Parallel Graph Searching Algorithms using CUDA

Student Name: J. M. Kemp

Supervisor Name: I. A. Stewart

Submitted as part of the degree of BSc Software Engineering to the

Board of Examiners in the School of Engineering and Computing Sciences, Durham University

***Abstract* —** **Context** - Graph searching is a common activity in computer science. Fast algorithms that search large graphs are not always cost effective, requiring supercomputers to achieve results in a practical amount of time. Graphics Processing Units (GPUs) provide a cost alternative to supercomputers, allowing parallel algorithms to be executed on the GPU.

Aims - To create a library of parallel graph searching algorithms using NVIDIAs Compute Unified Device Architecture (CUDA) as well as their sequential equivalents in order to compare their running times when searching identical graphs.

Method - Breadth-first search, depth-first search, Prim's and Kruskal's minimum spanning tree algorithms and Dijkstra's algorithm were all implemented sequentially and with CUDA. Optimising the CUDA implementations was very important to increase their performance and efficiency in terms of running time, and, as such, needed solid designs before they could be effectively implemented. A graph creation application was also created along with a graph loader. Finally, a graphical user interface was implemented to demonstrate each of the algorithms effectively.

Results - Each algorithm was timed in milliseconds to enable statistical analysis of the running time of each algorithm. CUDA and sequential algorithms were compared directly when searching identical graphs. This showed whether CUDA could improve the running time of these algorithms over the sequential versions. The type of data collected lends itself to be plotted onto scatter graphs, providing meaningful information and a clear visual representation of the data. Each algorithm has its own scatter graph, with data connected via a line showing sequential and CUDA running times when searching graphs of varying size. The results show that for algorithms such as breadth-first search, Prim's algorithm and Dijkstra's algorithm, speedup over the sequential version is achievable to great success.

Conclusions - This project aimed to find that creating parallel graph searching algorithms using CUDA produces decreased running times in comparison to existing sequential algorithms on the CPU when searching large graphs. The results show that breadth-first search and Dijkstra's algorithm are always faster than their sequential counterparts. Prim's algorithm is sometimes faster than its sequential equivalent whilst the remaining two do not appear to perform well against the sequential algorithms. The results show that there is very strong potential for CUDA when the right problems are solved with it. This project has provided successful algorithms that are highly beneficial to the area of graph searching.

***Keywords* —** CUDA, GPU, graph searching, parallel algorithms, speedup

I INTRODUCTION

Graph searching is a common problem throughout many different subject areas and used daily in applications from vehicle routing (Applegate et al. 2002) to gene mapping (Hitte et al. 2003). Compute Unified Device Architecture (CUDA) is a powerful technology which allows parallel algorithms to be created that execute on modern NVIDIA Graphics Processing Units (GPUs), utilising the parallel hardware architecture of CUDA enabled GPUs.

Parallel computing is “a form of computation in which many calculations are carried out simultaneously” (Almasi & Gottlieb 1988). Figures 1 and 2 show a visual comparison between a standard sequential algorithm and a parallel algorithm running on the CPU. 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. For example, this paper is being written while at the same time, the earth is orbiting the sun and cars are being driven.

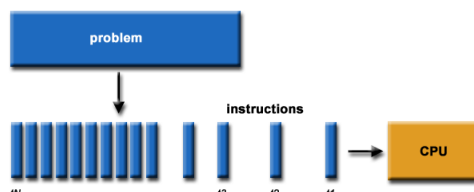


Figure 1: A representation of a sequential algorithm on the CPU (Barney 2010)

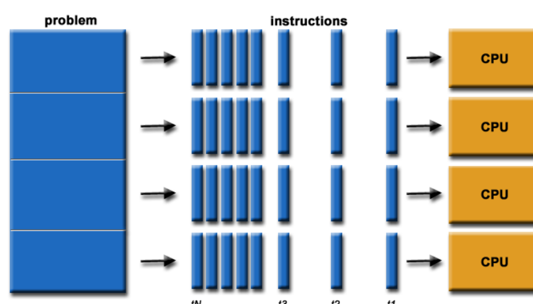


Figure 2: A representation of a parallel algorithm on multiple CPUs (Barney 2010)

Parallel computing on the GPU has become very relevant in recent years, providing a cost effective alternative for parallel algorithms that would normally be exclusive to supercomputers, with a low-end CUDA enabled GPU costing approximately £25. Modern graphics cards have a specialised hardware architecture that can be represented as a parallel computer. Unlike traditional graphics cards, cards such as NVIDIA’s 8800GTX are equipped with 16 multiprocessors, each with 8 cores, providing 128 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 the parallel architecture.

CUDA enabled GPUs now have an install base of at least 100 million units (NVIDIA 2009), showing that new, parallel algorithms can be distributed easily to a large number of machines. As CUDA is a relatively new technology, only being released in 2007, limited work has been accomplished in the field of graph searching using CUDA. On a broader scale, research that has been undertaken using CUDA covers a wide variety of subject areas but in little depth.

Since CUDA’s inception, there has been strong evidence to show that parallel algorithms on the GPU improve the performance on classic problems when compared to equivalent sequential algorithms. This project will look into the area of graph searching using CUDA in areas that have had little or no research completed. This will be accomplished by implementing CUDA ver-

sions of Breadth-First Search (BFS), Depth-First Search (DFS), Prim's and Kruskal's Minimum Spanning Tree (MST) algorithms and Dijkstra's algorithm as well as their sequential equivalents.

CUDA uses an extension of the C programming language, allowing parallel functions, known as kernels, to be written. Thus, all of the CUDA algorithms will be implemented in C/C++. Additionally, a graph generator will be created that allows graphs to be written to a text file as well as a graph reader that will take these graph files as input. This project will specifically look at improving the performance of these algorithms when searching large graphs of perhaps tens of thousands of vertices, when compared to the performance of their sequential solutions.

A Parallel Algorithms: A Silver Bullet?

Parallel algorithms may be harder to create than sequential algorithms as new problems are introduced that are not an issue with sequential algorithms. To solve a problem using a parallel algorithm, the problem must be decomposed into smaller sub problems which can then be solved in parallel (Xavier & Iyengar 1998, p11). This in itself could pose a difficulty as identifying where to break the problem into smaller sub tasks can be a significant obstacle. Obtaining an efficient level of synchronisation and communication between subtasks is a large problem which needs to be overcome for an algorithm to be effective. Also, inexperienced parallel programmers may resort to using sequential techniques, potentially having a negative effect on the overall performance of the algorithm.

Parallel algorithms may also be considered to not be portable, i.e. if the parallel algorithm is hardware specific, what may work on one machine may not work on another. This could be overcome if a standard was introduced for which all parallel hardware architectures could follow, e.g. Von Neumann architecture for CPUs. This is not so much of a problem for CUDA, as NVIDIA claim that existing CUDA algorithms will always work on future iterations of CUDA enabled GPUs. In an optimal world, parallelising a sequential algorithm would create linear speedup, i.e. doubling the number of processors should half the runtime. This level of speedup is hard to achieve however and requires an expertly written algorithm and cannot always be achieved.

