Classification of the difficulty in accelerating problems using GPUs

Submitted in fulfilment
of the requirements of the degree of

Master of Science

of

Rhodes University

Uvedale Roy Tristram

Grahamstown, South Africa
December 2013

Abstract

Scientists continually require additional processing power, as this enables them to compute larger problem sizes, use more complex models and algorithms, and solve problems previously thought computationally impractical. General-purpose computation on graphics processing units (GPGPU) can help in this regard, as there is great potential in using graphics processors to accelerate many scientific models and algorithms. However, some problems are considerably harder to accelerate than others, and it may be challenging for those new to GPGPU to ascertain the difficulty of accelerating a particular problem or seek appropriate optimisation guidance. Through what was learned in the acceleration of a hydrological uncertainty ensemble model, large numbers of k-difference string comparisons, and a radix sort, problem attributes have been identified that can assist in the evaluation of the difficulty in accelerating a problem using GPUs. The identified attributes are inherent parallelism, branch divergence, problem size, required computational parallelism, memory access pattern regularity, data transfer overhead, and thread cooperation. Using these attributes as difficulty indicators, an initial problem difficulty classification framework has been created that aids in GPU acceleration difficulty evaluation. This framework further facilitates directed guidance on suggested optimisations and required knowledge based on problem classification, which has been demonstrated for the aforementioned accelerated problems. It is anticipated that this framework, or a derivative thereof, will prove to be a useful resource for new or novice GPGPU developers in the evaluation of potential problems for GPU acceleration.

ACM Computing Classification System (CCS)

Thesis classification under ACM CCS (1998 version, valid through 2013) [1].

C.1.2 [Processor Architectures]: Multiple Data Stream Architectures—*Single-instruction-stream, multiple-data-stream processors (SIMD)*
F.2.0 [Analysis of Algorithms and Problem Complexity]: General

General Terms: Design, Experimentation, Measurement

Acknowledgements

I give my thanks to everyone who has contributed in some way to the production of this thesis.

I would like to express my sincere gratitude to my supervisor, Dr. Karen Bradshaw, for the tremendous support and guidance she provided throughout this research, whilst still giving me the freedom to find my own way. Her positivity, remarkable dedication, and attention to detail have also been greatly appreciated.

I am thankful to Prof. Denis Hughes for his willingness to assist with my first case study and his input on the related work.

I also thank my family and friends for their continued support and encouragement.

This work was undertaken in the Distributed Multimedia CoE at Rhodes University, with financial support from Telkom SA, Tellabs, Genband, Easttel, Bright Ideas 39, THRIP, and NRF SA (UID 75107). The authors acknowledge that opinions, findings and conclusions or recommendations expressed here are those of the author(s) and that none of the above mentioned sponsors accept liability whatsoever in this regard.

Contents

List of Figures

List of Tables

Listings

List of Algorithms

Chapter 1

Introduction

Scientists continually have a need or desire for additional processing power, as it enables them to compute larger problem sizes, use more complex models and algorithms, and solve problems previously thought computationally impractical. In recent years, the graphics processing unit (GPU) has become a powerful parallel processor that can be used for general-purpose computation, in addition to its traditional function of graphics processing [58]. With the high availability and cost-effectiveness of massively parallel GPUs, these devices have become popular for the acceleration of applications that benefit from such parallelism [48, 58]. This is demonstrated by the Cray XK7 (Titan) supercomputer, featuring 18,688 NVIDIA K20 GPUs [61], placing second in the world for speed[1] and first for power efficiency[2] (as of November 2013). However, while GPUs have been successfully used to increase application performance by an order of magnitude (or even two), such speedups are not possible for all applications [43, 48, 82]. The use of GPUs may even result in a decrease in performance [15]. In addition to possible performance uncertainty, extensive time, effort, and expert knowledge is sometimes required to develop a GPU implementation with acceptable performance [10, 15, 67]. This presents a challenge to scientists who are interested in accelerating their applications using GPUs, but are unsure of the difficulty and performance gain of doing so.

A number of general-purpose computation on graphics processing units (GPGPU) performance modelling and prediction solutions have been proposed that go some way in addressing performance uncertainty [10, 15, 29, 40, 48, 73]. In addition to performance prediction of current program code, several models even provide performance predictions

[1] `http://www.top500.org/lists/2013/11/`
[2] `http://www.green500.org/lists/green201311`

for optimisations that would help eliminate identified bottlenecks [48, 73], which is of particular benefit to novice GPGPU programmers. However, these models do not provide any analysis of the difficulty involved in developing the accelerated solution. Even if a prediction model were able to provide information on possible optimisations to address identified bottlenecks, the developer would still be required to understand and implement them for the target GPU.

1.1 Problem Statement and Research Goals

Given the large variance in the difficulty of accelerating different problems using GPUs, a viable means of estimating the implementation complexity with respect to accelerating existing models or algorithms on GPUs would be highly beneficial to those new to the GPGPU paradigm. A survey of the literature indicates that no formal method for doing this currently exists. Therefore, our objective is to make this possible through the creation of a problem difficulty classification framework.

The creation of such a framework would first require the identification of algorithm attributes that have a significant impact on GPU acceleration difficulty, and an analysis of why these attributes contribute to overall problem difficulty.

The research goals of this thesis are threefold:

1. Identify the attributes of algorithms that have a significant impact on GPU acceleration difficulty through the acceleration of three problems with varying difficulties.
2. Determine the reasons for the identified attributes' contributions to problem difficulty.
3. Construct a problem difficulty classification framework that uses the identified difficulty indicators to determine overall GPU acceleration difficulty.

Henceforth in this thesis, use of the term *difficulty* relates to the difficulty in accelerating an existing CPU program using a GPU. The same interpretation is meant by the phrase *problem difficulty* – the problem referred to is that of accelerating an existing program. We also express *attributes of problem solution* as simply *problem attributes*.

1.2 Thesis Organisation

Chapter 2 introduces parallel computing and describes a number of important parallel programming concepts.

Chapter 3 provides an overview of a modern GPU architecture and GPGPU frameworks, and discusses the performance benefit from GPU acceleration.

Chapter 4 describes the experimental design and methods used in this study.

Chapter 5 gives a detailed account of how a hydrological uncertainty ensemble model can be accelerated using GPUs.

Chapter 6 describes the acceleration of large numbers of k-difference comparisons using two different algorithms.

Chapter 7 discusses the reimplementation of an optimised GPU radix sort and provides a comparison with a naïve solution.

Chapter 8 presents an analysis of the accelerated problems, identifies and analyses the most important problem attributes in determining problem acceleration difficulty, and proposes a problem difficulty classification framework.

Chapter 9 summarises this thesis, presents the conclusions drawn, and identifies possible future work.

Appendix A gives the supporting calculations for difficulty indicator measurements used in Chapter 8.

Chapter 2

Parallel Computing

In traditional computing, a program is executed sequentially on a single processor. This model of computation is highly restrictive since it constrains what can be achieved by a program to the speed of a single processor. The availability of multiprocessor systems has given rise to a new form of computing, *parallel computing*, where computational tasks within a program are executed simultaneously (or in parallel) on multiple processors. This form of computing can result in significantly faster program execution and allows the computation of problem sizes previously thought infeasible. This chapter reviews the history of parallel computing and gives a broad overview of fundamental parallel computing concepts.

2.1 Parallel Architectures

Computer architectures can be broadly classified according to Flynn's taxonomy [25], which describes two types of parallel architectures. An overview of this taxonomy is given, followed by a brief history of parallel architectures with emphasis on CPUs and GPUs.

2.1.1 Flynn's Taxonomy

The different architectures in Flynn's taxonomy are described as computers that operate on streams of instructions and data in different configurations [25].

Single Instruction, Single Data stream (SISD): An entirely sequential computer that has a single processor executing instructions on a single data stream one datum at a time [45]. The first personal computers are an example of this architecture.

Single Instruction, Multiple Data streams (SIMD): A computer in which multiple data streams are acted upon by a single instruction stream simultaneously to perform naturally parallel operations [45]. GPUs and vector processors use this architecture.

Multiple Instruction, Single Data stream (MISD): An architecture in which a single data stream is operated on by multiple instruction streams. This is not really used in practice.

Multiple Instruction, Multiple Data streams (MIMD): Multiple processors operate independently and in parallel, executing different instruction streams on different data [11]. Multi-core CPUs in modern desktop computers use this architecture.

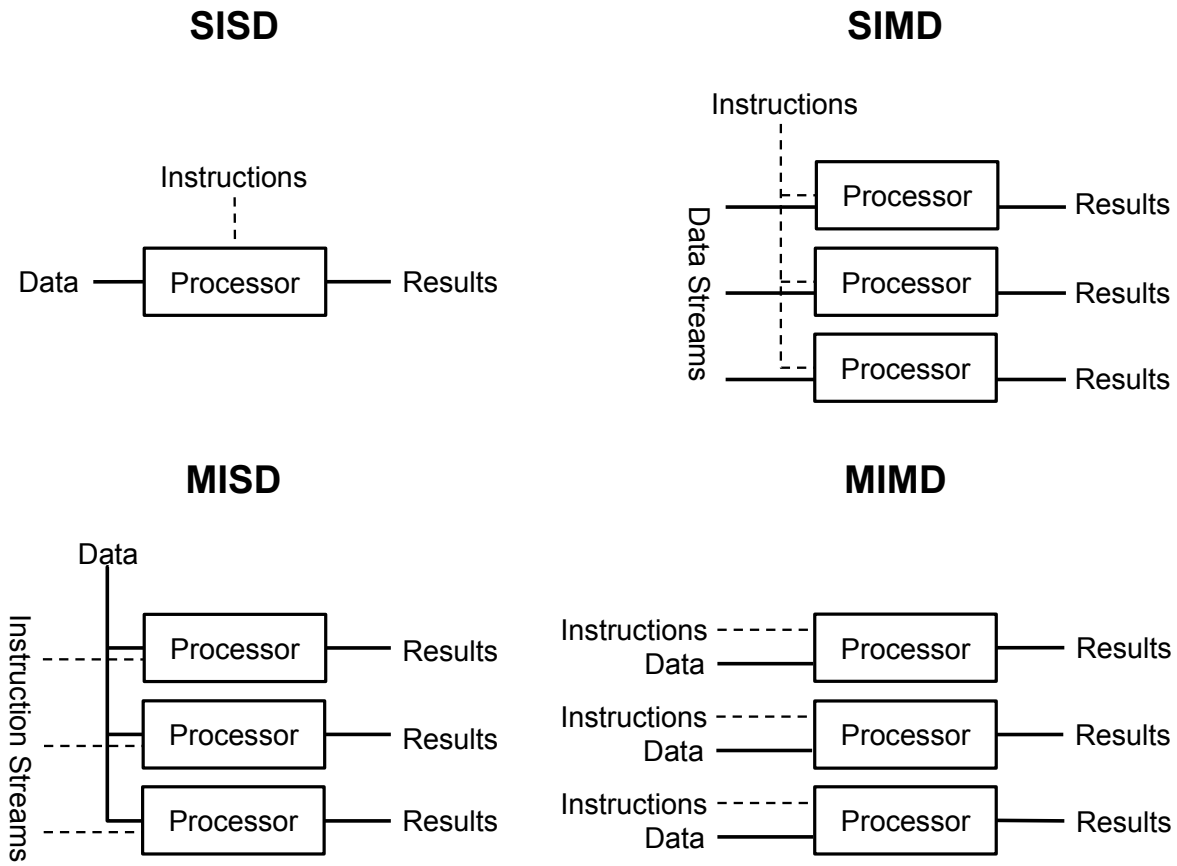


Figure 2.1: An illustration of Flynn's Taxonomy ([25]).

2.1.2 Central Processing Units

Multiprocessor computers with parallel computation capabilities first emerged in the 1960s and were based on the MIMD architecture [22]. The first "true" symmetric multiprocessor system is regarded by some as the Burroughs D-825 (released in 1962) [22]. These early multiprocessor computers did not scale very well; one example cited by Enslow [22] reported a 1.8x performance improvement with two processors, but only a 2.1x improvement with three.

The 1970s and 1980s brought the first SIMD computers in the form of *vector processors*, which operated on a vector of data with a single instruction [45]. The Cray-1 was one of the earliest of these computers [24], and probably the most well-known. However, interest in processors based purely on the SIMD architecture waned as MIMD processors became significantly cheaper, making them a more cost-effective option. Vector processing was later incorporated into conventional processors as an enhancement to reduce instruction fetches on vector operations [37].

Uniprocessor personal computers became prevalent in the 1980s, and the performance of these sequential computers doubled approximately every 18 months as a result of improvements in transistor and silicon technology [9]. While software architects continued to use a sequential programming model, microprocessor manufacturers continued to innovate on sequential performance, even if these innovations were inefficient in terms of transistor and power usage [9]. This trend ended when microprocessor manufacturers were confronted with the *power wall* [9, 47], meaning that single-core frequency improvements could no longer easily be made because of power and heat constraints [65]. The industry was thus forced to consider new computing paradigms to sustain the regular improvements in computing power, and it was concluded that the only way forward was to replace power-inefficient processors with efficient multi-core processors [9].

Multi-core processors brought about a new era in parallel computing. Ordinary desktop computers began to feature powerful multi-core CPUs, which required software developers to embrace parallel programming strategies to make use of the additional processing cores. In modern computing, multi-core CPUs are prevalent. The move to multi-core systems has meant that parallel computing is no longer the domain of a niche community, but something that needs to be considered by almost all application developers who require high performance.

2.1.3 Distributed Computing

Distributed computing can be seen as a loosely coupled form of parallel computing. In this model, the computational workload is distributed between multiple computers, or *nodes*, connected through an interconnection network [45]. There is no shared memory in a distributed system; each node has its own local memory. Consequently, data sharing between nodes must occur by means of passing messages over the interconnection network, commonly known as *message passing* [45]. The loosely coupled nature of distributed computing allows these systems to scale to very large sizes relatively easily [21]. For this reason, high performance computing centres use distributed computing in the form of large compute clusters with fast interconnection networks for maximum performance [14, 20].

2.1.4 Massively Parallel Architectures

CPUs have been designed for general purpose computation and have thus focused on low latency rather than high throughput computation. However, certain problems benefit greatly from high throughput, massively parallel architectures. The need for such architectures has given rise to devices such as graphics processors, massively parallel field programmable gateway arrays (FPGAs), and network flow processors.

Graphics Processing Units

The early graphics accelerators of the 1990s comprised a fixed function pipeline that greatly restricted what could be processed on these devices [58]. As the demand for better graphics grew, these devices became increasingly programmable [58, 62]. The first device to be labelled a GPU was the NVIDIA Geforce 256 launched in 1999 [58]. It featured a configurable integer pixel-fragment pipeline as well as a configurable 32-bit floating point vertex transform and lighting processor [58]. The NVIDIA Geforce 3, launched in 2001, was the first GPU to feature a programmable vertex processor that executed vertex shader programs [58].

GPUs were built to process graphics, and were thus designed to handle applications with the following characteristics [62]:

- *Large computational requirements.* Real-time graphics rendering requires tens of millions of pixels to be updated every second, and each pixel requires hundreds of operations [62]. This requires a substantial amount of processing power.

- *Throughput is more important than latency.* As a result of the human perception system being approximately six orders of magnitude slower than operations within modern processors, a high degree of latency is tolerable [62]. This means that throughput can be prioritised at the cost of latency.

- *Abundant parallelism.* Graphics operations typically require a number of small tasks to be performed on large sets of data in parallel [62], otherwise known as fine-grained data parallelism. With the large number of operations that need to be performed, this creates an abundance of available parallelism for massively parallel architectures.

These characteristics are not only found in graphics processing. Various other applications were identified to have similar characteristics, making them desirable to run on GPUs. With the development of the Cg language for programming GPUs, programmers were finally able to use the power of graphics processors to accelerate other kinds of workloads, marking the beginning of a new field of high performance parallel computation. However, the development of general-purpose programs for these devices was challenging, as it was still a requirement to express non-graphics computations with a graphics application programming interface (API) [58, 69]. This changed in 2006 with the launch of NVIDIA's Geforce 8800, the first GPU to feature a unified graphics and compute architecture. GPGPU programs could then be written in a variation of C with extended syntax using the compute unified device architecture (CUDA) GPU parallel programming API, which greatly simplified the writing of general-purpose programs on the GPU. With the addition of integer arithmetic, load/store memory access instructions with byte addressing, IEEE 754 floating-point arithmetic, thread arrays, shared memory, and fast barrier synchronisations [58, 69], the use of GPUs for general-purpose computations became increasingly viable and alluring. Researchers began to use GPUs to accelerate a range of different applications, publishing hundreds of papers on GPU acceleration [58], thus furthering interest in the field. The two biggest GPU manufacturers, AMD and NVIDIA, now regard the compute capabilities of their GPUs to be an important consideration, and are continuously working toward more compute friendly architectures [2, 60].

Other Parallel Devices

Field Programmable Gate Array: These devices contain a large number of programmable logic blocks and interconnects that enable developers to create custom hardware configurations [17]. With the flexibility to create custom hardware designs, developers

can choose to create massively parallel architectures. However, high performance FPGAs are more expensive than GPUs, and programming the hardware configuration of these devices is costly and labour intensive [17, 74].

Network Flow Processor: There are a number of applications for on-line network flow processing, such as secure socket layer inspection, OpenFlow routing, forensics, and intrusion detection [57]. However, performing live flow processing on networks with line speeds in the tens to hundreds of gigabits per second requires a substantial amount of processing power. Network flow processors achieve this through massively parallel throughput-oriented architectures [57].

2.2 Parallel Programming

Parallel architectures have introduced new challenges to programmers attempting to make the best use of the processing power available to them. This section serves to introduce basic parallel programming concepts, which form the foundation of GPU programming.

2.2.1 Speedup from Parallelism

As a consequence of programs containing inherently sequential code, parallelisation does not scale program performance linearly. Amdahl's and Gustafson's Laws attempt to predict the performance benefit of parallelism in a perfect environment given the portion of the program that is inherently sequential.

Amdahl's Law

Given that a certain fraction of a concurrent program is inherently sequential, it follows that the addition of processors will only reduce the execution time of the parallel section of the program provided the problem size is kept constant. If it is assumed that there is no overhead from parallelisation, the speedup that can be achieved through parallelisation can be found with Eq. (2.1), where the execution time of the best sequential version of the program is t_s, the sequential portion of the program is f, and the number of processors is p.

$$Speedup(p) = \frac{t_s}{ft_s + \frac{(1-f)t_s}{p}} = \frac{p}{1 + (p-1)f} \tag{2.1}$$

Eq. (2.1) is known as *Amdahl's law* [11]. The maximum speedup achievable by a program according to this equation can be found by setting p to ∞.

Gustafson's Law

Amdahl's law makes the assumption that the problem size is constant, while the parallel processing time changes with the number of processors in the system. Gustafson made the argument that the selected problem size typically scales with the number of processors in a system, and the parallel processing time is kept constant [11, 21]. He also made the case that the serial portion of the program does not increase with problem size [11]. These different assumptions resulted in *Gustafson's law*, shown below.

$$Speedup(p) = p + (1 - p)ft_s \tag{2.2}$$

These different assumptions mean that Gustafson's law estimates considerably greater speedups than Amdahl's law. For example, if the serial fraction of a program is $\frac{1}{10}$ and there are 20 processors, the maximum speedup is 18.1x using Gustafson's law and 6.9x using Amdahl's law.

2.2.2 Interprocess Communication

Many parallel programs require communication between the concurrent processes. Shared memory and message passing are the two fundamental methods for achieving this [11, 21].

Shared Memory

Computers with a global memory that is shared by all processors are known as *shared memory systems* [11, 21]. They have a unified address space, which means every memory location has a unique address that is accessible by the processors in the system [11]. The availability of memory that is shared by all processors allows interprocess coordination to take place, and allows for the most general form of MIMD computing [21, 45].

Message Passing

Message passing is typically used in multicomputer parallel computations where a global shared memory is not available [21]. Communication between compute nodes is achieved though message passing on an interconnection network [21]. Unlike shared memory systems, the communication between processes in a message passing system must be explicitly written into the software, which makes it harder to use than shared memory [21, 45]. However, message passing is seen as the only interprocess communication method that scales efficiently with additional processors in a distributed parallel system [21]. To bring the simplicity of shared memory programming to scalable message passing systems, hybrid distributed shared memory systems have been created. These systems use message passing for communication between processes, but this is abstracted away by software and made to look and behave like a shared memory system [21].

2.2.3 Parallel Program Decomposition

The available parallelism in programs can be expressed in a variety of ways. Three of the relevant parallel decompositions are task, data, and instruction parallelism.

Task Parallelism

Task-level parallelism or *function* parallelism results from executing independent, related tasks in parallel [19]. Pipelining is a form of this parallelism, since each task in the pipeline can be executed concurrently. There is usually only a modest amount of task parallelism available in a program, and it does not typically increase much with larger problem sizes [19]. Since different tasks have different computational requirements and scaling characteristics, it can be challenging to load balance task parallel applications efficiently [19].

Data Parallelism

Data or *thread-level parallelism (TLP)* expresses parallelism by executing the same code on multiple threads with different input data [11, 45]. This usually takes place at the statement level of the program, thus making it fine-grained parallelism [45]. This kind of

parallelism scales very well, and is typically found in massively parallel SIMD computers [11, 45], which are very well suited to data parallel computations. In these computers, this form of parallelism is essentially implemented in hardware by executing the same instructions on different data in lock-step [11, 45].

Instruction Parallelism

Instruction-level parallelism (ILP) results from concurrency at the instruction or statement level [11]. For example, statements $c = a + b$ and $d = e \times f$ do not depend on each other and can thus be executed in parallel.

2.2.4 Parallel Programming Terminology

We end this discussion by defining relevant parallel programming terminology.

Process: This is a completely independent program with its own personal memory allocation, variables, and stack [11]. For a process to do any work, it must contain at least one thread [21]. Different processes do not naturally share memory, but memory can be shared between processes through system calls [11].

Thread: Sometimes known as a *lightweight process*, a thread is an independent sequence of execution that resides within a process [21]. A process can manage many threads, and the threads within a process share the memory space and global variables of their parent process [11]. Threads are considerably faster to create than processes, use less resources, and can be synchronised much more efficiently than processes because of their access to shared resources [11]. Threads within a process can be executed concurrently on different processors [21].

Critical Section: A region of code in which a shared resource is accessed that should not be executed by more than one thread concurrently [11].

Semaphore: This is a basic synchronisation mechanism that ensures only a certain number of threads perform a particular action (such as access a particular resource or execute a critical section) simultaneously [8]. On a more fundamental level, it is a non-negative integer variable, which is manipulated by two functions [8]. The first function atomically decreases the value of the semaphore if its value is

greater than zero, and otherwise delays the executing thread until such a time as it can be decreased [8]. The second function atomically increases the value of the semaphore [8]. A semaphore that is initialised to one is commonly known as a *binary* semaphore, while a semaphore that can be initialised to other values is known as a *general* or *counting* semaphore [8].

Barrier: A synchronisation mechanism that prevents concurrent threads from proceeding until all threads (or a specified number) have reached the barrier [11]. Once this condition is met, all the threads at the barrier are awakened and continue execution [11]. This behaviour is useful in applications where threads need to share data at set checkpoints while being in a common state [11].

2.3 Summary

In this chapter, we gave a brief history of parallel processing and described important parallel programming concepts. We started off with a description of the types of parallel architectures according to Flynn's taxonomy, and followed this with a brief history of how the parallel architectures evolved. This led into a discussion of massively parallel architectures such as GPUs and FPGAs. Moving from hardware to software, we described well known equations for estimating the speedup of an application attainable through parallelism, and described methods for interprocess communication. This was followed by a brief description of three types of parallel program decomposition. Finally, we provided a list of pertinent parallel programming terminology and their associated definitions.

Chapter 3

GPU Computing

The massive parallelism and low cost of GPUs has made them appealing accelerators for highly parallel programs. With the support of GPU manufacturers, general-purpose computation on these devices has become an important field in high performance computing. In this chapter, we provide an overview of a modern GPU architecture and GPGPU.

3.1 Modern GPU Architecture

The GPU used in this study is an AMD Radeon HD 7970, which belongs to the Southern Islands series of AMD GPUs [3]. For brevity, this GPU is hereafter referred to as HD7970. This section provides an overview of the architecture of this GPU to give context to discussions on GPU performance. This is split into a discussion of the GPU's processing and memory models.

3.1.1 Processing Model

To provide a clear picture of how processing occurs on the HD7970, the organisation of the processors is first given, followed by an explanation of how work is scheduled on these processors.

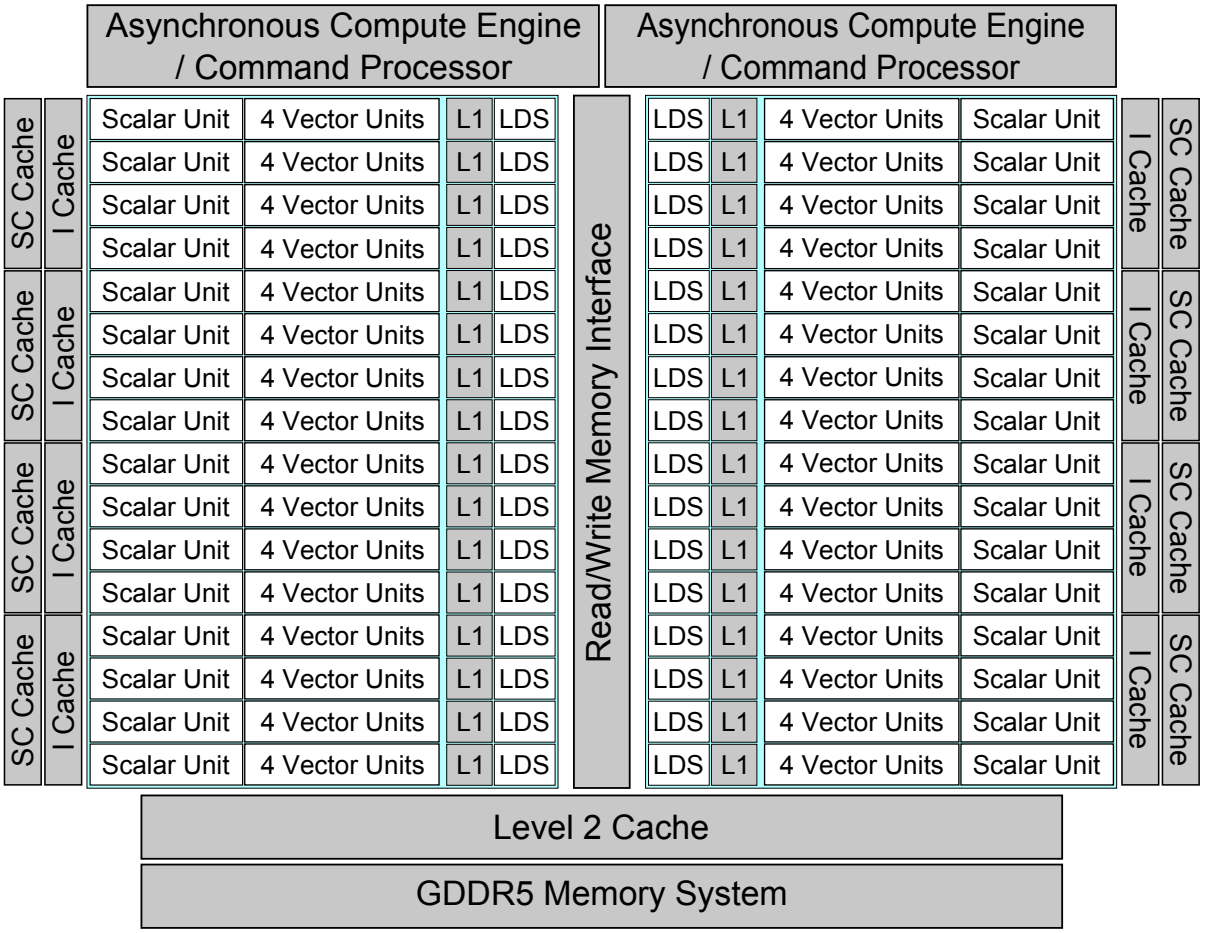


Figure 3.1: A partial block diagram of the AMD Radeon HD79xx architecture that shows the layout of the processing units, memories, and caches (adapted from [3]).

Organisation of Stream Processors

A block diagram of the HD7970 is given in Figure 3.1. The stream processors within the HD7970 are distributed among a number of compute units [2]. Within each compute unit (Figure 3.2), there are four vector (or SIMD) units and one scalar unit. Each vector unit is comprised of 16 stream processors, giving each compute unit a total of 64 stream processors. The purpose of the scalar unit is to handle branch instructions, constant cache accesses, and other scalar operations [3]. The HD7970 is made up of 32 compute units, giving it a total of 2,048 stream processors. These processors are typically clocked at 925 MHz, which gives the HD7970 a theoretical peak performance of 3.79 TFlops/s [3]. It should be noted that the term *stream processor* is used interchangeably with *stream core* and *processing element* in the literature [3]; we will only use stream processor (SP) going forward.

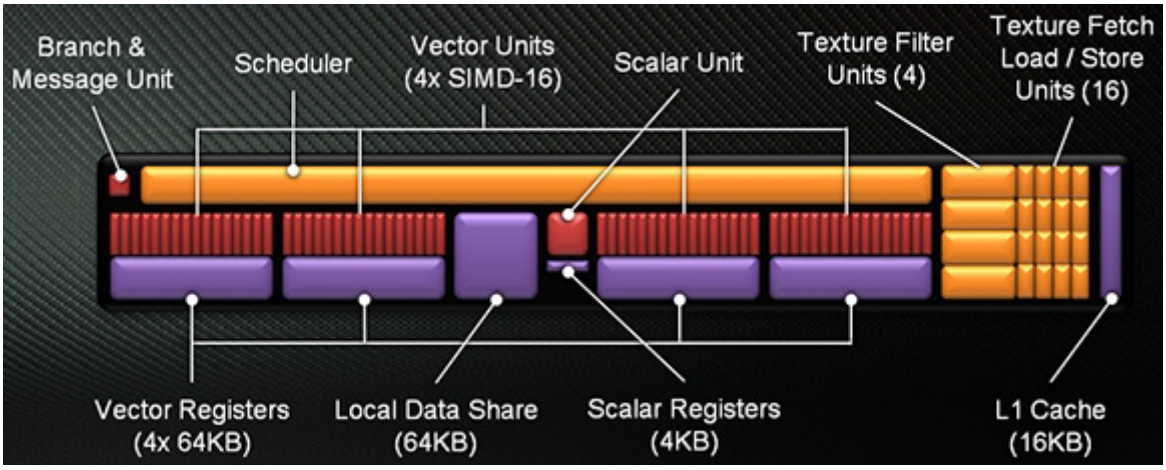


Figure 3.2: The organisation of an AMD Southern Islands GPU compute unit (taken from [83]).

Scheduling

The HD7970 executes work tasks in groups of 64 *work-items* called *wavefronts* [2]. Work-items can be thought of as lightweight threads. Wavefronts are distributed to the available compute units for processing. As is the case with SIMD processing, the stream processors within a vector unit all process the same instruction simultaneously. However, the different vector units within a compute unit are able to process independent instructions, and thus independent wavefronts [2]. It should be noted that SIMD units found on GPUs are different to traditional SIMD units in that they allow the processing elements some level of independent behaviour as well, such as processing separate code branches [59]. This has resulted in the use of a new term for this architecture, namely, single-instruction, multiple thread (SIMT) [59]. Since there are four times as many work-items in a wavefront as there are stream processors in a vector unit, a single instruction in a wavefront is executed over four cycles [3]. Each compute unit is able to schedule 10 wavefronts per vector unit, and therefore 40 wavefronts in total. Vector units are able to swap between these wavefronts as needed [3], which means the HD7970 can process up to 1,280 wavefronts or 81,920 work-items concurrently. Supporting the aforementioned number of wavefronts is contingent on the compute units having sufficient resources (i.e. registers and local memory) for all of them. The limited maximum wavefronts expressed as a percentage of the hardware maximum is known as GPU *occupancy* [3].

Although the stream processors within a vector unit all process the same instructions, different work-items within a wavefront are able to execute different instruction branches. This is achieved by combining all the instruction paths relevant to the wavefront and

processing them serially [3]. To prevent stream processors from executing the instructions of branches not relevant to their corresponding work-items, only the relevant stream processors are enabled through the use of an execution mask [3]. This is illustrated in Figure 3.3; the grey curved lines represent stream processors that have been masked out. In the example given, the condition only applies to a single thread, which means only 12.5% of the SPs are active for that section of the branch. If this was the case for a full sized wavefront of 64 threads, only 1.6% of the SPs would be active. Thus, while divergence within a wavefront is possible, it can result in significantly reduced performance.

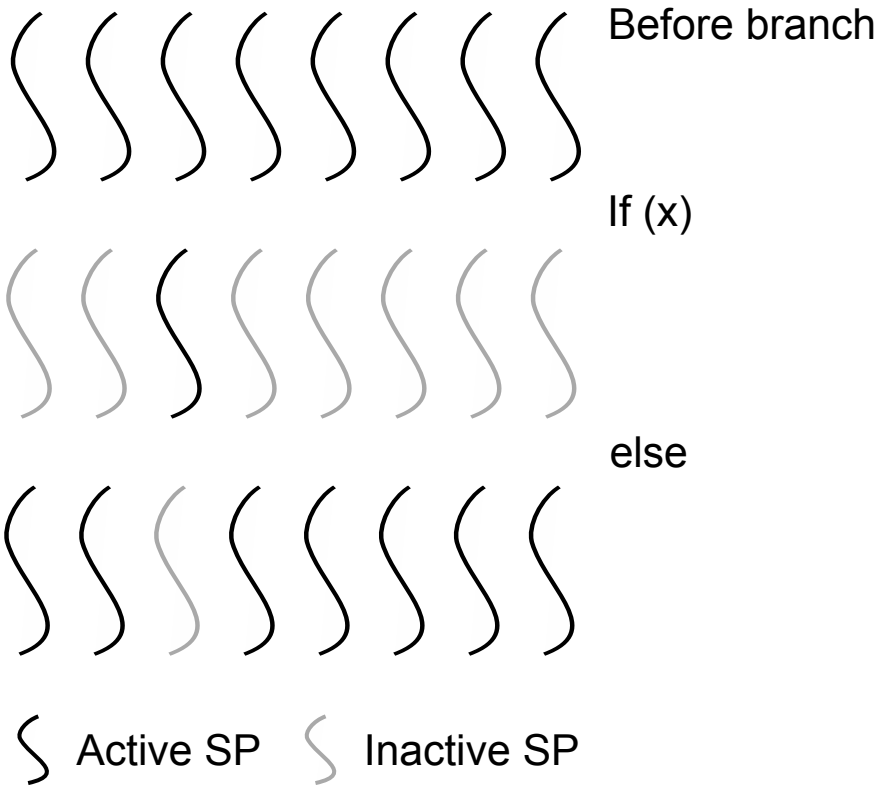


Figure 3.3: An illustration of how branches are executed within a wavefront. The curved lines represent stream processor threads of execution. Only eight threads are shown for simplicity.

3.1.2 GPU Memory Model

The GPU memory hierarchy consists of four memories: register memory, local memory, global memory, and constant memory [3]. These memories and their relationships are depicted in Figure 3.4, and their specifications are given in Table 3.1. The purpose and characteristics of each of the memory regions is given below, followed by a discussion on the L1 and L2 caches.

