONV. The analysis demonstrates that the GPU execution was significantly fast by the maximum number of loop unrolling and two Dimension of data in 2DCONV application. All nested loops are unrolled in 2DCONV kernel program.

5.3 FDTD-2D

FDTD-2D kernel program is executed on CPU and GPU with the different input size. But all the time it executes fast on GPU.for a small input size there is no big difference between CPU and GPU execution time but when input size increases the performance of GPU increase as compare to CPU performance. It is clearly shown in 5.3 graph when the input size is 12000 GPU takes 29.2373 sec to execute FDTD-2D but on other hand, CPU takes 169 sec for execution. In

FDTD-2D input, size is multiple of two and start graph from 256 for input size 1-128 execution time for CPU and GPU is less than one sec so it is not visible in a graph that why we start it from input size 1024.Figure 5.3 shows the result for FDTD-2D implementation. To examine why the program was always running

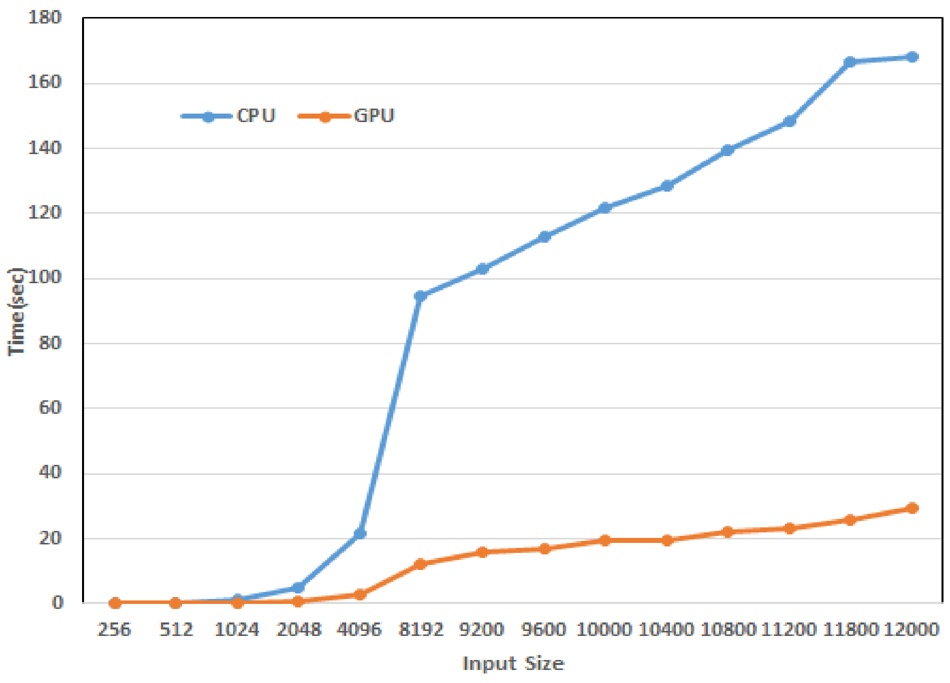


FIGURE 5.3: CPU-GPU execution time for FDTD-2D application.

faster on the GPU as compared to the CPU, analyze the host program and kernel program of FDTD-2D. The analysis demonstrates that the GPU execution was significantly fast by the maximum number of loop unrolling and two Dimension of data in FDTD-2D application. All nested loops are unrolled in FDTD-2D kernel program.

5.4 ATAX

ATAX application is executed on CPU and GPU with the different input size. In the previous OpenCL kernels like 2DCONV, 3DCONV and FDTD-2D those kernel execution remains fast all the time on GPU but ATAX OpenCL kernel execution is opposite to previous execution it remains fast all the time on CPU. For a small input size, there is no big difference between CPU and GPU execution time and when input size increases the performance of CPU increase as compare

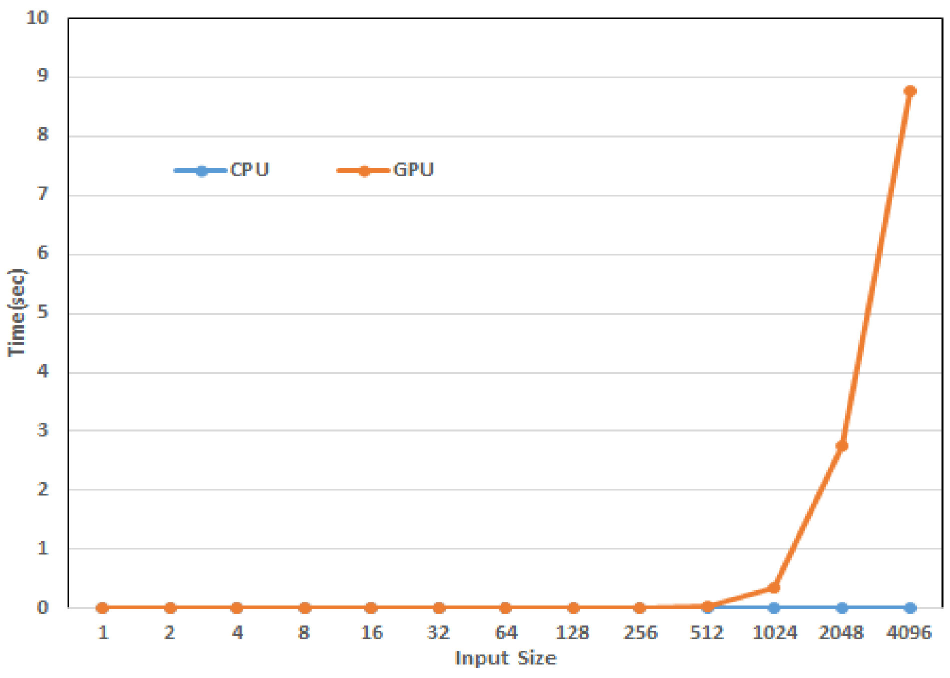


FIGURE 5.4: CPU-GPU execution time for ATAX application.

to GPU performance. It is clearly shown in a graph when the input size is 4096 CPU takes 0.018337 sec to execute it but on other hand, GPU takes 8.756636 sec for execution. Figure 5.4 show the result for ATAX implementation.

To examine why the program was always running faster on the GPU as compared to the CPU, analyze the host program and kernel program of ATAX. The analysis demonstrates that the CPU execution was significantly fast because there is no loop unrolled in its kernel program and one dimension data is used in the ATAX application.

In previous there are three OpenCL application 2DCOV,3DCONV and FDTD-2D always execute fast on GPU.when analysis the OpenCL program of those application demonstrates that the GPU remains all the time fast for these application first all nested loops in the host program are unrolled in the kernel program secondly there two-dimension data is used in these OpenCL application which executes fast on GPU for all input size on other hand ATAX execute all the time on CPU than GPU when analysis the ATAX application indicate the nested loop in ATAX host program didn't unroll in ATAX kernel program and second in ATAX application dimension of data is one. Those are the feature that effecting the ATAX performance on CPU and GPU. It mean less of loop unrolling and one dimension in data is good for CPU.

5.5 2MM

2MM Linear algebra application is executed on CPU and GPU with the different input size. In the 2MM experiment result there are two types of variations that have been seen 1) for small and large input size GPU performance is better than CPU 2) For some middle input size CPU performance is better than GPU. Figure 5.5 shows the 2MM application execution time of CPU-GPU for 1-128 input size, figure 5.6 shows the 2MM application execution time of CPU-GPU for 243-1024 input size, figure 5.7 the shows 2MM application execution time of CPU-GPU for 1024-4096 input size. To examine why the program was always running faster on

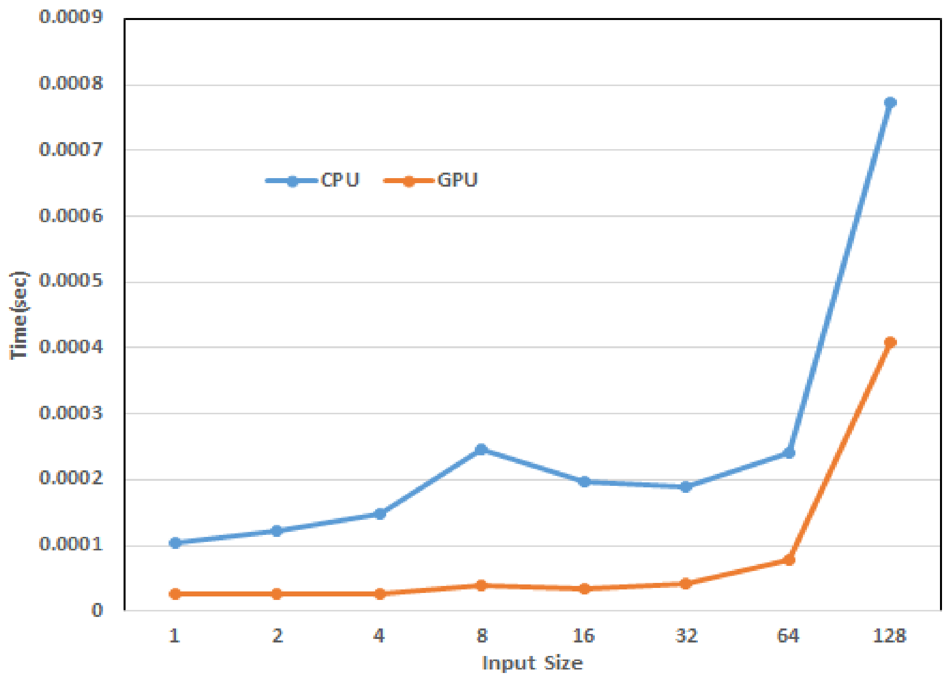


FIGURE 5.5: 2MM application execution time of CPU-GPU for 1-128 input size

the GPU for small and large input sizes and why the program was running faster on CPU for middle input size. To find out the reason for such variation analysis of the host and kernel program of 2MM is done. There are two nested loops in the 2MM host program and two dimensions in the data. The analysis demonstrates that the GPU execution was significantly fast for small and large input size and CPU execution was fast for middle input size because there are two loops unrolled from each nested loop and data in portioned in two dimensions. Much as in the previous application result 2DCONV, 3DONV and FDTD-2D GPU remain

