Comparison of Pathfinding Algorithms Using the GPGPU

Craig McMillan
10004794

Submitted in partial fulfilment of
the Requirements of Edinburgh Napier University
for the Degree of BSc (Hons) Games Development

School of Computing

April 28, 2014

Authorship Declaration

I, Craig McMillan, confirm that this dissertation and the work presented in it are my own achievement.

Where I have consulted the published work of others this is always clearly attributed;

Where I have quoted from the work of others the source is always given. With the exception of such quotations this dissertation is entirely my own work;

I have acknowledged all main sources of help;

If my research follows on from previous work or is part of a larger collaborative research project I have made clear exactly what was done by others and what I have contributed myself;

I have read and understand the penalties associated with Academic Misconduct.

I also confirm that I have obtained informed consent from all people I have involved in the work in this dissertation following the School's ethical guidelines.

Signed:

Date:

Matriculation no:

Data Protection Declaration

Under the 1998 Data Protection Act, The University cannot disclose your grade to an unauthorised person. However, other students benefit from studying dissertations that have their grades attached.

Please sign your name below one of the options below to state your preference.

The University may make this dissertation, with indicative grade, available to others.

The University may make this dissertation available to others, but the grade may not be disclosed.

The University may not make this dissertation available to others.

Abstract

AI algorithms are one of the last areas within a typical game engine to exploit the relatively new and emerging hardware of the GPGPU, with pathfinding algorithms being a core aspect within many games, developers need to harness the power of this extremely powerful parallel processor to create better experiences for their players.

This dissertation documents the research, design, implementation and evaluation of two pathfinding algorithms running on the GPGPU using the CUDA platform. We aim to identify which algorithms are suitable for a GPGPU implementation and attempt to identify what factors effect the parallelization of these algorithms.

We look at a number of parallel systems and GPGPU frameworks along with a number of pathfinding algorithms which have been used within games. We investigate previous attempts at parallelizing these algorithms using the GPGPU and identify that collaborative diffusion and Dijkstra may be suitable for a GPGPU implementation based upon this work.

A detailed analysis of the CUDA architecture is provided outlining the execution and syntax of CUDA kernel functions and the advantages of using the current Kepler architecture with the introduction of dynamic parallelism. We also discuss the maps that each algorithm will be tested against and the metrics which will be recorded.

The implementation provides a detailed description of the sequential and parallel implementations of the algorithms and highlights the challenges encountered when parallelizing the algorithms.

Analysing the results we were able to draw a number of conclusions; We found that both collaborative diffusion and Dijkstra can be parallelized using the GPGPU resulting in speed ups over their sequential versions and that the degree to which terrain costs vary within the map can increase the time taken to find a path within the parallel versions of both algorithms.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank my supervisor, Professor Emma Hart, for all her support, guidance, advice and patience throughout this project and my final year at Edinburgh Napier University.

I would also like to thank my second marker, Dr Neil Urquhart for his encouragement and support.

The NVIDIA Corporation also agreed to donate a Tesla K40 to support the project, I would like to thank them for this as without the generous donation certain aspects of the project would not have been possible.

I would also like to thank Dr Kevin Chalmers for his continued support throughout my time at Edinburgh Napier University and for his patience and willingness to take the time to answer every question I have had over the years.

Finally I would like to thank the School of Computing for providing the equipment, environment and knowledge to enable me to produce this body of work and make the most from my education.

Chapter 1 | Introduction

From the outset of the project the aim has always been to identify suitable pathfinding algorithms which can successfully be run in parallel on the GPGPU and to gain an understanding of the factors which can effect the parallelization of the algorithms. The body of work documented below describes in detail the process involved in achieving this.

1.1 Motivation

With physics and graphics having successfully taken advantage of the GPU to improve their performance, AI algorithms are one of the last areas within a typical game engine to exploit the relatively new and emerging hardware of the GPGPU.

Forming a relatively small part within the field of artificial intelligence yet key to the success of any game with computer controlled characters are path finding algorithms. Developers need algorithms which can find intelligent and realistic paths through complex terrain in order to create fun and immersive experiences for their players.

With the Graphics Processing Unit (GPU) becoming increasingly more common being found in everything from desktop computers, laptops, games consoles and mobile devices and the introduction of hardware to support general purpose computations (GPGPU). Utilising the immense parallel processing power of the GPGPU to perform pathfinding could allow developers to not only find realistic paths quicker but free up the Central Processing Unit (CPU) for other more complex AI operations leading to better games in the future.

1.2 Aim & Objectives

The aim of this project is to implement, test and evaluate a piece of software which compares multiple pathfinding algorithms running in parallel using the GPGPU. The following objectives have been identified in order to achieve this.

- Learn to use a GPGPU framework.
- Research what work has already been carried out in the area.

- Identify possible algorithms which could be suited to a GPGPU implementation.
- Design a number of tests to effectively compare and understand the algorithms
- Implement sequential and parallel versions of the algorithms
- Test the algorithms and gather data.
- Analyse the data and gain an understanding of the algorithms
- Identify areas for improvement.

1.3 Research Questions

Over the course of the project there are three main questions which will be considered.

1. Identify which pathfinding algorithms may be suited to a GPGPU implementation?
2. How do the algorithms compare in terms of performance and quality of path?
3. What factors effect the parallelization of the algorithms?

1.4 Scope

1.4.1 Deliverables

The items of the project to be delivered are: One traditional pathfinding algorithm implemented on both the CPU and the GPGPU; One other algorithm implemented on both the CPU and GPGPU; The results and analysis of the algorithms.

1.4.2 Boundaries

Only the algorithms themselves are being tested and no graphical simulations are required in order to achieve this. The final deliverable is not a game or game engine and no user interaction is required.

1.5 Constraints

There are a number of limiting factors which could affect the success of the project; The most costly among these is time. Numerous factors can effect the time constraint such as the time required to learn a GPGPU framework and the time needed to implement both CPU and GPGPU versions of the selected algorithms, particularly overcoming problems with the GPGPU version as there is very little previous work been done in this area.

1.6 Chapter Outlines

- Background - outlines the research into the topic.
- Methodology - Defines the methods used during the implementation.
- Implementation - Details the specifics of the sequential and parallel implementations.
- Results - Presents the results from the tests carried out.
- Conclusion - Discusses the results obtained, reflects on the aims and objectives and outlines future work to be carried out.

Chapter 2 | Background

In this chapter we will explore why the GPGPU is becoming an increasingly viable option for computation over the CPU. We will also draw a conclusion as to which pathfinding algorithms will be best suited to a parallel implementation and provide an overview of the current GPGPU APIs available to developers.

2.1 Sequential vs Parallel

Between 1986 and 2002 the speed of Central Processing Units (CPU's) increased by 50% per year (Herlighy & Shavit 2008) The reason for this increase is due to Moore's law (Moore 2013a) which states that the numbers of transistors per silicon chip will double every year. However manufactures have now reached the stage where they can no longer fit more transistors on a chip due to the heat generated by the large clock speeds as the CPU's cannot be cooled fast enough. Due to this, manufacturers have turned to putting multiple processors on a single integrated circuit.

Today the majority of devices including mobile phones, notebooks, tablets and home PCs now have at least 2 cores, with high end super computers such as Titan (Oak Ridge National Laboratory 2013) having 16-core AMD Opteron CPU's. As a result software developers are now moving towards parallel systems in order to achieve an increase in performance as we can no longer rely on CPU's becoming faster each year. However the multicore nature of modern CPU's lend themselves to particular types of parallel systems such as data parallelism and SIMD operations which developers can exploit.

2.1.1 Data Parallelism

Typically situations arise when a developer is presented with a large set of data which requires the same operations to be performed on each of the data elements. In a traditional sequential system the developer would have to loop over each of the data elements and perform the required operation until all the elements have been processed, however through the use of data parallelism these operations can be performed much faster in parallel.

Data parallelism works by distributing the data across multiple processors or processing cores and performing the computations on the data elements simultaneously. By doing so it is possible to significantly reduce the time

taken to complete calculations on the entire data set. For example a quad core system processing 64,000 data elements could theoretically see a speed up of a factor of four when the work is split up across the four processors. However the actual speed up will be affected by additional factors.

2.1.2 SIMD(Single Instruction Multiple Data)

In 1966 Michael J. Flynn proposed a classification to categorise the architectures of computer systems known as Flynn's Taxonomy (Flynn 1969). There are four categories' that describe the various architectures; one of these is SIMD which stands for Single Instruction Multiple Data. SIMD works on the principle of data parallelism where each processing unit (PU) performs the same instruction on a distinct chunk of data from the data pool simultaneously.

SIMD is supported by hardware which includes instruction sets such as Intel's Streaming SIMD Extensions (SSE) which work by performing the same instruction on data stored in 128-bit wide registers. This is also known as short-vector SIMD due to the limit on the size of the dataset and is an instruction level optimization. However GPU's implement wide-vector SIMD architectures which allow data to be stored in 256-bit wide registers allowing much larger datasets to be processed and whole algorithms to be parallelised.

Figure 2.1 shows the flow of data to the PUs to be processed within a single instruction.

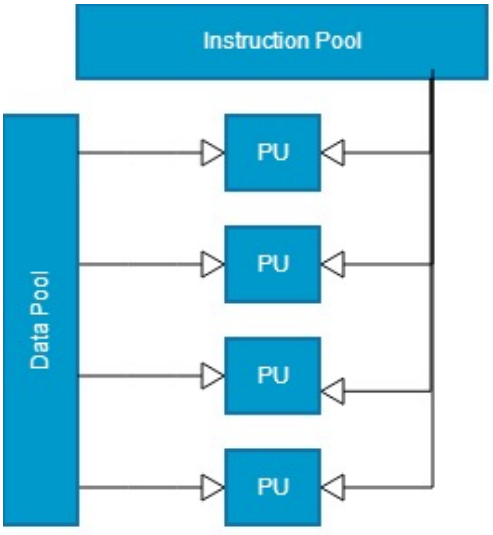

Figure 2.1: Single Instruction Multiple Data SIMD Diagram

2.2 General Purpose Processing on Graphics Processing Units

GPU hardware has become increasingly more common in modern computing systems, found in almost everything from high end enthusiast desktop computers, games consoles and even mobile devices.

By their very nature GPUs are massively parallel and follow the SIMD architecture which allows the processing of hundreds or thousands of threads simultaneously. NVidia Tesla GPUs such as the K20 which is used in the Titan super computer (Oak Ridge National Laboratory 2013) have 2496 programmable cores. This is a massive increase over the 2 or 4 core CPUs that are found in the majority of machines and as such developers are able to access increasingly more powerful computation for their parallel applications.

There are two main APIs which are used for developing GPGPU applications; these will be described in the following sections.

2.2.1 CUDA

The Compute Unified Device Architecture (CUDA) platform is a general purpose computing API that has been designed for performing tasks that would traditionally be executed on the CPU, on the GPU. CUDA was first released in 2007 by NVidia (NVidia 2013a) and is a proprietary platform that requires users to own an NVidia graphics card with CUDA capabilities.

Users can write code which can be run on the GPU using C/C++ or FORTRAN; In order to do this users are able to download the CUDA Development Tool Kit (NVidia 2013a) for free from the NVidia website. CUDA applications consist of host code which is executed on the CPU and device code, also known as kernel code which is executed on the GPU. The CUDA toolkit contains a compiler (NVCC 2013) which works by separating host code from device code during compilation, device code is then compiled by NVCC whilst the host code is sent to the configured C/C++ compiler.

The host system communicates with the device by copying over a set of data and then giving the device a task to perform on the data. This is typically done by splitting the work into multiple threads and each thread then performing the same operation on different parts of the dataset. Each thread must execute the same instructions as one another; however it is possible for threads to diverge. This will occur if there is a conditional statement such as an ‘If’, ‘While’ or ‘for’ when this happens the threads can no longer be run in parallel and their execution is serialized. This can have a large impact on

performance and as such should be avoided as much as possible.

2.2.2 OpenCL

The Open Computing Language or OpenCL was first developed by Apple in 2008 as a cross platform solution for programming heterogeneous platforms such as CPUs, GPUs and digital signal processors.

The standard is now maintained by the Khronos Group (Khronos Group, 2013), who are also responsible for maintaining the OpenGL standard. Although sharing many of the same features as CUDA, OpenCL aims to go further by providing a unified framework for programming multiple kinds of processors and not just GPUs.

OpenCL kernel code is derived from the C99 standard (OpenCL 2013) with additional keyword extensions that allow code to be programmed for parallelism.

2.3 Pathfinding Algorithms in Games

Many game developers are now making use of GPGPU technology to offload typical CPU activities such as physics and animation. In turn this is allowing more time for processing AI algorithms and as such in a typical game 20% to 50% of the CPU processing time each frame is available to AI developers (Millington & Funge 2009a). However many AI algorithms still take a very long time to run, for example a typical pathfinding algorithm can take tens of milliseconds for each character and in a game with a 100 characters this is a lot longer than the available time each frame.

Due to the advances in CPU and GPU hardware developers are now looking at moving the processing of certain AI algorithms to a parallel implementation to try and increase the performance of the algorithms.

In many games a key part to the AI is pathfinding algorithms that allow characters to safely find a path from one location to another while avoiding collisions with obstacles or other characters. This section will give an overview of some different types of pathfinding algorithms and the different environments and scenarios they are typically presented with in a game.

2.3.1 Dijkstra's Algorithm

Dijkstra's algorithm was first presented by Edsger Dijkstra in 1959 as a path finding algorithm that solves the single-source shortest path problem for a non-negative weighted graph (Dijkstra 1959). Nodes in the graph can represent either a 2-dimensional grid or points in 3D space. In many examples it is often used to find paths between connected cities where each city is represented as a node in the graph and each edge is a road with an associated cost. For a given source node Dijkstra is able to find the lowest cost path between that node and every other node in the graph.

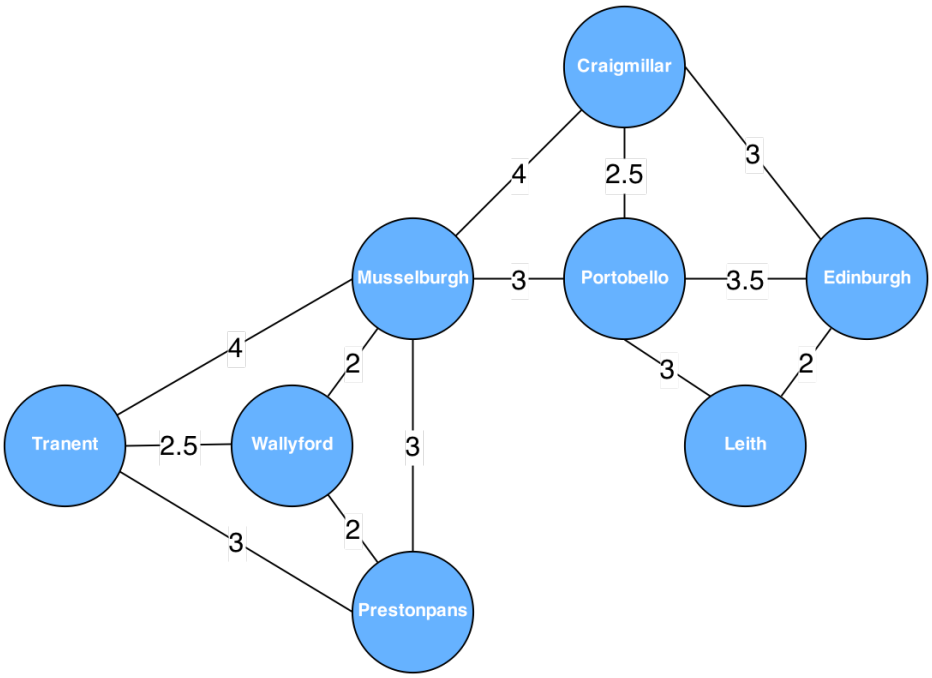

Figure 2.2: Non-negative weighted graph of connected cities

The algorithm itself works by assigning a distance value to each node in the graph, setting the value for our source node to zero and every other node to infinity. Initially we mark all nodes as unvisited and store them in a priority queue called the unvisited set. For our current source node we look at all of its unvisited neighbours and calculate the distance from our current node to the neighbouring node. For example, in figure 2.2 above if our current node was Tranent and we were looking at neighbour Musselburgh our distance would be 4. If this distance is less than our previous shortest path we add our current node to a shortest path tree which we can later use to backtrack a path to the source node. Once we have looked at all the neighbouring nodes

we remove our current node from the unvisited set and it will not be checked again. Next we set our current node to the node with the smallest distance in the unvisited set and repeat the process with this nodes neighbours until all the nodes have been visited. At this point our shortest path graph will now contain all of the lowest cost paths from our source node to every other node, we can then perform a depth-first search over this graph to find a path from any node to the source.

2.3.2 A Star Pathfinding (A*)

The A* search algorithm aims to find the best path from a starting state to a given goal state by traversing along the edges of a directed non-negative weighted graph. A* was first described in 1968 by Hart, Nilsson and Raphael as an extension to Dijkstra's algorithm (Hart, Nilsson & Raphael 1968).

Nodes in the graph can represent anything from a 2-dimensional grid to points in 3D space. A* works by expanding the nodes surrounding the start state based on a heuristic which is an estimate of how far away the goal node is from the current state. A* has two main parts that allow it to find the best path in an efficient manner, these are the open list and the closed list. The open list contains a collection of nodes that are still to be checked by the searching process and the closed list contains all the nodes that have already been checked. Nodes are also given scores or costs which help the searching algorithm to find the optimal path from the start and goal states, each node has a G, H and F cost. The G cost of a node represents the cost of getting to this node from the starting state and the H cost represents the cost from the current node to the goal state. The H value is also known as the heuristic which can be calculated in a number of ways and can determine how the algorithm is likely to behave. The F cost is a combination of the G and H costs which is calculated by simply adding the G and H values together.

The F cost is used to guide the search toward the goal node, at each iteration the node with the lowest F cost is removed from the open list and its neighbours searched. This search process involves checking if the node is not passable or is on the closed list, if it is we ignore it, otherwise we check to see if it is on the open list. If it is not on the open list we calculate its F, G and H scores and add it to the open list. If it is already on the open list we check to see if the G cost to this node is lower than our current G cost, if it is then this node provides a better path than our current one. This process continues until we remove the goal node from the open list or the open list becomes empty in which case there is no path from the start to the goal.

The chosen heuristic is a very important part of A* in order to determine the efficiency and behaviour of the algorithm. The chosen heuristic must follow a strict set of rules and cannot just be a randomly chosen value; it must be both consistent and admissible. In order for the value to be considered consistent when it is calculated for a given node N it must be non-decreasing as we move along any given path towards the goal state. For the value to also be considered admissible the heuristic function must never over estimate the cost of reaching the goal state from the current node as this can greatly affect the efficiency of the algorithm and the best path will not be found. (Millington & Funge 2009b).

There are two main methods that are typically used for calculating heuristics for A* these are Euclidean distance and Manhattan Distance. Euclidean Distance works by using Pythagorean Theorem to get the distance between the two points, this is one of the most commonly used heuristics as it can never overestimate the distance between the points as the straight line distance is always the shortest between two points. (Millington & Funge 2009c).