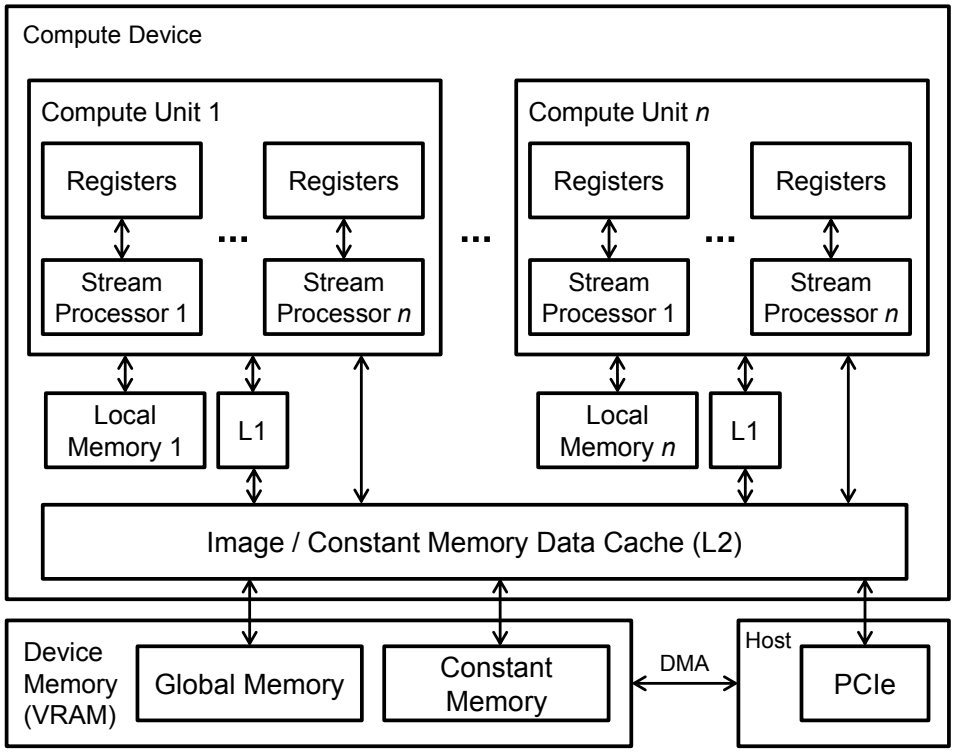


Figure 3.4: The memory hierarchy of Southern Islands devices (adapted from [3]).

Registers

Each compute unit has a number of vector and scalar general-purpose registers (VGPRs and SGPRs) that can be used by the scheduled wavefronts [3]. The VGPRs are distributed between the four vector units, and are different from SGPRs in that they are replicated for each of the stream processors. The SGPRs are typically used to store data that is common to an entire wavefront, such as constant data and the execution mask [3].

Registers act as temporary private memory for the work-items, and are thus also referred to as *private memory* [3]. The throughput of registers is 6x greater than any other kind of memory on the GPU, which means efficient use of them is crucial to achieving high performance for most applications. The number of registers usable by each work-item changes depending on the number of wavefronts scheduled on the compute units (i.e. occupancy). The HD7970 has 256 KB of VGPR register space per compute unit, which is 65,536 32-bit registers. If the maximum of 40 wavefronts are scheduled on each compute unit, this only permits each work-item to use ~25 32-bit registers. Higher register use thus typically comes at the cost of decreased TLP, but this is often a worthwhile trade-off [80]. When register use is excessive and exceeds the available register space, external global memory must be used instead. This is commonly referred to as “register spilling”, and

Table 3.1: Tahiti memory specifications (sourced from [3]).

Memory	**Size**	**Peak Read Bandwidth**	**Peak Read Bandwith / Stream Core**
L1 Cache	16 KB/CU	1.9 TB/s	1 byte/cycle
L2 Cache	768 KB	710 GB/s	~0.4bytes/cycle
Registers (VGPR)	256 KB/CU	22.7 TB/s	12 bytes/cycle
Local Memory	64 KB/CU	3.8 TB/s	8 bytes/cycle
Global Memory	3 GB	264 GB/s	~0.14 bytes/cycle

can have a substantial negative impact on performance given the order(s) of magnitude differences between global memory and register memory bandwidth and access latency [3].

Local Memory

Like register memory, each compute unit has its own local data share (LDS), more commonly referred to as local memory [2, 3]. However, unlike register memory, it is shared rather than private. This facilitates thread cooperation between work-items by allowing them to write to and read from a common memory area. Such thread cooperation is also possible with global memory, but local memory is over an order of magnitude faster (see Table 3.1), making it preferable for such operations.

The compute unit on an HD7970 has 32 banks of local memory storage, each containing 512 32-bit entries, that can each serve one request per cycle [2]. The LDS typically serves the requests from two different vector units per cycle, which means it is important to ensure that the local memory requests from a quarter wavefront correspond to different memory banks [2, 3]. This can be achieved through the use of a simple 4-byte linear access pattern. When LDS bank conflicts do arise, they are serialised and serviced over consecutive cycles, thus significantly reducing local memory throughput [3]. One noteworthy exception is when all requests access the same bank – this results in a broadcast of the requested data that carries no penalty [3].

Global Memory

The HD7970 has 3 GB of global memory, making it by far the largest area of memory on the GPU. As illustrated in Figure 3.4, it is located off-chip, unlike the previous two memories. Consequently, it has a significantly higher memory access latency of between 400 to 600 cycles, as well as over an order of magnitude lower memory bandwidth [3]. Since global memory is used to store input and output data, its high access latency and low throughput per stream processor can be a significant bottleneck for many applications. It is thus crucial to design memory access patterns that utilise all of the available bandwidth.

Access to global memory is facilitated through 12 memory channels [3]. To maximise throughput, memory access patterns should be designed to minimise channel conflicts. This can be achieved by designing efficient memory stride patterns, where a memory stride is, "the increment in memory address, measured in elements, between successive elements fetched or stored by consecutive work-items in a kernel [GPU program]" [3]. A one-unit memory stride, or *coalesced* access pattern, minimises channel conflicts on the HD7970 [3], and usually provides the best performance for GPUs in general.

Constant Memory

Rather than having its own memory area, constant memory resides within the same physical memory as global memory [3]. It differs from global memory in that it is read-only and usually benefits from caching [3]. This makes it ideal for read-only data that is common to all the work-items.

L1 Cache

Each compute unit has 16 KB of read/write L1 data cache that operates on a least recently used replacement policy [2]. The cache lines are 64 bytes long, which means that coalesced memory requests benefit the most from cache hits [2]. Cache misses are sent back to the L2 cache [2]. Global memory writes are written through the L1 cache, and are also eventually written back to the L2 cache when all wavefront stores have completed [2].

L2 Cache

The L2 cache is common to all compute units, and acts as a central point of coherency for the GPU [2]. Like the L1 data cache, it is read/write, uses 64 byte cache lines, and

operates on a least recently used replacement policy [2]. The cache size totals 768 KB on the HD7970, but this is physically partitioned between the six memory controllers and coupled with each memory channel [2].

Host-GPU Transfers

The transfer of data between the host and the GPU (global memory) takes place over the PCI Express bus. The HD7970 supports PCI Express 3.0, which allows it to transfer up to 16 GB/s to and from the host simultaneously [3]. This is over an order of magnitude slower than global memory, and the transfer speed can be significantly lower if the host does not support PCI Express 3.0, or does not have sufficient host memory bandwidth to saturate the PCI Express bus. As a result, transfers between the host and the GPU can be a significant performance bottleneck for bandwidth intensive applications [27].

3.2 GPGPU Programming Frameworks

A number of GPGPU frameworks have been developed to simply the creation of GPU targeted general-purpose applications. These frameworks provide useful hardware abstractions and a familiar programming environment that enable developers to spend more time focusing on an efficient parallel decomposition for a problem [39, 59]. We briefly discuss two of these, CUDA and C++ Accelerated Massive Parallelism (AMP), before giving a more detailed overview of OpenCL, which was the GPGPU framework used in this study. To show the differences between the programming APIs, a naïve single-precision alpha X plus Y (SAXPY) example is also given in each section.

3.2.1 CUDA

CUDA is a GPU architecture developed by NVIDIA that includes several components designed specifically for GPU computing [59]. The general aim of CUDA is to make general-purpose computation more practical on graphics processors whilst still giving the programmer access to low-level features [59, 69]. The launch of CUDA and CUDA C was a milestone for GPU computing; they no longer required users to express GPGPU problems as graphics problems, and gave users access to specialised GPU features [69]. Furthermore, CUDA C is based on the familiar C language with added extensions to

take advantage of the CUDA architecture [69]. An example CUDA program is given in Listing 3.1 to show the level of abstraction provided by the programming API.

The CUDA C language and compiler have been designed specifically for NVIDIA's CUDA hardware [59]. This means that CUDA applications do not run natively on GPUs created by other manufacturers, or any other kind of parallel processor. Although this enables the programming API to closely match the targeted hardware, this is a significant restriction given the availability of affordable high-performance GPU hardware from other manufacturers, and is the reason we opted not to use this framework. However, there have been efforts to increase the portability of CUDA programs through just-in-time translation[1] and compilation of the kernels into other instruction sets[2].

CUDA introduced a number of terms to GPU programming that have become pervasive in the discussion of GPGPU problems. Some of these are used in the discussion of GPU solutions from other authors; we thus provide a brief description of the common terms:

Thread Block: A one-to-three dimensional grid of threads [59].

Grid: A one-to-three dimensional grid made up of a number of thread blocks [59].

Streaming Multiprocessor: A processing unit that is able to execute multiple thread blocks concurrently, and through the use of a SIMT architecture, can also execute the threads within a thread block concurrently [59].

Warp: A group of 32 threads that are executed together on a streaming multiprocessor [59].

3.2.2 C++ AMP

Unlike CUDA and OpenCL, C++ AMP is a relatively high-level GPGPU capable language. It aims to simplify the task of writing programs that execute on data-parallel hardware (such as GPUs) through abstraction [51]. Programs are predominantly written in standard Microsoft Visual C++ with a few added keywords, but the data-parallel sections of code have added restrictions [51]. The comparative simplicity of developing GPU programs using this framework is clearly seen when comparing the example C++ AMP program in Listing 3.2 with the examples of the other frameworks in Listings 3.1 and 3.3.

[1] https://code.google.com/p/gpuocelot/
[2] http://www.pgroup.com/resources/cuda-x86.htm

```
#include <cuda.h>

__global__ void saxpy(float *a, float *x, float *y, int N) {
  int id = blockIdx.x * blockDim.x + threadIdx.x;
  if (id < N) y[id] = a[id]*x[id] + y[id];
}

int main(void) {
  float *a_h, *a_d, *x_h, *x_d, *y_h, *y_d;
  const int N = 1 << 18;
  a_h = (float *)malloc(N * sizeof(float));
  cudaMalloc((void **) &a_d, N * sizeof(float));
  x_h = (float *)malloc(N * sizeof(float));
  cudaMalloc((void **) &x_d, N * sizeof(float));
  y_h = (float *)malloc(N * sizeof(float));
  cudaMalloc((void **) &y_d, N * sizeof(float));

  for (int i = 0; i < N; i++) { a_h[i] = i; x_h[i] = i; y_h[i] = i; }
  cudaMemcpy(a_d, a_h, N * sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(x_d, x_h, N * sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(y_d, y_h, N * sizeof(float), cudaMemcpyHostToDevice);

  int block_size = 256;
  int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
  saxpy <<< n_blocks, block_size >>> (a_d, x_d, y_d, N);

  cudaMemcpy(y_h, y_d, sizeof(float)*N, cudaMemcpyDeviceToHost);

  free(a_h); cudaFree(a_d);
  free(x_h); cudaFree(x_d);
  free(y_h); cudaFree(y_d);
}
```

Listing 3.1: Naïve CUDA SAXPY program.

Applications written with C++ AMP are runnable on any DirectX 11 or later hardware [51]. While development of GPGPU applications using C++ AMP is typically simpler than using CUDA or OpenCL, it requires the use of proprietary software[3]. Initial empirical performance testing also revealed it to have lower performance than OpenCL (similar to the results of others[4]).

[3] Microsoft Windows 7 or later and Microsoft Visual Studio 2012 or later

[4] `http://codinggorilla.domemtech.com/?p=1135`

```
#include <amp.h>

using namespace concurrency;

int main()
{
  const int N = 1 << 18;
  auto ext = extent<1>(N).tile<256>();
  std::vector<float> a_h, x_h, y_h;
  a_h.resize(N);
  x_h.resize(N);
  y_h.resize(N);
  array_view<float, 1> a(ext, a_h), x(ext, x_h), y(ext, y_h);

  for (int i = 0; i < N; i++) { a[i] = i; x[i] = i; y[i] = i; }

  parallel_for_each(ext, [=](index<1> idx) restrict(amp)
  {
    if (idx[0] < N) y[idx] = a[idx] * x[idx] + y[idx];
  });
  y.synchronize();
}
```

Listing 3.2: Naïve C++ AMP SAXPY program.

3.2.3 OpenCL

OpenCL is an open, royalty-free standard developed by the Khronos Group, aimed at providing a single platform for parallel computation across heterogeneous computation devices, such as CPUs, GPUs, and other parallel processors [39]. The original OpenCL specification was released in 2008 [64], making it over five years old. In this time, the standard has gained a vast amount of support from both users and leading industry firms; the OpenCL working group members list includes the likes of Apple, Intel, AMD, NVIDIA, and IBM [64]. Since OpenCL is a standard, the burden is placed on parallel processor vendors to write compatible OpenCL compilers and runtimes. There are presently OpenCL compilers and drivers for mainstream vendors such as Intel[5], AMD, and NVIDIA [3, 59].

The flexibility of OpenCL is made possible by a number of model abstractions. These are the platform model, execution model, memory model, and programming model.

[5] http://software.intel.com/en-us/vcsource/tools/opencl-sdk

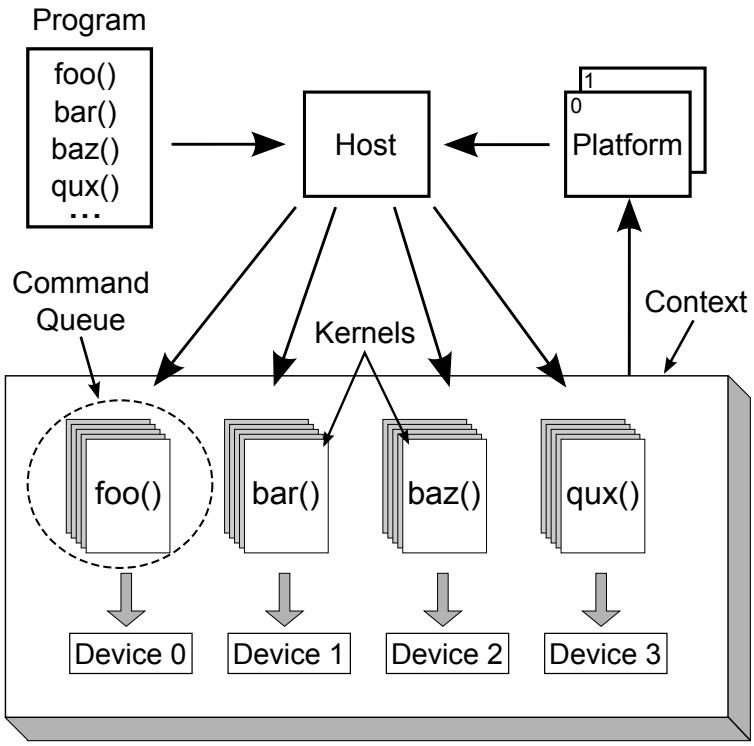


Figure 3.5: The execution of kernels on a number of OpenCL devices (adapted from [72]).

Platform Model

The platform model is a hardware abstraction. It specifies that there is a host that manages OpenCL execution on a number of OpenCL capable *devices*, such as CPUs and GPUs [26, 39]. Devices within a platform are made up of a number of compute units, which are in turn made up of a number of processing elements (i.e. stream processors in the HD7970). A host can have multiple platforms available; the different platforms typically support devices related to a particular hardware vendor (e.g. AMD and NVIDIA). Once a particular platform has been selected, an OpenCL *context* must be created to manage the related resources. This includes the set of devices, the device accessible memory and corresponding memory properties, and one or more *command queues* [39]. Command queues are the mechanism OpenCL uses to schedule commands to be executed on a specific device [39].

Execution Model

OpenCL code is executed by enqueuing *kernels* on an OpenCL device using a command queue [39]. Kernels are essentially functions that act as entry points into an OpenCL program. If multiple devices are available, multiple kernels can be launched simultaneously

using different command queues, as illustrated in Figure 3.5. To schedule a kernel for execution, the kernel's *NDRange* must be specified, which is simply an index range that can have between one and three dimensions [39]. Each index in the NDRange corresponds to a unique thread of execution to be scheduled on the OpenCL device. OpenCL divides the index space into groups known as *work-groups*, which are groups of *work-items* guaranteed to be executed together on the same compute unit in a series of wavefronts [39]. An illustration of an OpenCL NDRange is provided in Figure 3.6. This division of work-items into groups allows for divergent code to be executed efficiently by different work-groups, otherwise known as single program, multiple data execution (SPMD) [39]. A kernel is given access to data by specifying OpenCL buffers, images, or primitive variables as kernel arguments [39]. OpenCL memory objects can be created to reserve memory on the host or OpenCL device, or point to existing host memory [39]. When a kernel is run, each OpenCL thread executes exactly the same kernel program, with the only difference being its index in the NDRange. This index is used to differentiate thread behaviour, such as selection of input data.

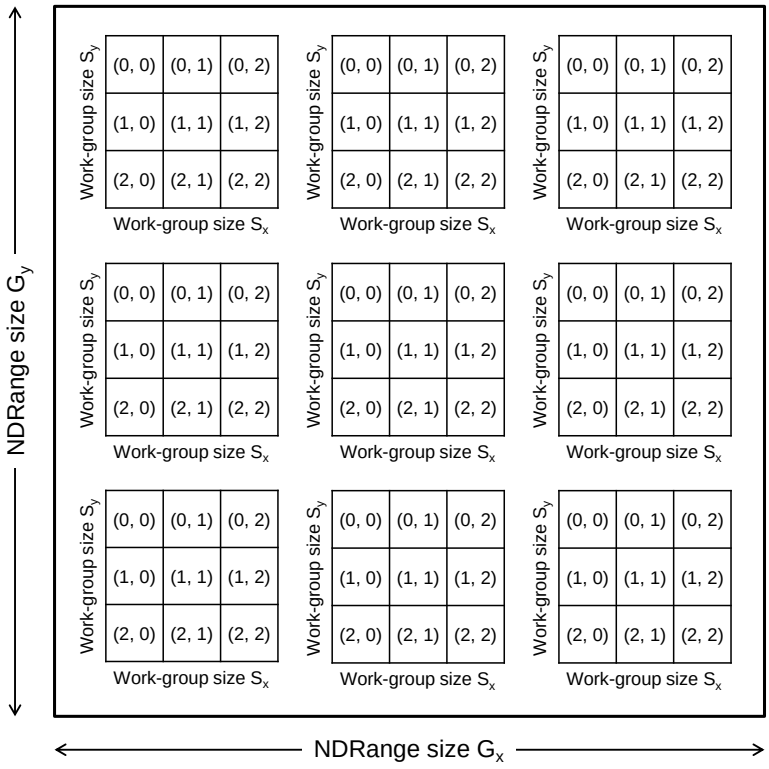


Figure 3.6: An example of what a 2-dimensional NDRange looks like, and how it maps to work-groups containing work-items (adapted from [39]).

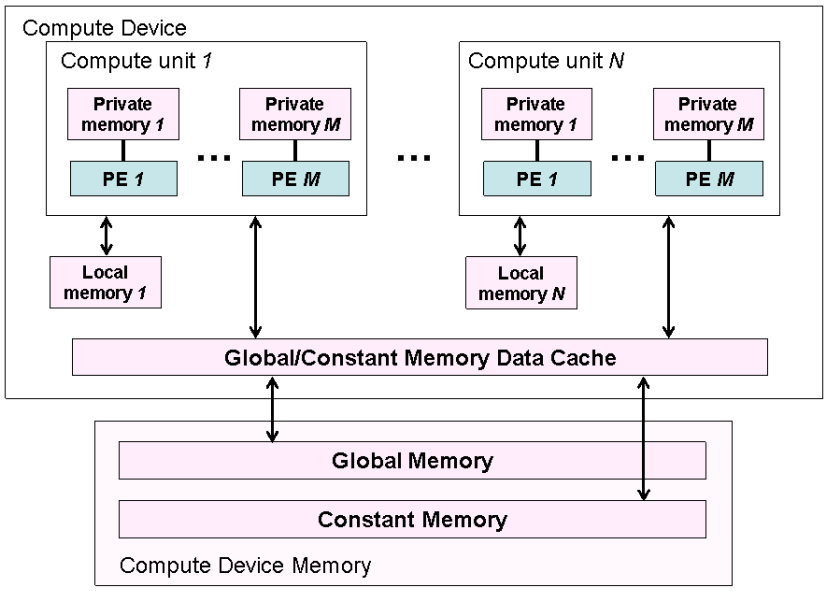


Figure 3.7: The OpenCL memory hierarchy (taken from [39]).

Memory Model

The OpenCL memory model, illustrated in Figure 3.7, maps closely to the memory model of the HD7970. The largest area of memory available is *global memory*, which is accessible by all processing elements. The next area of memory, *constant memory*, is similar to global memory, except it is guaranteed to stay constant during kernel execution, and is initialised by the host prior to kernel execution. *Local memory* is the next level of memory, and is distributed equally among the work-groups to facilitate data sharing within work-groups. Lastly, each work-item has its own *private memory*, which is often the smallest and fastest memory available. Although the local and global memories can be used for data sharing within a work-group by using synchronisation barriers, memory consistency is never guaranteed between work-groups.

Programming Model

The language used for writing OpenCL programs is a variation of the C99 specification, with added extensions for parallelism [39]. OpenCL programs can be written in a data-parallel style, task-parallel style, or a combination of the two. The data-parallel style, which is the most commonly used approach, expresses parallelism by executing the same code on multiple threads with different input data. In the task-parallel style, parallelism

is expressed by running different “tasks”, or OpenCL kernels, in parallel, and using vector data types [39]. An example of an OpenCL program is given in Listing 3.3.

```
#include "CL/cl.hpp"

using namespace cl;

static std::string CLCode =
"__kernel void saxpy(__global float *a, __global float *x,          \
                     __global float *y, int N) {                    \
  size_t id = get_global_id(0);                                     \
  if (id < N) y[id] = a[id] * x[id] + y[id];                        \
}";

int main(int argc, char **argv) {
  std::vector<Platform> platforms;
  Platform::get(&platforms);
  cl_context_properties cps[3] = {CL_CONTEXT_PLATFORM, (
    cl_context_properties)(platforms[0])(), 0} ;
  cl::Context context = Context(CL_DEVICE_TYPE_GPU, cps);
  std::vector<Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();
  CommandQueue queue = CommandQueue(context, devices[0],
    CL_QUEUE_PROFILING_ENABLE);
  Program::Sources source(1, std::make_pair(CLCode.c_str(), CLCode.length()
    +1));
  Program program = Program(context, source);
  program.build(devices);

  float *a_h, *x_h, *y_h;
  Buffer a_d, x_d, y_d;
  const int N = 1 << 18;
  a_h = (float *)malloc(N * sizeof(float));
  a_d = Buffer(context, CL_MEM_READ_ONLY, N * sizeof(float));
  x_h = (float *)malloc(N * sizeof(float));
  x_d = Buffer(context, CL_MEM_READ_ONLY, N * sizeof(float));
  y_h = (float *)malloc(N * sizeof(float));
  y_d = Buffer(context, CL_MEM_READ_WRITE, N * sizeof(float));

  for (int i = 0; i < N; i++) { a_h[i] = i; x_h[i] = i; y_h[i] = i; }

  queue.enqueueWriteBuffer(a_d, CL_TRUE, 0, N * sizeof(float), a_h);
  queue.enqueueWriteBuffer(x_d, CL_TRUE, 0, N * sizeof(float), x_h);
  queue.enqueueWriteBuffer(y_d, CL_TRUE, 0, N * sizeof(float), y_h);
```

```
38
39    Kernel clSAXPY = Kernel(program, "saxpy");
40    clSAXPY.setArg(0, a_d);
41    clSAXPY.setArg(1, x_d);
42    clSAXPY.setArg(2, y_d);
43    clSAXPY.setArg(3, N);
44
45    queue.enqueueNDRangeKernel(clSAXPY, NullRange, NDRange(N), NullRange);
46    queue.finish();
47    queue.enqueueReadBuffer(y_d, CL_TRUE, 0, N*sizeof(float), y_h);
48
49    free(a_h); free(x_h); free(y_h);
50  }
```

Listing 3.3: Naïve OpenCL SAXPY program.

3.3 The GPGPU Performance Myth

A number of academic papers have reported two orders of magnitude program speedup achieved through GPU acceleration [23, 46, 66]. Performance improvements of this magnitude have created the impression that orders of magnitude speedups are the status quo for GPU acceleration, and anything less is not particularly impressive. In response to this, Lee et al. [43] from Intel Corporation authored a paper, titled "Debunking the 100X GPU vs. CPU myth: An Evaluation of Throughput Computing on CPU and GPU" [43], in which they challenged the credibility of the orders of magnitude speedups reported through GPU acceleration. They did this by benchmarking commonly used throughput-oriented applications that had previously been accelerated on GPUs with massive speedups, ensuring that they had carefully optimised *both* the CPU and GPU implementations. The benchmarked applications included GPU kernels for SGEMM, Monte Carlo simulation, convolution, SAXPY, SpMV, sort, search, histogram, and ray casting. Their results revealed significantly lower speedups through GPU acceleration than previous authors. They made the argument that the CPU implementations of algorithms to which the GPU implementations are compared are often not adequately optimised, and through optimising data-level parallelism and thread-level parallelism, the GPU's speedup over the CPU is substantially reduced to ~2.5x on average [43]. Lee et al. [43]'s paper has been criticised for comparing a previous generation 65nm GPU to a current generation 45nm CPU (released 16 months apart), omitting information such as die size and power consumption,

and for not providing enough detail about the datasets or implementations of the algorithms tested [7]. Nevertheless, the core conclusion of the paper would still stand even if the GPU was of the same generation as the CPU, since the average speedup of the GPU would still be under an order of magnitude faster than the CPU.

Three other research groups have made notable contributions to this discussion. Vuduc et al. [82] arrived at similar results to Lee et al. [43] regarding what should be realistically expected from GPU acceleration, and note that it is important to consider more realistic application contexts where there is a mix of irregular and regular computations. Gregg and Hazelwood [27] highlight the importance of considering the location of program data when performing GPU vs. CPU comparisons, and contend that many GPU performance results are misleading for disregarding this information. They point out that the results of GPU computations are only useful if they are further utilised. If not utilised by another GPU kernel, the results must be fetched from the GPU, which is a step that is necessary to include in benchmarking for more accurate speedup reports. In some of their own tests, they found that the memory-transfer overhead of copying the data between the GPU and the host system added a significant amount of time to most of the applications they benchmarked. In a few cases, the combined memory-transfer and computation time was 50x that of the GPU processing time.

Lastly, Anderson et al. [7] discuss the difficulties of cross platform comparisons and identify two divergent, yet valid, viewpoints of conducting performance comparisons. The first viewpoint is that of the application developers, who have the goal of advancing their applications within the bounds of certain constraints, such as power consumption, cost, developer time, etc. The GPU performance results from application developers are likely the result of comparing the newly developed GPU implementation to the original CPU implementation, which may be sequential and lacking a similar degree of optimisation. As such, these results should not be interpreted as direct comparisons of the GPU and CPU architectures, but rather as the result of the developer effort involved in taking the application from performance level x to performance level y [7]. The results of these comparisons are particularly meaningful to other application developers in the field, and should be viewed strictly from the context in which the speedup claims are made. The second viewpoint is that of the architecture researchers, who are interested in the comparative performance of different architectures for a wide variety of applications spanning multiple domains [7]. To provide meaningful results and conclusions, comparisons of this kind need to consider a broader range of architectural features, such as die size, logic implementation, silicon process used, and power consumption figures [7].

In summary, it has been established that GPUs do not outperform CPUs by two orders of magnitude for a range of common applications run on GPUs, or even an order of magnitude in many cases. Given the diverse nature of these applications, it can be induced that these findings apply more broadly to GPGPU in general. However, this does not mean that reports of massive speedups achieved from GPU acceleration from application developers should be discounted; they should rather be interpreted as the result of developer effort in parallelising an existing application using GPUs.

3.4 Existing GPU Kernel Classification

A characteristic of GPU kernels that has been demonstrated to be highly relevant when considering overall GPU performance is the data transfer requirements to and from the GPU [15, 27]. To help identify and classify the different transfer requirements of different problems, Gregg and Hazelwood [27] created a taxonomy for memory overhead, which is reproduced below.

1. **Non-Dependent (ND):** Kernels that do not require data transfer to or from the GPU, or the data transfer is negligible (e.g. single integer input or output).
2. **Dependent-Streaming (SD):** Kernels that do require data transfer to or from the GPU, but this overhead is hidden with asynchronous streaming memory.
3. **Single-Dependent-Host-to-Device (SDH2D):** Kernels that require data transfer to the GPU.
4. **Single-Dependent-Device-to-Host (SDD2H):** Kernels that require data transfer from the GPU.
5. **Dual-Dependent (DD):** Kernels that require data transfer to *and* from the GPU.

A kernel may fall into multiple categories from this taxonomy depending on its use case. For example, using a GPU sort kernel to order data sent from the host would result in the kernel being classified as dual-dependent. However, if the data to be sorted was already on the GPU as the output from another operation, it would fall into single-dependent-device-to-host category. Further still, it could be classified as dependent-streaming if multiple sorts are required. From a performance perspective, the taxonomy is numbered from least to most performance impact.

3.5 Barriers to Entry

There are a number of barriers to entry into the field of GPU computing which can be problematic for many scientists.

Required architectural knowledge: Unlike CPU programming, GPU programmers are typically required to have a deeper understanding of the underlying architecture to develop their solutions.

Massive parallelism: It can be difficult for some programmers to come to terms with the massive parallelism of GPUs and how to write programs that use it effectively.

Debugging: The debugging tools for GPUs are not as mature and feature rich as those for the CPU. Coupled with the massive parallelism, this can make GPU code debugging challenging.

Problem dependent: The architecture of GPUs is not well suited to all kinds of parallel problems (such as tree and graph problems [82]). It is not always clear how beneficial the GPU acceleration of a particular problem will be, and it may be necessary to use performance projection tools to determine whether the likely speedup is worth the cost of development.

3.6 Summary

This chapter contextualised GPU computing by giving an overview of a modern GPU architecture, relevant GPGPU frameworks, and expected GPU performance. The architecture of the HD7970 was discussed, which included an overview of the organisation of its stream processors, the way in which work is scheduled, and its memory hierarchy. The CUDA and C++ AMP GPGPU frameworks were briefly described to contrast them with the GPGPU framework used in this study, OpenCL. CUDA is the prevailing framework and provides users with a relatively simple programming API, whilst still giving access to low-level functionality. However, the native use of CUDA is restricted to NVIDIA GPUs. C++ AMP greatly simplifies GPGPU development, but this comes at the cost of platform dependence and lower performance. OpenCL is similar to CUDA in the level of abstraction it provides, but is considerably more flexible with regard to the hardware on which it can run.

The large speedups claimed by application developers though GPU acceleration has been subject to some debate. Through extensive testing of well-known problems, it was found that the GPU speedups of these problems were significantly lower than advertised by previous studies as a result of more balanced comparisons where the CPU implementation is also optimised and data transfer overhead is included in benchmarking. It has been suggested that the speedup from unbalanced comparisons between CPUs and GPUs should be better contextualised. Finally, a GPU kernel classification framework was described that provides a simple way to categorise the data transfer requirements of a GPU kernel, and the typical barriers of entry into GPGPU were listed.

Chapter 4

Experimental Design and Methods

The first goal of this research was to identify problem attributes that can be evaluated to determine the overall problem difficulty. This chapter outlines how the experiments were designed to achieve this goal by describing the methods for GPU problem selection and problem acceleration. We include details about performance testing and identification of performance bottlenecks, as well as a listing of the tools used in this study.

4.1 GPU Problem Selection

The GPU problems accelerated in this study were selected for their perceived benefit from GPU acceleration, perceived acceleration difficulty, and lack of a freely available GPU solution with good performance. The final radix sort problem is an exception with regard to the lack of an existing GPU solution, as it was based on an efficient CUDA solution. This was simply because an algorithm of similar GPU acceleration difficulty would have been too time consuming to accelerate without prior work.

The first problem (Chapter 5) was selected for the perceived ease with which it could be accelerated on a GPU, the second (Chapter 6) for its perceived moderate acceleration difficulty, and the third (Chapter 7) for its perceived high acceleration difficulty. Since the selection of the moderate and hard difficulty problems followed the acceleration of the previous problem, the experience gained in GPGPU simplified the identification of problem characteristics that increased acceleration difficulty. Problems of increasing difficulty were selected to demonstrate the differences between hard and simple problems, and to enable testing of the resulting classification system's ability to distinguish problems of different difficulties.

4.2 GPU Problem Acceleration

The approach to the acceleration of the first two problems is broadly depicted in Figure 4.1. The CPU code was first converted to OpenCL without making any changes to improve GPU performance. This was to measure any speedup obtainable from GPU acceleration with a minimal amount of effort and applied GPGPU knowledge. Performance bottlenecks were then identified, followed by the implementation of optimisations in attempt to address some of these bottlenecks. The program was then re-benchmarked and the process repeated until a satisfactory speedup was obtained.

This approach was selected as it is likely the simplest and fastest method for obtaining a satisfactory speedup, presuming this is possible without significant code restructuring. This is in recognition of the fact that practitioners interested in GPU acceleration often do not have the time or expertise to completely redesign an existing program to complement the GPU architecture [55]. Once the required optimisations had been implemented, the solution was benchmarked on other criteria, such as its performance with different problem sizes and the influence of host-to-GPU data transfer time on overall performance.

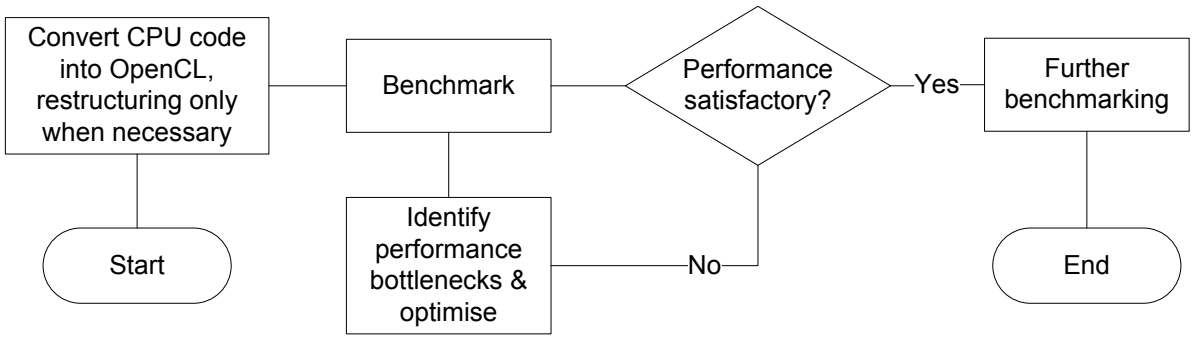


Figure 4.1: The broad approach to GPU acceleration for the first two case studies.

The final problem, radix sorting, warranted a different approach since the problem's difficulty would have prohibited a worthwhile solution within a reasonable amount of time for this study. Instead of creating a new GPU solution, a naïve and a highly optimised solution were reimplemented in OpenCL from other GPGPU languages and compared. The comparison provided similar insights into the optimisations and knowledge required to arrive at a solution with satisfactory performance.

4.2.1 Toolchain

The set of programming tools that are used in the development of software is known as the *toolchain*. The tools that comprised the toolchain for problem acceleration and evaluation are described below.

Microsoft Visual Studio

Microsoft Visual Studio is an integrated development environment for application development. Visual Studio 2010[1] was used for the creation of the GPU programs, and Visual Studio 2012[2] was used to create the CPU versions of the radix sort for Chapter 7. The streamlined integration of both the AMD and Intel OpenCL debugger made it an appealing integrated developer environment for OpenCL programming.

AMD CodeXL

CodeXL[3] is a suite of tools aimed at assisting OpenCL program development on CPUs, GPUs, and APUs. It consists of tools to aid in CPU profiling, GPU profiling, GPU debugging, and static OpenCL kernel analysis [4].

The GPU profiling tool, a facet of which is shown in Figure 4.2, extracts useful performance information from the execution of GPU programs. The output it provides includes information such as GPU occupancy, register usage, local and global memory usage, and stream processor usage [5].

The GPU debugging tool makes it possible to perform online OpenCL kernel debugging from any of the active GPU threads, which was used to identify problems in the GPU kernels. The debugger includes the ability to view the contents of global memory objects.

