UNIVERSITY OF BAHRAIN

College of Information Technology

"Enhancing CMS DAQ Systems Performance using Performance Profiling of Parallel Programs on GPGPUS"

A Thesis Submitted in Partial Fulfilment of the Requirements for the M.Sc. Degree in Software Engineering

Submitted by

Abdulla Ebrahim Ali Subah 20103079

Supervised by

Dr. Ahmed M. Zeki

Co-supervised by

Dr. Abdulla AlQaddoumi

Kingdom of Bahrain

June 2020

This thesis work titled "**Enhancing CMS DAQ Systems Performance using Performance Profiling of Parallel Programs on GPGPUS**") was carried out in the framework of the CMS induction project scheme. It is the result of a collaboration between the University of Bahrain (UoB), SEARCH Centre for Particle Physics a branch of Shaikh Ebrahim bin Mohammed Al Khalifa Centre for Culture and Research and the European Organization for Nuclear Research/Compact Muon Solenoid (CERN/CMS).

The University of Bahrain is greatly thankful to Shaikh Ebrahim bin Mohammed Al Khalifa Centre for Culture and Research for sponsoring Mr. Abdulla Subah to work on this CMS joint induction project. This induction project resulted in concrete contributions to CMS programs and helped the University of Bahrain to start the necessary network development within CERN/CMS and to fulfill its duties as CMS Associated Institute.

The University of Bahrain expressed its great gratitude to Sheikha Mai bint Mohammed Al Khalifa, President of the Board of Trustees of the Shaikh Ebrahim bin Mohammed Al Khalifa Centre for Culture and Research, for supporting the affiliation of the University of Bahrain with the CMS collaboration at CERN and supporting the execution of the CMS induction projects. This thesis work has been a mutually rewarding collaboration effort and will be a solid stepping stone to promote science and technology in the kingdom of Bahrain.

Defence Committee’s Approval

Abstract

Software engineers have been utilising Parallel Computing on General Purpose Graphics Processing Units (GPGPUs) in order to distribute the computing load on multiple processing units to meet increasing demand of processing powers. In order to get maximum performance from GPUs, researchers need to understand the architecture on the modern GPUs, how to optimise their programs to maximise the GPU utilisation, and how to measure the performance of GPU programs by using performance profiling tools.

The effectiveness of several GPU optimisation techniques is measured in this research through experimentations on the Data Acquisition (DAQ) system used by the Compact Muon Solenoid (CMS) experiment. Those techniques target memory access, control flow, and algorithmic optimisations. Multiple performance benchmarks are used in this research to compare the different GPU programs, such as the throughput and speedup. The benchmarking is done by using different performance profiling tools.

The results show that using the GPU shared memory decreases the number of executed instructions and clock cycles by more than 4% and 12% respectively. Using coalesced memory access pattern reduced the number of executed instructions and clock cycles by more than 71% and 44% respectively. However, using the Structure of Arrays (SoA) increased the number of executed instructions and clock cycles by less than 6% and 4% respectively. Furthermore, optimising the control flow by reducing the number of diverged threads in the GPU reduced the number of executed instructions and clock cycles by more than 57% and 68% respectively. As an algorithmic optimisation a grid data structure is developed. The grid data structure reduced the number of executed instructions and clock cycles by more than 98% and 95% respectively. All the results are in comparison to the previous optimisation iteration. All the optimisations combined resulted in more than 13 times speedup of the selected program compared to the CPU performance.

Publications

Subah, A., Zeki, A. M., Alqaddoumi, A., & Hammad, M. (2019). Using of GPUs to Enhance CMS Event Detection: Current Challenges and Future Prospective. In 2nd Smart Cities Symposium (SCS 2019) (pp. 34–38). IET.

Dedication

To my family, for their unconditional love and support.

To my teachers, for every piece of knowledge I have.

Acknowledgements

To my thesis supervisors Dr Ahmed M. Zeki and Dr. Abdulla AlQaddoumi the assistant professors at the University of Bahrain, I would like express my gratitude to them for their guidance, support, and patience.

I would also like to express my gratitude to my internship supervisor Dr. Andrea Bocci, the applied physicist at CERN for his guidance, support, and patience.

Thanks to all the staff of the Department of Computer Science and the Information Technology College in the University of Bahrain.

Thanks to all the Patatrack research group members, and CMS collaboration members for their support.

This work was supported by a grant from Shaikh Ebrahim bin Mohammed Al Khalifa Center for Culture and Research (SEARCH), many thanks to the center all its staff for their generous support.

To my family and friends, Thank you.

Abdulla Subah

Table of Contents

Chapter Two

Parallel Computing using GPGPUs in HEP, and Performance Profiling: A Literature Review

Chapter Three

Research Methodology

Chapter Four

Results and Analysis

Chapter Five

Conclusion

List of Tables

List of Figures

List of Listings

List of Abbreviations

ALU	Arithmetic Logic Unit
AoS	Array of Structures
API	Application Programming Interface
ASIC	Application Specific Integrated Circuits
CMS	Compact Muon Solenoid
CMSSW	CMS Software
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
CUPTI	CUDA Profiling Tools Interface
DAQ	Data Acquisition
ECAL	Electromagnetic Calorimeter
FP64	64-bit Floating Point Instruction
FPGA	Field Programmable Gate Arrays
GPGPUs	General Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HCAL	Hardon Calorimeter
HEP	High Energy Physics
HLT	High Level Trigger
HPC	High Performance Computing
L1	Level One Trigger
LHC	Large Hadron Colider
LS3	Long Shut Down 3
MIMD	Multiple Instructions Stream Multiple Data Streams
MISD	Multiple Instructions Stream Single Data Stream
MP	Multiprocessor
MPI	Message Passing Interface

OpenACC	Open Accelerators
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
OS	Operating System
PU	Processing Unit
SASSI	Streaming Assembler Instrumenter
SIMD	Single Instruction Stream Multiple Data Streams
SIMT	Single Instructions Stream Multiple Threads
SISD	Single Instruction Stream Single Data Stream
SM	Streaming Multiprocessor
SoA	Structure of Arrays
TAU	Tuning and Performance Analysis
TBB	Threading Building Blocks

Chapter One
Introduction

1.1. Overview

Around the year 2004 the number of transistors in a single Central Processing Unit (CPU) reached its limit due to thermal limitations (Holt 2016). Therefore, the clock speed of CPUs stopped increasing. CPU manufacturers overcame this limitation by increasing the number of Processing Units (PUs) in the CPU. This did not increase the clock speeds, but it allowed the developers to utilise parallel computing in their programs.

Parallel computing can be achieved by multi-core CPUs or co-processors such as Graphics Processing Units (GPUs) (Mittal and Vetter 2015). Software engineers have to design and implement software systems that can fully utilise the computing resources available for their software in order to reach the best possible performance.

By using performance profiling techniques and tools, software engineers are able to achieve the best possible performance and ensure maximum utilisation of the available hardware. Profiling tools can be used to measure the performance of parallel programs to find performance bottlenecks to optimise them (Shen et al. 2018).

Many scientific computing fields that require high processing powers resorted to GPUs in order to meet their computing needs. The filed of High Energy Physics (HEP) is one of them. In the HEP field, physicists study the behaviour of particles in high energies. It is a demanding field

when it comes to processing large amounts of data in limited time (Collins 2009).

This research aims at studying the use of performance profiling tools to optimise parallel programs on GPUs in the field of HEP. A program used by the HEP field is selected as a subject for the study. The program is ported, profiled, and optimised to take full advantage of parallel computing on GPUs. Each performance bottleneck is studied and an optimisation strategy is selected. The effect of the optimisations is reported. Different versions of the GPU program are compared and discussed.

1.2. Background

The Compact Muon Solenoid (CMS) is a particles detector in the Large Hadron Collider (LHC). It records the data of the generated particles from the collisions. Due to the large amount of data generated, the CMS detector software utilises parallel computing to process the data in the time limit.

Parallel computing is the field that studies how to utilise multiple processors to accelerate a single task (Blelloch 1996). GPUs are key enablers for parallel computing (Mittal and Vetter 2015). Hence, GPUs are used in the CMS detector software in order to utilise parallel computing.

GPUs have different architectures than CPUs. The GPUs are designed to maximise the throughput (Kirk 2007), which is the amount of data that can be processed in a unit of time. Therefore, to compare between a CPU and a GPU program that do the same task, throughput is used. Speedup is the measurement used to measure the improvement in the execution time

between two programs (Lee et al. 2010).

Throughput and speedup are computed with the help of programs called performance profilers. Profilers are tools used to compute metrics about a program, theses metrics can be general such as throughput, speedup, or specific such as instructions count and clock cycles. The profilers can be used to compare between the performance of different programs, and to locate performance bottlenecks.

In this project, a program from the CMS software is ported to the GPU. Porting the program is not simply rewriting it using a GPU framework. Porting involves optimising the program to fully utilise the GPU. The optimisation techniques are studied, their effect on the performance is measured and reported. The goal is to produce a program on the GPU with higher throughput than the CPU.

1.3. Problem

Currently the CMS detector has 40 million collisions per second. In the future, the events rate will increase by more than 7 times the current rate (Hegeman 2018). To keep up with the increasing data rate, the CMS is considering the use of parallel computing by utilising GPUs in the computing farm.

However, using the GPUs to process the experiment data requires re-engineering the current systems to be optimised for the GPU architecture. Before optimising the program, it is required to find a performance model that can be used to correctly measure the performance of the program and to identify the performance bottlenecks. Researchers provided different

performance models for GPUs that take different aspects in consideration. These performance models are then implemented by profiling tools (Malony et al. 2011; Stephenson et al. 2015).

In this project, the performance models are applied on the created GPU program using the profiling tools to find the performance bottlenecks. Optimisation techniques are used to overcome the performance limiters. The effect of the optimisations are measured and reported.

1.4. Objectives

The objectives of this research are:

1. Re-engineer parts of the CMS Data Acquisition (DAQ) and reconstruction systems to run on General Purpose Graphics Processing Units (GPGPUs).
    - Identify the software modules that can be transformed from sequential execution to parallel execution.
    - Identify suitable parallel algorithms to improve the modules.
    - Find strategies for data segmentation and synchronisation to deal with memory management issues.
2. Use parallel programming profiling techniques to enhance the modified modules.
    - Identify a suitable performance profiling model for parallel programs on GPGPUs.

- Profile the developed software modules.
- Use the produced metrics to look for optimisation opportunities.

3. Evaluate the produced modules and compare them to the baseline programs.

1.5. Research Questions

The research answers the following questions:

Main question: How to utilise parallel programming profiling techniques to re-engineer CMS DAQ and Reconstruction systems to achieve better performance?

Minor question (1): How to re-engineer the CMS DAQ and Reconstruction systems to use parallel computing?

Minor question (2): How to evaluate the performance of a parallel program on GPGPUs?

Minor question (3): What is a suitable model to profile the performance of parallel programs on GPGPUs?

Minor question (4): How to use the metrics obtained from the profiler to detect optimisation opportunities in parallel programs on GPGPUs?

Minor question (5): How to optimise parallel programs on GPGPUs to reduce the effect of bottlenecks?

1.6. Study Model

The model of the study is as follows:

1. **Literature Review:** The first stage is to survey the current literature to find the latest trends in parallel computing on GPUs and the use of performance profilers to optimise them.
2. **Dataset:** There are two main data sources for this study. The first one is the physics experiment data which is taken form the CMS experiment 2018 open data (CMS Collaboration 2019). The second source of data is the data extracted from profiling the parallel programs that are created.
3. **Dataset Cleaning:** The physics data is not altered. The parallel program profiling data is generated by the experiments, thus they are tailored to the needs of the research.
4. **Module Selection:** From a periodical benchmark exercise done to the CMS software, the FastJet software module was selected as a subject for this project (Benelli et al. 2011).
5. **Experiments:** The following points cover the process that governs the experimentation process:
    - The experimentation environment is described
    - The selected module is benchmarked using the selected dataset to see if the reported performance by the module developers can be reproduced.
    - A baseline parallel implementation is developed.
    - Using NVIDIA profiling tools, an experiment studies the effect of different bottlenecks and what optimisations can be done to

limit their effects.

- A profiling tool is developed based on call graph profiling techniques to help with algorithmic optimisations of the selected module.
- The experiments are repeated on different GPUs to see if the optimisations are effective on different architectures.

6. **Development Life Cycle:** A life cycle suggested by NVIDIA in their toolkit documentation (NVIDIA 2019a) is used to develop the parallel module. The life cycle consists of four iterative activities: Assess, parallelise, optimise, and deploy.
7. **Analysis and Discussion:**
    1. The results of profiling are presented, analysed, and discussed.
    2. The effect of parallel computing is analysed and discussed in terms of different performance models.
    3. The effect of different bottlenecks is studied.
    4. The effectiveness of algorithmic optimisation is analysed in light of the developed profiling tool.
    5. After each experiment, the parallel program is assessed, optimised, and compared with previous parallel and sequential implementations.

1.7. Significance

This project is significant because the need for parallelisation of data processing is a common desire for HEP experiments (Gohn 2015). Furthermore, there are several successful particle physics experiments using GPGPUs such as

- Online tracking of charged particle tracks in PANDA at FAIR (Bianchi et al. 2015).
- Reconstructing ring-shaped hit patterns in a Cerenkov detector in NA62 at CERN (Ammendola, Biagioni, et al. 2017).
- Track and vertex reconstruction for Mu3e at PSI (vom Bruch 2015).
- Track pattern recognition in LHCb (Gallorini 2015).

In addition, a speedup between 5-35 times have been achieved in different tasks in addition to reducing power consumption (Pantaleo 2017; Washbrook et al. 2011).

1.8. Scope

The following are the limitations of the project:

- One module of the CMS software is used only to experiment with.
- The testing and benchmarking is done on Monte Carlo generated data.
- The LHC was in a long shut down phase during the project, therefore the program was not tested in a production environment.

1.9. Structure

The thesis is structured into five chapters. The first chapter is an introduction to the thesis. The introduction provides a background to the problem addressed by the thesis, the objectives, and the research questions. The second chapter is a literature review. The literature review provides a com-

prehensive look to the use of performance profiling tools to optimise GPU programs in HEP. The third chapter presents the research methodology followed in the research. The fourth chapter presents the experiments conducted, the results of the experiments, and the analysis of the results. The fifth chapter concludes the thesis.

1.10. Conclusion

This chapter introduced the research by starting with the background of the study. After that, the background of the problem was stated.

A number of objectives were introduced as the main objectives of the project. The objective revolves around re-engineering software modules used in the CMS software in order to port them to GPUs.

Following the objectives, a list of research questions is listed to be answered by the researcher. After that, a study model is presented to be used to answer the research questions.

The significance of the study was shown by citing other projects which used parallel computing in the field of HEP successfully. Then the scope of the research is defined followed by the structure of the thesis.

The next chapter is a literature review of the use of parallel computing in HEP, and the use of performance profiling to optimise parallel programs on GPUs.

Chapter Two
Parallel Computing using GPGPUs in HEP, and Performance Profiling: A Literature Review

2.1. Introduction

This chapter introduces the background concepts of this study. Starting from HEP and how it introduces new computing challenges. Then moving towards how these challenges are being addressed in the field of High Performance Computing (HPC).

The chapter focuses on one concept from HPC, which is Parallel Computing. After that, GPGPUs are discussed as a key enabler for parallel computing. The discussion of GPGPUs takes this chapter through several HEP experiments that utilises GPGPUs in order to meet their computing needs.

The literature provides a resource for the main performance challenges the Parallel Computing on GPGPUs imposes. In addition, the literature guides the chapter to one of the solutions to the performance challenge which is Parallel Computing Performance Profiling.

2.2. High Energy Physics

HEP is the study of physics phenomena in particles that have been accelerated to high velocity using high level of energies. At this state, the particles are accelerated to speeds close to the speed of light in order to put them in an environment similar - as much as possible - to that of particles in the beginning of the universe (Collins 2009).

2.2.1. The Large Hadron Collider

The Large LHC is the largest physics experiment in the world. It is used to accelerate particles in high energies and collides theme to study new physics phenomena (Apollinari et al. 2015). This acceleration pushes the particles to speeds close to the speed of light. The LHC consists of a 27 kilometre ring of super conducting magnets used to boost the energy of the particles in tunnels buried 100 meter underground.

The particles are accelerated in two beams in opposite directions in the ring. Then in specific parts of this ring, these two beams are allowed to cross each other creating collision points. In these collision points, four detectors take place to take measurements which represent a snapshots of these collisions to study if there are any unknown physics phenomena in each collision. These four detectors are CMS, ATLAS, ALICE, and LHCb.

These detectors act like cameras that take snapshots of the collisions in the form of data measured using different sensors. The particles generated from the collisions interact with different types of sensors that convert these interactions to digital signals. Those signals are stored for later analysis by physicists.

2.2.2. The Compact Muon Solenoid

The CMS is one of the detectors that sits at one of the collision points along the LHC. Its objective is to observe new physics phenomena. Therefore it is referred to as a general purpose detector.

CMS is designed to work as a high speed camera that snaps three dimensional snapshots from each collisions. Each second, around forty million collisions happens inside the CMS. Most of these collisions generate an unstable particles that becomes stable in a fast transition.

After they are stabilised, the CMS is able to snap these snapshots of the collisions. Since all these collisions happen almost at the same time, the detector is required to measure the momenta and energies of each collision in different pieces and later put the pieces together to create on image for a collision to be saved for later analysis (Denegri 1995). The detector takes the shape of a cylinder with several layers of components (see Figure (2.1)).

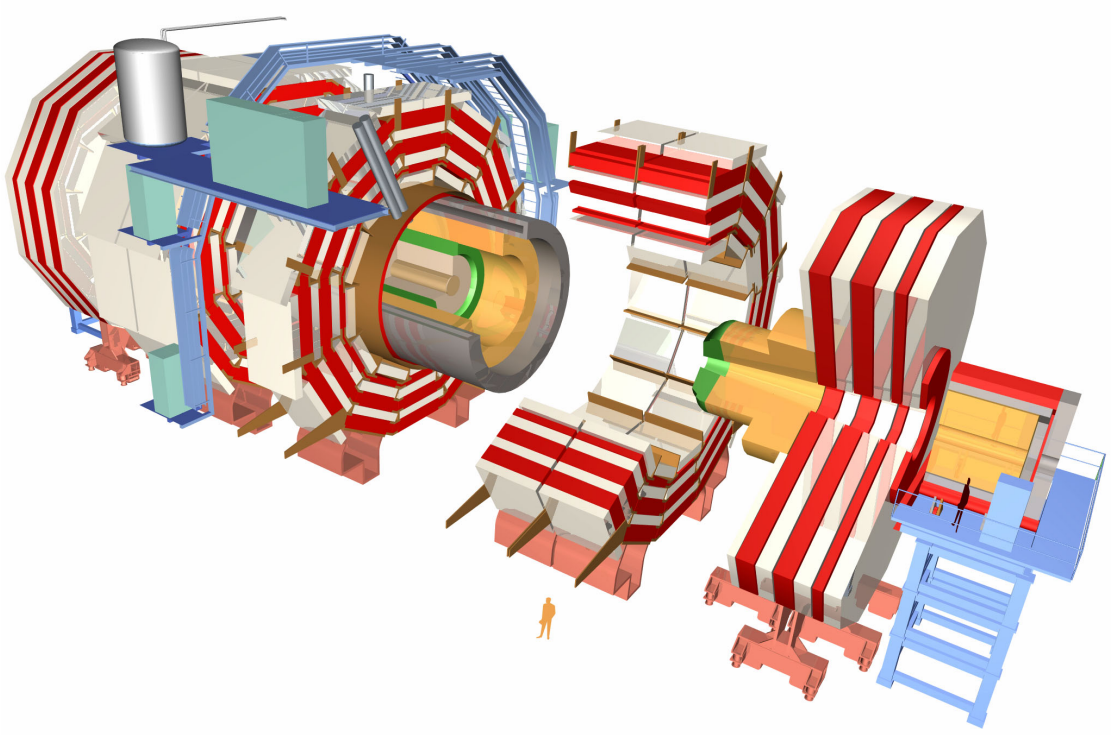

Figure (2.1): A drawing that shows the cylindrical multi layer design of the CMS detector (Collaboration 2012).

Each layer of the CMS detector is constructed by different types of sensors

that react to different types of particles. Other layers are there also to control the passing of particles through the detector layers. These layers have specific roles that can be summarised in these four points (Collaboration 1997):

i. Bending Particles

This role is carried by the solenoid magnet which this detector takes its name from. There are two benefits from bending the particles:

1. **Identifying the particle charge:** Positively charged particles bend in the opposite direction of the negative charged particles in the same magnetic field.
2. **Measuring the momentum of the particles:** Particles with higher momentum bend less than particles with low momentum in the same magnetic field.

Thus, the main functionality of the solenoid is to separate the particles generated in the collision based on their charge and momentum. These characteristics help during the analysis in determining the different types of the generated particles.

ii. Identifying Tracks

After bending the particles, the detector needs to record the paths that these particles took while they bent. This is done using a pixel-like sensors which are made of silicone. There are around 75 million electronic sensors that interact with the particles while they traverse from the innermost part of the detector to the outer most part of the detector. These silicone

sensors interact electromagnetically with the particles while they traverse through them. In each point of interaction, the sensors record a hit which is basically a dot in the 3D plane that represent the detector. Each hit is then connected with the next one creating a track for each particle that passes through the detector (see Figure (2.2)). The measurements of these tracks are used in the physics analysis to determine the particles types and their characteristics.

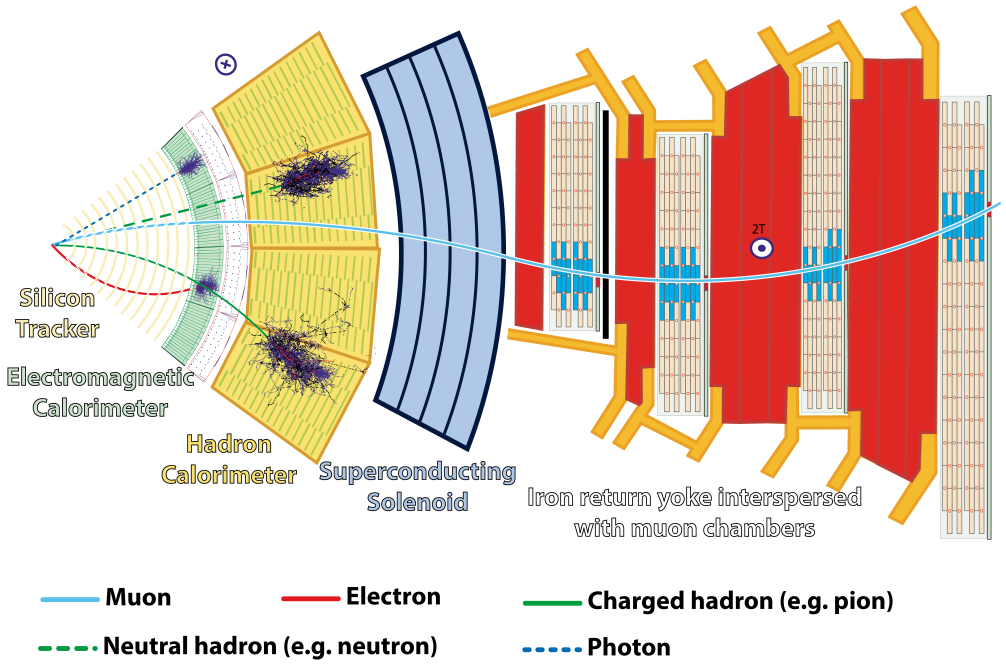

Figure (2.2): Slice showing CMS sub-detectors and how particles interact with them (Barney 2016).

iii. Measuring Energy

Calorimeters are used to stop different types of particles in the detector from escaping the detector chamber to the outside world. The particles are stopped by losing their energies while they transverse through the Calorimeter. There are two Calorimeters in CMS:

1. **Electromagnetic Calorimeter (ECAL):** It is the first Calorimeter

that interacts with the particles. It consists of crystals that interact with lighter particles with less energy such as photons and electrons. While passing through, the crystals generate fast bursts of light that allows precise measurements of the energy of the passing particles (Bloch et al. 2002).

2. **Hadron Calorimeter (HCAL):** Heavier particles with more energy such as hadrons get stopped and measured by the HCAL. Other particles that do not interact with the detector like neutrinos can be measured indirectly by the HCAL (Mans et al. 2012).

iv. Detecting Muons

Muons are particles that come from the same family of particles as the electrons, however they are 200 times heavier. Therefore, they pass through the detector without being absorbed. Thus, CMS is equipped with a sub-detector used to measure these particles' momentum.

2.2.3. Trigger and Data Acquisition

As stated in the previous section, the CMS detector works by reacting to the particles generated during collisions through different sensors that convert the reactions into digital data. These data represent an image of the collision. The system that is responsible of recording and performing data readouts from the detector electronics to the computing farm is called the Trigger and DAQ system. Unlike other triggering systems in similar physics experiments, which adopt the common three levels triggering architecture, the CMS Triggering and DAQ System adopt a two-levels triggering architecture (Andre et al. 2017).

i. Level One Trigger (L1)

This trigger mostly consists of Application Specific Integrated Circuits (ASICs), and Field Programmable Gate Arrays (FPGAs). L1 is the triggering level connected directly to the detector electronics. Therefore it performs readouts from three main components, ECAL, HCAL, and Muon detectors.

The main purpose of this trigger level is to adhere to the constraints imposed by the detector and the LHC experiment. As stated before, there are around 40 million collisions per seconds in the detector. This fact means that not all the data generated from the detector is worth keeping or can be kept due to computing and storage limitations. Another restriction is that the L1 trigger has around 4 microseconds to take a decision weather to keep the collision data or to discard it, otherwise it will not be able to keep up with the 40 MHz LHC clock (Bayatyan et al. 2000).

The L1 trigger hardware is programmed to perform specific physics tests to determine weather to keep or discard an event. Putting the 40 MHz events and the ~4 microseconds decision constraint results in 100 kHz maximum events output rate by the L1 trigger.

ii. High Level Trigger (HLT)

The HLT consists of computing nodes in a server farm. The server farm contains more than 30 thousand CPU cores. The main objective of the HLT is to provide an on-demand reconstructions and event selections. The computing nodes runs Scientific Linux operating system, and they do two

main jobs (Andre et al. 2017):

1. **Building Events:** It builds individual events from event fragments upon the request from filter units.
2. **Filtering Events:** The filter operation loads the raw data from the events into detector-specific data structures and performs the event filtering.

The HLT performs under two constraints. The first one is that it has to take a decision in an average time of 300 milliseconds. And it has an average output rate of 1 kHz. This means that out of 100 events passed from the L1 trigger to the HLT, only one is selected.

iii. CMS Software (CMSSW)

CMSSW is a modular C++ event reconstruction software package. It has more than 4000 modules written by hundreds of developers through the years. The package is configured using a standalone python library. The current running version of the software package uses multi-threading techniques to run multiple modules and to reconstruct multiple events concurrently.

iv. High Luminosity - LHC Phase 2

The LHC is upgraded and maintained in a periodical manner. These upgrades range from normal maintenance tasks to full upgrades that allows more powerful experiments to be carried. The next significant upgrade to the LHC will happen during what is known as the Long Shut Down 3 (LS3). The LS3 will take place from the end of 2022 to the end

of third quarter of 2025.

