Collaborative Diffusion on the GPU for Path-finding in Games

Craig McMillan, Emma Hart, Kevin Chalmers

Institute for Informatics and Digital Innovation, Edinburgh Napier University
Merchiston Campus, Edinburgh, EH10 5DT
{c.mcmillan,e.hart,k.chalmers}@napier.ac.uk

Abstract. Exploiting the powerful processing power available on the GPU in many machines, we investigate the performance of parallelised versions of pathfinding algorithms in typical game environments. We describe a parallel implementation of a collaborative diffusion algorithm that is shown to find short paths in real-time across a range of graph sizes and provide a comparison to the well known Dijkstra and A* algorithms. Although some trade-off of cost vs path-length is observed under specific environmental conditions, results show that it is a viable contender for pathfinding in typical real-time game scenarios, freeing up CPU computation for other aspects of game AI.

Keywords: GPU; Collaborative Diffusion; Path-finding; Parallel; Games

1 Introduction

AI algorithms are one of the last areas within a typical game engine to exploit the computational power found in modern GPUs. With pathfinding algorithms being used within many games for navigation of computer controlled characters, developers could potentially exploit GPU hardware for this processing, hence freeing up the CPU for other more complex operations.

However developers need to carefully consider the types of algorithms that can be run on the GPU. Traditional pathfinding algorithms are often considered difficult to parallelize due to their highly divergent nature [1, 2] which can have a significant impact on performance [3]. As a result it is necessary to consider algorithms with minimal divergence in which little or no effort is required to separate the problem into a number of parallel tasks. Such algorithms are described as *embarrassingly parallel* in order and are likely to result in the most effective use of the hardware.

We present a novel GPU implementation of the *collaborative diffusion* algorithm [4] which is embarrassingly parallel in its implementation and to the best of our knowledge has not been previously undertaken. We provide a comparison against two widely used pathfinding algorithms; A* [5] the most commonly used within games and parallel and sequential versions of Dijkstra [6] which is guaranteed to find the shortest path. The version of Dijkstra used exploits dynamic parallelism, an efficient use of hardware that makes it possible to launch kernels from threads running on the device. We evaluate the performance of each of the algorithms in a general path search across a range of different map sizes before looking at scenarios and environments which would typically be found within a game. Results show that a GPU implementation of diffusion is able to find paths within 60 frames per second, often considered real-time within the games industry and scalable to large maps. We also show that the path length and cost of diffusion is comparable to A* and Dijkstra within game environments.

The remainder of this work is organised as follows. The next section reviews related work in the area of GPU pathfinding. Section 3 provides an overview of the A*, Dijkstra and collaborative diffusion algorithms while Section 4 outlines the CUDA API. Section 5 gives a detailed description of parallelising the algorithms. Section 6 discusses the results from the experiments carried out. Finally Section 7 presents the concluding remarks and future work to be carried out.

2 Related work

Caggianese and Erra [7] describe a parallel version of A* which uses grid space decomposition to parallelise the algorithm such that it is suitable for the GPU using CUDA. They compared their approach to a GPU implementation of Real-Time Adaptive A* (P-RTAA*). Algorithms were tested on grids upto a size of 1024x1024 for multiple agents up to 262144. The approach proved faster than P-RTAA* for each group of agents tested achieving speed ups up to 45X. The algorithm found paths of similar length to A*, but showed a trade off in speed vs optimal path when using different sizes of planning blocks. However as speed is likely to be the hightest priority in real-time applications, longer sub-optimal paths may be an acceptable trade-off.

Ortega-Arranz *et al* [1] present a parallel implementation of Dijkstra's algorithm. Their implementation parallelizes the internal operations of the algorithm, exploiting the two internal loops within the algorithm. An *outer loop* is responsible for selecting a new current node while the *inner loop* checks each of the current nodes neighbours and calculates the new distance values. The *outer loop* is parallelized by simultaneously selecting multiple nodes that can be settled in parallel without affecting the algorithm correctness, and the *inner loop* by looking at each neighbour of a node in parallel. Ortega-Arranz *et al* compare their parallel implementation to an equivalent sequential implementation across a range of different graph sizes, achieving a 13X to 220x speed up with their GPU implementation compared to the CPU sequential implementation across the graph sizes tested.

