CHALMERS

A* Algorithm for Multicore Graphics Processors

RAFIA INAM

Master's Thesis

Networks and Distributed Systems Programme

CHALMERS UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering
Division of Computer Engineering
Göteborg 2009

A* Algorithm for Multicore Graphics Processors

RAFIA INAM,

Examiner: PHILIPPAS TSIGAS

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden July 2010

Abstract

Graphics processors are becoming more and more popular among many application developers as commodity data-parallel coprocessors. As a consequence, general purpose programming for graphics processors research is growing. *A** is a well known best-first shortest path finding algorithm, commonly used in computer games and robotics, to quickly find a good path in real time. We have implemented A* in CUDA, NVIDIA's programming platform for graphics processors. Our graphics processor A* implementation is faster and more efficient than a similar CPU version of the same algorithm.

We have provided three improvements to the parallel A* algorithm to allow it to work faster and on larger maps. The first improvement is the use of *pre-calculated paths* for commonly used paths. Secondly, we allow multiple threads to work on the same path and thirdly we have implemented a scheme for *Hierarchical Breakdown*. Instead of computing the complete path as a whole, the path is calculated in many segments. This makes it possible to calculate more paths concurrently on big maps than was possible before. Very large maps are broken down into many *clusters* and paths are computed at a higher level of abstraction using *path abstraction*. All the segments of a path are then joined together to make a complete path using *path refinements.*

ACKNOWLEDGEMENTS

First of all, I am grateful to Professor Philippas Tsigas for supervising my master's thesis and for all the inspirational discussions. I am also thankful to Daniel Cederman for the guidance in CUDA, and the time spent answering all my questions and removing errors from my code and proof-reading my thesis. Without both of them this thesis would not have been possible in this form.

Many thanks to all the members of the Department of Computer Science and Engineering. My gratitude to Dr. Tomas Olovsson, Director of Networks and Distributed Systems program, for his guidance, good discussions and writing a lot of officially required letters for me. My special thanks go to all members of the Distributed Computing and Systems Group from whom directly or indirectly I learned a lot.

Finally, I would like to extend my deepest gratitude to my friends and family. Many thanks go to my father, who always encouraged me in my studies and to my mother for her great love for me. And to my husband Inam who is always there in tough n rough, and to my daughters Youmna and Urwa I love the most, cannot express how you have filled my life with joy and happiness, so I am not trying.

Rafia Inam

CONTENTS

List of Figures

List of Tables

List of Graphs

1 Introduction

1.1 Background

At the start of multicore CPUs and GPUs the processor chips have become parallel systems. But speed of the program will be increased if software exploits parallelism provided by the underlying multiprocessor architecture [1]. Hence there is a big need to design and develop the software so that it uses multithreading, each thread running concurrently on a processor, potentially increasing the speed of the program dramatically. To develop such a scalable parallel applications, a parallel programming model is required that supports parallel multicore programming environment.

NVIDIA's graphics processing units (GPUs) are very powerful and highly parallel. GPUs have hundreds of processor cores and thousands of threads running concurrently on these cores, thus because of intensive computing power they are much faster than the CPU. At start, they were used for graphics purposes only. But now GPUs are becoming more and more popular for a variety of general-purpose, non-graphical applications too. For example they are used in the fields of computational chemistry, sparse matrix solvers, physics models, sorting, and searching etc. The algorithms designed for GPGPU (General Purpose GPU) run on the multi processors using many threads concurrently. As a result, these algorithms are extremely fast.

In this thesis we have used GPGPU in the field of searching. A* [2] is a standard search algorithm that finds shortest paths. We have created an implementation of the A* algorithm for multi-core graphics card to increase the efficiency of the algorithm using CUDA architecture. Different techniques to make this algorithm more optimal and fast are applied for different sized search spaces and then compared with each other to find which works well on multicore environment of graphics card. We have also found out the suitable sizes of search areas for these techniques. The methods we tried are calculating some pre-paths using waypoints, multiple threads helping each other per agent and the hierarchical breakdown of the A* algorithm.

The thesis is organized as follows. The rest of section 1 introduces the CUDA system model, search algorithms, A* algorithm and give the scope and objectives of this work. Section 2 describes the details of A* algorithm and its implementation. It further provides the three different methods used to improve

the algorithm. In section 3, the detailed description of the system model is given. Related work is specified in section 4. Experimental results from the A* implementations are found in section 5, followed by a discussion of the presented work in section 6. The thesis is concluded in section 7.

1.2 CUDA

CUDA stands for Compute Unified Device Architecture. It is a parallel programming paradigm released in 2007 by NVIDIA. It is used to develop software for graphics processors and is used to develop a variety of general purpose applications for GPUs that are highly parallel in nature and run on hundreds of GPU's processor cores.

CUDA uses a language that is very similar to C language and has a high learning curve. It has some extensions to that language to use the GPU-specific features that include new API calls, and some new type qualifiers that apply to functions and variables. CUDA has some specific functions, called *kernels*. A kernel can be a function or full program invoked by the CPU. It is executed N number of times in parallel on GPU by using N number of threads. CUDA also provides shared memory and synchronization among threads.

CUDA is supported only on NVIDIA's GPUs based on Tesla architecture. The graphics cards that support CUDA are GeForce 8-series, Quadro, and Tesla. These graphics cards can be used easily in PCs, laptops, and servers. More details about CUDA programming model are described in section 3.

1.3 Search Algorithms

Search algorithms are highly used in computer games, robotics, etc. They are used to find a path from a given 'start' node to a 'target' node. It searches the whole map area and finds each possible path from the start node to reach the target node and then gives the shortest path. One way of doing it is **uninformed search algorithms** also called brute-force search. These algorithms use the simplest method of the searching through the search area. These are not very efficient as a lot of time is required to search the whole map area.

Informed search algorithms are comparatively more efficient than uninformed search methods. These algorithms try to reduce the amount of time spent searching by using heuristic functions. Heuristic function means a method to find a solution that is close to the best possible answer in less time. In search algorithms, heuristic functions are used to calculate an estimate lowest path-cost only towards the target node. In this way the whole search area is not searched,

rather search is made on only the selected portion of area. This reduces the time of search dramatically. Some major informed search algorithms that work for graphs include best-first search, and A*.

1.4 A* Search Algorithm

A* (pronounced "A star") [2] is an informed search algorithm. It is formally defined as “best-first, graph search algorithm that finds the least-cost path from a given initial node to one goal node (out of one or more possible goals)” [3]. This algorithm is mainly used in computer games to find the shortest path at run time. It is also used to find shortest routes among cities, and in robotics, etc.

It is like other graph-searching algorithms in that it can potentially be required to search through a huge area of the map. The search space can be reduced by the use of an efficient heuristic function. Without heuristic or when the heuristic function equals to zero, A* becomes Dijkstra’s path finding algorithm, and on the other hand when it is extremely high, A* turns into BFS. Hence the heuristic function plays an important role in controlling the behavior of A*. If heuristic function gives a very low value, then A* will become slow to find the shortest path. If heuristic evaluates to be a very high value, then A* will become very fast but it will not calculate the shortest path. This reveals that the tradeoff between speed and accuracy of the algorithm is dependent on heuristic. Therefore, a heuristic should be chosen very carefully keeping in mind this tradeoff. A heuristic that is specific to the problem should be used in algorithms.

We are using the Manhattan heuristic [4] [5] for path-finding on maps. More details about A* algorithm and heuristic are described in next section.

1.5 Scope

Since multi-core technology is growing day by day, we wanted to develop the algorithm so that it exploits the multi threaded nature of the hardware. The intention of this project is the implementation of the A* shortest path finding algorithm on a multi-core graphics card using CUDA architecture provided by NVIDIA. Afterward different variations/optimizations of A* algorithm are studied and are implemented to more increase the speed and efficiency of the algorithm. These different optimizations done on A* are described as follow:

1. **Pre-Path Calculations:** Some *way points* are defined on the search area and shortest paths are calculated for each pair of way points and stored in the memory. These pre-stored paths are then used during online searching new paths. When a new path search

will start, the algorithm will check whether any path (or a portion of the path) is close to new search and already computed and stored. If yes, then that pre-computed path or portion of the path is simply added to the final path, and not searched online. This will reduce the total search time and effort effectively. If no pre-calculated path is matched with the new search, the new path will be computed completely.

2. **Threads Helping Each Other:** Many are searching a path using shared memory and thread synchronization. In our case, eight threads are used to search the optimal path per agent (agent characterizes a possible paths on the walkable nodes). Thread synchronizations are used at different stages on the A* algorithm. The purpose is to check the effects of using the shared memory and thread synchronizations on the total search time of A* algorithm.

3. **Hierarchical Breakdown:** Larger problems and big search spaces take a lot of time, effort and memory to calculate very long paths. One solution is to sub-divide the search space into many smaller parts called *clusters*. These clusters are joined with each other using some connection point called *exit points*, and a weighted abstract graph is built in this way. Optimal paths among the exit points per cluster are calculated and stored. Hence, using the method of *abstraction*, the whole search space is reduced into a small weighted abstracted graph that will be used for all searches. A specific path is searched in three stages. First the start and target points are connected to the abstract graph. Then the complete path from start to target point is searched on the abstracted graph. This path does not provide a detailed path. In third stage, the process of *path refinement* is used to make a detailed path. Already searched and stored paths among the exit points of clusters are then placed together at proper places, thus making the complete path.

1.6 Objectives

Since multi-core technology is growing day by day, we wanted to develop the algorithm so that it exploits the multi threaded nature of the hardware and in this way develop an implementation of the A* search algorithm that is very fast and efficient. It was also intended to find balance between each thread's resource usage and the number of simultaneously active threads.

We aim to adopt the simple A* algorithm for multicore platform running on many threads concurrently, and to observe its efficiency, thread synchronizations, and resource utilizations. Especially the memory requirement becomes very important for large search areas. We want to find out the effect of increase in search area on the efficiency and memory requirement of the algorithm. We want to explore which optimizations to be done on A* algorithm running on GPU to get efficient results for larger search spaces.

2 The System Model

2.1 The System Model

Graphics processors were mainly used only for graphics applications in the past. But now modern GPUs are fully programmable, highly parallel architectures that delivers high throughput and hence can be used very efficiently for a variety of general purpose applications.

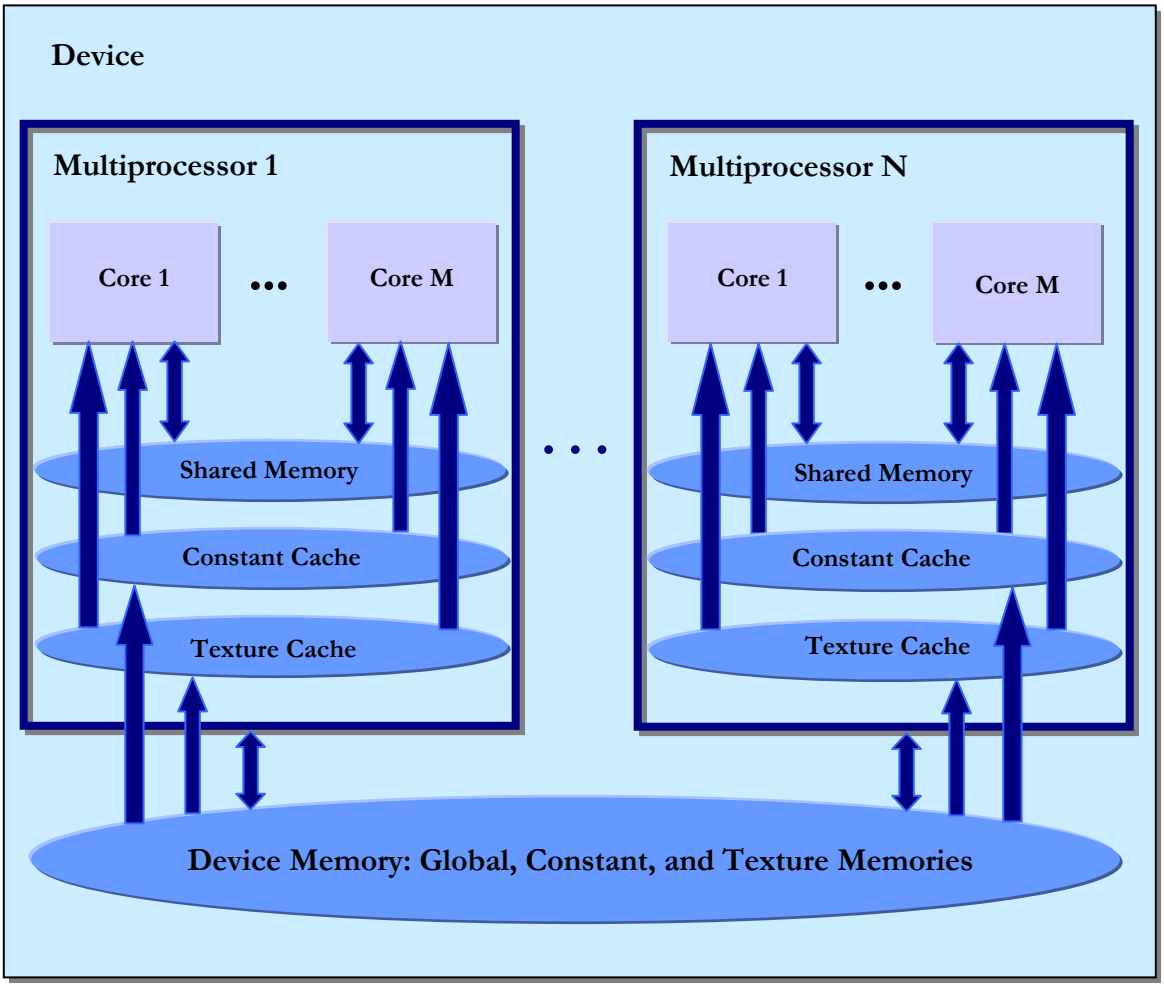


Figure 1: The CUDA System Model.

NVIDIA's graphics card is a new technology that is extremely multithreaded computing architecture. It consists of a set of parallel multiprocessors, that are

further divided into many cores and each core executes instructions from one thread at a time as described in Figure 1. Hence all those computations in which many threads have to execute the same instruction concurrently, also called data-parallel computations, are well-suited to run on GPU.

NVIDIA has designed a special C-based language CUDA to utilize this massively parallel nature of GPU. CUDA contains a special C function called *kernel*, which is simply a C code that is executed on graphics card on fixed number of threads concurrently. For defining threads, CUDA uses a grid structure.

