All-Pairs Shortest Path Algorithms Using CUDA

KEMP, JEREMY,MARK

All-Pairs Shortest Path Algorithms Using CUDA

Jeremy M. Kemp

Utilising graph theory is a common activity in computer science. Algorithms that perform computations on large graphs are not always cost effective, requiring supercomputers to achieve results in a practical amount of time. Graphics Processing Units provide a cost effective alternative to supercomputers, allowing parallel algorithms to be executed directly on the Graphics Processing Unit. Several algorithms exist to solve the All-Pairs Shortest Path problem on the Graphics Processing Unit, but it can be difficult to determine whether the claims made are true and verify the results listed. This research asks "Which All-Pairs Shortest Path algorithms solve the All-Pairs Shortest Path problem the fastest, and can the authors' claims be verified?" The results we obtain when answering this question show why it is important to be able to collate existing work, and analyse them on a common platform to observe fair results retrieved from a single system. In this way, the research shows us how effective each algorithm is at performing its task, and suggest when a certain algorithm might be used over another.

All-Pairs Shortest Path Algorithms Using CUDA

Jeremy M. Kemp

Submitted for the degree of MScR Computer Science to the School of Engineering and Computing Sciences, Durham University, 2012

Contents

List of Figures

List of Tables

Glossary

API Application Programming Interface.

APSP All-Pairs Shortest Path.

CPU Central Processing Unit.

CUDA Compute Unified Device Architecture.

device memory is the largest storage available on the GPU, similar to RAM on a PC.

DRAM Dynamic Random Access Memory.

GB Gigabyte (10^9 bytes).

GPGPU General-Purpose Computing on Graphics Processing Units.

GPU Graphics Processing Unit.

KB Kilobyte (10^3 bytes).

kernel is a CUDA function that allows code to be executed in parallel on the GPU.

MB Megabyte (10^6 bytes).

MIMD Multiple Instruction, Multiple Data.

MISD Multiple Instruction, Single Data.

RAM Random Access Memory.

SIMD Single Instruction, Multiple Data.

SISD Single Instruction, Single Data.

SSSP Single-Source Shortest Path.

thread block is a collection of CUDA threads that all execute on a single CUDA core.

warp is a group of CUDA threads that reside in a thread block, and that all execute the same instructions at the same time. Usually, in groups of 32.

Copyright

Acknowledgements

I would like to thank my supervisor, Professor Iain Stewart, for his many useful and insightful suggestions in regards to this work and the advice given during our many meetings! I would also like to thank my family for their continued support during my time at Durham, as well as Charlotte Hawkins for checking my work many times over!

Chapter 1

Introduction

1.1 Research Overview

Parallel computing on the Graphics Processing Unit (GPU) has been selected for use in an increasing number of systems and applications. The software supporting parallel computing on the GPU is becoming more and more comprehensive, with multiple Application Programming Interfaces (APIs) supporting the so-called General-Purpose Computing on Graphics Processing Units (GPGPU) era of parallel computing. With the growth in availability of said APIs, coupled with powerful, discrete GPUs, software developers and researchers are putting a greater amount of effort into parallelising their software to leverage the excellent performance benefits that GPGPU has to offer.

A lot of effort has gone into creating highly optimised solutions on the Central Processing Unit (CPU) in an attempt to squeeze as much performance out of existing CPU technology as possible. A common method of such optimisation is making the application dependent on a particular CPU architecture, in order to gain the benefits of using every feature available on that architecture. GPGPU programming allows applications to be offloaded onto the GPU and enjoy much greater performance than is currently available on the CPU by simply leveraging hardware that already exits, and has existed, in modern computers for many years.

Not all applications can enjoy this benefit however, as some problem areas are far more susceptible to parallel computing than others. The research area of graph theory has had some slight focus on GPGPU in recent years, with several solutions being developed for classic graph theory problems such as state space searching and implementing graph cuts (Vineet and Narayanan, 2008).

There has been an effort of research, investigating the All-Pairs Shortest Path (APSP) problem using Compute Unified Device Architecture (CUDA). CUDA is a GPGPU API from NVIDIA and is explained in much greater detail in Chapter 2. Judging the results of this research, and determining which algorithms are best to use in a given situation can be difficult, especially as they are often tested on completely different platforms, with different inputs and analysis. This research asks “Which All-Pairs Shortest Path algorithms solve their problem the fastest, and can the authors’ claims be verified?”.

1.2 Intended Outcomes

As discussed in Section 1.1, this research intends to look at APSP algorithms with CUDA, and compare their performance against each other. In doing so, several additional research questions were considered and formulated to deliver:

- How do CUDA algorithms compare against their CPU counterparts?
- Can these CUDA algorithms be improved or modified in any beneficial way?

1.3 Thesis Overview

This thesis aims to answer the research question that was asked in Section 1.1.

In Chapter 2, we look at key concepts behind parallel computing, GPGPU, and CUDA. In doing so, several frameworks around parallel computing and parallel computing classifications are examined. Some problems with parallel computing are examined, such as race conditions and deadlocks as well as the problem of barriers. In classifying a parallel computer, Flynn's Taxonomy is observed, providing a solid ground for parallel classification. Additionally, Chapter 2 looks at the world of GPGPU and describes in detail NVIDIAs CUDA API.

In Chapter 3, we investigate the APSP problem, and how it can be solved on the CPU with several different algorithms using varying techniques. The algorithms observed serve as the basis of the algorithms that this thesis will implement with CUDA and so are important to understand.

In Chapter 4, the APSP problem is looked at in greater detail, specifically in relation to how the problem can be solved with CUDA. Firstly, existing methods of storing graphs on the GPU are examined, weighing their benefits and shortcomings against each other. Finally, the CUDA implementations are described in detail, complete with algorithmic listings showing their pseudo code. Improvements to selected algorithms are also shown where possible, as well as limitations that hamper algorithms where applicable.

In Chapter 5, the results of all CUDA and CPU algorithms are analysed, comparing their results with each other. The authors' claims are also examined to see whether their comments can be verified. Each CUDA algorithm is cross examined with every other, in an attempt to determine if there is a clear winner amongst them, or if some are suited to specific tasks.

Finally, in Chapter 6, the findings of this thesis are summarised, providing a clear overview of the entire body of work, the considerations to be taken into account when creating CUDA algorithms, and areas for future work.

Chapter 2

Definitions

2.1 Parallel Computing

Traditionally, computer programs have been written for standard CPUs, i.e., they have been written with sequential execution in mind. A sequential program executes instructions in order, with each instruction occurring after the previous instruction has completed.

Parallel computing is "a form of computation in which many calculations are carried out simultaneously" (Almasi and Gottlieb, 1988). In order to obtain parallelism, the computer hardware must be designed with parallel execution in mind, so that many instructions can be executed at the same time. The hardware could simply include having multiple processors or cores in the CPU, having networked computers execute parallel executions, super computers, and now, GPU.

Figures 2.1 and 2.2 show a visual comparison between a standard sequential algorithm and a parallel algorithm running on a CPU with one core and a CPU with four cores respectively. Barney (2010) describes a useful example to help illustrate how a parallel algorithm relates to the real world. He states that parallel computing is simply an evolution of sequential computing that attempts to emulate what has always occurred in the real world with many complex, interrelated events happening at the same time while also in sequence.

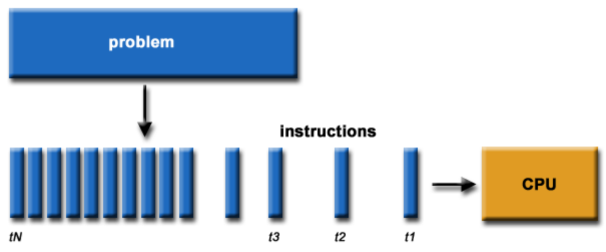

Figure 2.1: A representation of a sequential algorithm on the CPU (Barney, 2010)

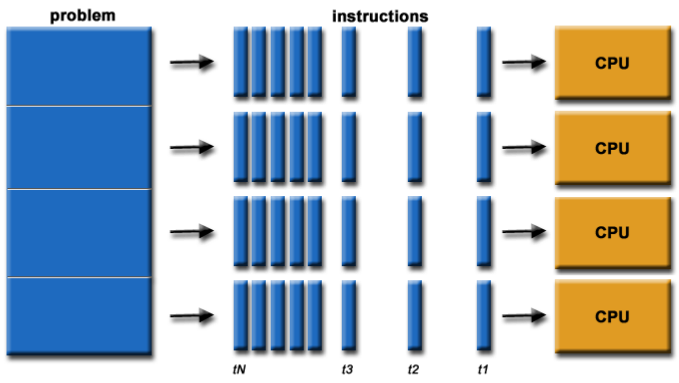

Figure 2.2: A representation of a parallel algorithm on multiple CPU (Barney, 2010)

2.1.1 Bernstein's Conditions

There are many challenges in creating parallel algorithms. Data dependency issues are key in implementing parallel algorithms, as not fully comprehending them can severely affect the performance of an algorithm. In understanding the data dependencies of a sequential program, we can see whether it can be successfully parallelised or not.

Bernstein (1966) devised a set of conditions that must exist if two or more processes can be executed in parallel. We say that I_i is the set of all inputs for a process P_i. Similarly, O_i is the set of all outputs for a process P_i.

When given two processes P_1 and P_2, they may execute in parallel if the following rules are observed:

$$I_1 \cap O_2 = \emptyset$$

$$I_2 \cap O_1 = \emptyset$$

$$O_1 \cap O_2 = \emptyset$$

The rules defined by Bernstein (1966) state that two processes cannot execute in parallel unless they are flow independent (rule one), anti independent (rule two) and output independent (rule three).

Flow Dependent S_1 precedes S_2 where at minimum one output of S_1 is an input to S_2

Anti Independent S_1 precedes S_2 where the output of S_2 overlaps input to S_1

Output Independent S_1 and S_2 write to the same unique output

For example, Algorithm 1 below cannot be implemented in parallel successfully as there are issues with flow dependency. If we look at line four, we can see that this line cannot be executed before line three, as line four requires an input that depends on the outcome of line three.

However, Algorithm 2 is an example of a program that may be implemented in parallel, as there are no dependencies between data and instruction. Each line is independent and does not depend on the outcome of any other line.

Algorithm 1 dependency(int i, int j)

```
1: int k;
2: int l;
3: k = i * j;
4: l = 3 * k;
```

Algorithm 2 noDependency(int i, int j)

```
1: int k;
2: int l;
3: int m;
4: k = i * j;
5: l = 3 * j;
6: m = i + j;
```

2.1.2 Common Problems with Parallel Programming

Often, when creating parallel algorithms, desired tasks are split into threads whose purpose is to solve some task in parallel with other threads. Often, multiple threads will want to read, and modify a common variable in order to perform some task. This can lead to a serious problem known as a race condition. Race conditions occur when separate threads both depend on a shared state. Without proper management, the threads can hold incorrect data or process incorrect data that has not been updated correctly by a different thread (Netzer et al., 1992).

Consider Algorithm 3 that helps to clarify race conditions. T_i refers to a resident thread of the parallel algorithm. Likewise, T_i refers to a register.

Algorithm 3 raceCondition()

```
1: int i = 0
2: $T_1$ reads i into $R_1$
3: $T_2$ reads i into $R_2$
4: $T_1$ i = i + 1 (in $R_1$)
5: $T_2$ i = i + 1 (in $R_2$)
6: $T_1$ writes $R_1$ back to memory
7: $T_2$ writes $R_2$ back to memory
```

In Algorithm 3, the result in i at the end of the algorithm is 1. However, the expected result is 2. To avoid this common problem, mutual exclusion must be provided by using a lock. The lock will allow a thread to assume control of a variable (in this case, i) and therefore stop any other thread from reading and/or writing to it until the controlling thread has released the lock.

In utilising locks to solve race conditions, another serious problem is introduced. Deadlocks occur when two or more threads are waiting for the other(s) to finish and neither of them ever do (Silberschatz et al., 2006). For example, imagine two threads (T_x) and two printers (R_x). Now, imagine each thread requesting the other's printer. This situation will cause a deadlock as the printers have not yet been released by the original threads. E.g. T_1 is in control of R_1, but is also requesting R_2. However, T_2 currently controls R_2; causing a deadlock. This form of deadlock is known as circular deadlock, or a circular chain, and can be seen in Figure 2.3.

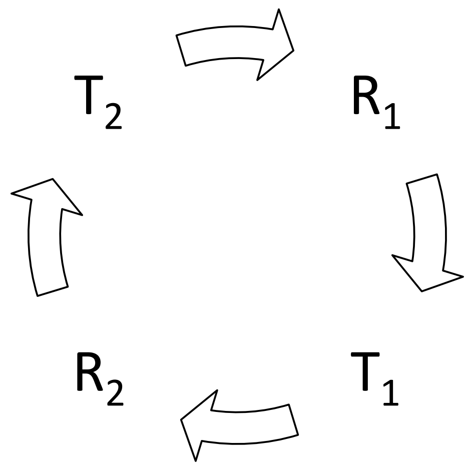

Figure 2.3: A Diagram Showing Two Threads in Circular Deadlock

2.1.3 Flynn's Taxonomy

When looking at the world of parallel computing, there are several ways in which you can classify a parallel computing machine. These classes could be based on the hardware architecture of the machine. For example, Flynn (1972) presents a method of classifying a parallel computing machine based on its hardware architecture, and therefore, programmability.

Flynn's classification is based on two separate dimensions, Instruction and Data. Furthermore, these dimensions are split into two states, Single or Multiple. This leads to four possible classifications that form Flynn's Taxonomy and can be seen in Table 2.1.

	Single Instruction	**Multiple Instruction**
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Table 2.1: Flynn's Taxonomy

As we will see in Section 2.4, the GPU used for this project is a Single Instruction, Multiple Data (SIMD) processor, capable of performing thousands of identical instructions on any number of pieces of data.

Single Instruction, Single Data (SISD)

- Only a **Single Instruction** is being executed by the CPU during any given clock cycle.
- Only a **Single Data** is being used as input for the current instruction during any given clock cycle.
- "Represents most conventional computing equipment available today" (Flynn, 1972).

Previous Instruction
load A(1)
load B(1)
C(1) = A(1) + B(1)
store C(1)
Next Instruction

Table 2.2: Single Instruction, Single Data

Single Instruction, Multiple Data (SIMD)

- Only a **Single Instruction** is being executed by the CPU during any given clock cycle.
- **Multiple Data** can be used by each processor to allow for multiple inputs.
- Best suited for systems with multiple streams of data, with a single instruction. E.g. a modern GPU.

Previous Instruction
load A(1)
load B(1)
C(1) = A(1) + B(1)
store C(1)
Next Instruction

(a) Processor 1

Previous Instruction
load A(n)
load B(n)
C(n) = A(n) + B(n)
store C(n)
Next Instruction

(b) Processor n

Table 2.3: Single Instruction, Multiple Data

Multiple Instruction, Single Data (MISD)

- **Multiple Instructions** are being executed by each processor during any given clock cycle.
- Only a **Single Data** is used by each processor for input in any given clock cycle.

Previous Instruction
load A(1)
load B(1)
C(1) = A(1) + 1
store C(1)
Next Instruction

(a) Processor 1

Previous Instruction
load A(1)
load B(1)
C(1) = A(1) + 1
store C(1)
Next Instruction

(b) Processor n

Table 2.4: Multiple Instruction, Single Data

Multiple Instruction, Multiple Data (MIMD)

- **Multiple Instructions** are being executed by each processor during any given clock cycle.
- **Multiple Data** can be used by each processor to allow for multiple inputs.
- Execution on a MIMD can be either synchronous or asynchronous.
- The most common form of parallel computer.
- The majority of the world's super computers follow the MIMD architecture.

Previous Instruction
load A(1)
load B(1)
C(1) = A(1) + B(1)
store C(1)
Next Instruction

(a) Processor 1

Previous Instruction
load Z(n)
foo()
bar()
while Z is true
Next Instruction

(b) Processor n

Table 2.5: Multiple Instruction, Multiple Data

2.2 GPGPU

In recent years, the advent of GPGPU has popularised the use of parallel computing on the GPU in achieving significant performance gains on a relatively cheap hardware device. GPGPU is a method of using the GPU to perform computations that would usually be executed by the CPU, rather than performing calculations to handle computer graphics, as is their traditional use. When the GPU is used for GPGPU, it can be viewed as a coprocessor to the CPU, offloading complex tasks that the GPU can tackle in parallel.

GPGPU provides an extremely cost effective alternative for parallel algorithms that would normally be exclusive to supercomputers, with a low-end CUDA enabled GPU costing approximately £25 compared to a super computer such as IBM's Blue Gene system at $1.3million.

Multiple GPUs in a single system can be utilised for a single problem, often increasing the performance of parallel applications. This project does not utilise multiple GPU however. The applications of GPGPU are far reaching and include some of the following:

- Graph Theory.
- Ray Tracing.
- Matrix and/or Vector Operations.
- Signal Processing.
- Image Processing.
- Speech Recognition.
- Physics Simulations.
- Medical Computation.