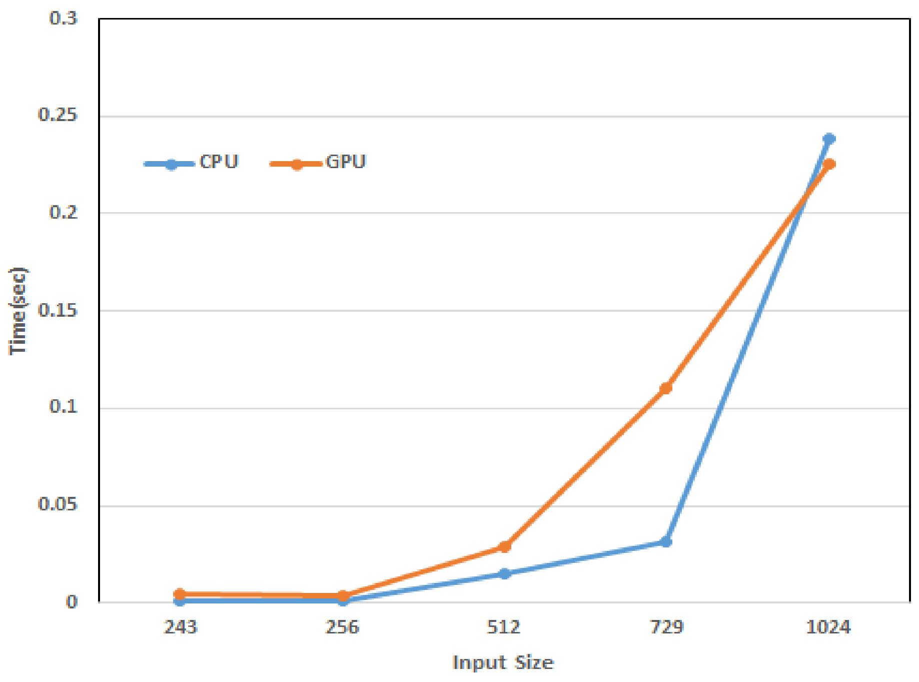


FIGURE 5.6: 2MM application execution time of CPU-GPU for 243-1024 input size

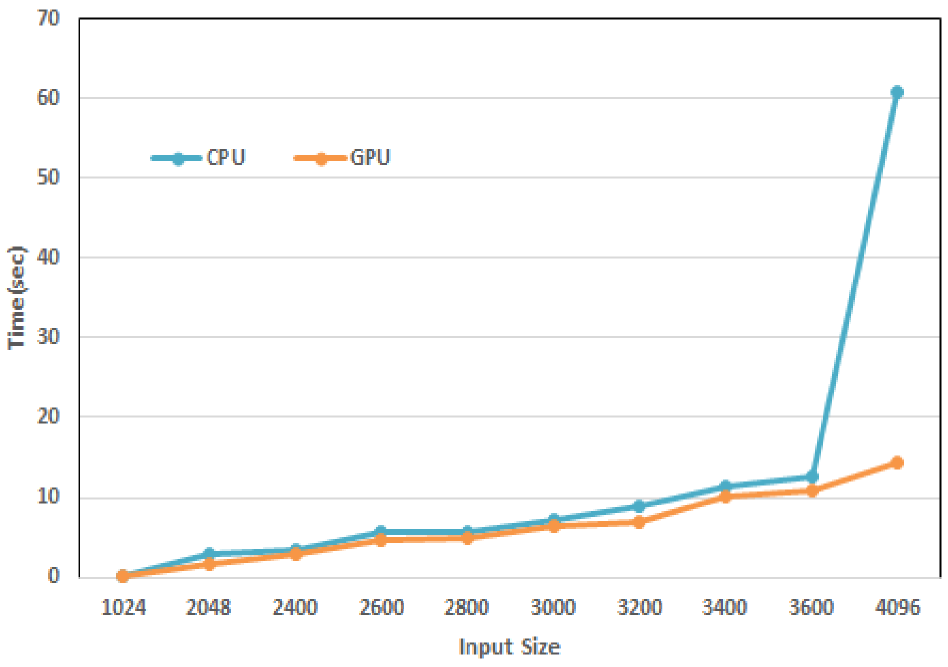


FIGURE 5.7: 2MM application execution time of CPU-GPU for 1024-4096 input size

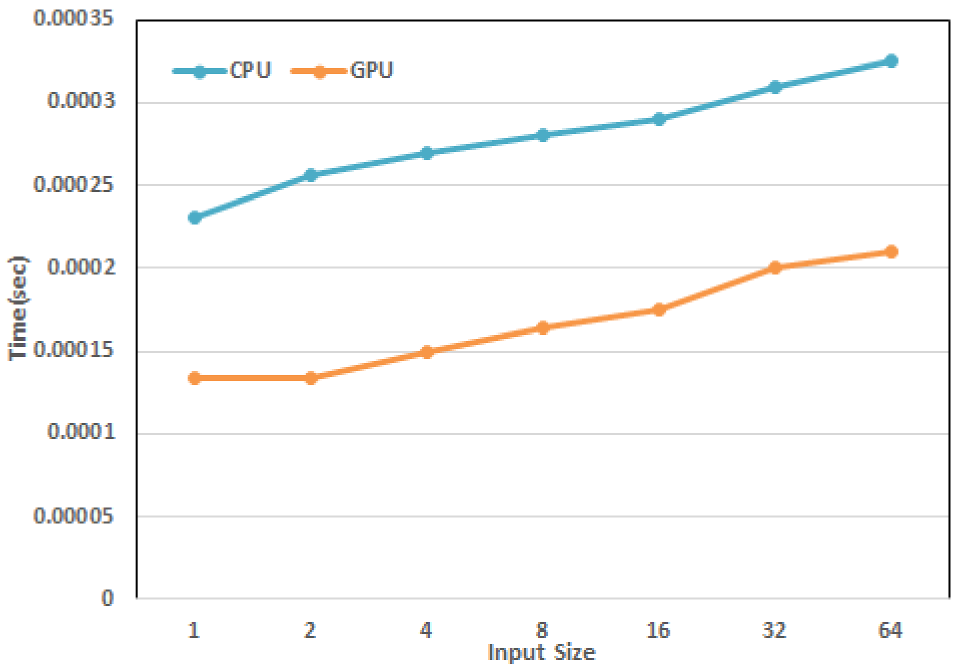


FIGURE 5.8: 3MM application execution time of CPU-GPU for 1-64 input size.

fast. all the time because all loops are unrolled in kernel function and data is portioned in two dimensions but in the 2MM application two-loop are unrolled but the innermost loop didn't unroll that why this study indicate GPU execution remain fast for small and large input size and CPU execution remain fast for some middle input size.

5.6 3MM

3MM Linear algebra application is executed on CPU and GPU with the different input size. In the 3MM experiment result there are two types of variations that have been seen 1) for small and large input size GPU performance is better than CPU 2) For some middle input size CPU performance is better than GPU. Figure 5.8 shows the 3MM application execution time of CPU-GPU for small input size, figure 5.9 shows the 3MM application execution time of CPU-GPU for middle input size, figure 5.10 the shows 3MM application execution time of CPU-GPU for large input size. To examine why the program was always running faster on the GPU for small and large input size and why the program was running faster on CPU for middle input size. To find out the reason for such variation

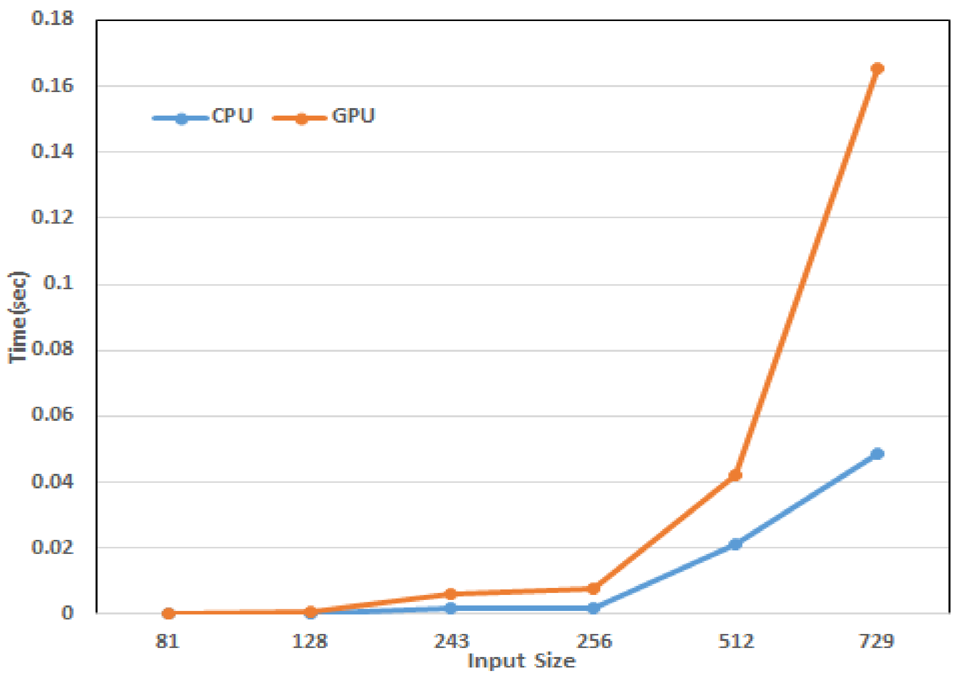


FIGURE 5.9: 3MM application execution time of CPU-GPU for 81-729 input size.

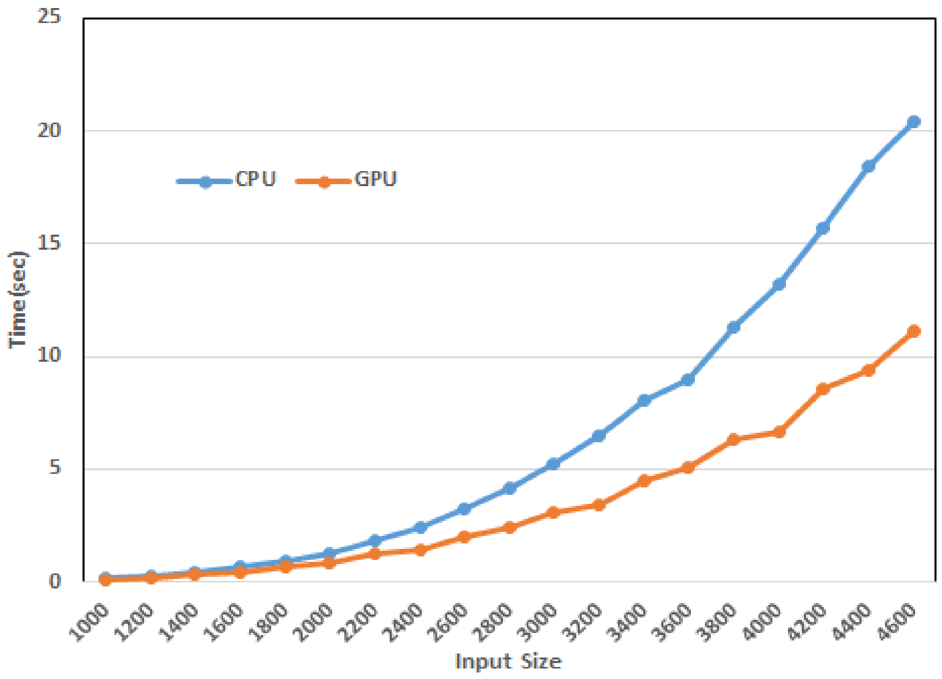


FIGURE 5.10: 3MM application execution time of CPU-GPU for 1000-4600 input size.

analysis of the host and kernel program of 3MM is done. There are three nested loops in the 3MM host program and two dimensions in the data. The analysis demonstrates that the GPU execution was significantly fast for small and large input size and CPU execution was fast for middle input size because there are two loops unrolled from each nested loop and data in portioned in two dimensions. Much as in the previous applications result from 2DCONV, 3DONV, and FDTD-2D GPU remain fast all the time because all loops are unrolled in kernel function and data is portioned in two dimensions but in the 3MM application two-loop are unrolled from each nested loop but the innermost loop of each nested loop didn't unroll that why this study indicate GPU execution remain fast for small and large input size and CPU execution remain fast for some middle input size.