The static OpenCL kernel analysis tool enables the user to compile a kernel for a number of GPU architectures. Compilation of a kernel using this tool provides lower level compiled versions of the kernel, a report on errors and warnings, and kernel statistics [4].

[1] `http://msdn.microsoft.com/en-us/library/dd831853(v=vs.100).aspx`
[2] `http://msdn.microsoft.com/en-us/library/dd831853(v=vs.110).aspx`
[3] `http://developer.amd.com/tools-and-sdks/heterogeneous-computing/codexl/`

Performance Counters

Show Zero Columns

	Method	ExecutionOrder	ThreadID	CallIndex	GlobalWorkSize	WorkGroupSize	Time	LocalMemSize	VGPRs	SGPRs	ScratchRegs	FCStacks	KernelOccupancy
1	upsweep_reduction__k1_Tahiti1	1	4368	134	{ 16384 1 1}	{ 256 1 1}	0.03956	4096	38	26	0	NA	60
2	toplevel_scan__k2_Tahiti1	2	4368	135	{ 256 1 1}	{ 256 1 1}	0.01615	2176	16	22	0	NA	100
3	downsweep_scanscatter__k3_Tahiti1	3	4368	136	{ 16384 1 1}	{ 256 1 1}	0.05911	5892	31	42	0	NA	80
4	upsweep_reduction__k1_Tahiti1	4	4368	139	{ 16384 1 1}	{ 256 1 1}	0.03244	4096	38	26	0	NA	60
5	toplevel_scan__k2_Tahiti1	5	4368	140	{ 256 1 1}	{ 256 1 1}	0.01526	2176	16	22	0	NA	100
6	downsweep_scanscatter__k3_Tahiti1	6	4368	141	{ 16384 1 1}	{ 256 1 1}	0.04889	5892	31	42	0	NA	80
7	upsweep_reduction__k1_Tahiti1	7	4368	144	{ 16384 1 1}	{ 256 1 1}	0.03467	4096	38	26	0	NA	60
8	toplevel_scan__k2_Tahiti1	8	4368	145	{ 256 1 1}	{ 256 1 1}	0.01304	2176	16	22	0	NA	100
9	downsweep_scanscatter__k3_Tahiti1	9	4368	146	{ 16384 1 1}	{ 256 1 1}	0.05170	5892	31	42	0	NA	80
10	upsweep_reduction__k1_Tahiti1	10	4368	149	{ 16384 1 1}	{ 256 1 1}	0.03511	4096	38	26	0	NA	60

Figure 4.2: An example of the output from a GPU performance counters profile.

Intel OpenCL CPU Debugger

The Intel SDK for OpenCL Applications[4] includes an OpenCL debugger for Intel CPUs that integrates into Microsoft Visual Studio 2010 (and 2012). This allows the debugging of OpenCL programs running on compatible Intel CPUs in much the same way as you would debug a normal program within Visual Studio. The only caveat is that you must specify the target GPU thread you would like to debug prior to running the application. The Intel OpenCL debugger enabled much faster code navigation than the AMD debugger, and was thus used in conjunction with the AMD debugger for particularly tough problems.

4.2.2 Performance Testing

The programs were benchmarked by calculating the arithmetic mean of the recorded time for at least five executions. For GPU benchmarks, this included the transfer of data to and from the GPU (unless otherwise specified), since this step is necessary for GPU computation in many cases. All benchmarks were performed on the same computer system, the specifications of which are given in Table 4.1, using the software listed in Table 4.2. When only testing with one GPU, the second GPU was removed from the system to maximise the available PCI Express bandwidth[5].

The impact of individual GPU optimisations can be measured using a number of different methods. For example, optimisations could be implemented and benchmarked on a base unoptimised implementation, a version with all other optimisations implemented, or

[4] `http://software.intel.com/en-us/vcsource/tools/opencl-sdk`

[5] The presence of two GPUs reduced the number of active PCI Express lanes from 16x to 8x on our motherboard.

Table 4.1: System hardware specification.

Component	**Description**
Motherboard	Intel 'Blue Hills' DZ77BH
CPU Model	Intel Core i7-3770
Graphics Processing Units	2 x Gigabyte GV-R797OC-3GD
RAM	2 x 4 GB DDRIII 1600 MHz

Table 4.2: System software specification.

Software	**Version**
Linux Mint 15	3.8.0-19-generic x86_64
Microsoft Windows 7 Enterprise	SP1 x86_64
AMD APP SDK Developer	2.8.1.0
AMD APP SDK Runtime	10.0.1084.4
Intel SDK for OpenCL Applications	2012 & 2013
Visual Studio	10.0.40219 SP1Rel
CodeXL	1.2.2484
GNU Compiler Collection	4.7.3

the partly optimised version of the implementation on which the optimisation was first implemented. The order in which the optimisations are applied affects the performance impact of individual optimisations and thus their perceived value.

When benchmarking optimisations on an otherwise unoptimised implementation, the impact of each optimisation is likely to be exaggerated because of the large scope for improvement. Conversely, the impact of optimisations on an otherwise fully optimised implementation might not portray the value of an individual optimisation in the absence of one of the already implemented optimisations (some of which may not be applicable to other applications). Several optimisations are also only applicable in certain scenarios. For example, loop unrolling is only applicable if loops were not already unrolled as a result of the use of vector types. We have thus used the third approach of benchmarking optimisations on the partially optimised implementations on which they were first attempted. If a similar order of optimisations is followed, this should provide the truest reflection of the impact of the optimisations.

Name	Description
Wavefronts	Total wavefronts.
VALUInsts	The average number of vector ALU instructions executed per work-item (affected by flow control).
SALUInsts	The average number of scalar ALU instructions executed per work-item (affected by flow control).
VFetchInsts	The average number of vector fetch instructions from the video memory executed per work-item (affected by flow control).
SFetchInsts	The average number of scalar fetch instructions from the video memory executed per work-item (affected by flow control).
VWriteInsts	The average number of vector write instructions to the video memory executed per work-item (affected by flow control).
LDSInsts	The average number of instructions to/from the LDS executed per work-item (affected by flow control).
VALUUtilization	The percentage of active vector ALU threads in a wave. A lower number can mean either more thread divergence in a wave or that the work-group size is not a multiple of 64. Value range: 0% (bad), 100% (ideal, meaning no thread divergence).
VALUBusy	The percentage of GPUTime vector ALU instructions are processed. Value range: 0% (bad) to 100% (optimal).
SALUBusy	The percentage of GPUTime scalar ALU instructions are processed. Value range: 0% (bad) to 100% (optimal).
FetchSize	The total kilobytes fetched from the video memory. This is measured with all extra fetches and any cache or memory effects taken into account.
CacheHit	The percentage of fetch, write, atomic, and other instructions that hit the data cache. Value range: 0% (no hit) to 100% (optimal).
MemUnitBusy	The percentage of GPUTime the memory unit is active. The result includes the stall time (MemUnitStalled). This is measured with all extra fetches and writes and any cache or memory effects taken into account. Value range: 0% to 100% (fetch-bound).
MemUnitStalled	The percentage of GPUTime the memory unit is stalled. Try reduce the number or size of fetches and writes if possible. Value range: 0% (optimal) to 100% (bad).
WriteUnitStalled	The percentage of GPUTime the Write unit is stalled. Value range: 0% to 100% (bad).
LDSBankConflict	The percentage of GPUTime LDS is stalled by bank conflicts. Value range: 0% (optimal) to 100% (bad).
WriteSize	The total kilobytes written to the video memory. This is measured with all extra fetches and any cache or memory effects taken into account.
GDSInsts	The average number of instructions to/from the GDS executed per work-item (affected by flow control). This counter is a subset of the VALUInsts counter.

Figure 4.3: A summary of the performance counters given by the CodeXL (taken from [5]).

The GPU implementations were compared to at least one multithreaded CPU implementation. If an existing multithreaded CPU implementation was not available, one was created by parallelising the program in much the same way as the GPU implementation. It should be noted that the comparisons between the CPU and GPU versions are made from the perspective of an application developer rather than an architect researcher. Consequently, the GPU speedups given are not accurate reflections of the relative speeds of the CPU and GPU, but instead reflect the performance improvement achieved through parallelisation using GPUs. The qualification that the comparisons are of the program implementations rather than the hardware is not always included to improve readability.

4.2.3 Identification of Performance Bottlenecks

Before optimisations can be attempted, the areas in need of optimisation must first be known. Performance bottlenecks were primarily identified through the review of the kernel performance counters given by the CodeXL kernel profiling tool. A summary of these counters is given in Figure 4.3. The counters most commonly reviewed were the *VALUUtilization*, *VALUBusy*, *SALUBusy*, *CacheHit*, and *MemUnitStalled*, as these give an indication of the branch divergence, stream processor utilisation, and efficiency of memory access patterns. Another performance indicator reviewed was the maximum kernel occupancy. This value is constrained by the number of general purpose registers used by each work-item, the quantity of shared memory allocated to each work-group, and the work-group size [5].

4.3 Summary

The problems accelerated in this study were selected based on their perceived benefit from GPU acceleration, difficulty, and lack of a freely available and GPU solution with good performance. With the exception of the radix sort case study, the problems were accelerated by modifying the existing CPU solution to work in OpenCL, and iteratively optimising the solution until a suitable speedup was achieved. Since the radix sort already had an optimised GPU solution in an alternative GPGPU framework, a naïve and a highly optimised implementation were ported to OpenCL and compared instead. All GPU solutions were compared with a multithreaded CPU implementation to obtain the GPU speedup. Since problem acceleration was done from the perspective of an application developer, the speedups obtained represent the performance gained from parallelising the existing implementation on a GPU, rather than the relative speeds of the CPU and GPU.

Chapter 5

Case Study 1: Hydrological Uncertainty Model

Hydrological models are simplified representations of certain processes within the hydrological cycle. These models are primarily used to increase our understanding of the observed processes and to make hydrological predictions or estimations [52]. A recent trend in hydrological modelling is the use of uncertainty analysis [33]. Using this approach, a model is run thousands of times using different input parameters. This can take a considerable amount of time on a CPU and could stand to benefit greatly from GPU acceleration, owing to the problem's SIMD-like nature. The hydrological uncertainty model accelerated here is based on an adapted version of the Pitman rainfall-runoff model [63] used for water resource estimation.

5.1 Pitman Hydrological Model

The Pitman model is a conceptual type, semi-distributed (sub-catchment), monthly time-step model that includes some 23 parameters that govern the algorithms defining the hydrological storages and processes such as evapotranspiration, interception, surface runoff, soil moisture storage, interflow, groundwater (GW) recharge and drainage, and catchment routing [33]. An overview of the hydrological processes and their relationships for this version of the model is illustrated in Figure 5.1. The full details of the model are not given here as the study could have used any model of this type. The conceptual diagram in Figure 5.1 and the brief explanation of the model are merely provided to illustrate the degree

of model complexity. The model is typically run over a period of 40 to 90 years (480 to 1,080 months) depending on the availability of input rainfall data. Each component of the model (Figure 5.1) consists of a set of sequential algorithms that generate either output data or the values of internal state variables that are used in the next time interval or as input to the next downstream sub-catchment. Most of the model components operate over four equal steps within the one-month main time step to avoid excessive changes in any of the state variables (storages or fluxes) before other components are updated. This approach is frequently used in coarse time step models [30] in recognition of the fact that, in nature, water balance components operate simultaneously.

The ability of the model to accurately represent the hydrological response of any given catchment is reliant on the correct specification of the model parameters. Estimation of these parameters is always problematic, even if they are calibrated against an observed stream flow time series. Many of the parameter estimation issues are associated with inter-relationships between model parameters and the problem of equifinality [13], whereby similar model outputs can be achieved with different parameter sets.

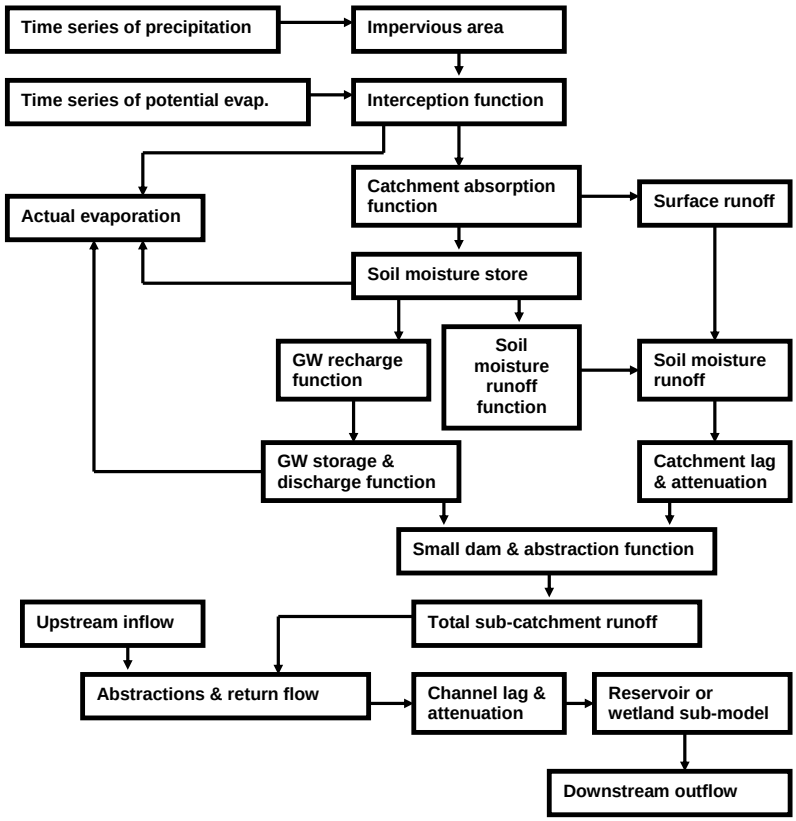


Figure 5.1: Conceptual process diagram of the version of the Pitman model in [31].

The uncertainty version of this model is designed to assist in the estimation of these parameters and allows the model results for many different options within the feasible parameter space to be explored [33]. The model has the goal of establishing parameter values, setting parameter uncertainty bounds [38], and exploring parameter inter-dependencies. Parameter inputs to the model are specified as either means and standard deviations of normal distribution functions, or minimum and maximum values of uniform distribution functions. If the normal distribution function option is used, the minimum and maximum values are used to constrain the tails of the distribution.

The Delphi code on which this model is based has evolved from the first version of the Spatial and Time Series Information Modelling (SPATSIM) system developed in the early 2000s [32], rather than having been meticulously designed from the perspective of efficient software architecture. The model is run many times (typically between 5,000 and 20,000) to generate ensembles of outputs, where each output is based on independent random samples from the defined parameter distributions. Running 10,000 ensembles for a basin with 30 sub-divisions over an 80 year input climate time series involves repeating the full set of model algorithms some 288 x 10^6 times, not to mention the time taken to access data from, and write the results to, the SPATSIM database tables. A second version of the uncertainty model also allows the precipitation inputs to the model to be considered with uncertainty and makes use of stochastically generated rainfall sequences rather than a single fixed time series. Typically, the model is run with 500 stochastic rainfall sequences (for each spatial sub-division or catchment within the basin), in which case the number of parameter samples is limited to 500, giving a total number of 250,000 ensembles or 72 x 10^8 operations of the model algorithms. Post-processing options available within the SPATSIM system include global sensitivity analysis [68] of the ensembles, frequency distribution analysis of selected output metrics, such as mean annual runoff, groundwater recharge, and several percentiles of the simulated flow duration curves, and a relatively simple approach to water resources yield uncertainty analysis. Figure 5.2 illustrates the software configurations of the Delphi (SPATSIM) versions of the two models. These configurations were designed for sequential execution, but they can be parallelised by executing the functions within the ensemble loop on different threads with different input parameters, as is illustrated in Figure 5.3.

The complexity of each model run and the design of the program can result in an undesirable model runtime of several hours on a modern CPU, even when running 10,000 ensembles (i.e., without stochastic rainfall inputs). An application of the stochastic rainfall version of the model to the Caledon River basin with 31 sub-catchments in Southern Africa takes approximately 45 hours to complete. While these model runs would not

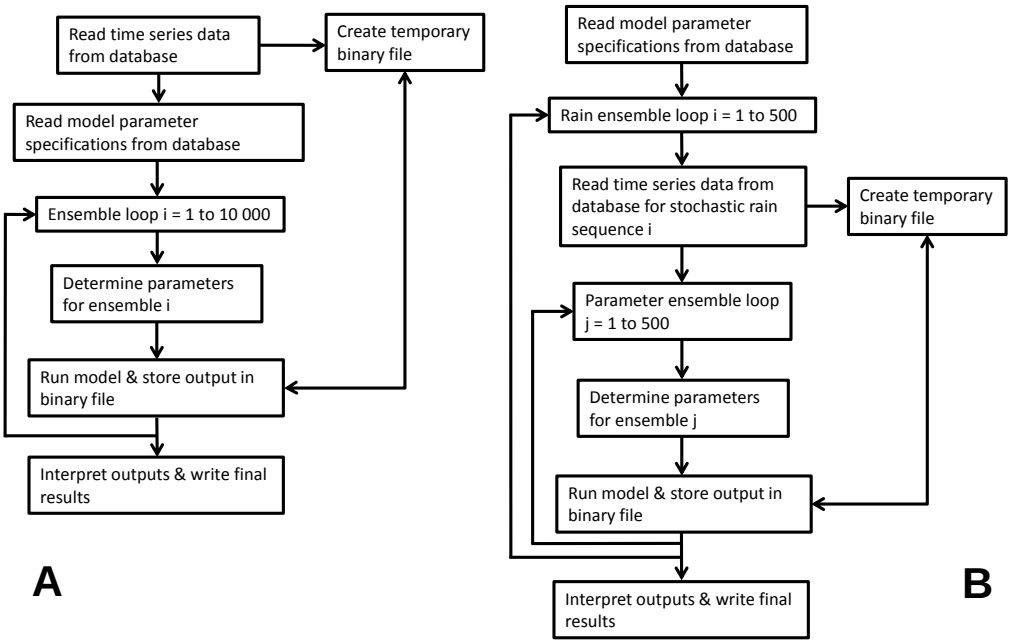


Figure 5.2: Software flow diagrams for the uncertainty version with single climate inputs (A) and the version using stochastic rainfall input sequences (B).

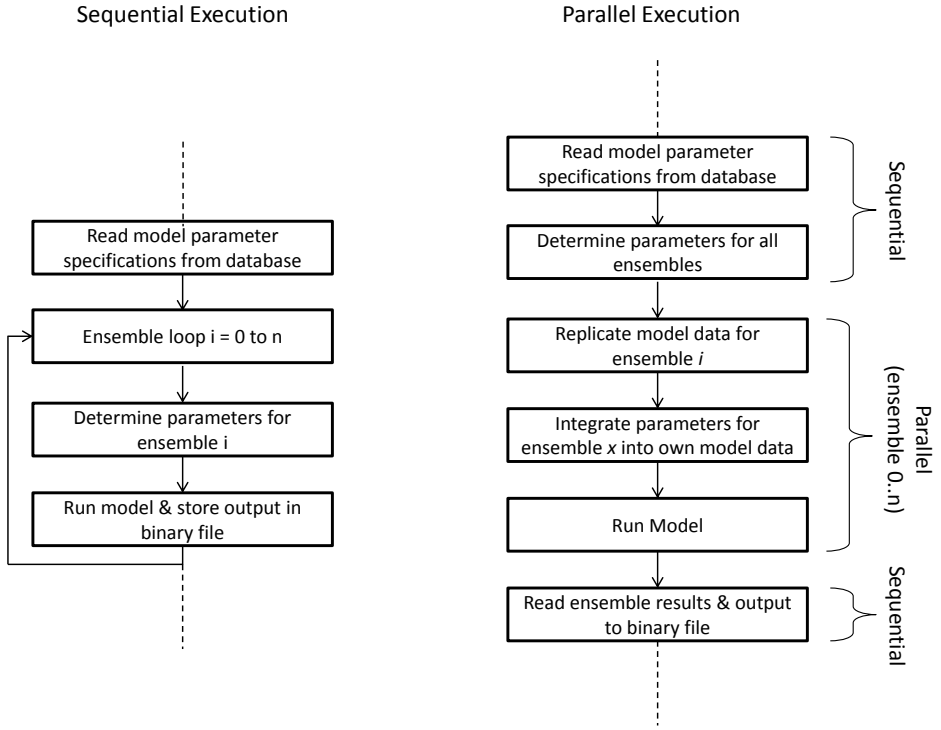


Figure 5.3: Comparison of the sequential and parallel implementations of the uncertainty version with single climate inputs.

normally be repeated many times, as would be the case with a purely manual parameter search and calibration approach, it is sometimes useful to run the uncertainty model several times to explore the effects of different combinations of constraints on the distributions of the different model parameters. Reducing the model runtime would therefore be extremely beneficial and, since each ensemble is computed in isolation, it is possible for this to be achieved by running ensembles in parallel.

5.2 GPU Implementation

For the model to be accelerated on a GPU, the core model code had to be extracted from the original Delphi project and implemented in OpenCL. The method used to do this is outlined below.

5.2.1 General Approach

A three-step approach was taken to accelerate the model with the aim of minimising debugging time. These steps are illustrated in Figure 5.4. The first step involved the extraction and conversion of the core model code into a new project or language. The motivation behind this step was to do a "trial run" implementation of the core model outside its original project with the luxury of CPU debugging tools. C# was used in this step, as the Visual Studio IDE has excellent debugging tools for the language and it is syntactically very similar to C and C++ (and hence OpenCL). The next step used the output of step one to create an OpenCL version of the model. Ideally, this step should involve less work because of the work done in step one. Finally, the OpenCL implementation was tuned to provide a satisfactory speedup on our test system without significantly modifying the original code. The proposed next step was to integrate the OpenCL solution into the original Delphi program.

5.2.2 Creating the C# Implementation

For the model to be run in an alternative language, the Delphi model data had to be exported at the time of model execution. For ease of development, these data were written to a binary file to be read by alternative implementations. The data were then imported into the Delphi equivalent data structures in C# to support the model code.

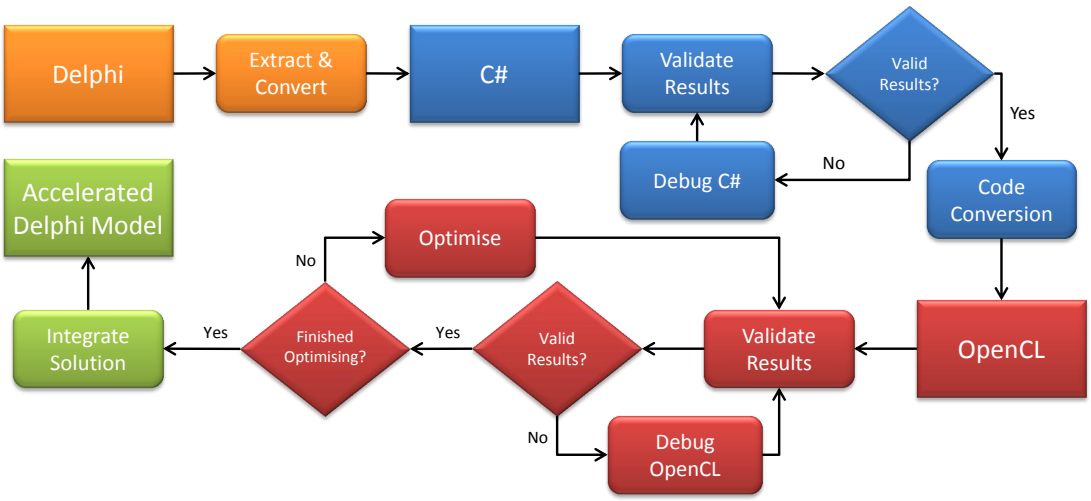


Figure 5.4: The approach taken to accelerate the uncertainty version of the adapted Pitman rainfall-runoff model.

Rather than rewriting the entire model line-by-line in C#, the Delphi code was copied, and its syntax iteratively translated into C# using regular expression replacements. Any syntax differences that could not be fixed this way were then changed manually. This resulted in a C# version of the model that ran, but produced results with significant numerical differences to the results of the original Delphi implementation. Even with the powerful debugging tools available for C#, it took a great deal of effort to debug the conversion errors that caused the differences in results. Given that CPU debugging is much easier than GPU debugging, it is worthwhile to do this intermediate step to identify such problems before the conversion to GPU code. For the purpose of performance comparisons, a multithreaded version of the C# implementation was also created. This was done by distributing the ensembles to be run among the threads in a thread pool.

5.2.3 Creating the OpenCL Implementation

Before work on the OpenCL model could begin, the managing host program had to be created. This was written in C++ to allow use of the OpenCL API more directly, although OpenCL API wrappers do exist for many of the popular programming languages. The model data were read into the program in the form of C++ structs, which were then written to OpenCL buffers for use in OpenCL kernels. A kernel was created to replicate the modifiable model data for the specified number of ensembles to enable their concurrent execution. This kernel also integrated the model parameters into the data, essentially preparing the data for each ensemble prior to model execution.

The model logic was added to a second kernel, which was created using the C# code as the starting point. The similarity of the C# syntax to OpenCL C made the conversion of the C# code relatively trivial, with most of the changes required being either function headers, mathematical operations, or the referenced location of variables. It was considerably more challenging to debug the GPU code than it was the C# code, owing to inferior debugging tools and scale of parallelism.

5.3 Results

The performance of the model was evaluated on a sample dataset that contained four sub-catchments for a variety of ensemble sizes. An additional dataset that represented the Caledon River basin with 31 sub-catchments was also tested to verify the consistency of the results. To gauge the impact and importance of GPU optimisations, the GPU implementation was evaluated both before and after optimisations had been applied.

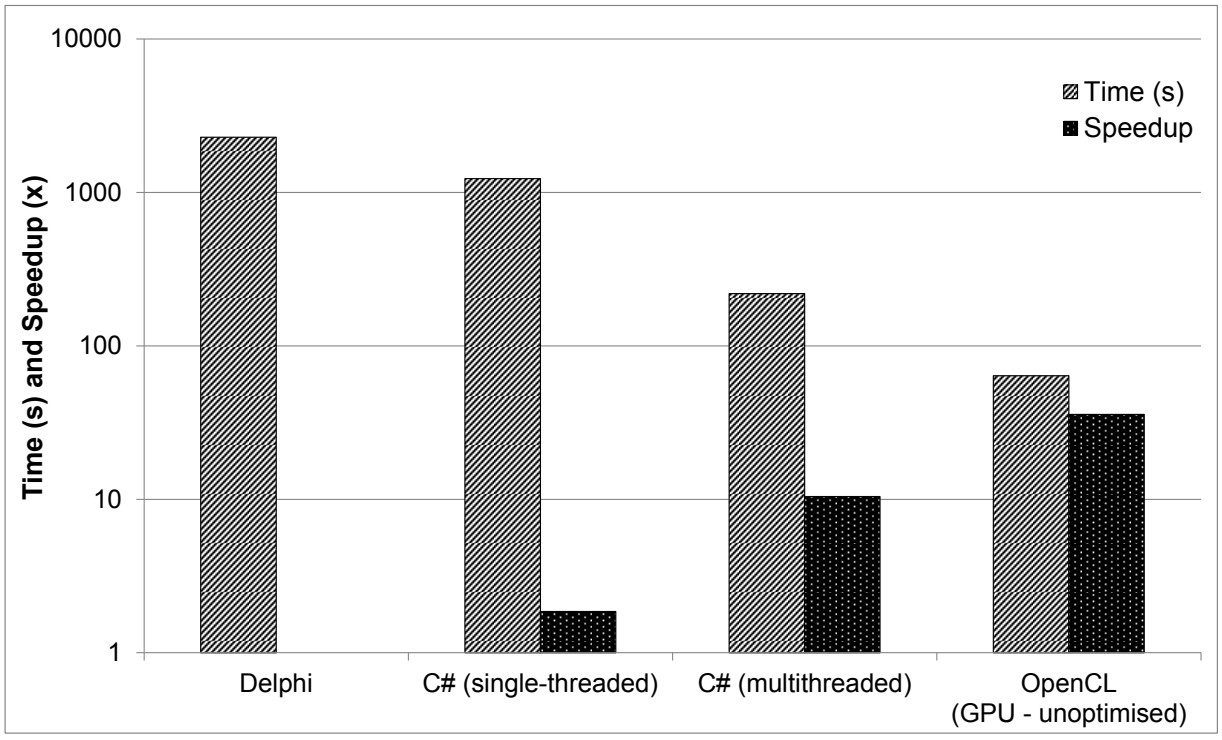


Figure 5.5: Model performance comparison for the execution of 50,000 ensembles on the sample dataset, with the speedups relative to the Delphi implementation plotted alongside.

The results in Figure 5.5 show that there is a significant improvement in speed from the original Delphi version to the OpenCL version running on a single HD7970 GPU.

Surprisingly, the single-threaded C# version is also noticeably faster than the Delphi version, even though the logic is identical. The OpenCL (GPU) implementation was 3.4x faster than the multithreaded C# implementation, and 35.8x faster than the original Delphi implementation. While the speedup over the original Delphi version is of practical significance, it is not a fair comparison since the Delphi version is single threaded. The unoptimised GPU version compared to the multithreaded C# version is a more relevant comparison and only results in a modest speedup of 3.4x.

5.3.1 Verifying the Results

When the results between the Delphi and C# implementations were compared, we were able to achieve a binary match. We were not able to achieve this with the OpenCL version; the results produced by the GPU differed very slightly from the Delphi and C# results. A frequency distribution of the size of these differences for the primary model outputs: mean monthly rainfall volume (MMRV), mean monthly groundwater recharge (MMGR), 10th percentile of the flow duration curve (10FDC), 50th percentile of the flow duration curve (50FDC), and 90th percentile of the flow duration curve (90FDC), can be seen in Figure 5.6. Leeser et al. [44] explain how the implementation flexibility of the IEEE 754 floating-point standard allows different hardware implementations to arrive at slightly different results, which could explain the observed differences. Since the slight variations in the results were acceptable for this model, a difference threshold was applied to verify subsequent OpenCL results.

5.3.2 Optimisations

Analysis of CodeXL's kernel profiling output revealed two characteristics of the GPU program that were hindering performance: register usage and memory latency.

Register Usage

Heavy register (local variable) usage restricts the number of wavefronts that can be scheduled on a compute unit, which results in the GPU operating at a lower occupancy [3]. Although this does not mean that compute units will be without work, it does mean that they are unable to swap between as many wavefronts to hide memory access latency.

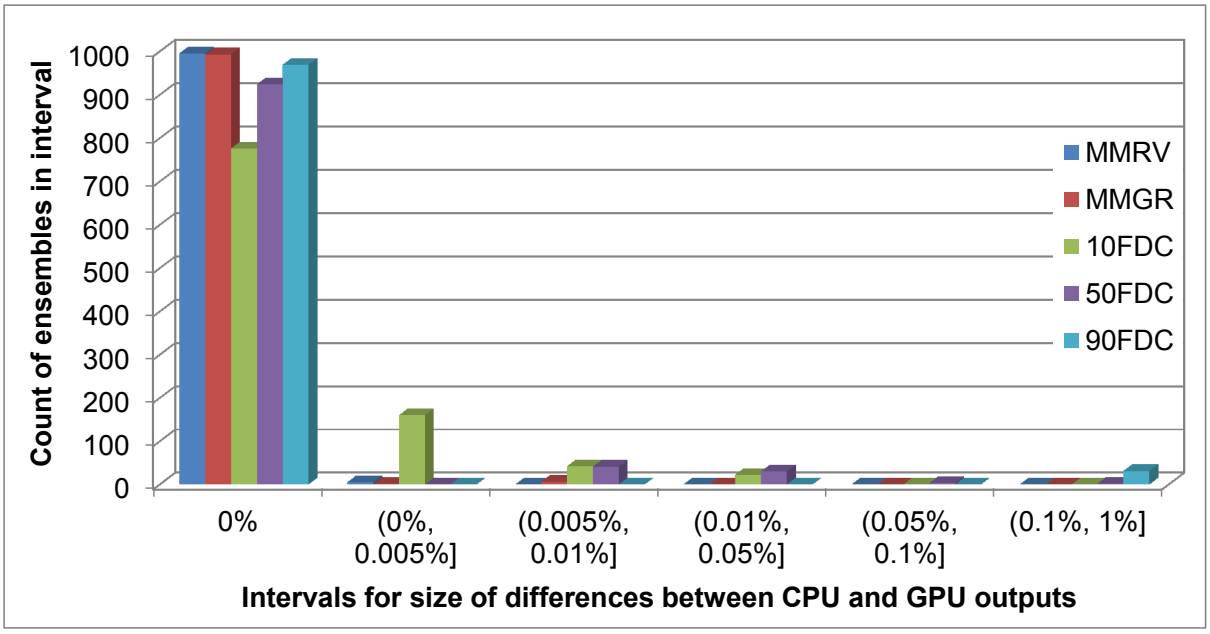


Figure 5.6: Frequency distribution showing the size of the differences between the primary CPU and GPU model outputs.

Excessive register usage results in registers spilling into slower memory, which incurs a significant performance penalty [3]. As can be expected with models of this kind, there were a large number of model state variables, which resulted in spilled registers and a low GPU occupancy of 10%.

Reducing register usage The program function that is responsible for running the model declares 22 private arrays that are significant contributors to the register usage of the program. These arrays were moved to global memory to determine whether manually moving less frequently used model state variables out of register space would improve performance by eliminating register spillage. The change did not, however, make a meaningful difference to the runtime of the model. Using local memory was also considered, but its limited size made its use impractical for this purpose. It is likely that the register usage could be reduced through code restructuring, but this was not attempted owing to the probable time requirement.

Memory Latency

Kernels that spend more time waiting for data transfers than performing computation are known as memory-bound kernels. CodeXL indicated that the utilisation of the GPU

stream processors was under 12%, making the kernel significantly memory bound. Any memory optimisations were therefore likely to yield a performance improvement. Two methods for reducing the impact of memory latency were identified: caching of frequently used data and optimising the memory layout.

Data Caching The kernel was memory bound because the model data were stored in global memory, which is the slowest GPU memory. Caching the most frequently accessed data in local memory would help to reduce the use of global memory, and also take advantage of local memory's higher bandwidth and lower access latency. An effort was made to do this, but it proved to be ineffective for this model because the size of the data that needed to be cached was too large, resulting in frequent swapping between local and global memory. Nevertheless, this optimisation is worth mentioning since it can be beneficial to other memory-bound applications.

Data Layout Optimising memory access patterns to avoid memory channel conflicts is another technique that can be used to improve memory performance [3]. The original data layout resulted in an undesirable many-unit stride between data accesses of consecutive work items, since the data for each ensemble were stored in large structs. To change this into a one-unit stride, the kernel that replicated the original model data was modified to store consecutive items in the data structs x places apart, where x is the number of model ensembles. A comparison between the original layout and this new layout is illustrated in Figure 5.7. This optimisation resulted in a considerable threefold performance improvement.

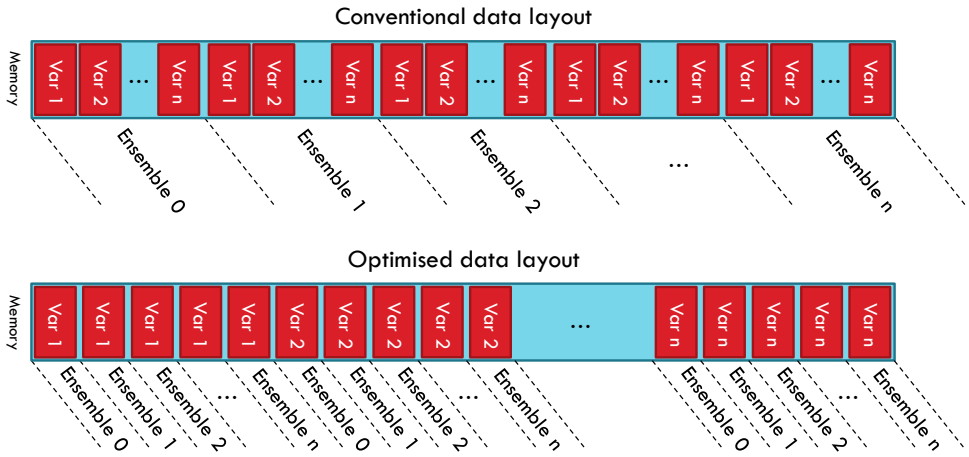


Figure 5.7: The original data layout in memory compared to the optimised layout.