During LS3 the LHC will be upgraded to work with High Luminosity which means more energy to boost the particles. This implies that the detectors will have more collisions per second than before. As shown in Table (2.1) (Hegeman 2018), the events rate will be more than 7 times the current rate. This will generate around 5.3 PB of data each day. The current computing capacity is 0.2 PB per day.

Table (2.1): The CMS Phase-2 DAQ System captured in numbers, and compared to the current Run-2 DAQ System.

CMS detector	LHC Run-2	HL-LHC Phase-2
Event size	2.0 MB	7.4 MB
Event network throughput	1.6 Tbit/s	44 Tbit/s
Event network buffer (60 seconds)	12 TB	333TB
HLT accept rate	1 kHz	7.5 kHz
Storage network throughput	2.5 GB/s	61 GB/s
Storage capacity needed (1 day)	0.2 PB	5.3 PB

2.3. High Performance Computing

The field where researchers work on technology, architecture, programming methods, algorithms, and system software to achieve the most out of the available computing capabilities is called **HPC** (Hager 2010).

The advancements of technology allowed us to fit powerful computing devices in small form factor to be portable and affordable. However, the demand for large computing farms still exist due to new and old application domains such as climate modeling, oil and gas exploration, quantum mechanics, and other fields that require substantial processing powers. The technology advancements allows more problems that were

not possible with old technologies to be approachable with new ones. Therefore, the need for HPC will always be valid no matter how small and how fast computing technology becomes.

2.3.1. Heterogeneous Computing

The ability of computing nodes to host more than one processor of different architectures is called **Heterogeneous Computing**, It required scientists and engineers to develop new methods to utilise all the available processors in order to achieve a single task. In addition, it is one of the main enablers of HPC (Mittal and Vetter 2015).

The different processors can be available on the same computing node such as the availability of multiple CPUs and GPUs. Moreover, due to the advancements in networking solutions between computing nodes, it is possible to harness more performance by orchestrating all the available computing nodes to work on the same computing task. This requires new methods and technologies to allow all the available computing resources to work together in order to achieve HPC.

2.3.2. Parallel Computing

Parallel Computing or **Parallel Programming** is the field where researchers study how to utilise multiple processors to accelerate a single task (Blelloch 1996). The availability of multiple processing units that can be utilised to achieve higher performance in computing tasks is one of the reasons why researchers study parallel computing. Gordon Moore predicted that the transistors count per microprocessor will double every two years. This is known as the Moore's law. This prediction still holds to

these days. However, in 2004 due to thermal limitations the clock speed of new processor plateaued (Holt 2016). The only reason Moore's law still hold is because microprocessor manufactures moved towards fitting more processing units on the same chip instead of increasing the clock speed.

Parallelisation can be done either on homogeneous or heterogeneous computing farm. In addition, it can be done task wise, which means breaking the task into smaller tasks that can run in parallel, or, breaking the problem set into pieces that can be processed in parallel. Michael J. Flynn proposed a classification for computer architecture which is known as Flynn's taxonomy (1972). The taxonomy classifies parallel computing into four main models:

1. **Single Instruction Stream, Single Data Stream (SISD):** Is the model used by sequential processors, thus not exploiting any parallelism.
2. **Single Instruction Stream, Multiple Data Streams (SIMD):** A single instructions can be pipe-lined or distributed on multiple units in order to operate on different data streams.
3. **Multiple Instructions Streams, Single Data Stream (MISD):** This model is usually used in fault tolerance. Homogeneous or Heterogeneous processing units can process the same data stream using different implementations, this can be used to check for faults in critical systems.
4. **Multiple Instructions Streams, Multiple Data Streams (MIMD):** This model represents the multi-processor system that

can be used to execute multiple tasks simultaneously on different data.

Modern computing solutions implement parallel architectures mentioned above in order to maximise the performance of the computing resources. This implementation happens on different levels:

1. **CPU Level:** Producing multi core CPUs that can run parallel instructions on parallel threads.
2. **Computing Farm Level:** Implementing network solutions and message passing standards that can be used to facilitate communications between processes on different computing nodes.
3. **Co-Processors:** Attaching a co-processor to the computing node. It can be used as an accelerator that communicate with the CPU to offload some of the tasks from the CPU to the accelerator. GPUs are an example of this implementation.

i. Multi-Threading

Modern computers implement the above classifications by using heterogeneous architectures or parallel computing programming models. For example, a multi-threading library such as Intel Threading Building Blocks (TBB) can be used to implement an SIMD program by using data vectorisation. Hence, each thread processes one vector of data. The same library can be used with multi-thread CPUs to implement MISD or MIMD models by using task parallelism (Reinders 2007).

ii. Cluster Computing

Another example of how parallel computing is implemented is the Message Passing Interface (MPI) (Gropp et al. 1996). It is a message passing standard implemented in different programming languages such as C++ and Fortran. The standard provides a mechanism of communication between different computing nodes which allows computer farms with many nodes to be utilised in implementing SIMD, MISD, and MIMD. Hence, it is widely used in cluster computing as communication standard between different processes in different computing nodes.

iii. Co-Processors

Co-Processors have been used as accelerators in computing nodes for HPC applications. The main processor, also known as the host, offloads part of the computations tasks to these accelerators in order to achieve better performance. The co-processor usually provides an architecture optimised for a certain task such as floating-point computations or graphics applications. Researchers in the early 2000 (Zhe Fan et al. 2004) have used GPUs as accelerators due to their parallel computing capabilities. In the past decade, GPU manufacturers produced programming models that allow the use of GPUs for general purpose applications (Kirk 2007; Munshi 2009).

2.3.3. Parallel Computing Using GPUs

Originally GPUs were meant as an application specific co-processors. They were used to offload the calculations needed for graphics applications such as rendering from the CPU. The GPU architecture was optimised for throughput of operations, unlike CPUs which are optimised

for faster clock cycles. In early 2000s researchers realised that the GPU architecture can be used by exploiting rendering libraries to accelerate scientific applications. They found that scientific applications can benefit from such parallel architecture (Zhe Fan et al. 2004).

2.3.4. GPUs Architecture

Since CPUs are designed to maximise the clock cycle, their architecture provides large cache memory in order to minimise the number of times the Arithmetic Logic Units (ALUs) needs to fetch data from slower memory types like the RAM. However, the GPUs provide more ALUs with smaller cache memory. This design allows for more data processed per unit of time instead of instructions processed per unit of time, and that is why GPUs are considered massively parallel processors. Figure (2.3) shows an illustration of CPU and GPU architectures.

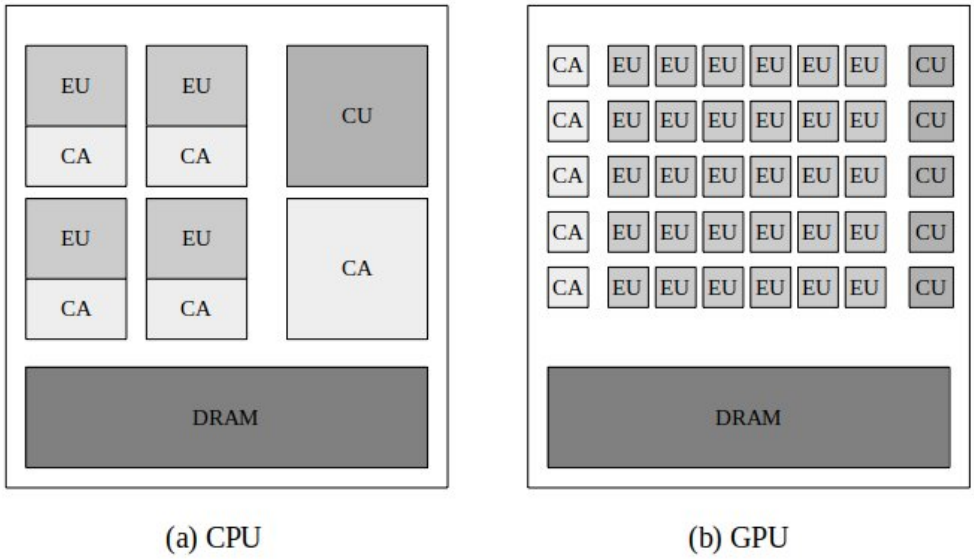

Figure (2.3): A comparison between the architecture of a GPU and a CPU.

The key difference between a CPU and a GPU is the processing capability

of a single ALU. Modern CPUs in general have more powerful (faster clock cycles) ALUs than those available in GPU cores. In addition, the size of fast access memory such as the L1 cache is larger in CPUs than in GPUs. Therefore, GPUs do not outperform CPUs in the speed of processing data but in the throughput (Ramroach et al. 2019).

Throughput is the amount of data processed in a unit of cost (Lee et al. 2010). The cost can be time, wattage, money, or any other unit of cost. Since GPUs do not outperform CPUs in clock cycle, they need to outperform them in the amount of data they process. Hence, the amount of data transferred from host memory (CPU RAM) to device memory (GPU DRAM) and the resulting data measures the effectiveness of GPUs against CPUs.

2.3.5. GPUs Programming Frameworks

After realising the potential of GPUs as general purpose processors, researchers started creating programming frameworks that allow writing programs that can be executed on GPUs without the need for using rendering libraries. Generally, these programming frameworks treat the GPU as a co-processor for the CPU. Therefore, they treat the CPU as a host that the main program is written for. Inside the main program some functions can be called to be executed in the GPU which is usually called the device. These parts are called device functions. The programming models rely on a run time that manages the memory allocation and transferring between the host and the device. Here are the most used programming frameworks to write programs for GPUs.

1. **Open Accelerators (OpenACC):** It is a variation of the Open Multi-Processing (OpenMP) application programming interface (API) (Wienke et al. 2012). While OpenMP targeted CPUs only in a homogeneous computing setup, OpenACC is a standard that expanded OpenMP to provide a simplified model to program heterogeneous CPU/GPU systems. OpenACC provides compiler automation and programming language extensions to allow parallel programming for heterogeneous processors.
    - ***Advantages:*** OpenACC abstracts most of the details of parallelising a program. This abstraction moves the burden of writing the details of the parallel program from the developer to the compiler. This also allows OpenACC to be portable between different architectures from different vendors. This property is referred to as Performance Portability. In addition it is an open standard that can be implemented by any vendor. Code listeing (2.1) shows a sample OpenACC program.
    - ***Disadvantage:*** Being general and abstract gives the developer little space of flexibility to exploit parallel computing to its most potential.

```
main()
{
    /* serial code */
    #pragma acc kernels
    // automatically runs on GPU
    {
        /* parallel code */
    }
}
```

Listing (2.1): OpenACC code example.

2. **Open Computing Language (OpenCL):** It is a framework that allows programmers to target massively parallel processors through a model that defines a set of run time APIs and language extensions as shown in code listing (2.2) (Munshi 2009). It targets heterogeneous computing nodes the have a CPU as a host and other accelerators as devices. The devices can be GPUs, FPGAs, or other types of accelerators.

```
kernel()
{
    /* parallel code
    to be executd on the GPU */
}

main()
{
    /* call to OpenCL or CUDA API
    to lunch the kernel on the GPU */
}
```

Listing (2.2): CUDA / OpenCL code example.

- ***Advantages:*** It depends more on the APIs rather than depending on the programming extensions. This makes it more portable and it makes providing support for it from different vendors an easier task. In addition, it is an open standard that can be implemented by any vendor.
- ***Disadvantage:*** While it is portable, it does not provide the same performance across different devices and architectures. Developers have to spend significant amount of time tuning their device functions to reach high performance on each processor.

3. **Compute Unified Device Architecture (CUDA):** It is a parallel computing platform developed by NVIDIA as a general purpose parallel programming interface to allow developers to write programs that run on GPU manufactured by NVIDIA (Kirk 2007). It provides language extensions to the C and C++ programming languages and several APIs to control the GPUs. It shares the same code layout with OpenCL (see code listing (2.2)), however CUDA relies more on the programming exertions rather than relying on the APIs.

    - ***Advantages:*** CUDA allows programmers direct access to specific hardware features, which allow the programmers to exploit the full potential of the GPU. Since it allows direct access to different memory levels on the GPU and specific GPU cores, CUDA is suitable to study parallel computing on GPUs. It gives the developer an in-depth understanding of the GPU architecture.

- ***Disadvantage:*** CUDA is not portable, it is a proprietary framework owned by NVIDIA, therefore it can only work on GPUs manufactured by NVIDIA. CUDA programs might even break moving between old and new NVDIA GPUs.

2.3.6. GPUs Memory Hierarchy

Each GPU programming framework has its own abstraction for the GPU hardware components starting from threads, cores, and memory types. This study focuses on the CUDA framework, hence this is a description of how the CUDA framework abstracts the GPU memory. In CUDA framework, the device consists of three types of memory (NVIDIA 2019a) (see Figure (2.4)):

1. **Global Memory:** The memory that is exposed to the host and can be accessed by all processing units. It is the slowest memory in terms of access time, but it is the largest in-terms of size (10s of GBs).
2. **Shared Memory:** The memory available for each block of threads. It is faster than the global memory, but it is significantly smaller than the global memory (1000s of KBs).
3. **Local Memory:** The memory available per-thread. It is the fastest memory in-terms of access time, but it is the smallest in size (10s of KBs).

2.3.7. GPUs Parallel Execution Model

Another difference between the CPU and GPU architectures is how the parallelism is handled. CPUs provide more flexibility in terms of parallel

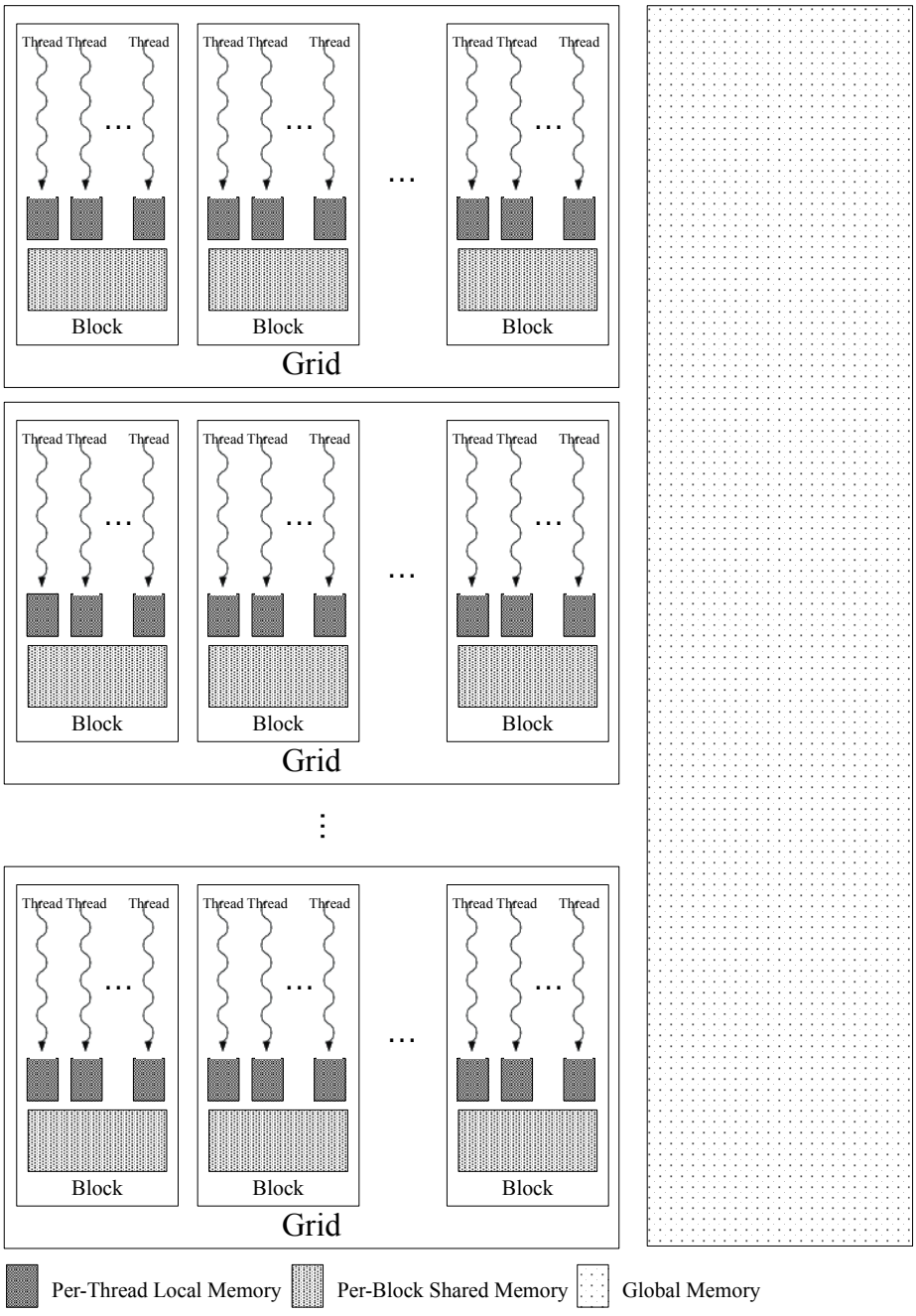

Figure (2.4): GPU memory hierarchy.

programming models. A CPU can be used for all the parallel models described earlier. However GPUs architecture is optimised for SIMD or Single Instruction Stream, Multiple Threads (SIMT) models.

SIMT is a parallel computing model that is used by GPUs generally. It is an extension of the the SIMD. From NVIDIA GPUs point of view, each parallel program consists of several instructions. These instructions are executed by threads in parallel using a streaming multi-processor (GPU processing core). The threads are grouped in warps of fixed size. The warp size differs between each generation of GPUs. Each warp has one program counter. Therefore, each group of threads can only advance in execution simultaneously (Kirk 2007).

2.3.8. GPUs in HEP

HEP is a demanding field when it comes to data processing. As mentioned before, the CMS experiment generates around 40 Terabytes of events data per second. The number is going to increase more than seven times in the next upgrade of the experiment. Therefore, researchers have been studying and proving that the use of parallel computing in HEP can be effective in dealing with such high data rates (Gohn 2015).

GPUs are key enablers for parallel computing. Hence, the use of GPUs in HEP has been increasing in the past few years. Both CMS and ATLAS have proven use cases for GPUs. Developers managed to gain speedups up to 35 times in some experiments (Pantaleo 2017; Washbrook et al. 2011). In addition, GPUs proved to be more efficient in terms of cost and energy consumption (Pantaleo 2017; Washbrook et al. 2011).

2.3.9. Parallel Computing Performance Challenges on GPUs

Writing parallel programs or converting sequential programs into parallel programs is not a straightforward task. There are many challenges that could make the end result not performing as expected. Therefore, developers need to spend a substantial amount of time to optimise the parallel program to fully utilise the GPUs potential.

This part of the literature review surveys the latest literature to identify the challenges in developing parallel programs for GPUs. From the CUDA toolkit documentation (NVIDIA 2019a) and the experiments that were surveyed, it is found that there are four main reasons behind the performance challenges for parallel programs on GPUs. These are the memory, instructions, execution configuration, and control flow. While these reasons are mostly related to GPUs manufactured by NVIDIA, they are applicable to other GPUs from other vendors due to the similarities in the overall architecture of the GPUs (Jang et al. 2011).

i. Memory

From a memory point of view, getting the best performance out of GPUs means optimising the memory access patterns to each of the memory types mentioned. The global memory needs to be accessed the least. In a perfect situation as in an embarrassingly parallel problem, the program accesses the global memory only to transfer data back and forth between the host and the device. The data need to be almost always in the local memory of the thread unless inter-thread communication is needed, then the shared memory can be used. This is known as the data locality (Jang

et al. 2011).

In addition, NVIDIA GPUs access the memory in terms of cache lines similar to CPUs. However, these cache lines can be used by a group of threads called warps. Each warp can use the same cache line in a single memory access. This memory access pattern is called a coalesced memory access (Xinxin Mei and Xiaowen Chu 2017). To ensure the GPU memory access is efficient, the developers need to make sure that their programs are accessing the memory using an optimal pattern.

ii. Instructions

Another reason for performance challenges in GPUs that might not be present or as effective in CPUs, is the type of instructions used. Some mathematical operations can be very slow on GPUs compared to other types of instructions. As an example is the division and the square root operations. While they are slow also on CPUs, since GPUs have significantly slower ALUs, these instructions can cause a performance bottleneck (Nickolls and Dally 2010).

Another issue can be the floating point operations. Some calculations need a certain level of precision used. Historically, GPUs were not designed to perform accurate floating point operations. However, this changed with increasing use of GPUs in general purpose programming. Single precision floating point operations are well supported in newer GPUs (Jia, Maggioni, Staiger, et al. 2018). However, it is not the case for double precision floating point operations. Double precision operations on GPUs are significantly slower. Some top tier modern GPUs have special double

precision processors to limit this problem (Nickolls and Dally 2010). In the most cases, developers have to use lower precision to maximise the performance if possible. Otherwise double precision operations can be a major bottleneck.

iii. Execution Configuration

Different parallel computing programming frameworks provide different capabilities and approaches towards parallelisation. On one hand, frameworks such as OpenACC provide an easier approach to parallel computing. However, the developers have to sacrifice the flexibility and the ability to fully utilise the massively parallel processors. On the other hand, frameworks such as CUDA and OpenCL provide the developers with more control over the massively parallel processor. However, this level of control comes with a price. The developers have to come up with new algorithms and data structures that are optimised for massively parallel processors such as GPUs.

Granularity is the property in a parallel problem that describes how big each parallel task is compared to the whole size of the problem. Any computing problem can be categorised into one of three categories when it comes to their tendency to parallelise and take the most advantage of a massively parallel processor (Lustig and Martonosi 2013).

1. **Fine-grained:** When the processing units spend more time on communication than on executing the work assigned to them.
2. **Coarse-grained:** When processing units communication time is insignificant compared to the time they spend executing the task.

3. **Embarrassingly Parallel:** Those problems that can be broken down into tasks that does not need or rarely communicate.

In frameworks such as CUDA and OpenCL, it is the developer's duty to express the problem in Coarse-grained or Embarrassingly Parallel manner to take the most advantage of massively parallel processors.

iv. Control Flow

Due to the SIMT parallel model used by NVIDIA GPUs, each thread in a warp shares the same program counter. Therefore, if the code contains branches, such as loops or conditional statements, and threads in the same warp have to take a different branch, the GPU serialises the execution of the instructions. This means the execution time of the branched code is the sum of execution times of all branches. This problem is called branch divergence (Han and Abdelrahman 2011).

2.3.10. Counter Measures

The literature contains many examples of using GPUs to utilise parallel computing in HPC. Each project usually faces one or more of the challenges mentioned above. Table (2.2) presents some of the literature that were surveyed, the challenges they faced, and the counter measures that were take to overcome these problems.

Table (2.2): GPUs challenges surveyed from different experiments.

Experiment	Challenges	Counter Measures
(Ammendola, Bauce, et al. 2013)	**Memory:** Current data structures are inherited from the sequential program. Most of the time is spent in preparing data to work with the parallel algorithm.	**Algorithmic Optimisations:** Develop new data structures and algorithms that are optimised for parallel programs.
(Bozza et al. 2015)	**Control Flow:** The code is highly divergent, hence the more cores the GPU have the slower the algorithm becomes.	**Micro Bench-marking:** By fine tuning the code the researchers managed to gain some wasted performance back, however they are questioning if the performance gain is worth the effort.
(Lopes et al. 2015)	**Control Flow:** The algorithm requires atomic transactions which causes the parallel code to serialise, and the code is highly divergent.	**Algorithmic Optimisations:** The researchers combined different parallel computing techniques to overcome the need for atomic transactions.
(Adinetz et al. 2015)	**Execution Configuration:** Comparing different execution configurations such as dynamic parallelism, joined kernel, and host streams. **Control Flow:** The code requires branch management in some cases. **Instructions:** The algorithm requires trigonometric functions and square roots.	**Algorithmic Optimisations:** Managed to reach higher GPU occupancy by splitting the data into bunches. **Trade-offs:** Implemented a work around to stop using expensive instructions.
(vom Bruch 2015)	**Control Flow:** 87% of the created threads were idle.	**Micro Bench-marking Optimisations:** Different launch parameters were tested with multiple launch configurations.

(Grasseau et al. 2015)	**Execution Configuration:** The algorithm produced varying performance results using different hardware and configurations.	**Algorithmic Optimisations:** Targeting optimal kernel sizes allowed for more consistent performance results.
(Schouten et al. 2015)	**Execution Configuration:** GPU utilisation is only 20%.	**Micro Bench-marking Optimisations:** The researchers suggested that performance tuning allows better GPU utilisation, however their study didn not contain any results that support this claim.
(Vogt and Schröck 2015)	**Memory:** Access patterns to memory locations were not optimal, the algorithm exhausted the local memory registers.	**Algorithmic Optimisations:** New algorithm was developed to distribute the load of local memory on several threads. In addition using coalesced memory access pattern helped enhance the performance.
(Spera 2015)	**Memory:** The data needs to be accessed frequently for the calculations which resulted in high amount of memory hits. **Instructions:** The algorithm require double precision instructions to meet its accuracy requirements.	**Algorithmic Optimisations:** The algorithm was written to use the shared memory as much as possible which reduces the memory access latency. **Trade-offs:** Some double precision instructions can be changed to single precision while maintaining the required final accuracy.
(Di Domenico 2015)	**Memory:** The access pattern enforced by the dataset is not optimal for GPUs, and the algorithm requires high number of memory access.	**Algorithmic Optimisations:** The algorithm was written to distribute the memory access operations to optimise the memory access pattern. In addition, it was optimised to use local memory as much as possible.
(Jun et al. 2019)	**Memory:** The needed to make multiple copies of the data was slowing data transfer to the GPU.	**Algorithmic Optimisations:** Use multiple data structure in order to limit the data transfers between the CPU and GPU.
(Yeo et al. 2019)	**Control Flow:** The performance degraded due to a highly diverged code.	**Algorithmic Optimisations:** The divergence was reduced by dividing the parallel kernels into multiple sub-kernels.

(Bi et al. 2020)	**Instructions:** The dependence on double precision instructions in the computations was a bottleneck on GPUs.	**Trade-offs:** The use of double precision calculations is not important to the researchers, therefore they chose to change it to single precision calculations.
(Rovere et al. 2020)	**Control Flow:** The algorithms in the literature depends on hierarchical execution method which is not optimal for GPUs.	**Algorithmic Optimisations:** Transforming the hierarchical clustering method to a parallel clustering method.

The counter measures in the literature can be summarised in three points:

1. **Algorithmic Optimisations:** The most effective optimisations that were found in the surveyed literature were due to optimising the algorithms to be more parallel or developing new algorithms that solve the problem in a parallel manner. This type of optimisation proved to be effective and portable between different parallel platforms.
2. **Micro Bench-marking Optimisations:** As a counter measure, the developers run micro benchmark tools to make sure that they are utilising the GPU to its full potentials. This requires focusing on the architecture low level optimisations. These optimisations provide significant performance gains. However, it is a hardware specific in most cases and is not be useful on different configurations other than the original configurations developers used.
3. **Trade-offs:** In some cases the developers have to sacrifice precision or relaxed some constraints in order to get the most out of the GPU. This might be acceptable in some cases, however in other cases there are constraints that can't be relaxed.

2.3.11. Performance Profiling

Profiling is a form of dynamic software analysis that measures time and memory complexity, or the usage frequency of instructions, components, or any other resources. It is usually used to optimise programs to achieve better performance results. Graham et. al. the authors of the previous definition (1982) classified performance profilers into two types based on the output they generated:

1. **Counters:** The profiling produces data about the frequency of instructions or function calls. It gives insight on how the algorithm behaves.
2. **Timers:** The profiling produces data about how long it takes a certain segment of the code to finish execution. It gives an insight on where the execution time is mostly spent.