Manhattan distance works by finding the distance between two points if a grid like path is followed. This is done by summing the differences between their corresponding components (Manhattan Distance Metric 2013).

2.3.3 Collaborative Diffusion

Traditionally in a game each agent will have its own pathfinding routine that will be called any time an agent needs to find a path to a specific goal state. Typically in the majority of games this will call the A* algorithm, this works well in most cases but as the number of agents increases the time spent pathfinding also increases which can have a detrimental effect on the performance of the game.

To combat this collaborative diffusion works on the idea of antiobjects, an antiobject is a type of object that appears to do the opposite of what we would generally think the object would be doing (Repenning 2006). For example in the game Pac-Man (Pac-Man 1980) the objective of each ghost is to find Pac-Man. Using traditional pathfinding like A*, each ghost would run its own pathfinding algorithm every time it needed a new path. From an object orientation stand point this makes sense to many programmers that the responsibility of finding a path would be down to each agent, however by removing the computation from the objects that typically appear to do the

search, it allows the computation to be redistributed in a parallel fashion to the objects that define the space to be searched instead.

Diffusion is a gradual process in which physical and conceptual matter, such as molecules, heat, light, gas, sound and ideas are spread over an N-dimensional physical or conceptual space over time (Repenning 2006), within collaborative diffusion this idea is used to transfer the scent of a goal state throughout the world in which pathfinding will be taking place. This diffusion value is calculated using the following equation.

$$D = \frac{1}{n}\sum_{i=0}^{n} ai$$

where:
D = diffusion value
n = number of neighbours
a = diffusion value of neighbour

Typically the world is organised as a grid representing either a 2D or 3D environment and each neighbour is defined according to the Moore neighbourhood (Moore 2013b) which comprises of the eight cells surrounding a central cell.

Each goal state is given an initially high diffusion value which is used as the starting state as shown by the green tile in Figure 2.3 below.

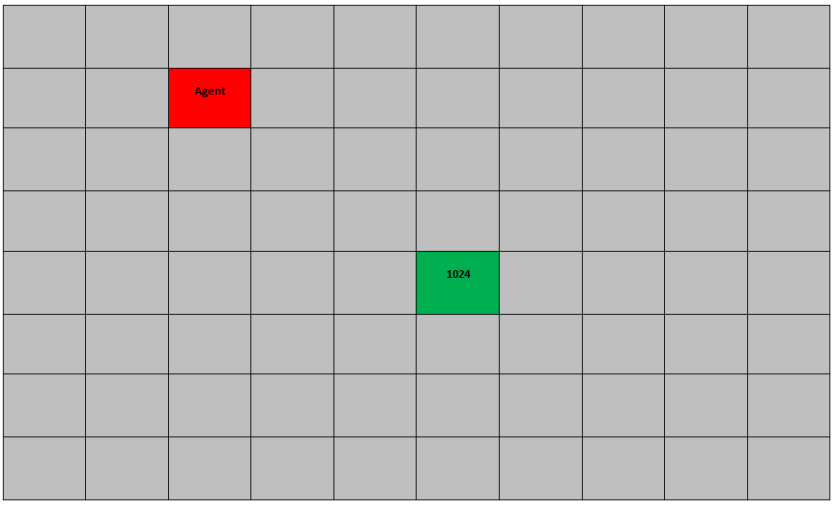

Figure 2.3: Starting state of goal and agent

During the first iteration the diffusion value for all the tiles are calculated using the diffusion equation. After this iteration the tiles surrounding the goal state now have a larger diffusion value than they had initially and the 'scent' of the goal state is beginning to move toward the agent as shown in Figure 2.4 below.

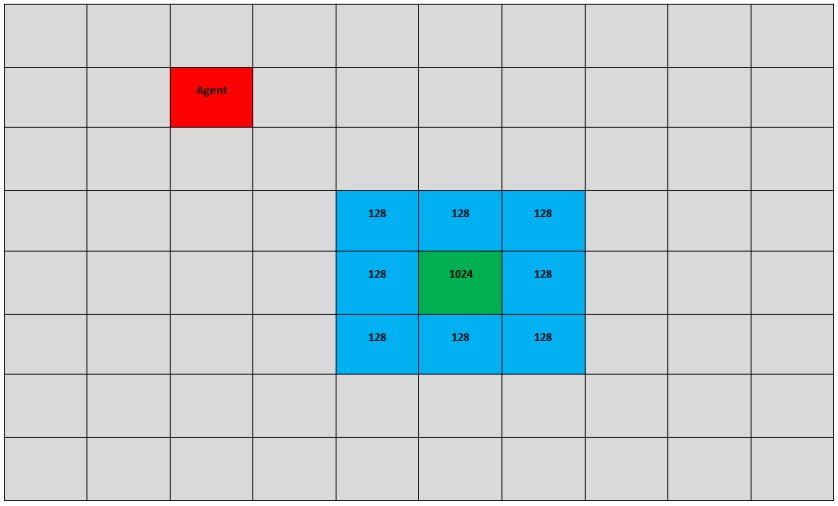

Figure 2.4: Grid after one iteration

This process continues again for all of the tiles in the world during the next iteration, as the diffusion value is calculated again for all the tiles it begins to spread out further from the goal state expanding the 'scent' of the goal further across the grid and towards the agent. During this iteration the diffusion value of the tiles surrounding the goal state becomes larger making the goal state appear more attractive to an agent. This second iteration is shown in Figure 2.5 below.

At each iteration the diffusion process repeats expanding further and further outwards across the tiles. Eventually the 'scent' produced by the tiles will reach an agent at which point a path can be found to the goal state.

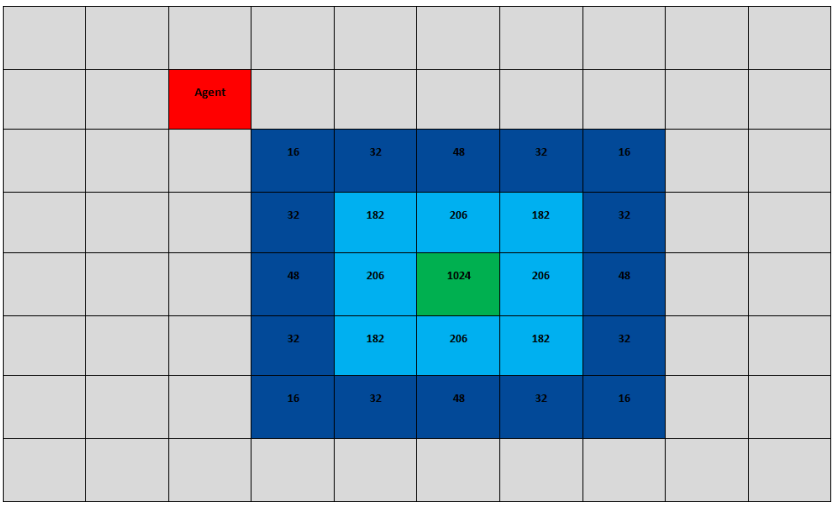

Figure 2.5: Grid after two iterations

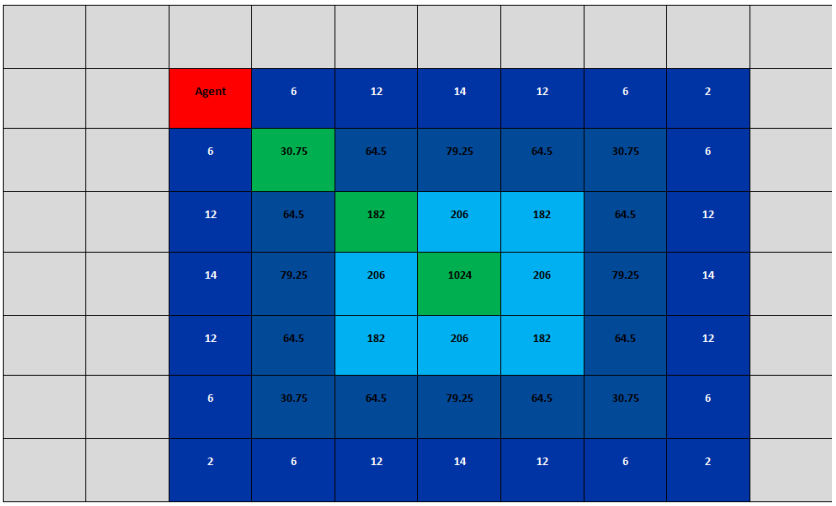

Figure 2.6: Path can be found after third iteration

Once the 'scent' has reached the agent, it is possible for a path to the goal state to be found, unlike pathfinding algorithms like A* which use a back tracking algorithm to find a path from the starting state to the goal state all the agent simply has to do is look up the diffusion values of the tiles neighbouring the tile it is currently on and move to whichever one has the highest diffusion value.

As shown in Figure 2.6 during the third iteration our agent is able to pick up the 'scent' of the goal state, by simply moving to the neighbour with the

highest diffusion value our agent is able to find the quickest path to the goal as represented by the green tiles.

Collaborative diffusion allows agents to collaborate together to complete objectives in a very simple manner. In his paper collaborative diffusion: programming antiobjects, Alexander Repenning states that there are four different types of agents that allow collaboration to occur:

1. **Goal Agents:** Goal agents are pursued by other agents and can be either static or mobile.

2. **Pursuer Agents:** A pursuer agents is an agent that is interested in one or more of the goal agents. Multiple pursuer agents may be interested in the same goal and may collaborate or compete with one another.

3. **Path Environment Agents:** A path environment agent allows pursuer agents to move towards a goal agent and participates computationally in the diffusion process. In a simple arcade game this could be a floor tile in a maze.

4. **Obstacle Environment Agents:** Work in a similar manner to path environment agents but rather than helping an agent to reach the goal they interfere with the agents attempt to reach a goal. This could be represented by a wall or obstacle in a room.

Making use of these different types of agents Repenning noticed that agents would appear to collaborate with one another to reach a goal. He also states that the collaboration between the agents is an emergent behaviour as no code is explicitly responsible for orchestrating a group of agents but is instead a result of the diffusion process and the way in which pursuer agents find a path to goal agents through the path environment agents. Another interesting observation by Repenning is that as the number of agents is increased the time taken to perform the pathfinding remains constant (Repenning 2006).

By its very nature collaborative diffusion lends itself very well to a data parallel implementation on the GPU, as the GPU works by splitting the work across a number of threads and having each thread perform the same task it should be more than possible to split the work of the tiles across a number of threads as each tile performs the same diffusion calculation.

2.4 Existing Work

In recent years as GPU technology has become more common place, many traditionally CPU based applications like physics and medical simulations have successfully made the transition to GPU implementations, however there has been very little research done in the area of game AI and pathfinding algorithms. The following papers present us with some background on GPU pathfinding and multi-agent systems.

Timothy Johnson and John Rankin from La Trobe University looked at parallel agent systems on a GPU for use with simulations and games. They looked at creating a system where multiple agents are able to interact with other agents and collaborate and negotiate with each other in order to achieve their goals. The main section of the paper outlines some of the problems that they encountered during the implementation of the system; one of the main issues they came across was how to handle branch divergence (Johnson & Rankin 2012). Branch divergence occurs when there is a conditional statement such as an 'If', 'for' or 'while' statement, when this occurs all the threads on the GPU must execute the divergent piece of code which can result in a massive waste of processing time, as traditional multi-agent systems that require collaboration are highly divergent this can have a very large overhead when running on the GPU.

Johnson and Rankin tested two implementations of the system, one intended to run on the CPU using C++ and OpenGL and one running on the GPU using C++, OpenGL and OpenCL. They tested the system for 1000, 5000, 10,000 and 25,000 agents. They found that in every case the sequential CPU version drastically outperformed the GPU version, however the GPU version scales a lot better than the CPU version, for 1000 agents to 5000 agents they found that the GPU required 48% more processing time to handle the extra agents compared to 498% extra processing time needed by the CPU. The code that was used during the GPU implementation had not been optimized specifically for the GPU and as such Johnson and Rankin conclude that had optimizations to handle the branch divergence been implemented and as GPU hardware continues to become more powerful the performance difference between the CPU and GPU will become less (Johnson & Rankin 2012).

Another interesting paper that looked at Multi-Agent Pathfinding over Real-World Data using CUDA presented an implementation of A* that uses road maps converted from real world data to find paths for multiple agents across city streets (McNally 2010). McNally used an optimized version of A* specif-

ically designed to be run on the GPU using the CUDA API and tested the system on both the CPU and GPU for a range of different sizes of graphs and agents. It was found that for a low number of agents the CPU significantly outperformed the GPU; however as the number of agent's increases the time taken to perform pathfinding on the GPU decreases and for approximately 1024 agents the GPU performs similarly to the CPU (McNally 2010). McNally concluded that the initially poor performance of the GPU was down to the highly divergent nature of the A* algorithm which results in threads being run sequentially when a conditional statement is met, however similarly to Johnson and Rankin it was found that the GPU scales a lot better for a large number of agents compared to the CPU and that as GPU hardware advances the performance difference between the CPU and GPU will become less.

Using the latest GPU architecture released by Nvidia (Kepler 2013) (Ortega-Arranz, Torres, Llanos & Gonzalez-Escribano 2013) present a parallel implementation of Dijkstras algorithm in their paper 'A New GPU-based Approach to the Shortest Path Problem' (Ortega-Arranz et al. 2013). They provide an implementation in which they have parallelized the internal operations of the algorithm through exploiting the inherent parallelism of the internal loops. They identify two loops within the algorithm which they refer to as the *outer loop* and the *inner loop*. The *outer loop* is responsible for selecting a new current node while the *inner loop* checks each of the current nodes neighbours and calculates the new distance values. (Ortega-Arranz et al. 2013) were able to parallelize the *outer loop* by selecting multiple nodes simultaneously that can be settled in parallel without affecting the algorithm correctness and the *inner loop* by looking at each neighbour of the node in parallel (Ortega-Arranz et al. 2013).

(Ortega-Arranz et al. 2013) compared their parallel implementation to an equivalent sequential implementation across a range of different graph sizes. They initially identified that branch divergence may cause a possible slow down to the algorithm when run on the GPU, however using the CUDA Visual Profiler (Nvidia Visual Profiler 2013) they were able to show that threads were only being serialized between 0.5% and 6% of the time, concluding that branching was having very little effect on performance. As a result they were able to achieve a 220x speed up with their GPU implementation compared to the CPU sequential implementation (Ortega-Arranz et al. 2013). This provides interesting evidence in support of McNally, Johnson and Rankin who concluded that as GPU architectures and hardware improves so may GPU performance for divergent algorithms, with the current Kepler architecture

this may be the case.

To the best of our knowledge no research has been undertaken into a parallel GPGPU implementation of collaborative diffusion to perform pathfinding for multiple-agents. However the book CUDA by Example by Jason Sanders and Edward Kandrot (Sanders & Kandrot 2010b) provides a GPU implementation of simulating heat transfer. In the example heaters are placed across a 2D grid and each heater is given an initially high temperature value, at each time step the heat is diffused to the neighbouring tiles using the following equation:

$$\tau new = \tau old + \sum_{NEIGHBOURS} \kappa.\Big(\tau neighbour - \tau old\Big)$$

where:
τnew = new temperature
τold = old temperature
κ = the rate at which heat flows through the system
$\tau neighbour$ = temperature of neighbouring tiles

This equation is very similar to that presented in section 2.3.3 where essentially all that is happening is we are taking the average of the neighbouring tiles and setting that as the temperature value. In the example Sanders and Kandrot discuss some performance considerations with how the grid is stored and provide example code using different types of GPU memory to store the grid (Sanders & Kandrot 2010b). From this example Sanders and Kandrot have shown that diffusion on the GPU is a feasible option and with some modifications the implementation they have presented could be used to perform pathfinding using the collaborative diffusion model.

The results shown from the papers described in this section help to provide evidence that using the GPU to perform multi-agent pathfinding could be a viable option. However the types of algorithms need to be considered carefully, as highly divergent algorithms can have an impact on performance when executed on the GPU. Sanders and Kandrot have also provided evidence that diffusion based algorithms are suited to a GPU implementation, while (Ortega-Arranz et al. 2013) has shown that unlike A*, Dijkstra can be successfully parallelized. As such with these points in mind the following questions will form the basis of investigation for the remainder of the document.

1. How does a GPU implementation of collaborative diffusion compare to a GPU implementation of Dijkstra?

2. What factors effect the parallelization of both algorithms?

Chapter 3 | Methodology

This section looks at the methods which will be used during the implementation. We will provide a detailed overview of the CUDA architecture and look at how the graphs used for testing the algorithms will be represented along with the metrics used to compare the algorithms.

3.1 CUDA

Of the two GPU API's CUDA 5.5 has been chosen for the GPU implementations of the algorithms. Although it is not cross-platform like OpenCL, it is the more mature of the two API's and provides a wealth of resources and documentation for developers through the NVidia CUDA zone. NVidia have also kindly agreed to donate a Tesla K40 GPU to support the project through their academic hardware donation program.

When writing CUDA applications it is common to refer to the CPU and system's memory as the host and the GPU and its memory as the device. Code is executed on the device using kernel functions which are sections of an application that can be run by thousands, sometimes millions of threads simultaneously. In the following sections we will review the CUDA architecture and some of the important features that it provides as well as some of the debugging and analysis tools provided by NVida.

3.1.1 Architecture

At the time of writing the most recent architecture released by Nvidia is the Kepler GK110. Claimed to be the fastest, most efficient HPC architecture ever built (NVidia 2013b) and the architecture used in the testing hardware. Figure 3.1 shows the full chip block diagram for the Kepler GK110.

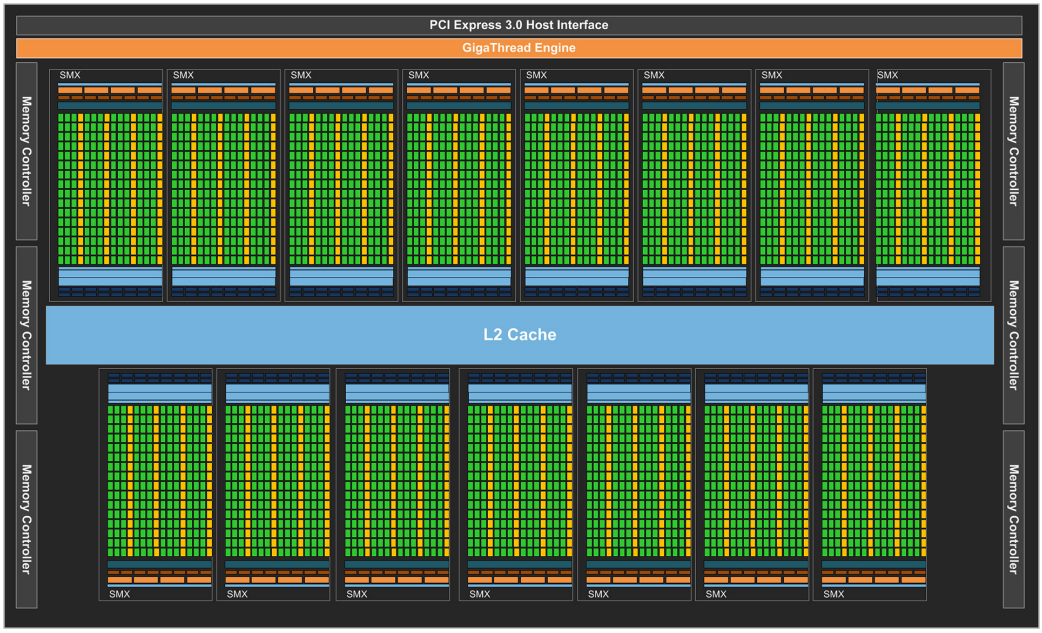

Figure 3.1: Kepler GK110 Full chip block diagram (NVidia 2013b)

Looking at the overall Kepler architecture the main feature we can see are the 15 streaming multiprocessors (SMX units), represented by the green columns which will be covered in more detail shortly. Alongside the SMX units we have 6 memory controllers which control read and write operations to the device memory (DRAM) attached to the SMXs through the L2 Cache. The remaining blocks are the Host Interface connecting the device to the host through the PCI-Express bus enabling application control and data transfer between host and device and the GigaThread Engine which is responsible for thread scheduling at the hardware level.