Multiple GPGPU APIs exist to utilise the GPU for parallel computing. Popular APIs include NVIDIAs CUDA, OpenCL Khronos (2011) and DirectX's DirectCompute platform (Microsoft, 2012). This research focuses solely on NVIDIAs CUDA API. Each have their advantages and disadvantages, but all provide a solid parallel computing API to utilise the powerful hardware of modern GPU.

Modern graphics cards have a specialised hardware architecture that can be represented as a parallel computer. Unlike traditional graphics cards, GPUs such as NVIDIAs 580GTX are equipped with 16 multiprocessors, each with 32 cores, providing an impressive 512 cores. Each core has access to a global bank of memory, much like the Random Access Memory (RAM) on a PC, as well as a block of shared memory per multiprocessor which provides fast storage that can be used to share data between parallel processes. The potential of GPGPU is extremely great, given this unique hardware architecture that can provide great performance benefits to algorithms at a relatively low cost.

2.3 What is CUDA?

CUDA is a parallel computing solution developed by NVIDIA, encompassing both a software and hardware architecture for using an NVIDIA GPU as a parallel computing device without the need for a graphics API. CUDA is available for all NVIDIA GPU following (and including) their G80 series of GPUs.

The CUDA API is an extension of the C programming language, providing programmers with a set of tools to create parallel algorithms. By providing the API in C, CUDA gives many programmers who already know C to quickly pick up their tools and begin creating CUDA applications.

CUDA enabled GPUs now have an install base of at least 100 million units (NVIDIA, 2009). Clearly, from this number, parallel algorithms utilising CUDA can be distributed easily to the mass market with a large number of machines supporting the technology, making CUDA an ideal candidate to boost the performance of a wide range of applications, both academically and commercially, examples of which are given in Section 2.2.

As a parallel computing platform, CUDA is designed to run thousands of threads at the same time, each thread executing the same code, but acting on multiple pieces of data, usually chosen programmatically to ensure that each thread works on a different pieces of unique data. Using this method, applications can be executed on the GPU, rather than the CPU as described above.

The CUDA API provides both high and low level APIs to suit the programmers needs. The lower level API provides a greater level of granularity and closeness to the underlying hardware, but decreases the readability and maintainability of CUDA code. These APIs are known as the runtime and driver APIs, respectively. In older versions of CUDA, the driver API provided a greater level of detail in querying the GPU memory, in providing more information than the runtime API. However, large strides have been made in the latest CUDA releases, both in API usability, and CUDA compiler performance. The two APIs are mutually exclusive however, and their use must never overlap.

Despite providing greater control, the lower level driver API does not provide a performance increase over runtime code, and should simply be used if a greater amount of control over the GPU is required. Older versions of CUDA provided an emulator, so that CUDA may be programmed without the presence of a CUDA GPU. Emulation was not supported by the driver API, and the emulation program was deprecated with CUDA 3.0.

Since its inception, there has been strong evidence showing that parallel algorithms on the GPU can greatly improve the performance of classic problems when compared to their sequential (CPU) equivalents, providing a justification for research in this area.

2.4 CUDA Hardware Model

The architecture of a CUDA enabled GPU can be represented as a massive SIMD processor, examined in further detail in Section 2.1.3. A CUDA device consists of a number of multiprocessors, each with an identical number of processors (cores). Key to the architecture of CUDA devices is the different types of memory available, and their layout. Or, in other words, CUDAs memory hierarchy.

Table 2.6 gives a brief overview of the differing memory types, their access types, scope and locality. Additionally, Figure 2.4 shows these different types of memory that form CUDAs memory hierarchy and how they interact with the CUDA architecture on a higher level.

Memory	**On/Off Chip**	**Cached**	**Access**	**Scope**	**Lifetime**
Register	On	N/A	R/W	Thread	Thread
Local	Off	Compute 2.x	R/W	Thread	Thread
Shared	On	N/A	R/W	Block	Block
Global	Off	Compute 2.x	R/W	Host + Device	Host
Constant	Off	Yes	R	Host + Device	Host
Texture	Off	Yes	R	Host + Device	Host

Table 2.6: Device Memory Features (NVIDIA, 2012)

Firstly and perhaps most importantly is shared memory. Shared memory is located directly on-chip with the multiprocessors, providing extremely fast read and write times to and from the processor. In utilising shared memory, an impressive performance gain of over 100x can be gained over global memory (NVIDIA, 2011a).

When possible, the greatest amount of data should be moved from device memory into shared memory to try and squeeze as much performance out of CUDA as possible. Once data is in shared memory, computations can be performed there before writing the results back to device memory and thus, reducing the effects of latency between the multiprocessors and device memory. Latency simply describes the delay in clock cycles between some action being requested, and the action completing.

Unfortunately, shared memory is very small in comparison to the other memory types in CUDA. Older compute devices had just 16kb per multiprocessor to leverage. Newer compute devices however are graced with an additional 16kb, totalling 32kb per multiprocessor. Using shared memory wherever possible in CUDA code is clearly very beneficial from a performance standpoint when utilised correctly, but the programmer should be wary of the memory constraints that go hand-in-hand with shared memory.

As well as shared memory, registers are located on-chip providing extremely quick access for local variables stored in CUDA code. The use of too many registers, increasing register pressure, can have a negative effect on system performance and is explained in more detail in Section 2.6.

As we can see in Figure 2.4, texture and constant caches are provided on top of shared memory, also located on-chip. Cache memory is read only, and must be populated with data before CUDA code is executed. This can be performed with one of the many memory allocation features provided by the CUDA driver and runtime APIs. Whilst not as fast as shared memory, they provide a greater amount of storage, and are significantly faster than device memory (global memory).

Device memory, (also known as global memory or Dynamic Random Access Memory (DRAM)) is available to all multiprocessors and their cores, effectively acting as the GPUs RAM. Device memory offers by far the greatest amount of storage capacity on the GPU but also suffers from being the slowest of all forms of memory. Device memory takes several clock cycles to both read and write data to and from the multiprocessors. It is often necessary to use device memory, due to its sheer capacity, so its speed must be kept in mind at all times to ensure the greatest performance benefits when programming for CUDA.

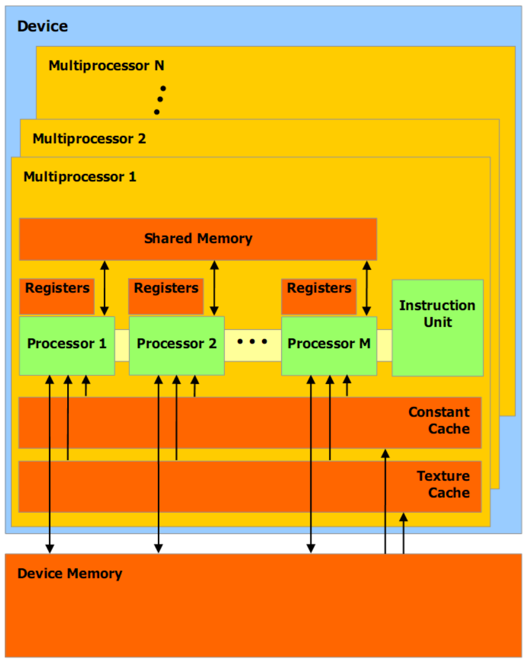

Figure 2.4: CUDA Hardware Model, Demonstrating Memory Hierarchy and Overall Hardware Architecture of CUDA GPUs (NVIDIA, 2009)

Fundamentally, understanding the benefits and drawbacks of these contrasting memory types is important and can greatly affect the performance of code. In knowing which storage type to use before creating a CUDA application, we can speed-up our applications as much as possible.

When programming for CUDA, it is important to take into consideration the time taken to physically move data between the host and the device. Minimising data transfer between host and device is important because those transfers are subject to much lower bandwidth than when moving data internally on the device (NVIDIA, 2011a). In some cases, it may be beneficial to just compute the data you need on the device, rather than copying it from host memory to device memory. Due to the high bandwidth, low latency of shared memory, it is always beneficial to use it wherever possible. Either by copying data from device memory to shared, and performing calculations there, or by simply computing the data required directly into shared memory (NVIDIA, 2011a).

2.4.1 Coalesced Memory Access

"Perhaps the single most important performance consideration in programming for the CUDA architecture is coalescing global memory accesses" (NVIDIA, 2011a).

Dehne and Yogaratnam (2010) state that the goal of coalescing memory access is to combine multiple global memory access requests, by multiple threads, concurrently, into a single memory transaction for an independent portion of memory. The benefits of using this technique are vast, greatly improving the performance of the application. Using coalesced memory access can be a difficult task to master, as the GPU hardware support for the system has changed quite significantly with each version of CUDA.

Early versions of CUDA (1.0/1.1) required that the kernel explicitly align memory access patterns so that each thread had to access consecutive memory blocks that related to the order of the threads. Kernels are explained in greater detail in Section 2.5. Imagine four threads, T_0 to T_3. To achieve coalesced memory access, each thread must access memory locations A_0 to A_3 where $A_0 < A_1 < A_2 < A_3$ and are in a block of contiguous memory (Dehne and Yogaratnam, 2010). The memory accesses by these threads are coalesced using a half-warp (explained a little further on) of threads, where a full warp consists of thirty two threads. In this way, sixteen thirty two bit reads are coalesced into one sixty four byte memory access. As noted by Dehne and Yogaratnam (2010), this method of memory coalescing is really quite inflexible and rather complicated to implement successfully.

With the release of CUDA 1.2, the rules for coalesced memory access were relaxed, allowing for easier, and more successful use of the system (NVIDIA, 2011b). With CUDA 1.2 and above, if sixteen data accesses fit into a thirty two byte memory segment, then a single memory access of thirty two bytes is performed. If however, those sixteen accesses do not fit into a thirty two byte segment, but do fit into a sixty four byte segment, a sixty four byte segment is performed instead.

If the data stored in global memory does not map well for coalescing, it can be beneficial to pad your data so that it may match the coalesced access patterns. Padding data simply means to add extra data that has no meaning in order to achieve some storage constraint. In that way, you can still benefit from the performance improvement of coalesced access. This is only possible however, if you have enough free memory that you can waste with data padding (NVIDIA, 2011b). Clearly, using this coalesced system allows for a significant performance improvement by allowing multiple pieces of data to be accessed in parallel, rather than sequentially.

2.5 CUDA Software Model

As explained in Section 2.3, two APIs are provided in order that CUDA might be programmed. Both APIs allow programmers to write special functions known as kernels that will be executed on the GPU. These kernels are structured in the same way as normal C functions, but are provided with additional intrinsics such as $threadIdx$ that allow the executing thread to access information about itself. In this case, a 3-Dimensional vector ($threadIDx$) that holds the current thread's address. Each component of the vector represents the threads x, y, and z co-ordiantes inside the thread block (explained below). This information can be used in a number of ways; most commonly in determining what data the thread should operate on.

In order to create a CUDA application, it is not necessary to understand the underlying hardware architecture, as it is hidden from the programmer. While a programmer does not need to understand the hardware architecture, it is extremely beneficial in being able to gain the most out of CUDA. By understanding the hardware architecture, as well as the intricate details of how CUDA threads operate and interact, the programmer can tailor his or her kernels to obtain the best performance possible from the code in utilising the many memory types and features of the CUDA architecture.

Instead of seeing the CUDA architecture when programming, threads are seen as being organised into blocks. Blocks are a convenient structure to think about threads in. A block is simply a 1, 2, or 3-Dimensional structure in which threads reside. In this way, groups of threads can easily be partitioned, allowing the programmer to easily decide how and where blocks should operate on data, and how shared memory should be utilised. Each thread is executed following the Single Program, Multiple Data (SPMD) model (see NVIDIA (2012)).

A programmer can define how many threads are executed for each kernel that is written. Taking the NVIDIA 8800GTX as an example, the programmer can define no more than 512 threads per block. Blocks can also be ordered into grids, with each grid holding at most 2^{32} blocks. Therefore, 2^{41} total threads can be executed per kernel. CUDA handles the assignment of threads and blocks to multiprocessors as well as other tasks including thread scheduling by utilising its GigaThread technology (see NVIDIA (2011b)).

The way in which threads are scheduled on the GPU differs greatly from the CPU. Whilst one might say that threads execute independently on the CPU, CUDA threads are scheduled in groups. These groups, known as warps, execute following the Single Instruction Multiple Thread (SIMT) model as described in Section 2.1.3. The minimum number of threads per warp is 32, with each thread inside the warp executing exactly the same instruction. Therefore, if the code being executed by the warp contains branches, each branch is expanded by filling in with null values where appropriate. Clearly, avoiding branches is critical as the performance of algorithms containing branches degrade as thread execution time is increased by expanding both branches.

CUDA provides functions that allow threads to be synchronised within a block. This synchronisation process acts as a barrier within the kernel which forces all threads in a block to hang until every thread in the block has reached the barrier. Blocks cannot be synchronised within a grid. Threads can be addressed using either a 1, 2, or 3-Dimensional index. Likewise, blocks may be addressed by a 1, 2, or 3-Dimensional index. CUDA provides thread and block

ID variables which can be used in a variety of ways to ensure that the threads in a kernel are performing the correct tasks on the correct piece or pieces of data.

A very important aspect of a block, is that threads within a block can communicate via the GPU shared memory. This is the only form of thread communication available with CUDA. The GPU automatically schedules where and how blocks should be executed on the device. Blocks are always contained to one core, i.e. a block and its threads can never be split between different cores. This restriction ensures that each thread in the block can communicate via shared memory as shared memory is located on-chip (as discussed in Section 2.4). Organising threads and the data allocated to shared memory is often a complex task, with additional issues such as avoiding bank conflicts.

Bank conflicts occur where one block's shared memory data overlaps another's. Bank conflicts are discussed in more detail in Section 2.8. Evidently, utilising shared memory is highly beneficial but requires skill to accomplish it successfully. As mentioned previously, the programmer specifies the number of threads that are used for each kernel. The programmer can also specify the block size that is to be used for each kernel. A kernel can also be executed multiple times with differing thread and block sizes for each execution to create the desired results.

CUDA refers to the GPU as the device, whereas the CPU is the host. Executing a kernel does not stop the host from executing it's own code, allowing both device and host code to run simultaneously. This feature was only introduced in a recent version of CUDA however, and kernel calls used to be blocking. Having blocking code means that the CPU would not be able to continue the CPU section of a CUDA program until the kernel returned control to the host.

If the host wishes to access device memory whilst a kernel is executing, it is necessary for the kernel to finish its execution. Therefore, the host code is blocked until kernel execution finishes, at which point the host may proceed. As of CUDA compute version 1.1 and above, asynchronous memory access is supported whilst a kernel is executing, allowing host code to access device memory during kernel execution. This feature can be useful in certain problem domains, but for this project, its use is limited at best.

If the programmer wishes, multiple kernels can be executed asynchronously, allowing multiple differing tasks to be completed at once. In this regard, the kernels must be carefully written to ensure that enough memory is available for each kernel and that performance isn't harmed by executing more than one kernel at any one time.

2.6 Occupancy

CUDA executes instructions sequentially within a thread block, so executing a warp whilst another is paused or blocked is the only way in which CUDA can hide latencies and attempt to keep the GPU busy. CUDA defines a metric, occupancy, that allows determination of how effectively the GPU is being kept busy by CUDA (NVIDIA, 2012).

NVIDIA define occupancy as the "ratio of the number of active warps per multiprocessor to the maximum number of possible active warps" (NVIDIA, 2012). Having a low occupancy can interfere with the ability of CUDA to hide the performance issues related to memory latency which in turn results in a decrease in performance of CUDA code. Conversely, having a high occupancy rating does not always equal a higher performance rating of CUDA code. NVIDIA (2012) state that there is a point in which additional occupancy does not improve CUDAs performance.

Register availability is one of the major factors that can affect the occupancy of CUDA code. Kernels use registers to enable threads to store local variables in extremely efficient memory, allowing for low latency access by the thread. The number of registers available are limited, making them a scarce resource for thread blocks. They must be shared between all threads and blocks that reside on a single multiprocessor. As registers are allocated by the CUDA compiler all at one time, the number of threads that may reside on a multiprocessor is reduced as there are a limited number of registers. This leads to a lower occupancy rating due to the simple fact that fewer threads can be allocated to a multiprocessor when lots of registers need to be allocated to a thread block.

In calculating occupancy, the number of registers used by a thread is very important. CUDA devices with compute capability 1.0/1.1 have 8,192 registers per multiprocessor and can also have at most 768 threads resident on a multiprocessor at any one time (NVIDIA, 2012). With these statistics, each thread would have to use at most 10 registers to achieve 100% occupancy.

The exact nature of the relationship between register use and occupancy can be difficult to determine (NVIDIA, 2012). Due to the fact that register allocation differs slightly between different compute versions of CUDA devices, and the fact that a multiprocessor's shared memory is also partitioned between differing thread blocks, exact occupancy calculation is a difficult task. To combat this, NVIDIA provide CUDA developers with a spreadsheet in which critical data about CUDA kernels can be entered to provide an occupancy rating for the code. Additionally, NVIDIA provide a profiling tool that allows the CUDA code to be executed and monitored to calculate an occupancy rating.

