Graphics Processing Units: Abstract Modelling and Applications in Bioinformatics

Thesis submitted in accordance with the requirements of the University of Liverpool for the degree of Doctor in Philosophy by

Thomas C. Carroll

March 2020

Abstract

The Graphical Processing Unit is a specialised piece of hardware that contains many low powered cores, available on both the consumer and industrial market. The original Graphical Processing Units were designed for processing high quality graphical images, for presentation to the screen, and were therefore marketed to the computer games market segment. More recently, frameworks such as CUDA and OpenCL allowed the specialised highly parallel architecture of the Graphical Processing Unit to be used for not just graphical operations, but for general computation. This is known as General Purpose Programming on Graphical Processing Units, and it has attracted interest from the scientific community, looking for ways to exploit this highly parallel environment, which was cheaper and more accessible than the traditional High Performance Computing platforms, such as the supercomputer. This interest in developing algorithms that exploit the parallel architecture of the Graphical Processing Unit has highlighted the need for scientists to be able to analyse proposed algorithms, just as happens for proposed sequential algorithms.

In this thesis, we study the abstract modelling of computation on the Graphical Processing Unit, and the application of Graphical Processing Unit-based algorithms in the field of bioinformatics, the field of using computational algorithms to solve biological problems. We show that existing abstract models for analysing parallel algorithms on the Graphical Processing Unit are not able to sufficiently and accurately model all that is required. We propose a new abstract model, called the Abstract Transferring Graphical Processing Unit Model, which is able to provide analysis of Graphical Processing Unit-based algorithms that is more accurate than existing abstract models. It does this by capturing the data transfer between the Central Processing Unit and the Graphical Processing Unit. We demonstrate the accuracy and applicability of our model with several computational problems, showing that our model provides greater accuracy than the existing models, verifying these claims using experiments. We also contribute novel Graphics Processing Unit-base solutions to two bioinformatics problems: DNA sequence alignment, and Protein spectral identification, demonstrating promising levels of improvement against the sequential Central Processing Unit experiments.

Acknowledgements

I would like to express my gratitude to my supervisors Professor Prudence Wong and Professor Paul Spirakis; their support and guidance throughout my study has been invaluable. Extra thanks go particularly to Prudence, who has supervised my work since my time as an undergraduate student; she has been a true inspiration, and I feel privileged to have worked with her.

I would like to thank my examiners for their diligent work in examining this thesis, and for their recommendations which lead to the final version.

I would like to thank my many colleagues for their support and friendship, and for making the sometimes arduous struggle of PhD study more bearable. In particular, I would like to thank Reino and Josh.

Finally, I would like to thank my friends and family for their unending support and encouragement.

Contents

List of Figures

List of Tables

List of Algorithms

Nomenclature

δE	Gap extension penalty
δO	Gap opening penalty
δ	PTM shift in spectrum
Ti_i	Input transfer cost for round i
$\hat{I}_i$	Number of input transfers for round i
I_i	Number of words of input transfer in round i
λ	Global memory access cost
γ	Operation rate
To_i	Output transfer cost for round i
$\hat{O}_i$	Number of output transfers for round i
O_i	Number of words of output transfer in round i
$\mathcal{X} = X_0, X_1, ..., X_{r-1}$	Set of pattern sequences / experimental spectra
$X_i = x_0, x_1, ..., x_{m-1}$	Pattern sequence / experimental spectrum $X_i \in \mathcal{X}$
q_i	I/O of round i
R	Number of GPU rounds
t_i	Time of round i
α	Data transfer staging cost
σ	Synchronisation cost
$\mathcal{T} = T_0, T_1, ..., T_{q-1}$	Set of text sequences / theoretical spectra (library)
β	Cost of transferring single word
$T_i = t_0, t_1, ..., t_{n-1}$	Text sequence / Theoretical spectrum $T_i \in \mathcal{T}$

$ATGPU(b, k, M, G)$	An instance of an ATGPU machine with b cores on each multprocessor, k multiprocessors, M words capacity in shared memory, and G words capacity in global memory. Parameters can be left out if they are not needed: a single parameter would stipulate the number of cores on a multiprocessor, two parameters would stipulate the number of cores on a multiprocessor and the number of processors, and so on.
b	Number of cores in a multiprocessor
G	Size of global memory of ATGPU machine, in words
k	Number of multiprocessors
M	Size of shared memory of each multiprocessor, in words
$p = kb$	Number of cores on ATGPU machine

Chapter 1

Introduction

Well known English proverbs such as *'many hands make light work'*, and *'a problem shared is a problem halved'* aim to teach how teamwork can accelerate the completion of a task, or can make a problem that initially seems difficult or daunting appear more manageable. Families exhibit workload sharing or concurrent working, in order to complete the day's errands quicker; Alice collects the children from school, taking 20 minutes, whilst Bob buys groceries from the shop next door, taking 30 minutes. By working together, Alice and Bob completed the day's tasks in 30 minutes, which is an improvement on the 50 minutes it could have taken, if only one person completed both tasks in sequence.

The above example is indeed simple, yet demonstrates how sharing the workload between workers can bring about improvement in running time; it demonstrates a principle that is also present in computing: it is sometimes possible to break down a computing task into smaller independent parts to be computed concurrently, thus improving the running time.

In computing, a Central Processing Unit (CPU) is the component that executes program instructions, which can include arithmetic operations, logic operations, and memory accesses. The CPU contains various other controllers and cache memory spaces, as well as connections to the rest of the computer system. The unit which performs the arithmetic and logic operations is often referred to as a *core*. A CPU with a single core performs a single task at a time and switches between tasks in order to multitask - yet multitasking is not considered as truly parallel computing. In fact, multiple cores are required to perform tasks in parallel, which mirrors our introductory example – we required two people to be able to complete both tasks concurrently.

Over time, as the operational speed of the CPU has plateaued, parallel processing has become more prevalent in the quest of improving the running time of computational tasks. In addition to improvement of running time, other benefits of parallel processing have been sought, including greater energy efficiency and better temperature control. There have been many pieces of specialist computing hardware released that are capable of parallel processing, such as the multicore CPU, the Field Programmable Gate Array (FPGA) and the supercomputer. In this thesis, we investigate the specialised parallel hardware of the Graphics Processing Unit (GPU), which was initially designed for graphical applications, yet has also found widespread adoption in general purpose programming applications, including scientific application. We examine the theoretical analysis of parallel algorithms and how this applies to computer programs that are run on the GPU. We then study algorithms that use computational approaches to solve problems which occur in biology, in a field known as *Bioinformatics*; we see how the effective use of the GPU can improve running time, when a large amount of data is required.

1.1 The Graphics Processing Unit

The GPU is a type of parallel processing hardware on which groups of low-powered cores work concurrently. The GPU was born out of the graphics pipeline, and so was originally purposed for presenting graphical images to the screen. Available in both consumer- and commercial-grade versions, the GPU is widely used as a co-processor in many High Performance Computing (HPC) applications (known as General Purpose Programming on GPU (GPGPU)) in addition to their original use in computer gaming. The nVidia Compute Unified Device Architecture (CUDA) GPU framework [66] is widely used for scientific computing. Advanced Micro Devices, Inc. (AMD) is another major manufacturer of GPU devices, who champion the Open Computing Language (OpenCL) [37] heterogeneous computing framework. GPUs are commonly utilised in either of two settings: either as a discrete device in a workstation (the workstation may have 1 or more discrete GPU devices connected to the CPU by the Peripheral Component Interconnect Express (PCIe) bus), or as part of a cluster.

The massively parallel architecture of the GPU has proven to be useful in accelerating many different types of tasks, and has produced some very impressive speed-up results, when comparing to sequential implementations.

However, due to the specialised and unique nature of the GPU architecture, there is a large learning curve to be overcome, when a programmer first starts to learn how to use these pieces of hardware. This means that the implementation of efficient algorithms on the GPU can be challenging, and there is a need for abstract models to aid in the design of these efficient algorithms.

1.2 Analysis of Parallel Algorithms

When scientists design an algorithm, there is a desire to theoretically analyse it before any work on implementation takes place. They would look to answer questions pertaining to the expected running time and the expected storage space required, and in particular how this compares to existing algorithms. A common way to analyse a sequential algorithm would be to count the total number of operations required, memory accesses, or the amount of storage used, and then observe the trend as the input size increases. This would then give a fair way to compare two algorithms. However, parallel programs are split between multiple cores, often running on specialised architectures, meaning that simply counting the total number operations, the total memory accesses, or the total amount of storage used, in the same way as if it was run sequentially, would not always give a truly accurate analysis of the algorithm's performance.

There are many parallel abstract models in existence, making it possible for scientists to theoretically analyse and model an algorithm as if running in a parallel environment. Well known examples include the Parallel Random Access Machine (PRAM) [25], the Bulk Synchronous Parallel Machine (BSP) [83] and the Parallel External Memory Machine (PEM) [4], yet none of these parallel abstract models capture *all* elements required to effectively model program execution on the GPU, as the GPU has a special architecture and execution pattern . There are also several parallel abstract models that are designed specifically for analysis of GPU, namely the Abstract GPU Model (AGPU) [39] and Sitchinava Weichert GPU Model (SW-GPU) [77], which both capture different elements of GPU execution, but there are still elements of the execution that are missing from one or both of these models, namely the data transfer between the CPU and GPU. This means, that there is currently no parallel abstract model that captures *all* elements required to theoretically analyse GPU execution in its entirety (see Chapter 2 for more details).

1.3 Bioinformatics Problems

The Bioinformatics field looks to use algorithms and computational thinking to solve problems that occur in biology. In this thesis, we study two such problems: the sequence alignment problem, and the protein spectral alignment problem. More specifically, we investigate the design and application of bioinformatics algorithms on the GPU. Parallelism is exploited by nature everyday, and we are looking to exploit the many-core and highly parallel computational nature of the GPU to accelerate the computation.

1.3.1 DNA Sequence Alignment

The problem of finding alignment between two biological sequences (such as Deoxyribonucleic acid (DNA)) has been extensively studied, with the two most famous alignment algorithms being the Smith-Waterman algorithm [79] and the Needleman-Wunsch algorithm [65]. An alignment allows highlight of common areas between sequences, on the premise that homology between two sequences can show some sort of connection, or in the case of an unknown gene sequence, can indicate what gene the sequence is most related to. Roughly speaking, aligning a short pattern sequence to a longer text sequence is to determine whether the pattern exists in the text and if so the positions where it occurs.

With the advances in sequencing technologies, the amount of data that requires alignment has increased drastically. These DNA sequences are obtained using equipment known as *sequencers*. Sequencers take physical DNA samples, analyse them, and output the sequences of nucleobases of the DNA samples as character strings. The Illumina HiSeqX Ten sequencer is an example of a modern sequencer that can produce three billion reads (sequences) of length 250 bp (base pairs) in less than three days.

The *re-sequencing* problem is to assemble short reads produced by the sequencer into a genome sequence by referring to a reference genome, requiring “mapping” or “aligning” short reads back to reference sequences. The task is challenging due to the vast amount of data and the large genome sizes.

There is a wide range of short-read alignment tools available, e.g., Bowtie [44], BWA [46], GenomeMapper [74], MAQ [47], SOAP2 [48], SHRiMP [73], Stampy [57], REAL [28], addressing different aspects of the problem. Due to the data size, faster tools are needed. This asserts not just speed requirements on the processors but also leads to high power/energy requirements; furthermore, this potentially causes too high temperature that may damage the processors. To solve this problem, it is nowadays common to exploit multiprocessors such as the GPU. There are many alignment tools available, which use the GPU in order to achieve increase in speed and SOAP3 [52] is currently among the best GPU-based short-read alignment tools available.

Because of mutations and other biological mechanisms, it is common that sequences in comparison may not be exact match but may have some mismatches or gaps. It is important to take into account mismatches or gaps, otherwise some vital information may be missing. However, allowing mismatches or gaps greatly increases the complexity of the problem and algorithms detecting mismatches or gaps are significantly slower than their counterparts that detect exact matches. Existing short-read alignment tools including those mentioned above usually only allow at most a small number of mismatches or gaps because of this.

Therefore, there is a need for a short read alignment tool that utilises the GPU effectively, which allows mismatches and gaps in the computed alignments.

1.3.2 Protein Spectral Alignment

The problem of aligning two protein spectra is different to that of aligning two protein sequences, as a protein spectra is made up of numerical masses, as opposed to singular bases. Tandem Mass Spectroscopy (MS/MS) is routinely used in proteomic studies to measure sample protein data or sample peptide data, generating spectra (lists of mass peaks corresponding to the weight of fragments of the sample protein, which is broken up and measured in a mass spectrometer) which are then analysed by software tools in order to identify the input sample.

In MS/MS spectral identification, the spectrum of the sample data is searched against a database, computing similarity scores for the sample against each item in the database.

Within the life cycle of the protein modification can occur whereby molecules will attach onto the protein; this is known as Post Translational Modification (PTM), of which there are over 200 known types. PTM affects many areas of cell and protein functionality, such as tagging proteins for destruction, and altering cell metabolism. A simple comparison of the modified spectrum against the unmodified form is not sufficient, as the sample has undergone the PTM process, which adds molecules to part of the protein, thereby affecting the resulting spectrum data. PTMs therefore make identification of proteins and peptides more challenging.

As with sequence alignment, the amount of data that is created is ever increasing, yet there has been relatively little study into the effective use of GPU for the protein spectral alignment problem; there has been some work towards using GPU to accelerate some scoring schemes for the spectral alignment problem [6] [61], but efforts have more generally geared towards heuristic methods to reduce the search space, as opposed to looking to parallelise the calculations. There is therefore scope to study the effective use of GPU in computing the protein spectral alignment database search.

1.3.3 Research Questions

In this thesis we seek to answer the following two research questions:

1. Are we able to create a parallel abstract model that gives improved and accurate complexity analysis and predicted running time trend for GPU programs, compared to the analysis given by existing models?
2. Can new GPU-based algorithms improve on existing solutions to bioinformatics tasks within the Sequence Alignment problem and the Protein Spectral Alignment problem?

1.4 Contribution of Thesis and Author's Published Work

The contribution of this thesis fits into three areas:

1. A new parallel abstract model for GPU computation.
2. The proposal and study of a GPU-based sequence alignment tool called `GPUGapsMis`.
3. The proposal and study of GPU-based algorithms for the protein spectral alignment problem.

1.4.1 Abstract Transferring GPU Model

Our contribution consists of an abstract model, called Abstract Transferring GPU (ATGPU), which is an extension of previous models. We introduce new components to capture data transfer between the CPU and GPU.

We extend the SWGPU and AGPU architectures, introducing a size constraint on global memory, making the model more realistic. We extend the pseudocode of AGPU to capture data transfer, and we extend the SWGPU cost function to model data transfer and to simulate the cost on a particular GPU. To our knowledge, ATGPU is the first abstract model with this comprehensive array of analysis and design capabilities, and the first abstract GPU model to capture data transfer.

We demonstrate the use of ATGPU and evaluate several computational problems using the model. We show via experiments that existing models are not able to sufficiently model the actual running time in all cases, as they do not capture data transfer. We show that by capturing data transfer using our model, we are able to obtain more accurate predictions of the actual running time.

The model, along with the analysis and verification of vector addition, matrix multiplication, and reduction appears in the following published paper:

- Thomas C Carroll and Prudence W H Wong. An Improved Abstract GPU Model with Data Transfer. In *Proceedings of the International Conference on Parallel Programming Workshops*, pages 113–120. IEEE, 2017. doi: 10.1109/ICPPW.2017.28

1.4.2 Sequence Alignment Problem

Our contribution is a study of our proposed data-parallel GPU-based algorithm for the pair-wise sequence alignment problem with multiple gaps. The algorithm, which we call `GPUGapsMis`, is based on the GapsMis and GapsPos algorithms in [5]. GapsMis uses a dynamic programming approach to compute the gapped alignment of two sequences, by looking for the best gap insertion point at each step. GapsPos uses the information from GapsMis to then compute the optimal alignment of the sequences. We give analysis of `GPUGapsMis` on the Abstract Transferring GPU Model (ATGPU) model, and give analysis of of observed results with respect to the different approaches.

To achieve greater improvement over the CPU, we try to maximise the amount of parallelism by using appropriate data structures to store the data and hence decrease the I/O to shared and global memory, which could cause a bottleneck in performance. To allow flexibility of dealing with real data, we also extend the algorithm to allow the use of scoring matrix (which is a table allowing for customised scores by biologists) in addition to the Hamming distance that is considered in GapsMis [5]. We implement our algorithm and a modified version of the sequential algorithm GapsMis with the scoring matrix; we call the extended algorithm `CPUGapsMis`. We also enable the functionality to compute the optimal alignment, as in GapsPos [5], and investigate using a Hybrid backtracking method and a GPU backtracking method. Further to this, we investigate allowing a single text and multiple text sequences to be aligned on the device at one time, with different batching methods.

We compare the performance of `GPUGapsMis` and `CPUGapsMis` and the peak empirical speedup achieved on our system (detailed in Table A.1) is 11× in computing the alignment score matrix, and 10.4× when backtracking is also computed (this peak result is achieved for a particular input size, with more details in Chapter 5). We show that by lowering the amount of communication and data transfer between the GPU and CPU, we are able to yield the most improvement. We also show that despite the backtracking being sequential and inefficient on the GPU (when compared to performing the backtracking on the CPU), it is more beneficial to perform this on the GPU, rather than returning to the CPU for performing the backtracking.

The GPU-based algorithm `GPUGapsMis`, along with analysis on the AGPU model and comparison against sequential results appears in the following published paper:

- Thomas C Carroll, Jude-Thaddeus Ojiaku, and Prudence W H Wong. Pairwise Sequence Alignment with Gaps with GPU. In *2015 IEEE International Conference on Cluster Computing*, pages 603–610. IEEE, 2015. doi: 10.1109/CLUSTER.2015.10

The extended study, incorporating the backtracking and the various batching approaches appears in the following accepted journal manuscript:

- Thomas C Carroll, Jude-Thaddeus Ojiaku, and Prudence WH Wong. Semiglobal Sequence Alignment with Gaps using GPU. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2019. doi: 10.1109/TCBB.2019.2914105. (To Appear)

Further to this, we also give analysis on the ATGPU model, showing that it is able to distinguish between two similar GPU-based approaches, which differ only in the data transfer requirements. We show that this is not shown on existing models, and that the ATGPU model gives a more accurate analysis of the algorithm.

1.4.3 Protein Spectral Identification

We investigate using the GPU to accelerate and solve the Match Score Identification problem, which computes similarity between a database of theoretical known spectra, and a set of experimental modified spectra. This particular algorithm has been shown to perform well against existing tools, maintaining accuracy levels and decreasing running time for identification of spectra.

We discuss how the best performing CPU implementation of this algorithm is not the best performing on the GPU as it does not use the GPU resources effectively, and how modification of a different sequential approach is required in order to effectively exploit the GPUs resources. We propose the algorithm GPU-MSI, which solves the match spectral identification on the GPU. We give theoretical analysis of GPU-MSI and verify performance of GPU-MSI using experiments, showing that GPU-MSI achieves promising level of speedup, with upto 22$\times$ speedup (for a particular input size) compared to the best performing CPU implementation on our system. (see Chapter 3 for more detail).

In addition to the protein spectral identification problem, the Match Spectral Identification algorithm is also suited to be used as a filter for heuristic approaches to the spectral alignment problem, which means that this tool can also be built into a pipeline to further speed up other existing tools.

1.5 Thesis Outline

The outline of the rest of the thesis is as follows:

In Chapter 2, we discuss and review the GPU architecture and execution of programs on the GPU. We then proceed to review areas where GPU acceleration has proven to be useful. We discuss classical parallel abstract models and GPU-specific parallel abstract models, comparing them, and discussing how well suited they are to analysing execution of programs on the GPU. We then review other analytical tools designed to provide predictive information on how an existing GPU program should run, before reviewing how GPU-CPU data transfer can affect the running of programs, and the efforts towards its analysis.

In Chapter 3 we begin by discussing the background of DNA sequence alignment, introducing the reader to the common forms of alignment, and discussing the tools available. We then proceed to review the use of parallel processing within the sequence alignment problem, before introducing and reviewing the *Semi-global sequence alignment with bounded number of gaps* problem, which the work in Chapter 5 is based upon. The chapter then moves the discussion toward the protein spectral alignment problem, introducing the reader to the alignment of protein spectra obtained from MS/MS. We review the methods used for spectral alignment, and introduce the reader to the heuristic methods and the pipe-lining that is used to speed up the process when dealing with large amounts of data, as well as discussing the existing efforts that use the GPU to improve running time. Finally, we introduce the reader to the *match spectrum identification* problem, on which the work in Chapter 6 is based.

Chapter 4 introduces our newly proposed GPU abstract model, called the ATGPU. The chapter starts by highlighting the scope within the existing literature for an improved GPU parallel abstract model. We then introduce the architecture of the model, followed by the components of analysis that are included in the model, drawing comparison against the existing models from the literature. We then demonstrate the usefulness of the model compared to existing models with theoretical analysis and empirical results of several different algorithms.

Chapter 5 presents a study into the DNA sequence alignment problem, where we propose a new GPU algorithm for the semi-global sequence alignment with multiple gaps problem. The chapter first highlights the need for an improved GPU-based algorithm for this problem from within the literature, before giving the theoretical problem definition. We then introduce the GPU-based algorithm, followed by theoretical analysis of the proposed algorithm on the AGPU and ATGPU models. The experimental setting, along with the approaches that are used for the experiments are then given, before the analysis of empirical results, comparing to existing GPU and CPU implementations are discussed.

Chapter 6 presents a study into the match spectral identification problem. The chapter first gives analysis of the existing sequential approaches for solving this problem. After that, it proceeds to consider how these sequential solutions would map onto the GPU, showing why particular solutions are not feasible for the GPU, despite them being faster on the CPU. Then, the algorithm `GPU-MSI` is introduced; analysis is given on the ATGPU model. We then verify performance and the ATGPU analysis is using experiments, showing that `GPU-MSI` outperforms all sequential implementations.

Finally, Chapter 7 gives concluding remarks and directions of future work.

Chapter 2

Graphics Processing Units and Parallel Architectures

2.1 Introduction

In this chapter, we begin by describing the nVidia Compute Unified Device Architecture (CUDA) framework (consisting of the GPU hardware architecture, memory access model, execution model, and programming model) which is implemented on nVidia GPUs, which are used in this thesis. We then review the many different ways in which Graphics Processing Units (GPUs) have been effectively used as an accelerator, demonstrating its applicability to many real-world problems. We then proceed to discuss the differences between sequential algorithms and parallel algorithms, in the theoretical sense, and review several parallel abstract models. With these parallel abstract models, we demonstrate that they are not suitable for the effective and accurate modelling and analysis of algorithms designed for the GPU. In light of this, we review two parallel abstract models that are designed specifically with the analysis of GPU algorithms in mind, and we refer back to the CUDA framework to demonstrate that these models also miss key elements of the GPU execution model, meaning that they are unable to provide a full and accurate picture for analysis. We then look at another class of GPU algorithm analysis tools, namely predictive analytic tools, which are software based, requiring program code to be written and interpreted or compiled, in order for analysis to take place. We argue that these tools are often complicated and that the usefulness of these particular tools is apparent at a later stage in the software development cycle, than the stage when we look to use abstract models, meaning that they are complimentary to any abstract model, but not a replacement for them. Finally, we focus our attention on data transfer between the CPU and the GPU, which is one of the first steps of the GPU execution model. We argue that ignoring the data transfer between the CPU and GPU can provide inaccurate experiment results, as they are some of the most expensive operations in the whole GPU execution model, and that by optimising the data transfer between CPU and GPU (in addition to the kernel code), can a GPU program garner much improvement.

2.2 Graphics Processing Units (GPU)

The GPU is a specialised piece of hardware that contains many low-powered cores designed for parallel processing of high-quality graphical images, for example in computer games and photo editing software. The nVidia Compute Unified Device Architecture (CUDA) framework consists of numerous libraries, an Application Programming Interface (API) and a programming language, which allows general purpose programming and solving of many computational non-graphical problems on nVidia GPUs, which are used for experimental work within this thesis. In this section we describe the nVidia GPU hardware architecture [66] [1]. In addition to the GPU hardware architecture, we describe the CUDA framework with regards to the execution model, the memory access model, and the programming model.

2.2.1 GPU Hardware Architecture

The GPU sits as a discrete peripheral device on the machine, connected to the Central Processing Unit (CPU) by the Peripheral Component Interconnect Express (PCIe) bus. The GPU is made up of many low powered cores, arranged in groups on *Streaming Multiprocessor*s (SMs), of which each GPU has several. The GPU will run many threads in parallel on the cores, which is discussed further in Section 2.2.2. The GPU has a large amount of slow off-chip memory, known as *global memory*; Global memory is accessible by threads on all cores on the GPU and by the CPU; it is normally in the order of gigabytes in size, and organised in fixed size *memory blocks*. All data allocated on the global memory are aligned to a memory block, and all memory requests from global memory are served at the memory block level, i.e: a request for a single value would return the whole memory block in which the value resides. Memory requests from threads within the GPU are queued globally and served by one of the memory controllers.

[1]other GPU manufacturer's architectures (such as those by Advanced Micro Devices, Inc. (AMD), and competing APIs such as Open Computing Language (OpenCL) are similar

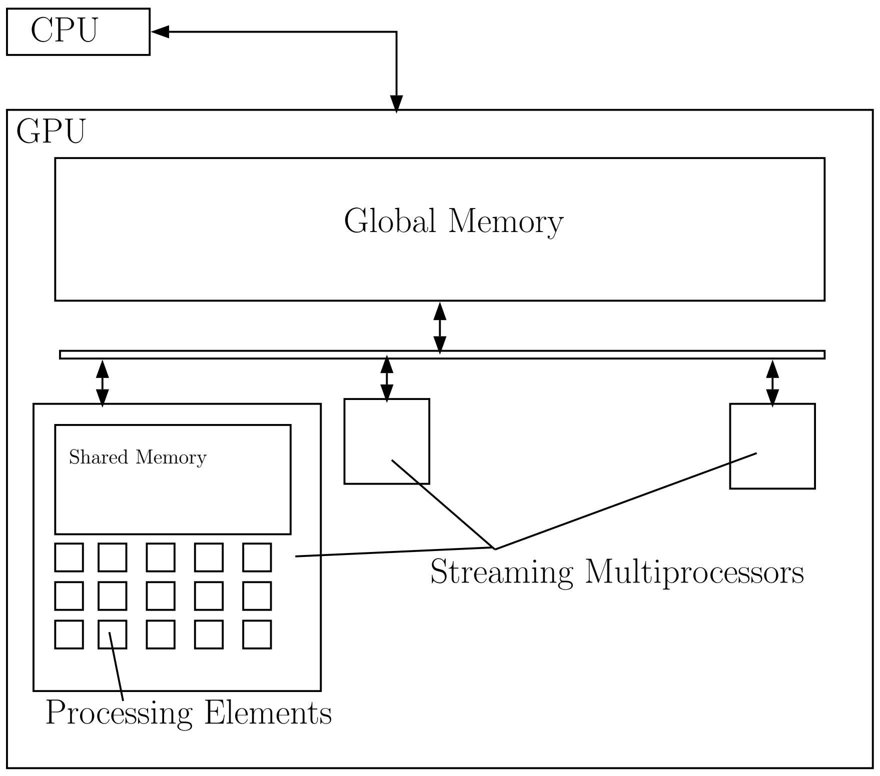


Figure 2.1: Conceptual view of the GPU architecture

Each SM contains a small amount of fast on-chip memory, known as *shared memory*, accessible only to cores on that particular SM, and in the order of kilobytes in size. The shared memory is split into several *banks*, the size of each bank corresponding with the memory block size. Each bank can serve one memory request in unit time, meaning that if threads request values from distinct banks, then the request will complete in unit time, yet if they request values from non-distinct banks, this is known as a *bank conflict*, and the requests are serialised. Part of the shared memory is reserved as private register space for the cores. Part of the global and shared memory is reserved for *constant memory*, which is heavily cached by the hardware and designed for quick access to program constants. Figure 2.1 demonstrates the GPU architecture.

2.2.2 GPU Programming and Execution

A programmer writes *kernels* (analogous to functions) in the CUDA C programming language, and uses the CUDA API to send both the compiled kernel instructions and the required data to the GPU. A programmer first specifies the launch configuration of the kernel, as a *grid* of *blocks* of threads on the GPU. First, the program data and input data is sent to the GPU over the PCIe bus. These data transfers are often the slowest and most costly operations of the entire program. After the data and kernel have been transferred, the execution of the kernel will begin.

A *thread block* is a collection of threads which work in cooperation and are run on a single SM, in a single instruction multiple data (SIMD) fashion, with inter-thread communication only possible via shared memory, accessible only to the threads of the thread block. The thread block *conceptually* runs concurrently, yet in reality is divided into *warps*, which are arrays of 32 threads, each run in lock step with one another. The instructions of the kernel for each warp are placed in an instruction queue, and are scheduled for execution on lanes of CUDA cores. Once the instruction has executed, there may be the need to *wait* on a shared memory request or a global memory request. Once the request has been serviced, the next instruction is ready to be scheduled for execution. When a shared memory request is placed by a warp, it is serviced in unit time should each address be within distinct banks. If this is not the case, then a *bank conflict* occurs, and the request is serialised by the hardware into as few non-conflicting requests as possible. When a global memory request is placed by the warp, then it is put into as few memory-block-wide transactions as possible. If all requests by the warp are for addresses within the same memory block, then this is serviced by a single transaction, this is known as memory *coalescing*. Accessing global memory is very expensive, taking up to 800 cycles per memory block requested, therefore it is wise to access global memory with as much coalescing as possible, otherwise the global memory access can throttle a program's performance.

Once an operation has been executed by a warp, the next instruction (possibly from a different warp) in the instruction queue is then scheduled for execution. It is possible to have multiple thread blocks resident on a single SM, provided there are enough shared memory resources for them to execute, with each GPU having also an upper limit to the amount of concurrent thread blocks on an SM. If a program is able to hold the maximum number of blocks on each SM, it is said to have full *occupancy*. Full occupancy ensures there is an increased number of warps available to execute whist other long-latency operations (such as global memory requests) are being serviced, which will go to hide the latency of these said long latency operations. Due to the way that the instruction queue is populated with ready-state instructions, it is important to ensure that each warp is independent of the rest. It is possible to synchronise the threads within the thread block, using barrier operations. Likewise, blocks must be independent of one another; currently the best way to synchronise blocks is to terminate and then relaunch

Table 2.1: CUDA thread indexing example

Block ID	0							1						2	
Local ID	0	1	2	3	4	...	31	0	1	2	3	...	31	0	...
Global ID	0	1	2	3	4	...	31	32	33	34	35	...	63	64	...

the kernel.

2.2.3 Applications of The GPU

We discuss later in Chapter 3 the application of GPU to bioinformatics problems, but it is also important to note that the GPU has been applied to many types of computational problems, a few examples of which are given here.

The authors in [10] study a dynamic programming method for knapsack problems, achieving a 15× speed-up with experiments using a single GPU, and 30× speed-up with experiments using 2 GPU devices, when compared to CPU control experiments. The authors in [11] develop a GPU-based Markov clustering algorithm for operations on sparse matrices, with application to bioinformatics. The authors in [19] develop a GPU-based dynamic programming algorithm for robot path planning in a multiagent environment, with speed-up of an order of magnitude. The authors in [21] develop a GPU algorithm to compute graph diameter, demonstrating experiments achieving 21× speed-up, compared to CPU control experiments. CAMPAIGN [38] is a library of GPU-optimised clustering algorithms. The authors in [63] design efficient sorting algorithms for the GPU, publishing the fastest merge sort GPU algorithm, at the date of publication. The authors in [80] develop a GPU tool for generic parallelization of certain algabreic dynamic programming tasks, such as RNA folding. They demonstrate experiments which achieve speedup of between $6\times - 25\times$, dependant on the application.

2.3 Classical Abstract Parallel Models

When scientists design algorithms, they often write pseudocode and will perform theoretical analysis on the algorithm, to find out how the performance of the algorithm will handle an increase in input size. This analysis ususally focusses on Big-Oh notation, with regards to running time and storage space required. An algorithmic model helps us to therefore formalise this process; the RAM algorithmic model can help us to encapsulate the fundamental aspects of computing, such as arithmetic operations,

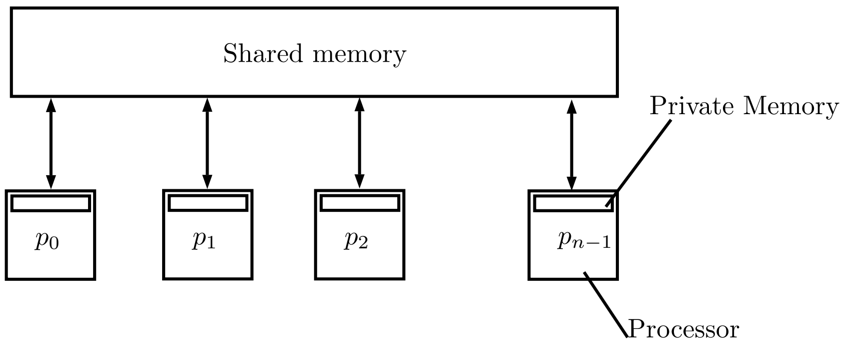


Figure 2.2: Conceptual diagram of the Parallel Random Access Machine (PRAM) and the Bulk Synchronous Parallel Random Access Machine (BSPRAM) machines showing the processors with the shared memory unit.

loops, memory operations, and so forth. Using Big-Oh notation with the RAM model assumes that each operation takes unit time, and that the number of operations then relates directly to the run time. However, when we start to consider parallel architectures with multiple processors, the RAM model loses its usefulness. When we consider what is required of a abstract parallel model, we look for usability, portability, and predictability; is it easy to use? do the algorithm analyses hold on multiple hardware? can the model predict performance? The RAM model is unable to capture concurrent execution of parallel architectures, therefore can not correctly model the performance of a parallel algorithm. There have been many parallel algorithmic models, some of which we review below.

2.3.1 Parallel Random Access Machine (PRAM)

The *Parallel Random Access Machine* (PRAM), proposed by Fortune et al. [25] is a shared memory model, containing synchronous processors with their own private memory, and shared memory which is accessible to all (Figure 2.2). As the processors are synchronous, they all run on a common clock, i.e. in *lockstep* with one another. Communication between the processors is only possible by *read* and *write* operations to the shared memory unit.

When read/write operations occur in the PRAM, there is the chance that multiple processors attempt to access the same memory address at the same time. This is known as a *simultaneous read/write*; there are several variants of the PRAM that dictate how this situation is handled:

Exclusive Read Exclusive Write PRAM (EREW) does not allow simultaneous access to a distinct memory location.

Concurrent Read Exclusive Write PRAM (CREW) allows multiple processors to concurrently read the same location, but only allows a single processor to write to a memory location at one time.

Concurrent Read Concurrent Write PRAM (CRCW) allows multiple processors to read and write to a single memory location at the same time. The way the multiple write operations are handled is dictated by three sub-variants:

- **Common CRCW** concurrent writes can only occur if **all** processors attempt to write the same value.
- **Arbitrary CRCW** allows an arbitrary processor to succeed in writing to the memory location.
- **Priority CRCW** allows the processor with the lowest ID to succeed in writing to the memory location.

Analysis of algorithms on the PRAM are similar to that of the RAM model, involving the counting of work, processors, and communication. Efficiency and speed-up of algorithms on PRAM can be calculated when comparing the relationship between the best sequential time complexity, the time complexity on one PRAM processor, and the total work carried out by the PRAM for a particular algorithm.

The PRAM is a well adopted and well studied parallel algorithmic model, which captures many important parameters of parallel programs, such as allocating work to processors and counting work at each time step.

When we consider the sub-variants of the PRAM, the GPU accesses memory in an Arbitrary CREW manner, where in the case of multiple cores reading an address simultaneously, a broadcast occurs; in the case where multiple cores write to the same address simultaneously, then an arbitrary core will succeed.

The PRAM models computation in many ways differently to how a GPU executes in reality. Firstly, the architecture of the processors does not match the GPU, as the streaming multiprocessor is not captured. Secondly, the memory architecture does not match, meaning bank conflicts and coalesced memory operations would not be captured. The concept of a warp is not captured on the PRAM, meaning that latency hiding operations can not be captured, and finally, data transfer and synchronisation operations between the CPU and GPU are not captured. This means that the PRAM does not capture everything that is needed to accurately model GPU computation.