5.3.3 Optimised GPU Implementation

A performance comparison between the optimised GPU implementation and the CPU implementations for the sample dataset is given in Figure 5.8. The performance of the GPU implementation was also tested on a Caledon River basin dataset for different problem sizes, the results of which can be seen in Figure 5.9.

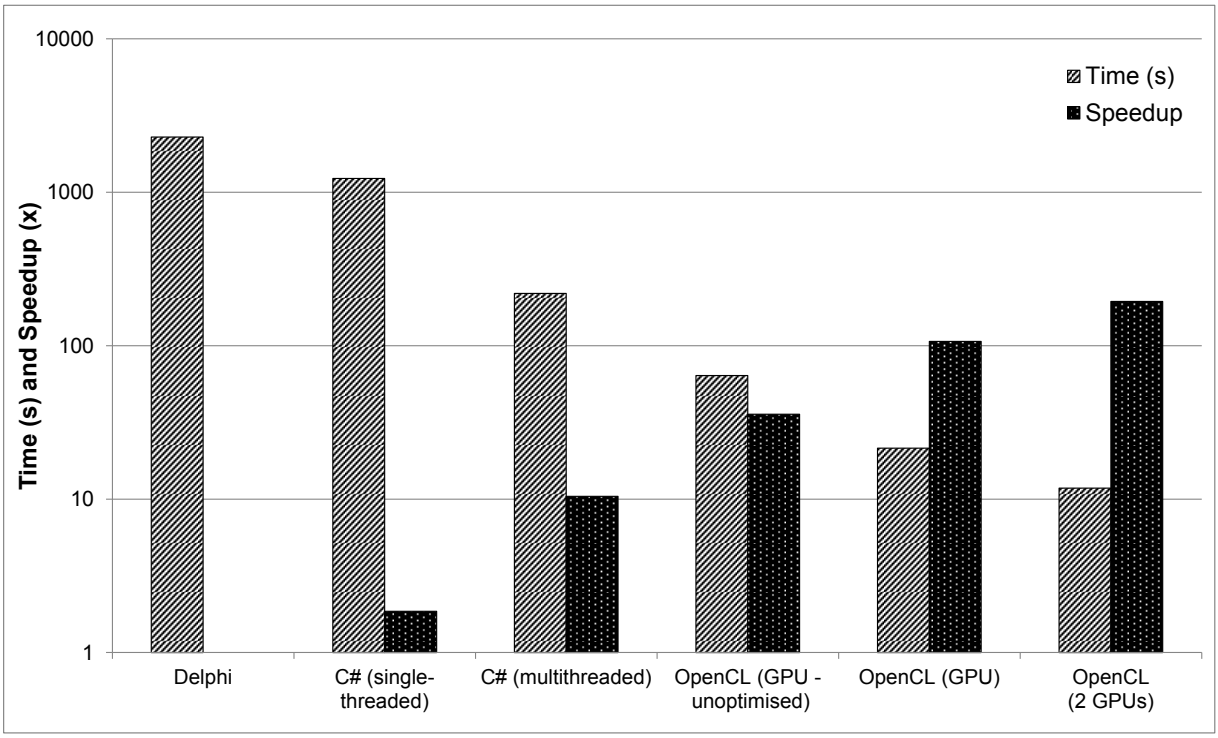


Figure 5.8: Model performance comparison for the execution of 50,000 ensembles on the sample dataset after optimisation, with the speedups relative to the Delphi implementation plotted alongside.

As shown in Figure 5.8, the data layout optimisation made a considerable difference to GPU performance. The optimised version is 3x faster than the unoptimised version, and consequently 107x faster than the Delphi implementation and 10x faster than the multithreaded C# version. The addition of a second identical GPU scaled the performance almost linearly, with the speedup over the C# version improving to 18.6x.

The results for the Caledon basin dataset in Figure 5.9 reveal that if 15,000 or more ensembles are run, the GPU's speedup over the CPU implementations is consistent with the results in Figure 5.8. However, if fewer than 15,000 ensembles are run, the speedup starts declining. The decline in performance is the result of idle stream processors after the first 8,192 ensembles[1] have been processed. Considering that running a large number

[1]This the minimum number of work-items needed to provide work to all the stream processors.

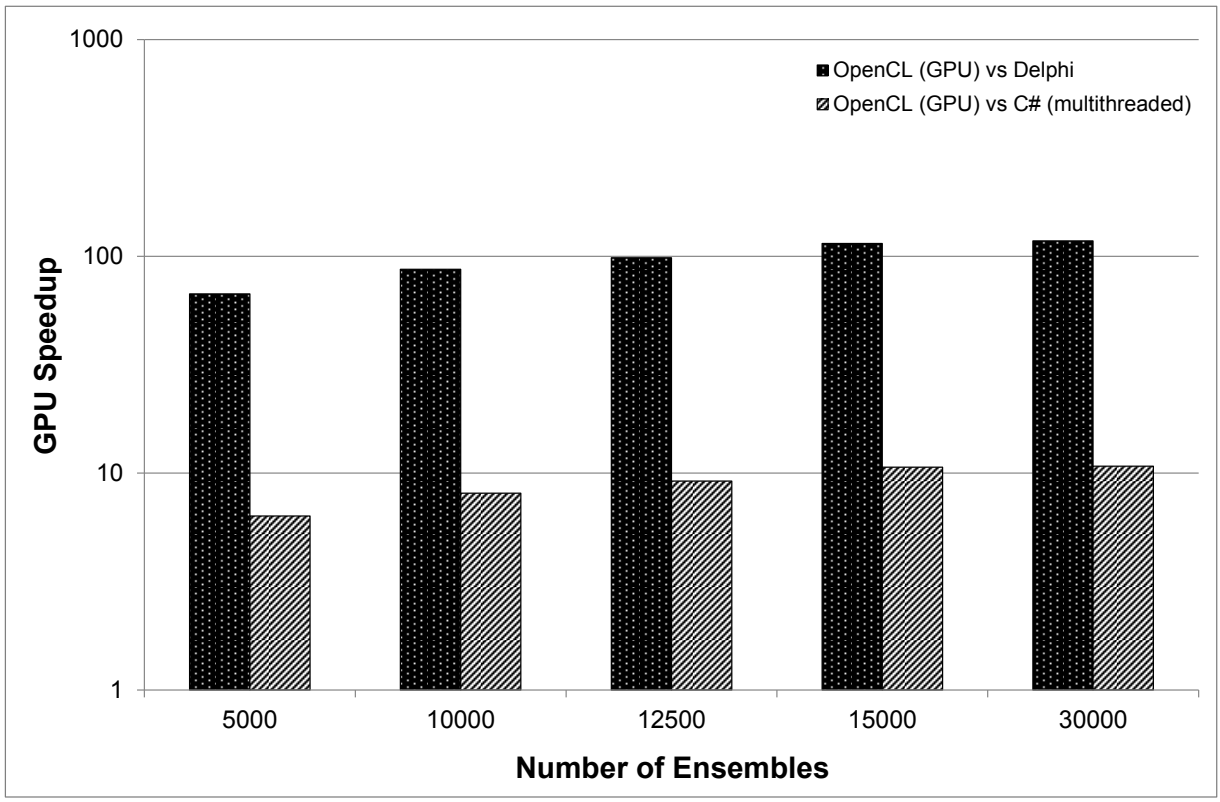


Figure 5.9: The GPU speedup when running 5,000 to 30,000 ensembles of the model on the Caledon basin dataset.

of ensembles is desirable in uncertainty modelling, this should not be a cause for concern. The performance impact of data transfers to and from the GPU was also measured and found to account for less than 0.1% of the total program execution time, and this value decreased with higher ensemble counts.

5.4 Summary

The goal of this study was to determine the difficulty of using a GPU to accelerate an adapted Pitman rainfall-runoff uncertainty model without significantly changing the original code and obtain a worthwhile speedup. This is an extension of our previously published work on this hydrological model [78]. Through a three-step approach, a tenfold speedup over a multithreaded C# implementation of the model was achieved by using a commodity GPU, whilst minimising the work needed to create the GPU version. One significant optimisation was implemented to achieve this result, that is, the rearrangement of the model data in global memory to improve global memory throughput. The speedup was also found to scale almost linearly with the addition of a second identical GPU.

Additional testing of the model on an alternative dataset for a number of different problem sizes revealed the performance to be consistent with the initial dataset when running at least 15,000 ensembles, and the speedup remained constant with larger problem sizes. Smaller problem sizes resulted in declining performance as a result of insufficient parallel work for the GPU. Given the work required to achieve these results, it was estimated that users would need to have a basic understanding of OpenCL and parallel programming as well as knowledge of an efficient global memory data layout to achieve similar results with a comparable model. As the model code was not redesigned specifically for GPUs, it is likely that there is still potential for further improvement.

Chapter 6

Case Study 2: *K*-Difference String Matching

The matching of strings with an allowance for small differences, or errors, forms an integral part of many data processing algorithms. Its use is particularly evident in bioinformatics, signal processing, and spelling correction [56]. Other areas in which it is used include handwriting recognition, intrusion and virus detection, spam detection and filtering, data mining, pattern recognition, and image compression [6, 56, 84]. However, calculating the k-difference result of two strings is time consuming and thus impractical for many applications that require performing k-difference comparisons between a large number of strings. Scenarios where this may be required include improving the accuracy of email spam detection techniques [6] and malware detection on network stream data [85].

The layout of a sequential algorithm for this use case may be similar to Algorithm 6.1.

Algorithm 6.1 Comparing input strings to a number of test patterns.

```
function PATTERN_MATCH(inputStrings, testPatterns, results, threshold)
    for i ∈ 0..len(testPatterns) do
        for j ∈ 0..len(inputStrings) do
            results_{i,j} ← kdiff(testPatterns_i, inputStrings_j, threshold)
end function
```

This use case is somewhat different to the type of approximate string matching typically utilised in bioinformatics, where an approximate match of a short string is searched within a comparatively long text.

In this chapter, we investigate the effectiveness of using a GPU to perform large numbers of k-difference comparisons using two different algorithms. In the benchmarking of the

CPU and GPU implementations, we evaluate the performance impact of string length, the size of the string alphabet, and the chosen cut-off threshold. Since the structure of the problem is well matched to the GPU's architecture, it is expected that the GPU will provide a significant performance improvement.

6.1 Approximate String Matching

There are two primary types of approximate string matching, namely k-difference and k-mismatch [56]. A k-mismatch algorithm calculates the *hamming distance* between two strings, which is the minimum number of substitutions that need to be made to make the compared strings identical. Algorithms that use k-difference matching calculate the *Levenshtein distance* or *edit distance* between two strings, which is similar to the hamming distance, except that it also allows deletions and insertions. The k value specifies the maximum number of weighted errors between two strings after which the strings are considered sufficiently different to be classed as non-matching. Since each of the operations in a k-difference algorithm usually has a configurable cost, these algorithms can be easily adapted into k-mismatch algorithms by giving the deletion and insertion operations sufficiently large cost values.

Performing k-difference string matching quickly and efficiently is challenging, and it has long been a classic computer science problem. Many different algorithms of varying complexities have been developed over the past three decades [56], but for this study, we are only interested in so-called "online" algorithms that do not perform any pre-processing.

The original solution to calculating the Levenshtein distance between two strings used a dynamic programming approach [56]. The algorithm involves building a difference matrix $\mathrm{C}_{0..i,0...j}$, where *0..i* represents the characters in the test string m, and *0..j* represents the characters in the input string n. The first row $\mathrm{C}_{0,j}$ is populated with its column index, and the first column $\mathrm{C}_{i,0}$ is populated with its row index. The rest of the table is populated by applying the algorithm shown in Algorithm 6.2.

Algorithm 6.2 Populating the dynamic programming matrix.

if $n_i == m_i$ **then**
    $C_{i,j} \leftarrow C_{i-1,j-1}$
else
    $C_{i,j} \leftarrow 1 + min(C_{i-1,j-1}, C_{i-1,j}, C_{i,j-1})$

After the table is populated in this way, the last cell in the table, $C_{i-1,j-1}$, contains the difference value between the two strings. This value can then be checked against a threshold, k, to determine whether the strings are an approximate match. An example of this method is illustrated in Figure 6.1. Although this approach is simple and flexible, it is $O(mn)$ in both time and space, which means it is impractical for many applications on traditional hardware. Since the original algorithm, there have been two significant improvements to k-difference string matching that are useful for problems that do not require text searching.

Test pattern

Input string

		d	y	n	a	m	i	c
	0	1	2	3	4	5	6	7
d	1	0	1	2	3	4	5	6
d	2	1	1	2	3	4	5	6
y	3	2	1	2	3	4	5	6
a	4	3	2	2	2	3	4	5
m	5	4	3	3	3	2	3	4
i	6	5	4	4	4	3	2	3
d	7	6	5	5	5	4	3	3

Figure 6.1: The dynamic programming approach to calculating the Levenshtein distance between two strings. The final difference value is indicated by the shaded block.

6.1.1 The Cut-Off Heuristic

In 1985, Ukkonen observed that the dynamic programming matrix property $D_{i,j} \geq D_{i-1,j-1}$ allows Levenshtein distance algorithms to avoid unnecessary calculation of cells in a column where the value is always greater than the applied threshold, k [79]. This column calculation cut-off heuristic can be applied as follows:

Algorithm 6.3 Ukkonen's cut-off heuristic.

$active \leftarrow k$
for $j \in 1..n$ **do**
  calculate $D_{1..a+1,j}$, where $a == active_{j-1}$
  $active_j \leftarrow$last cell in column j where $D_{i,j} \leq k$

This optimisation has been proven to reduce the average expected time complexity of the algorithm to $O(kn)$ [16]. A faster algorithm that uses Ukkonen's cut-off was developed

that further improves the average time to $O(kn/\sqrt{\alpha})$, where α represents the alphabet size [16]. However, this algorithm has the drawback of reduced flexibility [56].

6.1.2 Bit Parallelism

It was discovered that existing string matching algorithms could be accelerated by exploiting the parallelism inherent in operations on computer words [56]. Myers [54] used this parallelism effectively for k-difference matching by encoding the differences along columns of the dynamic programming matrix using two bits, and transitioning from column to column using bit operations. This solution is best suited to short strings where ideally $m \leq$ *word size*, as such strings would not require the use of multiple words to emulate a word of size m or greater. Myers' algorithm has a much improved average expected time complexity of $O(mn/w)$, but can be difficult to adapt to different distance functions [54, 56].

6.1.3 Existing GPU Solutions

There are many existing GPU solutions for k-difference string matching, where a pattern is searched for within a very long text (e.g. large DNA sequence). However, there are few GPU solutions designed for the problem of k-difference matching between large numbers of aligned short texts[1]. While the existing k-difference string matching solutions could conceivably be adapted for this problem, they would not be very efficient. This is because they utilise the vast parallelism of GPUs to cooperatively solve a single k-difference problem, rather than solve many problems concurrently. For shorter texts, this would result in underutilisation of the GPU because of the lack of available parallelism in each comparison.

Other than our own work, the only published account of accelerating this particular problem on a GPU was found to be a Master's thesis by Langner [41]. Langner implemented Myers' bit-parallel algorithm with Ukkonen's cut-off in CUDA and tested it with an NVIDIA Geforce 460 GTX. He reported speedups of up to 4x over an i7 930 CPU with an OpenCL implementation. This is lower than we would expect from GPU acceleration, but there appear to be opportunities for improving his implementation. Possible improvements include computing multiple comparisons per thread, using multiple words

[1] The test data we use consists of strings that are less than 560 characters in length, which is considerably shorter than the long DNA sequences typically used in approximate string matching.

rather than increasing the word size for longer strings, and the use of thread cooperation. Through the use of a modern GPU and creation of a GPU implementation that includes these improvements, we expect to achieve a larger speedup.

6.2 GPU Implementations

Two GPU solutions were created to solve this problem. The first solution used a standard dynamic programming matrix algorithm that is easily adaptable to different distance functions, and the second solution used an adaptation of Myers' bit-parallel algorithm, which is much faster, but not as simple to modify. The GPU implementations do not parallelise this algorithm, but rather run many instances of it concurrently.

6.2.1 Simple Dynamic Programming Matrix Implementation

One of the simplest and most flexible methods for finding the k-difference between two strings is building a dynamic programming matrix with an algorithm such as the one described in Algorithm 6.2. However, accelerating this approach to string matching on a GPU is challenging, as each cell in the matrix requires a character comparison, which results in a very low ratio of computation instructions to memory transactions.

A simplified version of the GPU implementation of this algorithm (including Ukkonen's cut-off improvement) is provided in Algorithm 6.4. Since filling column i only requires the values in column $i-1$, the amount of memory used by the algorithm can be reduced significantly by only storing two columns of the matrix at one time. However, the amount of data transferred to and from global memory is a bottleneck. The reason for this can be explained by reviewing the core function of the algorithm, `calc_col`. In this function, a loop iterates over the number of char16 vectors in the string. Within the loop, a batch of 16 characters (16 bytes) and 16 short integers (32 bytes) are read from global memory, and the characters are compared to the reference character, c. The results of these comparisons are saved back to global memory. This amounts to $\lceil \frac{patternLen}{16} \rceil \times (48)$ bytes read from global memory and $\lceil \frac{patternLen}{16} \rceil \times (32)$ bytes saved to global memory per `calc_col` function call. Without Ukkonen's cut-off, a string of 256 characters in length would result in `calc_col` reading and writing a total of

$$\left\lceil \frac{256+15}{16} \right\rceil \times (48+32) = 1280 \text{ bytes}$$

Algorithm 6.4 A simplified representation of the GPU implementation of the standard algorithm.

1: **function** CALC_COL(global $strTest$, $patternLen$, c, global $prevCol$, global $curCol$, i, $bestk$, $active$)
2: ▷ $strTest$ is the test pattern, $patternLen$ is the length of the shortest pattern, c is character in the input string currently tested against, $prevCol$ is the previously calculated column, $curCol$ is the column to be calculated, and i is the column number.
3: $pos \leftarrow 0$
4: **while** $pos \leq (patternLen + 15)/16$ and $pos < active + 1$ **do**
5: $testVec \leftarrow strTest_{pos \times 16..pos \times 16+15}$
6: $prevVec \leftarrow prevCol_{pos \times 16..pos \times 16+15}$
7: **for** $p \in pos \times 16..pos \times 16 + 15$ **do**
8: **if** $p == 0$ **then**
9: $curCol_{i,0} \leftarrow (testVec_0 \neq c \ ? \ 1 : 0)$
10: **else**
11: $curCol_{i,p} \leftarrow min((testVec_p \neq c \ ? \ prevVec_{p-1} + 1 : prevVec_{p-1}), min(prevVec_p, testVec_{p-1}) + 1)$
12: $pos{+}{+}$
13: **for** $p \in 15..0$ **do**
14: **if** $curCol_{i,pos \times 16+p} <= maxdiff$ **then**
15: $curActive = pos$
16: **break**
17: $active \leftarrow curActive$
18: $bestk \leftarrow curCol_{i,(pos-1) \times 16+(queryLen+15\%16)}$
19: **end function**
20: **function** COMPARE_STRINGS(global $strInput$, global $strTest$, global $patternLen$, global $tempCol0$, global $tempCol1$, $maxdiff$)
21: ▷ $strInput$ is the input string, $strTest$ is the test string, $patternLen$ is the shorter of the input and test string, $tempCol0$ and $tempCol1$ are used to temporarily store two columns in a matrix.
22: **for** $j \in 0..patternLen$ **do**
23: $tempCol0 \leftarrow j + 1$
24: $bestk \leftarrow patternLen + 1$
25: $i \leftarrow 0$
26: $active = (patternLen + 15)/16$
27: **while** $i \leq ((patternLen + 15)/16)$ **do**
28: $inpVec \leftarrow strInput_{i \times 16..i \times 16+15}$
29: $count \leftarrow (patternLen - i \times 16) \ \% \ 16$
30: **for** $j \in 0..count$ in steps of 2 **do**
31: calc_col($strTest$, $patternLen$, $inpVec_j$, $tempCol0$, $tempCol1$, $i \times 16 + j$, $bestk$, $active$, $maxdiff$)
32: calc_col($strTest$, $patternLen$, $inpVec_{j+1}$, $tempCol1$, $tempCol0$, $i \times 16 + j + 1$,$bestk$, $active$, $maxdiff$)

```
33:         if count % 2 ≠ 0 then
34:             calc_col(strTest, patternLen, inpVec_j, tempCol0, tempCol1, i × 16 +
                    count − 1, bestk, active, maxdiff)
35:         i++
36:     if bestk ≤ maxdiff then
37:         return True
38:     else
39:         return False
40: end function
```

per `calc_col` function call. According to [3], global memory bandwidth per stream processor is ~0.14 bytes per cycle. Each thread compares its own input string to the same test pattern, so this would most likely benefit from L1 caching, which can transfer an average of one byte per cycle. Transferring 1,024 bytes uncached and 256 bytes cached would therefore require 7,570 cycles. In contrast to the large number of memory transactions, the amount of computation in this function is almost negligible. Problems such as this where the GPU spends most of its time servicing memory requests are known as memory-bound. This makes it clear that optimisations should target improving the efficiency of memory transactions. The aggregate time and space complexities for this algorithm can be represented as $(\sum_{i=1}^{t} O(k_i n_i))/p$ and $\sum_{i=1}^{pd} O(m_i)$, respectively, where t represents the total number of comparisons, p represents the number of processors, and d represents how many comparisons each processor executes concurrently.

6.2.2 Bit-Parallel Implementation

Hyyrö's adaptation of Myers' bit-parallel algorithm [35], hereafter referred to as HBP, is used as the basis for this implementation, which is an extension of our previous attempt to accelerate this algorithm on the GPU [77]. Further adaptations of this algorithm by Hyyrö are specific to certain use cases and are thus not relevant to this work. The size of the word type was chosen to be a 32-bit unsigned integer for the GPU implementation rather than a 64-bit unsigned int as used in the CPU implementation. This is because the HD7970 is primarily built for 32-bit operations and only uses 32-bit registers [3]. Since the solution should work for texts greater than the size of the words used to store the column data, multiple words are used for each column vector. This adds a few extra operations as overflows between neighbouring words must be taken into account when addition and bit shift operators are used. Myers' solution to the cut-off heuristic was applied to this algorithm, where only the required word blocks are calculated, as illustrated in Figure 6.2.

This results in an $O(kn/w)$ expected time and $O(\alpha m/w)$ space complexity [34, 54]. The aggregate time and space complexities can be represented as $(\sum_{i=1}^{t} O(k_i n_i/w))/p$ and $\sum_{i=1}^{pd} O(\alpha_i m_i/w)$, respectively.

The HBP algorithm represented in Algorithm 6.5 requires considerably fewer global memory transactions than Algorithm 6.4. This is because its effective use of computer words means fewer memory transactions are needed to transfer the same amount of information, and more data can be stored in private registers. If the Ukkonen cut-off optimisation is omitted for the sake of comparison, this algorithm would result in $\frac{patternLen+31}{32} + 2$ unsigned integer (4 bytes) loads from local or global memory. A string of 256 characters would therefore result in 72 bytes read from global memory in 72 clock cycles (cached). In practice, it is only $b + 2$ with the cut-off optimisation, where b is the current number of active blocks (illustrated in Figure 6.2).

6.2.3 Parallelisation Approach

GPUs make extensive use of SIMD processing, which means they are well suited to running the same algorithm on large datasets. Depending on the algorithm, each GPU thread can be independent, or groups of threads can work cooperatively. Existing GPU solutions to a similar problem where an approximate match of a pattern is searched within a large text have used the latter approach. For this problem, we use the former in a data-parallel style, as the individual comparisons are too short to benefit from work-group parallelism.

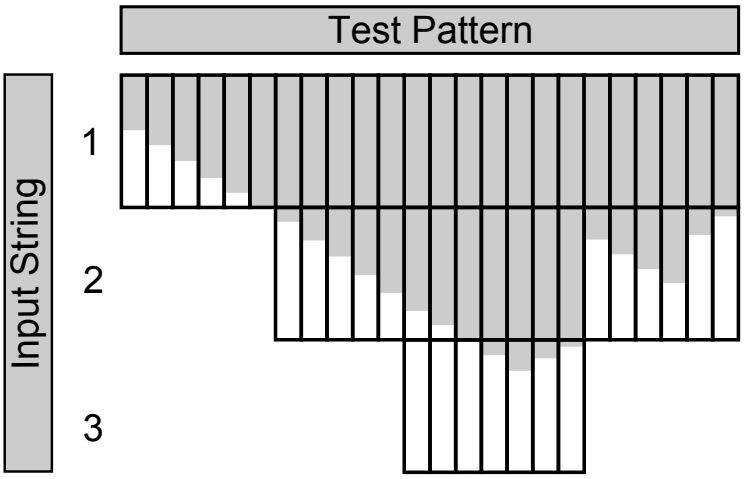


Figure 6.2: The dynamic programming matrix with columns divided into blocks the size of a word (adapted from [34]). The shaded region indicates the area that would be calculated using the cut-off heuristic without bit-parallelism. The blocks indicate the region that would be calculated using the cut-off heuristic incorporated into the bit-parallel algorithm.

Algorithm 6.5 A simplified representation of the GPU implementation of the HBP algorithm.

```
1: function ADVANCE_BLOCK(b, ref, PM, lastVP, lastVN, N, hin)
2:     ▷ b is the active block depth, ref is the string character to compare against, PM
         is the processed test string, lastVP contains the vertical positive changes from
         the last column, lastVN contains the vertical negative changes from the last
         column, N is the shorter string's length, and hin is a carry from a previous
         block.
3:     out ← 0
4:     X ← PM_{ref,b}
5:     if hin == −1 then
6:         X |= 1
7:     D0 ← (((X & lastVP_b) + lastVP_b) ^ lastVP_b) | X | lastVN_b
8:     HP ← lastVN_b | ~(D0 | lastVP_b)
9:     HN ← D0 & lastVP_b
10:    num ← (b + 1 == (N + 31)/32) ? N & 31 : 32
11:    if HP & 1 << num − 1 then
12:        out ← 1
13:    else
14:        if HN & 1 << num − 1 then
15:            out ← −1
16:    HP <<= 1
17:    HN <<= 1
18:    if hin == 1 then
19:        HP |= 1
20:    lastVP_b ← HN | ~(D0 |= 1)
21:    if hin == −1 then
22:        lastVP_b |= 1
23:    lastVN_b ← D0 & HP
24:    return out
25: end function
26: function CALC_COL(ref, PM, N, lastVP, lastVN, DT, b, strindex, k, pass)
27:    ▷ ref is the character in the input string currently tested against, PM is the
         processed test string, lastVP contains the vertical positive changes from the
         last column, lastVN contains the vertical negative changes from the last column,
         DT stores block error count values, b is the active block depth, strindex is the
         current index of the input string, k is the maximum k value, and pass is an
         output parameter used to indicate a positive or negative match.
28:    carry ← 0
29:    for i ∈ 0..b do
30:        carry ← advance_block(i, ref, PM, lastVP, lastVN, N, carry)
31:        DT_i ← DT_i + carry
32:    if DT_{b−1} − carry ≤ k and b < (N + 31)/32 and PM_{ref,b} & 1 then
33:        b++
34:        for i ∈ 0..32 do
35:            lastVP_{b−1} |= (1 << i)
```

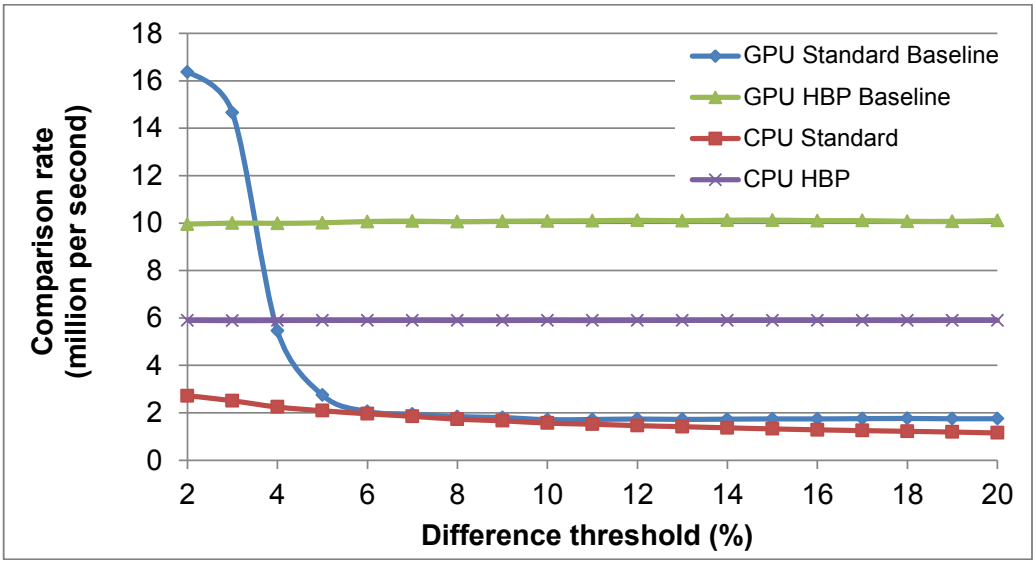
```
36:         lastVN_{b-1} ← 0
37:         DT_{b-1} ← DT_{b-2} + 32 − carry +
                advance_block(b − 1, ref, PM, lastVP, lastVN, N, carry)
38:     else
39:         while DT_{b-1} ≥ k + 32 do
40:             b--
41:     if b == (N + 31)/32 and DT_{b-1} ≤ k then
42:         pass ← 1
43:         return
44:     if strindex == N − 1 then
45:         if b == (N + 31)/32 and DT_{b-1} − b × 32 − N ≤ k then
46:             pass ← 1
47:         else
48:             pass ← −1
49: end function
```

6.3 Results

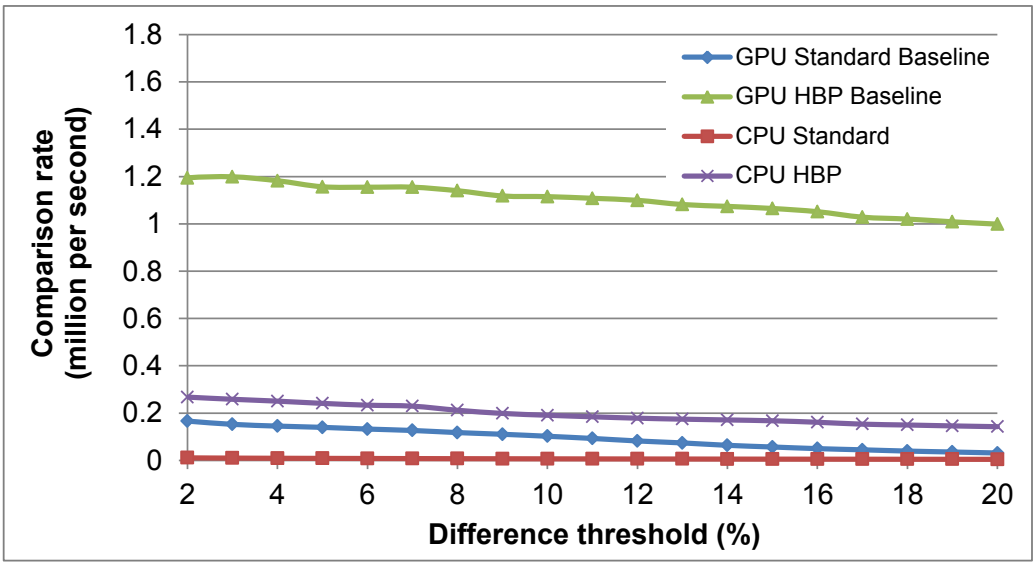
The GPU implementations were benchmarked against the same algorithms implemented for the CPU in the C programming language. Four categories of test data were used: short texts of no more than 64 characters, long texts of between 256 and 560 characters, a small alphabet of 4 characters, and a large alphabet of 64 characters. For brevity, transitional results only include the extreme combinations of short texts with a small alphabet and long texts with a large alphabet. Each dataset was evaluated with threshold (k) values from 2% to 20% of the string length. Sample datasets were used for the different alphabet and string length configurations. The datasets consisted of 8,192 strings, which were compared to a sample of 500 strings when testing long strings and 2,000 strings when testing short strings. The results for the baseline implementations of both algorithms can be seen in Figure 6.3.

Both baseline GPU implementations were faster than their CPU counterparts, but not by much. The GPU speedup was particularly low for short strings, where the average speedup was 1.8x for the standard algorithm[2] and 1.7x for the HBP algorithm. Longer strings resulted in comparatively better performance for the GPU, with the average speedups increasing to 12.7x for the standard algorithm and 5.8x for the HBP algorithm. However, these GPU implementations were naïve and stood to benefit from a number of optimisations.

[2]If the first three data points are omitted from the calculation, this is reduced to 1.2x.



(a) The comparative CPU and GPU performance for short strings using a small alphabet.



(b) The comparative CPU and GPU performance for long strings using a large alphabet.

Figure 6.3: The baseline performance of both algorithms for short strings (a) and long strings (b).

6.3.1 Optimisations

The performance of both algorithms was greatly improved through the implementation of a number of optimisations.

Data Layout

It is important to ensure that any accesses to global memory are designed as efficiently as possible to achieve good performance [3], especially when this memory is accessed often. Each GPU thread calculates the k-difference value between a common test pattern and its assigned input string, both of which are stored in global memory. The standard algorithm also uses global memory to temporarily store two columns in the dynamic programming matrix. The initial implementations of these algorithms simply stored the string and column data linearly in a section of global memory assigned to each thread. According to the AMD guide, global memory throughput is maximised on the HD7970 when each wavefront accesses consecutive groups of 256 bytes (4 bytes per thread), as this results in each wavefront accessing a different memory channel [3]. However, it was found that reading 1,024 bytes per wavefront resulted in better performance[3]. This was achieved by writing the input string and column data to global memory in the order in which they were read.

Caching

Although each compute unit on the HD7970 has 16 KB of L1 cache, the explicit caching of global memory data in registers can increase performance. This is because registers can serve up to 12 bytes per cycle compared to a single byte per cycle from the L1 cache, an order of magnitude difference. To ensure the string data in global memory was read as fast as possible, the string characters were loaded in batches into temporary private memory (registers). The data layout optimisation meant that each load from global memory was 16 bytes. The highest throughput from global memory was achieved when multiple loads of 16 bytes were cached in private memory, but it was only practical to cache a single load when other components of the algorithm were added because of the limited number of registers available. Another advantage of using registers for cache is they are not prone to cache misses.

[3]The AMD guide also mentions that there are 12 memory channels on the HD7970, and increments of 256 bytes usually result in the use of a different memory channel. Since 12 is not a power of two, 16 consecutive sections of 256 bytes would need to be read to access all the memory channels. Therefore, it makes sense that reading 1,024 bytes instead of 256 bytes per wavefront would result in superior performance, since doing so would result in the use of all 12 of the memory channels instead of only 10.

Vector Types

Previous generation AMD GPUs that use the very long instruction word (VLIW) architecture can benefit substantially from vectorisation as a result of more efficient use of the VLIW processing units [3]. The HD7970 used in this study no longer uses the VLIW architecture and thus does not benefit from explicit vectorisation [3]. However, the use of certain vector types can still improve performance as a result of their clear identification of 16 or 32 contiguous bytes of memory that can make use of 16-byte memory operations [3]. To leverage the memory efficiency benefit of vector types, `char16` and `short16` vector types were used instead of `char` and `short` arrays in the standard algorithm. As a side effect of this optimisation, loops had to be manually unrolled because vector components are not enumerable.

Loop Unrolling

Loop unrolling (otherwise known as loop unwinding) is used in many high performance programs. It is the practice of minimising or removing loop control flow logic by directly embedding multiple iterations of the loop into the code, which can reduce dynamic instruction count, improve ILP, and exploit memory locality [28, 53, 75]. It has been shown to provide significant improvements in performance if implemented correctly [53, 81]. However, loop unrolling can result in increased register and instruction cache usage, possibly leading to instruction cache misses and spilled registers [75]. This optimisation was attempted on the standard algorithm as it utilised many loops containing few instructions.

