Multi-Agent Pathfinding over Real-World Data Using CUDA

by

Owen McNally, B.A.Mod Computer Science

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science

University of Dublin, Trinity College

September 2010

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Owen McNally

September 13, 2010

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Owen McNally

September 13, 2010

Acknowledgments

I would like to thanks my friends, family and classmates for all their help and support.

Owen McNally

University of Dublin, Trinity College
September 2010

Multi-Agent Pathfinding over Real-World Data Using CUDA

Owen McNally
University of Dublin, Trinity College, 2010

Supervisor: John Dingliana

This project will implement a framework to perform pathfinding over real-world data using CUDA. The system will read in XML maps from Open Street Map and convert them into a roadmap format that we can use to perform pathfinding. The system uses the CUDA architecture to perform A* searches for many agents in parallel while allowing the user to set start and goal states manually or have the system generate random start and goal states automatically. The implementation will be tested on a variety of maps with differing numbers of agents in order to ascertain the viability of using the GPU to perform pathfinding in real-world situations.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

In recent years there has been a blurring of the lines between traditional CPUs and GPUs and the work that each of them can perform. With the advent of programmable shaders and the increasing complexity and flexibility of modern graphics cards, more and more tasks that were historically CPU based are being ported over to the GPU. Many of these tasks that are now being run on the GPU are achieving superior performance compared to their CPU counterparts. All of this is due to the massively parallel processing power that is now available to us since the introduction of the General Purpose GPU (GPGPU).

GPGPU applications first appeared as scientific number crunching programs which showcased the raw performance of the devices that most users simply associated with drawing impressive graphics. In October 2006 Stanford's Folding@Home[29] project started using GPU clients when performing protein folding to achieve performance of over 70x that of their CPU contributions[40]. Soon people realised that these devices could be used for more than simple scientific calculations, and could be used to enhance not only the graphics in modern games, but also the physics and AI as well.

Games today operate on a larger scale than ever, with some real-time strategy games reaching to thousands of units battling across massive maps with tens of thousands of pathfinding nodes. In these modern games the artificial intelligence component of the game engine takes up only a small portion of the overall available time per frame. Most modern video game developers aim to produce games that run at a constant 60 frames per second, to ensure a smooth and responsive playing experience for the user. With the introduction of consoles that play games in high definition resolutions up to 1080p, and video game budgets in the hundreds of millions of dollars, production qualities and visuals are of the utmost importance. Along with this, many other aspects of a AAA video games such as networking, sound, and animation are also contending for CPU time, leaving the AI portion of the engine with somewhere in the range of 1-3ms[5] of time to perform all pathfinding calculations needed for the current frame.

As we can see, any speedup that may be available through the use of the GPU to perform pathfind-

ing computations would be useful. If we are able to reduce the time needed to calculate paths for large numbers of agents, then we can free up more time for other activities, or add in an extra feature that could put us one step ahead of the competition. And in today's fiercely competitive gaming marketplace, any advantage in performance or graphical fidelity that can be gained over the competition is vitally important.

1.2 Dissertation Overview

This work presents an implementation of the A* algorithm geared towards the CUDA platform, using graphs generated from data taken from Open Street Maps. Data is first taken from the Open Street Map servers in XML format before being parsed into a CUDA-friendly roadmap representation that can be used to perform pathfinding with the A* algorithm. This roadmap is then displayed by the testing framework using the SDL library. The user can then manually create paths through the environment or have them generated randomly by the application. Finally the user can set the number of agents in the simulation and execute the A* pathfinding algorithm on the GPU. A series of test cases were set out and we used our framework to perform multiple simulations using different variables to attempt to better understand how GPU pathfinding can be utilized to improve performance.

1.3 Project Layout

The remainder of this paper is set out as such; Chapter 2 gives background discusses the state of the art in A* pathfinding, GPGPU application, and projects using Open Street Maps. Chapter 3 discusses the design decisions taken when planning the implementation, and some of the compromises that had to be made in porting the highly divergent A* to the GPU. Chapter 4 then describes the implementation used in the final application and details each of its features. Chapter 5 gives the results of our tests and evaluates our findings. Chapter 6 then gives the conclusions we have taken from the work and discusses what might be possible in terms of future work.

Chapter 2

Background

2.1 The A* Search Algorithm

The A* Search algorithm was first proposed in [10] and aims to find the optimal path from a starting state to a given goal state by traversing along edges of a node graph, and to do this as efficiently as possible. A* is based on Dijkstra's search algorithm[19] that expands all nodes surrounding the starting state until it reaches a goal state, however A* expands states in a more directed manner using a heuristic or estimate of how far the goal is away from the current state. Nodes in the graph can represent anything from a 2-dimensional grid of tiles to arbitrary points in 3D space. The algorithm has two main components that allow it to search through the graph in an efficient manner; the open and closed lists. The open list is a collection of nodes that have yet to be expanded by the searching process, while the closed list is the collection of nodes that we have already been inspected.

As well as these open and closed lists the A* algorithm also keeps track of each nodes G and H values. These values are used to keep track of various costs for a particular node. The G value of a given node respresents the cost of getting to this node from the starting state. While the H value is an estimate of the cost of getting from this node to the goal. This H value, also known as the heuristic, can be computed in a number of different ways depending on how one wishes the search to behave. Nodes may also store their F value, which is an overall estimate of how likely the node is to bring the search towards the goal. The F value is simply a combination of the G and H values attained by adding the G and H values together.

The A* search then begins with a graph and an open and closed list, both empty. Start and goal states are then set for the current search query and the algorithm can begin calculating the optimal path from start to goal. The first step of the search is to add the starting state to the closed list and expand each of its neighbouring states. These states are then added to the open list ordered by their F values. The search continues then by selecting the next node from the open list with the lowest F value. The node is removed from the open list and compared to the goal state. If it is not the goal state then it is added to the closed list. We then take each of its neighbours and insert them into their corresponding position in the heap. This process continues until we either reach the goal state,

or there are no nodes remaining on the open list. If we reach the goal state then we have computed the optimal path to the goal from the initial starting state and we can return the completed path. If our open list has become empty then there is no path from the starting state to the goal state.

2.1.1 Heuristics

Heuristic values cannot be arbirarily computed and must follow a strict set of rules to be useful in the A* search. For a heuristic to work correctly, it must be both *consistent* and *admissable*. A heuristic can be considered *consistent* if when calculated for a given node N it is non-decreasing as we move along any given path towards the goal. This means that the H value for a node close to the goal state cannot be greater than the H value of a node further away from the goal. As well as this, for a heuristic to be useful it must also be *admissable*. For a heuristic to be classed as *admissable*, the heuristic fuction must never over-estimate the cost of reaching the goal from the current node. This means that the estimation should be optimistic, and should always be less than the actual cost of reaching the goal from the current state. If the heuristic is not *admissable*, the search may never find the goal and end up in a dead end of a graph or get stuck switching back and forth between two states endlessly[19].

A common way to compute the heuristic in a grid or tile based search where every move has a cost of 1 is to use a heuristic known as the "Manhattan Distance". Manhattan Distance is a form of taxicab geometry that basically treats the tiles like the grid layout of the streets in Manhattan. In this way the H value between two given tiles is computed by summing the absolute differences of their coordinates [39].

Another simple heuristic that may be used to estimate the distance between the current node and the goal node is the straight line or Euclidean distance. This is performed by simply using the Pythagorean formula to get the disance between the two points in either 2D or 3D space. This is a commonly used heuristic because it can never overestimate the distance between the two points because the straight line distance between two points is the shortest distance possible.

Other more complicated heuristics can be used to solve search problems such as the 8-Puzzle Problem in which a set of tiles must be slid around a board to arrange them in a certain order (either ascending of descending). With search problems such as this a 'distance' heuristic is not really applicable and we must use a different measure to determine how 'close' we are to the goal state, such as how many of the tiles are on the same square as they would be if the puzzle was in a solved state.

2.1.2 Open and Closed Lists

The closed list structure in the A* algorithm can be implemented in a number of ways, and simply records whether or not a given node has already been inspected or not. Nodes that have been inspected will be on the closed list, while nodes that have no will be absent. A common way of implementing such a structure is a simple hash table or bitset. Note that just because a node is not on the closed list does not mean that it is on the open list, and vice versa.

The open list on the other hand is a far more interesting structure. Nodes are added to the open list as the algorithm expands each edge of the current node being inspected, however they cannot be simply added to a hash table or bitset as with the closed list. For the sake of efficiency the open list is generally implemented as a priority queue structure, with nodes with lower F values coming before nodes with higher F values. In this way the node which has the lowest combined G and H value will be the next to be inspected by the algorithm. This works because nodes that are close to both start and goal states will lie somewhere in between the two, and moving towards the goal will make the G values increase and the H values decrease, while moving further away from the goal state will see an increase in both the G and H values. Because of this these two values help to guide the search towards the goal state, and allow us to avoid unneccesary states that are out of the way as we move towards our goal. While it is possible to have an open list that is unordered, such lists are highly inefficient as extracting the element with the lowest F value requires searching through the entire list.

The most common ways of implementing the priority queue are to use a simple linked list structure or a heap structure. With a list structure the program must keep track of the head of the list, i.e. the node with the current lowest F value. This allows the node with lowest F score to be easily removed from the list and then replaced by the next node in line. However, inserting new elements into a linked list can expensive, especially as the size of the list increases. This can become troublesome on larger graphs where searches regularly require a large open list. To insert a new node into the linked list, the program must iterate through the list from one node to the next and check the F value of each node. When it finds a node with a higher F value than the new node being inserted the new node can then be placed in this position in the list, and the remainder of the list appended after. This search through the list can prove quite costly over large structures, with insertion time increasing linearly as the size of the linked list grows.

Another way to structure the priority queue is to use a heap. In a heap each node has two children, and each child has a higher F value than its parent node. In this way, the top node on the heap always has the lowest F value, and as you traverse down the heap the F values of the nodes at different levels increase. Note that there is no specific ordering of nodes other than that each child is of greater value than its parent. To remove the node with the smallest F value from the top of the heap is slightly more complicated than with a linked list. When the top node is removed from the heap it must be replaced by a node from from a bottom of the heap. This node is then bubbled down through the heap by testing whether it has a higher F value than either of its children. If it has, it is then swapped with the child and the process is repeated on the next level down. This process continues until the node is in its correct position. Similarly when adding a node to the heap it is added to the bottom of the pile. It is then compared with the value of its parent node, and if it has a smaller F value, the two nodes are swapped. Again, this process continues up the levels of the heap until the node reaches its final position.

The heap structure can provide a significant preformance increase over a typical linked list with insertion and removal times of O(log n)[19]. We can see why this is the case if we closely examine what's going on when performing a insertion or removal. If, for example, we wished to insert a new node into our open list which is an ordered queue of 1000 nodes, and our queue was implemented as a

linked list, it would take on average 500 comparisons to find the appropriate place in the list for our new node. However, if the queue was implemented as a heap structure, we would need to perform an average of 1-3 comparisons to find the correct place for our new node. This is because as we move down the levels of the binary heap the number of nodes at each step doubles, and also the heap, unlike the linked list, does not require that each of our nodes is perfectly ordered, just that it has a greater value than that of its parent[19].

However, binary heaps are not perfect. When using a linked list the removal of the first item from the list is extremely simple, however when removing the first item from a heap structure we are required to re-sort the heap completely. This is achieved by moving the last node in the heap to the top and then bubbling this node down through the heap until it reaches its final position. This would take an average of 9 comparisons on a heap with 1000 nodes, which is quite a lot of work compared to the linked list implementation.

The reason then that the heap implementation is still more efficient than the linked list is the number of times during a search that an element is removed from the open list, compared to the number of times an element is added. On each iteration of the A* algorithm only a single node is removed from the open list, while in a grid based map 2-4 nodes could be added. If the graph is something more complicated than a square grid such as a hexagonal grid or a free graph, then the number of addations to the list could be anywhere from 0 to 10 or more. Because of this, the number of elements being added to the open list during any iteration will almost always be much greater than the number of elements being removed. It stands to reason then that because the heap structure performs insertions more efficiently, it will be the more efficient implementation overall. This turns out to hold true in the majority of cases, however the benefit of using a heap greatly increases as the size of the graph increases, and using a heap structure for the open list on a small graph can actually be slower than a simple linked list implementation. As graph sizes increase and the number of potential nodes in the open list reaches the tens of thousands, other approaches such as multi-level buckets [8] or Fibonacci heaps [1] can prove more efficient than simple binary heaps. However in this work we chose to stick with binary heaps as only one of the graphs being tested had over ten thousand nodes.