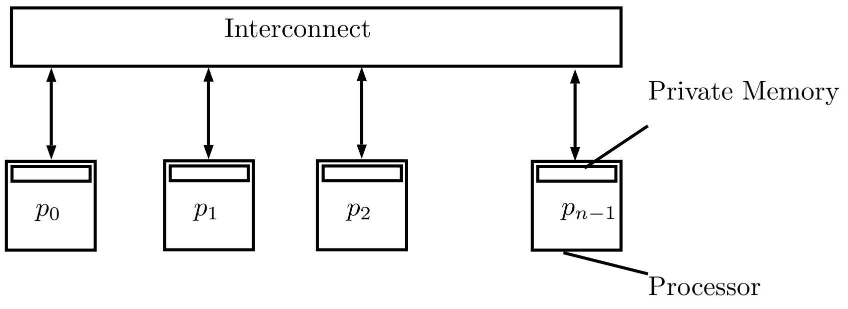


Figure 2.3: Conceptual diagram of the BSP machine, with processors $p_0, ..., p_{n-1}$.

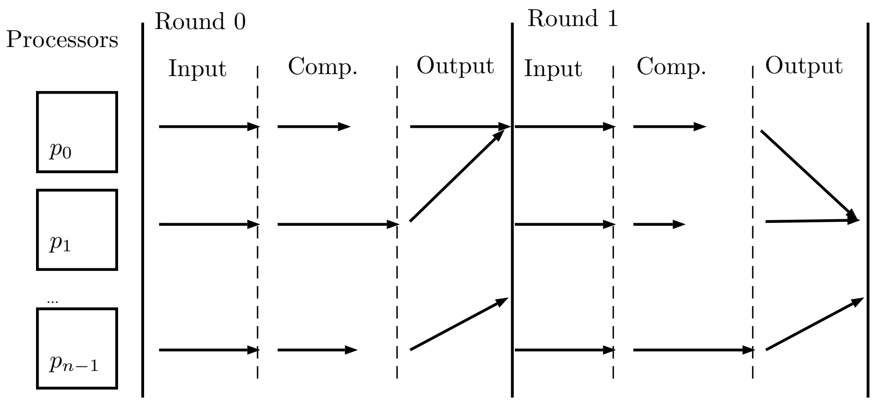


Figure 2.4: Execution model of the BSP and BSPRAM machines, showing the communication (input/output), computation and synchronisation (between rounds) steps. Each arrow represents a computational thread on a processor, and related memory transactions.

2.3.2 Bulk Synchronous Parallel Machine (BSP)

The Bulk Synchronous Parallel Machine (BSP), proposed by Valiant [83], is a distributed memory model, consisting of a machine and a cost function. The machine contains interconnected processors with their own private memory. A processor accesses its own memory with low latency, and accesses another processor's memory with higher latency. Algorithms are executed in rounds consisting of steps of computation, communication, and synchronisation. Algorithms are analysed with a cost function, which is a function taking the longest running processor at each computation round, the number and size of communications at each communication step, and the cost of synchronisations at synchronisation steps.

The BSP's concept of steps and rounds works in a similar way to how a warp functions on a GPU: all threads in the warp do some computation and wait until all are complete, however, computation is asynchronous between processors and processors can run distinct programs. In a similar way to the computation step, all processors perform memory access operations and wait until all are complete, as happens in a GPU warp. Synchronisation operations function in a similar way, except with a GPU, the synchronisation is across all threads in the block. Global synchronisation is only possible by termination of execution and kernel relaunch. It is also the case that synchronisation must happen on the BSP in every round, whereas it is not the case when we consider GPU execution.

The memory structure of the BSP is quite different from that of the GPU. The BSP allows direct communication between processors by means of accessing memory contents of another processor, whereas the GPU does not allow direct one-to-one communication between processors; threads must communicate via the shared memory, which is not present on the BSP, neither is a global memory unit present of the BSP.

The rounds and steps of the BSP mean that it captures something of GPU execution that the PRAM did not, yet the memory structure of the BSP as well as the intra-processor communication means that it is unable to effectively capture elements required to model a GPU. Additionally, the authors in [8] examine the original BSP cost function and show that the predictability of the cost function can be improved.

2.3.3 Bulk Synchronous Parallel Random Access Machine (BSPRAM)

Tiskin [82] proposed the BSPRAM, which is an extension of the BSP with elements of the PRAM. The machine consists of processors with fast private memory, and shared memory accessible to all. The architecture is identical to the PRAM, as shown in Figure 2.2. The execution runs in rounds, similar to BSP, as shown in Figure 2.4. The difference between the BSP and the BSPRAM is that the communication between processors happens in the shared memory unit, rather than via the interconnect, meaning each round begins with input from the shared memory, ending in output to the shared memory.

The BSPRAM is closer to the GPU than the PRAM and the BSP, in that there is a round-based execution and communication between processors via memory, as opposed to explicit intra-processor communication. However, the BSPRAM has asynchronous processors (as with the BSP) and each processor can run a different program (as with both the BSP and PRAM), meaning that the processors working in cooperation in a warp is not captured here in the BSPRAM. Additionally, access to memory does not follow the same particulars as the GPU, in that the memory architecture does not match,

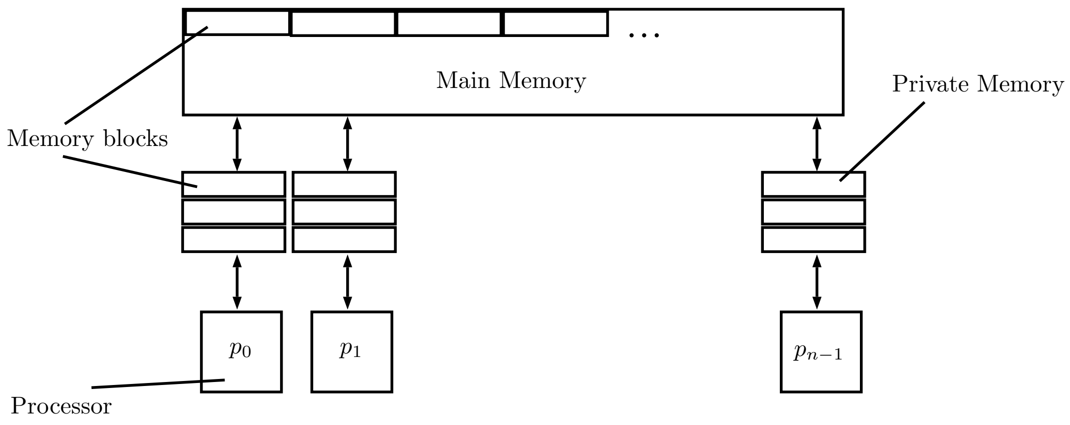


Figure 2.5: The Parallel External Memory Machine

and coalesced access or bank-conflicts are not modelled in the BSPRAM. Therefore, the BSPRAM is not suitable for modelling GPU computation.

2.3.4 Parallel External Memory Machine (PEM)

The Parallel External Memory Machine (PEM), proposed by Arge et. al. [4], contains processors and a formal memory hierarchy; each processor has private memory, and there is main memory accessible to all. Both memories are partitioned into equal sized memory blocks. Figure 2.5 demonstrates this architecture.

Processors cannot directly communicate in the PEM, and must communicate via read and write operations to the main memory. Memory transactions transfer entire memory blocks from the main memory to the private memory of a processor, and processor cannot use the data until it is in their private memory. The complexity analysis of algorithms on the PEM is by their *I/O complexity*. Algorithms are analysed not on the raw number of memory transactions from main memory to the private memories, but on the number of *parallel* memory transactions, i.e. n processors each transferring a single distinct block from main memory to their individual private memories has an I/O complexity of $O(1)$, rather than $O(n)$.

The two level memory hierarchy of a private memory and global memory, together with the memory transfers having the granularity at the memory block level is the same as how the GPU manages transactions between global memory and shared memory of the SM. However, attached to each of these private memories is a single processor, whereas the SM of a GPU has multiple processors working in cooperation, meaning that the SIMD operation of an SM is not captured by the PEM. Further to this, each processor in the PEM can execute a different program, which is not possible on the GPU. Therefore, the PEM is not able to capture everything needed to effectively model computation on the GPU.

2.4 Modelling and Analysing GPU Computation

As detailed in Section 2.3 there are no classical parallel models that capture everything required to model computation on the GPU. There has been considerable recent effort towards modelling computation on the GPU. Recent progress on modelling GPU computation has come in two main areas: analytical or predictive tools, and abstract models. Abstract models look to analyse pseudocode before any actual computer code has been written, providing information on how the pseudocode *is expected to run* if it is to be implemented. On the other hand, the software aided analytic and predictive tools analyse code that has already been written and compiled, meaning that the two categories of effort occupy different areas of the software design process.

2.4.1 Abstract Models

Two prominent abstract models for GPGPU are the Abstract GPU Model (AGPU) [39], and an unnamed model proposed by Sitchinava and Weichert [77], which we refer to as the Sitchinava Weichert GPU Model (SW-GPU). Both models share the same abstract architecture, which is shown in Figure 2.6, and the same execution model; the difference in the two models comes in the design and analysis of algorithms. We now discuss both the AGPU and SW-GPU in detail.

The models capture a host (CPU) and a device (GPU). The device contains a conceptually unlimited amount of global memory (split into memory blocks of b) and k Multiprocessors (MPs). Each MP contains b cores and M words of shared memory capacity, which is split evenly into b memory banks. The global memory can be accessed by the host and by all cores on the device. The shared memory is accessed only by cores on the SM.

A GPU-based algorithm runs in parallel on the cores of MPs; cores within an MP all perform the same instructions at the same time (i.e. in *lockstep*), therefore modelling the concept of a warp. Global memory requests transfer an entire memory block between global memory and the shared memory of the particular MP. If requested addresses are within the same memory block, this completes as a single operation, otherwise, multiple operations are required; this therefore models coalesced global memory access.

Shared memory requests complete in constant time provided requested addresses are in distinct banks, otherwise the requests are serialised; this models bank conflicts in shared memory. Shared memory requests are assumed to be bank conflict free, as bank conflicts are difficult to analyse. The MP waits until all memory requests by the cores have been resolved before proceeding to the next instruction.

Abstract GPU (AGPU) Model

Koike and Sadakane [39] proposed a theoretical model for GPUs called the AGPU. Using the AGPU, it is possible to design algorithms using the specific pseudocode and to analyse the asymptotic computational complexity of GPU algorithms. We discuss the specific pseudocode as part of the Abstract Transferring GPU Model (ATGPU) in Chapter 4, and we discuss the analysis of algorithms on the AGPU below.

In the AGPU model, GPU algorithms are measured by the following metrics:

- Time complexity
- I/O complexity
- Global memory space complexity
- Shared memory space complexity

The *time complexity* measures the number of instructions each multiprocessor executes. Should there be thread divergence within a multiprocessor, all paths are counted for the time complexity. Where the time complexity of multiple multiprocessors vary, the largest complexity is used, meaning the time complexity is not a sequential measure, but a parallel measure of the time complexity.

The *I/O complexity* measures the total number of global memory blocks accessed by all multiprocessors. Because the amount of parallelism for memory requests to be fulfilled is dependent on the bandwidth of the architecture, the I/O Complexity is defined as the summation of all global memory block requests from all multiprocessors.

The *global memory space complexity* measures the global memory usage, in words, of the algorithm. Likewise, the *shared memory space complexity* measures the shared memory usage, in words, of the algorithm. If the amount of shared memory used varies amongst the MPs, the largest value is taken.

The AGPU does not explicitly capture synchronisation (though analysing multiple AGPU algorithms in succession would go some way to capture this metric), and disallows algorithms where shared memory used exceeds capacity. Occupancy is measured as a function of shared memory usage and shared memory capacity. The AGPU gives pseudocode for designing algorithms on the model.

The AGPU was successfully used to design new GPU-based algorithms that were faster than the CUDA library implementations, meaning the AGPU is shown to be a well suited model for designing and analysing GPU algorithms; the pseudocode gives the designer a direct link to the architecture, making it easier to appreciate how the operations will affect the running of the program, and allowing for easier translation into CUDA or OpenCL code. However, the AGPU does not convey the true cost of accessing the global memory on the GPU, which is often orders of magnitude more expensive than a regular compute operation, in terms of clock cycles until completion. Likewise, the model does not take into account any host device communication, which is an important and often overlooked aspect of using the GPU as a coprocessor for accelerating a computational task.

SW-GPU Model

Sitchinava and Weichert proposed an abstract GPU model[77], which we refer to as the SW-GPU model. The SW-GPU does not provide a pseudocode, but analyses algorithms using a cost function. The SW-GPU models execution in rounds, delimited by explicit synchronisation with the host. The cost function uses metrics very similar to the ATGPU:

- Number of rounds
- Number of operations performed by the MPs
- Number of memory requests
- Synchronisation with host

In a GPU algorithm on the SW-GPU, there are R *rounds*. For a given round i, The *number of operations performed* by the MPs is denoted as t_i. Should there be thread divergence within a multiprocessor, all paths are counted. Where the number of operations by multiple MPs vary, the largest is used, meaning the number of operations is not a sequential measure, but a parallel measure. The *number of memory requests* measures the total number of global memory blocks accessed by all MPs in round i, and is denoted as q_i. The memory access operations on a GPU are very expensive, so each operation is assigned the cost λ. Finally, *synchronisation* with the host is also a costly operation, which is assigned the cost σ.

Therefore, the cost of a GPU algorithm on the SW-GPU model is given by Equation 2.1.

$$\sum_{i=1}^{R}(t_i + q_i\lambda + \sigma) \tag{2.1}$$

The SW-GPU is effective for modelling GPU computation, which is demonstrated in Chapter 4. However, like the AGPU, not every element of GPU execution is captured. For someone to be able to effectively design *and* analyse a GPU algorithm, then a pseudocode, such as that provided by the AGPU is useful - as it stands, the SW-GPU focusses purely on the analysis of an algorithm, and does not help with the design aspect. We discuss in Chapter 4 the scope for an improved abstract model for GPU computation.

2.4.2 Analytical or Predictive Tools

The abstract models discussed in Sections 2.3 and 2.4.1 analyse algorithms as they are designed on paper, not requiring the implementation of any code; This is useful for comparing two ideas. The analytical and predictive tools that are now described in this section analyse code that has already been implemented, and look to make conclusions on the performance or expected performance of the code on the GPU.

Hong and Kim [34] create a predictive tool which can be built into compilers. The tool predicts kernel latency at compile time by analysing the compute and memory access operations of the compiled code, yet the model is not simple, as many calculations are required. Their model predicted the observed latency to within a 5.14% error.

Konstandinidis et al. [40] propose a method to predict performance of compute bound or memory bound kernels, using the Quadrant-split method as a visualisation technique. The model is for use on developed kernels, and uses the profile of the device and the operations within the kernel to predict the performance on a particular device. Once the execution on one GPU has been profiled, the model is then successfully used to predict the performance on other GPUs, reporting a difference between predicted and observed results within 25.8%, averaging an error of 10.1%.

The authors in [41] propose a GPU time prediction model that is shown to give good and accurate results, however it does not have a defined pseudocode, is complicated (as in it needs very exact program instructions), and does not take into account the data transfer between GPU and CPU.

These tools are able to provide extra insight into developed GPU programs, however they are of little use during the early design stage, and therefore do not compete with or remove the need for abstract models for modelling GPU computation.

There have also been efforts that look to improve GPU performance on-the-fly for a running program, yet this is beyond the scope of this thesis. GPU programs execute in lockstep, yet *if-statements* can cause branch divergence, where some threads evaluate to true and some evaluate to false, so causing two branches of execution. The authors in [30] identify that branch divergence causes performance degredation and develop a system that reconfigures the warps of the execution on the fly, to then reduce the divergence that was introduced.

2.5 CPU GPU Data Transfer

When a GPU program executes, once the environment has been set up, the first step is to transfer program data and input data from the CPU to the GPU. When a GPU kernel has finished executing, signalling either the end of a particular stage of the program, or the end of the program all together, there must be synchronisation operations and data transfer operations that take place, either transferring intermediate data between the CPU and GPU, or transferring final output data. The size of this data transfer can be very large for some applications, for instance in a matrix multiplication application where two $n \times n$ matrices are multiplied into a single $n \times n$ matrix; For sufficiently large values of n, this data transfer requirement could reach into the gigabytes in size.

Data transfer between the CPU and GPU has been shown to affect the performance of a GPU program under normal usage. Gregg and Hazelwood [32] demonstrate that data transfer between CPU and GPU can affect reported performance, and argue that when reporting results, the GPU speed-up should include time taken for data transfer operations. Martin et. al. [60] study an n-body simulation on the GPU, with relation to the impact that data transfer between CPU and GPU has on the applicaiton performance. They conclude that it would be helpful to scientists to be able to quantify and estimate beforehand how much of an impact the data transfer would have on performance of GPU applications. The authors in [35] demonstrate that CPU-GPU communication can be detremental to the performance of the GPU program, and develop a tool that automatically optimises the communication between the two.

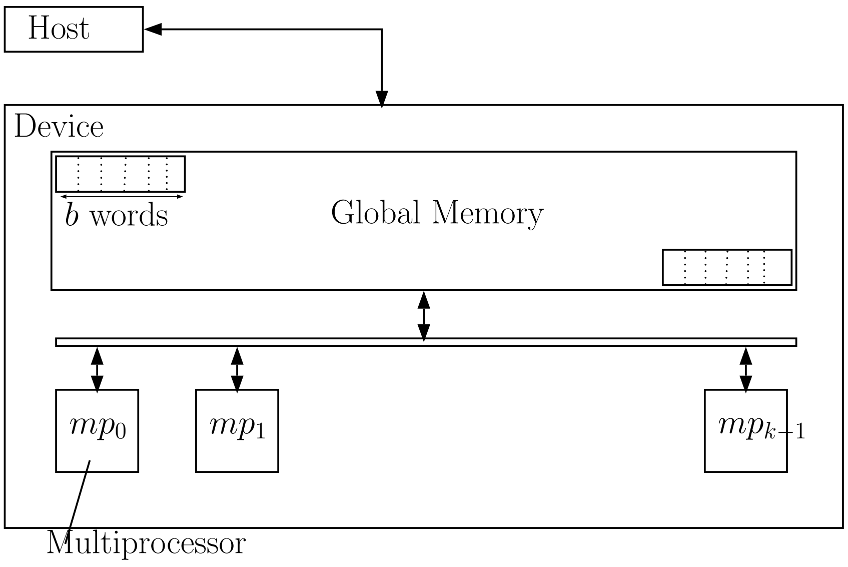
A bottleneck was experienced in [36] transferring data between CPU and GPU. The authors demonstrate that reducing the overhead of copying data between the CPU and GPU can significantly improve the perfomance of the application. In their paper, they demonstrated a performance increase of 33% by mapping the system memory to that of the GPU, thereby reducing the overhead.

Several techniques have been proposed to find the best technique for transferring data between CPU and GPU. Fujii et al. [29] identify that direct memory access, where the GPU and CPU share a unified address space, offers the best performance for large amounts of data transfer.

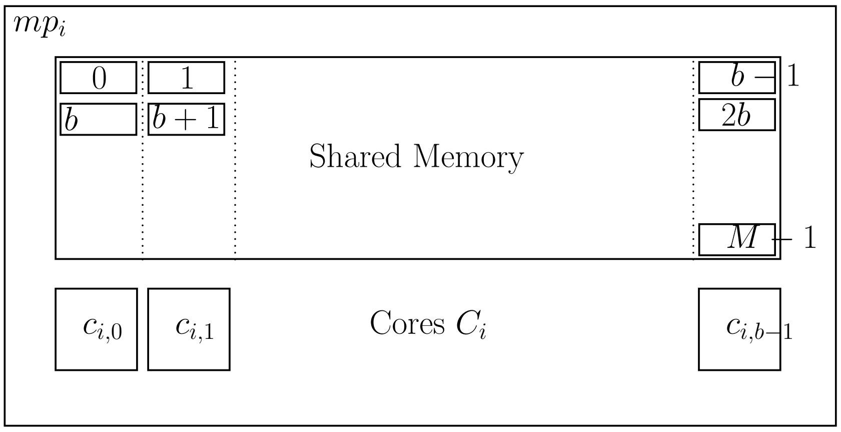
There has been many attempts to model CPU GPU data transfer using both software based tools and cost functions, yet to our knowledge, there are no existing works which look to model CPU GPU data transfer and execution in the same tool. Van Werkhoven et al. [85] produce an analytical tool modelling the data transfer and predicting the best data transfer technique for a given GPU program, as it is generally not feasible to program and test all techniques of data transfer between the CPU and GPU. Boyer et al. [9] propose a function to predict latency of data transfer operations. Their function lowered the difference between predicted and observed speed-up from 255% to 9%.

2.6 Conclusion

In this chapter, we review the architecture of the GPU, the way that programs execute on the GPU, and we discuss areas where the GPU has been used as an accelerator. We then review several classic parallel abstract models, highlighting ways in which they do not capture everything that is required to accurately model computation on the GPU. Following that, we reviewed some existing GPU-specific abstract models, yet we see that both models miss data transfer between the CPU and GPU, as well as seeing how the models seek to capture different aspects of the GPU computation. Finally we highlight why data transfer between the CPU and GPU is important, and review the different efforts that have been made towards analysing these operations. In this chapter, we therefore identify scope for an improved abstract model that is specific to GPU computation, which captures more than the individual GPU abstract models, and that also captures the data transfer between GPU and CPU.



(a) Device.



(b) Multiprocessor.

Figure 2.6: Abstract architecture of the GPU, which is shared by the AGPU and the SW-GPU models.

Chapter 3

DNA Sequence Alignment and Protein Spectral Identification

3.1 Introduction

In this chapter, we introduce two specific bioinformatics problems, namely *DNA Sequence Alignment* and *Protein Spectral Identification.* Both of these problems are centred around computing a similarity score between two items, building this similarity calculation into a database or library search, in order to determine the best fitting candidate. The chapter begins by discussing the DNA Sequence alignment problem and reviewing algorithms and research efforts that look to solve this particular problem, as well as algorithms designed to take advantage of parallel architectures, such as the GPU. The chapter then proceeds to discuss of Protein Spectral Identification by reviewing existing algorithms for this process, including those which take advantage of parallel architectures, such as the GPU, before introducing the *Match Score Identification problem*, which is a version protein spectral identification problem which we study further in Chapter 6.

3.2 DNA Sequence Alignment

Deoxyribonucleic acid (DNA) is the backbone to life; it encodes all the genes of an organism (the *genome*). DNA is made up of a string of bases: *adenosine, cytosine, guanine,* and *thymine*, with each human having a unique genome. A sequencer is able to analyse DNA samples and obtain the code that represents an organism's genome. It is then of use to biologists and clinicians to analyse this data to find gene mutations and other conditions, using a process called *alignment*, which looks for similarity between sequences. The premise is that homology between two sequences can signal a connection between them, or in the case of an unknown gene sequence, can indicate which gene the sequence is most related to.

Computing alignment between two biological sequences (be that DNA sequences of bases, or protein sequences of amino acids) has been extensively studied, with the two most famous alignment algorithms being the Smith-Waterman algorithm [79] and the Needleman-Wunsch algorithm [65], both of which use a dynamic programming approach to compute a similarity score in a pairwise manner. Sequence alignment is an application of the string matching problem, whereby a short pattern sequence is compared to a longer text sequence to determine whether the pattern exists in the text, and if so, the positions where it occurs. When aligning two sequences, there are several variations of alignment that can be computed:

Global Alignment Attempts to align the entirety of the two sequences end-to-end (such as in the Needleman-Wunsch algorithm [65])

Local Alignment Looks to align local regions of the sequences that have high similarity (such as in the Smith-Waterman algorithm [79])

Semi-Global Alignment Looks to align the entirety of one of the sequences, with a prefix of the other (such as GapsMis [5]).

Because of mutations and other biological mechanisms, it is common that sequences in comparison may not be exact match but may have some mismatches. It is important to take into account mismatches otherwise some vital information may be missing. However, allowing mismatches greatly increases the complexity of the problem and algorithms detecting mismatches are significantly slower than their counterparts that detect exact matches. Existing short-read alignment tools only allow a small number of mismatches (or disallow mismatches altogether) because of this.

Differences may also appear in the form of a *gap*, which is a consecutive region that appears in the text but not in the pattern or vice versa (i.e., a consecutive sequence of insertions or deletions of letters in the text or the pattern). Gaps may occur because of mutation event that a segment of DNA sequence is copied or inserted, replication process that a segment is missing, or genetic transposition that a segment changes position on chromosomes.

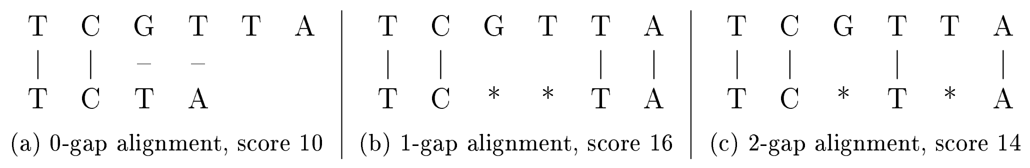

T C G T T A
T C T A
(a) 0-gap alignment, score 10
T C G T T A
T C * * T A
(b) 1-gap alignment, score 16
T C G T T A
T C * T * A
(c) 2-gap alignment, score 14

Figure 3.1: Examples of valid alignments for text $TCGTTA$ and pattern $TCTA$.

3.2.1 The Sequence Alignment Problem

A string a is a *substring* of string b if there exist two (possibly empty) strings s_1 and s_2 such that $s_1s_2 = b$. Furthermore, a is a *prefix* (*suffix* resp.) of b if s_1 (s_2 resp.) is an empty string.

Consider an alphabet consisting of the four DNA bases $\Sigma = \{A, C, T, G\}$. Let $*$ represent the gap character and $\Sigma' = \Sigma \cup \{*\}$. *An aligned pair* is a pair of letters (x, y) such that $(x, y) \in \Sigma' \times \Sigma' \setminus \{*, *\}$. In other words, an aligned pair may involve at most one gap character. An alignment of two strings T and X is a string of aligned pairs (t_1, x_1), (t_2, x_2), $\cdots$, (t_ℓ, x_ℓ) such that removing all the gap characters $*$ from $t_1t_2 \cdots t_\ell$ gives T (similarly for X). Note that there are $\ell - |T|$ gap characters in the alignment. In the alignment of T and X, we say that t_i *matches* x_i if $t_i = x_i$; t_i is *substituted* by x_i if $t_i \neq x_i$ and both are not $*$; x_i is *inserted* if $t_i = *$; t_i is *deleted* if $x_i = *$.

A sequence of ℓ aligned pairs (t_1, x_1), (t_2, x_2), $\cdots$, (t_ℓ, x_ℓ) is called a *gap sequence* if either all t_i equal $*$ or all x_i equal $*$. The sequence is called a *gap-free sequence* if none of the t_i nor x_i equals to $*$. In other words, an alignment can be viewed as $z_0g_0z_1g_1...z_{\alpha-1}g_{\alpha-1}z_\alpha$ where z_0 is a possibly empty gap-free sequence, $z_1...z_\alpha$ are non-empty gap-free sequences, and $g_0...g_{\alpha-1}$ are gap sequences. In this case, the alignment has α gaps.

Figure 3.1 demonstrates the following example: suppose we have two sequences `TCGTTA` and `TCTA`. If we do not allow a gap, we can align `TCGT` with `TCTA` with two matches. If we allow a gap of any length, we can align `TCGTTA` with `TC**TA` with four matches, where `*` represents a gap character. If we allow two gaps, we can align `TCGTTA` with `TC*T*A`, also with four matches.

For each possible alignment of two sequences, they must be quantified in a way that communicates their similarity, where the most suitable alignment will have the highest (lowest) score, in the case of a maximising (minimising) score function. Given two strings T and X, we can measure the quality of an alignment of T and X by a score function $\delta(\cdot)$. The score of an alignment is calculating by adding the scores of all gap sequences and gap-free sequences in the alignment. Scoring functions are discussed in greater detail in Section 3.2.2.

When the sequence alignment problem is applied to a database or library search, the problem is to report the best scoring database reference sequence (text) for a given sample sequence (pattern). With the advances in sequencing technologies, the amount of data that requires alignment has increased drastically. For example, the GenBank public database contains over 162,000,000 sequences and the Illumina HiSeqX Ten sequencer can produce three billion reads (sequences) of length 250 bp (base pairs) in less than three days. The *re-sequencing* problem is to assemble short reads produced by the sequencer (an equipment that takes a physical biological sample and outputs the sequence of nucleobases as a character string) into a genome sequence by referring to a reference genome, requiring “mapping” or “aligning” of the short reads back to a database of reference sequences. The task is therefore challenging due to the increasing amount of data and the large genome sizes.

3.2.2 Scoring Functions

The way in which an alignment is scored, and how the algorithm deals with the given score, has a huge effect on the accuracy of the alignments that are produced. There are several ways in which an alignment can be scored; some are taken from stringology (such as the edit distance), and others have been created to maximise biological accuracy. The alignment scoring scheme can be separated from the alignment algorithm, and an alignment algorithm can be adapted to use each particular scoring scheme.

As set out in Section 3.2.1 scoring function is represented as $\delta(x, y)$, where $x, y \in \Sigma' = \Sigma \cup \{*\}$. This function will map to a scoring matrix, which is $|\Sigma'|^2$ and maps each possible combination of characters to a score value. Additionally, factors within the alignment such as gaps will change the scoring function. The scoring function chosen will dictate the values within $\delta(\dot{})$, with some common scoring functions described below.

The *Levenshtein distance* is defined as the minimum number of operations required to transform string T into string X. These operations can be to *substitute* an existing character, to *insert* a new character, or to delete a character. The Levenshtien distance is a *minimising* function, that is strings with a *higher* degree of similarity will have a *lower* Levenshtein distance.

The *Hamming distance* between two strings of equal length is defined as the number of mismatching characters. For example, the strings *aabaa* and *aacaa* have a hamming distance of 1; the strings *abbaa* and *aaabb* have a hamming distance of 4. As with the Levenshtein distance, the Hamming distance is a minimising function, where strings with a *high* degree of similarity have a *lower* Hamming Distance.

The use of specialist biological scoring matrices, such as the BLOSUM series, and the PAM series are substitution matrices that contain scores to accurately reflect alignments in biology. These matrices reward alignment that is good and biologically accurate alignment, penalising either bad alignments or those which are biologically inaccurate. As a result of this, the values in the scoring matrices are much more varied (BLOSUM62 has a range of $-4 - 11$) than the substitution matrices used when calculating either Hamming or Levenshtein distances, which would typically comprise of 0s and 1s. It is also the case, that when using such scoring matrices, the scoring function is a *maximising* function, and that two sequences with good alignment have a higher score.

Scoring for Gaps Up until now, we have seen that substitution matrices can score an alignment based upon the matched character pairs. We now discuss how the alignment scoring schemes can account for gaps.

By adding scores for a *gap character* into the substitution matrix, substitution matrices can account for gaps using a *constant gap penalty* i.e. a gap character would be assigned a particular cost. This means that two gaps of length 3 would score the same as one gap of length 6, yet It has been claimed that it can be desirable to penalise the occurrence of gap as a whole instead of individual alternations [24].

The *affine gap cost* is one of the most popular gap scoring functions, which heavily penalises the opening of a gap, whilst penalising less severely for extending a gap. This favours fewer, longer gaps in the final alignments that are reported. Affine gap penalty scores are shown to increase the sensitivity of detecting biological relationships and constructing biologically accurate alignments[2].

A study [87] into the accuracy of a generalised affine gap penalty (allowing a gap sequences to occur in both sequences) as opposed to a traditional affine gap penalty shows that generalised affine gap cost will produce shorter but more accurate alignments, and that traditional affine gap penalty costs will provide slightly less accurate, but longer alignments, when aligning protein sequences.

There are several other gap scoring functions in the literature, such as *logarithmic* scoring and *log-affine* scoring.

Some have argued that non-affine gap scoring increases biological accuracy of the resulting alignments: The authors in [31] argue for logarithmic gap penalty when aligning protein sequences, and the authors in [62] produced a gapped local sequence alignment tool that uses a non-affine gap penalty, which increases the likelihood of finding a good alignment containing a large gap. PLANAR [7] is a sequence alignment tool for RNA that also takes into account the RNA secondary structure (therefore a more complex problem than regular sequence alignment) using trees and linked lists, with a non-affine gap cost. The authors in [58] propose a variable gap penalty function that is sensitive to the structure of proteins, which is shown to increase accuracy of generated alignments when aligning protein sequences.

However, it has been shown in [15] that logarithmic gap costs decrease the accuracy of computed alignments, despite the distribution of gap sizes being shown to follow a power law, which lead some scientists to use a logarithmic gap cost when constructing alignments. The authors show that logarithmic gap costs actually *decrease* the accuracy of the constructed alignments. They studied three types of gap costs and found that Log-Affine gap cost was the most accurate, followed by affine gap cost, and lastly by logarithmic gap cost. They show that affine gap costs (the most popular type used) produce a good approximation of biological alignments, and could be used with confidence. The authors later produced a sequence alignment tool called “Ngila”[16] which uses log-affine gap costs to score alignments.

In addition to more specialised scoring for gaps, there has been some limited effort to incorporate the quality of the experimental data and the quality of the equipment used into the scoring of the alignment in general.

RMAP[78] is a tool that uses quality scores and longer reads to improve the mapping of read to reference alignments, and [27] demonstrates how DNA sequence quality scoring can be incorporated into sequence alignment tools. They shows that this can improve the quality of the alignment results obtained, increasing correctly mapped reads of a real data set from 49% to 66%.

3.2.3 Existing CPU-based Sequence Alignment Tools

There is a wide range of existing CPU-based alignment tools available; some are designed for serial execution, some are designed to take advantage of multicore CPUs, and others are designed to run at scale on a cluster or a supercomputer. Each addresses different challenges of the sequence alignment problem. A large amount of tools use a dynamic programming approach, yet there are other methods used which employ hashing, Burrows-Wheeler transformation, and graphs.

The Basic Local Alignment Search Tool [3] is a widely used local alignment tool that uses a dynamic programming approach to produce local alignment library search. It heuristically reduces the search space from all entries in the library, down to those that are most likely to provide a region with high similarity to the sample sequence, but scanning for segments within the library that exceed a particular scoring threshold.

Crochemore et al.[20] utilise text compression (using LZ78 technique) is used to successfully speed up the Smith Waterman alignment between two sequences with an additive gap penalty. The repeating structure and the potential of unrestricted indels/substitutions of the Smith Waterman algorithm allow for this optimisation to work. Farrar et al.[23] developed an SIMD-based implementation of the Smith-Waterman algorithm, with experiments achieving up to $8\times$ speed up over existing SIMD implementations.

FOGSAA[17] is a sequence alignment tool that uses a branch and bound method to accelerate the sequence alignment task. It is shown to give an improvement of $70-90\%$ for highly similar sequences, $54-70\%$ for those of mid-range similarity, and terminates with an approximate score for sequence pairs of minimal similarity. Protein sequences are shown to obtain an improvement of $23-53\%$ against existing heuristic global alignment methods. This tools performance is dependant on the content of the sequences, and in some cases will only provide an approximation of the similarity score. Therefore it does not provide a reliable and repeatable improvement level against existing tools.

SOAP [47] looks to efficiently align large sets of sequence data, which other alignment tools had struggled to do. It uses a seed-and-hash technique to accelerate alignment of upto two reads to a reference sequence, allowing either a limited number of mismatches, or a single gap of upto length 3 in the alignment. It was shown that in addition to the speed improvement on existing tools, the option to allow restricted gaps increased the accuracy of the alignments, whereas previous tools would only calculate with or without gaps (not both).

The BOWTIE[44] short read sequence alignment tool which uses Burrows-Wheeler transformation (which prepares strings for compression by reversibly-rearranging them in order to be easier to compress, whilst having a lower memory footprint when compared to hashing fragments of strings). It does not support insertions or deletions (gaps), but provides speed-up against existing tools like SOAP and Maq, and can be run on a regular desktop computer because of lower memory footprint and the ability to be parallelised over a number of threads on the CPU. SOAP2 [48] further improves upon SOAP by using Burrows-Wheeler Transformation. The Burrows-Wheeler Alignment (BWA) short read alignment tool [46] uses Burrows-Wheeler Transformation to align short reads to long references, allowing mismatches and gaps. Experiments carried out as part of the study show $10\times-20\times$ improvement on MAQ, retaining similar accuracy of alignments. SOAPdenovo[49] is an extension of SOAP2 that successfully assembles the human genome in a supercomputer. REAL[28] is a short-read alignment tool, allowing for at most k mismatches, based on the signature of various sequence fragments which outperforms SOAP2. SHRiMP [73] uses a mix of a hashing technique along with a vectorised implementation of Smith-Waterman algorithm and an alignment technique that takes advantage of output from particular equipment. They demonstrated the accuracy of their approach, and that the majority of speed-up against existing tools occurred during the vectorised implementation of the Smith-Waterman algorithm. GenomeMapper [74] uses a hash based graph approach to simultaneously align short reads against multiple genomes, by indexing the multiple genomes before aligning the reads against the index. Stampy [57] is a tool which aligns short reads with a hashing technique and incorporates statistical analysis with high accuracy and speed. [76] Parallelised Short-read sequence denovo assembler using distributed debrijn graph method on a cluster.