Intra-Group Cooperation

Although each thread processes its own k-difference problem, one common element between the threads is the test pattern. This was leveraged by using intra-group thread cooperation to collectively read the relevant test string from global memory. In the standard algorithm, this string was then stored in local memory to be used by all the threads within the group. Since each thread reads the same test pattern from local memory during the running of the algorithm, local memory loads benefited from broadcast reads [3]. The HBP algorithm first required the string to be processed into bit-vector arrays (in the form of unsigned integers) before the processed string could be saved in local or global memory.

Scheduling

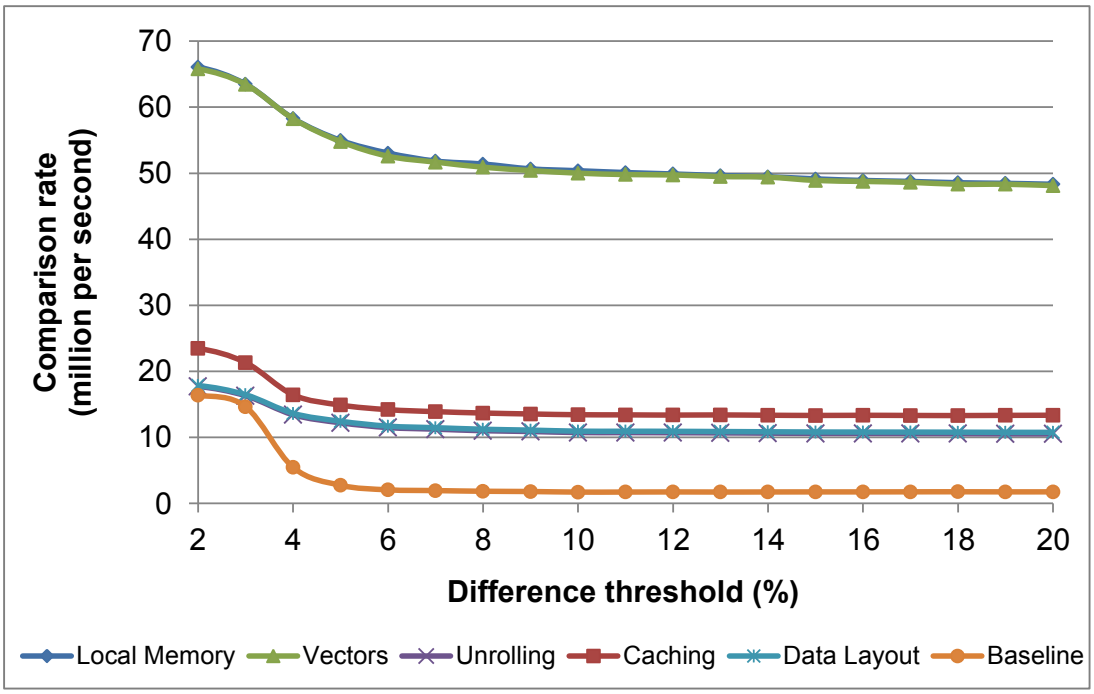
An advantage of OpenCL is that the programmer can specify a very large NDRange and let the hardware manage its execution. However, we found that scheduling a set NDRange yielded optimum performance if the NDRange was calculated using the expected number of wavefronts scheduled per GPU compute unit, as this reduced scheduling overhead. Using this method, each thread performs multiple k-difference calculations.

Simple Algorithm Optimisations

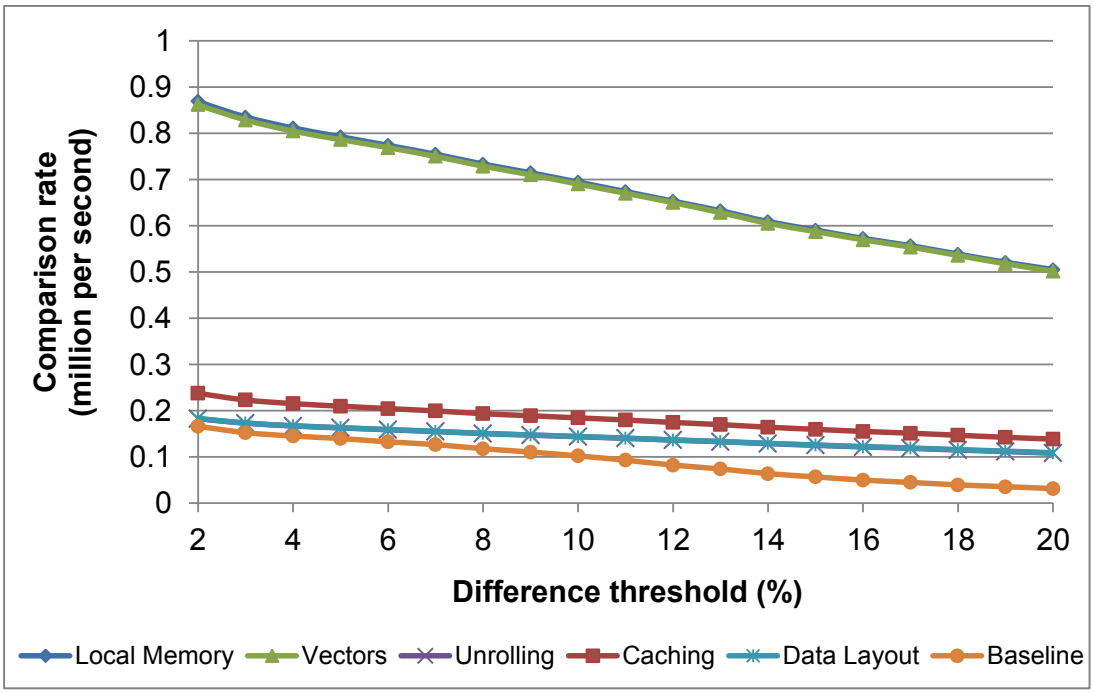
The performance improvements observed by implementing the optimisations described above are illustrated in Figure 6.4(a) and (b) for short and long strings, respectively. The data layout optimisation resulted in a 12% to 514% improvement for short strings and 14% to 246% improvement for long strings. Comparisons between the longer strings at higher difference thresholds benefited the most from this optimisation as they transferred a much greater amount of data to and from global memory. Loop unrolling reduced the performance by 1% to 2% for short strings and under a percent for long strings, which is not too surprising considering the bottleneck was with memory latency rather than control flow overhead. Caching provided the third biggest improvement of between 24% to 31% for short strings and 28% to 29% for long strings, even though it resulted in four registers being spilled into global memory. The vector data type optimisation surprisingly gave the second biggest performance improvement of between 197% and 272% for short strings and 262% to 276% for long strings. The impact of the final local memory optimisation was negligible (under a percent). Although local memory is considerably faster than global memory, only ~11% of the global memory data used in the core algorithm could be placed in local memory and this had to be initialised from global memory. Furthermore, the global memory accesses are likely to benefit from the L1 and L2 caches, since all work-groups would be accessing the same test strings in the same order.

HBP Algorithm Optimisations

The performance improvements observed by implementing the optimisations described above, with the exception of loop unrolling and vector types, are illustrated in Figure 6.5(a) and (b) for short and long strings, respectively. The loop unrolling and vector type optimisations were not as relevant to this implementation because of differences in

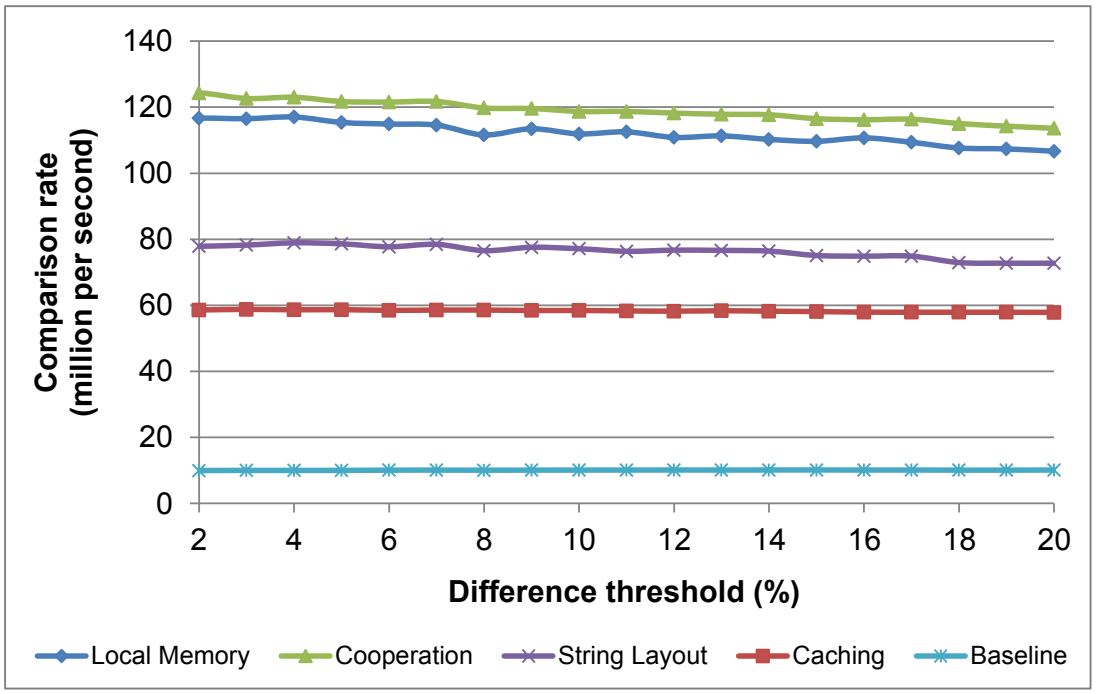


(a) The impact of individual optimisations on the standard implementation for short strings with a small alphabet.

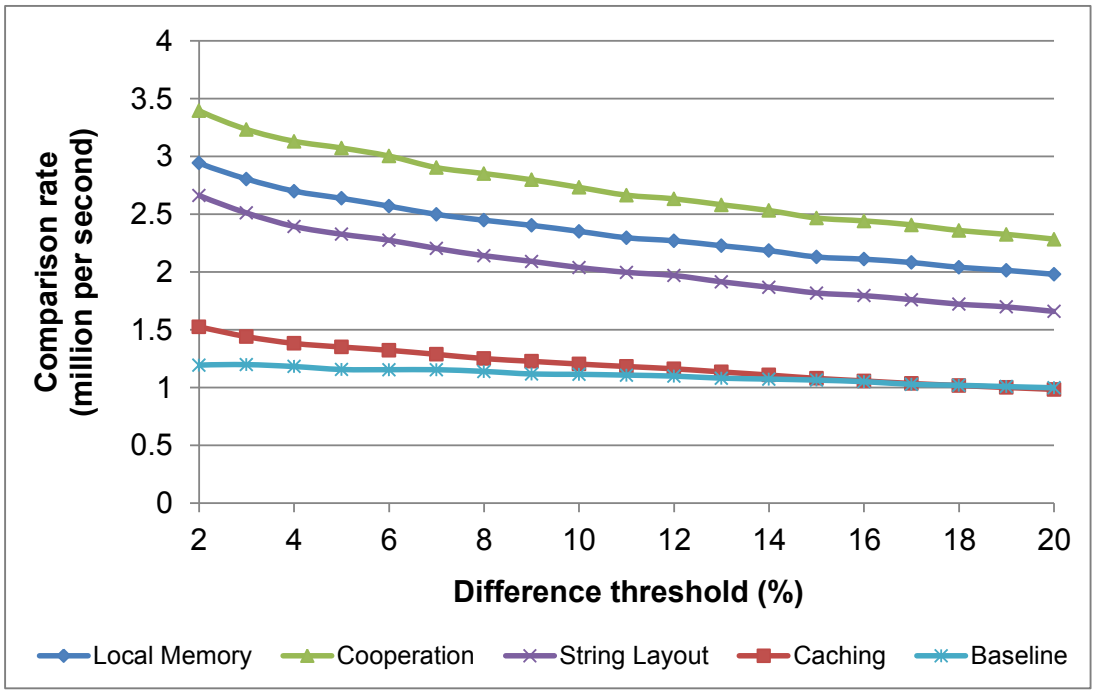


(b) The impact of individual optimisations on the standard implementation for long strings with a large alphabet.

Figure 6.4: The impact of different optimisations on the performance of the standard algorithm.



(a) Impact of individual optimisations on the HBP implementation for short strings with a small alphabet.



(b) Impact of individual optimisations on the HBP implementation for long strings with a large alphabet.

Figure 6.5: The impact of different optimisations on the performance of the HBP algorithm.

the algorithm implementation. The data layout optimisation resulted in a 620% to 690% improvement for short strings and 66% to 109% improvement for long strings. Unlike what was observed with this optimisation in the standard algorithm, its impact decreases with longer strings and higher difference thresholds. When the algorithm operated on short strings of less than 64 characters, only two unsigned integers were required to cover each position in the string (2 x 32 bits). This meant that there was little variance in the section of PM array (containing the processed test string) read by each work-item since successful matching could only result in a depth increase of one unsigned integer. The data layout optimisation ensured that accesses to the PM array also had high spatial locality. However, the longer strings required up to 18 unsigned integers to cover each position in the string. The different matching success rates of the string comparisons could thus result in a high variance in the depth of the PM array read by each work-item, and higher difference thresholds permitted larger gaps between the lowest permissible depth and the highest (perfect match). This significantly reduced the regularity of the memory reads and thus the effectiveness of the data layout optimisation.

Caching of string data resulted in a 20% to 26% performance decline for short strings and a 41% to 43% performance decline for long strings. Better caching is usually expected to result in better performance, but the implementation of caching required the use of additional GPU registers. GPU kernel profiling revealed that the use of extra registers resulted in 60 registers being spilling into global memory, which was responsible for the decline in performance.

Intra-thread cooperation in reading test strings improved the performance of short strings by 96% to 110% and long strings by 124% to 133%. Saving the cooperatively read and processed test strings into local memory instead of global memory reduced the performance by between 5% to 7% for short strings and 13% to 14% for long strings. This is contrary to local memory performance benchmarks done using a similar memory access pattern in which local memory was just over 2x faster than global memory. Despite our efforts, we have yet to discover the reason for this performance anomaly.

6.3.2 Optimised Results

The final optimised versions of the GPU implementations were compared to their CPU-based counterparts for small and short alphabets and short and long strings. The standard GPU implementation included the data layout, caching, vector type, and local memory

optimisations, while the HBP GPU implementation included the data layout and thread cooperation optimisations.

Standard Algorithm

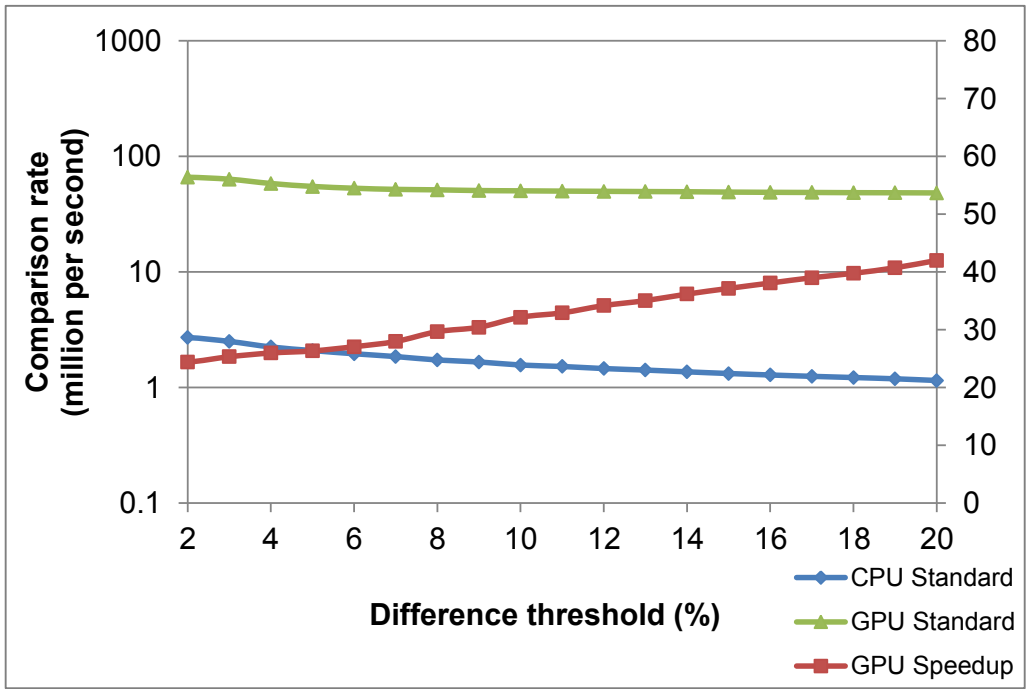
The similarity of graph (a) to (b) in Figures 6.6 and 6.7 indicates that alphabet size has a minimal impact on the relative performance of the CPU and GPU for the standard algorithm. The average GPU speedup differed by less than a percent between performance tests on the small and large alphabets. Since the algorithm simply compares one character to another numerically, this is unsurprising.

A clear trend that can be seen from the results is the decrease in comparison throughput with higher difference thresholds. This is the result of previously failed comparisons requiring additional work to reach a result because of the increased tolerance for mismatches. As indicated by the speedup curve, the performance of the GPU does not decline as fast as the CPU's performance with higher difference thresholds. From the lowest threshold to the highest, the CPU's throughput decreases by 2.4x while the GPU's throughput decreases by 1.5x.

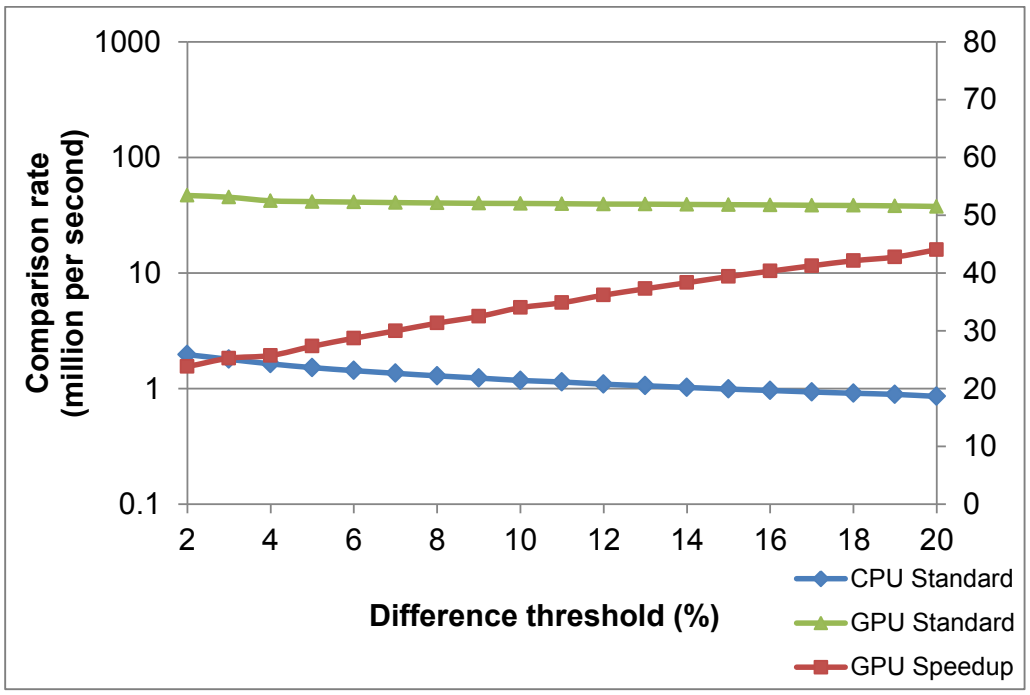
The number of string comparisons per second decreases considerably from short strings (Figure 6.6) to long strings (Figure 6.7). However, the GPU's speedup over the CPU more than doubles, increasing from an average of 33x to an average of 100x. In many ways, increasing the string length is synonymous with increasing the acceptable difference threshold, since both result in an increased average number of comparisons per string. It is therefore clear that the performance of the GPU implementation scales considerably better than the CPU implementation with larger problem sizes and difference thresholds for this algorithm.

HBP Algorithm

Unlike the standard algorithm results, the larger alphabet resulted in a decrease in GPU speedup for the HBP algorithm. This can be seen when comparing graph (a) to (b) in Figures 6.8 and 6.9. The comparison rate of the GPU decreased by an average of 9.8%, while the CPU comparison rate only decreased by and average of 2.7%, resulting in a decrease in the average GPU speedup of 7.8%. The GPU profiling information revealed a ~15% average reduction in the number of active stream processors for the dataset with

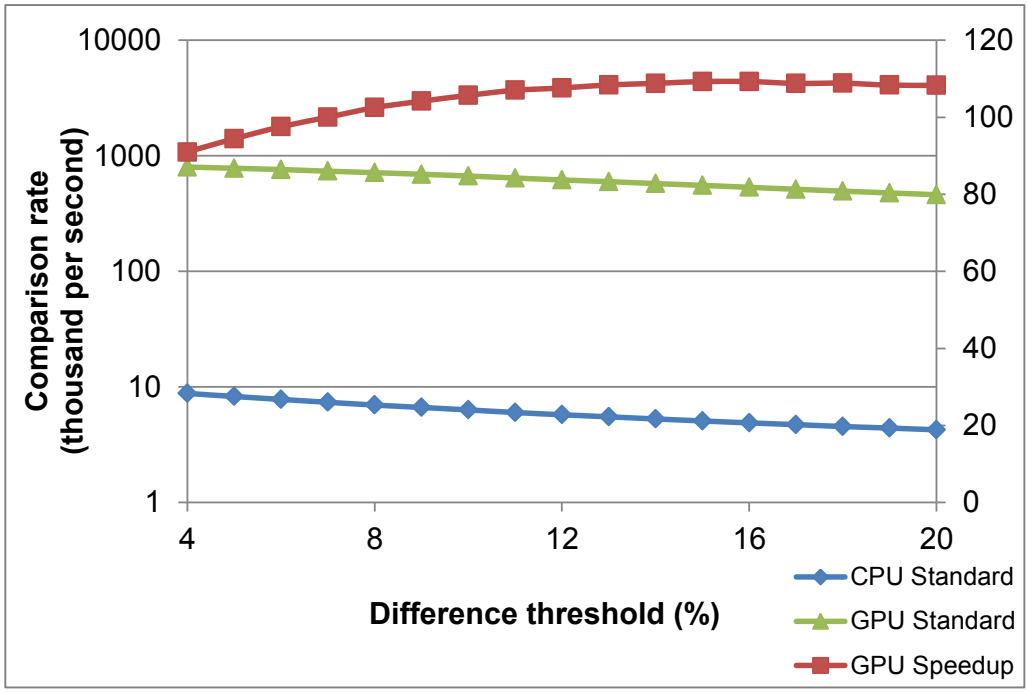


(a) The standard algorithm tested with a small alphabet and short strings.

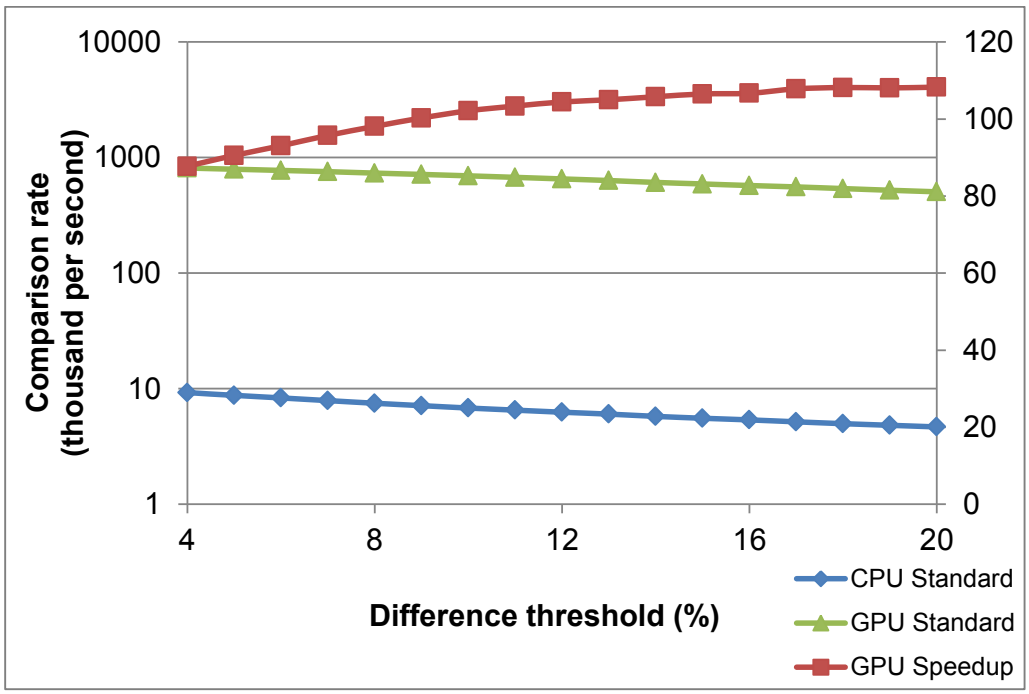


(b) The standard algorithm tested with a large alphabet and short strings.

Figure 6.6: The performance of the standard algorithm for short strings, small and large alphabets, and varying difference thresholds. The comparison rate of the GPU and CPU (green and blue lines) are labelled on the left y-axis, and the GPU speedup (red line) is labelled on the right y-axis.

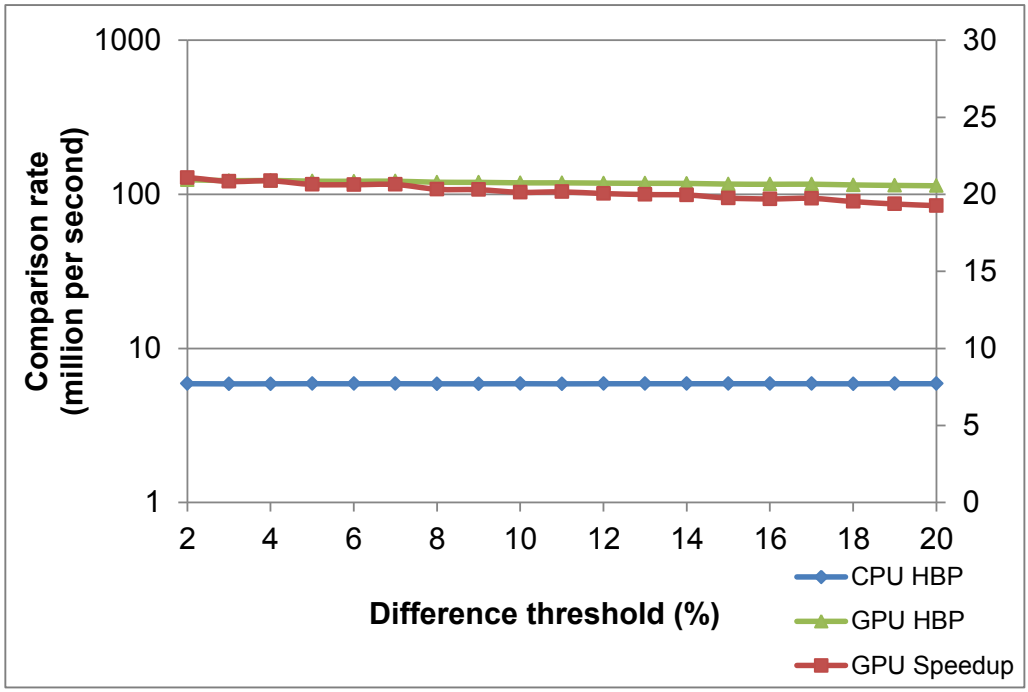


(a) The standard algorithm tested with a small alphabet and long strings.

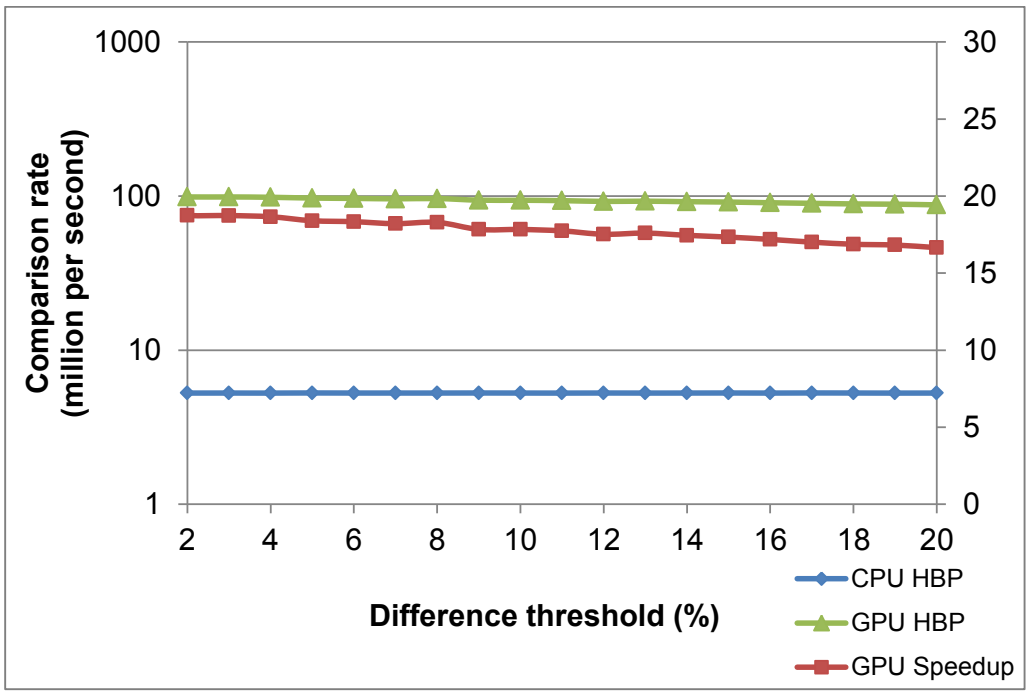


(b) The standard algorithm tested with a large alphabet and long strings.

Figure 6.7: The performance of the standard algorithm for long strings, small and large alphabets, and varying difference thresholds. The comparison rate of the GPU and CPU (green and blue lines) are labelled on the left y-axis, and the GPU speedup (red line) is labelled on the right y-axis.

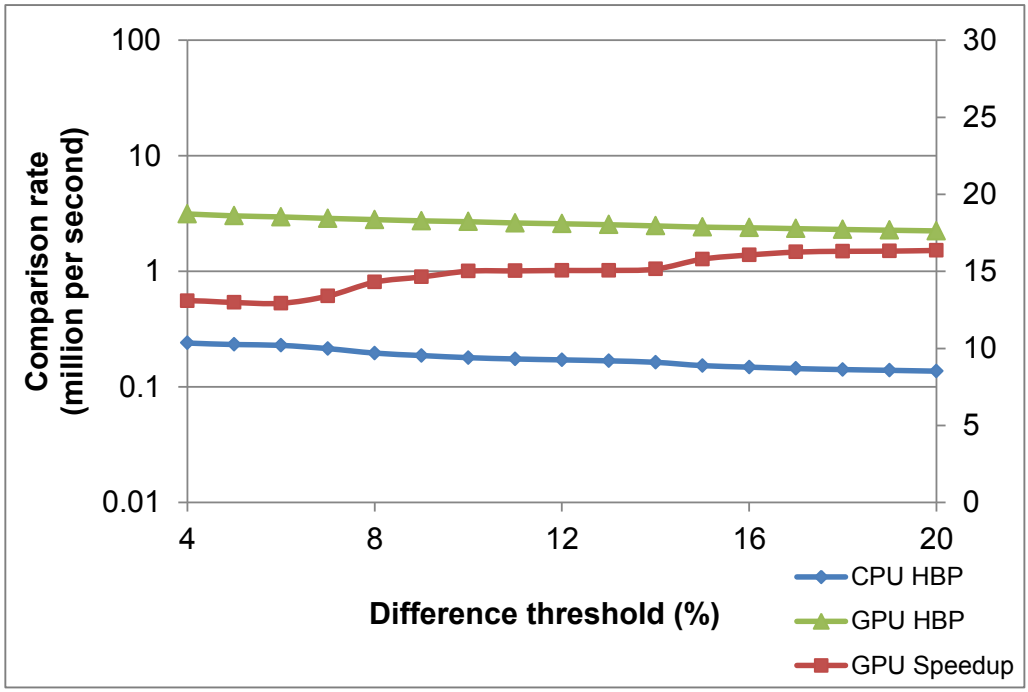


(a) The HBP algorithm tested with a small alphabet and short strings.

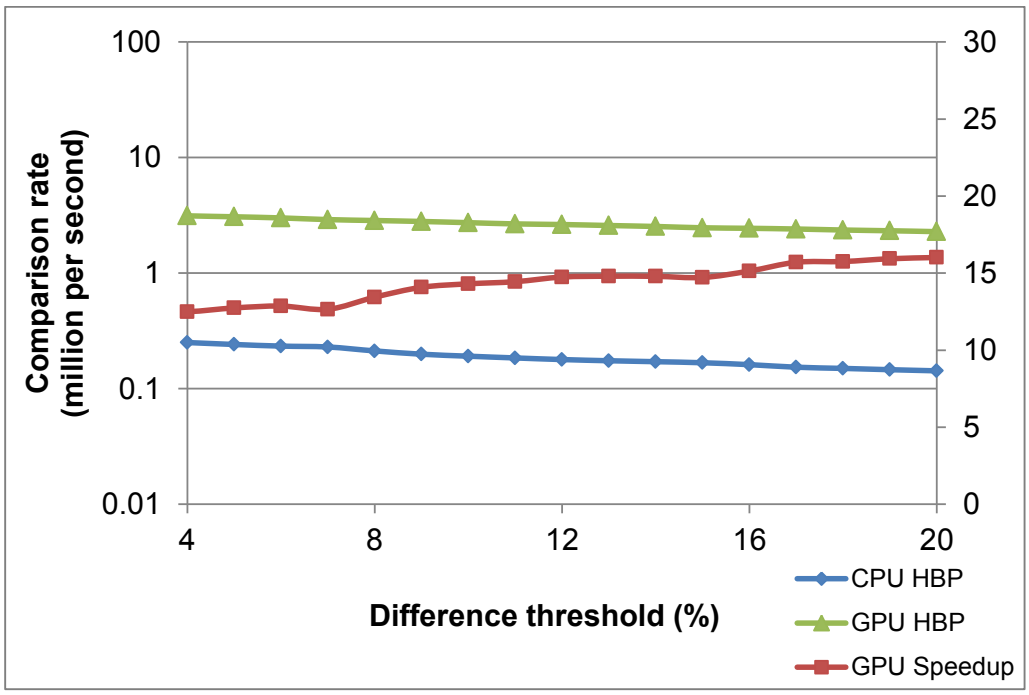


(b) The HBP algorithm tested with a large alphabet and short strings.

Figure 6.8: The performance of the HBP algorithm for short strings, small and large alphabets, and varying difference thresholds. The comparison rate of the GPU and CPU (green and blue lines) are labelled on the left y-axis, and the GPU speedup (red line) is labelled on the right y-axis.



(a) The HBP algorithm tested with a small alphabet and long strings.



(b) The HBP algorithm tested with a large alphabet and long strings.

Figure 6.9: The performance of the HBP algorithm for long strings, small and large alphabets, and varying difference thresholds. The comparison rate of the GPU and CPU (green and blue lines) are labelled on the left y-axis, and the GPU speedup (red line) is labelled on the right y-axis.

a large alphabet. The GPU performance decrease is thus likely the result of increased thread divergence from differences in the data rather than a larger alphabet.