Clearly, the main advantage of using parallel algorithms is to increase the performance for any given task if the difficulties of creating the algorithms can be overcome. Utilising a modern GPU as a platform for parallel computing presents a far more cost effective alternative for parallel algorithms than the vastly more expensive alternative of a super computer. It is doubtful that using parallel algorithms is a silver bullet (Brooks 1987) to obtain speedup over any CPU algorithm due to the difficulties in creating parallel algorithms. However, there is certainly potential to improve the performance of many applications by creating parallel algorithms. Creating multi-threaded algorithms for multi-cored CPUs does not provide the amount of speedup available when utilising the GPU due to the sheer number of cores available on the GPU.

II RELATED WORK

Due to the emerging nature of CUDA and its infancy in terms of research effort, a limited amount of published work in the area of graph searching using CUDA is available. A large amount of unpublished documentation from NVIDA is however available in the form of Application Programming Interfaces (APIs), technical documents, and conference presentations. The following section collates the available work and reviews it objectively.

A Graph Storage on the GPU

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).

The way in which the subject graph to be searched is stored on the GPU is of critical importance. Traditional methods of storing graphs ($G = (V, E)$) such as using an adjacency matrix provides a constant time method, $O(1)$, of determining whether there is an edge between two vertices but compromises on memory usage, $O(|V|^2)$, when an entry in the matrix represents no edge between vertices. Adjacency lists provide a method of storing graphs which requires less memory than an adjacency matrix, $O(|V| + |E|)$, as there is no wasted 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. Harish & Narayanan (2007) and Harish et al. (2009) describe the use of a compacted adjacency list represented in Figure 3. 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 compacted adjacency list where all of the information that would usually be stored in several lists, is compacted into a single, one dimensional array.

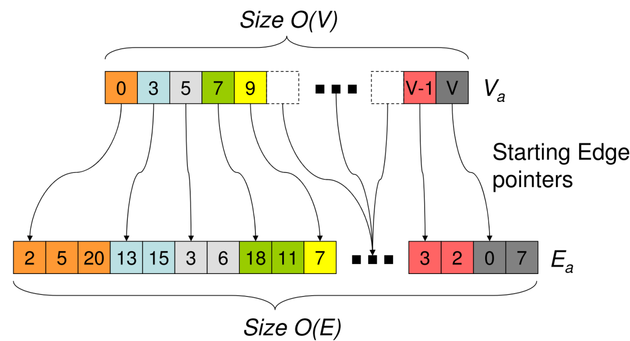


Figure 3: Graph representation as a compacted adjacency list (Harish et al. 2009)

Demonstrating a compacted adjacency list, figure 3 shows the vertex list (V_a) and the edge list (E_a). The vertex list points to a starting index in the edge list 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 the edge list to represent edge weights.

Katz & Kider (2008) and Buluc et al. (2009) look at an alternative way of storing their graphs on the GPU, opting to use the standard adjacency matrix representation. Their implementations make use of the shared memory (explained in Section III) that is utilised by CUDA. The adjacency matrix can be partitioned to sit in shared memory allowing for faster operations on the graph data. Katz & Kider (2008) manage to store graphs with just over 11 thousand vertices, occupying a considerable 1.015 Gigabytes (GB) of data. Using the compacted adjacency list, graphs of 10 million vertices with a degree of 6 are possible on a GPU with just 768 Megabytes (MB) of memory (Harish & Narayanan 2007).

B Graph Searching Using CUDA

Several successful attempts have been made at implementing traditional graph searching algorithms on the GPU using CUDA, some of which do not even use the advanced features provided by CUDA such as shared memory.

BFS has been researched and implemented using CUDA by Harish & Narayanan (2007) with very promising results. Using the compacted adjacency list, their CUDA implementation is "20-50 times faster than its CPU counterpart". Their implementation searches the entirety of an undirected, weighted graph, choosing the least cost path once the search is complete. Further speedup could therefore be achieved if an unweighted graph was searched, allowing the search to halt as soon as the goal vertex was discovered, potentially reducing the search time significantly. Their implementation of BFS does not make use of the GPUs shared memory. Finding an implementation that exploits shared memory could improve the performance of the algorithm drastically. However, they argue that using shared memory is not possible for BFS as the data required at any one time could be located anywhere in the compacted adjacency list. "Finding the locality of data to be collectively read into the shared memory is as hard as the BFS problem itself" (Harish & Narayanan 2007). The nature of BFS implies that the algorithm will have a higher performance rate when searching graphs with a higher degree per vertex and they acknowledge this before describing their implementation.

Using the compacted adjacency list, Vineet et al. (2009) present a "Fast Minimum Spanning Tree" algorithm for discovering a MST on the GPU using CUDA. When searching undirected, weighted graphs, a speedup of "30 to 50 times" is achieved when compared to their CPU implementation when using a very powerful CUDA GPU comprising of 240 cores and approximately 4GB of memory. The solution presented by Vineet et al. (2009) also makes use of an external library, CUDA Data Parallel Primitives Library (CUDPP), that provides parallel algorithms such as scan and radix sort. Using an external library such as CUDPP could affect the overall performance of the algorithm and should be analysed to see what effect they have. Vineet et al. (2009) do not analyse the algorithms that they use from CUDPP, potentially threatening the validity of their results. They also compare their implementation with a previous GPU implementation and show that they achieve "2 to 3" times speedup over it. The implementation presented is far more complex than the standard sequential implementations of a MST algorithm such as Prim's algorithm. Their graph data is partitioned and partition is stored in a bank of shared memory. Once the graph data has been operated upon by the algorithm, it then needs to be moved back into the GPUs global memory, whereby it can be collated with the rest of the data to begin forming a MST. This method of graph searching does help to provide a great deal of speedup for an inherently sequential graph problem, but comes at a cost of a complex algorithm.

C Summary of Related Work

Related work suggests that using advanced CUDA features such as shared memory is not always necessary to achieve a significant amount of speedup over CPU algorithms ("20-50 times faster" (Harish & Narayanan 2007)). Ideally, shared memory would be used in an attempt to achieve greater speedup, but this may not be possible due to the unknown location of certain graph data.

Choosing how to store graph data on the GPU is an important consideration. Using either a compacted adjacency list or the standard adjacency matrix appear to be the two common forms

of graph storage on the GPU, each with their own benefits and shortcomings. The faster adjacency matrix implies more memory usage, whereas the slower compacted adjacency list provides more efficient use of the GPUs memory. The trade-off between time and space complexity is an important consideration for this project.

Parallel algorithms clearly have the potential to improve the performance of existing sequential algorithms significantly. As mentioned in the Introduction, parallel algorithms have the potential to improve the performance of any graph searching technique significantly and many benefits of using CUDA are expected. Harish & Narayanan (2007) and Vineet et al. (2009) both show that graph searching on the GPU is possible and presents significant speedup over CPU implementations with very little cost. They also demonstrate that different categories of graph search can be accomplished; searching for a specific vertex as well as searching and building a MST are possible with significant speedup. The related work presented emphasises the relevancy of this project, providing a solid ground for new algorithms in the field of graph searching with a strong potential for speedup.