Because of the importance of gaps, the alignment problem has been considered in the presence of gaps [24, 1, 5]. In addition to allowing mismatches in the form of edit distance or score, the problem also allows for a bounded number of gaps (of any length). Usually the number of gaps allowed is a small constant independent of the length of the text or pattern. Dynamic programming algorithms have been proposed to find the alignment with the best alignment “score” with a bounded number of gaps. In [24, 1], a single gap is allowed and the algorithm GapMis is proposed; while in [5], multiple gaps are allowed and the algorithm GapsMis is proposed. The algorithms GapMis and GapsMis have been implemented and are shown to perform well against other approaches like EMBOSS water [72] and EMBOSS needle [72].

3.2.4 Existing GPU-based Sequence Alignment Tools

Due to the ever-increasing data size, faster sequence alignment tools are needed. This asserts not just speed requirement on the processors but also leads to high power/energy requirements; furthermore, this potentially causes the processors to reach too high a temperature, potentially damaging them. To solve this problem, it is nowadays common to exploit multi-processors such as the GPU. There are many alignment tools available, which use the GPU in order to achieve increase in speed.

Various sequence alignment problems have been tackled using GPU-based algorithms, including BLAST (the Basic Local Alignment Search Tool) [84, 89], the Smith-Waterman global alignment algorithm [59, 51, 56, 81], Needleman-Wunsch local alignment [22, 71] ([75] studies GPU implementation of Smith-Waterman and Needleman-

Wunsch with focus towards a hybrid model).

SOAP3 [52] is currently among the best short-read alignment tools available, adapting the Burrows-Wheeler Transformation technique of SOAP2. CuHMMer [86] is CPU-GPU hybrid tool for calculating sequence similarity using hidden Markov models. It load-balances the tasks on both CPU and GPU, so that the CPU does not sit idle whilst the GPU is computing. CuHMMer claims upto 45× speed-up over CPU control experiments, and outperforms existing GPU-only implementations. Zhang et al.[88] develop a GPU-based pairwise statistical significance estimation tool, which determines whether a particular local alignment (produced by an alignment tool) can occur by chance alone. claiming 180× speed-up against the CPU.

For Semi-global sequence alignment with a single gap, a tool called libgapmis [1] has been developed, which uses a task-parallel GPU-based method, for which an 11× speed-up over CPU control experiments has been reported. With multiple gaps, there is a data parallel implementation [68] of GapsMis, for which a 5× speed-up over the CPU has been reported, yet this implementation does not use the GPU to compute the backtracked optimal alignment and uses sequential techniques for finding the optimal gap insertion point.

3.3 Protein Spectral Identification

Proteins provide function, structure, and repair to biological cells. Proteins are created during *translation*, where Ribonucleic acid (RNA) is traversed by a ribosome, stringing together amino acids that correspond to the sequence of the RNA. Strings of amino acids form *peptides*, a string of which forms each protein. After translation occurs, various chemical modifiers can attach themselves to points along the protein, which could provide different functionality to the protein. This is known as Post Translational Modification (PTM).

It is of interest to biologists and clinicians to be able to identify a protein from a given sample, and to also identify any PTM present. Understandably, PTM can make this problem more challenging and complex, meaning that a challenge is to identify a protein that contains PTM. There are over 300 different types of PTM which are known to occur, so it is also of interest to be able to characterise these modifications and to discover new ones.

Mass Spectrometry is a technique in examining the chemical make-up of a sample, whereby the sample is fractured and each part measured, outputting a *Spectrum* of peaks which corresponding to the molecular masses of parts found within the sample.

Tandem Mass Spectroscopy (MS/MS) attaches two mass spectrometers together and gives finer detail to the mass spectrometry process and is often used in the sampling of proteins and peptides.

There are several large public databases containing spectra corresponding to known proteins and peptides, against which it is possible to compare spectra obtained from protein or peptide samples in order to identify them. However, a simple value-by-value comparison of a sample against a library of known proteoform spectra is normally not sufficient, as the sample has undergone *Post Translational Modification (PTM)*, which changes molecular weight of some peptides within the protein, therefore affecting the whole sample spectrum.

Spectral Alignment (often solved using dynamic programming) is used to compare the query spectrum and a set of target spectra (the library), allowing unexpected PTM. A similarity score for each spectral pair is computed, under the premise that the higher similarity two spectra posses, the more likely they are to be related. These similarity scores are generally centered around counting the number of similar mass peaks in the two spectra, allowing for modifications. This similarity score is then used in database search, in order to identify the sample protein.

Matching the spectrum of a sample protein against a database of known protein spectra has become a point of bottleneck in many modern mass spectrometry experiments [6]. To alleviate this bottleneck, there has been lots of study into reducing the search space of the database, and into speeding up the computation, using serial computer programs.

3.3.1 Sequential Approaches to Spectral Alignment and Library Search

There are many CPU-based tools that compute spectral alignment, using different methods and scoring schemes.

Mascot[69] is a well regarded protein spectral identification tool that uses a probability based scoring function. Pevzner et al [70] propose a dynamic programming method for PTM-tolerant spectral alignment, which computes a spectral similarity score, shown to be effective for upto two PTMs. The approach looks to make switches between diagonals through a spectral graph (that is, insert a PTM between aligned sections of the spectral pair). This approach has found itself to be quite popular for other tools to be based upon.

ProsightPTM [45] is a well regarded web-based top-down protein spectral alignment tool which uses sequence tag search and probability based scoring.

Frank et al. [26] present a study that shows how spectral alignment is able to identify PTMs in samples of intact proteins that are generated by higher resolution instruments, using a top-down approach called MS-Top-Down, as opposed to the bottom-up approach that was more commonly used. The advantage of protein identification using top-down approach is that it is possible to characterise the proteins exact form, as opposed to merely identify the protein as in the bottom-up approaches.

A tool called PSAwEL [18] presents a spectral alignment algorithm that allows unexpected PTM (that is, the user does not need to specify them beforehand), which uses a new scoring function to improve on their previous work.

The alignment of spectra with a PTM has been accelerated using linked lists [53] in order to characterise the values within a set of spectra. Here, the authors show how by breaking away from the commonly used dynamic programming methods, and by indexing in linked lists which particular spectra have a peak at a given value, can the alignment be accelerated, requiring only traversal of relevant linked lists. In addition to performing the spectral alignment task with a single PTM, this approach is also used as a filtering technique for other protein spectral alignment tools, such as [54]. We study this particular approach in further detail in Chapter 6.

MS-Align+ [54] is a spectral alignment algorithm that filters the search space by characterising each diagonal crossing through the spectral grid (possible using the method in [53]), using a similar alignment method to [26]. If a candidate protein has low scoring diagonals, then it is not likely to match, and is discarded. For the well scoring candidate proteins, only the top scoring diagonal crossings are then used for the alignment, which is shown to both accelerate the alignment process and retain a high level of accuracy.

MS-Align-E [55] overcomes the limitation that is encountered with unexpected PTMs, which is that a good scoring diagonal to serve as evidence for the PTM may not exist in the spectral graph, which is common when histones occur ultra-modified proteins. The tool overcomes this limitation by considering diagonal fragments and using a list of expected PTMs to construct directed edges in the spectral graph, solving the problem by scoring paths from the source to the sink in the graph.

Raw output from the mass spectrometry equipment is processed into XML and then transformed into monoisotopic mass lists (an ascending list of numbers which correspond to the mass peak values of the obtained spectrum), using spectral deconvolution tools such as MS-Deconv+ [42], before performing spectral alignment on the mass lists as a scoring for database search. Thus, the MS/MS pipeline is assembled.

TopPIC [43] is a protein spectral identification tool that improves on MS-Align+ [54], which heuristically filters out unsuitable candidate proteins, before performing spectral alignment, and finally performing statistical analysis on the reported alignments. They reduce the memory requirement of the database by using an indexing method.

3.3.2 Spectral Alignment and Library Search on GPU

Whilst the GPU has been used extensively in sequence alignment, to our knowledge there has been relatively little work on using the GPU to accelerate spectral alignment for identification of proteins or peptides.

To our knowledge, the first attempt to implement a protein spectral library search on the GPU was that by Baumgardner et.al. [6]. They develop a tool called FastPaSS (*Fast Parallelized Spectral Searching*), which implements in CUDA the SpectraST spectral searching algorithm. There are three scoring metrics that are used by SpectraST (and therefore also by FastPaSS): the dot product, the Δ-dot, and the F-value. The Δ-dot and F-value are performed on the CPU, and the dot product is performed on the GPU. The dot product uses matrix multiplication, which lends itself well to the GPU, as matrix multiplication is easily parallelisable, being a common student exercise when learning General Purpose Programming on GPU (GPGPU). FastPaSS successfully uses the GPU to accelerate a computationally intensive part of the spectral library search using the GPU, with experiments demonstrating an improvement of up to 26×, when compared to CPU control experiments.

Milloy et. al. develop a GPU-based peptide matching tool called Tempest [61]. Tempest computes the digestion of candidate proteins (into candidate peptides) on the CPU, and the computes the similarity score between sample peptide spectrum and selected candidate spectra on the GPU. This is in contrast to [6], which only computes *some* of the similarity score on the GPU, and does not perform *in-silico* digestion of proteins. The similarity score used by Tempest is a dot-product type scheme, based upon the SEQUEST XCorr scoring metric. The GPU uses a single thread to compute the similarity score between a single candidate peptide and the sample peptide serially, computing in batches until all scoring is complete, before reporting the highest scored matches. By accelerating the scoring using the GPU, experiments show that Tempest is able to achieve speed-up of up to 15× against single-threaded control CPU experiments.

Li et al. [50] design an efficient GPU-based spectral dot product scoring module and demonstrate experiments which achieve up to 60× speed-up against CPU control experiments on a single GPU, and also demonstrate favourable speed-up when deployed at scale on a GPU cluster of four nodes.

$$T' = 010101010000$$
$$T = 2, 4, 6, 8, 12$$

Figure 3.2: Example of Spectrum as mass-list and as vector.

3.3.3 Match Score Identification Problem

We now describe the Match Score Identification problem, introduced in [53], which we study in Chapter 6.

A spectrum $T = t_1 < t_2 < ... < t_n$ is a list of prefix residue masses. That is, each element $t_i \in T$ represents the cumulitve mass of the first i fragment ions within T, with t_n representing the mass of the entire sample, known as the *precursor mass*. T can be represented as a *mass-list* of integers, being $t_1, t_2, ..., t_n$, or as a *0-1 vector* T'. $T' = t'_1, t'_2, ..., t'_{t_n}$ where $\forall t'_i \in T' : t'_i = 1$ if $i \in T$ and 0 if otherwise meaning if there is a peak in the spectrum at mass ℓ, then there is a 1 in the vector at position ℓ.

Consider two spectra $T = t_1, t_2, ..., t_n$ and $X = x_1, x_2, ..., x_m$. A mass pair (t_i, x_j) is *matched* if and only if:

- $t_i = x_j$
- $t_i \neq t_n$
- $x_j \neq x_m$

A *post translational modification* (PTM) can occur to a protein, changing the spectrum from its previous unmodified form, changing the precursor mass of the spectrum, the values of the peaks within the spectrum, and possibly even adding new peaks to the spectrum. In the 0-1 vector representation of an unmodified spectrum, the modification would manifest itself as insertion of a string of 0s and 1s. A spectrum T can be *shifted* by an integer value $\delta > 0$, denoted as $T(\delta)$. The shifted spectrum $T(\delta)$ is generated as $T(\delta) = t_1 + \delta, t_2 + \delta, ..., t_n + \delta$, inserting δ 0s at the start and shifting the entire spectrum δ places to the right in the 0-1 vector operations.

The Match Score Identification Problem

We investigate using a scoring scheme called the *match counting score*, which is then used in the Match Score Identification Problem.

Match Counting Score The match counting score is the number of matched mass peaks between two spectra T and X, where T is the spectrum of a modified protein

$$\begin{array}{ccccccccccc}
0 & 0 & 1 & 0 & 1 & 0 & [0 & 0 & 0] & 1 & 0 \\
 & & | & & | & & & & & & \\
0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & & &
\end{array}$$

(a) $CT, X) = 2$

$$\begin{array}{ccccccccccc}
0 & 0 & 1 & 0 & 1 & 0 & [0 & 0 & 0] & 1 & 0 \\
 & & & & & & & & & | & \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0
\end{array}$$

(b) $C(T, X(3)) = 1$

Figure 3.3: Calculating match score for $T = \{2, 4, 9, 10\}, X = \{2, 4, 6, 7\}$, where $\delta = 10 - 7 = 3$. The PTM in T is surrounded by square brackets.

containing a PTM and X is the spectrum of the unmodified form of the protein. The PTM has inserted a mass of value $\delta = t_n - x_m$ into T. The mass pairs containing t_n or x_m are never matched, because by inserting a shift of δ into T, the mass pair (t_n, x_m) will always be matched, therefore not providing useful information about the similarity of the two spectra T and X.

$C(T, X)$ is the number of matched pairs between the spectra T and X, excluding the precursor masses. $C(T, X)$ therefore gives the number of matched mass pairs to the left of the PTM, yet not to the right. $C(T, X(\delta))$ gives the number of matched mass pairs to the right of the PTM.

The match counting score for two spectra T, X is calculated as $C(T, X)+C(T, X(\delta))$. An example is given in Figure 3.3.

Problem 1. *The Match Score Identification Problem (MSI) is to calculate the match score between each spectrum $T_i \in \mathcal{T}$ and a set of spectra $\mathcal{X}$ and to return the spectrum in $\mathcal{X}$ with the maximum match score, where $\mathcal{T} = T_0, T_1, ..., T_{q-1}$, $T_i = t_0, ..., t_{n-1}$ and $\mathcal{X} = X_0, ..., X_{r-1}$, where $X_j = x_0, ..., x_{m-1}$. That is, we wish to find $X_\alpha \in \mathcal{X} : \alpha = \max_{j=1}^{r}(C(T_i, X_j) + C(T_i, X_j(\delta))$ for each $T_i \in \mathcal{T}$.*

3.4 Conclusion

In this chapter, we reviewed DNA sequence alignment algorithms and protein spectral alignment algorithms. We began by reviewing sequential solutions to the problems, before reviewing solutions to the problems which take advantage of the GPU. Finally, we identified scope for two particular problems for us to study further the acceleration using GPU: the sequence alignment problem (we study this problem in Chapter 5), and the match score identification problem (we later study this problem in Chapter 6).

Chapter 4

The Abstract Transferring GPU Model

Introduction

GPUs are commonly used as coprocessors to accelerate a compute-intensive task, taking advantage of their massively parallel architecture. There is study into different abstract parallel models, which allow researchers to design and analyse parallel algorithms. However, most work on analysing GPU algorithms has been software based tools for profiling a GPU algorithm. Recently, some abstract GPU models have been proposed, yet they do not capture all elements of a GPU. In particular, they miss the data transfer between CPU and GPU, which in practice can cause a bottleneck and reduce performance dramatically. In this chapter, we propose a comprehensive model called ATGPU which to our knowledge is the first abstract GPU model to capture data transfer between CPU and GPU. We show via experiments, that existing abstract GPU models cannot sufficiently capture all of the actual running of a GPU algorithm time in all cases, as they do not capture data transfer. We show that by capturing data transfer with our model, we are able to obtain more accurate predictions of the GPU algorithm actual running time. It is expected that our model helps improve design and analysis of heterogeneous systems consisting of CPU and GPU, and will allow researchers to make better informed implementation decisions, as they will be aware how data transfer will affect their programs.

4.1 Introduction

We observe in Chapter 2 that the SW-GPU and the AGPU model different aspects of the GPU execution; SW-GPU models the trend of overall running time, whereas AGPU has focused on design and analysis of individual elements of the kernel. Both are equally important, so there is scope for a more comprehensive model combining all elements. Using GPU as a coprocessor requires data transfer between CPU and GPU and this can lower performance if not properly considered. To our knowledge, this is not captured in any abstract GPU model, though it has been well studied in software based tools. Our contribution consists of an abstract model, called Abstract Transferring GPU (ATGPU), which is an extension of previous models. We introduce new components to capture data transfer between CPU and GPU.

We extend the SWGPU and AGPU architecture, introducing a size constraint on global memory, making the model more realistic. We extend the pseudocode of AGPU to capture data transfer, and we extend the SWGPU cost function to model data transfer and to simulate the cost on a particular GPU. To our knowledge, ATGPU is the first abstract model with this comprehensive array of analysis and design capabilities, and the first abstract GPU model to capture data transfer. A comparison of models is provided in Table 4.1.

We demonstrate the use of ATGPU and evaluate several example computational problems using the model. We show via experiments that existing models are not able to sufficiently model the actual running time in all cases, as they do not capture data transfer. We show that by capturing data transfer using our model, we are able to obtain more accurate predictions of the actual running time.

The remainder of the chapter is organised as follows: Section 4.2 discusses our

Table 4.1: Comparison of GPU abstract models

Item	**AGPU**[39]	**SW-GPU**[77]	**ATGPU**
Pseudocode	✓		✓
Time Complexity	✓	✓	✓
I/O Complexity	✓	✓	✓
Space Complexity	✓		✓
Shared Memory Limit	✓		✓
Synchronisation		✓	✓
Cost Function		✓	✓
Global Memory Limit			✓
Host/ Device Data Transfer			✓

model, and Section 4.3 describes how algorithms are analysed in our model. Section 4.4 analyses and evaluates computational problems using our model, and Section 4.5 gives concluding remarks on the chapter.

4.2 The Model

We now describe the architecture, execution, and usage of the ATGPU model.

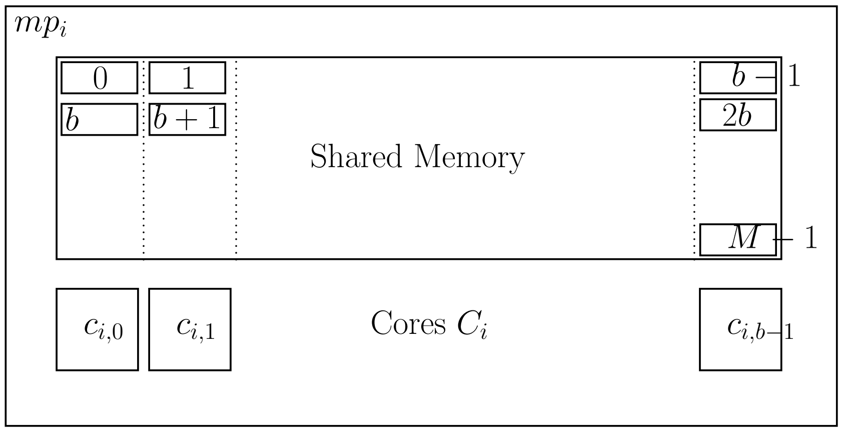
Architecture. The architecture of ATGPU (depicted in Figure 4.1) is similar to SWGPU and AGPU, with an additional constraint on global memory size. The model captures a *host* (CPU) and a *device* (GPU). Let $ATGPU(b, k, M, G)$ be an instance of the model with b cores on k Multiprocessors, giving bk cores in total,shared memory of M words per MP, and global memory of G words.

Let $MP = \{mp_1, mp_2, ..., mp_k\}$ be the set of MP. Let $C_i = \{c_{i,1}, ..., c_{i,b}\}$ be the set of cores of $mp_i \in MP$. All $c_{i,j} \in C_i$ execute the same set of instructions in lockstep. The shared memory of each $mp_i \in MP$ is split into b memory banks, such that b successive words reside in distinct banks. Only C_i can access the shared memory of mp_i. The global memory is divided into memory blocks of b words. The host and all $mp_i \in MP$ can access global memory.

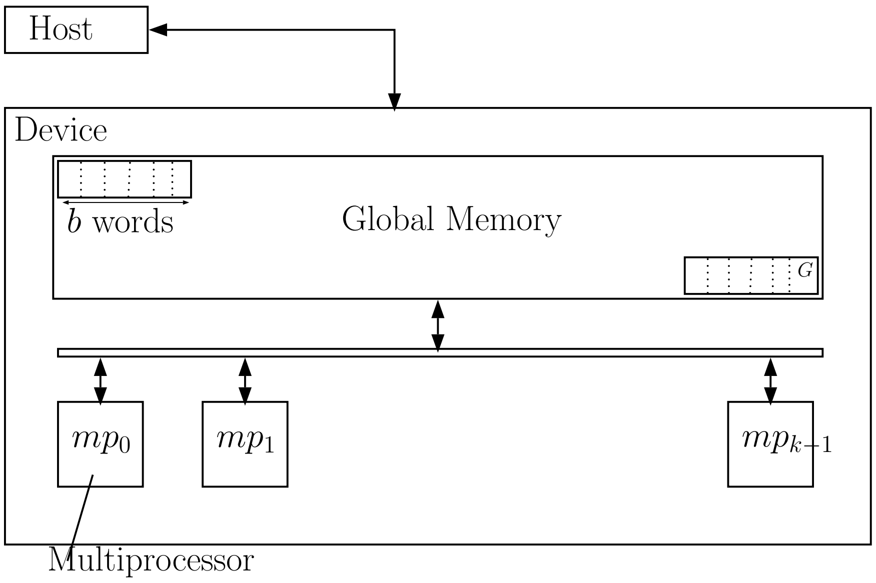
Execution of Algorithms on the Model. ATGPU executes algorithms in *rounds*, similar to SWGPU. A round begins by the host transferring data to the device global memory. The kernel is then run on all (or a subset of) $mp_i \in MP$, and on all cores $c_{i,j} \in C_i$. Instructions are executed on C_i in lockstep. If execution paths diverge, all paths are executed. Data must be moved from global memory to shared memory in order for C_i to access it. Upon a memory access instruction, C_i waits until all cores have their memory request resolved.

In a global memory access instruction, if C_i requests words within the same memory block, instructions *coalesce* and complete as a single transaction. If requested words are in l separate memory blocks, l separate transactions occur. This is depicted in Figure 4.2.

In a shared memory access instruction, if all $c_{i,j} \in C_i$ access words in distinct memory banks, the request completes in constant time. If there is access to words in the same memory bank, a *bank conflict* occurs and requests are serialised. We assume that bank conflicts do not occur, as these are difficult to analyse. It is worth noting, that the existing models (AGPU and SW-GPU) also assume that execution of the GPU program is bank conflict free, and do not capture bank conflicts for the same reason.



(a) Multiprocessor of our model. Note the b banks (shown as columns in the shared memory) and the b cores.



(b) Device view of our model, with the $k = \frac{p}{b}$ multiprocessors, and the global memory of size G, in blocks of b words

Figure 4.1: Abstract architecture of the ATGPU Model

The round ends with output data being transferred from global memory to the host. Synchronisation operations occur, and the subsequent round commences.

Notation for Pseudocode. We extend the pseudocode from the AGPU model, allowing for explicit data transfer. Each GPU kernel is placed inside a *wrapper loop*, to execute the instructions on MP. If instructions are to be run on only a subset of MP, then this is specified within the wrapper loop.

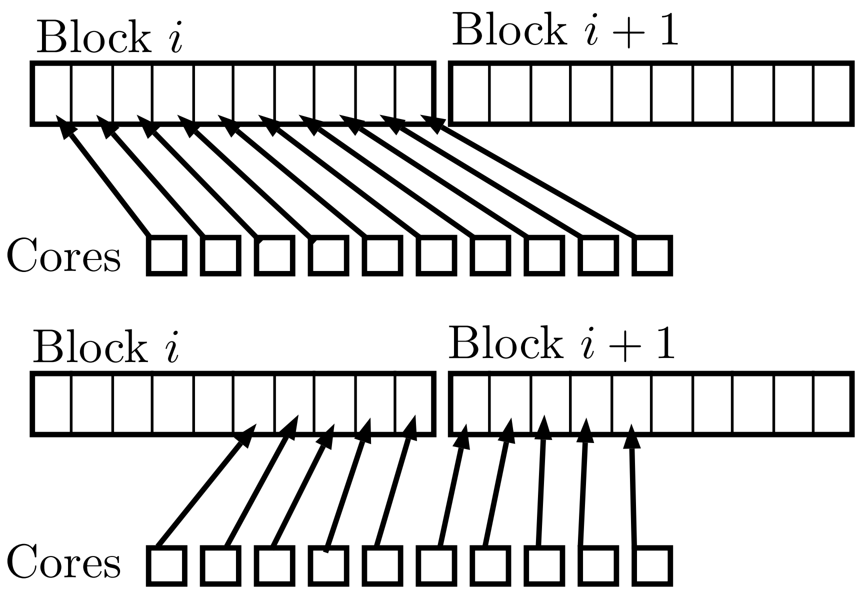


Figure 4.2: Examples of coalesced global memory access, where all cores request memory locations from the same memory block and thus completes in a single transaction (top), and non-coalesced global memory access, where cores request memory locations from two different blocks and thus completes in two transactions (bottom).

Algorithm 1 ATGPU Wrapper Loop

```
for all $mp_\rho \in MP[mp_0, ..., mp_{b-1}]$ in parallel do
    for all $c_{\rho,\epsilon} \in C_\rho$ in parallel do
        Instructions ....
```

In our model, any primitive data type, with vectors and arrays thereof, are allowed as variables. Our model defines three types of variable scope. *Host* variables reside in host memory, only accessible to the host, and their names begin with capital letter. *Global* variables reside in global memory, accessible by the host and all MPs, and their names begin with lower case letter. *Shared* variables reside in shared memory, accessible only by C_i for shared variable on mp_i, and their names begin with an underscore. Memory access syntax is *<destination><operator><source>*. Data transfer between host and device uses the $\Lleftarrow$ operator, global memory access uses the $\Leftarrow$ operator, and shared memory access uses the $\leftarrow$ operator. The *if-statement* can also be used as an $if-then-else$ statement, but it is important to note that all branches of program flow will be executed where any of the cores enter the branch, therefore it is very important to reduce diverging execution paths.

4.3 Analysing Algorithms on ATGPU

Our model defines the metrics below for an algorithm running on ATGPU. Asymptotic complexity can be measured both on a per-round basis and across the entire algorithm.

Number of Rounds R. The number of rounds R in the program defines how many rounds are required. As data transfer and synchronisation take a non trivial amount of time, we look to minimise this value. This is from SW-GPU.

Time t_i. The maximum number of operations across all MPs executed in round i. This is from both AGPU and SW-GPU. In the case of an $if - then - else$ statement, **all branches** of the statement must be counted, as the cores will execute all branches that any one of the cores enter.

I/O q_i. The total number of global memory blocks accessed in the round, by all MP. This is from both AGPU and SW-GPU. In the case of an $if-then-else$ statement, **all branches** of the statement must be counted, as the cores will execute all branches that any one of the cores enter.

Global Memory Space Used. The number of words stored in global memory for each round i. If there is difference between rounds, then the largest value is taken. If this is greater than G, the algorithm cannot be run on our model.

Shared Memory Space Used. The maximum number of words stored in shared memory across all MPs in round i. If there is difference between rounds, then the largest value is taken. If this is greater than M, the algorithm cannot be run on our model.

Data Transfer. We introduce analysis of data transfer to the model. For round i, let I_i (O_i resp.) be the number of words transferred from the host to device (device to host resp.) at the start (end resp.) of the round, referred to as *inward* (*outward* resp.) transfer. The total amount of words transferred between the host and device for all rounds can be measured as: $\sum_{i=1}^{R}(I_i + O_i)$.

Cost Functions. The cost function is adapted from the SW-GPU, with modifications of data transfer, operation rate, and a GPU-cost.

Operation Rate γ. The cost for a multiprocessor to execute instructions is represented by the variable γ. This variable corresponds to the number of instructions executed by the device in a time-cost unit, hence why the cost function *divides* by this, as opposed to *multiplying*. We see that this corresponds to the clock rate of the GPU. The operation rate γ can be set to a value corresponding to a particular GPU for calculating the cost.

Global Memory Access Cost λ. The cost to access a memory block in global memory is non-trivial; accessing shared memory, when no bank conflicts occur can take 4 cycles, whereas global memory can take in the region of $400 - 800$ cycles. We denote by λ this cost, being the number of cycles to access a global memory block.

Fixed Synchronisation Cost σ. The fixed cost synchronisation tasks that need to take place, such as resetting the device, de-allocating and reallocating of data structures, clearing queues, etc. take a non-trivial amount of time. This is represented by σ.

Host Device Data Transfer. Boyer et al. [9] gave a function to determine the time of data transfer between CPU and GPU. We use this to assign cost to data transfer stages. Let α represent the initial overhead of a data transfer transaction, β represent the cost of sending a word, and $\hat{I}_i$ ($\hat{O}_i$ resp.) represent the number of data transfer transaction of inward (outward resp.) transfer in round i. The function Ti_i gives the cost of inward data transfer for round i: $Ti_i = \hat{I}_i\alpha + I_i\beta$. Likewise, the function To_i gives the cost outward data transfer for round i: $To_i = \hat{O}_i\alpha + To_i\beta$.

Cost Function. We say that the cost of an algorithm is upper bounded by Expression (4.1):

$$\sum_{i=1}^{R}\left(Ti_i + \frac{t_i + \lambda q_i}{\gamma} + To_i + \sigma\right) \quad . \tag{4.1}$$

GPU-Cost Function. Expression (4.1) gives the cost as ran on a "perfect GPU" — a GPU with sufficient resources to concurrently run every thread block in the algorithm. This is an impossible machine, with an unlimited amount of memory and multiprocessors. Like how the AGPU allows a k multiprocessor machine to simulate $w > k$ MPs, we can alter the ATGPU cost function so that it simulates a GPU with $k' < k$ MPs. Each MP on a GPU can accommodate $\ell = \min(\lfloor \frac{M}{m} \rfloor, H)$ blocks concurrently, where m represents the shared memory usage by a GPU program, and H represents a hardware imposed limit. The GPU cost function is given as shown in Expression (4.2), which captures the concept of occupancy:

$$\sum_{i=1}^{R}\left(Ti_i + \frac{\lceil \frac{k}{k'\ell} \rceil t_i + \lambda q_i}{\gamma} + To_i + \sigma\right) \quad . \tag{4.2}$$

4.4 Evaluation of Our Model

We evaluate our model using several computational problems: vector addition, reduction, prefix scan, and matrix multiplication. These algorithms have been well studied in the past, and we use our model to focus on the effect of data transfer on their actual running times. We measure the effect on overall running time when the data transfer is included, compared to when the data transfer is not included. We scrutinise the effectiveness of our model in capturing data transfer and providing a more accurate prediction of overall running time than the SW-GPU, which does not capture data transfer.

To do this, we examine the trends of the SW-GPU cost function, the ATGPU cost function, the observed total running time, and the observed kernel running time as the input size increases. We use the GPU cost function of our model as the ATGPU cost, and the GPU cost function of our model minus the data transfer as the SW-GPU cost. Our model can be shown as useful in cases where the rate of growth for the ATGPU cost function is closer to the actual running time than the SW-GPU cost function.

4.4.1 Vector Addition

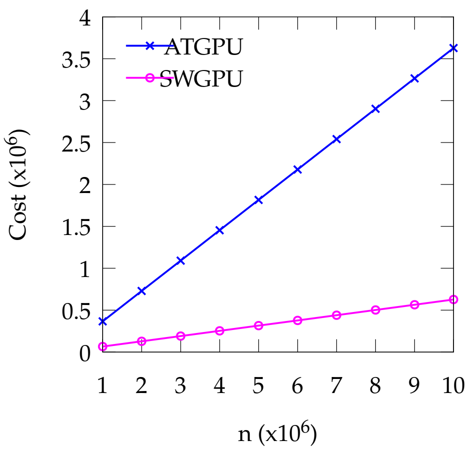
For two vectors $A = (a_1, a_2, ..., a_n), B = (b_1, b_2, ..., b_n)$, the addition is given as $A + B = (a_1 + b_1, a_2 + b_2, ..., a_n + b_n)$. We implement a simple GPU kernel that adds two Vectors of n integers. An element of the answer vector c_i is independent, making this an embarrassingly parallel problem. We assign n threads, with each thread i calculating the value $c_i = a_i + b_i$.

ATGPU Analysis. We give pseudocode in Algorithm 2, with analysis on the ATGPU model below. The *for all* loop on lines 3-11 is executed once, therefore the number of rounds is 1. The cores each execute 13 operations with no divergence, so the parallel time complexity is $O(1)$. Global memory access operations on lines 5,6, and 9 are coalesced meaning only a single block per instruction is accessed by the multiprocessor. As there are k multiprocessors, the I/O complexity is $O(\lambda k)$. There are $3n$ words stored in global memory, meaning the global memory complexity is $O(n)$. Each core stores one value in shared memory, so the shared memory complexity is $O(b)$. There are 3 transfer operations of $3n$ words in total, giving a transfer complexity of $O(\alpha + \beta n)$. The cost is $O\left(\alpha + \beta n + \frac{1+\lambda k}{\gamma} + \sigma\right)$. The GPU-cost is $O\left(\alpha + \beta n + \frac{k}{k'\ell}\frac{1+\lambda k}{\gamma} + \sigma\right)$.

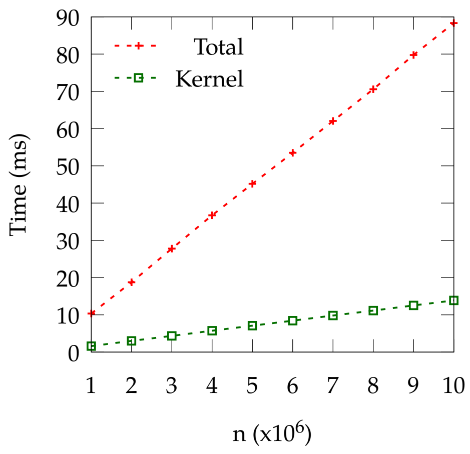
We plot the GPU-cost function in Figure 4.3a.

Experimental Setting. We run the Vector Addition kernel on randomly generated data sets, from $n = 1{,}000{,}000 \rightarrow 10{,}000{,}000$ with results shown in Figure 4.3b.

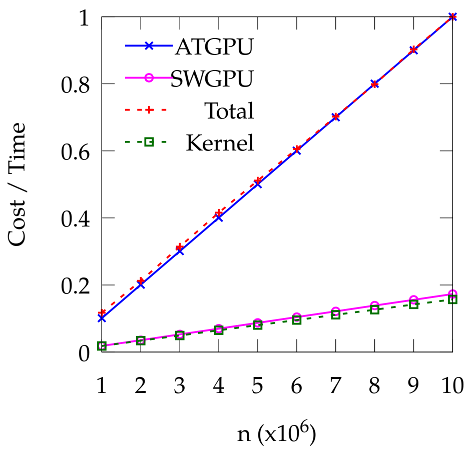
Discussion. Figure 4.3b shows that the growth of total running time is much steeper than the kernel running time. Data transfer takes an average of 84% of the total time, meaning data transfer between CPU and GPU has significantly affected



(a) Predicted results.



(b) Observed results.



(c) Normalised results.