2.2 The CUDA Platform

The CUDA (Compute Unified Device Architecture) platform is a general purpose computing architecture designed for performing tasks that would be normally executed on the CPU, on the GPU. The applications are usually massively parallel and attain their high performance from executing hundreds or thousands of threads at once. CUDA is based on a SIMT (Single Instruction Multiple Threads) instruction set that can have multiple threads perform the same operation over a large set of data, much like a vector processor. In this way CUDA can execute multiple threads that all perform the same task on differing sets of data, and can execute these threads all in parallel. By doing this CUDA can outperform single-threaded, or even multi-threaded CPU applications at the same task by having greater throughput, due to having more threads "in flight" at any given time. When using a CUDA

enabled GPU it is seen by the system as an external device that can execute highly parallel workloads. The host system then communicates with the device by copying over data and then giving the device a task to perform. The device then executes this task and returns the results to the host.

When running code on a CUDA GPU, each thread must execute the same instructions. That is to say, there is no way to tell one thread to execute code A, while another executes code B. CUDA programs consist of a 'kernel' that is executed on each thread on the GPU in parallel, and the code in this kernel defines what actions a single thread should perform. Different threads can work on different sets of data then by using their unique thread IDs. The thread IDs can then be used as a parameter for the data set, and have each thread working on its own seperate piece of the overall problem.

While threads cannot execute completely different code from one another, they can diverge. Diverging happens when there is a conditional statement such as 'if', 'while' or 'for' in the code and one thread follows one path while another thread follows a different one. When this happens with two threads in the same group (or 'warp'), these threads can no longer be run in parallel and their execution must be serialized. This is quite detrimental to the overall performance of the program, so thread divergance should be avoided at all costs.

CUDA threads are then grouped into warps, with each warp containing 32 threads. Warps are grouped into 'blocks' that are assigned to each Multi-Processor. The programmer can specify the number of blocks in the application and the number of threads per block. Threads and blocks can also be assigned in up to 3 dimensions, so you can have a 2x3x4 block of threads running on 6x7x8 blocks per grid. Each of these dimensions is specified by x, y and z attributes. Threads and blocks can be initialised like this so as to allow for easier thread ID calculations when using different sets of data. For example if you are working on a one dimensional set of data you can simply use one dimensional threads and blocks. However, if the data you are working on uses two or three dimensional arrays you can use two or three dimensional threads and blocks.

2.2.1 Processing Units

Each CUDA enabled graphics card has a number of processing units that make up the CUDA architecture. At the lowest level, each graphics card has a set number of CUDA cores. Each of these cores can execute one thread at a time and each has its own floating point arithmetic logic unit (ALU). These CUDA cores are then grouped together into Multi-Processor (MP) blocks, each of which contains 8 CUDA cores. More recent GPUs such as the Nvidia GTX series that support CUDA 2.0 can have varying number of cores per MP. For example the Quadro FX580 used in the testing of this implementation contains 32 CUDA cores, split into 4 Multi-Processor blocks of 8 cores each, while the Nvidia GTX470 has 14 MPs each with 32 cores each. Each Multi-Processor then contains extra elements to help manage its own set of cores such as a thread scheduler and on some newer GPUs each Multi-Processor also has its own double precision floating point ALU that is shared between all of the cores.

The warp is essentially the basic unit of work on the CUDA architecture as no less than one warp

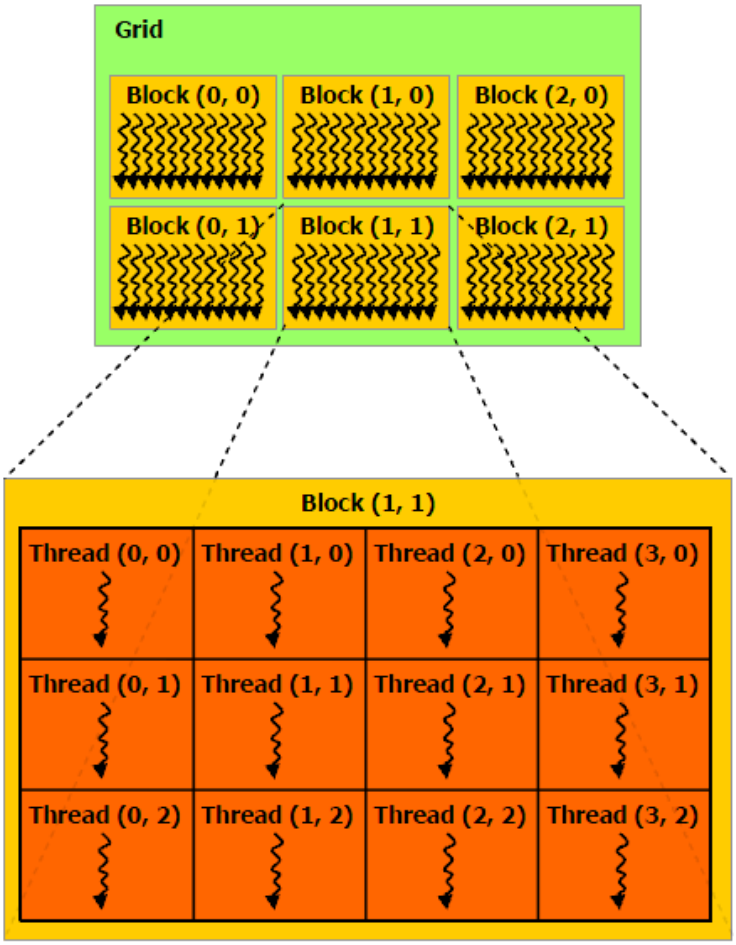

Figure 2.1: CUDA Thead and Block Organization.

of threads can exist at any one time. That is to say that if you attempt to create a CUDA application that uses less than 32 threads, CUDA will automatically create a warp of 32 threads, assign x number of threads to the work of the application, and the remaining threads in the warp will simply be empty threads that perform no operations. In the best case scenario, each thread in the warp contains the exact same instructions to be executed on different sets of data, without any diverging branches, and can be executed in parallel as one chunk.

At run time, each warp is then assigned to a Mutli-Processor which will schedule it on the CUDA cores it manages. Because each Mutli-Processor only has 8 CUDA cores, and each warp has 32 threads, each warp is executed in stages using a pipeline. The process of pipelining involves splitting up the task of performing an operation into multiple stages. For example a simple pipeline could involve only three stages. The first stage of executing an instruction could be to load the values in memory, the second stage might be to perform the operation using the ALU, and the third would be to store the results in memory. By performing the task in this pipelined fashion we can hide any latencies that

may occur the execution of an instruction, and increase the overall clock speed of the system because instead of an operation that takes 9us to execute in a single stage, it now takes three stages of 3us.

In this way each warp is split into 4 batches, threads 0-7, threads 8-15, threads 16-23 and threads 24-31. At the first clock cycle the first batch no.1 will be begin executing. At the second batch no.2 will begin executing. At the third batch no.3 will begin executing, and at the fourth batch no.4 will begin executing. At the fifth cycle batch no.1 will finish execution and store its results. In this way the entire warp is being executed at the same time, although it is not quite truely in parallel. This works slightly differently with the newer GPUs that can have 32 cores on a single MP.

2.2.2 Memory Hierarchy

There are a number of different types of memory on the GPU that can be assigned and accessed by CUDA applications, each with differing properties and capabilities. The types of memory available in the CUDA environment are Global, Constant, Shared, Local and Texture. Each of which will be explained in the following sections.

Global, Constant and Texture Memory

Global memory space acts much like global memory in any standard C program. It is accessable to any thread being executed at any given time and is persistant over all threads during the kernel execution. Global memory may or may not be cached depending on the compute capability of the GPU. Compute capability is a measure of the CUDA feature set that each GPU supports. For all devices of compute capability up to 2.0, global memory is not cached. However for device of compute capability 2.0 and above global memory is cached[22]. Global memory is relatively slow as it is off-chip, with access times in the range of 200 to 300 cycles of latency[33], and allows both read and write access from threads.

Constant and Texture memory are similar to global memory in that they are globally accessable across blocks and threads and are persistant across kernal launches. However, they do not provide write access outside of the host environment, meaning that they are set once on kernel launch and then cannot be changed during the running of the CUDA application. Both constant and texture memory are cached on all CUDA devices regardless of compute capability.

The difference between constant and texture memory lies in how they are optimized for different types of access. The texture cache is optimized for 2D spacial locality because textures generally consist of a 2-dimensional array of values, so threads in the same warp that read texture addresses that are close together will see greater speedups due to a greater number of cache hits [22]. The hardware also provides a number of additional capabilites when using specific instructions to access texture memory such as filtering, normalization, and special addressing modes. These are most commonly used to perform special operations on the texture in hardware before they are returned to the calling function [22].

Constant memory on the other hand takes up only 64KB of memory on a CUDA device. It is used for variables defined in the kernal to be of constant type and provides cached access to all threads

running on the device. Reading from constant memory is as fast as reading for a register for all threads in a warp, as long as they are all reading from the same memory location. However if the threads read from different memory locations their accesses must be serialized.

Register, Local and Shared Memory

Registers are on-chip memory locations that each thread uses to store local variables in the kernel. They are very fast and act similarly to any other registers one would find in a common CPU. Each register set has a scope of a single thread, meaning that one thread cannot access registers belonging to a different thread. Registers retain their value for the lifetime of the associated thread.

Local memory on the other hand is situated off-chip. The term 'local memory' is not meant to imply a spacial locality to the CUDA core, but rather the fact that the scope of the memory is at the thread level, so each thread has its own local memory. Local memory is read and write accessable, has a lifetime of the associated thread, and similar to global memory is not cached on devices of compute capability less than 2.0.

The final type of memory is Shared memory. Shared memory like the name implys has a scope of an entire thread block. Shared memory is located on-chip so is therefore quite fast, has a lifetime of the entire block, and is accessable to every thread in the block in both read and write mode. Shared memory can then be used for threads to communicate or cooperate with one another, as values can be set in the shared memory by one thread and then read by another. In this way an entire block of threads can work as a single cohesive unit to achieve a common goal, even if their work requires inter-thread communication.

2.3 Open Street Map

Open Street Map is a global collaborate project that provides a free editable map of the whole world. OSM was founded in July 2004 by Steve Coast with the aim of providing free geographical data such as street maps to anyone who wants them. The project was started because most mapping services such as Google Maps that you may think of as free actually have some legal or technical restrictions on their use which prevent them from being used in certain ways. Open Street Map has taken inspiration from sites such as Wikipedia and provides an interface that allows users to easily edit any map they wish. As well as this OSM also provides a revision history to allow rollbacks of unintended or malicious changes to the map. User can manually change waypoints, known as 'nodes', or even upload their own GPS information to the database to allow for more accurate editing of the map.

Open Street Map data has not only been used in many academic and non-academic projects alike, but has also been the subject of many studies on the accuracy of crowd sourced data and the legitimacy of the community driven model that is used to update and create the maps in the first place. Many useful services such as Open Route Service[25] base their entire system on data provided by Open Street Maps and its contributors, showing that there is strong confidence in the accuracy and robustness of the OSM data. The University of Heidelberg currently has a project which is working

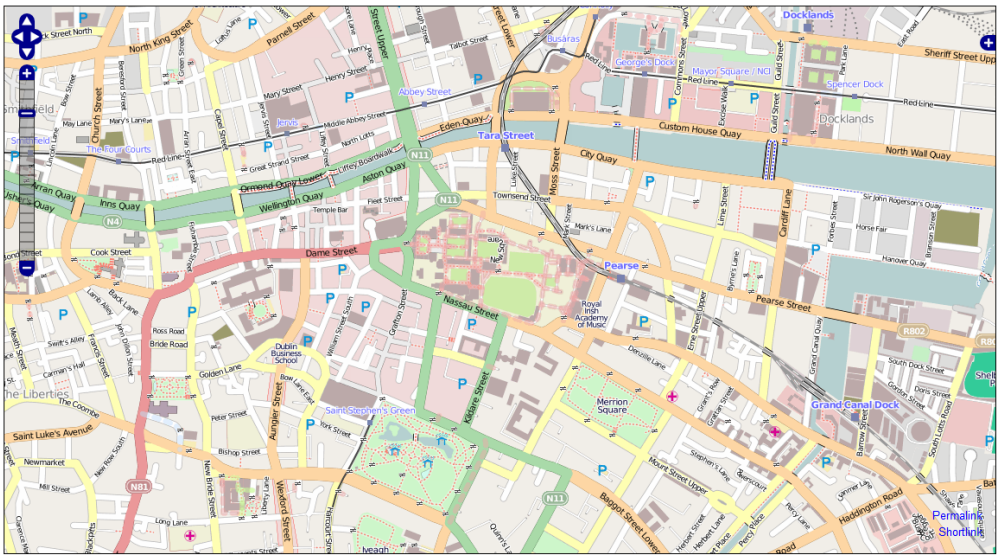

Figure 2.2: Open Street Map view of Dublin City Center.

on combining Open Street Maps data with a digital elevation database to create service oriented interoperable 3D city models [7]. Currently only available for cities in Germany, the website provides a 3D map view as part of a Java applet in which the user can move around and zoom in and out, much like in Google Earth. As well as this, entire papers have been dedicated to the idea of Open Street Maps[9], user generated content and why it is beneficial to us all.