3 Pathfinding Algorithms

We describe two potential but contrasting approaches for pathfinding in a game environment that lend themselves to parallelization on the GPU. For completeness, A* is also described given its frequent use in games.

3.1 Traditional Pathfinding Algorithms

Dijkstra's algorithm [6] solves the single-source shortest path problem for a non-negative weighted graph. Nodes in the graph can represent either a 2-dimensional grid or points in 3D space. For a given source node the algorithm finds the lowest cost path between that node and every other node in the graph. As Dijkstra will evaluate every node in the graph it can be computationally expensive. As a result it is rarely used within games.

A* [5] is an extension to Dijkstra's algorithm. Unlike Dijkstra, A* uses heuristics to determine which nodes in the graph to settle. A node is considered settled once the shortest path between that node and the source node has been found. This allows A* to search only a fraction of the nodes in a graph in order to find a low cost path between two nodes. This has led to A* becoming commonly used within many games due to its low computational overhead and ability to find low cost paths in real-time.

3.2 Collaborative Diffusion

Repenning [4] introduces Collaborative Diffusion, inspired by physical processes that spread matter over an N-dimensional physical or conceptual space over time. Repenning introduced the idea of using diffusion based techniques and the idea of antiobjects to find paths as a means to address some of the limitations of traditional path search algorithms [4]. *Collaborative diffusion* is based on the notion of programming concept of *antiobjects* — an object that appears to do the opposite of what might be expected [4]. For example, in the game Pac-Man it might be natural to assign path-finding computation to each of the ghost agents. However, by assigning this computation to the objects that define the background instead, opportunities for redistributing the computation in a manner that can be parallelized may improve performance. This results in a great deal of the computational intelligence being shifted into the environment; although the ghost agents appear to be intelligent, in reality the intelligence lies in the background which becomes a computational reflection of the game topology, including the player location and their state. This allows agents to collaborate and compete, resulting in sophisticated emergent behaviours. In addition to naturally emergent behaviours occurring, an added benefit is that the number of agents pathfinding within an environment also has no effect on the time taken by the algorithm to find a path as path searching is handled independently of the agents. As a result the use of antiobjects enables games with large numbers of agents to be built.

Collaborative Diffusion is typically used within a tile-based environment, each goal node is given an initially high diffusion value which is used as the starting node: over time this value is spread throughout the world according to equation 1 until each passable tile obtains its own value. D refers to the diffusion value, n the number of neighbouring nodes, a the diffusion value of a neighbour, and κ is the cost of the node. Neighbours are defined by a metric appropriate to the graph; in a typical 2-dimensional grid-world, the *Moore* neighbourhood comprising of the eight cells surrounding any given cells is typically used. Once values have diffused across the complete graph, a path can be backtracked to the goal from any node by simply moving to the neighbouring node with the largest value.

$$D = (\frac{1}{n}\sum_{i=0}^{n} a_i) \cdot \kappa \tag{1}$$

To the best of our knowledge no research has been undertaken into a parallel GPGPU implementation of diffusion to perform searching. However, Sanders *et al* [8] provide an example of a GPU implementation of simulating heat transfer via a similar diffusive process, and discuss performance considerations relating to how a grid is stored.