They also classify them by the method they generate data:

1. **Sampling:** Collecting the data from call stack or from hardware counters by issuing operating system interrupts during the execution of the program.
2. **Instrumentation:** Manually or automatically inserted instructions to collect information about the program execution.

In terms of data presentation, generated data profiles can be classified into tow classes:

1. **Flat Profile:** When the profiled components are shown independent of each other in such a way that the effect of one component on the

others is not presented, this is called a flat profile.

2. **Call Graph Profile:** When the profile data is presented in a way that shows the relationship between the components and how they are affected by each other, this is called a call graph profile.

Profiling tools usually use a mix of the listed methods of profiling and presentation in order to provide the intended profiling functionality.

Parallel Computing Profiling: To overcome the performance challenges imposed by developing parallel programs, developers have been using standard and specialised profiling techniques to be able to achieve most of the performance capabilities of GPUs. This part of the literature review surveys several profiling models, tools, and techniques used on GPU programs.

- **TAU:** The Tuning and Performance Analysis (TAU) system is a collection of profiling tools that work with different languages. It is portable between different devices and architectures. In addition, it provides visualisation tools for performance metrics in single device, across multiple devices, or computing nodes. Through the CUDA Profiling Tools Interface (CUPTI), TAU has been extended to work on NVIDIA GPUs using CUDA (Malony et al. 2011).

- **G-HPCToolkit:** The GPU HPC Toolkit (G-HPCToolkit) is an extension to the HPCToolkit that is focused on GPU powered supercomputers. It uses CUPTI to support CUDA powered GPUs. The tool uses different sampling techniques asynchronously to collect data from different computing nodes then aggregate them using a

profiling engine. The main target of this tool is supercomputers or computing clusters (Chabbi et al. 2013).

- **SASSI:** The Streaming Assembler Instrumenter (SASSI) is developed by NVIDA research lab as an instrumentation tool for the code generated by the CUDA assembler. SASSI dose not depend on the interface provided by CUPTI. Hence, it instruments on a lower level providing access to the underlying assembly code. Since it is tied to the assembly code, it works only on CUDA programs and therefore supports NVIDIA GPUs only (Stephenson et al. 2015).

- **CUDAAdvisor:** CUDA Adviser is a tool created to profile GPU programs with a similar approach to SASSI. However, it is on a higher level which makes it portable by design and supports different GPUs and different programming languages and frameworks. It is not developed by NVIDIA, hence it does not provide the same profiling capabilities as SASSI (Shen et al. 2018).

2.3.12. Parallel Profiling Data Visualisation

Performance analysis is a key factor in finding bottlenecks in programs. The reviewed tools provide different range of profiling capabilities that ranges from profiling on single node to profiling computing clusters. The generated metrics are key components in any performance analysis endeavour. However, these metrics can be overwhelming and especially in parallel programs where there are large number of concurrent processors, processes, or thread.

To put the generated metrics in useful perspective, packages such TAU

and G-HPCToolkit provide visualisation tools that show the behaviour of the program across multiple machines. While this is useful in assessing the scale-ability of the parallel program, it does not provide useful insight about how the algorithm itself behaves. These tools do not provide an effective way of targeting algorithmic optimisations.

The literature shows that algorithmic optimisations provide more performance advantages than other types of optimisation (refer to Table (2.2)). However, the surveyed tools do not provide sufficient capabilities in terms of studying the behaviour of the algorithm. The tools are more focused on the metrics sampled from the hardware counters and providing proper explanation of these metrics.

Hatchet is a visualisation framework that takes the output of profilers and feed it to data analysis library in order to provide insight about how the parallel program behaves for different configurations or input data (Bhatele et al. 2019). The framework provides programmable interface for the developers to provide the needed visualisations (see Figure (2.5)) for their parallel programs. It is dependant on the data provided by the profilers and does not provide any capabilities that allows producing more data.

2.4. Conclusion

This chapter provided an introduction about the computing problem imposed by the requirements of HEP applications. It moves to describe how the CMS detector as an HEP experiment provides a sizeable challenge when it comes to computing. The chapter introduced HPC as the comput-

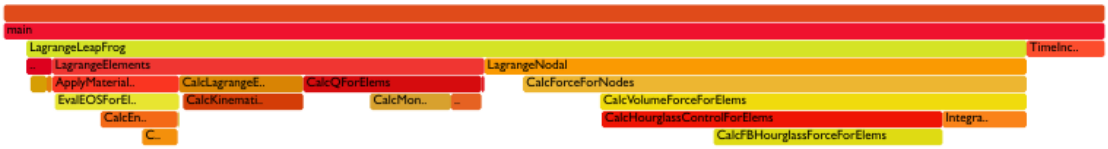

Figure (2.5): Using a flame graph to visualise the behaviour of an algorithm using the Hatchet visualiser (Bhatele et al. 2019). The intensity of the color indicate the frequincy of exeucution of that part of the program while the length of the bar shows the relative duration.

ing paradigm that solves the large scale computing challenges imposed by HEP.

New advancements in HEP requires more sophisticated computing challenges. Parallel computing is a prime candidate in taking these challenges. The chapter discussed the different parallel computing models and their applications such as multi threading, cluster computing, and co-processors or accelerators.

The literature showed that HEP field has been experimenting with parallel computing to accelerate their programs. One platform of choice for HEP applications are GPUs. The chapter also reviewed a number of HEP experiments in order to scout for different experiences and challenges. A summery of the most occurring challenges was provided alongside the proposed counter measures.

One important counter measure for parallel programs challenges is to use performance profiling tools to identify and optimise parallel program bottlenecks. Several parallel profiling tools were reviewed and compared. The major feature that these tools does not provide is the ability to visualise the behaviour of the parallel algorithm. Hatchet is a tool that provides this

ability as a standalone solution, however it is dependant on the output of profiling tools.

Based on this literature review, chapter three introduces the methodology that is used to answer the research questions of this work.

Chapter Three
Research Methodology

3.1. Introduction

In this chapter the research methodology that is followed throughout the project is presented. First, a summary of how this research project is provided, the following sections break this summary into details. The third section in the chapter discusses the sources of data that are used in the project. Then, section four goes through several benchmarks that are used to measure the performance of the developed modules. Section five introduces theoretical limits for parallel computing in light of the benchmarks discussed in the benchmarks section. After that, the selected module from CMSSW is explained. The development life cycle of the parallel implementation of the selected module is illustrated in section seven. The next section introduces the profiling tool that are used in this project followed by the experimental framework.

3.2. Research Design

This research is designed to follow a quantitative research methodology. The work seeks to collect data using defined tools and experiments. Then the data is analysed and discussed. The following steps represent a summary of the research stages that were followed:

1. **Literature Review:** The first stage is to survey the current literature to find the latest trends in parallel computing on GPUs. The survey then looks for example experiments that applied parallel

computing especially on GPUs. From those experiments a list of common challenges are extracted alongside the solutions that were implemented by the researchers. Then, the review lists several tools that can be used to profile parallel programs on GPUs as a mean to overcome the surveyed challenges. Finally, a potential research area is selected as a bases for the study. The literature review was presented in Chapter 2.

2. **Dataset:** There are two main data sources for this study. The first one is the physics experiment data which is used as an input to the parallel computing problem that is studied. The physics experiment data is taken form the CMS experiment 2018 open data. The second source of data is the data extracted from profiling the parallel programs that are created. This data is extracted using tools available or built for this experiment. The data differs depending on the experiment that is carried.
3. **Dataset Cleaning:** The physics data is not altered, it is assumed that the dataset is accurate from the physics point of view due to the physics part being outside the scope of this study. The parallel program profiling data is generated by the experiments, thus they are tailored to the needs of the research.
4. **Module Selection:** The CMSSW goes through periodical profiling exercises to benchmark its performance. From this exercise, a list of potential applications is extracted for possible improvements (Benelli et al. 2011). One module that was selected as a potential subject for improvements was the FastJet module. Hence, the module was selected as a candidate for the study. The FastJet

software package is discussed in detail later on in this chapter.

5. **Experiments:** The following points cover the process that governs the experimentation process:
    - The experimentation environment is described
    - The selected module is benchmarked using the selected dataset to see if the reported performance by the module developers can be reproduced.
    - A baseline parallel implementation is developed.
    - Using NVIDIA profiling tools, an experiment studies the effect of different bottlenecks and what optimisations can be done to limit their effects.
    - A profiling tool is developed based on call graph profiling techniques to help with algorithmic optimisations of the selected module.
    - The experiments are repeated on different GPUs to see if the optimisations are effective on different architectures.
6. **Development Life Cycle:** A life cycle suggested by NVIDIA in their toolkit documentation (NVIDIA 2019a) is used to develop the parallel module. The life cycle consists of four iterative activities: assess, parallelise, optimise, and deploy.
7. **Analysis and Discussion:**
    1. The results of profiling are presented, analysed, and discussed.
    2. The effect of parallel computing is analysed and discussed in terms of different performance models.
    3. The effect of different bottlenecks is studied.
    4. The effectiveness of algorithmic optimisation is analysed in

light of the developed profiling tool.

5. After each experiment, the parallel program is assessed, optimised, and compared with previous parallel and sequential implementations.

3.3. Data

The purpose of this section is to describe the data sources that are used in this study. As stated previously, the research depends on two main data sources. The first is extracted from the CMS experiments 2018 data and it is used as an input to the parallel program that is developed. The second source of data is generated by existing profiling tools or profiling tools created as a mean to conduct this study.

3.3.1. Physics Data

The physics dataset is generated using a Monte Carlo simulation that corresponds with the data which is collected by the CMS experiment during Phase 2 (CMS Collaboration 2019). It was released as part of the 2018 open data and it contains 20,000 events data that are 641 GB in size. The dataset consists of files, each file represent one event. Each line in the file represent one particle. Events are representations of collisions in the detector. The events consist of several particles, which are described by their x, y, z coordinates. In addition, each particle has its measured energy deposit reported. Listing (3.1) shows a sample of an event data file. The columns in the listing represent the data x, y, z coordinates, and the last one is the Energy deposit (E).

```
2.0746400356 1.0617550611 -0.1100711301 2.3331460953
0.7080608606 0.4309902787 -0.0106400996 0.8289849162
-0.2444184721 0.7751963735 -0.0353877954 0.8135859370
0.6506414413 1.0213022232 -0.0527213514 1.2120941877
-0.5917757750 0.6458099484 -0.0381360054 0.8767687678
```

Listing (3.1): Physics data sample.

The data represent proton-proton collisions, these events can be up to 2000 particles in size. The FastJet software package deals with events that are more than 10,000 particles in size. To generate events with sizes larger than 2000, a methodology of combining events with each other is used. The methodology simply combines unique events of various sizes to generate larger events. This is used to make sure that the algorithm is profiled properly under different work loads. The combination methodology is taken from the FastJet package (Cacciari, Salam, and Soyez 2012).

3.3.2. Profiling Data

The second source of data for this project is generated from profiling tools. Two main tools are used to profile the developed program:

1. **Perfworks:** The main tool used is the NVIDIA Perfworks C++ API. The API provides many capabilities that can be used to benchmark and monitor applications on NVIDIA GPUs. Since it is not publicly available, it is used through some other wrapper applications such as the NVIDIA Nsight Compute.
2. **Call Graph Profiling:** The NVIDIA profiler provides many metrics that can be used to benchmark a GPU program. However, it does not provide call graph profiling capabilities (NVIDIA 2019b).

Thus, a profiling tool has to be implemented based on the call graph profiling technique. The purpose of this tool is to bridge the gap and provide extra profiling capabilities that can be especially useful for algorithmic optimisations.

3.4. Benchmarks

In order to measure the effectiveness of a parallel program a set of metrics are needed. There are many metrics that can be used to benchmark the performance of a parallel program. This project focuses on two benchmarks: Speedup and Throughput.

Speedup: The improvement in execution speed between two programs used to process the same data (Lee et al. 2010). It is a fraction or percentage that shows the relative improvement in performance.

Throughput: The number of units of data processed in a unit of time (Lee et al. 2010). In this case, it is the number of events processed in a unit of time which represents the rate of events processed.

3.5. Parallel Computing Limits

Throughput and Speedup can be predicted before the implementation of the parallel program. Programmers can use Amdahl's law and Gustafson's law to get a glimpse of the effectiveness of parallel computing on the program. These two laws set up a theoretical limit to how much can a program benefit by parallelisation.

3.5.1. Amdahl's Law

If a program consists of a sequential part (a part that can be only processed sequentially) and a parallel part (a part that can be processed in parallel). Assuming it is possible to throw an infinite number of processors on the parallel part to reduce its execution time to almost 0 seconds. The maximum speedup that a programmer can reach with infinite number of processors is 1 divided by the fraction of the sequential part (Amdahl 1967).

Therefore, if a program consists of 50% sequential part, and a 50% parallel part the maximum theoretical speed up is 1 / 0.5 which is 2 times faster execution. Figure (3.1) shows an illustration of Amdahl's law.

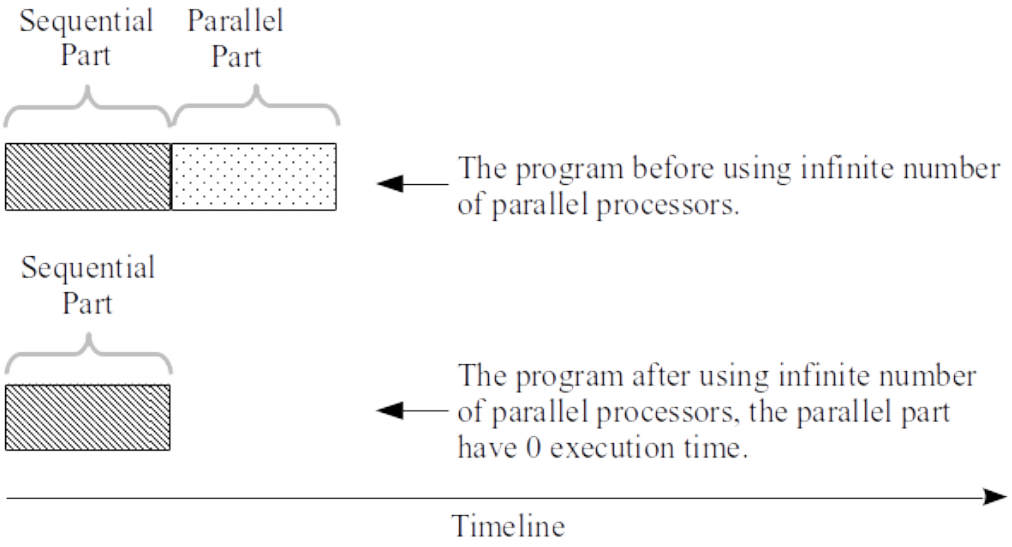

Figure (3.1): An illustration of Amdahl's law.

3.5.2. Gustafson's Law

According to Amdahl's law, the maximum benefit of using parallel computing is bound by the sequential part of the program. Gustafson's law

is used to overcome this limit by assuming that the parallel part of the program is not necessarily fixed in size and it can be changed dynamically by redesigning the program or using larger datasets (Gustafson 1995).

Assuming it is possible to increase the size of the parallel part of the program to be 75% of the execution time. A parallel program theoretically can reach 4 times speedup in execution time according to Amdahl's law. Figure (3.2) shows an illustration of Gustafson's law.

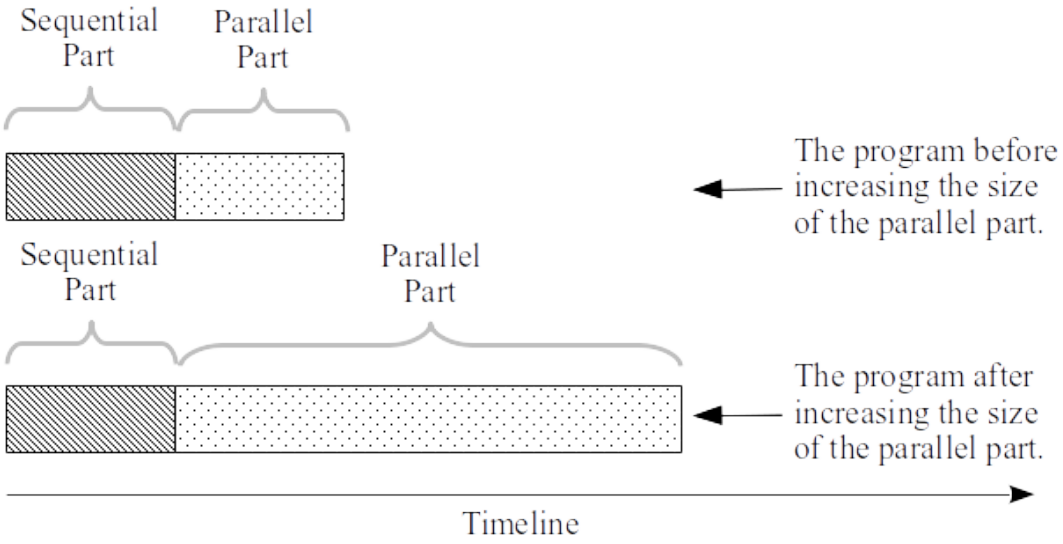

Figure (3.2): An illustration of Gustafson's law.

In parallel computing, both Amdahl's law and Gustafson's laws play major role in program design. The developers should aim to reduce the parallel part execution time alongside increasing the size of the parallel problem. This way, the theoretical limit imposed by Amdahl's law can be overcome, and maximum benefit of parallel computing can be realised.

Amdahl's and Gustafson's laws impose theoretical limits on parallel programs. In practice, the speedup that can be achieved might be more or

less than the limits predicted if other aspects were taken into account. For example, in the case of super linear speedup, higher speedup can be reached due to the instructions being in the cache most of the time (Jiang et al. 2019).

3.6. FastJet Software Package

FastJet is an open source software package used to find jets in proton-proton or electron-positron collisions (Cacciari, Salam, and Soyez 2012). It provides highly optimised native implementations of several sequential recombination and clustering algorithms.

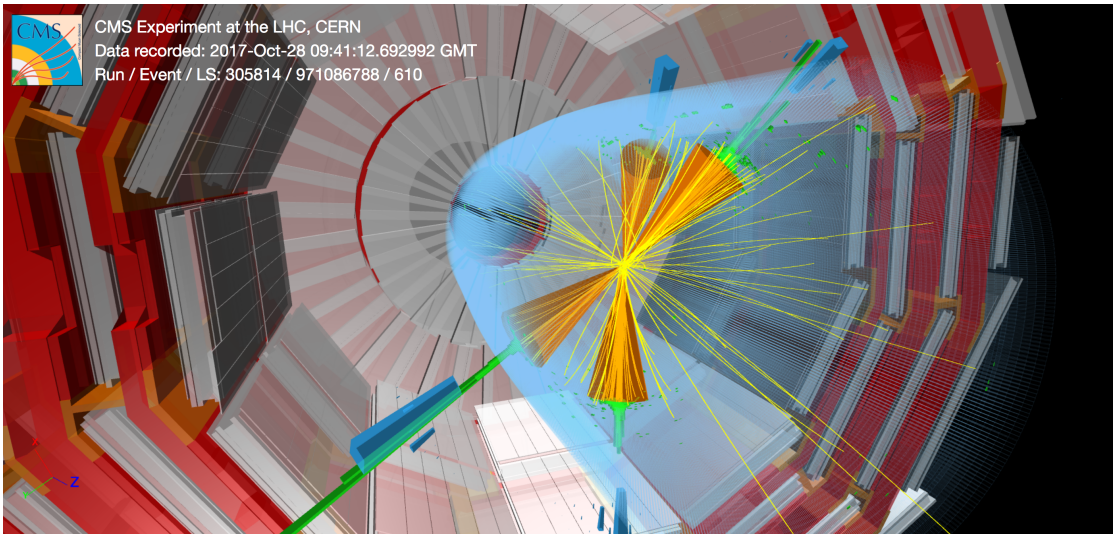

Figure (3.3): Display of two pairs of jets from the CMS experiment (Mc Cauley 2019).

Jets are collection of hadrons that travel in narrow cone structures in the same direction as a result of particle collisions. The jets are measured in particle detectors to determine the properties of original elementary particles. Figure (3.3) shows an example of two pairs of jets from the CMS experiment.

FastJet provides an interface for physicists to use different clustering algorithms. The physicists have to provide a set of meta data that specify the algorithm, hyper parameters, and implementation. The program runs the clustering algorithm and report back the jets found as x, y, z, and Transverse Momentum (pT).

3.6.1. Clustering Algorithm

In addition to the input particles, the clustering algorithm requires two main parameters. The first is the radius of the cluster (R in Equation (1)). The second parameter is the minimum transverse momentum (pT). The pT plays the role of a threshold, every jet that has pT less than the threshold is discarded from the output.

The algorithm performs the following steps (refer to code listing (3.2) for a pseudo code):

1. Calculate the distance (using a distance algorithm) between each particle and the other particles, and between each particle and the jet beam.
2. Find the minimum distance.
3. Recombine:
    (a) If the minimum distance is between two particles:
        i. combine the two particles into one particle (using a recombination scheme).
        ii. Remove the two recombined particles from the list.
        iii. Add the new particle to the list.
    (b) If the minimum distance is between a particle and the beam:

i. Mark the particle as a jet.

ii. Remove it from the list of particles.

4. Repeat the steps until the list is empty.

```
# Repeat untill all the particles are combined
for i = 0 to number_particles
  min_d = inf
  min_p_i = -1
  min_p_j = -1

  # Find the minimum
  for p_i in particles
    for p_j in particles
      d = 0

      if p_i == p_j
        d = compute_iB_distance(p_i)
      else
        d = compute_ij_distance(p_i, p_j)

      if d < min_d
        min_d = d
        min_p_i = p_i
        min_p_j = p_j

  # Combine
  if min_p_i == min_p_j
    combine_iB(min_p_i)
  else
    combine_ij(min_p_i, min_p_j)
```

Listing (3.2): Clustering algorithm pseudo code.

The distance between particles i and j can be calculated using (Cacciari and Salam 2006):

$$d_{ij} = min(d_{iB}, d_{jB}) \frac{\Delta R_{ij}^2}{R^2} \tag{1}$$

Where:

$$d_{iB} = \frac{1}{k_{ti}^{2p}} \tag{2}$$

$$k_{ti}^{2p} = x_i^2 + y_i^2 \tag{3}$$

$$\Delta R_{ij}^2 = (\eta_i - \eta_j)^2 + (\phi_i - \phi_j)^2 \tag{4}$$

$$\eta_i = \frac{0.5 * \log\left(d_{ib} + max(0, (E_i + z_i) * (E_i - z_i) - d_{iB})\right)}{(E_i + |z_i|)^2} \tag{5}$$

$$\phi_i = \text{atan2}(y_i, x_i) \tag{6}$$

Note that some calculations details and special cases where left out for simplicity.

The FastJet package provides many particles recombination schemes and allow the users to implement their own schemes through plugins, this project considers the basic default scheme only which is a simple summation of the particles properties. Combining particles p_i and p_j results in the following particle p_k:

$$x_k = x_i + x_j \tag{7}$$

$$y_k = y_i + y_j \tag{8}$$

$$z_k = z_i + z_j \tag{9}$$

$$E_k = E_i + E_j \tag{10}$$

3.6.2. Sequential Clustering Implementation

The FastJet package provide's multiple sequential implementations for the clustering algorithm. All the algorithms share the same steps listed in the previous subsection, they differ only in the second step, finding the minimum distance. This project focuses on three main implementations (Cacciari and Salam 2006):

1. N^3Dumb: This implementation searches for the nearest neighbour for each particle using linear search algorithm ($O(N)$), applying it for all the particles results in $O(N^2)$ time complexity. Looping on all the list requires $O(N)$. Hence, the total is $O(N^3)$.
2. N^2Tiled: This method implements a tiling algorithm in order to find the nearest neighbours. The tiling algorithm have a time complexity of $O(N)$. Looping on all the particles results in a total of $O(N^2)$ time complexity.
3. $N \ln N$: This method uses triangulation techniques (Voronoi Diagrams) to find nearest neighbours. The diagram construction needs $O(N \ln N)$ time. The diagram construction is done once then used $O(N)$ times to find the nearest neighbours, therefore, the total time required becomes $O(N \ln N)$.

Where N is the number of particles in each event.

3.6.3. Parallel Clustering Algorithm

The sequential algorithm mentioned in sub-section 3.6.2 can be summarised in the following steps:

1. Calculate distances.
2. Find particles nearest neighbours.
3. Find minimum distance.
4. Recombine
5. Repeat.

Using an iterative development life cycle, each step in the algorithm can be rewritten to utilise parallel computing. This project takes an inside-out approach for the development of the parallel module. Therefore, the first part that is experimented with is the inner most task which is the distance calculation. Next, is finding a parallel implementation for the nearest neighbour problem. The final part, is finding the minimum distance between all the particle pairs.

The recombination step can also be parallelised. However, this requires changing the algorithm fundamentally in a way that it is not testable against the sequential implementation for correctness. This requires implementation and proof of correctness which is out of the scope of this project.

As mentioned in the previous chapter, parallel processors benefit from increasing the throughput of the problem rather than increasing the processing speed of one event. Therefore, this project works on implementing a throughput-oriented implementation of the algorithm that makes it possible to process multiple events at the same time.

3.7. Development Life Cycle

This project follows an iterative development life cycle proposed by NVIDIA as a best practice (NVIDIA 2019a). The life cycle consists of four main activities: assessment, parallelisation, optimisation, and deployment (see Figure (3.4)).

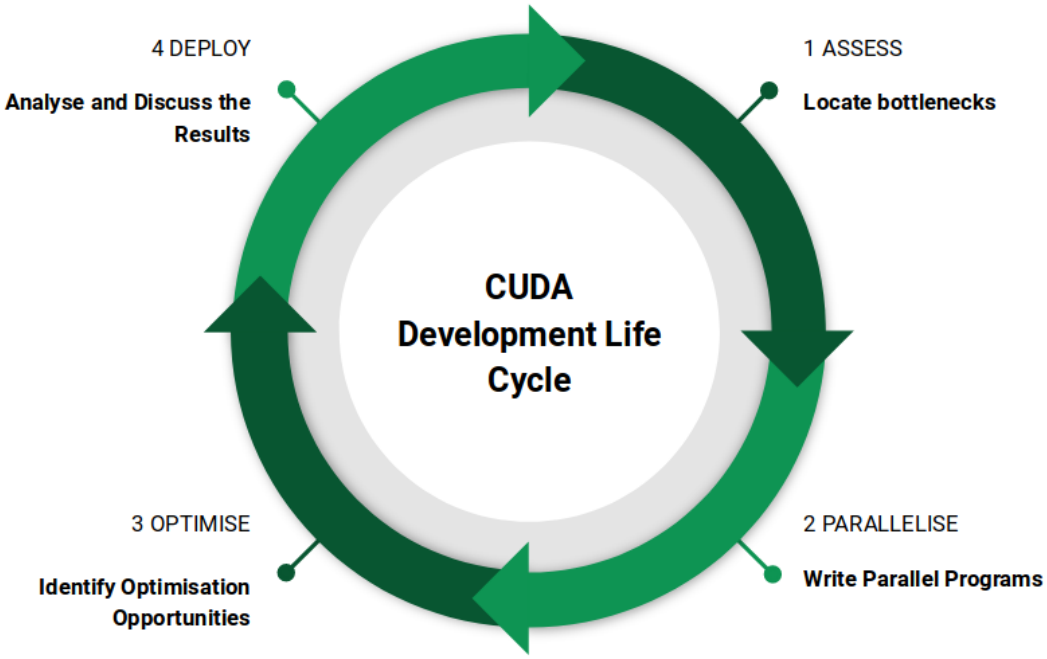

Figure (3.4): CUDA development life cycle.