Figure 3.2: Full chip block diagram of Kepler SMX unit (NVidia 2013b)

Figure 3.2 shows the block diagram of an individual SMX unit in the Kepler architecture. The most prominent aspect of the SMX units are the 192 single-precision CUDA cores split into four 48 core blocks each with its own warp scheduler and two dispatch units. Alongside the CUDA cores are 64 double-precision units to handle double precision arithmetic, 32 load/store units to read/write from memory and 32 special function units to handle transcendental operations such as sine, cosine and square root. The instruction cache, register file, shared memory/L1 cache and the read-only data cache are shared across all the CUDA cores. Altogether a full Kepler GK110 GPU consists of 2880 CUDA cores.

3.1.2 Device Memory

Within the CUDA API there are different types of memory available to developers. Host memory refers to the memory attached to the CPU while device memory is attached to the GPU and accessed directly through the dedicated memory controllers. Within this section we will briefly look at the four main types of device memory.

Global Memory
Global memory refers to a block of memory that is accessible by all threads that are being executed and remains persistent during the kernel execution while allowing both read and write access from threads. Global memory is relatively slow in comparison to the other types of memory available on the GPU and has a latency of 400-800 cycles (Zaharan 2012). However the peak bandwidth is extremely high, for example in the Tesla K40 donated by NVidia the memory bandwidth is 288 GB/sec compared with approximately 50 GB/sec for a high end CPU. Global memory is accessed through CUDA kernels using device pointers similar to C/C++ pointers. Typically most cards have upwards of 1GB of global memory.

Shared Memory
Shared memory is a type of on chip scratchpad memory used for fast data interchange between threads running on the same SMX. In the Kepler architecture each SMX has 64K of on chip memory which can be used as either 48K shared memory / 16K L1 cache or 48K L1 cache / 16K shared memory depending on the needs of the application. Using shared memory can allow communication between threads on an SMX allowing threads to collaborate on computations which can be useful for many applications.

Constant Memory
Constant memory is an optimized form of read only memory with the ability to broadcast data to multiple threads in a single operation. Constant memory is most commonly used to hold data which will not change during a kernels execution. Developers are able to access 64K of constant memory within their applications. The advantage of constant memory is that it can conserve bandwidth when compared to global memory which can result in a speedup for applications which are able to make effective use of the available 64K.

Texture Memory
Texture memory within the GPU was originally designed for graphics applications, however when used correctly it can be exploited by CUDA developers

to gain additional speedups within their applications. Texture memory is read-only and is cached on chip which allows a high bandwidth as it reduces reads to off chip DRAM. Texture memory is most effectively utilised when memory access patterns exhibit a great deal of spatial locality where threads next to one another read addresses close together.

3.1.3 Kernel Execution

When a CUDA kernel function is executed the number of parallel blocks and threads must be specified in either one, two or three dimensions. The launch of CUDA kernels is highly configurable and different variations of parallel threads and blocks can result in vastly different performance from an application.

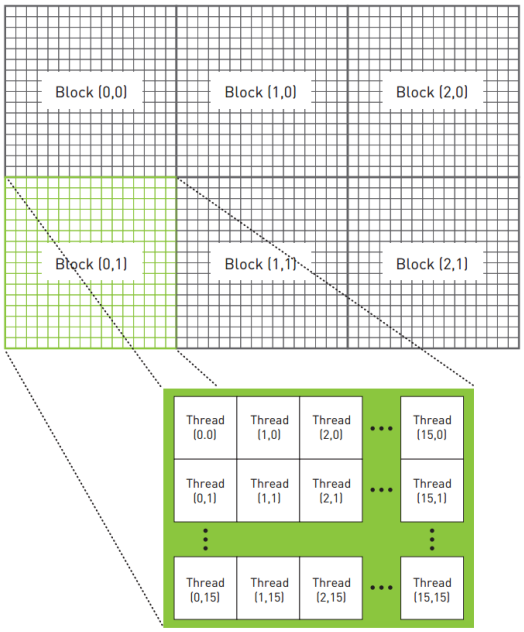

Figure 3.3: 2D hierarchy of blocks and threads that could be used to process a 48x32 grid (Sanders & Kandrot 2010a)

Figure 3.3 shows a two dimensional grid of 3x2 blocks each with 16x16 threads. This particular execution could be used to process a 48x32 grid where each thread corresponds to an individual position in the grid. CUDA uses the idea of warps to block work together, in the NVidia architecture a warp refers to a group of 32 threads that are executed in lockstep. At every line in a CUDA kernel each thread in a warp executes the same instruction but on a different part of the data.

CUDA Kernel code is derived from C with added keywords to allow sections of programs to be run on the device. Kernel functions are defined within source code using the *__global__* keyword and must have a *void* return type. Figure 3.4 below shows an example of a kernel function which performs vector addition.

```
8  __global__ void addVectors(int *a, int *b, int *c, int size)
9  {
10     // Get the id of the current thread
11     int tid = blockIdx.x;
12
13     // Check we are still in range of the vector
14     if (tid < size)
15     {
16         // Add a and b, store result in c
17         c[tid] = a[tid] + b[tid];
18     }
19 }
```

Figure 3.4: Add vectors a and b together and store result in vector c

Some initialisation needs to be performed before the *addVectors* kernel can be executed. Firstly the vectors need to be defined on both the host and the device and space allocated within global memory on the device to store the vectors. Finally the vectors need to be filled with data on the host and then copied from the host to the device as shown in Figure 3.5.

```
21  int main()
22  {
23      // Size of our vectors
24      const int size = 64;
25
26      // Host vectors
27      int a[size], b[size], c[size];
28
29      // Device vectors
30      int *dev_a, *dev_b, *dev_c;
31
32      // Allocate space within global memory on the device
33      // To store the vectors
34      cudaMalloc((void**)&dev_a, size * sizeof(int));
35      cudaMalloc((void**)&dev_b, size * sizeof(int));
36      cudaMalloc((void**)&dev_c, size * sizeof(int));
37
38      // Fill the vectors a and b on the host
39      for(int i = 0; i < size; i++)
40      {
41          a[i] = i;
42          b[i] = i;
43      }
44
45      // Copy the vectors a and b from the host to the device
46      cudaMemcpy(dev_a, a, size *sizeof(int), cudaMemcpyHostToDevice);
47      cudaMemcpy(dev_b, b, size *sizeof(int), cudaMemcpyHostToDevice);
48
```

Figure 3.5: Initialise vectors and copy data to the device

Once our vectors have been initialised and the data copied to the device we are able to set up our grid of blocks and threads. In this instance we create a one dimensional grid with 1 block and 1 thread for each position in our vector, as a result each thread will add one value from vector a and one value from vector b while storing the result in vector c. To launch our *addVectors* kernel we require special syntax which follows the rule.

function_name«<grid_dim, block_dim»>(parameters...)

```
49      // Define our work dimensions
50      dim3 grid_dim(size, 1, 1);
51      dim3 block_dim(1, 1, 1);
52
53      // Call the add vector kernel
54      addVectors<<<blocks, threads>>>(dev_a, dev_b, dev_c, size);
55
```

Figure 3.6: Launch *addVectors* kernel

Once the kernel execution has completed the results stored on the device are copied back to the host and displayed, finally the global memory allocated on the device is freed.

```
56      // Copy our data from vector dev_c back to the host
57      cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);
58
59      // Display the results
60      for(int i = 0; i < size; i++)
61      {
62          cout << a[i] << " + " << b[i] << " = " << c[i] << endl;
63      }
64
65      // Free the memory allocated on the device
66      cudaFree(dev_a);
67      cudaFree(dev_b);
68      cudaFree(dev_c);
```

Figure 3.7: Retrieve results and free allocated memory

3.1.4 Dynamic Parallelism

New with the Kepler GK110 architecture dynamic parallelism allows the GPU to generate new work for itself dynamically without ever having to involve the CPU. This results in the ability to run more of an application on the GPU improving both scalability and performance through support for more varied parallel workloads.

Before the introduction of Kepler, kernels would typically be launched from the host and run to completion before returning a result to the host which could be analyzed and further kernels launched. However with dynamic parallelism kernels now have the ability to call other kernels directly from the device, allowing the host to be free to complete other tasks.

Taking advantage of dynamic parallelism to reduce the amount of read and writes between host and device could result in significant speed ups for parallel applications. Figure 3.8 shows CPU involvement when launching multiple kernels on a Kepler GPU compared with previous architectures.

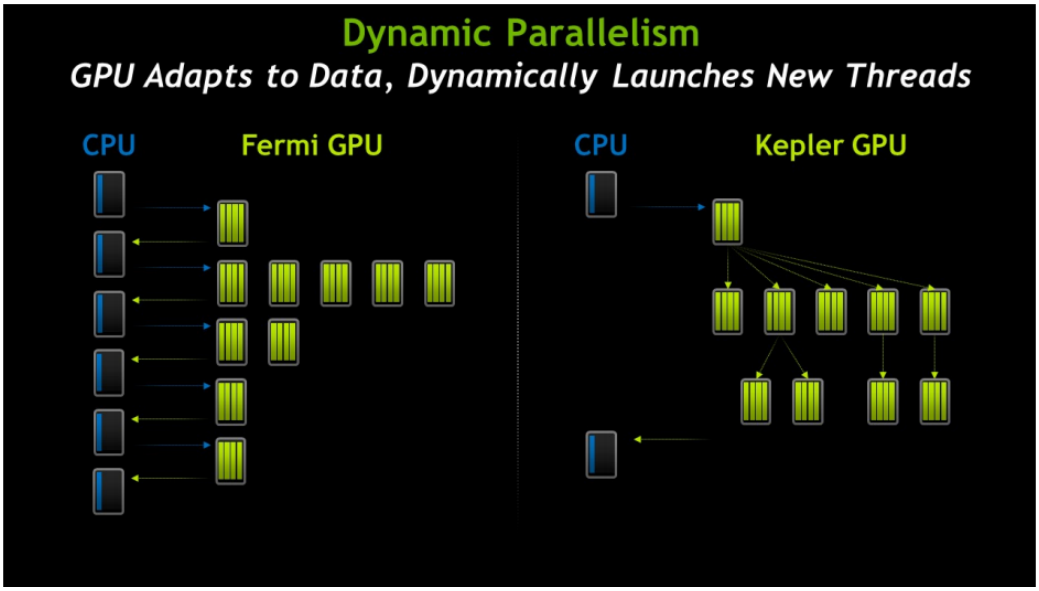

Figure 3.8: GPU launches new work for itself in Kepler architecture (NVidia 2013b)

3.2 Graph Representation

For both the diffusion and Dijkstra implementations graphs are represented as two-dimensional grids, with each node connected to its neighbouring nodes based on the moore neighbourhood. Each node in the graph also has an associated cost with different costs used to represent varying types of terrain within a game environment. For the graphs used in the Dijkstra implementations larger costs represent more hazardous terrain whereas for the diffusion graphs costs are represented as a value between zero and one with zero being impassable.

Two sets of maps have been chosen to test the algorithms against, the first set contains maps where every node has an equal terrain cost but the maps vary in size, the second set is made up of maps of equal size but contain varying terrain costs.

Set One - Varying Grid Sizes

32x32 64x64 128x128 256x256 512x512 1024x1024 2048x2048

Set Two - Varying Terrain Costs

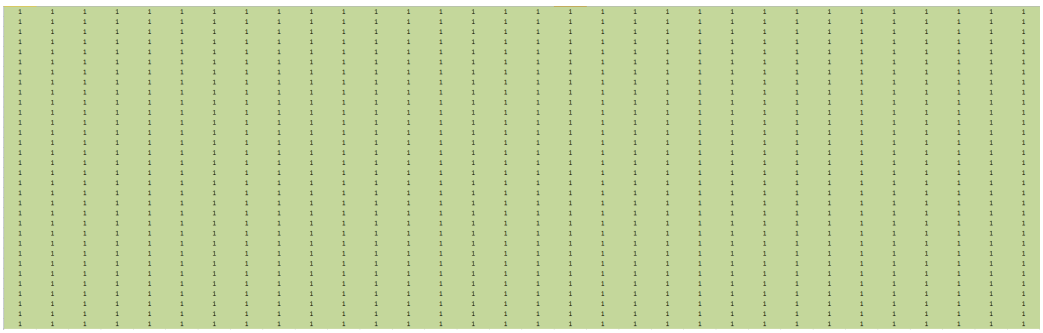

Figure 3.9: Test map 1

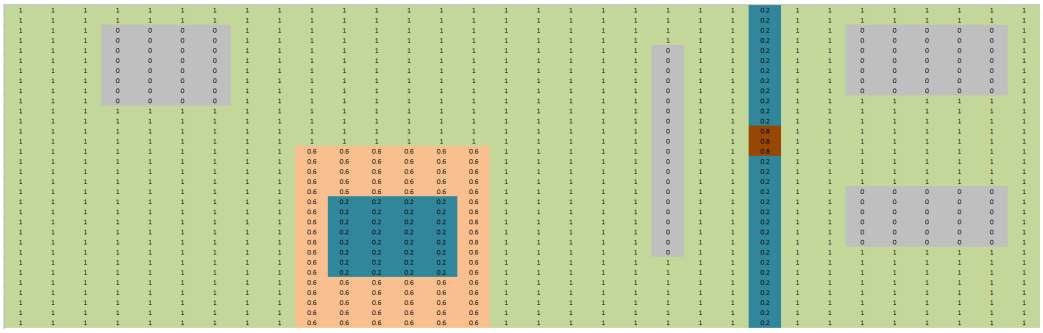

Figure 3.10: Test map 2

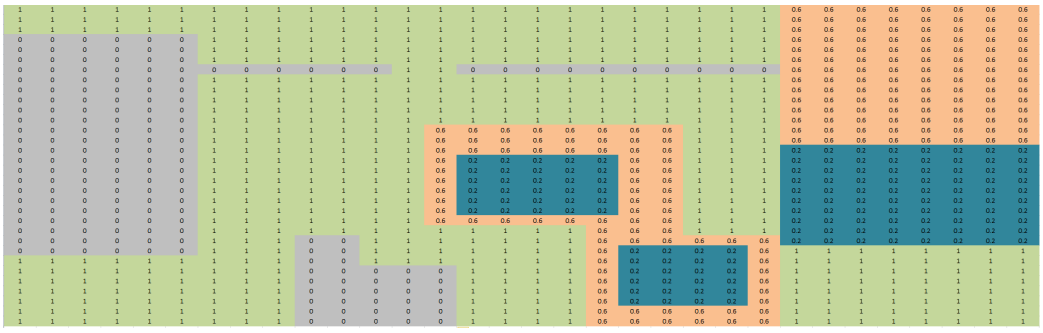

Figure 3.11: Test map 3

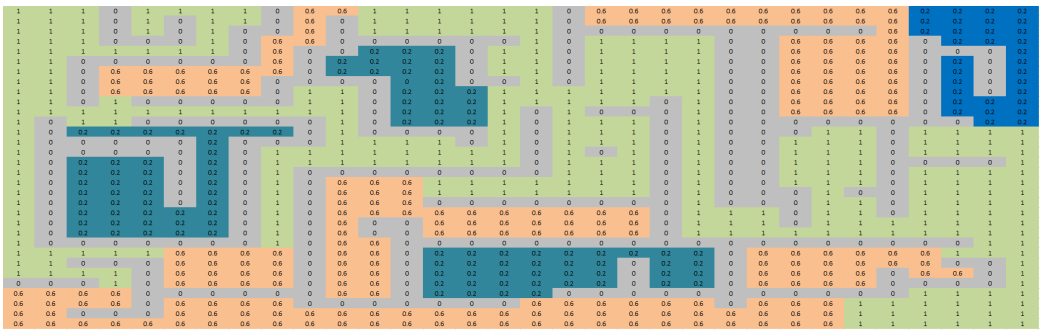

Figure 3.12: Test map 4

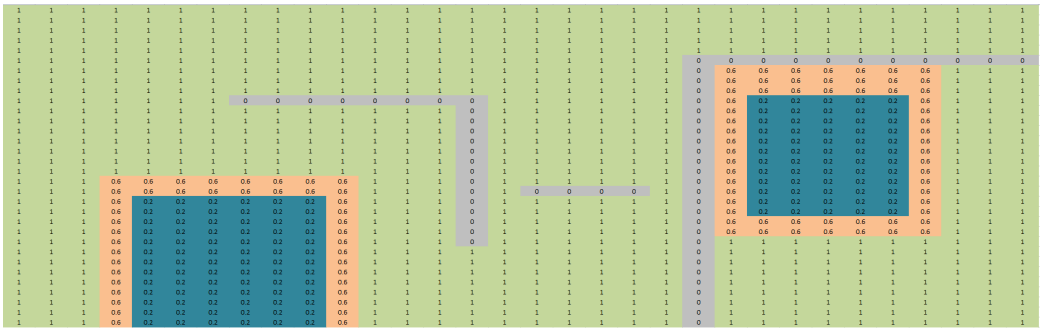

Figure 3.13: Test map 5

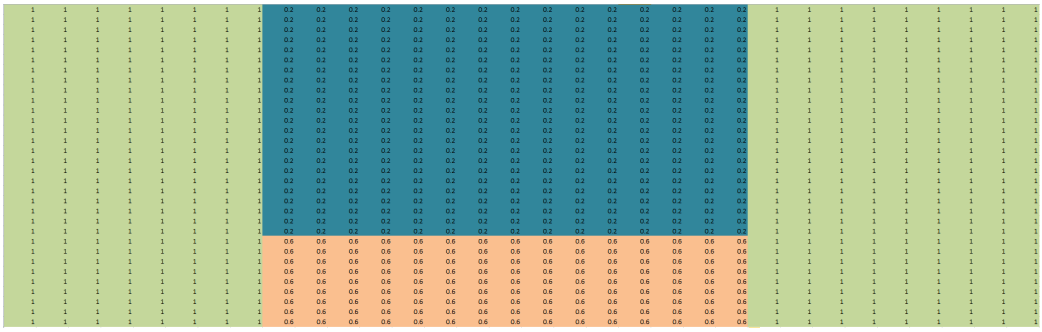

Figure 3.14: Test map 6

In each of the above maps grey areas are impassable while blue and yellow areas have a high cost of passing through compared to the green areas.

Start and Goal Nodes

For each of the maps the start node is the top left corner of the map while the goal node is the bottom left, these positions have been chosen as they

have the largest euclidean distance between them in the map.