Figure 4.3: Results for vector addition.

the running time of the algorithm. We compare this to Figure 4.3a, where we see that ATGPU function grows at a much quicker rate than the SW-GPU function. In Figure 4.3c, we normalise all data on a $0 \rightarrow 1$ scale. We see that the SW-GPU function has a much slower rate of growth than the total running time, and that the ATGPU function has a rate of growth which is much closer to the actual total running time. This means that by capturing the data transfer, the ATGPU is able to better predict the total running time of this algorithm than the SW-GPU, which does not capture

Algorithm 2 Vector Addition on ATGPU

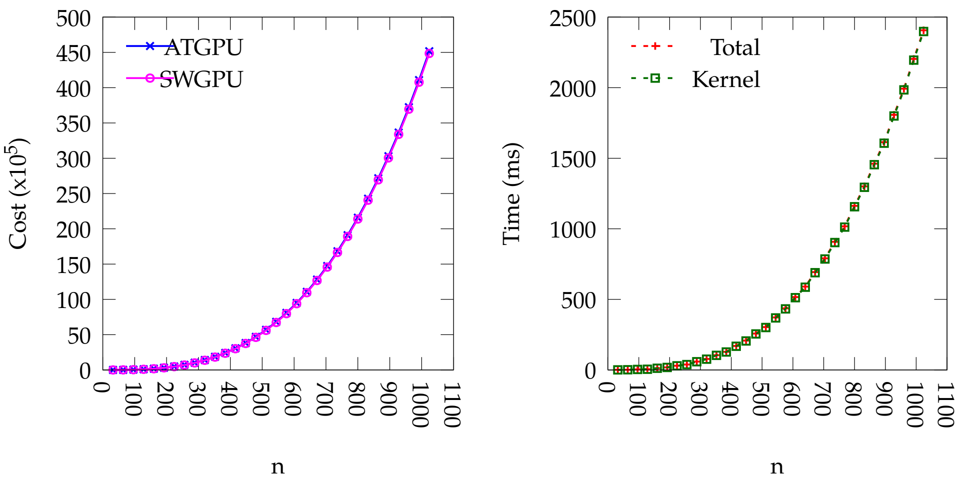
Input: Two vectors A, B of length n

Output: $C = A + B$

```
1: a ⇚ A                                                   ▷ Transfer data to Device
2: b ⇚ B
3: for all $mp_i \in MP[mp_0, ..., mp_{k-1}]$ in parallel do     ▷ Start GPU
4:     for all $c_{i,j} \in C_\rho$ in parallel do
5:         $\_a[j] \Leftarrow a[ib + j]$
6:         $\_b[j] \Leftarrow b[ib + j]$                     ▷ Work in shared memory
7:         $\_c[j] \leftarrow \_a[j] + \_b[j]$                ▷ Output to
8:                                                            Global memory
9:         $c[ib + j] \Leftarrow \_c[j]$
                                                           ▷ Transfer output to Host
10: $C \Lleftarrow c$
```

data transfer.

4.4.2 Matrix Multiplication



(a) Predicted results. (b) Observed results.

Figure 4.4: Results for matrix multiplication.

We now investigate matrix multiplication. For two matrices A, B, we multiply them into the matrix C. The matrix multiplication $AB = C$, where A, B are $n \times n$ matrices, is given in Equation 4.3 .

$$c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} b_{k,j} \tag{4.3}$$

We give ATGPU pseudocode and analysis below. We use a well known GPU method for matrix multiplication in shared memory (introduced in CUDA Programming Guide [67]), modified for the single warp per multiprocessor of our model.

Algorithm 3 Matrix Multiplication on ATGPU

Input: Two $n \times n$ matrices of integers, A and B.
Output: $C = A \times B$ **Assert:**
W is n rounded to the highest b (padded side length)
$T = \frac{W}{b}$ (num tiles in each dimension)
$k = T^2$ (each MP calculates a single tile)
$x = \frac{\rho}{T}$ (x coordinate of tile)
$y = \rho \mod T$ (y coordinate of tile)

```
 1: _A ⇚ A                                                ▷ Transfer input data to Device
 2: _B ⇚ B
 3: for all $mp_\rho \in MP[mp_0, ..., mp_{k-1}]$ in parallel do
 4:     for all $c^\rho_\epsilon \in C^\rho$ in parallel do
 5:         for $i = 0 \to b$ do                          ▷ Partial answer must be initialised to 0
 6:             $\_c[ib + \epsilon] = 0$
 7:         for $t = 0 \to T - 1$ do                      ▷ Calculate a tile at a time
 8:             for $i = 0 \to b - 1$ do      ▷ Copy a $b \times b$ block of $\_A$ and $\_B$ to Shared Memory
 9:                 $\_a[ib + \epsilon] \Leftarrow \_A[(x + i)W + tb + \epsilon]$
10:                 $\_b[ib + \epsilon] \Leftarrow \_B[(tb + i)W + y + \epsilon]$
11:             for $i = 0 \to b$ do                      ▷ Calculate this part of the solution
12:                 for $j = 0 \to b$ do
13:                     $\_c[ib + \epsilon] \leftarrow \_c[ib + \epsilon] + (\_a[ib + j] * \_b[jb + \epsilon])$
14:         for $i = 0 \to b$ do                          ▷ Copy $\_c$ to $\_C$ in Global Memory
15:             $\_C[(x + i)W + yb + \epsilon] \Leftarrow \_c[ib + \epsilon]$
16: C ⇚ _C                                                ▷ Transfer answer data from Device to Host
```

ATGPU Analysis. The *for all* loop on lines 3-15 executes only once, meaning the number of rounds is 1 The loop on lines 7-15 runs for $O(T) = O(\frac{n}{b})$ time. The inner loops on lines 8-10 and 11-13 run for $O(b)$ time, meaning the code inside these loops run for $O(n)$ time. The loop on line 12 runs $O(\frac{n}{b} \times b \times b) = O(nb)$ time, meaning the parallel time complexity of the algorithm is $O(nb)$. The I/O complexity for a single multiprocessor is $O(n)$ (from lines 9 and 10), yet as the number of multiprocessors

$k = O(n^2)$, the I/O complexity of the program is therefore $O(\lambda n^3)$. The global memory used is $O(n^2)$. The shared memory used is $O(b^2)$. The transfer complexity is $O(\alpha+\beta n^2)$.

The cost is $O\left(\alpha + \beta n^2 + \frac{nb+\lambda n^3}{\gamma} + \sigma\right)$. We plot the GPU-cost in Figure 4.4a.

Experimental Setting. We run the matrix multiplication kernel on randomly generated square matrices of side length $n = \{32, 64, ..., 1024\}$. We plot the observed results in Figure 4.4b.

Discussion. We can see from Figure 4.4b that there is little difference between the kernel running time and the total running time. This means that the data transfer does not affect the running time of this algorithm, and is a reflection of the analysis given on the ATGPU model. This particular application is an example where the ATGPU model gives analysis that is accurate, though no more useful than the existing AGPU and SW-GPU models. This is due to where the work of the algorithm is most heavily weighted; by considering the ATGPU analysis given, it can be seen that the I/O complexity (global memory access) is the fastest growing of all metrics, which is also captured by the existing models.

4.4.3 Reduction

The reduction of a n-sized vector A, for some operator $\oplus$, is calculated as $\oplus_{i=1}^{n} a_i$. We implement a reduction kernel [33] using the addition operator, to sum an array of n integers, using a tree-based method. The reduction kernel which we use has two distinct parts: the *local reduction* part, and the *global reduction* part. We implement each part as a separate kernel, which we describe below and demonstrate in Figures 4.5 and 4.6 .

The *local* reduction kernel initially creates a number of partial reduction values by reducing values as if they are arranged in a table, each core of the MP reducing a single column of data. The result of this is b partial reduction values within the shared memory of each MP. The kernel then proceeds to use a parallel tree-based method to reduce the b values to a single value, which is placed into global memory. The kernel then terminates.

The *global* reduction kernel is then executed, which uses multiple rounds to execute successive tree-based reduction kernels upon the contents of the memory, until a single value remains. This answer is then transferred back to the host.

ATGPU Analysis. The algorithm runs as in the "Reduction" pseudocode, each round using the output from the previous round as input.

The number of rounds is $O(\log k)$, the global memory complexity is $O(n)$, the shared memory complexity is $O(b)$, the parallel time complexity is $O(\log k \log b)$, the transfer

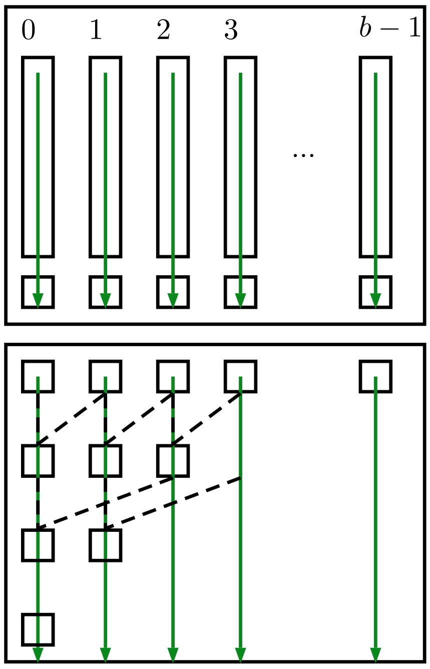


Figure 4.5: Demonstration of Local reduction kernel (green arrows represent threads).

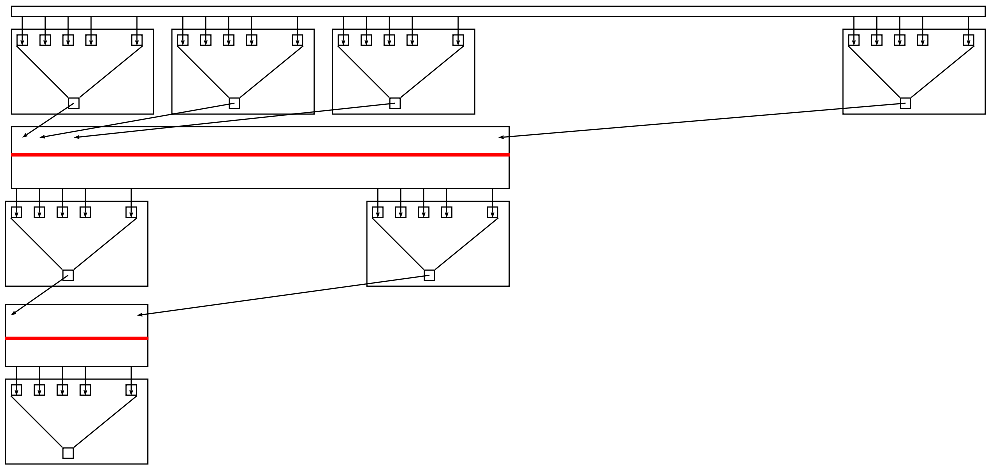


Figure 4.6: Demonstration of Global reduction kernel, where k values from the local reduction kernel are further reduced (red lines represent synchronisation operations).

complexity is $O(\alpha + \beta n)$ and the I/O complexity is $O\left(\lambda \frac{n}{b}\right)$. The cost is:

$$O\left(\alpha + \beta n + \frac{(\log k \log b) + \lambda(\frac{n}{b})}{\gamma} + \sigma \log k\right) \quad .$$

Algorithm 4 Reduction on ATGPU

Input: n integers allocated on GPU.

Output: $\oplus_{i=1}^{n} A[i]$

```
1: _A ⇚ A                                                  ▷ Transfer input data
2: for all $mp_\rho \in MP[mp_0, ..., mp_{k-1}]$ in parallel do   ▷ Start GPU Local reduction
3:     for all $c^\rho_\epsilon \in C^\rho$ in parallel do
4:         $\_ans[\epsilon] \leftarrow 0$                    ▷ Initialise Reduction for column
5:         for $i = 0 \rightarrow \lceil \frac{n}{p} \rceil$ do     ▷ Each core reduces a column of input
6:             $\_ans[\epsilon] \leftarrow \_ans[\epsilon] \oplus \_A[\rho b + \epsilon + ip]$
7:         for $i = 0 \rightarrow \log_2 b$ do              ▷ Reduce $b$ values using Tree method
8:             if $\epsilon < \frac{b}{2^i}$ then
9:                 $\_ans[\epsilon] \leftarrow \_ans[\epsilon] \oplus \_ans[\epsilon + \frac{b}{2^i}]$
10:        if $\epsilon == 0$ then
11:            $\_A[\rho] \Leftarrow \_ans[\epsilon]$ ▷ Place calculated reduction value into Global memory
12: for $i = 0 \rightarrow \log_{2b} k$ do                  ▷ Start GPU Global reduction
13:     for all $mp_\rho \in MP[mp_0, ..., mp_{(k(\frac{1}{2b})^i)} - 1]$ in parallel do
14:         for all $c^\rho_\epsilon \in C^\rho$ in parallel do
15:             $\_ans[\epsilon] \Leftarrow \_A[\rho 2b + \epsilon] \oplus \_T[\rho 2b + b + \epsilon]$
16:             for $j = 0 \rightarrow \lfloor \log_2 b \rfloor$ do
17:                 if $\epsilon < \frac{b}{2^j}$ then
18:                     $\_ans[\epsilon] \leftarrow \_ans[\epsilon] \oplus \_ans[\epsilon + \frac{b}{2^j}]$
19:             if $\epsilon == 0$ then
20:                 $\_A[\rho] \Leftarrow \_ans[\epsilon]$
21: Ans ⇚ _A[0]                                             ▷ Transfer answer
```

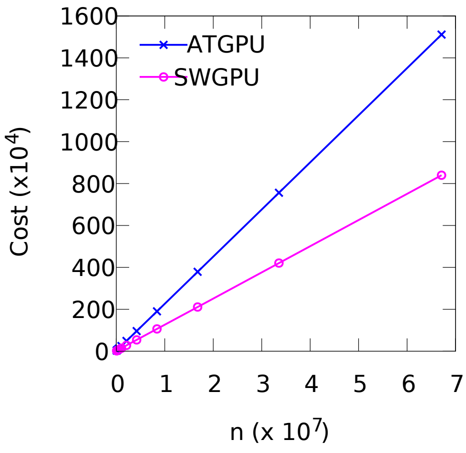
The GPU-cost is:

$$O\left(\alpha + \beta n + \frac{(\lceil \frac{n}{bk'\ell} \rceil \log k \log b + \lambda(\frac{n}{b})}{\gamma} + \sigma \log k\right) .$$

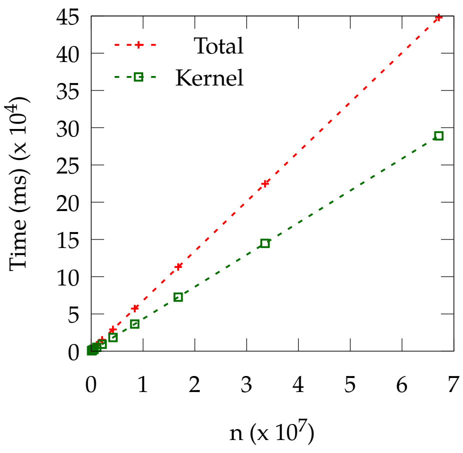
We plot the GPU cost in Figure 4.7a.

Experimental Setting. We run the reduction kernel on randomly generated vectors of 0/1 values, being sizes $n = \{2^{16}, 2^{17}, ..., 2^{26}\}$. This range of input sizes was chosen to create a large enough amount of data transfer and computation so that the differences in run time can be reliably observed, whilst not exhausting the GPU resources. We plot the observed results in Figure 4.7b.

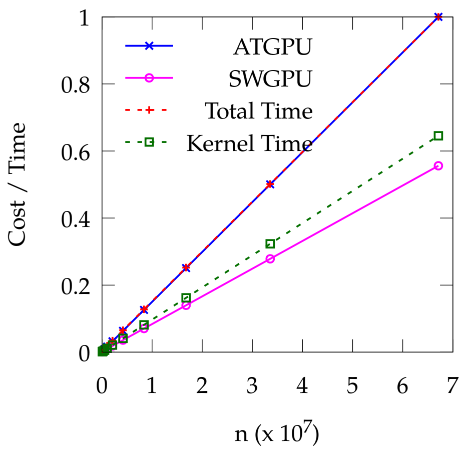
Discussion. Figure 4.7b shows that the growth of total running time is steeper than the kernel running time,though there is not as stark a difference as in vector addition. On average, the data transfer takes 35% of the total running time. We compare this to Figure 4.7a, where we see that ATGPU function grows at a quicker rate than the SWGPU function. We see in Figure 4.7c that the ATGPU function has a rate of growth closer than the SWGPU function to the actual total running time. Therefore, as in the vector addition example, capturing the data transfer gives a more accurate prediction of the actual running time.



(a) Predicted results.



(b) Observed results.



(c) Normalised results.

Figure 4.7: Results for reduction.

Table 4.2: Comparison of asymptotic complexity of various components for the ATGPU analysis of several algorithms.

Component	**Vector Addition (Algorithm 2)**	**Matrix Multiplication (Algorithm 3)**	**Reduction (Algorithm 4)**
Parallel Time Complexity	$O(1)$	$O(nb)$	$O(\log k \log b)$
I/O Complexity	$O(\lambda k)$	$O(\lambda n^3)$	$O(\lambda \frac{n}{b})$
Transfer Complexity	$O(\alpha + \beta n)$	$O(\alpha + \beta n^2)$	$O(\alpha + \beta n)$
Dominant Function	Transfer Complexity	I/O Complexity	Transfer Complexity

4.4.4 Summary

We now give a summary of the experimental results which evaluate the ATGPU model. We do this by considering the ATGPU cost functions in finer detail, and analysing how each component contributes to the trend of observed running time. We also demonstrate the accuracy of our model by comparing the proportion of time taken for the GPU CPU data transfer with that which was predicted using the ATGPU cost function.

In the computational problems studied, we see that for vector addition and reduction, it is not sufficient to simply capture the kernel execution for predicting the actual running time. We show that by capturing the data transfer in addition to the kernel, it is possible to obtain a trend that is much closer to the actual running time, than if the data transfer was not captured. We also show a case where our model useful in predicting the trend of running time, yet proves to be no more useful than the existing models; in matrix multiplication, there is little difference between the kernel and total running times, so the kernel can provide an accurate prediction of the total running time in this case.

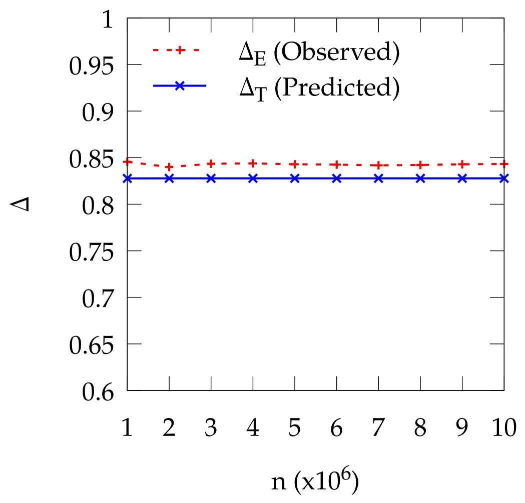
When we consider the ATGPU cost function analyses for these problems in more detail, we can see that in each case, the dominating complexity metric (Transfer complexity, I/O complexity, Time complexity) dictates whether or not the data transfer between CPU and GPU has a significant effect upon the true running time of the GPU program. This is demonstrated in Table 4.2.

The ability to distinguish a computational problem as *transfer bound* is a new capability that distinguishes the ATGPU from the existing models, and allows algorithm designers to further focus their effort on optimising particular parts of their code. The ability to compare the trend of the data transfer against that of time and I/O is also important in cases where the trends of all metrics are similar, distinguished only by the parameters related to transfer operations and memory access costs. In cases like this, the data transfer between CPU and GPU would be dominant because the values of α and β (memory transaction staging cost and transfer cost for a single word) are typically much larger than the cost associated with accessing a global memory block (λ). The ATGPU model is therefore able to distinguish between these cases (this is demonstrated in Chapter 5), where existing models can not.

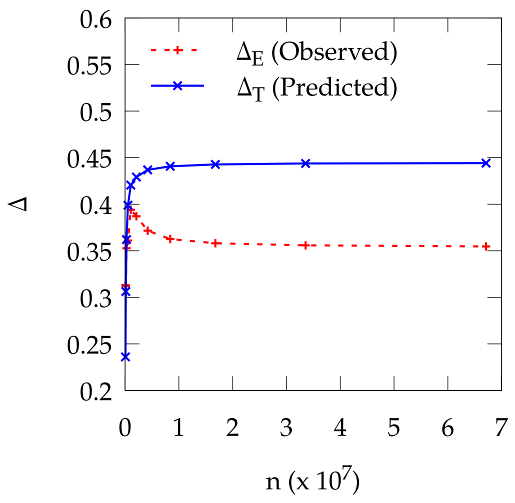
To demonstrate the accuracy of our model, we have also calculated the relative proportions of time/cost allocated to data transfer, and we see that our model has a good level of accuracy, as seen in Figure 4.8. We see that the predicted proportions of cost allocated to data transfer are on average to within 1.5% of observed proportions for vector addition, to within an average of 0.76% for matrix multiplication, and to within an average of 5.49% for reduction. We also calculate that the SWGPU captures on average only 16% of the actual running time for the vector addition example, and only 58% of actual running time for the reduction example, with 89% of the actual time being captured in the matrix multiplication example.

4.5 Conclusion

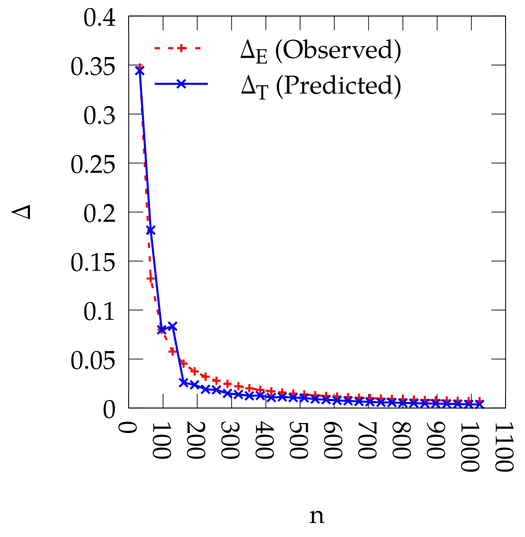
In this chapter, we introduce a model called Abstract Transferring GPU (ATGPU), applicable to design and analysis of GPU algorithms. The model is an extension of existing abstract models. ATGPU is, to our knowledge, the first GPU abstract model containing data transfer between host and device as an integral part. The model contains an architecture, a pseudocode and cost functions, allowing an algorithm to be analysed on a “perfect GPU" and simulated on a real GPU. We show via experiments that existing models cannot sufficiently capture all of the actual running time of a GPU algorithm in all cases, as they do not capture data transfer. We show that by capturing data transfer with our model, we are able to obtain more accurate predictions of the GPU algorithm actual running time. We demonstrate two cases where capturing both the kernel and data transfer in our model is useful for better predicting the actual running time, and one case where capturing only the kernel running time is sufficient.



(a) Vector addition.



(b) Reduction.



(c) Matrix multiplication.

Figure 4.8: Proportions (Δ) of time/cost for data transfer.

Chapter 5

Semi-global Sequence Alignment with Gaps with GPU

5.1 Introduction

In this chapter we consider the pair-wise semiglobal sequence alignment problem with gaps,which is motivated by the *re-sequencing* problem that requires assembly short read sequences into a genome sequence by referring to a reference sequence. The problem has been studied before for single gap and bounded number of gaps. For single gap, there is a GPU-based algorithm proposed [5]. In this chapter we propose a GPU-based algorithm for the bounded number of gaps case, called `GPUGapsMis`. We implement the algorithm and compare the performance with the CPU-based algorithm, called `CPUGapsMis`; The algorithm has two distinct stages: the *alignment phase*, and the *backtrack phase.* We investigate several different approaches, in order to determine the most favourable for this problem, by means of a Hybrid model or a wholly-GPU based model, as well as the alignment of single text sequences or multiple text sequences on the GPU at a time. We show that the alignment phase of the algorithm is a good candidate for parallelisation, with peak speed up of 11 times achieved on our system. We show that although the backtracking phase is sequential, it is more beneficial to perform it on the GPU, as opposed to returning to the CPU and performing the operation there. When performing both phases on the GPU, `GPUGapsMis` achieves a peak speed up of 10.4 times on our system against `CPUGapsMis`. Our data parallel GPU algorithm achieves results which are an improvement on those reported by an existing GPU data parallel implementation [68]. Further to this, we give analysis on both the AGPU model and the ATGPU model, showing that the ATGPU model is able to more accurately predict the best performing GPU approach, thanks to the ability to capture data transfer.

Organisation of Chapter

This chapter is organised as follows: Section 5.2 provides the notations required and the problem definition; Section 5.3 details our proposed solution; Sections 5.4 and 5.6 detail experimental evaluation and discuss the results obtained; Finally, Section 5.7 concludes the chapter.

5.2 Problem Definition and Preliminaries

5.2.1 Problem Definition

Given the preliminary discussion in Section 3.2.1, we are ready to define the pair-wise sequence alignment with bounded number of gaps problem.

Definition 1. *Given a text T of length n, a pattern X of length $m < n$, and an integer $z > 0$, the problem is to find all prefixes T' of T where the corresponding alignment of T' and X in the form $z_0 g_0 z_1 g_1 ... z_{\alpha-1} g_{\alpha-1} z_\alpha$ satisfies the property that $\alpha \leq z$ and the score is the maximum.*

We are required to find the prefixes of text T which satisfy the properties described, because we use the *seed and extend* strategy [3] for alignment, whereby a high quality alignment seed (at the start of the sequences) is matched, and the alignment is then extended. This involves aligning prefixes of the text T with the entirety of the pattern X, known as a *semi-global* alignment. This is as opposed to a *global* alignment, which aligns the entirety of T and X, and opposed to a *local* alignment, which aligns substrings of both T and X.

5.2.2 Dynamic Programming Algorithm

Adapting the dynamic programming algorithm in [5] to allow general score function, our algorithm is based on the following dynamic programming framework. We keep a matrix $G_q[i, j]$, which stores the maximum alignment score between the prefixes $t_1 t_2 \cdots t_i$ of the text T and $x_1 x_2 \cdots x_j$ of the pattern X, allowing up to q gaps, where $0 \leq q \leq z$. We assume that the *gap extension penalty* is the same regardless of which letter is aligned with the gap character, i.e., there exists a constant δ_E such that $\delta(x, *) = \delta(*, x) = \delta_E$ for all $x \in \Sigma$.

Note that the restriction on the number of gaps can be observed by calculating the matrix up to G_z.

$$G_0[i,j] = \begin{cases} G_0[i-1,j-1] + \delta(t_i, x_j) & \text{if } i == j \\ -\infty & \text{otherwise} \end{cases}$$

$$G_q[i,j] = \max \begin{cases} \delta_O + \sum_{l=0}^{j-2} \delta(*, x_l) & \text{if } i == 0 \\ \delta_O + \sum_{l=0}^{i-2} \delta(t_l, *) & \text{if } j == 0 \\ \max_{r=1}^{j-i}(G_{q-1}[i, j-r] + \delta_O + \sum_{l=j-r+2}^{j} \delta(*, x_l)) & \\ \qquad \text{if } i < j & \\ \max_{r=1}^{i-j}(G_{q-1}[i-r, j] + \delta_O + \sum_{l=i-r+2}^{i} \delta(t_l, *)) & \\ \qquad \text{if } i > j & \\ G_q[i-1, j-1] + \delta(t_i, x_j) & \text{otherwise} \end{cases}$$

A naïve implementation of the dynamic programming recurrences could result in an algorithm of $O(znm(n+m))$ time, yet it was demonstrated in [5] that storing the information of the gap insertion points (the value of r which maximises the scores on lines 3 and 4 of the recurrence) would make the look-up possible in $O(1)$ time, giving an improved time complexity of $O(znm)$.

We keep a matrix H_q which stores information on gap length and placement (at which position and in which sequence does the gap occur), for the alignment up to and including the pair (t_i, x_j) which includes at most q gap sequences, for $0 \leq q \leq z$. The cells are populated as shown in the recurrence, with $H_q[i,j]$ being populated after $G_q[i,j]$ has been calculated.

$$H_q[i,j] = \begin{cases} 0 & (t_i, x_j) \text{ in alignment} \\ r > 0 & (t_i, *) \text{ in alignment, gap of } r \\ r < 0 & (*, x_j) \text{ in alignment, gap of } r \end{cases}$$

The alignment is retrieved using the linear time algorithm *GapsPos* [5]. Starting from the position of the alignment score reported by GapsMis, the alignment is built backwards, moving towards the start of the sequences. The value within each cell of H_q dictates how the row and column indices are adjusted; either both are decremented by

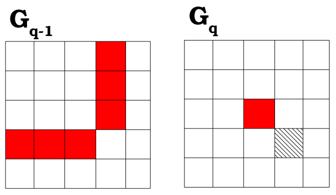


Figure 5.1: The dependencies whilst calculating cell $G_q[i, j]$ (hatched) are shown in solid filled cells.

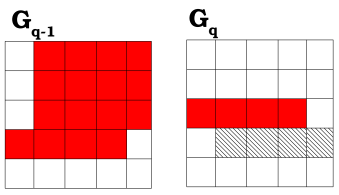


Figure 5.2: The dependencies whilst calculating the row $G_q[i, *]$ (hatched) are shown in solid filled cells.

one in the case of no gap, or the column index (row index) is decreased by the absolute value of the cell to give a gap in the pattern (text).

5.3 Our Solution

In the following section we describe `GPUGapsMis`, our solution to the semi-global sequence alignment with bounded gaps problem. We also give theoretical analysis of the proposed solution on the AGPU model in Section 5.3.2 and the ATGPU model in Section 5.5.

5.3.1 Idea of Parallelisation

As the recurrence in Section 5.2.2 shows, the dependencies for the cell $G_q[i, j]$ lie within the cell $G_q[i - 1-, j - 1]$ and the range of cells $G_{q-1}[0...i, 0...j]$, where $0 < q \leq z$. Therefore as shown in Figures 5.1 and 5.2, we are able to express parallelism along each row of the dynamic programming matrix in order to create a data-parallel solution. As the dependencies required for calculating cells within G_q all lie either in G_q or G_{q-1}, we only require the current and previous one G matrix for computation to be stored.

We keep the following data in the global memory: text sequence data, pattern sequence data, score data and matrices G_q, G_{q-1}, H data for each sequence pair being aligned. Pointers kept in private memory, which point to G_q and G_{q-1} in global memory, are updated at each iteration of the number of gaps calculated, and the H matrix is only used on the final iteration, as for q gaps, only the data in H_q is required when computing the optimal alignment.

The shared memory space contains the pattern data, the text character for current matrix row i, and the buffers required for our aggressive double buffer technique. This double buffer technique is laid out as follows: $currRow, prevRow$ hold rows $i, i-1$ of G_q, $prevGprevRow, prevGcurrRow$ hold rows $i, i-1$ of G_{q-1}, along with $maxIVal$, $maxILoc$, $maxJVal$, $maxJLoc$ hold the information relating to optimal gap insertion points from G_{q-1}. As with the global memory pointers, *currRow*, *prevRow*,*prevGprevRow*, *prevGcurrRow* are updated at each row iteration, and filled with any required data. The double buffer technique allows us to re-use the same two allocated memory locations for any number of gaps when solving the sequence alignment problem. It not only allows us to reliably calculate the memory footprint, but enables the GPU to be used for aligning with any number of gaps. If the double buffer technique was not used, then the memory footprint would become very large for even small numbers of gaps, meaning the GPU resources would be exhausted and therefore unable to be used. In order to maximise use of global memory access bandwidth, we need to use vectorised memory access operations. In order for vectorised memory accesses to be made possible, we pad with dummy data the shared memory row caches, the patterns, and the matrix rows.

We now explain the intuition behind the parallelisation for a single sequence pair, executed by a single thread block on the GPU. This is repeated for additional sequence pairs in a separate thread block per sequence pair. Initially, the *pattern sequence* is fetched from *global memory* into the shared memory. We calculate matrix G_0 followed by $G_1, G_2, ..., G_z, H_z$, for up to z gaps. Each matrix is calculated in a row-wise, data parallel fashion, with parallelism being expressed along each row. As each matrix is being calculated, the row number is iterated, and the number of gaps is iterated.

To calculate a row of G_q, we fetch the text character from the global memory, and the relevant gap insertion data relating to G_{q-1} . We then initialise the first cell of the row, and proceed to iterate across the row for all threads in a tiling fashion. The data required for the calculation is held in shared memory. At the end of row calculation, we copy the values to global memory and retain in shared memory for the next row, discarding the previous row. At the end of a matrix calculation, the pointers to the current G matrix and previous G matrix are updated, so we are using a double buffer approach on several levels.

For a number of gaps $0 < q \leq z$, we calculate the matrices G_q (H_q) in the following way, which is explained visually in Figure 5.3:

- Initialise the first row ($G_q[0,*]$) by storing the values into shared memory $previousRow, hRow$, with each warp of the block taking a tile.
- Store data of $previousRow, hRow$ in global memory.
- Fetch data of $G_{q-1}[0,*]$ from global memory into shared memory $prevGprevRow$, in preparation for calculating the subsequent rows of G_q
- Loop for each row $1 \leq i \leq n$

  a. Fetch $G_{q-1}[i,*]$ into shared memory $prevGcurrRow$.

  b. Calculate the best gap insertion point into the pattern, for each position $0 \leq j \leq m$, in $O(\log m)$ time. We use a tree-based method for finding the maximal gap insertion point from $prevGprevRow$. The maximal gap insertion point for $G_q[i,j]$ exists in the range $G_{q-1}[i,0,...,j-1]$. We are able to calculate the maximal insertion points for an entire row in the same routine. We calculate, for each position $0 < j < m$ the alignment score and location of the best point, up to but not including j itself. We modify a parallel prefix scan algorithm to use the max operator as opposed to the summation operator to calculate this.

  c. Update the gap insertion points into the text, if this is required, by comparing $maxIVal$, $maxILoc$, $prevGprevRow$.

  d. Compare values in shared memory, for the three options of alignment: continue the current alignment ($prevRow$), insert gap in text ($maxIVal, maxILoc$), or insert gap in pattern ($maxJVal, maxJLoc$). Place optimal value into $currRow$ and relevant gap value into $hRow$. Now place $currRow, hRow$ into Global Memory.

  e. Update the pointers of ($prevGcurrRow, prevGprevRow$) ($prevRow, currRow$) in preparation for calculating row $i+1$

The algorithm *GapsPos* calculates the optimal alignment path for the two sequences, which we refer to as *backtracking*. GapsPos is performed sequentially using a single thread.

Difference from existing data-parallel implementation. Ojiaku [68] proposed a data-parallel solution to this problem, reporting experimental results of a 5 times speed

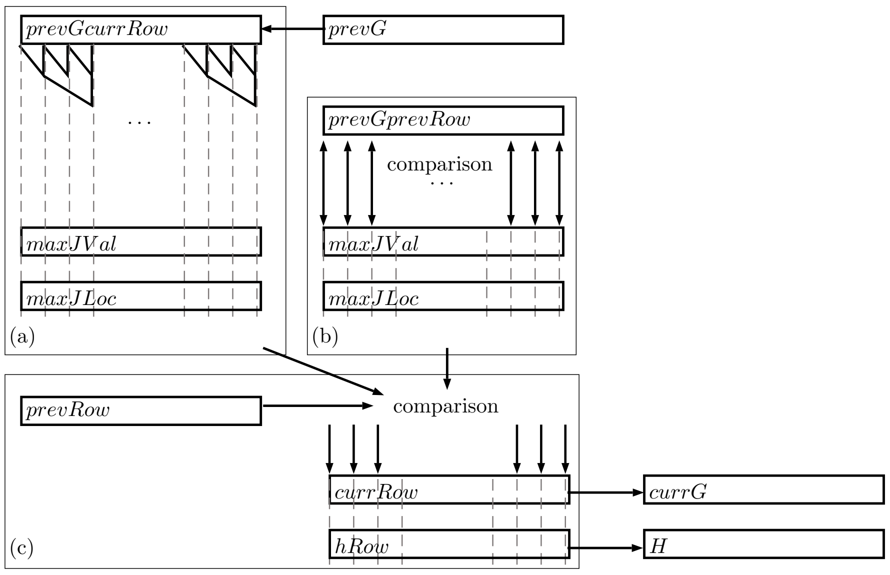


Figure 5.3: Idea of parallelisation for GapsMis. (a) Best gap insertion points in pattern are found. (b) Best gap insertion points in text are updated, if needed (c) Best score is calculated, and placed into global memory.

up against a single thread of the CPU. We evaluate `GPUGapsMis` using a similar environment as that used in [68]. Our solution differs in that we reduce the amount of host device communication by running for all z gaps in a single kernel run, therefore not requiring any global synchronisation or data transfer between subsequent gap numbers. We also use a parallel tree-based method for finding the optimal gap insertion point, where as [68] uses a sequential method. Further to this, we investigate several approaches to calculating the backtracking, by performing this on the GPU. This is opposed to calculating the backtracking on the CPU only, as in [68].