3.7.1. Assess

In this activity, the developer assesses the application to locate the parts that consume the majority of the execution time. Then investigate the feasibility of GPU acceleration for the problem at hand. This part was done in the previous subsection and is repeated after each iteration of the development.

3.7.2. Parallelise

After identifying the execution bottlenecks of the program, the developer proceeds to writing the parallel program. This part consists of writing CUDA kernels that produces the same results as the original program while taking advantage of parallel computing.

3.7.3. Optimise

After writing the parallel programs, it is essential to make sure that they take full advantage of the parallel processor. In this part, the developer needs to identify optimisation opportunities. These opportunities can be easy to implement as fixes to the current code or they might need further assessment which is done in the next step.

3.7.4. Deploy

Since this is a research project, the developed modules are not deployed to the production environment. Therefore, this step is re-purposed to analyse and discuss the results of the built program.

3.8. Profiling Tools

The main objective of this research is to use profiling tools in order to achieve better performance. Different methods and profiling tools were discussed in chapter 2, this research focuses on two main profiling tools: NVIDIA Nsight Compute and the call graph profiler developed for this study.

3.8.1. NVIDIA Nsight Compute

Based on Perfworks, NVIDIA implements the Nsight Compute profiling tool. The tool is used to profile programs running on NVIDIA manufactured GPUs. It can be used from the command line or from the Graphical User Interface (GUI) applications provided by NVIDIA.

The following is a summary of the data that can be extracted using the Nsight Compute program directly or using a GUI application that utilises it:

1. **Duration:** The time the application or a single kernel took to finish execution.
2. **Utilisation:** How much of the GPU hardware is used during the execution. This gives information about utilisation of both Streaming Multiprocessors (SM) (Computing Units) and Memory.
3. **Stall Statistics:** Statistics about the execution pauses (stalls) due to memory operations, synchronisation, or other reasons.
4. **Instructions Statistics:** Instructions counters, how many times each instruction is executed.

3.8.2. Parallel Call Graph Profiling Tool

Call graph profiling was introduced in the previous chapter as a program that can be used to produce statistics about the execution of a program functions or branches. Using some of the functions provided by the CUDA framework such as `clock()` and `printf()` it is possible to build a call graph profiling tool that can provide useful insight.

These two functions can be used to implement a Data Generator that generates profiling data during the execution of the parallel program. Then it is possible to implement a visualiser for the data which takes the profiling data as an input and helps in querying and reasoning about the data. The data generator and the visualiser are independent of each other in terms of implementation.

Data Generator: A C++ helper class is proposed as a profiling data generator. It is implemented as a wrapper for the clock and print functionality provided by the CUDA framework.

1. `clock()`: The clock function allows the developer to find out how many clock cycles it takes an instruction to complete. By calling the function before the instruction or set of instructions that needs to be inspected, it is possible to find out the elapsed clock cycles.
2. `printf()`: The ability to print messages from inside kernels while they are being executed in the GPU helps in mapping the execution paths that the program takes.

Visualiser: The visualiser can be any program that can receive a textual data as input. Any data manipulation program such as database clients and worksheets applications can be used to analyse the data produced by the data generator. However, for larger programs it is more efficient to write scripts that does the visualisation part.

Python along side some data manipulation and visualisation libraries, was used to write visualisation scripts for this project. These visualisation tools provide statistics about the branches and function calls in the parallel

program. These visualisations provide an insight about the behaviour of the kernel during run time.

3.9. Experimental Framework

This section defines the experimental framework followed in this project. Starting with the physics data as the input of the parallel program. Then the parallel program implementation framework is specified starting by the development language and some assumptions regarding the development process. After that, the profiling process is discussed in terms of timing tools, data collection procedures, and assumptions. The bottleneck and optimisation selection are listed after that followed by the definitions of the used benchmarks. Finally the analysis and presentation of the results are discussed.

3.9.1. Physics Dataset Selection

As mentioned in chapter 3, the physics data is from a Monte Carlo simulation that represent how the data is going to be during Phase 2. The largest event in the dataset contains ~2000 particles. Therefore, multiple events were merged to generate larger events. The method of merging the events was taken from the FastJet package itself.

The generated events start from 1000 particles in size. Then the event size is increased by ~1000 particles for each larger event. For the GPU experiments a set of 10 events was generated. Meaning the sizes of the Events increases from ~1000 to ~10,000 particles. For the CPU experiments, events up to 20,000 particles were generated. The FastJet algorithms show different performance characteristics after the 10,000-

particles mark while the GPU implementations does not show this change in behaviour.

For the throughput experiments, a single event is used multiple times to ensure proper comparison between the different algorithms.

3.9.2. Implementation

To develop the parallel program, CUDA framework was used through the C++ programming language. The CUDA framework supports only the C++14standard, hence all the C++ code that was written followed this standard.

During the implementation, it was made sure that the parallel algorithms generates the correct final output and the output could be traced with the sequential implementation in every step of the algorithm. In addition, all the data types and precision constraints imposed by the sequential program were maintained in the implementation unless it was specified for the sake of experimentation.

i. Variations of Parallel Algorithms

Two parallel algorithms were developed for this project. The algorithms differ in the methods used to find the particles nearest neighbours and finding the global minimum distance because they are the most intensive computing tasks in the original algorithm.

1. **Tri-Matrix Algorithm:** This algorithm puts the distances between the particles in a tri-matrix and then performs a reduction operation in-order to find the minimum distance.

2. **Grid Algorithm:** This algorithm models the trigger surface as a grid and distributes the hits data on the grid by their coordinates. Then it uses the grid to find the nearest neighbours of the particles.

ii. Optimisation

Optimising the algorithm is done during the implementation activity in the experimental framework. After each development iteration, profiling and benchmarking is done. A bottleneck is selected with the help of the profiling data. The bottleneck is analysed and experiments to optimise it are carried. These experiments are what is described and reported in this project in the Experiments section (section 4.3).

iii. Code Compilation

The code is compiled multiple times, each time specifying different GPU target architecture. This means that multiple versions of the program were generated in-order to make sure that maximum compatibility with the target hardware. The compilation details are listed in the Experimental Environment section (see section 4.2).

3.9.3. Profiling

The profiling process consists of four parts:

i. Timing

This project compares a CPU version against a GPU version of the problem. Thus, different tools to record the time on the CPU and on the GPU is used. On the CPU the Chrono class from the standard library

for time taking as shown in Listing (3.3) is used. On the GPU the time measurement API provided by the CUDA framework as shown in Listing (3.4) is used.

```cpp
// Record the starting time
auto start = std::chrono::high_resolution_clock::now();
// Run the clustring
cluster();
// Record the end time
auto end = std::chrono::high_resolution_clock::now();
// Compute the elapsed time
std::chrono::duration<double, std::milli> elapsed = end - start;
```

Listing (3.3): C++ timing example.

```cpp
// Define and create events for the API to track
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
// Record the starting time
cudaEventRecord(start);
// Launch the kernel
clustering_kernel<<<1,256>>>();
// Record the end time
cudaEventRecord(stop);
// Since the kernel launch is asynchronous by default
// Make sure that the kernel finished the job and returned
cudaEventSynchronize(stop);
// Compute the elapsed time
float milliseconds;
cudaEventElapsedTime(&milliseconds, start, stop);
```

Listing (3.4): CUDA timing example.

ii. Data Collection

All the tools used in the project to collect data are specified in the Experimental Environment section (section 4.2) along their version numbers. To insure consistency in the collected data, no major process were running on the same hardware during the data collection except the process needed by the Operating System (OS). In addition, each experiment was repeated 10 times in-order to have an insight in the performance fluctuation. The average of the runs is reported as the final result.

iii. Call Graph Profiling

Whenever more precise or detailed information about the behaviour of the algorithm is needed, call graph profiling is used. Call graph profiling includes inserting `printf()` and `clock()` statements which affects the timing results. Therefore, call graph profiling is disabled during timing. The `printf()` function is limited by the default output buffer size. Thus, `cudaDeviceSetLimit(cudaLimitPrintfFifoSize, 10 * 1024 * 1024)` is used to increase the limit.

iv. Call Graph Profiling Aggregation and Visualisation

The call graph data in a multi-threaded setting can be overwhelming. To make it useful, visualisation scripts are created to aggregate the data and visualise it in a useful manner. All the scripts are written in the Python programming language.

Figure (3.5) shows a summary of the dynamic profiling process used in this project.

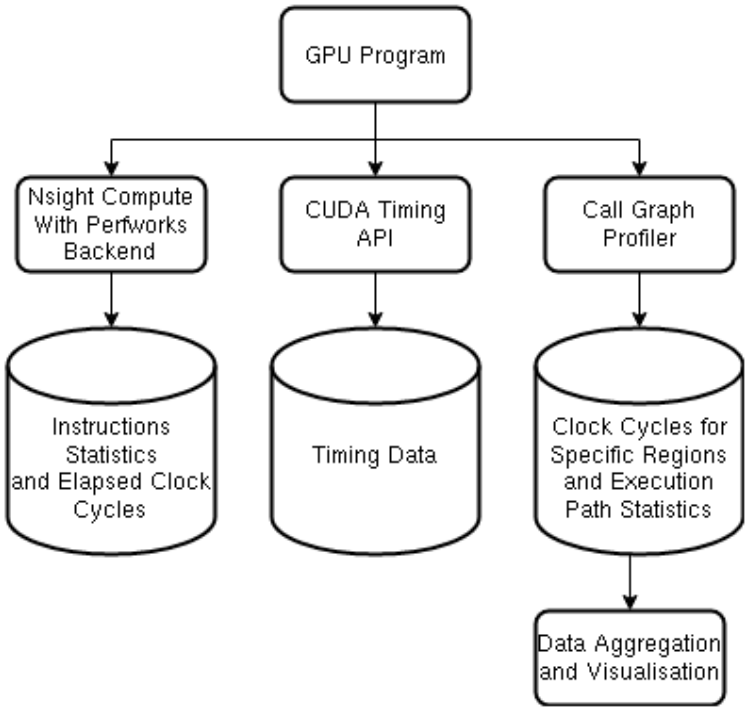

Figure (3.5): An illustration of the dynamic profiling process used in this project.

3.9.4. Benchmarks

To compare the CPU and GPU implementations of the algorithm and the different GPU implementations, the run time of each program is measured on the same input. The run time can be used to derive more representative measurements. Such as:

1. **Speedup**: It is the fraction of speed gained or lost after a program modification (Lee et al. 2010). If the run time of program 1 is T_1 and the run time of program 2 is T_2 and the programs run on the same workload, the speedup is $S = \frac{T_1}{T_2}$. If $S > 1$, then there is an S times improvement in run time.
2. **Throughput:** The throughput is the rate of data processed in a unit

of time (Lee et al. 2010). If N is the number of events processed by the program and T is run time of the program. The throughput is $Th = \frac{N}{T}$. With the assumption that all the events are of uniform size. This assumption is important to make this measurement meaningful.

In addition, different memory and computing resource utilisation benchmarks are used. These benchmarks are taken by querying performance counters through the Nsight Compute profile.

3.9.5. Analysis and Presentation

The different implementations are analysed and compared in terms of speedup and throughput (Lee et al. 2010). By visualising the performance metrics of the different implementations, it is possible to draw conclusions about the effectiveness of each implementation or optimisation. In addition, other results generated from the call graph profiler are presented in order to better understand the performance.

Figure (3.6) shows a summary of the experimental framework described earlier.

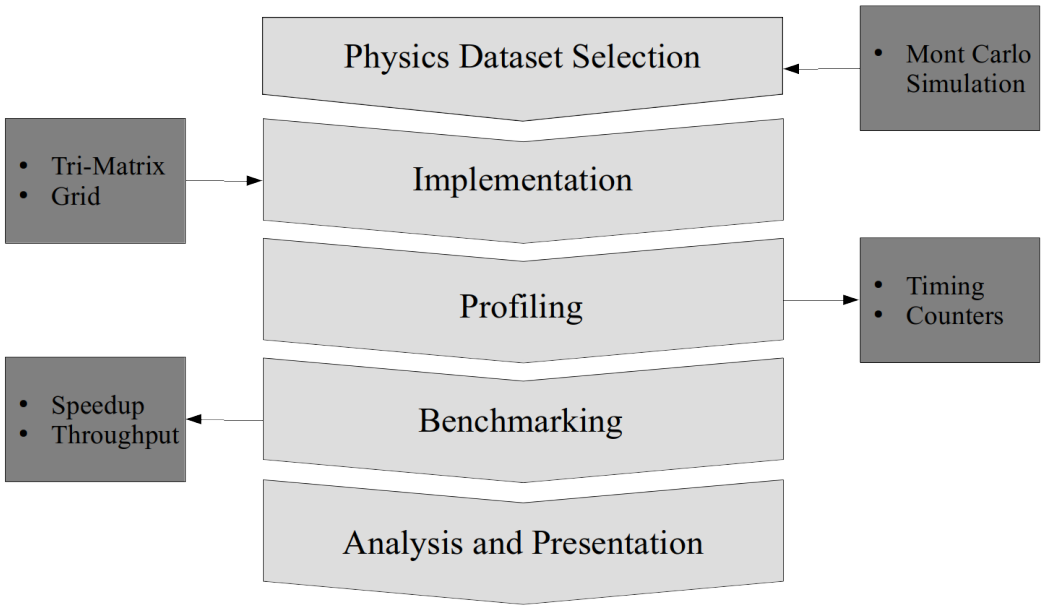

Figure (3.6): The experimental framework summarised.

3.10. Conclusion

This chapter started by presenting the Methodology behind this study. Section 3.2 presented the stages of the research design. Those stages are selected as the bases of conducting this quantitative research.

After presenting the methodology stages, the chapter presents the stages in details. Starting with the data used in the study, the sources of data are listed and described. The data comes from the physics experiment as an input to the parallel program, then the profiling data is generated from the parallel programs in-order to study them.

The next part of the chapter lists different benchmarks used in this project to measure the performance of the parallel programs created. The speedup and throughput benchmarks were defined this part of the chapter.

In-order to gain a better understanding of the parallel programs performance, a methodology to predict parallel computing programs limits was discussed in section 3.5. In practice those limits might not hold, however they provide a useful insight about the behaviour of parallel programs.

Section 3.6 of the chapter presents the software module that is used as the subject of this study. After that, the selected development life cycle for this study is described.

Then, the Experimental Framework that was followed in-order to perform the experiments is described. The framework consisted of data and implementations description. In addition, the performance metrics used in the experiments and how they are compared and analysed was illustrated.

The next chapter presents the results of conducting this study alongside a thorough discussion.

Chapter Four
Results and Analysis

4.1. Introduction

This chapter presents the core of the experiments carried in this study followed by the results and discussion. It starts by explaining the experimental framework in terms of the hardware, software, and configurations used followed by the definition of the experiments. This section describes each experiment and lists the goals of performing the experiment. The final section in the chapter presents and discusses the results from each experiment.

4.2. Experimental Environment

This section presents the environment which is used to carry the experiments. The environment consists of multiple hardware and software components. All experiments are done under the specified hardware specifications and software versions.

4.2.1. Hardware

Two computing nodes equipped with GPUs are used in the experiments. Each computing node has different type of GPUs and CPUs. The first node (NODE A) is used for the CPU version of the program and the GPU versions. The second node (NODE B) is used only for the GPU versions of the program.

Table (4.1) lists the specifications of the compute nodes A and B. It

is worth noticing while each node contains multiple GPUs and CPUs, the programs are tested by running them on a single GPU and CPU. Distributing the work on multiple GPUs is not in the scope of the study.

Table (4.1): Computing nodes general hardware specifications.

Specificatioon	NODE A	NODE B
CPUs	2x Intel Xeon Silver 4114 2.20 GHz	2x Intel Xeon E5-2650 v4 2.20 GHz
RAM	92 GB	251 GB
Hard Disk	894.3 GB SSD	447.1 GB SSD
GPUs	1x Tesla V100 1x Tesla T4	8x GeForce GTX 1080 Ti

Table (4.2) lists all the GPUs that were used in this study. Each GPU is intended for different type of tasks. The V100 is top of the tier scientific computing GPU, therefore it has the most computing power. The T4 is also a scientific GPU, however it is designed to be small in terms of form factor and it consumes much less power (wattage) than the V100 (Jia, Maggioni, Smith, et al. 2019). The GeForce GPU is a normal consumer gaming GPUs. The intention of this mix is to have different variety of GPUs to observe how our program behave under different GPU types.

Table (4.2): GPUs specifications comparison.

Specification	Tesla V100	Tesla T4	GeForce GTX 1080 Ti
Multiprocessors (MP)	80	40	28
CUDA Cores / MP	64	64	128
Threads / MP	2048	1024	2048
Clock Speed	1.38 GHz	1.59 GHz	1.58 GHz
VRAM	32.51 GB	15.11 GB	11.178 GB
Capability Number	7.0	7.5	6.1
Driver Version	10.1	10.1	10.1
Runtime Version	10.1	10.1	10.1

Table (4.3) shows the different CPUs in the computing nodes A and B. Only the "Intel Xeon Silver 4114" CPU from computing node A is used to test the CPU version of FastJet. In addition, the FastJet software package current implementation runs only on a single thread. Therefore the multiple cores of the CPU are not used.

Table (4.3): CPUs specifications comparison.

Specification	Intel Xeon Silver 4114	Intel Xeon E5-2650
Cores	10	12
Hardware Threads / Core	2	2
Clock Speed	2.2 GHz	2.2 GHz
Architecture	x86_64	x86_64

4.2.2. Software

Table (4.4) shows the software packages that are used in the experiments. All the software packages were compiled and configured for the architecture that they were used on.

Table (4.4): Software packages used in the experiment.

Package	Version
FastJet	3.3.2
OS	CentOS Linux release 7.7.1908
CUDA	10.1
GCC Compiler	8.3.1 20190225
Nvcc Compiler	10.1.243
Nsight Compute	2019.4.0

4.2.3. Compilation Configuration

For every version created of the parallel program, it is necessary to use consistent and optimal compiler configurations. The NVIDIA compiler (nvcc) was used to compile all the programs created during the study. The compiler provides a set of flags to target certain compile time optimisations for different GPU architectures. To make sure the compiled programs run smoothly as intended on all the GPUs, nvcc compiler flags were sat to match each GPU architecture. For example, compiling for the Tesla T4 GPU the `arch=compute_75,code=sm_75` flags were used to tell the compiler which architecture it is compiling for. This has to be done due to some noticeable change in performance when the default flags are used. In addition, the `O2` compiler flag was used for other compilation time code optimisations. This flag allows the compiler to change the code in order to optimise the program performance (GCC Team 2020).

4.3. Experiments

This section provides a description of the multiple experiments carried during this study. The previous section described the environment, while this one lists configurations of the experiment itself. The goal is to make the experiments as reproducible as possible.

The first thing that is mentioned is the input data: is it single or multiple events and the characteristics of the chosen event. The FastJet algorithm requires two hyper parameters: *radius*, and minimum pT. These parameters are fixed at 0.4 and 1.0 respectively throughout the experiments unless otherwise specified. The parameters were selected to match the

configurations used in the CMS experiment (CMS Collaboration 2019).

Furthermore, the main goals of each experiment is specified. This means that each experiment is designed to address one or more of the research questions mentioned earlier in chapter 1 as an experimental goal. The description acts only as a mapping between the experiments and the research questions, however, the effectiveness of the experiment in answering the question is discussed further in the Results and Discussion section and in the Conclusion chapter (chapter 5).

The experiments are arranged into two parts. The first part holds the CPU implementation related experiments. This part is used to set the baseline for the main goal of the experiments which is the parallel computing implementations. The second part is related to the GPU experiments. This part focuses on comparing the baseline implementation with the parallel implementation and improving the parallel implementation after each optimisation iteration.

4.3.1. CPU Experiments

The following experiments are used as sanity check and to set the baseline benchmarks for the CPU algorithm.

i. Reproducing the FastJet Paper Results

In the paper where the FastJet authors introduced their algorithm (Cacciari and Salam 2006), they introduced some benchmarks results that shows how the different algorithms variations (N3Dumb, N2Tiled, NlnN) scale up as the events' sizes increase. In this experiment, the current version of

the FastJet implementation is tested to see if the software still reproduces the same results.

- **Experimental Goal:** Sanity check and setting a baseline, by reproducing the results of the FasJet software package.
- **Related Research Questions:** None. This is just to make sure that the old results match with the current implementation and hardware.

ii. Benchmarking Using Amdahl's Law and Gustafson's Law

Amdahl's and Gustafson's Laws are used to predict the effectiveness of parallelising part of a program (Amdahl 1967; Gustafson 1995). In this experiment a methodology and a model are proposed to apply these laws. A simple program that simulate the running of the parallel part of the program on a GPU while assuming that the parallel part takes no time to execute.

- **Experimental Goal:** Applying the predictive models introduced by Amdahl and Gustafson.
- **Related Research Questions:** Question 1, and 3.

4.3.2. GPU Experiments

This part of the experiments uses the Tri-Matrix implementation on FastJet in order to test different GPUs bottlenecks, how much they are effective, and how to mitigate them.

i. GPU Baseline

This is used to set-up the baseline for the GPU implementations. The

baseline is an attempt to create a correct implementation of the algorithm. The performance is not the goal of this implementation.

- **Experimental Goal:** Creating a baseline program on the GPU and identifying the first optimisation opportunity.
- **Related Research Questions:** Question 1, 2, 4, and 5.

ii. Global Memory vs Shared Memory

In an attempt to reduce the memory access latency, this experiment measures the effect of using shared memory vs global memory.

- **Experimental Goal:** Measure the clock cycles required to access the global memory and the shared memory, and how they affect the GPU implementation.
- **Related Research Questions:** Question 1, 2, 4, and 5.

iii. Interleaved vs Coalesced Memory Access Patterns

The pattern used to access the GPU memory can be a performance limiter. This experiment is designed to compare the Interleaved and the Coalesced memory access patterns. Both patterns are used and compared to measure the performance advantages of memory access pattern optimisations.

- **Experimental Goal:** Compare the interleaved and coalesced memory access patterns and how they impact the performance.
- **Related Research Questions:** Question 1, 2, 4, and 5.

iv. Array of Structures (AoS) vs Structure of Arrays (SoA)

This experiment compares two methods of storing data to the GPU memory. These two methods impact the performance of accessing the GPU memory, therefore, this impact is measured.

- **Experimental Goal:** Compare the AoS and SoA memory storage methods and how they impact the performance.
- **Related Research Questions:** Question 1, 2, 4, and 5.

v. Branch Divergence and Idle Threads

This experiment studies the problem of divergent branches. The divergence of execution branches can lead to idle threads and sequential execution of code on the GPU.

- **Experimental Goal:** Measure the effect of branch divergence, and implement a method to mitigate them.
- **Related Research Questions:** Question 1, 2, 4, and 5.

vi. Algorithmic Optimisation Using a Grid Data Structure

This experiment implements a grid data structure that can be used to store and retrieve the particles' nearest neighbours. The aim of creating such structure is to limit the number of distance computations required to find the nearest neighbours.

- **Experimental Goal:** Testing the effectiveness of a grid data structure in limiting the number of distance computations.
- **Related Research Questions:** Question 1, 2, and 5.

4.4. Results and Discussions

The results and discussion of each experiment are presented in this section. Each experiment is presented in details in a separate subsection. After presenting the experiment in details, the output results of the experiment is presented followed directly in the same sub-section by a discussion of the results.

4.4.1. CPU Experiments

This subsection presents the experiments carried on the CPU version of the FastJet program. Those are small experiments to baseline the work and to make sure the reported results match the experimental results. In addition, those experiments are used to predict the outcome of parallelising the algorithm.

i. Reproducing the FastJet Paper Results

The FastJet software package provides different implementations for the jet clustering algorithm. They all share the same steps as described in Chapter 3. However they differ in the process of finding the particles' nearest neighbours.

N3Dumb: In this implementation, the nearest neighbour is located by exhaustive search. Therefore, taking each particle and searching the list of particles for its nearest neighbour.

N2Tiled: This implementation creates a set of tiles around each particle. The number of tiles by default is nine tiles. The list of tiles is filled for

each particle once, and then updated after each iteration of recombination. Therefore, the program have to search only nine tiles for the nearest neighbour of the particle.

N*ln*N: This implementation generates the Voronoi diagram for the set of particles. Then the diagram is used to query the nearest neighbour of each particle. The diagram can be constructed in *lnN* time.

The FastJet algorithm paper (Cacciari and Salam 2006) shows that the N*ln*N implementation is the best in terms of scaling to large event sizes (see Figure (4.1)). However, for smaller ones, the N2Tiled is most optimal. The paper was published in 2006. Many versions of the FastJet software package has been released since then. In this experiment the FastJet algorithm is benchmarked again to see if the same observations hold.

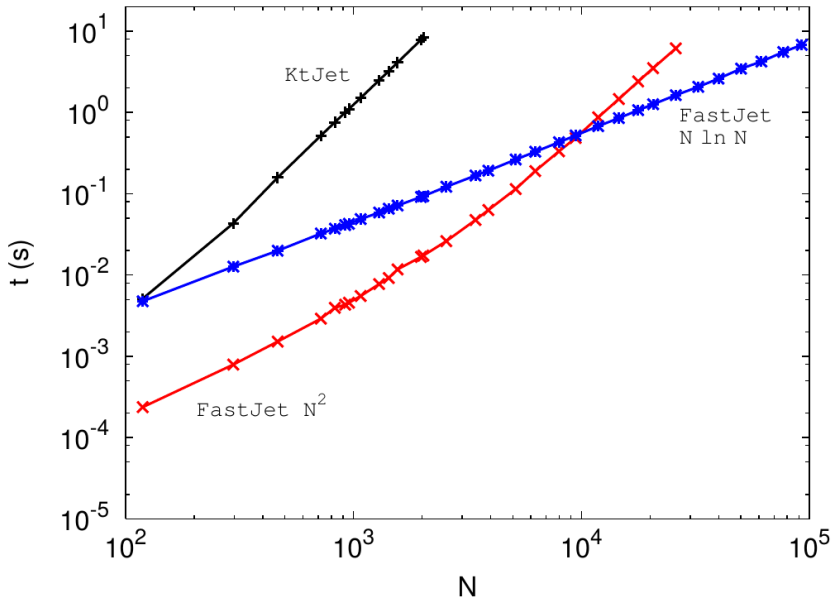

Figure (4.1): FastJet benchmark results (Cacciari and Salam 2006).

To benchmark the software, events were generated with sizes ranging

from 1k to 20k particles. The generation process was described in section 3.9.1. Figure (4.2) shows that the software package still reproduces a similar behaviour for the different implementations. It is worth noting that the smallest event used in reproducing the data was of size 1k particles, unlike the original benchmark which starts with events of size around 100 particles.