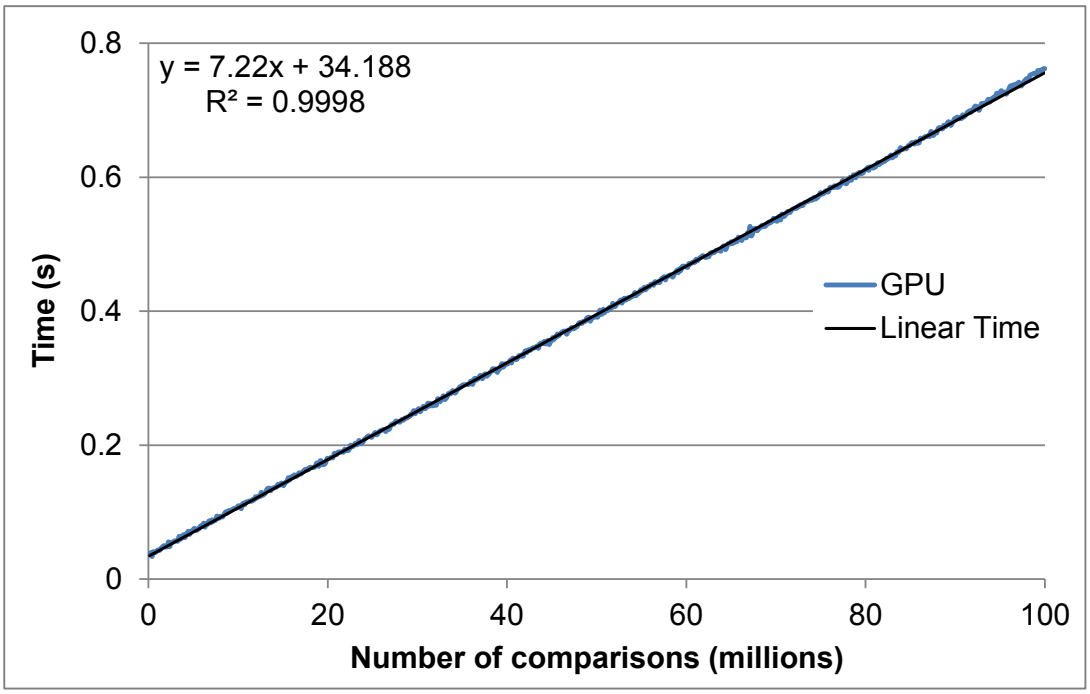
The impact of higher difference thresholds on the GPU speedup was mixed. For short strings, the CPU performance remained constant as the difference threshold increased, while the performance of the GPU gradually declined. Both the CPU and GPU performance declined with higher difference thresholds when operating on long strings, but the CPU performance declined more rapidly. The consistent performance of the CPU with higher thresholds on the small alphabet dataset is contrary to the norm. However, it is explained by the fact that the CPU implementation uses 64-bit unsigned integers for storing the bit-vectors, which contain a sufficient number of bits to represent each position in the short strings. This means that higher difference thresholds do not result in the need for additional bit vectors to be read. The same does not apply to the GPU implementation since it uses 32-bit unsigned integers.

Comparisons between long strings (Figure 6.9) reduced the performance of the GPU implementation more than they did the CPU implementation, resulting in a decline in GPU speedup from an average of 19x to an average of 14.4x. This is the opposite of what is typically expected from GPUs given their throughput-oriented architecture. GPU kernel profiling revealed the cause of this to be a 20% reduction in the GPU occupancy, which reduced the ability of the GPU to hide memory access latency. The occupancy decrease was caused by an increase in the number of registers needed to support the bit-vector arrays that represented the longer strings. Comparisons between even longer strings would exacerbate the issue and result in register spilling.

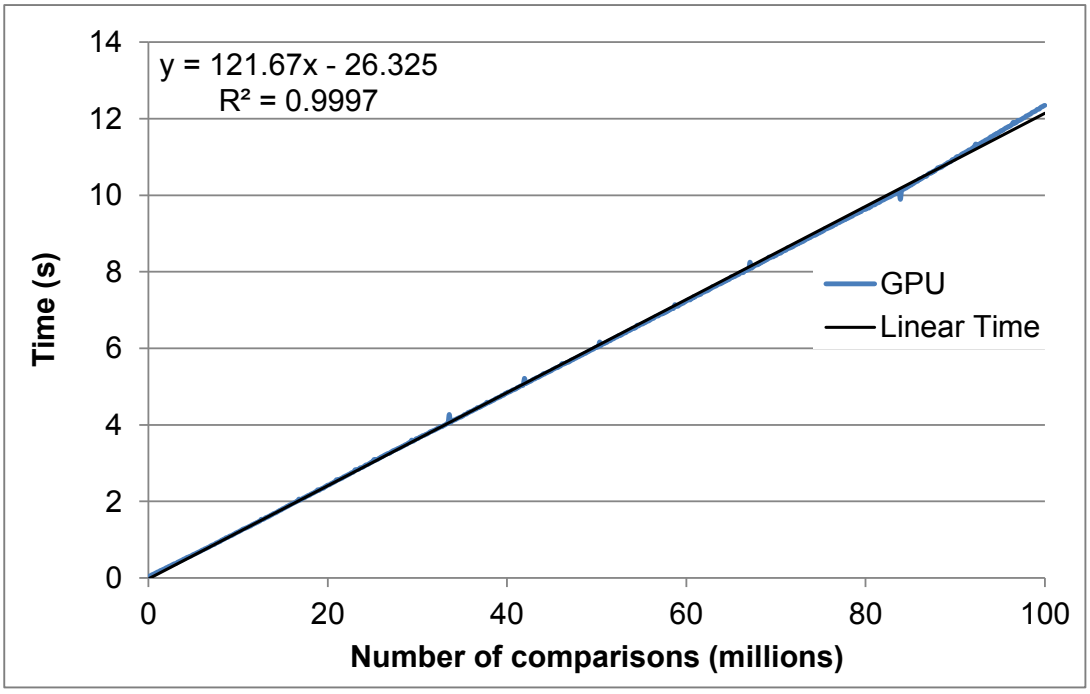
6.3.3 Impact of Problem Size

To determine the impact of problem size on GPU performance, the HBP algorithm was benchmarked with the number of required comparisons increasing from 131,000 to 100 million in increments of 131,000. Both short and long strings were tested using a small alphabet and a difference threshold of 4%. The results, given in Figure 6.10, show that the GPU's performance is remarkably consistent as the problem size increases. Given the strong evidence for a linear relationship between GPU execution time and problem size, the `LINEST`[4] function in Microsoft Excel was used to calculate the linear regression equation and coefficient of determination (R^2) statistic for (a) and (b), which are shown

[4]http://office.microsoft.com/en-za/excel-help/linest-HP005209155.aspx



(a) Comparisons between short strings.



(b) Comparisons between long strings.

Figure 6.10: The impact of problem size on the HBP algorithm when using a difference threshold of 4%.

on their respective graphs. Table 6.1 also gives the F probability and the standard m and c errors. The calculated R^2 statistics and F probabilities indicate that the execution

time for comparisons between both short and long strings can be predicted with the given linear equations within a 99% confidence interval. Graph (a) conforms to the linear equation better than (b) though, which is most likely due to the larger variance in possible comparison times permitted by comparisons between longer strings.

Table 6.1: Regression analysis for a linear relationship between problem size and performance.

String Length	R^2	F Probability	m Std. Error	c Std. Error
Short	0.999826	0	0.003448	0.199281
Long	0.999725	0	0.073084	4.224020

6.3.4 Data Transfers

Table 6.2: The data transfer overhead.

Test Strings	1,000	2,000	3,000	4,000	5,000
Small α, short strings	10.1%	6.8 %	6.2 %	6.5%	5.7%
Large α, long strings	0.3%	0.2%	0.2%	0.2%	0.1%

The benchmarks were carried out with the GPU program configured as dual dependent, which is the least desirable configuration for data transfers. The overall contribution of data transfers to the total runtime of the HBP GPU implementation for various configurations at a difference threshold of 4% can be seen in Table 6.2. These results show that the data transfer time impacts comparisons between short strings the most; the transfer time for short strings contributed up to 11.65% of the total GPU time compared to up to 0.16% for long strings. This difference is the result of the significantly longer comparison time between long strings compared to that of short strings, which reduces the contribution of data transfer overhead to overall GPU execution time.

6.4 Discussion

Prior to optimisation, the GPU implementations of both algorithms outperformed their CPU counterparts, but not by a significant margin. After the implementation of four optimisations, the performance of the standard algorithm improved by an average of

18.3x for short strings and 7.8x for long strings. Two optimisations improved the average performance of the HBP algorithm by 11.9x for short strings and 2.4x for long strings. Overall, these optimisations resulted in the GPU being between 23.9x and 109x faster than the CPU for the standard algorithm and between 12.5x and 21x faster for the HBP algorithm.

6.5 Summary

Determining whether two strings approximately match each other using the k-difference algorithm is an operation that has uses in many applications. However, because of the time complexity of the k-difference algorithm, it is not usually feasible to incorporate it in programs that require high throughput processing, such as malware detection and spam filtering. The goal of this study was to evaluate the applicability of GPU acceleration to large numbers of k-difference string comparisons using a standard dynamic programming matrix algorithm and a bit-parallel algorithm. The bit-parallel algorithm had already been accelerated on a GPU using OpenCL, but inefficiencies in the existing implementation were found and the speedup was considerably lower than we believed was achievable. Initial GPU implementations of these algorithms only resulted in a speedup over the CPU of between 1.8x and 12.7x for the standard algorithm and between 1.7x and 5.8x for the HBP algorithm. Through the implementation of two to four memory related optimisations, the speedups were improved to between 23.9x and 109x for the standard algorithm and between 12.5x and 21x for the HBP algorithm. The speedups obtained are likely to be significant for many applications of the algorithms, and could potentially make previously impractical applications feasible.

Chapter 7

Case Study 3: Radix Sort

The need for sorting is found in many computer science problems. Although highly optimised sorting algorithms have been developed for CPUs, sorting very large datasets such as those found in existing GPU applications can still be a performance bottleneck [50]. Unlike the problems in Chapters 5 and 6, sorting does not map easily onto the GPU's architecture because of the inherent and irregular data dependence between the records to be sorted [50]. However, using the right techniques, a modern GPU can achieve a sizeable speedup over a modern CPU for certain use cases.

7.1 Radix Sort Algorithms

A least significant digit radix sort algorithm works by repeatedly sorting the input keys based on increasingly higher value sections of the physical representation of the keys. Keys that have the same value at the same section of the key do not change positions relative to each other. The number of sections, and consequently sorting passes, is determined by the width of the section, otherwise known as the radix. A simple sequential radix sort of four values is illustrated in Figure 7.1, and a basic algorithm is given in Algorithm 7.1.

Efficiently parallelising this algorithm is significantly more challenging than parallelising the problems presented in Chapters 5 and 6, owing to the large amount of synchronisation and cooperation needed between participating threads. The basic parallel approach to radix sorting separates the problem into three steps. These steps, repeated for each radix segment of the key as outlined by Satish et al. [71], are:

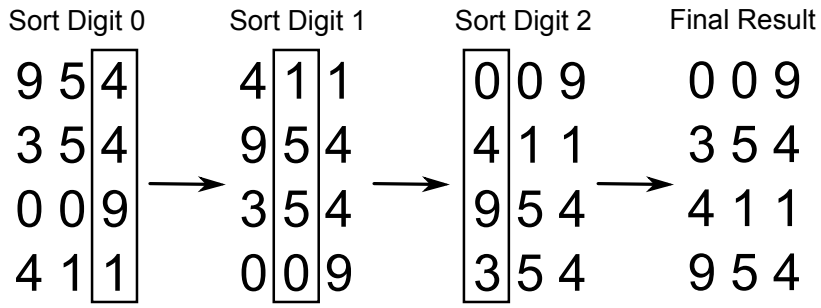


Figure 7.1: A simple illustration of the steps performed in a radix sort. In this example, the radix is a single digit, i.e., units, 10's and 100's are sorted in turn.

Algorithm 7.1 A basic sequential radix sort for 32-bit integers.

```
1:  function SeqRadixSort(keys, numkeys)
2:      radix ← 4
3:      numpasses ← 32/radix
4:      numbuckets ← pow(2, radix)
5:      ▷ Appopriate allocation and zeroing of index and bucket arrays.
6:      for sortpass ∈ numpasses do
7:          for i ∈ numkeys do
8:              ▷ getbits extracts the value from the key at current radix
9:              k ← getbits(keys_i, radix, sortpass * radix)
10:             buckets_{k,index_k++} ← keys_i
11:         keyindex ← 0
12:         for j ∈ numbuckets do
13:             for i ∈ index_j do
14:                 keys_keyindex ← buckets_{j,i}
15:                 keyindex++
16: end function
```

1. Assign an even portion of the input keys to each thread and make each thread calculate a private histogram of its allocated keys.

2. Use a parallel prefix sum to calculate a global histogram from the private histograms created by each thread.

3. Use parallel prefix sums to determine the offset for each key within each thread's individual histogram partitions, add this offset to the global partition offset calculated in Step 2, and scatter accordingly.

7.2 GPU Implementations

Two GPU radix sort algorithms were implemented for comparison. The first was a simple GPU radix sort based on a naïve C++ AMP implementation [12], and the second was a highly optimised GPU radix sort based on a CUDA implementation created by Merrill and Grimshaw (MG) [49]. A comparison of the performance and implementation of these two GPU radix sorts should provide some insight into the problem difficulty and knowledge required for successfully accelerating a radix sort.

7.2.1 Simple Radix Sort

The simple GPU radix sort [12] has three kernels, which are straight-forward implementations of the three steps outlined for parallel radix sorts in the previous section. However, there are two notable differences in the kernels for Steps 1 and 3. The first difference is that each GPU thread only processes a single key rather than a fraction of the total number of keys divided equally among the GPU processing elements. This approach simply scales the size of the NDRange with problem size, thereby taking advantage of the GPU's ability to handle large problem sizes and ensuring very high GPU occupancy. The second difference is the use of intra-group thread cooperation to produce aggregated work-group results in Step 1, which are then processed in Step 2 and used in Step 3. This is done to reduce the amount of data copied to and from global memory.

7.2.2 MG Radix Sort

To the best of the author's knowledge, MG's CUDA radix sort is the fastest implementation of a radix sort on GPUs. Much like the simple radix sort, it has three kernels corresponding to the three steps in a parallel radix sort algorithm and uses intra-group thread cooperation to aggregate thread results into work-group results. A very high-level overview of the algorithm implementation is illustrated in Figure 7.2. A more detailed description of the implementation can be found in [50].

The three phases, each implemented in a separate GPU kernel, are described in more detail below.

Kernel 1: The purpose of this kernel is to determine the aggregate number of keys that fall into the different bit-pattern buckets for each of the work-groups, where the keys

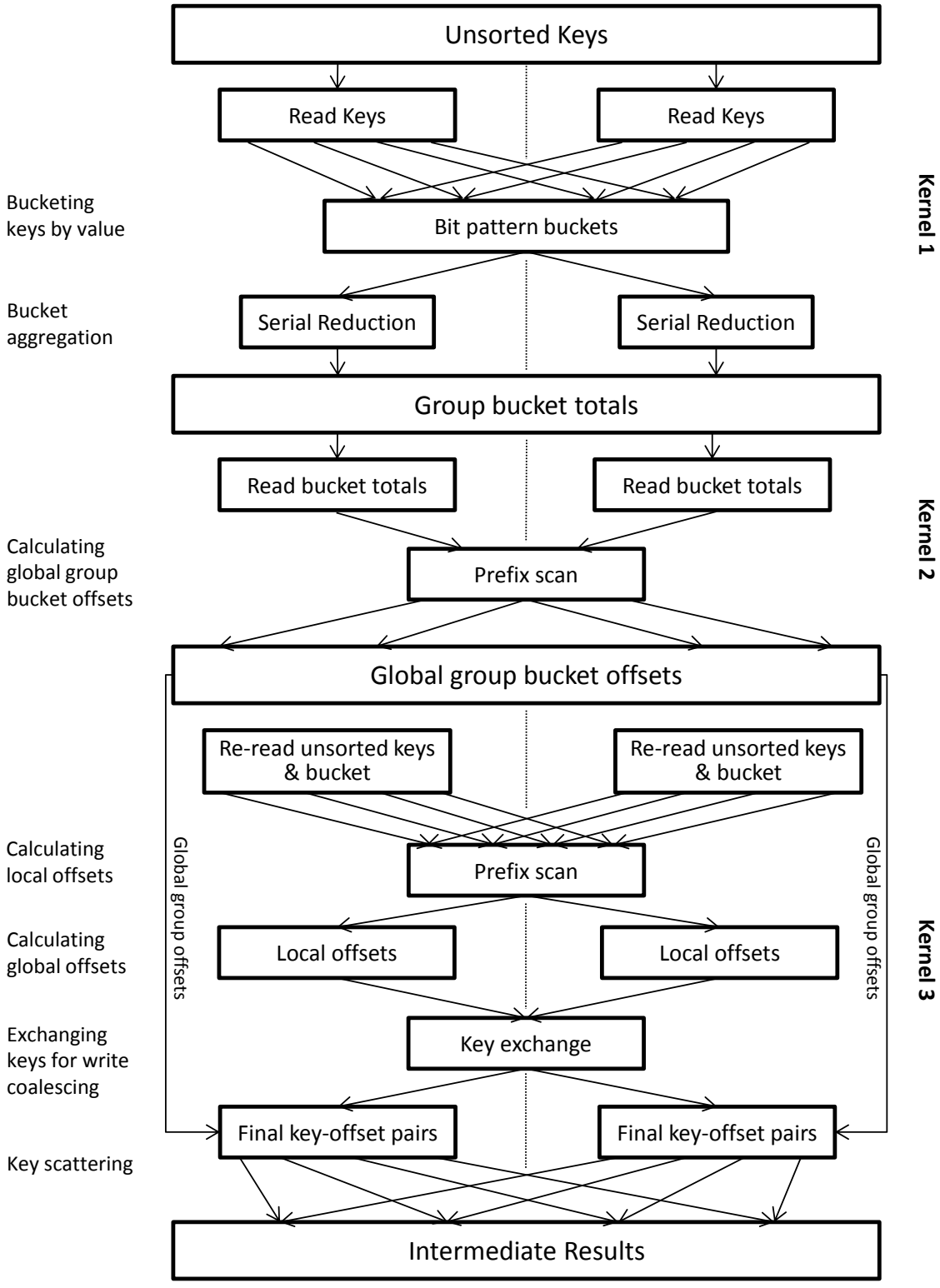


Figure 7.2: A high-level overview of the steps performed in Merrill and Grimshaw's GPU radix sort algorithm. The dotted line in the centre demarcates sections of independent computation.

are decoded based on the current offset in the key and the chosen radix. Each thread reads a number of keys from global memory and increments the appropriate bucket in local memory depending on the key's decoded value. The decoded bit pattern is simply the value of the relevant section of the key as determined by the radix and current sorting pass. The buckets in local memory are then serially reduced to obtain the final bucket counts for the work-group (or CUDA thread block), which are saved in global memory.

Kernel 2: This kernel performs a prefix scan of the bit-pattern buckets saved by the first kernel to obtain the global bucket offsets for each of the work-groups. This is done to provide each work-group with an offset in global memory for each bit-pattern bucket to which it can scatter its keys.

Kernel 3: The final kernel can be conceptually separated into four phases. In the first phase, the keys are re-read from global memory, decoded, and bucketed according to their bit pattern. This is essentially redundant computation since this was done in Kernel 1, but is repeated because it is faster than saving and loading the results to and from global memory. In the second phase, the threads within the work-groups cooperate to determine their inter-group bucket offsets using prefix scans. The third phase serves to optimise the scattering of the keys by performing an intra-group key exchange that results in the threads holding keys with offsets that would result in writes to global memory that are more ordered. The fourth and final phase adds the global group bucket offsets to the local offsets to get the final global memory offsets, and scatters the keys accordingly.

Since the kernels only operate on a section of the input keys, the results output by Kernel 3 are intermediate and are used as the input for the next iteration. The number of iterations required to fully sort the keys depends on the size of the key and the chosen radix. For a typical key size of 32 bits, eight iterations of the kernels would be needed to fully sort the keys using a radix of four.

Since the sort was implemented in CUDA, we had to reimplement a revision[1] of it in OpenCL for AMD's Tahiti range of GPUs to provide a fair comparison. Apart from the standard CUDA to OpenCL syntax changes, many of the required changes involved modifying sections of code to use a wavefront of 64 instead of 32 threads and modifying

[1]Revision 256 was used as the code in later revisions was significantly more fragmented and templated, making it harder to port to OpenCL.

GPU architecture-specific settings that governed the memory usage pattern of the algorithm. Even though the algorithm was explicitly designed for the GPU, implementing it in OpenCL was still challenging because of the low-level optimisations that needed to be understood and adapted for the HD7970 architecture.

7.2.3 Comparison Between GPU Sorts

The simple radix sort and the MG radix sort both consisted of three phases and followed roughly the same steps for a parallel radix sort as listed in Section 7.1. Three of the biggest differences in the MG sort implementation were found to be the addition of a key exchange step in the final phase, scaling the amount of work done per thread with problem size rather than scaling the number of threads, and loading and processing multiple keys per thread rather than a single key at a time. Many of the other differences related to the MG sort's processing of multiple keys per thread and techniques used to perform the same operations as in the simple sort in a more efficient manner.

We identified a number of strategies and techniques used in the MG sort that greatly improve GPU utilisation. Some of these are described below.

Computational Granularity

Each GPU work-group processes a portion of the total number of keys, which is in the thousands for problem sizes large enough to warrant the use of GPUs. To ensure efficient use of both the stream processors and GPU memory, these keys are processed in batches of an appropriate size tied to the target GPU's architecture. Scheduling more parallel work per thread has been shown to maximise performance by hiding more memory latency [80]. Furthermore, throughout the radix sort algorithm the number of memory loads done prior to computation has been set to depend on the target GPU architecture. This is because different GPU architectures have different efficient ratios of computation to memory transactions as a result of different memory and stream processor configurations. It is therefore necessary to optimise the computational granularity based on the target GPU architecture for best performance.

Synchronisation-Free Cooperation

It is common for synchronisation barriers to be used when GPU threads write to local memory to ensure memory consistency is maintained. These barriers can unnecessarily reduce performance in situations where not all work-items are required to participate in the cooperation and the number of work-items exceeds the size of a wavefront. This is because wavefronts are usually able to process instructions independently of other wavefronts in the same work-group, and barriers prevent them from doing this [3, 73]. This was avoided by using synchronisation-free thread cooperation throughout the radix sort algorithm by keeping the number of threads participating in thread cooperation to within a single wavefront. Since wavefronts are executed atomically [3], synchronisation is not needed between the threads.

Loop Unrolling

Wherever it was possible, loops were unrolled either through a compiler directive or manually by using a tiered function hierarchy. With the tiered function hierarchy, a particular tier calls lower tier functions until the lowest function is reached, which contains the unrolled code. A basic example of this can be seen in Listing 7.1.

Memory Packing

When working with values much smaller than can be held by the value type in local or global memory, packing multiple values into a single word can be an effective way of reducing memory load. This technique is used with local memory repeatedly in the algorithm, where an integer is re-interpreted as four separate char values.

Kernel Fusion

In programs where there is a requirement to perform common operations on the GPU data, it may seem sensible to use existing optimised solutions provided by libraries such as Boost.Compute[2] for OpenCL, and the Data-Parallel Primitives Library[3] for CUDA, as has been done in previous solutions [70]. However, there is a significant performance penalty

[2] `https://github.com/kylelutz/compute`
[3] `https://code.google.com/p/cudpp`

```
void ProcessDataItem(int item); //Defined elsewhere

void Process4Items(int *data, offset) {
    ProcessDataItem(data[offset]);
    ProcessDataItem(data[offset + 1]);
    ProcessDataItem(data[offset + 2]);
    ProcessDataItem(data[offset + 3]);
}

void Process8Items(int *data, offset) {
    Process4Items(data, offset);
    Process4Items(data, offset + 4);
}

void Process16Items(int *data, offset) {
    Process8Items(data, offset);
    Process8Items(data, offset + 8);
}

void ProcessData(int numitems, int *data) {
    int offset = 0;
    while (offset + 16 < numitems) {
        Process16Items(data, offset);
        offset += 16;
    }
    if (offset + 8 < numitems) {
        Process8Items(data, offset);
        offset += 8;
    }
    if (offset + 4 < numitems) {
        Process4Items(data, offset);
        offset += 4;
    }
    for (int i = offset; i < numitems; i++) //Process remaining elements
        ProcessDataItem(data[i]);
}
```

Listing 7.1: An example of how a tiered function hierarchy can be used to increase the amount of work done per loop iteration.

for doing so as it requires that all the data be passed from one kernel instance to another through global memory. This implementation integrates all of the required operations into existing kernels, thereby reducing the aggregate memory workload by allowing the results from one step to be passed to the next through local or private memory [50].

Ordered Writes

In the final phase of the sort, the newly ordered keys are written back to global memory. Since the order of the keys may have changed significantly from their order when the

threads initially read them, this can result in highly scattered writes and thus low global memory throughput. The degree of orderliness of global memory writes can be greatly improved by exchanging keys within work-groups so that the keys within work-groups are ordered. Even though this requires additional computation and use of local memory, the benefit of improved global memory throughput far outweighs the cost of the additional work.

Pinned Memory

The need for GPU accelerated sorting is likely to only arise with very large datasets. Consequently, it is important to ensure that the transfer of data to and from the GPU is as fast as possible. To this end, pinned host memory was used to ensure the host could transfer data to and from the host memory regions using the maximum available memory bandwidth [3]. Pinned memory enables this since it is unpageable and has a fixed memory address [3].

7.3 Results

The performance of the GPU implementations were compared to `std::sort` found in Microsoft's standard template library[4] (sequential) and the highly optimised parallel CPU radix sort found in Microsoft's Parallel Patterns Library (PPL)[5]. The keys to be sorted were generated randomly, and the results were averaged over five runs for each problem size tested.

As shown by the results in Figure 7.3, the simple GPU sort performed poorly; even the PPL radix sort was on average 2.2x faster than the simple GPU sort for the tested problem sizes. The performance of the MG sort was significantly better – an average of 2.5x faster than the PPL radix sort.

Despite the GPU radix sort being by far the most optimised of the GPU implementations presented here, its performance benefit over an efficient CPU solution is considerably less than the order of magnitude speedups achieved by the other GPU problems presented in Chapters 5 and 6. Compared to the PPL radix sort, the performance of our adaptation of MG's radix sort ranged from a slowdown of 0.92x when sorting 2^{17} elements to a speedup of 3.7x when sorting 2^{25} elements.

[4] `http://msdn.microsoft.com/en-us/library/vstudio/c191tb28(v=vs.110).aspx`
[5] `http://msdn.microsoft.com/en-us/library/vstudio/dd492418(v=vs.110).aspx`

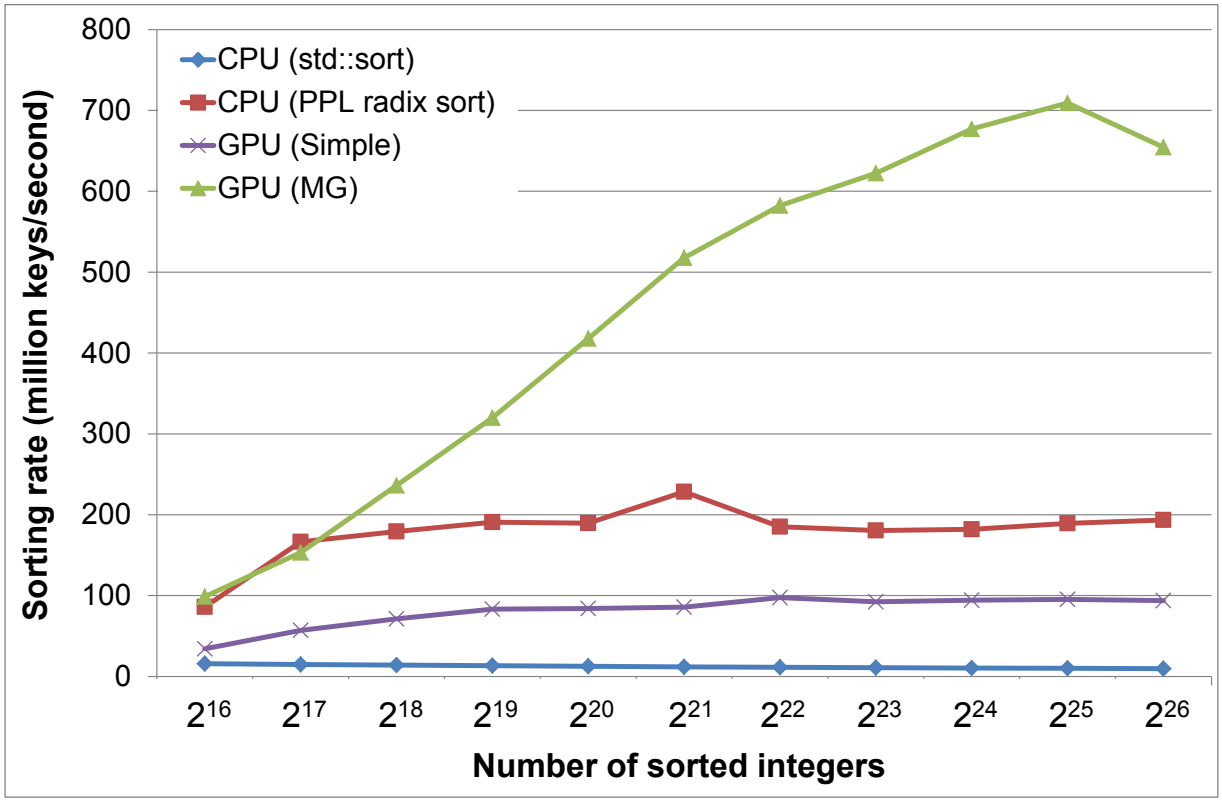


Figure 7.3: A comparison between the performance of the simple GPU radix sort, the MG radix sort, and two CPU sorting algorithms for different problem sizes.

7.3.1 Data Transfers

The MG radix sort results are considerably better if only the compute time of the GPU is considered. The results demonstrating this are illustrated in Figure 7.4.

Excluding the time it takes to transfer the data to and from the GPU, the GPU's advantage over the CPU ranges from 1.6x to 7.9x over the same problem sizes. An unfortunate limitation of discreet GPUs is that the transfer of data to and from the GPU can be a significant bottleneck, and indeed, this was the case for the GPU radix sort. For the smallest problem size tested, 2^{16}, data transfers accounted for 36% of the total GPU time, and for problem sizes greater than 2^{21}, the data transfer time actually exceeds the compute time and continually grows in proportion to compute time with larger problem sizes. This trend is shown in Table 7.1. The GPU's 'sweet spot' is when sorting 2^{25} elements; greater problem sizes result in a decrease in performance because the additional work no longer results in better GPU utilisation and creates more overhead from data transfers.

While the impact of data transfers on performance was severe for a standalone sort, other use cases provide opportunities to mitigate the impact of data transfers or remove the

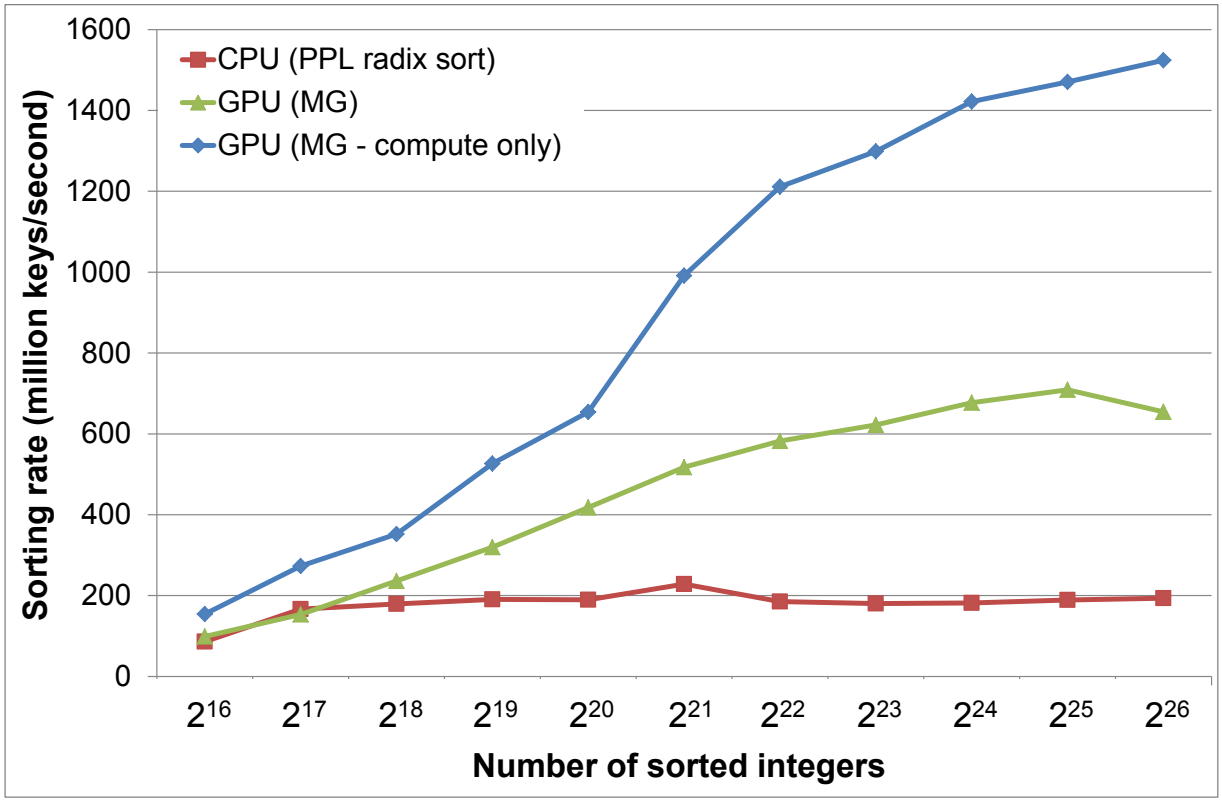


Figure 7.4: The performance of the MG radix sort when only compute time is considered, compared to the data transfer inclusive version and PPL radix sort.

requirement completely. In use cases where the radix sort is a component of a larger GPU program, it may not be required to transfer the input data or results between the host and GPU depending on where the sort is needed. For use cases where a number of datasets require sorting, the impact of data transfers can be reduced by overlapping data transfers with kernel execution. If this is done when the execution time exceeds the data transfer time, all but the first and last data transfers are hidden by computation, which can result in a sizeable performance improvement. This is illustrated in Figure 7.5 for three identical sorts, where (a) represents a completely sequential implementation and (b) represents an implementation using overlapped transfers. Each computation phase is preceded by data transfer for the input keys and followed by the transfer of the sorted results. As can be seen in (b), only the first and last data transfers are not hidden by

Table 7.1: The data transfer contribution to the total GPU time of a number of different problem sizes.

No. Keys	2^{16}	2^{17}	..	2^{25}	2^{26}
Transfer Overhead	36%	43%	..	52%	57%

computation, thus resulting in a significant reduction in overall execution time compared to (a).

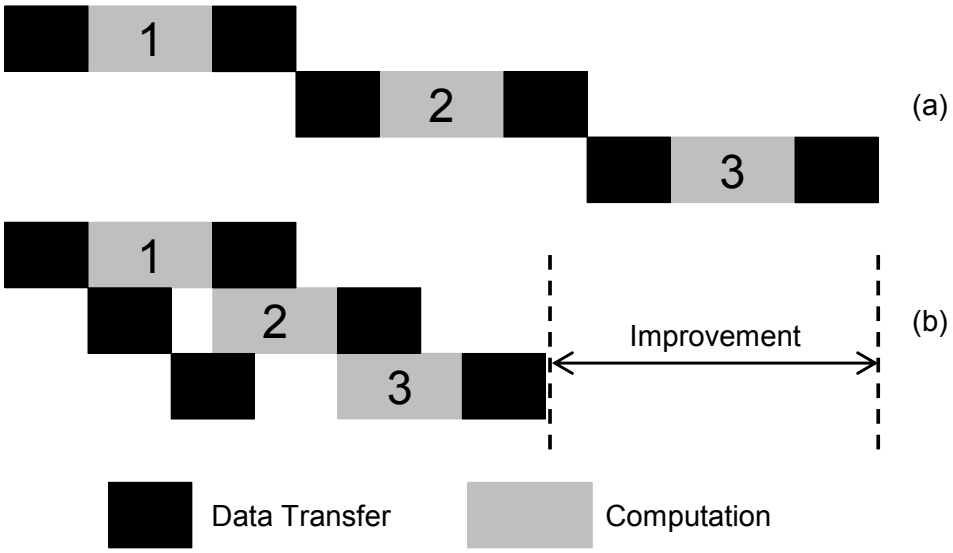


Figure 7.5: An illustration of how overlapped transfer with kernel execution hides data transfer overhead.