III SOLUTION

A Solution Overview

The project was split into three main categories; basic, intermediate and advanced objectives. The basic objectives were to implement CUDA and sequential breadth-first search and depth-first search algorithms as well as to evaluate their performance. Intermediate objectives required the implementation of a graph reader as well as Prim's and Kruskal's minimum spanning tree algorithms, both in CUDA and sequential. Finally, the advanced objectives were to implement Dijkstra's algorithm sequentially and with CUDA as well as perform significant evaluation of all five search algorithms. The basic objectives were completed first, followed by intermediate and the advanced to ensure that all objectives were met successfully.

B CUDA Hardware Model

CUDA enabled GPUs are manufactured by NVIDIA, starting from their G80 range of GPUs and all subsequent versions. It is worth noting that an NVIDIA 8800GTX GPU (G80 range) will be used for this project. GPUs are specifically designed to compute large amounts of data-parallel work whereas CPUs are designed for large amounts of sequential computation; "this fundamental difference is reflected in both the memory system and the way instructions are executed" (Breitbart 2008).

The architecture of a CUDA enabled GPU can be represented as a massive Single Instruction Multiple Data (SIMD) processor. Each core within a multiprocessor has to execute the same instruction at the same time but on potentially different data according to the SIMD model. This concept is known as execution hierarchy. Memory hierarchy is also present on the GPU with different categories of memory. Device memory, much like RAM on a PC, is available to all multiprocessors for read and write access but also provides the slowest form of memory to the multiprocessors. Texture and constant caches are also available to each multiprocessor but only provide read only memory. Shared memory is available to every multiprocessor with each multiprocessor having its own block of shared memory which only that multiprocessor can read and write to. Finally, each multiprocessor has access to its own registers for data storage. All of

these concepts can be seen in Figure 4. Shared memory resides on each multiprocessors chip, providing just 16 kilobytes (KB) of very fast memory for each multiprocessor. Device memory is vastly larger than shared memory, providing 768MB of storage for an 8800GTX GPU. It is favourable to use shared memory as the time required to read and write data to it is much less than that of device memory, however this is difficult.

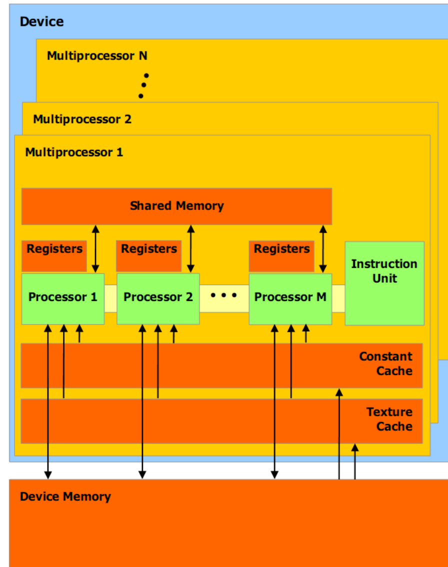


Figure 4: CUDA hardware model, demonstrating memory hierarchy and overall hardware architecture of CUDA GPUs (NVIDIA 2009*b*)

C CUDA Software Model

CUDA allows programmers to write functions known as kernels that will be executed on the GPU. A programmer does not see the hardware architecture of the GPU. Instead, they see a number of threads which are organised into blocks. Each thread is executed following the Single Program Multiple Data (SPMD) model. A programmer can define how many threads are executed for each kernel that is written. On an NVIDIA 8800GTX, the programmer can define no more than 512 threads per block. Blocks can also be ordered into grids, with each grid holding 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 like scheduling. While a programmer does not need to understand the hardware architecture, it is very useful in being able to gain the most out of CUDA.

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. 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, branch Θ and branch Ω, each branch is serialised. Therefore, Θ will execute first, followed by Ω. Clearly, avoiding branches is critical as the performance of algorithms containing branches will be degraded as thread execution is serialised between differing branches.

CUDA provides functions that allow threads to be synchronised within a block. This synchronisation process acts as a barrier within the kernel which all threads in a block must reach

before the kernel can progress any further. Blocks cannot be synchronised within a grid. Threads can be addressed using either a 1, 2, or 3 dimensional index. Additionally, blocks may be addressed by a 1 or 2 dimensional index. CUDA provides thread and block ID variables which are used to compute the addresses of threads and blocks.

Threads within a block can communicate via the shared memory. The GPU schedules blocks to execute on the device with each block executing entirely on one multiprocessor, allowing the threads in a block to utilise the shared memory on said multiprocessor. Organising threads and the data allocated to shared memory is often a complex task, with additional issues such as avoiding bank conflicts, where one blocks shared memory data overlaps another's. Clearly, 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 a kernel. A kernel can also be executed multiple times with differing thread and block sizes for each execution.

CUDA refers to the GPU as the device whereas the CPU is the host. Executing a kernel does not stop the host from executing its own code, allowing both device and host code to run simultaneously. 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. CUDA 1.1 and above does however support asynchronous memory access whilst a kernel is executing, allowing host code to proceed. The GPU used for this project only supports CUDA 1.0 and so cannot perform this function, potentially limiting obtainable speedup.

D Solution Design

From the outset, it was clear that each algorithm would perform a number of similar operations, mainly in relation to data management. As a result, the solution was designed with a modular approach in mind to avoid duplicate code. A number of software engineering practices were employed to ensure high quality algorithms were created.

Sommerville (2007, p25) talks of a layered model approach to the architecture design which "organises a system into layers, each of which provide a set of services". This project was designed with a layered approach in mind. Each layer can be seen in Figure 5.

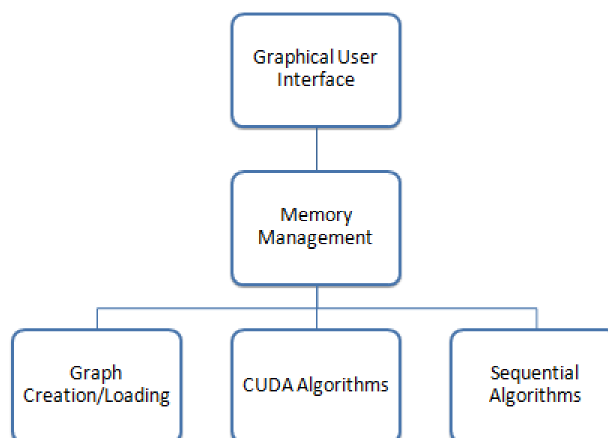


Figure 5: Layered system architecture

A "MemoryManagement" class was created to handle all operations regarding any data

throughout the project, including all the arrays necessary for each algorithm, initialisation of those arrays, memory allocation and de-allocation as well as copying data to and from the GPU. Having a single class for these operations allowed for a single instance of the class to be used by all algorithms, utilising the same data structures rather than copying the relevant data for each algorithm. At the beginning of each algorithm a simple "initialiseArrays" method can be called, setting up the arrays for first use by that algorithm. This modular approach ensured that there was no code duplication as all 5 algorithms perform very similar tasks with regards to the graph arrays and data copying to and from the device. One of the benefits of this was the ability to change the layout of data on the GPU or other aspects of the memory management very easily, only modifying a single class with the changes being reflected across all algorithms. The "MemoryManagement" class can easily be re-used in future projects, lowering development and maintenance costs.