2.7 Thread and Block Heuristics

When choosing the number of threads per block, a multiple of 32 threads is recommended by NVIDIA as to ensure "optimal computing efficiency" and facilitate coalescing (NVIDIA, 2012). By ensuring the correct parameters for the number of threads per thread block, the balance between the latency of a CUDA application and resource utilisation can be found.

Occupancy and latency hiding depend on the number of active warps per multiprocessor which in turn depends on the register and shared memory constraints set by the compute capability of the GPU. To balance occupancy with resource allocation, the correct execution parameters should be chosen (NVIDIA, 2012).

Kernels should be designed to try and keep the GPU as active as possible, ensuring that there is as little idle time as possible whilst a kernel is executing. A simple way of doing this is to ensure that the number of blocks specified is greater than the number of multiprocessors on the GPU. Different GPUs have contrasting numbers of multiprocessors however, which is important to keep in mind. In this way, each multiprocessor has at least one thread block to execute. Increasing the number of thread blocks so that each multiprocessor is assigned multiple thread blocks by the compiler is important. In doing this, if a thread block is forced to wait by a *__syncthreads*() command, execution can be switched to another thread block, thus helping to keep the GPU busy at all times. NVIDIA recommend using thousands of thread blocks per kernel launch to ensure scalability with future GPUs (NVIDIA, 2011a).

Clearly, occupancy is not just determined by block size as many blocks may be present on a single multiprocessor at any one time. NVIDIA (2012) give the example that having a block size of 512 threads may result in occupancy of 66% as the maximum number of threads is 768. Therefore, only one active block would reside on a multiprocessor. However, a smaller block of 256 threads could result in 100% occupancy as there would be three active blocks on the multiprocessor.

Selecting the correct block size is important for the reasons described above, but there are several factors in choosing the block size, depending on the task at hand. Currently, experimenting with block sizes is needed to obtain the best performance, but NVIDIA (2012) provide the following rules that should be followed to ensure block and thread heuristics are set correctly.

- Threads per block should be a multiple of warp size to avoid wasting computation, and to facilitate coalescing.
- At least 64 threads per block should be used.
- Between 128 and 256 threads per block is a better choice however. This provides a good base range for initial experimentation of different block sizes.
- If latency is an issue, use smaller thread blocks rather than one large one. This is especially useful if *__syncthreads*() is frequently used.

2.8 Bank Conflicts

As we know from Sections 2.4 and 2.5, shared memory has a much higher bandwidth and lower latency than global memory. This is not the case however where bank conflicts occur. Shared memory is divided into equally sized block (banks) that are accessible simultaneously by threads resident on a single multiprocessor. “Therefore, any memory load or store of n addresses that spans n distinct memory banks can be serviced simultaneously” (NVIDIA, 2012). As a result, thread blocks can achieve a bandwidth that is n times as great as the bandwidth capabilities of a single bank on its own.

If however, there are several memory addresses in a request that map to the same bank of shared memory, a bank conflict occurs and the access to the bank is serialised, impacting the performance of the kernel. In an attempt to reduce the effects of bank conflicts, the GPU will attempt to split each memory request that will result in a bank conflict into as many requests as necessary, so as to avoid conflicts, thus decreasing the bandwidth of the memory access.

2.9 Chapter Summary

In this chapter, a general overview of parallel computing was discussed, as well as an in-depth look at GPGPU and more specifically, CUDA.

We have seen how Bernstein's Conditions can be used to identify whether an algorithm or process can be implemented in parallel by understanding what inherent dependencies are present in the process. In order that a process might be implemented in parallel, it must be flow independent, anti independent, and output independent.

Many new and interesting problems may present themselves in parallel computing. We have seen how race conditions can drastically effect how a program operates, in potentially resulting in incorrect data being read/written. This problem can lead to deadlock, whereby different threads end up waiting for other threads to finish a task, and therefore stall due to neither of them ever finishing.

Flynn's Taxonomy is an important framework for identifying how a parallel computer might be implemented. Flynn (1972) presents three classifications, SIMD, MISD, MIMD for identifying a parallel computer, and how it operates. As well as one classification, SISD, which identifies a traditional sequential computer. These classifications were later used in identifying how a CUDA GPU operates.

On looking at the GPGPU space, we have seen many useful applications of GPGPU, as well as several high profile APIs available for utilising the technology. CUDA was identified as a GPGPU API, as well as the one that will be utilised for this thesis. We have seen in great detail how the CUDA hardware and software models are composed, providing a parallel platform in which many thousands of resident threads may be executed in order to improve the performance of a subject problem. Technicalities of CUDA were presented, such as bank conflicts, and deciding on what memory type(s) should be utilised when implementing applications for CUDA.

Chapter 3

Introduction to the All-Pairs Shortest Path Problem

Graphs are an extremely common data structure in the field of computer science. There are many different problems that can and are represented as graphs, and algorithms to manipulate them are very important and widely used. This section looks at graph algorithms in terms of the APSP problem.

3.1 What is the APSP Problem?

Imagine trying to find the shortest distance between all pairs of cities in an atlas. This problem can be solved using an APSP algorithm by representing the cities and roads between them as vertices and edges respectively. More formally, given a directed, weighted graph $G = (V, E)$, we wish to find for every pair of vertices $u, v \in V$, a least weight (shortest) path from u to v, whose weight is the sum of all edges in the path (Cormen et al., 2001). $|V|$ denotes the number of vertices in the set and $|E|$ similarly denoting the number of edges in G.

The problem can be solved by running an algorithm that solves the Single-Source Shortest Path (SSSP) problem by running it on every vertex in G. A popular way of solving the problem using this means is using Dijkstra's SSSP algorithm. The edges' weights must be non-negative in order to use Dijkstra's algorithm for APSP. If negative edge weights are allowed, the slower Bellman-Ford algorithm must be used (Cormen et al., 2001). Where negative cycles are allowed, there is no shortest path as the traversal of such a cycle continually reduces the cost of the path. An algorithm using this approach is described in Section 4.3. A "true" APSP algorithm does not take this approach however. The following algorithms are expanded on in the following sections.

3.2 Sequential Algorithms

3.2.1 The Floyd-Warshall APSP Algorithm (Floyd, 1962)

The majority of CUDA algorithms described in Chapter 4 are originally based upon this algorithm so it is important to fully understand the theory behind it, and how it is implemented correctly.

The Floyd-Warshall algorithm utilises a dynamic programming technique and runs in $O(n^3)$ time (Cormen et al., 2001). The algorithm operates on a directed graph $G = (V, E)$ with non-negative edge weights. The algorithm can however operate if required with negative edge weights. If a cycle was to exist with total negative weight, the Floyd-Warshall algorithm can be used to detect them. Initially, all path lengths are 0. If negative cycles exist between two vertices, the path length between those two vertices will be negative.

Using the observations in Cormen et al. (2001), for any pair of vertices u and v, observe all paths from u to v where the intermediate vertices are from some subset of V $\{1, 2, ..., k\}$ for any k. Additionally, let p be a path amongst u and v that is of a minimum weight.

Floyd-Warshall uses the relationship between p and all shortest paths between both u and v with all of the intermediate vertices in the set $\{1, 2, ..., k-1\}$. Depending on whether k is an intermediate vertex or not, one of two things can happen. Where k is not an intermediate vertex of p, all of the intermediate vertices in p must be in $\{1, 2, ..., k-1\}$. To that end, the least cost path between u and v will also be in the set $\{1, 2, ..., k\}$.

However, if k is an intermediate vertex of p, p can be split into two paths such that p_1 is that path from u to k and the path p_2 is from k to v. As k is no longer an intermediate vertex, p_1 is the shortest path from u to k where all intermediate vertices are in $\{1, 2, ..., k-1\}$ and p_2 is the shortest path from k to v where all intermediate vertices are in $\{1, 2, ..., k-1\}$. This observation holds in that a subpath of a shortest path is in itself, a shortest path, as described by Cormen et al. (2001).

Algorithm 4 Floyd-Warshall

```
1: for k= 0 to n
2:     for u = 0 to n
3:         for v = 0 to n
4:             graph[u][v] = min(graph[u][v], graph[u][k] + graph[k][v])
5:         end
6:     end
7: end
```

Algorithm 4 demonstrates how simple the Floyd-Warshall algorithm is to implement on the CPU. The computations can be done in place, meaning that the graph does not need to be copied and therefore, keeping the same amount of memory. This basic construct is used and modified by the majority of the following CUDA algorithms, and is a classic solution to the APSP problem.

3.2.2 Dijkstra's Algorithm (Dijkstra, 1959)

As mentioned in Section 3.1, Dijkstra's algorithm, which solves the SSSP problem, can be used to solve APSP by repeating the algorithm on every vertex in a graph. Harish and Narayanan (2007) describe a CUDA algorithm based on Dijkstra's algorithm which is detailed in Section 4.3. A good implementation of Dijkstra's algorithm has a lower running time than that of the Bellman-Ford algorithm described in Section 3.2.3.

Dijkstra's algorithm maintains a set S of vertices whose shortest path weights from the source vertex s have already been determined. The algorithm repeatedly selects a vertex $v \in V - S$ with "the minimum shortest path estimate" and then adds u to S, finally relaxing the edges that leave u. To improve on the Bellman-Ford algorithm, a minimum-priority queue Q is used. The original algorithmic description by Dijkstra does not use a minimum-priority queue (Dijkstra, 1959).

Algorithm 5 Pseudo Code for Dijkstra's Algorithm (Cormen et al., 2001)

```
1: $S \leftarrow \emptyset$
2: $Q \leftarrow V$
3: while $Q \neq \emptyset$
4:     do $u \leftarrow$EXTRACT-MIN($Q$)
5:         $S \leftarrow S \cup \{u\}$
6:         for each vertex $v \in adj[u]$
7:             do RELAX($u, v, w$)
```

We can see from Algorithm 5 that the loop invariant on line three will be true at the start of the algorithm. As the algorithm proceeds through the loop, a vertex u is extracted from Q and immediately added to S. This process maintains the invariant, and from it, we can see that the algorithm loops exactly $|V|$ times. This holds as each vertex is removed from Q and added to S exactly once each. As edges are relaxed on lines four to seven, the path cost estimate is updated if the shortest path can be improved by passing through u to get to v (Cormen et al., 2001).

Dijkstra's algorithm uses a greedy approach in that it chooses the next closest or lightest vertex. Greedy algorithms are not always optimal but in Dijkstra's case, this algorithm does present an optimal solution. A proof for this can be found in the work by Cormen et al. (2001).

Dijkstra's algorithm gives us a running time of $O(|E| + |V| \log |V|)$ when implemented using a minimum-priority queue that is represented by a Fibonacci heap. With this method, each call to EXTRACT-MIN only takes $O(\log |V|)$ and each RELAX call takes just $O(1)$ of which there are $|E|$ calls. The Relax algorithm is explained in Section 3.2.3 and shown in Algorithm 7.

The running time of the algorithm changes when solving the APSP problem. As the algorithm is run $|V|$ times for each vertex, the new running time is $O(|V|^2 \log |V| + |V||E|)$ (Cormen et al., 2001).

3.2.3 The Bellman-Ford Algorithm (Bellman, 1958)

The Bellman-Ford algorithm solves the SSSP problem but like other SSSP algorithms, it can be used to solve APSP as well, by running on each vertex in G. Unlike Dijkstra's algorithm, the Bellman-Ford algorithm may operate on graphs with negative edge weightings and also has an asymptotic running time that is worse than Dijkstra's. For that reason, the Bellman-Ford algorithm is usually used where negative edge weights may be present, as Dijkstra's cannot operate on such a graph.

The algorithm will return a boolean variable that specifies whether the graph contains a negative weight cycle that may be reachable from the source vertex s. If a cycle is found, the algorithm states that there is no solution and ceases execution. If however, there is no cycle, the shortest paths and their weights are calculated.

During the algorithm, all edges in G are relaxed (as shown in Algorithm 7). The Relax algorithm looks at whether the path to v can be improved if the path goes through u. If so, both $d[v]$ and $\pi[v]$ are updated. This continues until the shortest paths have been calculated. Once completed, the algorithm will return *true* if, and only if, no negative weight cycles are detected.

Algorithm 6 Pseudo Code for the Bellman-Ford Algorithm (Cormen et al., 2001)

```
1: INITIALISE SINGLE SOURCE($G, s$)
2: for $i \leftarrow 1$ to $|V[G]| - 1$
3:     do for each edge $(u, v) \in E[G]$
4:         do RELAX($u, v, w$)
5: for each edge($u, v$) $\in E[G]$
6:     do if $d[v] > d[u] + w(u, v)$
7:         then return FALSE
8: return TRUE
```

The Bellman-Ford algorithm runs in $O(|V||E|)$ time. Line one in Algorithm 6 takes $\Theta(|V|)$ time. Following that, each of the edge relaxations in line four take $\Theta(|E|)$ time and the for loop over lines five - seven take $O(|E|)$ time. Thus, the running time results in $O(|V||E|)$.

Algorithm 7 Pseudo Code for the Relax Algorithm (Cormen et al., 2001)

```
1: RELAX($u$, $v$, $w$)
2: if $d[v] > d[u] + w(u, v)$
3:     then $d[v] \leftarrow d[u] + w(u, v)$
4:          $\pi[v] \leftarrow u$
```

3.2.4 Blocked Algorithm (Venkataraman et al., 2003)

The Blocked Algorithm by Venkataraman et al. (2003) was designed, and optimised, with CPU cache in mind, rather than optimising for RAM, which was the standard at the time. In utilising cache, Venkataraman et al. (2003) can achieve a substantial speed-up over the standard Floyd-Warshall algorithm in solving the APSP problem.

The algorithm begins by partitioning the adjacency matrix into sub matrices of size $B \times B$ where B is known as the blocking factor. Usually, it is normal for B to divide wholly into $|V|$. The algorithm performs B iterations of line one in Algorithm 4 on each $B \times B$ block of D before it proceeds to the next B iterations (Venkataraman et al., 2003).

To increase understanding of the algorithm, Venkataraman et al. (2003) suggest thinking of each set of B iterations as being split into three separate phases. In phase 1 of the first iteration, Algorithm 4 is used to compute the elements within the sub matrix located at $(0, 0)$. As this set of iterations only accesses the elements within this block, Venkataraman et al. (2003) state that the sub matrix is called the self-dependent block. In the following code snippets we say that k is $1 \leq k \leq B$.

In phase 2, a modified version of Algorithm 4 is used to compute the remaining sub matrices that are on the same row and column as the self-dependent block. For the sub matrices on the same row, the computation

$$D^k(i,j) = min\{D^{k-1}(i,j), D^B(i,k) + D^{k-1}(k,j)\}$$

is used. Likewise, for the remaining sub matrices on the same column as the self-dependent block, the computation

$$D^k(i,j) = min\{D^{k-1}(i,j), D^{k-1}(i,k) + D^B(k,j)\}$$

is used.

Finally, in phase 3, the remaining sub matrices are computed. Like phase 2, this computation is completed using a modified version of Algorithm 4:

$$D^k(i,j) = min\{D^{k-1}(i,j), D^B(i,k) + D^B(k,j)\}$$

Now that phase 3 is complete, the next round of B iterations is computed by the Algorithm 8. This time however, the self-dependent block is located at $(1, 1)$. The process then repeats until B iterations of the algorithm have been performed. We are left with an algorithm that solves the APSP problem with the intention of improving running times when compared to the standard Floyd-Warshall algorithm (Algorithm 4).

Algorithm 8 Code for the Blocked Floyd-Warshall Algorithm (Venkataraman et al., 2003)

```
for(round = 1; round$\leq$ n / B; round ++)
    for(k = (round - 1) * B + 1; k != round * B; k ++)
        for(all i and j in block)
            D[i][j] = min(D[i][j], D[i][k] + D[k][j]);
        end
    end

    do the remaining for all remaining blocks:
        phase 2 blocks
        phase 3 blocks
        for(k = (round - 1) * B + 1; k $\leq$ round * B; k ++)
            for(all i and j in block)
                D[i][j] = min(D[i][j], D[i][k] + D[k][j]);
            end
        end
end
```

Proving the Blocked Algorithms Correctness

We begin with a bit of notation. Suppose that P_1, P_2 and P_3 are sets of paths in G. Suppose that some path p is the concatenation of a path p_2 in P_2 with a path p_3 in P_3; so, in particular, the last vertex of p_2 must be equal to the first vertex of p_3. We say that p_2 is a *prefix* of p, with p_3 the *complementary suffix*. Define $P_2 \cdot P_3$ as the set of paths in G having a prefix in P_2 so that the complementary suffix is in P_3 (note that we disallow walks, obtained in this way, that are not paths).

The following definitions are needed for this section.

1. We denote by $P^c(u,v)$ the set of all paths from vertex u to vertex v in G where the internal vertices come from $\{1,2,\ldots,c\}$.

2. For any set of paths P, we denote by $\min P$ the minimum weight of the paths of P or ∞ if P is empty.

3. We define the $n \times n$ matrix A^c so that $A^c(u,v) = \min P^c(u,v)$; that is, $A^c(u,v)$ is the value $l^c_{u,v}$ (see the previous section), with $A^c(u,v) = \min\{A^{c-1}(u,v), A^{c-1}(u,c) + A^{c-1}(c,v)\}$.