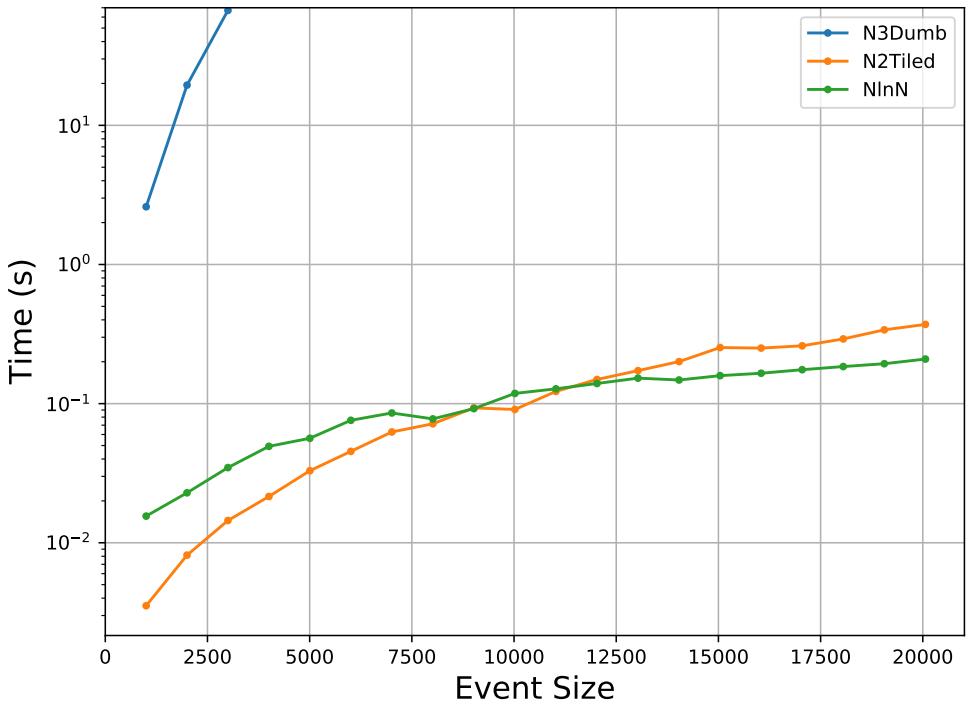

Figure (4.2): FastJet results reproduced.

It is clear that the N3Dumb implementation shots from the beginning to more than 1s run time. While the N2Tiled and the N*ln*N keep the run time well bellow the 1s mark. Furthermore, the N2Tiled implementation still performs worse than the N*ln*N implementation after when the event size is approaches 10k.

The only noticeable difference between the two figures (Figure (4.1) and Figure (4.2)) is the improvement in processing time between the old and the reproduced results. In the new benchmark, the events of size 10k get processed in 0.1s. While in the old figure (4.2), it takes more than 0.5s to process them. This might be due to the difference of hardware used, and the general improvements in the implementations.

Table (4.5) shows the throughput of the different implementations and Figure (4.3) shows a graphical comparison between them. The advantage of the parallel program is showen when it comes to the amount of events that can be processed in a unit of time. For this benchmark, an event of size 1k particles is used for this and the next experiments. The reason an event of fixed size is used is to simplify the computation and comparison between the different implementations. The N2Tiled throughput is used as the baseline to compare the performance of the parallel implementations.

Table (4.5): The throughput of processing an event of size 1k particles using the CPU implementations.

Implemenation	**Throughput (event / second)**
N3Dumb	0.38
N2Tiled	283.43
NlnN	64.31

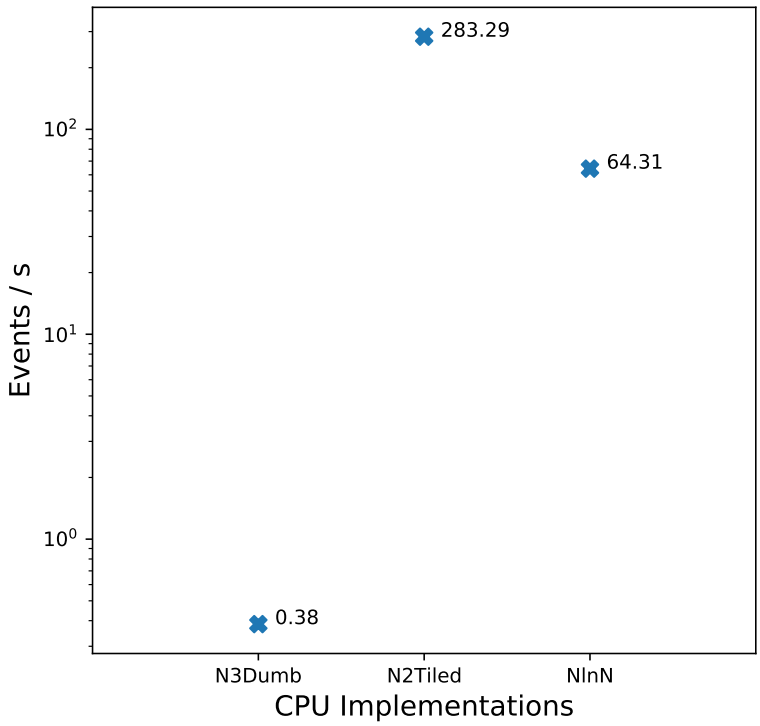

Figure (4.3): A comparison between the throughput of processing an event of size 1k particles using the CPU implementations.

ii. Benchmarking Using Amdahl's Law and Gustafson's Law

This experiment is designed to predict the maximum theoretical speedup that can be reached by applying Amdahl's Law and Gustafson's Law. Since the parallel implementation of the jet clustering algorithm performs the entire clustering operation on the GPU, the only sequential part remaining for the CPU to process is reading the events from the disk and copying them to the GPU and copying the results back from the GPU to the CPU memory.

The process of reading the events from the disk and reporting the results after the clustering is done is timed. The time of reading and reporting is considered the sequential part of the program. It is assumed that the

clustering part is processed by infinite number of parallel processors which makes the processing time go to zero, which is an application of Amdahl's Law.

The number of events submitted to the GPU is 1000 simultaneous events, therefore increasing the size of the parallel problem which is an application of Gustafson's Law. Using this experiment, it is possible to predict the maximum theoretical speedup possible by parallelising the problem. The number of events is chosen so that all the events can fit on the GPU memory at the same time. If the number of events exceeds the GPU RAM size, then the GPU memory bandwith will be a variable that needs to be accounted for. Profiling the GPU memory bandwith is outside the scope of the study.

The experiments data presented in Table (4.6) and Figure (4.4) shows that the theoretical speedup can reach 150 times. An important thing to note is that in reality the speedup is much lower than the reported. For more accurate predictions of the speedup, more factors needs to be considered. One thing to consider is the parallel execution time. The GPUs at hand can only processes a limited number of blocks (events in this case) in parallel. Which means reducing the parallel time to zero is not a practical assumption.

Table (4.6): The speedup of processing 1000 events in parallel assuming the parallel part is processed in no time.

GPU	Predicted Speedup	Predicted Throughput (event / second)
V100	150.26	42587.17
T4	136.69	38742.55
GTX1080	138.59	39280.03

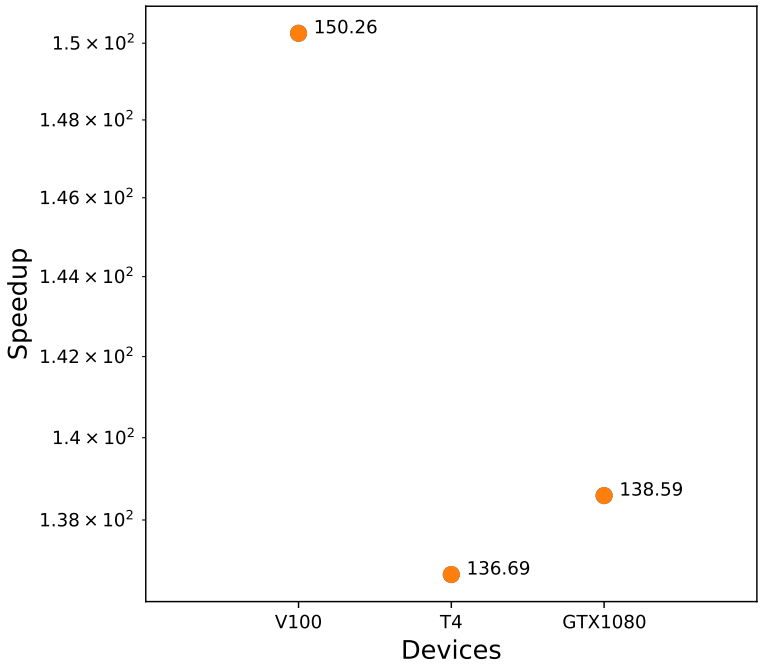

Figure (4.4): A comparison of the speedup of processing 1000 events in parallel assuming the parallel part is processed in no time.

However, this experiment is useful to make sure that the time required to copy the data from the host to the device is, by a comfortable margin, less than the time of processing the set of events sequentially. If the time required to get the data to the GPU is more than the time of sequentially processing them or very close to it, then it might be useless to try to parallelise the problem on the GPUs. Therefore, this exercise is important to test the feasibility of attempting to parallelise the algorithm.

4.4.2. GPU Experiments

This subsection presents the experiments carried on the GPU implementation. Each experiment represents a development cycle. As described in the Research Methodology Chapter (see chapter 3), after each development iteration, the implementations are profiled and compared against the baseline implementation. The experiments are designed to identify performance bottlenecks in parallel programs on GPUs, and how to mitigate them.

i. GPU Baseline

The first implementation of the parallel program uses the parallel reduction pattern (Schmidt et al. 2018) to find the particles nearest neighbours. The distances between the particles is computed in $N * (N+1)/2$ steps representing a Tri-Matrix of distances, where N is the number of particles. While computing the distances, an array of the smallest distances is created. The minimum of the array is found using the reduction process as illustrated in Figure (4.5).

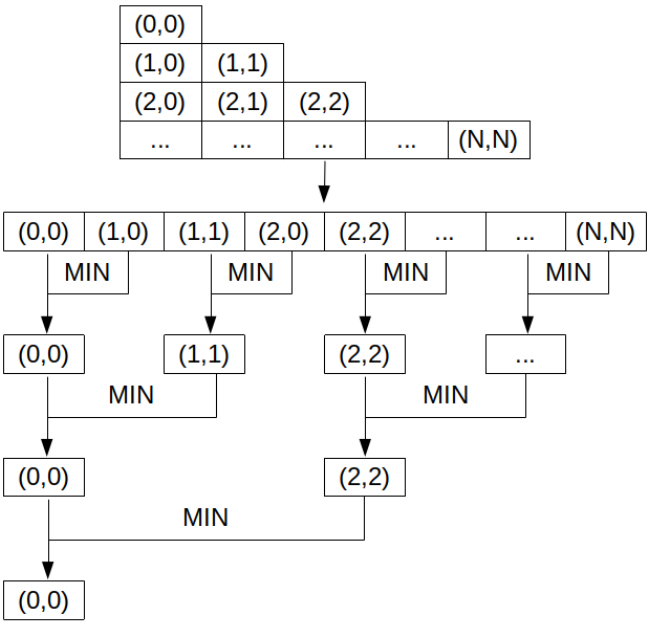

Figure (4.5): Parallel reduction to find the nearest neighbour.

While the distance computation and the nearest neighbour searching is done in parallel on the GPU, Table (4.7) and Figure (4.6) shows that the GPU baseline is not even close to the CPU throughput. This is expected at this stage. The aim of the baseline is to create a correct program which is profiled and optimised in the next development iterations.

Table (4.7): Throughput of the CPU and GPU implementations.

Implementation	Time (s)	Events / s	Speedup (vs CPU)
CPU N2Tiled	3.53	283.43	1.0
GPU V100	15.71	63.67	0.22
GPU T4	31.95	31.30	0.11
GPU GTX1080	42.81	23.36	0.08

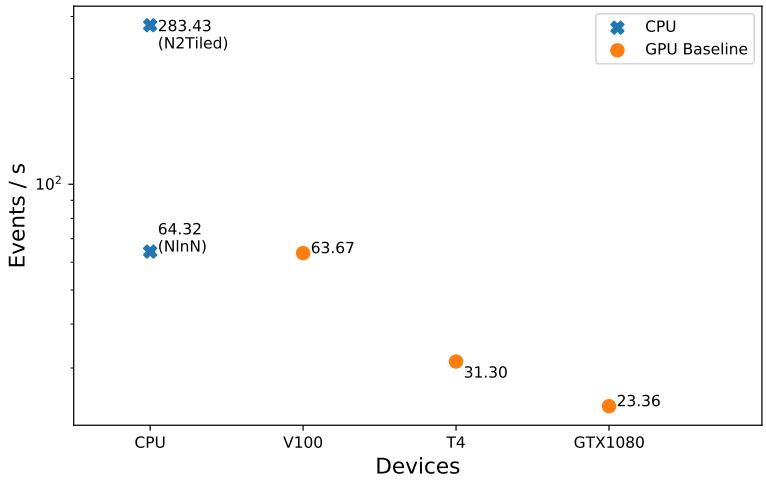

Figure (4.6): CPU vs GPU baseline throughput.

To select a bottleneck to optimise, the profilers were used. Table (4.8) shows that the program executes more memory instructions than double precision computing instructions (FP64). This information is not enough to conclude that the bottleneck is memory operations, because each instruction type takes different number of clock cycles. Hence, the call graph profiler was used to pinpoint the average clock cycles each instruction consumes. The global memory access requires around 70 to 85 clock cycles to be executed as shown in Table (4.8) across the different GPU architectures. Following the methodology suggested by Wong et. al. Wong et al. 2010 and later enhanced for new GPU architectures by Wu et. al. Wu et al. 2019, it is possible to see that the memory operations are responsible for most of the clock cycles in the program. Therefore, the next experiments are used to optimise the memory access operations.

Table (4.8): GPU baseline profiling information.

Metric	Count
Memory Instructions Count	1.85E+08
FP64 Instructions Count	3.62E+07
Avg. Cycles / Global Memory Instruction (V100)	69
Avg. Cycles / Global Memory Instruction (T4)	78
Avg. Cycles / Global Memory Instruction (GTX1080)	84.125
Avg. Cycles / FP64 instruction (V100)	259.75
Avg. Cycles / FP64 instruction (T4)	255.25
Avg. Cycles / FP64 instruction (GTX1080)	295.25

ii. Global Memory vs Shared Memory

The memory hierarchy in GPUs impose an important limitation on parallel programs. Unlike CPUs, The ALUs of the GPU have access to significantly smaller fast access memory caches (Serpa et al. 2019). Therefore, making sure that the required data for processing is present in the caches is mandatory to ensure the maximum performance on GPUs.

In a perfect situation, each thread has its required data in SM registers, which are the fastest memory that can be accessed on a GPU. However, the register sizes are small, therefore in a real world application, threads almost always have to access the data from slower memory resources. Also, in many occasions, GPU threads have to communicate data to other threads, this communication can not be done using the local registers because they can be accessed by the thread itself only.

To allow inter-block communication without paying high performance cost, the shared memory can be used. The shared memory is a special type of memory that can be accessed by the threads in the same GPU

block. While the shared memory is not as fast as the registers' memory, it is much faster than accessing the global memory (X. Mei and X. Chu 2017). Therefore, using the shared memory as a mean of communication between the threads in the same block can be effective in improving the performance.

In this experiment, the effect of using the global memory and the shared memory is studied. The reduction part of the algorithm is moved entirely to the shared memory as shown in Figure (4.7). This should reduce the latency of accessing the memory and it reduces the number of global memory access instructions. As shown in Table (4.8), the global memory access requires at least 69 clock cycles on average.

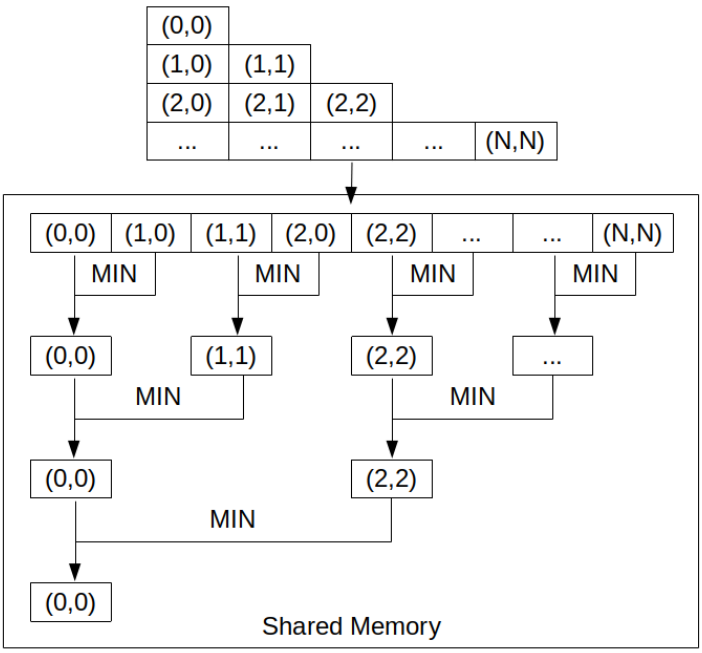

Figure (4.7): Parallel reduction done in shared memory.

After the shared memory optimisation, the CPU version remains faster than the GPU implementation as shown in Table (4.9) and Figure (4.8).

However, the shared memory optimisation is proven effective compared to the baseline GPU implementation as shown in Table (4.9) and Figure (4.8). With almost half the run time on the V100 and the GTX1080. On the other hand, the T4 did not benefit as much as the other GPUs from this optimisation. This is partially because the T4 has a newer architecture than the other cards, therefore it is more optimised to mitigate the problems of the older GPUs architectures which makes the optimisations not as effective as the older architectures (Shahneous Bari et al. 2018). Moreover, the T4 has significantly slower shared memory access as shown Table (4.10).

Table (4.9): Throughput of the GPU implementations after applying the shared memory optimisation.

Implementation	Time (s)	Events / s	Speedup (vs GPU Baseline)	Speedup (vs CPU)
GPU V100	8.72	114.67	1.80	0.40
GPU T4	26.86	37.24	1.19	0.13
GPU GTX1080	26.91	37.16	1.59	0.13

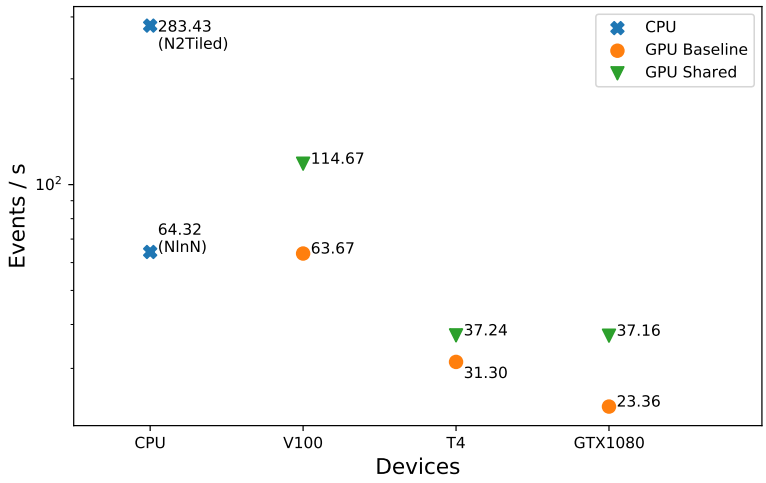

Figure (4.8): GPU shared memory optimisation throughput.

In terms of the instructions, Figure (4.9) shows that the total number of instructions needed to execute the program was reduced significantly.

This is due to the fact that using better memory access patterns allows the GPU to optimise the number of memory access instructions required by the programs (Jia, Maggioni, Smith, et al. 2019; X. Mei and X. Chu 2017). In addition, The number of instructions is lower on the newer GPU architectures. In this experiment, the T4 has the newest architecture, followed by the V100, then the GTX1080. New architectures allow for more compile time optimisations than older architectures (Jia, Maggioni, Smith, et al. 2019).

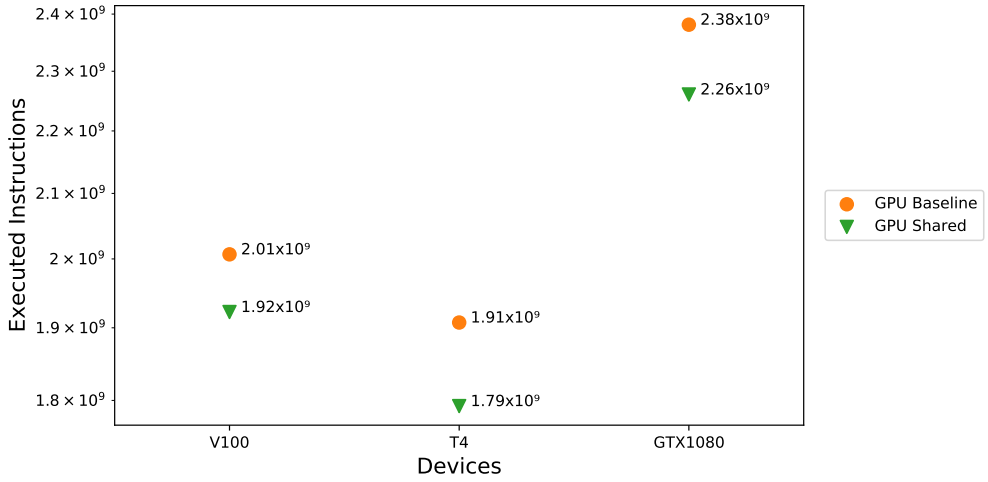

Figure (4.9): GPU optimisations effect on the number of executed instructions.

With the decrease of the global memory instructions, the total number of elapsed clock cycles decreased as well (see Figure (4.10)). The decrease in the total number of clock cycles can be explained by Table (4.10), which shows that the single shared memory instruction is half the cost of a single global memory instruction in the worst case (refer to Table (4.8)). Note that the elapsed clock cycles reading for the GTX1080 baseline is missing in Figure (4.10), because the profiler can not report back very high numbers on older architectures and considers them as infinity (NVIDIA

2019c)

The Volta (V100) and the Pascal (GTX1080) architectures have lower shared memory latency by design than the Turing (T4) architecture which explains the higher shared memory access latency in the T4 GPU (Jia, Maggioni, Smith, et al. 2019).

The reduced number of instructions is due to the shared memory banks implemented in the GPU architectures. The memory banks allow multiple memory access requests to share the same memory access instruction, therefore limiting the required number of memory access instructions. The next experiment discusses the memory banks in more details.

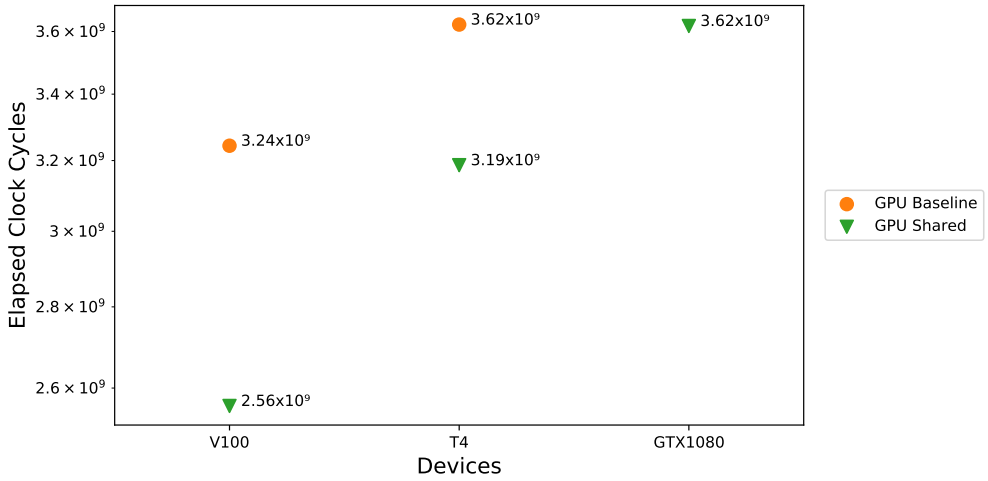

Figure (4.10): GPU optimisations effect on the number of elapsed clock cycles.

Table (4.10): GPU shared memory profiling information.

Metric	Count
Avg. Cycles / Shared Memory Instruction (V100)	17.125
Avg. Cycles / Shared Memory Instruction (T4)	48.625
Avg. Cycles / Shared Memory Instruction (GTX1080)	12

iii. Interleaved vs Coalesced Memory Access Patterns

In each memory access operation, the data is be transferred between the SM and the memory in chunks. Those chunks are called cache lines (NVIDIA 2019a). The size of the cache line differs depending on the GPU architecture. The GPUs used in this study have a cache line of size 128 Bytes. Therefore, if one memory operation requested a 32 double precision values each of size 8 Bytes (total of 256 Bytes). The SM has to hit the memory twice to fetch the whole requested piece of data. Figure (4.11) shows how the memory is accessed using cache lines, where each colour represents a cache line, which represents a single memory access (X. Mei and X. Chu 2017).

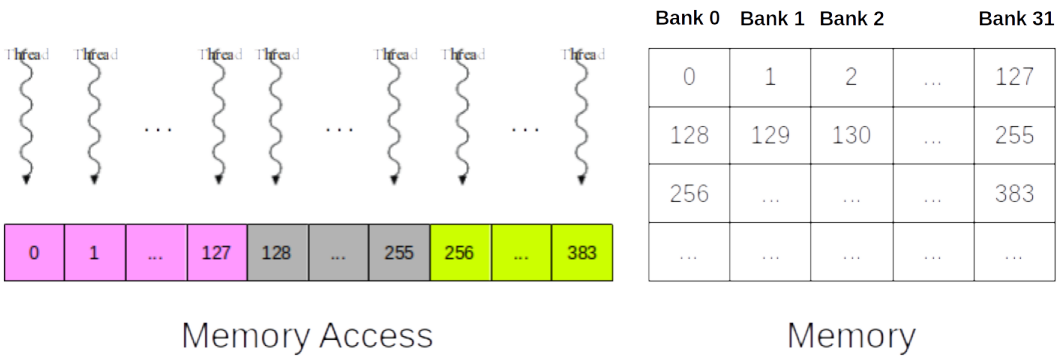

Figure (4.11): An illustration of how the memory is accessed in the threads using cache lines and memory banks.

The NVIDIA GPUs that are used in this experiment organise the threads when they execute them as warps (group of threads). The warp size varies between different GPU architectures. The GPUs used in this experiment have a warp size of 32 threads. This means that each SM issues instructions for each 32 threads (warp) in parallel.

Since the GPU fetches the data from the memory in cache lines and the threads are organised in warps, if the threads in one warp requested data that are adjacent to each other in the memory, the adjacent threads can use the data fetched in a single memory cache hit. This means that if each thread in a warp accessed the data adjacent to the data used by the thread next to it, it reduces the number of memory accesses required to execute an instruction (Schmidt et al. 2018). This memory access pattern is called a Coalesced memory access pattern (Li et al. 2019). Figure (4.12) illustrates the difference between the interleaved access pattern which has been used in the previous experiments and the coalesced access patterns, which is implemented in this experiment.