3.3 Metrics

A number of metrics have been chosen in order to evaluate and compare the algorithms against. These metrics should provide an insight into how each algorithm performs and what factors effect the parallelization of the algorithms.

Iterations of Diffusion Loop to Fill Map

This metric is specific to only the diffusion implementations and is used to determine how many iterations of the diffusion loop is required to spread the goal value throughout the map so that a path can be found from every node in the map to the goal. This should provide insight into what factors effect the speed at which the goal value is diffused through the map.

Iterations to Evaluate Nodes in Queue

This metric is specific to the Dijkstra implementation and is used to determine how many iterations of the outer loop is needed to evaluate all the nodes that are added to the queue. The parallel implementation is concerned with evaluating multiple nodes in the queue at the same time, this metric will give an indication of how successful the parallelization has been.

Path Length

The length of each path found by both diffusion and Dijkstra is recorded, ideally we want to find the shortest path between the two points.

Path Quality

Path quality is based upon which algorithm produces the most intelligent and realistic path through the map as would be chosen by a human.

Time

The time to find each path is recorded, the faster the time the better the algorithm performs. Ideally we want paths to be found within real time if they are to be used within a game environment.

Chapter 4 | Implementation

This chapter contains a detailed description of both the sequential and parallel implementations of the diffusion and Dijkstra algorithms. We will present the CUDA kernels required to run the algorithms on the GPU and highlight the differences between the sequential and parallel implementations. To ensure that the parallelizations have been successful both sequential and parallel applications were tested on a series of test maps to ensure that the same path was found by both implementations.

4.1 Programming Environment

The applications were developed using Visual Studio 2012 Ultimate (Microsoft 2012) with the sequential applications written in C++ while the parallel applications were written in a combination of C++ and CUDA C/C++. V5.5 of the NVidia CUDA SDK is used and provides integration with visual studio allowing the build process to be simplified and syntax highlighting for CUDA specific keywords to be enabled. Through adding the .cu and .cuh file type extensions to the list of of Visual Studios C++ supported syntax file types it is also possible to make use of the intellisence auto-completion while working on CUDA kernel code.

4.2 Map Representation

As outlined in section 3.2 maps are represented as 2D grids where each cell in the grid represents a node in the graph with an associated terrain cost. The maps used within set one are generated at run time while the maps within set two have been pre-defined and are read in from a .CSV file where each cell in the file represents the cost of an associated node.

Both the diffusion and Dijkstra applications use the same sets of maps and the same methods to generate the maps, the only difference between the applications is the way in which the maps need to be stored for both the sequential and parallel versions. These differences will be highlighted in the following sections.

4.2.1 Sequential Representation

Within the sequential applications maps are represented using the C++ vector data structure contained within the C++ standard library. For each of

the maps a two-dimensional vector is created where each position in the vector represents the equivalent position within the map. Table 4.1 shows how nodes would be represented within the vector.

1	2	3
4	5	6
7	8	9

Table 4.1: Representation of nodes in 2D vector

4.2.2 Parallel Representation

Within the parallel applications the maps must be represented differently so that they can be stored and accessed correctly within the kernels. As a result we can only use one-dimensional data structures and the size of the map must be known at runtime. Table 4.2 shows how the map would be represented within a one dimensional array so that it can be accessed on the GPU.

1	2	3	4	5	6	7	8	9

Table 4.2: Representation of nodes in 1D array

4.3 Collaborative Diffusion

As Figure 4.1 shows there are three main stages which both the sequential and parallel diffusion applications follow. The application begins by generating the maps and initialising the grids before entering the main diffusion loop, this continues until a diffuse value has been calculated for every node in the graph; finally the path is backtracked and drawn to a file before the memory is freed.

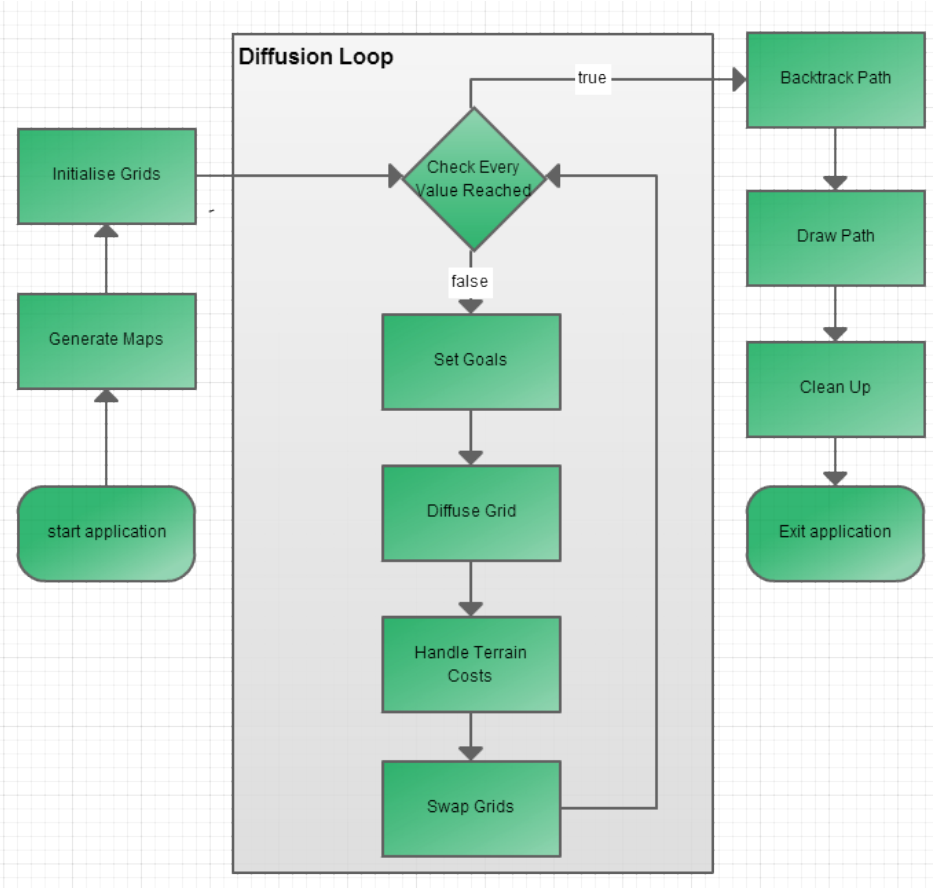

Figure 4.1: Diffusion Application Flow Diagram

4.3.1 Grid Initialisation

In order to complete the process correctly the application requires four grids to store the necessary data required to calculate the diffuse values for each tile in the grid. The size of each grid is equal to the size of the map in which a path is being calculated. These grids are;

Input and Output Grids
: The input and output grids contain the diffuse values calculated at each iteration of the diffusion loop.

Goal Grid
: The goal grid contains the initial values of the nodes within the map, the goal node has a large initial value.

Terrain Grid
: The terrain grid contains the terrain cost of each node in the map.

Each of these grids are set up in the initialisation stage with each position in the grid containing a single *float* value. The input and output grids are all initialised to zero along with every value in the goal grid; apart from the goal position which is set to an initially large value. The values in the terrain grid are either read in from a .CSV file which represents one of the six maps or set to one if no map is specified.

In the parallel version of the application once the grids are set up on the host we need to allocate space for them on the device and copy the data from host to device. This is done using the *cudaMalloc* and *cudaMemcpy* keywords.

Before we are able to launch our kernels within the parallel version we must first define our blocks and threads with which we will launch each kernel with. In order for the application to run correctly we must define enough threads so that there is one thread launched for each node in the grid. For each of the maps we launch blocks of 16x16 threads with the number of blocks being launched determined by the size of the map. For example for the 32x32 maps, 2x2 blocks with 16x16 threads are launched whereas for the 512x512 maps we launch 32x32 blocks with 16x16 threads. Previous testing determined that this combination resulted in the best performance for the application.

4.3.2 Diffusion Loop

Once the grids are set up we are able to begin the main diffusion loop; this loop runs until every node in the output grid which is not impassible no longer contains a zero value or until we reach a user defined maximum number of iterations. The first part of this process involves setting the goal values within the input grid, the reason this is done each iteration is that we want the goal value to remain constant while the values spread throughout the grid.

Figure 4.2 shows the kernel responsible for this process, this is done by getting the thread id which corresponds to the position of the node in the grids and checking if there is a goal at this position in the goal grid, if there is we copy the value into the corresponding position in the input grid. Sequentially we follow a similar process, the main difference however is that we must loop through every position in the grid and set each value one by one.

```
__global__ void copy_const_kernel(float *input, float *goals)
{
    // Get thread id and offset
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    // Copy our constant values (goals) into our grid
    if(goals[offset] != 0)
        input[offset] = goals[offset];
}
```

Figure 4.2: Copy Goals into Input Grid

Once the goal values have been set within the input grid we are able to calculate the diffuse values for each of the nodes in the grid, this is achieved using the diffusion equation;

$$D = \frac{1}{n}\sum_{i=0}^{n} ai$$

where:
D = diffusion value
n = number of neighbours
a = diffusion value of neighbour

This is done by getting the diffuse value for each of the eight neighbours of the node and setting its value to the average of the neighbours. It is for this process that we require both the input and output grids, in order for the values to diffuse correctly we need to take the average of the values from the input grid and store the answer in the corresponding position of the output grid. Figure 4.3 shows the kernel responsible for this process. As before we calculate the id of the current thread to get the position of the node within the grid. Within the sequential application we must loop through every node in the grid and calculate its diffuse value accordingly.

```
__global__ void diffuse_kernel(float *outputGrid, const float *inputGrid, int gridSize)
{
    //Get the thread id and offset into array
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    // Get the index of left and right neigbours
    int left = offset - 1;
    int right = offset + 1;
    if (x == 0) left++;
    if (x == gridSize - 1) right--;

    // Get the index of top and bottom Neigbours
    int top = offset - gridSize;
    int bottom = offset + gridSize;
    if (y == 0) top += gridSize;
    if (y == gridSize - 1) bottom -= gridSize;

    // Get the corner neighbours
    int topright, topleft, bottomright, bottomleft;
    if(x != gridSize - 1 && y != 0)
        topright = offset - gridSize + 1;
    if( x != 0 && y != 0)
        topleft = offset - gridSize - 1;
    if(x != 0 && y != gridSize - 1)
        bottomleft = offset - 1 + gridSize;
    if(x != gridSize - 1 && y != gridSize - 1)
        bottomright = offset + 1 + gridSize;

    // Calculate the diffuse value
    outputGrid[offset] = 0.125 * (inputGrid[left] + inputGrid[right] +
                                  inputGrid[top] + inputGrid[bottom] +
                                  inputGrid[topleft] + inputGrid[topright] +
                                  inputGrid[bottomleft] + inputGrid[bottomright]);
}
```

Figure 4.3: Calculate diffuse value for each node

Terrain costs are handled in a post process stage once the diffuse values for the iteration have been calculated. Terrain costs are a value between zero and one with zero being impassable, the diffuse value for each node is adjusted by multiplying its value by the terrain cost. This results in the goal value spreading more slowly across areas which have a terrain cost close to zero than those with a cost closer to one, when a path is backtracked this results in the path favouring areas where the terrain is easily passable and avoids walls and rough terrain. Figure 4.4 shows the kernel responsible for handling terrain costs.

```
__global__ void tile_cost_kernel(float *data_grid, float *wall_grid)
{
    // Get thread id and offset
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    // Multiply the diffuse value by the terrain cost.
    data_grid[offset] = data_grid[offset] * wall_grid[offset];
}
```

Figure 4.4: Multiply the diffuse value by the terrain cost

Finally the input and output grids are swapped ready for the next iteration of the diffusion loop to take place if there are still values in the grid which have not been diffused. If all the values have been diffused we are now able to find a path from any node in the grid to our goal node. Figure 4.5 shows the entire diffusion loop with the kernel calls.

```
//Diffusion Loop
for(int i = 0; i < iterations; i++)
{
    // Copy our goals into our start grid
    copy_const_kernel<<<blocks, threads>>>(dev_inputGrid, dev_goalGrid);

    // Calculate our diffusion values
    diffuse_kernel<<<blocks, threads>>>(dev_outputGrid, dev_inputGrid, gridSize);

    //Copy our terrain and wall into the grid
    tile_cost_kernel<<<blocks, threads>>>(dev_inputGrid, dev_wallgrid);

    // Swap our grids
    swap(dev_inputGrid, dev_outputGrid);

    //Copy our databack
    cudaMemcpy(temp, dev_outputGrid, gridSize*gridSize*sizeof(float), cudaMemcpyDeviceToHost);

    //If the grid contains no zero values
    bool zeroValues = checkforZeroValues(temp, tempWalls, gridSize);
    if(!zeroValues)
    {
        iterations = i;
        break;
    }
}
```

Figure 4.5: The Diffusion Loop

4.3.3 Backtracking the Path

Before we are able to backtrack a path in the parallel version of the application we must copy our output grid back from the device to the host so that we can access our diffuse values for the map. This is done using the *cudaMemcpy* keyword to copy the data back into a 1D array.

In order to backtrack a path we pick a starting position within the grid; for each of the maps in the testing sets this is always the top left hand corner of the map. To find the path; starting from the source position we check each of our neighbouring nodes and add which ever node has the larger diffuse value to the path, we continue doing this until we reach the goal node which will have the largest diffuse value in the map. At this point we now have a list containing every position in the path which is then written to a .CSV file along with the length of the path and the time taken for the path to be found. If no path to the goal exists a message explaining this is written

to the console.

4.3.4 Summary

Between both the sequential and parallel diffusion applications there is very little difference in terms of the process involved and the actual code. The main differences are the way in which the grids must be stored in memory so that they can be accessed appropriately and a little extra set up involved in the parallel version to pass the grids between host and device and vice versa, however CUDA makes this process very simple with its added keywords. The power of the parallel version comes from the ability to process each node in the grid at the same time, the inherent nature of the diffusion process makes this easy to achieve as each node has the same calculation applied to it.

4.4 Dijkstra

Unlike diffusion which lends itself nicely to a parallel implementation, Dijkstra requires a number of changes in order for it to be parallelized correctly. Before looking at these changes we will first outline the sequential implementation of the application.

4.4.1 Sequential Implementation

The sequential application follows the same process as outlined in section 2.3.1. For each of the maps we create a node graph which is then used to perform the Dijkstra algorithm against, the node graph itself is a two-dimensional vector of equal size to the map. Each node in the graph contains the cost of passing through the particular node, the id of the node and a list of its neighbouring nodes. The list of neighbours is made up of the nodes eight surrounding neighbours based on the moore neighbourhood. Figure 4.6 shows the node struct.

```
struct node
{
    // Cost to travel through node
    cost _cost;

    // ID, made up of nodes position in grid
    id _id;

    //List of nodes neighbours
    neighbourData neighbours;

    // Constructor
    node(cost _cost, id _id)
        : _cost(_cost), _id(_id) { }
};
```

Figure 4.6: The Node Data Structure

Once the node graph is set up we need two other grids, a minimum distance grid to store the distances between the goal node and every other node and a previous grid to store the id of the next node to move to in order to reach the goal node. Each position in the minimum distance grid must be initially set to infinity with the cost of the goal node being set to zero. Each id within the previous grid is initially set to null.

Dijkstra itself works based upon the idea of a queue, where each node is added to the queue and at each iteration the node with the lowest cost is removed and evaluated. In order to achieve this we need to use a priority queue so that when nodes are added to the queue they remain ordered. Within the sequential version this is achieved using the *set* data structure within the C++ standard library as it provides good performance for adding and removing nodes. Within the queue we store the nodes id and the cost of reaching the goal from this node.

Next we add the goal node to the queue and then enter the outer loop. Within this loop we select the node in the queue which has the lowest cost and then for each of its neighbours check if the cost of passing through this neighbour is less than our current minimum distance for the node. If the distance is less we update the minimum distance and the parent of the neighbour node and add it to the queue before repeating the process until the queue is empty.

```
//loop until the queue is empty
while (!vertex_queue.empty())
{
    // get the cost for the first node in the queue
    cost dist = vertex_queue.begin()->first;

    // get the id of the first node in queue
    id _id = vertex_queue.begin()->second;
    node currNode = nodeGraph[_id.first][_id.second];

    // remove the first node from the queue
    vertex_queue.erase(vertex_queue.begin());

    //Visit each of the neighbours of the current node
    for(auto& neighbours : currNode.neighbours)
    {
        // Get the node for the current neighbour
        node neighbour = nodeGraph[neighbours.first.first][neighbours.first.second];
        cost neighbourCost = neighbours.second;
        id neighbourID = neighbour._id;

        // Get the distance of passing through this neighbour
        cost distanceThrough = dist + neighbourCost;

        // Check if this distance is shorter than minimum distance
        if(distanceThrough < min_distance[neighbourID.first][neighbourID.second])
        {
            // remove this neighbour from the queue if it exists there
            vertex_queue.erase(make_pair(min_distance[neighbourID.first][neighbourID.second], neighbourID));

            // set the minDistance for this position to distanceThrough
            min_distance[neighbourID.first][neighbourID.second] = distanceThrough;

            // set previous at this position to the current node
            previous[neighbourID.first][neighbourID.second] = currNode._id;

            // Add this neighbour to the queue
            vertex_queue.insert(make_pair(min_distance[neighbourID.first][neighbourID.second], neighbourID));
        }
    }
```

Figure 4.7: Check Each Node in the Queue Until it is Empty

Once each node in the queue has been checked our previous grid now contains a shortest path tree which we can perform a breadth first search over in order to find a path from any node in the grid to the goal node. Similar to the diffusion application a path is then backtracked from the top left corner of the map to the goal node and the path, path length and time taken to find the path are all written to a .CSV file.

4.4.2 Parallel Implementation

For the parallel implementation we follow a similar process as outlined by (Ortega-Arranz et al. 2013) as discussed in section 2.4. In an attempt to simplify the process detailed by (Ortega-Arranz et al. 2013), we have made use of dynamic parallelism within the Kepler architecture to reduce the number of read and writes between host and device and reduce the involvement of the CPU.

Unlike diffusion Dijkstra is not inherently parallel, however as outlined by (Ortega-Arranz et al. 2013) it is possible to select a number of nodes from the queue and process them in parallel without affecting the algorithms correctness, this set of nodes are known as the frontier set. As a result this changes the overall process of the algorithm compared to the sequential version as

we now must find a means of determining which nodes can be settled at the same time and can no longer just select the node with the lowest cost from the queue. Figure 4.8 shows the flow of the parallel Dijkstra application.