5.3.2 AGPU Analysis

We now give analysis of `GPUGapsMis` using the AGPU model [39] which has been discussed in Chapter 2. We present AGPU Pseudocode in Algorithm 5 for `GPUGapsMis` aligning one sequence pair on a single multiprocessor. This is replicated for all qr sequence pairs in the input set (hence the qr multiprocessors), with Algorithm 5 corresponding to code run by a single CUDA thread block. Theoretical results are presented in Theorem 1.

Let $CORE[1, ..., b]$ be the set of cores within each multiprocessor, $\mathcal{T} = T_1, T_2, ..., T_q$ be the set of texts - each of length n, $\mathcal{X} = X_1, X_2, ..., X_r$ be the set of patterns - each of length m, where $n \geq m$, $z > 0$ be max number of gaps, δ_O be the gap opening penalty, and δ_E be the gap extension penalty.

Theorem 1. *The performance of* `GPUGapsMis` *on the AGPU model satisfies the following properties.*

(i) The time complexity is $O(zn\frac{m}{b})$.

(ii) The I/O complexity is $O(zqrn\frac{m}{b})$.

(iii) The global memory usage is $O(qrnm)$.

(iv) The shared memory usage is $O(m)$.

Proof. We now give a proof of the claims in Theorem 1, with line references to Algorithm 5.

(i) We see that the "Gaps" loop (lines 5-42) iterates z times in total with an additional procedure for initialising G_0. We see that the "row" loop (lines 15-41) is iterated n times in total, for all matrices $G_0 \rightarrow G_z$. When we examine the contents of the "row" loop, we see that there are several smaller loops each with $O(\frac{m}{b})$ iterations, and the procedure of finding the best gap insertion point takes time $O(\log m)$. The variable b corresponds to the number of cores present in the AGPU multiprocessor, is dictated by the architecture in use, and is typically much smaller than m. Therefore $O(\frac{m}{b}) \geq O(\log m)$, meaning the "row" loop interior is $O(\frac{m}{b})$.

Thus, a single multiprocessor executes in $O(zn\frac{m}{b})$ time.

(ii) We see that a multiprocessor accesses the entire pattern, meaning $\frac{m}{b}$ blocks are accessed. Further, for each individual row, we see that there are $4\frac{m}{b} + 1$ blocks of global memory accessed (for the text character, for fetching *prevGCurrRow*, for storing *currRow* and for storing *hRow*). Therefore, we see that each multiprocessor accesses $zn4\frac{m}{b} + zn$ blocks of global memory. Across the entire algorithm aligning qr sequence pairs, $qrzn4\frac{m}{b} + qrzn = O(qrzn\frac{m}{b})$ global memory blocks are accessed.

(iii) We see that for a multiprocessor aligning a sequence pair, the amount of global memory used is $2(n+1)(m+1)$ for the two G matrices, plus n integers for the text and m integers for the pattern, therefore for qr multiprocessors aligning qr sequence pairs, the amount of global memory used is $O(qrnm)$.

(iv) We see that for the shared memory data structures, no index over the value of m is ever read or written in any multiprocessor, this makes the complexity of shared memory used to be $O(m)$. □

Algorithm 5 `GPUGapsMis`on AGPU

```
1: for all $MP_\rho \in MP[0, ..., qr-1]$ do in parallel
2:     for all $core_\epsilon \in CORE[0, ..., b-1]$ do in parallel
3:         Initialise $G_0$ matrix into $previousG$
4:         // Calculate G Matrices for up to $z$ gaps
5:         for $q = 1 \to z$ do
6:             // Initialise row 0 of $G_q$
7:             if $\epsilon == 0$ then
8:                 $prevRow[0] \leftarrow 0$
9:             for $j = \{\epsilon+1, b+\epsilon+1, b+\epsilon+1, .....\} \leq m$ do
10:                $prevRow[j] \leftarrow \delta_E(j-1) + \delta_O$
11:            // Place $prevRow$ into $currG$
12:            for $j = \{\epsilon+1, b+\epsilon+1, 2b+\epsilon+1, .....\} \leq m$ do
13:                $currG[j] \Leftarrow prevRow[j]$
14:            // Calculate each row $i$ of Matrix $q$
15:            for $i = 1 \to n+1$ do
16:                $t \Leftarrow t[i]$ //Get Text Char
17:                // Fetch PrevGCurrRow
18:                for $j = \{\epsilon+1, b+\epsilon+1, 2b+\epsilon+1, .....\} \leq m$ do
19:                    $prevGcurrRow[j] \Leftarrow prevG[i, j]$
20:                // Calculate MaxILoc and MaxIVal from PrevGPrevRow
21:                for $j = \{\epsilon+1, b+\epsilon+1, 2b+\epsilon+1, ...\} \leq m$ do
22:                    $p[\epsilon] \leftarrow (i - maxILoc[j] - 1) * \delta_E$
23:                    if $maxIVal[j] + p[\epsilon] < prevGprevRow[j]$ then
24:                        $maxIVal[j] \leftarrow prevGprevRow[j]$
25:                        $maxILoc[j] \leftarrow i-1$
26:                // Calculate MaxJLoc and MaxJVal from PrevGCurrRow
27:                for $j = \{\epsilon+1, b+\epsilon+1, 2b+\epsilon+1, .....\} \leq m$ do
28:                    $maxJLoc[j] \leftarrow j$
29:                    $maxJVal[j] \leftarrow prevGcurrRow[j]$
30:                Use Tree based method to calculate the Max values
31:                // Now calculate the Values to place into the cells
32:                Initialise cell $G_q[i, 0]$
33:                if $\epsilon == 0$ then
34:                    $currRow[0] \leftarrow ((i-1) * \delta_E) + \delta_O$
35:                for $j = \{\epsilon, b+\epsilon, 2b+\epsilon, .....\} \leq m$ do
36:                    Look in $prevRow[j-1]$ to continue alignment
37:                    Look in $maxJVal$ for Gap in Pattern, applying penalty
38:                    Look in $maxIVal$ Gap in Text, applying penalty
39:                    Place max in $currRow[j]$
40:                    Calculate $hRow[j]$
41:                Copy $currRow$ to $currentG[i, *]$, $hRow$ to $H[i, *]$
42:                Update $currRow$ and $prevRow$ pointers
43:                Update $prevGprevRow$ and $prevGcurrRow$ pointers
44:            Update $currentG$ and $previousG$ pointers
45:     end parallel for
46: end parallel for
47: Report alignment score: $\max_{0 \leq \gamma \leq n} G_z[\gamma, m]$
```

5.4 Experimental Setting

Sequence alignment tools are typically used to search databases of known sequences, in order to find the best match for a query sequence, or set of query sequences.

Multiple Pairwise Sequence Alignment. In order to simulate a database search for the most optimal alignment for a set of query sequences, we align a set of query (pattern) sequences with a set of target (text) sequences.

Let $\mathcal{T} = T_1T_2, ..., T_q$ be the set of text sequences, and $\mathcal{X} = X_1, X_2, ..., X_r$ be the set of pattern sequences. We want to simulate searching in a database for the text sequence which gives the best alignment score for each individual pattern sequence. Let $\mathcal{S} = s_1, s_2, ..., s_{qr}$ be the set of sequence pairs, that is $\mathcal{S} = \mathcal{T} \times \mathcal{X}$. For each $s_i \in \mathcal{S}$, we solve the Semiglobal Sequence alignment with a bounded number of gaps problem, with either `GPUGapsMis` or `CPUGapsMis` - a sequential implementation of GapsMis on a single CPU thread.

Input Data. The sequence data used is taken from the NCBI DNA sequence database *GenBank* [64]. From the database, we choose from a selection of genomic data, namely *e.coli* and *Ralstonia solancearum*. We randomly select sequences from the database and further process each sequence by randomly removing some bases such that the length of the sequence becomes the length of the specific experiment sequence pair. This process produces synthetic data, yet since it is taken from real data, it is more realistic than that which is randomly generated (it is much more difficult to generate accurate and realistic patterns). The synthetic data used will give a good view of the performance of `GPUGapsMis` with real sequence data, as all data is treated identically by the algorithm.

For our experiments, we consider different input sets of text sequences and pattern sequences and for each set of sequences, we measure the performance of aligning all the sequence pairs in the set. E.g., for an input set of q text sequences and r pattern sequences, we align all $q \times r$ sequence pairs.

The sequences are stored in text files containing one sequence per line. There are eight input files for text sequences; each file contains $16, 32, 64, ..., 2048$ sequences, and each text sequence is 250bp in length. There are four input files for pattern sequences; the length of pattern sequences in each file is $50, 100, 150, 200$ bp, and each pattern file contains 100 pattern sequences. Each input set is formed by taking one text sequence file and one pattern sequence file.

Approaches. For evaluating the most effective way to use the GPU device as a co-processor for `GPUGapsMis`, we use several approaches detailed below, summarised in Table 5.1. We run control experiments with two versions of `CPUGapsMis`; `CPU-A` computes the alignment scores only, and `CPU-B` computes the alignment with backtracking.

There are in total six distinct approaches used in experiments with `GPUGapsMis`. The approaches for `GPUGapsMis` consist of a *batching method* and, where appropriate, a *backtracking method.* `GPU-A` computes the alignment scores only. Two approaches are considered for the batching method used when computing the alignment; *single text batching method* denoted by `-S`, and *multiple text batching method* denoted by `-M`. There are two approaches considered when we compute backtracking: `GPU-B` computes alignment with backtracking entirely on the GPU (we refer to this as the *GPU backtracking method*), and `GPU-H` computes the alignment scores on the GPU and computes backtracking on the CPU (we refer to this as the *Hybrid backtracking method*).

Single Text Batching Method. In the single text batching method, single text sequence is sent to the GPU, along with all pattern sequences. It is then aligned with all pattern sequences, before the next text is sent to the GPU for alignment with all pattern sequences. More precisely, the text data for $T_i \in \mathcal{T}$ is sent to the GPU, along with all pattern data. The kernel is run, and any output data is returned to the host. This is repeated for subsequent text sequences, meaning sequence data requires $O(qrm)$ words transferred to the GPU, and $O(rm)$ space allocated on the GPU. Single text batching method is denoted by `(s)` against the algorithm name.

Multiple Text Batching Method. In the multiple text batching method, we send multiple text sequences, along with all pattern sequences to the GPU, then allocate space in the GPU memory for ℓ sequence pairs to be aligned. The sequence data requires $O(qn+rm)$ words transferring to the GPU and $O(qn+rm)$ space allocated on the GPU. The qr alignment tasks required for aligning all sequence pairs in $\mathcal{S}$ are executed in $\lceil \frac{qr}{\ell} \rceil$ batches to ensure enough global memory is available to store the required matrices. The kernel is run for each batch, returning any output data to the host.

GPU Backtracking Method. In the GPU backtracking method, the backtracking algorithm GapsPos is performed on the GPU inside the same Kernel as the alignment scores calculation, by a single thread. The calculated data of size $O(qrz)$ is then returned to the host.

Hybrid Backtracking Method. In the hybrid backtracking method, the alignment score calculation is performed on the GPU. The backtracking H matrices of size $O(qrnm)$ are returned to the host asynchronously at the end of the kernel execution for each thread block, and GapsPos is performed on the CPU.

Table 5.1: Summary of approaches.

	Batching	*Backtracking*
`CPU-A`	-	-
`CPU-B`	-	On CPU
`GPU-S-A`	Single Text	-
`GPU-M-A`	Multiple Text	-
`GPU-S-H`	Single Text	Hybrid
`GPU-M-H`	Multiple Text	Hybrid
`GPU-S-B`	Single Text	GPU
`GPU-M-B`	Multiple Text	GPU

Verification of Correctness. Testing was carried out, whereby output matrices were compared between the CPU and GPU in order to verify the correctness of the calculations. This verification was done using 16 text sequences of length 250bp and 100 pattern sequences of each available length.

Performance Measurement. To evaluate the performance, we compare three measurements. *Latency* is measured as the total time taken. *Throughput* is a measure of how fast the data matrices are filled and is measured in Mega Cell Updates per Second (MCUPS), that is how many *millions of cell updates per second* occur. Precise throughput is calculated by dividing the total number of cells of G and H matrices to be updated in the entire execution, by the time taken to compute them. *Improvement ratio* is calculated as $\frac{CPU Latency}{GPU Latency}$, yet as this compares the performance of `CPUGapsMis` and `GPUGapsMis`, it could be calculated using throughput to obtain identical values. If this improvement ratio value is greater than 1, then `GPUGapsMis` has yielded an improvement against `CPUGapsMis`.

5.5 ATGPU Analysis of Different Approaches

In Section 5.3.2, we used the AGPU model to give theoretical analysis of `GPUGapsMis`. We highlighted in Chapter 2, and demonstrated in Chapter 4 that existing GPU abstract models do not capture everything that is required to give a fully accurate analysis of GPU computation; between the AGPU and the SW-GPU, none measured the data transfer between the CPU and GPU. Therefore, we now extend the existing AGPU analysis from Section 5.3.2 by analysing `GPU-S-A` and `GPU-M-A` on our ATGPU model, which we introduced in Chapter 4, comparing the resulting cost function for each approach. We analyse these two particular approaches as both contain only GPU execution, meaning we can fully analyse them using the ATGPU model. It is currently not possible to analyse CPU execution using the ATGPU model, therefore we are unable to fully analyse approaches that perform significant work on the CPU.

We note that both approaches perform a similar amount of work on the GPU (they align the same amount of sequence pairs) yet have differing data transfer requirements between the CPU and GPU. It is expected that there is a difference shown between the two approaches, which would lead a developer to the decision of which approach is best to use. It is also hoped that this difference would either not be evident, or be less evident, in the $SW - GPU$ cost function i.e. the ATGPU cost function without the data transfer, as in Chapter 4. For the analysis, we assume that all data that is required can fit onto the GPU memory.

`GPU-S-A` The approach of `GPU-S-A` has q rounds, and on each round, must transfer a single text (n words) and all pattern data (rm words), followed by the score data (r words) at the end of the round. Therefore, for each of the q rounds, the data transfer cost results to: $3\alpha + \beta(n + r + rm)$.

By taking elements of the AGPU analysis from Section 5.3.2, the resulting cost function for `GPU-S-A` is given in Expression 5.1, where α is the ATGPU cost to stage a Host Device memory transaction, β is the cost of transferring a single word to the device, b is the number of cores on each multiprocessor of the device, λ is the cost for accessing a block of global memory on the device ,γ is the operation rate (i.e. the number of operations completed per time-cost unit), and σ is the cost for synchronisation operations at the end of a computation round.

$$O\left(\alpha q + \beta(qrm) + \frac{qzn\frac{m}{b} + \lambda(zqr\frac{m}{b})}{\gamma} + \sigma q\right) \tag{5.1}$$

`GPU-M-A` Conversely, the approach of `GPU-M-A` has a single round, in which it transfers all text data (qn words), all pattern data (rm words) and the score data (qr words). Therefore the data transfer cost for this single round is $3\alpha + \beta(qn + rm + qr)$. By taking elements of the AGPU analysis from Section 5.3.2, the resulting cost function for `GPU-M-A` is given in Expression 5.2, where α is the ATGPU cost to stage a Host Device memory transaction, β is the cost of transferring a single word to the device, b is the number of cores on each multiprocessor of the device, λ is the cost for accessing a block of global memory on the device ,γ is the operation rate (i.e. the number of operations completed per time-cost unit), and σ is the cost for synchronisation operations at the end of a computation round.

$$O\left(\alpha + \beta(qn + rm + qr) + \frac{qzn\frac{m}{b} + \lambda(zqr\frac{m}{b})}{\gamma} + \sigma\right) \tag{5.2}$$

Discussion From our analysis using the ATGPU cost functions, we can see that both approaches have the same amount of computation and global memory access, both growing in a linear fashion, yet the `GPU-S-A` approach has a greater amount of synchronisation and data transfer by a factor of q, when compared to the `GPU-M-A` approach. We therefore expect to see in our experimental results, that the `GPU-M-A` approach outperforms the `GPU-S-A` approach and that both approaches have a linear trend in running time.

If we were to compare the two cost functions without the data transfer (so as to simulate the SW-GPU), then we would still see that the `GPU-M-A` approach is better performing, due to the lower amount of synchronisation that is required, however the synchronisation cost is negligible when compared to data transfer.

We discuss this particular hypothesis with regards to experimental results in Section 5.6.2.

5.6 Results

In this section, we present and discuss results from experiments carried out as described in Section 5.4. Following from the AGPU analysis in Section 5.3.2, and the we expect that the latency of `GPUGapsMis` is lower than `CPUGapsMis`, that latency increases linearly as input size increases, and that the improvement ratio of `GPUGapsMis` against `CPUGapsMis` decreases as the pattern length increases, because the amount of shared memory used corresponds with the pattern length, thereby affecting the occupancy level on the GPU.

We look to evaluate the performance change of `GPUGapsMis` as the input size increases, and to validate the AGPU analysis given in Section 5.3.2. We carry out all experiments described in Section 5.4, with all results presented in the supplementary material. In order to look closely at the trends, we focus in this section discussion on two settings: (i) increasing number of sequence pairs with pattern length fixed at 200bps; and (ii) increasing pattern length with number of sequence pairs fixed at 204800. Both settings investigate the effect of increasing data size. The results presented here appear in Tables in Appendix Chapter B as either the final rows, or the bottom-right sub tables. These results are representative of all other experiment results obtained. We also compare the performance of `GPUGapsMis` against the algorithm presented in [68].

Each figure in this section is made up of three subfigures; (a) latency results; (b) calculated throughput; (c) calculated improvement ratio.

5.6.1 Single Text Batching Method Results

First, we investigate results achieved by `GPUGapsMis` using the single text batching approach. We begin by discussing results obtained computing the alignment scores only, including comparison of our results against those obtained by the existing work in [68]. We then proceed to discuss the results obtained by computing alignment scores together with backtracking, before giving a summary of results obtained.

Alignment Scores Only

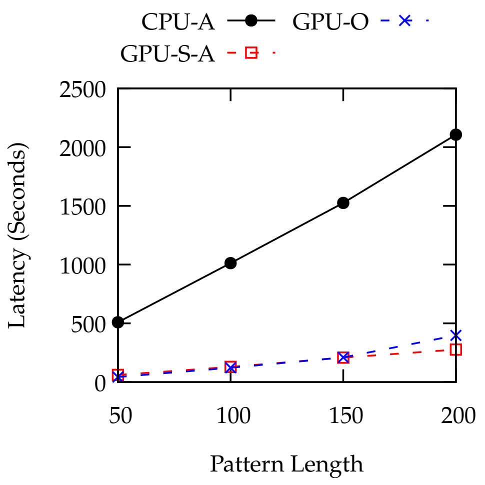
Results. Figures 5.4 and 5.5 show that the latency of `CPU-A` (black solid curve) and `GPU-S-A` (red dotted curve) increase linearly with the increase in size of input data. `GPU-S-A` has smaller latency than `CPU-A` in all cases and therefore outperforms `CPU-A` in all cases.The rate of increase in latency is 7.3 higher for `CPU-A` than for `GPU-S-A`.

This agrees with the AGPU analysis given in Section 5.3.2.

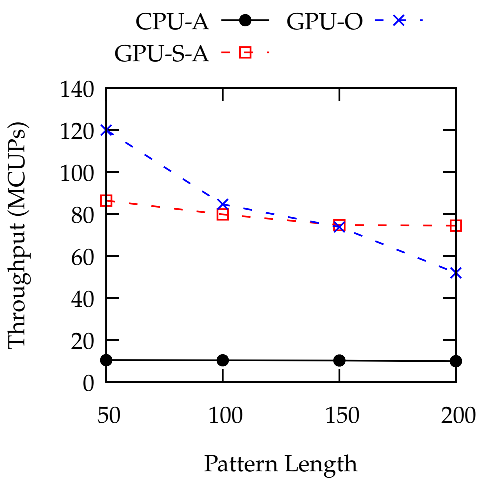
Figure 5.4 shows that the throughput of `CPU-A` stays constant while the improvement ratio and the throughput of `GPU-S-A` decrease as the pattern length increases. The throughput drops from 86.3 MCUPS at pattern length 50, to 74.5 MCUPS at pattern length 200, with improvement dropping from 8.4 to 7.3 times. Figure 5.5 shows that for increasing number of sequence pairs, the throughput (around 74 MCUPS) and the improvement ratio (around 7.6) of `GPU-S-A` remain stable.

Discussion. We see that the throughput and improvement ratio of `GPU-S-A` relative to `CPU-A` is sensitive to increasing pattern length, yet not sensitive to increasing number of sequence pairs to align. These performance metrics are less stable for increase in pattern length because shared memory use increases with pattern length, lowering the occupancy rate. This means less warps are available for hiding the latency of global memory access operations. In turn, input sets will take longer to process as the number of sequence pair alignment tasks concurrently run on the SM is decreased.

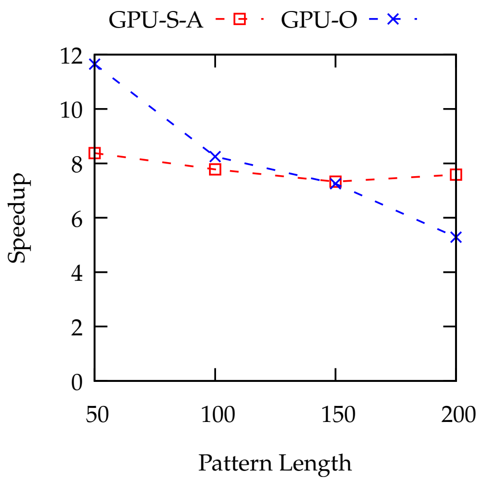
Comparison against existing work. The blue dotted curve in Figures 5.4 and 5.5 show the performance of the algorithm proposed in [68], `GPU-O`. We see that for some smaller pattern lengths, there is no improvement achieved, however as the pattern length is increased, we see that the performance level of `GPU-O` drops. `GPU-S-A` is less sensitive to increase in pattern length and for pattern lengths 150 or greater, `GPU-S-A`



(a) Latency



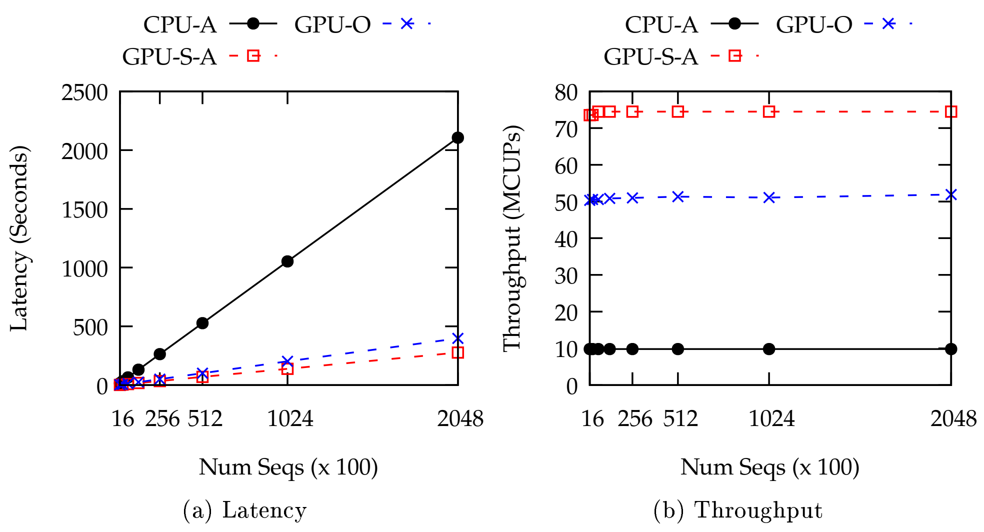
(b) Throughput



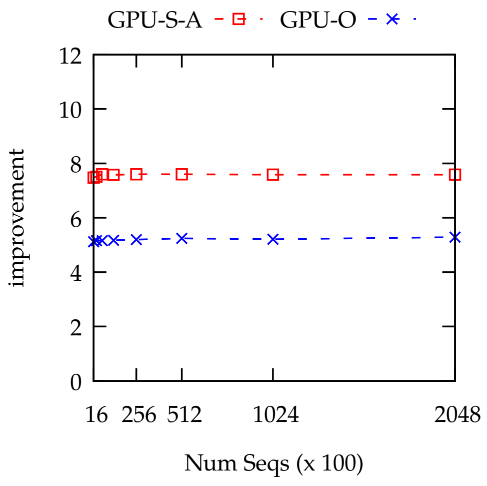
(c) Improvement vs CPU

Figure 5.4: Result for `GPU-S-A` and `GPU-O`, for input sets containing 204800 sequence pairs.

out performs `GPU-O`. Figure 5.4a shows the trend of `GPU-S-A` latency is the less steep of all. At its peak, `GPU-S-A` achieves throughput 23MCUPS higher than `GPU-O`, and a greater speed up of 7.59 against 5.29 of `GPU-O`.



(a) Latency (b) Throughput



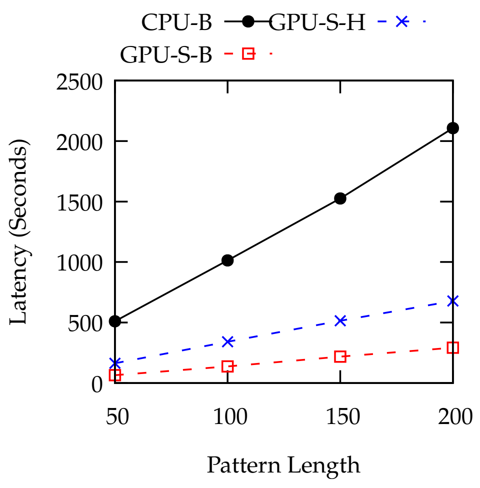
(c) Improvement

Figure 5.5: Result for `GPU-S-A` and `GPU-O`, for input sets containing patterns of length 200.

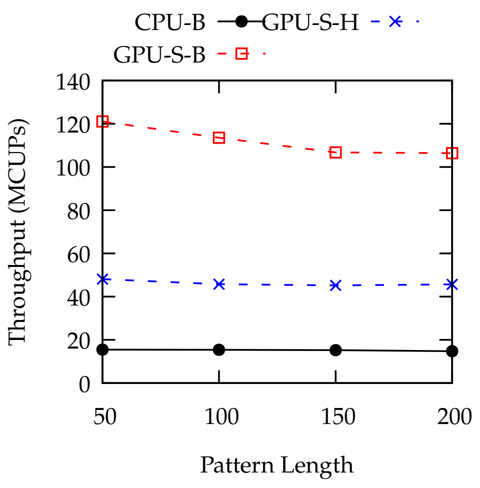
Alignment Scores with Backtracking

Results. Figures 5.6 and 5.7 show that when backtracking is also calculated, similar trends occur.

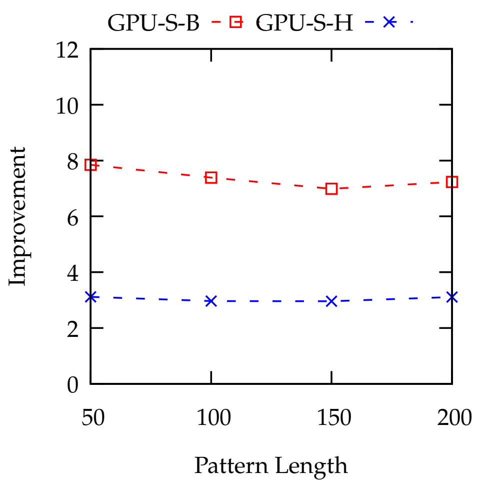
When we compare `GPU-B` and `GPU-H`, we see that the GPU backtracking approach (`GPU-B`) always outperforms the hybrid backtracking approach (`GPU-H`). In more details, Figure 5.6 shows when the pattern length increases, `GPU-H` achieves an improvement ratio of about 3.1 times while `GPU-B` achieves 7.0-7.8 times. With increasing number of sequence pairs, Figure 5.7 shows the improvement ratios of `GPU-H` and `GPU-B` are 3.1 times and 7.2 times, respectively.



(a) Latency



(b) Throughput



(c) Improvement vs CPU

Figure 5.6: Result for `GPU-S-B` and `GPU-S-H`, for input sets containing 204800 sequence pairs.

Discussion. We note that when backtracking is included, the throughput achieved is higher; see `GPU-S-A` vs `GPU-S-B` in Figures 5.5b and 5.7b and `CPU-A` vs `CPU-B` in Figures 5.4b and 5.6b. This is because the additional requirement to populate the H

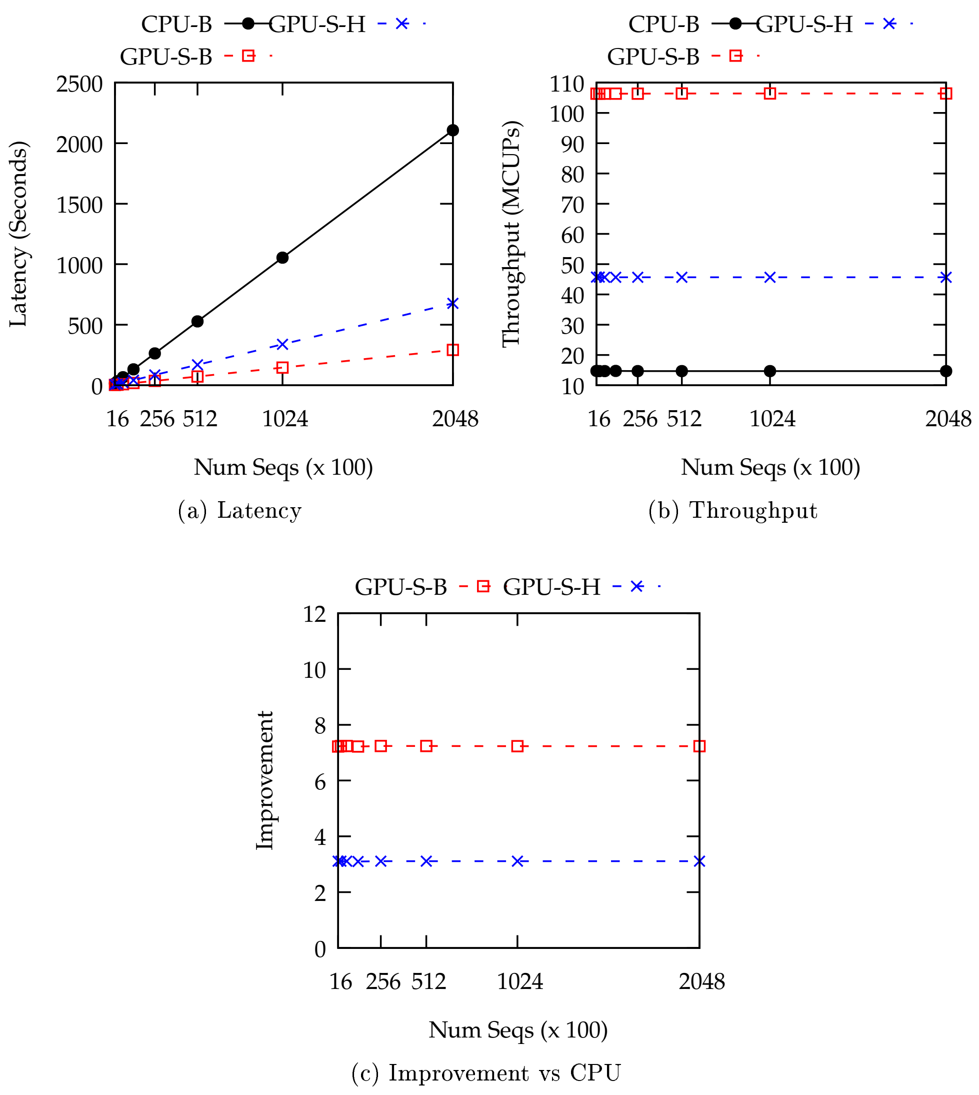


Figure 5.7: Result for `GPU-S-B` and `GPU-S-H`, for input sets containing patterns of length 200.

matrices require less work per cell than when populating the G matrices. Each row of the G matrices requires $O(\log m)$ computation by the multiprocessor, yet only $O(1)$ additional computation is required to calculate the values for each row of the H matrices.

The improvement ratio achieved by `GPU-S-B` was slightly lower than `GPU-S-A`, as shown in Figure 5.8. The backtracking algorithm GapsPos is a serial computation which has not been parallelised, and is not efficient on the GPU. Therefore it is faster on the CPU than on the GPU, giving rise to the lower improvement ratio exhibited by

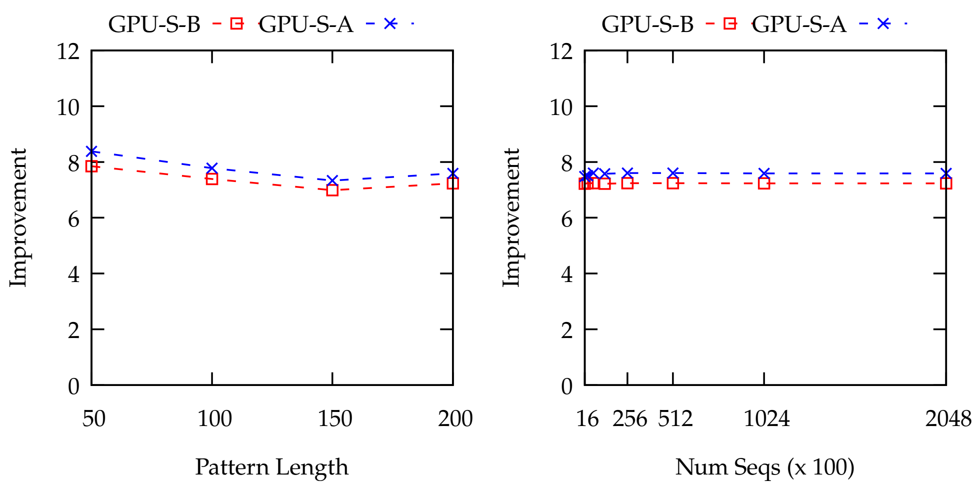


Figure 5.8: Comparison of improvement ratio between `GPU-S-A` and `GPU-S-B`.

`GPU-S-B` compared to `GPU-S-A`.

Figures 5.6 and 5.7 show that `GPU-S-H` achieved lower throughput than all other `GPUGapsMis` approaches, and exhibit lower sensitivity to increasing pattern length. The reason for this is the higher amount of data transfer between the CPU and the GPU. The cost associated with data transfer between CPU and GPU is very high, and can create a bottleneck in a GPU program.

Summary

In summary, taking into account of all experimental results presented in the supplementary material, `GPU-S-A` is on average 7.7 times faster than `CPU-A`. The peak improvement ratio is 8.4 times, when the pattern length is 50 and number of sequence pairs is 204800. Note that the throughput achieved in this setting is 86.4 MCUPS. On the other hand, when backtracking is considered, the peak throughput is increased to 121 MCUPS, though the improvement ratio is 7.8 times which is lower than the 8.4 times without backtracking. This peak occurs at the same input setting as above. This higher throughput but lower improvement ratio is due to less work required to calculate the additional cells during the backtracking phase, and the sequential backtracking algorithm being inefficient on the GPU.

On average, over all experiment settings we see that the throughput increases by 33.4 MCUPS when backtracking is considered, compared to the alignment scores only counterpart. The improvement ratio of `GPU-S-B` decreases by 0.4 on average, when compared to `GPU-S-A`. The improvement ratio of `GPU-S-H` decreases by 4.3 on average, when compared to `GPU-S-B`.

5.6.2 Multiple Text Batching Results

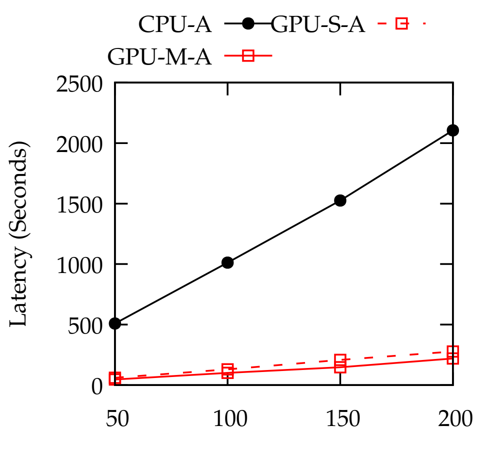
We now investigate results achieved by `GPUGapsMis` using the multiple text batching approach.