2.4 State Of The Art

Traditionally, GPUs were used solely for graphical functions such as matrix manipulations and large scale vector calculations. However, in the last few years there have been an increasing number of computationally intensive tasks migrating from the CPU to the GPU to take advantage of the available computing power. A prime example of this would be the PhysX physics engine[23] that was bought by Nvidia and now runs on their CUDA enabled cards for increased performance. While products like PhysX and other physics engines such as Bullet[34] have begun to realise the increased performance that's available by putting their systems on graphics hardware, there have not been many attempts to perform pathfinding or artificial intelligence tasks in general on the GPU. This may be due to the fact that GPUs have not always been a very hospitable place for applications that require a lot of branching, with 'if' statements only becoming properly supported in recent shader models. However there is no longer any reason why this should be the case. While performance increase would not be as great as compared to an application with few conditional statements, the performance increase for a divergent algorithm such as A* should still be noticable.

2.4.1 Pathfinding

Many improvements have been made in recent years in the area of pathfinding, with new implementations providing the ability to discover new environments[15], learn about their changing surroundings[17] or perform partial planning of routes using graph abstraction [6]. However through all of this one thing has remained relatively constant, and that is the use of the original A* algorithm[10] as the basis for these approaches. While the original algorithm may seem crude and unflexible since it was developed, it has maintained its status as the premier algorithm when it comes to performing fast, efficient discovery of optimal paths.

In[15], the D* algorithm was presented which behaves like a reverse A* algorithm. This algorithm is designed to work in unknown environments where an agent is presented with a start and goal position but does not know the intermediate geography of the scene and must learn as it navigates through the world. This algorithm is quite popular in robotics especially in search and rescue type applications where a robot would enter a collapsed building that may be too dangerous for a human being in an attempt to locate survivors. It can also be used in strategy games where there is a fog-of-war present and the agent does not know what sort of terrain lies between it and its objective. While this algorithm is excellent at allowing agents to find their way through unknown environments, in our work we are performing a search over a graph in an environment that is completely known to the agent at search time, so this technique is not of any real benefit to us.

Papers such as [17] and [16] present us with A* based algorithms that allow agents to move through dynamic scenes with ever changing environments, and in real-time. This is achieved by abandoning the planning of the complete optimal paths in favour of planning a shorter path that simply brings us closer to our goal. Using a 'lookahead' we can specifiy to these algorithms how far ahead in the graph we wish to search, and we can adjust this according to how much time we have available for planning. If we need the planning done quickly we can lower the lookahead, while if we have more time available we can make use of it to plan further in the environment. Not planning the full path obviously has its downsides as well and will usually result in the path being sub-optimal. However, lots of work has been done to reduce this to a minimum and algorithms such as Learning Real-Time A* and Real-Time Adaptive A* can generate paths usually no more than 10% worse than the optimal solution[14]. While these algorithms provide an interesting avenue to explore on the GPU, the fact that the basic A* algorithm is as of yet unproven in real-time scenarios on the GPU meant that we decided to put them aside in favour of exploring the usefullness of the classic A* algorithm, although a very basic implementation of a Real-Time A* algorithm was tested with the implementation.

Another area in which A* has progressed in recent years is with path abstraction. Path abstraction involves creating a number of different resolutions in the node graph at which searches can be performed. In [36], a method known as *clique abstraction* is used to group nodes into sets according to their connectivity. The grouping follows two simple rules - any nodes, up to a maximum of four, that are all interconnected are all abstracted into the same group. Any nodes with only a single connected edge (known as an 'orphan' node) is abstracted into the same group as its neighbour. Using these rules we can guarantee that any two nodes that share a parent node in a higher level of abstraction

can be reached in a single move from one another, with the exception of orphan nodes. This method continues until every node in the graph has been abstracted to the next level, producing a simpler graph that we can perform pathfinding over. These levels of abstraction are used to find higher level paths that can then be refined down to lower levels as needed, thus allowing us to perform pathfinding for sections of our journey, rather than for the whole thing in one execution. While this method of path abstraction is undoubtedly useful for planning paths over large maps, in this project we were more interested in testing the raw performance of the GPU when executing the pathfinding algorithm, specifically in relation to the number of nodes in the graph, so using techniques such as this to reduce the overall number of nodes over which we must search would be counter-productive.

Despite all these improvements in the functionality of the classic A* algorithm, our goal in this project was to test the viability of performing pathfinding in general on the GPU, and since A* is the basis for most modern pathfinding algorithms and is still largely untested on the GPU, we decided that investigating the standard A* algorithm would provide us with a greater insight into how the GPU handles pathfinding in general than testing a particularly algorithm that may provide specific pitfalls due to its complexity. Performing pathfinding on the GPU is still quite a novel idea, and while varitions of the A* algorithm may work well on modern CPUs, there is no gaurantee that porting these improvements to the GPU will provide the same benefits.

2.4.2 GPGPU and AI

Many traditionally CPU related tasks have now successfully made the jump to the GPU, and as new architectures continue to blur lines between CPU and GPU more and more applications are finding a new home on the GPU. Multinational companies such as Nvidia have also been supporting this initiative by releasing GPU AI technology previews[21] demonstrating what is possible on their hardware as well as giving developers a taste of what will be possible in the coming years. While there has been very little reasearch done in the area of highly divergent path finding algorithms such as A*, the following papers presented us with some good background on multi-agent simulations and crowd behaviors.

In [4] we are presented with a GPU based multi-agent navigation system that uses vector obstacles based on the RVO model as presented in [38]. Each agent in the simulation has a goal to which is must move towards, while also avoiding static scene geometry and dynamic objects such as other agents. The RVO model had to be adapted for the GPU and the simulation consisted of up to 10,000 agents navigating through the scene simultaneously. Techniques such as a hash based nearest neighbour checks and loop unrolling were implemented to achieve an overall speedup of almost 48x over CPU implementations. The hash map nearest neighbour check was used to examine the area surrounding an agent and perform local avoidance of dynamic objects in the scene based on their positions and velocities relative to the agent. This provided a speedup of 4x over standard proximity checking methods alone.

In [32] we see another method of creating realistic crowd simulations, this time using a continuum based global path planner. This method of performing path finding was first developed by [37] and

uses a dynamic potential field to perform global navigation with moving objects such as other agents or scene geometry, without the need for explicit local avoidance. Spatial algorithms such as this create a flow-like movement through the environment which agents follow like a stream following the curvature of a rock. This can lead to some interesting emergent behavior such as lane formation and queueing. The main function of the system is solving a non-linear partial differential equation known as the Eikonal Equation that describes how waves of the potential field propogate through the environment. Luckily there has been some research done on solving this quickly on the GPU [13] and they have shown it is possible to simulate up to 65,000 agents on a commodity GPU using this method. The demo itself presented in the paper shows 3,000 agents in the scene running at 20-25 fps on a Radeon HD4870.

The Nvidia paper [3] which was the basis for much of the work of this project, presented a method of performing global pathfinding using classic A* and Dijkstra algorithms. The algorithms were modified to take advantage of data parallelism wherever possible and used clever implementations of edge lists using adjacency tables to achieve high cache hit rates during the inner A* loop. The data used in their testing however consisted of some unusual graph sizes, ranging from just 8 nodes up to only 340. This represents a graph size of less than 18x19 nodes, which is not consistent with the large graph sizes used in modern video games which would be upwards of 2,000 nodes. As well as this the paper only gives the results of 5 graphs that were tested, each with a single number of agents equal to the number of nodes in the graph squared. It was our aim to expand on this work with some more meaningful figures and performing tests on a wider range of graphs with differing numbers of agents.

2.4.3 AI in Games

With the cost of game development increasing on a year to year basis the trend to go to middleware companies for specific components of your game engine is becoming more attractive by the day. Game AI is not immune to this and companies like Havok and Autodesk have produced incredibly robust and cross platform AI solutions that game developers can essentially buy off the shelf and integrate state of the art AI tools and algorithms into their games. Havok's latest incarnation of its Havok AI module includes automatic navigate mesh generation, streaming support for naviagation through different parts of the level, full support for dynamic environments and dynamic local steering [11].

Autodesk's Kynapse AI solution also provides an impressive feature set including 3D Pathfinding, spatial awareness for AI characters and team coordination. This allows for impressive levels of immersion where teams of AI bots can work together in environments they have never seen before and still be able to naviagate through the scene seamlessly while also being able to identify key areas where a threat might approach or function as an escourt for a player controlled character. Autodesk have also released a number of white papers detailing some of their thoughts on traditional pathfinding techniques such as A* and its place in modern video games. These papers provide some useful insight into how the industry leaders view the field of game AI as well as provide information on where and when optimizations can be used to give us maximum benefit [30].

These products are the pinnacle of current game AI and provide an impressive array of features for

the modern game developer to handle. They have been used in a multitude of games on all platforms [2][12] and allow developers to quickly and easily get to grips with the state of the art in AI techniques without having to invest the time and money to develop such complex systems themselves.

Chapter 3

Design

3.1 OSM XML Format

The Open Street Map servers deliver the map data from their servers in a standard XML format. These XML maps are either retrieved from the Open Street Map website by selecting a region to export, or by downloading larger map areas from one of the many Open Street Map storage sites. Downloads directly from the OSM website are limited to a maximum of 50,000 nodes to reduce the load on their live servers.

The main component used to parse the raw XML data from the OSM maps is the Xerces XML library. The Xerces is a library used for parsing, validating, serializing and manipulating XML documents and is part of the Apache XML Project [35]. Xerces provides robust, flexible and efficient parsing of XML documents and in our work is used to quickly and efficiently parse the OSM data and convert them into an intermediate format for further processing.

The OSM XML format consists of three main elements; Nodes, Ways and Tags. Along with each of these, every OSM XML map begins with an 'osm' element specifying the version of the document and the program used to generate the XML file, and a 'bounds' element giving the minimum and maximum latitudes and longitudes of all the nodes in the file.

These Node and Way will then make up the roadmap graph that we will be performing the pathfinding over. Each node in the map will be converted to a state in our graph, and each way will be converted into a collection of edges between states. In this way, we can build up a full graph representation of the XML map provided by the OSM servers and use it to search for routes between different locations.

3.1.1 Nodes

A node in the Open Street Map database can represent many things. A node in its most basic form simply specifies a specific point in the world with a latitude and longitude. Nodes can then be part of a group that defines a road or railway, or can be tagged to represent a public service such as a

```
<osm version="0.6" generator="CGImap 0.0.2">
 <bounds minlat="53.1959600" minlon="-6.1096500" maxlat
 <node id="89053144" lat="53.1973572" lon="-6.0975252"
28T23:18:25Z"/>
 <node id="458490131" lat="53.1957894" lon="-6.1081295"
 <node id="458490135" lat="53.1958683" lon="-6.1079126"
 <node id="458484754" lat="53.1985739" lon="-6.1084566"
 <node id="322711169" lat="53.2025423" lon="-6.1056700"
29T00:43:22Z"/>
 <node id="458487724" lat="53.1974091" lon="-6.1055718"
 <node id="458484712" lat="53.1989927" lon="-6.1081038"
 <node id="232274166" lat="53.1976198" lon="-6.1100914"
28T23:16:11Z"/>
 <node id="89052731" lat="53.2010270" lon="-6.1110276"
06T12:55:06Z">
  <tag k="highway" v="traffic_signals"/>
 </node>
 <node id="254364910" lat="53.2023762" lon="-6.1004343"
28T03:38:56Z">
  <tag k="created_by" v="Potlatch 0.8"/>
  <tag k="name" v="Ideal Stores"/>
  <tag k="shop" v="convenience"/>
 </node>
```

Figure 3.1: An Example of an OSM XML file.

post office, a historic monument, or even a local shop or pub. In this way nodes are the atomic building block of the Open Street Maps database on top of which all other more complex elements are constructed.

Each node in the OSM document contains a unique global ID of integer type, greater than or equal to 1. These ID's can be used to identify any given node in the OSM database and are numbered individually as they are added to the system. While the ID's are globally unique with respect to other node IDs, they are not unique with respect to 'way' IDs. This means that it is possible to have a way and node with the same ID in a given map[26]. Node IDs are persistent on the OSM servers so that even if a node is moved or modified its ID will remain the same. Deleted node IDs are not recycled and will not be used again unless the deletion of the node is undone. This removes the possibility of new nodes added to the system having conflicting IDs with nodes that have been deleted and then re-added.