Define
$$\varphi(P_1,P_2,P_3) = P_1 \cup (P_2 \cdot P_3).$$
Note that if $P_1 \subseteq P_1'$, $P_2 \subseteq P_2'$ and $P_3 \subseteq P_3'$ then $\varphi(P_1,P_2,P_3) \subseteq \varphi(P_1',P_2',P_3')$ (we shall return to this fact later). Also, note that
$$P^c(u,v) = \varphi(P^{c-1}(u,v), P^{c-1}(u,c), P^c(c,v)),$$
or
$$P^c_{x,y}(i,j) = \varphi(P^{c-1}_{x,y}(i,j), P^{c-1}_{x,z}(i,k), P^{c-1}_{z,y}(k,j))$$
where $u = xb+i$, $v = yb+j$ and $c = zb+k$.

Suppose that we amend the recurrence within Floyd-Warshall slightly so that if $u = xb+i$, $v = yb+j$, $c = zb+k$ and $\bar{c} = (z+1)b$ (and so $c \leq \bar{c}$) then we define the following sets of paths:

- $\tilde{P}^0_{x,y}(i,j) = P^0_{x,y}(i,j)$
- $\tilde{P}^c_{z,z}(i,j) = P^c_{z,z}(i,j)$
- $\tilde{P}^c_{x,z}(i,j) = \varphi(\tilde{P}^{c-1}_{x,z}(i,j), \tilde{P}^{c-1}_{x,z}(i,k), P^{\bar{c}}_{z,z}(k,j)), \text{if } x \neq z$
- $\tilde{P}^c_{z,y}(i,j) = \varphi(\tilde{P}^{c-1}_{z,y}(i,j), P^{\bar{c}}_{z,z}(i,k), \tilde{P}^{c-1}_{z,y}(k,j)), \text{if } y \neq z$
- $\tilde{P}^c_{x,y}(i,j) = \varphi(\tilde{P}^{c-1}_{x,y}(i,j), P^{\bar{c}}_{x,z}(i,k), P^{\bar{c}}_{z,y}(k,j)), \text{if } x \neq z \neq y$

with $\tilde{A}^c_{x,y}(i,j) = \min \tilde{P}^c_{x,y}(i,j)$.

We now detail two inductions. Our first induction hypothesis is that no matter what the values of u and v, we have that the set of paths $\tilde{P}^{c-1}_{x,y}(i,j)$ from u to v in G is such that all internal vertices on any path come from the set of vertices $\{1,2,\ldots,\bar{c}\}$. The base case of the induction (when $c = 0$) holds by definition, and the construction above immediately yields that we must have that the set of paths $\tilde{P}^c_{x,y}(i,j)$ from u to v in G must be such that all internal vertices on any path come from the set of vertices $\{1, 2, \ldots, \bar{c}\}$. Thus, in particular, $\tilde{P}^{c}_{x,y}(i,j) \subseteq P^{\bar{c}}_{x,y}(i,j)$.

Suppose as our induction hypothesis that no matter what the values of u and v, we have a set of paths $\tilde{P}^{c-1}_{x,y}(i,j)$ from u to v in G such that all internal vertices on any path come from the set of vertices $\{1, 2, \ldots, \bar{c}\}$ and such that $P^{c-1}_{x,y}(i,j) \subseteq \tilde{P}^{c-1}_{x,y}(i,j)$. From the definitions above, we trivially have that every path in $\tilde{P}^{c}_{x,y}(i,j)$ only has internal vertices from the vertex set $\{1, 2, \ldots, \bar{c}\}$. Furthermore, we have the following:

- if $x \neq z$ then

$$\begin{aligned} P^{c}_{x,z}(i,j) &= \varphi(P^{c-1}_{x,z}(i,j), P^{c-1}_{x,z}(i,k), P^{c-1}_{z,z}(k,j)) \\ &\subseteq \varphi(\tilde{P}^{c-1}_{x,z}(i,j), \tilde{P}^{c-1}_{x,z}(i,k), P^{\bar{c}}_{z,z}(k,j)) \\ &= \tilde{P}^{c}_{x,z}(i,j) \end{aligned}$$

- if $y \neq z$ then

$$\begin{aligned} P^{c}_{z,y}(i,j) &= \varphi(P^{c-1}_{z,y}(i,j), P^{c-1}_{z,z}(i,k), P^{c-1}_{z,y}(k,j)) \\ &\subseteq \varphi(\tilde{P}^{c-1}_{z,y}(i,j), P^{\bar{c}}_{z,z}(i,k), P^{c-1}_{z,y}(k,j)) \\ &= \tilde{P}^{c}_{z,y}(i,j) \end{aligned}$$

- if $x \neq z \neq y$ then

$$\begin{aligned} P^{c}_{x,y}(i,j) &= \varphi(P^{c-1}_{x,y}(i,j), P^{c-1}_{x,z}(i,k), P^{c-1}_{z,y}(k,j)) \\ &\subseteq \varphi(\tilde{P}^{c-1}_{x,y}(i,j), P^{\bar{c}}_{x,z}(i,k), P^{\bar{c}}_{z,y}(k,j)) \\ &= \tilde{P}^{c}_{x,y}(i,j). \end{aligned}$$

So, by induction, we have that $P^{c}_{x,y}(i,j) \subseteq \tilde{P}^{c}_{x,y}(i,j)$ no matter what the values of u and v and for all values c from $\{0, 1, \ldots, n\}$.

However, when $c = zb$ we also have that $\tilde{P}^{c}_{x,y}(i,j) \subseteq P^{c}_{x,y}(i,j)$, and so for such c, $\tilde{P}^{c}_{x,y}(i,j) = P^{c}_{x,y}(i,j)$ with $\tilde{A}^{n}_{x,y}(i,j) = A^{n}(u,v)$. Consequently, we can use the computations of $\tilde{A}_{x,y}$ in order to compute A^{n}.

3.3 Chapter Summary

In this chapter, we have investigated the formulation of the APSP problem as well as several sequential solutions to the problem.

In identifying the APSP problem, we have seen that it is a way of finding the shortest path between every pair of vertices in a given weighted graph G. This can be solved by utilising specific APSP algorithms, or by performing algorithms that solve the SSSP problem on every vertex in G.

Several APSP algorithms are presented which will later be discussed in relation to CUDA algorithms. These CUDA algorithms are originally based the sequential algorithms presented in this chapter. We have seen a very popular APSP algorithm presented by Floyd (1962) which solves the problem in place in $O(V^3)$. Additionally, we have seen the classic SSSP algorithm presented by Dijkstra (1959) and how it can be utilised for the APSP problem. The Bellman-Ford algorithm is also an SSSP algorithm that can solve the APSP problem by executing it on every vertex in G. However, we have seen that the Bellman-Ford algorithm has the added benefit of working on graphs with negative edge weights, whereas Dijkstra's algorithm does not have this capability. We have seen that Dijkstra's algorithm has a better running time than Bellman-Ford and so should always be used unless there are negative edge weights.

Finally, a blocked CPU algorithm based on Floyd-Warshall's APSP algorithm is observed, which provides an improved running time over the Floyd-Warshall algorithm. A proof of correctness is provided for this algorithm, as it is based on the well documented Floyd-Warshall algorithm, but forms the basis of a critical CUDA algorithm.

An understanding of these algorithms is critical in presenting CUDA implementations of APSP algorithms, as many of them are based on these classic CPU algorithms.

Chapter 4

Implementation of APSP Algorithms Using CUDA

4.1 Graph Representations with CUDA

GPU memory layout is optimised for rendering graphics and cannot support user-defined data structures efficiently (Harish et al., 2009). Whilst data structures for use on the CPU have been studied extensively, the use of hash tables (Hyvonen et al., 2008) for example, that are efficient on the CPU, are not suitable for the GPU (Harish et al., 2009).

Of the algorithms detailed in this chapter, two different graph representations are used. These implementations are described here, to provide a full understanding that may be used when describing the APSP algorithms. The way in which a graph to be searched is stored on the GPU is critically important. Trade-offs between time and memory constraints have to be considered when choosing the memory layout for a graph. Let $G = (V, E)$ be a graph with the vertex set V and the edge set E. We will use this notation for the rest of the paper to denote the graphs that we will perform searches on for all APSP algorithms. In that way, we can maintain a coherent approach to the way in which we describe all algorithms.

4.1.1 Adjacency Lists

Traditional methods of storing graphs, such as using an adjacency matrix, provide a constant time method, $O(1)$, of determining whether there is an edge e between two vertices, u and v, but compromises on memory usage, $O(|V|^2)$. For sparse graphs, adjacency matrices are largely wasteful, holding data where no edges exist in the graph.

Adjacency lists provide a method of storing graphs which requires vastly less memory than an adjacency matrix, $O(|V|+|E|)$. An adjacency list achieves this low memory usage by only holding vital information. However, the drawback of using such a list is a more expensive lookup time to determine if there is an edge between two vertices $O(|V|)$. Harish and Narayanan (2007) and Harish et al. (2009) describe the use of a compacted adjacency list represented in Figure 4.1. They argue that due to the variable nature of graphs (number of vertices and edges per graph), using an adjacency list representation may not be completely efficient on the GPU. Therefore, they use a modified version of an adjacency list, known as a compacted adjacency list where all of the information that would usually be stored in several lists, is compacted into a single, one dimensional array.

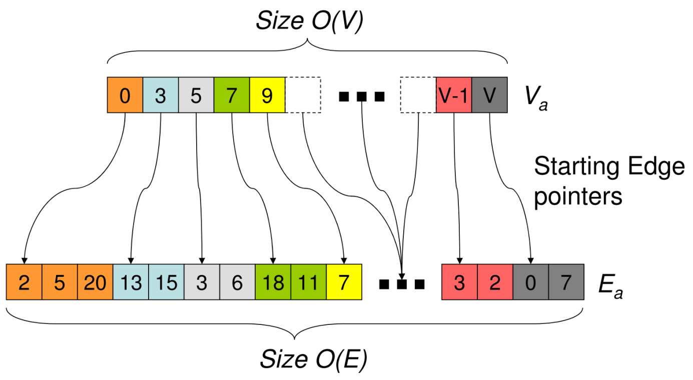

Figure 4.1: Graph Representation as a Compacted Adjacency List (Harish et al., 2009)

Demonstrating a compacted adjacency list, Figure 4.1 shows the vertex list (V_a) and the edge list (E_a). The vertex list points to a starting index in E_a which then represents the vertices incident to it. This method of storing an input graph is suitable for both undirected and directed graphs and can be expanded with a further array of equal size to $|E_a|$ to represent edge weights (W_a). Using several one dimensional arrays is suitable for use in CUDA as several blocks of contiguous memory can easily be allocated onto the GPU. When using a two dimensional array with CUDA, the compiler simply allocates a contiguous block of memory anyway, and automatically converts the two dimensional array into a one dimensional array, adding a hidden cost to running times.

4.1.2 Adjacency Matrices

Katz and Kider (2008) and Buluc et al. (2010) look at an alternative way of storing their graphs on the GPU, opting to use the standard adjacency matrix representation. Despite the downsides to using an adjacency matrix as described in section 4.1.1, this is an appropriate choice due to the way the computations are calculated in the relevant algorithms. Their implementations make use of shared memory which maps extremely well to the nature of an adjacency list.

Katz and Kider (2008) manage to store graphs with just over 11,000 vertices, occupying a considerable 1.015Gigabyte (10^9 bytes)s (GBs) of data. Using the compacted adjacency list, graphs of 10million vertices with a degree of six are possible on a GPU with just 768Megabyte (10^6 bytes)s (MBs) of memory (Harish and Narayanan, 2007).

4.2 Mapping Threads to Vertices

It is often necessary to map each thread t to a vertex v such that each t has one unique v corresponding to it. When dealing with thousands of blocks, threads and vertices, it becomes convenient to have a common function to calculate this for us. To that end, we created a function that takes the current information about t, including its position in a thread block, as well as block and grid dimensions, to return an integer value that will assign t to v. As described in Section 2.5, CUDA provides multidimensional thread organisation as well as the necessary naming constructs to identify thread and block co-ordinates. Using $threadID.x$ for example, will return that threads current x coordinate. The same principle applies to thread blocks and grids, as well as y and z co-ordinates. The function listed in Algorithm 9 is used throughout this paper and will be referred to again several times as "*obtainThreadID*".

Algorithm 9 This Function is used by Algorithms that use the Graph Structure Described in Section 4.1.2

```
1: vertex = 0
2: vertex = vertex + threadIDX + (blockIDX * blockDimension.x)
3: vertex = vertex + (threadIDY + (blockIDY * blockDimension.y))
4: vertex = vertex + blockDimension.x * gridDimension.x
5: return vertex
```

4.3 Harish and Narayanan's Algorithm (Harish and Narayanan, 2007)

This particular algorithm is the only CUDA algorithm implemented in this paper that it is actually comprised of the solution to the SSSP problem and simply repeated for each vertex in the graph, forming a complete solution to the APSP problem.

Using the compacted adjacency list graph representation described in section 4.1.1, additional arrays are used to hold the extra data required by the algorithm. A boolean mask array, M_a, of size $|V|$ is constructed, as well as a weight array, W_a, of size $|E|$. Additionally, a cost array and updating cost array are created. C_a and U_a respectively, each of size $|V|$. Each array holds standard 32-bit integers, excluding M_a, which as previously described is a boolean array.

The kernels in this algorithm utilise a one dimensional thread block structure (Algorithms 10 and 11). Each thread block consists of 32 threads, with as many thread blocks as necessary to accommodate $|V|$. Once the number of blocks has been allocated, $|V|$ may be padded to match the number of threads so as to avoid memory access violations, for all edge weights w, $0 < w \leq 1000$.

The algorithm begins by setting the source vertex's (s) entry in M_a to true, where all other entries are false. During the first kernels execution, each vertex u looks at M_a to see if it has a corresponding entry as true. If this is the case, u will read its current cost from C_a. This cost is the current cost of getting from s to u at the current stage of the algorithm. Additionally, u fetches its neighbours, v, weight from W_a. If the cost of travelling from u to v is less than travelling from s to u, its cost is updated in U_a. Finally, v sets its entry in M_a to false so that it is not examined again.

The second kernel takes the values in U_a, and updates C_a with any changes detected. Each vertex v checks if its entry in C_a is greater than in U_a. If so, the cost in C_a is updated. Crucially, M_a is also updated to true, so that in the next iteration, v may be examined.

These two kernels are repeated until all entries in M_a are false, at which point, SSSP has been solved and the algorithm can stop. However, to solve the APSP problem, the entire process is simply repeated $|V|$ times, with a different source vertex for each iteration. In that way, a solution to the APSP problem is solved, using an algorithm that would otherwise solve SSSP.

Algorithm 10 APSP Kernel 1 $(V_a, E_a, W_a, C_a, U_a, M_a)$

```
1: threadID =obtainThreadID
2: if $M_a$[threadID] == true then
3:     $M_a$[threadID] = false
4:     for all neighbours nThreadID of threadID do
5:         if $U_a$[nThreadID] > $C_a$[threadID] + $W_a$[nThreadID] then
6:             $U_a$[nThreadID] = $C_a$[threadID] + $W_a$[nThreadID]
7:         end
8:     end
9: end
```

Algorithm 11 APSP Kernel 2 ($V_a, E_a, W_a, C_a, U_a, M_a$)

```
1: threadID = obtainThreadID
2: if $C_a$[threadID] $>$ $U_a$[threadID] then
3:     $C_a$[tid] = $U_a$[tid]
4:     $M_a$[tid] = true
5: end
6: $M_a$[threadID] = false
```

After each iteration, the algorithm must copy C_a back to host memory, so that the result can be saved, and the path costs kept. If this were not completed, due to the nature of the algorithm, the results would be overwritten by the next iteration of the algorithm, thus resulting in only the final iterations results being saved.

4.3.1 Modifications

When Harish and Narayanan (2007) designed the algorithm, CUDA had not developed as it has today, and was vulnerable to race conditions. This resulted in the need for U_a as well as the second kernel. It was not safe to simply update C_a during the first kernel, as there was no way to synchronise between multiprocessors with CUDA 1.0 (Harish and Narayanan, 2007). As of CUDA 1.1 however, atomic read and write operations were added to the architecture, meaning that the algorithm could be modified to eliminate the need for both U_a and the second kernel. As well as improving the running time of the algorithm, this simple modification also allowed for slightly larger graphs as, U_a would no longer be present.

Using some newer CUDA functions such as copying data whilst a kernel is executing is not possible. For example, if we wished to eliminate the delay of copying C_a by transferring the data during kernel execution, only incomplete data would be returned as the kernel must finish to obtain the complete result set. However, to try and reduce the effects of this data transfer with each iteration, C_a is allocated using pinned memory, allowing CUDA to have direct access to the relevant memory areas for faster data transfer between host and device by providing bandwidth of at least 5GBps (NVIDIA, 2011a).