When the project was nearing completion, there was time to create a Graphical User Interface (GUI) to enable easier demonstration of the algorithms. Having used a layered approach in the design process, it was very easy to place the GUI as the "top" layer, enabling the GUI to call upon all other layers. The GUI allows the user to create or load a graph, select algorithms to run, enter a source vertex and, where appropriate, a goal vertex. Finally, the user has the results of each algorithm displayed to them in the results tab.

A number of pre-existing components were also used in this project. The Thrust Template Library for CUDA (Hoberock & Bell 2009) was used and its use within specific algorithms is explained later in this section. Qt, a cross-platform User Interface (UI) was also used to develop the GUI for the project. Both Thrust and Qt have been developed by experts in their respective fields and provide high-quality, reliable components for use in this project.

To provide version control for the project, a Subversion repository was created. This tool proved to be immensely useful during the projects life cycle. Due to the experimental nature of the project, changes would often be made to the CUDA code which resulted in the algorithms' performance decreasing. The version control software enabled the changes to be rolled back to a previous state very easily, reducing the implementation time of the project that would have otherwise been wasted on reversing changes manually. Using Subversion also helped productivity on the project, being able to develop the project from anywhere with an internet connection provided several opportunities for development. Utilising Subversion also subverted issues with data loss as all of the code was backed up on an external server.

E BFS

This implementation of BFS using CUDA is based upon the work by Harish & Narayanan (2007). However, instead of searching a weighted, undirected graph, this implementation searches an unweighted, undirected graph. As a result, as soon as the goal node is found, the search can halt, meaning that BFS will always find the fastest route to the goal node.

For BFS, there is one thread per vertex, demonstrated in Figure 6. Using a block size of 512, the number of blocks and therefore threads are calculated via the number of vertices in the graph that is to be searched. This means that there is the potential to have 511 redundant threads that are idle during the kernel execution.

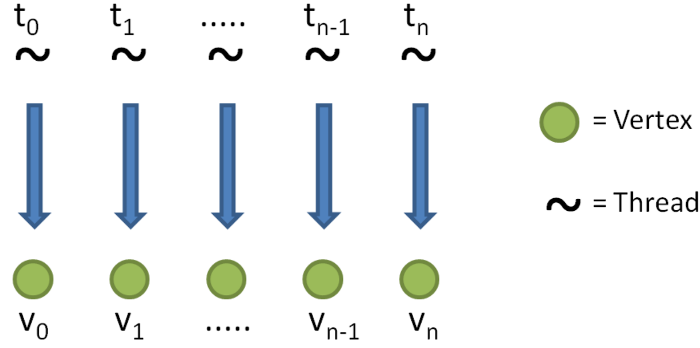


Figure 6: Example mapping of threads to vertices

As well as the graph arrays, V_a and E_a, BFS makes use of a frontier array (boolean), f_a, which holds the vertices that have been discovered so far. For this implementation, each vertex has its own thread. The thread ID (tid) is an integer that can be mapped to a vertex, as the vertex identifiers are also integers (demonstrated in Figure 6). During each iteration of the kernel, each vertex looks at its entry in f_a. If it is true, the vertex proceeds to mark its neighbours as true in f_a, thus adding them to the fringe of the search. Finally, once the neighbours have been added to f_a, the kernel checks to see if the goal vertex's entry is true in f_a. If it is, the kernel halts and the search is complete. Otherwise, the process is continually repeated until the goal vertex is found or all entries in f_a are true.

Originally, the kernel call was placed inside a loop of host code. After each kernel execution, f_a was copied back to the host to check whether the goal vertex had been found or if each entry was true. The time spent copying data to and from the device after each kernel execution was expensive so the design of the algorithm was changed to keep this check inside the kernel, eliminating the unnecessary data transfer and improving the performance of the algorithm dramatically.

F DFS

DFS proved to be a very difficult algorithm to design for use with CUDA. Originally, the algorithm began by discovering the neighbours of the source vertex and then performing DFS on each of those newly discovered neighbours. This proved to be very problematic as several arrays had to be created for each separate DFS search, using a large amount of memory. Additionally, the source vertex needed a very high degree, making it an unsuitable solution. This approach was adapted so that BFS was executed on the graph until a certain number of vertices were on the fringe of the search. From here DFS was executed on each of the fringe vertices. This however, was even more problematic as even more memory was needed than with the previous solution. Freeman (1991) states that DFS is an inherently sequential process and that researchers believe has no parallel solution. As a result, it was decided that an implementation of DFS using CUDA, that closely followed the sequential implementation, would be created. The underlying data structures were created manually so that they could perform parallel functions in an attempt to improve the CUDA algorithm over the sequential algorithm.

As with BFS, DFS requires as many threads as there are vertices in the graph. A new boolean array, vI_a, is utilised for this kernel. vI_a tracks whether a given vertex has been visited or not by the search. Additionally, a stack array, s_a, was created. New kernels were written in order to

treat this array as a standard Last In First Out (LIFO) stack by providing functions to push and pop data as well as check if the stack is empty.

The actual DFS search utilises a single thread and proceeds in a fashion very similar to sequential DFS. At the first stage of the kernel, a vertex is popped from s_a and the neighbours of said vertex are then pushed to s_a. When pushing vertices to s_a, a new kernel is invoked, utilising as many threads as there are neighbours to the original vertex. Each thread then pushes a single vertex to s_a. This process of popping from s_a and then adding neighbours is repeated until s_a is empty or the goal vertex has been found. To check if s_a is empty, a third kernel is called, utilising $|V|$ threads. Each thread checks its corresponding vertices entry in s_a to see whether it is present in the stack or not. If all threads return true, s_a is empty and DFS can then halt.

G Prim's Algorithm

Prim's algorithm provides a greedy approach to building a MST as at each step, the edge with the least weight is added to the MST. During the design of Prim's algorithm, several unanticipated challenges were encountered that added time to the implementation stage. It became apparent that underlying algorithms would need to be developed in order to develop a parallel version of Prim's algorithm. As the algorithm is an inherently sequential problem, with the edge with the least weight being added to the MST, it was necessary to develop a method of finding the least weight edge quickly. As a result, rather than creating an algorithm to find the least cost edge, an existing library, Thrust (Hoberock & Bell 2009), was used which provides a parallel algorithm for finding the smallest element in an array.

As with BFS and DFS, the number of threads is equal to $|V|$. To begin with, E_a is sorted using a parallel bubble sort so that the edges are sorted in edge weight order. The need for bubble sort was unanticipated and was therefore chosen because it provided a small implementation time. Sorting E_a assists the process of finding the least weight edge in a smaller amount of time. The bubble sort algorithm operates on each vertex in V_a, looking at each vertices neighbours in E_a. These neighbours are then sorted in order of edge weight. This results in $|V|$ bubble sorts operating in parallel.