Along with their unique ID, each node in the system has a specific latitude and longitude to give it a position on the OSM map of the globe. The latitude and longitude values of each node are stored as floating point numbers accurate to 7 decimal places, with latitude in the range of -90 to +90, and longitude in the range of -180 to +180.

Finally each node also contains zero or more 'tag' elements. This tag element is used to store any additional data about the node that might be relevant to the map rendering software. Each tag contains a key and a value ('k' and 'v' respectively) which are commonly used to store the software used to create the node, but are also used to specify that a node represents an object such as traffic lights on a highway, the name and type of a shop, or the web address and phone number of a restaurant.

Each node also contains some information that is not useful for the purpose of our work such as

the user who created the node, their user ID, whether the node is visible or not, a version number for the node, the changeset that the node belongs to, and a timestamp of when the node was created.

3.1.2 Ways

Ways in the Open Street Maps database are defined as a directed list of nodes. Because of this, each 'way' describes a generic path and can then represent anything from a road, railway track, or river to a coastal boarder or outline of a building or park. Ways can then be further grouped into 'relations' that define higher lever uses of collections of ways such as specific train routes. However, they are beyond the scope of our work. Each way may then consist of from 2 to 2,000 nodes [28] and defines a linear path that has the same use or properties. Areas can be defined by creating a 'closed' way which has both the first and last nodes in the way the same, and tagging it as such.

Like a node, each 'way' then has a unique global ID that is unique between all other ways, allowing it to be identified from any other way in a map. Ways do not contain their own longitude and latitude coordinates, but their position is defined by the nodes which they contain. The nodes that are members of a given way are then contained in a list of 'nd' elements, each of which has node ID 'ref' attribute. Each ref then specifies an individual ID of a node which is part of a way. Using this list, the map renderer can draw each way by beginning at the start of the node list and connecting each node to the next with a line until it reaches the last node in the list.

Similar to nodes, ways can also contain tag elements with key and value pairs to help describe the particulars of the way. Ways usually contain more tag elements than nodes due to the many different classifications a way may have. For example, a road representing a street may have tags for the name of the street, the classification of the road (primary, secondary, tertiary) and whether it is a one way road or not. Many of these tags also have default values, such as the 'oneway' tag which defaults to 'no'.

Again there are a number of superfluous attributes attached to each way element that we do not use in our implementation such as the user who added the way, the changeset it belongs to, and the timestamp at which it was created.

3.2 XML Parsing

The parsing of the Open Street Map XML document is then performed by the Xerces XML library. There are two main types of parser that are used when parsing an XML document; DOM parsers and SAX parsers, both of which will be described in the following sections.

3.2.1 DOM Parsers

Document Object Model (or DOM) parsers perform their task by scanning through the XML file and creating a hierarchical object model in memory to represent that file. Once the file has been completely parsed and stored in memory, the user can then move through the tree of nodes accessing

different elements or attributes at will, regardless of whether the original information was in a tree format or not.

This is obviously advantageous if the original data being in the XML was in a hierachical tree-like format, however that is not quite the case with our OSM documents. While there is a hierarchy present in the documents they are organised in more of an array or list structure, with many repeated elements and very little depth to the hierarchy.

The DOM method of parsing an XML file also requires that the entire document be read into memory before any of the information in the document can be accessed. This proved to be quite troublesome for our purposes as many of the files tested during the implementation of this work were quite large in size. For example the OSM XML data for Ireland consisted of 1.7million nodes and almost 4million edges. This resulted in an XML file in the region of 380MB in size. To completely parse this entire file using the DOM model and store the resulting model in memory took over 5 minutes of parsing time and 4GB of RAM. For these reasons the DOM parsing method was deemed to be too cumbersome.

3.2.2 SAX Parsers

Simple API for XML (or SAX) parsers are slightly more complex to use than the DOM model, however they provide greater flexibility and efficiency when working with large files. Instead of creating an entire model of the document in memory, SAX parsers work by providing a stream interface as the parser scans through the XML document, allowing the user access to the various elements and attributes of the document on the fly. In this way the user can create a handler that tracks the current location in the XML document, and then retrieves whatever relevant information they need from the various elements and attributes.

This SAX method of parsing an XML file was more suited to our needs when implementing the OSM map parser due to the fact that the OSM map data is largely represented in list format. Also, because a large amount of the data in the OSM document is irrelevant to us, using a SAX parser allows us to easily bypass unwanted attributes without them being fed into a bulky object model.

3.3 A* using CUDA

Traditional object oriented implementations of the A* algorithm involve many different classes and structures. Objects are defined to represent nodes in the graph, the graph itself, paths traversing the graph, the priority queue, and even the open and closed lists. Due to the limitations of CUDA C, many of these structures cannot be created or used in the traditional way. CUDA C requires that all data structures be declared and initialised before the kernel begins to run, and allocating new memory during run-time is not allowed. Because of this we are required to allocate the maximum possible space for each of our data structures before our program runs to ensure that we do not run out of memory during run-time. This is a design choice made by Nvidia that trades space complexity for performance. With all the data structures in place and no need to allocate or delete memory

during the running of the kernel, CUDA applications can achieve high performance by sacrificing some flexibility and usability in terms of high level concepts such as classes and memory management.

3.3.1 Roadmap Format

The roadmap then constitutes of all the elements in the system that describe the graph over which we are performing pathfinding. This consists of three main elements; the node list, the edge list, and the adjacency list. Each of these are, in their most basic form, simply a large array of floating point or integer values. The structure of these arrays was first used in the Nvidia paper [3], although the structure has been optimized for our own purposes. Each of the main lists will be explained in detail in the following sections.

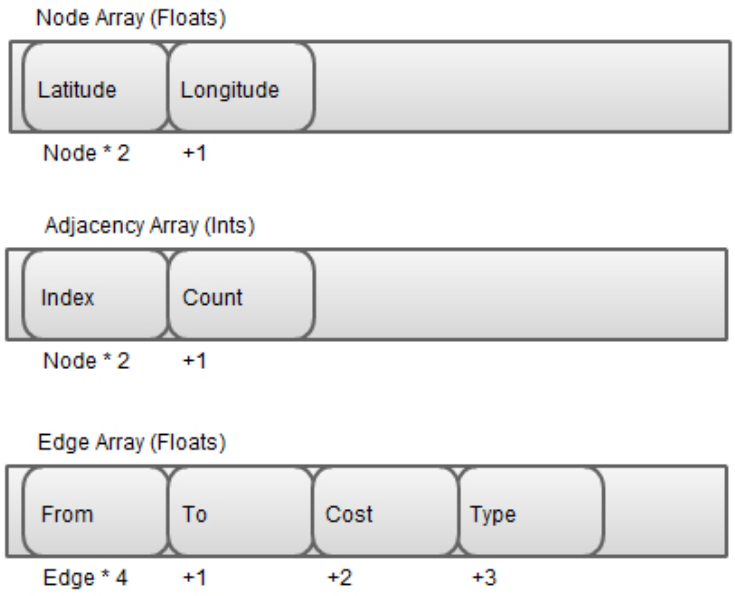

Figure 3.2: The Roadmap Format Used.

We use these large arrays of floats because of the nature of the graphics cards we are working with. GPUs are designed to operate on large data sets and have many hardware optimizations for working with floating point values. This is because most graphics workloads consists of performing the same set of instructions on large sets of points or vectors, which are typically defined as floating point numbers. More modern GPUs such as the GTX470 provide support for double precision floating point operations with version 1.3 of the CUDA toolkit. However, each Multi-Processor on the GPU has only one double precision floating point unit which is shared by all threads on the MP. On the other hand each core has its own single precision FPU, which allows for each of the threads being processed to perform single precision floating point operations simultaneously[22]. Because of this it is much more efficient to use single precision floating point numbers rather than the more accurate double precision floats.

This format is used to take advantage of CUDAs coalesced memory reads. When performing reads from global memory CUDA reads data in chunks of 32 or 64 words. What this means is that if a number of threads in a warp all request data from within the same block, their requests can be amortized into a single read instruction that encapsulates all of the data requested. This gives huge performance increases as instead of having to wait for multiple fetches from global memory, all the threads in the warp can be grouped together and simply make a single fetch that will return all of the requested data[22]. As well as this the roadmap is accessed using texture memory. This allows for fetches to be cached so that we do not waste cycles refetching data that has been recently used.

The Node List

The node list simply comprises of an array of floating point values describing each node in the graph. Each node element is made up of two floating point values, so that a single node takes up two floats in the array. For example node 0 is described by the two floating point values at index 0 and 1 in the node list array, while node 5 is described by the values at index 10 and 11. These two values represent the latitude and longitude of the give node. In this way nodes can be indexed in the array by multiplying their node number by 2, and their latitudes or longitudes can be accessed by adding 0 or 1 to the resulting value. These values are rounded to single precision as the Open Street Map coordinates go slightly beyond this, however single precision is perfectly accurate for our purposes.

The Edge List

The edge list used in the implementation is the largest part of our roadmap. It consists of four floating point values and edges in the list are accessed by multiplying the ID if the desired edge by 4 and adding 1, 2, or 3 to access the various values of the floats in the structure.

The first value in an edge element corresponds to the node ID from which the edge begins. While this is stored as a floating point value, the actual node ID itself is an integer and must be cast as such before use. The second value is the ID of the node where the edge ends, which must also be cast to an integer. The third value in the edge list is the cost associated with traversing the edge. This could be computed in a number of different ways such as the Euclidean distance from the starting node to the end node, or by a function of the type of edge and the transportation methods allowed by the agents. The final value in the edge element describes the type of the edge. When converting from OSM map data this value is used to classify the edges as different types of roads. Each road type, such as primary, secondary, tertiary or unclassified, is given a unique value associated with it and this value is then stored in the fourth position in the edge element so that we may query it during searches.

The Adjacency List

Unlike the node and edge lists, the adjacency list works more like a look-up directory for nodes and their associated edges in the edge list. Each element in the adjacency list consists of two integer values. The first value is an index into the edge list of where this node's set of edges begins. The

second value is a count of the number of edges associated with this node. Because the edge list is serialized, each subsequent edge belonging to a node occurs four positions past the current edge. In this way the adjacency list will give for a given node, the position in the edge list where its first edge element begins, and the number of subsequent edge elements that belong to this node.

3.3.2 Agent Data

As mentioned earlier, each agent in system acts independently and without interfering with one another's paths, so each agent must have its own private set of data to keep track of its individual A* search. This comprises of G values for each node in the graph which will be unique for each different start state, open and closed sets, a priority queue structure and a parentage array.

The G values for each node is simply an array of floating point values with the same number of elements are there are nodes in the graph. Using this, the agent can quickly query or update G values for any node in the system. The parentage array also consists of the same number of elements are there are nodes in the graph, however this array stores integer IDs of the parent node rather than floating point costs. In this way each node that has been expanded keeps track of which node caused it to be inspected, so that we may trace a path back to the start state if necessary.

The priority queue array again has the same number of elements as there are nodes in the graph, however each element in the priority queue consists of a struct with two members; an integer ID for the node, and a floating point F value. The integer ID allows us to identify which node this value refers to, while the F value itself keeps track of the nodes combined G and H score. Because we keep track of G and F values in this way, saving the H value of each node is unnecessary and would just take up extra memory.

Finally there are the open and closed sets. During our testing we implemented these set in two different ways. The first using a bitset to save memory in favour of more complex setting and checking of set membership, and the second using full byte chars to sacrifice memory in favour of faster setting and checking.

Bitsets

The open and closed lists were first implemented as bitsets, where each bit in the set is associated with a single node. If this bit is 1 then the node is part of the set, if the bit is 0 then the node is absent from the set. These take up in the region of one eighth of the total number of nodes, in bytes of memory. To check if a node is on a certain list then, we need to perform a small number of calculations. The bitsets themselves are made up of an array of *chars*, so to determine which char in the bitset contains the bit we are interested in, we need to divide the index of the node by 8 (the number of bits in a char). It should be noted that while shifting right by 3 positions would seem like a more efficient choice, upon performing a number of tests it would seem that the CUDA compiler automatically converts divisions by a power of two into shifts, as the timings when testing each of the cases produced almost identical results.