2.2 Heterogeneous Architecture

CUDA programming paradigm is a combination of serial and parallel executions. Figure 2 shows an example of this heterogeneous type of programming. The simple C code runs serially on CPU also called *host* [6].

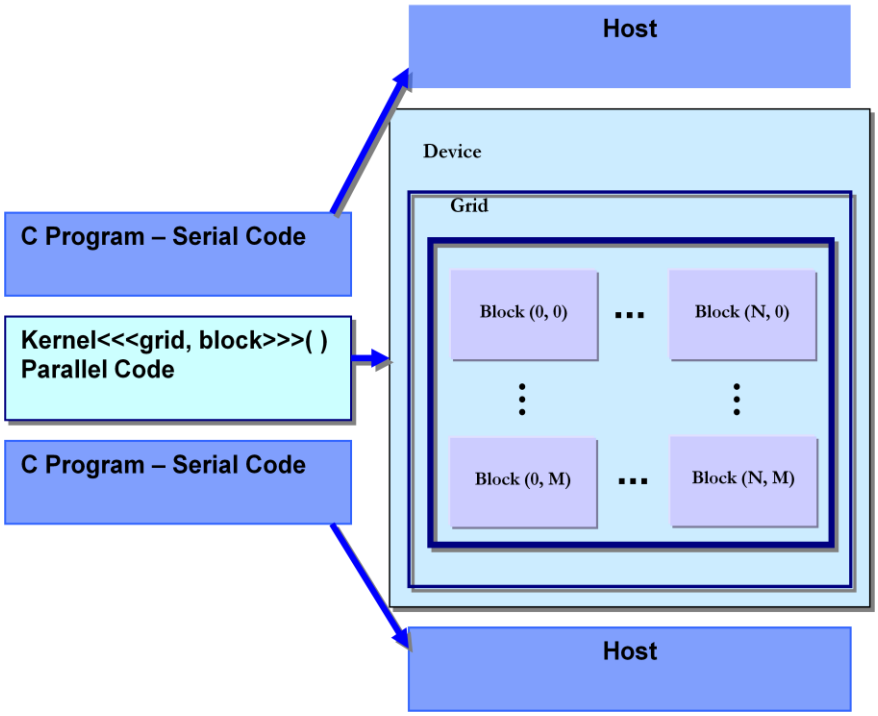


Figure 2: Heterogeneous Architecture

Parallel execution is expressed by the *kernel function* that is executed on a set of threads in parallel on GPU; GPU is also called *device*. This kernel code is a

C code for only one thread. The numbers of thread blocks, and the number of threads within those blocks that execute this kernel in parallel are given explicitly when this function is called.

The kernel function can only be invoked by serial code from CPU. To call the kernel function, the execution configuration must be specified, i.e., the number of threads in a thread block and number of threads within a grid. To declare grid and thread blocks CUDA has a predefined data type *dim3*, an integer vector type that specifies the dimensions of the grid and thread blocks. In the kernel function call grid and block variables are written in three angular brackets <<< grid, block >>> as shown in Figure 2**Error! Reference source not found.**. In this invocation, grid and thread blocks are created dynamically. The value of this grid and block variables must be less than the allowed sizes which are given in next section. The threads are scheduled in hardware and not in software. Kernel function has always a return type *void*. It has a qualifier __global__ that means this is a kernel function to be executed on GPU. See Figure 3 for a graphical description of grid and thread blocks.

2.3 The Grid and block structures

The Grid consists of one-dimensional, two-dimensional or three-dimensional thread blocks. Each thread block is further divided into one-dimensional or two-dimensional threads. A thread block is a set of threads running on one processor. Figure 3 describes a two-dimensional grid structure and a two-dimensional block structure. Within a thread block, threads are organized together in warps. Normally 32 threads are grouped in one warp. All threads of a warp are scheduled together for execution.

All threads of a single thread block can communicate with each other through shared memory; therefore they are executed on the same multiprocessor. In this way it becomes possible to synchronize these threads.

The CUDA paradigm provides some built-in variables to use this structure efficiently. To access the id of a thread block the *blockIdx* variable (values from 0 to gridDim-1) is used and to access its dimension the *blockDim* variable is used while *gridDim* gives the dimensions of the grid. Each individual thread is identified by *threadIdx* variable, can have values from 0 to blockDim-1. *WarpSize* specifies warp size in the threads. All these variables are built-in in kernel. The maximum allowed sizes of each dimension of grid is 65535, and x, y, and z dimensions of a thread block are 512, 512, and 64, respectively [1] [6].

The allocation of the number of thread blocks to each multiprocessor is dependent on the necessity of the shared memory and registers by each thread block. More memory and registers requirement by each thread block means allocation of less thread blocks to each multiprocessor. In this case the remaining thread blocks have to wait for their turn for execution.

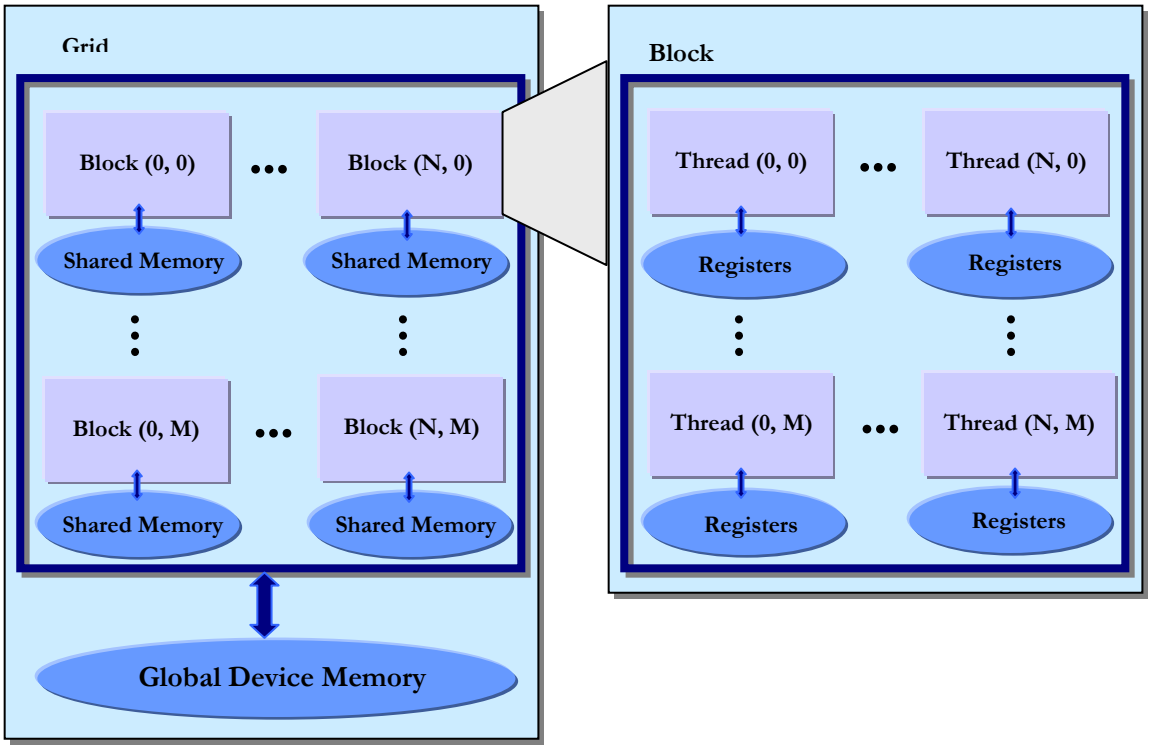


Figure 3: The CUDA Grid Structure and Block Structure.

All this threads creations, their execution, and termination are automatic and handled by the GPU, and is invisible to the programmer. The user only needs to specify the number of threads in a thread block and the number of thread blocks in a grid.

2.4 Memory Model

All multiprocessors access a large global device memory for both gather and scatter operations. Memory model is described graphically in Figure 1. This memory is relatively slow because it does not provide caching.

Shared memory is fast as compared to device memory and normally takes the same amount of time as required to access registers. It is also called parallel data cache (PDC). Shared memory is "local" to each multiprocessor unlike device memory and allows more efficient local synchronization. It is divided into many parts. Each thread block within multiprocessor accesses its own part of shared memory and this part of shared memory is not accessible by any other thread block of this multiprocessor or of some other multiprocessor. All threads within a thread block that have the same life time as of the block, share this part of memory for both read and write operations. As shared memory space is only 16KiB, so it must be used efficiently. To declare variables in shared memory __shared__ qualifier is used and to declare in global memory __device__ qualifier is used.

Each multiprocessor also has its own read only caches to speed up read operation. These are constant cache and texture cache memories.

Each thread also contains its own local memory. Normally local variables of the kernel functions are allocated here. Sometimes they are allocated on global memory.

2.5 Thread Synchronization

For synchronization purpose among threads CUDA API provides a hardware thread-barrier function *syncthreads()* that acts as synchronization point. As threads are scheduled in hardware, this function is implemented in hardware. The threads will wait at the synchronization point until all of the threads have reached at this point. The communication among threads (if required) is possible through per-block shared memory. Hence thread synchronization is possible only at thread block level. Since threads of a thread block may communicate with each other, these threads must execute on same processor. That is why thread block is guaranteed to execute on one processor.

2.6 Control Flow

As the kernel function runs on the device, memory must be allocated on device in advance before kernel function invocation and if the kernel function has to execute on some data then the data must be copied from the host memory to the device memory. Device memory can be allocated either as *linear memory* or as *CUDA arrays*. Qualifier __device__ at the start of a variable specifies that space for this variable is allocated on the device memory. CUDA API [6] also has

functions to allocate and de-allocate device memory at run time like cudaMalloc(), cudaFree(), etc. Similarly, after the execution of kernel function, data from device memory must be copied back to host memory in order to get results. To copy data to and from the device to host CUDA API provides functions for example cudaMemCpyToSymbol(), cudaMemCpyFromSymbol(), cudaMemCpy(), etc. Keeping all this in view the processing flow is as follows:

1. Allocate memory on host and device separately. Device memory is readable and writable by the host through the memory copy functions.
2. Copy data from host to device using CUDA API if required.
3. Kernel function executes parallel on each core.
4. Copy data back from device to host using CUDA API.

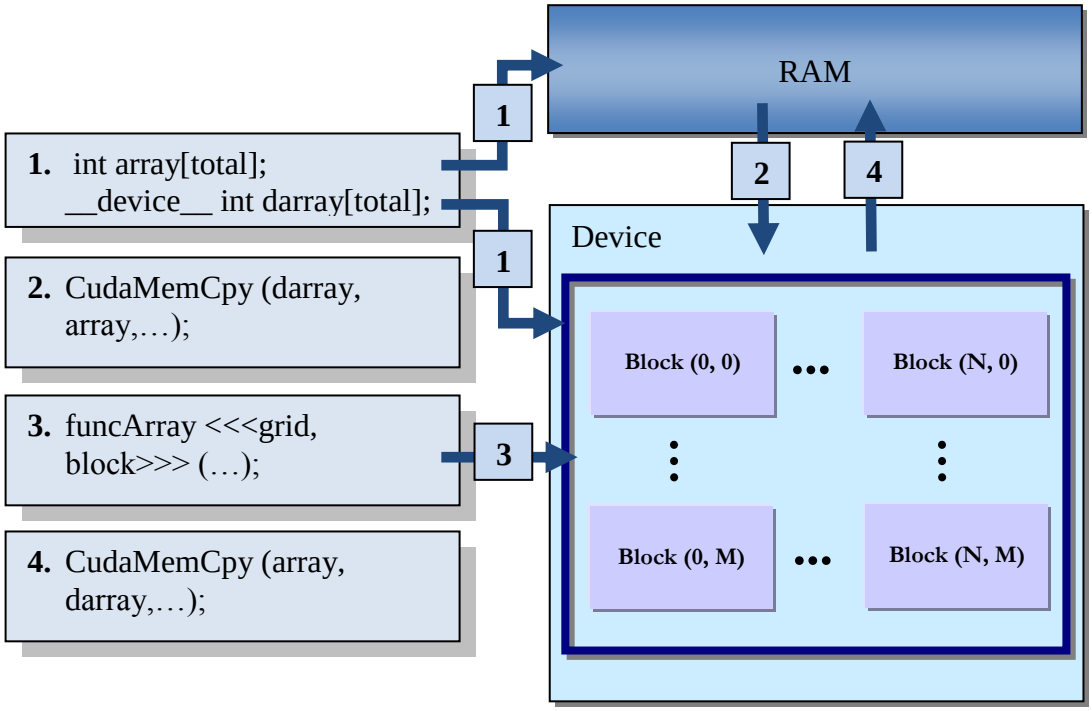


Figure 4: An example of processing flow.

Figure 4 illustrates an example of processing flow of CUDA. In first step two arrays of same size are declared, one on the host and one on the device. The data from the host is copied to the device using CUDA API cudaMemCpy(). The kernel function runs in parallel on the device and in last step the results are copied back to the host using cudaMemCpy() function.

2.7 Restrictions

To use general purpose GPU we must follow the restrictions of the CUDA programming paradigm. Some of the restrictions are given below:

Simple C programming is supported by the CUDA compiler. It lacks the use of object-oriented or C++ features in device code.

Heterogeneous architecture is used to make an interaction between CPU and GPU programming models. Data may be copied from host memory to device memory and the results are copied back to host from the device memory. Heterogeneous programming is discussed in section 2.2 and described graphically in Figure 2.

Kernel function invocation: The grid, thread blocks, and threads are created by the kernel function invocation from the host. This is the only way to create them. They cannot be created inside the kernel function. The grid, and thread blocks are discussed in section 2.3 and Figure 3. Moreover the number of grids and thread blocks must not exceed their maximum allowed values.

The kernel functions do not return any results, i.e. its return type is always void. Further the kernel function call is asynchronous. It means that control returns back before the completion of the kernel function on the device. More information can be found in CUDA programming guide [1]. All functions with the __device__ qualifier are by default inline.

Recursion is simply not allowed within kernel functions because of the large amount of memory requirement for the thousands of thread.

The device memory allocation and de-allocation at run-time is possible only when using host code and before calling the device code. It means that within the device code, the device memory cannot be allocated nor de-allocated using the functions like cudaMalloc(), cudaFree(), etc. All the allocations required for a specific kernel function are done before calling that kernel function in the host code and similarly all that allocated device memory is de-allocated after the completion of that kernel function in the host code.