4 CUDA Overview

The Compute Unified Device Architecture (CUDA) platform [9] is a general purpose computing API that has been designed for performing tasks that would traditionally be executed on the CPU, on the GPU. CUDA was first released in 2007 by NVidia and is a proprietary platform for Nvidia hardware. CUDA applications consist of host code which is executed on the CPU and device code, also known as *kernel* code, which is executed on the GPU. The host system communicates with the device by copying over a set of data and then giving the device a task to perform on the data. This is typically done by splitting the work into multiple threads and each thread then performing the same operation on different parts of the dataset. While offering obvious opportunities for speed-up in comparison to the CPU, care must be taken in porting code. For example, branch divergence can lead to threads no longer being executed in parallel, with their computation becoming serialized. This can result in lower performance as the execution units on the GPU are not being fully utilized.

New with NVidia's Kepler GK110 architecture [10] is a *dynamic parallelism* feature that allows the GPU to launch threads directly and generate new work for itself without ever having to involve the CPU. This results in the ability to run more of an application on the GPU as kernels running on the device are able to adjust the number of threads being launched depending upon the needs of the application, improving both scalability and performance through support for more varied parallel workloads and freeing up the CPU for other operations.

5 Parallelising the Path-Finding Algorithms

5.1 Dijkstra

The Dijkstra algorithm partitions all nodes into two distinct sets, *unsettled* and *settled*. Initially all nodes are in the unsettled sets, e.g. they must be still evaluated. A node is moved to the settled set if a shortest path from the source to this node has been found. As outlined by Ortega-Arranz *et al* [1], Dijkstra can be thought of as operating around two loops: the *outer loop* selects the lowest cost node from the unsettled set while the unsettled set is not empty whilst the *inner loop* evaluates each of the current lowest cost nodes neighbours. Each node in the unsettled set must be checked one at a time with the fastest sequential implementations being based around an efficient implementation of a minimum priority queue to allow fast retrieval of the lowest cost node. Algorithm 1 shows the pseudo code for the *outer* and *inner* loops of the sequential Dijkstra implementation.

Algorithm 1 Sequential Dijkstra Outer and Inner Loops

```
1: while unsettledSet != empty do
2:     currentNode = node in unsettledSet with minimum distance[d]
3:     remove currentNode from unsettledSet
4:     for each neighbour n of currentNode do
5:         //Calculate the cost of passing through the neighbour
6:         cost = distance[currentNode] + length(currentNode, n)
7:         if cost < distance[n] then
8:             //Update the cost and parent of the neighbour
9:             distance[n] = cost
10:            previous[n] = currentNode
11:            add n to unsettledSet
12:        end if
13:    end for
14: end while
```

The outer `while` loop can be parallelised by selecting multiple nodes from the unsettled set and settling them at the same time without effecting the algorithm's correctness. This set of nodes are known as the frontier set which is generated by selecting any node in the unsettled set which has a tentative distance of the current *minimum* + 1.

Our parallel implementation makes use of CUDA *dynamic parallelism* allowing the entire Dijkstra process to be run directly on the GPU. Algorithm 2 outlines the parallel Dijkstra implementation.

5.2 Diffusion

The parallel diffusion algorithm is given in Algorithm 3. The sequential version differs in the manner in which the grids are stored in memory so that they can be accessed appropriately, with an additional process in the parallel version to pass the grids between host and device and vice versa. The speedup of the parallel version comes from the ability to process each node in the grid at the same time — this is easy to achieve as diffusion is embarrassingly parallel, meaning the problem can be split into a number of parallel tasks as each node has exactly the same calculation applied to it.

At the start of each iteration of the `while` loop, the value of the goal node is set so that it remains constant and spreads evenly throughout the grid. The goal value is then diffused over the grid according to Equation 1. The diffusion

Algorithm 2 Parallel Dijkstra Outer and Inner Loops

```
1: while unsettledSetEmpty == FALSE do
2:     //Get the minimum cost in the unsettled set
3:     for all nodes in unsettled Set do                          ▷ Executed in parallel
4:         getMinimumCost()
5:     end for
6:     //Create the frontier set
7:     for all nodes in unsettled Set do                          ▷ Executed in parallel
8:         if nodeCost[threadID] <= globalMin + 1 then
9:             addToFrontier[threadID]
10:        end if
11:    end for
12:    //Evaluate frontier set
13:    for all nodes in frontier Set do                           ▷ Executed in parallel
14:        evaluateNode()
15:    end for
16:    //Check if the unsettled set is empty
17:    if unsettledSet == EMPTY then unsettledSetEmpty = TRUE
18:    end if
19: end while
```