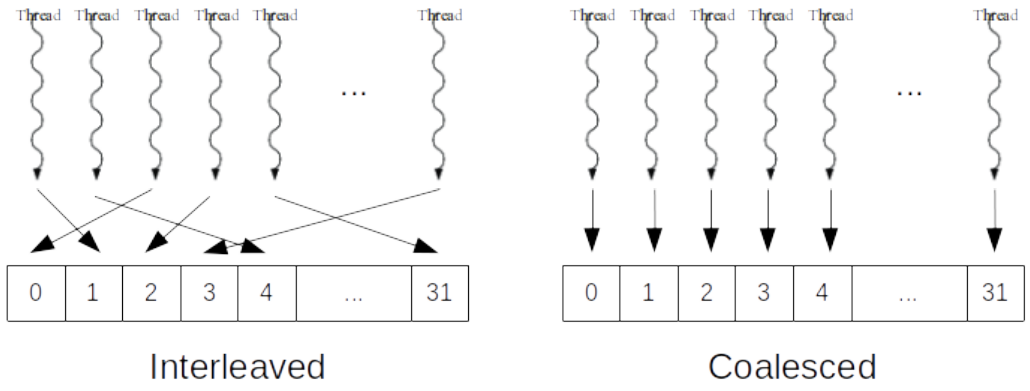

Figure (4.12): An illustration of the difference between interleaved and coalesced memory access patterns.

Another reason why coalesced memory access is effective is, it reduces bank conflicts (Jia, Maggioni, Staiger, et al. 2018). The GPU shared memory is divided into memory banks (see Figure (4.11)). Each word of memory is contained in a bank. The word size and the number of

banks differ from one architecture to the other, for example the Volta architecture (V100) has a 8 Bytes word size and 32 memory banks / SM. If two threads tried to access different words from the same bank, the access to the bank is serialised. This is called a bank conflict. Therefore, accessing the memory in a coalesced pattern ensures that each thread reads a consecutive memory location, which also ensures that the memory bank conflict is kept to a minimum.

To measure the effect of optimising the memory access pattern, the reduction part of the tri-matrix algorithm was re-implemented to use a coalesced memory access pattern as shown in Figure (4.13). This was done by modifying the indices which are used in the loop that accessed the data in shared memory.

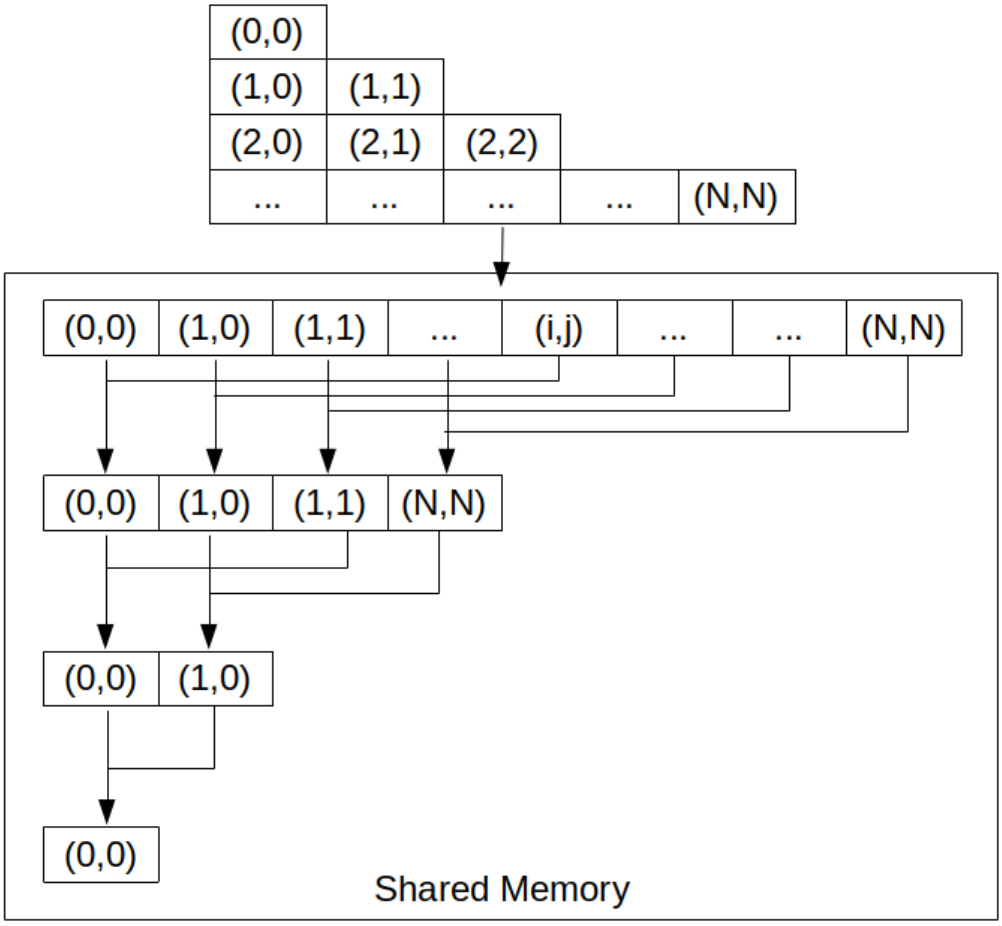

Figure (4.13): The reduction process done using a coalesced memory access pattern.

The throughput gap between the GPU version and the CPU version is closing due to the coalesced memory optimisation as seen in Table (4.11) and Figure (4.14). While the V100 and the GTX1080 doubled their throughput, the T4 is not benefiting from the memory access optimisations as much as the other two models.

Table (4.11): Throughput of the GPU implementations after applying the coalesced memory optimisation.

Implementation	Time (s)	Events / s	Speedup (vs GPU Shared)	Speedup (vs CPU)
GPU V100	4.88	204.98	1.79	0.72
GPU T4	14.43	69.32	1.86	0.24
GPU GTX1080	12.40	80.64	2.17	0.28

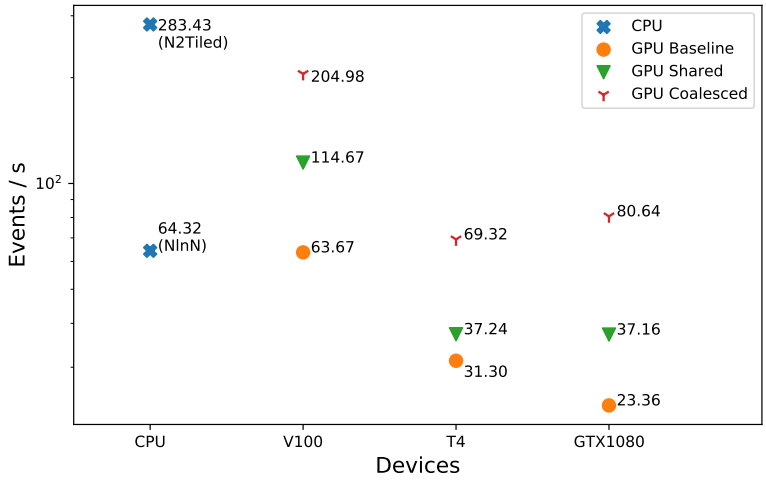

Figure (4.14): GPU coalesced memory optimisation throughput.

The increase in throughput can be explained by the profiling information presented in Figure (4.15). The number of executed instructions is much lower than what it used to be in the share memory version of the algorithm. This is because the coalesced memory access pattern reduces the number of memory instructions required to execute the program (Li et al. 2019).

Moreover, the significant reduction in the number of instructions caused a significant reduction in the number of elapsed clock cycles as shown in Figure (4.16).

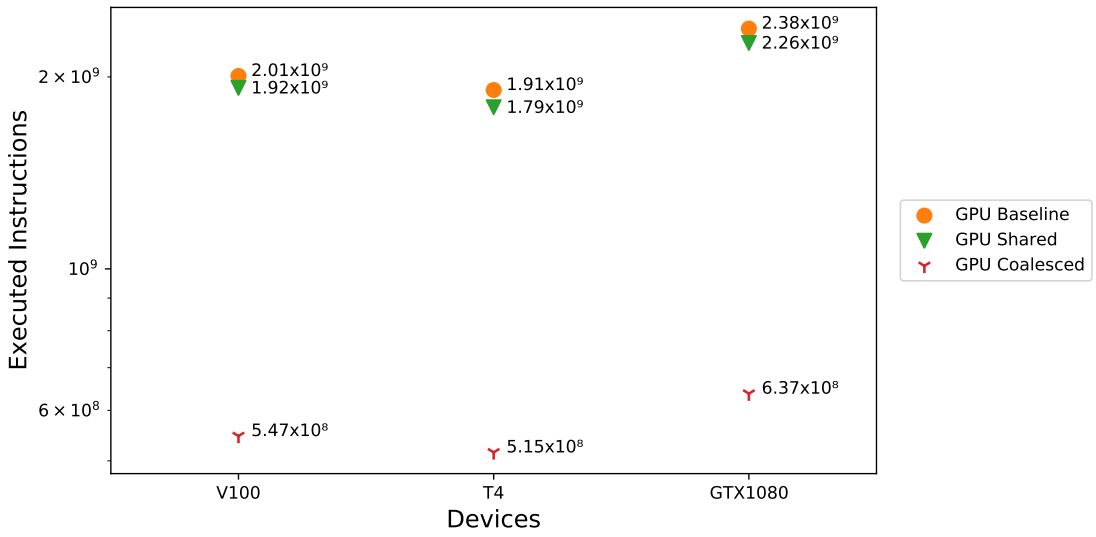

Figure (4.15): GPU optimisations effect on the number of executed instructions.

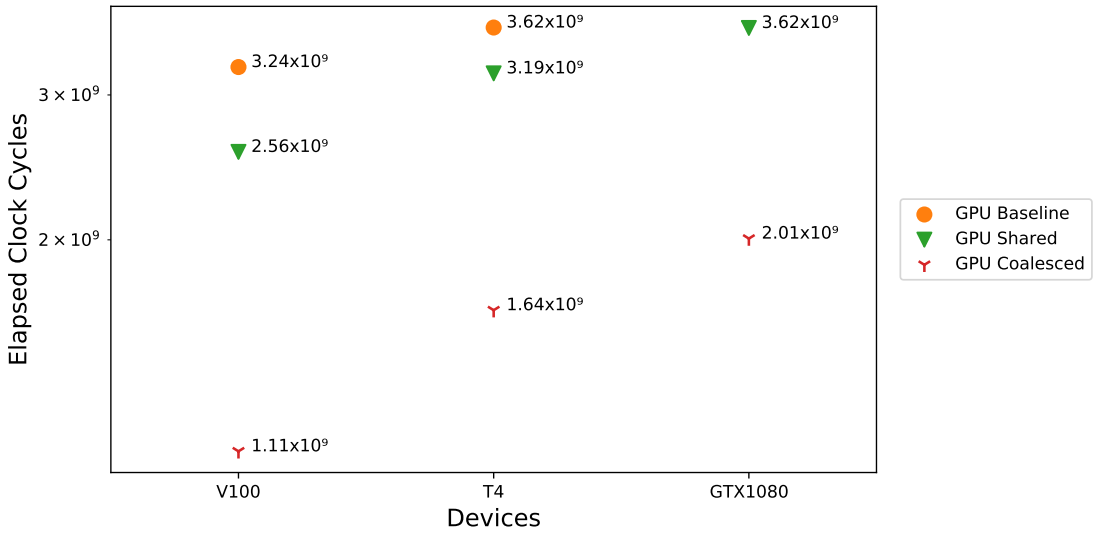

Figure (4.16): GPU optimisations effect on the number of elapsed clock cycles.

Using the call graph profiler, it is possible to map the memory access patterns of the Interleaved and Coalesced GPU implementations. Figure (4.17) is a visualisation for the two versions generated using the call graph profiler. The visualisation is done by getting the access indices for each

iteration and plotting them against the thread that accessed them.

The visualisation shows that after the first iteration (iteration 0 in Figure (4.17)) the coalesced version reduced its shared memory accesses by half. However, the interleaved keeps almost the same number of shared memory accesses until the third iteration.

Such visualisations are useful to make sure that the correct and expected memory access pattern is used. And they can be generated by feeding the execution data into a visualisation engine.

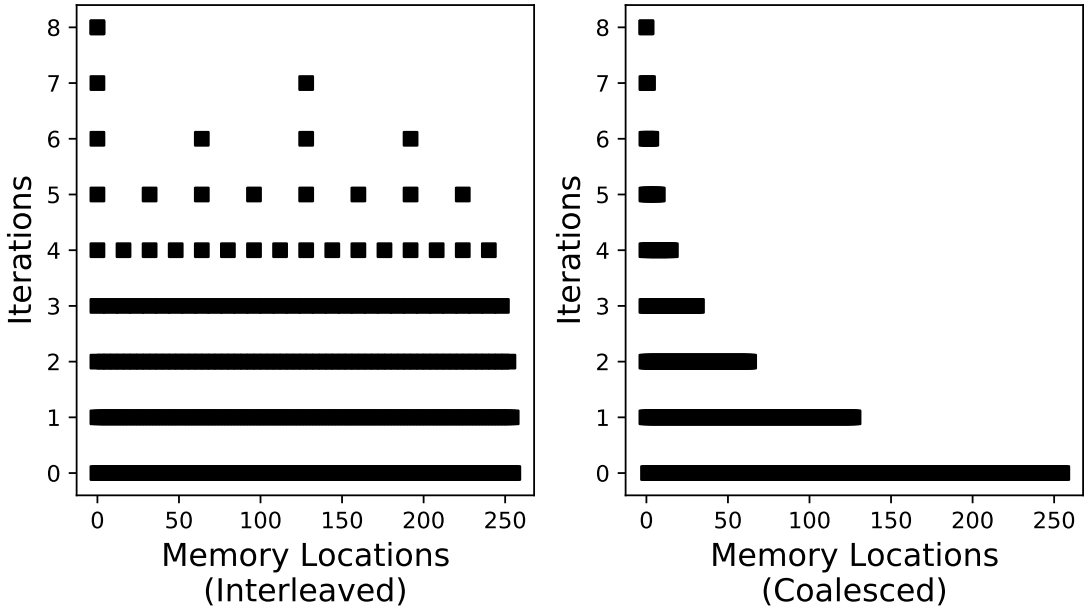

Figure (4.17): Visualisation of two memory access patterns produced by the call graph profiler. The sub-plot on the left shows an interleaved access pattern visualised. The sub-plot on the right shows a coalesced access pattern visualised.

iv. Array of Structures (AoS) vs Structure of Arrays (SoA)

In the previous experiments, different memory types and methods of accessing the data were benchmarked. In this experiment, the data structure

used to store the data is profiled.

On GPUs, it is recommended to have the data stored in a way that supports the efficient access pattern discussed earlier. Therefore the idea of Array of Structures (AoS) vs Structure of Arrayes (SoA) rises (Kirk 2007). In an AoS setting, the data is stored in structures that hold multiple scalars. For each instance of the object needed, a new object of the structure is created and added to an array of objects.

On the other hand, SoA is when one structure is used to hold multiple arrays of different scalar values. Each time a new object needs to be stored, an entry for each scalar is added to the arrays inside the original structure instance. Code listing (4.1) shows an implementation example for AoS and SoA.

The advantage that SoA has over AoS in GPU programs, is that SoA implementations are by default coalesced. Instead of loading the whole data structure every time it's required, in a SoA setting, the GPU can load only a single scalar, and because the scalars are located next to each other in the memory, the GPU can fetch them for the entire warp in an optimal number of memory instructions.

```
1  // AoS example
2  struct AOSDistance {
3      double distance;
4      int index_i;
5      int index_j;
6  };
7
8  // Creating the array
9  AOSDistance aos_distances[10];
10
11 // Accessing the data of the first element
12 aos_distance[0].distance = compute_distance();
13
14 // SoA example
15 struct SOADistance {
16     double distance[10];
17     int index_i[10];
18     int index_j[10];
19 }
20
21 // Creating an instance of the structure
22 SOADistance soa_distances;
23
24 // Accessing the data of the first element
25 soa_distances.distance[0] = compute_distance();
```

Listing (4.1): AoS vs SoA example.

In order to apply the SoA optimisation on the problem at hand, the data structure that holds the distances is redefined to be in a SoA setting (refer to Figure (4.13) and Code listing (4.1)).

The results in Table (4.12) and Figure (4.18) looks contradicting to the initial assumptions. The throughput of the SoA version of the program is lower than the previous optimisation iteration. To explain these results,

the profiling metrics have to be compared for both versions.

Table (4.12): Throughput of the GPU implementations after applying the SoA data structure optimisation.

Implementation	Time (s)	Events / s	Speedup (vs GPU Coalesced)	Speedup (vs CPU)
GPU V100	5.45	183.57	0.90	0.65
GPU T4	14.07	71.07	1.03	0.25
GPU GTX1080	12.41	80.55	1.00	0.28

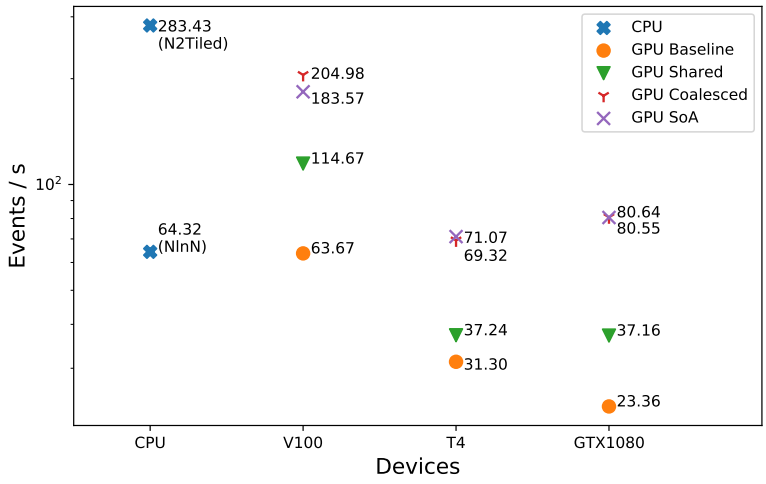

Figure (4.18): GPU SoA optimisation throughput.

Looking at the profiling metrics from Figure (4.19), it is clear that the SoA optimisation is not doing what it is supposed to do. The number executed instructions increased in comparison to the coalesced optimisation, which increased the total number of clock cycles (see (4.20)).

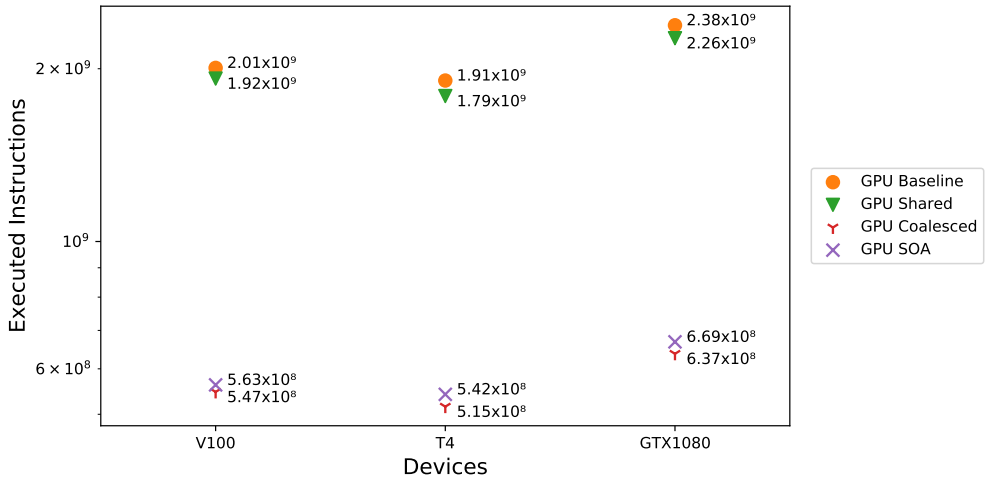

Figure (4.19): GPU optimisations effect on the number of executed instructions.

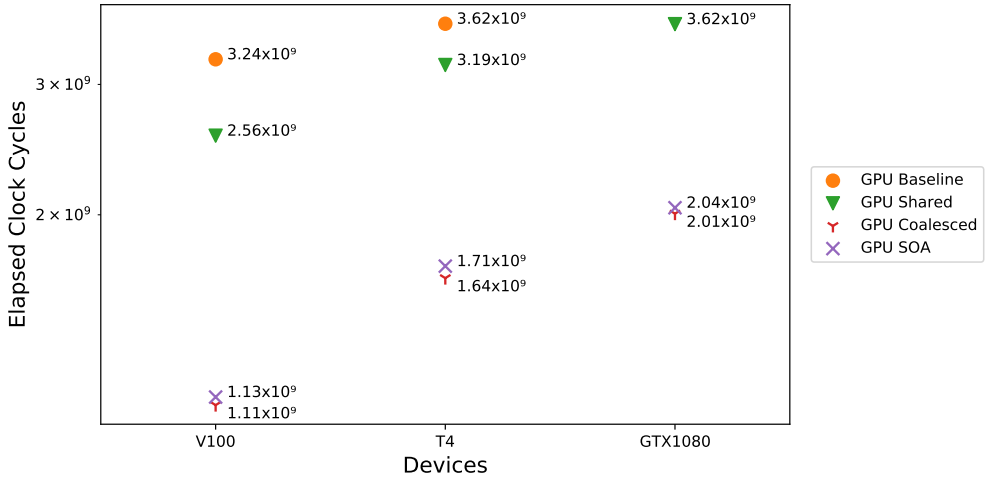

Figure (4.20): GPU optimisations effect on the number of elapsed clock cycles.

This problem has been encountered by Mei and Tian (2016) in their work about the impact of data layouts on the efficiency of GPU accelerated programs. They suggested that if a AoS structure is aligned to the memory bank word size, then the GPU run time can optimise the access to the structure, and this optimisation is done better than the SoA optimisation.

Looking back at the distance structure used in this study shown in Code

listing (4.1), the size of the structure in the AoS versions is 16 Bytes (8-byte double + 2 4-byte integers). Which means that the AoS is in fact aligned with the GPU bank word size, therefore it is optimised.

To test this hypothesis, a sub-experiment was conducted. In this experiment, the size of the structure was changed to 12 Bytes by changing the integers types to short type which has a size of 2 Bytes. This change broke the alignment with the GPU bank size which should stop the GPU from optimising the access to the structure.

The results in Figure (4.21) show that the structure size indeed affects the performance of the SoA and AoS implementations. The SoA data structures are better for non-aligned data structures. However, if the data structure is aligned the GPU compiler and run time will probably optimise it. Therefore using an AoS in this case is preferred. This result matches the one from the results by Mei and Tian (2016).

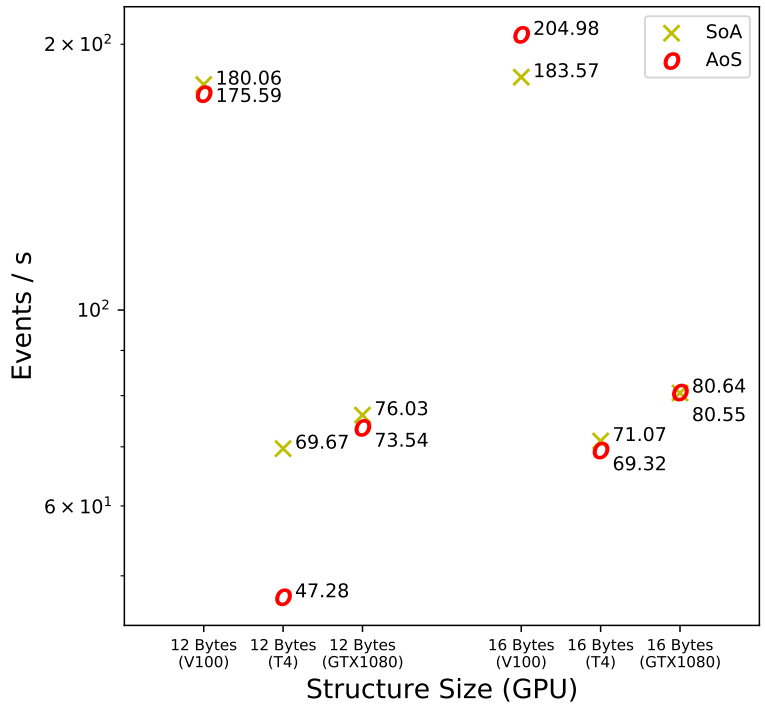

Figure (4.21): Comparison between the AoS and SoA implementations with different structure sizes.

v. Branch Divergence and Idle Threads

Branch divergence can be a major performance limiter for a program running on the GPU. When the threads in the same warp take different execution paths due to a branching instruction (see Listing (4.2)), the SM ends up executing both of the branches in a serial manner (Lin and Wang 2020), which means that the run time is the sum of executing all the branches instead of being the run time of the critical path (see Figure (4.22)). The effect of branch divergence can be predicted by performing a static analysis on the code or by dynamic analysis using profiling.

```
1 // Get the thread ID
2 int tid = threadIdx.x;
3
4 if (tid == 0) {
5     // execute thread 0 code
6 } else if (tid == 1) {
7     // execute thread 1 code
8 }
```

Listing (4.2): Branch divergence example.

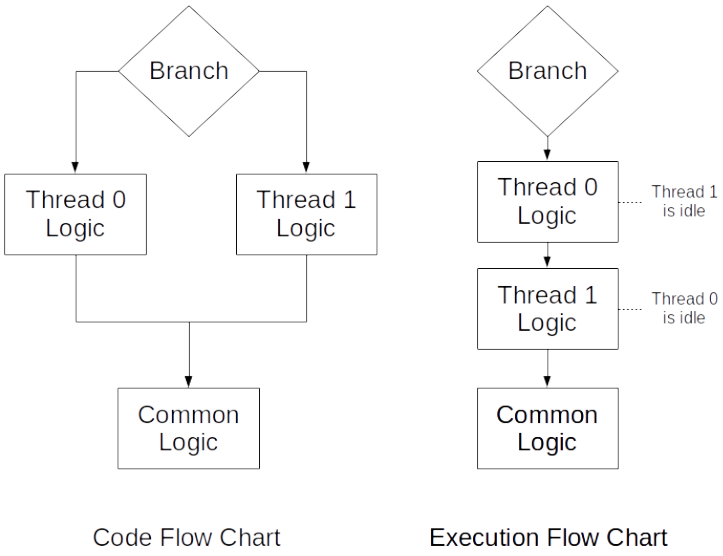

Figure (4.22): Diverged code flow chart vs execution flow chart.

In this experiment, the call graph profiler is used to visualise the branch instructions and their effect on the execution time. The aim of the visualisation is to give a visual representation of how much work is done by every GPU thread in a block. Threads that are idle are a waste of resources and need to be kept at minimum.

Figure (4.23) shows the call graph profiler output limited to three reduction

operations, each black square represent an active thread doing work, white spaces are idle threads. It is clear that after the first reduction step in each loop, half the threads are idle which is a waste of the GPU resources.

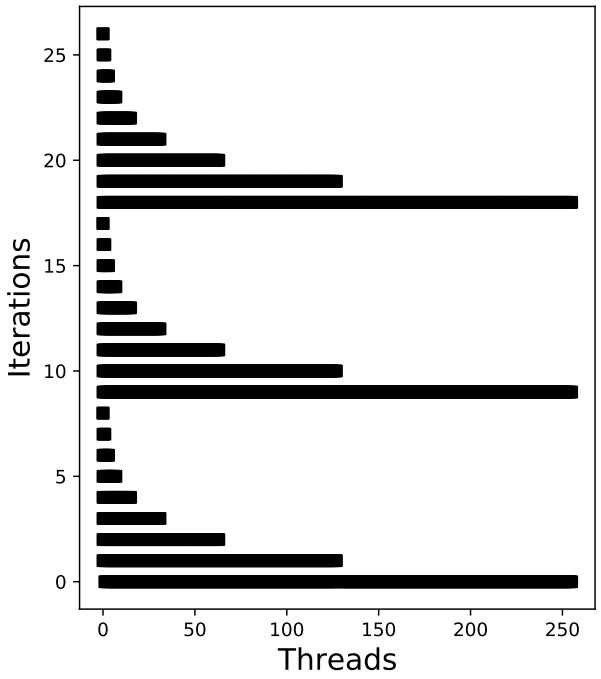

Figure (4.23): Diverged code visualised.