Shared memory is shared among the threads on the same thread block only. Threads from different thread block cannot share it. This concept is discussed in section 2.4 and graphically shown in Figure 3.

Built-in variables such as blockIdx, threadIdx, etc, cannot be assigned any values. Further it is not possible to take their address.

The variables declared with __device__, __shared__, or __constant__ qualifiers also have some restrictions [1]. Address of a variable with any one of these qualifiers can only be used within the device code.

Communication and synchronization among threads are only possible at thread block level. Communication among thread blocks is not allowed. Section 2.5 explains the thread synchronization.

3 A* Algorithm

3.1 A* Path finding Algorithm

A real challenge for an agent in real time games is to find the route from the start node to the goal node in presence of other agents and obstacles. In the presence of obstacles, the path moves around the obstacle and reaches the goal. This path should be of minimum cost or in other words it should be the shortest possible distance.

A* is a shortest path finding algorithm that uses informed search technique to find the least-cost path from the start node to the goal node. The classic representation of the A* algorithm is as follow [7]:

"f(x) = g(x) + h(x)"

f(x): is called the distance-plus-cost heuristic function (or simply F cost) and it is the sum of path-cost function g(x) and heuristic function h(x).

g(x): the path-cost function (or simply G cost) is the actual total cost of the path to reach the current node x from the start node.

h(x): is the estimated cost (or simply H cost) of the path from current node x to the goal node. An estimate is made that tells how far the goal node is from the current node x. h(x) must be an admissible heuristic estimate. A heuristic function is said to be admissible if the cost of path estimated by it never exceeds the lowest-cost path. Since h(x) is part of f(x), f(x) is dependable on h(x) for the lowest cost of path. It means when h(x) is admissible, A* algorithm is guaranteed to give the shortest path if one exists. Therefore, h(x) must not overestimate the cost.

There are many different heuristic functions used for the grid maps. Some famous heuristics are Manhattan distance, diagonal distance, Euclidean distance. We are using the Manhattan distance to estimate h(x) because it works better on squared grids [4] [5]. It is the direct distance from current node to the goal node without considering obstacles in the path. In this way h(x) is giving us the lowest possible cost to reach the goal node.

3.2 Implementing Parallel A*

3.2.1 Map Representation

Generally computer games use grids for the world representation. We are considering the map as a uniform two-dimensional grid that is subdivided into small square shaped tiles, as shown in Figure 5. Map has walkable and non-walkable tiles; the obstacles are represented as non-walkable tiles. In Figure 5**Error! Reference source not found.**, the obstacles are represented by the gray tiles. The algorithm searches only walkable tiles of the map. All non-walkable tiles are simply ignored. Hence, each tile represents a node in the algorithm, and is either walkable, or non-walkable.

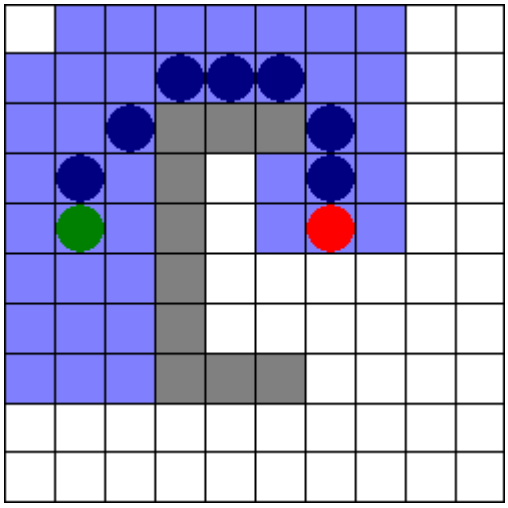


Figure 5: A algorithm search area in light blue; Green is start node; Red is goal node; Gray represents unwalkable nodes.*

Units can move to the adjacent tiles (eight adjacent tiles in our case) including diagonals too. Cost to move straight to the adjacent tile is 10, while the diagonal movement cost is kept 14.

3.2.2 The Nodes

A* search algorithm finds the optimal path in the presences of obstacles depending on F cost value of nodes. The nodes with lower F cost values are

remembered and searched first. The nodes that have already been visited are also remembered, so that they are not checked repeatedly. In this way each node gets one of the following statuses, ‘not visited’, ‘open’, or ‘closed’. All the visited nodes to find a path from start node to the goal node are called search space and they are represented in blue in Figure 6**Error! Reference source not found.**.

3.2.3 Open List

The node that has been visited is called ‘open’ and is placed on the open list. When all neighbor nodes of an open node have been visited (means placed on open list), its status is changed to ‘closed’ and this node is removed from open list. Hence on each pass, one node is removed from the open list and at maximum eight neighbor nodes are placed on it.

The open list array is sorted using binary heap sort. It sorts the nodes according to their F cost values. An optimized way to maintain this sorted list is the use of priority queue [8].

3.2.4 Algorithm flow

Algorithm starts when the current node (start node at the beginning) is placed on open list. Then its eight adjacent neighbor nodes are visited and are put on the open list, their status becomes open, their G cost, H cost, and F cost values are computed and G cost and F cost values are stored. The parent of all these neighbor nodes is the current node. Therefore, current node’s id is stored as Parent of these neighbor nodes. The current node is done at this stage and its status is changed to closed, and it is removed from the open list.

Using lowest F cost values, A* algorithm keeps on moving towards the goal node and finds the shortest possible path in presence of obstacles (non-walkable nodes).

3.2.5 Storing the final path

We are storing the parent of each node. It helps in retaining a path at the end of the search, if a path exists. As the search starts from the start node, thus start node has no parent. By visiting the walkable neighbor nodes of this start node, these neighbors become child nodes of the start node, or in other words, all walkable neighbor nodes has a parent that is the start node. In the next step, these neighbor nodes become parent nodes of other visited nodes, and so on. In Figure 6, arrows point towards the parents.

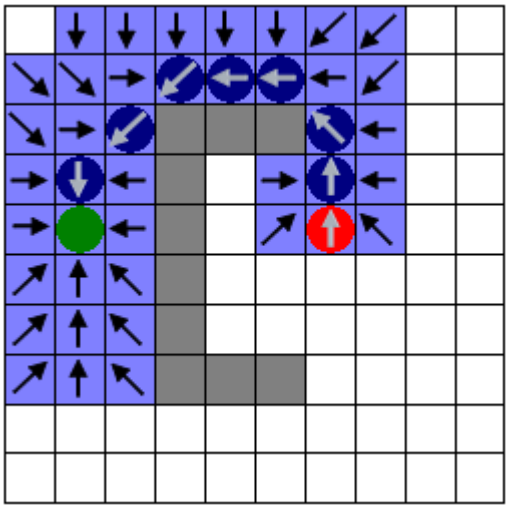


Figure 6: Arrows are pointing towards parent nodes; final path is represented using cyan arrows.

At the end of the search, if the path is found, the optimal path is retrieved by moving backwards from the parent of target node towards the start node. This optimal path is stored in the path array. In Figure 6 the optimal path is shown by arrows of cyan color.

3.2.6 Structure

The structure used to represent a walkable node is C struct type named "node". It contains the following fields:

1. Status – store the status of current node. It can take one of these three values, not-visited, open, or closed. At start all nodes have status not-visited.
2. Parent – stores the parent node's id.
3. Gcost – stores the actual cost to travel from start node to the current node.
4. Fcost – records the F cost value for node.

An array of node type is used in the algorithm for all nodes of the map. The index of the array represents the unique id of each node.

3.3 Priority Queue as Binary Heap

A* algorithm needs a list to insert and remove elements/nodes/items according to their F cost value that is f(x). A* generally removes one node on every pass, and usually adds from zero to eight new nodes to the list. The node with the lowest F cost value is removed first and the list should be rearranged after insertion of every new node to it. This takes a lot of time, so some efficient way should be used to maintain this list. Moreover, for larger maps it becomes more essential. The use of a priority queue is well-suited in this situation [8]. We are implementing a priority queue as a sorted list and are using binary heap to sort the nodes according to the lowest F cost value.

The priority queue is maintained to fetch the node with the lowest F cost value and to insert the node at its appropriate position in the queue according to its F cost value. Binary heap is very fast in this situation of numerous insertions and deletions to queue that are dependent on F cost. If priority queue has N number of elements, then average efficiency to insert and remove the elements on the priority queue is O (logN).

For example consider the heap with 1000 nodes on it. Using a binary heap, an average of three comparisons are needed to insert it in the right place, starting from the bottom of queue, and in worst case nine comparisons are required. And an average of about nine comparisons is needed to remove an item from the open list and reorder the heap appropriately. Worst case requires 18 comparisons.

3.3.1 Inserting Items to the Priority Queue

The priority queue is implemented as a sorted list using binary heap sort. Figure 7 shows a simple queue containing n-1 elements. A new element is inserted at the tail end of the queue. This shown is Figure 8Figure 8; total number of elements is n now.

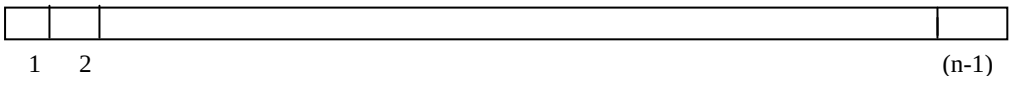


Figure 7: Binary heap with (n-1) elements

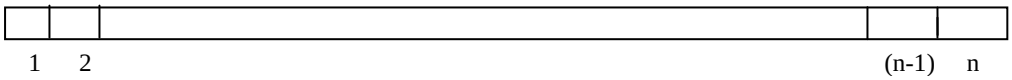


Figure 8: Inserting nth element at the end of heap.

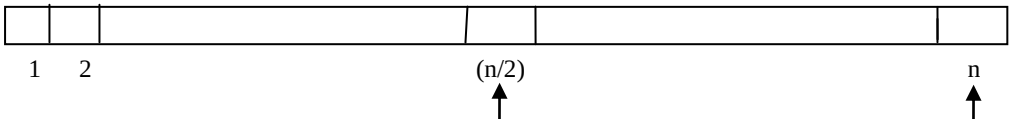


Figure 9: Comparing nth element with the element at (n/2) position.

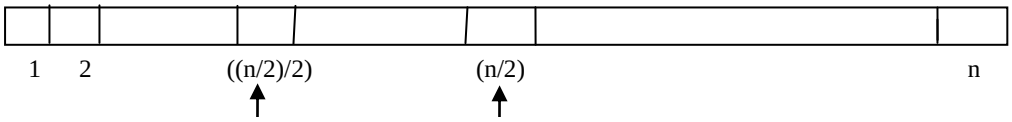


Figure 10: Comparing element at (n/2) with the item at position ((n/2)/2).

Now we need to place this element according to its priority, the F cost value. So we start by comparing the F cost value of this element with its parent's F cost value. If its F cost value is lower than its parent's F cost, the two elements are swapped. This is described in Figure 9. The newly inserted element is now at the position n/2. This element is again compared with its new parent (at position n/2/2), and if its F cost value is lower than its parent's F cost, they are swapped as described in Figure 10. This process continues until the item is not lower than its parent, or until the element has reached to the top of array.

3.3.2 Removing Items from the Priority Queue

In A* path finding we need to delete the lowest F cost item. This element is the top of the queue at position 1, shown in Figure 11. So first, the item in position 1 is removed, which then becomes empty. Now we need to resort the list to place the element of lowest F cost value at first position. We start it by moving up the last item of the heap to position 1 as given in Figure 12. Next we compare F cost value of this element with its two children's F cost values. If its F cost is lower than both the children's F cost values, the process finishes, otherwise it is

swapped with the lowest F cost value child. This is revealed in Figure 13 and Figure 14. This process is continued until the list finishes.

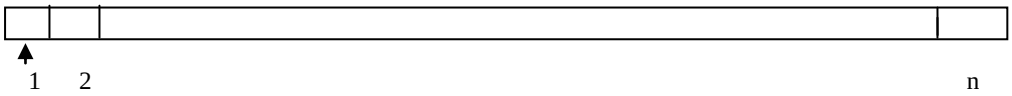


Figure 11: Deleting lowest F cost element from position 1.

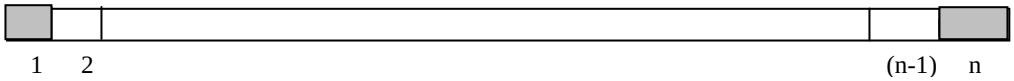


Figure 12: Move nth element at the 1st position of heap.

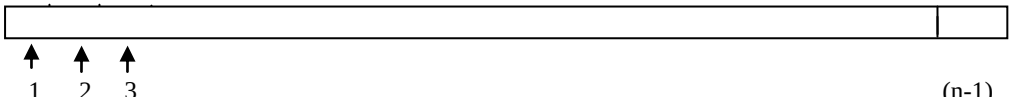


*Figure 13: Comparing Ist element with its two children at positions (current position*2) and (current position*2+1) and then swaping with lower FCost of the two children.*

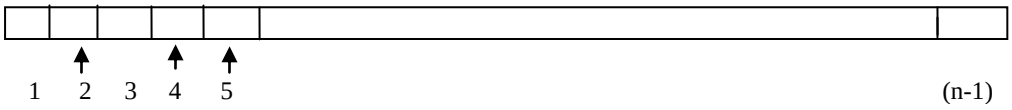


Figure 14: Comparing next element with its two children and swapping with lower Fcost of the two children and so on until list finishes.

3.4 **Pseudo code**

Summarizing everything the pseudo code is given here:

1. Generate the map with walkable and non-walkable nodes.
2. Create open list that is initially empty.

3. Create start node and target node. Place start node on open list.
4. While open list does not become empty and path not found, do the following:
   4.1. Remove the node with lowest F cost value from the open list. This node is now called *current node*.
   4.2. Change the status of current node as closed.
   4.3. Do the following for each adjacent node (eight in our case) of the current node:
      4.3.1. If adjacent node is within the map boundaries and it is walkable and its status is not closed then do the following:
      4.3.2. If adjacent node is not on open list then place it on open list, make current node as parent of this node, and store its G cost and F cost values. Also change its status to open.
      4.3.3. If adjacent node is on open list (means its status is already open) then recalculate G cost value. If new G cost is less than already computed G cost then change the parent node and store newly computed G cost and F cost values and resort open list.
   4.4. When the target node's status becomes open, the path is found.
5. When open list becomes empty, it means path does not exist.

3.5 Admissibility and Optimization

"Any graph search algorithm is said to be admissible if it always returns an optimal solution that is the one with the lowest cost, if a solution exists at all" [9].