5.7 GEMM

GEMM Linear algebra application is executed on CPU and GPU with the different input size. In the GEMM experiment result there are two types of variations that have been seen 1) for small and large input size GPU performance is better than CPU 2) For some middle input size CPU performance is better than GPU. Figure 5.11 shows the GEMM application execution time of CPU-GPU for small input size, figure 5.12 shows the GEMM application execution time of CPU-GPU for middle input size, figure 5.13 shows GEMM application execution time of CPU-GPU for large input size. To examine why the program was always running faster on the GPU for small and large input size and why the program was running faster on CPU for middle input size. To find out the reason for such variation analysis of the host and kernel program of GEMM is done. there are one nested loop in the GEMM host program and two dimensions in the data. The analysis demonstrates that the GPU execution was significantly fast for small and large input size and CPU execution was fast for middle input size because there are two loops unrolled from nested loop and data in portioned in two dimensions. Much as in the previous applications result from 2DCONV, 3DCONV and FDTD-2D. GPU remain fast all the time because all loops are unrolled in kernel function and data is portioned in two dimensions but in the GEMM application two-loop

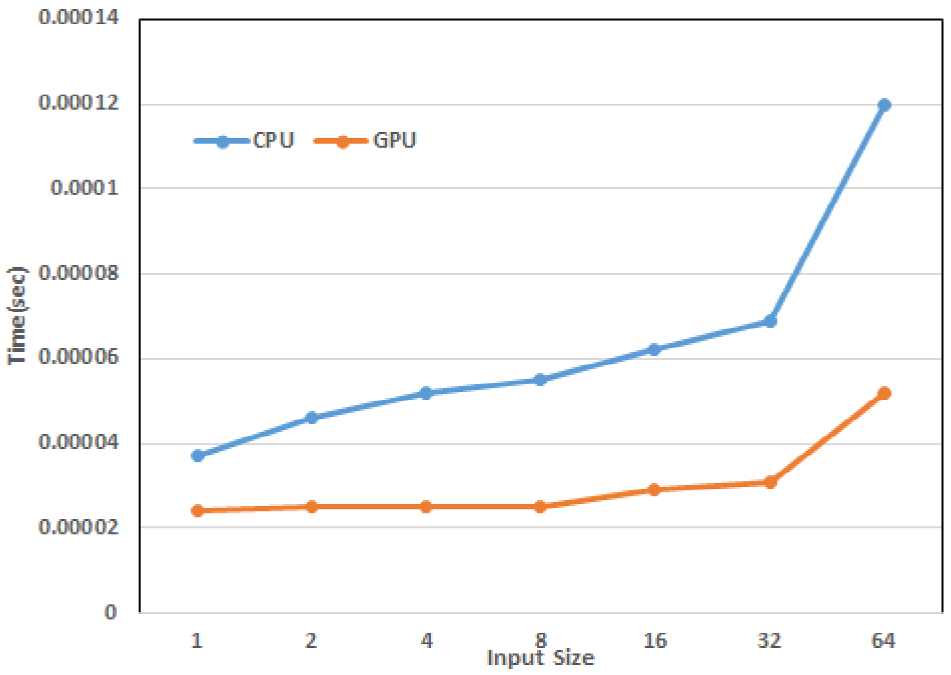


FIGURE 5.11: GEMM application execution time of CPU-GPU for 1-64 input size.

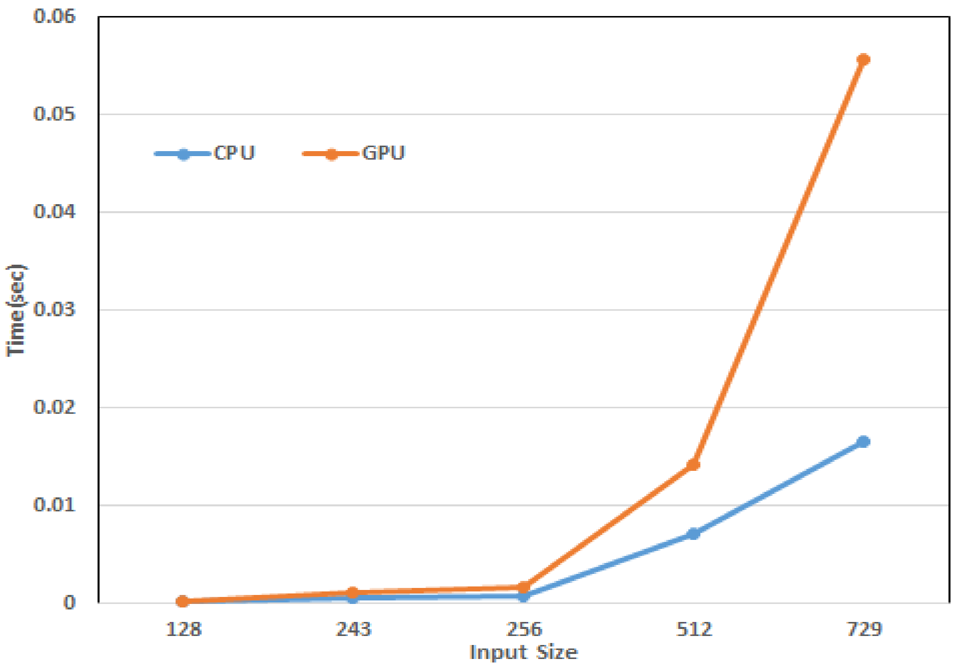


FIGURE 5.12: GEMM application execution time of CPU-GPU for 128-729 input size.

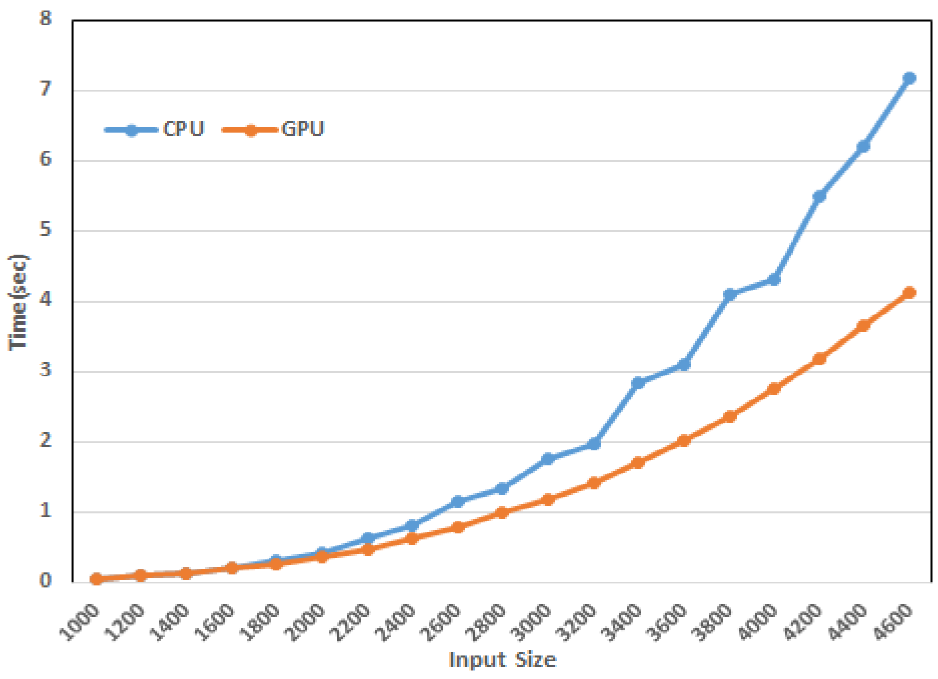


FIGURE 5.13: GEMM application execution time of CPU-GPU for 1000-4600 input size.

are unrolled from each nested loop but the innermost loop of each nested loop didn't unroll that why this study indicate GPU execution remain fast for small and large input size and CPU execution remain fast for some middle input size. After execution of GEMM, 3MM and 2MM the results of all these applications are the same for example GPU execution impressively fast for small and large input but as compared to CPU execution in same CPU execution remains fast for middle input size as compared to GPU. after analyzing that OpenCL applications host and kernel programs two things common in all application first two dimensions in data second in host program of each application there are nested loop in 2MM two nested loops, 3MM three nested loops and in GEMM one nested loop and two loops are unrolled in the kernel program of these applications.

5.8 MVT

MVT Linear algebra application is executed on CPU and GPU with the different input size. In the GEMM experiment result, there are two types of variations that have been seen 1) for small input size GPU performance is better than CPU 2) but as input size increase CPU perform better than GPU. Figure 5.14 shows the

execution time of CPU-GPU for MVT application for small input size figure 5.15 shows the execution time of CPU-GPU for MVT application for large input size.
To examine why the program was always running faster on the GPU for small

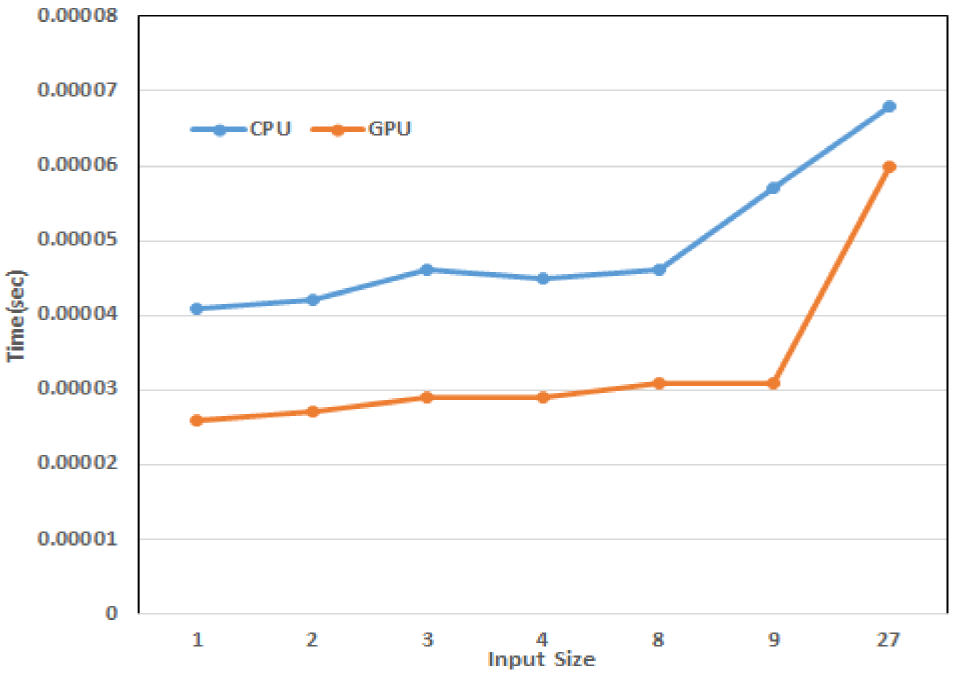


FIGURE 5.14: MVT application execution time of CPU-GPU for 1-27 input size.

input size and why the program was running faster on CPU for large input size. To find out the reason for such variation analysis of the host and kernel program of MVT is done. There are two nested loops in the MVT host program and one dimension in the data.
The analysis demonstrates that the GPU execution was significantly fast for small input size and CPU execution was fast for large input size because there is one loop unrolled from each nested loop and data is portioned in one dimension.
Much as in the previous application result from ATAX CPU remain fast all the time because there is no loop unrolled in its kernel and data is portioned in one dimension.
But in the MVT application one-loop is unrolled from each nested loop but the inner loop of each nested loop didn't unroll that why this study indicates GPU execution remains fast for small and but CPU execute MVT kernel program fastly than GPU when input size increase. Such variation has been seen when in put size.