Figure (4.24) shows the needed modification to the implementation to limit divergent and idle threads. Instead of performing the reduction of each 256 (number of threads / block) elements from the tri-matrix. The algorithm keeps fetching the elements from the matrix, keeping the element with minimum distance, and discarding other elements. After all the elements of the matrix are processed, and 256 minimums have been found, the reduction step happens.

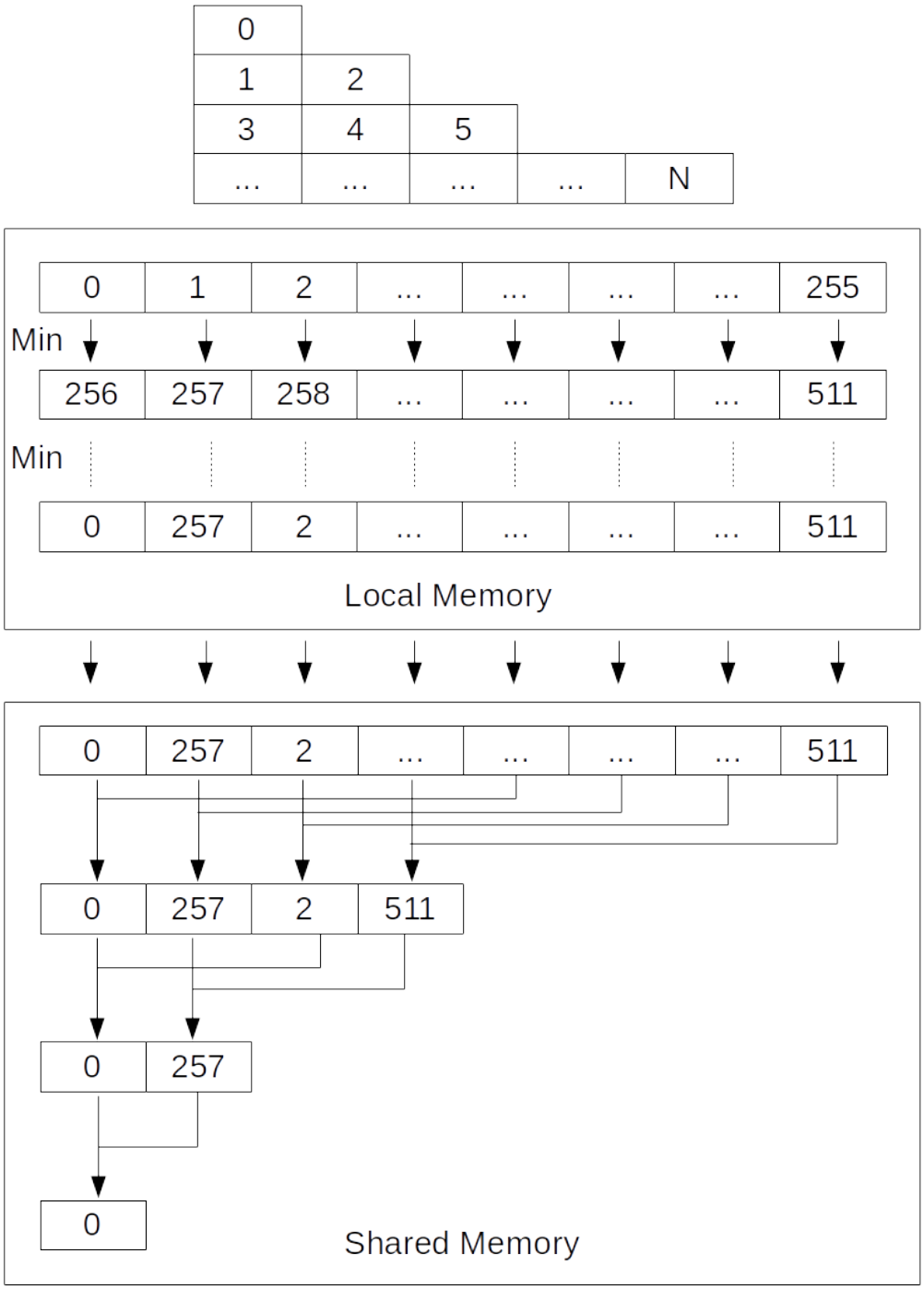

Figure (4.24): The reduction process done using minimum divergent threads.

Figure (4.25) shows a visualisation of the work done by the threads in the last iterations before the reduction. The total number of iterations is

the same as the diverged code. However, the way the work is distributed between the threads has changed.

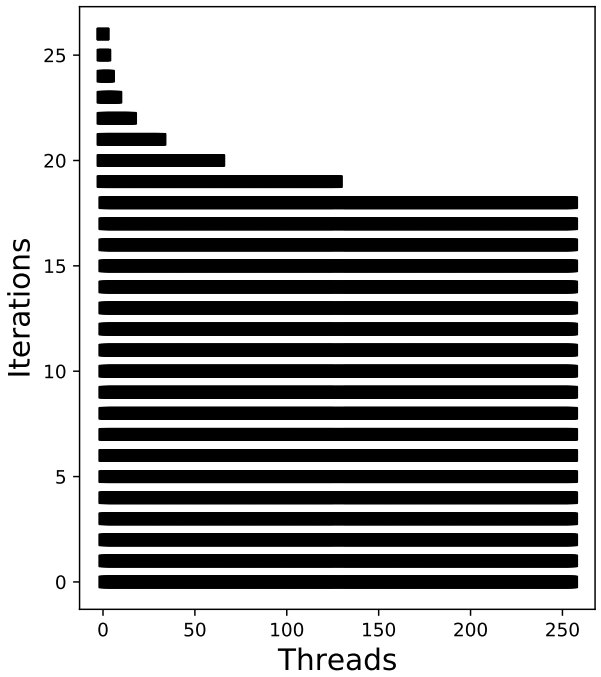

Figure (4.25): Minimum diverged code visualised.

After the divergent threads optimisation, still the CPU version is outperforming the GPU version as shown in Table (4.13)) and Figure (4.26).

After this optimisation, the number of instructions executed and the number of clock cycles elapsed is order of magnitudes lower than the baseline as shown in Figure (4.27) and Figure (4.28).

The current implementation performs shared memory operations only on the last step (see Figure (4.24)). The most executed loop dose double precision operations. Therefore, the next optimisation aims at reducing the number of double precision operations.

Table (4.13): Throughput of the GPU implementations after minimising the diverged threads.

Implementation	Time (s)	Events / s	Speedup (vs GPU Coalesced)	Speedup (vs CPU)
GPU V100	4.03	248.21	1.21	0.88
GPU T4	12.31	81.23	1.17	0.29
GPU GTX1080	8.41	118.94	1.47	0.42

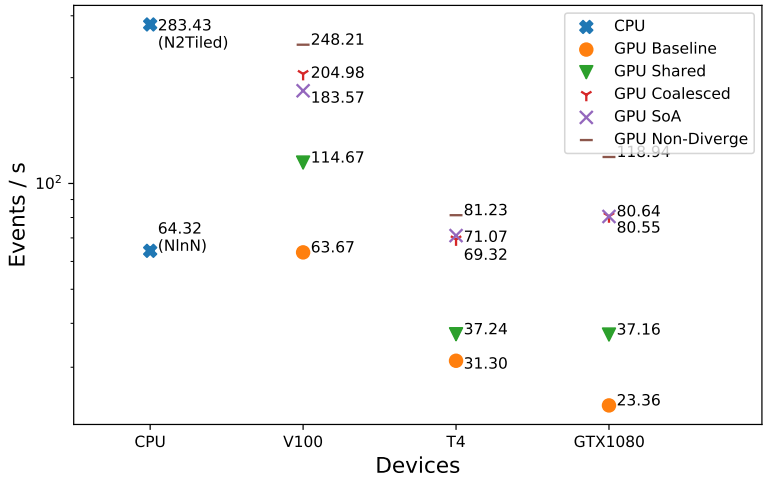

Figure (4.26): Throughput of the GPU implementations after minimising the diverged threads.

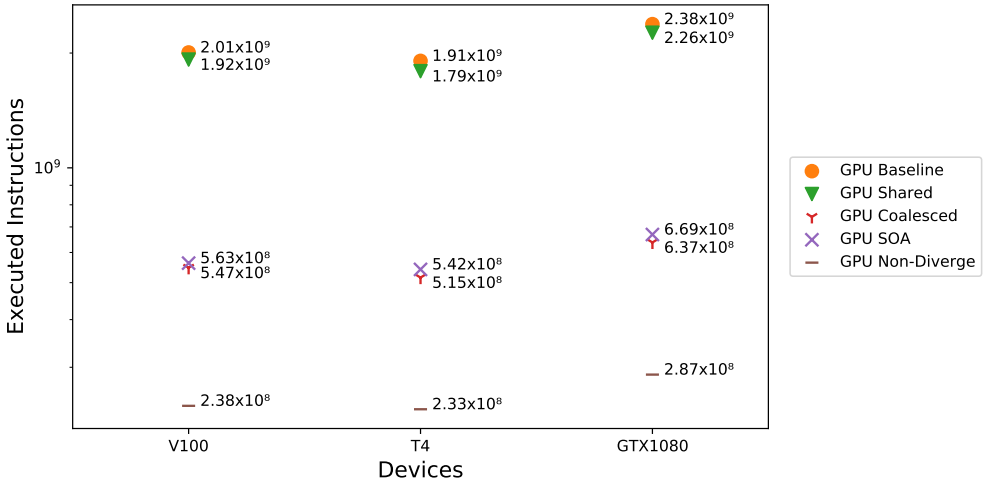

Figure (4.27): GPU optimisations effect on the number of executed instructions.

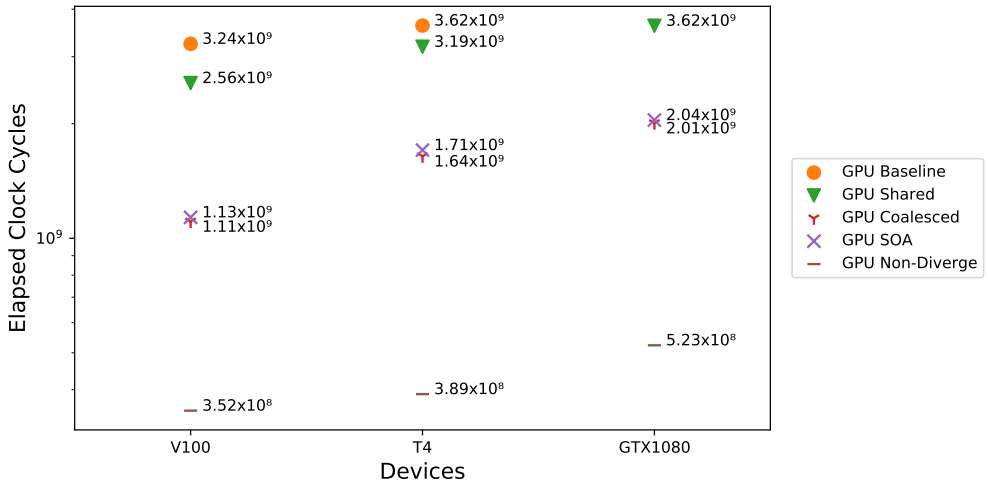

Figure (4.28): GPU optimisations effect on the elapsed clock cycles.

vi. Algorithmic Optimisation Using a Grid Data Structure

The Tri-Matrix implementation searches for a particle nearest neighbour by measuring the distance between the particle and all the particles before it in the list. This results in a $\frac{n(n+1)}{2}$ distance calculations for every recombination step in the worst case. However, since the maximum jet cone radius is predefined as an input to the clustering algorithm, it is possible to exploit this information in order to limit the number of distance calculations, which results in reducing the number of double precision calculations.

It is possible to think of the internal cylinder of the detector as a grid (see Figure (4.29)). The particles then can be distributed on the grid based on their η and ϕ values that represent their position on the grid. The cell size can be computed in a way to limit the particle from searching for its nearest neighbour to 8 cells around the cell containing it (see Figure (4.29)). Therefore, the grid can be used to limit the number of distance calculations in the search for the particle nearest neighbour.

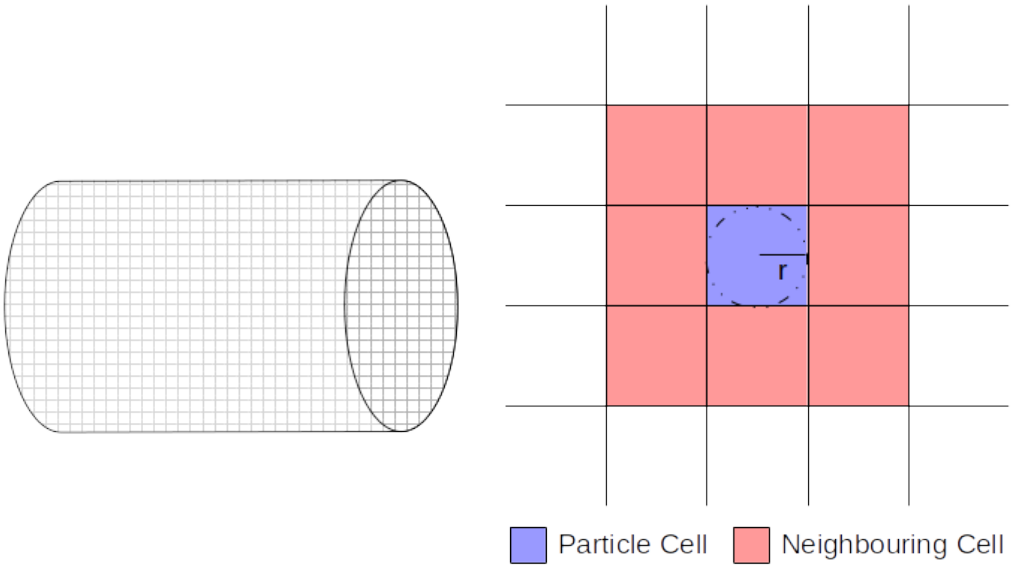

Figure (4.29): A grid data structure to store minimum distances between particles.

After finding the smallest distance and recombining particles, the main advantage provided by the grid algorithm is in limiting the recalculation of the distances between the newly generated particle and the other particles, to the particles in the same cell and in the surrounding 8 cells, this can lead to computing distances in 27 cells in the worst case.

The grid data structure reduced the number of instructions and clock cycles significantly (see Figure (4.31)). This allowed the GPU implementation to outperform the CPU best implementation by 13 times the throughput in the worst case (see Table (4.14) and Figure (4.30)).

Since this is an algorithmic optimisation, the T4 is showing much more improvement in performance compared to the memory access optimisations. The V100 being the GPU with the best 64bit floating point calculations

capabilities is able to achieve up to 28 times the CPU throughput.

Figure (4.31) and Figure (4.32) shows the significant reduction in the number of instructions and the elapsed clock cycles the grid optimisations achieves.

Table (4.14): Throughput of the GPU implementations after using the grid data structure.

Implementation	Time (s)	Events / s	Speedup (vs GPU Non-Diverge)	Speedup (vs CPU)
GPU V100	0.12	8198.00	33.03	28.92
GPU T4	0.23	4353.90	53.60	15.36
GPU GTX1080	0.26	3864.07	32.49	13.63

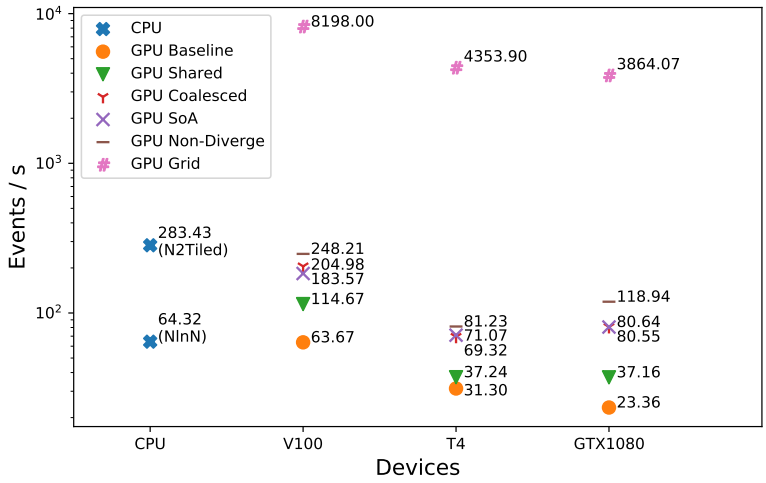

Figure (4.30): Throughput of the GPU implementations after using the grid data structure.

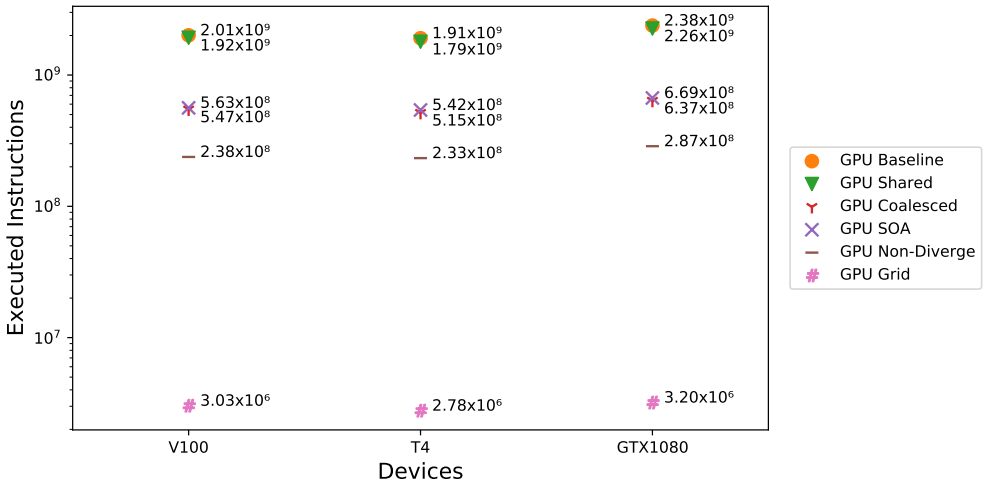

Figure (4.31): GPU optimisations effect on the number of instructions executed.

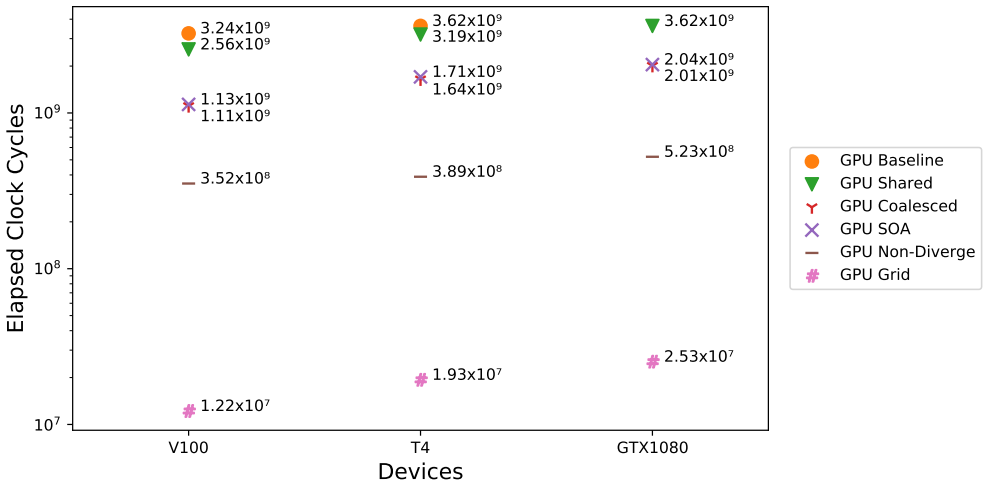

Figure (4.32): GPU optimisations effect on the elapsed clock cycles.

4.5. Conclusion

This chapter started by explaining the experimental environment. To apply the experimental framework, an Experimental Environment was introduced and described. The experimental environment consists of different hardware and software, each hardware component was specified along its specifications. Moreover, each software component used was

specified alongside its corresponding versions and configurations.

After setting up the framework and the environment, the chapter proceeded to list a description of all the experiments followed by their goals and the research questions they tackle. The experiments start by exploring the baseline implementations on CPU followed by a GPU baseline. The GPU baseline is optimised through number of optimisations that allows the algorithm to better utilise the GPU resources. Followed by experiments that improve the algorithm itself by implementing algorithmic optimisations.

The chapter finally presents the results of the experiments and the observations gathered from conducting them. The experiments proved that memory locality is crucial in decreasing the GPU memory latency. In addition, better memory access methods such as coalesced memory access, can reduce the memory latency.

Furthermore, by testing the SoA data structures, it is found that they are useful for non-aligned data structures, however the AoS data structure performers better with smaller aligned data structures.

Reducing the memory latency proved to be effective in optimising the GPU program, however it was not enough to outperform the CPU implementation. Therefore, algorithmic-oriented optimisations were used. Reducing the divergent threads was useful, but the optimisation that made the big difference was using a grid data structure to minimise the computations required to reach a solution. All the optimisations put together were enough to produce a program that has at least a throughput 13 times more than the CPU best implementation.

The optimisation process was guided by profiling metrics, the Nsight profiler was used to get a feel of where the GPU is spending most of its clock cycles. In addition, a call graph profiler was used to do more detailed profiling and behavioural mapping of the GPU performance. The data form both profilers used to achieve the final result.

Next chapter is a summary of the work done in this project, the questions answerd, and the results acheived.

Chapter Five
Conclusion

5.1. Introduction

This chapter summarises the work done in this project. The first section summarises the steps followed to conduct this project. The second section summarises the results obtained from conducting the experiments and how they answer the research questions. The third section compares between the results from this experiment and other experiments surveyed earlier in Chapter 2. The contributions and the future work are mentioned in sections 4 and 5 respectively.

5.2. Summary of the Project

The first part of the project is the literature review. In this step, problem domain was surveyed. The need for parallel computing using GPUs in the field of HEP was discussed with a focus on the CMS experiment. Moreover, several experiments that utilised GPUs in the HEP field were discussed to understand the challenges of this field. Different parallel computing frameworks were surveyed to understand what they can deliver to meet the computing needs of HEP. From the literature, it was found that getting the best performance in parallel programs is a major challenge. The different experiments that utilised GPUs used performance profilers to overcome the performance challenges imposed by parallel computing on GPUs. Therefore, the main focus area chosen for this research work is how to utilise performance profiling to get better performance from GPU programs.

After identifying the research problem, a methodology was introduced to describe the data that is used in the experiments, the CPU and GPU programs used from the CMS experiment, and the performance models, and the profilers that are used to benchmark the programs. The FastJet program was the software module selected from the CMS experiment to be parallelised. The NVIDIA Nsight Compute profiler was used as the main profiler. In addition, a Call Graph profiler was implemented to get better understating of the performance issues and the optimisation techniques.

The next step was to generate the physics data using a Monte Carlo simulation. The generated data was the input for the implemented parallel programs. After preparing the input data, the CPU version of the FastJet program was benchmarked. Followed by a GPU baseline implementation to be used as a starting point for the profiling and optimisation experiments.

After getting the CPU and GPU baselines, multiple experiments were used to benchmark the effect of different GPU optimisations. The first optimisation was done using the shared memory instead of global memory. Then, interleaved and coalesced memory access patterns were compared. Another experiment was used to measure the effect of using SOA data structures instead of AOS data structures. Then, an optimisation to reduce the number of diverged and idle threads was used followed by using a grid data structure to limit the number of distance calculations in the program.

The final step was to compare the effect of each optimisation on the performance of the parallel program. Each optimisation was added to

the GPU version of the program incrementally until the GPU program outperformed the CPU program. After adding each optimisation, the GPU programs were profiled and the new iteration is compared to the previous increments. Both, the Nsight Compute and the Call Graph profilers were used to guide the optimisation process of the GPU program.

5.3. Summary of the Results

The main objective of this project, is to utilise GPU performance profiling tools in order to enhance the performance of CMS software modules. Using Amdahl's and Gustafson's Laws with performance profilers was useful to measure how much it takes the data to be copied from the CPU main memory to the GPU main memory. If the over head of copying the data is more than the time it takes the CPU program to finish executing, then the GPU program will never outperform the CPU program. In the case of the FastJet program, it was found that copying the data from the CPU to the GPU requires only a small fraction of time compared to the CPU processing time. Therefore, it was a good candidate to be ported to the GPU.

By using the profilers, it was found that the single global memory operation required from 69 to 85 clock cycles to be executed on the GPU. On the other hand, a single FP64 operation required from 255 to 295 clock cycles to be executed on the different GPU models. Using this profiling information, and the fact that the memory operations count for most of the instruction executed by the GPU, it was concluded that the memory operations are a bottleneck.

To optimise memory access time, the faster shared memory was used. It was found by profiling that the average cycles required to execute a shared memory access is between 12 and 48 clock cycles. Therefore, using the shared memory whenever possible reduced the number of clock cycles required by the GPU program. This increased the throughput of the program.

In order to reduce the number of memory operations, a coalesced memory access pattern was used. The coalesced access pattern provided better utilisation for the memory cache lines and memory banks on the GPU. This optimisation reduced the number of memory operations required to run the GPU program and increased the throughput. In addition, a capability of the Call Graph profiler was implemented to visualise the memory access pattern on the GPU.

The GPU program was optimised using a SOA data structure. The optimisation was not useful. Upon further inspection, it was found that the SOA optimisation is useful only for unaligned data structures. For example, a data structure with size 12 bytes was profiled against a data structure of size 16 bytes. The SOA was faster with the 12 bytes data structure and the AOS was faster with the 16 bytes data structure because the 16 bytes is aligned with the GPU memory banks sizes.

To reduce the number of diverged branches in the GPU program, the Call Graph profiler was used to visualise the work done by each thread. Reducing the number of divergent and idle threads resulted in reducing the number of instructions executed by the GPU program. This optimisation increased the throughput across all the GPU models.

After optimising the memory operations, the FP64 operations became the bottleneck. From the literature (refer to Table (2.2)), it was found that the largest performance optimisation can be obtained from algorithmic optimisations instead of micro-benchmarking. Therefore, the algorithm was updated to use a grid data structure. The main benefit provided by the grid data structure was decreasing the number of FP64 operations required to execute the program. This optimisation made the GPU performance 15 times better than the CPU performance.

The effectiveness of the optimisations differed between the different GPUs used in this project. The T4 was not affected by the memory access optimisations as much as the V100 and the GTX1080. However, the T4 performance improved significantly by the algorithmic optimisation. The V100 has the highest throughput among all the used GPUs because of its superior specifications.