However, A* is only admissible if the heuristic you use never over-estimates the distance to the goal. To estimate h(x) we are using Manhattan rule [4] [5]. It calculates the shortest possible distance between two nodes without considering obstacles in the path and gives us the lowest possible cost to reach the goal node. Hence A* is admissible and considers few nodes because it only searches the nodes with lowest path costs to reach the target. When A* terminates its search, it has found a shortest path or the path does not exist.

Some optimizations should be done to improve the performance of the algorithm. Some optimizations done in the code of A* algorithm are given here.

1. The node with the lowest F cost is to delete always from the open list in each search loop. A priority queue is used to keep the nodes sorted according to the lowest F cost value. This priority queue is implemented using binary heap whose average efficiency O (logN). This enables the items be removed efficiently always.

2. Another optimization is done in storing the path after A* algorithm has found it. To store path, we need to backtrack it from target node towards the start node. To make this backtracking easy and efficient, the parent node's id of each node along the path is stored. This parent id is used to retrieve the final path at the end of the algorithm if the path is found.

3.6 Different Variations of the Algorithm

We have made the following three improvements to the parallel A* algorithm to allow it to work faster and on larger maps.

- **Pre-stored Paths:** It is the use of pre-calculated paths for commonly used paths.

- **Multiple Threads per Agent:** multiple threads work together on the same path using shared memory and thread synchronization.

- **Hierarchical Breakdown:** It is path slicing and path putting together activity. Instead of computing the complete path as a whole, the path is calculated in many segments. This makes it possible to calculate more paths concurrently on big maps than was possible before. Very large maps are broken down into many clusters and paths are computed at a higher level of abstraction using path abstraction. All the segments of a path are then joined together to make a complete path using path refinements.

All these three improvements are explained in details in the following sections.

3.7 Pre-stored Paths

When many agents are finding paths in parallel on a search area, some paths are repeated either fully or partially. So it is only waste of time and effort to calculate all paths completely every time. One way to solve this is to compute some paths in advance and store them. The algorithm will run in two phases: first to find some paths and store them, and second to run all agents concurrently and finding their respective paths with the usage of these pre-stored paths.

The first phase is very simple and includes only few paths to be computed. In second phase, when a new agent starts to find a path, first it checks in pre-

stored paths whether this path has been already computed and stored. If yes then there is no need to compute it again, hence the search is stopped and path is simply copied. If no then agent will check for some partially pre-computed paths. It will check whether any pre-stored path's end point and it's to be computed path's target point are same. After this, agent starts calculating its path. Every time at the selection of 'current node' agent checks whether this current node exists on the pre-stored path. Its existence means that the rest of the path has already been computed, so path finding is stopped at this point. The first portion of the final path is computed and the last portion of the path is simple taken from the pre-stored path and appended with first one. In this way effort is put to compute only a portion of the path. In case no pre-stored path is matched fully or partially, the new path is computed completely.

3.8 **Multiple Threads per Agent**

To exploit the parallel hardware architecture in a true sense, the software running on it should be adopted to run concurrently using multiple threads. The software should be changed to utilize multithreading, shared memory access, and achieve concurrency controls. For the adoption of A* using multiple threads, some improvements are required in the basic algorithm. When many threads are finding a particular path, and all these threads are accessing some shared memory, thread synchronization becomes essential for correct execution of the algorithm.

For implementation of A* algorithm running with Multiple threads per agents, I am using eight threads in parallel to find an optimal path. Eight threads are used because the grid illustration is used for the map representation in which each node has maximum eight neighbors. Now these eight threads work concurrently on these neighbors of current node instead of one thread working in a loop. A data structure called a 'temporary list' on the CUDA shared memory is used by these eight threads. It is an array of eight places; one place is accessible to each thread.

In this new multiple threads implementation of the algorithm some changes are made. One thread starts execution of the initial portion of algorithm and executes until the current node is selected. Other threads remain in waiting state. Thread synchronization is used here to accomplish this. After selecting current node, all the eight threads run in parallel to work simultaneously on all the eight neighbors of the current node. All these adjacent or neighbor nodes are checked, their G cost and F cost values are calculated, their statuses are checked and/or changed and they are put on the temporary list. Thread synchronization is done here so that all the threads finish their execution before going further. After

this step only one thread runs for the remaining portion of the algorithm and places all the values of temporary list onto the open list.

3.8.1 Pseudo Code

Summarizing all this, the modified algorithm is given below. Eight threads run in parallel on portion of pseudo code written in italics.

1. Generate the map with walkable and non-walkable nodes.
2. Create *open list* that is initially empty.
3. Create start node and target node. Place start node on open list.
4. While open list does not become empty and path not found, do the following:
   4.1. Remove the node with lowest F cost value from the open list. This node is called now *current node.*
   4.2. Change the status of current node as closed.

   4.3. Thread Synchronization is done here. Eight threads run in parallel, each works on one adjacent node of the current node.
      4.3.1. If adjacent node is within the map boundaries and it is walkable and its status is not closed then do the following:
      4.3.2. If adjacent node is not on open list then place it on temporary list, make current node as parent of this node, and store its G cost and F cost values. Also change its status to open.
      4.3.3. If adjacent node is on open list (means its status is already open) then recalculate G cost value. If new G cost is less than already computed G cost then change the parent node and store newly computed G cost and F cost values and resort temporary list.
   4.4. Thread Synchronization done here.

   4.5. Temporary list is stored on open list.
   4.6. When the target node's status becomes open, the path is found.
5. When open list becomes empty, it means path does not exist.

3.9 Hierarchical Breakdown of A*

Two improvements made on simple A* algorithm, i.e. Pre-path calculations and Multiple threads per agents have shown significant improvements in calculating many paths in less time. Simple A* algorithm and these two improvements on it give good results only on small sized images. As the size of the images increases, the memory requirements of the algorithm also increases which results in less number of thread groups to run parallel on CUDA

architecture, hence not only the time required to find paths increases but also limits the total number of threads that can run concurrently. Thus for larger maps some techniques must be used to overcome the memory limitations of underlying graphics card and CUDA architecture.

One way to solve this problem for large sized images is "finding A* path hierarchically" [10]. It is finding paths in small parts or slices and then putting those path slices together. We have implemented hierarchical path finding A* on graphics card. The search space is divided into smaller portions called clusters. Instead of applying search on whole map, search is applied on smaller portions of the map, hence fulfilling the memory limitations of underlying graphics card and CUDA architecture. The whole process of path finding is done in two steps; Path Abstraction and Path Calculation.

Path Abstraction: also called path slicing, is a onetime activity in which an abstract weighted graph is made from the grid map representation. This graph is stored in memory and all the further path finding is done at a higher level of abstraction using this weighted graph.

Path Calculation: All the actual paths are computed after path abstraction. This is done in the following three steps:

1. In the first step all the start and target nodes are added to the abstract weighted graph.
2. Complete abstract paths are computed on abstract weighted graph at a higher abstraction level. Abstract paths do not include low-level path details.
3. Third step is called path refinement in which all the abstract paths are refined to the low level paths. Detailed paths are patched up to abstract path to give a complete path.

Path abstraction and path calculation processes are discussed below in details.

3.9.1 Path Abstraction

Path Abstraction is the procedure to make an abstract weighted graph from a grid representation of a map. Figure 15 shows grid representation of a map used to take the results. The whole grid map is divided into smaller parts called clusters. These clusters are connected to each other at specific points on the borders of the clusters. These points are called the exit points. Clusters of sizes 20*20 pixels are given in Figure 16. Different clusters are shown in different colors in and the exit points at borders of these clusters are shown in the red color. These exit points are connected in the two following ways:

1. Exit point at the border of one cluster is connected to the adjacent exit points at the border of adjacent cluster if both are walkable. In this way two adjacent clusters get connected to one another and the cost/weight of this connection/edge is 10. These exit points are not connected to the diagonal clusters at the corners. This is called an inter-edge [10] as it connects two different clusters together. One cluster can be connected to its adjacent cluster at maximum three different places, at top, at bottom, and at the middle. If any node at these places is unwalkable, then an adjacent place is considered for the exit point and so on.

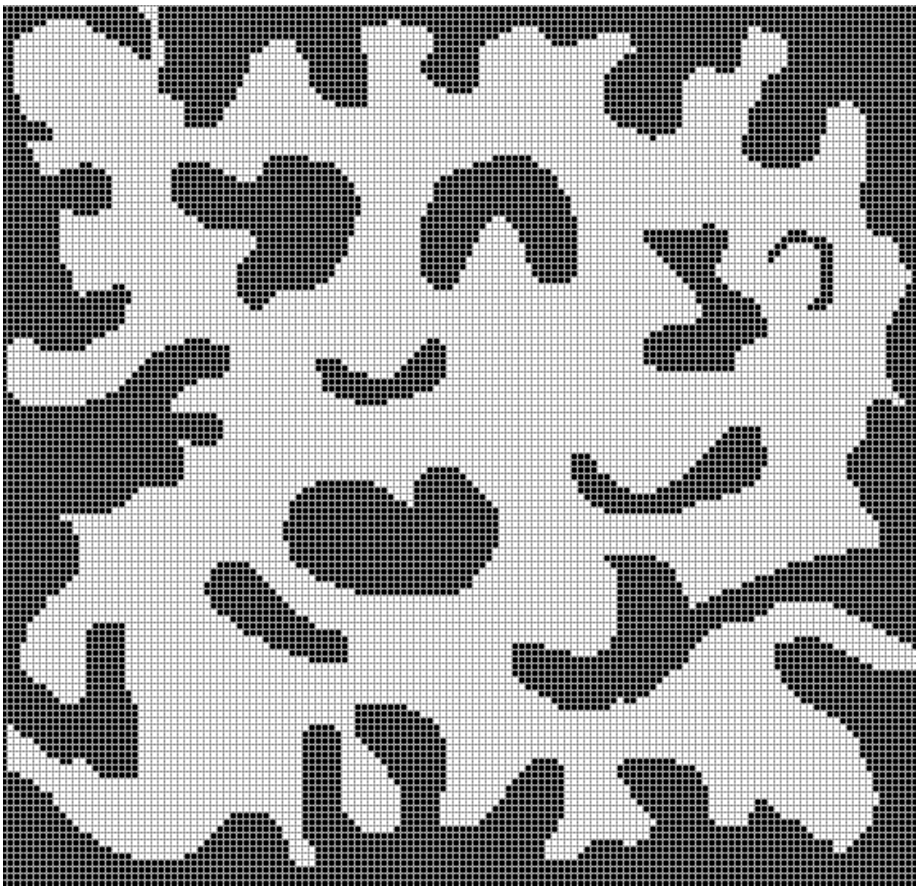


Figure 15: Grid representation of a map used to take the results.

For example consider the top left cluster of the Figure 16. It is adjacent to two other clusters, one at its right and one at its bottom. Looking at top left cluster

and the cluster at its right, we see that the top eight adjacent nodes are unwalkable therefore they cannot be considered for the exit points. The ninth node at the border line of both clusters is walkable therefore they are taken as the exit points. These two clusters are connected at two more exit points; in the middle and at bottom of the clusters. Hence in this way each cluster can be connected to maximum four adjacent clusters at maximum 12 exit points. Figure 16 is generated from the system and displays all the clusters with the connecting exit points.

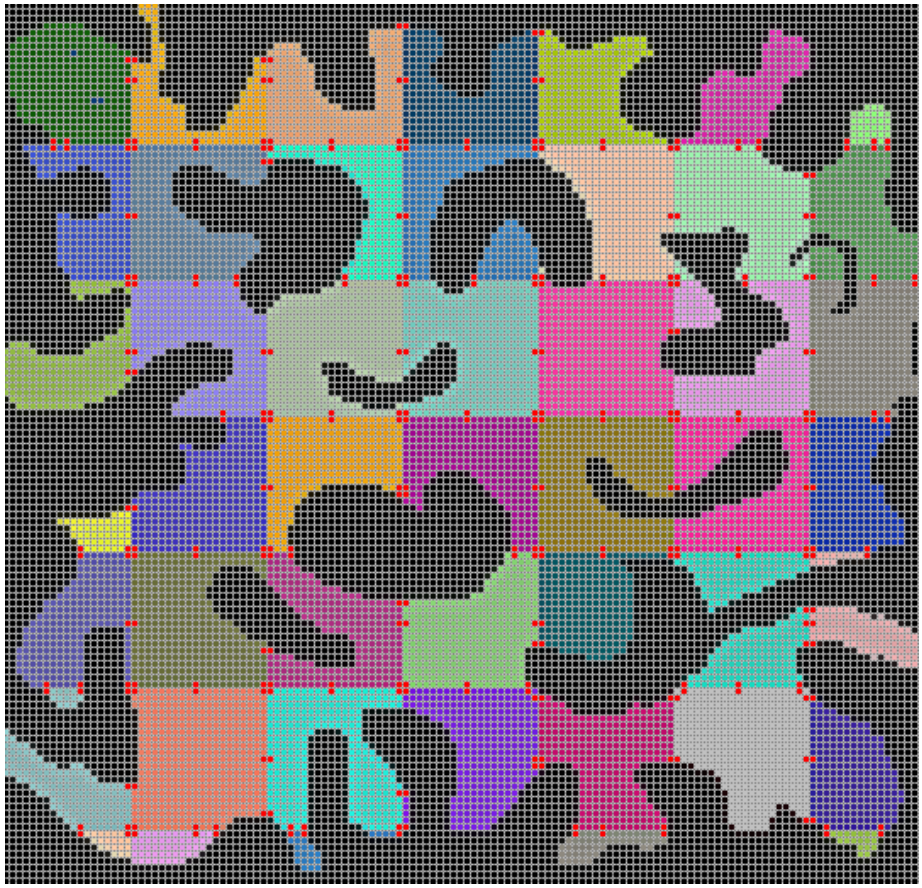


Figure 16: All the clusters with the connecting exit points.

2. In the second step optimal paths among exit points of each cluster are calculated using simple A* algorithm implementation on GPU and then stored. Each exit point of a particular cluster is connected directly to all other exit points in the same cluster. As these edges are connecting exit points within the clusters, therefore, they are, called the intra-edges. Figure 17

illustrates intra-edges of the top left corner of the map. The actual costs of these paths are computed. This cost becomes the weight of edge.