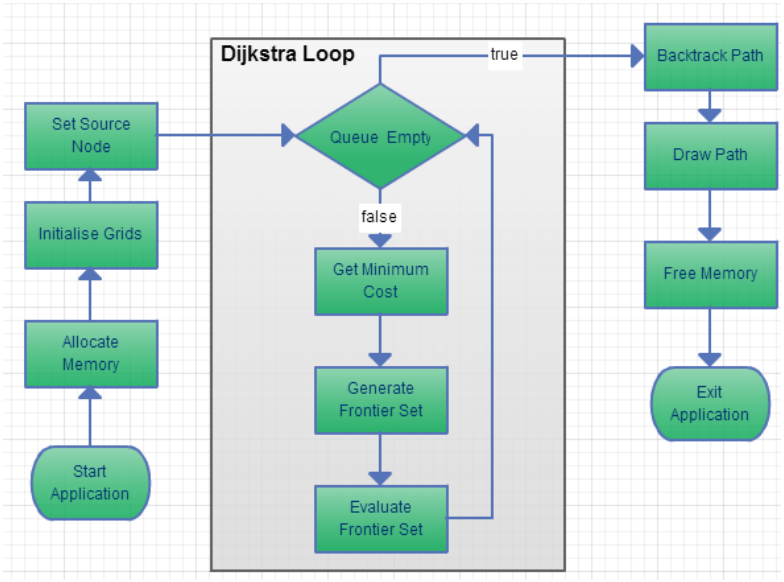

Figure 4.8: Flow of Parallel Dijkstra Application

The first thing we need to do is allocate memory on the GPU to store our grids, like in the sequential version we need to allocate space for our node graph, previous and minimum distance grids. This is done using the *cudaMalloc* keyword to allocate enough space for each grid within global memory. We also reserve space on the host so we can copy the data back once the Dijkstra algorithm has completed.

Due to the nature of the GPU there is no support for a priority queue or the C++ standard library, as a result of this we need to implement our own way of handling which nodes are in the queue. This is achieved through using a grid of boolean values which correspond to a node in the node graph. If the value is true the node is in the queue. We also have a similar grid for determining which nodes are in the frontier set.

Once space has been allocated appropriately for each of the grids we then set up our blocks and threads using the same combination as the diffusion application creating blocks of 16x16 threads and determining the blocks by the size of the map being processed so there is one thread for each node in the grid. After this we then make a single kernel call from the host to run

the Dijkstra algorithm, this kernel is launched with only a single block and thread while the blocks and threads are passed as parameters along with the grids and the source node. It is within this kernel that dynamic parallelism takes place.

```
//Run dijkstra on GPU
runDijkstraKernel<<<1, 1>>>(dev_previous, dev_min_distance, dev_nodeGraph,
                                        dev_inQueue, dev_inFrontier, blocks, threads,
                                        sourceNode, gridSize, dev_iterations);
```

Figure 4.9: Calling the Dijkstra Kernel From the Host

The reason that this kernel is launched with only a single block and thread is that we only need a single thread to handle the control of the application on the GPU side when using dynamic parallelism, if we were to launch this kernel with multiple threads we could introduce race conditions which would effect the correctness of the algorithm.

The first thing that we do within the run Dijkstra kernel is call the initialise data kernel to initialise each of the grids. Within the kernel we get the thread id which represents the nodes position in the grid, we set the nodes value in the previous grid to null, the minimum distance grid to infinity, add the node to the node graph and set both its position in the queue and the frontier set to false. Figure 4.10 shows the initialise data kernel.

```
__global__ void initialiseDataKernel(nodeID *previous, float* min_distance, node *nodeGraph,
                                     bool *inQueue, bool *inFrontier)
{
    // Get the thread ID in the grid
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    // previous is the offset of the parent in the node array
    // init to -1 as there is no parent initially
    previous[offset] = nodeID(-1);

    // min_distance is the minimum distance from the source to each node
    // Initially set to infinity
    min_distance[offset] = INFINITY;

    // node_graph contains all the nodes in the graph
    nodeGraph[offset] = node(1.0f, nodeID(offset));

    // inQueue is a bool which says if the node at offset is inQueue
    inQueue[offset] = false;

    // inFrontier is a bool which says if the node is in the frontier set
    inFrontier[offset] = false;
}
```

Figure 4.10: Initialise the Grids on the GPU

Whenever calling a kernel from within another kernel when using dynamic parallelism we must always call *cudaDeviceSyncronize()* immediately after, this ensures that we wait for all threads to complete the operation before we move onto the next set of instructions. Failure to do this would introduce race conditions stopping the algorithm from performing correctly.

Next we call the set goal kernel passing in the id of the goal node. We get the thread id and check to see if this matches the id of the goal node, if it does we set the minimum distance of the corresponding node to zero and add it to the queue by setting the corresponding position in the queue to true.

```
__global__ void setGoalKernel(float* min_distance, bool *inQueue, int sourceOffset)
{
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    if(offset == sourceOffset)
    {
        //Set the minDistance of the source node to zero
        min_distance[offset] = 0;

        //add it to the queue
        inQueue[offset] = true;
    }
}
```

Figure 4.11: Set Goal kernel

Once the goal has been set we then enter the main loop which runs until the queue is empty. Unlike the sequential version where we simply take the node from the top of the priority queue which handles the ordering of the nodes, we need to manually determine which node has the lowest cost in the queue so that we can determine which nodes are added to the frontier set.

This is done through launching a kernel which checks if the node is in the queue using the thread id and if it is performing an atomic min operation to store the value of the lowest cost if it is less than current minimum. As each node is being checked in parallel we need to use an atomic operation so that the entire read, execute and write operation is handled in a single step otherwise race conditions could cause the program to perform incorrectly.

```
__global__ void getMinKernel(int *globalMin, float* min_distance, bool *inQueue)
{
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    //Get the cost for this node
    int distance = min_distance[offset];

    if(inQueue[offset] == true)
    {
        //If it is less than the current global min, set it as the global min
        atomicMin(globalMin, distance);
    }
}
```

Figure 4.12: Find the Minimum Cost in the Queue

The frontier set determines which nodes in the queue can be processed at the same time without affecting the algorithms correctness. (Ortega-Arranz et al. 2013) outline two methods for evaluating which nodes can be added to the frontier set, for our implementation we have chosen the simplest of the two. A node is added to the frontier set if it is in the queue and it has a cost less than or equal to the lowest cost in the queue plus one. This process is handled in the frontier set kernel.

```
__global__ void frontierSetKernel(float *min_distance, node *nodeGraph, bool *inQueue,
                                  bool *inFrontier, float globalMin)
{
    // Add any node in the queue which has a distance
    // of min_distance <= min_distance + 1 to the frontier set

    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    if(inQueue[offset] == true)
    {
        if(min_distance[offset] <= globalMin + 1)
        {
            //Add to frontier set
            inFrontier[offset] = true;
        }
    }
}
```

Figure 4.13: Add Nodes to the Frontier Set

Evaluating the nodes in the frontier set is the most involved process on the GPU. For each node that is in the frontier set we check each of its eight neighbours and see if the cost of passing through the neighbour is less than the current lowest cost, if it is we update the neighbours parent and minimum cost and add the neighbour to the queue.

(Ortega-Arranz et al. 2013) define this process as the *inner loop* and check each of the nodes neighbours in parallel. Within our implementation we have two versions of the application, one which looks at each neighbour in parallel and another which checks each neighbour one by one. To allow the neighbours to be processed in parallel CUDA streams were used. CUDA streams are a sequence of operations that execute in issue-order on the GPU; CUDA operations in different streams may run concurrently. Figure 4.14 shows the potential difference in execution time when using CUDA streams. The differences between these two implementations will be discussed in section 5.

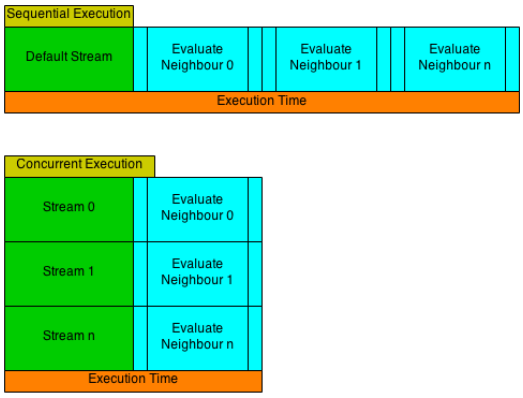

Figure 4.14: CUDA Streams

Finally we call a kernel which checks if there are any nodes remaining in the queue, if there are we repeat the loop again otherwise we end the kernel and control is returned back to the host. The entire run Dijkstra kernel is shown in figure 4.15.

```
__global__ void runDijkstraKernel(nodeID *previous, float* min_distance, node *nodeGraph,
                                  bool *inQueue, bool *inFrontier, dim3 blocks, dim3 threads, int sourceOffset,
                                  int gridSize, int *iterations)
{
    iterations[0] = 0;

    //Initialise the data
    initialiseDataKernel<<<blocks, threads>>>(previous, min_distance, nodeGraph, inQueue, inFrontier);
    cudaDeviceSynchronize();

    //Set the sourceNode
    setGoalKernel<<<blocks, threads>>>(min_distance, inQueue, sourceOffset);
    cudaDeviceSynchronize();

    queueEmpty = false;
    while(queueEmpty == false)
    {
        //Find the minimum global distance
        globalMin = INFINITY;
        getMinKernel<<<blocks, threads>>>(&globalMin, min_distance, inQueue);
        cudaDeviceSynchronize();

        //Add eligible nodes to the frontier set
        frontierSetKernel<<<blocks, threads>>>(min_distance, nodeGraph, inQueue, inFrontier, globalMin);
        cudaDeviceSynchronize();

        //Evaluate Each of the nodes in the frontier set
        evaluateFrontierSetKernel<<<blocks, threads>>>(previous, min_distance, nodeGraph,
                                                       inQueue, inFrontier, blocks, threads,
                                                       gridSize);
        cudaDeviceSynchronize();

        //At the end check if the queue is empty, if it is we have evaluated everynode
        queueEmpty = true;
        nodesinQueue = 0;
        queueEmptyKernel<<<blocks, threads>>>(&queueEmpty, inQueue, &nodesinQueue);
        cudaDeviceSynchronize();

        iterations[0]++;
    }
}
```

Figure 4.15: Run Dijkstra Kernel

Once control is returned to the host we then need to copy our data from the device back to the host and backtrack a path using a breadth first search. If no path to the goal exists a message is written to the console otherwise the path is written to a .CSV file along with the length of the path and the time taken to find the path. Finally the memory on both the host and the device is freed.

4.4.3 Summary

Unlike diffusion Dijkstra does not lend itself well to a parallel implementation and a number of changes are required to the overall process of the algorithm so that paths can be found correctly. One of the biggest changes over the sequential implementation is the way in which the priority queue is handled on the GPU and the means of determining which nodes can be added to the frontier set. However through making use of dynamic parallelism within the Kepler architecture this process was somewhat simplified as we were able to call kernels to handle the operations of the queue directly from the GPU.

Chapter 5 | Results

Both the sequential and parallel diffusion and Dijkstra applications were tested on both sets of maps outlined in section 3.2 and the metrics specified in section 3.3 recorded for each of the maps. Each map in both sets were run 10 times and the average taken for each of the metrics.

5.1 Set One Results

The purpose of the maps in set one were to test the time taken for each algorithm to find a path as the size of the maps increase. As each node in the map has an equal terrain cost, both diffusion and Dijkstra found identical paths moving in a straight line from the top left corner to the bottom right. The length of each path is equal to the width and height of each of the maps tested.

Collaborative Diffusion

Figure 5.1 shows the time taken for each map from both the sequential and parallel diffusion implementations. The time taken axis increments using a base 10 logarithmic scale.

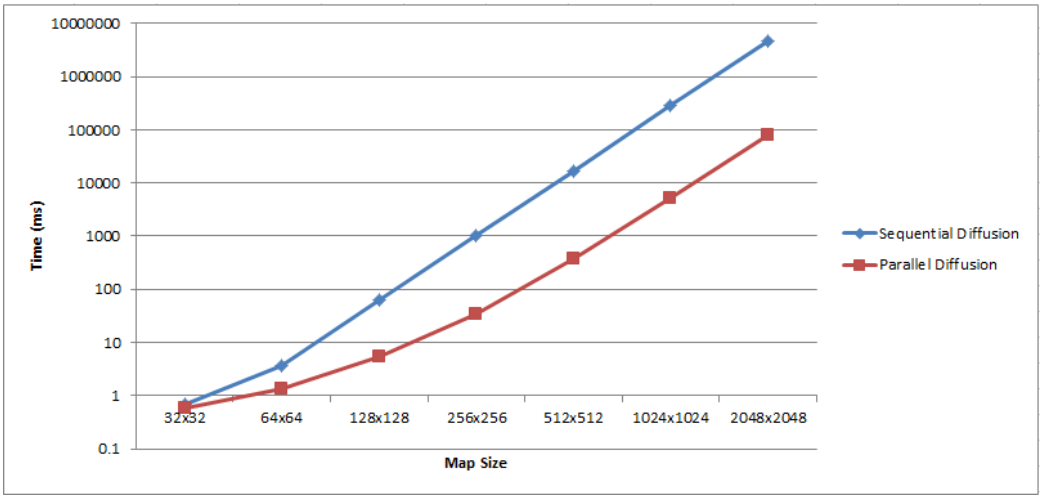

Figure 5.1: Sequential vs Parallel Diffusion

It is clear that for each of the map sizes, the parallel version outperforms the sequential version taking significantly less time to diffuse the goal value

throughout the grid and find a path. For the largest map size tested the parallel version takes 98% less time to find the path over the sequential version. The iterations required to fill the grid are shown in table 5.1.

Map	Sequential Iterations	Parallel Iterations
32x32	31	31
64x64	68	75
128x128	224	261
256x256	866	1026
512x512	3507	4175
1024x1024	14382	17176
2048x2048	59256	65091

Table 5.1: Iterations to Diffuse Goal Value

Even though the parallel version is able to find a path quicker, the number of iterations required to diffuse the goal value throughout the grid is much larger than in the sequential version. This is surprising as it would be expected that both versions should take the same number of iterations to diffuse the value.

As a result of this a number of other tests were carried out to determine what factors could effect the time taken to diffuse the value throughout the grid. The first test involved looking at the size of the goal value and how this effects the time taken, the results from this are shown in figure 5.2.

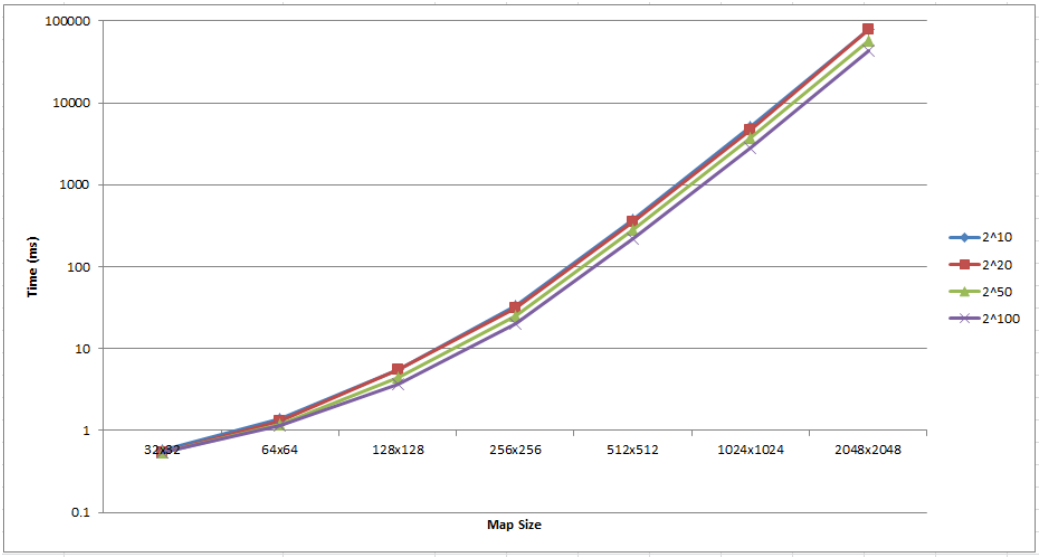

Figure 5.2: Time to Fill With Different Goal Values

As can be seen the larger the initial goal value the less iterations needed to fill the grid and the less time taken to find a path. For the largest map size when the initial value is increased from 1024 to 2^{100} the time to find a path is reduced from 1 minute 17 seconds to 42 seconds, this indicates that larger values diffuse more quickly.

The second test involves looking at how the goals position within the grid can effect the time taken to find a path. For this test the goal was moved to the centre of each map as due to the nature of the diffusion process this should theoretically reach every node quicker than when the goal is the bottom right node. Figure 5.3 shows the results from this test, the time axis increments using a base 10 logarithmic scale.

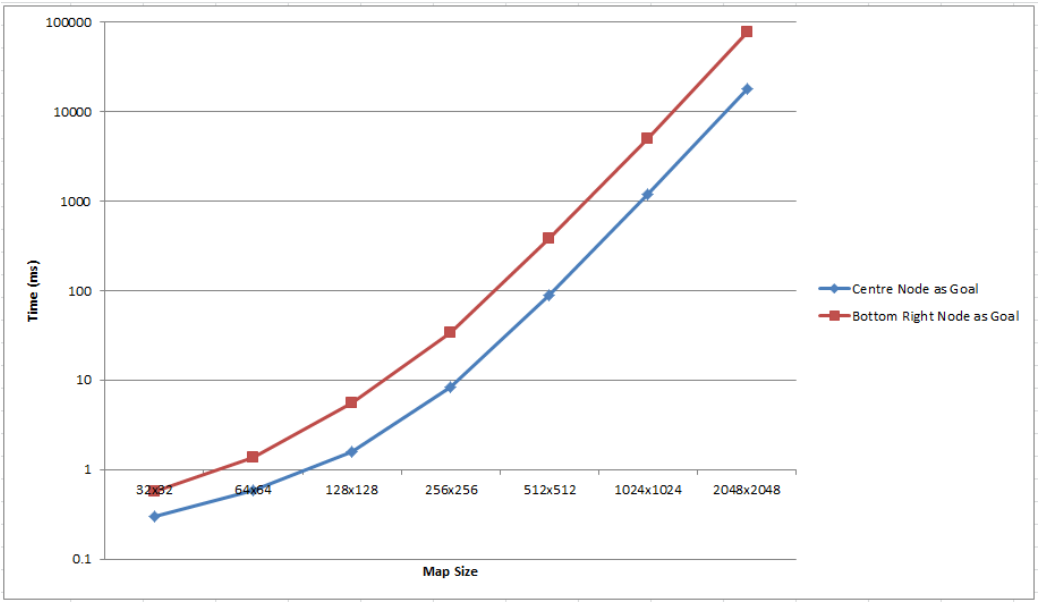

Figure 5.3: Time to Find Path With Centre as Goal Node

As expected when the goal node is in the centre of the map the time taken to find the path is improved reducing the time taken by 76% for the largest map size. Due to the nature of the diffusion algorithm we would expect the centre goal position to give the best performance while the corner nodes of the map should give the worst performance, it would be reasonable to expect that every other position in the grid would fall within this range.

Dijkstra

Two versions of the parallel Dijkstra application were implemented, one version attempted to parallelize both the inner and outer loops of the Dijkstra algorithm using CUDA streams while the other only parallelized the outer-loop. The time taken for both these applications are shown in figure 5.4. The time elapsed axis increments using a base 10 logarithmic scale.