Alignment Scores Only

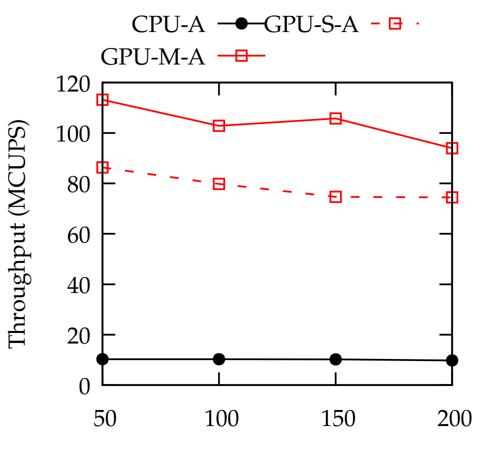
As shown in Figures 5.9 and 5.10, there are similar trends in latency, throughput and improvement ratio exhibited by `GPU-M-A` to those exhibited by `GPU-S-A` discussed in Section 5.6.1.

By examining Figures 5.9b and 5.10b closer, we see that `GPU-M-A` achieves greater throughput than `GPU-S-A`. This is because `GPU-M-A` requires less host device communication than `GPU-S-A`. In Section 5.6.1 `GPU-S-H` was negatively affected by increased host device data transfer and therefore exhibited lower sensitivity to increasing pattern length with fixed number of sequence pairs, being shown as a flatter and lower trend in throughput and improvement ratio when compared to `GPU-S-B`. This is a similar scenario, as `GPU-S-A` has a greater host device data transfer requirement than `GPU-M-A`. This is amplified by the lower number of host device synchronisations required by `GPU-M-A` compared to `GPU-S-A`.

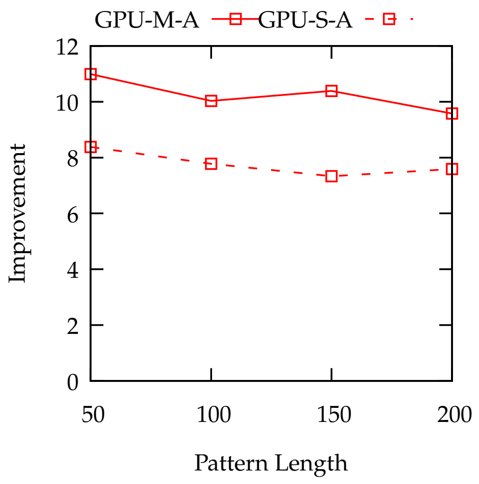
Discussion of ATGPU Analysis In Section 5.5, we used the ATGPU model to analyse the `GPU-S-A` and `GPU-M-A` approaches, by extending the existing AGPU analysis and comparing the resulting cost functions, which take into account the data transfer requirements of each approach. Our analysis showed that both approaches had the same amount of computational work and the same amount of global memory accesses, yet they differed in the amount of synchronisation and data transfer, with `GPU-S-A` having up to a factor of q more data transfer and synchronisation to complete. Our analysis predicted that the `GPU-M-A` approach would therefore outperform the `GPU-S-A` approach. By examining Figures 5.9 and 5.10, it can bee see that our hypothesis is confirmed. This shows us that the ATGPU can show difference between two GPU approaches that require the same amount of computation, but different amounts of data transfer, and that it is able to help in the decision of which solution should be implemented, giving a fuller picture of the execution when compared to the other existing abstract models.



(a) Latency



(b) Throughput



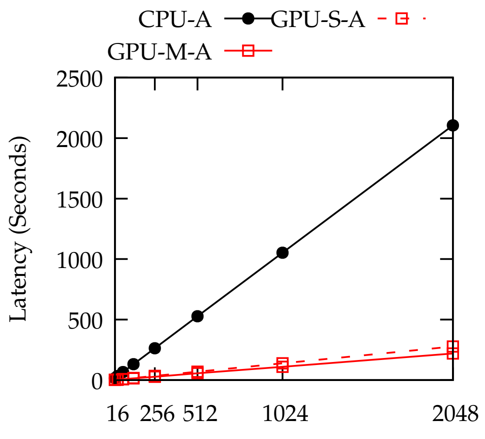
(c) Improvement

Figure 5.9: Result for `GPU-M-A`, performing alignment scores phase only with multiple text batching, for input sets containing 204800 sequence pairs.

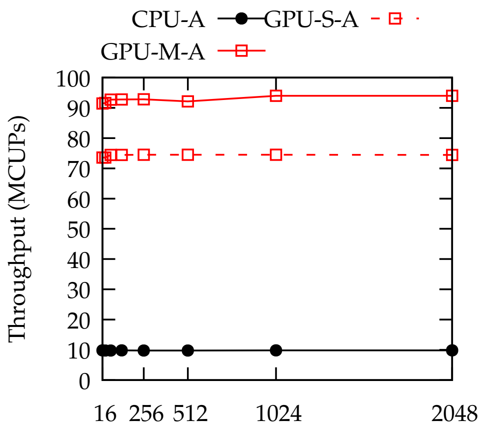
Alignment Scores with Backtracking

We see in Figures 5.11 and 5.12 that `GPU-M-B` and `GPU-M-H` exhibit trends similar to their respective single text batching counterparts, `GPU-S-B` and `GPU-S-H`.

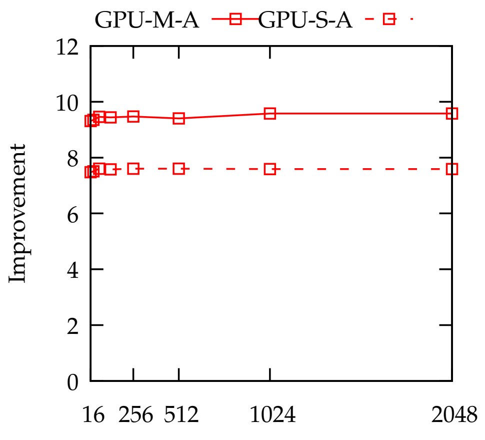
Similar to Section 5.6.2, the multi text batching `GPU-M-B` and `GPU-M-H` perform consistently better than the single text counterpart `GPU-S-B` and `GPU-S-H`, respectively. This is because each of the multi text approaches require less host device communica-



(a) Latency



(b) Throughput

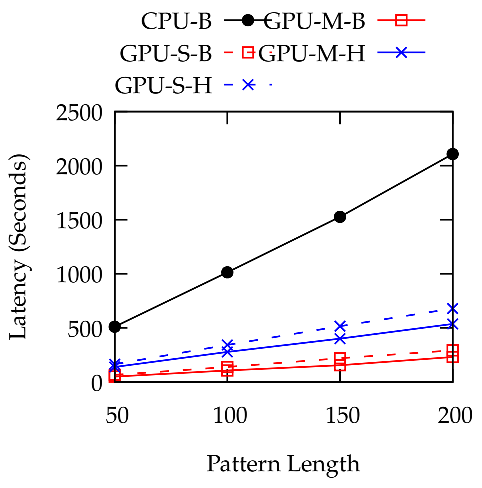


(c) Improvement

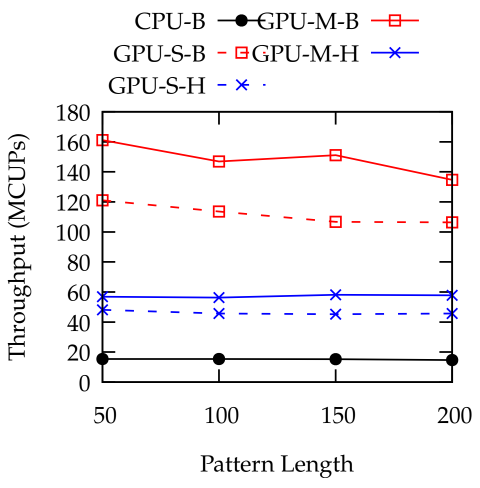
Figure 5.10: Result for `GPU-M-A`, performing alignment scores phase only, with multiple text batching, for input sets containing patterns of length 200.

tion and data transfer than their single text counterpart. As previously explained, the data transfer between host and device is very expensive and can be detrimental to the performance, therefore reducing the amount of this type of data transfer as much as possible would benefit the improvement ratio against the CPU, as has been demonstrated here.

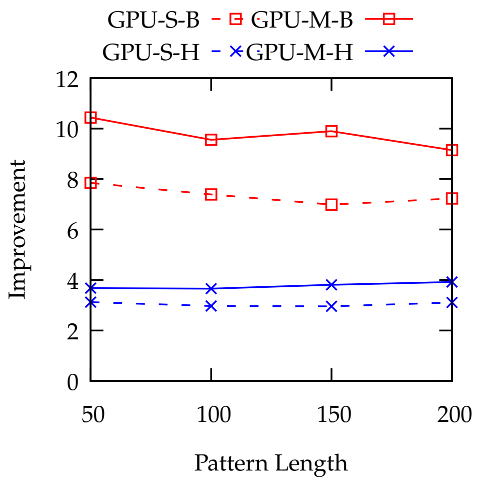
An interesting result is the throughput and improvement ratio of `GPU-M-H`, which monotonically increases as pattern length is increased, as shown in Figures 5.11b and 5.11c.



(a) Latency



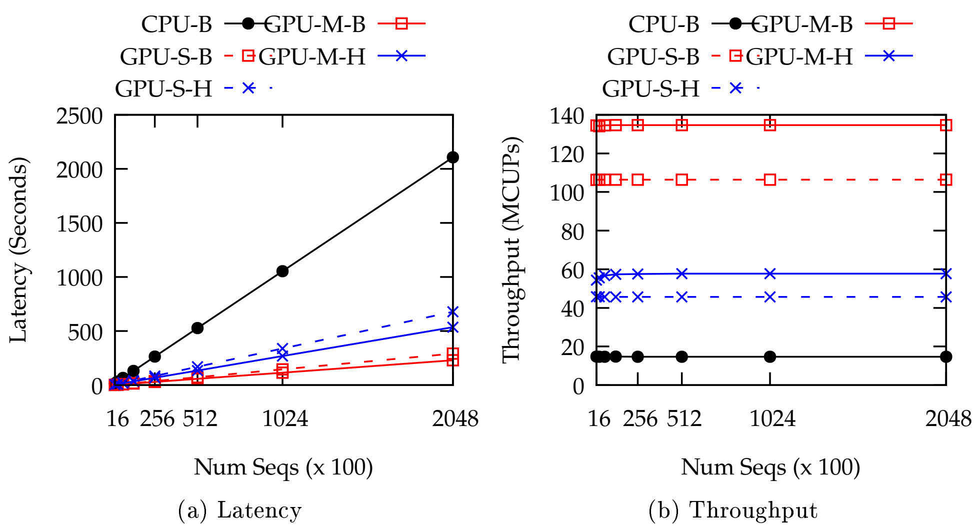
(b) Throughput



(c) Improvement

Figure 5.11: Result for `GPU-M-B` and `GPU-M-H`, with multiple text batching, for input sets containing 204800 sequence pairs.

This is the only GPU approach to exhibit such a characteristic. `GPU-M` can schedule at most qr threadblocks on the GPU in a single batch, whereas `GPU-S` is more limited and can only schedule up to r threadblocks in a single batch. Therefore when H matrices are returned asynchronously to the host upon termination of the kernel, there are more threadblocks ready for execution in `GPU-M-H` than `GPU-S-H`, meaning `GPU-S-H` is not able to hide the latency of asynchronous data transfer as effectively as `GPU-M-H`.



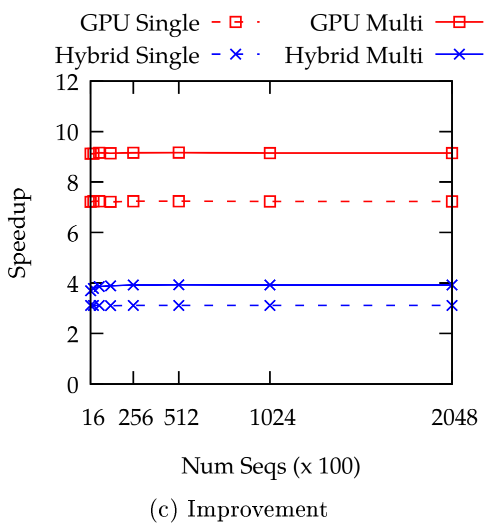


Figure 5.12: Result for `GPU-M-B` and `GPU-M-H`, with multiple text batching, for input sets containing patterns of length 200.

Summary

In summary, taking into account of all experimental results presented in the supplementary material, we see that the peak performance of `GPU-M-A` and `GPU-M-B` occur in the same setting; when pattern length is 50, for 204800 sequence pairs. `GPU-M-A` is on average 10.1 times faster than `CPU-A` and increases the improvement ratio on average by 2.3 compared to `GPU-S-A`. The peak improvement ratio is 11 times, when the pattern length is 50 and number of sequence pairs is 204800. Note that the throughput achieved in this setting is 113.2 MCUPS. On the other hand, when backtracking is computed, the peak throughput is increased to 161 MCUPS, though the improvement ratio is 10.4 times which is lower than the 11 times without backtracking. As with single text batching, this higher throughput but lower improvement ratio is due to less work required to calculate the additional cells for backtracking, and the sequential backtracking algorithm being inefficient on the GPU.

The improvement ratio of `GPU-M-H` decreases by 6.1 on average, when compared to `GPU-M-B`. On average, `GPU-M-H` causes an increase in improvement ratio by 0.6 and an increase in throughput by 9.8 MCUPS when compared to `GPU-S-H`.

We see that `GPU-M-B` increases throughput yet lowers the improvement ratio achieved, when compared to `GPU-M-A`. Throughput of `GPU-M-B` increases on average by 45.8 MCUPS compared to `GPU-M-A`, and the improvement ratio decreases by 0.3 on average. `GPU-M-H` achieved higher throughput and higher improvement ratio than `GPU-S-H`, yet does not outperform `GPU-B`.

5.6.3 Improvement on Different GPU Devices

By running `GPUGapsMis` on GPUs with more resources, it is expected that a higher level of improvement against `CPUGapsMis` would be achieved, however some parallel algorithms are not able to take advantage of extra resources past a certain point, due to excessive communication overhead. We wish to investigate whether a GPU with more resources is negatively affected in performance gained, when compared to a lower specification GPU, due to finite global memory access bandwidth and costly access latency. The increased number of alignment tasks (threadblocks) running concurrently on the GPU could create a communication bottleneck when serving global memory requests.

We test this by investigating how results of `GPUGapsMis` on GTX680 (already discussed) compare to results on GTX650. GTX650 and GTX680 have 2 and 8 SMs, clock speed of 1.2GHz and 1 GHz, and global memory of 1GB and 2GB, respectively. GTX680 has more Streaming Multiprocessors than GTX650, so it can run more alignment tasks concurrently than GTX650. Therefore we expect GTX680 to outperform GTX650 when running `GPUGapsMis`. Assuming that all data fits on the GPU memory, we must decide how much we expect GTX680 to outperform GTX650. GTX680 has 4 times the resources of GTX650, but a clock speed that is only 83% of GTX650. Therefore we can estimate that GTX680 will be around 3.3 times faster than GTX650. The global memory bandwidth of GTX680 is only 2.4 times of GTX650, so there is potential for some applications to encounter a bottleneck in global memory access on GTX680, yet not GTX650.

We run the best performing approach of `GPUGapsMis`, `GPU-M-B` on GTX650. If `GPUGapsMis` has 3.3 or greater improvement on GTX680, compared to GTX650, then we should expect that running `GPUGapsMis` on a Kepler GPU with specifications higher than GTX680 would yield greater improvement still. The results obtained achieved are summarised in Table 5.2.

Table 5.2: Comparison of resources for GTX650 and GTX680 and associated performance of `GPU-M-B`.

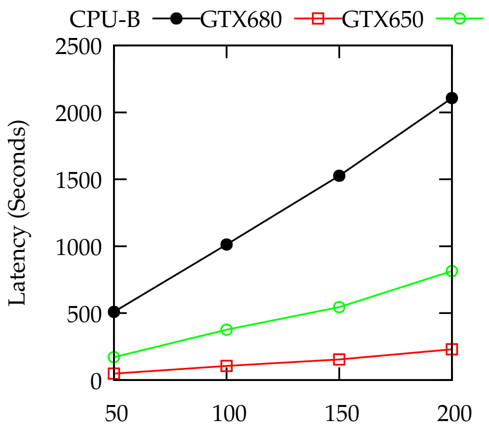
GPU	GTX650	GTX680
Num SM	2	8
Clock Speed	1.2GHz	1GHz
Resource Ratio	1	4
Expected Improvement	1	3.3
Observed Improvement	1	3.5

GTX680 outperforms GTX650 in all cases, by a ratio of 3.5 times. This ratio remains constant throughout increase in pattern length and throughout increase in number of sequence pairs. Figures 5.13 and 5.14 demonstrate that the performance of `GPU-M-B` exhibits similar trends on GTX650 as on GTX680, and show the ratio of improvement between the two GPUs unaffected by input data size.

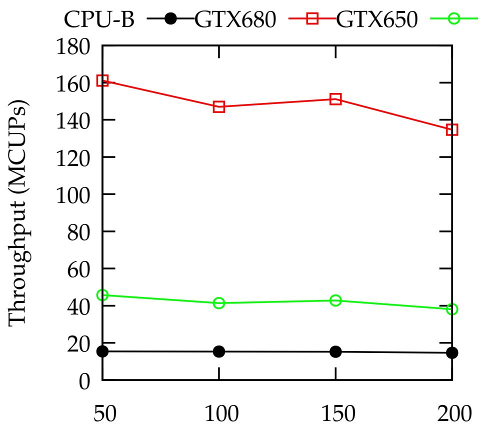
We are able to conclude that `GPUGapsMis` adapts to a GPU of different specification well, and that any communication overhead is not exaggerated by a disproportionate amount, as resources available are increased. Therefore, we are able to have confidence that proportionally better speed up would be possible, should higher specification GPUs be used to run `GPUGapsMis`.

5.7 Conclusion

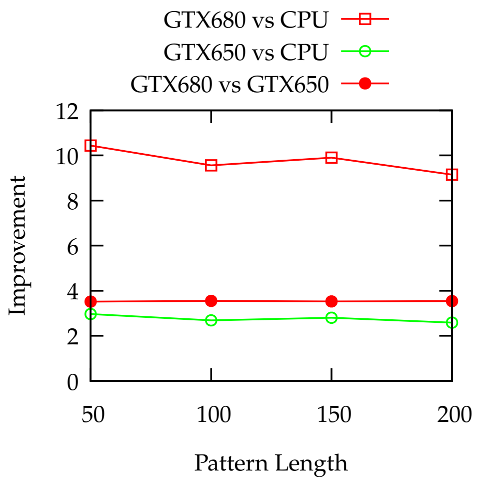
In this chapter, we presented a study on a GPU-based algorithm to solve the pairwise semi-global sequence alignment with bounded number of gaps problem, using a data-parallel approach. We analysed our algorithm `GPUGapsMis` on both the AGPU and ATGPU models, with theoretical analysis confirmed by observed results. We achieved greater speed up compared to a previous data-parallel approach. On our system, we achieved peak speed up against the CPU of 11 times when only the alignment scores



(a) Latency



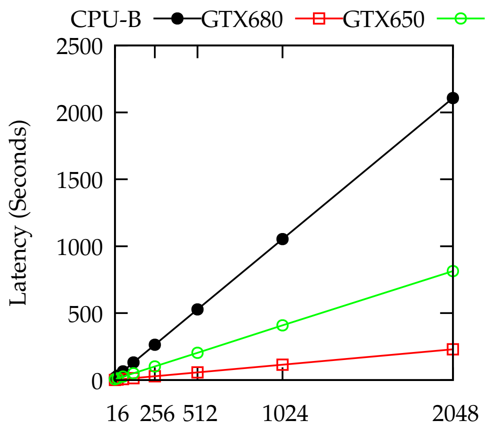
(b) Throughput



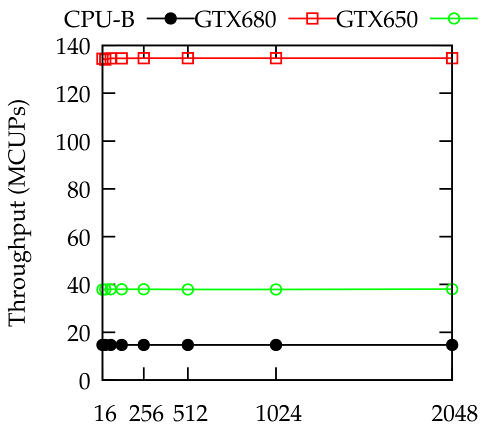
(c) Improvement

Figure 5.13: Result for `GPU-M-B`, with multiple text batching, running on GTX 650 for input sets containing 204800 sequence pairs.

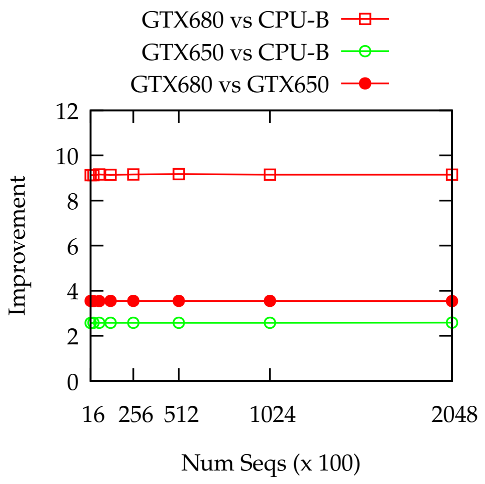
were computed, and 10.4 times speed up when the backtracking was also computed. We achieved greater levels of speed up compared to a previous existing data-parallel approach [68] on a similar system. We successfully used the ATGPU model (introduced in Chapter 4) to accurately predict the best performing GPU approach, which was not possible using existing abstract GPU models. This is because the different GPU approaches had the same amount of computational work, but differing amounts of data transfer. We showed that the best performance was achieved by `GPU-M-B`, with



(a) Latency



(b) Throughput



(c) Improvement vs CPU

Figure 5.14: Result for `GPU-M-B`, with multiple text batching running on GTX 650 for input sets containing patterns of length 200.

multi text batching and backtracking computed on the GPU. Of all approaches we considered, `GPU-M-B` required the least host device communication, and we showed that the performance scaled well on a GPU of better specification. We note that there will exist a point where an increase in resources (i.e. number of cores, amount of global memory) on the GPU will fail to yield ever increasing improvement (using the same input data). This is because the maximum amount of parallelism would have been achieved for that particular input data set.

Chapter 6

Protein Spectral Identification on GPU

6.1 Introduction

The characterisation of proteins or peptides is often carried out using mass spectrometry, a process which fractures a sample along various cleavage points (generally along the peptide bonds). These segments of the sample are then measured by means of their molecular weight and the intensity of which they appear in the sample. The data is then mapped out as a *spectrum*, which shows the information relating to the molecular weight of the constituent parts. By analysing the data points on the spectrum, it is then possible to infer the identity of the sample protein, by means of its constituent peptides. However, modification in the form of PTM can make unexpected changes to the spectrum, meaning that a simple comparison of experimental values with known values is not sufficient, so there are a range of spectral identification algorithms which look to match modified experimental sample spectra with a known theoretical spectrum in a database. There has been relatively little work into accelerating this process using the GPU.

In this chapter, we investigate using the GPU to accelerate and solve the Match Score Identification problem (introduced in Chapter 3), which computes similarity between a database of theoretical known protein or peptide spectra, and a set of experimental modified spectra. This particular algorithm has been shown to perform well against existing tools, maintaining accuracy levels and speeding up the identification of protein spectra. We propose the algorithm `GPU-MSI`, which solves the match spectral identification on the GPU. We provide a theoretical analysis of `GPU-MSI` using the ATGPU

Table 6.1: Comparison of sequential MSI Algorithms for single experimental spectrum S, and same length theoretical spectra T

Algorithm	Time
List Match	$O(qrn)$
Vector Match	$O(qrm)$
Index Match (Worst Case)	$O(qrn)$

model, comparing with sequential solutions and showing that it is optimal. We then verify performance of `GPU-MSI` using experiments, confirming the hypotheses that were generated by the analysis on the ATGPU model, and showing that `GPU-MSI` achieves promising level of speed up, with up to 22× speed up on our system, compared to the best performing CPU implementation.

Organisation of Chapter

The remainder of the chapter is organised as follows: We first analyse the existing sequential approaches to the Match Spectral Identification problem in Section 6.2. In Section 6.3, we consider how each of the sequential approaches from Section 6.2 will run on the GPU, before proposing the algorithm `GPU-MSI` and giving analysis on the ATGPU model. Section 6.5 details the setting of the experiments used and Section 6.6 presents and discusses experimental results to verify the performance of `GPU-MSI`. Finally, Section 6.7 concludes the chapter.

6.2 Sequential Approaches

In order to solve Problem 1, there exists several sequential algorithms, which we now analyse. Algorithms 6, 8, and 10 were previously analysed in [53], yet we also give pseudocode and analysis here. A summary of the analyses of the different algorithms is given in Table 6.1.

List Match Algorithm The list match algorithm (Algorithm 6) calculates the match score between all experimental spectra in $\mathcal{T}$ and all library spectra in $\mathcal{X}$, all stored as ordered mass-lists.

For computing the match score for q experimental spectra of n masses, and r target spectra, each of $m \leq n$ masses, the simple list match algorithm requires $O(qrn)$ time.

Algorithm 6 List Match

Input: Experimental spectra $\mathcal{T} = T_0, ..., T_{q-1}$
Theoretical spectra $\mathcal{X} = X_0, ..., X_{r-1}$ as mass-lists.
Output: Spectrum $X_\alpha \in \mathcal{X}$ with the highest match score with each $T_i \in \mathcal{T}$.

```
for $i = 0 \rightarrow q-1$ do
    for $j = 0 \rightarrow r-1$ do
        $\delta = T_i[n] - X_j[m]$
        $x = 1$
        $y = 1$
        while $x < n$ AND $y < m$ do
            if $T_i[x] == X_j[y]$ then
                $score = score + 1$
                $x = x + 1$
                $y = y + 1$
            else if $T_i[x] < X_j[y]$ then
                $x = x + 1$
            else
                $y = y + 1$
        $x = 1$
        $y = 1$
        while $x < n$ AND $y < m$ do
            if $T_i[x] == X_j[y] + \delta$ then
                $score = score + 1$
                $x = x + 1$
                $y = y + 1$
            else if $T_i[x] < X_j[y] + \delta$ then
                $x = x + 1$
            else
                $y = y + 1$
```

Vector Match Algorithm The vector match algorithm calculates the match score between experimental spectra $\mathcal{T}$ stored as 0-1 vector and all spectra in $\mathcal{X}$, stored as mass lists. The vector representation $\mathcal{T}'$ of $\mathcal{T}$ is generated as per Algorithm 7, with Algorithm 8 performing the calculation.

Algorithm 7 Generate Vector representation of $\mathcal{T}$

```
Input: Query spectra $\mathcal{T} = T_0, T_1, ..., T_{q-1}$ Output: $T_i' = T_i \in \mathcal{T}$ as 0-1 vector.
  for $i = 0 \to q-1$ do
      Create $T_i'[0, ..., T_i[n] - 1 = 0$
      for $j = 0 \to n-1$ do
          $T_i'[T_i[j] - 1] = 1$
```

Algorithm 8 Vector Match

```
Input: Query spectra $\mathcal{T}' = T_1', ..., T_q'$
Target spectra $\mathcal{X} = X_1, ..., X_r$
Output: Spectrum $X_\alpha \in \mathcal{X}$ with the highest match score with each $T \in \mathcal{T}$.
1: for $i = 1 \to q$ do
2:     $max = 0$
3:     $\alpha = 0$
4:     for $j = 1 \to r$ do
5:         $\delta = T_i[n] - X_j[m]$
6:         for $k = 1 \to m$ do
7:             if $T_i'[X_j[k]] == 1$ then
8:                 $score++$
9:             if $T_i'[X_j[k] + \delta] == 1$ then
10:                $score = score + 1$
11:        if $score > max$ then
12:            $max = score$
13:            $\alpha = j$
14:        Record $\alpha$
```

We consider computing the match score for q experimental modified spectra of n masses, and r target spectra, each of $m \leq n$ masses. For generating the vector, $n-1$ accesses are required and $N + n - 2$ updates, with no comparisons. For generating the vector, $O(qn)$ time is required, and for computing the score using Algorithm 8, $O(qrm)$ time is required, which dominates the running of the two algorithms. This is because the number of target spectra in the database r would *typically* be much larger (an order of magnitude larger) than the lengths of the spectra (n or m).

6.2.1 Index Match Algorithm

The main contribution of [53] was the index match algorithm, which calculates the match score for theoretical spectrum X against query spectra $\mathcal{T}$, using linked lists. The linked lists index T and $\mathcal{X}$ by the columns of their 0-1 vector representation, allowing traversal of each linked list to quickly establish which spectra in $\mathcal{X}$ contain a mass peak at a particular column.

To generate the linked lists, each spectrum in $\mathcal{X}$ is traversed, with nodes being added to the linked list corresponding to the mass value of each peak. This is shown in more detail in Algorithm 9. In order to calculate $C(T, X_*(\delta))$, the linked lists are re-calculated with the relevant shifts inserted, and then traversed at the same time as the non-shifted lists. We discuss this in more detail in Section 6.3, as the algorithm `GPU-MSI` requires a shifted representation of the library to be calculated.

Algorithm 9 Generate Linked List representation of $\mathcal{X}$

Input: $\mathcal{X}'$

Output: The set of linked lists $\mathcal{L} = L_1, ..., L_M$ where M is the maximum precursor mass of $\mathcal{X}$.

```
1: for $i = 1 \to M$ do
2:     Initialise linked list $L_i$
3: for $i = 1 \to r$ do
4:     for $j = 1 \to m$ do
5:         Add new node "$i$" to $L_{X_i[j]}$
```

The generation of the linked lists require $O(rm)$ time. The match score is then calculated for $(\mathcal{T}, \mathcal{X})$ using the index match algorithm, as shown in Algorithm 10.

The maximum size of each linked list is r, as there are r target spectra in $\mathcal{X}$; however, the average size of each linked list is $\frac{r(m-1)}{M}$ as there are M linked lists in total and $r(n-1) + M$ nodes in total created (each linked list has a null node). The only linked lists that are traversed are those $qn - 1$ lists which correspond to masses in $\mathcal{T}$, therefore algorithm 10 requires $O(qnr)$ time in the worst case, yet linked list traversal would be reduced to $O(\frac{r(m-1)}{M})$ in the average case.

It was shown in [53] that the index match algorithm was, in practice, faster than the vector match algorithm when the generated linked lists were reused for scoring many experimental spectra against a library.

Algorithm 10 Index Match

```
Input: Query spectra $\mathcal{T} = T_1, ..., T_q$ as mass list.
Linked lists $\mathcal{L} = L_1, ..., L_M$ from $\mathcal{X}$
Output: Spectrum $X_\alpha \in \mathcal{X}$ with the highest scoring match score with each $T \in \mathcal{T}$.
1:  for $i = 1 \to q$ do
2:      for $j = 1 \to n$ do
3:          for Each non-null node $l_k$ in $L_{T_i[j]}$ do
4:              $score[l_k.label] = score[l_k.label] + 1$
5:      $\alpha = 0$
6:      $max = 0$
7:      for $j = 1 \to r$ do
8:          if $score[j] > max$ then
9:              $\alpha = i$
10:             $max = score[j]$
11:     Report $\alpha$
```

6.3 Match Score Identification Problem on The GPU

We now study how the *match spectral identification* problem can be solved on the GPU. We discuss how each of the sequential solutions presented in Section 6.2 would execute on the GPU; we see that the list match algorithm and the index match algorithm is not well suited to execution on the GPU. We adapt the vector match algorithms for the GPU, and propose the new algorithm `GPU-MSI`, giving analysis on the Abstract Transferring GPU model [12] (see Chapter 4) and verifying performance with experimental results.

6.3.1 List Match Algorithm on GPU

For a particular thread block i, have each constituent thread j calculate the match score for the spectrum pair $(T_i, X_j) \in \mathcal{T} \times \mathcal{X}$. The spectral data would be held in shared memory. X_j would reside completely in bank j, meaning bank-conflict free access to X_j, Each thread would keep its own score value in shared memory, thus avoiding bank conflicts encountered when accessing the score value.

As in the sequential version a loop of $(n + m)$ operations would be required for each thread, with each thread accessing the experimental spectrum in a non-pattered manner. Figure 6.1 demonstrates how the non-patterned access to T_i could potentially cause bank conflicts. Up to a b-way bank conflict could occur at each access operation to T_i, when $n \geq b^2$, and up to a $\lceil \frac{n}{b} \rceil$-way bank conflict when n is smaller.

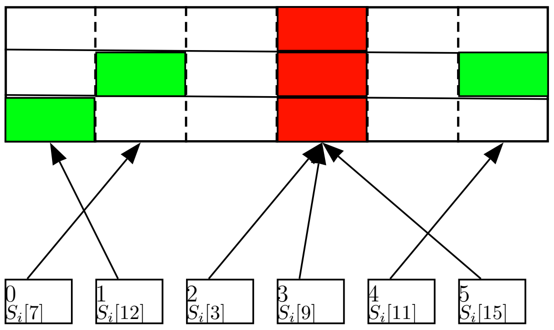


Figure 6.1: Non-patterned access to T_i can cause bank conflicts in shared memory. $b = 5, n_i = 18$. A 3-way bank conflict (red) is shown.

6.3.2 Index Match on GPU

Linked lists are not always best suited to the GPU as they can not be accessed randomly, can inhibit techniques used for addressing the data structure, and require more space than a regular array of structs. Therefore, we model the linked lists of the index match algorithm as arrays of integers; the integer value i in array j represents a node of label i in linked list j. Arrays would be of size r, with dummy data (integer values set to -1) occupying the remaining elements of array j.

In a data parallel implementation, the cores would take each array of r and process elements in $\frac{r}{b}$ steps. An uneven workload between the cores of the multiprocessor would occur, as the column position being considered is not guaranteed to have a multiple of b library sequences to be updated. In the case where $x < b$ elements are present, this means that some cores would lie idle, therefore under utilising resources. Should the elements represent $c, c+b, c+(2b)$... (where c is a constant, and b is the number of cores) then bank conflicts would occur and the operations would serialise. Without padding to the arrays, there would be non-coalesced global memory access, which could create a bottleneck in the program; both of these memory access constraints could remove the time-saving incentive of using the index match algorithm, and adding padding to the arrays could then remove the space-saving incentive of using the index match algorithm.

6.4 Vector Match on GPU

We now introduce the `GPU-MSI` algorithm, which solves the *Match Score Identification problem* introduced in Chapter 2. To our knowledge, this is a novel GPU algorithm which calculates the match score for a set of query spectra $\mathcal{T} = T_1, ..., T_q$ and target spectra $\mathcal{X} = X_1, ..., X_r$.

Table 6.2: Example of vector match calculation against experimental spectrum $T = \{2, 6, 8, 12\}$, where the library spectra have different precursor masses. $X_1 = \{2, 6, 8, 10\}, X_2 = \{2, 3, 5, 9\}, X_3 = \{6, 9, 11\}, X_4 = \{2, 3, 5, 8\}, X_5 = \{3, 6, 7\}$. Traversing column 7 (shown in grey) allows all matched mass points at position 8 to be found in the single column, yet the differing δ values means that the same can not be achieved for finding matched points against $X_*(\delta)$ (cells shown in black), requiring a different column for each library spectrum. $\delta_1 = 2, \delta_2 = 3, \delta_3 = 1, \delta_4 = 4, \delta_5 = 5,$.

Col	0	1	2	3	4	5	6	7	8	9	10
X_1	0	1	0	0	0	1	0	1	0	0	x
X_2	0	1	1	0	1	0	0	0	0	x	x
X_3	0	0	0	0	0	1	0	0	1	0	0
X_4	0	1	1	0	1	0	0	0	x	x	x
X_5	0	0	1	0	0	1	0	x	x	x	x

Table 6.3: By right-aligning the spectral library vector representation to the precursor mass of the experimental spectrum T, all δ-shifted matched pairs for $t_3 = 8$ are located in column $8 - (12 - 11) - 1 = 6$ (black). All non-shifted matched pairs are shown in grey (T_5 is out of range).

Col	0	1	2	3	4	5	6	7	8	9	10
X_1	x	0	1	0	0	0	1	0	1	0	0
X_2	x	x	0	1	1	0	1	0	0	0	0
X_3	0	0	0	0	0	1	0	0	1	0	0
X_4	x	x	x	0	1	1	0	1	0	0	0
X_5	x	x	x	x	0	0	1	0	0	1	0

We give explanation of the algorithm for calculation only one query spectra T_i against $\mathcal{X}$, as the calculation for a whole set of query spectra is very similar. The input data is $\mathcal{T}, \mathcal{X}', \mathcal{X}'_\delta$ (i.e. a 0-1 vector representation of the theoretical library). The GPU algorithm runs with b cores per multiprocessor. Each multiprocessor mp_i will calculate the match score between T_i and $\mathcal{X}$.