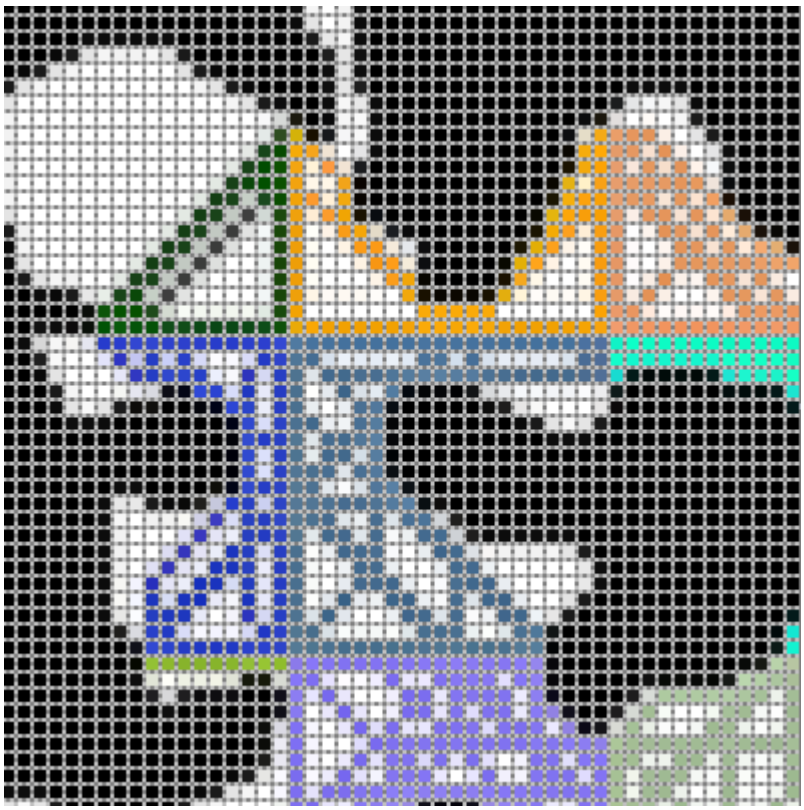


Figure 17: Illustrates intra-edges of the top left corner of the map.

Consequently an abstract weighted graph is made using all inter-edges and intra-edges. An adjacency list is used to store this abstract weighted graph. In first step, as both connecting exit points are adjacent to each other, therefore; there is no need to calculate the path among them. Therefore, they are simply stored on the adjacency list and the cost of edge between them is 10. In step 2, the simple A* algorithm implementation for GPU is used to calculate the optimal paths between exit points. All these paths are computed concurrently on GPU. The algorithm searches for each path only on a small portion of the map, i.e. within one cluster and this one-time activity is very fast. The actual costs of paths, i.e. G costs are stored as weights of the edges on adjacency list. The paths are stored on a separate list called path list.

3.9.2 Path Calculation

All the actual paths are computed in this step. It is accomplished in three steps.

In first step all the start and target nodes are added to the abstract graph. It is done by connecting each start and target point to all exit points of their respective clusters. For example, if a start node lies on top left cluster, then this start node is connected to all the exit points of that top left cluster. This start node, with the cost to reach each exit point is appended to the adjacency list. In this way the start node is attached to the abstract weighted graph. Paths from this start node to the exit points are stored on the path list. The Simple A* algorithm implementation for GPU is used to achieve this.

In second step the complete abstract paths are calculated on abstract weighted graph at higher level of abstraction. A* search is made on adjacency list instead of actual map. Paths found at this higher level are optimal and small. They only include the high level moves, i.e. moving from one cluster to another cluster until target node is reached without considering low-level detailed paths within the clusters. The adjacency list is much smaller in size as compared to the actual map size, therefore; the search is fast. Further the smaller size of adjacency list also overcomes the memory limitations of GPU architecture.

Third step is the refinement of abstract paths. It just patches up all the low-level paths already stored in path list to make a complete path. Paths at lower level of abstraction are not optimal.

3.10 Literature Review

A* algorithm has been very famous for its reduced search space by the use of heuristic function [2]. Further priority queue's implementation as binary heap reduces complexity to O (logN) for N elements. It is fast but one problem with this algorithm is the exponential growth of memory requirement. Iterative-Deeping A*, IDA* [11] uses less memory as it does not remember the visited nodes, but it needs exponential time to run and reach the goal node. Learning Real Time Algorithm LRTA* [12] learns an exact heuristic function and presents better results in both space and time complexities over the previous two algorithms, but it can take a lot of time to learn an accurate heuristic function. [13] provide different factors that affect the performance of Pathfinding especially in game environment.

Hierarchical A* [14] is a technique represents space in an hierarchical fashion. To reduce search space it uses a heuristic to create homomorphism abstractions automatically. It uses algorithms for caching to reduce the number of visited nodes. The method of graph abstraction and refinement used in Hierarchical A* was designed by [15].

Hierarchical Pathfinding (HPA*) [10] divides the search space into sectors with a limited numbers exit points connecting the sectors, instead of abstracting the graph directly from search space. The path quality is improved by using path smoothing. This method increases the performance at the cost of path optimality. Hierarchical Annotated A* (HAA*) [16] is an extension of HPA* and provides a hierarchical path finding using multi-size agents (agents can be of different sizes) and using heterogeneous environment (terrain can be of different types, e.g. plain, tree, water, hill, unwalkable).

Sturtevant and Buro provide a method of automatic state-space abstraction called Partial Refinement A* (PRA*) [17] by which a multi-level graph abstractions to reduce the search space. The complete abstract graph is made by mapping one or more states to a single abstract state and mapping adjacent vertices to adjacent or identical vertices at higher abstraction level. The paths are computed at higher abstractions and then refined. The drawback is the loss of path optimality. These abstractions can also be built dynamically or combined with LRTA* [18]. [19] implements the Windowed Hierarchical Cooperative A* (WHCA*) and Cooperative Partial-Refinement A* (CPRA*) algorithms where path are found in eight directions instead of four, and the agents travel freely in the environment and share information to avoid collisions. A comprehensive empirical evaluation is performed [20] on 3000 problems using different map sizes from 139 * 148 to 193 * 193. Experiments are performed using state-space abstraction [17] and evaluating its effects on two real-time heuristic search methods, Learning Real-Time Search (LRTS) which is an extension of LTRA* and Path Refinement Learning Real-Time Search (PR LRTS). In [21] authors present a technique reduce the number of collisions between agents through the sharing their direction of movements with each other.

All of these algorithms and their results are implemented on single core processor. Avi Bleiweiss [22] implemented the A* algorithm on the graphics card and showed that the results are much faster than the CPU implementations. He used graphs to represent the search space, and thousands of agents run in parallel on GPU each finding one path. Our implementation of simple A* algorithm on graphics card gives approximately same results as that of [22]. We have used array for search space. The problem with running simple A* algorithm on GPU is the increasing memory needs with the increase in map size (search space). Moreover these two implementations do not utilize multithreading or shared memory of the graphics processors.

To find path on bigger maps, we implemented HBDn A* on graphics processors. For hierarchical breakdown we use the path abstraction and refinement technique, very much influenced from [10]. We are dividing the search space into clusters which are connected to each other at three different points as compared to [10] where clusters are connected at only two points. As a consequence the abstract weighted graph contains greater number of nodes than [10]; hence helps in increasing the path optimality. We have used only two-level graph abstraction to keep it simple on GPU.

4 Experimental Evaluation

The results taken for all the four implementations i.e. parallel A*, Pre-stored Paths, Multiple threads per agent, and implementation of hierarchical breakdown of A* are given in the following sections.

4.1 Graphics Card Used

The graphics card used to run the experiments was a NVIDIA's GeForce GTX 260 with 24 multi-processors; each multiprocessor contains 8 processor cores, so it becomes total 192 processor cores. It has 576 MHz Graphics Clock, 1242 MHz Processor Clock, 896 MB standard memory, and 36.9 (billion/sec) Texture Fill Rate.

4.2 Parallel A* Algorithm

The experimental results for the simple A* implementation on GPU using CUDA architecture are presented here. For benchmarking of simple A* algorithm we used the following distributions which are done by Avi Bleiweiss [22] given in Table 1.

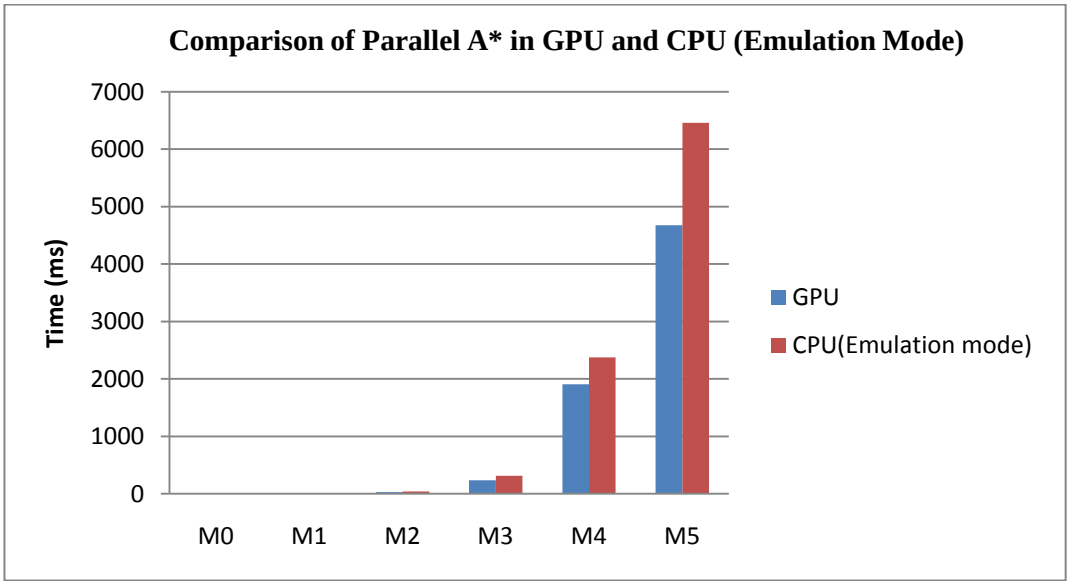
Map	Size (Total no. of nodes)	Walkable Nodes	Agents	Launches
M0	3*3	8	64	1
M1	6*6	32	1024	1
M2	9*9	64	4096	1
M3	13*13	129	16641	1
M4	17*17	245	60025	1
M5	20*20	340	115600	2

Table 1: Benchmarks for parallel A algorithm; Describes size of map array, number of walkable nodes, number of agents (blocks: 1 thread per block), and launches.*

Size gives the size of the map used; nodes are the walkable nodes in the array; agents are the number of all possible paths on walkable nodes, and launches shows that the threads are divided into following number of launches to run. The results of parallel A* algorithm are presented in *Table 2*.

Map	GPU Time (ms)	CPU(Emulation mode) Time (ms)
M0	0.2331	2.098
M1	3.7417	7.8273
M2	27.9296	41.1469
M3	236.227	315.993
M4	1908.219	2372.86
M5	4677.509	6458.94

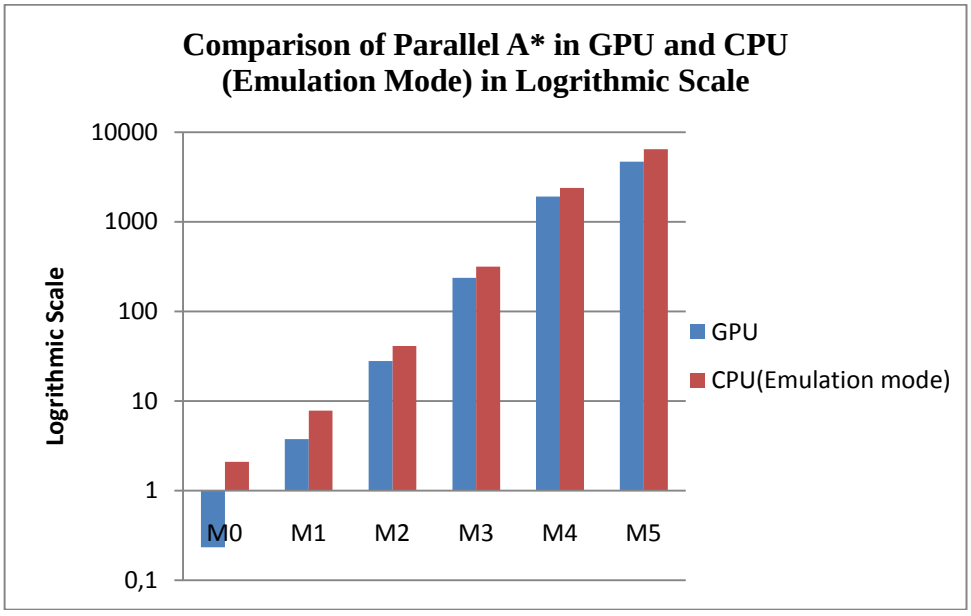
Table 2: Experiment results for parallel A algorithm; Describes time (in milli-seconds) for GPU and CPU (Emulation mode)*



Graph 1: Comparative performance of A for GPU and CPU (Emulation mode).*

Our implementation of the A* algorithm is different in a couple of ways from that of Avi's. Avi is representing the map using a graph with nodes and edges while we are representing it as a map of two-dimensional grid. For this

reason we are using the name Map instead of Graph as used in [22]; we name our Maps as M0, M1, and so on, while Avi uses G0, G1, and so on. Further Avi implemented the A* algorithm using adjacent lists while we have implemented A* using arrays. In our implementation, the total number of nodes is equal to the size of the map. This map contains walkable nodes and some obstacles or non-walkable nodes. The number of walkable nodes in our benchmark is kept the same as the nodes in Avi's [22] benchmark. Another difference is in the number of launches for M4 and M5. Avi is using 2 launches for G4 and 3 launches for G5, whereas we are running M4 in one launch and M5 in 2 launches. Moreover the graphics card used by Avi is also different from us. He has used NVIDIA 8800 GT with 14 multiprocessors thus a total of 112 processor cores. We are using NVIDIA's GeForce GTX 260 with 24 multi-processors so total 192 processor cores.