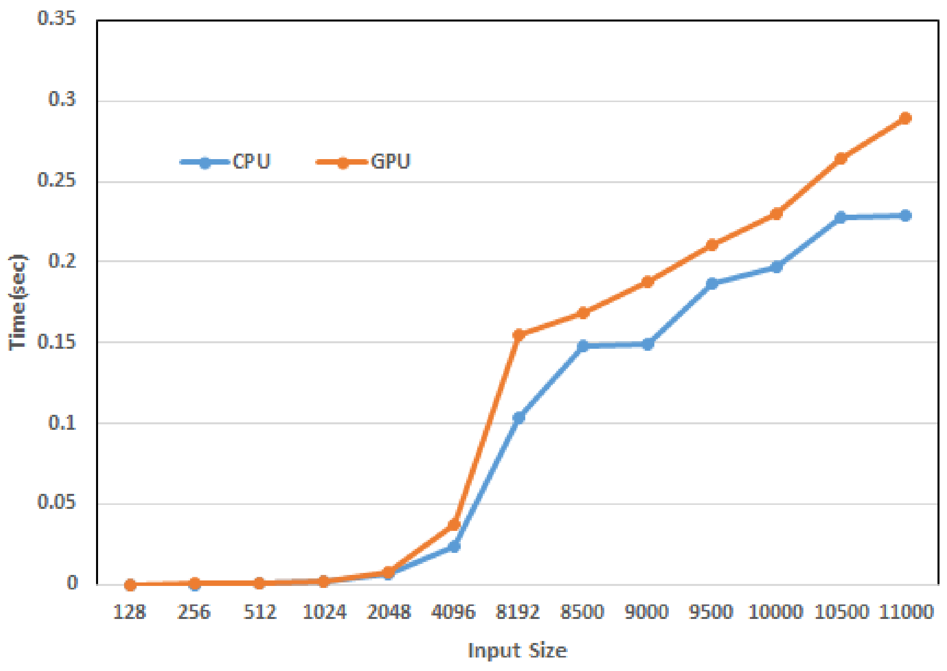


FIGURE 5.15: MVT application execution time of CPU-GPU for 128-11000 input size.

5.9 BICG

BICG Linear algebra application is executed on CPU and GPU with the different input size. In the BICG experiment result, there are two types of variations that have been seen 1) for small input size GPU performance is better than CPU 2) but as input size increase CPU perform better than GPU. Figure 5.16 shows the execution time of CPU-GPU for BICG application for small input size figure 5.17 shows the execution time of CPU-GPU for BICG application for large input size. To examine why the program was always running faster on the GPU for small input size and why the program was running faster on CPU for large input size. To find out the reason for such variation analysis of the host and kernel program of BICG is done.

There are one single loop and one nested loop in the BICG host program and one dimension in the data. The analysis demonstrates that the GPU execution was significantly fast for small input size and CPU execution was fast for large input size because there is one loop unrolled from each nested loop and data is portioned in one dimension. Much as in the previous application result from ATAX CPU remain fast all the time because there is no loop unrolled in its kernel and data

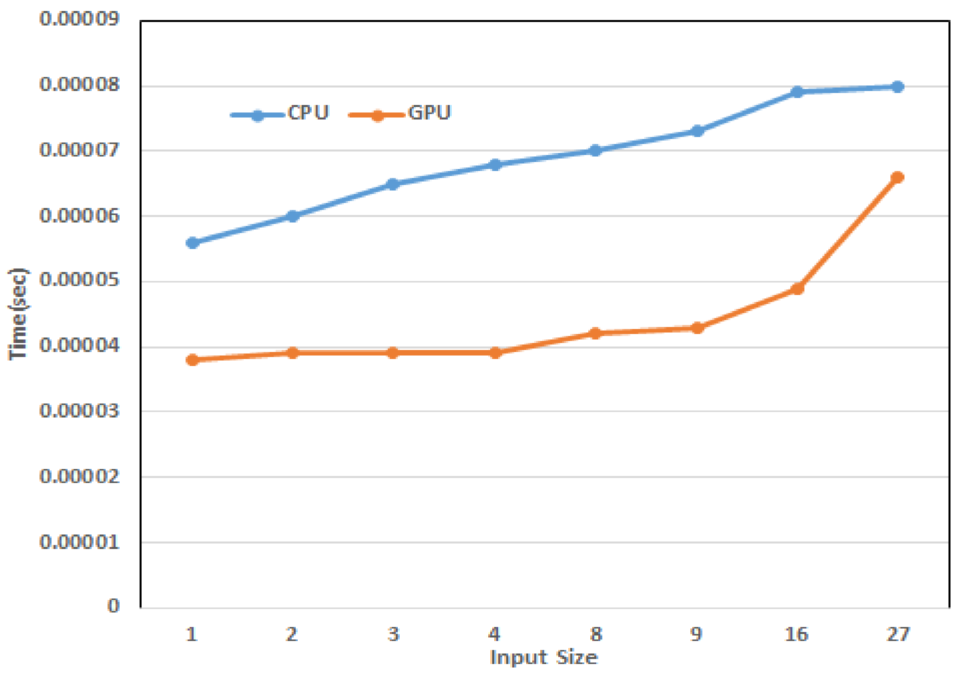


FIGURE 5.16: BICG application execution time of CPU-GPU for 1-27 input size.

is portioned in one dimension. But in the BICG application one-loop is unrolled from each nested loop but the inner loop of each nested loop didn't unroll that why this study indicates GPU execution remains fast for small and but CPU execute BICG kernel program fastly than GPU when input size increase.

5.10 GESUMMV

GESMMV Linear algebra application is executed on CPU and GPU with the different input size. In the BICG experiment result, there are two types of variations that have been seen 1) for small input size GPU performance is better than CPU 2) but as input size increase CPU perform better than GPU. Figure 5.18 shows the execution time of CPU-GPU for GESUMMV application for small input size 5.19 shows the execution time of CPU-GPU for GESUMMV application for large input size. To examine why the program was always running faster on the GPU for small input size and why the program was running faster on CPU for large input size. To find out the reason for such variation analysis of the host and kernel program of GESUMMV is done. There is one nested loop in the GESUMMV host

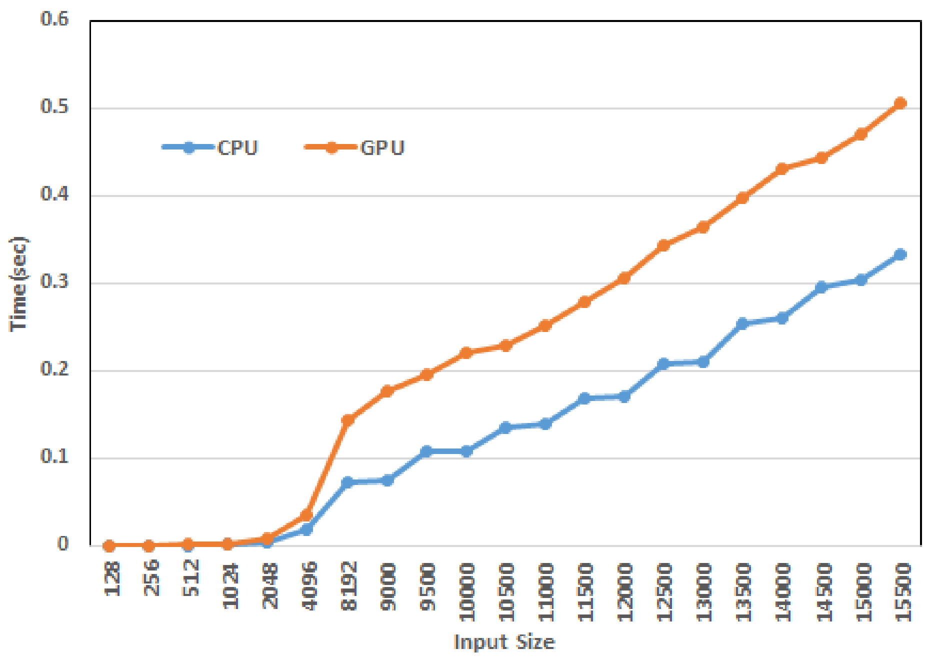


FIGURE 5.17: BICG application execution time of CPU-GPU for 128-15500 input size.

program and one dimension in the data. The analysis demonstrates that the GPU execution was significantly fast for small input size and CPU execution was fast for large input size because there is one loop unrolled from nested loop and data is portioned in one dimension.

Much as in the previous application result from ATAX CPU remain fast all the time because all loops are unrolled in kernel function and data is portioned in one dimension but in the GESUMMV application one loop is unrolled from nested loop but the inner loop of the nested loop didn't unroll that why this study indicate GPU execution remain fast for small input size and CPU execution remain fast for large input size.

After execution of MVT,and BICG the results of all these applications are the same for example GPU execution impressively fast for small input but as compared to CPU execution in the same way CPU execution remains fast for large input size as compared to GPU. after analyzing that OpenCL applications host and kernel programs two things common in all application first one dimension in data second in host program of each application there are nested loop in MVT two nested loops, BICG one nested loops and in the kernel program of these application one loop unrolled from each nested loop.

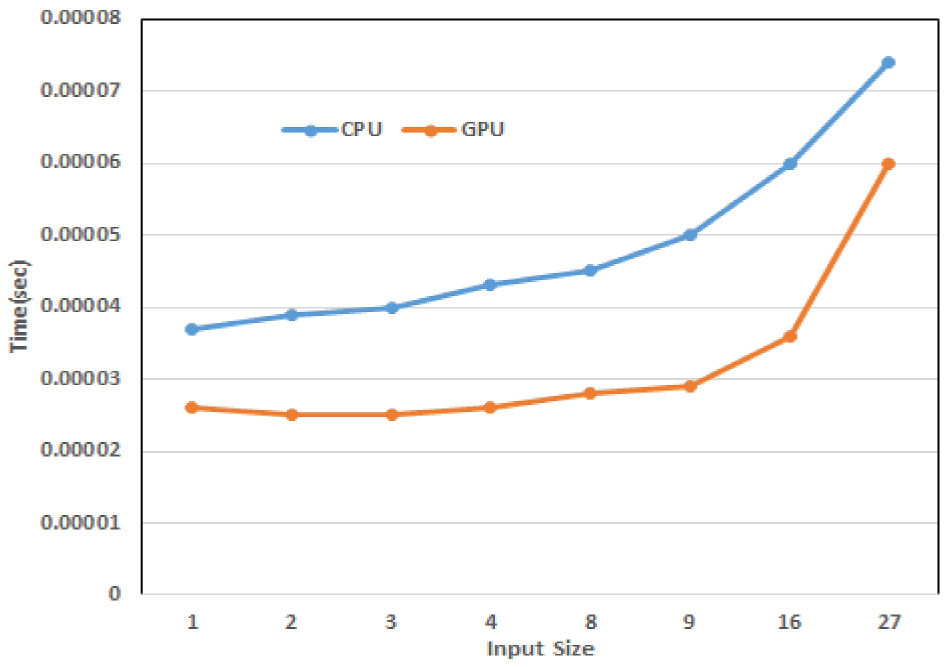


FIGURE 5.18: GESUMMV application execution time of CPU-GPU for 1-27 input size.