4.4 Quoc-Nam Tran's Floyd-Warshall Algorithm (Tran, 2010)

Tran (2010) describe an algorithm to simply modify the Floyd-Warshall algorithm so that it becomes parallelised on the GPU. Utilising the fact that all threads can communicate via device memory, the algorithm essentially computes the k^{th} round of the Floyd-Warshall algorithm in one parallel step. Shared memory is utilised by the algorithm in that each thread maps to its own vertex, so inter-block communication can be used to compute Floyd-Warshall on small chunks of the graph at one time.

The algorithm uses a duplicate copy of the graph, that is identical to the original graph when the algorithm starts. To avoid global memory communication, the use of this identical graph P and the original graph G are interchanged. The algorithm performs a series of iterations, where P initially holds the results for the first iteration, its use is swapped so that the algorithm thinks P is now G for the second iteration. This swapping occurs for every iteration until the complete result is formulated. Utilising two data structures such as this ensures that read-after-write inconsistencies are eliminated.

A thread block size of 16 x 16 is utilised for this algorithm. The intention is that threads calculating the results will fit into a sub-block of the graph that is dynamically assigned to the thread block. The kernel for this algorithm is executed $|V|$ times, with the use of G and P alternating with each execution.

As described in Section 2.5, CUDA threads within a block are executed in a group of 32, known as a warp. As each half-warp share the row

$$G[k, j], j = blockIdx.x * 16, blockIdx.x * 16 + 1, ..., blockIdx.x * 16 + 15$$

and one element $G[i, k]$ where

$$i = blockIdx.y * 16, blockIdx.y * 16 + 1, ..., blockIdx.y * 16 + 15$$

the row $G[k, j]$ and the column $G[i, k]$ are copied into shared memory for every thread block.

Threads whose corresponding row address, *blockIdx.y*, is 0, copy the row data, and threads with 0 for their column address, *blockIdx.x*, copy the column data. The data for both row and column is copied from G. Once this is complete, the standard Floyd-Warshall computation is performed on the data that has been copied into shared memory. The result is then written back to P as shown in line nine of Algorithm 13.

Algorithm 12 CUDAFloyd-Warshall (G, P)

```
1: Allocate and copy $G, P$ to device
2: for $i = 0$ to $|V|$
3:     if $i\%2 \neq 0$ then
4:         CUDAFWKernel$(G, P)$
6:     else
7:         CUDAFWKernel$(P, G)$
```

Algorithm 13 CUDAFWKernel (G, P)

```
1: threadID = obtainThreadID
2: row[16] **Shared
3: column[16] **Shared
4: if threadIdx.y = 0 then
6:     row[threadIdx.x] = G[threadID]
7: if $threadIdx.x = 0$ then
8:     column[threadIdx.y] = G[threadID]
9: $P$[threadID] = min(G[threadID], row[threadIdx.x] + column[threadIdx.y])
```

Tran (2010) state that the running time of this algorithm is $O(n^3/|P|)$ where $|P|$ is the number of cores on the GPU. Clearly, this provides a slight asymptotic speed-up over the standard CPU version of Floyd-Warshall that runs in $O(n^3)$ time.

The algorithm has been optimised to ensure that read and write operations from global memory are coalesced, so as to access as much data as possible in one processor cycle, thus reducing the latency on memory accesses. Additionally, Tran (2010) have stopped bank conflicts from occurring in shared memory, ensuring that the efficiency of the memory is as great as possible. As with the algorithm described in Section 4.7, this algorithm (Algorithm 12) can be expanded to work across multiple GPUs, or simply with larger graphs than the GPU can store at one time. However, Tran (2010) state that this has a significantly negative effect on this algorithm.

4.5 Quoc-Nam Tran's CUDA APSP Algorithm (Tran, 2010)

Tran (2010) describes a second algorithm that claims to provide an extremely impressive speed-up of 2,500x over its sequential, single-core implementation. However, the sequential algorithm runs in $O(n^3 \log n)$ which is worse than Floyd-Warshall's $O(n^3)$ running time. Similar in construct to the algorithm described in Section 4.4, this algorithm (Algorithm 14) differs slightly in the CUDA kernel, as well as the CPU loop that calls the kernel.

As we know from our description of Floyd-Warshall in Section 3.2.1, the shortest path with only two edges between any vertices u and v will go through exactly one other vertex. We also know that the adjacency matrix A^1 of shortest paths that have a length of at most two is constructed as:

$$A^1[u,v] = min(A^0, A^0[u,k] + A^0[k,v]), whereA^0 = A$$

Following from this, A^2 could be constructed from A_0 and A_1 by agreeing that the shortest path with three or less edges will be the shortest path with two or less edges followed by the shortest path of length at most one (Tran, 2010). Due to the fact that a path with N edges where $N \geq |V|$ must have a cycle, A^{N-1} is the adjacency matrix which must hold the shortest path throughout the entire graph. This algorithm by Tran (2010) constructs adjacency matrices $A^1, A^3, A^4, ..., A^N$ for the formulation of the APSP problem.

As each row in A^k is related to every column in A^k when calculating A^{k+1}, A^k is partitioned into tiles that conveniently relate to the architecture of thread blocks as in Section 4.4. These tiles are referred to as "A's" and "B's" for rows and columns respectively. Each thread block in the kernel copies one tile from the current row of A^k, and one column tile into shared memory. The data is processed here and the result is simply written back to global memory. Tran (2010) state that the algorithm is very similar to matrix multiplication, except that the minimum value is taken rather than the summation. A fact that makes implementation easier to complete.

As with the algorithm in Section 4.4, this algorithm uses two copies of the graph, interchanging their use for storing results, and calculating necessary data. The algorithms kernel is called $\lceil \log_2(n-1) \rceil$ times.

Algorithm 14 CUDAAPSP (G, P)

```
1: Allocate and copy $G, P$ to device
2: **for** $i = 0$ **to** $\lceil \log_2(n-1) \rceil$
3:     **if** $i\%2 \neq 0$ **then**
4:         CUDAAPSPKernel$(G, P, i)$
6:     **else**
7:         CUDAAPSPKernel$(P, G, i)$
```

Algorithm 15 APSPKernel (G, P, i)

```
1: threadID = obtainThreadID
2: minimumValue = $\infty$
3: A[16][16] **Shared
4: B[16][16] **Shared
5: for each tile A and B in $G$
6:     minimumValue = min(A[threadIdx.y, k + B[k, threadIdx.x], minimumValue)
7: $P$[threadID] = minimumValue
```

4.6 Katz and Kider's Algorithm (Katz and Kider, 2008)

The algorithm described by Katz and Kider (2008) utilises CUDAs shared memory constructs to a greater extent than those algorithms described by Harish and Narayanan (2007), as well as the divide and conquer approach that maps well to parallel programming. The algorithm is based upon a sequential algorithm that was designed to make better use of the CPU cache (Venkataraman et al., 2003). It is easy to see how this algorithm maps to CUDA by utilising shared memory instead of CPU cache, and its inherent divide and conquer nature. The algorithm described by Venkataraman et al. (2003) is itself based upon the Floyd-Warshall algorithm, and aims to improve its performance by splitting the graph into several blocks, and computing the Floyd-Warshall on smaller sets of data.

To begin, the algorithm partitions the graph into blocks of equal size. For this implementation, each block is 16 x 16 vertices in size. In this way, a graph block can easily be mapped to a CUDA thread block. The algorithm proceeds in rounds, with each round consisting of 3 phases. Each of the 3 phases are an individual CUDA kernel.

Each round gives the algorithm a new *primary block*. This primary block is set along the diagonal axis of the graph, with the first primary block starting at location $(0, 0)$ and the final block at $(|V| - 16, |V| - 16)$. From this, we can see that the algorithm consists of $|V|/16$ rounds.

Phase 1 is the simplest of the phases, and simply computes the standard Floyd-Warshall algorithm for the current primary block. Only one multiprocessor is active for this phase, as only one block (and therefore one CUDA block) is being computed. The computation takes place in shared memory, so as to speed-up the phase as much as possible. As only one thread block is currently active, it is a trivial task for each thread to read its data into shared memory using *obtainThreadID*. Each thread then computes its value, and saves it back to global memory.

In Phase 2, blocks whose values are dependant on the primary block are computed. In other words, all blocks on the same row and column as the primary block are computed. To accomplish this, each CUDA block loads its relevant graph block into shared memory, as well as the primary block; therefore creating two arrays in shared memory. Once again, *obtainThreadID* is used to ensure that each thread loads the correct graph data into shared memory. The result is stored in the current block, and not the primary block. Once the computation is complete, the result is written back to the graph in global memory.

Phase 2 employs a CUDA grid and block layout that helps to ease the data processing (Katz and Kider, 2008). For a graph with n blocks per axis, $2 * (n-1)$ blocks are processed. In this case, the blocks are organised into a grid with the dimensions $(n - 1, 2)$. The first row in this grid processes the data that occupies the same row as the primary block. Likewise, the second row of the grid, handles the data on the same column as the primary block. In this way, the kernel reads the current CUDA blocks y co-ordinate. As the value will be a 0 or 1, it is easy for the block to determine whether it should be handling a row or a column. Ensuring that each CUDA block skips the primary block is extremely important. Katz and Kider (2008) give the equation shown in Figure 4.2 to ensure that this is accomplished successfully. If the equation returns 0, the current block is to the left of the primary block (using x co-ordinates). However, if 1 is returned, the current block is either in the same location as the primary block, or to the right of it. In this case, the 1 is added to the addressing system, so that the CUDA block can skip over the primary block.

$$skipPrimaryBlock(x) = min\left(\frac{blockIDx.x + 1}{primaryBlockID + 1}, 1\right)$$

Figure 4.2: Equation to Skip Over x Thread Blocks

Finally, phase 3 computes the values for all remaining blocks that haven't yet been computed. In this phase, the primary block is not needed. Instead, values from the same row and column as the current block that were computed in phase 2 are loaded into shared memory, along with the current block. Therefore, we have three arrays in shared memory. Figure 4.4 visualises this memory access.

This final phase has a much larger grid and block layout than previous phases. In this case, the CUDA blocks are organised into a grid with dimensions $(n-1, n-1)$. As with the equation in Figure 4.2, phase 3 must also skip over the primary block. For this phase, two equations are needed, rather than just the one in phase 2. The original equation in Figure 4.2 is used for the x direction, however a new one is needed to skip over in the y direction. The equation is listed in Figure 4.3.

$$skipPrimaryBlock(y) = min\left(\frac{blockIDx.y + 1}{primaryBlockID + 1}, 1\right)$$

Figure 4.3: Equation to Skip Over y Thread Blocks

Once phase 3 has finished, the primary block is relocated diagonally down the graph. This begins the next round. Once all rounds have completed, the APSP problem has been solved. The nature of the algorithm shows that, whilst it is hard to reduce the Floyd-Warshall algorithm into smaller parts due to its large data dependency, it is possible to do so in ways such as these to improve the performance of the APSP problem and as such, reducing the $O(n^3)$ running time of the standard Floyd-Warshall algorithm.

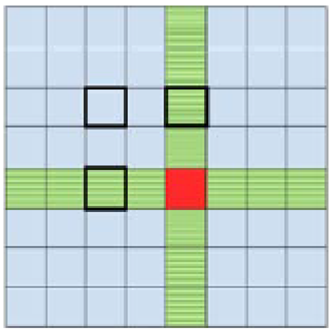

Figure 4.4: Overview of Katz's Data Access During Phase 3 (Katz and Kider, 2008)

In each phase, every block computes the Floyd-Warshall algorithm. Inside the loop that computes these values, each thread within the block must be synchronised to ensure that all the data is up to date for the next loop of the algorithm. Without the synchronisation, incorrect values may be used for the computation, resulting in an invalid result. Figure 4.5 helps to provide a higher level view of the execution of all 3 phases where primary block is in location (0, 0).

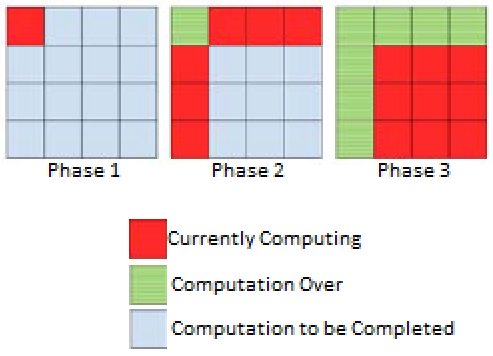

Figure 4.5: Overview of Katz's Algorithm Executing where 0,0 is the Primary Block (Katz and Kider, 2008)

Algorithm 16 Phase1 Kernel $(G, |V|, k)$

```
1: threadId = obtainThreadID
2: primaryBlock[16][16] **Shared
3: Compute the Primary Block
4: G[threadId] = primaryBlock[threadIdx.x][threadIdx.y]
```

Algorithm 17 Phase2 Kernel $(G, |V|, k)$

```
1: threadId =obtainThreadID
2: primary[16][16] **Shared
3: current[16][16] **Shared
4: primary = retrievePrimaryBlock
5: if $blockIdx.y == 0$ then
6:     for $i = 0$ to 16
7:         current[threadIdx.x][threadIdx.y] = min(current[threadIdx.x][threadIdx.y],
           current[threadIdx.x][j] + primary[i][threadIdx.y])
8:     else
9:         for $i = 0$ to 16
10:            current[threadIdx.x][threadIdx.y] = min(current[threadIdx.x][threadIdx.y],
               primary[threadIdx.x][j] + current[i][threadIdx.y])
11: G[threadId] = primary[threadIdx.x][threadIdx.y]
```

Algorithm 18 Phase3 Kernel $(G, |V|, k)$

```
1: threadId = obtainThreadID
2: primary[16][16] **Shared
3: row[16][16] **Shared
4: column[16][16] **Shared
5: primary = retrievePrimaryBlock
6: for $i = 0$ to 16
7:     currentBlock[threadIdx.x][threadIdx.y] = min(currentBlock[threadIdx.x][threadIdx.y],
       rowBlock[threadIdx.x][j] + columnBlock[i][threadIdx.y])
8: G[threadId] = primary[threadIdx.x][threadIdx.y]
```

4.6.1 Limitations

Whilst this algorithm is very efficient and is shown to provide some very good results, it is limited by the fact that the entire graph must be stored in global memory. As such, the size of the graph is limited to whatever global memory the GPU has. In the case of a GTX 470, this is 1.25GB. Additionally, the way in which the graph is partitioned is limited in that it can only be $\frac{1}{3}$ of the total shared memory size which is just 16kB for old CUDA devices, and 32kB for newer devices. This is due to the fact that phase 3 requires three equally sized blocks to be stored in shared memory simultaneously.

In Section 4.7, we describe a significant modification to this algorithm that allows graphs of any size, rather than being limited by the amount of global memory on the GPU.

4.7 Modified Katz and Kider's Algorithm

As mentioned in Section 4.6.1, the entire graph must be able to fit onto the GPU global memory. This is quite a restriction, as many graphs may be vastly larger than the amount of memory available on the GPU. To that end, we have created a modification that allows larger graphs to work with the algorithm.

In principle, it is the same algorithm, with some added steps. To accommodate larger graphs, each round and phase is split into stages. At each stage, the large graph is partitioned so that as much of the graph can occupy as much memory on the GPU as possible. For example, if we have a graph that occupies 2GB of memory, but only 1GB of memory is available on the GPU, the graph will be partitioned into so that it occupies $\frac{1}{3}$ of total global memory. In this case, that would be approximately 333MB. Unfortunately, the graph cannot be partitioned into a larger size (1GB), as the kernels require up to three different sets of data (primary block, row block etc.).

During each phase of the algorithm, there are several stages that result in large portions of the graph are copied to the GPU, whereby the algorithm then proceeds as normal. The result is then copied back to host memory and the process repeats until the entire graph has been used for that phase. This process repeats until all three phases have run across the entire graph, and for as many rounds as are needed. Undoubtedly, this version of the algorithm will be slower than the one detailed in Section 4.6 due to the added operations of copying data to and from the GPU, but it facilitates the operation on graphs of any size which is a noticeable and worthy benefit. Katz and Kider (2008) also state that a method similar to this can be used to run the algorithm across multiple GPUs in parallel.

Algorithm 19 Modified Algorithms Host Code

```
1: for all rounds do
2:     IDENTIFY PARTITION SIZE
3:     IDENTIFY PARTITION LOCATION
4:     for all partitions in $G$ do
5:         phase1Kernel($partition, |V|, k$)
6:     end
7:     for all partitions in $G$ do
8:         phase2Kernel($partition, |V|, k$)
9:     end
10:    for all partitions in $G$ do
11:        phase3Kernel($partition, |V|, k$)
12:    end
13: end
```

4.8 Chapter Summary

In this chapter, we have observed the APSP problem in more detail, specifically relating to how the APSP problem can be solved with CUDA, looking at CUDA implementations, as well as the necessary data structures and considerations taken into account when implementing the CUDA algorithms.

There were no major problems observed when implementing APSP solutions. The most major obstacle was in ensuring our code was as close to the original implementation as possible. Problems occurred due to the fact that descriptions in some journals could be a little vague as to the technical details in how the original algorithms were implemented.

We have seen four CUDA implementations of different APSP algorithms, as well as two of our own modifications on those algorithms in an effort to improve them, using new and contemporary techniques. The solutions described highlight the different needs of each algorithm on memory utilisation and data structures.