6.4.1 Calculating and Aligning the Shifted Vector representation

Table 6.2 demonstrates the access pattern to the cells of the vector representation of $\mathcal{X}$ that would be required, when calculating $C(T_*, \mathcal{X})$, in a situation where the spectra in $\mathcal{X}$ have different masses (this would be almost all cases of input data). It shows that the shifted value that is required for the score is resident in different columns of the vector representation. This presents a problem when we want to use the GPU, as all threads must perform the same action, and correctly patterned data access is required for both global and shared memory access. The aligned data in a single column (that is for the non-shifted value) will work well for the GPU, as it all resides in contiguous cells. We are therefore required to calculate a second vector representation of $\mathcal{X}$, which aligns all spectra values to the *right*; this then places all shifted values in the same column, as is demonstrated in Table 6.3. To use this with an entire set of experimental spectra, we align this shifted vector representation to the maximum mass value found in $\mathcal{T}$. This technique can also be applied to the index match algorithm (algorithm 10) to calculate the linked lists for the shifted spectra.

6.4.2 Reducing Space Requirement on the GPU

With the current data storage scheme used in [52], r elements are required for not only each peak value in the spectrum, but for all values between 0 and W (where W is the maximum mass of the experimental spectra). This therefore requires $O(Wr)$ space. This is equivalent to $O(nD(a+\Delta)r)$ space, where r is the number of library spectra, n is the size of the experimental spectra, D is the discretisation factor, a is the maximum mass of the amino acids used in the experimental spectra, and Δ is the maximum weight which a PTM adds to the affected spectrum.

W can become very large when we consider long spectra, large PTM values, and large discretisation factors; a spectrum could contain anything from 50 peaks to thousands of peaks, depending on the sample; PTM values can also be large, with the *N-terminal palmitoylation* PTM adding 238 Daltons to the mass of the spectral fragments; the discretisation factor can range from 1 in the case of *ion trap spectra* experiments, to 100 for *ion cyclotron resonance spectra* experiments, and no doubt greater, depending on the resolution on the spectra and the quality of the equipment used. This means that the library representation can become very sparse and very large.

When accessing the data, the total of accessed values is $O(qnr)$, we see that the amount of unaccessed (therefore useless) data is $O(D(a+\Delta))$.

From our work on the ATGPU model in Chapter 4, and work concerning amounts of data transfer to and from the GPU in Chapter 5, we know that we should always look to minimise the number of transfers and the amount of transfers. We also know that GPU memory is much more limited than memory available to the CPU, so we should be motivated to use this as efficiently as possible. At the moment, the amount of unaccessed data that is transferred to the GPU is something we can look to improve.

We now present a data storage scheme that reduces the storage requirement and improves on the ratio of accesses to stored words. We serialise the mass peak positions

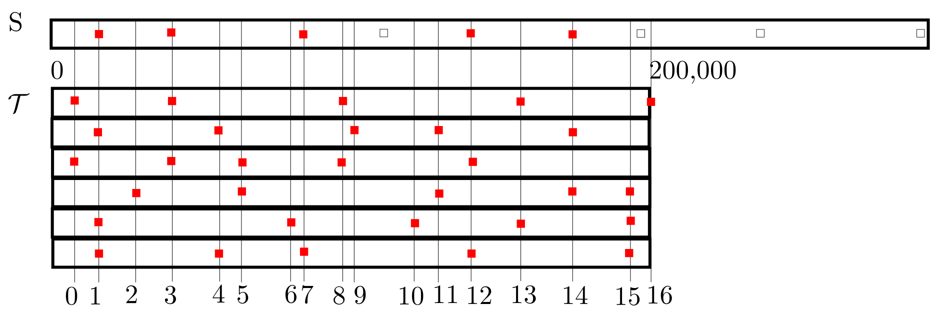


Figure 6.2: The space saving scheme only considers columns which match with mass peaks present in the library.

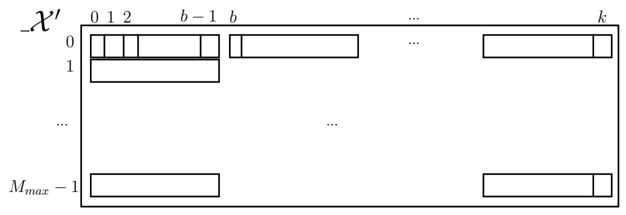
in the theoretical library, such that each subsequent unique peak position has the subsequent serial number. For example, consider the mass peaks in the set $\mathcal{T}' = \{T_1 \cup T_2\} = \{120, 146, 160, 200, 250, 300, 343, 411\}$. When we serialise these positions, any instance of 120, in either the set of experimental spectra or the library, would be mapped to 0; any instance of 146 would be mapped to 1, and so on. Both the experimental spectra and the library of theoretical spectra (and the shifted copies) are then mapped to these values, which excludes mass peaks within the library that will never be matched with the experimental spectra. The vector representation and index representation are then created as earlier using the new mapping. This new mapping requires r elements per mass peak, of which there are now $O(qn)$ unique mass peak columns being stored. We note that $O(qnr) \leq O(Mk)$ meaning the space required has now been lowered. We also note that the amount of access now matches the size of the library, meaning we have increased the ratio of stored words to accessed words, and use the memory more efficiently.

Figure 6.2 demonstrates how only columns corresponding with a mass peak in the library are taken for the space saving solution, and how only these positions are considered when calculating the match score against an experimental spectrum, lowering the amount of column fetches needed and removing the need to check for empty columns.

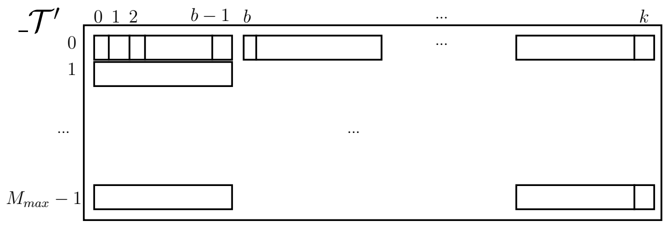
Idea of Parallelisation

Global Memory $\mathcal{T}$ is held in global memory, such that T_i is stored in $\lceil \frac{n}{b} \rceil$ consecutive blocks. $\mathcal{X}'$ (resp. $\mathcal{X}'(\delta)$) is stored in global memory, such that $X'_j[T_i], ..., X'_{j+b-1}[T_i]$ (resp. $\mathcal{X}'(\delta)_j[T_i], ..., \mathcal{X}'(\delta)_{j+p-1}[T_i]$) is stored in the same block. $Score$ is stored such that $Score[i][j]$ being the score between T_i and X_j is in the same block as $Score[i][j+$

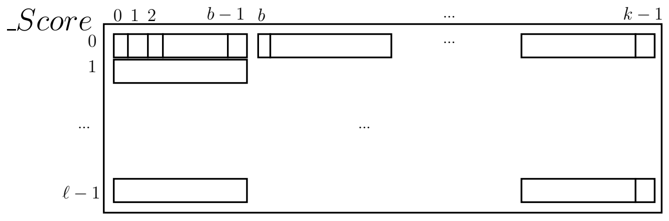
$b-1]$. Global memory storage is demonstrated in Figure 6.3



(a) $\mathcal{X}'$ (identical for $\mathcal{X}'(\delta)$) in global memory. Each row contains the vector for all spectra at a particular position. Each column keeps the vector for a particular spectrum.



(b) $\_\mathcal{T}$ in global memory. Each row contains the mass list for T_i.



(c) $\_Score$ in global memory. Each row contains the scores for T_i against $\mathcal{X}$.

Figure 6.3: Global memory of GPU Reverse Vector Match

Shared Memory Shared memory will hold in $\_T[0, ..., b-1]$, b consecutive mass values of T_i. In $\_X[0, ..., b-1]$ will reside a b-long segment of a row of X' or $X'(\delta)$. Score values for library spectra will be held in $\_score[0, ..., b-1]$.

Kernel Threadblock i of b threads will calculate the match score between a query spectrum T_i and all target spectra $\mathcal{X}$.

The kernel will calculate the score data to go into $\_Score[i][j]$ for all $0 \le j < r$, with the main outer loop iterating through the row $\_Score[i][*]$, calculating b values per iteration. During the calculation, score values are held in shared memory and are accessed in a bank-conflict free manner. Upon completion of the iteration, the scores for the b library sequences is complete and then the values are placed into global memory in a single coalesced memory access operation.

During each iteration, the experimental spectrum is fetched from global memory a block at a time, containing b mass values. These mass values are looped over, and each relevant b-sized column segment is pulled from $\mathcal{X}'$ (and for $\mathcal{X}'(\delta)$) into shared memory. The values pulled from $\mathcal{X}'$ represent the presence/absence of a peak in each of the b library sequences at this particular position, determined by the values in T_i. If 0, there is no peak. If 1, there is a peak. Therefore, we simply add this value to the score values, in a bank conflict free manner. The subsequent values of T_i are dealt with in the same fashion, and then the iteration completes. We then proceed to calculate for the next b library sequences. Output data is returned to the host and the kernel is then terminated.

ATGPU Analysis We now give analysis of `GPU-MSI` using the ATGPU model with b cores per multiprocessor and q multiprocessors, and present the pseudocode in Algorithm 11. Note that $\Lleftarrow$ represents a transfer between GPU and CPU memory; $\Leftarrow$ represents a global memory access operation; $\leftarrow$ represents a shared memory access operation.

Transfer Complexity There are in total 5 transfer operations. The $\_\mathcal{T}, \_\mathcal{T}(\delta)$ data structures each require qn words. The $\_\mathcal{X}, \_\mathcal{X}(\delta)$ data structures each require qnr words, and the $\mathcal{s}core$ data structure requires qr words. Therefore, the transfer cost is $\alpha 5 + \beta(2qn + 2qnr + qr)$, which is upper bounded by $O(\alpha + \beta(qnr))$.

Time Complexity The inner-most loop (i.e. lines 13-17) is executed $\lceil \frac{nr}{b^2} \rceil$ times. The inner-most loop itself iterates for b times, meaning the code within the loop executes $\lceil \frac{nr}{b} \rceil$ times. This inner-most loop contains 16 individual operations.

The fetching of the T values (lines 11 and 12)and outputting to the $\_Score$ matrix (line 18) are executed $\lceil \frac{nr}{b^2} \rceil$ times, where there are 10 operations.

The time cost is therefore $16\lceil \frac{nr}{b} \rceil + 10\lceil \frac{nr}{b^2} \rceil$, which is upper bounded by $O(\frac{nr}{b})$.

I/O Complexity Each of the q multiprocessors, in the process of aligning the spectra accesses the T and $\_Score$ data as $3\lceil \frac{nr}{b^2} \rceil$ blocks. Each multiprocessor accesses the X data as $2\lceil \frac{nr}{b} \rceil$ blocks. This gives the amount of global memory blocks accessed as $q(3\lceil \frac{nr}{b^2} \rceil + 2\lceil \frac{nr}{b} \rceil)$, which is upper-bounded by $O(\frac{qnr}{b})$

Algorithm 11 `GPU-MSI` on ATGPU

Input: Normalised mass list representation of experimental spectra $\mathcal{T}, \mathcal{T}(\delta)$ **Input:** Vector representation of Library spectra: $\mathcal{X}', \mathcal{X}'(\delta)$

Output: Score matrix for all spectral pairs

```
 1: //Copy Input data to Device
 2: $\_\mathcal{T} \Lleftarrow \mathcal{T}$
 3: $\_\mathcal{T}(\delta) \Lleftarrow \mathcal{T}(\delta)$
 4: $\_\mathcal{X} \Lleftarrow \mathcal{X}'$
 5: $\_\mathcal{X}(\delta) \Lleftarrow \mathcal{X}'(\delta)$
 6: //Begin GPU Work
 7: for all $mp_i \in MP[0, ..., q-1]$ in parallel do
 8:     for all $c_{i,j} \in C_i$ in parallel do
 9:         for $y = 0 \rightarrow \lceil \frac{r}{b} \rceil$ do
10:             for $x = 0 \rightarrow \lceil \frac{n}{b} \rceil - 1$ do
11:                 $\_t[j] \Leftarrow \_\mathcal{T}[i][yb+j]$
12:                 $\_td[j] \Leftarrow \_\mathcal{T}d[i][yb+j]$
13:                 for $z = 0 \rightarrow b$ do
14:                     $\_x[j] \Leftarrow \_\mathcal{X}'[\_T[z]][yb+j]$
15:                     $\_score[j] \leftarrow \_score[j] + \_X[j]$
16:                     $\_xd[j] \Leftarrow \_\mathcal{X}'(\delta)[\_Td[z]][yb+j]$
17:                     $\_score[j] \leftarrow \_score[j] + \_pSeqd[j]$
18:             $\_Score[i][j] \Leftarrow \_score[j]$
19: //Return answer to Host
20: $Score \Lleftarrow \_Score$
```

Global Memory Used As per the description of the storage scheme in Section 6.4.2, and in a similar vain to the I/O complexity, the $\mathcal{X}$ data requires $2qnr$ words. The $\mathcal{T}$ data requires $2qn$ words, and the *_Score* data structure requires qr words. Therefore, the total global memory used is $2qnr + 2qn + qr$, which is upper-bounded by $O(qnr)$.

Shared Memory used In Algorithm 11, no shared memory data structure requires more than b words, meaning the shared memory used is $O(b)$, and therefore the occupancy on the real GPU should be optimal.

Cost Function We see that there is a single round in this algorithm, so the cost function is upper-bounded as in Expression 6.1, where α is the ATGPU cost to stage a Host Device memory transaction, β is the cost of transferring a single word to the device, b is the number of cores on each multiprocessor of the device, λ is the cost for accessing a block of global memory on the device ,γ is the operation rate (i.e. the number of operations completed per time-cost unit), and σ is the cost for synchronisation operations at the end of a computation round.

$$O\left(\alpha + \beta(qnr) + \frac{\frac{nr}{b} + \lambda\frac{qnr}{b}}{\gamma} + \sigma\right) \tag{6.1}$$

Discussion Because the global memory access is able to coalesce, because there are no bank conflicts, and because the shared memory usage is linear with the number of cores in the multiprocessor, we expect `GPU-MSI` to achieve high occupancy on the GPU, meaning that latency of global memory access should be well hidden by computation tasks, and that the algorithm will use the GPU resources well, yielding good performance.

We now compare `GPU-MSI` in Algorithm 11 with the existing sequential algorithms of the simple vector match algorithm (Algorithm 8) and the index match algorithm (Algorithm 10). The processor time product of `GPU-MSI` is qnr (there are qb processors, multiplied by $\frac{nr}{b}$ time). We therefore see, compared to the time complexity of the simple vector match algorithm (Algorithm 8) and the index match algorithm (Algorithm 10) of $O(qnr)$, that `GPU-MSI` performs no more work than the sequential algorithms, and so is optimal.

When we consider the ATGPU cost function of `GPU-MSI`, we see that the work carried out by the GPU is $O(\frac{nr}{b} + \lambda\frac{qr}{b})$. Generally, as the cost of global memory access is large, we expect that the global memory access will dominate this function, and therefore that the performance of GPU-MSI will be more sensitive to an increase in r, than it would be to an increase in n.

6.5 Experimental Setting

Spectral identification tools are typically used to search databases of known spectra, in order to find the best match for a query spectrum, or set of query spectrum. In order to simulate a database search for the most optimal alignment for a set of query sequences, we align a set of query (pattern) sequences with a set of target (text) sequences.

Let $\mathcal{S} = s_1, s_2, ..., S_L$ be the set of experimental spectra and let $\mathcal{T} = t_1, t_2, ..., t_r$ be the set of theoretical library spectra. We look to simulate searching in a database for the theoretical spectrum which gives the best score for each individual experimental spectrum. For each pair of spectra in $\mathcal{S} \times \mathcal{T}$, we solve the *Match Spectral Identification Problem*, with either GPU-MSI - or - with CPU-MSI-V or CPU-MSI-I - a sequential implementation of the vector match algorithm and the index match algorithm on a single CPU thread.

Input Data. We randomly generate a single spectrum to the length that we specify in the experiments, by picking amino acids at random, until the given amount are picked. We then insert a shift of a random value between 10 and 50, at a random position in the spectrum. We do this many times over, creating a new spectrum each time. We use some of these generated spectra as the theoretical library, and some as the experimental spectra. This ensures that we are using spectra that are reletivley similar to each other for our simulated database search - this matches the use case that would be encountered by biologists. The synthetic data used will give a good view of the performance of GPU-MSI with real sequence data, as all data is treated identically by the algorithm.

For our experiments, we consider different sizes of theoretical library, and different lengths of spectra, measuring the performance of aligning all the spectral pairs. We use a set of $q = 500$ experimental spectra and a database of $r = \{250, 500, 750, 1000\}$ theoretical spectra, of length $m = \{50, 75, 100, 125, 150, 175, 200\}$. We align all $q \times r$ spectral pairs. The spectral data is generated at runtime, though the program can be extended to allow input from disk.

Performance Measurement. To measure the performance of the algorithms, we first process the input data so that it is in the form required for the particular algorithm (vector representation, or index representation). The timer then measures time taken

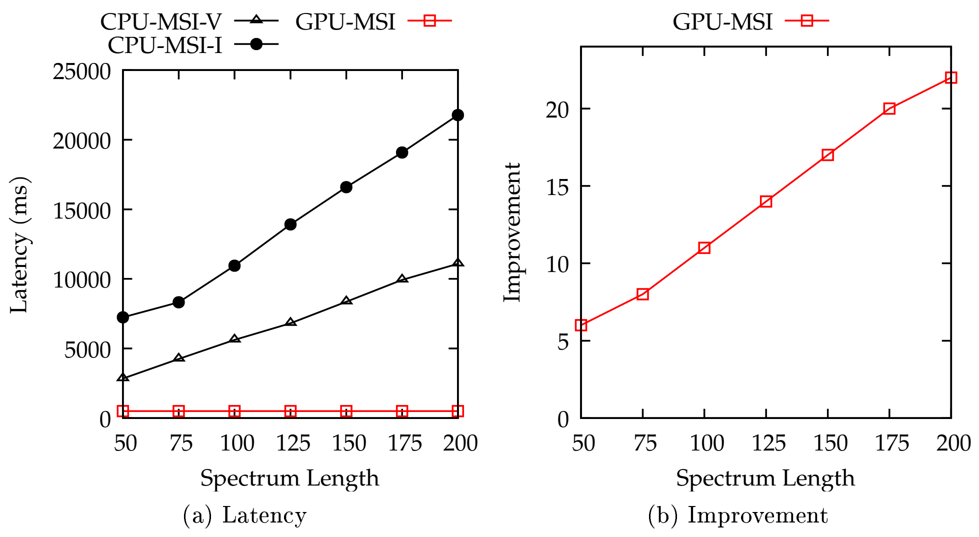


Figure 6.4: Result for `GPU-MSI` for a library of 1000 spectra.

for all GPU operations (such as data transfer and compute) to finish, as well as the entire CPU-based method.

6.6 Results and Discussion

In this section, we present and discuss results from experiments carried out as described in Section 6.5. Following from the ATGPU analysis in Section 6.4, we expect that the performance of `GPU-MSI` will be more sensitive to an increase in r, than it would be to an increase in n.

We look to evaluate the performance change of `GPU-MSI` as the input size increases, and to validate the ATGPU analysis given in Section 6.4. We carry out all experiments described in Section 6.5, with all results presented in Table C.1. In order to look closely at the trends, we focus on two settings: (i) increasing number of library spectra with spectral length fixed at 200 amino acids; and (ii) increasing spectral length with number of library spectra fixed at 1000. Both settings investigate the effect of increasing data size. The results presented here appear in the final rows and final columns of Table C.1, and are representative of all other experiment results obtained.

Figures 6.4 and 6.5 show that the latency of `CPU-MSI-V` `CPU-MSI-I` increase linearly with the increase in size of input data. `GPU-MSI` has considerably smaller latency than all results by the CPU algorithms and therefore outperforms both `CPU-MSI-I` and

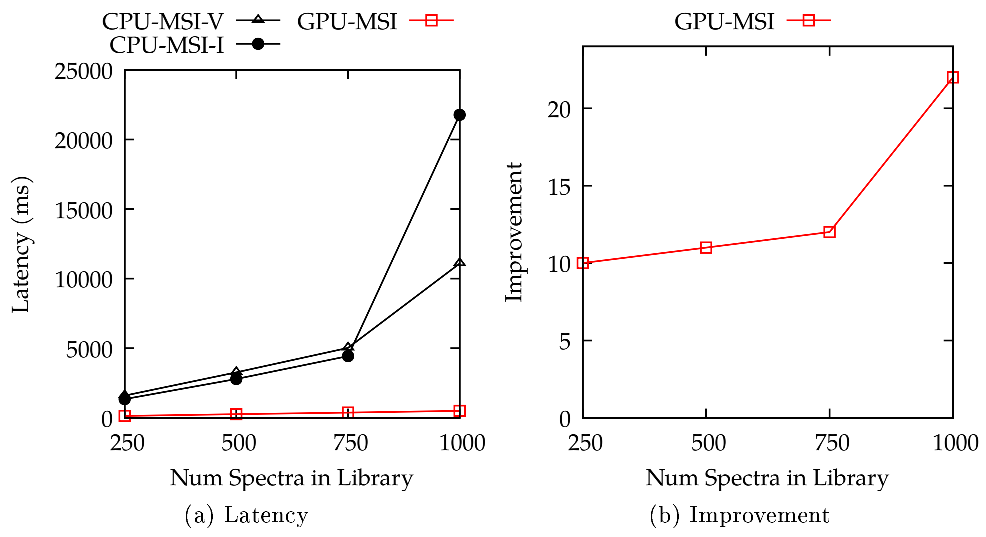


Figure 6.5: Result for `GPU-MSI` for spectra of 200 length.

`CPU-MSI-V` in all cases. Furthermore, we can see in Figure 6.4b that the improvement ratio increases as the pattern length increases. This stands to reason, as Figure 6.4a shows an almost flat trend in latency for `GPU-MSI`, as opposed to the linear growth shown for the CPU counterparts. Figure 6.5 shows that when the number of library spectra is increased, the running time increases linearly. This is reinforced by Figure 6.5b which shows the improvement ratio is much flatter than in Figure 6.4b . This suggests that the performance of `GPU-MSI` is not sensitive to the spectral length, but *is* sensitive to the size of the theoretical database. In comparison, the CPU counterparts are sensitive to both theoretical library size and spectral length. When we compare the trends of the results to the ATGPU analysis in Section 6.4, we see that our hypothesis has been confirmed, and that `GPU-MSI` is indeed more sensitive to an increase in the size of the theoretical library, than to an increase in the length of the spectra to be aligned.

The peak improvement that we achieved with `GPU-MSI` on our system was 22× over the best performing CPU control algorithm, which occurred with spectral length of 200 and 1000 theoretical library spectra.

6.7 Conclusion

In this chapter, we introduced the algorithm `GPU-MSI`, which computes the match score identification problem for a set of experimental protein spectra and a set of known theoretical library spectra. The chapter reviewed and analysed the existing sequential solutions, and then considered how each of the sequential solutions would run on the GPU. We then introduced the algorithm `GPU-MSI`. We gave analysis and developed hypotheses about the performance of the algorithm using the ATGPU model that was introduced in Chapter 4, and showed that the `GPU-MSI` algorithm is optimal, in that it performed no more work than the sequential counterparts. We then discussed the experimental setting and presented the experimental results. We verified the performance of `GPU-MSI` against two existing sequential solutions using experiments. We showed that the hypotheses created using the ATGPU model were confirmed by the experimental results, and that the algorithm achieved promising levels of improvement on our system when compared to the CPU control experiments. `GPU-MSI` achieved a peak improvement of 22$\times$ over the best performing CPU control experiment on our system.

Chapter 7

Conclusion

In this chapter, we look to draw conclusions on the thesis.

In Chapter 1, there were two research questions set out:

1. Are we able to create a parallel abstract model that gives improved and accurate analysis of GPU programs, compared to the analysis given by existing models?
2. Can new GPU-based algorithms improve on existing solutions to bioinformatics tasks within the Sequence Alignment problem and the Protein Spectral Alignment problem?

We now discuss how well these questions have been answered.

In Chapter 2, we reviewed several existing abstract parallel models, both general parallel models, and those that are designed with the GPU specifically in mind. We highlighted that no model existed which considered the data transfer between the CPU and the GPU, before then bringing attention to the arguments in the literature regarding consideration of CPU GPU data transfer: namely, that the costs involved with transferring data from the CPU to the GPU (and vice-verse) should always be considered, as they are expensive operations and can greatly reduce an application's performance level, once these operations are also taken into account.

In Chapter 4, we proposed a new GPU abstract model, which we called ATGPU. The ATGPU differs from existing models in the fact that it is the first abstract GPU model (to our knowledge) which considers CPU GPU data transfer. The ATGPU also combines the capabilities of existing models, making it more comprehensive than the existing models. We demonstrated the use of the ATGPU model in a number of settings:

In Chapter 4 we used our model to analyse a number of computational problems, demonstrating that for some computational problems, the CPUGPU data transfer operations that are captured by the ATGPU model give a more full and accurate analysis than the existing models, and that for some other problems, the model is *as useful* as the existing models. We successfully verified these findings with experiments.

In Chapter 5, we considered the performance of our GPU-based sequence alignment algorithm `GPUGapsMis`, and extended analysis on the existing AGPU model, by the use of ATGPU cost function to model different experimental approaches. By using the ATGPU cost function, we were able to show that the computation carried out by these approaches was the same, but the CPU GPU data transfer differed. This meant that the existing models did not distinguish between the approaches. We demonstrated that the ATGPU cost function successfully distinguished between the different approaches, and that it produced the correct result in identifying the best performing approach.

In Chapter 6, we considered the performance of our GPU-based spectral identification algorithm `GPU-MSI`, and used the analysis of this algorithm on the ATGPU model to generate hypotheses about the performance. These hypotheses were then successfully confirmed using experimental results.

We can therefore conclude that, in proposing the ATGPU model in Chapter 4, we have successfully created a parallel abstract model that gives improved and more accurate analysis of GPU programs, compared to the analysis given by existing models.

In Chapter 3, we introduced two specific bioinformatics problems, namely: DNA Sequence Alignment, and Protein Spectral Identification. We highlighted that there has been a wealth of research in the area of sequence alignment using GPU, and decided to focus on one particular problem: semi-global sequence alignment with a bounded number of gaps. For this particular problem, we have shown that there was existing GPU algorithms for the single gap case, and that for the multiple gap case, there was also an existing GPU algorithm, but we identified scope where it could be improved. We then reviewed research relating to protein spectral identification, and discussed how there is only a small amount of research relating to the use of GPU in solving this problem, which mostly focussed on matrix multiplication operations to speed up particular scoring schemes. We identified a scoring scheme called the *match spectral identification*, which successfully accelerated the identification of protein spectra using linked lists; we identified scope to propose a GPU solution to accelerate this further, as none existed.

In Chapter 5, we proposed the algorithm `GPUGapsMis`, which is a GPU-based algorithm that solved the semiglobal sequence alignment with a bounded number of gaps problem. We studied a number of approaches for using the GPU to solve this problem, investigating differences in batching the sequence alignment jobs, and in computing the optimal alignment on the GPU and the CPU. The empirical results that we obtained improved on the existing GPU work on this problem, obtaining a peak improvement of 11× on our system over the CPU control experiments in the case where alignment scores were calculated on the GPU, using the multiple text batching approach, and for a particular input size. We also conducted a case study of the various approaches, concluding that the best performance for this problem was to load as many jobs on the GPU as possible, and to perform all calculation on the GPU, even in the case of computing the optimal alignment backtracking, which is a sequential computation. We also demonstrated that our algorithm's performance scaled as expected on GPU of different specification.

In Chapter 6, we proposed `GPU-MSI`, a GPU-based algorithm that solves the match spectral identification problem for identifying protein spectra in a database. To our knowledge, this is the first GPU-based solution to date. We gave analysis of the algorithm using the ATGPU model, and successfully verified the performance using experiments. Our experiments shown that, for the different input sizes tested, this algorithm yielded a promising 22× improvement at its peak over the best performing CPU control on our system.

We can therefore conclude that we have successfully proposed two GPU-based algorithms that improve on existing solutions to bioinformatics tasks.

We now discuss future work.

For the ATGPU model that was introduced in Chapter 4, it would be desirable in the future to carry out further experiments on other computational problems to verify our model; We think there is potential to look for ways in which global memory management on the ATGPU can be improved, in order to analyse global memory usage in a better way. Furthermore, it is desirable to verify the model using other GPUs. It is also interesting to consider extending the model to capture bank conflicts, though these are hard to analyse and difficult to predict. In particular, it would be good to investigate the use of multiple streams, as it is common in CUDA applications to have a compute stream and a memory stream operating concurrently, which can hide latency of expensive data transfer operations and give further practical improvement. We would also be interested in extending the model to capture a multiple GPU setting.

For `GPUGapsMis`, introduced in Chapter 5, it would be interesting in the future to investigate different data-parallel approaches to lower the amount of shared memory required, as well as investigate task parallel methods. In addition to this, it would also be interesting to look at ways to improve the performance of the backtracking phase, possibly by using a task-parallel GPU kernel. We use only a single GPU device in this thesis, so it would be interesting to investigate using multiple GPU devices to test further scalability, as well as to use higher specification GPUs to verify the improved speed up claim. Furthermore, it would be interesting to consider GPU variants for other alignment problems, e.g. those that may replace BWA or Bowtie.

In the future, it would be interesting to investigate `GPU-MSI` (introduced in Chapter 6) in calculating the *diagonal score identification* problem, also in [53], which uses the same *match score identification* scheme that was studied, with extended inputs of the spectra. This allows the algorithm to find PTMs at the edges of alignments and also to score individual diagonal paths through the alignment graph. It is the application of the match score identification algorithm as the diagonal score identification scoring scheme, which is used for filtering in other approaches, as detailed in Chapter 3. Additionally, it would be good to investigate this problem using real datasets of experimental protein spectra, to see how the algorithm performs in a more likely real-world setting.

The limitations of the ATGPU model mean that bank conflicts and uneven work on the cores make analysing some algorithms difficult, as this is difficult to predict. In the future, it would be interesting to implement the index match algorithm and the list match algorithm on the GPU, and to give analysis on an improved ATGPU model that captures bank conflicts and uneven work loads, to investigate this problem further.

Bibliography

[1] Nikolaos Alachiotis, Simon Berger, Tomáš Flouri, Solon P Pissis, and Alexandros Stamatakis. libgapmis: extending short-read alignments. *BMC Bioinformatics*, 14 (Suppl 11):S4, 2013. doi: 10.1186/1471-2105-14-S11-S4.

[2] Stephen F Altschul. Generalized affine gap costs for protein sequence alignment. *Proteins: Structure, Function, and Bioinformatics*, 32(1):88–96, 1998. doi: 10.1002/(SICI)1097-0134(19980701)32:1<88::AID-PROT10>3.0.CO;2-J.

[3] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3): 403–410, 1990. ISSN 0022-2836. doi: 10.1016/S0022-2836(05)80360-2.

[4] Lars Arge, Michael T Goodrich, Michael Nelson, and Nodari Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 197–206. ACM, 2008. ISBN 9781595939739. doi: 10.1145/1378533.1378573.

[5] Carl Barton, Tomáš Flouri, Costas S Iliopoulos, and Solon P Pissis. GapsMis: flexible sequence alignment with a bounded number of gaps. In *Proceedings of the International Conference on Bioinformatics, Computational Biology and Biomedical Informatics*, BCB'13, pages 402–411, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2434-2. doi: 10.1145/2506583.2506584.

[6] Lydia Ashleigh Baumgardner, Avinash Kumar Shanmugam, Henry Lam, Jimmy K. Eng, and Daniel B Martin. Fast parallel tandem mass spectral library searching using GPU hardware acceleration. *Journal of Proteome Research*, 10(6):2882–2888, 2011. doi: 10.1021/pr200074h.

[7] Bhubaneswar Mishra and O Gill and N Ramakrishnan. PLANAR: RNA sequence alignment using non-affine gap penalty and secondary structure. In *International*

symposium on computational biology and bioinformatics (ISBB '06), Bhubaneswar, India December 15-17, 2006",2006, 2006.

[8] Mauro Bianco and Geppino Pucci. On the predictive quality of BSP-like cost functions for NOWs. In *European Conference on Parallel Processing*, pages 638–646. Springer, 2000. doi: 10.1007/3-540-44520-X_89.

[9] Michael Boyer, Jiayuan Meng, and Kalyan Kumaran. Improving GPU performance prediction with data transfer modeling. In *IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 1097–1106. IEEE, 2013. doi: 10.1109/IPDPSW.2013.236.

[10] Vincent Boyer, Didier El Baz, and Moussa Elkihel. Dense dynamic programming on multi GPU. *19th International Euromicro Conference on Parallel, Distributed, and Network-Based Processing*, pages 545–551, 2011. ISSN 1066-6192. doi: 10.1109/PDP.2011.25.

[11] Alhadi Bustamam, Kevin Burrage, and Nicholas A Hamilton. Fast parallel Markov clustering in bioinformatics using massively parallel computing on GPU with CUDA and ELLPACK-R sparse format. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 9(3):679–692, 2012. doi: 10.1109/TCBB.2011.68.

[12] Thomas C Carroll and Prudence W H Wong. An Improved Abstract GPU Model with Data Transfer. In *Proceedings of the International Conference on Parallel Programming Workshops*, pages 113–120. IEEE, 2017. doi: 10.1109/ICPPW.2017.28.

[13] Thomas C Carroll, Jude-Thaddeus Ojiaku, and Prudence W H Wong. Pairwise Sequence Alignment with Gaps with GPU. In *2015 IEEE International Conference on Cluster Computing*, pages 603–610. IEEE, 2015. doi: 10.1109/CLUSTER.2015.10.

[14] Thomas C Carroll, Jude-Thaddeus Ojiaku, and Prudence WH Wong. Semiglobal Sequence Alignment with Gaps using GPU. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2019. doi: 10.1109/TCBB.2019.2914105. (To Appear).

[15] Reed A. Cartwright. Logarithmic gap costs decrease alignment accuracy. *BMC Bioinformatics*, 7(1):527, 2006. ISSN 1471-2105. doi: 10.1186/1471-2105-7-527.

[16] Reed A. Cartwright. Ngila: global pairwise alignments with logarithmic and affine gap costs. *Bioinformatics*, 23(11):1427–1428, 2007. ISSN 13674803. doi: 10.1093/bioinformatics/btm095.

[17] Angana Chakraborty and Sanghamitra Bandyopadhyay. FOGSAA: Fast optimal global sequence alignment algorithm. *Scientific Reports*, 3(Suplementary material): 1746, 2013. ISSN 2045-2322. doi: 10.1038/srep01746.

[18] Freddy Cliquet, Guillaume Fertin, Irena Rusu, and Dominique Tessier. Proper alignment of MS/MS spectra from unsequenced species. In *11th International Conference on Bioinformatics and Computational Biology*, pages 766–772, 2010.

[19] Stephen Cossell. Parallel Evaluation of a Spatial Traversability Cost Function on GPU for Efficient Path Planning. *Journal of Intelligent Learning Systems and Applications*, 03(04):191–200, 2011. ISSN 2150-8402, 2150-8410. doi: 10.4236/jilsa.2011.34022.

[20] Maxime Crochemore, Gad M Landau, Michal Ziv-Ukelson, and J Comput Siam. A subquadratic sequence alignment algorithm for unrestricted scoring matrices. *SIAM journal on computing*, 32(6):1654–1673, 2003. doi: 10.1137/S0097539702402007.

[21] Giso H. Dal, Walter A. Kosters, and Frank W. Takes. Fast diameter computation of large sparse graphs using GPUs. pages 632–639, 2014. doi: 10.1109/PDP.2014.17.

[22] Reza Farivar, Harshit Kharbanda, Shivaram Venkataraman, and Roy H. Campbell. An algorithm for fast edit distance computation on GPUs. *2012 Innovative Parallel Computing*, pages 0–8, 2012. doi: 10.1109/InPar.2012.6339593.

[23] Michael Farrar. Striped Smith–Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, 23(2):156–161, 2007. doi: 10.1093/bioinformatics/btl582.