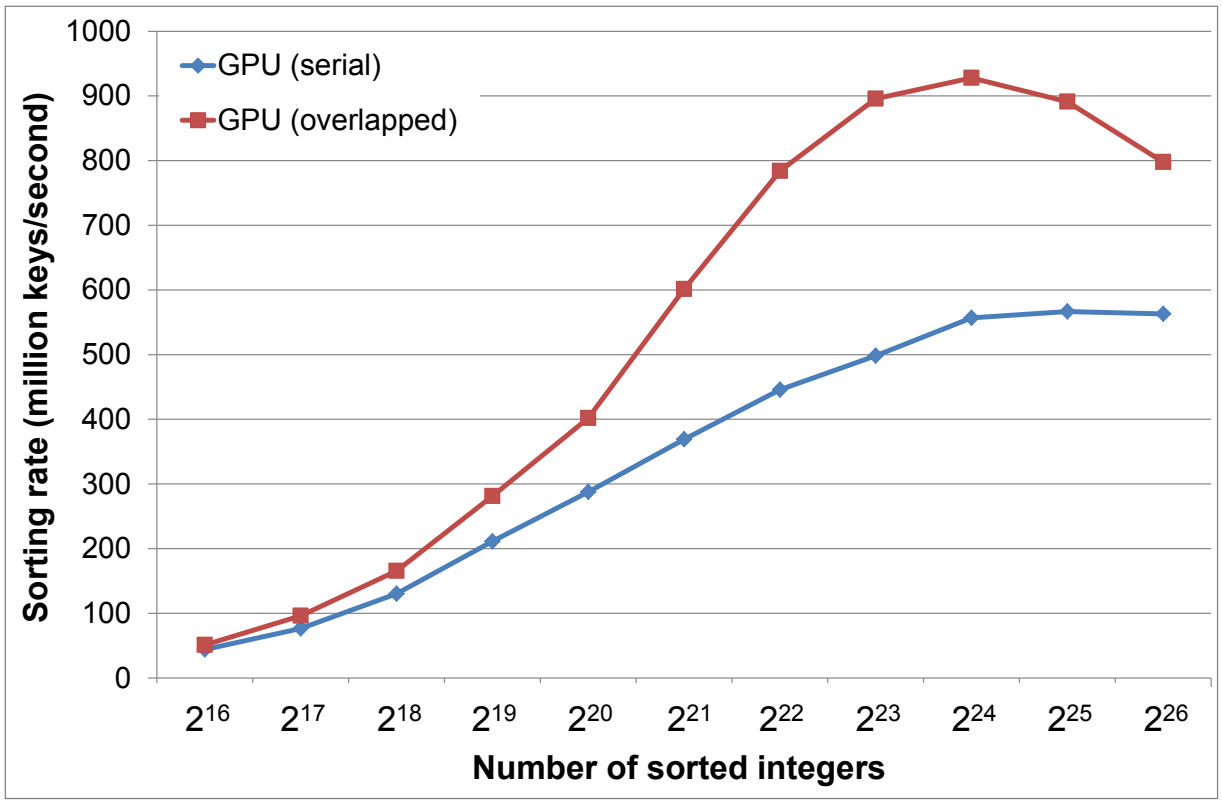


Figure 7.6: A comparison of how overlapped transfer and execution of multiple sorts compares to sequential scheduling.

To gauge the performance benefit of this optimisation for the MG radix sort, ten identical sorts were executed both sequentially and using overlapped transfers. The results in Figure 7.6 show a performance improvement from 1.16x when sorting 2^{16} elements to 1.8x when sorting 2^{23} elements. Larger problem sizes resulted in greater speedups, as the longer data transfer times impacted the sequential version significantly more than the version using the optimisation. However, the speedup was not sustained with problem sizes greater than 2^{23} elements, as the time spent on data transfers exceeded the time spent on computation, and thus the entire data transfer time could no longer be hidden.

7.4 Summary

Fast sorting is an operation that is required in many scientific programs. This chapter reviewed the radix sort algorithm and how it could be parallelised for GPUs to provide a speedup over an efficient CPU radix sort implementation. The parallelisation of the radix sort was seen to be significantly more complicated than the parallelisation of the problems in Chapters 5 and 6. Two GPU implementations were attempted. The first implementation was a relatively straight-forward implementation of the parallel radix sort steps on a GPU, while the second implementation was a conversion from a highly optimised CUDA radix sort. The optimised CUDA radix sort used techniques such as carefully selected computational granularity, synchronisation-free thread cooperation, loop unrolling, tight memory packing, kernel fusion, and ordered writes to ensure optimal performance. Pinned memory was also used to reduce the data transfer overhead. The results revealed the simpler implementation to be inadequate in providing a speedup over an efficient CPU radix sort, as it was on average 2.2x slower. The speedup of the optimised GPU radix sort ranged from a slowdown of 0.92x to a speedup of 3.7x. When only GPU computation time was measured, the speedup increased to between 1.6x and 7.9x faster, which shows that data transfer time between the host and GPU can be a significant contributor to the total GPU processing time. The speedup from using overlapped transfer and execution to reduce the impact of data transfers was between 1.16x and 1.8x when sorting 2^{16} and 2^{23} keys, respectively.

Chapter 8

Discussion

The wealth of published success stories of gaining orders of magnitude speedups through the use of GPU computing has undoubtedly caught the attention of many scientists. However, anyone new to the field of GPGPU could be forgiven for feeling apprehensive about the steep learning curve and low-level documentation, which act as barriers to entry into the field. This need not be so in all cases. To accelerate problems that do not map well to the GPU's architecture, it may be necessary to understand the low-level details of thread scheduling, memory transactions, and so forth. For problems that map relatively well to the GPU's architecture, having such in-depth knowledge of the functioning of the GPU is not a prerequisite for obtaining a satisfactory speedup, as was found with the uncertainty model discussed in Chapter 5. Consequently, it would be beneficial to be able to classify a problem as belonging to a particular GPU acceleration difficulty level, and identify guidance appropriate to the given classification. Here we expand on our preliminary work [76] on the identification of problem attributes influencing problem difficulty to provide a more complete solution. This chapter begins with an overview of the knowledge required to achieve the speedups obtained for each of the problems accelerated. This is followed by a detailed explanation of the problem attributes selected as problem difficulty indicators. Finally, the classification framework built around these difficulty indicators is discussed.

8.1 Reflection on Required GPGPU Knowledge

The problems accelerated in Chapters 5, 6, and 7 were found to require different levels of GPGPU knowledge to achieve the speedups obtained. This can be seen as corresponding

to problem difficulty. A brief summary of the knowledge required for each case study is outlined below.

8.1.1 Case Study 1: Hydrological Uncertainty Model

Of all the case studies, acceleration of the hydrological uncertainty ensemble model (Chapter 5) required the least amount of GPU-specific knowledge. The required knowledge can be broken down as follows:

OpenCL: A basic understanding of the OpenCL framework and API was needed to create a program capable of launching the required OpenCL kernels. This includes understanding the role and use of OpenCL platforms, contexts, devices, command queues, programs, kernels, buffers, and NDRanges. However, much of the boilerplate OpenCL initialisation and configuration could have been avoided if an OpenCL wrapper such as OpenCLHelper[1] had been used. To create the OpenCL kernels, knowledge of the C language was required, along with the syntax additions and added constraints of OpenCL C. It was also necessary to have a high-level understanding of the way in which OpenCL kernels are executed in parallel.

GPU Architecture: In porting this model to a GPU, a data layout optimisation was applied that significantly improved the program's performance. Such an optimisation could only have been attempted with knowledge of either the most efficient memory access pattern or the way in which memory requests are allocated to different memory controllers on the GPU.

8.1.2 Case Study 2: *K*-Difference String Matching

The speedup obtained in the acceleration of large numbers of k-difference comparisons (Chapter 6) was achieved as a result of several optimisations. The implementation of these optimisations required additional knowledge to that identified for the first case study.

[1] `https://github.com/hughperkins/OpenCLHelper`

OpenCL: Over and above the basic knowledge required for case study 1, acceleration of the standard algorithm also required knowledge of the properties of local memory.

Thread Cooperation: For best performance, threads were made to cooperate on reading data from global memory. To even be aware that such an optimisation was possible, it was necessary to understand the respective characteristics of local and global memory. Implementation of the optimisation further required knowledge of the memory consistency model of local and global memory, and the correct use of barriers.

GPU Architecture: The GPU architecture knowledge required for this problem built on the knowledge required for case study 1. To recognise the benefit of explicit caching, it was necessary to know the respective speeds of global and register or private memory. Furthermore, the effect of register usage on performance had to be understood to achieve a good balance between caching and GPU occupancy. Lastly, knowledge of vector types and the benefit of their use was required for an optimisation applied to the standard algorithm.

8.1.3 Case Study 3: Radix Sort

Unlike the problems in case studies 1 and 2, the acceleration of a radix sort (Chapter 7) was based on an existing GPU solution. The identification of the knowledge required for this case study was thus based on a combination of analysis of the existing implementation and the experience gained through re-implementation of the radix sort in OpenCL.

OpenCL: In addition to the knowledge required for case study 2, knowledge of the benefits of using pinned memory for transfers between the host and GPU and how this could be implemented was necessary.

GPU Architecture: The MG radix sort incorporated many additional GPU architecture specific optimisations, including optimised memory load granularity, synchronisation-free cooperation, and maximisation of GPU occupancy. Implementation of these optimisations required knowledge of the target GPU's memory architecture and processing speed, thread scheduling, number of compute units, and maximum number of wavefronts per compute unit.

8.1.4 Implication of Required Knowledge

From the discussion above, it is clear that the knowledge required for each subsequent case study expanded on the knowledge requirement of the previous case study, with the knowledge becoming increasingly specific to the target GPU. The need for additional knowledge is a direct result of the optimisations implemented to improve the baseline performance. This increasing knowledge requirement is thus relevant to problem difficulty, since in addition to the requirement of a greater understanding of GPGPU, it correlates with a higher degree of GPU-specific optimisation of an increasingly complex nature. It follows that if one could identify the need for such optimisations, the difficulty of GPU problems could be approximated.

8.2 Important Problem Difficulty Factors

Through the acceleration of the problems in Chapters 5 to 7, seven problem attributes relevant to problem difficulty have been identified, namely, inherent parallelism, branch divergence, problem size, required computational parallelism, memory access regularity, data transfer overhead, and thread cooperation. These difficulty indicators correspond with previously identified factors for GPU acceleration viability and performance projection [15, 48, 73]. It is easy to see the relation between GPU acceleration performance and our notion of difficulty in parallelising problems – factors that decrease estimated GPU performance would require more attention during solution optimisation. Each difficulty indicator is described below with reference to its relevance to the accelerated problems. The best methods for accurate quantitative measurement of the indicators are not readily apparent. This is an area that requires further research; however, possible methods have been suggested.

8.2.1 Inherent Parallelism

The traditional definition of inherent parallelism refers to the structure of an algorithm that enables it to be decomposed into a number of tasks that can be executed independently without sharing data [36]. This is similar to $(1 - f)$ in Amdahl's and Gustafson's laws. For the purposes of this study, a relaxed definition of inherent parallelism is used that places more emphasis on the quantity of parallel tasks and does not require complete

independence between the tasks. This work specifically considers the inherent parallelism of the algorithm used, and not that of the problem itself, since these may differ.

Some algorithms may have an adequate amount of parallelism for CPUs, but lack the massive amount of parallelism required by GPUs. As an example of this, consider Algorithm 8.1. On a standard multi-core CPU, this algorithm could be parallelised in its current form by sharing the iterations of the primary loop between the CPU cores. The same approach would not be suitable for implementation on a GPU, owing to the insufficient number of iterations in the primary loop to satisfy the data parallel requirements of a GPU. However, it is sometimes possible to redesign or transform algorithms in a way that dramatically increases inherent parallelism. Algorithm 8.2 shows how this could be applied to the example problem. Instead of having sufficient inherent parallelism to keep only 32 processors occupied, as is the case in the first algorithm, the transformed algorithm now has enough parallelism for n processors. The inherent parallelism is now directly linked to problem size. This transformation does mean that the initialisation and configuration work is duplicated. However, given the massive processing power of GPUs, there is little need to prevent work duplication in GPU programs if it avoids unfavourable data transfers or code layouts. In some cases, this duplication can be avoided by using shared memory. Low inherent parallelism therefore means that effort must be directed towards re-evaluating the algorithm to ascertain whether it can be restructured or redesigned to increase the inherent parallelism.

Algorithm 8.1 An algorithm with low inherent parallelism.

```
for i ∈ 0..31 do
    ▷ Initialisation and configuration based on i
    for j ∈ 0..n do ▷ Where n is a large number
        ▷ Independent Core Code
```

Algorithm 8.2 Transformed version of Algorithm 8.1 with high inherent parallelism.

```
for i ∈ 0..n do ▷ Where n is a large number
    for j ∈ 0..31 do
        ▷ Initialisation and configuration based on j
        ▷ Independent Core Code
```

The solutions to the hydrological uncertainty ensemble model in Chapter 5 and the large-scale k-difference matching in Chapter 6 are *embarrassingly parallel*, since the inherent parallelism is directly related to the problem size. However, other solutions that were considered would have resulted in a considerably lower amount of inherent parallelism[2].

[2]For example, parallelising the k-difference comparison itself for the problem in Chapter 6.

Unlike the first two problems, the radix sort in Chapter 7 contains sections of code in which only a limited number of threads can participate, and hence, this problem has less inherent parallelism.

An evaluation of inherent parallelism should be possible with a performance projection tool such as GROPHECY, which was designed to statically evaluate skeleton CPU code for GPU acceleration [48].

8.2.2 Branch Divergence

Within compute units, wavefronts are executed by SIMD stream processors. Consequently, *branch divergence* (or control flow divergence) within a wavefront results in lower GPU utilisation, as stream processors that do not follow a branch are forced to idle [18]. This is clearly illustrated in Figure 3.3.

The k-difference string matching problem was parallelised by assigning each work-item different k-difference problems to solve, thus making the problem embarrassingly parallel. This solution also allowed for a significant amount of branch divergence, as each work-item processes strings with potentially different lengths to its neighbours. Ukkonen's cut-off heuristic further increased branch divergence by adding variation based on the input data. This resulted in an average active number of work-items in a wavefront of 55% or lower, which is a significant performance bottleneck. Since branch divergence can be difficult to avoid without significant code refactoring, it was not one of the problems we attempted to address in our GPU solutions. Nevertheless, it is an important difficulty indicator, as it can have a significant impact on GPU performance, and it can be avoided or reduced through code refactoring and methods such as branch fusion [18].

The amount of branch divergence within a program typically changes depending on the input data. Typical divergence statistics for a program would therefore have to be determined through dynamic analysis using typical input data. It may be possible to do this by using an automatic conversion tool to obtain a basic functioning GPU program (e.g. OpenMP to CUDA [42]), which can then be used by a GPU performance analysis tool (e.g. GPUPerf [73]) to extract this information. Familiarity with a program may also enable one to make a reasonable estimate of the severity of branch divergence.

8.2.3 Problem Size

Problem size is the amount of work that can be parallelised. If managed correctly, an abundance of work enables the GPU to hide much of the memory access latency (given the resource constraints) [3]. Conversely, a low amount of parallel work increases the likelihood of compute units stalling on pending memory operations, or idling without work.

An abundance of parallel work is typically used to improve TLP, ILP, or both. Improving TLP involves scheduling additional work-groups to compute units. This gives the SIMD units within the compute units the opportunity to process alternative work-groups when the resident work-group has stalled, thereby improving stream processor utilisation and GPU performance. However, TLP cannot be improved indefinitely. Compute units are able to schedule a limited number of work-groups and wavefronts. On AMD GPUs, the number of work-items (threads) needed to fully occupy the GPU can be calculated by Eq. (8.1):

$$Work\text{-}items = \frac{max\_wavefronts\_per\_CU \times number\_of\_CU}{\left\lceil \frac{work\_group\_size}{64} \right\rceil} \times work\_group\_size \tag{8.1}$$

The AMD HD7970 has 32 compute units, each of which can schedule 40 wavefronts [3]. A typical work-group size of 64 would therefore require 81,920 work-items to fully occupy the GPU. However, because compute unit resources are shared between the work-groups active on that compute unit [3], the number of active work-groups each compute unit can manage may be significantly lower than the maximum. For example, the GPU HBP implementation of k-difference string matching in Chapter 6 has a maximum occupancy of 40% for short strings, owing to register usage restricting the number of work-groups that can be scheduled on a compute unit to 16. This means 32,768 work-items would be sufficient to reach the restricted maximum occupancy. Since GPU resource usage is not known prior to implementation, estimations would have to be used based on GPU performance projection tools such as GROPHECY [48].

Increasing ILP and using additional registers is another approach to improving GPU utilisation [80]. Instead of using many additional wavefronts to hide memory access latency, ILP hides memory access latency by enabling stream processors to execute alternative independent instructions in the same work-item. This approach has been shown to provide high GPU utilisation even with very low GPU occupancy [80].

While ILP and TLP differ in the way in which they utilise work-items to improve utilisation, they can both be improved by increasing the number of independent work problems processed by each stream processor. Thus, to maximise GPU utilisation by leveraging either ILP, TLP, or optimally a combination of the two, the number of work problems to be processed should be multiple times larger than the number of stream processors in the target GPU. Estimation of the number of work problems required to cover memory access latency is discussed in the next section. In the absence of an abundance of work problems, ILP can also be improved by ensuring that independent instructions follow each other as often as possible.

The impact of insufficient problem size can be seen in case studies 1 and 3. In Section 5.3.3, the speedup of the GPU implementation of the hydrological model steadily declined with problem sizes lower than 15,000 ensembles. The performance of the GPU radix sort in Section 7.3 steadily increased with larger problem sizes. It is therefore clear that having a sufficiently large problem size is important when accelerating a program using GPUs.

8.2.4 Required Computational Parallelism

We define *required computational parallelism (RCP)* as the number of wavefronts or warps required to cover memory access latency. This is calculated as $RCP = (a+m)\ /\ a$, where a is the total arithmetic latency and m is the total global memory latency for a single work-item. This is identical to *computation warp parallelism* [29]; we have used a different name to avoid association with a particular brand of GPUs (*warp* is a CUDA term). RCP is similar to arithmetic intensity, except that it relates to time rather than quantity.

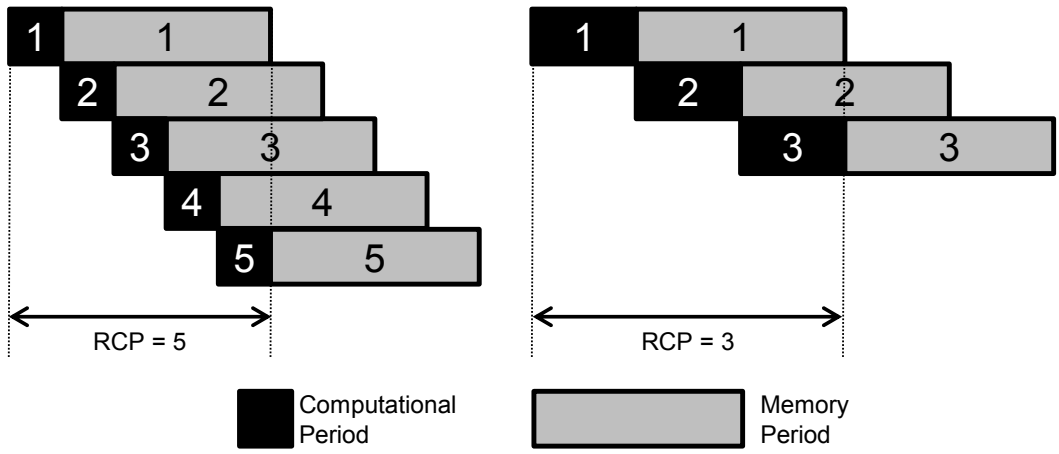


Figure 8.1: An illustration of how the length of computational periods affects RCP. The numbers within the blocks represent different wavefronts or warps.

The latencies of global memory reads and writes are orders of magnitude higher than those of computational instructions [3], which means it can be quite difficult to hide memory latency with computation. RCP is essentially a measure of how many alternative wavefronts are required to cover memory latency. This is clearly illustrated in Figure 8.1. Programs with a low RCP are preferable as this simplifies the hiding of memory latency. High RCP values typically require more effort to be directed towards adding TLP and ILP to hide memory latency, and ensuring memory requests are as efficient as possible. The acceleration of the standard algorithm compared to the HBP algorithm in Chapter 6 is a good example of this. The standard algorithm required a much larger number of global memory requests than the HBP algorithm, and as a result, more effort was spent on reducing the number of memory requests and improving the efficiency of these requests. The radix sort also has a high RCP, and many of the radix sort optimisations are memory related.

```
1  void reduce(float *x, float *output, float z, int dataitems)
2  {
3    for (int i = 0; i < dataitems; i ++)
4    {
5        float result = 0;
6        float input = x[i];
7
8        for (int j = 0; j < 4; j++) {
9            result += z * i;
10       }
11
12       result *= input;
13       output[i] = result;
14   }
15 }
```

Listing 8.1: Simple C++ function illustrating the calculation of RCP.

Listing 8.1 is used to demonstrate how RCP is calculated. In a multithreaded CPU application, the `reduce` function can be called by multiple threads with a subset of the total number of data items to achieve parallelism. A GPU solution, on the other hand, may remove the primary loop and assign a GPU thread to each work item to be processed. Doing so would result in each thread processing roughly 13 arithmetic operations[3] and one memory read instruction. The number of clock cycles these instructions require varies depending on the target GPU. On the HD7970, most arithmetic instructions have a latency of four cycles, while global memory fetch instructions have a latency of 400 to 600 cycles [3]. The aggregate latency of the compute instructions is therefore ~52 cycles

[3]Lines 8 to 10 contain 12 arithmetic operations (three per iteration, one for the loop counter and two for line 9), and line 12 adds one more.

while the aggregate latency of the memory instructions is ~500 cycles. This results in an RCP of 10.6.

A limitation of calculating RCP using the above approach is that it assumes the algorithm has perfect ILP and contains no global read-after-write dependencies. It is also assumed that the program does not spill registers into global memory; if this occurs, variables would have to be manually moved into global data structures to ensure global memory access patterns are predictable. Thus, accurately calculating this value by hand may be difficult, especially with the size of some programs and the limited information available on the latency of GPU operations. However, tools such as GROPHECY can provide estimations of the number of GPU memory load and computational instructions given annotated input from the user [48]. If a base GPU program already exists, kernel analysis tools are capable of providing estimates as well. For example, AMD's kernel analyser estimates the number of stream processor instructions and fetch instructions for OpenCL kernels from static analysis[4]. There are also performance analysis tools that can do this for CUDA programs [29, 73].

RCP can be used to estimate the problem size needed to hide the latency of global memory requests. However, an accurate interpretation of RCP can only be done with extra problem specific information, such as the maximum number of wavefronts or warps that can be scheduled by the hardware, the number of wavefronts in a compute unit that can simultaneously access global memory during the time it takes for a memory request to complete, problem size, and maximum memory bandwidth [29, 73]. With further work, it should be possible to adapt existing GPU program analytical models for this purpose.

8.2.5 Memory Access Pattern Regularity

GPUs are able to provide the highest memory bandwidth when memory access patterns are regular and have high spatial locality. Irregular access patterns or patterns with low spatial locality prevent memory requests from being coalesced into fewer memory transactions, or result in unbalanced utilisation of the memory controllers [3, 48]. This can sometimes be addressed by preprocessing the input data to group similar data inputs, or rearrangement of the data items within a work-group.

The original solutions to the first two case studies included memory access patterns with low spatial locality, which significantly reduced performance and were addressed through

[4] `http://developer.amd.com/tools-and-sdks/heterogeneous-computing/codexl/`

data-layout optimisations. Branch divergence in the solution for the second case study can also result in irregular memory access patterns. For the most part, the memory access patterns in the radix sort are regular and have high spatial locality; however, the writing of the output data in the key scattering kernel is inherently irregular. This was partially addressed in the MG radix sort solution through inter-group data swapping. This problem is more challenging to address in algorithms that have inherently irregular data structures, such as those that involve tree or graph traversal [82].

The overall rating of this difficulty indicator is a combination of the ratings for memory access regularity and memory access locality. The measurement of these prior to implementation is difficult, since it typically involves dynamic variables. However, the GROPHECY tool is able to measure the prevalence of memory access irregularity to some extent through static analysis and deductions [48].

8.2.6 Data Transfer Overhead

Most GPU programs require data to be transferred to and from the GPU. The overhead of these transfers and the context in which the GPU program is run are important considerations that can affect problem difficulty. In the context of running a standalone GPU program, the time taken to transfer data to and from the GPU is pure overhead. Thus, GPU programs with large data requirements will be harder to accelerate than those with smaller data requirements.

The importance of reviewing kernel data requirements was recognised by Gregg and Hazelwood [27], prompting the creation of a taxonomy to describe the data transfer requirements of GPU kernels. This taxonomy is outlined in Section 3.4, where the categories are listed in order of preference from a performance perspective. Programs that fall into the single-dependent-host-to-device, single-dependent-device-to-host, or dual-dependent categories require unhidden data transfers between the host and the GPU. This data transfer overhead can greatly reduce the performance benefit of using GPUs, and can even result in a performance slowdown [15]. Data transfer overhead had a crippling impact on the performance of the GPU radix sort (Section 7.3.1). A measure of the data transfer overhead can therefore be useful in estimating difficulty, since high values may require data restructuring to ensure that the smallest amount of data is transferred between the GPU and host, use of pinned memory, or implementation of optimisations such as overlapped data transfer with kernel execution.

An estimation of data transfer overhead (in percent) can be obtained using a tool such as GROPHECY++ [15]. By building on the analysis and CPU code transformations performed by the original GROPHECY tool, GROPHECY++ can estimate data transfer time by measuring the data transfer rate and identifying data that must be transferred between the host and GPU [15]. The data transfer rate will vary from host to host owing to differences in the specification of the host and device hardware. In the absence of the relevant hardware to measure the transfer rate, default hardware specific values can be used. The tool can also estimate the execution time of a CPU program after GPU acceleration (with some optimisation) [15, 48]. Using the data transfer and program execution time estimates, the estimated data transfer overhead can be calculated. The tool's prediction errors in estimating data transfer and computation times have been measured at an average of 8% and 9%, respectively [15].

The potential severity of data transfer overhead can also be determined manually by calculating the quantity of data that would need to be transferred between the host and the GPU for a desired problem size and then estimating the GPU speedup. Once the quantity of data to be transferred has been calculated, it can be converted into a time estimate using the average data transfer rate between the host and the GPU[5]. The compute time can be estimated by reducing the execution time of the CPU version by estimated GPU speedup factors. The data transfer overhead can then be estimated by calculating the data transfer time as a percentage of the combined compute and data transfer time.

8.2.7 Thread Cooperation

Cooperation between threads can occur within the same work-group (intra-group) or between different work-groups (inter-group). Intra-group thread cooperation is commonly used in conjunction with local memory, while inter-group cooperation takes place over multiple kernel executions through global memory for reasons of data consistency.

The difficulty of thread cooperation lies in its use of local and global memory, and synchronisation primitives such as barriers and atomic operations. Local and global memories are orders of magnitude slower than private memory, and atomic operations serialise concurrent memory requests. This necessitates giving careful thought to the granularity and structure of thread cooperation to ensure that performance penalties are kept to a minimum. This includes understanding how best to arrange data in shared memory regions to

[5]The data transfer rate of similar systems could be used if the target system is unavailable.

avoid memory channel conflicts that serialise the memory requests [3]. The more prevalent thread cooperation there is in a program, the more important it is to ensure that the cooperation is designed as efficiently as possible to prevent unnecessary performance loss.

The final solution to the large-scale k-difference problem discussed in Chapter 6 used thread cooperation to reduce duplication of work and memory reads. Although this was a relatively simple use of thread cooperation, the implementation thereof provided some difficulty relating to the use of local versus global memory. Conversely, the MG radix sort in Chapter 7 made extensive use of thread cooperation for several different purposes. This necessitated the use of optimisations and techniques to ensure that this cooperation was as fast as possible, which greatly increased the complexity of the GPU solution.

8.3 Classification Framework

It would be beneficial to aggregate the analysis of the different problem characteristics discussed in the previous section into a simple classification framework that could be used to describe overall problem difficulty. This would require quantifying the analysis of each difficulty indicator using a rating that appropriately describes its relevance to problem difficulty; an attempt at doing this is given below. This is followed by an initial framework design that incorporates these ratings to describe overall problem difficulty, application of the classification framework to the accelerated problems in Chapters 5 to 7, and limitations of the proposed design.

8.3.1 Difficulty Categories

For the purposes of the initial classification framework, ordinal ratings are used for all the difficulty indicators, namely, 'Negligible', 'Low', 'Moderate', or 'High', as quantitative measurement of the difficulty indicators and determination of applicable thresholds are beyond the scope of this research. These ratings have different meanings for the different difficulty indicators, as discussed below.

Inherent Parallelism: 'Negligible' means there is no inherent parallelism. A 'Low' rating means that inherent parallelism is not linked to problem size, and there is an insufficient number of parallel tasks for the number of processors on the GPU. A 'High' rating means there is an abundance of parallel tasks, and the inherent parallelism is typically linked to problem size.

Branch Divergence: 'Negligible' means there is virtually no branch divergence, while 'High' denotes an abundance of branch divergence.

Problem Size: 'Negligible' means there are virtually no work tasks. A 'Low' rating means the envisaged problem size is not sufficiently large to provide work for all the stream processors in the GPU, or not large enough to take advantage of TLP or ILP. A 'High' rating means the problem size allows for an abundance of TLP and ILP.

Required Computational Parallelism: A 'Negligible' RCP means no TLP is needed, while a 'High' RCP means it is difficult or impossible to hide memory access latency.

Memory Access Regularity: 'Negligible' means memory transactions are completely irregular and have no spatial locality, whereas 'High' means memory accesses are predominantly or always regular and have high spatial locality.

Data Transfer Overhead: 'Negligible' means the data transfer overhead is not a consideration, whereas 'High' means the data transfer overhead contributes significantly to overall program execution time.

Thread Cooperation: 'Negligible' means there is virtually no thread cooperation, while 'High' means thread cooperation is prevalent throughout the solution.

8.3.2 Framework Design

Based on the difficulty indicators described in the previous section, a difficulty classification framework was constructed, as illustrated in Table 8.1. Despite its simplicity and use of ordinal ratings, this framework can provide the user with an idea of overall problem difficulty, particularly for extreme cases where problem acceleration is either very simple or very difficult, and serves as a starting point for future work on GPU problem difficulty classification.

To enable computation of an overall difficulty value, values between zero and three are associated with each of the ordinal ratings. Where higher evaluations result in increased difficulty, the 'Negligible' rating is assigned zero and the 'High' category three. The reverse is true for indicators where higher evaluations result in decreased difficulty. Since all indicators are weighted equally in this framework, the overall difficulty value can be obtained by simple addition of the values assigned to the indicators.

Table 8.1: The problem difficulty classification framework.

Difficulty Indicator	**Rating**			
	Negligible	Low	Moderate	High
Inherent Parallelism				
Branch Divergence				
Problem Size				
Required Computational Parallelism				
Memory Access Pattern Regularity				
Data Transfer Overhead				
Thread Cooperation				

8.3.3 Classification of Accelerated Problems

Given that the classification framework was modelled on what was learned from accelerating the three case studies, evaluation of these problems using the framework should provide a difficulty estimation similar to what was actually experienced. Owing to the unavailability of the GROPHECY or GROPHECY++ tool for measurement of the applicable indicators, performance metrics estimated by the CodeXL kernel profiling tool are used instead. The reader is referred to Appendix A for an explanation of how RCP and data transfer overhead figures were calculated. We have attempted to classify the problems without using knowledge gained during the actual problem acceleration.

Case Study 1: Hydrological Uncertainty Model

Inherent Parallelism: Since the model uses uncertainty analysis, many independent instances of the model are run with different input data. Each of these model instances can be regarded as a parallel task to be run as a separate work-item, which links the parallelism with problem size. This makes the problem embarrassingly parallel.

Branch Divergence: There is unlikely to be much branch divergence between ensembles since the primary loop iteration counts are identical, and the data-dependent

branches typically have a low depth. Kernel profiling confirmed this by indicating an average branch divergence of under 3%.

Problem Size: Since the purpose of this program is uncertainty analysis, the parameters for each model run are generated randomly (within certain bounds). The problem size is therefore only restricted by the number of possible parameter configurations; this number is large enough not to be of concern. Before acceleration, the model was typically run with between 5,000 and 20,000 ensembles. The limit on the number of ensembles run was solely due to the high execution time of the sequential solution.

RCP: This ratio was evaluated to be 4.7. We consider this to be 'Moderate', as it would require TLP amounting to just under half of the maximum number of wavefronts that can be scheduled on a SIMD unit.

Memory Access Pattern Regularity: Other than a small amount of branch divergence, the memory access pattern is regular. However, the memory access locality is low because of the large data structures that store the model data. Thus, we classify this as 'Moderate'.

Data Transfer Overhead: The data transfer to the GPU consists of the core model data and the parameter sets for each ensemble. Since the parameter sets are small, the quantity of input data transferred between the host and the GPU does not scale significantly with problem size. With model runs being computationally intensive, the data transfer overhead should only constitute a small portion of the overall

Table 8.2: Classification of the hydrological uncertainty model.

Difficulty Indicator	**Rating**			
	Negligible	Low	Moderate	High
Inherent Parallelism				✔ 0
Branch Divergence	✔ 0			
Problem Size				✔ 0
Required Computational Parallelism			✔ 2	
Memory Access Pattern Regularity			✔ 1	
Data Transfer Overhead	✔ 0			
Thread Cooperation	✔ 0			

GPU execution time. If the GPU speedup factor is estimated at 10x or 100x, the data transfer overhead is estimated at less than 0.1% for a problem size of 50,000 ensembles. We therefore classify this as 'Negligible'.

Thread Cooperation: Each work-item calculates its own model ensemble, and thus no thread cooperation is required.

Table 8.2 shows the difficulty classification of the hydrological uncertainty model. The only non-zero difficulty indicators are RCP and memory access pattern regularity. The overall difficulty according to this evaluation is 3, which suggests a relatively low acceleration difficulty.

Case Study 2: *K*-Difference String Matching

Inherent Parallelism: Each k-difference comparison can be run independently of all others, and the number of comparisons is linked to problem size. This gives the problem 'High' inherent parallelism.

Branch Divergence: Comparisons between strings of different lengths within a wavefront will result in some work-items finishing before others. Coupled with Ukkonen's cut-off, this results in high branch divergence. According to kernel profiling, the average amount of branch divergence is over 55%.

Problem Size: The number of strings compared is expected to be in the millions for practical applications of this program. There is thus an abundance of parallel work available.

RCP: The RCP for comparing short and long strings was calculated as 3.4 and 2.7 for the HBP algorithm, and 27.1 and 28.6 for the standard algorithm, which may be considered as 'Low' and 'High', respectively.

Memory Access Pattern Regularity: The original solution used a linear layout for storing the string data and partial results in memory. The length of the strings results in low memory access locality, as each work-item accesses different strings. The access pattern regularity would also be low since the strings do not have a set length. Branch divergence can also cause irregularity in the memory access pattern as some work-items request more data than others. Therefore, we classify this as 'Low'.

Data Transfer Overhead: The input consists of the strings to be compared, and the output for each comparison is an integer corresponding to a positive or negative match. If a 10x speedup is assumed for the HBP algorithm with a problem size of 2,000 short test patterns and 8,192 short input strings, the transfer overhead is estimated at 2.22%. We classify this as 'Low'. The overhead decreases to under a percent when applying this estimation to long strings and the standard algorithm, which we classify as 'Negligible'.

Thread Cooperation: Inter-group cooperation between work-items is not needed in this algorithm, but can be recognised as greatly beneficial for data sharing purposes. Since this might not be known prior to implementation, this has been classified as 'Negligible'.

Table 8.3: Classification of the k-difference string matching problem. Ratings that are specific to an algorithm are annotated; the standard algorithm has been abbreviated to STD.