Algorithm 3 Parallel Diffusion

```
1: while gridDiffused != TRUE do
2:     setGoals(inputGrid, goalGrid)          ▷ Executed in parallel for all nodes in map
3:     diffuseGrid(outputGrid, inputGrid)     ▷ Executed in parallel for all nodes in map
4:     if CheckGridDiffused(outputGrid) then
5:         gridDiffused == TRUE
6:     end if
7:     swapGrids(inputGrid, outputGrid)
8: end while
```

value is multiplied by the node cost κ in each calculation. Costs are set between 0 and 1 (where zero represents an impassable tile and 1 a fully passable tile). The diffusion value thus decays quickly in areas with a value closer to zero. Once the goal value has spread to every tile in the graph, it is possible to backtrack a path from any point in the graph.

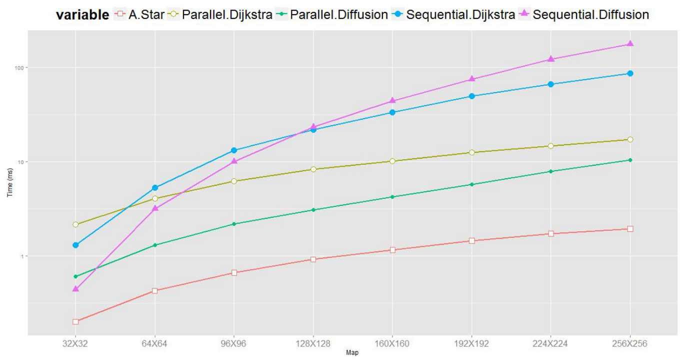
6 Results

Sequential and parallel versions of Dijkstra and diffusion were implemented, alongside a sequential version of the A* algorithm. Sequential versions were written in C++, and parallel versions using the CUDA API.

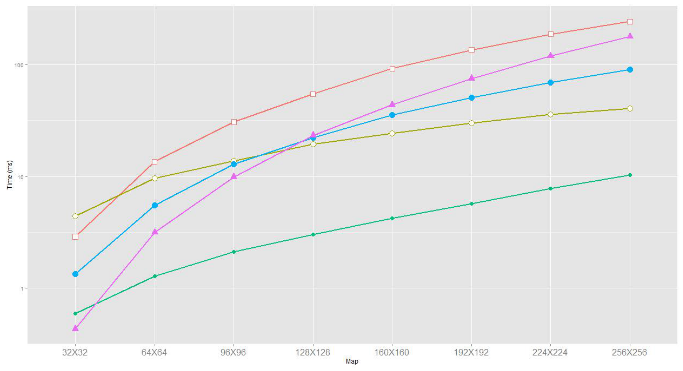
Each algorithm was tested on 2D-grids ranging in size from 32×32 to 256×256. Two scenarios were tested; in the first each node had equal weight, while in the second each node was assigned a random weight between one and ten. For the diffusion algorithms this weight was normalized and inverted so that the weight is a value between zero and one. The source node was always in the top-left corner, and the goal in the bottom-right, giving paths with the maximum possible Euclidean distance. Sequential versions of the algorithms were run on a Intel(R) Core(TM) i7-2600K 3.4GHz CPU while the parallel versions were run on an NVidia Tesla K40 GPU. All experiments were repeated 10 times and the time in milliseconds to obtain a path to the goal recorded. In each experiment the *cost* of a path was also noted.

6.1 Time to find Goal Node

Figure 1 compares the results of all five algorithms, showing the average time taken to find a path across a range of graph sizes in which (a) all nodes had equal weight (b) nodes were assigned a random weight between one and ten.