Then to determine which bit in the char is associated with the node we need to *mod* the index of the node with 8. Once we have these two pieces of information we can then extract the associated char from the array. We do this by using the result of the division as the index, and then taking the returned char and shifting it to the right by the result of the mod operation. This will leave us with the bit were are interested in the least significant bit position. With the bit in place all we need to do then is to perform a bitwise AND with the char and a char of hex value 0x01. This will blank out all the bits except the one we are interest in. Then we simply test the resulting value, if it 1 then the node is in the set and we return true, if it is zero then the node is not in the set and we return false.

Setting a bit in one of the bitsets works similarly. We perform both the division and mod by 8 to assert which char and bit we are interested in, and then copy the char at this position from the bitset array. However, we then create a temporary char with the value of 1 and shift this left by the result of the mod. This gives us a char will all zeros except for a 1 at the bit we are interested in. If we want to set the value in the bit array to true, we can then simply perform a bitwise OR with our temporary char and the char from the bitset. This will result in the bit we are interest in being set to 1, while all other bits in the char will remain untouched. If instead we wish to unset the bit for this node, saying that it is no longer in this set, we must first invert all the bits in our temporary char so that all bits will be 1 except for the bit we are interest in, and then perform a bitwise AND with the char from the array. This will cause the node associated bit to be set to zero, while leaving the rest of the bits untouched.

Fully Byte Chars

Our second implementation simply used a full char to represent whether a node was in a set or not. If the char had a value of 1 it is in the set, and if it had a value of 0 it is not. This implementation is much simpler than using bitsets as it does not require the use of shifting and bitwise manipulation to perform operations on the desired bit. However, this does mean that for each node in the set we require a full 8-bit char, which is very wasteful of space. So the idea with this implementation was then to increase our efficiency by sacrificing memory in favour of extremely quick checking and setting of membership. In the end this was the implementation that we went for. After running a number of tests we concluded that the saving of space gained from using the bitsets did not make up for the extra instructions that had to be executed each time and check or setting of a bit occurred. While using a full byte char did not result in a huge performance increase, the speedup was noticeable.

3.4 Memory Requirements

To calculate the total amount of memory required for our kernel to run we must find the total memory required for a single agent, multiply this by the number of agents in the system, and then add to the total memory requirements of the roadmap.

Roadmap

The node array for the roadmap then requires two floats for each node in the system, the adjacency array requires two integers for each node, and the edge array requires four floats for each edge. The formula for calculating the roadmap size is then -

$$\begin{aligned} NodeArray &= 2 * sizeof(float) * N \\ EdgeArray &= 4 * sizeof(float) * E \\ AdjacencyArray &= 2 * sizeof(int) * N \end{aligned}$$

$$TotalRoadmap = (2 * sizeof(float) * N) + (2 * sizeof(int) * N) + (4 * sizeof(float) * E)$$

Where N is the number of nodes in the graph, and E is the number of edges.

Agents

Each agent then requires a float for each node in the graph for its G values, an integer for each node in the graph for the parentage array, two floating point values for each node in the graph for its F array, and a character value for each node for both the open and closed sets. The total size requirements for a single agent are then -

$$\begin{aligned} G &= sizeof(float) * N \\ F &= 2 * sizeof(float) * N \\ P &= sizeof(int) * N \\ Sets &= sizeof(char) * N \end{aligned}$$

$$SingleAgent = (3 * sizeof(float) * N) + (sizeof(int) * N) + (2 * sizeof(char) * N)$$

The combined size of the roadmap plus the agents must then be less than the total memory on the GPU. This results in the maximum number of agents depending mainly on the number of nodes in the graph, as when this increases the size of each agent increases as well.

3.5 Issues

One issue that we discovered while implementing this system is that floating point numbers do not have enough accuracy to accommodate the large integer values being used by Open Street Map to give their nodes unique IDs. Because of the sheer number of nodes that must have unique IDs in the OSM database (a total of 747,513,499 at the time of writing [27]), some node IDs are rounded to the

nearest value that is possible to be represented using a floating point number. This is due to floating point numbers being better suited to represent numbers containing many decimal places rather than large integer values.

To accommodate for this, each node being read in from the OSM XML data is assigned a unique 'alternate' ID beginning from zero. As each node is read in from the file it is assigned a unique ID by being added to an array of integers. The index of the array corresponds to the new alternate ID, while the value of the of integer at a given index corresponds to its original OSM ID. A second 'map' structure is also created to convert original OSM IDs to the new alternate ones. This is simply created by using a map object from the C++ STL and using the original integer values from the OSM data as the key, and the new alternate IDs as the corresponding value. Using these two data structures we can easily convert back and forth between the original OSM IDs and the alternative IDs used in our roadmap.

When reading in data from the OSM XML files there were also issues with the accuracy of the longitudes and latitudes. While the values stored in the XML were accurate to 7 decimal places, when converting them to floating point numbers the last digit would sometimes round itself up or down. Luckily due to the accuracy of these numbers this rounding did not seem to have any significant measurable impact on the performance of the implementation. When drawing the node positions on the screen all nodes appeared within an acceptable tolerance of their real positions on the OSM map.

It should also be noted than when using outside sources such as OSM, we are essentially at the mercy of the accuracy of their data. If a part of a major road is not tagged correctly in the OSM data, then it will not be used correctly in our application. For performing benchmarking tasks such as ours this isn't really a big problem as we are just generating random start and goal states. However, if our system needed to guarantee that the routes returned would provide the fastest way to get from point A to point B in the real-world, such as for an emergency services routing system for example, then this might become an issue.

Chapter 4

Implementation

The application works in a number of stages that will be explained fully in the following sections. First the raw XML data taken from the Open Street Map servers is parsed into an intermediate data structure. This data is then converted into the CUDA friendly roadmap. The user can then set the start and goal states using the testing framework and the GPU will perform the A* algorithm on the data for the given number of agents. The results are then returned to the testing framework where they are displayed to the user.

4.1 Parsing the XML Data

When processing a new file using Xerces, we first need to create a new parser object. As we are dealing with relatively large files that could potentially take up a large amount of memory using a DOM style parser, we instead chose to use a SAX interface to parse the OSM data as described in section 3.2.2. Using the SAX interface allowed us to quickly and easily pick out the specific pieces of information in the XML document that were of interest to us while discarding any superflous information in the document that would not aid us in performing the pathfinding.

4.1.1 Setup

To create the SAX interface using Xerces we implemented a custom handler class that derived from the Xerces *DefaultHandler* class. This *DefaultHandler* class contains methods that perform the bare minimum processing and error handling to effectively parse an XML file. We can then override the *startElement* method of the *DefaultHandler* class. This method is called every time a new element is encountered in the XML file, so by overriding it we can provide our own implementation of how the handler should process the individual elements in the XML.

In the *startElement* method we are then given access to the name of the element currently being parsed by the SAX parser. Using this variable we can check the name of the element which is being parsed and act accordingly based on what actions we need to perform on the element in question.

However, before we perform any real parsing we first need to count the number of nodes in our XML file. To do this we simply run through the file using the SAX parser and increment a counter every time we encounter an element with the name *node*.

Once this has been done we can then set up some of the basic data structures required by the program to parse the XML. These data structures are in an intermediate format that provides a stepping stone between the pure XML representation in the OSM document and the floating point arrays required by the CUDA section of the implementation. While these intermediate structures are not an essential part of the system, they provide a more structured approach to parsing the XML into node and edge objects, rather than directly transferring the values into a large array.

An array of simple *node_data* structs that contain a floating point longitude and latitude is then created to manage the node information from the XML document. We also create an array of *edge_data* structs to hold the edge information for each node. Each of these *edge_data* structs contains two integer IDs signifying at which nodes the edge originates from and ends, labelled as 'to' and 'from'. Each struct also contains a float 'cost' for the edge and float 'type' field that allows us to specify extra information about the classification of the edge. Along with this we also create an array of integers to translate alternative node IDs into their original OSM IDs. In this array each index corresponds to the alternate ID of the node, while the value at the index corresponds to its original ID in the OSM database. We then parse the file again and begin to fill these structures.

4.1.2 Node Data

Filling the *node_data* array with the node information is a relatively straight forward process. We simply parse through the XML until we come across a *node* element and then retrieve its longitude and latitude from the element attributes. During this process we also assign each node with its alternate ID. Alternate IDs are simply determined by keeping a counter starting at zero and incrementing it as each new node is read in to the system. We keep track of these IDs by adding the alternate and original pairs to both the 'alternate-to-original' and 'original-to-alternate' structures so that we may translate back and forth later in the application.

4.1.3 Way Data

Reading in *ways* then is slightly more complicated. The edges of the graph connecting the two nodes are then added to the database in pairs. We do this by moving through the list of way nodes with a two node 'buffer'. At each pair of nodes, we create two new *edge_data* structs and set their 'to' and 'from' fields to the corresponding node IDs. These two are then added to the *edge_map* for each of the nodes. The edge map contains an array of vectors, one for each node in the XML file. As new edges are read in for each node, the edge is added on to the vector corresponding to that node. In this way when the parsing is finished the *edge_map* will contain a vector of *edge_node* structs for each node. Each connection the node has to another node in the system will be represented here as an individual *edge_struct*. As we are adding these edges to the associated node's edge vector, we also keep track of

the IDs for each node in the current way by adding their IDs to a vector. We can use this information later to add extra information to each edge on the *way*.

Finally we need to look at the *tag* element for the current *way*. This *tag* element contains extra information about the *way* to help describe its real world use. For us, we are only interested in whether or not the *way* is part of a road that has a specific classification. To do this we simply check the *tag* element attributes for the OSM keyword *highway*. This OSM keyword is used to assign roads and ways to different groups depending on the quality and classification of the road in the OSM database. If a *highway* key is found, we then check the value of the key and assign the edges a value in our system based on the classification. The OSM values we check for are *primary*, *secondary*, *tertiary* and *trunk*. Once we have identified what classification this way had, we can then iterate through the edges that were part of the way and assign them the appropriate roadtype value.

4.1.4 Roadmap Generation

Next we need to convert our array of *edge_node* vectors into a single serialized *edge_node* array. This is done by creating a new array of *edge_node* structs which is the total size of the edge vectors combined. While doing this we also want to create and initialize our *adjacency* list. This list acts as a map into our serialized *edge_node* array so that we can easily look up the associated edges for any node in the system. To do this then, we simply iterate through the array of *edge_node* vectors that we created from the XML data stored in the *edge_map*. We set the current position in our new edge array as the index for the current node in the adjacency table, and set its edge count as the size of the vector. We can then iterate through the vector itself and copy all of the data from each of the *edge_node* structs in the vector to the new serialized *edge_node* array. When complete we then have our serialized *edge_node* array along with adjacency information, which gives the index into the edge array where this node's edges begins, and the number of edges associated with this node, for each individual node in the system.

To complete the parsing of the OSM XML data then, we only need to copy the data from our intermediate structures to the final floating point and integer arrays. We first need to create the three arrays for our final node, edge and adjacency information. The node and adjacency arrays are arrays both twice the size of the number of nodes in the system, the node array being an array of floating point values, while the adjacency array is an array of integers. This is because each piece of node and adjacency data contains two unique attributes. The edge array on the other hand is an array of floating point values, which has a size of four times the number nodes in the system because it has four attributes. To fill our finals arrays then, we simply iterate through our intermediate structures and add each of them to the final array using offsets for each element in the structure. For example in the edge array the 'from' is stored at (e*4), the 'to' is stores at (e*4+1), and the cost at (e*4+2), where e is the current edge number. Once this operation has been completed we are then ready to use the data in our CUDA pathfinding algorithm.

4.2 Displaying Maps

Figure 4.1: The Testing Framework.

Once all of the of XML data has been read into the system we can then begin to display the map on the screen so that the user can set up and visualize routes. To do this we simply convert the latitude and longitude coordinates from the nodes into screen space coordinates. We can compute this using the boundary information in the XML file. This information stores the minimum and maximum values for both latitude and longitude of all the nodes in the XML. Using this information we can calculate the width and height of the of the data being respresented by the XML and from this can create a ratio that converts from the original longitude and latitude values to screen space coordinates. This is done by simply dividing the width and height of the screen in pixels by the width and height on the map area. While this is not the most accurate convertion possible it provides a perfectly usable visual interface for the user and for the purposes of this work provides adequit usability.

4.2.1 SDL and SDLDraw

Two main libraries are used to draw the map on the screen. These are the standard SDL library[31] and in addition the SDLDraw library[18]. The SDL (Simple DirectMedia Layer) is a bare-bones graphics library that provides a robust, flexible and easy to use API that allows the user to access and draw on-screen primitives. The SDLDraw library then adds some additional basic shapes and

functionality to the system, allowing the user to draw lines and circles by simply specifying end points or position and radius. This was used to help speed up the development of the visual interface and we only required it be a functional representation of the data in question, without requiring a visually impressive display.