A second kernel is then executed inside a loop of host code. Here, two new integer arrays are introduced, sE_a, and sEW_a which hold the smallest edge and the smallest edge weight respectively. Each vertex then looks at its neighbours in parallel using tid, adding the first valid edge to sE_a and sEW_a. The act of bubble sorting means that this kernel does not have to look at every neighbour, it can just add the first valid edge that it finds, rather than scanning all of them and choosing the least cost edge. After this kernel execution, Thrust's "min_element" algorithm, also inside the loop, scans through sE_a, selecting the least cost edge and adding it to the MST. This loop halts when it has executed $|V| - 1$ times, once for each edge added to the MST.

H Kruskal's Algorithm

Kruskal's algorithm was the most complicated for this project, requiring three kernels and the use of the Thrust library (Hoberock & Bell 2009). One of these three kernels is called during the execution of a different kernel, from inside that kernels code, and is not called in the host code. When implementing Kruskal's algorithm, it is important to avoid creating cycles when building the MST so a sub graph array, sG_a, is needed to keep track of which vertices belong to which sub graphs, if any at all.

sG_a is used to hold whichever sub graph a vertex belongs to. As Kruskal's algorithm chooses the least weight edge in the graph to add to the MST, sub graphs are often created. sG_a is then used to check if any particular edge can be added to the MST.

As with Prim's algorithm, each kernel and the Thrust algorithm, "min_element" is executed inside a host loop which loops $|V| - 1$ times, once for each edge added to the MST. Using a thread for each edge (u, v), the first kernel checks to see if u is true in f_a. It also calls another kernel, which checks to see if the edge (u, v) can be added to the MST by using sG_a. If the edge (u, v) can be added to the MST, it is added to sE_a and sEW_a.

After this step, Thrust's "min_element" algorithm is used to find the edge with the least weight in sE_a. The edge selected is then passed on to the next kernel which adds the edge to the MST as well as updating the sub graph array sG_a.

I Dijkstra's Algorithm

The implementation of Dijkstra's algorithm is very similar to that of BFS. However, additional arrays are required to keep track of extra data required by the algorithm.

An integer array, c_a, keeps track of the cost of getting to a vertex from the source vertex whilst a boolean flag array, fl_a, keeps track of which elements in c_a are currently being updated.

As with BFS, there is one thread per vertex with a block size of 512 where the number of blocks is calculated via the number of vertices in the graph. During the kernels execution, each vertex looks at its entry in f_a using tid. If the value is true, it marks all of its neighbouring vertices as true in f_a. From here, it marks the vertices corresponding entry in fl_a as true. The vertex then proceeds to look at those neighbouring vertices in c_a and updates the entry if the cost of getting from the current vertex to a neighbour is less than the entry in c_a. The corresponding entry in fl_a is then set to false. The purpose of fl_a is to block other threads from accessing a particular vertex in c_a at any given time if another thread is currently reading and or updating that same entry in c_a. Without this block, there may have been inconsistencies, with two threads reading the same value at the same time and updating accordingly, resulting in an incorrect value.

J Sequential Graph Searching Algorithms

To be able to analyse the results of each CUDA algorithm, sequential algorithms were created that performed the same task. Avoiding bias by creating slow sequential algorithms was important. Therefore, each sequential algorithm was designed from the ideas presented in the textbook by Cormen et al. (2001) which details how each algorithm should be implemented. Each sequential algorithm utilises a single core of the CPU, with no form of multi-threading, making them truly sequential. However, standard time sharing on the CPU means that the search will not be running constantly although its effect should be negligible.

Rather than using an adjacency matrix for the graph representation in the sequential algorithms, the same compacted adjacency lists that are used by the CUDA algorithms are utilised. The use of the compacted adjacency list was decided as to avoid converting the graph data into an adjacency matrix, potentially harming the running time of the sequential algorithm.

K Graph Creation and Graph Loading

Clearly, this project requires graphs in order to function. As a result, it was necessary to build a graph creator. Additionally, a graph reader was built in order to load pre-existing graphs.

To create a graph, the user is prompted to enter the number of vertices that they desire into the GUI, as well as the maximum and minimum degree per vertex that they would like. A boolean 2D array is then created which represents an adjacency matrix. From here, each vertex in the matrix has a number of vertices, between the minimum and maximum degree values, marked as true in the matrix. These true values represent edges between the vertices.

The adjacency matrix is then read and written to a text file. The text file takes the form of an adjacency list, with each value separated by a comma. Text files were chosen for their simplicity and the ability to read the file in any text editor. The process of creating a graph can take several seconds when building a graph with tens of thousands of vertices.

Loading a graph is more complicated than graph creation and as such, requires more time. The process begins by reading a graph file and counting the number of vertices that there are in the graph. V_a can then be allocated once $|V|$ is known. The graph file is then read again, reading values from the file in order to populate V_a. $|E|$ is also calculated in this step, allowing for E_a to be allocated in memory. The graph file is then read once more, populating E_a with all of the edge data for the graph.

L Graphical User Interface

Towards the end of the project, there was enough time to create a simple GUI to better demonstrate each algorithm. The GUI makes use of Qt, a cross-platform User Interface (UI) framework. The GUI was designed to follow a tabbed structure, with different operations such as graph creation, graph loading and viewing the results (shown in Figure 7) appearing on separate tabs. Multi-threading was utilised for the GUI, allowing each graph searching algorithm to be executed whilst enabling the GUI to be manipulated by the user.

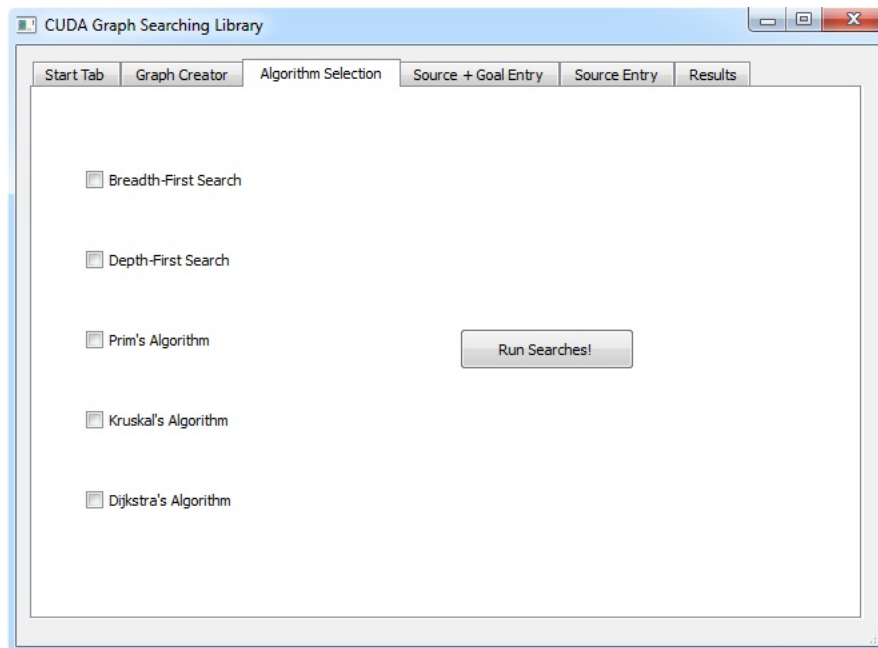


Figure 7: GUI algorithm selection tab

Human-Computer Interaction (HCI) design principles were taken into account when designing the GUI. At each stage of the GUI, feedback is provided to the user about any current op-

erations. Be it an error that they may have made, or if graph searches are currently executing, messages are displayed on the screen for the users convenience. Error messages were designed to be as helpful as possible, providing the user with a solution to the current error. For example, when creating a graph, if the user inputs an invalid number of vertices for the graph, "Please enter a valid number of vertices between 2 and 30,000 (inclusive)" is displayed, demonstrating to the user that they have made a mistake and how to fix it.