Graph 2: Comparative performance of A algorithm for GPU and CPU using logarithmic scale*

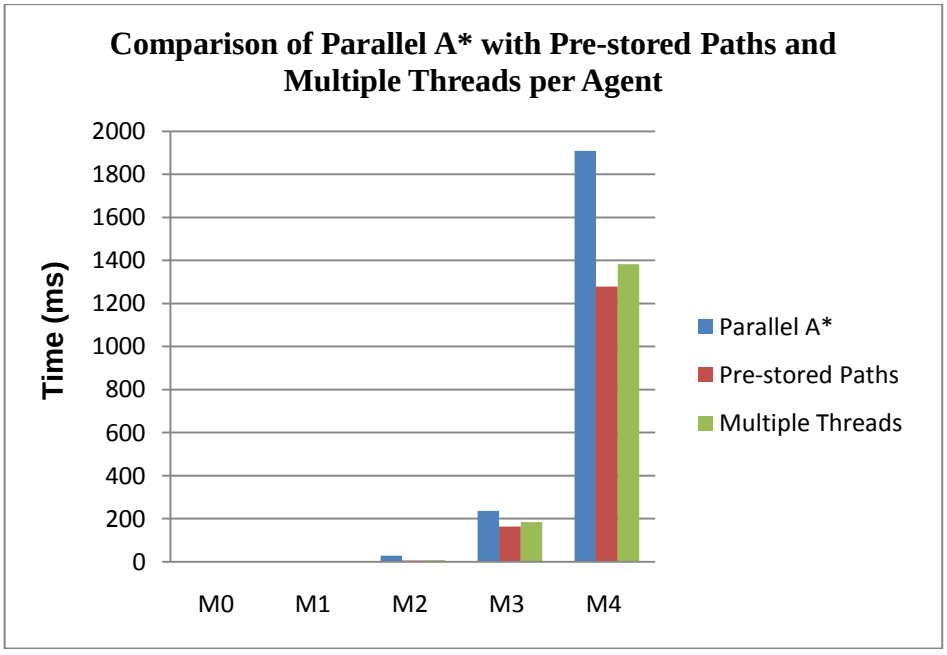
Despite of all these differences, we found that results of both implementations are more or less the same. Graph 1 and Graph 2 represent the results in graphical form.

4.3 Pre-stored Paths and Multiple Threads per Agent

We use the same benchmarks as presented in Table 1. Results are taken for maps M0 till M4 for the Pre-stored paths and Multiple threads per agent. These results are presented in Table 3 and also compared to the simple A* implementation of A* algorithm on GPU.

Map	Parallel A* Time(ms)	Pre-stored Paths Time(ms)	Multiple Threads Time(ms)
M0	0.2331	0.188	0.26
M1	3.7417	2.66	3.15
M2	27.9296	6.62	7.35
M3	236.227	163.7	184.6
M4	1908.219	1279	1382.6

Table 3: Comparison of GPU time of Parallel A to Pre-stored Paths and Multiple Threads per Agent.*



Graph 3: Comparative performance of Parallel A to Pre-stored Paths and Multiple Threads per Agent.*

It is obvious from results that pre-stored paths and Multiple threads implementation are much faster and efficient than the simple A* implementation for GPU. These results are presented graphically in Graph 3 for five maps. Pre-stored path implementation gives the most efficient results. It is because all the paths are not computed fully or partially. Implementation of Multiple threads per agent takes less time than simple A* algorithm, but a little more when compared to pre-stored path. Binary heap becomes the bottle neck in Multiple thread implementations. 8 threads run in parallel but when it comes to put values in binary heap, only one thread remains active and all the other seven threads wait.

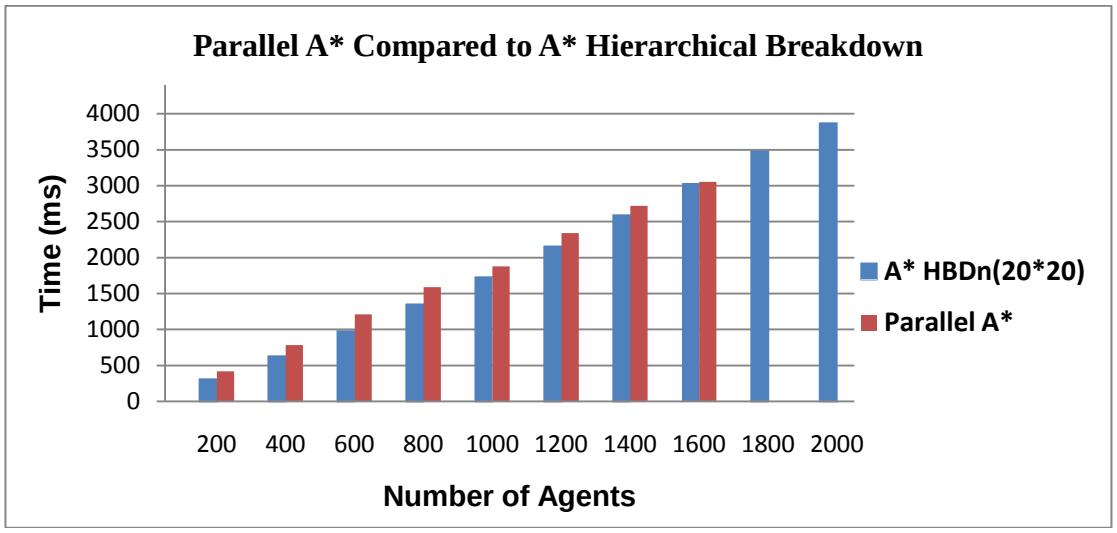
4.4 **Hierarchical Breakdown of A***

The maps used to take results for parallel A*, Pre-stored paths and Multiple Threads are very small in size. Small sized maps are not very practical. As discussed in last chapter, for bigger sized maps, the memory limitations of GPU architecture is an obstacle. Hierarchical breakdown (HBDn) of A* algorithm is used and the results for it are presented here.

Figure 18: Image 1 (Less walkable nodes)

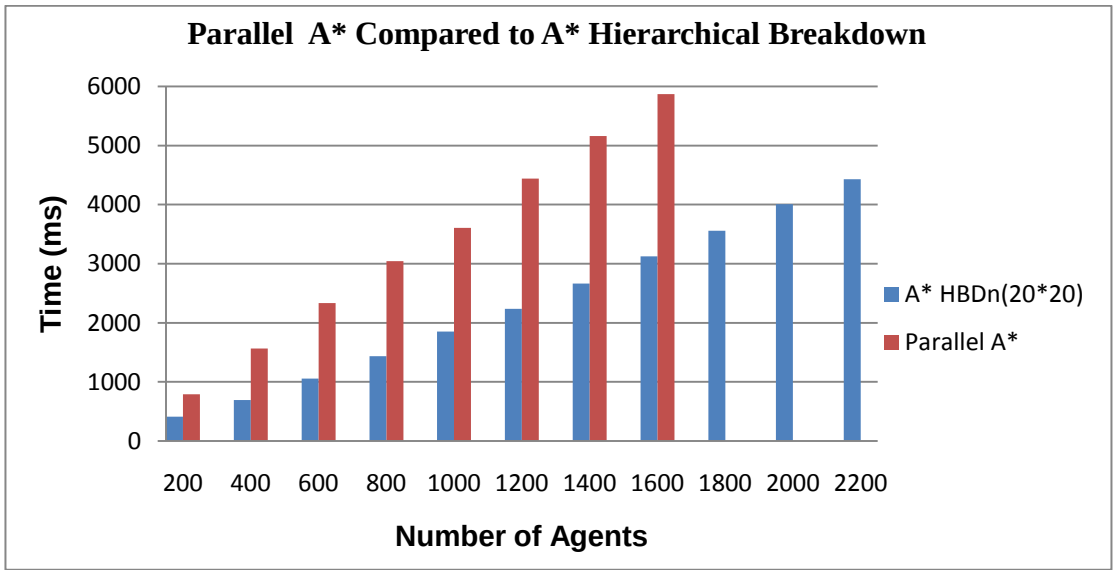
Figure 19: Image 2 (More walkable nodes)

The two main images used to take results for hierarchical breakdown of A* algorithm are shown in Figure 18 and Figure 18. Image 1 has more black (unwalkable) area than the Image 2; hence results are taken for both less walkable and more walkable maps and its effect on the HBDn A* and parallel A* are observed. A third image that is all white is used in few experiments just to check the extreme conditions (all walkable) effect on the efficiency of HBDn of A*. All the three maps are of the size 140 * 130 pixels.



*Graph 4: Comparison of HBDn A*with Parallel A* for Image 1*

In first step, results are taken for Parallel A* implementation and HBDn A* implementation on both images (Image 1 and Image 2) and in second step, they are compared with each other. The results for Image 1 and Image 2 are provided in Table 6 and Table 7 respectively and shown graphically using Graph 4 and Graph 5 respectively.



*Graph 5: Comparison of HBDn A*with Parallel A* for Image 2*

For Image 1 (with less walkable area) both the implementations have approximately same results as shown in Graph 4 except that the Parallel A* implementation does not run for more than 1600 agents because of high memory requirements.

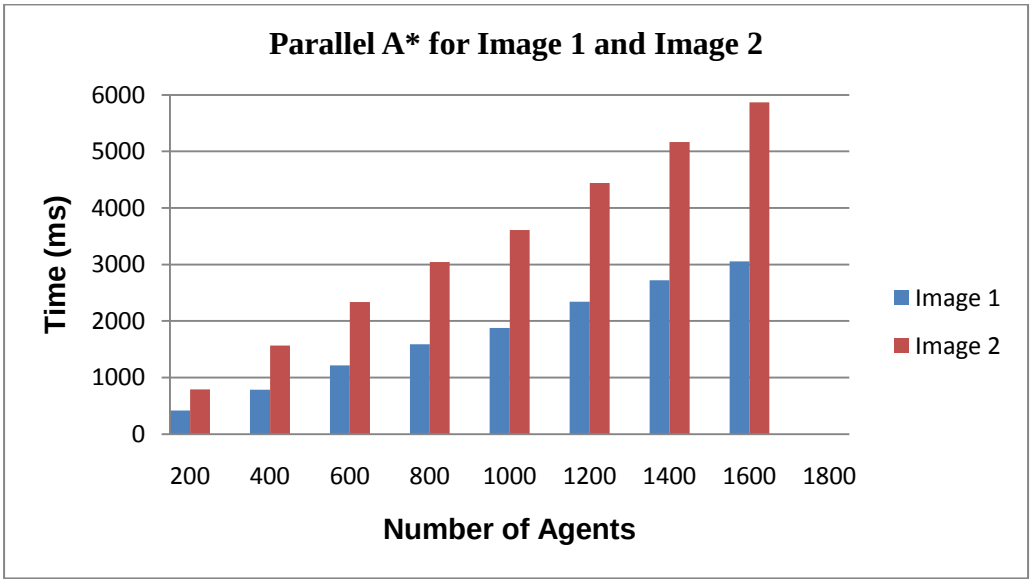
For Image 2 (with more walkable area) the results reveal a drastic change in the behavior of the Parallel A* implementation. It not only stops at 1600 agents but also takes much more time to calculate the paths for the fewer number of agents as compared to HBDn A* implementation. It is clear from the Graph 5. Thus it demonstrates that the Parallel A* implementation has major variations with the increase in the walkable area.

In the second step a comparison is made for the following:

- Parallel A* implementation for Image 1 and Image 2
- HBDn A* implementation for Image 1 and Image 2

4.4.1 Comparison of Parallel A* Results for Image 1 and for Image 2

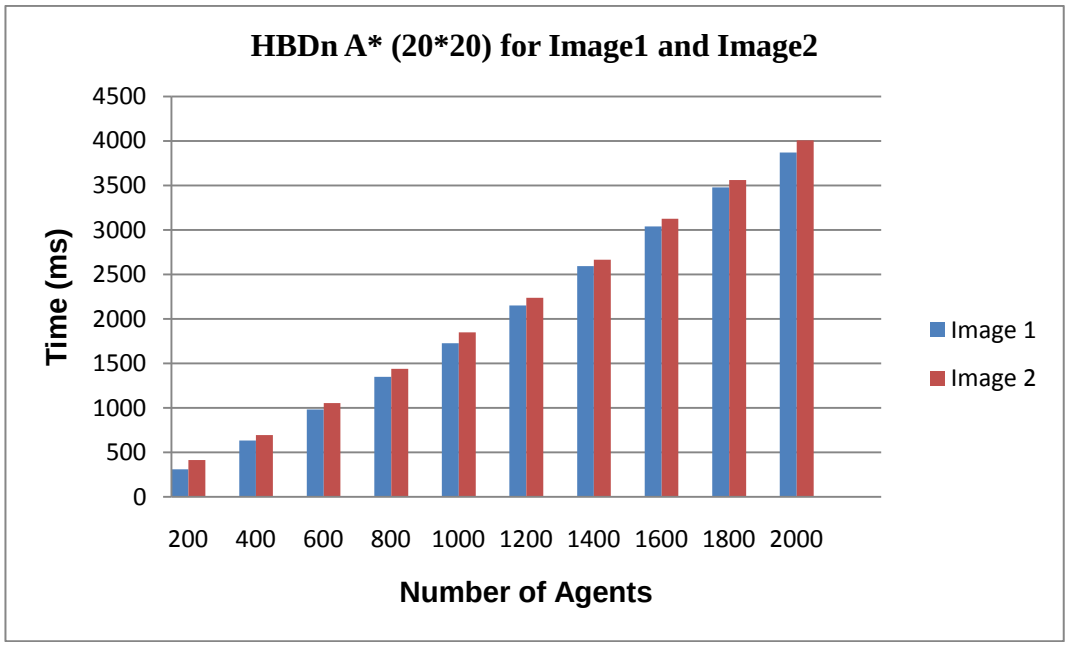
Parallel A* implementation provides irregular and changeable results for both the images as shown in Graph 6. When the walkable area increases (for the Image 2), the time for path calculation increases and at 1600 agents it becomes almost double as compared to time of Image 1.



Graph 6: Comparison of Parallel A for Image 1 and Image 2.*

4.4.2 Comparison of HBDn A* (20*20) Results for Image 1 and for Image 2

HBDn A* implementation provides more consistent and stable results for both the images as shown in Graph 7. It is not affected by the increase in the walkable area. Hence it is good for calculating paths on big images or images with more walkable or white space.



Graph 7: Comparison of HBDn A for Image 1 and Image 2.*

All these results explains that for the bigger sized images HBDn A* works much better than the Parallel A*. Moreover it gives very steady and consistent results even when the walkable area increases.

4.4.3 Suitable Cluster Sizes

Next we wanted to figure out the suitable cluster size for the Hierarchical breakdown A* implementation for GPU. We took results for these cluster sizes; 10*10, 15*15, 20*20, and 25*25. The total number of clusters for the Image 1, Image 2, and white image are given in Table 4 and Graph 8.