The maps are then drawn to the screen by using SDLDraw's *DrawLine* function. Because each edge in our map data provides us with 'to' and 'from' coordinates all we then need to do is to iterate through the edge data and call the *DrawLine* function for each edge with the 'to' and 'from' coordinates as parameters. We also perform a check on both of the coordinates to ensure that they are inside the screen space being drawn, as the bounds limits supplied in the OSM XML document are not always completely accurate and some nodes can appear outside of the drawable area after being converted to screen space. While drawing the edges we can also perform a check on their road type and use this to colour code each edge depending on their classification. This allows us to be able to visually distinguish between low priority unclassified or tertiary roads and high priorty primary or trunk roads.

4.3 Setting Up Routes

Once we have the map displayed on the screen, we can then proceed to have the user set up the desired routes on which the agents in the simulation should travel. We do this by defining a set of start and goal points from which the agents should begin and end their paths. The number of start points does not have to be equal to the number of goal points, nor does the the number of start and goal points have to be equal to the number of agents desired in the simulation. If the number of start and goal points is less than the number of agents in the simulation, the agents will be split into groups, each group following a different path. If there are more routes than agents in the simulation however then the agents will just follow one path each up to the total number of agents, any extra paths being ignored.

4.3.1 Manually Setting Routes

Starting points for the agents can then be added to the simulation by pressing the 's' key. This tells the program that we are now beginning to select nodes from the graph from which we would like agents to begin their search. A message will appear in the console window alerting the user that they are now adding starting points to the simulation. Nodes are added then by simply clicking on the location of a node on the display. To accomodate for the fact that nodes are essentially a single pixel in size and therefore very difficult to accurately click, the program will then automatically take the x and y coordinates of the mouse click position and match it to the closest node it can find to these coordinates. Because of this no matter where you click on the screen *some* node will be added to the list, because the program will always select the closest node to the location in which you clicked, even if that node is quite far away. This is determined by a simple Euclidean distance formula.

Nodes are added to the goal list in much the same way. To set the application to begin added goal

nodes we simply press the 'g' key and begin clicking on nodes on the map that we would like to be used as goal nodes in our search. It should be noted that for setting up very specific routes, the first goal node will match up with the first start node. The second goal node will be paired up with the second start node, and so on throughout all the start and goal nodes. If the number of nodes in either the start and goal lists differs, then the nodes on the smaller list will be reused by wrapping back around to node 0 to accomodate for the missing nodes. For example if the user adds three starting nodes and two goal nodes to the simulation, goal node 0 will be paired up with starting node 3 to create the third route.

4.3.2 Random Routes

As well as adding start and goal nodes in this manner, it is also possible to ask the application to create a random set of start and goal points for the simulation. We can do this quite easily due to the fact that we are using alternate node IDs ranging from 0 to the totaly number of nodes in the map. To create a number of random paths then we simply press the 'r' key. We are then promted by the program to enter the number of random paths we require. Again, this does not have to be the same as the number of agents in the system, however the random path generator will always select the same number of start and goal node for the simulation. While state IDs are generated randomly, there is no guarantee that all start and goal states generated will be unique, however for our work this is not really an issue. The application then simply generates a random number between 0 and the number of nodes and assigns this ID to either the start nodes or the goal nodes. Once this process is finished the application will display a message telling the user that the random generation of start and goal states is complete.

4.4 CUDA Setup

Once we have set all of the desired start and goal states, we can then begin to perform the actual pathfinding. To do this we press the 'a' key and then tell the program the number of agents we wish to have in the system. The first step in this processes is to then set up the appropriate data structures for each of our agents in the system. This entails creating G, F, Open and Closed lists for each agent as described in section 3.3.2. Each of these lists is then part of a higher 2-dimensional array construct, in which the first dimension allows us to specify which agent's data we wish to access, and the second dimension gives access to the elements in the list themselves. In this way each element in the first dimension points to a different list associated with a specific agent. We can then index into each of the arrays using the thread ID to access that thread's associated array.

The mechanism to create two dimensional arrays such as these in CUDA can be confusing at first glance due to some unusual details of how the *cudaMalloc* function works. The *cudaMalloc* function takes a void** pointer and a size as a parameter, and returns a pointer to an address in the CUDA device's memory space. Because of this we cannot allocate space for the first dimension using *cudaMalloc* and then loop through this array using *cudaMalloc* again to allocate more space for each

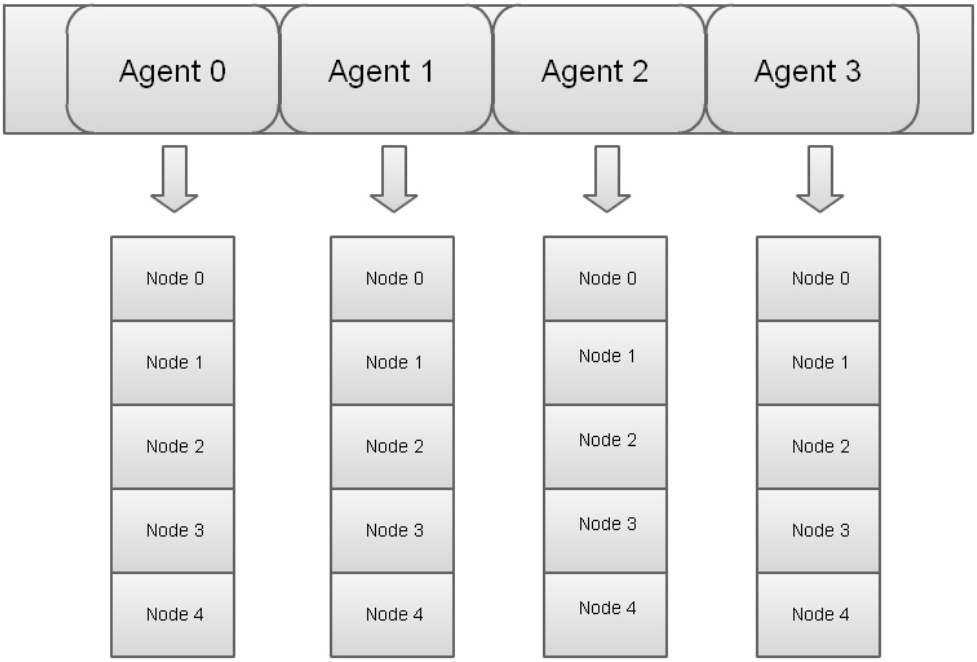

Figure 4.2: 2-dimensional Agent List Setup.

of the agent's arrays. This does not work because the first dimension that we allocated creates space in the device's memory, so we cannot assign the agent specific arrays to these memory locations because we do not yet have access to the CUDA device memory. Instead we must allocate the first dimension on the host itself. We can then loop through each of the elements of this array using *cudaMalloc* to map space on the device for the second dimension. The array on the host will then contain the memory locations of the second dimension arrays on the device. We then simply malloc space for the first dimension on the device, and copy over the locations from the host array, leaving us with a two dimensional array on the device.

We also need to allocate space on the device for the result array. This is structured in the form of a one dimensional array with the same number of elements as there are nodes in the graph. This result array is then used to keep track of the parentage of each node in the graph. We keep track of this by saving the ID of a node's parent node in this result array. As the A* algorithm moves through the graph each new node that is added to the open list has its parentage value set to the ID of the node that caused it to be expanded. In this way when we find that we have reached the goal node and the search has finished, we can then check the parent of each node starting with the goal and follow the trail all the way back to our starting node. By doing this we can quickly and easy generate a list of nodes that constitute the optimal path returned by our algorithm.

As well as this, space is also allocated on the device for the individual start and goal states for each agent, which are simply two large one dimensional integer arrays. These arrays are then filled with the values passed in by the test framework, or are randomly generated according to the number of

routes specified during the setup. Finally the roadmap arrays for each the nodes, edges and adjacency are transferred to the CUDA device themselves before the kernel is executed.

4.5 The Kernel

The kernel then consists of an optimized A* algorithm modified to run on the CUDA platform. The first thing we must do when each kernel starts is decide what data we need to be working on. To do this we need to compute the index of the thread. Luckily, CUDA provides a very simple way to calculate an index of any given thread in the launch. Each thread has access to a number of variables that can give information about the grid and block in which this thread is located. The *threadIdx* variable tells us a unqiue index for this thread inside of its block, while the *blockIdx* variable gives us a unqiue ID for the block itself. We also have access to a *blockDim* variable that gives us the dimensions of the blocks in the grid. Each of these three variables also has an x, y and z component that allows us to use two and three dimensional layouts of threads and blocks. So to calculate the x component of the current thread, we need to to multiply the *blockIdx.x* by the *blockDim.x* and add the *threadIdx.x*. The y and z components are computed in the same way. On top of these three variables there is also a *gridDim* variable that is at most two dimensional and allows us to check the dimensions of the grid in blocks. However, this is not used in our implementation.

After asserting which data we should be working on we then need to set the size of our priority queue to 1 and add the starting node to the queue. We then set the flag to show that the starting node is on the open list. The open and closed lists work as full byte chars, as described in section 3.3.2, where each char in the set is associated with a single node.

The priority queue in our program was implemented as a heap structure. As mentioned before we cannot dynamically allocate space while the CUDA kernel is running so the priority queue has space allocated up to its maximum size before the launch. To keep track of how many element are in the heap then, each thread keeps its own integer count of how many items are currently in the queue. Each element in the queue consists of two parts; an integer index of the node in question, and the F value of the node. The reader will remember from 2.1 that the F value consists of the cost to get to this node, the G value, added to the estimated cost of reaching the goal, the H value. The main extract and insert functions in the implementation are based on the pseudo-code provided in Nvidia[3].

4.6 The Main Loop

The very fact that the main part of the A* algorithm is a loop speaks to the fact that this is highly divergent algorithm which would not usually be associated with a SIMT architecture such as CUDA. The main loop's terminating condition then is that the priority queue must always have at least 1 element in the queue, and as soon as it is empty the kernel will terminate (after doing some small cleanup operations).

On each iteration of the loop, the top node from the priority queue is extracted from the heap, removed from the open set and added to the closed set. We then check to see if this node is the goal node which we are searching for. If it is the goal node then we can finish our search and return the resulting parentage array, however if it is not we must add the edge nodes associated with this node to the priority queue. To get the edges for the node we have just extracted, we need to get the index into the edge array of where these edges begin, and a count of how many edges are associated with this node. All of this information is stored in the adjacency array. To access the index and count for the associated edges we perform texture lookup using the *tex1Dfetch* command. Because each element in the adjacency array consists of both a index and a count, we must multiply this node's index by 2 to obtain the right edge index from the lookup, and by adding 1 we can obtain the count of the number of edges. With this information we can then interate through each edge belonging to this node in the edge list.

We then need to check what nodes these edges connect to. Once again we use a *tex1Dfetch* to access the ID of the node that the first edge connects to. This information is stored in the second value of the edge element. Using this we can then check if the node is on the closed list. If it is, we can skip over this node and move on to the next edge because it has already been inspected, however if it is not we need to do some further processing. Applying the heuristic function we can obtain a H value for the new node. In our implementation the H value is computed using a simple Euclidean distance formula involving the longitudes and latitudes of the node in question and the longitude and latitude of the goal. We also compute a G value for the node by taking the G value of current node and adding on the cost of traversing this edge.

Before adding this edge to the priority queue then, we must first test whether the edge is already in the open set (and therefore already in the queue). If it is not then we can insert it into the queue, however if it *is* in the open set we must compare its current G value with the new G value we have computed. If the G value we have just calculated is lower than the G value currently stored for the node, then we must update its value in the queue and reorder the heap appropriately, which is described in section 2.1.2. Finally if the edge is being added to the open list, or its G value has been rescomputed, then the we must set its parent in the result array as the current node. We then move on to the next edge by incrementing the edge index by 4. We do this because each edge in the array has 4 associated floating point values.

As an aside, it is possible to implement this A* algorithm in such as way as to avoid the need for a closed set. In fact the nvidia paper on GPU pathfinding does not show a closed set in its pseudo-code [3]. However, we were unable to obtain a full source code listing from Nvidia to examine their full implementation. There is a downside however to using the implementation that does not use a closed set, and that is that routes must be guaranteed to exist between start and goal states in the graph, and using our open street maps data this is not always possible, especially when generating random start and goal states in the environment. At the same time the closed set of a given agent only takes up less than 1% of the total memory footprint of an agent when imlpemented as a bitset, so it would not constitute any real meaningful gain in terms of saving memory.