(a) Equal weighted graphs



(b) Random weighted graphs

Fig. 1: Time (ms) to find goal node for sequential and parallel algorithms across equal (a) and random (b) weighted graphs

In 1(a) we clearly see that A* outperforms both the sequential and parallel versions of Dijkstra and Diffusion. Collaborative Diffusion is faster than both the parallel and sequential versions of Dijkstra. In contrast, sequential diffusion fares worst of all the algorithms except in the single case of the smallest graph.

The parallel version of Dijkstra scales better than its corresponding sequential version. This is due to the fact that when weights are equal sequentially Dijkstra requires n^2 iterations of the outer loop to find a path compared to only n iterations in the parallel version for an $n \times n$ grid like those used in the tests. The situation is reversed however when using graphs with randomly weighted nodes.

In the weighted graphs 1(b), A* is the slowest algorithm in all but the smallest graph; upto $5\times$ slower than parallel Dijkstra and upto $24\times$ slower than the parallel diffusion implementation. Parallel diffusion is fastest in all cases except for the smallest graph. Parallel Dijkstra is slower than its sequential counterpart until a graph size of 160×160 is reached.

The poor performance of both A* and parallel Dijkstra in weighted graphs can be easily explained. A* is particularly successful in graphs with nodes of equal cost due to the use of a heuristic that enables it to search only a small fraction of the graph. However when node costs are random, the algorithm needs to search many more nodes. A* evaluated more than 90% of the nodes in each of the weighted graphs compared to between 0.5% and 4% of nodes when the weights are equal. The performance of Dijkstra suffers when there is a large variation in weight between nodes due to the way in which the *outer loop* and *frontier set* works as when there is a large variation in weight between nodes, fewer can be settled simultaneously as they do not meet the criteria to be added to the frontier set, resulting in extra iterations of the *outer loop* required to process all the nodes in the graph. The performance of parallel diffusion remains relatively unchanged between the equal and randomly weighted graphs unlike that of A* and Dijkstra suggesting that while A* is faster in maps where all weights are equal, diffusion may be the better choice of algorithm in cases where weights are all random or where maps within a game are likely to change between areas of equal and random weight. Thus the performance of diffusion is not effected by the weights within a graph.

6.2 Path Length vs path cost

In many computer games, it is necessary to distinguish between paths of short length and paths of low cost. For example, the shortest path to a goal might involve crossing terrain such as water or a swamp that expends more energy than moving across grass. Therefore, it is of interest to compare the length of paths found by each of the algorithms to the cost. 20 random 128×128 grids were generated. In each instance, costs were randomly assigned to each node in the range one to ten. Figure 2 shows the path length (number of nodes in path) plotted against the path cost (summed cost of nodes in path) for each of the algorithms. As expected both A* and Dijkstra find low cost paths through each of the graphs. However, when compared to diffusion, the paths are of a greater length, with diffusion finding paths that are between 30 and 51 nodes shorter. This comes with the trade-off however of paths being between 90 and 195 units greater than those found by A* or Dijkstra. In games where the cost of the path is critical to the game play diffusion may not be the most suitable algorithm.