IV RESULTS

In this section, the results of the project are presented. Results stem from a number of graph searches that were conducted multiple times to record an average whilst recording the execution time, allowing for comparison between CUDA and sequential algorithms.

A 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 CUDA and sequential algorithms. Seven graphs were searched, each with an increasing number of vertices but all with a similar average degree per vertex that ranged from 11.9 to 14.02.

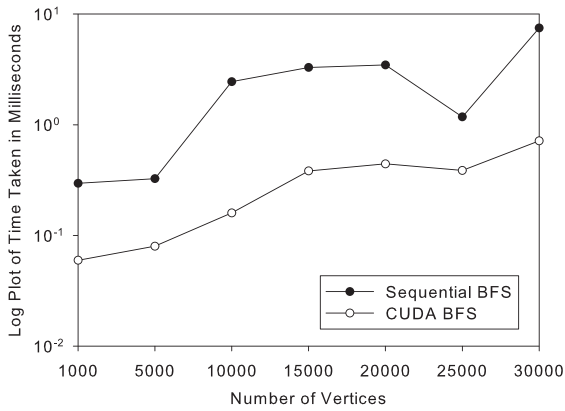
It is important to realise that both the GPU and CPU are also performing other tasks; not just graph searches. The GPU displays the output to the screen whilst the CPU handles the Operating System (OS) and other commands. The GPU often has just 80% of its global memory free when displaying output to the screen. As such, if a search is repeated on the same graph with the same search parameters, different running times may be recorded. Therefore, each search was conducted five times in order to record an average execution time. Seven graphs were created, with 1000, 5000, 10,000, 15,000, 20,000, 25,000 and 30,000 vertices. For each search conducted, the source vertex was always 0, and, where appropriate, the goal vertex was the vertex with the highest ID (e.g. the goal node when searching the 5000 vertex graph was 4999).

Due to a weakness in the graph creator explained in Section V, only graphs with a maximum of 30,000 vertices can be created. As a result, the seven graphs that were created have 5000 vertex increments so as to ensure coverage of all the possible graphs that can be created by the application. Despite this slight limitation, the results in Figure 8 are more than adequate to effectively demonstrate all five algorithms.

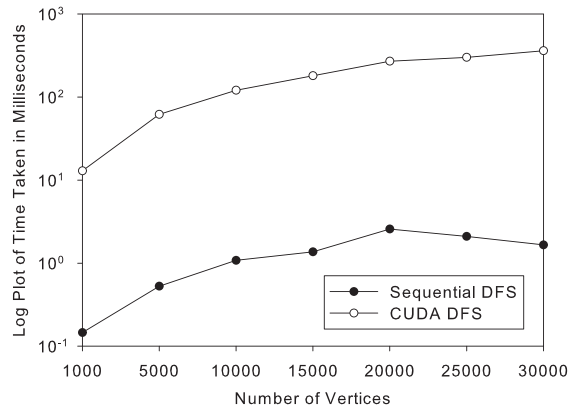
B Evaluation Setup

All CUDA and sequential experiments were conducted on a PC with 4 GB RAM and an Intel Core 2 Duo E6750 2.66GHz processor running Windows 7 64-bit. An NVIDIA 8800GTX GPU with 768MB of memory was used for all experiments. All of the CUDA algorithms were written using CUDA version 2.3 in C while the sequential algorithms were written in C++ using Visual Studio 2008. NVIDIA graphs driver version 197.45 was used for the final build of the project. Thrust version 1.2 was used for the applicable algorithms. Finally, Qt version 4.6.2 was used to create the GUI.

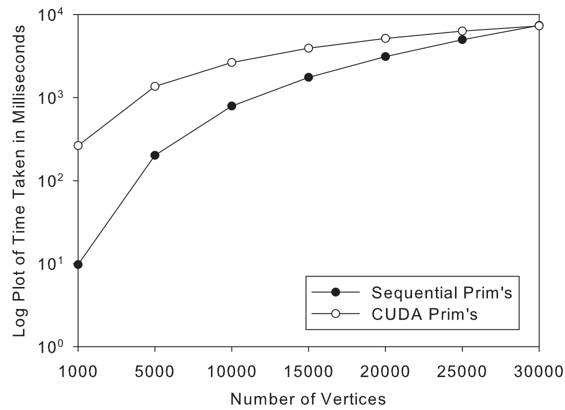
The results show that BFS and Dijkstra's algorithm are always significantly faster than their sequential counterparts while DFS, Prim's algorithm and Kruskal's algorithm are slower than their sequential equivalent. There is one exception however, Prim's algorithm is slightly faster for the 30,000 vertex graph. The results for all experiments can be seen in Figure 8.



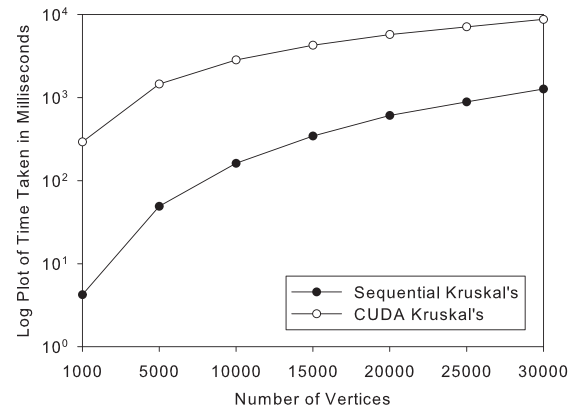
(a) CUDA and sequential BFS



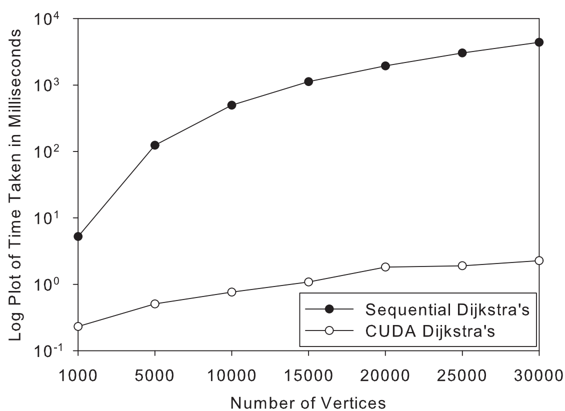
(b) CUDA and sequential DFS



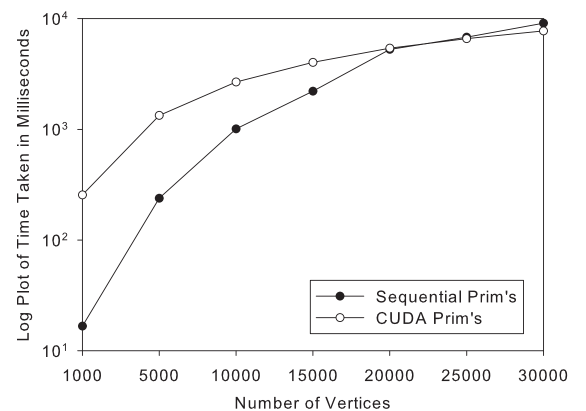
(c) CUDA and sequential Prim's algorithm