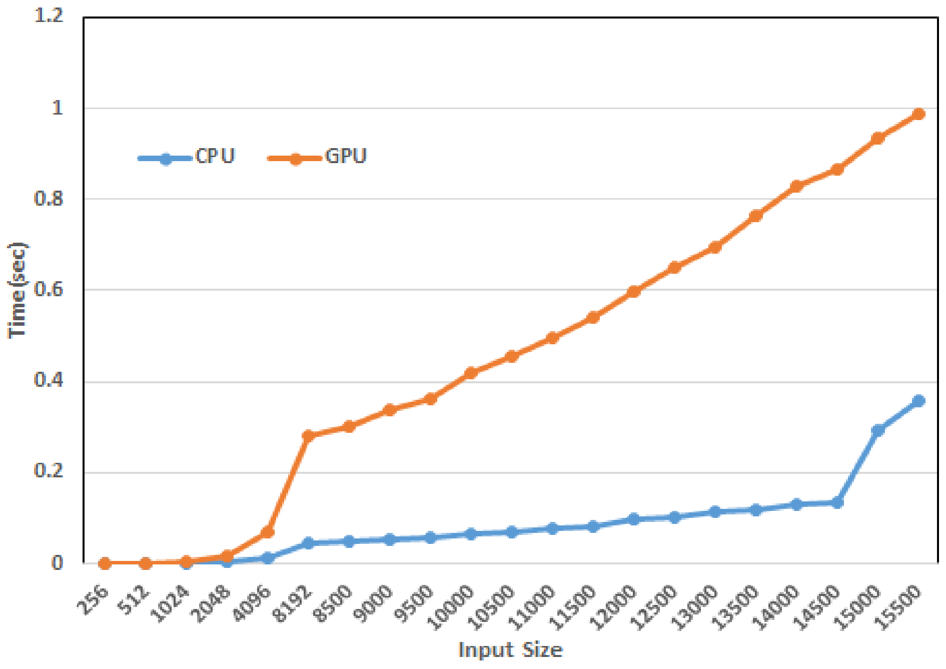


FIGURE 5.19: GESUMMV application execution time of CPU-GPU for 256-15500 input size.

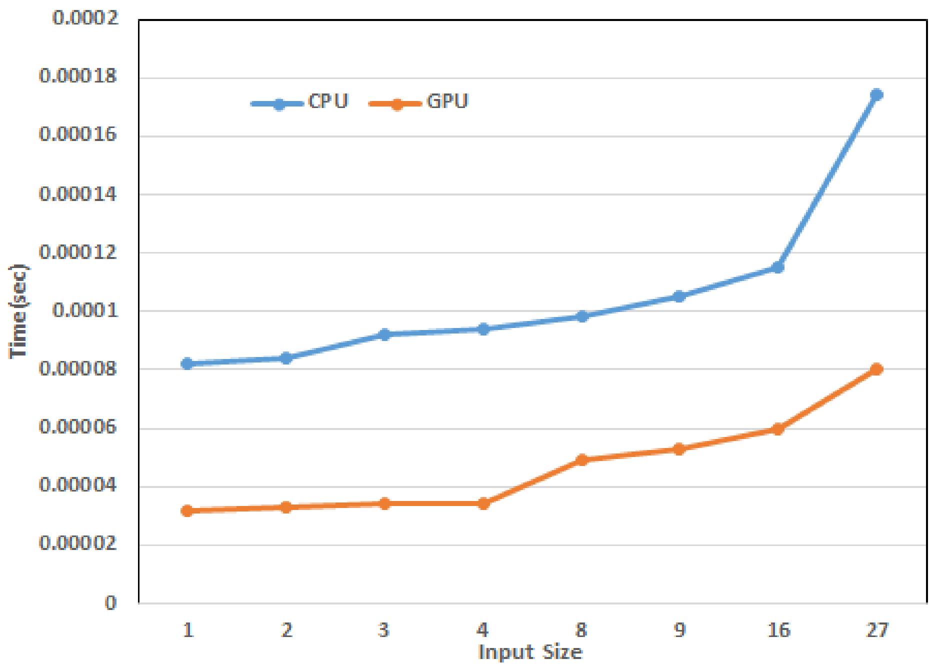


FIGURE 5.20: Covariance application execution time of CPU-GPU for 1-27 input size.

5.11 Covariance

Covariance Datamining application is executed on CPU and GPU with the different input size. In the Covariance experiment result, there are two types of variations that have been seen 1) for small input size GPU performance is better than CPU 2) but as input size increase CPU perform better than GPU. Figure 5.20 shows the execution time of CPU-GPU for Covariance application for small input size figure 5.21 shows the execution time of CPU-GPU for Covariance application for large input size. To examine why the program was always running faster on the GPU for small input size and why the program was running faster on CPU for large input size. To find out the reason for such variation analysis of the host and kernel program of Covariance is done. There are three nested loops in the Covariance host program and one dimension in the data. The analysis demonstrates that the GPU execution was significantly fast for small input size and CPU execution was fast for large input size because there is one loop unrolled from each nested loop and data is portioned in one dimension. Much as in the previous application result from ATAX CPU remain fast all the time because there is no loop unrolled in its kernel function and data is portioned in one dimension.

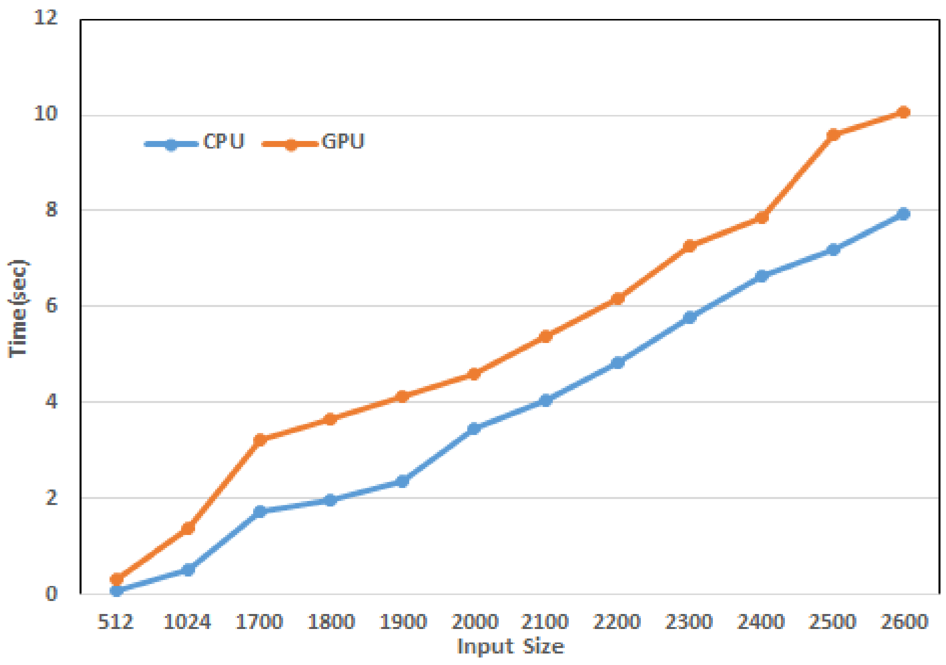


FIGURE 5.21: Covariance application execution time of CPU-GPU for 512-2600 input size.

But in the Covariance application one-loop is unrolled from each nested loop but the inner loop of each nested loop didn't unroll that why this study indicates GPU execution remains fast for small and but CPU execute Covariance kernel program fastly than GPU when input size increase.

5.12 Correlation

Correlation Datamining application is executed on CPU and GPU with the different input size. In the Correlation experiment result, there are two types of variations that have been seen 1) for small input size GPU performance is better than CPU 2) but as input size increase CPU perform better than GPU. Figure 5.22 shows the execution time of CPU-GPU for Correlation application for small input size figure 5.23 shows the execution time of CPU-GPU for Correlation application for large input size. To examine why the program was always running faster on the GPU for small input size and why the program was running faster on CPU for large input size. To find out the reason for such variation analysis of the host and kernel program of Correlation is done. There are four nested loops in the Covariance host program and one dimension in the data. The analysis

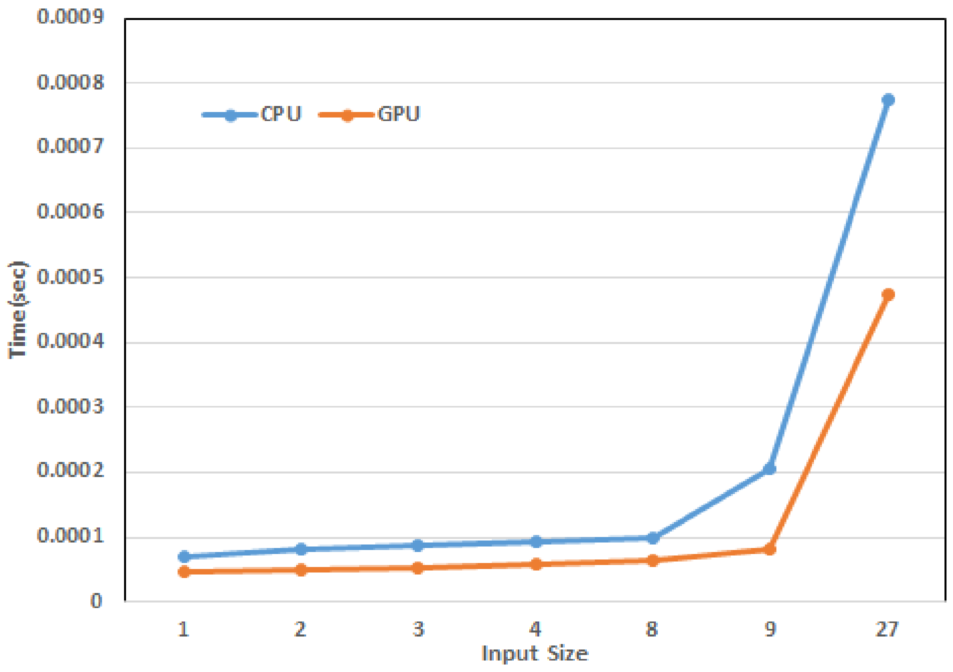


FIGURE 5.22: Correlation application execution time of CPU-GPU for 1-27 input size.

demonstrates that the GPU execution was significantly fast for small input size and CPU execution was fast for large input size because there is one loop unrolled from each nested loop and data is portioned in one dimension. Much as in the previous application result from ATAX CPU remain fast all the time because there is no loop unrolled in its kernel function and data is portioned in one dimension. But in the Covariance application one-loop is unrolled from each nested loop but the inner loop of each nested loop didn't unroll that why this study indicates GPU execution remains fast for small and but CPU execute Covariance kernel program fastly than GPU when input size increase. After execution of MVT, BICG and Covariance the results of all these applications are the same for example GPU execution impressively fast for small input but as compared to CPU execution in the same way CPU execution remains fast for large input size as compared to GPU. after analyzing that OpenCL applications host and kernel programs two things common in all application first one dimension in data second in host program of each application there are nested loop in MVT two nested loops, BICG one nested loops, Covariance three nested loop and Correlation application consists of four nested loops. in the kernel program of these applications, one loop is unrolled from each nested loop.

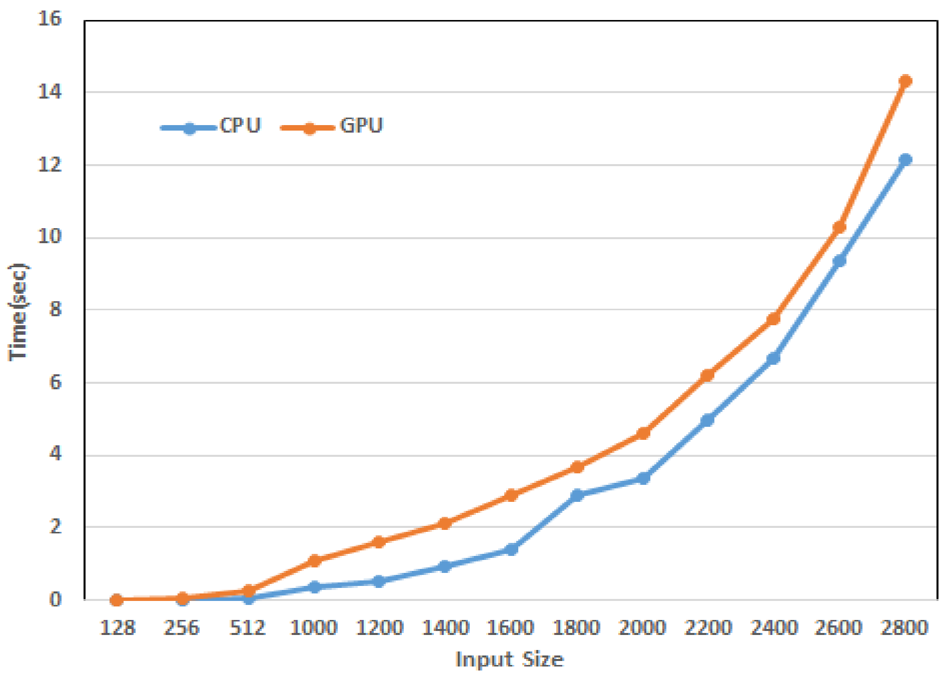


FIGURE 5.23: Correlation application execution time of CPU-GPU for 128-2800 input size.