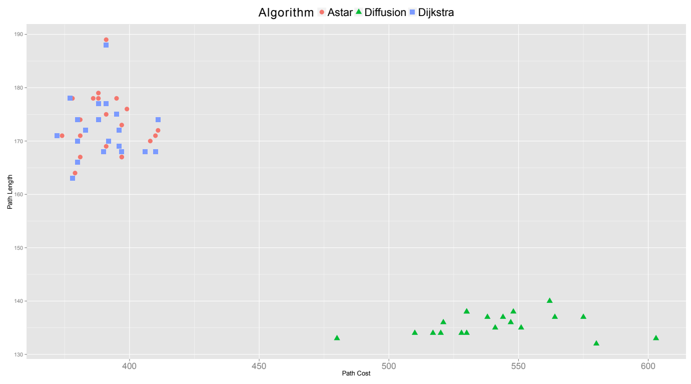


Fig. 2: Path Length vs Path Cost

6.3 Obstacles

In addition to varying terrain types, the environment in a computer game is also likely to contain obstacles which are impassable such as walls or large buildings. Computer controlled characters must be able to find paths around these obstacles. To evaluate the performance of A*, parallel Dijkstra and parallel Diffusion in more realistic scenarios, we generate 4 maps with varying types of terrain and obstacles. For each of the maps we record the average time in milliseconds across 10 runs to find a path, along with the path length and the path cost. In each case the start node is the top left corner and the goal node is the bottom right corner. Figure 4 compares the paths found by both A* and Diffusion. Dijkstra has been omitted from this comparison as the paths found were near identical to A* and it took over 5× longer to find the paths than either of the other algorithms. Figure 3 records the time, length and cost of each experiment. In each case, t-tests show that the time difference is statistically significant at a 95% confidence level.

Algorithm	Time (ms)	Path Length	Path Cost
Diffusion	0.61	**32**	50
A*	**0.52**	39	**42**
Dijkstra	2.87	39	**42**

(a) Map 1

Algorithm	Time (ms)	Path Length	Path Cost
Diffusion	**0.59**	**49**	**113**
A*	1.53	**49**	**113**
Dijkstra	6.08	**49**	**113**

(b) Map 2

Algorithm	Time (ms)	Path Length	Path Cost
Diffusion	0.75	**39**	**39**
A*	**0.30**	41	41
Dijkstra	2.34	**39**	**39**

(c) Map 3

Algorithm	Time (ms)	Path Length	Path Cost
Diffusion	1.27	**75**	147
A*	**1.04**	114	**114**
Dijkstra	8.07	114	**114**

(d) Map 4

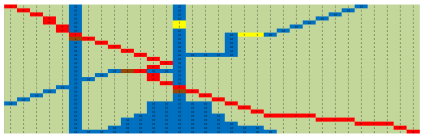
Fig. 3: Time (ms), path length and path cost for the different algorithms across the different maps.

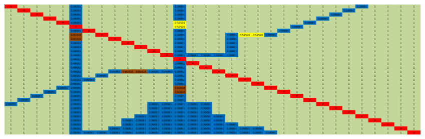
- **Map 1** : the path choice involves crossing a stream or using a lower cost bridge. A* opts for the lower cost moves, resulting in crossing the bridges (cost 42, path length 39), whereas diffusion 4(b) moves in a straight line (cost 50, path length 32), and is 1.17 times slower than A*.
- **Map 2**: approximately 50% of the map is covered with high cost terrain. Both A* 4(c) and diffusion 4(d) found a path to the goal by moving through the area of lower cost terrain taking 49 nodes to reach the goal with a cost of 113. Although the paths are of equal length and cost, diffusion takes a straighter path with fewer turns, which is found 2.6 times faster than A*.
- **Map 3** : this contains a large number of impassable obstacles spread out across the map resulting in a large number of possible routes to the final goal. Diffusion 4(f) finds a shorter and cheaper path than A* 4(e) (cost, length = 39 vs 41 for A*). In this case A* was able to find a path 2.5 times faster than diffusion, albeit only with a difference of 0.45ms.
- **Map 4**: this contains a large number of nodes with high cost terrain separated by areas of impassable obstacles. This map provided the largest variation between the path length and path cost for each of the algorithms. A* 4(e) found a longer path taking 114 nodes to reach the goal compared to diffusions 4(h) 75 nodes. However the path found by A* is less costly than diffusion (114 vs 147). A* avoids all areas of terrain whereas diffusion takes short-cuts through the high cost areas. A* is 1.2 times faster in this case.