(d) CUDA and sequential Kruskal's algorithm



(e) CUDA and sequential Dijkstra's Algorithm



(f) CUDA and sequential Prim's algorithm on graphs with an increased average degree per vertex

Figure 8: Results of graph searching experiments

C Results Analysis

Figure 8a shows a comparison between CUDA and sequential BFS. The graph shows that for all seven graphs, CUDA BFS was significantly faster than sequential; providing up to 15x speedup. The results also show that whilst the overall running time for CUDA BFS generally increases with $|V|$, the increase in time is far more linear than that of the sequential algorithm which has large jumps between running times. For the 25,000 vertex graph, both CUDA and sequential BFS were faster than the previous (20,000 vertex) graph with the sequential algorithm demonstrating nearly a 3x speedup over the previous graph. This demonstrates that larger graphs do not necessarily mean that the running time of a search will increase.

As can be seen in Figure 8b, the CUDA implementation of DFS provides no speedup over the sequential version. The results show that, for the CUDA version, running times increase as the size of the graph increases. The running times between smaller graphs are quite different, with each time increasing by quite a large factor. This begins to peter out as the graph size increases and this can be seen in Figure 8b with the line beginning to level out. The results for sequential DFS are slightly more erratic than the CUDA version, with running times reaching a peak when searching the 20,000 vertex graph and steadily decreasing with subsequent graphs.

The results for Prim's algorithm, shown in Figure 8c, are interesting in that the sequential algorithm begins by being much faster than CUDA for smaller graphs. But, as graph size increases, the gap between running times decreases at an increasing rate and eventually, for the 30,000 vertex graph, CUDA is slightly faster than the sequential version. These results suggest that the CUDA implementation of Prim's algorithm may offer significant speedup over the sequential version when searching much larger graphs, perhaps with 100,000 vertices or more. Unfortunately, due to limitations in the graph creation application, graphs with more than 30,000 vertices cannot be created. A higher average degree per vertex can however be achieved. Figure 8f shows the effect of increasing the average degree per vertex. As can be seen, the rate at which the CUDA algorithm closes the gap over the sequential algorithms speedup is vastly reduced, with CUDA offering a small speedup on the 25,000 vertex graph and an increased speedup on the 30,000 vertex graph over Figure 8c.

As with Prim's algorithm, the results shown in Figure 8d for Kruskal's algorithm show that the sequential algorithm is faster than the CUDA implementation. The results shown for Prim's and Kruskal's algorithms mirror certain traits in regards to running times including that the CUDA implementation of Kruskal's algorithm appears to improve its performance over the sequential version as the size of the graphs searched increases. Much like Prim's algorithm, the results suggest that CUDA Kruskal's would indeed be faster than the sequential algorithm, but would require very large graphs to search; perhaps with at least 100,000 vertices.

When looking at the results for Dijkstra's algorithm in Figure 8e, it is clear that the CUDA implementation offers a large amount of speedup over the sequential algorithm. When searching the 30,000 vertex graph, CUDA offered a fantastic speedup over the sequential algorithm. The sequential algorithm results in a large jump between execution times from searching the 1000 vertex graph to the 5000 vertex graph with the running time steadily increasing with each new graph. The CUDA algorithm does not exhibit this jump in running time. Instead, Figure 8e shows that running times steadily increase whilst the algorithm is solving the problem, with the speedup offered increasing as the size of the graph increases. At best, the CUDA implementation of Dijkstra's algorithm shows a 1939.72x speedup over sequential when searching the 30,000 vertex graph with CUDA completing the search in 2.2723ms versus the sequential versions 4407.634ms.

V EVALUATION

Upon completion, the project implemented all of the basic, intermediate and advanced objectives that were detailed in section III. This section evaluates the weaknesses, strengths and limitations of the project.

A Weaknesses of the Project

Currently, the graph creation process can only support graphs of approximately 30,000 vertices with an average degree per vertex of between 11.9 and 14.02. This is due to limitations in the graph loading process. Larger graphs often result in the application crashing, due to errors in memory allocation. With further work, this problem could be fixed to allow for much larger graphs, perhaps with millions of vertices. Despite this, the graph creation and loading process has been improved many times and can now support much larger graphs than when it was first created. Additionally, when specifying the minimum and maximum degree per vertex, the maximum degree may be more than specified. This is due to a limitation in the graph creation process.

CUDA DFS, Prim's and Kruskal's algorithms are all slower than their sequential equivalents apart from the final search on Prim's algorithm. DFS was very challenging to design, with no apparent way of parallelising any of its operations. Unfortunately, this resulted in an algorithm that lacks any speedup when compared to normal sequential DFS. As can be seen in Figure 8c, the CUDA version of Prim's algorithm is slower than the sequential version apart from when searching the 30,000 vertex graph. However, as the size of the graph increases, the CUDA implementation of Prim's algorithm provides speedup over the sequential version. Additionally, Figure 8f provides promising results in that increasing degree of the graph can obtain further speedup for CUDA Prim's algorithm.

The use of bubble sort in the CUDA version of Prim's algorithm was an unanticipated step for implementation and so was used to provide a fast implementation time. Using bubble sort could well be replaced by a much faster sorting algorithm such as quick sort, in order to improve the running time of the sorting process.

B Strengths of the Project

The results show in Figures 8a and 8e clearly show that BFS and Dijkstra's algorithm respectively are a resounding success when using CUDA, providing substantial speedup over their sequential versions. This project set out to create fast, parallel graph searching algorithms for large graphs which is exactly what these two do. This is especially true for Dijkstra's algorithm which demonstrates that it is very effective for large graphs.

Despite achieving undesirable results for Prim's and Kruskal's algorithms, Figures 8c and 8d show that there is potential for each algorithm to provide speedup over the sequential algorithms when searching larger graphs or perhaps improving the algorithms themselves.

The project also allows highly customisable graphs to be created and saved for future use, allowing a diverse nature of graphs to be searched. Large graphs with a varying average degree per vertex can be created and loaded, with the user being able to identify each graph with a customisable name. Each graph file can also be edited manually with any text editor for even greater flexibility with the graph searching algorithms.

The addition of a GUI late in the project allows the algorithms to be demonstrated in a much more suitable environment, away from command line. The clear display of results allows the user to quickly see the performance of each search as soon as it has finished executing. Additionally, users can repeat searches on the same graph with different search parameters without having to reload the graph; something that was not possible before the implementation of the GUI.