Chapter 5

Results and Evaluation

In this Section, we will look at the results of running both CUDA and CPU algorithms which will allow us to compare the claims made by each algorithm's author(s), as well as providing a common test suite for them. The results stem from a number of graph searches that were conducted across a maximum of 21 graphs.

5.1 Evaluation Method

To evaluate each algorithm, a number of tests were conducted in which the running time of each algorithm was recorded in order to compare and contrast between each CUDA algorithm, as well as the CPU algorithms. 21 graphs were searched, each with an increasing number of vertices but all with a similar average degree per vertex. To create the graphs, a graph creator was implemented. This graph creator accepted as arguments: number of vertices required, minimum and maxiumum degree per vertex, and minimum and maximum weight per edge. In this way, we could experiment with graph sizes and density very easily. Average edge weights per graph always varied between 2 and 6 edges.

Of course, whilst running the algorithms, both GPU and CPU are performing other tasks such as handling the operating system, and displaying output on a monitor. Several MBs of the GPU memory are used by the computer in order to create an output on the monitor and as such, less memory is available for graph searching. 21 graphs were searched with differing numbers of vertices $\{1024, 2048, 3096, ..., 21504\}$. The GPU used for the experiments was equipped with 1.25GBs of global memory, meaning that a graph with at most $18,000$ vertices could be present on the GPU when using the adjacency matrix method of storage described in Section 4.1.2. Modifications to certain algorithms such as that described in Section 4.7 allow for larger graphs, hence the greater number of vertices tested.

5.2 Algorithm Summary

Table 5.1 gives a brief overview of the algorithms inplemented in this paper, allowing for quick reference to the location of the pseudo code for each algorithm.

Name	Algorithm Number(s)	Pseudo Code Page(s)
Harish and Narayanan's	41 and 42	10 and 11
Quoc-Nam Tran's Floyd-Warshall	12 and 13	43 and 44
Quoc-Nam Tran's CUDA APSP	14 and 15	45 and 46
Katz and Kider's	16, 17 and 18	49, 50 and 50
Modified Katz and Kider's	19	51

Table 5.1: A Sumary of Each Implemented Algorithm

5.3 Evaluation Setup

All CUDA and CPU experiments were conducted on the same PC with the specifications shown in Figure 5.1.

- CUDA 4.0
- NVIDIA 280.26 GeForce Driver
- NVIDIA GeForce GTX 470 GPU
- 1.2GB Dedicated GPU Memory
- Intel Core i5 CPU
- 4GB RAM 1333Mhz
- Windows 7 Professional x86
- Visual Studio Professional 2008

Figure 5.1: Experimental PC Specifications

In order to answer the research question posed in Section 1.1, it was necessary to test all algorithms on the same computer, in order to be able to fairly compare the results presented. If different computers had been used, external factors would have affected the results, such as GPU speed, meaning that each algorithm would not be comparable to another. Indeed, the same factors are relevant for the CPU algorithms that were tested.

5.4 Quoc-Nam Tran's Floyd-Warshall Results

Tran (2010) do not provide any statistical evidence as to any performance increase or decrease when compared to Floyd-Warshall despite being based on that algorithm. Instead, they simply state that their CUDA Floyd-Warshall algorithm provides a speed-up of between 48x and 52x when compared to a single core (sequential) Floyd-Warshall algorithm (Section 3.2.1).

Shown in Figure 5.2, we can see that the algorithm provided by Tran (2010) provides a seemingly constant speed-up over the standard sequential Floyd-Warshall algorithm. As the number of vertices in the graph increases, the time taken to complete the APSP problem reduces when compared to the previous graph, resulting in a moderately steep curve at the beginning of the graph, that begins to flatten as the number of vertices increases.

The x axis of the graph in Figure 5.2 shows us that the graph with the most vertices searched had 11,264 vertices. Due to the fact that graphs are represented by this algorithm as an adjacency matrix consisting of integers, each of which is four bytes, a larger graph could not be used. This is because not enough global memory was present on the GPU. Such a comparatively low number of vertices is achieved as Algorithm 13 utilises both the graph, and a copy of that graph on the GPU; thus halving the number of possible vertices due to there being effectively two graphs stored in device memory.

Based on the data shown in Figure 5.2 we can presume that the curve would continue, with running times very gently flattening out as the graph suggests, until a constant increase is achieved as running times will always increase slightly for larger graphs.

The results show that Algorithm 13 provides on average, a 51x speed-up over the CPU algorithm, with speed-ups in the range of between 49x and 53x. These results are consistent with the claims made by Tran (2010) and validate their results for the algorithm that we have implemented.

The GPU used in the experiments contains double the number of cores that were used for Tran (2010) experiments, possibly accounting for the 2x speed-up found in this research over the results presented by Tran (2010) which would suggest that a linear speed-up can be achieved with this algorithm by simply adding more cores. However, other factors may have resulted in this speed-up. As the problem being solved is APSP, the degree per vertex in the graph has no effect, as every element in the adjacency matrix is examined independently, resulting in the same execution time for both dense, and sparsely populated graphs.

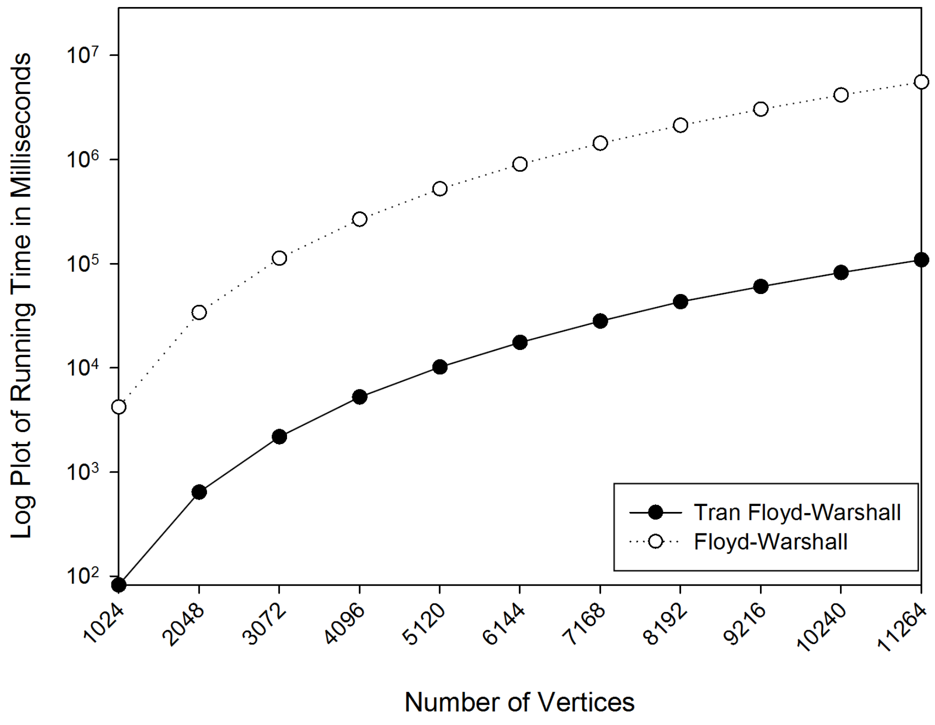

Figure 5.2: Quoc-Nam Tran CUDA Floyd-Warshall and CPU Floyd-Warshall

5.5 Quoc-Nam Tran's CUDA APSP Results

As in Section 5.4, Tran (2010) do not provide any numerical data that shows the performance of Algorithm 14 in comparison to a CPU algorithm. Again, they state that their CUDA APSP algorithm provides a speed-up of an impressive 2,500x over a CPU algorithm.

However, as stated in Section 4.5, this CPU algorithm has a running time of $O(n^3 \log n)$ which is significantly worse than that of the standard Floyd-Warshall algorithm that runs in $O(n^3)$ time. As Algorithm 4 is much faster asymptotically, these experiments compare the running time of this algorithm against Floyd-Warshall, rather than the slower $O(n^3 \log n)$ algorithm.

As we can see from Figure 5.3, the speed-up attained by this algorithm follows the same curve as the execution times for Algorithm 4. The increase in execution time between the 1024 vertex graph, and the 2048 vertex graph is noticeably greater than that of other graphs, and highlights that larger graphs have a smaller jump in execution time. We can presume from the curve displayed in Figure 5.3 that for larger graphs than were tested, this trend would continue.

Several algorithms have been presented and implemented in this research, with the aim of comparing their performance on a single platform, as well as attempting to improve them where possible, and analyse their performance. As seen in Chapter 5, some algorithms are better suited to specific graphs. For example, we know that utilising Algorithm 16 is best suited to smaller graphs, as the performance gained is much greater than any other algorithm. Similarly, Algorithms 14 and 12 are best suited to smaller graphs, offering similar execution times to Algorithm 16.

Additionally, Algorithm 10 is better suited to large graphs, offering speed ups of at least 377x over the CPU Floyd-Warshall algorithm. For denser graphs, this algorithm may not be better suited as it will have to process many more elements. This restriction does not apply to our modified algorithm (Algorithm 19) however, and therefore this Algorithm (Algorithm 19) may prove to be a better choice where graph density is unknown.

All of our algorithms provide significant performance increases over their CPU counterparts, with an average speed-up of 154x over CPU Floyd-Warshall.

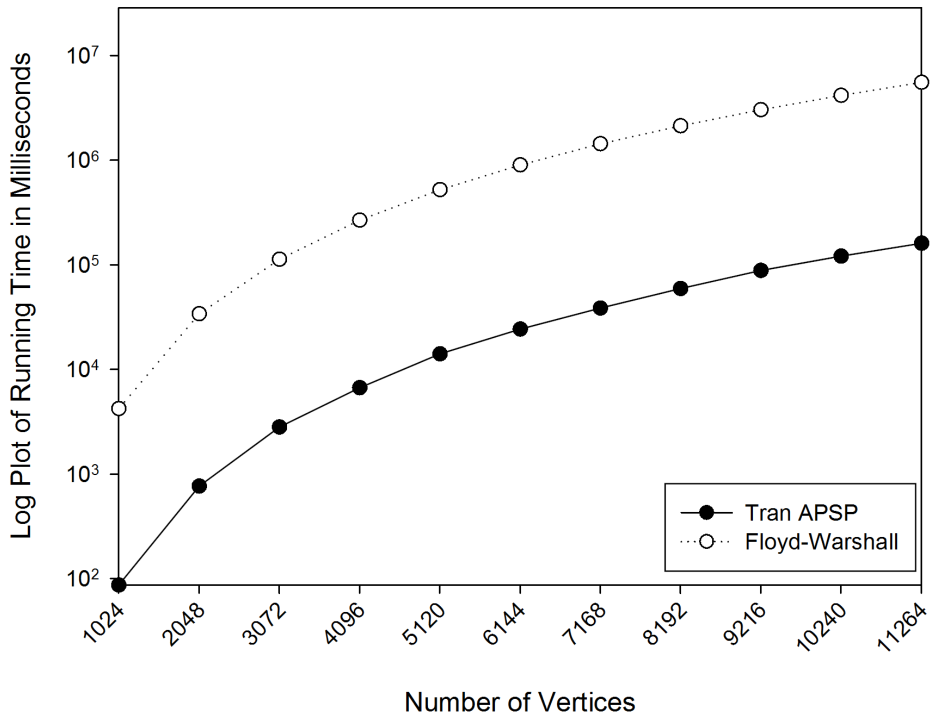

Figure 5.3: Quoc-Nam Tran CUDA APSP and CPU Floyd-Warshall

The results show that Algorithm 14 provides an average speed-up of 39x over Algorithm 4, with speed-ups between 34x and 49x. The claims made by Tran (2010) of a 2,500x speed-up over a CPU algorithm are somewhat irrelevant, as that CPU algorithm is inherently worse than the $O(n^3)$ Floyd-Warshall algorithm and would obviously provide a much greater speed-up over it. As we will see in Section 5.9, this algorithm is not the best in class for solving the APSP problem.

5.6 Katz and Kider's Results

The results presented in Katz and Kider (2008) state that their CUDA algorithm provides a speed-up of between 60x and 130x over the standard Floyd-Warshall algorithm described in Algorithm 4. Additionally, a speed-up of between 45x and 100x is offered over the blocked algorithm presented in Section 3.2.4 (on which this CUDA algorithm is based). Furthermore, Katz and Kider (2008) claim that their implementation is between 5x and 6x faster than the algorithm by Harish and Narayanan (2007) discussed in Section 4.3. This final claim is discussed in Section 5.9.

Figure 5.4 shows that Algorithm 16 provides an excellent speed-up over Algorithm 4, with an apparent increase in speed-up as the size of the graph increases. Indeed, with 1024 vertices, we see a speed-up of 104x, whilst when we have 17,408 vertices, a speed-up of 153x is achieved by the algorithm.

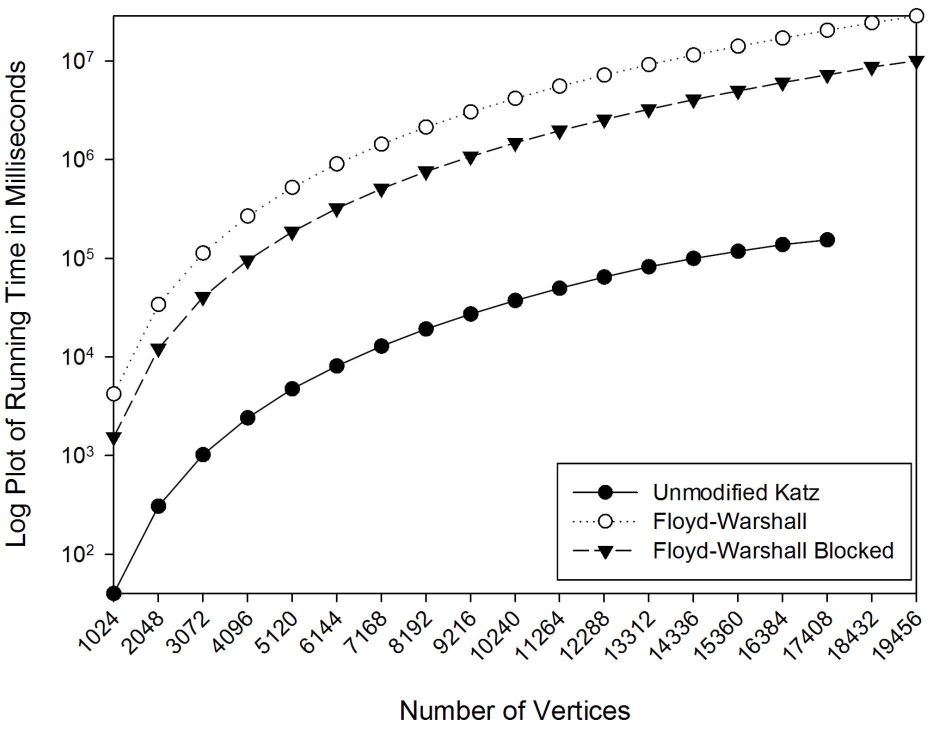

Figure 5.4: Katz and Kider's Algorithm, CPU Floyd-Warshall and CPU Blocked Floyd-Warshall

We see a similar pattern when comparing Algorithm 16 against Algorithm 8. Although, instead of seeing speed-ups of 104x and 153x, we see 38x and 54x respectively. The cache efficient CPU algorithm presented in Section 3.2.4 does not provide a linear speed-up over the Floyd-Warshall algorithm. Instead, as the size of the graph increases, the speed-up gained over Floyd-Warshall increases. This explains why the algorithm presented by Katz and Kider (2008) does not provide as stark a speed-up over Algorithm 8 as it does over Algorithm 4, due to the fact that there is less performance difference between them as a result of Algorithm 8's speed-up.

As we can see in Figure 5.4, only 17,408 vertices were available in the largest graph searched. This is due to the fact that the next largest graph, with 18,432 vertices, occupies just under 1.3GB which is too great for the GPU to hold. As a result, searching was not attempted with any larger graphs. This problem is solved by the modified version of this algorithm however, and its results are presented in Section 5.7.

The results presented are slightly better than those claims made by Katz and Kider (2008), in that speed-ups range between 104x and 153x when compared to the standard Floyd-Warshall algorithm. Conversely, the results presented in Figure 5.5 for comparison to the blocked algorithm (Algorithm 8) are slightly worse than those presented by Katz and Kider (2008). For these results, we see speed-ups of between 38x and 54x whereas Katz and Kider (2008) observed between 45x and 100x.

A possible reason for this may be that the CPU used for this experiment has a much better system for handling its cache, for which this CPU algorithm was designed, as the CPU used for this test is many generations better than that used by Katz and Kider (2008). This may have resulted in a speed-up in the blocked algorithm's code, as it was able to better utilise the features of the CPU in comparison to the standard Floyd-Warshall algorithm, potentially biasing the results of this test slightly.