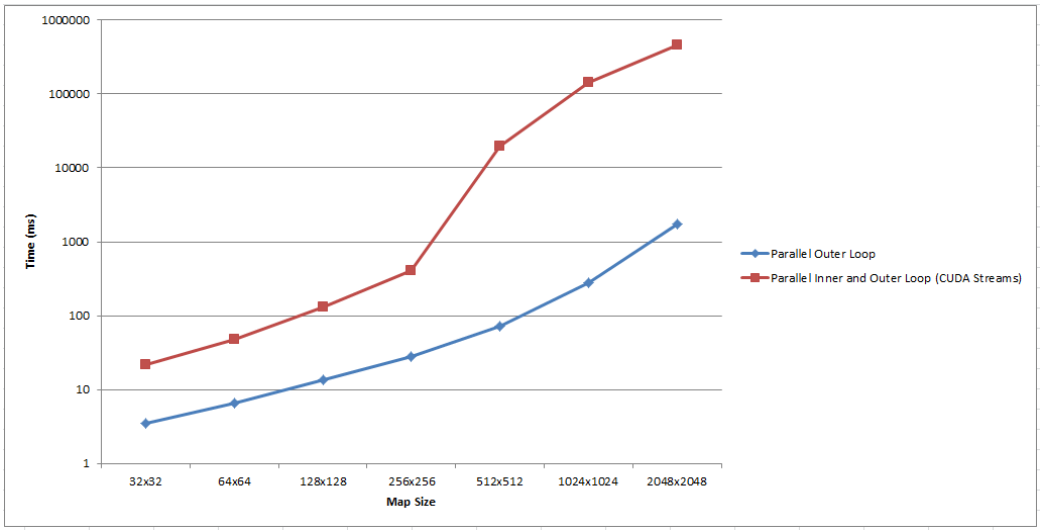

Figure 5.4: Parallelized Outer loop vs Parallelized Outer and Inner Loop

Contrary to what would have been expected, evaluating each of the nodes neighbours in parallel is actually slower than simply looking at each neighbour one by one. Investigating further and looking at the time taken to create and launch a stream compared to calling a function to evaluate the nodes neighbours we can see that streams incur a large overhead. This is shown in figure 5.5.

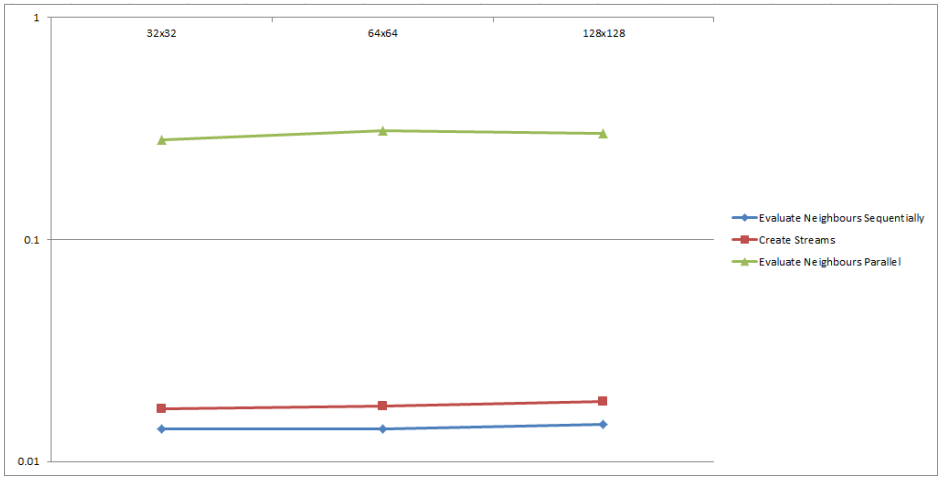

Figure 5.5: Evaluate Neighbours Sequentially vs Creating Streams

Looking at the graph we can see that we can call the function to evaluate all eight of the nodes neighbours quicker than we can create eight streams to handle the neighbours. On top of creating the streams we also have to launch eight kernels which results in us launching a new block and thread to handle each neighbour, this process is extreamly slow and takes almost 18X longer than it does to simply call the function which does not result in any new threads being launched. As a result of this, the version which only parallelizes the outer loop will be used to compare against the seqential counterpart and the diffusion application.

Figure 5.6 shows the time taken for each map in set one for both the sequential and parallel implementations of the Dijkstra applications. The time elapsed axis increments using a base 10 logarithmic scale.

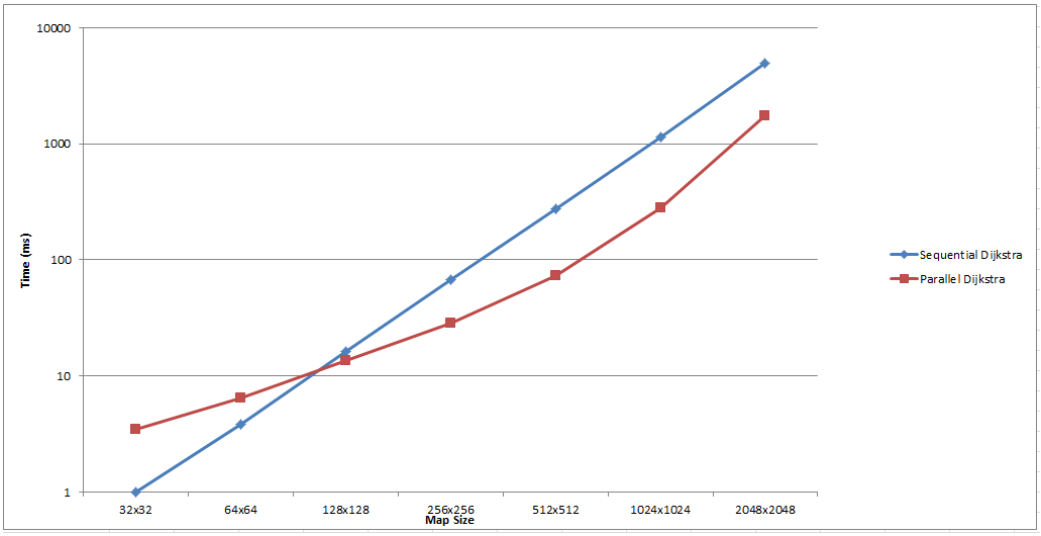

Figure 5.6: Sequential vs Parallel Dijkstra

Unlike the parallel diffusion implementation which outperforms its sequential counterpart for each of the maps, the sequential Dijkstra implementation outperforms the parallel implementation for both the 32x32 and 64x64 maps. However for the larger graph sizes the parallel implementation takes an average of 75% less time for the 1024x1024 map and 65% less time for the 2048x2048 map.

Table 5.2 shows the iterations of the outer loop required to evaluate every node that is added to the queue.

Map	Sequential Iterations	Parallel Iterations
32x32	1024	32
64x64	4096	64
128x128	16384	128
256x256	65536	256
512x512	262144	512
1024x1024	1048576	1024
2048x2048	4194304	2048

Table 5.2: Iterations to Evaluate All Nodes in Queue

Interestingly the number of iterations required to evaluate all the nodes in the queue tie in with the size of the map in which the algorithm is being run against. For the nxn maps in set one, the number of iterations required by

the sequential version is equal to n*n while the parallel version only requires n iterations where the terrain cost for each node is equal.

Comparison of Diffusion and Dijkstra

As we have seen both of the parallel implementations outperform their sequential counterparts for the large graph sizes, This section will look at diffusion and Dijkstra perform against one another for the maps in set one. Figure 5.7 shows the time taken for all the versions of the application to find a path for the different map sizes.

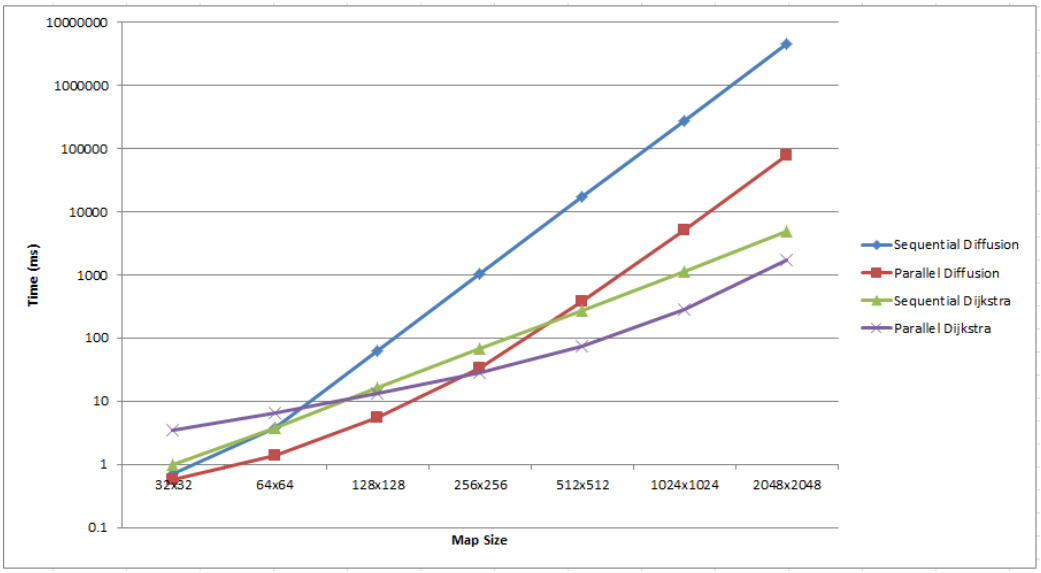

Figure 5.7: Diffusion vs Dijkstra

As we can see for both the 32x32 and 64x64 maps parallel Dijkstra is the slowest of the algorithms taking 6X longer than the parallel diffusion implementation. The time taken for the diffusion applications increases much quicker as the size of the map increases than the Dijkstra implementations and as a result both the sequential and parallel Dijkstra perform much faster than the parallel diffusion for the larger map sizes with the parallel Dijkstra performing 45X faster for the 2048x2048 grid than the parallel diffusion. This indicates that although diffusion provides the best performance for small maps it does not scale well as the size of the maps increase.

5.2 Set Two Results

This section looks at the result collected from the maps in set two, the purpose of these map were to investigate the differences between the paths found by both diffusion and Dijkstra and the effect that varying terrain costs has on the algorithms.

A survey was handed out to fellow students in which they were shown two paths through each map and asked to select which of the paths through the map they felt produced the more realistic path that they would expect a character within a game to follow. Students did not know which paths belonged to either the diffusion or Dijkstra implementation. 18 students responded to the survey.

Within this set there are six maps which each of the algorithms were tested against, both sequential and parallel implementations found the same paths through each of the maps.

Map One

Map one was the simplest of the maps containing no obstacles or costly terrain. Both diffusion and Dijkstra implementations found an identical path taking 32 nodes to move directly from the top left corner to the bottom right.

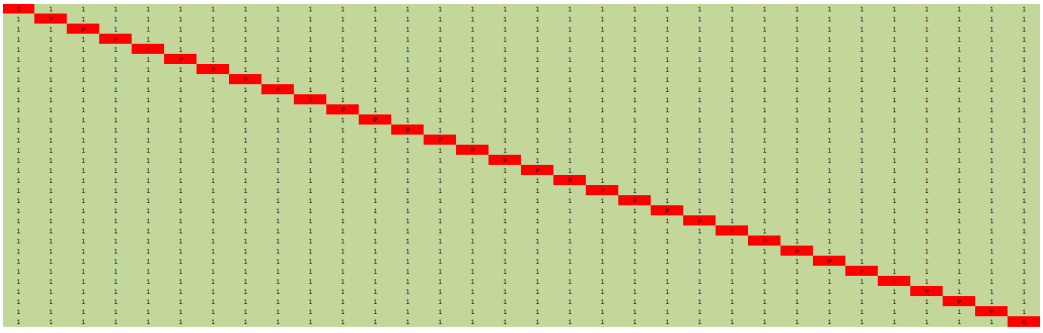

Figure 5.8: Diffusion Path Map One

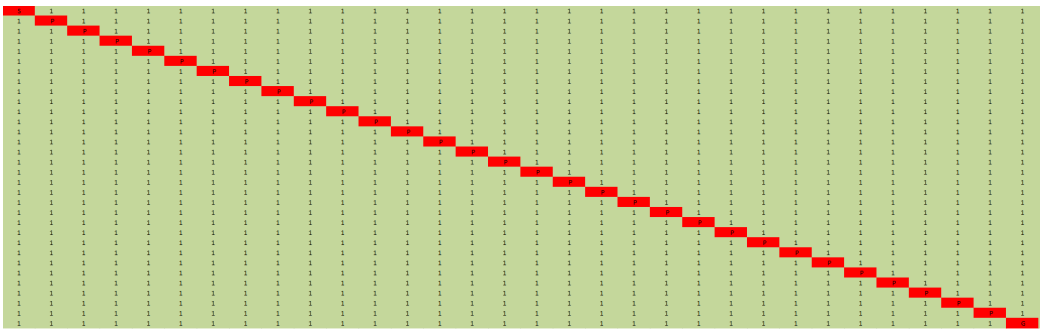

Figure 5.9: Dijkstra Path Map One

Map two

Map two introduces a number of obstacles and costly terrain, there is a small beach with a lake in the centre and a river with a small wooden bridge over it along with a number of impassable areas.

The path found by the diffusion implementation was significantly shorter than that found by Dijkstra, traversing only 39 nodes to reach the goal compared to the 49 taken by Dijkstra.

Looking at figures 5.10 and 5.11 we can clearly see the differences in the paths with diffusion choosing to move over the more costly terrain and going straight through the river while Dijkstra takes the longer route avoiding the river and moving over the bridge to reach the goal. When asked 72% of students felt that the diffusion path was more realistic.

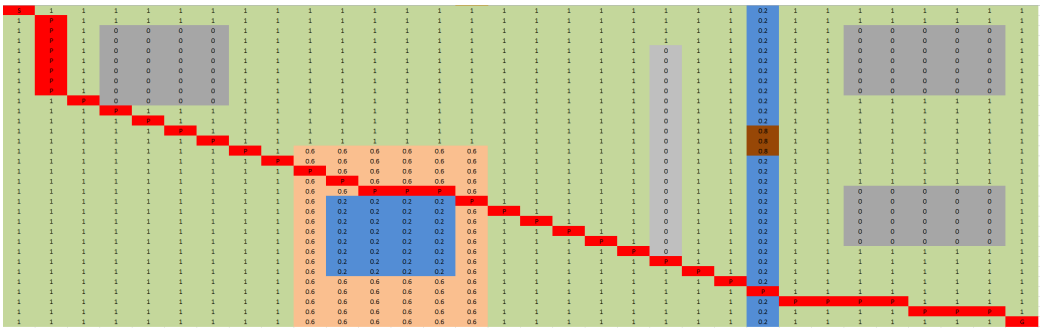

Figure 5.10: Diffusion Path Map Two

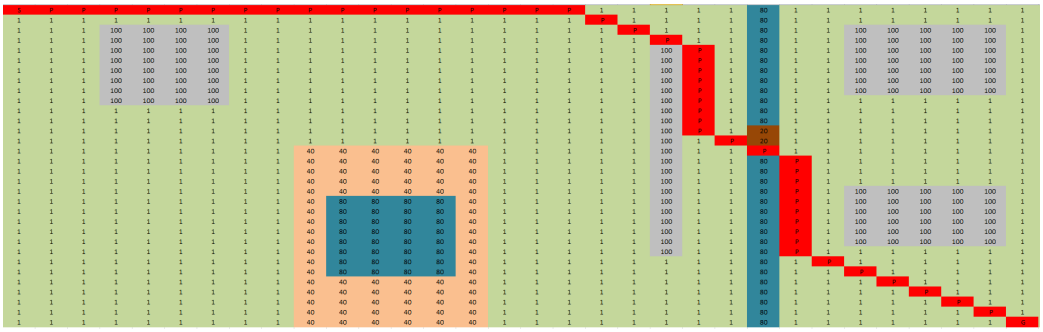

Figure 5.11: Dijkstra Path Map Two

Map Three

Map three introduces more terrain and obstacles over map two with approximately 50% of the nodes having a terrain cost other than one.

Similar to map two diffusion was able to find the shorter path of the two algorithms traversing only 38 nodes to reach the goal compared to Dijkstras 41, again this was achieved by moving over certain areas with a higher terrain cost while Dijkstra takes the longer route in order to avoid any areas of costly terrain.

As with map two 72% of students asked felt that diffusion provided the more realistic path.

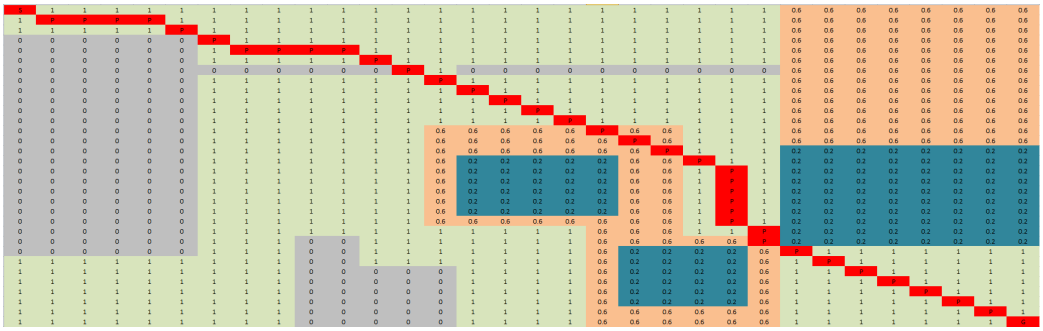

Figure 5.12: Diffusion Path Map Three

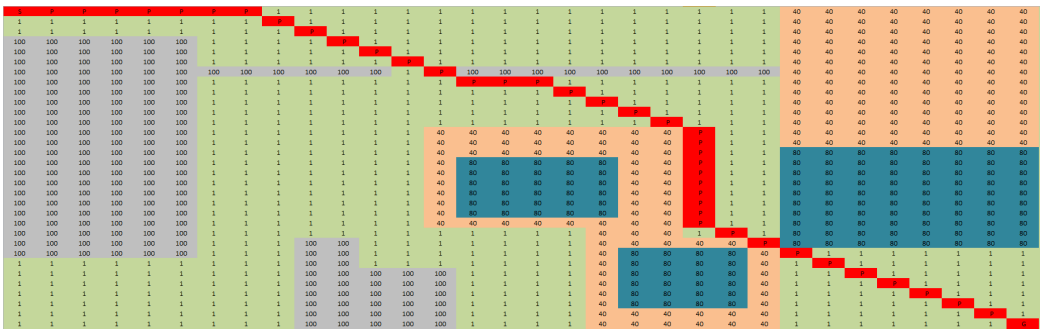

Figure 5.13: Dijkstra Path Map Three

Map Four

Map four was the most complex of the maps, providing a maze like structure for the algorithms to solve with a number of high terrain cost areas. In this example both diffusion and Dijkstra found near identical paths taking 61 nodes each to reach the goal. 67% of the students asked felt that Dijkstra provided the more realistic path.