Throughout the project, work was completed as detailed in the project plan. However, towards the end of the project, certain aspects of the implementation took a little longer than anticipated. Luckily, "backup" time had been allocated on the plan which accounted for this possibility. As a result, everything was completed on time thanks to the additional time allocated in the plan.

Due to the nature of CUDA, all of the algorithms will remain relevant as they are guaranteed to work on all future CUDA enabled GPUs from NVIDIA. Therefore, the number of machines that can benefit from these algorithms will only increase with time. The algorithms presented here are suitable for the underlying algorithms in commercial applications such as vehicle route planning due to the number of CUDA enabled machines across the globe. CUDA has already provided speedup to commercial applications such as the video conversion suite, Badaboom, highlighting the relevancy of this project.

C Limitations of the Project

Despite the success of meeting all of the project objectives, there are several limitations that if overcome, could have improved the project. However, some of these limitations were out of the control of this project.

The GPU used for the project was the first commercially available CUDA enabled GPU. As a result, some of the functions available on newer CUDA GPUs are not available. These include asynchronous data transfer from the GPU whilst a kernel is executing, asynchronous kernel execution, pinned memory and double precision floating point numbers. A new NVIDIA GTX480 for example, would provide access to these features as well as providing 480 cores over the 128 that were used for this project. The additional cores alone would substantially improve the performance of the algorithms presented here.

Using CUDA for this project has resulted in the algorithms being bound to CUDA enabled GPUs. Utilising a technology such as OpenCL would allow parallel algorithms to run on GPUs from other manufacturers such as ATI and not just NVIDIA as it is a cross platform parallel computing technology.

As mentioned in Section II, it is difficult to make use of the shared memory provided by CUDA for graph searching. If a way to utilise the shared memory for these graph searching algorithms could be found, there is the potential for even further performance increases over sequential algorithms.

Additionally, when the project began, it required a lot of research in order to understand parallel algorithms and how to design them as it was an entirely new concept to come to terms with. Design and implementation of the algorithms could have been accomplished in a shorter space of time if knowledge of parallel algorithms had been acquired before the project began. The time allowed to complete the project was also a limitation in itself. If more time was available, significant optimisation of the algorithms produced could be conducted, such as the addition of quick sort mentioned in Section V, potentially improving each algorithms performance further. Additionally, greater analysis of the results could be accomplished, if more results were collected.

VI CONCLUSION

Graph searching 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 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 CUDA enabled GPUs 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.

This project has fulfilled all of the objects that it set out to achieve. BFS, DFS, Prim's and Kruskal's minimum spanning tree algorithms and Dijkstra's algorithm were implemented, both sequentially and using CUDA. Additionally, a graph creation application was also implemented as well as the tools to load previously created graphs into memory. A simple GUI was also created to help better demonstrate the functionality of each algorithm by allowing the user to select algorithms, as well as display running times on the screen.

There are several areas in which this project could be extended with further work. Firstly, investigating the use of shared memory across all five graph searching algorithms will almost certainly improve the performance of all algorithms. Section II explained the difficulties of using shared memory with graph searching algorithms. If this could be overcome, speedup could easily be increased for each CUDA algorithm presented here. Another interesting route of investigation would be to apply the algorithms presented to a larger problem such as vertex cover or the travelling salesman problem; both of which can possess large running times which could potentially be reduced by using the CUDA algorithms described and implemented in this project. Branching in each CUDA algorithm has the potential to slow each algorithm down if each thread follows a different execution path. Discovering ways to reduce branching in each algorithm may provide further speedup.

While DFS and Kruskal's algorithms do not appear to benefit from CUDA implementation, Prim's algorithm shows a limited amount of speedup for large graphs with the potential for greater speedup for even larger graphs than those searched in Section IV. BFS and Dijkstra's algorithms were a resounding success when implemented with CUDA, providing a significant performance boost of up to 1939.72x in the best result; highlighting the potential for algorithms designed to run on the GPU to be used in both commercial and scientific environments.

The main findings of the project show that CUDA can be used to great success when utilised for appropriate problems. CUDA is a challenging technology to master. Overcoming issues such as utilising shared memory, avoiding branches, choosing appropriate data structures and designing algorithms correctly are key to success with CUDA. Some problems appear to be inherently sequential, and are difficult if not impossible to implement in parallel. This is unfortunate as there is clearly massive potential for parallel algorithms on the GPU as demonstrated by BFS and Dijkstra's algorithm.

References

Almasi, G. S. & Gottlieb, A. (1988), *Highly parallel computing*, Benjamin-Cummings Publishing Company.

Applegate, D., Cook, W., Dash, S. & Rohe, A. (2002), 'Solution of a min-max vehicle routing problem', *Informs Journal on Computing* **14**(2), 132–143.

Barney, B. (2010), 'Introduction to parallel computing'. Unpublished.
*https://computing.llnl.gov/tutorials/parallel_comp/

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

Brooks, F. (1987), 'No silver bullet essence and accidents of software engineering', *Computer* **20**, 10–19.

Buluc, A., Gilbert, J. R. & Budak, C. (2009), 'Solving path problems on the gpu', *Parallel Computing* **In Press, Corrected Proof**, –.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2001), *Introduction to Algorithms*, second edn, The MIT Press.

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

Harish, P. & Narayanan, P. J. (2007), 'Accelerating large graph algorithms on the gpu using cuda', *International Conference on High Performance Computing (HiPC)* **4873**, 197–208.

Harish, P., Vineet, V. & Narayanan, P. J. (2009), Large graph algorithms for massively multithreaded architectures, Technical report, International Institute of Information Technology Hyderabad.

Hitte, C., Lorentzen, T. D., Guyon, R., Kim, L., Cadieu, E., Parker, H. G., Quignon, P., Lowe, J. K., Gelfenbeyn, B., Andre, C., Ostrander, E. A. & Galibert, F. (2003), 'Comparison of multimap and tsp/concorde for constructing radiation hybrid maps', *J Hered* **94**, 9–13.

Hoberock, J. & Bell, N. (2009), 'Thrust: A parallel template library'. Version 1.3.
*http://www.meganewtons.com/

Hyvonen, J., Saramaki, J. & Kaski, K. (2008), 'Efficient data structures for sparse network representation', *International Journal of Computer Mathematics* **85**(8), 1219–1233.

Katz, G. J. & Kider, Jr, J. T. (2008), All-pairs shortest-paths for large graphs on the gpu, *in* 'GH '08: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware', Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, pp. 47–55.

NVIDIA (2009), 'C on the gpu'. Unpublished.
*http://developer.download.nvidia.com/presentations/2009/SIGGRAPH/asia/
2_SGA_CUDA_C_final.pdf

Sommerville, I. (2007), *Software Engineering*, eighth edn, Pearson Education Limited.

Vineet, V., Harish, P., Patidar, S. & Narayanan, P. J. (2009), Fast minimum spanning tree for large graphs on the gpu, *in* 'HPG '09: Proceedings of the Conference on High Performance Graphics 2009', ACM, New York, NY, USA, pp. 167–171.

Xavier, C. & Iyengar, S. S. (1998), *Introduction to Parallel Algorithms*, 1st edn, John Wiley & Sons Inc.