5.13 Results Summary

In this work analysis of the different nature of OpenCL applications is done by executing these OpenCL applications on different architecture to find out which application is suitable for which architecture. Some applications like 2DCONV, 3DCONV and FDTD-2D are GPU suitable because when those applications were run on CPU and GPU with different input sizes. GPU performance is always better than CPU. To examine why these applications were running faster on the GPU, see the host program and kernel program of 2DCONV, 3DCONV, and FDTD. Those applications remain fast all the time because all loops are unrolled in kernel function and data is portioned in two dimensions. On other hand, ATAX is CPU suitable application because its execution is always fast on the CPU for all input sizes as compared to GPU. To examine why these applications were running faster on the CPU, see the host program and kernel program of ATAX. This application remains fast all the time because no loop is unrolled in its kernel function and data is portioned in two dimensions. There is also some application like BICG, GESUMV, MVT, Covariance, and Correlation which executes fast on GPU for small input size but when input size increase CPU become fast as compared to GPU. To examine why GPU perform better for small input size of these applications and CPU perform better for when input size increase for this see the host and kernel programs of these applications two things common in all application first one dimension in data second in host program of each application there are nested loop in MVT two nested loops, BICG one nested loops, Covariance three nested loop and Correlation application consists of four nested loops. In the kernel program of these applications, one loop is unrolled from each nested loop. In the experiment result, there is some application like 2MM, 3MM and GEMM which execute fast on GPU for small and large input size but some middle input size CPU perform better than GPU. To examine this type of result this sees the host and kernel programs of these applications. after analyzing that OpenCL applications host and kernel programs two things common in all application first two dimensions in data second in the host program of each application there are nested loop in 2MM two nested loops, 3MM three nested loops and in GEMM one nested loop and kernel program of these application two loops unrolled from each nested loop.

Table 5.1: Critical Table of Result.

Application	**Dimension in Data**	**# loop unrolled in Kernel funtion/ nested loop**	**Result**
2DCONV	2	All loops unrolled	GPU performs better than CPU for all input size
3DCONV	2	All loops unrolled	GPU performs better than CPU for all input size
FDTD-2D	2	All loops unrolled	GPU performs better than CPU for all input size

ATAX	1	NILL	CPU performs better than GPU for all input size
2MM	2	2	GPU perform better for small and large input size CPU performs better for some middle input size
3MM	2	2	GPU perform better for small and large input size CPU performs better for some middle input size
GEMM	2	2	GPU perform better for small and large input size CPU performs better for some middle input size
BICG	1	1	GPU perform better for small input size but CPU performs better when input size increase
GESUMMV	1	1	GPU perform better for small input size but CPU performs better when input size increase

MVT	1	1	GPU perform better for small input size but CPU performs better when input size increase
Correlation	1	1	GPU perform better for small input size but CPU performs better when input size increase
Covariance	1	1	GPU perform better for small input size but CPU performs better when input size increase

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This work analyzes the performance of OpenCL applications by executing each application on different architecture Intel i7-6700 and NVIDIA Geforce GT 740 architecture is used to find out which application GPU suitable and which one is CPU suitable after executing applications on CPU-GPU then compare the result of all applications and after that find out which software feature affecting the performance and following questions is addressed.
Research questions were the following:

.

- which application execution is fast on GPU than CPU and which applications execution is fast on GPU than CPU ?
- Which are the most important factors determining the execution time of different OpenCL applications on GPU compared to the factors impacting the execution time on the CPU?

The answer to the first question, the result of the experiment shows that there are many OpenCL applications of pollybench suit which is suitable for GPU and

some are suitable for CPU. For example, when execute 2DCONV, 3DCONV and FDTD-2D on both architecture GPU performance is better than CPU for all input size in the same way execution of ATAX application remain fast all the time on CPU compare to GPU.
There is some application like 2MM, 3MM and GEMM which execute fast on GPU for small and large input size but some middle input size CPU perform better than GPU, in the same way, some application like BICG, GESUMV, MVT, Covariance, and Correlation which executes fast on GPU for small input size but when input size increase CPU become fast as compared to GPU.
The answer to the first question is, that the number of dimensions in data and loop unrolling features affects the performance of an application when it executes on CPU and GPU. .

- If there are two dimensions in the host program and all loops are unrolled in kernel program of OpenCL application so that type applications execute fast on GPU as compare to CPU.i.e 2DCONV,3DCONV, and FDTD-2D.
- If there is one dimension in the host program and no loop is unrolled in kernel program of OpenCL application so that type applications execute fast on CPU as compare to GPU.i.e ATAX.
- If there are two dimensions in the host program and two loops are unrolled from each nested loop in kernel program of OpenCL applications so that type applications execute fast on GPU for small and large input size and execute fast on CPU for some middle input size .i.e. 2MM, 3MM and GEMM. Execution time of CPU for middle value (81-729). Those application for some value (81-729) performance is better on CPU. It mean some other feature also affecting the performnce that's why future investigation needed.
- If there are one dimension in the host program and one loop is unrolled from each nested loop in the kernel program of OpenCL applications so that type applications execute fast on GPU for small input size but when input size increase CPU perform better than GPU i.e BICG, GESUMMV, MVT, Correlation and Covariance.

6.2 Future Work

Future works within the same area would be to test more applications on CPU and GPU, to obtain some kind of general classification of what kind of algorithms are suitable for GPUs. Also test polybench suit on integrated GPU to find out the performance of these OpenCL application on integrated GPU. Then you could set up a classification for this specific GPU and draw conclusions about similarly powerful integrated GPUs. A strong starting point could be the testing of the AMD benchmark.

Bibliography

[1] J. L. Manferdelli, N. K. Govindaraju, and C. Crall, "Challenges and opportunities in many-core computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 808–815, 2008.

[2] R. Vuduc and J. Choi, "A brief history and introduction to gpgpu," in *Modern Accelerator Technologies for Geographic Information Science*, pp. 9–23, Springer, 2013.

[3] S.-A. Stefaniga and M. Gaianu, "Performance analysis of morphological operation in cpu and gpu for medical images," in *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pp. 377–384, IEEE, 2017.

[4] H. Anzt, T. Cojean, C. Yen-Chen, J. Dongarra, G. Flegar, P. Nayak, S. Tomov, Y. M. Tsai, and W. Wang, "Load-balancing sparse matrix vector product kernels on gpus," *ACM Transactions on Parallel Computing (TOPC)*, vol. 7, no. 1, pp. 1–26, 2020.

[5] N. Naz, A. Haseeb Malik, A. B. Khurshid, F. Aziz, B. Alouffi, M. I. Uddin, and A. AlGhamdi, "Efficient processing of image processing applications on cpu/gpu," *Mathematical Problems in Engineering*, vol. 2020, 2020.

[6] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *2012 innovative parallel computing (InPar)*, pp. 1–10, Ieee, 2012.

[7] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, *et al.*, "Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu," in *Proceedings of the 37th annual international symposium on Computer architecture*, pp. 451–460, 2010.

[8] N. H. Prayank, B. D. Bhalchandra, V. Bhatnagar, R. Tomer, and S. Agarwal, "Models for parallel computing review and perspectives," *International Journal of Recent Advances in science and technology*, vol. 1, no. 1, pp. 12–18, 2014.

[9] S. Ji, N. Satish, S. Li, and P. K. Dubey, "Parallelizing word2vec in shared and distributed memory," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 9, pp. 2090–2100, 2019.

[10] L. Howes and A. Munshi, "The opencl specification. khronos opencl working group," 2015.

[11] H. Lang, L. Passing, A. Kipf, P. Boncz, T. Neumann, and A. Kemper, "Make the most out of your simd investments: counter control flow divergence in compiled query pipelines," *The VLDB Journal*, vol. 29, no. 2, pp. 757–774, 2020.

[12] S. G. Shiva, *Advanced computer architectures.* CRC Press, 2018.

[13] C. A. Navarro, N. Hitschfeld-Kahler, and L. Mateu, "A survey on parallel computing and its applications in data-parallel problems using gpu architectures," *Communications in Computational Physics*, vol. 15, no. 2, pp. 285–329, 2014.

[14] "Intel® core™ i7-6700 processor." https://ark.intel.com/content/www/us/en/ark/products/88196/intel-core-i7-6700-processor-8m-cache-up-to-4-00-ghz.html. accessed: jan. 1 ,2020.

[15] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[16] M. Garland and D. B. Kirk, "Understanding throughput-oriented architectures," *Communications of the ACM*, vol. 53, no. 11, pp. 58–66, 2010.

[17] K. Raju and N. N. Chiplunkar, "A survey on techniques for cooperative cpu-gpu computing," *Sustainable Computing: Informatics and Systems*, vol. 19, pp. 72–85, 2018.

[18] "Nvidia geforce gt 740." https://www.geforce.com/hardware/desktop-gpus/geforce-gt-740/specifications. accessed: jan. 1 ,2020.

[19] C. Bertoni, J. Kwack, T. Applencourt, Y. Ghadar, B. Homerding, C. Knight, B. Videau, H. Zheng, V. Morozov, and S. Parker, "Performance portability evaluation of opencl benchmarks across intel and nvidia platforms," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 330–339, IEEE, 2020.

[20] F. Mayer, M. Knaust, and M. Philippsen, "Openmp on fpgas—a survey," in *International Workshop on OpenMP*, pp. 94–108, Springer, 2019.

[21] S. Memeti, L. Li, S. Pllana, J. Kołodziej, and C. Kessler, "Benchmarking opencl, openacc, openmp, and cuda: programming productivity, performance, and energy consumption," in *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, pp. 1–6, 2017.

[22] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling many-core performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[23] "Khronos group web page,"." https://www.khronos.org/. Available: https://www.khronos.org/.

[24] J. Tompson and K. Schlachter, "An introduction to the opencl programming model," *Person Education*, vol. 49, p. 31, 2012.

[25] M. Scarpino, "Opencl in action: how to accelerate graphics and computations," 2011.

[26] A. E. De Giusti, "Structured parallel programming: patterns for efficient computation.," *Journal of Computer Science and Technology*, vol. 15, no. 01, pp. 43–44, 2015.

[27] D. Grewe and M. F. O'Boyle, "A static task partitioning approach for heterogeneous systems using opencl," in *International conference on compiler construction*, pp. 286–305, Springer, 2011.

[28] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pp. 73–82, 2008.

[29] S. Ryoo, C. Rodrigues, S. Stone, S. Baghsorkhi, S.-Z. Ueng, and W.-m. W. Hwu, "Program optimization study on a 128-core gpu," in *The First Workshop on General Purpose Processing on Graphics Processing Units*, pp. 30–39, Citeseer, 2007.

[30] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu, "Program optimization space pruning for a multithreaded gpu," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pp. 195–204, 2008.