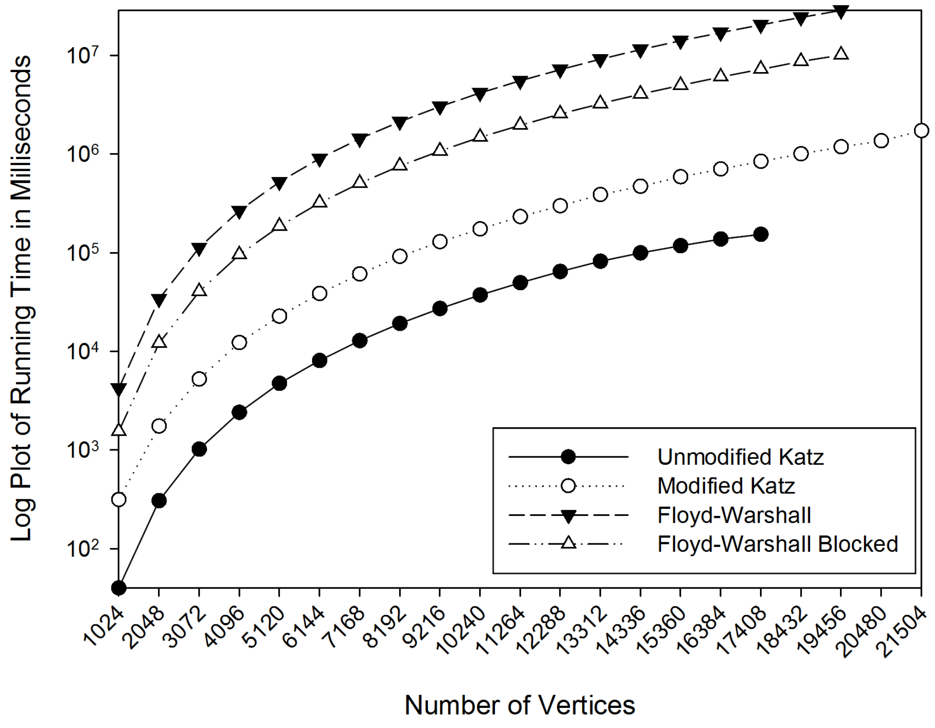

Figure 5.5: Katz and Kider's Algorithm, Modified Algorithm, CPU Floyd-Warshall and CPU Blocked Floyd-Warshall

5.7 Modified Katz and Kider's Results

As we know from Section 4.7, the subject graph is partitioned before kernel execution, and relevant parts of the graph are copied to and from the device to accommodate graphs whose size is larger than the capacity of device memory. As a result, execution time is increased, due to the additional copies to and from the device.

For this experiment, we ran the algorithm on all graph sizes, including "small" graphs that would normally fit onto device memory. For these small graphs, they were partitioned in the same way as a large graph would be, to simulate the number of transfers between host and device for each kernel execution. In this way, we can compare the two algorithms directly, to see what effect these extra memory copies have on execution.

As we can see in Figure 5.6, Algorithm 19 closely follows the same pattern as Algorithm 16. As we get to graphs whose size is larger than can fit onto device memory, the execution times continue along the same curve, showing a predictable increase in execution time as graph size continues to increase.

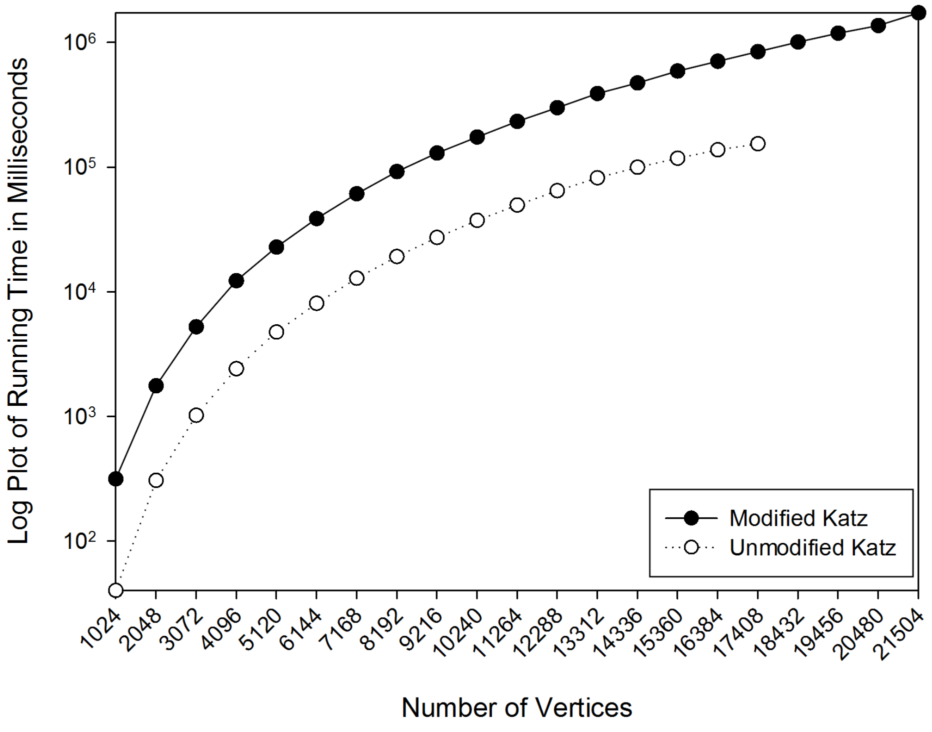

Figure 5.6: Katz and Kider's Shared Algorithm and Modified Algorithm

Despite being noticeably slower than the standard algorithm by Katz and Kider (2008), roughly between 2x and 8x, there is still a vast improvement over both Algorithm 4 and Algorithm 8. So much so that we could run experiments on graphs larger than either CPU algorithm simply because the execution time was significantly faster. Both CPU algorithms (Algorithms 4 and 8) ran out of system memory due to the requirements of their implementations whereas Algorithm 19 was not bound by these requirements, and could continue for larger graphs.

Figure 5.7 presents a speed-up of between 13x and 24x over the standard Floyd-Warshall algorithm, and speed-ups of 4x and 9x over the Blocked Floyd-Warshall algorithm presented in Section 3.2.4.

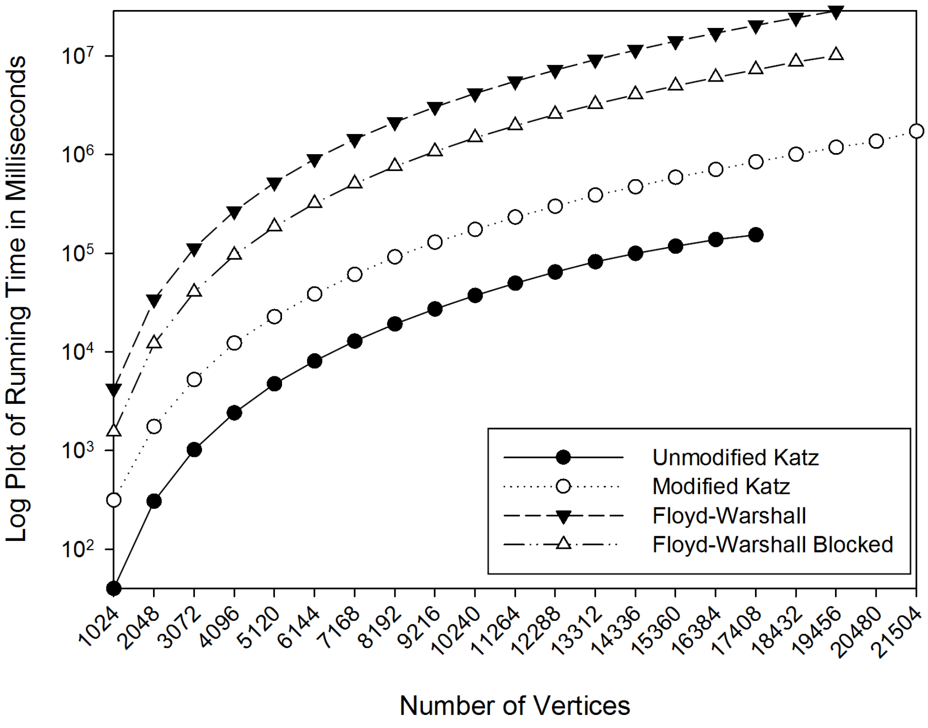

Figure 5.7: Katz and Kider's Shared and Modified Algorithms with Floyd-Warshall and Blocked Floyd-Warshall

5.8 Harish and Narayanan's Results

Algorithm 10 is the only algorithm described in Chapter 4 that utilises the compacted adjacency list structure described in Section 4.1.1 and as such, is able to search graphs with many more vertices. This is due to the memory constraints being just $O(|E| + |V|)$, rather than $O(|V|^2)$ as with a traditional adjacency matrix.

As we can see in Figure 5.8, as the number of vertices in the graph increases, the speed-up gained by Algorithm 10 also increases. For example, the graph with 1024 vertices provides a speed-up of just under 2x, but we see a speed-up of 7x for the next graph of 2048 vertices. This trend continues throughout each graph searched, until the final graph where Floyd-Warshall was able to search with 19,456 vertices. This provided a speed-up of 377x.

These results match closely with those presented by Harish and Narayanan (2007), and confirm their claims. They did not however, run the Floyd-Warshall algorithm on graphs with more than approximately 5000 vertices, whereas our experiments continued until 19,456 vertices. In running our experiments on larger graphs, we can see more evidence that greater speed-ups are attained as the size of the graph increases.

It would appear that the memory transfer between host and device after each execution of Algorithm 10 and Algorithm 11 does not have a great impact on execution time. Due to the fact that the majority of the data copied is of boolean type, the size of data transfer is limited, thus helping to reduce the impact of memory transfer latency on the execution time of the algorithm.

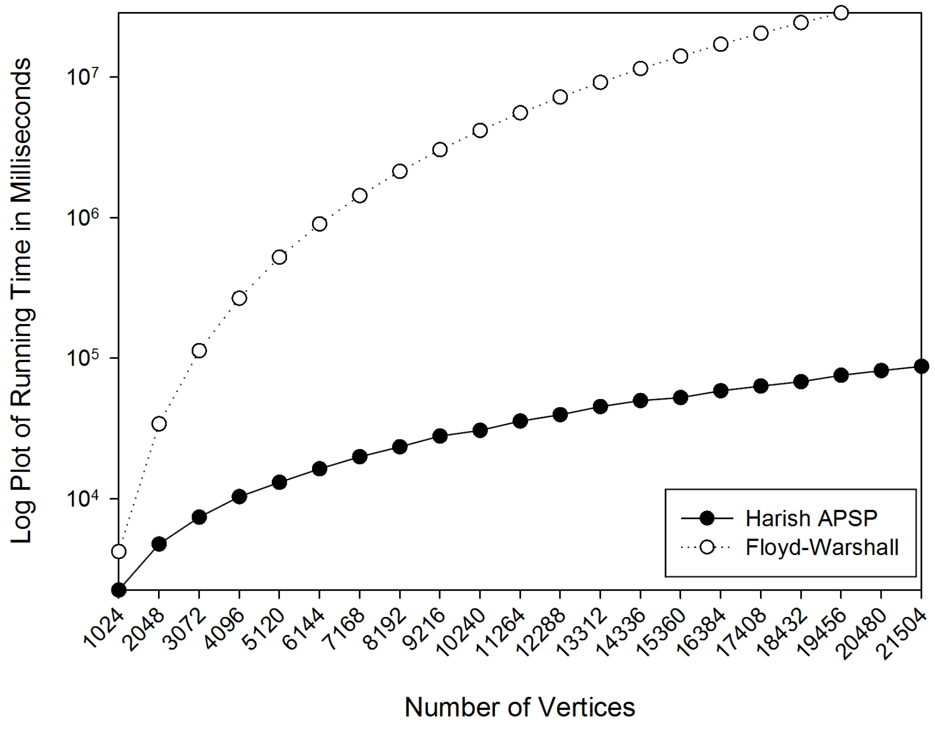

Figure 5.8: Harish and Narayanan's Algorithm with CPU Floyd-Warshall

5.9 CUDA Results Comparison

Figure 5.9 gives us an overview of every CUDA algorithm detailed in Chapter 4, comparing their running times with the number of vertices in the graph.

When comparing all CUDA algorithms to Floyd-Warshall, we find that the algorithm with the best average speed-up over Floyd-Warshall is Algorithm 16 (Katz and Kider, 2008) with an average speed-up of 154x. However, we can see that Algorithm 10 by (Harish and Narayanan, 2007) provides the best improvement for large graphs of at least 9,216 vertices. This improvement is relative however, where denser graphs will decrease the benefit of Algorithm 10 due to the increased memory requirements of the algorithm. For denser graphs, our modified Algorithm (Algorithm 19) would be better suited. This highlights the improtatnce of memory optimisations when considering GPGPU.

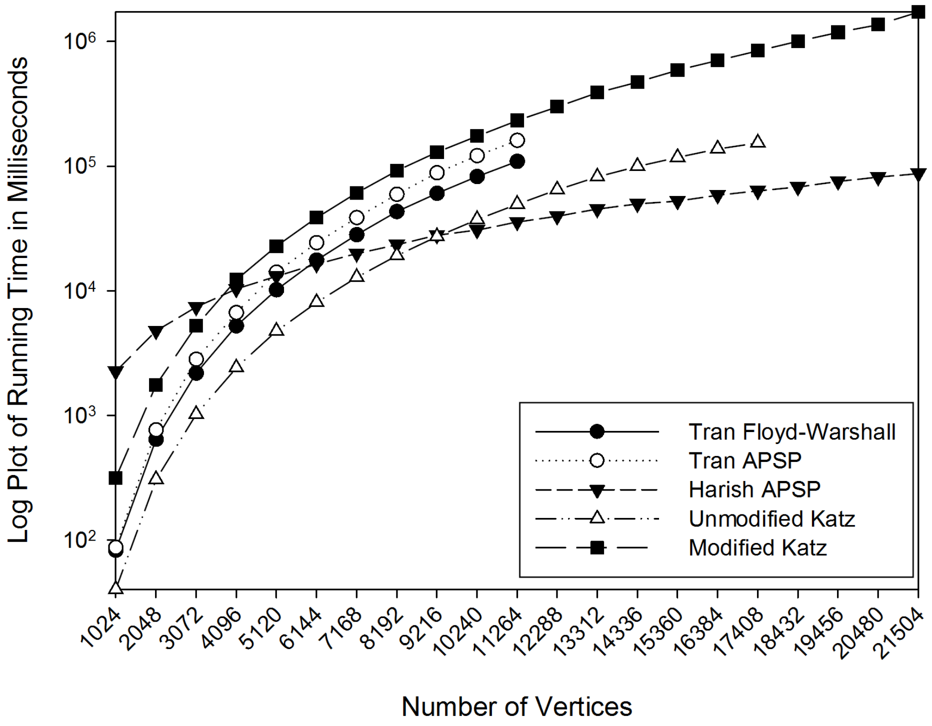

Figure 5.9: All CUDA Algorithms

5.9.1 Quoc-Nam Tran's Algorithms

In this section, we will look at both CUDA Floyd-Warshall and CUDA APSP, Algorithm 12 and Algorithm 14, respectively, both of which are presented by Tran (2010). We look at both algorithms in this section. Both algorithms offer extremely similar results, and are also architecturally similar.

If we look at Figure 5.9, we can see that Algorithm 12 starts off extremely well, beaten only by Algorithm 16. In fact, this trend continues until we reach a graph with 6,144 vertices at which point, Algorithm 10 gains an advantage.

After this point, Algorithm 12 remains the faster of the remaining two algorithms (Algorithms 14 and 19). These results suggest that the algorithm is best suited to smaller graphs, where it is known that the subject graph will never exceed a certain size. In this case, approximately 6000 vertices.

As we know from Section 4.4, the algorithm does not copy data to and from the device between kernel calls. Additionally, the kernel also utilises shared memory. From this, we know that data transfer between host and device is not affecting the performance of the algorithm and does not account for the algorithm starting so well, but falling short as the size of the graph increases.

Looking more closely at the algorithm, we know that each kernel call results in 24 reads from device memory per block, and 24 writes to device memory per block where the block size is 16 x 16. Additionally, only one computation is performed by each thread in the block, whereas other algorithms perform multiple computations.

The ratio between computations in shared memory, and the reading/writing to device memory would appear to have a negative effect on the performance of the algorithm as the graph size increases, resulting in more overall reads/writes to device memory as more data is needed.

This is likely to be the reason why Algorithm 16 is faster than either Algorithm 12 or 14 for large graphs. If we look at Algorithm 16, we know that it makes extensive use of shared memory in order to perform its computations and also breaks the graph into manageable chunks. Additionally, every thread in Algorithm 16 performs the same instruction, whereas both of the algorithms presented by Tran (2010) have instructions that only a subset of threads perform. Adding to the performance decrease, both algorithms perform multiple conditional checks, whereas Katz and Kider (2008) have no conditionals in phases 1 and 2.

As we know from Figure 5.9, both of the algorithms given by Tran (2010) are faster than Algorithm 16 for smaller graphs. This can be explained by the offset of cost between conditionals as used by Tran (2010), and loops utilised by Katz and Kider (2008). For small graphs, the cost of the conditional statements is lower than that of the for loops as there is not as much data to process. However, as the graph sizes increase, we can see that utilising loops becomes more effective than the conditional checks, and provides greater performance.