Chapter 5

Results and Evaluation

The results presented in the chapter were tested on two separate GPUs, the Quadro FX580[24], an entry level workstation GPU, and a GeFore GTX 470[20], a top of the range GPU aimed at gaming enthusiasts. Both cards fully support the CUDA computing engine, however the GeForce GTX470 has a compute capability of 2.0, while the FX580 only has a compute capability of 1.1. This is a general measure of the "generation" of the GPU in question and specifies which sets of features each card is capable of. More recent cards such as the Nvidia GTX470 have added improvements such as fully cached global memory and increased numbers of cores per Multi-Processor. The biggest factor separating these two GPUs is the total number of CUDA cores, the Quadro FX580 having only 32 cores while the GeForce GTX470 has 448. This results in a huge increase in raw computing power, although for highly divergent algorithms such as A* pathfinding this would probly not have as big an impact on performance as when performing something as embarrassingly parallel as matrix multiplication. However, when implementing the work detailed here the code was optimized to perform best on devices of compute capability 1.1 such as the Quadro FX580. Because of this, the results seen here for the GeForce GTX470 may be below what could theoretically be achieved had the code been optimized for more modern GPUs. That said, the performance gain from running the code on the higher spec GPU is still substantial and results in a marked improvement over the Quadro FX580 in almost all situations.

The CPU used in the benchmarks was a Intel Core2 Duo T6600 @ 2.2Ghz. The CPU version of the A* search is a single-threaded direct port of the GPU version, using the same roadmap structures as described in 3.3.1. Care was taken to ensure that the CPU implementation was as close to the original GPU version as possible. To perform the benchmark then, the CPU simply performs the A* search for each agent in serial using a 'for' loop.

The testing was performed using the framework described earlier, and consisted of generating x start and goal states for the scene and then performing the CUDA implementation of the A* search using x agents. This means that each agent in the simulation will have its own unique path and no two agents should be following identical paths. The start and goal state are generated randomly as described in section 4.3.2, so while we cannot guarantee that no two agents will have the same start

and goal states, it is highly unlikely when dealing with graphs with high numbers of nodes. The numbers of agents used in the testing ranged from 32 to 8,192 and increased in powers of two. Due to GPU memory limitations, not all graphs were able to run with the full 8,192 agents, in these cases we simply ran the tests for up to the maximum number of agents possible in a single launch.

There were a total of five different maps tested ranging from 348 nodes to over 12,000. Each of these were taken from the Open Street Map database and represent real-world geography including classifications of streets and roadways. Each of the maps included a number of different classifications of road, with agents associating a lower cost to high priority edges when pathfinding towards their goal. The two areas used in the maps are the city centre of Dublin, Ireland, and a portion of the town of Bray, Co.Wicklow in Ireland. Due to the different population densities of the areas there are differing rates of connectivity to the associated graphs. The graphs based on maps of the city center have a higher degree of connectivity due to the complex road infrastructure, which results in the average path lengths being shorter than might be expected. On the other hand the graphs of the town of Bray have fewer main roads, and because of this connectivity is limited. With much of the graph consisting of unclassified roads leading into housing estates or cul de sacs, the path lengths tend to be longer than for a highly connected environment. A small number of nodes also exist on each map that are unconnected, these appear due to how nodes are used on Open Street Map to represent places of interest that may just be a point in space. However, the number of unconnected nodes is so small that they are irrelevant to our testing.

For each map we ran a series of tests wherein a number of random start and goal states would be generated, beginning with 32, and then that same number of agents would perform the A* search on the map. Each of these tests would be performed 10 times and resulted in a total number of A* searches equal to 10 times the agent count. For each test we record the total search time associated with the run and then divide by 10 to get an average search time for that number of agents. We also record the average length of the paths generated by the agents so that we may compare average path length to execution time. Because the average path length is computed over *all* paths, even those that returned a path of length zero, we also decided to record the total number of 'zero paths'. Zero paths occur when the goal state is unreachable from the start state. This allowed us to qualify a lot of our average path length results by also displaying how many paths were unreachable. In the tables in the appendix we show the total times for each number of agents to search each graph and also the total 'time per agent'. We decided to use this 'time per agent' metric as it is a good indicator of how well the GPU can handle differing number of agents, and shows what type of workload works best on the GPU, and what is better left to the CPU.

A second set of results for each graph is also shown where we forced each start and goal state to be placed on a classified road. This was done because with our purely random start and goal state placement we were finding large number of the goal states were returning as unreachable, in some cases upwards of 60% of the total routes. Because of this our numbers for average path lengths in each graph were being skewed by large numbers of 0's. While this does solve the issue, bringing the percentage of 'zero paths' down below 1% in most cases, it also has the effect of making paths slightly *easier* to solve, so we decided to split it into a separate section of results.

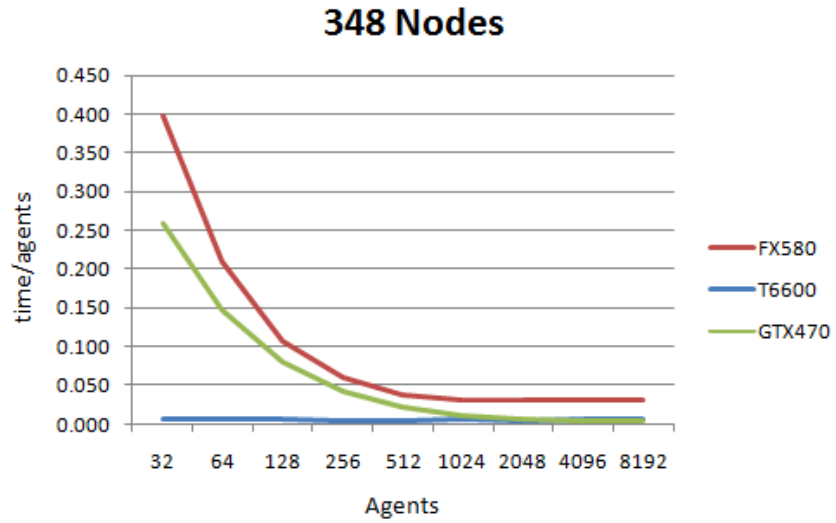

Figure 5.1: Timer per agent compared to number of agents for the 348 Node graph.

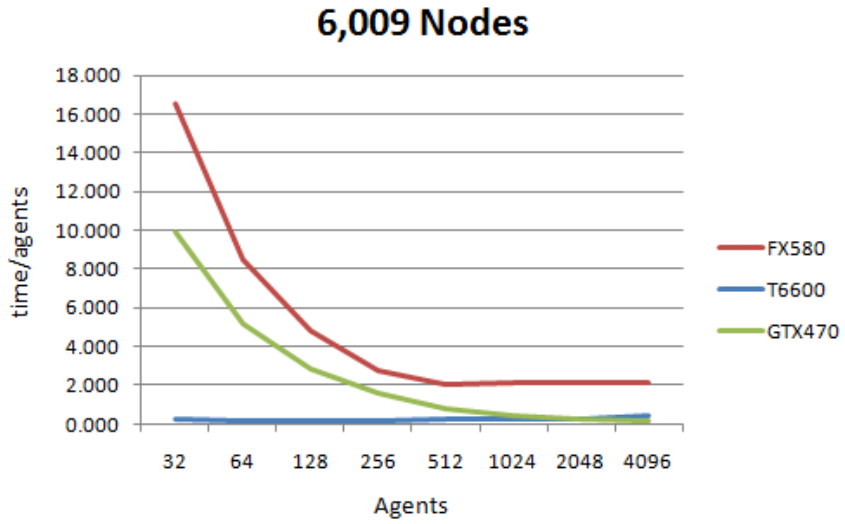

Figure 5.2: Timer per agent compared to number of agents for the 6k Node graph.

As can be seen from the results, we are still not in the realm of real-time pathfinding on the GPU, although we can see from the performance gains of the high end GPU that that goal is in sight and real-time A* pathfinding on the GPU may indeed be possible in a few generations of GPU. The main area where the GPU suffers is when dealing with low numbers of agents. Anything below 512 agents on even the smaller graphs results in the CPU significantly outperforming the GPU on both total time and time per agent. This is due to the nature of GPUs throughput based design. Without all of the GPU cores being saturated with agents running paths, much of the GPUs computing power is

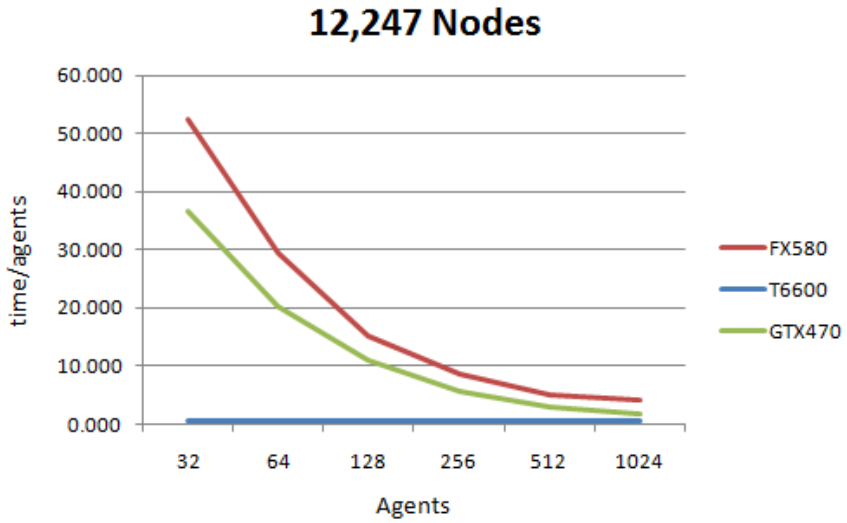

Figure 5.3: Timer per agent compared to number of agents for the 12k Node graph.

wasted. This is not simply a matter of running more agents than there are cores on the GPU. CUDA makes use of having thousands of threads in flight at once by scheduling new threads to run while older threads wait for memory transactions to return. Without these extra threads waiting to execute we must sit idle while the memory transaction completes, which is detrimental to performance.

As well as this the highly divergent nature of A* results in forcing much of the GPUs SIMT architecture to slow to a crawl and perform each thread in serial. Due to the lower clock speeds, smaller caches and longer memory access times, the GPU simply cannot compete with a modern CPU with high clock speed and a large cache when executing threads in serial. This can be seen from the correlation between graph size and execution time in the results. Because the main block of the CUDA kernel consists of a 'while' conditional statement, larger graphs can be directly associated with an increase in the number conditional statements being processed, and therefore the number of divergent agents. As the number of agents, and therefore threads, in the system increases however, the GPU begins to come into its own. As the cores become more and more saturated we can see a steady increase in performance in terms of time per agent, even overtaking the CPU counterparts after enough agents are added to the system. This shows us that the GPU may indeed have some advantages over a CPU even when executing highly divergent algorithms. When running simulations with large numbers of agents such as crowd simulations or complex game environments it is worth considering what advantages could be gained from performing the pathfnding on the GPU, with the high end GPU producing results on par with our CPU implementation when pathing over 2,000 agents in almost all the graphs tested.

		Random		Classified	
Nodes	Edges	Avg.Path	Zero Paths	Avg.Path	Zero Paths
348	758	13.43	0.44	19	0.05
2148	4456	26.86	0.53	31.55	0.29
6009	13262	54.82	0.46	84.03	0
7,928	16,122	15.6	0.69	46.72	0.0005
12,246	25,458	25.52	0.64	59.21	0.001

Table 5.1: Average Path Lengths and Number of Zero Paths for Random and Classified Start and Goal states.

Chapter 6

Conclusions and Future Work

6.1 Conclusion

In this disseration we designed and implemented a framework for testing the efficiency of performing A* searches on real-world data using CUDA. The system accepts any XML map file from the Open Street Map database and parses it into a CUDA-friendly roadmap format. This roadmap is then used as a graph over which we can perform the A* searches. The map is displayed in a SDL window where the user can see the different road and motorway types based on their colour on the screen. We implemented a CUDA based A* algorithm that allows multiple A* seaches to be performed at the same time by multiple agents. These agents each have their own start and goal nodes as specified by the user. The framework can also create random sets of start and goal nodes, and specify whether these nodes should appear on major roads or not. The framework also gives the ability to set certain nodes in the graph as blocked, preventing agents from traversing edges connected to these nodes. We are then able to display the resulting routes to the user on the map screen, and animate the agents as they move along their paths. Our results showed that while GPU pathfinding may not yet outperform the current CPU implementations, the gap is closing, and as GPU and CPU become more alike we can only predict that this gap will soon disappear altogether.

6.2 Future Work

6.2.1 Graph Abstraction

As was mention in 2.4.1, many modern implementation of the A* algorithm involve performing graph abstraction to plan a given route in steps. It would be interesting to see how this would perform using the CUDA implementation, as one of the main drawbacks to using CUDA for pathfinding at the moment is that its performance decreases as the size of the map increases. Using graph abstraction it might be possible to simulate a small graph while actually working on a larger one.