Difficulty Indicator	**Rating**			
	Negligible	Low	Moderate	High
Inherent Parallelism				✔ 0
Branch Divergence				✔ 3
Problem Size				✔ 0
Required Computational Parallelism		✔ 1 (HBP)		✔ 3 (STD)
Memory Access Pattern Regularity		✔ 2		
Data Transfer Overhead	✔ 0	✔ 1 (HBP)		
Thread Cooperation	✔ 0			

Table 8.3 shows the difficulty classification of this problem. Unlike the hydrological uncertainty ensemble model, there are many non-zero difficulty indicators in this classification. If these are summed, the overall difficulties of the standard and HBP algorithms are 8 and 7, respectively.

Case Study 3: Radix Sort

Inherent Parallelism: Parallelism is obtained by partitioning the key space between a number of work-items and sharing information at key points in the algorithm. There are many sections of code in which thread cooperation occurs involving a limited number of work-items, which reduces the inherent parallelism to 'Moderate'.

Branch Divergence: As mentioned for inherent parallelism, there are many sections of code in which a limited number of threads participate, which implies a moderate amount of branch divergence. This is supported by kernel profiling, which indicates divergence to be as high as 43.6% for the first kernel, but less than 13% for the second and third kernels.

Problem Size: GPU radix sorting is typically only required for very large numbers of keys, which means problem size is 'High'.

RCP: The average RCP for the primary kernels[6] was calculated as 11. This is 'High', since it exceeds the maximum number of wavefronts that can be scheduled on a SIMD unit.

Memory Access Pattern Regularity: Irregular access patterns are found in the final kernel when writing the results of a sorting pass to global memory. Given that this code is visited a number of times from multiple sorting passes, this algorithm has been classified as having 'Moderate' memory access pattern regularity.

Data Transfer Overhead: Other than several non-vector input variables, the input and output data sizes are identical. For a problem size of 2^{26} 32-bit integers, the data transfer overhead is estimated at 59%, which is easily classified as 'High'.

Thread Cooperation: A significant amount of both intra- and inter-group thread cooperation is used in this algorithm to enable parallel execution.

Table 8.4 shows the difficulty classification of the radix sort. It is clear from this classification that the radix sort is a much harder problem to accelerate than the first two problems. It has a lower amount of inherent parallelism, high RCP, high data transfer overhead, and a much higher use of thread cooperation. The overall difficulty according to this classification is 13.

[6]The second kernel was omitted from the calculation because it is designed to only use a single work-group.

Table 8.4: Classification of the MG radix sort.

Difficulty Indicator	Rating			
	Negligible	Low	Moderate	High
Inherent Parallelism			✔ 1	
Branch Divergence			✔ 2	
Problem Size				✔ 0
Required Computational Parallelism				✔ 3
Memory Access Pattern Regularity			✔ 1	
Data Transfer Overhead				✔ 3
Thread Cooperation				✔ 3

8.3.4 Reflection

Classification of the hydrological uncertainty ensemble model, large-scale k-difference string matching problem, and radix sort using the proposed framework resulted in difficulties of 3, 7 - 8, and 13, respectively. Relative to each other, these ratings are well matched to the actual problem difficulties experienced. Furthermore, the individual difficulty indicator ratings correspond to either the evaluation of the knowledge requirement for replicating our GPU solutions, or identified bottlenecks. This shows that the evaluation of the identified difficulty indicators can give an idea of the difficulty of problem acceleration using GPUs.

8.3.5 Limitations

The present classification system is an initial attempt at estimating the difficulty associated with implementing algorithms on GPU hardware. The relatively high correlation between predicated and actual difficulty of the test cases described shows that this approach has potential. However, a number of limitations exist that need to be addressed in future revisions.

Evaluation methods: One of the most significant limitations is the lack of reliable quantitative evaluation methods for the problem difficulty indicators. Possible methods

have been identified, but these need to be explored in more detail. Once the best methods have been identified, they would need to be built into an evaluation tool to enable quick and easy evaluation of candidate problems for GPU acceleration.

Unclear Classifications: The current classification options are too vague. Other than extreme cases, it is not clear how the indicator evaluations relate to the framework ordinal ratings. Further work is needed to determine appropriate boundaries for each category rating.

Uniform Weighting: In the final difficulty calculation based on a classification, each indicator is weighted equally. In reality, the difficulty impact of some indicators is likely to be more significant than others. This also requires further research.

8.4 Classification-Based Optimisation Guidance

If a GPU problem classification system such as the one described above were standardised, guides could be written to suggest optimisations or strategies to address particular classifications. As shown below, many of the optimisations identified in the case studies could be included as suggestions for certain classifications. The sections in which these optimisations are described are given in brackets.

8.4.1 Extensive Thread Cooperation

Efficient Memory Packing (7.2.3): To reduce the impact of the data transfers necessary for thread cooperation, the data in the shared memory region could be packed as efficiently as possible.

Synchronisation-Free Thread Cooperation (7.2.3): In programs where the work-group size is greater than the wavefront size, synchronisation-free thread cooperation can reduce the performance penalty of barriers.

8.4.2 High Data Transfer Overhead

Batched Processing (7.3.1): The impact of data transfers can be significantly reduced by overlapping data transfer with kernel execution.

Pinned Memory (7.2.3): Pinned memory can improve the data transfer speed between the host and the GPU.

8.4.3 High Required Computational Parallelism

Data Layout (5.3.2, 6.3.1): Memory read bandwidth can be improved by ensuring memory access patterns and data layouts are efficient for the target GPU.

Caching (6.3.1): Storing previously requested data in local or private memory can reduce the number of global memory requests.

Vector Types (6.3.1): The use of vector types can improve the efficiency of memory requests.

Efficient Memory Packing (7.2.3): Efficient memory packing can reduce the number of memory requests by better utilisation of memory space.

Thread Cooperation (6.3.1): In situations where different work-items require the same data from global memory, thread cooperation can be used to read the data once from global memory and share it between threads using faster local memory.

Ordered Writes (7.2.3): Through the use of thread cooperation, global memory writes can be re-ordered to improve memory write bandwidth.

8.5 Summary

The difficulty of GPU acceleration depends on the problem considered for acceleration. For new or novice GPGPU developers, it may not be easy to distinguish problems that would be difficult to accelerate from those that could be accelerated relatively easily. As such, it would be useful to be able to determine the probable difficulty of accelerating a particular problem using a classification framework based on the problem's attributes. To do this, important difficulty indicators for classifying overall problem difficulty were identified by reviewing the problems accelerated in Chapters 5, 6, and 7, and selecting the problem attributes relevant to acceleration difficulty. The identified indicators were inherent parallelism, branch divergence, problem size, required computational parallelism, memory access pattern regularity, data transfer overhead, and thread cooperation. A simple difficulty classification framework was created based on these difficulty indicators

and ordinal rating categories. This framework was applied to the accelerated problems, and the classifications were found to correspond relatively well with the difficulty actually experienced in accelerating these problems. However, three limitations of the framework were identified, namely, the lack of reliable quantitative difficulty indicator evaluation methods, vague classification ratings, and unrealistic uniform indicator weightings. It was suggested that all of these could be addressed with further research. Finally, it was shown that specific problem difficulty indicator classifications can be used to suggest specific optimisations to improve program performance.

Chapter 9

Conclusion and Future Work

GPUs have improved tremendously in their ability to perform general-purpose computation in recent years; their massively parallel and throughput-oriented architecture provides many scientists with an opportunity to significantly improve the performance of their applications using commodity hardware. However, the architecture of GPUs is not well suited to all kinds of problems, and some problems require extensive developer effort and GPU knowledge to achieve a satisfactory speedup. Past research has made progress in projecting the probable GPU performance given a skeleton implementation, but no work has been done on estimating the difficulty of GPU acceleration. Such an estimation, along with a breakdown thereof, would give those interested in GPU acceleration an idea of what to expect in terms of the required development effort and GPGPU knowledge.

We set out to address the lack of a means to formally estimate problem difficulty through the identification of problem attributes that are important in determining problem difficulty; evaluation of the reasons behind the identified attributes' contribution to problem difficulty; and creation of an initial difficulty classification framework. The identification of relevant problem attributes was achieved through the acceleration and review of three problems of increasing difficulty, namely, a hydrological uncertainty ensemble model, a comparison of large numbers of strings using k-difference matching, and a radix sort algorithm. The speedups achieved under ideal circumstances were 10.2x for the hydrological model, 109x (standard algorithm) and 21x (HBP algorithm) for computing large numbers of k-difference comparisons, and 3.7x for the radix sort.

The review of the aforementioned case studies revealed seven problem attributes to be important factors in GPU acceleration difficulty. These are inherent parallelism, branch

divergence, problem size, required computational parallelism, memory access pattern regularity, data transfer overhead, and thread cooperation. Their contribution to overall problem difficulty was explored by evaluating the reason for their impact on GPU performance and the work required to address unfavourable evaluations.

The identification of appropriate problem difficulty indicators enabled the creation of an initial problem difficulty classification framework. Since quantitative methods for evaluating the difficulty indicators were not available, the framework was constructed based on ordinal rating categories: 'Negligible', 'Low', 'Moderate', and 'High'. Classifications are thus acknowledged as being subjective, since it may not be clear where the boundary lies between adjacent ratings. Overall problem difficulty is determined by simple addition of the numeric representations of the difficulty ratings, which are weighted equally. Application of the classification framework to the three accelerated problems produced difficulty classifications of 3, 7 - 8, and 13, respectively, which are relatively accurate evaluations of the difficulty actually experienced in accelerating the problems.

Despite a lack of quantifiable difficulty indicators, a difficulty classification framework has been created that clearly differentiates problems at the extremes of the difficulty spectrum. Furthermore, application of the framework to the accelerated problems resulted in difficulty indicator ratings for each case study that matched either our actual experience in problem acceleration, or performance bottlenecks identified in our solutions. With the adaptation of our framework for quantitative measurements, we firmly believe that a more accurate difficulty classification is possible across the full range of difficulties.

The work presented in this thesis represents a preliminary exploration into problem difficulty classification. During the research, a number of opportunities were identified that can be addressed in future work. Firstly, a more comprehensive study on the attributes of problem solutions that increase acceleration difficulty could be done to ensure that all important difficulty indicators are present in the classification framework. Following this, each of the attributes needs to be studied in more detail to determine appropriate quantitative measurement methods. Given the effort that may be required to manually assess each of the problem difficulty indicators, an existing performance analysis tool could be extended, or a new tool created, to automate this assessment. With the availability of quantitative measurement methods for the difficulty indicators, the classification options may need to be revised to better reflect the range of possible values, and appropriate boundaries for each category rating would need to be determined. It would also be beneficial to investigate the relative importance of the difficulty indicators to assign appropriate weightings to them in the classification framework. Finally, extensive user

testing could be carried out to determine the accuracy of difficulty estimations given by the framework.

References

[1] ACM. The ACM Computing Classification System [1998 Version]. 1998. Online: http://www.acm.org/about/class/1998 [Accessed 10/12/13].

[2] Advanced Micro Devices. AMD Graphics Core Next (GCN) Architecture. 2012. Online: http://www.amd.com/us/Documents/GCN_Architecture_whitepaper.pdf [Accessed 31/07/13].

[3] Advanced Micro Devices. AMD Accelerated Parallel Processing OpenCL Programming Guide. August 2013. Online: http://developer.amd.com/wordpress/media/2013/08/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf [Accessed 01/08/13].

[4] Advanced Micro Devices. CodeXL. 2013. Online: http://developer.amd.com/tools-and-sdks/heterogeneous-computing/codexl/ [Accessed 19/07/2013].

[5] Advanced Micro Devices. CodeXL User Guide. November 2013. Online: http://developer.amd.com/wordpress/media/2013/11/CodeXLHelp.chm [Accessed 01/08/13].

[6] Aleen, F., and Mahalingam, K. Improving Bayesian Spam Filters Using String Edit Distance Algorithm. In *International Conference on Internet Computing* (2008), CSREA Press, 121–125.

[7] Anderson, M., Catanzaro, B., Chong, J., Gonina, E., Keutzer, K., Lai, C., Murphy, M., Sheffield, D., Su, B., and Sundaram, N. Considerations when evaluating microprocessor platforms. In *Proceedings of the 3rd USENIX conference on Hot topic in parallelism* (Berkeley, CA, USA, 2011), HotPar'11, USENIX Association, 1.

[8] Andrews, G. R., and Schneider, F. B. Concepts and Notations for Concurrent Programming. *ACM Comput. Surv. 15*, 1 (Mar. 1983), 3–43.

[9] Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., and Yelick, K. A view of the parallel computing landscape. *Commun. ACM 52*, 10 (Oct. 2009), 56–67.

[10] Baghsorkhi, S. S., Delahaye, M., Patel, S. J., Gropp, W. D., and Hwu, W. W. An Adaptive Performance Modeling Tool for GPU Architectures. *SIGPLAN Not. 45*, 5 (Jan. 2010), 105–114.

[11] Barry, W. amd Allen, M. *Parallel Programming: Techniques And Applications Using Networked Workstations And Parallel Computers, 2/E.* Pearson Education, Upper Saddle River, NJ, 2006.

[12] Basu, D. Parallel Radix Sort on the GPU using C++ AMP. 2013. Online: `http://www.codeproject.com/Articles/543451/Parallel-Radix-Sort-on-the-GPU-using-Cplusplus-AMP` [Accessed 09/02/13].

[13] Beven, K. J. A manifesto for the equifinality thesis. *Journal of Hydrology 320*, 1–2 (2006), 18–36.

[14] Bland, A. S., Wells, J., Messer, O. E., Hernandez, O., and Rogers, J. Titan: Early experience with the Cray XK6 at Oak Ridge National Laboratory. In *Cray User Group 2012* (2012).

[15] Boyer, M., Meng, J., and Kumaran, K. Improving GPU Performance Prediction with Data Transfer Modeling. In *IEEE 27th International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)* (2013), 1097–1106.

[16] Chang, W., and Lampe, J. Theoretical and empirical comparisons of approximate string matching algorithms. In *Combinatorial Pattern Matching* (1992), Springer, 175–184.

[17] Che, S., Li, J., Sheaffer, J., Skadron, K., and Lach, J. Accelerating compute-intensive applications with GPUs and FPGAs. In *Symposium on Application Specific Processors (SASP)* (2008), IEEE, 101–107.

[18] Coutinho, B., Sampaio, D., Pereira, F. M. Q., and Meira Jr., W. Divergence Analysis and Optimizations. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques* (Washington, DC, USA, 2011), PACT '11, IEEE Computer Society, 320–329.

[19] Culler, D. E., Singh, J. P., and Gupta, A. *Parallel computer architecture: a hardware/software approach*. Gulf Professional Publishing, 1999.

[20] Dongarra, J. Visit to the National University for Defense Technology Changsha, China. Tech. rep., University of Tennessee, 2013.

[21] El-Rewini, H., and Abd-El-Barr, M. *Advanced computer architecture and parallel processing*, vol. 42. John Wiley & Sons, Inc., 2005.

[22] Enslow, Jr., P. Multiprocessor Organization - a Survey. *ACM Comput. Surv. 9*, 1 (March 1977), 103–129.

[23] Fang, Q., and Boas, D. A. Monte Carlo simulation of photon migration in 3D turbid media accelerated by graphics processing units. *Opt. Express 17*, 22 (Oct. 2009), 20178–20190.

[24] Flynn, M. *Computer Architecture: Pipelined and Parallel Processor Design*. Computer Science Series. Jones and Bartlett, 1995.

[25] Flynn, M. J. Some computer organizations and their effectiveness. *IEEE Trans. Comput. 21*, 9 (Sept. 1972), 948–960.

[26] Gaster, B. *Heterogeneous Computing with OpenCL*. Morgan Kaufmann, 2012.

[27] Gregg, C., and Hazelwood, K. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software* (2011), ISPASS '11, IEEE Computer Society, 134–144.

[28] Hennessy, J. L., and Patterson, D. A. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[29] Hong, S., and Kim, H. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. *SIGARCH Comput. Archit. News 37*, 3 (June 2009), 152–163.

[30] Hughes, D. A. Incorporating groundwater recharge and discharge functions into an existing monthly rainfall–runoff model. *Hydrological Sciences Journal 49*, 2 (2004), 297–311.

[31] Hughes, D. A., Andersson, L., Wilk, J., and Savenije, H. H. Regional calibration of the Pitman model for the Okavango River. *Journal of Hydrology 331*, 1–2 (2006), 30–42.

[32] Hughes, D. A., and Forsyth, D. A. A generic database and spatial interface for the application of hydrological and water resource models. *Computers & Geosciences 32*, 9 (2006), 1389–1402.

[33] Hughes, D. A., Kapangaziwiri, E., and Sawunyama, T. Hydrological model uncertainty assessment in Southern Africa. *Journal of Hydrology 387*, 3–4 (2010), 221–232.

[34] Hyyrö, H. Explaining and Extending the Bit-parallel Approximate String Matching Algorithm of Myers. Tech. rep., Dept. of Computer and Information Sciences, University of Tampere, 2001.

[35] Hyyrö, H. A bit-vector algorithm for computing Levenshtein and Damerau edit distances. *Nordic J. of Computing 10*, 1 (Mar. 2003), 29–39.

[36] Jaques, M., Ross, C., and Strickland, P. Exploiting inherent parallelism in non-linear finite element analysis. *Computers & Structures 58*, 4 (1996), 801–807.

[37] Jotwani, N. D. *Computer System Organization.* Tata McGraw-Hill Education, 2009.

[38] Kapangaziwiri, E., Hughes, D. A., and Wagener, T. Constraining uncertainty in hydrological predictions for ungauged basins in Southern Africa. *Hydrological Sciences Journal 57(5)* (2012), 1000–1019.

[39] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.2, Rev 15.* Khronos Group, 15 November 2011.

[40] Kothapalli, K., Mukherjee, R., Rehman, M., Patidar, S., Narayanan, P. J., and Srinathan, K. A performance prediction model for the CUDA GPGPU platform. In *International Conference on High Performance Computing (HiPC)* (2009), 463–472.

[41] Langner, L. Parallelization of Myers Fast Bit-Vector Algorithm using GPGPU. Master's thesis, Freie Universität Berlin, 2011.

[42] Lee, S., Min, S., and Eigenmann, R. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. *SIGPLAN Notices 44*, 4 (Feb. 2009), 101–110.

[43] Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., and Dubey, P. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *SIGARCH Comput. Archit. News 38*, 3 (June 2010), 451–460.

[44] Leeser, M., Ramachandran, J., Wahl, T., and Yablonski, D. OpenCL Floating Point Software on Heterogeneous Architectures — Portable or Not? In *Workshop on Numerical Software Verification (NSV)* (2012).

[45] Lewis, T. G., and El-Rewini, H. *Introduction to parallel computing*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[46] Lu, P., Oki, H., Frey, C., Chamitoff, G., Chiao, L., Fincke, E., Foale, C., Magnus, S., McArthur, WilliamS., J., Tani, D., Whitson, P., Williams, J., Meyer, W., Sicker, R., Au, B., Christiansen, M., Schofield, A., and Weitz, D. Orders-of-magnitude performance increases in GPU-accelerated correlation of images from the International Space Station. *Journal of Real-Time Image Processing 5*, 3 (2010), 179–193.

[47] Meenderinck, C., and Juurlink, B. (When) Will CMPs Hit the Power Wall? In *Euro-Par 2008 Workshops - Parallel Processing*, vol. 5415 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2009, 184–193.

[48] Meng, J., Morozov, V. A., Kumaran, K., Vishwanath, V., and Uram, T. D. GROPHECY: GPU performance projection from CPU code skeletons. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (2011), ACM, 14.

[49] Merrill, D., and Grimshaw, A. Revisiting Sorting for GPGPU Stream Architectures. Tech. Rep. CS2010-03, University of Virginia, Department of Computer Science, Charlottesville, VA, USA, 2010.

[50] Merrill, D., and Grimshaw, A. High Performance and Scalable Radix Sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters 21*, 2 (2011), 245–272.

[51] Microsoft Corporation. C++ AMP : Language and Programming Model. Tech. rep., Microsoft Corporation, `http://download.microsoft.com/download/4/0/E/40EA02D8-23A7-4BD2-AD3A-0BFFFB640F28/CppAMPLanguageAndProgrammingModel.pdf`, 2012.

[52] Moradkhani, H., and Sorooshian, S. General Review of Rainfall-Runoff Modeling: Model Calibration, Data Assimilation, and Uncertainty Analysis. In *Hydrological Modelling and the Water Cycle*, vol. 63 of *Water Science and Technology Library*. Springer Berlin Heidelberg, 2008, 1–24.

[53] Murthy, G., Ravishankar, M., Baskaran, M., and Sadayappan, P. Optimal loop unrolling for GPGPU programs. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on* (2010), 1–11.

[54] Myers, G. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. ACM 46*, 3 (May 1999), 395–415.

[55] Natoli, V. Top 10 Objections to GPU Computing. June 2011. Online: `http://archive.hpcwire.com/hpcwire/2011-06-09/top_10_objections_to_gpu_computing_reconsidered.html` [Accessed 15/11/13].

[56] Navarro, G. A guided tour to approximate string matching. *ACM Comput. Surv. 33*, 1 (Mar. 2001), 31–88.

[57] Netronome. Netronome NFP-6xxx. August 2013. Online: `http://www.netronome.com/files/file/Product%20Briefs/Netronome%20NFP-6xxx%20Product%20Brief%208-13.pdf` [Accessed 19/11/13].

[58] Nickolls, J., and Dally, W. The GPU computing era. *Micro, IEEE 30*, 2 (2010), 56–69.

[59] NVIDIA Corporation. *CUDA C Programming Guide*, 4.2 ed. NVIDIA Corporation, April 2012.

[60] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. Tech. rep., NVIDIA Corporation, `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`, 2012.

[61] Oak Ridge Leadership Computing Facility. ORNL Debuts Titan Supercomputer. Tech. rep., Oak Ridge Leadership Computing Facility, 2012.

[62] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. GPU Computing. *Proceedings of the IEEE 96*, 5 (May 2008), 879–899.

[63] Pitman, W. V. *A mathematical model for generating monthly river flows from meteorological data in South Africa.* Hydrological Research Unit, University of the Witwatersrand, Johannesburg, South Africa, 1973.

[64] Rosenberg, O. OpenCL Overview. November 2011. Online: `http://www.khronos.org/assets/uploads/developers/library/overview/opencl-overview.pdf` [Accessed 01/05/12].

[65] Ross, P. Why CPU frequency stalled. *Spectrum, IEEE 45*, 4 (2008), 72–72.

[66] Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and Hwu, W. W. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming* (New York, NY, USA, 2008), PPoPP '08, ACM, 73–82.

[67] Ryoo, S., Rodrigues, C. I., Stone, S. S., Baghsorkhi, S. S., Ueng, S., Stratton, J. A., and Hwu, W. W. Program optimization space pruning for a multithreaded GPU. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization* (2008), CGO '08, ACM, 195–204.

[68] Saltelli, A., Ratto, M., Andres, T., Campolongo, F., Cariboni, J., Gatelli, D., Saisana, M., and Tarantola, S. *Global Sensitivity Analysis. The Primer.* John Wiley & Sons, 2008.

[69] Sanders, J., and Kandrot, E. *CUDA by example: an introduction to general-purpose GPU programming.* Addison-Wesley Professional, 2010.

[70] Satish, N., Harris, M., and Garland, M. Designing efficient sorting algorithms for manycore GPUs. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing* (Washington, DC, USA, 2009), IPDPS '09, IEEE Computer Society, 1–10.

[71] Satish, N., Kim, C., Chhugani, J., Nguyen, A. D., Lee, V. W., Kim, D., and Dubey, P. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (2010), SIGMOD '10, 351–362.

[72] Scarpino, M. A Gentle Introduction to OpenCL. August 2011. Online: `http://www.drdobbs.com/parallel/a-gentle-introduction-to-opencl/231002854` [Accessed 29/11/13].

[73] Sim, J., Dasgupta, A., Kim, H., and Vuduc, R. A performance analysis framework for identifying potential benefits in GPGPU applications. *SIGPLAN Not. 47*, 8 (Feb. 2012), 11–22.

[74] Smith, R., Goyal, N., Ormont, J., Sankaralingam, K., and Estan, C. Evaluating gpus for network packet signature matching. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2009), 175–184.

[75] Sreenivasa Murthy, G. Optimal loop unrolling for GPGPU programs. Master's thesis, Ohio State University, 2009.

[76] Tristram, D., and Bradshaw, K. Evaluating the acceleration of typical scientific problems on the GPU. In *SAICSIT '13: Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference* (New York, NY, USA, 2013), ACM, 17–26.

[77] Tristram, D., and Bradshaw, K. Parallelising k-difference calculations on the GPU. In *Southern African Telecommunications Networks and Applications Conference* (2013), 155 – 160.

[78] Tristram, D., Hughes, D., and Bradshaw, K. Accelerating a hydrological uncertainty ensemble model using graphics processing units (GPUs). *Computers & Geosciences 62* (2014), 178–186.

[79] Ukkonen, E. Finding approximate patterns in strings. *Journal of algorithms 6*, 1 (1985), 132–137.

[80] Volkov, V. Better performance at lower occupancy. In *Proceedings of the GPU Technology Conference, GTC* (2010), vol. 10.

[81] Volkov, V. Unrolling parallel loops. November 2011. Online: `http://www.cs.berkeley.edu/~volkov/volkov11-unrolling.pdf` [Accessed 01/11/13].

[82] Vuduc, R., Chandramowlishwaran, A., Choi, J., Guney, M., and Shringarpure, A. On the limits of GPU acceleration. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism* (2010), USENIX Association, 13–13.

[83] Walton, S. AMD Radeon HD 7970 Review. December 2011. Online: `http://www.techspot.com/review/481-amd-radeon-7970/` [Accessed 03/12/13].

[84] Wang, A. H. Detecting spam bots in online social networking sites: a machine learning approach. In *Proceedings of the 24th annual IFIP WG 11.3 working conference on Data and applications security and privacy* (2010), DBSec'10, Springer-Verlag, 335–342.

[85] Yen, T., and Reiter, M. K. Traffic Aggregation for Malware Detection. In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (Berlin, Heidelberg, 2008), DIMVA '08, Springer-Verlag, 207–227.

Appendix A

Classification Calculations

A.1 Required Computational Parallelism

As defined in Section 8.2.4:

$$RCP = \frac{(a+m)}{a}$$

where

a is the latency of arithmetic instructions,
and m is the latency of memory instructions.

To calculate RCP using the performance metrics given by the AMD Kernel Profiling Tool:

Let

n be the number of kernels;
$instructions_A$ be the total number of arithmetic instructions;
$instructions_M$ be the total number of memory instructions;
$latency_A$ be the latency of arithmetic instructions;
$latency_M$ be the latency of memory instructions;
$VALUInst_i$ be the number of arithmetic instructions for kernel i;
$VFetchInst_i$ be the number of vector memory instructions for kernel i; and
$SFetchInst_i$ be the number of scalar memory instructions for kernel i.

Then:

$$
\begin{aligned}
instructions_A &= \sum_{i=0}^{n-1} VALUInst_i \\
instructions_M &= \sum_{i=0}^{n-1} (VFetchInst_i + SFetchInst_i) \\
\therefore RCP &= \frac{(instructions_A \times latency_A) + (instructions_M \times latency_M)}{instructions_A \times latency_A}
\end{aligned}
$$

For the HD7970, we set $latency_A$ to 4 and $latency_M$ to 500 (see Section 8.2.4). The RCP calculations for each problem are given below using the above equations and the $VALUInst$, $VFetchInst$, and $SFetchInst$ figures given by the AMD Kernel Profiler for the unoptimised implementations of the problems.

A.1.1 Case Study 1

$$
\begin{aligned}
instructions_A &= \sum_{i=0}^{1} VALUInst_i \\
&= 2792 + 71828357 \\
&= 71831149 \\
instructions_M &= \sum_{i=0}^{1} (VFetchInst_i + SFetchInst_i) \\
&= (442 + 13) + (2144804 + 87) \\
&= 2145346 \\
\therefore RCP &= \frac{(instructions_A \times latency_A) + (instructions_M \times latency_M)}{instructions_A \times latency_A} \\
&= \frac{(71831149 \times 4) + (2145346 \times 500)}{71831149 \times 4} \\
&= 4.7
\end{aligned}
$$

A.1.2 Case Study 2

Standard Algorithm - Short Strings

$$
\begin{aligned}
instructions_A &= \sum_{i=0}^{0} VALUInst_i \\
&= 21468587 \\
instructions_M &= \sum_{i=0}^{0} (VFetchInst_i + SFetchInst_i) \\
&= (4484684 + 1014) \\
&= 4485698 \\
\therefore RCP &= \frac{(instructions_A \times latency_A) + (instructions_M \times latency_M)}{instructions_A \times latency_A} \\
&= \frac{(21468587 \times 4) + (4485698 \times 500)}{21468587 \times 4} \\
&= 27.1
\end{aligned}
$$

Standard Algorithm - Long Strings

$$
\begin{aligned}
instructions_A &= \sum_{i=0}^{0} VALUInst_i \\
&= 455474468 \\
instructions_M &= \sum_{i=0}^{0} (VFetchInst_i + SFetchInst_i) \\
&= (100558278 + 264) \\
&= 100558542 \\
\therefore RCP &= \frac{(instructions_A \times latency_A) + (instructions_M \times latency_M)}{instructions_A \times latency_A} \\
&= \frac{(455474468 \times 4) + (100558542 \times 500)}{455474468 \times 4} \\
&= 28.6
\end{aligned}
$$

HBP Algorithm - Short Strings

$$
\begin{aligned}
instructions_A &= \sum_{i=0}^{0} VALUInst_i \\
&= 5849788 \\
instructions_M &= \sum_{i=0}^{0} (VFetchInst_i + SFetchInst_i) \\
&= (111184 + 1016) \\
&= 112200 \\
\therefore RCP &= \frac{(instructions_A \times latency_A) + (instructions_M \times latency_M)}{instructions_A \times latency_A} \\
&= \frac{(5849788 \times 4) + (112200 \times 500)}{5849788 \times 4} \\
&= 3.4
\end{aligned}
$$

HBP Algorithm - Long Strings

$$
\begin{aligned}
instructions_A &= \sum_{i=0}^{0} VALUInst_i \\
&= 72950241 \\
instructions_M &= \sum_{i=0}^{0} (VFetchInst_i + SFetchInst_i) \\
&= (988806 + 226) \\
&= 989032 \\
\therefore RCP &= \frac{(instructions_A \times latency_A) + (instructions_M \times latency_M)}{instructions_A \times latency_A} \\
&= \frac{(72950241 \times 4) + (989032 \times 500)}{72950241 \times 4} \\
&= 2.7
\end{aligned}
$$

A.1.3 Case Study 3

$$
\begin{aligned}
instructions_A &= \sum_{i=0}^{1} VALUInst_i \\
&= 150 + 224 \\
&= 374 \\
instructions_M &= \sum_{i=0}^{1} (VFetchInst_i + SFetchInst_i) \\
&= (1 + 11) + (5 + 13) \\
&= 30 \\
\therefore RCP &= \frac{(instructions_A \times latency_A) + (instructions_M \times latency_M)}{instructions_A \times latency_A} \\
&= \frac{(374 \times 4) + (30 \times 500)}{374 \times 4} \\
&= 11
\end{aligned}
$$

A.2 Data Transfer Overhead

A 'what-if' approach to estimating data transfer overhead:

Let

n be the number of inputs;
$data_{in}$ be the data transferred to the GPU (in bytes) independent of n;
$datapi_{in}$ be the extra data transferred to the GPU (in bytes) per increment of n;
$datapi_{out}$ be the extra data transferred from the GPU (in bytes) per increment of n;
$rate$ be the estimated rate of transfer to and from the GPU (in bytes/ms);
t_{CPU} be the execution time of the existing implementation (in ms);
t_{GPU} be the estimated execution time of GPU implementation (in ms);
t_{data} be the estimated data transfer time (in ms); and
$speedup$ be a desired or estimated speedup over the existing implementation.

Then:

$$
\begin{aligned}
t_{data} &= \frac{data_{in} + n \times (datapi_{in} + datapi_{out})}{rate} \\
t_{GPU} &= \frac{t_{CPU}}{speedup} \\
\therefore transfer\_overhead &= \frac{t_{data}}{t_{GPU} + t_{data}} \times 100
\end{aligned}
$$

We estimate our *rate* to be 10 GB/s for our system. The data transfer overhead calculations for each case study are given below.

A.2.1 Case Study 1

When n is 50,000 ensembles on a test dataset:

$data_{in}$ is 57,540,954 bytes,
$datapi_{in}$ is 164 bytes,
$datapi_{out}$ is 212 bytes, and
t_{CPU} is 219,500 ms.

Assume *speedup* is 10x:

$$
\begin{aligned}
t_{data} &= \frac{data_{in} + n \times (datapi_{in} + datapi_{out})}{rate} \\
&= \frac{57540954 + 50000 \times (164 + 212)}{10 \times 2^{30} \times 10^{-2}} \\
&= 7 \\
t_{GPU} &= \frac{t_{CPU}}{speedup} \\
&= \frac{219500}{10} \\
&= 21950 \\
\therefore transfer\_overhead &= \frac{7}{21950 + 7} \times 100 \\
&= 0.03\%
\end{aligned}
$$

A.2.2 Case Study 2

These calculations are based on the scenario where n is 8,192 strings, the number of test strings is 2,048, and the difference threshold is 4%.

HBP Algorithm - Short Strings

With an average string length of 64 characters:

$data_{in}$ is 131,072 bytes,
$datapi_{in}$ is 64 bytes,
$datapi_{out}$ is 8,192 bytes, and
t_{CPU} is 2,781 ms.

Assume *speedup* is 10x:

$$\begin{aligned} t_{data} &= \frac{data_{in} + n \times (datapi_{in} + datapi_{out})}{rate} \\ &= \frac{131072 + 8192 \times (64 + 8192}{10 \times 2^{30} \times 10^{-2}} \\ &= 6.3 \\ t_{GPU} &= \frac{t_{CPU}}{speedup} \\ &= \frac{2781}{10} \\ &= 278.1 \\ \therefore transfer\_overhead &= \frac{6.3}{278.1 + 6.3} \times 100 \\ &= 2.22\% \end{aligned}$$

HBP Algorithm - Long Strings

With an average string length of 560 characters:

$data_{in}$ is 1,146,880 bytes,
$datapi_{in}$ is 560 bytes,
$datapi_{out}$ is 8,192 bytes, and

t_{CPU} is 15,303 ms.

Assume *speedup* is 10x:

$$\begin{aligned} t_{data} &= \frac{data_{in} + n \times (datapi_{in} + datapi_{out})}{rate} \\ &= \frac{1146880 + 8192 \times (560 + 8192}{10 \times 2^{30} \times 10^{-2}} \\ &= 6.8 \\ t_{GPU} &= \frac{t_{CPU}}{speedup} \\ &= \frac{15303}{10} \\ &= 1530.3 \\ \therefore transfer\_overhead &= \frac{6.8}{1530.3 + 6.8} \times 100 \\ &= 0.44\% \end{aligned}$$

Standard Algorithm

The same amount of data is transferred for the HBP and standard algorithms, and thus the data transfer overhead of the standard algorithm can be expressed in terms of the overhead calculated for the HBP algorithm:

Short Strings The CPU HBP algorithm is 2.35x faster than the standard algorithm. Therefore, for a 10x speedup, the data transfer overhead is $\frac{2.27}{2.35} = 0.96\%$.

Long Strings The CPU HBP algorithm is 27x faster than the standard algorithm. Therefore, for a 10x speedup, the data transfer overhead is $\frac{0.44}{27} = 0.016\%$.

A.2.3 Case Study 3

When n is 2^{26} integers:

$data_{in}$ is 0 bytes;

$datapi_{in}$ is 4 bytes;
$datapi_{out}$ is 4 bytes; and
t_{CPU} is 347 ms.

Assume *speedup* is 10x:

$$\begin{aligned}
t_{data} &= \frac{data_{in} + n \times (datapi_{in} + datapi_{out})}{rate} \\
&= \frac{0 + 2^{26} \times (4 + 4)}{10 \times 2^{30} \times 10^{-2}} \\
&= 50 \\
t_{GPU} &= \frac{t_{CPU}}{speedup} \\
&= \frac{347}{10} \\
&= 34.7 \\
\therefore transfer_overhead &= \frac{50}{34.7 + 50} \times 100 \\
&= 59\%
\end{aligned}$$