When comparing Algorithms 12 and 14, we see that their results are very similar. Each share the same number of threads and blocks, in the same layout. Additionally, kernel structure is very similar, with rows and columns being copied into shared memory for computation. However, Algorithm 14 utilises a loop structure with conditionals nested inside the loop. Algorithm 12 simply utilises a conditional without the need for the looping structure. The lack of loop could account for the increased performance which is displayed in Figure 5.9.

As we can see, both algorithms follow an extremely similar curve on the graph, suggesting that the performance penalty paid by having a looping structure is related to the size of the graph and accounts for the difference in performance.

5.9.2 Katz and Kider's Algorithm

In this section, we look at the results of the algorithm presented by Katz and Kider (2008) (Algorithm 16) and compare the results against the other CUDA algorithms implemented.

When we observe Figure 5.9, we can see that Algorithm 16 starts off extremely well, offering the lowest execution time of all algorithms. During that time, execution time steadily increases in a fairly steep curve, reflecting every other algorithm (excluding Algorithm 10). When we reach 10,240 vertices, the algorithm becomes slower than Algorithm 10 for all remaining graphs. At its peak, this algorithm is at least 2.3x faster than any other algorithm implemented.

Despite being overtaken performance-wise by Algorithm 10, this algorithm (Algorithm 16) still remains a good performer in comparison to the remaining algorithms, ensuring that it is always faster than them. If we look at the curve presented in Figure 5.9, it appears to be levelling out towards the end, for the largest graphs searched. For example, when performing the algorithm on 16,384 vertices, the decrease in performance to the next graph (17,408 vertices) is less than from 13,312 vertices to 16,384. If this trend were to continue, we could surmise that the performance of this algorithm may once again overtake the performance of the algorithm given by Harish and Narayanan (2007) (Section 4.3). This is due to the fact that Algorithm 10 appears to have a linear increase in execution time of 1.1x for larger graphs of at least 12,000 vertices.

In order to be able to test this, a GPU with a much larger global memory is required, and may not be available for several years depending on the GPUs released by NVIDIA in that time.

5.9.3 Harish and Narayanan's Algorithm

Looking at Figure 5.9, we can see that Algorithm 10 is by far the slowest algorithm initially, where graph sizes are small. However, the curve is much more gentle than other graphs, as its performance decreases only slightly as the graph size increases. In fact, the algorithm given by Harish and Narayanan (2007) is only 32x slower for the largest graph searched when compared to the smallest graph. However, if we look at Algorithm 12, we see an increase of 1322x.

This results in the algorithm becoming the fastest for larger graphs of approximately 10,000 vertices and above. The structure of the algorithm is drastically different to all other implementations, as it utilises the compacted adjacency list structure as described in Section 4.1.1. Additionally, rather than utilising one data structure, supplemental information is required such as the M_a array (as described in Section 4.3). Despite the extra arrays required, the physical size of the graph is still a lot smaller when compared to the traditional adjacency matrix. $O(|V| + |E|)$ as opposed to $O(V^2)$.

This may explain why there is only a comparatively small decrease in performance as graph size increases. Due to the reduced storage requirements of the compacted adjacency list, increasing graph size has a smaller effect on the number of elements that the algorithm must process.

This algorithm is able to handle the largest graphs of all other implementations, excluding the modifications that we made to the original algorithm implemented by Katz and Kider (2008). This is solely down to the fact that the compacted adjacency list is used, which has the benefit of eradicating redundant data. However, if an intensely dense graph were to appear, the modified algorithm (Algorithm 19) would continue to be able to operate, whereas this algorithm would not, as the storage required would be too great. To that end, it can be said that this algorithm operates best on graphs that are not densely populated.

When comparing our modified Katz and Kider (2008) algorithm with the algorithm given by Harish and Narayanan (2007), we notice that Algorithm 10 is on average, 7.5x faster. However, at its peak, the algorithm is actually 19x faster for a graph with 21,504 vertices. A major factor contributing to the difference in execution times is the copying of data to and from the device for Algorithm 19. As we know from Section 4.7, in order to facilitate large graphs, Algorithm 19 partitions the graph into sections, and copies it to the GPU in stages, building up the computation piece by pieces. This results in many large data transfers, impacting significantly on execution times, providing an added speed-up for Algorithm 10.

From Figure 5.9, we see that this algorithm is initially slower than both algorithms given by Tran (2010). However, this is not the case when the graphs have at least 7,168 vertices as the algorithm provides an increasing speed-up as graph size increases. As only 11 graphs searched are operated on by Tran (2010), this algorithm only provides a speed-up on six of them despite utilising more graphs overall. This is simply due to the fact that the adjacency matrix method of storing graphs used by Tran (2010) requires $O(V^2)$ as two copies of the graph are stored in device memory. We have seen that Algorithm 10 operates well on the graphs utilised for these tests. However, further work needs to be conducted to assess its suitability on densely populated graphs when compared to our modified algorithm (Algorithm 19).

Chapter 6

Conclusions

In this thesis, the research documented was designed to look at the APSP problem in the GPGPU space, investigating existing work, determining which algorithms are best, and attempting to improve upon the authors' work investigated where possible.

The APSP problem is a common problem throughout many different subject areas. Fast, sequential algorithms are often expensive to run as they require a very fast computer, such as a supercomputer, to execute on. Parallel computing on the GPU provides a much cheaper alternative, allowing for fast algorithms to be written at a fraction of the cost. CUDA is one such parallel computing technology that enables parallel algorithms to be written for CUDAs enabled GPU manufactured by NVIDIA. Using CUDA to create these parallel algorithms allows them to be distributed to at least 100 million machines with CUDA enabled GPUs and so are not solely restricted to scientific applications due to their commercial status.

We have seen several APSP algorithms implemented in Chapter 4 as well as their results in Chapter 5. Each algorithm was tested on the same machine, so that their performance might be critically and fairly compared and analysed so as to establish patterns and trends between each algorithm.

The same graphs were used on each algorithm so as to ensure that different data could not affect the results, and the machine was left in a constant state so as to ensure fairness and comparability between results.

As we saw in Chapter 5, different algorithms are best suited to different tasks. For example, Figure 5.9 shows us that Algorithm 16 is best suited to graphs whose total number of vertices is comparatively small. The algorithm gives the best performance boost when compared to any other of the CUDA algorithms that were tested and analysed. However, Algorithms 14 and 12 also provide a significant performance increase over their CPU counterparts.

For larger graphs, Algorithm 10, originally described by Harish and Narayanan (2007), is the best of the five algorithms, providing a significant performance increase over its CPU counterpart (Dijkstra's algorithm). However, Algorithm 19 also provides a performance increase, but not as much as Algorithm 10. Moreover, as we saw in Chapter 5, we know that Algorithm 10 is dependent on the density of the graph. If there are more edges on the graph, it will take longer to execute. Our modified algorithm (Algorithm 19) does not suffer from this restriction, and will have the same execution time no matter what the density of the graph. For that reason, it may be beneficial to use our modified algorithm over Algorithm 10 when the density of the graph is unknown.

All of our implementations provide a significant performance increase over their CPU counterparts, with an average performance increase of 154x over Floyd-Warshall. We have seen in Chapter 5 that all the authors' claims have been verified, in that similar results were observed in our codes, and when comparing the CUDA algorithms against each other. It is clear that different graphs are best used by different algorithms, with no one algorithm being a clear winner. An informed decision should be made before a particular algorithm is used, based on the properties of the subject graph and the intended application of the algorithm.

Additionally, the machine used to execute the algorithms must be taken into consideration. A data transfer heavy algorithm such as Algorithm 10 should not be used where the data bus does not have a high enough bandwidth, as data transfer times will be impacted significantly during kernel execution.

Our modified algorithm presented in Section 4.7 provides an excellent improvement over the traditional algorithm in allowing for much larger graphs to be utilised. The small performance impact of allowing these graphs is offset by the fact that the algorithm is much more versatile in the operations that it can perform.

Furthermore, this work provides application for many important research areas and spans many subject areas, including, but not limited to, medical research in the analysis of biological networks (Shi and Zhang, 2011). The vast utility of such applications means that any future research into such an area would be extremely beneficial. Section 6.1 details such examples of future work.

6.1 Future Work

The research presented in this thesis was successful in implementing APSP algorithms and comparing their performance on a common platform, as well as improving them where possible. The work completed and the solutions implemented can provide a strong basis for future work where improvements in CUDA may apply. Additionally, new solutions can be implemented following the sentiments in this thesis, so that they might be compared to the work presented here.

Solutions for creating a new data structure for graph storage could be implemented. New data structures would give way to a host of new CUDA APSP algorithms with the potential of improving on the performance of traditional APSP algorithms. The focus of new research would benefit from these implementations by giving a wider variety of options for graph storage to choose from. Currently a compromise must be made between access times and storage requirements. Researching a new storage method that combines the two may be beneficial in certain scenarios.

More tests could be conducted on Algorithm 10 (Section 4.3) to determine what effect the density of subject graphs has on the performance of the algorithm. Out of the CUDA algorithms implemented, Algorithm 10 is the only one that presents differing performance based on the density of the graph. As described in Chapter 4, algorithms utilising the adjacency matrix method of graph storage operate over all the data in the matrix. Algorithm 10 operates on a compacted adjacency list which eradicates redundant data meaning that it only operates on the necessary data, potentially reducing execution times for sparse graphs.

6.2 Final Conclusion

The research questions posed in Chapter 1 have been answered successfully by this research. To show this, the questions are repeated here with a summarised view of their answers.

1. **Which All-Pairs Shortest Path algorithms solve their problem the fastest, and can the authors' claims be verified?**

   We have seen that all CUDA algorithms solve the APSP problem faster than their CPU counterparts. Algorithms 16 and 10 in particular solve the APSP problem in the fastest amount of time, providing average speed-ups of 113x and 129x respectively. Chapter 5 clarifies that the authors' original claims are valid, with our results reflecting their claims. From this, we can deem that our original question posed in Chapter 1 has been answered successfully.

2. **How do CUDA algorithms compare against their CPU counterparts?**

   Chapter 5 gives us plenty of evidence that all CUDA algorithms are significantly faster than their CPU counterparts. We see that over all the CUDA algorithms implemented, there is an average speed-up of 76x over the Floyd-Warshall algorithm (Algorithm 4) and 27x over the Blocked Floyd-Warshall algorithm (Algorithm 8).

3. **Can these CUDA algorithms be improved or modified in any beneficial way**?

   We have seen from Chapter 4 that we have improved the algorithm given by Harish and Narayanan (2007). Additionally, we have dramatically modified the blocked algorithm presented by Katz and Kider (2008) in order that graphs larger than DRAM might be used. Both of these modifications are beneficial to the algorithm. Firstly, the modifications to Algorithm 10 allow the algorithm to better utilise CUDA hardware in order to achieve improved performance. Finally, the modification to Algorithm 16 is extremely beneficial in giving us a flexible algorithm capable of being utilised on a much larger number of graphs.

Bibliography

George S. Almasi and Allan Gottlieb. *Highly parallel computing.* Benjamin-Cummings Publishing Company, 1988.

Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. doi: http://doi.acm.org/10.1145/1465482.1465560. URL `http://doi.acm.org/10.1145/1465482.1465560`.

David Applegate, William Cook, Sanjeeb Dash, and Andre Rohe. Solution of a min-max vehicle routing problem. *Informs Journal on Computing*, 14(2): 132–143, 2002.

Blaise Barney. Introduction to parallel computing, January 2010. URL `https://computing.llnl.gov/tutorials/parallel_comp/`. Unpublished.

R. E. Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16: 87–90, 1958.

A. J. Bernstein. Analysis of programs for parallel processing. *Electronic Computers, IEEE Transactions on*, EC-15(5):757 –763, oct. 1966. ISSN 0367-7508. doi: 10.1109/PGEC.1966.264565.

Jens Breitbart. Case studies on gpu usage and data structure design. Master thesis, Universitat Kassel, Kassel, Germany, 2008.

Fred Brooks. No silver bullet essence and accidents of software engineering. *Computer*, 20:10–19, 1987. ISSN 0018-9162. doi: http://doi.ieeecomputersociety.org/10.1109/MC.1987.1663532.

AydIn Buluc, John R. Gilbert, and Ceren Budak. Solving path problems on the gpu. *Parallel Computing*, 36(5-6):241 – 253, 2010. ISSN 0167-8191. Parallel Matrix Algorithms and Applications.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, 2001.

Frank Dehne and Kumanan Yogaratnam. Exploring the limits of gpus with parallel graph algorithms. *Arxiv preprint arXiv10024482*, abs/1002.4, 2010.

E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. ISSN 0029-599X. URL `http://dx.doi.org/10.1007/BF01386390`. 10.1007/BF01386390.

Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5:345–, June 1962. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/367766.368168. URL `http://doi.acm.org/10.1145/367766.368168`.

Michael J. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948 –960, sept. 1972. ISSN 0018-9340. doi: 10.1109/TC.1972.5009071.

Jon Freeman. Parallel algorithms for depth-first search. Technical report, University of Pennsylvania, 1991.

Pawan Harish and P. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In Srinivas Aluru, Manish Parashar, Ramamurthy Badrinath, and Viktor Prasanna, editors, *High Performance Computing HiPC 2007*, volume 4873 of *Lecture Notes in Computer Science*, pages 197–208. Springer Berlin / Heidelberg, 2007.

Pawan Harish, Vibhav Vineet, and P. J. Narayanan. Large graph algorithms for massively multithreaded architectures. Technical report, International Institute of Information Technology Hyderabad, India, 2009.

C. Hitte, T. D. Lorentzen, R. Guyon, L. Kim, E. Cadieu, H. G. Parker, P. Quignon, J. K. Lowe, B. Gelfenbeyn, C. Andre, E. A. Ostrander, and F. Galibert. Comparison of multimap and tsp/concorde for constructing radiation hybrid maps. *J Hered*, 94:9–13, 2003.

Jared Hoberock and Nathan Bell. Thrust: A parallel template library, 2009. URL `http://www.meganewtons.com/`. Version 1.3.

Jorkki Hyvonen, Jari Saramaki, and Kimmo Kaski. Efficient data structures for sparse network representation. *International Journal of Computer Mathematics*, 85(8):1219–1233, 2008.

Swapnil D. Joshi and Mrs. V. S. Inamdar. Performance improvement in large graph algorithms on gpu using cuda: An overview. *International Journal of Computer Applications*, 10(10):10–14, November 2010. Published By Foundation of Computer Science.

Gary J. Katz and Joseph T. Kider, Jr. All-pairs shortest-paths for large graphs on the gpu. In *GH '08: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 47–55, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association. ISBN 978-3-905674-09-5.

Khronos. *The OpenCL Specification*, June 2011.

Paulius Micikevicius. General parallel computation on commodity graphics hardware: Case study with the all-pairs shortest paths problem. In *PDPTA*, pages 1359–1365, 2004.

Microsoft. *DirectCompute Shader Overview*, September 2012.

Robert H.B. Netzer, Sanjoy Ghosh, Robert H. B, and Netzer Sanjoy Ghosh. Efficient race condition detection for shared-memory programs with post/wait synchronization, 1992.

NVIDIA. *NVIDIA CUDA. Programming Guide Version 2.3*, July 2009.

NVIDIA. *CUDA C Best Practices Guide*, May 2011a.

NVIDIA. *NVIDIA C CUDA Programming Guide*, May 2011b.

NVIDIA. *CUDA C Best Practices Guide*, January 2012.

T. Okuyama, F. Ino, and K. Hagihara. A task parallel algorithm for computing the costs of all-pairs shortest paths on the cuda-compatible gpu. pages 284 –291, dec. 2008.

Zhiao Shi and Bing Zhang. Fast network centrality analysis using gpus. *BMC Bioinformatics*, 12(1):149, 2011. ISSN 1471-2105. doi: 10.1186/1471-2105-12-149. URL `http://www.biomedcentral.com/1471-2105/12/149`.

A. Silberschatz, P.B. Galvin, and G. Gagne. *Operating System Principles, 7th Ed.* Wiley student edition. Wiley India Pvt. Limited, 2006. ISBN 9788126509621. URL `http://books.google.co.in/books?id=WjvX0HmVTlMC`.

Ian Sommerville. *Software Engineering.* Pearson Education Limited, eighth edition, 2007.

Quoc-Nam Tran. Designing efficient many-core parallel algorithms for all-pairs shortest-paths using cuda. pages 7–12, April 2010.

Gayathri Venkataraman, Sartaj Sahni, and Srabani Mukhopadhyaya. A blocked all-pairs shortest-paths algorithm. *J. Exp. Algorithmics*, 8:2.2, 2003. ISSN 1084-6654.

V. Vineet and P.J. Narayanan. Cuda cuts: Fast graph cuts on the gpu. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, pages 1 –8, june 2008. doi: 10.1109/CVPRW.2008.4563095.

Vasily Volkov and James W. Demmel. Lu, qr and cholesky factorizations using vector capabilities of gpus. Technical report, University of California, Berkeley, May 2008.

C. Xavier and S. S. Iyengar. *Introduction to Parallel Algorithms.* John Wiley & Sons Inc., 1st edition, 1998.