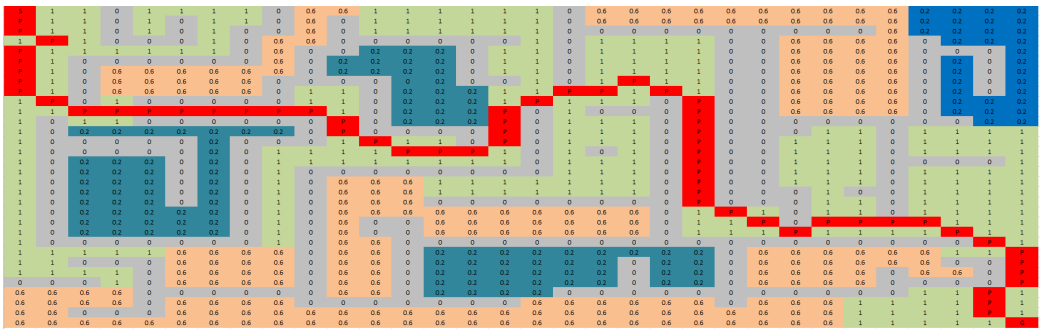

Figure 5.14: Diffusion Path Map Four

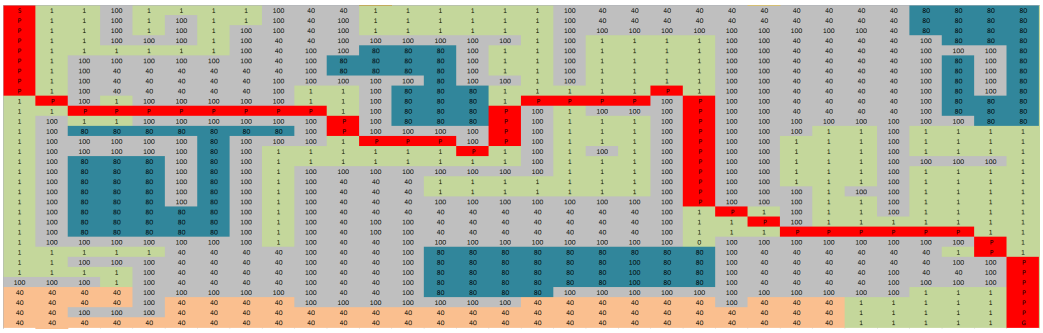

Figure 5.15: Dijkstra Path Map Four

Map Five

Map five was designed to test how each algorithm behaves when no path to the goal exists. In both cases diffusion and Dijkstra performed as expected and were unable to find a path to the goal.

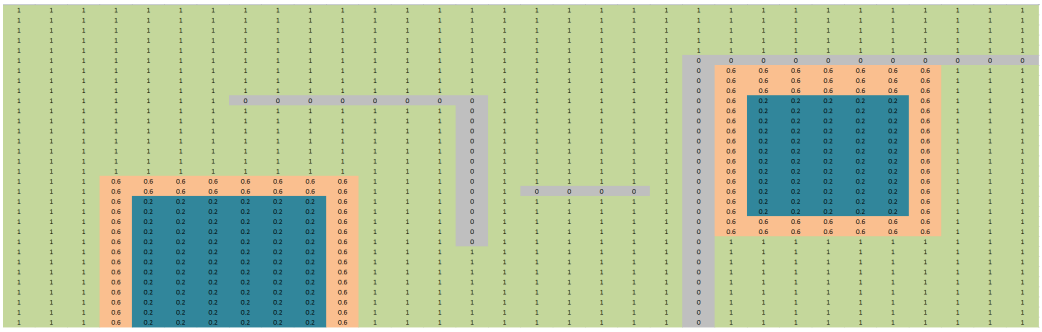

Figure 5.16: No Possible Path to Goal in Map Five

Map Six

Map six looks at how each algorithm behaves when we force it to move through a section of terrain to reach the goal. Within this map we have a large section of water with a high terrain cost and an area of sand with a lower cost than the water covering the centre portion of the map.

Both algorithms were able to reach the goal in 47 nodes and both chose to move over the lower cost sand area than the water however looking at figures 5.17 and 5.18 we can see that they both take fairly different paths through the map even though they are the same length. When asked 50% of the 18 students felt the diffusion path was more realistic while the other 50% prefered Dijkstra.

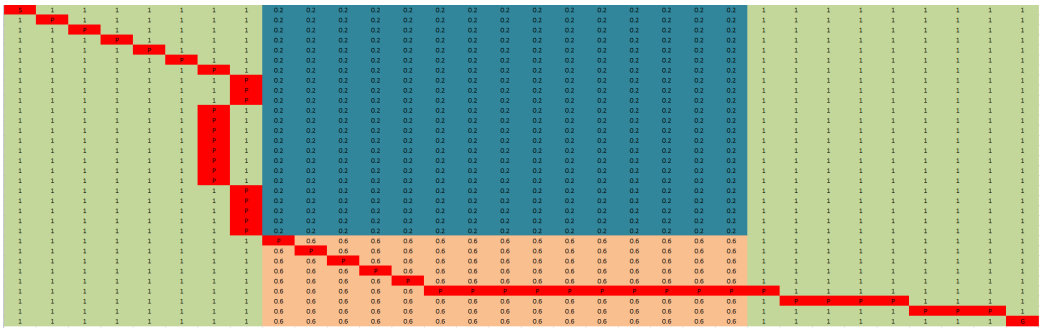

Figure 5.17: Diffusion Path Map Six

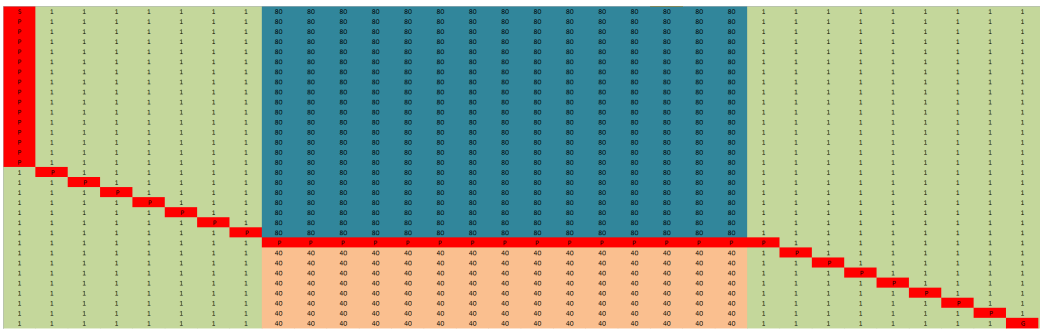

Figure 5.18: Dijkstra Path Map Six

5.2.1 Performance

When looking at the performance of the parallel implementations to solve each of the maps we can see a large difference between both the diffusion and Dijkstra implementations. Figure 5.19 shows the time taken for each of the maps.

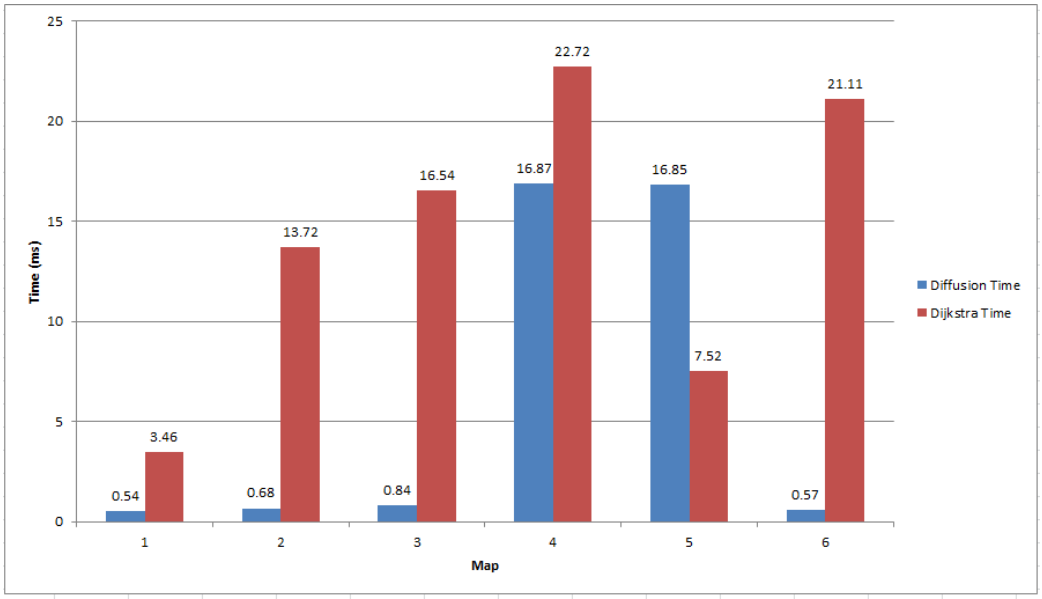

Figure 5.19: Dijkstra Path

As we can see for each of the maps apart from map five, parallel diffusion vastly outperforms the parallel Dijkstra implementation. Looking back at the results from figure 5.7 we can see that for the 32x32 maps diffusion was the quickest while Dijkstra was the slowest, what is interesting here however is the variation in time between the different maps.

Looking at diffusion, for maps 1, 2, 3 and 6 we take around 0.6 ms to find a path between the start and the goal yet for maps 4 and 5 we take roughly 27X longer to find the path. Looking at table 5.3 which shows the number of iterations of the diffusion loop required to find the path we can see that for both maps 4 and 5 we are reaching the user defined maximum number of iterations which was set to 1000. Looking at map five we can see that no path to the goal exists and as such our diffuse value will never spread to every tile so our diffusion loop will not terminate early. For map four it is possible that due to the complexity of the map the diffuse values are so small that they are not picked up by the termination function resulting in the diffusion loop running until it reaches the user defined maximum.

Map	Parallel Diffusion Iterations
1	31
2	38
3	49
4	1000
5	1000
6	31

Table 5.3: Iterations of Diffusion Loop to Find Path

When looking at the time taken by the Dijkstra implementation for each of the maps in figure 5.19 we can see a much larger variation between the time taken for each of the maps ranging between 3.46 ms to 22.7ms.

Looking at the number of iterations of the outer loop required to solve each map between the sequential and parallel Dijkstra implementations in table 5.4 we can see that the iterations required by the sequential version remains consistent at 1024 iterations for each of the maps apart from map five which takes 701 iterations. Interestingly this is the same number of nodes as are added to the queue, as we seen in table 5.2 for an nxn grid sequentially we take n*n iterations, in the case of map five we only ever add 701 nodes to the queue as a section of the map is unreachable.

However looking at the parallel version the number of iterations jumps between 32 and 219 for the different maps. Further investigation determined that this is due to the way in which nodes are added to the frontier set. At each iteration any node with a cost of less than the minimum cost plus 1 will be added to the set, in each of these maps there is a large variation in terrain

cost between nodes which will result in fewer nodes added to the set each iteration as they will not satisfy this function resulting in a greater number of iterations required to evaluate all the nodes and more time taken to find a path.

Looking at the results from table 5.2 and 5.4 we can conclude that for an nxn grid best case performance for the parallel application where each node has an equal terrain cost will be n iterations of the outer loop while worst case will be n*n should each node have a different terrain cost.

Map	Parallel Dijkstra Iterations	Sequential Dijkstra Iterations
1	32	1024
2	132	1024
3	159	1024
4	219	1024
5	72	701
6	203	1024

Table 5.4: Iterations of Outer Loop for Parallel and Sequential Dijkstra

Chapter 6 | Conclusion

6.1 Discussion of results

The results from set one showed that for the smaller graph sizes parallel diffusion vastly outperforms both the sequential and parallel Dijkstra implementations and when looking at the speed up over the sequential implementations compared to Dijkstra, diffusion results in the larger speed up over its sequential counterpart achieving speed ups of 1.2X to 59X compared to Dijkstra only achieving speed ups of 1.2X to 4X, indicating that diffusion is much more suited to a GPGPU implementation than Dijkstra. However the performance of diffusion appears to scale very poorly as the size of the maps increase resulting in both the sequential and parallel Dijkstra implementations being much quicker for the largest graph sizes tested.

When looking at the length of the paths found by both the implementations diffusion never found a path that took more steps to reach than the goal than Dijkstra and was able to finder shorter paths for two of the six maps than Dijkstra was able to.

However this does come with a trade off, in order to achieve this diffusion chose to move across areas with a higher terrain cost, whereas Dijkstra found the lowest cost path if we were to look at the accumulated terrain costs of the traversed nodes. From a games standpoint this is not always bad as we are more concerned with finding a more intelligent and realistic looking path than one with the lowest accumulated cost which diffusion was able to achieve with 57% of the 18 students asked feeling this way.

The results have also shown that terrain costs can significantly effect the performance and the behaviour of the algorithms. For both diffusion implementations terrain costs can significantly effect the time taken to diffuse the goal value throughout the map as areas of high terrain costs will spread the value much much more slowly than areas of a lower cost.

Similarly the parallel Dijkstra implementation is effected by terrain costs due to the nature of the frontier set selection method with maps containing a large amount of terrain taking significantly longer than those with no terrain, unlike the sequential implementation where performance is largely unaffected by terrain costs. From a games perspective this can make the job

of the map designers much more difficult as complex maps could significantly effect the performance of the game should paths need to be recalculated frequently.

6.2 Fulfilment of Aims and Objectives

This project has resulted in the successful implementation of two different pathfinding algorithms running on the GPGPU using the CUDA architecture. A number of test maps were designed and each algorithm tested against each of the maps in order to collect a number of metrics so that the algorithms can be compared in terms of both performance and quality.

The results collected from these tests allowed us to gain an understanding of how the algorithms performed and behaved when faced with different types of obstacles allowing us determine the factors which effect the parallelizations and the areas which could be improved. As a result of the work carried out all eight of the aims and objectives outlined in section 1.2 have been successfully completed.

In section 1.3 we posed three questions with which the research aimed to answer. The first asked "Identify which pathfinding algorithms may be suited to a GPGPU implementation?". Three algorithms were discussed and an investigation into previous attempts at parallelizing them was carried out, from this we concluded that both collaborative diffusion and Dijkstra could be suited to a GPGPU implementation. Our results obtained in section 5 provide further evidence to support this as both our diffusion and Dijkstra implementations resulted in a speed up over the sequential counterparts.

Our second question asked, "How do the algorithms compare in terms of performance and quality of path?". From the results obtained we discovered that for the smallest of the graph sizes diffusion is 6X faster in parallel than Dijkstra while Dijkstra is 45X faster than diffusion for the largest of the graphs. Through asking a number of students we also found that diffusion results in more realistic and intelligent paths than Dijkstra.

Our final question asked, "What factors effect the parallelilzation of the algorithms?". Through a series of tests we were able to determine that both of the parallel implementations can be significantly effected by the degree to which terrain cost varies within the map, with higher degrees of variation increasing the time required to find a path.

We also identified that collaborative diffusion is an inherently parallel algorithm requiring very little modifications to the process involved in order to parallelize it on the GPGPU. Unlike Dijkstra which requires significant changes to the algorithm in order to parallelize it correctly, however the introduction of dynamic parallelism within the Kepler architecture simplified this process making it easier to implement on the GPGPU.

6.3 Future Work

Both of the parallel implementations could benefit from a number of optimizations, primarily the parallel Dijkstra implementations. Even though we follow a similar approach to (Ortega-Arranz et al. 2013) with our parallelization we were unable to see similar speed ups over our sequential implementation. In their paper they report seeing speed ups of 13X to 220X whereas we have only been able to achieve speed ups of 1.2X to 4X.

There are a number of differences between their implementation and ours, biggest among them is the use of dynamic parallelism to simplify the process. It is possible that this could have had an effect on the overall performance of the algorithm however this may be a trade off developers could be willing to make due to the ease of implementation.

The other big change between the (Ortega-Arranz et al. 2013) implementation and ours is the way in which the graphs are represented and stored on the GPU. Their implementation looks at the differences between using adjacency lists and matrices to store the neighbours while we calculate each nodes neighbours every time within the evaluate frontier set kernel. Making use of either of these data structures could result in further speed ups for the algorithm.

Within their implementation (Ortega-Arranz et al. 2013) define two methods of selecting which nodes should be added to the frontier set, of the two methods we chose to implement the simpler and less aggressive method. Implementing alternate selection methods could result in speed ups as if we are able to select more nodes each iteration we should be able to reduce the time taken to find a path in maps which have a high degree of variation between terrain costs.

Finally the Dijkstra implementation could benefit from reducing the amount of branching occurring within the kernels, as we seen in section 2.4 a number of previous attempts at parallelizing pathfinding algorithms failed to see

speed ups due to the branching that was occurring. When looking at our kernels we can see a large amount of 'if' statements within each kernel, if we were able to remove these in some way we could expect to see significant speed ups.

The diffusion application could most likely benefit from making use of the memory types specific to the GPU. Due to the access patterns and the way certain grids are stored it should be more than possible to make use of both constant and texture memory which should result in speed ups as we would reduce the amount of reads to global memory which can be slow in comparison.

6.4 Critical Assessment of Project

Selecting both collaborative diffusion and Dijkstra appear to have been the correct choice and no major problems or obstacles were met with the implementation of the algorithms, the generous donation of the Tesla K40 from the NVidia Corporation significantly helped this by allowing us the hardware to take advantage of dynamic parallelism and although at the time of writing there were no resources of anyone having parallelized pathfinding algorithms using this method, the supporting CUDA documentation along with CUDA by Example (Sanders & Kandrot 2010b) and the CUDA Handbook (Wilt 2013) were more than adequate to gain an understanding of the technology.

Using the accompanying project blog to document the project and keep track of the work that needed to be carried out proved to be an effective method of managing the project. By analysing and writing up the results for each test as they were completed allowed us to gain a better understanding of how the algorithms were performing and identify any bottlenecks in the implementations, from this we were able to direct the project and create new tests accordingly allowing us to refine and improve the algorithms and better understand the factors which were effecting the parallelizations.

Overall the project went very well, we were able to remain on track with the original plan and meet all the major deadlines even being ahead of schedule at points allowing us extra time to rigorously test the algorithms, there were no significant changes in direction to the project from what we originally set out to achieve and with hindsight there would be no major changes to the overall process that was followed in managing the project. The only regret is that there was not more time available to carry out the work described in section 6.3.

Bibliography

Dijkstra, E. (1959). A note on two problems in connexion with graphs, *Numerische Mathematik* pp. 269–271.
URL: *http://www-m3.ma.tum.de/foswiki/pub/MN0506/WebHome/dijkstra.pdf*

Flynn, M. J. (1969). Very high-speed computing systems, *Proceedings of the IEEE* pp. 1901–1909.

Hart, P., Nilsson, N. & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths, *Systems Science and Cybernetics, IEEE Transaction On* **4**(2): 100–107.

Herlighy, M. & Shavit, N. (2008). *The Art of Multiprocessor Programming*, Morgan Kaufmann.

Johnson, T. & Rankin, J. (2012). Parallel agent systems on a gpu for use with simulations and games, *Latest Advances in Information Science and Applications* .

Kepler (2013). Kepler - the world's fastest, most efficient hpc architecture. Accessed: 7th January 2014.
URL: *http://www.nvidia.com/object/nvidia-kepler.html*

Manhattan Distance Metric (2013). Improved outcomes software. Accessed: 24th October 2013.
URL: *http://www.improvedoutcomes.com*

McNally, O. (2010). Multi-agent pathfinding over real-world data using cuda.

Microsoft (2012). Microsoft visual studio ultimate 2012. Accessed: 14th April 2014.
URL: *http://www.microsoft.com/en-gb/download/details.aspx?id=30678*

Millington, I. & Funge, J. (2009a). *Artificial Intelligence for Games*, Morgan Kaufmann.

Millington, I. & Funge, J. (2009b). *Artificial Intelligence for Games*, Morgan Kaufmann.