[24] Tomas Flouri, Kimon Frousios, Costas S Iliopoulos, Kunsoo Park, Solon P Pissis, and German Tischler. GapMis: a tool for pairwise sequence alignment with a single gap. *Recent patents on DNA & gene sequences*, 7(2):84–95, 2013.

[25] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118. ACM, 1978. doi: 10.1145/800133.

[26] Ari M Frank, James J Pesavento, Craig A Mizzen, Neil L Kelleher, and Pavel A Pevzner. Interpreting Top-Down Mass Spectra using Spectral Alignment. *Analytical Chemistry*, 80(7), 2008. doi: 10.1021/ac702324u.

[27] Martin C Frith, Raymond Wan, and Paul Horton. Incorporating sequence quality data into alignment improves DNA read mapping. *Nucleic Acids Research*, 38(7): e100–e100, 2010. doi: 10.1093/nar/gkq010.

[28] Kimon Frousios, Costas S. Iliopoulos, Laurent Mouchard, Solon P. Pissis, and German Tischler. REAL: an efficient REad ALigner for next generation sequencing reads. pages 154–159, 2010. doi: 10.1145/1854776.1854801.

[29] Yusuke Fujii, Takuya Azumi, Nobuhiko Nishio, Shinpei Kato, and Masato Edahiro. Data Transfer Matters for GPU Computing. *Proceedings of the International Conference on Parallel and Distributed Systems*, pages 275–282, 2013. ISSN 15219097. doi: 10.1109/ICPADS.2013.47.

[30] Wilson W L Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. *Proceedings of the Annual International Symposium on Microarchitecture*, pages 407–418, 2007. ISSN 10724451. doi: 10.1109/MICRO.2007.30.

[31] Nalin C W Goonesekere and Byungkook Lee. Frequency of gaps observed in a structurally aligned protein pair database suggests a simple gap penalty function. *Nucleic Acids Research*, 32(9):2838–2843, 2004. ISSN 03051048. doi: 10.1093/nar/gkh610.

[32] Chris Gregg and Kim Hazelwood. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 134–144. IEEE, apr 2011. ISBN 978-1-61284-367-4. doi: 10.1109/ISPASS.2011.5762730.

[33] Mark Harris. Optimizing Parallel Reduction in CUDA. Technical report, nVidia Inc, 2008. URL `http://www.mendeley.com/research/optimizing-parallel-reduction-cuda/`.

[34] Sunpyo Hong and Hyesoon Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 152–163. ACM, 2009. doi: 10.1145/1555815.

[35] Thomas B. Jablin, Prakash Prabhu, James a. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. Automatic CPU-GPU communication management and optimization. *ACM SIGPLAN Notices*, 47(6):142, 2012. ISSN 03621340. doi: 10.1145/2345156.1993516.

[36] Shinpei Kato, Jason Aumiller, and Scott Brandt. Zero-copy I/O processing for low-latency GPU computing. In *2013 ACM/IEEE International Conference on Cyber-Physical Systems*, pages 170–178, 2013. ISBN 9781450319966. doi: 10.1109/ICCPS.2013.6604011.

[37] Khronos. The OpenCL Specification Version 1.2. Technical report, Khronos OpenCL Working Group, 2012. URL `http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`.

[38] Kai J Kohlhoff, Marc H Sosnick, William T Hsu, Vijay S Pande, and Russ B Altman. CAMPAIGN: an open-source library of GPU-accelerated data clustering algorithms. *Bioinformatics*, 27(16):2321–2322, 2011. doi: 10.1093/bioinformatics/btr386.

[39] Atsushi Koike and Kunihiko Sadakane. A Novel Computational Model for GPUs with Application to IO Optimal Sorting Algorithms. In *2014 IEEE International Parallel and Distributed Processing Symposium Workshhops*, pages 614–623, 2014. doi: 10.1109/IPDPSW.2014.72.

[40] Elias Konstantinidis and Yiannis Cotronis. A practical performance model for compute and memory bound GPU kernels. In *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2015. doi: 10.1109/PDP.2015.51.

[41] Kishore Kothapalli, Rishabh Mukherjee, M Suhail Rehman, Suryakant Patidar, P. J. Narayanan, and Kannan Srinathan. A performance prediction model for the CUDA GPGPU platform. In *2009 International Conference on High Performance Computing*, pages 463–472. IEEE, 2009. ISBN 9781424449224. doi: 10.1109/HIPC.2009.5433179.

[42] Qiang Kou, Si Wu, and Xiaowen Liu. A new scoring function for top-down spectral deconvolution. *BMC Genomics*, 15(1):1, 2014. doi: 10.1186/1471-2164-15-1140.

[43] Qiang Kou, Likun Xun, and Xiaowen Liu. Toppic: a software tool for top-down

mass spectrometry-based proteoform identification and characterization. *Bioinformatics*, 32(22):3495–3497, 07 2016. doi: 10.1093/bioinformatics/btw398.

[44] B Langmead, C Trapnell, M Pop, and S Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3): R25, 2009. ISSN 1465-6914. doi: 10.1186/gb-2009-10-3-r25.

[45] Richard D. LeDuc, Gregory K. Taylor, Yong-Bin Kim, Thomas E. Januszyk, Lee H. Bynum, Joseph V. Sola, John S. Garavelli, and Neil LS Kelleher. ProSight PTM: an integrated environment for protein identification and characterization by top-down mass spectrometry. *Nucleic Acids Research*, 32(suppl_2):W340–W345, 07 2004. ISSN 0305-1048. doi: 10.1093/nar/gkh447.

[46] Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009. ISSN 13674803. doi: 10.1093/bioinformatics/btp324.

[47] Ruiqiang Li, Yingrui Li, Karsten Kristiansen, and Jun Wang. SOAP: short oligonucleotide alignment program. *Bioinformatics*, 24(5):713–714, 2008. ISSN 13674803. doi: 10.1093/bioinformatics/btn025.

[48] Ruiqiang Li, Chang Yu, Yingrui Li, Tak Wah Lam, Siu Ming Yiu, Karsten Kristiansen, and Jun Wang. SOAP2: An improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009. ISSN 13674803. doi: 10.1093/bioinformatics/btp336.

[49] Ruiqiang Li, Hongmei Zhu, Jue Ruan, and Wubin Qian. De novo assembly of human genomes with massively parallel short read sequencing. *Genome Research*, 20:265–272, 2010. doi: 10.1101/gr.097261.109.

[50] You Li, Hao Chi, Leihao Xia, and Xiaowen Chu. Accelerating the scoring module of mass spectrometry-based peptide identification using GPUs. *BMC Bioinformatics*, 15(1):121, 2014. doi: 10.1186/1471-2105-15-121.

[51] Lukasz Ligowski, Witold Rudnicki, Łukasz Ligowski, and Witold Rudnicki. An efficient implementation of Smith Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. In *IEEE International Symposium on Parallel & Distributed Processing, 2009*, pages 1–8. IEEE, 2009. doi: 978-1-4244-3750-4.

[52] Chi Man Liu, Thomas Wong, Edward Wu, Ruibang Luo, Siu Ming Yiu, Yingrui Li, Bingqiang Wang, Chang Yu, Xiaowen Chu, Kaiyong Zhao, Ruiqiang Li, and Tak Wah Lam. SOAP3: ultra-fast GPU-based parallel alignment tool for short reads. *Bioinformatics*, 28(6):878–879, 2012. doi: 10.1093/bioinformatics/bts061.

[53] Xiaowen Liu, Alessandro Mammana, and Vineet Bafna. Speeding Up tandem mass spectral identification using indexes. *Bioinformatics*, 28(13):1692–1697, 2012. doi: 10.1093/bioinformatics/bts244.

[54] Xiaowen Liu, Yakov Sirotkin, Yufeng Shen, Gordon Anderson, Yihsuan S Tsai, Ying S Ting, David R Goodlett, Richard D Smith, Vineet Bafna, and Pavel A Pevzner. Protein identification using top-down spectra. *Molecular & Cellular Proteomics*, 11(6):M111–008524, 2012. doi: 10.1074/mcp.M111.008524.

[55] Xiaowen Liu, Shawna Hengel, Si Wu, Nikola Tolic, Ljiljana Pasa-Tolic, and Pavel A Pevzner. Identification of ultramodified proteins using top-down tandem mass spectra. volume 12, pages 5830–5838. ACS Publications, 2013. doi: 10.1021/pr400849y.

[56] Yongchao Liu, Adrianto Wirawan, and Bertil Schmidt. CUDASW++3.0 accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. *BMC Bioinformatics*, 14(117), 2013. doi: 10.1186/1471-2105-14-117.

[57] Gerton Lunter and Martin Goodson. Stampy: A statistical algorithm for sensitive and fast mapping of Illumina sequence reads. *Genome Research*, 21(6):936–939, 2011. ISSN 10889051. doi: 10.1101/gr.111120.110.

[58] M. S. Madhusudhan, Marc A. Marti-Renom, Roberto Sanchez, and Andrej Sali. Variable gap penalty for protein sequence-structure alignment. *Protein Engineering, Design and Selection*, 19(3):129–133, 2006. ISSN 17410126. doi: 10.1093/protein/gzj005.

[59] Svetlin A Manavski and Giorgio Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, 9(Suppl 2):S10, 2008. doi: 10.1186/1471-2105-9-S2-S10.

[60] Sergio Martin, Fernando Gustavo Tinetti, Nicanor Casas, Graciela De Luca, and Daniel Alberto Giulianelli. N-body simulation using GP-GPU: Evaluating host/device memory transference overhead. In *XVIII Congreso Argentino de Ciencias de la Computación*, 2013.

[61] Jeffrey A Milloy, Brendan K Faherty, and Scott A Gerber. Tempest: GPU-CPU computing for high-throughput database spectral matching. *Journal of Proteome Research*, 11(7):3581–3591, 2012. doi: 10.1021/pr300338p.

[62] Richard Mott. Local sequence alignments with monotonic gap penalties. *Bioinformatics*, 15(6):455–462, 1999. ISSN 13674803. doi: 10.1093/bioinformatics/15.6.455.

[63] Michael Garland Nadathur Satish Mark Harris. Designing Efficient Sorting Algorithms for Manycore GPUs. Technical report, nVidia and UC Berkeley, 2008. URL `http://www.nvidia.com/docs/io/67073/nvr-2008-001.pdf`.

[64] NCBI. NCBI Genbank. URL `http://www.ncbi.nlm.nih.gov/genbank`.

[65] S B Needleman and C D Wunsch. A general method applicabe to the search for similarities in the amino acid sequence of two proteins. *Journal of Mollecular Biology*, 48:443–453, 1970.

[66] nVidia. Programming Guide:: CUDA Toolkit Documentation. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`, aug 2017. URL `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

[67] nVidia Inc. Programming Guide:: CUDA Toolkit Documentation, aug 2017. URL `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

[68] Jude-Thaddeus Ojiaku. *A Study of Time and Energy Efficient Algorithms for Parallel and Heterogeneous Computing.* PhD thesis, University of Liverpool, Dec 2014.

[69] David N. Perkins, Darryl J. C. Pappin, David M. Creasy, and John S. Cottrell. Probability-based protein identification by searching sequence databases using mass spectrometry data. *Electrophoresis*, 20(18):3551–3567, 1999. doi: 10.1002/(SICI)1522-2683(19991201)20:18<3551::AID-ELPS3551>3.0.CO;2-2.

[70] Pavel A Pevzner, Vlado Dancík, Chris L Tang, Vlado Dan, Ï Cík, and Chris L Tang. Mutation-Tolerant Protein Identification by Mass Spectrometry. *Journal of Computational Biology*, 7(6):777–787, 2000. doi: 10.1089/10665270050514927.

[71] Sooraj Puthoor, Ashwin M Aji, Shuai Che, Mayank Daga, Wei Wu, Bradford M Beckmann, and Gregory Rodgers. Implementing directed acyclic graphs with the heterogeneous system architecture. In *Proceedings of the 9th Annual Workshop on*

General Purpose Processing using Graphics Processing Unit, pages 53–62. ACM, 2016. doi: 10.1145/2884045.2884052.

[72] Peter Rice, Ian Longden, and Alan Bleasby. EMBOSS: The European Molecular Biology Open Software Suite. *Trends in Genetics*, 16(1):276–277, 2000. ISSN 13590294. doi: 10.1016/j.cocis.2008.07.002.

[73] Stephen M. Rumble, Phil Lacroute, Adrian V. Dalca, Marc Fiume, Arend Sidow, and Michael Brudno. SHRiMP: Accurate mapping of short color-space reads. *PLoS Computational Biology*, 5(5):1–11, 2009. ISSN 1553734X. doi: 10.1371/journal.pcbi.1000386.

[74] Korbinian Schneeberger, Jörg Hagmann, Stephan Ossowski, Norman Warthmann, Sandra Gesing, Oliver Kohlbacher, and Detlef Weigel. Simultaneous alignment of short reads against multiple genomes. *Genome Biology*, 10(9):R98, 2009. ISSN 1474-760X. doi: 10.1186/gb-2009-10-9-r98.

[75] Mohammed A Shehab, Abdullah A Ghadawi, Luay Alawneh, Mahmoud Al-Ayyoub, and Yaser Jararweh. A hybrid CPU-GPU implementation to accelerate multiple pairwise protein sequence alignment. In *8th International Conference on Information and Communication Systems*, pages 12–17. IEEE, 2017. doi: 10.1109/IACS.2017.7921938.

[76] Jared T Simpson, Kim Wong, Shaun D Jackman, Jacqueline E Schein, Steven JM M Jones, and Inanç Inanc Birol. ABySS: a parallel assembler for short read sequence data. *Genome Research*, 19(6):1117–1123, 2009. doi: 10.1101/gr.089532.108.

[77] Nodari Sitchinava and Volker Weichert. Provably efficient gpu algorithms. *arXiv preprint arXiv:1306.5076*, 2013.

[78] Andrew D Smith, Zhenyu Xuan, and Michael Q Zhang. Using quality scores and longer reads improves accuracy of Solexa read mapping. *BMC Bioinformatics*, 9:128, 2008. ISSN 1471-2105. doi: 10.1186/1471-2105-9-128.

[79] Temple F Smith and Michael S Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[80] Peter Steffen, Robert Giegerich, and Mathieu Giraud. GPU parallelization of algebraic dynamic programming. *Parallel Processing and Applied Mathematics*, 6068

LNCS(PART 2):290–299, 2010. ISSN 03029743. doi: 10.1007/978-3-642-14403-5_31.

[81] Gregory M Striemer and Ali Akoglu. Sequence alignment with GPU: Performance and design challenges. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–10. IEEE, 2009. doi: 10.1109/IPDPS.2009.5161066.

[82] Alexandre Tiskin. The bulk-synchronous parallel random access machine. *Theoretical Computer Science*, 196(1):109–130, 1998. ISSN 03043975. doi: 10.1016/S0304-3975(97)00197-7.

[83] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990. doi: 10.1145/79173.79181.

[84] Panagiotis D. Vouzis and Nikolaos V. Sahinidis. GPU-BLAST: using graphics processors to accelerate protein sequence alignment. *Bioinformatics*, 27(2):182–188, 11 2010. ISSN 1367-4803. doi: 10.1093/bioinformatics/btq644.

[85] Ben. Van Werkhoven, Jaso. Maassen, Frank. J. Seinstra, and Henri. E. Bal. Performance models for CPU-GPU data transfers. In *14th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing,*, pages 11–20, 2014. ISBN 9781479927838. doi: 10.1109/CCGrid.2014.16.

[86] Ping Yao, Hong An, Mu Xu, Gu Liu, Xiaoqiang Li, Yaobin Wang, and Wenting Han. CuHMMer: A load-balanced CPU-GPU cooperative bioinformatics application. In *2010 International Conference on High Performance Computing & Simulation*, pages 24–30. IEEE, 2010. doi: 10.1109/HPCS.2010.5547159.

[87] Marcus A. Zachariah, Gavin E. Crooks, Stephen R. Holbrook, and Steven E. Brenner. A generalized affine gap model significantly improves protein sequence alignment accuracy. *Proteins: Structure, Function, and Bioinformatics*, 58(2):329–338, 2005. ISSN 08873585. doi: 10.1002/prot.20299.

[88] Yuhong Zhang, Sanchit Misra, Daniel Honbo, Ankit Agrawal, Wei-keng Liao, and Alok Choudhary. Efficient pairwise statistical significance estimation for local sequence alignment using GPU. In *2011 IEEE 1st International Conference on Computational Advances in Bio and Medical Sciences*, pages 226–231. IEEE, 2011. doi: 10.1109/ICCABS.2011.5729885.

[89] Kaiyong Zhao and Xiaowen Chu. G-BLASTN: accelerating nucleotide alignment by graphics processors. *Bioinformatics*, 30(10 2014):1384–1391, 2014. ISSN 14602059. doi: 10.1093/bioinformatics/btu047.

Glossary

AGPU Abstract GPU Model. 3, 9, 21–24, 27, 43, 44, 48, 53, 60, 63, 66, 71–73, 80, 108

AMD Advanced Micro Devices, Inc.. 2, 12

API Application Programming Interface. 12, 13

ATGPU Abstract Transferring GPU Model. 7–10, 22, 23, 42, 43, 47–50, 52, 53, 57, 58, 60, 63, 71–73, 80, 97, 105, 107–110

BSP Bulk Synchronous Parallel Machine. 3, 18, 19

BSPRAM Bulk Synchronous Parallel Random Access Machine. 16, 18–20

CPU Central Processing Unit. 1–3, 9, 10, 12, 15, 17, 21, 24–26, 33, 34, 36, 37, 39, 42, 43, 57, 58, 60, 66, 70–72, 90, 97, 106–109

CRCW Concurrent Read Concurrent Write PRAM. 17

CREW Concurrent Read Exclusive Write PRAM. 17

CUDA Compute Unified Device Architecture. 2, 12, 13, 23, 39

DNA Deoxyribonucleic acid. 4, 9, 28–30, 33

EREW Exclusive Read Exclusive Write PRAM. 16

FPGA Field Programmable Gate Array. 2

GPGPU General Purpose Programming on GPU. 2, 39

GPU Graphics Processing Unit. 2–15, 17–28, 35, 36, 39, 41–44, 47, 48, 55, 57, 58, 60, 64, 66, 70–72, 80, 87–90, 94–97, 102, 106–110, 112

HPC High Performance Computing. 2

MP Multiprocessor. 21–23, 47, 48, 53

MS/MS Tandem Mass Spectroscopy. 5, 9, 37, 38

OpenCL Open Computing Language. 2, 12, 23

PCIe Peripheral Component Interconnect Express. 2, 12, 14

PEM Parallel External Memory Machine. 3, 20, 21

PRAM Parallel Random Access Machine. 3, 16, 17, 19

PTM Post Translational Modification. 5, 36–38, 89, 97, 110

RNA Ribonucleic acid. 36

SM *Streaming Multiprocessor*. 12–14, 20, 21, 112

SW-GPU Sitchinava Weichert GPU Model. 3, 21, 23, 24, 27, 43, 44, 47, 49, 50, 53, 71–73

Appendix A

Specification of Hardware and Software Setting for Experiments

In this thesis, we utilise the AMD A10 5800-K[1] CPU, Nvidia GTX 650[2] and the Nvidia GTX 680[3] GPUs installed in custom build workstations. Table A.1 gives the specifications of the machines.

Table A.1: Specifications of the custom build workstations used for experiments within this thesis.

Item	**Machine 1**	**Machine 2**
CPU	AMD A10 5800-K	AMD A10 5800-K
RAM	16GB	16GB
GPU	Nvidia GTX 650	Nvidia GTX 680
OS	Ubuntu	Ubuntu
Other	nVidia CUDA proprietary drivers	nVidia CUDA proprietary drivers

[1]4 cores, clocked at 3.8 GHz
[2]2 SMs, 384 CUDA cores, clocked at 1.2 Ghz, 1GB device memory
[3]8 SMs, 1536 CUDA cores, clocked at 1 Ghz, 2GB device memory

Appendix B

Full Results Tables for GPUGapsMis

This appendix contains the full results tables for all experiments detailed for `GPUGapsMis` in Chapter 5. Several approaches are used:

- `GPU-S-A` computes only the alignment phase, with a single text on the GPU (Table B.1)
- `GPU-S-B` computes the alignment phase and backtracking phase on the GPU, with a single text on the GPU (Table B.2).
- `GPU-S-H` computes the alignment phase on the GPU with at most a single text, and the backtracking phase on the CPU (Table B.3).

Likewise, the approaches `GPU-M-A` (Table B.4), `GPU-M-B` (Table B.5) and `GPU-M-H` (Table B.6) compute for multiple text on the GPU. The approaches `CPU-A` and `CPU-B` are used as control for alignment phase only, and for including backtracking phase, respectively.

Table B.1: Results for `GPU-S-A`

Patt Length	50					100				
	Latency (seconds)		Throughput (MCUPS)		Speedup	Latency (seconds)		Throughput(MCUPS)		Speedup
Num Seqs	`CPU-A`	`GPU-S-A`	`CPU-A`	`GPU-S-A`		`CPUGapsMis`	`GPU-S-A`	`CPUGapsMis`	`GPU-S-A`	
1600	4.0	0.4	10.3	84.9	8.2	8.0	1.0	10.2	78.8	7.7
3200	7.9	1.0	10.4	85.3	8.2	15.9	2.0	10.2	78.8	7.7
6400	15.9	1.9	10.3	85.3	8.3	31.6	4.0	10.3	79.5	7.7
12800	31.8	3.8	10.3	86.3	8.4	63.6	8.1	10.2	79.8	7.8
25600	63.5	7.6	10.3	86.3	8.4	126.4	16.2	10.3	79.8	7.8
51200	127.3	15.2	10.3	86.3	8.4	253.7	32.5	10.2	79.8	7.8
102400	254.2	30.4	10.3	86.4	8.4	508.5	65.0	10.2	79.8	7.8
204800	509.0	60.7	10.3	**86.4**	**8.4**	1012.2	130.1	10.3	79.8	7.8

Patt Length	150					200				
	Latency (seconds)		Throughput (MCUPS)		Speedup	Latency (seconds)		Throughput (MCUPS)		Speedup
Num Seqs	`CPU-A`	`GPU-S-A`	`CPU-A`	`GPU-S-A`		`CPU-A`	`GPU-S-A`	`CPU-A`	`GPU-S-A`	
1600	12.3	1.7	9.9	73.7	7.5	16.4	2.2	9.8	73.5	7.5
3200	23.7	3.3	10.2	73.8	7.2	33.0	4.4	9.8	73.6	7.5
6400	47.6	6.5	10.2	74.6	7.3	65.9	8.7	9.8	74.5	7.6
12800	94.7	13.0	10.3	74.7	7.3	131.4	17.3	9.8	74.5	7.6
25600	200.0	26.0	10.2	74.7	7.4	263.6	34.7	9.8	74.5	7.6
51200	380.8	52.0	10.2	74.7	7.3	527.3	69.4	9.8	74.5	7.6
102400	759.7	104.0	10.2	74.7	7.3	1053.3	138.8	9.8	74.5	7.6
204800	1525.0	207.9	10.2	74.7	7.3	2106.0	277.5	9.8	74.5	7.6

Table B.2: Results for `GPU-S-B`

Patt Length	50					100				
	Latency (seconds)		Throughput (MCUPS)		Speedup	Latency (seconds)		Throughput(MCUPS)		Speedup
Num Seqs	`CPU-B`	`GPU-S-B`	`CPU-B`	`GPU-S-B`		`CPU-B`	`GPU-S-B`	`CPU-B`	`GPU-S-B`	
1600	4.0	0.5	15.5	120.9	7.8	8.0	1.0	15.3	113.5	7.4
3200	7.9	1.0	15.5	121.0	7.8	15.9	2.1	15.3	113.5	7.4
6400	15.9	2.0	15.4	121.0	7.8	31.6	4.3	15.4	113.6	7.4
12800	31.9	4.0	15.4	121.0	7.8	63.7	8.6	15.3	113.5	7.4
25600	63.6	8.1	15.5	121.0	7.8	126.5	17.2	15.4	113.6	7.4
51200	127.5	16.2	15.4	121.0	7.8	253.9	34.3	15.3	113.6	7.4
102400	254.5	32.5	15.5	121.0	7.8	508.9	68.6	15.3	113.6	7.4
204800	509.7	65.0	15.4	**121.0**	**7.8**	1013.0	137.2	15.4	113.6	7.4

Patt Length	150					200				
	Latency (seconds)		Throughput (MCUPS)		Speedup	Latency (seconds)		Throughput (MCUPS)		Speedup
Num Seqs	`CPU-B`	`GPU-S-B`	`CPU-B`	`GPU-S-B`		`CPU-B`	`GPU-S-B`	`CPU-B`	`GPU-S-B`	
1600	12.3	1.7	14.8	106.8	7.2	16.4	2.3	14.7	106.3	7.2
3200	23.7	3.4	15.3	106.8	7.0	33.0	4.6	14.7	106.3	7.2
6400	47.6	6.8	15.3	106.8	7.0	65.9	9.1	14.7	106.4	7.2
12800	94.7	13.6	15.4	106.8	6.9	131.5	18.2	14.7	106.4	7.2
25600	191.1	27.3	15.2	106.8	7.0	263.7	36.4	14.7	106.4	7.2
51200	381.0	54.5	15.3	106.8	7.0	527.6	72.8	14.7	106.4	7.2
102400	760.2	109.0	15.3	106.8	7.0	1053.8	145.7	14.7	106.4	7.2
204800	1525.9	218.2	15.3	106.7	7.0	2107.0	291.4	14.7	106.4	7.2

Table B.3: Results for `GPU-S-H`

Patt Length	50					100				
	Latency (seconds)		Throughput (MCUPS)		Speedup	Latency (seconds)		Throughput(MCUPS)		Speedup
Num Seqs	`CPU-B`	`GPU-S-H`	`CPU-B`	`GPU-S-H`		`CPU-B`	`GPU-S-H`	`CPU-B`	`GPU-S-H`	
1600	4.0	1.3	15.5	48.1	3.1	8.0	2.7	15.3	45.7	3.0
3200	7.9	2.6	15.5	48.1	3.1	15.9	5.3	15.3	45.7	3.0
6400	15.9	5.1	15.4	48.1	3.1	31.6	10.7	15.4	45.7	3.0
12800	31.9	10.2	15.4	48.1	3.1	63.7	21.3	15.3	45.7	3.0
25600	63.6	20.4	15.5	48.1	3.1	126.5	42.6	15.4	45.7	3.0
51200	127.5	40.9	15.4	48.1	3.1	253.9	85.2	15.3	45.7	3.0
102400	254.5	81.8	15.5	48.1	3.1	508.9	170.4	15.3	45.7	3.0
204800	509.7	163.5	15.4	**48.1**	**3.1**	1013.0	340.7	15.4	45.7	3.0
Patt Length	150					200				
	Latency (seconds)		Throughput (MCUPS)		Speedup	Latency (seconds)		Throughput (MCUPS)		Speedup
Num Seqs	`CPU-B`	`GPU-S-H`	`CPU-B`	`GPU-S-H`		`CPU-B`	`GPU-S-H`	`CPU-B`	`GPU-S-H`	
1600	12.3	4.0	14.8	45.2	3.1	16.4	5.3	14.7	45.8	3.1
3200	23.7	8.1	15.3	45.2	3.0	33.0	10.6	14.7	45.7	3.1
6400	47.6	16.1	15.3	45.2	3.0	65.9	21.2	14.7	45.7	3.1
12800	94.7	32.2	15.4	45.2	2.9	131.5	42.4	14.7	45.7	3.1
25600	191.1	64.4	15.2	45.2	3.0	263.7	84.8	14.7	45.7	3.1
51200	381.0	128.8	15.3	45.2	3.0	527.6	169.6	14.7	45.7	3.1
102400	760.2	257.6	15.3	45.2	3.0	1053.8	339.2	14.7	45.7	3.1
204800	1525.9	515.3	15.3	45.2	3.0	2107.0	678.3	14.7	45.7	3.1

Table B.4: Results for `GPU-M-A`

Patt Length	50					100				
	Latency (seconds)		Throughput (MCUPS)		Speedup	Latency (seconds)		Throughput(MCUPS)		Speedup
Num Seqs	`CPU-A`	`GPU-M-A`	`CPU-A`	`GPU-M-A`		`CPU-A`	`GPU-M-A`	`CPU-A`	`GPU-M-A`	
1600	4.0	0.4	10.3	105.4	10.2	8.0	0.8	10.2	99.5	9.8
3200	7.9	0.8	10.4	109.0	10.5	15.9	1.6	10.2	99.8	9.8
6400	15.9	1.5	10.3	109.7	10.6	31.6	3.2	10.3	101.0	9.8
12800	31.8	2.9	10.3	111.7	10.8	63.6	6.4	10.2	101.5	9.9
25600	63.5	5.9	10.3	111.7	10.8	126.4	12.8	10.3	101.5	9.9
51200	127.3	11.8	10.3	111.3	10.8	253.7	25.9	10.2	100.4	9.8
102400	254.2	23.2	10.3	113.2	11.0	508.5	50.5	10.2	102.9	10.0
204800	509.0	46.3	10.3	**113.2**	**11.0**	1012.2	101.0	10.3	102.9	10.0

Patt Length	150					200				
	Latency (seconds)		Throughput (MCUPS)		Speedup	Latency (seconds)		Throughput (MCUPS)		Speedup
Num Seqs	`CPU-A`	`GPU-M-A`	`CPU-A`	`GPU-M-A`		`CPU-A`	`GPU-M-A`	`CPU-A`	`GPU-M-A`	
1600	12.3	1.2	9.9	101.6	10.3	16.4	1.8	9.8	91.5	9.3
3200	23.7	2.4	10.2	102.9	10.0	33.0	3.5	9.8	91.6	9.4
6400	47.6	4.7	10.2	104.3	10.2	65.9	6.9	9.8	92.7	9.5
12800	94.7	9.3	10.3	104.4	10.2	131.4	13.9	9.8	92.8	9.4
25600	191.0	18.6	10.2	104.5	10.3	263.6	27.8	9.8	92.8	9.5
51200	380.8	37.6	10.2	103.2	10.1	527.3	56.1	9.8	92.1	9.4
102400	759.7	73.4	10.2	105.7	10.4	1053.3	110.0	9.8	93.96	9.58
204800	1525.0	146.8	10.2	105.8	10.4	2106.0	219.9	9.8	94.0	9.6

Table B.5: Results for `GPU-M-B`

Patt Length	50					100				
	Latency (seconds)		Throughput (MCUPS)		Speedup	Latency (seconds)		Throughput(MCUPS)		Speedup
Num Seqs	`CPU-B`	`GPU-M-B`	`CPU-B`	`GPU-M-B`		`CPU-B`	`GPU-M-B`	`CPU-B`	`GPU-M-B`	
1600	4.0	0.4	15.5	154.2	10.0	8.0	0.8	15.3	145.3	9.5
3200	7.9	0.8	15.5	158.7	10.2	15.9	1.7	15.3	146.3	9.6
6400	15.9	1.5	15.4	160.4	10.4	31.6	3.3	15.4	146.5	9.5
12800	31.9	3.1	15.4	160.6	10.4	63.7	6.6	15.3	146.7	9.6
25600	63.6	6.1	15.5	160.9	10.4	126.5	13.3	15.4	146.9	9.6
51200	127.5	12.2	15.4	161.1	10.4	253.9	26.5	15.3	146.9	9.6
102400	254.5	24.4	15.5	161.1	10.4	508.9	53.0	15.3	147.0	9.6
204800	509.7	48.8	15.4	**161.1**	**10.4**	1013.0	106.0	15.4	147.0	9.6
Patt Length	150					200				
	Latency (seconds)		Throughput (MCUPS)		Speedup	Latency (seconds)		Throughput (MCUPS)		Speedup
Num Seqs	`CPU-B`	`GPU-M-B`	`CPU-B`	`GPU-M-B`		`CPU-B`	`GPU-M-B`	`CPU-B`	`GPU-M-B`	
1600	12.3	1.2	14.8	149.2	10.1	16.4	1.8	14.7	134.5	9.1
3200	23.7	2.4	15.3	150.9	9.9	33.0	3.6	14.7	134.0	9.1
6400	47.6	4.8	15.3	150.9	9.9	65.9	7.2	14.7	134.6	9.2
12800	94.7	9.6	15.4	151.1	9.8	131.5	14.4	14.7	134.6	9.1
25600	191.1	19.3	15.2	151.0	9.9	263.7	28.8	14.7	134.7	9.2
51200	381.0	38.5	15.3	151.1	9.9	527.6	57.6	14.7	134.6	9.2
102400	760.2	77.1	15.3	151.1	9.9	1053.8	115.1	14.7	134.6	9.2
204800	1525.9	154.1	15.3	151.2	9.9	2107.0	230.2	14.7	134.7	9.2

Table B.6: Results for `GPU-M-H`

Patt Length	50					100				
	Latency (seconds)		Throughput (MCUPS)		Speedup	Latency (seconds)		Throughput(MCUPS)		Speedup
Num Seqs	`CPU-B`	`GPU-M-H`	`CPU-B`	`GPU-M-H`		`CPU-B`	`GPU-M-H`	`CPU-B`	`GPU-M-H`	
1600	4.0	1.3	5.5	45.8	3.0	8.0	2.5	6.1	49.53	3.2
3200	7.9	2.4	11.1	50.4	3.3	15.9	4.6	12.2	53.4	3.5
6400	15.9	4.6	22.2	53.8	3.5	31.6	9.0	24.4	54.3	3.5
12800	31.9	8.9	44.3	55.0	3.6	63.7	17.6	49.0	55.4	3.6
25600	63.6	17.5	88.8	56.1	3.6	126.5	34.8	97.5	56.0	3.6
51200	127.5	34.7	177.1	56.6	3.7	253.9	69.3	195.6	56.2	3.7
102400	254.5	69.2	352.5	56.8	3.7	508.9	138.5	391.6	56.2	3.7
204800	509.7	138.4	712.0	**56.8**	**3.7**	1013.0	276.8	779.5	56.3	3.7
Patt Length	150					200				
	Latency (seconds)		Throughput (MCUPS)		Speedup	Latency (seconds)		Throughput (MCUPS)		Speedup
Num Seqs	`CPU-B`	`GPU-M-H`	`CPU-B`	`GPU-M-H`		`CPU-B`	`GPU-M-H`	`CPU-B`	`GPU-M-H`	
1600	12.3	3.4	7.3	53.8	3.6	16.4	4.5	8.3	54.3	3.7
3200	23.7	6.4	14.6	56.5	3.7	33.0	8.7	16.6	55.6	3.8
6400	47.6	12.7	29.0	57.3	3.8	65.9	17.1	33.4	56.8	3.9
12800	94.7	25.2	58.0	57.7	3.8	131.5	33.8	66.3	57.4	3.9
25600	191.1	50.3	116.4	57.9	3.8	263.7	67.3	133.9	57.6	3.9
51200	381.0	100.2	232.5	58.0	3.8	527.6	134.3	268.0	57.7	3.9
102400	760.2	200.4	464.5	58.1	3.8	1053.8	268.7	532.2	57.7	3.9
204800	1525.9	400.3	928.5	58.2	3.8	2107.0	536.9	1062.8	57.7	3.9

Appendix C

Full Results Tables for GPU-MSI

This appendix contains the full results tables for all experiments detailed for GPU-MSI in Chapter 6.

Table C.1: Results for `GPU-MSI`

Num Lib Spectra	250				500			
Length	`CPU-MSI-V`	`CPU-MSI-I`	`GPU-MSI`	Speedup on GPU	`CPU-MSI-V`	`CPU-MSI-I`	`GPU-MSI`	Speedup on GPU
50	395	332	127	3	811	695	252	3
75	583	499	128	4	1211	1078	253	4
100	794	662	129	5	1645	1377	254	5
125	961	834	130	6	1999	1782	255	7
150	1176	1015	131	8	2440	2099	256	8
175	1358	1173	132	9	2905	2477	257	10
200	1590	1347	132	10	3270	2795	258	11

Num Lib Spectra	750				1000			
Length	`CPU-MSI-V`	`CPU-MSI-I`	`GPU-MSI`	Speedup on GPU	`CPU-MSI-V`	`CPU-MSI-I`	`GPU-MSI`	Speedup on GPU
50	1211	1092	376	3	2837	7238	500	6
75	1822	1643	377	4	4244	8326	501	8
100	2518	2185	371	6	5622	10944	502	11
125	3117	2768	380	7	6813	13909	504	14
150	3625	3311	371	9	8361	16593	505	17
175	4272	3855	382	10	9931	19081	507	20
200	5022	4437	383	12	11092	21774	508	22