[31] V. Volkov and J. W. Demmel, "Benchmarking gpus to tune dense linear algebra," in *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pp. 1–11, IEEE, 2008.

[32] J. H. Lee, K. Patel, N. Nigania, H. Kim, and H. Kim, "Opencl performance evaluation on modern multi core cpus," in *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pp. 1177–1185, IEEE, 2013.

[33] G. S. Murthy, M. Ravishankar, M. M. Baskaran, and P. Sadayappan, "Optimal loop unrolling for gpgpu programs," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 1–11, IEEE, 2010.

[34] A. Salah, K. Li, K. M. Hosny, M. M. Darwish, and Q. Tian, "Accelerated cpu–gpus implementations for quaternion polar harmonic transform of color images," *Future Generation Computer Systems*, vol. 107, pp. 368–382, 2020.

[35] H. Li, H. Yang, and J. Zhao, "An image-space parallel convolution filtering algorithm based on shadow map," in *Ninth International Conference on Digital Image Processing (ICDIP 2017)*, vol. 10420, p. 104201O, International Society for Optics and Photonics, 2017.

[36] H. Jiang and N. Ganesan, "Cudampf: a multi-tiered parallel framework for accelerating protein sequence search in hmmer on cuda-enabled gpu," *BMC bioinformatics*, vol. 17, no. 1, pp. 1–16, 2016.

[37] A. Di Biagio and G. Agosta, "Improved programming of gpu architectures through automated data allocation and loop restructuring," in *23th International Conference on Architecture of Computing Systems 2010*, pp. 1–8, VDE, 2010.

[38] D. Kim, S. Kang, J. Lim, S. Jung, W. Kim, and Y. Park, "Resource-aware device allocation of data-parallel applications on heterogeneous systems," *Electronics*, vol. 9, no. 11, p. 1825, 2020.

[39] F. Kreiliger, J. Matejka, M. Sojka, and Z. Hanzálek, "Experiments for predictable execution of gpu kernels," *OSPERT 2019*, p. 23, 2019.

[40] T. Yuki and L.-N. Pouchet, "Polybench 4.0," 2015.

[41] M. Daga, A. M. Aji, and W.-c. Feng, "On the efficacy of a fused cpu+ gpu processor (or apu) for parallel computing," in *2011 Symposium on Application Accelerators in High-Performance Computing*, pp. 141–149, IEEE, 2011.

[42] S. Azmat, L. Wills, and S. Wills, "Accelerating adaptive background modeling on low-power integrated gpus," in *2012 41st International Conference on Parallel Processing Workshops*, pp. 568–573, IEEE, 2012.

[43] S. Kim, J. Bottleson, J. Jin, P. Bindu, S. C. Sakhare, and J. S. Spisak, "Power efficient mapreduce workload acceleration using integrated-gpu," in *2015 IEEE First International Conference on Big Data Computing Service and Applications*, pp. 162–169, IEEE, 2015.

[44] E. Ching, N. Egi, M. Mortazavi, V. Cheung, and G. Shi, "Unleashing the hidden power of integrated-gpus for database co-processing," *Informatik 2014*, 2014.

[45] F. Li, Y. Ye, Z. Tian, and X. Zhang, "Cpu versus gpu: which can perform matrix computation faster—performance comparison for basic linear algebra subprograms," *Neural Computing and Applications*, vol. 31, no. 8, pp. 4353–4365, 2019.

[46] Z. Huang, N. Ma, S. Wang, and Y. Peng, "Gpu computing performance analysis on matrix multiplication," *The Journal of Engineering*, vol. 2019, no. 23, pp. 9043–9048, 2019.

[47] V. Saahithyan and S. Suthakar, "Performance analysis of basic image processing algorithms on gpu," in *2017 International Conference on Inventive Systems and Control (ICISC)*, pp. 1–6, IEEE, 2017.

[48] W. Thomas and R. D. Daruwala, "Performance comparison of cpu and gpu on a discrete heterogeneous architecture," in *2014 International Conference on Circuits, Systems, Communication and Information Technology Applications (CSCITA)*, pp. 271–276, IEEE, 2014.

[49] U. Ahmed, J. C.-W. Lin, G. Srivastava, and M. Aleem, “A load balance multi-scheduling model for opencl kernel tasks in an integrated cluster,” *Soft Computing*, vol. 25, no. 1, pp. 407–420, 2021.

[50] U. Ahmed, M. Aleem, Y. Noman Khalid, M. Arshad Islam, and M. Azhar Iqbal, “Ralb-hc: A resource-aware load balancer for heterogeneous cluster,” *Concurrency and Computation: Practice and Experience*, p. e5606, 2019.

[51] Y. N. Khalid, M. Aleem, U. Ahmed, M. A. Islam, and M. A. Iqbal, “Troodon: A machine-learning based load-balancing application scheduler for cpu–gpu system,” *Journal of Parallel and Distributed Computing*, vol. 132, pp. 79–94, 2019.

Appendix A

Data Set : CPU-GPU Execution Time

Table A.1: CPU-GPU Execution Time

Benchmark	Application	Input Size	CPU-Run-Time	GPU-Run-Time
Convolution:	2DCONV	1	0.000182	0.000026
Convolution:	2DCONV	2	0.000189	0.000028
Convolution:	2DCONV	4	0.000123	0.000027
Convolution:	2DCONV	8	0.000092	0.000029
Convolution:	2DCONV	16	0.000089	0.000028
Convolution:	2DCONV	32	0.000972	0.000028
Convolution:	2DCONV	64	0.000215	0.000031
Convolution:	2DCONV	128	0.000296	0.000038
Convolution:	2DCONV	256	0.000349	0.000069
Convolution:	2DCONV	512	0.000701	0.000305
Convolution:	2DCONV	1024	0.002418	0.000816
Convolution:	2DCONV	2048	0.01027	0.002878
Convolution:	2DCONV	4096	0.040567	0.01156
Convolution:	2DCONV	8192	0.159683	0.044596
Convolution:	2DCONV	3	0.00019	0.00003
Convolution:	2DCONV	9	0.000906	0.00003
Convolution:	2DCONV	27	0.00019097	0.00003
Convolution:	2DCONV	81	0.0002391	0.000033
Convolution:	2DCONV	243	0.0003058	0.000071
Convolution:	2DCONV	729	0.00116801	0.000498
Convolution:	2DCONV	2187	0.011599	0.00346
Convolution:	2DCONV	6561	0.116789	0.030473
Convolution:	2DCONV	8500	0.192306	0.050473
Convolution:	2DCONV	9000	0.220476	0.055317
Convolution:	2DCONV	9500	0.237208	0.063071
Convolution:	2DCONV	10000	0.252689	0.066886
Convolution:	2DCONV	10500	0.288684	0.076927
Convolution:	2DCONV	11000	0.307197	0.08228
Convolution:	2DCONV	11500	0.339185	0.092343
Convolution:	2DCONV	12000	0.3728	0.09524
Convolution:	2DCONV	12500	0.398343	0.108873
Convolution:	2DCONV	13000	0.436943	0.114922
Convolution:	2DCONV	13500	0.455054	0.12688
Convolution:	2DCONV	14000	0.501917	0.130404
Convolution:	2DCONV	14500	1.172719	0.146193
Convolution:	2DCONV	15000	2.698431	0.15298
Convolution:	2DCONV	15500	20.103212	0.168965
Convolution:	3DCONV	8	0.000217	0.00004
Convolution:	3DCONV	16	0.000269	0.000068
Convolution:	3DCONV	32	0.00048	0.000157
Convolution:	3DCONV	64	0.001308	0.000708
Convolution:	3DCONV	128	0.0006279	0.002052
Convolution:	3DCONV	256	0.042658	0.014073
Convolution:	3DCONV	512	0.334873	0.111196
Convolution:	3DCONV	27	0.000391	0.000121
Convolution:	3DCONV	81	0.002154	0.000741
Convolution:	3DCONV	243	0.036858	0.012211
Convolution:	3DCONV	550	0.427389	0.146884
Convolution:	3DCONV	580	0.5071	0.170609
Convolution:	3DCONV	600	0.573949	0.182079

Convolution:	3DCONV	600	0.573949	0.182079
Convolution:	3DCONV	610	0.777658	0.199798
Convolution:	3DCONV	620	0.779186	0.206137
Linear Algebra	2MM	1	0.000105	0.000026
Linear Algebra	2MM	2	0.000121	0.000027
Linear Algebra	2MM	4	0.000149	0.000027
Linear Algebra	2MM	8	0.000247	0.000039
Linear Algebra	2MM	16	0.000196	0.000034
Linear Algebra	2MM	32	0.000189	0.000041
Linear Algebra	2MM	64	0.000241	0.000079
Linear Algebra	2MM	128	0.000773	0.00041
Linear Algebra	2MM	256	0.001149	0.003325
Linear Algebra	2MM	512	0.014797	0.028786
Linear Algebra	2MM	1024	0.238547	0.225124
Linear Algebra	2MM	2048	5.824906	1.8382
Linear Algebra	2MM	4096	91.840692	14.489396
Linear Algebra	2MM	3	0.000113	0.000028
Linear Algebra	2MM	9	0.00182	0.00003
Linear Algebra	2MM	27	0.0002028	0.000042
Linear Algebra	2MM	81	0.000287	0.000222
Linear Algebra	2MM	243	0.0011079	0.004299
Linear Algebra	2MM	729	0.0313191	0.110288
Linear Algebra	2MM	2187	2.64732	2.944499
Linear Algebra	3MM	1	0.000231	0.000137
Linear Algebra	3MM	2	0.000256	0.000134
Linear Algebra	3MM	4	0.000191	0.000169
Linear Algebra	3MM	8	0.00026	0.000134
Linear Algebra	3MM	16	0.000258	0.000039
Linear Algebra	3MM	32	0.000223	0.00007
Linear Algebra	3MM	64	0.000279	0.000149
Linear Algebra	3MM	128	0.00046	0.000658
Linear Algebra	3MM	256	0.001774	0.007918
Linear Algebra	3MM	512	0.021494	0.042176
Linear Algebra	3MM	1024	0.362389	0.337826
Linear Algebra	3MM	2048	7.572493	2.757657
Linear Algebra	3MM	4096	137.119246	8.756636
Linear Algebra	3MM	3	0.000256	0.000037
Linear Algebra	3MM	9	0.00027	0.000036
Linear Algebra	3MM	27	0.000221	0.000055
Linear Algebra	3MM	81	0.000312	0.00035
Linear Algebra	3MM	243	0.001686	0.006427
Linear Algebra	3MM	729	0.048758	0.165286
Linear Algebra	3MM	2187	3.973749	4.416818
Linear Algebra	3MM	2500	6.084055	6.329312
Linear Algebra	3MM	2700	7.825406	7.945314
Linear Algebra	3MM	3000	10.681407	10.343447
Linear Algebra	3MM	3500	21.060085	17.285947
Linear Algebra	3MM	3700	27.016805	10.333313
Linear Algebra	3MM	4000	26.732305	12.023552
Linear Algebra	Atax	1	0.000111	0.000137