According to the obtained results, the research questions can be answered as follows:

- **Main Question: How to utilise parallel programming profiling techniques to re-engineer CMS DAQ and Reconstruction systems to achieve better performance?**

  Profiling techniques can be used to identify the software modules that could benefit from parallel computing as the work done by Benelli et. al. (2011). In addition, profiling techniques can be used to predict the performance of the parallel program on GPUs as seen in the experiment in section ii. in 4.4.1. Moreover, profiling can be

used during the GPU program development to find performance bottlenecks and optimisation opportunities as seen in the experiments in section 4.4.2.

- **Minor Question (1): How to re-engineer the CMS DAQ and Reconstruction systems to use parallel computing?**

  After selecting a program to enhance using parallel computing, it is necessary to create a baseline parallel program that ensures the correctness of the parallel program before applying any kind of optimisation. This approach was used by multiple other experiments mentioned in Table (2.2), and it was utilised in this project as seen in GPU Baseline experiment (see section i. in 4.4.2).

  After implementing the baseline, a performance model can be used to benchmark the performance of the parallel program in order measure its performance and find optimisation opportunities (see sections ii. - vi. in 4.4.2).

- **Minor Question (2): How to evaluate the performance of a parallel program on GPUs?**

  In the case of evaluating a GPU program on the same GPU model, the number of executed instructions or the number of elapsed clock cycles can be enough to compare between two different programs, lower count means better performance. However, evaluating the performance between different GPU models or between the GPU and CPU, the throughput is the evaluation criteria that can be used. In addition, the Speed up which can be derived from the throughput

can be used to compare between two different implementations (refer to the experiments in section 4.4.2).

- **Minor Question (3): What is a suitable model to profile the performance of parallel programs on GPUs?**

  In the experiment in section ii. in 4.4.2 an attempt to predict the maximum possible throughput was done. The maximum throughput found is much higher than the final throughput reached in section vi. in 4.4.2. Therefore, the model used was not useful in predicting the maximum speedup. However, the experiment was useful in determining the time required to move the data from the CPU main memory to the GPU main Memory. Timing this task is necessary to make sure that porting the program to the GPU is feasible.

- **Minor Question (4): How to use the metrics obtained from the profiler to detect optimisation opportunities in parallel programs on GPUs?**

  Using the instructions counters, it is possible to know which instructions are more expensive in terms of clock cycles. As seen in the GPU Baseline experiment (see section i. in 4.4.2), the counters showed that the global memory instructions consumed most of the GPU clock cycles.

  In the Global vs Shared memory experiment (see section ii. in 4.4.2), the call graph profiler was used to measure the average clock cycles required to execute the different memory operations. This helped in determining which memory type is faster and how much

it is faster on each GPU model.

In addition, the metrics collected by the call graph profiler were used to visualise the causes of performance issues such as interleaved memory access patterns or divergent threads (refer to sections iii. and v. in 4.4.2).

- **Minor Question (5): How to optimise parallel programs on GPUs to reduce the effect of bottlenecks?**

  The following GPU optimisations were effective in this project:

  - Using memory types with lower latency. For example, using shared memory over global memory (refer to section ii. in 4.4.2).
  - Using optimised memory access patterns. For example, using coalesced over interleaved memory access patterns (refer to section iii. in 4.4.2).
  - Using SOA data structures for data that does not align with the GPU memory banks size and AOS otherwise (refer to section iv. in 4.4.2).
  - Reducing the number of diverged and idle threads in the program (refer to section v. in 4.4.2).
  - Performing algorithmic optimisations to reduce the number of instructions to perform the task. Therefore, reducing the number of clock cycles, which improves the performance

(refer to section vi. in 4.4.2).

5.4. Contributions

This project provided the following contributions:

- Multiple optimisations were tested in order to measure their effectiveness and the results are summarised in Table (5.1). The table shows the type of the optimisation, speedup compared to the previous optimisation, reduction in instructions and clock cycles as a percentage from the previous optimisation, and the effectiveness of the optimisation.

Table (5.1): A summary of the obtained results from the experiments.

Optimisation	Optimisation Type	GPU	Speedup (Relative to GPU Baseline)	Speedup (Relative to Previous Optimisation)	Instructions Reduction	Clock Cycles Reduction	Effective?
Shared Memory	Memory Access	V100	1.80	1.80	4.20%	21.14%	Yes
		T4	1.19	1.19	6.03%	12.05%	
		GTX1080	1.59	1.59	5.06%	None	
Coalesced Access	Memory Access	V100	3.22	1.79	71.55%	56.74%	Yes
		T4	2.21	1.86	71.24%	48.46%	
		GTX1080	3.45	2.17	71.81%	44.53%	
SOA	Memory Access	V100	2.88	0.90	-2.87%	-2.55%	No
		T4	2.27	1.03	-5.07%	-3.81%	
		GTX1080	3.45	1.00	-4.92%	-1.87%	
Non-Diverged	Control Flow	V100	3.90	1.21	57.75%	68.95%	Yes
		T4	2.60	1.17	57.01%	77.16%	
		GTX1080	5.09	1.47	57.07%	74.41%	
Grid	Algorithmic	V100	128.76	33.03	98.73%	96.54%	Yes
		T4	139.12	53.60	98.81%	95.03%	
		GTX1080	165.43	32.49	98.88%	95.17%	

- A parallel implementation on GPU for the FastJet software package. The parallel GPU program can be used as a standalone solution[1] or as a plug-in to the FastJet package[2]. In addition, a call graph profiler prototype that uses source code instrumentations as a data engine

[1] https://github.com/asubah/fastjet_gpu

[2] https://github.com/asubah/FastJet-GPU-Plugin

and feeds the data to a visualisation engine. Other alternatives mentioned in Chapter 2, focused on instrumenting the PTX code or the output from other profilers.

- Other minor contributions were made, such as proving that SOA optimization is not effective in the case of small aligned data structures (G. Mei and Tian 2016). In addition, proving that the newer GPU architectures are less effected by the memory access optimisations (Shahneous Bari et al. 2018). Furthermore, proving the new GPU architectures compilers does a better job in optimising the program during compile time (Jia, Maggioni, Smith, et al. 2019).

5.5. Future Work

In this project, it was possible to outperform the CPU implementation by implementing a grid based nearest neighbour algorithm. Other options might provide better performance such as the Voronoi algorithm used by the NlnN CPU implementation. Porting the Voronoi algorithm to the GPU might provide better performance than the grid algorithm.

Another direction that might be worth investigating is performance portability. This project focused on GPUs from Nvidia. Other GPU vendors or other parallel processors can be used to test the difference in performance between the different platforms. In addition, parallel computing frameworks that support performance portability can be investigated.

References

Adinetz, A., Herten, A., Kraus, J., Mertens, M. C., Pleiter, D., Stockmanns, T., & Wintz, P. (2015). Triplet finder: On the way to triggerless online reconstruction with gpus for the p¯anda experiment. *Journal of Computational Science*, *10*, 317–326. doi:https://doi.org/10.1016/j.jocs.2015.03.010

Amdahl, G. M. (1967). Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the Spring Joint Computer Conference* (pp. 483–485). AFIPS '67 (Spring). Atlantic City, New Jersey: Association for Computing Machinery.

Ammendola, R., Bauce, M., Biagioni, A., Fantechi, R., Fiorini, M., Giagu, S., ... Vicini, P. (2013). The GAP project - GPU for realtime applications in high energy physics and medical imaging. In *2013 ieee nuclear science symposium and medical imaging conference (2013 nss/mic)* (pp. 1–7). doi:10.1109/NSSMIC.2013.6829757

Ammendola, R., Biagioni, A., Chiozzi, S., Cretaro, P., Lorenzo, S. D., Fantechi, R., ... Vicini, P. (2017). GPU real-time processing in NA62 trigger system. *Journal of Physics: Conference Series*, *800*.

Andre, J. M., Behrens, U., Branson, J., Chaze, O., Cittolin, S., Contescu, C., ... Velez, C. V. (2017). *The Phase-2 Upgrade of the CMS DAQ Interim Technical Design Report* (tech. rep. No. CERN-LHCC-2017-014. CMS-TDR-018). CERN. Geneva. Retrieved from https://cds.cern.ch/record/2283193

Apollinari, G., Béjar Alonso, I., Brüning, O., Lamont, M., & Rossi, L. (2015). *High-luminosity Large Hadron Collider (HL-LHC): Preliminary Design Report*. Fermi National Accelerator Lab (FNAL). Batavia, IL, United States.

Barney, D. (2016). *CMS Detector Slice*. CMS Collection. Retrieved from https://cds.cern.ch/record/2120661

Bayatyan, G. L., Grigorian, N., Khachatrian, V. G., Margarian, A. T., Sirunyan, A. M., Stepanian, S., & Adam. (2000). *CMS TriDAS Project: Technical Design Report, Volume 1: The Trigger Systems* (tech. rep. No. CERN-LHCC-2000-038). Retrieved January 31, 2020, from https://cds.cern.ch/record/706847

Benelli, G., Bozsogi, B., Pfeiffer, A., Piparo, D., & Zemleris, V. (2011). Measuring CMS software performance in the first years of LHC collisions. In *2011 IEEE Nuclear Science Symposium Conference Record* (pp. 108–112).

Bhatele, A., Brink, S., & Gamblin, T. (2019). Hatchet: Pruning the Overgrowth in Parallel Profiles. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (20:1–21). SC '19. Denver, Colorado: ACM.

Bi, Y.-J., Xiao, Y., Gong, M., Sun, P., Xu, S., & Yang, Y.-B. (2020). Lattice qcd package gwu-code and quda with hip. *arXiv preprint arXiv:2001.05706*.

Bianchi, L., Herten, A., Ritman, J., Stockmanns, T., Adinetz, A., Kraus, J., & Pleiter, D. (2015). Online Tracking Algorithms on GPUs for the PANDA Experiment at FAIR. *J. Phys. Conf. Ser. 664*(8). doi:10.1088/1742-6596/664/8/082006

Blelloch, G. E. (1996). Programming Parallel Algorithms. *Commun. ACM*, *39*(3), 85–97. doi:10.1145/227234.227246

Bloch, P., Brown, R., Lecoq, P., & Rykaczewski, H. (2002). *Changes to CMS ECAL Electronics: Addendum to the Technical Design Report* (tech. rep. No. CERN-LHCC-2002-027). Geneva: CERN. Retrieved from https://cds.cern.ch/record/581342

Bozza, C., Kose, U., De Sio, C., & Stellacci, S. M. (2015). GPU-based quasi-real-time Track Recognition in Imaging Devices: from raw Data to Particle Tracks. In *Proceedings, GPU Computing in High-Energy Physics (GPUHEP2014): Pisa, Italy, September 10-12, 2014* (pp. 9–16). doi:10.3204/DESY-PROC-2014-05/2

Cacciari, M., & Salam, G. P. (2006). Dispelling the N3 myth for the kt jet-finder. *Physics Letters B*, *641*(1), 57–61.

Cacciari, M., Salam, G. P., & Soyez, G. (2012). FastJet User Manual. *Euro Physics Journal*, *C72*, 18–96.

Chabbi, M., Murthy, K., Fagan, M., & Mellor-Crummey, J. (2013). Effective Sampling-driven Performance Tools for GPU-accelerated Supercomputers. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (43:1–12). SC '13. Denver, Colorado: ACM.

CMS Collaboration. (2019). TTToHadronic_TuneCP5_13TeV-powheg-pythia8 in FEVTDEBUGHLT format for LHC Phase2 studies. Retrieved March 15, 2020, from http://opendata.cern.ch/record/12303

Collaboration, C. (1997). *The CMS Magnet Project: Technical Design Report* (tech. rep. No. CERN-LHCC-97-010). Geneva: CERN. Retrieved January 31, 2020, from https://cds.cern.ch/record/331056

Collaboration, C. (2012). *Detector Drawings*. CMS Collection. Retrieved from https://cds.cern.ch/record/1433717

Collins, P. D. B. (2009). *An Introduction to Regge Theory and High Energy Physics*. Cambridge, UK: Cambridge University Press.

Denegri, D. (1995). *The CMS Detector and Physics at the LHC* (tech. rep. No. CMS-TN-1995-167). Retrieved January 31, 2020, from https://cds.cern.ch/record/1365780

Di Domenico, G. (2015). Fast Cone-beam CT reconstruction using GPU. In *Proceedings, GPU Computing in High-Energy Physics (GPUHEP2014): Pisa, Italy, September 10-12, 2014* (pp. 193–198). doi:10.3204/DESY-PROC-2014-05/35

Flynn, M. J. (1972). Some Computer Organizations and Their Effectiveness. *IEEE transactions on computers*, *100*(9), 948–960.

Gallorini, S. (2015). Track pattern-recognition on GPGPUs in the LHCb experiment. In *GPU Computing in High-Energy Physics* (pp. 38–43).

GCC Team. (2020). GCC Online Documentaions: Options That Control Optimization. Retrieved May 5, 2020, from https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Gohn, W. (2015). Data Acquisition for the New Muon g-2 Experiment at Fermilab. *Journal of Physics: Conference Series*, *664*(8), 082014.

Graham, S. L., Kessler, P. B., & Mckusick, M. K. (1982). Gprof: A Call Graph Execution Profiler. *The ACM Special Interest Group on Programming Languages (SIGPLAN)*, *17*(6), 120–126.

Grasseau, G., Lisniak, S., & Chamont, D. (2015). Hybrid implementation of the VEGAS Monte-Carlo algorithm. In *Proceedings, GPU*

Computing in High-Energy Physics (GPUHEP2014): Pisa, Italy, September 10-12, 2014 (pp. 103–108). doi:10.3204/DESY-PROC-2014-05/19

Gropp, W., Lusk, E., Doss, N., & Skjellum, A. (1996). A High-performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, *22*(6), 789–828.

Gustafson, J. L. (1995). Reevaluating Amdahl's Law. In *Multiprocessor Performance Measurement and Evaluation* (pp. 92–93). Washington, DC, USA: IEEE Computer Society Press.

Hager, G. (2010). *Introduction to High Performance Computing for Scientists and Engineers*. Chapman Hall/CRC Computational Science. Hoboken, NJ: CRC Press.

Han, T. D., & Abdelrahman, T. S. (2011). Reducing Branch Divergence in GPU Programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units* (3:1–8). GPGPU-4. Newport Beach, California, USA: ACM.

Hegeman, J. G. (2018). *The CMS Data Acquisition System for the Phase-2 Upgrade* (tech. rep. No. CMS-CR-2018-099). CERN. Geneva. Retrieved January 31, 2020, from https://cds.cern.ch/record/2629375

Holt, W. M. (2016). Moore's law: A path going forward. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)* (pp. 8–13).

Jang, B., Schaa, D., Mistry, P., & Kaeli, D. (2011). Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures. *IEEE Transactions on Parallel and Distributed Systems*, *22*(1), 105–118.

Jia, Z., Maggioni, M., Smith, J., & Scarpazza, D. P. (2019). *Dissecting the Nvidia Turing T4 GPU via Microbenchmarking*. arXiv: 1903.07486

Jia, Z., Maggioni, M., Staiger, B., & Scarpazza, D. P. (2018). *Dissecting the Nvidia Volta GPU Architecture via Microbenchmarking*. arXiv: 1804.06826

Jiang, W., Sha, E. H.-M., Zhang, X., Yang, L., Zhuge, Q., Shi, Y., & Hu, J. (2019). Achieving Super-Linear Speedup across Multi-FPGA for Real-Time DNN Inference. *ACM Trans. Embed. Comput. Syst. 18*(5s).

Jun, S., Canal, P., Apostolakis, J., Gheata, A., & Moneta, L. (2019). Vectorization of Random Number Generation and Reproducibility of Concurrent Particle Transport Simulation. *J. Phys. Conf. Ser.*

Kirk, D. (2007). NVIDIA CUDA Software and GPU Parallel Computing Architecture. In *Proceedings of the 6th International Symposium on Memory Management* (pp. 103–104). ISMM '07. Montreal, Quebec, Canada: ACM.

Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., ... et al. (2010). 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). ISCA '10. Saint-Malo, France: Association for Computing Machinery.

Li, B., Wei, J., Sun, J., Annavaram, M., & Kim, N. S. (2019). An Efficient GPU Cache Architecture for Applications with Irregular Memory Access Patterns. *ACM Trans. Archit. Code Optim. 16*(3).

Lin, H., & Wang, C.-L. (2020). On-GPU Thread-data Remapping for Nested Branch Divergence. *Journal of Parallel and Distributed Computing*, *139*, 75–86.

Lopes, R. H. C., Reid, I. D., & Hobson, P. R. (2015). Tree Contraction, Connected Components, Minimum Spanning Trees: a GPU Path to Vertex Fitting. In *Proceedings, GPU Computing in High-Energy Physics (GPUHEP2014): Pisa, Italy, September 10-12, 2014* (pp. 30–35). doi:10.3204/DESY-PROC-2014-05/5

Lustig, D., & Martonosi, M. (2013). Reducing GPU Offload Latency via Fine-grained CPU-GPU Synchronization. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)* (pp. 354–365). HPCA '13. Washington, DC, USA: IEEE Computer Society.

Malony, A. D., Biersdorff, S., Shende, S., Jagode, H., Tomov, S., Juckeland, G., … Lamb, C. (2011). Parallel Performance Measurement of Heterogeneous Parallel Systems with GPUs. In *Proceedings of the 2011 International Conference on Parallel Processing* (pp. 176–185). ICPP '11. Washington, DC, USA: IEEE Computer Society.

Mans, J., Anderson, J., Dahmes, B., de Barbaro, P., Freeman, J., Grassi, T., … Yetkin, T. (2012). *CMS Technical Design Report for the Phase 1 Upgrade of the Hadron Calorimeter* (tech. rep. No. CERN-LHCC-2012-015. CMS-TDR-10). Retrieved from https://cds.cern.ch/record/1481837

Mc Cauley, T. (2019). *Display of an Event in a Dijet Resonance Search in CMS*. CMS Collection. Retrieved January 5, 2020, from https://cds.cern.ch/record/2681224

Mei, G., & Tian, H. (2016). Impact of Data Layouts on the Efficiency of GPU-accelerated IDW Interpolation. *SpringerPlus*, *5*(1), 104.

Mei, X. [X.], & Chu, X. [X.]. (2017). Dissecting GPU Memory Hierarchy Through Microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, *28*(1), 72–86.

Mei, X. [Xinxin], & Chu, X. [Xiaowen]. (2017). Dissecting GPU Memory Hierarchy Through Microbenchmarking. *IEEE Trans. Parallel Distrib. Syst. 28*(1), 72–86.

Mittal, S., & Vetter, J. S. (2015). A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Computing Surveys (CSUR)*, *47*(4), 1–35.

Munshi, A. (2009). The OpenCL Specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)* (pp. 1–314).

Nickolls, J., & Dally, W. J. (2010). The GPU Computing Era. *IEEE Micro*, *30*(2), 56–69.

NVIDIA. (2019a). CUDA C++ Best Practices Guide. Retrieved January 5, 2020, from https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

NVIDIA. (2019b). Perfworks. Retrieved January 5, 2020, from https://developer.nvidia.com/perfworks

NVIDIA. (2019c). The user manual for NVIDIA Nsight Compute. Retrieved January 5, 2020, from https://docs.nvidia.com/nsight-compute/NsightCompute/index.html

Pantaleo, F. (2017). *New Track Seeding Techniques for the CMS Experiment* (Doctoral dissertation). Retrieved January 31, 2020, from https://cds.cern.ch/record/2293435

Ramroach, S., Dhanoo, A., Cockburn, B., & Joshi, A. (2019). CUDA Optimized Neural Network Predicts Blood Glucose Control from Quantified Joint Mobility and Anthropometrics. In *Proceedings of the 2019 3rd international conference on information system and data mining* (pp. 32–36). ICISDM 2019. doi:10.1145/3325917.3325940

Reinders, J. (2007). *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc.

Rovere, M., Chen, Z., Di Pilato, A., Pantaleo, F., & Seez, C. (2020). CLUE: A Fast Parallel Clustering Algorithm for High Granularity Calorimeters in High Energy Physics. arXiv: 2001.09761 [physics.ins-det]

Schmidt, B., González-Domínguez, J., Hundt, C., & Schlarb, M. (2018). Chapter 8 - Advanced CUDA Programming. In B. Schmidt, J. González-Domínguez, C. Hundt, & M. Schlarb (Eds.), *Parallel programming* (pp. 287–313). Morgan Kaufmann.

Schouten, D., DeAbreu, A., & Stelzer, B. (2015). GPUs for Higgs boson data analysis at the LHC using the Matrix Element Method. In *Proceedings, GPU Computing in High-Energy Physics (GPUHEP2014): Pisa, Italy, September 10-12, 2014* (pp. 109–118). doi:10.3204/DESY-PROC-2014-05/20

Serpa, M. S., Moreira, F. B., Navaux, P. O. A., Cruz, E. H. M., Diener, M., Griebler, D., & Fernandes, L. G. (2019). Memory Performance and Bottlenecks in Multicore and GPU Architectures. In *27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)* (pp. 233–236). Pavia, Italy.

Shahneous Bari, M. A., Stoltzfus, L., Lin, P., Liao, C., Emani, M., & Chapman, B. (2018). Is Data Placement Optimization Still Relevant on Newer GPUs? In *IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)* (pp. 83–96). Dallas, TX, USA.

Shen, D., Song, S. L., Li, A., & Liu, X. (2018). CUDAAdvisor: LLVM-based Runtime Profiling for Modern GPUs. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (pp. 214–227). CGO 2018. Vienna, Austria: ACM.

Spera, M. (2015). Using Graphics Processing Units to solve the classical N-body problem in physics and astrophysics. In *Proceedings, GPU Computing in High-Energy Physics (GPUHEP2014): Pisa, Italy, September 10-12, 2014* (pp. 187–192). doi:10.3204/DESY-PROC-2014-05/34. arXiv: 1411.5234 [astro-ph.IM]

Stephenson, M., Sastry Hari, S. K., Lee, Y., Ebrahimi, E., Johnson, D. R., Nellans, D., … Keckler, S. W. (2015). Flexible Software Profiling of GPU Architectures. *SIGARCH Comput. Archit. News*, *43*(3), 185–197.

Vogt, H., & Schröck, M. (2015). cuLGT: Lattice Gauge Fixing on GPUs. In *Proceedings, GPU Computing in High-Energy Physics (GPUHEP2014): Pisa, Italy, September 10-12, 2014* (pp. 175–180). doi:10.3204/DESY-PROC-2014-05/31. arXiv: 1412.3655 [hep-lat]

vom Bruch, D. (2015). Track and Vertex Reconstruction on GPUs for the Mu3e Experiment. In *Proceedings, GPU Computing in High-*

Energy Physics (GPUHEP2014): Pisa, Italy, September 10-12, 2014 (pp. 71–76). doi:10.3204/DESY-PROC-2014-05/13

Washbrook, A., Clark, P., Jones, C., Rovatsou, M., & Emeilyanov, D. (2011). *Algorithm Acceleration from GPGPUs for the ATLAS Upgrade* (tech. rep. No. ATL-DAQ-PROC-2011-010). CERN. Geneva. Retrieved January 31, 2020, from https://cds.cern.ch/record/1323859

Wienke, S., Springer, P., Terboven, C., & an Mey, D. (2012). OpenACC: First Experiences with Real-world Applications. In *Proceedings of the 18th International Conference on Parallel Processing* (pp. 859–870). Euro-Par'12. Rhodes Island, Greece: Springer-Verlag.

Wong, H., Papadopoulou, M., Sadooghi-Alvandi, M., & Moshovos, A. (2010). Demystifying GPU Microarchitecture Through Microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)* (pp. 235–246).

Wu, J., Yang, X., Zhang, Z., Chen, G., & Mao, R. (2019). A Performance Model for GPU Architectures that Considers On-Chip Resources: Application to Medical Image Registration. *IEEE Transactions on Parallel and Distributed Systems*, *30*(9), 1947–1961.

Yeo, B., Lee, M. J., Semertzidis, Y. K., & Kuno, Y. (2019). GPGPU Tracking in the COMET Phase-I Cylindrical Drift Chamber. In *Connecting the Dots and Workshop on Intelligent Trackers*. arXiv: 1908.01949 [physics.ins-det]

Zhe Fan, Feng Qiu, Kaufman, A., & Yoakum-Stover, S. (2004). GPU Cluster for High Performance Computing. In *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing* (pp. 47–47).

الملخص

التجارب الفيزيائية عالية الطاقة تولد كميات هائلة من البيانات التي تحتاج إلى التحليل من قبل علماء الفيزياء. التزايد المستمر في أحجام البيانات يدفع علماء الحاسب لاستخدام الحوسبة المتوازية لتغطية لتوفير ما يكفي من القوة الحسابية للتجارب الفيزيائية. تعد معالجات الرسوميات من أفضل المعالجات للحوسبة المتوازية، لذلك يلجأ الباحثون إلى محاولة تحقيق أقصى استفادة منها.

في هذا البحث تمت دراسة استخدام معالجات الرسوميات في تجربة كاشف الميون العملاق بغرض تحليل البيانات. يهدف البحث إلى اختيار أحد البرامج المستخدمة في التجربة بغرض إعادة هندسته ليتوافق مع معالجات الرسوميات بهدف تسريعه وزيادة إنتاجه للبيانات. يتطرق البحث إلى كيفية تقييم وقياس أداء البرامج المتوازية على المعالجات الرسومية وكيفية تحسين أدائها لتحقق أقصى استغلال لموارد المعالج.

تمت دراسة تقنيات تحسين مختلفة، منها التحسينات المتعلقة بذاكرة معالج الرسوميات، والتحسينات المتعلقة بترتيب الأوامر، والتحسينات المتعلقة بالخوارزميات المتوازية. كما تم استخدام مقاييس مختلفة وأدوات قياس متعددة بغرض مقارنة البرامج التي تمت دراستها مع بعضها البعض.

تشير النتائج المستفادة من إجراء التجارب في هذا البحث إلى أن التحسينات المتعلقة باستخدام الذاكرة المشتركة في معالج الرسوميات يخفض عدد العمليات ودورات المعالجة بنسبة 4% و12% على التوالي. واستخدام النمط الملتحم للولوج إلى الذاكرة يخفض عدد العمليات ودورات المعالجة بنسبة 71% و44% على التوالي. كما تبين أن استخدام هياكل بيانات المصفوفات يزيد عدد العمليات ودورات المعالجة بنسبة 6% و4% على التوالي. بالإضافة إلى ذلك، تبين أن تقليل التشعبات في البرنامج يخفض عدد العمليات ودورات المعالجة بنسبة 57% و68% على التوالي. أخيرا، تم التوصل إلى أن استخدام هياكل البيانات الشبكية يخفض عدد العمليات ودورات المعالجة بنسبة 98% و95% على التوالي. هذه النسب هي بالمقارنة مع أداء دورة التحسين السابقة لكل دورة. عند جمع جميع التحسينات مع بعضها البعض، تمت زيادة إنتاجية البرنامج بمقدار 13 ضعفا على الأقل مقارنة بالنسخة غير المتوازية.

جامعة البحرين

كلية تقنية المعلومات

”تحسين أداء أنظمة الحصول على البيانات في تجربة كاشف الميون العملاق باستخدام أدوات قياس أداء البرمجة المتوازية عند استخدام المعالجات الرسومية متعددة الأغراض“

أطروحة مقدمة كجزء من متطلبات الحصول على درجة الماجستير في هندسة البرمجيات

إعداد

عبد الله إبراهيم علي صباح 20103079

إشراف

د. أحمد محمد زكي

مشرف مشارك

د. عبد الله القدومي

مملكة البحرين

يونيو 2020