Image 1 has less number of clusters because of less walkable area, as no clusters are required on unwalkable areas. Image 2 is divided in more clusters as compared to Image 1 because it has more walkable area than Image 1. White Image (having all walkable area) is used to check the extreme values and to decide about the most appropriate cluster size; it is divided into the maximum possible number of clusters.

It is clear that the total number of clusters for the cluster size 10*10 is huge as compared to the other sizes; for Image 2 and white image (whole apace is walkable) it becomes more than double. The increase in the number of clusters also increases the memory requirements to calculate the paths that further reduces the number of agents that can run in parallel.

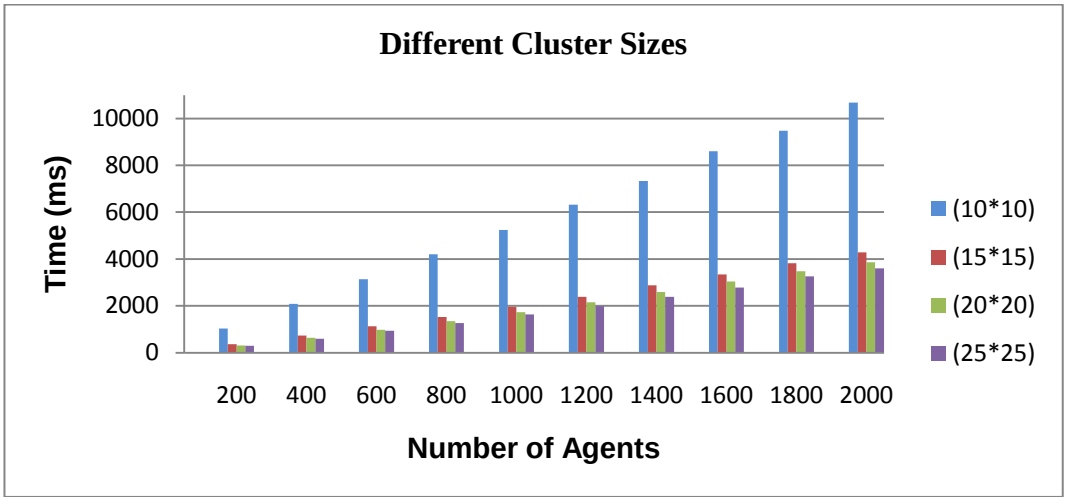
Image	No. of Clusters for Cluster Size (10*10)	No. of Clusters for Cluster Size (15*15)	No. of Clusters for Cluster Size (20*20)	No. of Clusters for Cluster Size (25*25)
Image 1	112	51	32	22
Image 2	174	71	42	25
White Image	182	81	46	30

Table 4: Different Cluster Sizes.

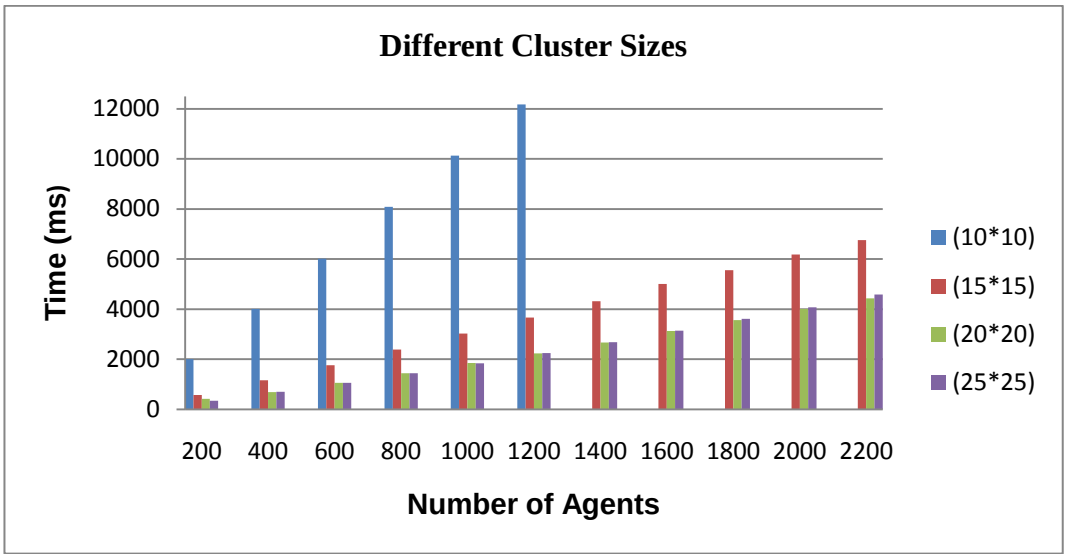
Graph 8: Different Cluster Sizes.

It is obvious from the Graph 9 (Table 10), and Graph 10 (Table 11) that the cluster size 10*10 takes much more time than the other cluster sizes for both Images 1 and 2. For Image 2 it cannot run for more than 1200 agents which mean that with the increase in walkable (or white) area the memory requirements for 10*10 cluster size increases rapidly and execution of more than 1200 agents becomes impossible.

The efficiency of the HBDn A* implementation is a tradeoff between speed and optimality. Path at the abstract graph is optimal but path at the lower level of hierarchy is sub-optimal. This sub-optimality increases with the increase in cluster size. We want to calculate path that is more close to the optimal path and takes less time to calculate. With the increased cluster size, the time to calculate path reduces but it gives less optimal path. While decreasing the cluster size, increases the optimality of the path but it also increases the time to calculate the path.



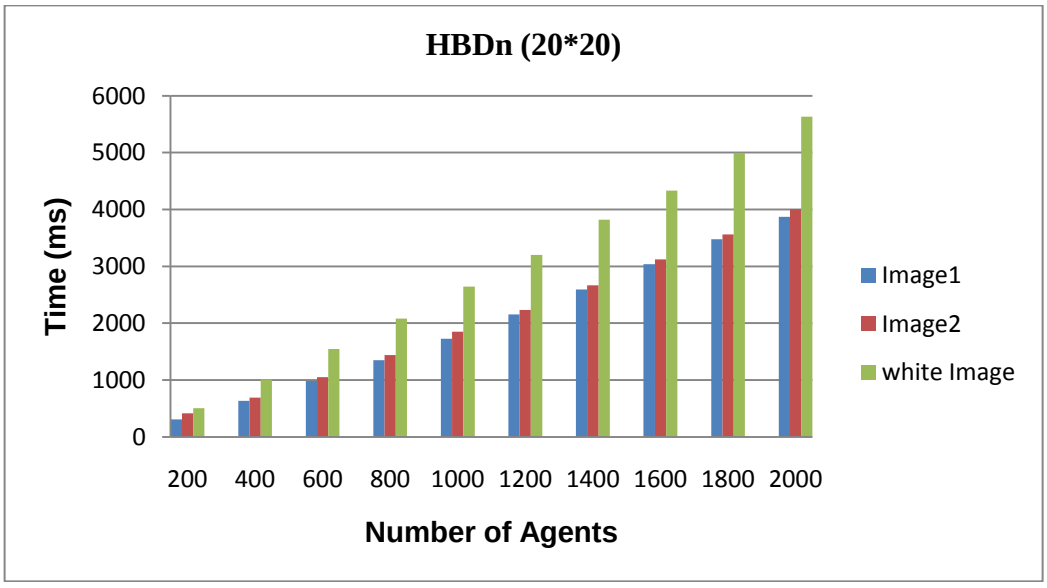
Graph 9: Results of Different Cluster Sizes on Image 1.



Graph 10: Results of Different Cluster Sizes on Image 2.

Appropriate Cluster Size

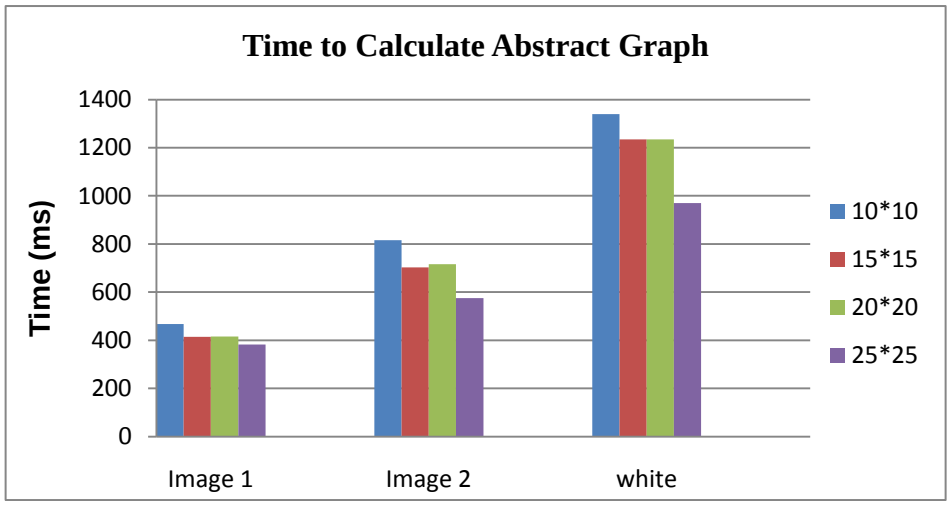
Therefore, examining these results, we can say that the cluster size 20*20 is appropriate. It takes less time to calculate paths than cluster sizes 10*10 and 15*15 and calculates more optimal paths than cluster size 25*25. Graph 11 presents the run of HBDn A* for the cluster size 20*20 for all the three images. It gives the stable results and for increased walkable area (like Image 2) and also for the white image the results are regular.



*Graph 11: Comparison of 20*20 Cluster Size for three Images*

4.4.4 Calculating the Abstract Graph

Abstract path calculation is one time activity that is performed at the start of the HBDn A*. The total amount of time to calculate the abstract graph for Image 1, Image 2, and white image for all cluster sizes is given in Table 5 and Graph 12. Results indicate that it takes very less time to calculate. Further different cluster sizes do not affect much the time required to calculate the abstract path.



Graph 12: Comparison of Time to Calculate Abstract Graph for three Images for different Cluster Sizes.

Figure 20 shows graphically all the abstract weighted graphs for Image 1 and Image 2 with the cluster size 20*20. It is clear that no cluster and graph is made for un-walkable area.

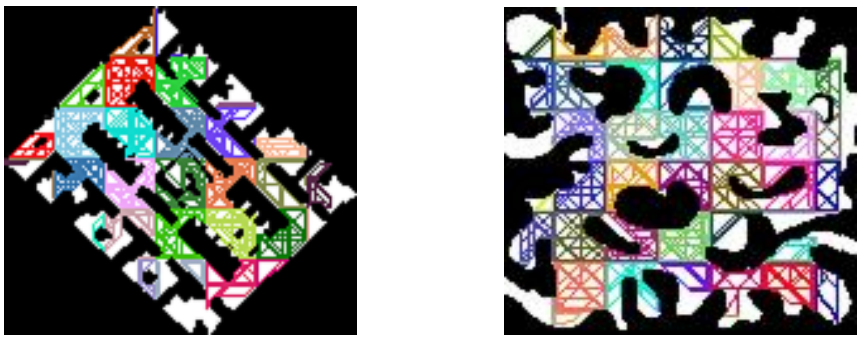


Figure 20: Abstract Weighted Graph for Image 1 and Image 2

Figure 21and Figure 22 represents the complete paths computed on Image 1 and Image 2 using HBDn with 500 agents and 1000 agents respectively.

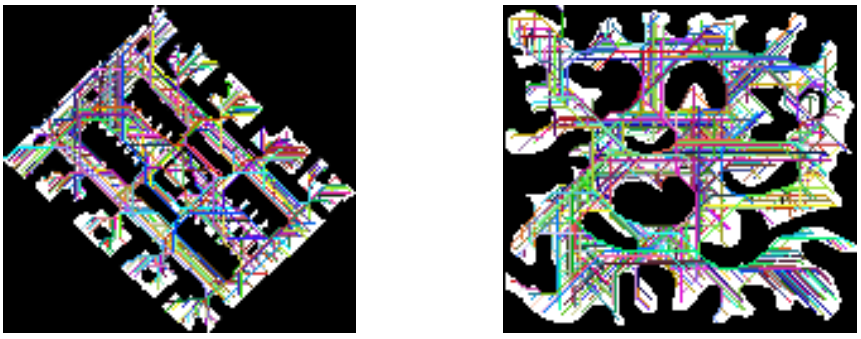


Figure 21: Complete paths for 500 Agents on Image 1 and Image 2

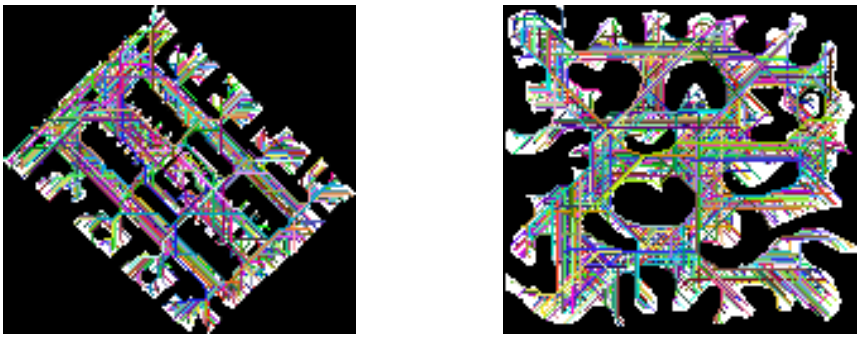


Figure 22: Complete paths for 1000 Agents on Image 1 and Image 2

4.5 Memory Footprints

Memory allocated on GPU for any A* implementation is actually the memory required for the arrays. The same amount of memory is allocated on CPU too. The arrays and their memory requirement for all implementations of the A* algorithms are given below:

4.5.1 Parallel A* Implementation

The arrays used are given here; n is the total number of agents that run in parallel.

Array	**Data type**	**Memory**
The Map	Char	mapWidth*mapHeight
Open list	Short	2 * n * mapWidth*mapHeight
Node List	Struct node	12 * n * mapWidth*mapHeight
Path list	Short	2 * n * mapWidth*mapHeight
Path Length	Short	2 * n
Start nodes	Short	2 * n
Target nodes	short	2 (ignoring constant)

Total memory allocated= ((16 * n +1)* mapWidth*mapHeight)) + (4 * n)

4.5.2 PrePath A* Implementation

The arrays used are given here, n is the total number of agents that run in parallel to calculate complete paths, P is the total number of agents run in parallel to compute pre paths.