6.2.2 Agent Cooperation

It would be interesting to see if a form of cooperative pathfinding could be ported over to CUDA, and what sort of impact this would have on the results we attained in this work. With the large numbers of agents present in simulations such as this, if agents were able to share information they may be able to find routes more efficiently than in standard implementations.

6.2.3 Agent Interference

In our work each agent operates alone, without taking into account what other agents in the environment are doing. An implementation where agents attempted to avoid crowded paths or moved in groups would be interested especially when performing crowd simulations.

Appendix

	FX580		GTX470		T6600	
Agents	Total(ms)	Time/Agents	Total(ms)	Time/Agents	Total(ms)	Time/Agents
32	12.75	0.398	8.29	0.259	0.2	0.006
64	13.49	0.211	9.41	0.147	0.4	0.006
128	13.75	0.107	10.28	0.080	0.7	0.005
256	15.61	0.061	10.75	0.042	1.3	0.005
512	18.82	0.037	11.24	0.022	2.6	0.005
1024	31.88	0.031	12	0.012	5.7	0.006
2048	63.07	0.031	12.38	0.006	10.8	0.005
4096	127.48	0.031	15.18	0.004	23.1	0.006
8192	246.28	0.030	31.84	0.004	46	0.006

Table 1: Bray Town - 348 Nodes

	FX580		GTX470		T6600	
Agents	Total(ms)	Time/Agents	Total(ms)	Time/Agents	Total(ms)	Time/Agents
32	114.59	3.581	83.49	2.609	1.7	0.053
64	129.97	2.031	83.68	1.308	2.8	0.044
128	130.19	1.017	84.35	0.659	5.4	0.042
256	159.69	0.624	85.7	0.335	9.6	0.038
512	264.73	0.517	96.26	0.188	20.5	0.040
1024	560.92	0.548	101.11	0.099	42.6	0.042
2048	1121.3	0.548	107.83	0.053	87.7	0.043
4096	2255.37	0.551	131.38	0.032	175.6	0.043
8192	4530.94	0.553	276.39	0.034	349.3	0.043

Table 2: Bray Town - 2,148 Nodes

	FX580		GTX470		T6600	
Agents	Total(ms)	Time/Agents	Total(ms)	Time/Agents	Total(ms)	Time/Agents
32	530.05	16.564	316.58	9.893	7.8	0.244
64	544.22	8.503	331.45	5.179	12.8	0.200
128	616.76	4.818	360.01	2.813	24.4	0.191
256	695.9	2.718	405.11	1.582	52.9	0.207
512	1045.12	2.041	424.63	0.829	108.1	0.211
1024	2197.14	2.146	474.18	0.463	224.4	0.219
2048	4424.8	2.161	520.24	0.254	449.7	0.220
4096	8664.43	2.115	637.89	0.156	1805.2	0.441

Table 3: Bray Town - 6,009 Nodes

	FX580		GTX470		T6600	
Agents	Total(ms)	Time/Agents	Total(ms)	Time/Agents	Total(ms)	Time/Agents
32	933.72	29.179	608.53	19.017	12.8	0.400
64	1053.3	16.458	626.45	9.788	21.8	0.341
128	1109.26	8.666	681.96	5.328	37.6	0.294
256	1184.56	4.627	757.57	2.959	81.3	0.318
512	1534.16	2.996	731.8	1.429	130.5	0.255
1024	2634.09	2.572	850.39	0.830	298.6	0.292
2048	5294.03	2.585	939.18	0.459	596.3	0.291

Table 4: Dublin City Centre - 7,928 Nodes

	FX580		GTX470		T6600	
Agents	Total(ms)	Time/Agents	Total(ms)	Time/Agents	Total(ms)	Time/Agents
32	1674.15	52.317	1173.07	36.658	14.9	0.466
64	1892.95	29.577	1290.76	20.168	36.4	0.569
128	1928.62	15.067	1390.58	10.864	81.1	0.634
256	2215.85	8.656	1436.19	5.610	149.5	0.584
512	2540.7	4.962	1451.2	2.834	316.5	0.618
1024	4221.1	4.122	1756.94	1.716	616.4	0.602

Table 5: Dublin City Centre - 12,247 Nodes

	FX580		GTX470		T6600	
Agents	Total(ms)	Time/Agents	Total(ms)	Time/Agents	Total(ms)	Time/Agents
32	12.77	0.399	8.53	0.267	0.1	0.003
64	12.88	0.201	9.94	0.155	0.3	0.005
128	13.12	0.103	9.43	0.074	0.7	0.005
256	14.11	0.055	9.13	0.036	1.3	0.005
512	17.82	0.035	9.85	0.019	2.7	0.005
1024	30.49	0.030	11.27	0.011	5.2	0.005
2048	60.9	0.030	11.66	0.006	10.6	0.005
4096	119.6	0.029	13.97	0.003	21.5	0.005
8192	235.77	0.029	30.54	0.004	45.1	0.006

Table 6: Bray Town - 348 Nodes, Classified Start and Goals

	FX580		GTX470		T6600	
Agents	Total(ms)	Time/Agents	Total(ms)	Time/Agents	Total(ms)	Time/Agents
32	111.79	3.493	62.02	1.938	0.9	0.028
64	115.9	1.811	70.14	1.096	2.4	0.038
128	116.49	0.910	74.86	0.585	4.8	0.038
256	150.22	0.587	77.22	0.302	10	0.039
512	231.63	0.452	77.72	0.152	19	0.037
1024	466.68	0.456	89.79	0.088	41.2	0.040
2048	930.37	0.454	97.58	0.048	74.1	0.036
4096	1876.76	0.458	116.21	0.028	149.1	0.036
8192	3761.1	0.459	235.08	0.029	300.6	0.037

Table 7: Bray Town - 2,148 Nodes, Classified Start and Goals

	FX580		GTX470		T6600	
Agents	Total(ms)	Time/Agents	Total(ms)	Time/Agents	Total(ms)	Time/Agents
32	415.48	12.984	226.61	7.082	6.6	0.206
64	425.54	6.649	263.2	4.113	15.1	0.236
128	471.55	3.684	264.93	2.070	29.3	0.229
256	567.02	2.215	296.88	1.160	59.1	0.231
512	817.98	1.598	308.97	0.603	123.6	0.241
1024	1605.32	1.568	331.51	0.324	233.1	0.228
2048	3175.29	1.550	364.43	0.178	460.2	0.225
4096	6274.4	1.532	452.76	0.111	960.5	0.234

Table 8: Bray Town - 6,009 Nodes, Classified Start and Goals

	FX580		GTX470		T6600	
Agents	Total(ms)	Time/Agents	Total(ms)	Time/Agents	Total(ms)	Time/Agents
32	786.32	24.573	439.79	13.743	6.1	0.191
64	862.74	13.480	544.42	8.507	14.2	0.222
128	883.29	6.901	505.24	3.947	34	0.266
256	994.37	3.884	539.02	2.106	60	0.234
512	1337.12	2.612	577.71	1.128	114.1	0.223
1024	2425.24	2.368	662.37	0.647	242.1	0.236
2048	4677.27	2.284	719.41	0.351	463.3	0.226

Table 9: Dublin City Centre - 7,928 Nodes, Classified Start and Goals

	FX580		GTX470		T6600	
Agents	Total(ms)	Time/Agents	Total(ms)	Time/Agents	Total(ms)	Time/Agents
32	1457.76	45.555	919.9	28.747	18.8	0.588
64	1670.59	26.103	1020.83	15.950	34.5	0.539
128	1679.51	13.121	1022.83	7.991	59.6	0.466
256	1870.24	7.306	1148.76	4.487	123.1	0.481
512	2232.26	4.360	1221.09	2.385	242.9	0.474
1024	3618.83	3.534	1354.1	1.322	485.5	0.474

Table 10: Dublin City Centre - 12,247 Nodes, Classified Start and Goals

Bibliography

[1] Diab Abuaiadh, Diab Abuaiadh, Jeffrey H. Kingston, and Jeffrey H. Kingston. Are Fibonacci Heaps Optimal? In *ISAAC'94, LNCS*, pages 442–450, 1994.

[2] Autodesk. Kynapse Game Titles, 2010.

[3] Avi Bleiweiss. GPU accelerated pathfinding. In *GH '08: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 65–74, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[4] Avi Bleiweiss. Multi Agent Navigation on the GPU. White paper, GDC '09, February 2009.

[5] Vadim Bulitko, Mitja Luštrek, Jonathan Schaeffer, Yngvi Björnsson, and Sverrir Sigmundarson. Dynamic control in real-time heuristic search. *J. Artif. Int. Res.*, 32(1):419–452, 2008.

[6] Vadim Bulitko, Nathan Sturtevant, Jieshan Lu, and Timothy Yau. Graph abstraction in real-time heuristic search. *J. Artif. Int. Res.*, 30(1):51–100, 2007.

[7] GDI-3D. GDI-3D - Spatial Data Infrastructure for 3D-Geodata, 2010.

[8] Andrew V. Goldberg and Craig Silverstein. Implementations of Dijkstra's Algorithm Based on Multi-Level Buckets, 1995.

[9] Mordechai M. Haklay and Patrick Weber. OpenStreetMap: User-Generated Street Maps. *IEEE Pervasive Computing*, 7(4):12–18, 2008.

[10] Peter Hart, Nils Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, February 1968.

[11] Havok. Havok AI Features, 2010.

[12] Havok. Havok Game Titles, 2010.

[13] Won ki Jeong, Ross, and T. Whitaker. A fast eikonal equation solver for parallel systems. In *In SIAM conference on Computational Science and Engineering*, 2007.

[14] Sven Koenig. A Comparison of Fast Search Methods for Real-Time Situated Agents. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 864–871, Washington, DC, USA, 2004. IEEE Computer Society.

[15] Sven Koenig and Maxim Likhachev. D*lite. In *Eighteenth national conference on Artificial intelligence*, pages 476–483, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[16] Sven Koenig and Maxim Likhachev. Real-time adaptive A*. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 281–288, New York, NY, USA, 2006. ACM.

[17] Richard E. Korf. Real-time heuristic search. *Artif. Intell.*, 42(2-3):189–211, 1990.

[18] Pepe Gonzlez Mario Palomo, Jos de la Huerga. SDL_Draw, 2010.

[19] Ian Millington. *Artificial Intelligence for Games (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[20] Nvidia. GeForce GTX470 Specifications, 2010.

[21] NVIDIA. GPU AI - technology preview, 2010.

[22] Nvidia. Nvidia CUDA Best Practices Guide, 2010.

[23] Nvidia. PHYSX, 2010.

[24] Nvidia. Quadro FX580 Specifications, 2010.

[25] ORS. Open Route Service, 2010.

[26] OSM. Node - Open Street Map Wiki, 2010.

[27] OSM. Open Street Maps Statistic Page, 2010.

[28] OSM. Way - Open Street Map Wiki, 2010.

[29] Vijay Pande. Folding@Home - Distributed Computing, 2010.

[30] Pierre Pontevia. Pathfinding is Not A Star. White paper, March 2008.

[31] SDL. Simple DirectMedia Layer, 2010.

[32] Jeremy Shopf, Joshua Barczak, Christopher Oat, and Natalya Tatarchuk. March of the Froblins: simulation and rendering massive crowds of intelligent and detailed creatures on gpu. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, pages 52–101, New York, NY, USA, 2008. ACM.

[33] Jakob Siegel, Juergen Ributzka, and Xiaoming Li. CUDA Memory Optimizations for Large Data-Structures in the Gravit Simulator. *Parallel Processing Workshops, International Conference on*, 0:174–181, 2009.

[34] Game Physics Simulation. Game Physics Simulation, 2010.

[35] Apache Software. Xerces-C++ XML Parser, 2010.

[36] Nathan Sturtevant and Michael Buro. Partial pathfinding using map abstraction and refinement. In *AAAI'05: Proceedings of the 20th national conference on Artificial intelligence*, pages 1392–1397. AAAI Press, 2005.

[37] Adrien Treuille, Seth Cooper, and Zoran Popović. Continuum crowds. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 1160–1168, New York, NY, USA, 2006. ACM.

[38] Jur van den Berg, Sachin Patil, Jason Sewall, Dinesh Manocha, and Ming Lin. Interactive navigation of multiple agents in crowded environments. In *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 139–147, New York, NY, USA, 2008. ACM.

[39] Eric W. Weisstein. "Taxicab Metric." From MathWorld–A Wolfram Web Resource, 2010.

[40] Wikipedia. Folding@Home - Graphics Processing Units, 2010.