Linear Algebra	Atax	2	0.000221	0.000134
Linear Algebra	Atax	4	0.0001	0.000169
Linear Algebra	Atax	8	0.000124	0.000134
Linear Algebra	Atax	16	0.000107	0.000039
Linear Algebra	Atax	32	0.000097	0.00007
Linear Algebra	Atax	64	0.000118	0.000149
Linear Algebra	Atax	128	0.000115	0.000658
Linear Algebra	Atax	256	0.000176	0.007918
Linear Algebra	Atax	512	0.000298	0.042176
Linear Algebra	Atax	1024	0.000763	0.337826
Linear Algebra	Atax	2048	0.004135	2.757657
Linear Algebra	Atax	4096	0.018337	8.756636
Linear Algebra	Atax	3	0.000094	0.000037
Linear Algebra	Atax	9	0.000114	0.000036
Linear Algebra	Atax	27	0.000198	0.000055
Linear Algebra	Atax	81	0.00012	0.00035
Linear Algebra	Atax	243	0.000175	0.006427
Linear Algebra	Atax	729	0.000457	0.165286
Linear Algebra	Atax	2187	0.003748	4.416818
Linear Algebra	Atax	6561	0.050802	11.125473
Linear Algebra	Atax	9000	0.070925	6.329312
Linear Algebra	Atax	9500	0.102996	7.945314
Linear Algebra	Atax	10000	0.103657	10.343447
Linear Algebra	Atax	10500	0.135692	17.285947
Linear Algebra	Atax	11000	0.120393	10.333313
Linear Algebra	Atax	11500	0.175205	12.023552
Linear Algebra	BICG	1	0.000056	0.000038
Linear Algebra	BICG	2	0.000086	0.000039
Linear Algebra	BICG	4	0.000059	0.000039
Linear Algebra	BICG	8	0.000062	0.000042
Linear Algebra	BICG	16	0.000079	0.000049
Linear Algebra	BICG	32	0.000075	0.000079
Linear Algebra	BICG	64	0.000092	0.000118
Linear Algebra	BICG	128	0.000054	0.000202
Linear Algebra	BICG	256	0.000143	0.000388
Linear Algebra	BICG	512	0.000237	0.000765
Linear Algebra	BICG	1024	0.000788	0.001657
Linear Algebra	BICG	2048	0.004115	0.007129
Linear Algebra	BICG	4096	0.017699	0.034627
Linear Algebra	BICG	8192	0.071557	0.143663
Linear Algebra	BICG	3	0.000068	0.000039
Linear Algebra	BICG	9	0.000073	0.000043
Linear Algebra	BICG	27	0.00008	0.000066
Linear Algebra	BICG	81	0.000074	0.000143
Linear Algebra	BICG	243	0.000134	0.000358
Linear Algebra	BICG	729	0.000383	0.0012
Linear Algebra	BICG	2187	0.003682	0.008378
Linear Algebra	BICG	6561	0.050631	0.085715
Linear Algebra	BICG	9000	0.073766	0.176264
Linear Algebra	BICG	9500	0.108158	0.195892

Linear Algebra	BICG	10000	0.103786	0.221009
Linear Algebra	BICG	10500	0.134935	0.229253
Linear Algebra	BICG	11000	0.119226	0.250893
Linear Algebra	BICG	11500	0.168973	0.27923
Linear Algebra	BICG	12000	0.158881	0.304955
Linear Algebra	BICG	12500	0.207425	0.343368
Linear Algebra	BICG	13000	0.191958	0.363237
Linear Algebra	BICG	13500	0.253128	0.397921
Linear Algebra	BICG	14000	0.243223	0.429866
Linear Algebra	BICG	14500	0.294599	0.442492
Linear Algebra	BICG	15000	0.273251	0.470395
Linear Algebra	BICG	15500	0.333892	0.505108
Linear Algebra	Gemm	1	0.000046	0.000024
Linear Algebra	Gemm	2	0.000037	0.000025
Linear Algebra	Gemm	4	0.000052	0.000025
Linear Algebra	Gemm	8	0.000038	0.000025
Linear Algebra	Gemm	16	0.000062	0.000029
Linear Algebra	Gemm	32	0.000069	0.000031
Linear Algebra	Gemm	64	0.000152	0.000052
Linear Algebra	Gemm	128	0.000181	0.000214
Linear Algebra	Gemm	256	0.000686	0.001713
Linear Algebra	Gemm	512	0.007078	0.014248
Linear Algebra	Gemm	1024	0.120715	0.114008
Linear Algebra	Gemm	2048	2.693363	0.927407
Linear Algebra	Gemm	4096	42.031056	7.326181
Linear Algebra	Gemm	8192	456.837723	25.070733
Linear Algebra	Gemm	3	0.000037	0.000024
Linear Algebra	Gemm	9	0.000041	0.000026
Linear Algebra	Gemm	27	0.000047	0.000033
Linear Algebra	Gemm	81	0.000132	0.000128
Linear Algebra	Gemm	243	0.000591	0.002187
Linear Algebra	Gemm	729	0.016496	0.055592
Linear Algebra	Gemm	2187	1.326202	1.490213
Linear Algebra	Gemm	6561	204.251776	9.421209
Linear Algebra	Gemm	8500	482.773177	25.143003
Linear Algebra	Gesummv	1	0.000059	0.000026
Linear Algebra	Gesummv	2	0.000039	0.000025
Linear Algebra	Gesummv	4	0.000041	0.000026
Linear Algebra	Gesummv	8	0.000043	0.000028
Linear Algebra	Gesummv	16	0.000038	0.000036
Linear Algebra	Gesummv	32	0.000069	0.000091
Linear Algebra	Gesummv	64	0.000045	0.000159
Linear Algebra	Gesummv	128	0.000079	0.000297
Linear Algebra	Gesummv	256	0.000552	0.000579
Linear Algebra	Gesummv	512	0.000245	0.001187
Linear Algebra	Gesummv	1024	0.000834	0.003303
Linear Algebra	Gesummv	2048	0.002636	0.015501
Linear Algebra	Gesummv	4096	0.010898	0.068479
Linear Algebra	Gesummv	8192	0.043417	0.279759
Linear Algebra	Gesummv	3	0.000044	0.000025

Linear Algebra	Gesummv	9	0.00004	0.000029
Linear Algebra	Gesummv	27	0.000074	0.000066
Linear Algebra	Gesummv	81	0.000073	0.000194
Linear Algebra	Gesummv	243	0.000107	0.000564
Linear Algebra	Gesummv	729	0.000432	0.001877
Linear Algebra	Gesummv	2187	0.003127	0.019049
Linear Algebra	Gesummv	6561	0.028596	0.17571
Linear Algebra	Gesummv	8500	0.048667	0.299259
Linear Algebra	Gesummv	9000	0.054379	0.337371
Linear Algebra	Gesummv	9500	0.057835	0.363864
Linear Algebra	Gesummv	10000	0.067241	0.419504
Linear Algebra	Gesummv	10500	0.070705	0.455109
Linear Algebra	Gesummv	11000	0.079535	0.498202
Linear Algebra	Gesummv	11500	0.083427	0.542799
Linear Algebra	Gesummv	12000	0.09934	0.596713
Linear Algebra	Gesummv	12500	0.10158	0.650584
Linear Algebra	Gesummv	13000	0.112641	0.694818
Linear Algebra	Gesummv	13500	0.117648	0.765832
Linear Algebra	Gesummv	14000	0.132437	0.828024
Linear Algebra	Gesummv	14500	0.134861	0.864967
Linear Algebra	Gesummv	15000	0.294369	0.933314
Linear Algebra	Gesummv	15000	0.357517	0.989695
Linear Algebra	MVT	1	0.000041	0.000026
Linear Algebra	MVT	2	0.000042	0.000027
Linear Algebra	MVT	4	0.000045	0.000029
Linear Algebra	MVT	8	0.000046	0.000031
Linear Algebra	MVT	16	0.000046	0.000048
Linear Algebra	MVT	32	0.000047	0.000071
Linear Algebra	MVT	64	0.000051	0.000116
Linear Algebra	MVT	128	0.000055	0.000205
Linear Algebra	MVT	256	0.000144	0.00043
Linear Algebra	MVT	512	0.000391	0.000847
Linear Algebra	MVT	1024	0.001526	0.001924
Linear Algebra	MVT	2048	0.006108	0.008164
Linear Algebra	MVT	4096	0.024259	0.037905
Linear Algebra	MVT	8192	0.103468	0.155138
Linear Algebra	MVT	3	0.000046	0.000029
Linear Algebra	MVT	9	0.000057	0.000031
Linear Algebra	MVT	27	0.000068	0.00006
Linear Algebra	MVT	81	0.000076	0.000165
Linear Algebra	MVT	243	0.000143	0.000429
Linear Algebra	MVT	729	0.000699	0.00132
Linear Algebra	MVT	2187	0.007191	0.009635
Linear Algebra	MVT	6561	0.094506	0.093204
Linear Algebra	MVT	8500	0.148368	0.169061
Linear Algebra	MVT	9000	0.139123	0.187617
Linear Algebra	MVT	9500	0.186788	0.210559
Linear Algebra	MVT	10000	0.177559	0.230276
Linear Algebra	MVT	10500	0.238171	0.244849
Linear Algebra	MVT	11000	0.216483	0.269188

Data mining	Correlation	1	0.000071	0.000046
Data mining	Correlation	2	0.000092	0.000049
Data mining	Correlation	4	0.000094	0.000043
Data mining	Correlation	8	0.00009	0.00006
Data mining	Correlation	16	0.000069	0.000137
Data mining	Correlation	32	0.0000103	0.000751
Data mining	Correlation	64	0.000262	0.003764
Data mining	Correlation	128	0.001179	0.32531
Data mining	Correlation	256	0.009103	0.074786
Data mining	Correlation	512	0.067057	0.32531
Data mining	Correlation	1024	0.521597	1.389249
Data mining	Correlation	2048	8.775795	9.383656
Data mining	Correlation	3	0.000087	0.000051
Data mining	Correlation	9	0.000085	0.000065
Data mining	Correlation	27	0.000974	0.000475
Data mining	Correlation	81	0.000416	0.006343
Data mining	Correlation	243	0.007378	0.066169
Data mining	Correlation	729	0.133603	0.68484
Data mining	Correlation	2187	13.618366	7.590744
Data mining	Correlation	2000	8.134508	8.984424
Data mining	Correlation	2500	19.25735	9.407149
Data mining	Correlation	3000	27.800201	10.662702
Data mining	Correlation	3500	52.399901	11.404163
Data mining	Correlation	4000	106.146994	10.22719
Data mining	Correlation	4500	162.325073	25.414885
Data mining	Covariance	1	0.000082	0.000032
Data mining	Covariance	2	0.000092	0.000033
Data mining	Covariance	4	0.000094	0.000034
Data mining	Covariance	8	0.00009	0.000049
Data mining	Covariance	16	0.000069	0.000128
Data mining	Covariance	32	0.0000103	0.000751
Data mining	Covariance	64	0.000262	0.003762
Data mining	Covariance	128	0.001179	0.016578
Data mining	Covariance	256	0.009103	0.074877
Data mining	Covariance	512	0.067057	0.326137
Data mining	Covariance	1024	0.521597	1.377272
Data mining	Covariance	2048	8.775795	9.396772
Data mining	Covariance	3	0.000087	0.000032
Data mining	Covariance	9	0.000085	0.000053
Data mining	Covariance	27	0.000974	0.00046
Data mining	Covariance	81	0.000416	0.006278
Data mining	Covariance	243	0.007378	0.065267
Data mining	Covariance	729	0.133603	0.685455
Data mining	Covariance	2187	13.618366	9.143673
Data mining	Covariance	2000	8.134508	8.958877
Data mining	Covariance	2500	19.25735	19.125275
Data mining	Covariance	3000	27.800201	10.084549
Data mining	Covariance	3500	52.399901	10.20873
Data mining	Covariance	4000	106.146994	11.120783
Data mining	Covariance	4500	162.325073	9.870089
Data mining	Covariance	5000	277.80133	11.814984
Data mining	Covariance	5500	386.431369	7.425238