Millington, I. & Funge, J. (2009c). *Artificial Intelligence for Games*, Morgan Kaufmann.

Moore (2013a). Moores law. Accessed: 16th October 2013.
URL: *http://www.britannica.com/EBchecked/topic/705881/Moores-law*

Moore (2013b). Moores neighbourhood. Accessed: 24th October 2013.
URL: *http://mathworld.wolfram.com/MooreNeighborhood.html*

NVCC (2013). Cuda compiler driver. Accessed: 17th October 2013.
URL: *http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/*

NVidia (2013a). Cuda toolkit. Accessed: 17th October 2013.
URL: *https://developer.nvidia.com/cuda-toolkit*

NVidia (2013b). Nvidia's next generation cuda compute architecture kepler gk110, *Technical report*, Nvidia.
URL: *http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf*

Nvidia Visual Profiler (2013). Nvidia visual profiler. Accessed: 7th January 2014.
URL: *https://developer.nvidia.com/nvidia-visual-profiler*

Oak Ridge National Laboratory (2013). Introducing titan the worlds number 1 open science supercomputer. Accessed: 24th October 2013.
URL: *http://www.olcf.ornl.gov/titan/*

OpenCL (2013). Opencl reference pages. Accessed: 24th October 2013.
URL: *http://www.khronos.org/registry/cl/sdk/2.0/docs/man/xhtml/*

Ortega-Arranz, H., Torres, Y., Llanos, D. R. & Gonzalez-Escribano, A. (2013). A new gpu-based approach to the shortest path problem, *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, IEEE, pp. 505–511.

Pac-Man (1980). History of pac-man. Accessed: 24th October 2013.
URL: *http://pacman.com/en/pac-man-history/*

Repenning, A. (2006). Collaborative diffusion: Programming antiobjects, *Technical report*, University of Colorado.

Sanders, J. & Kandrot, E. (2010a). *Cuda by Example*, Addison Wesley.

Sanders, J. & Kandrot, E. (2010b). *Cuda by Example*, Addison Wesley.

Wilt, N. (2013). *The CUDA Handbook*, Addison Wesley.

Zaharan, M. (2012). Cuda memories, *Technical report*, New York University.

Appendices

Appendix A | Initial Project Overview

A.A Title of Project

Comparison of Pathfinding Algorithms Using the GPGPU

A.B Overview of Project Content and Milestones

The aim of the project is to implement, test and evaluate a piece of software which compares multiple pathfinding algorithms that will be run in parallel using the GPGPU. The algorithms will be compared against CPU based implementations of themselves and against each other in a range of different pathfinding scenarios.

For the project to be successful the following aspects must be implemented. One traditional path finding algorithm implemented on both the CPU and GPGPU. One other pathfinding algorithm implemented on the CPU and GPGPU, Test both algorithms in a range of different scenarios from simple flat surfaces with few obstacles to varying terrain with a wide range of static and moving obstacles, the accompanying report and documentation.

A.C The Main Deliverable(s)

Report showing an understanding of the project and an analysis of the data gathered comparing the chosen algorithms. To hopefully show that making use of the GPGPU for computation is a feasible option for path finding algorithms. The accompanying software containing parallel and sequential implementations of multiple pathfinding algorithms.

A.D The Target Audience for the Deliverable(s)

Game developers, AI researchers, parallel computing researchers, GPU manufactures.

A.E The Work to be Undertaken

Investigate current projects and literature which have similar goals to this and gain an understanding of their findings and what limitations and problems they faced, also to identify suitable metrics to compare the algorithms against.

Implement the chosen algorithms both on the CPU and GPGPU and test the differences in performance between the algorithms and their sequential and parallel implementations. Evaluate the data gathered to provide an understanding of the benefits/limitations of parallel pathfinding algorithms. Document the findings and the software to be included in the accompanying analysis and report.

A.F Additional Information / Knowledge Required

To complete the project and achieve the main goals, an understanding of programming for the GPGPU will be required. Knowledge of artificial intelligence and the different pathfinding algorithms will also be necessary.

A.G information Sources that Provide a Context for the Project

Information about GPGPU Programming

Nvidia - http://www.nvidia.co.uk/page/home.html

Khronos Group(OpenCL) - http://www.khronos.org/opencl/

Cuda by Example written by Jason Sanders and Edward Kandrot

Heterogeneous Computing with OpenCL written by Benedict R. Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry and Dana Schaa.

Information about AI and Pathfinding

Artificial Intelligence For Games written by Ian Millington and John Funge.

AI Game Dev - http://aigamedev.com

A.H The Importance of the Project

A.I algorithms are one of the last parts of a typical game engine to be exploited by the GPU. By researching into this area and determining if it is possible and worthwhile to make use of these techniques could benefit game developers by allowing better and more efficient AI in games which will create a better experience for the player specifically in the mobile section where costly A.I can be a problem due to constraints on processing power and memory.

A.I The Key Challenge(s) to be Overcome

The biggest difficulty may be implementing the algorithms so they run on the GPGPU. There are limited resources on parallel implementations of traditional pathfinding algorithms and implementing traditional sequential algorithms in an efficient parallel way may be a difficult challenge to overcome.

Appendix B | Project Management and Diary

B.A Gantt Chart

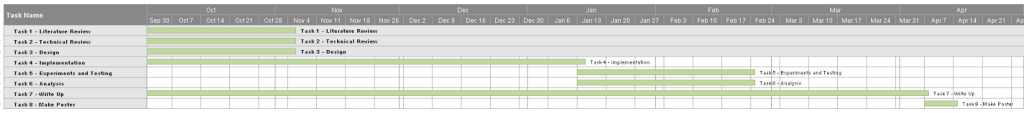

Figure B.1: Gantt Chart Showing Timeline for Project

Break Down

- Task 1
  - Literature Review: 01/10/13 - 05/11/13 (5 Weeks)
- Task 2
  - Technical Review: 01/10/13 - 05/11/13 (5 Weeks)
- Task 3
  - Design: 01/10/13 - 05/11/13 (5 Weeks)
- Task 4
  - Implementation: 01/10/13 - 14/01/14 (15 Weeks)
- Task 5
  - Experiments and Testing: 14/01/14 - 24/02/14 (6 Weeks)
- Task 6
  - Analysis: 14/01/14 - 14/02/14 (6 Weeks)
- Task 7
  - Write Up: 01/10/14 - 07/04/14 (28 Weeks
- Task 8

- Make Poster: 07/04/14 - 14/04/14 (1 Week)

A blog was also set up where a project diary and updates on the technical work along with code examples and testing results were documented.

B.B Project Diary

B.B.1 Friday 20th September 2013

Went over first draft of initial project overview and discussed what changes needed to be made before handing in on Friday 27th September.

For the following week a wiki and project plan needs to be created.

B.B.2 Friday 27th September 2013

Looked over changes made to the IPO and agreed it was ready to be handed in. Submitted it to the school of computing office at 13:30.

Discussed impact that offloading AI onto the GPU may have an effect on rendering if too much of the resource is taken up by the pathfinding. May need to be aware of and test this during the testing and evaluation stage of the project. possibly create a graph showing how much time is taken up by rendering vs pathfinding.

create gantt chart of project plan for next Friday. Look into implementing terrain, doesn't need to be complicated, don't actually need to render anything at this stage just need to be able to access the data correctly on the GPU with minimal passing of data between the CPU and GPU as this will cause a major slowdown and greatly effect performance. Look into getting interoperability working and storing the data in VBO's etc.

Keep reading papers for the lit and tech review. Start draft chapters for the week 9 review and keep a log of any technical work done.

B.B.3 Friday 4th October 2013

Went over Gantt chart and reviewed dates and times for each section. Agreed that the write up should be a continual process throughout the entire project and not just at the end of the project. Gantt chart will be updated to reflect this.

Discussed how data structures and the chosen heuristics could affect the implementation of A* if done poorly. One possible solution to this is to find

an implementation of A* that is assumed to be efficient and use it for comparisons.

Began a sequential implementation of A* for a 2D map during the week, aim is to complete this for Friday 11th October and then continue work on it to lead to pathfinding over a dynamic 3D map. Look into papers about parallel AI and continue with writing up lit review.

B.B.4 Friday 11th October 2013

Presented a draft of the sections that are to be included in the lit review. Agreed that don't need to go into too much detail about the different types of parallel systems only need to give a brief overview.

Look more closely at why using the GPU and why the particular technology could be useful for running pathfinding algorithms. highlight the differences and the advantages/disadvantages over the CPU.

Look at the different APIs that are available for programming on the GPU and give a justification about the chosen API.

Look at the different types of pathfinding algorithms which are available and what is traditionally used in games.

Highlight the advantages/disadvantages in the context of the project and look at the existing work that has been done and pick out what the findings/conclusions are and how they relate to the current project.

Discussed how the algorithms are going to be compared and what the best method of comparing the algorithms would be. Decided that A* is going to be very difficult to paralyze and that the best method if running a GPU implementation of A* would be to test it for a large number of agents and look at the scalabilty in comparison to the GPU. This has already been a few times in the past so it may be better to compare a CPU implementation of A* for a large number of agents against a GPU implementation of a diffusion based algorithm and see which has the better scalabilty and performance. The reason for this is that a CPU A* implementation is what is traditionally used in games so by comparing it against a different algorithm implemented on the GPU it may be possible to find a better algorithm for performing pathfinding within games?

Discussed how rendering could be effected by running the algorithms on the gpu for a large number of agents. Concluded that rendering is not important as the problems are the same for the CPU and GPU implementation, at this stage is has been decided that no rendering of agents will be carried out however if the project is successful and there is time left over it is something that could be implemented to make the application more visually appealing.

Aims for the coming week are to complete a first draft of the lit review in preparation for the week 9 meeting.

B.B.5 Friday 18th October 2013

Went over the work that has been done on the lit review, agreed that it is along the right tracks and to continue with it, with the aim to be completed by week 9 review. Agreed that more detail is needed on the working of the GPU as this will be specific to the implementation.

Found an A* CPU implementation that has been used in professional triple A games. This will be modified to suit the needs of the project and used as a benchmark to compare the GPU implementation of a diffusion based algorithm.

Asked to get a copy of the work on pathfinding from the computational intelligence module as this may be helpful.

Still wait on feedback from IPO from second marker.

B.B.6 Friday 25th October 2013

Went over the current draft of the background. Agreed that this should be completed by Friday 1st of November to be sent to second marker in time for the week 9 review.

For the week 9 review need to complete a report on what other work has been carried out during the project, including any practical work or other research, also need to create a revised plan of what work is to be done and estimated dates of when this will be completed.

Received feedback from IPO. Main feedback was that the project could be too complex a project for honours level, it is important to establish where this project is re-implementing existing algorithms or develops new ones.

B.B.7 Friday 1st November 2013

Not much work was done on the background for the project this week due to other coursework deadlines.

Went over techniques that could help with the flow of the document as it is felt that this could be better. This includes writing everything up and then going back at the end and adding in a couple sentences to each section to help with the flow. It may also be helpful to create a flow diagram for each section and follow this.

Also discussed the existing work chapter, rather than having a heading for each paper instead talk about the main points of the paper and use the paper as a reference. For example talk about parallel AI and some of the considerations that need to be taking into account as referenced in the paper by Timothy Johnson and John Rankin and then talk about a* pathfinding and reference the paper by Owen Mcnally. Rather than talking about the paper on collaborative diffusion in this section, discuss it in the section on diffusion.

The week 9 meeting is scheduled to be for Friday the 8th if the second marker is available. For this the draft of the background is to be sent to the second marker the day before and a 1 to 2 page report on the technical work needs to be written up.

B.B.8 Week 9 Review

Had the week 9 review with the second marker. Discussed the lit review and aims of the project and received feedback on the project.

Feedback

Changes heading on lit review section Artificial intelligence in games to pathfinding algorithms in games.

Add a section on Dijkstra's algorithm.

Test cases need to be well designed, Take as many timings as possible and get an average and use t testing to determine if the data sets are significantly different from one another.

Overall the feedback was positive and constructive, the project is on track and has clear goals about what needs to be achieved for the project to be

successful.

B.B.9 Friday 22nd November 2013

Discussed the technical work that has been done so far and went over the initial timings that were taken for both the sequential and parallel versions of the diffusion application.

Agreed that due to the nature of the diffusion pathfinding it would be unfair to compare to A* as the diffusion process returns all paths to a goal node whereas A* only returns a single path. Due to this it may be better to compare against an implementation of Dijkstra instead.

Need to come up with a plan of tests which need to be run that includes a range of challenges for the algorithm to deal with such as static and dynamically changing goals etc.

B.B.10 Friday 29th November

Discussed the list of tests that were outlined in the testing plan and went over what measurements should be recorded from the tests.

Not a lot of work has been completed on the project over the last week due to upcoming coursework and exams.

B.B.11 Friday 6th December

Looked at some of the results from Nsights performance analysis tools for CUDA and looked at how they could be used within the project to give an insight into how well the application is performing in terms of branching and memory usage.

Also decided to look at a parallel implementation of Dijkstra as well, having a quick look around Google there appear to be lots of example implementations where people have attempted to parallelize the algorithm.

Aim for this week is to read over some of the existing work that has been done using parallel Dijkstra implementations.

B.B.12 Monday 20th January 2014

Discussed the work that has been carried out during the winter break.

During the holidays i applied for an academic hardware donation from Nvidia,

they have kindly agreed to donate a Tesla K40 to help with the project. This will provide the power of dynamic parallelism which should simplify the parallel Dijkstra application and provide a significant speed up. However the card will not be arriving until February which adds an extra time pressure to the project.

Currently have the following applications completed

- Sequential Diffusion
- Parallel Diffusion
- sequential Dijkstra
- Sequential A*
- Parallel Euclidean Distance

Until the card arrives the time will be spent writing up the methodology and the implementation for the applications which have been completed. Once the card has arrived the planned tests can be carried out and the results written up.

B.B.13 Monday 27th January 2014

Until the new GPU arrives the aim is to continue writing up the methodology and the experiments that have been carried out with the current complete applications. Discussed ways that experiments should be presented and how the results should be analysed.

B.B.14 Monday 10th February 2014

The Tesla K40 donated by NVidia arrived last friday and was fitted into the testing hardware, we discussed the results from the initial tests which were carried which involved running the parallel diffusion for a range of graph sizes and discussed the results of the survey that was handed out to several students asking them which path they felt was the most 'realistic' for an agent to take through a map.

The aim for the remaining 7 weeks of the project is to implement and test a parallel Dijkstra and finish writing up the report.

B.B.15 Tuesday 18th February 2014

Went over the initial results from the parallel Dijkstra where the outer loop of the algorithm has been parallelized. The initial results are promising having seen a speed up of about 60% for the larger graph sizes.

Further work can be done to parallelize dijkstra through parallelization of the inner loop which should allow for a greater speed up.

Work can also be done to improve the frontier set selection process which could result in better performance.

The dates for the honours project have now changed as well. The poster presentation is now the Wednesday of week 13 and the hand in of the report is the 28th of April.

B.B.16 Tuesday 25th February 2014

Discussed the results from attempting to parallelize the inner loop of the Dijkstra application. Using cuda streams has resulted in a slow down of the application so further tests to determine why this is happening need to be carried out. Possible reason for this is that to simply create the streams on the GPU takes much longer than it does to simply evaluate an individual neighbour.

The paper 'A New GPU-based Approach to the shortest path problem' uses a similar method and reported a speedup of 220x, it may be worth while emailing and asking to see the source code to perform a comparison between their implementation and my own.

B.B.17 Monday 10th March 2014

Discussed the results from the following set of tests and concluded that there is not much other work that can be done on this area at this time. http://10004794-honours-project.blogspot.co.uk/2014/03/streams-vs-device-functions-2.html

All that is really left to do is test the parallel implementation on the 5 test maps and then continue writing up the results. This week will most likely be the final week of coding for the project.

B.B.18 Monday 17th March 2014

Discussed the results obtained from last weeks experiments and concluded that all the implementation, results and testing have all been completed. The remainder of the project will be focused on the write up and the poster session.

Discussed that it should be possible to use the coursework for computational intelligence to show a robot moving through the maze using collaborative diffusion.

B.B.19 Monday 24th March 2014

Discussed the poster presentation and what needs to be included in the poster. The aim is to have an initial draft by Monday 31st March.

Also need to revise the initial questions outlined during the background. Possible questions are.

1. Find a suitable algorithm for path finding on the GPU.

2. Quantify in terms of
a) speed
b) Layout of path - scalability
c) trade-off : Speed vs Path style

3. What contributes to the GPU - Mapping of algorithms to the GPU

B.B.20 Wednesday 9th April 2014 - Poster Presentation

The poster was shown and discussed at the poster presentation session

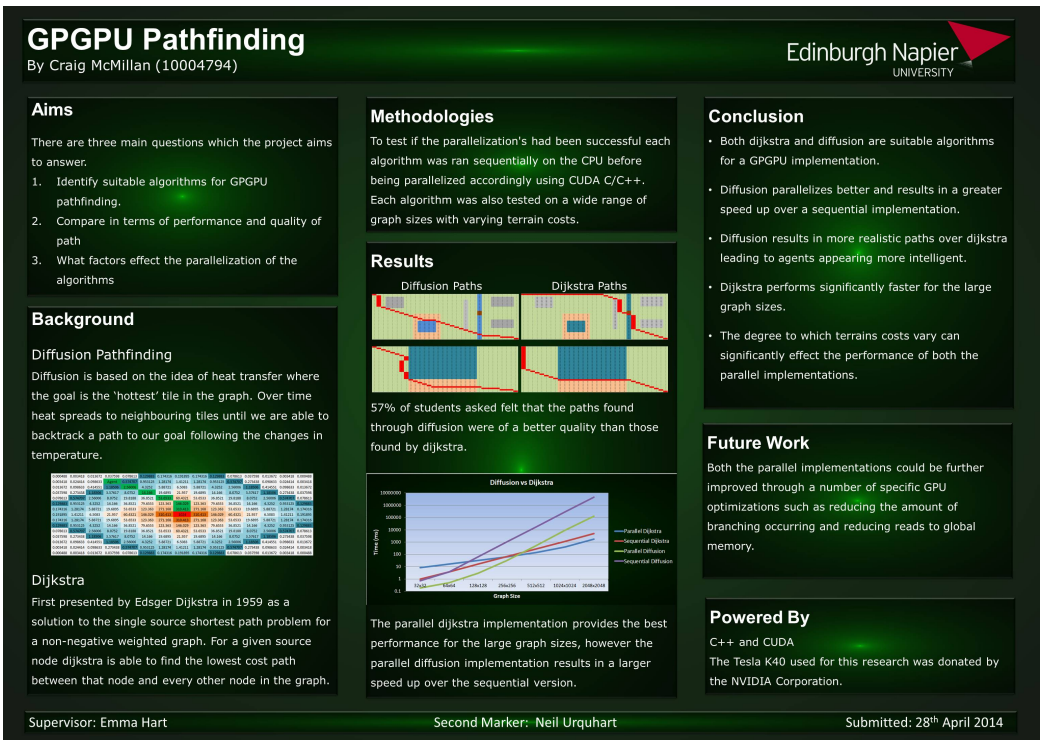

Figure B.2: Honours Poster

All that remains is the final write up and hand in for Monday 28th April 2014.