Array	**Data type**	**Memory**
The Map	Char	mapWidth*mapHeight
Open list	Short	2 * n * mapWidth*mapHeight
Node List	Struct node	12 * n * mapWidth*mapHeight
Path list	Short	2 * n * mapWidth*mapHeight
PreOpen list	Short	2 * P * mapWidth*mapHeight
Pre list	Short	12 * P * mapWidth*mapHeight
Pre Path list	short	2 * P * mapWidth*mapHeight
Pre Path Lenth	Short	2 * P
Path Length	Short	2 * n
Pre start nodes	short	2 * P
Pre target nodes	short	2 * P
Start nodes	Short	2 * n
Target nodes	short	2 * n

Total memory allocated = ((16 * (n +P) + 1) * mapWidth*mapHeight) +
(6 *(n+P))

4.5.3 Multiple Threads/Agent A* Implementation

Array	Data type	Memory Requirement
The Map	Char	mapWidth*mapHeight
Open list	Short	2 * n * mapWidth*mapHeight
Node List	Struct node	12 * n * mapWidth*mapHeight
Path list	Short	2 * n * mapWidth*mapHeight
Path Length	Short	2 * n
Start nodes	Short	2 * n
Target nodes	short	2 * n

Total memory allocated = ((16 * n +1)* mapWidth*mapHeight)) + (6 * n)

4.5.4 HBDn A* Implementation

Path calculations are done in three steps; therefore, momory requirements for these three steps are given separately. The variable bSIZE is the size of the cluster. Each cluster has maximum 12 exit points. The variable startcount provides the number of walkable nodes within a particular cluster. Ncluster represents the total number of clusters.

1. **Abstract Path Computation:**

Array	Data type	Memory
The Map (adjacency graph)	Struct listnode	18 * (bSIZE * bSIZE)
Open list	Short	2 * startcount * (bSIZE * bSIZE)
Node List	Struct node	12 * startcount * (bSIZE * bSIZE)
Path list	Short	2 * (12*11) * (bSIZE*bSIZE)
Path cost	int	4 * (12*11)
Path Length	Short	2 * (12*11)
Start nodes	Short	2 * (12*11)
Target nodes	short	2 * (12*11)

Total memory allocated= 1320 + (($bSIZE^2$) * (282 + (14*startcount)))

2. Complete Path Computation:

First start and target nodes are added to the abstract graph.

Array	Data type	Memory
The Map (adjacency graph)	Struct listnode	18 * Ncluster * (bSIZE * bSIZE)
Open list	Short	2 * 24 * n * (bSIZE * bSIZE)
Node List	Struct node	12 * 24 * n * (bSIZE * bSIZE)
Path list	Short	2 * 24 * n * (bSIZE*bSIZE)
Path cost	int	4 * 24 * n
Path Length	Short	2 * 24 * n
Map Cluster	Short	Ncluster * (bSIZE*bSIZE)
Start nodes	Short	2 * 24 * n
Target nodes	short	2 * 24 * n

Total memory allocated = $(240 * n) + (384 * n * bSIZE^2) + (19 * Ncluster * bSIZE^2)$

Then complete path is calculated on the abstract graph. Here adjCount gives the total number of nodes on the abstract weighted graph.

Array	Data type	Memory
The Map (abstract weighted graph)	Struct listnode	34 * Ncluster * 12 * n
Open list	Short	2 * adjCount * n
Node List	Struct node	12 * adjCount * n
Path list	Short	2 * 12 * n
Path cost	int	4 * n
Path Length	Short	2 * n
Start nodes	Short	34 * n
Target nodes	short	34 * n

Total memory allocated = (98+ 14*adjCount + 408*Ncluster) * n

5 Conclusions

5.1 Summary of Contributions and Results

A* path-finding search algorithm is very famous in games for finding shortest distance between two nodes. Today's games have thousands of agents moving at a same time in the presence of obstacles. Thus it has become very important to find shortest paths concurrently and in a speedy way. Making use of GPU's highly parallel multi-threaded nature suits this scenario perfectly.

Implementing Simple A* algorithm using arrays (Parallel A*) has approximately the same results as compared to A* implementation using adjacent lists in [22]. Both implementations are greedy for space. Increase in the size of map increases the memory requirements and thus decreases the speed of algorithm.

To further increase the overall performance of algorithm, the memory requirements must be reduced. One option is to use the fast, read-only constant memory for storing the map. Pre-computing some paths and then sharing this already computed information with other agents further increases the efficiency.

Another solution to this problem is to exploit the parallel hardware architecture in a true sense. Some improvements are made in the basic A* algorithm to calculate each path using multiple threads that run concurrently and use shared memory and thread synchronization. It reduces the total search time of A* algorithm as compared to the Parallel A* implementation. Binary heap is the bottleneck for multiple threads, as only one thread can access the heap at a time and all the other threads have to wait; thread synchronization is done to achieve this.

Hierarchical Breakdown of A* algorithm (HBDn) is faster and more consistent solution for big-sized maps. It gives long paths on big search areas. To overcome the high memory needs for the larger maps, the search space is divided into smaller areas called clusters. The path is calculated using two levels of hierarchy. At the higher level, an abstract weighted graph is made that connects these clusters. Then multiple paths are computed on this abstract weighted graph which is much smaller in size than the original map. And in the last, the complete paths are found using path refinements. The abstract weighted graph calculation is one time activity and takes very less time. Our results show that the most appropriate cluster size using HBDn for the graphics processors is 20*20. There is

a trade-off between performance and optimality using HBDn. The path at abstract graph level is optimal but the path at lower level of hierarchy is sub-optimal.

5.2 Limitations

The biggest overhead in implementing A* algorithm on graphics processor is the memory requirements for the algorithm. Memory needs increase with the increase of agents and with the increase in map size. To run more number of agents in less time Pre-Path, Multiple threads, and HBDn techniques are adopted.

Pre-Path uses already computed and stored paths, but gives static results. In case of Multiple threads the binary heap becomes the bottleneck for the multiple threads as only one thread at a time can access it. Both Pre-Path and Multiple threads are feasible for small sized maps. HBDn is a good solution for large search spaces. The limitation here is the compromise on path optimality. The path calculated at lower level of hierarchy is sub-optimal.

5.3 Future Recommendations

Many ideas related to the efficient A* implementation on graphics cards can be tried in future. Some of these ideas are discussed below.

- To overcome bottleneck of the binary heap while using multiple threads, the Lock-free or wait-free implementations of the binary heap could be done.

- Many levels of hierarchies could be used in Hierarchical breakdown implementation of A* algorithm.

- Some heuristic method can be explored that is more suitable to calculate the abstract weighted graph in HBDn.

- Pre-Paths can be combined with the Multiple threads and HBDn implementation.

- Maps used only walkable or un-walkable. Maps with Heterogeneous environment (of different weights) can be used.

6 Appendix A

We provide the actual results we have taken. For each value, we took at least 20 results and calculated the average value of it. All the values in the following tables are average of 20 results.

Image	10*10 Time (ms)	15*15 Time (ms)	20*20 Time (ms)	25*25 Time (ms)
Image 1	467.1384013	414.9552805	415.5135423	382.557685
Image 2	816.312173	702.9571197	716.256325	575.3688007
White Image	1339.498187	1234.609144	1235.187572	970.0900207

Table 5: Comparison of Time to Calculate Abstract Graph for three Images for different Cluster Sizes.

No. of Agents	A* HBDn(20*20) Time (ms)	Parallel A* Time (ms)
200	310.295697	419.492927
400	633.430503	786.2296727
600	981.5383923	1212.887928
800	1349.686914	1589.904878
1000	1726.962503	1878.784448
1200	2152.610175	2341.177945
1400	2591.491325	2719.14503
1600	3039.83062	3053.868284
1800	3477.315319	Cannot run
2000	3867.716315	Cannot run

Table 6: Comparison of Hierarchical Breakdown of A with Simple Parallel A*for Image 1.*

No. of Agents	A* HBDn(20*20) Time (ms)	Parallel A* Time (ms)
200	414.259284	793.1963873
400	693.5839555	1563.613424
600	1054.029649	2333.547868
800	1438.116202	3041.44362
1000	1850.667655	3609.353704
1200	2235.941427	4442.295688
1400	2663.860195	5162.952208
1600	3124.887066	5869.93564
1800	3560.26511	Cannot run
2000	4007.588992	Cannot run
2200	4430.429601	Cannot run

Table 7: Comparison of Hierarchical Breakdown of A with Simple Parallel A*for Image 2.*

No. of Agents	Time (ms) for Image 1	Time (ms) for Image 2
200	419.492927	793.1963873
400	786.2296727	1563.613424
600	1212.887928	2333.547868
800	1589.904878	3041.44362
1000	1878.784448	3609.353704
1200	2341.177945	4442.295688
1400	2719.14503	5162.952208
1600	3053.868284	5869.93564

Table 8: Comparison of Parallel A for Image 1 and Image 2.*

No. of Agents	Time (ms) for Image 1	Time (ms) for Image 2
200	310.295697	414.259284
400	633.430503	693.5839555
600	981.5383923	1054.029649
800	1349.686914	1438.116202
1000	1726.962503	1850.667655
1200	2152.610175	2235.941427
1400	2591.491325	2663.860195
1600	3039.83062	3124.887066
1800	3477.315319	3560.26511
2000	3867.716315	4007.588992

Table 9: Comparison of HBDn A for Image 1 and Image 2.*

No. of Agents	(10*10)	(15*15)	(20*20)	(25*25)
200	1032.261977	358.0015175	310.295697	298.5838325
400	2082.390204	727.648158	633.430503	599.381222
600	3138.962177	1121.668306	981.5383923	930.538533
800	4201.372564	1524.015241	1349.686914	1258.350996
1000	5240.296618	1967.859478	1726.962503	1632.391788
1200	6326.44594	2386.658102	2152.610175	1992.89369
1400	7327.145975	2873.504065	2591.491325	2389.658523
1600	8606.622241	3346.890597	3039.83062	2785.949073
1800	9473.542007	3816.284394	3477.315319	3265.061032
2000	10678.64857	4283.857284	3867.716315	3601.496391

Table 10: Results of Different Cluster Sizes on Image 1.

No. of Agents	(10*10)	(15*15)	(20*20)	(25*25)
200	2006.904477	578.591651	414.259284	343.1814625
400	4015.2253	1162.843245	693.5839555	699.837726
600	6018.922134	1766.11961	1054.029649	1055.378761
800	8083.187373	2383.093149	1438.116202	1442.894864
1000	10135.05026	3022.72967	1850.667655	1842.208581
1200	12172.85929	3663.664697	2235.941427	2249.605382
1400	Can't run	4313.022332	2663.860195	2687.098331
1600	Can't run	5006.114418	3124.887066	3146.321965
1800	Can't run	5555.239019	3560.26511	3607.322152
2000	Can't run	6183.904121	4007.588992	4077.845432
2200	Can't run	6761.307409	4430.429601	4583.212843

Table 11: Results of Different Cluster Sizes on Image 2.

References

[1] NVIDIA CORPORATION, CUDA Programming Guide, http://developer.nvidia.com/cuda

[2] P.E. Hart, N.J.Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths". *IEEE Transactions on System Science and Cybemetics*. 4 pp 100-107, 1968.

[3] A* search algorithm, Wikipedia, http://en.wikipedia.org/wiki/A*, Last visited 11th Jan, 2009

[4] Amit's A* Pages http://theory.stanford.edu/~amitp/GameProgramming/, last visited 11th Jan, 2009

[5] Rina Dechter and Judea Pearl, "Generalized best-first search strategies and the optimality of A*", *Journal of The ACM*, Volume 32, Issue 3, Pages: 505 - 536, 1985.

[6] NVIDIA CORPORATION, CUDA Reference Manual, http://developer.nvidia.com/cuda

[7] Path Finding - A* Algorithm http://www.edenwaith.com/products/pige/tutorials/a-star.php, last visited 11th Jan, 2009

[8] T. Cazenave, "Optimizations of Data Structures, Heuristics, and Algorithms for Path-Finding on Maps". *IEEE Symposium on Computational Intelligence and Games*, 2006

[9] A* algorithm tutorial http://www.geocities.com/jheyesjones/astar.html, last visited 12th Feb, 2009

[10] A. Botea, M. Muller, and J. Schaeffer. "Near Optimal Hierarchical Path-Finding". *Journal of Game Development*, 1(1), pp. 7-28, 2004.

[11] R. E. Korf, "Depth First Iterative Deeping: An Optimal Admissible Tree Search", *Journal of Artificial Intelligence*, pp. 97-100, 1985

[12] R. E. Korf, "Real-time Heuristic Search". *Artificial Intelligence*, 42(2-3), pp. 189-211, 1990.

[13] B. Reese and B. Stout, "Finding a pathfinder". *Proceedings of the AAAI Symposium on Artificial Intelligence and Computer Games*, 1999.

[14] R. Holte, M. Perez, R. Zimmer, and A. MacDonald. "Hierarchical A*: Searching Abstraction Hierarchies Efficiently". *In Proceedings AAAI-96*, pp. 530-535, 1996.

[15] R. Holte, T. Mkadmi, R. Zimmer, and A. MacDonald. "Speeding Up Problem-Solving by Abstraction: A Graph Oriented Approach". *Artificial Intelligence Journal*, 85(1-2), pp. 321-361, 1996.

[16] D. Harabor, A. Botea. "Hierarchical Path Planning for Multi-Size Agents in Heterogeneous Environments". *IEEE Symposium on Computational Intelligence and Games*, 2008

[17] N. Sturtevant and M. Buro, "Partial Pathfinding Using Map Abstraction and Refinement", *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 1392-1397, 2005.

[18] V. Bulitko, N. Sturtevant, and M. Kazakevich, "Speeding up learning in real-time search via automatic state abstraction". In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1349–1354. 2005

[19] N. Sturtevant, and M. Buro. "Improving Collaborative Pathfinding Using Map Abstraction". *In AIIDE*, pages 80--85, 2006

[20] V. Bulitko, N. Sturtevant, J. Lu, T. Yau. "Graph Abstraction in Real-Time Heuristic Search". *Journal of Aritifial Intelligence*, 30, pp. 51-100, 2007.

[21] R.Jansen, N. Sturent, "A new approach to cooperative pathfinding". *Proceedings of the 7th conference on Autonomous agents and multiagent systems*. Pp. 1401-1404, 2008.

[22] GPU Accelerated Pathfinding, Avi Bleiweiss, In proceedings of Graphics Hardware, 2008, Pages 65 – 74, 2008