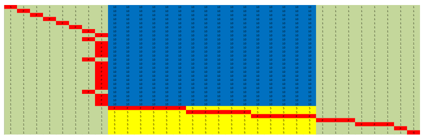
The results obtained in Sections 6.1 and 6.2 show that in grids with nodes of equal weight, A* outperforms the parallel algorithm, with the converse being true on grids with randomly assigned weights. However, on maps 1-4 which attempt to reflect typical game scenarios, although diffusion performs fastest in only 1 out of 4 scenarios, all paths are found comfortably within 'real-time'. Figure 2 which analysed randomly weighted graphs suggests that diffusion finds paths of short-length at the expense of high-cost. Although this is the case in maps (1) and (4), in map (3) diffusion was able to find a path both shorter and less costly than A* and a path of equal length and cost in map 2. It appears that in an environment in which nodes are weighted in a structured rather than random manner, diffusion is a viable choice of method. Ultimately, the choice of algorithm will depend in the specific game in question, but the performance of diffusion in this respect may outweigh cost issues in many cases.

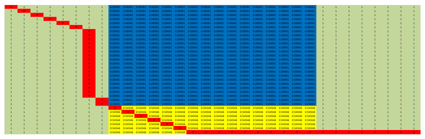
7 Conclusions

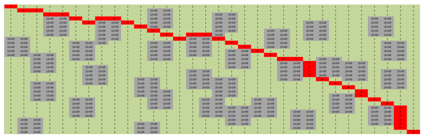
This paper has analysed the ability of three different path-finding algorithms to find paths through environments that are representative of those encountered in typical computer games, i.e. contain varied terrain or obstacles. In particular, we investigate whether path-finding algorithms such as Djikstra that traditionally might be thought of as too computationally expensive to be utilised within a game can be made tractable through exploiting the GPU. Additionally, we evaluated a path-finding algorithm called Collaborative Diffusion that has received little attention within the AI/Games literature.

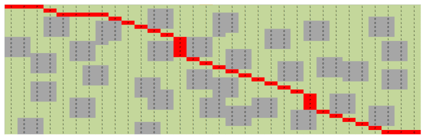

(a) A* Map 1

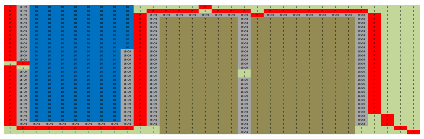

(b) Diffusion Map 1


(c) A* Map 2


(d) Diffusion Map 2


(e) A* Map 3


(f) Diffusion Map 3


(g) A* Map 4

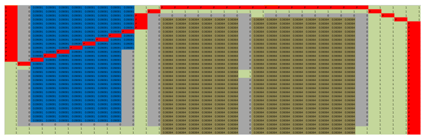

(h) Diffusion Map 4

Fig. 4: Paths for A* and Diffusion through different map types

A GPU implementation of diffusion performs particularly well in environments in which nodes do not have equal costs in terms of the time taken to find a path. This is particularly relevant for the majority of computer games in which environments consist of mixed terrain with variable energy costs for passing through. Timing results show that diffusion can be run on the GPU in real-time for graph sizes up to approximately 256x256 nodes: these sizes are well within the bounds of typical computer games — games such as Civilisation, StarCraft and SimCity all run in environments that are smaller than this. Diffusion typically finds shorter paths than either A* or Dijkstra, both of which favour paths with low cost — we note that from a game-playing perspective, the shortest path can often appear more realistic to an observer in taking a direct route. In environments containing obstacles, diffusion finds results comparable to A* in terms of cost and in a time-scale appropriate for real-time path finding. In addition, diffusion lends itself to use with multiple agents, opening up the possibility of emergent behaviours occurring, thus further enhancing the game-playing experience from a player perspective.

Currently, the evironments in which the algorithms have been evaluated are rather contrived. Future work will evaluate the performance of the parallelized algorithms within real game environments, using open-source APIs. Similarly it would be interesting to look at each of the algorithms in multi-agent scenarios, particularly diffusion as we have shown that the number of agents should have no effect on the time taken to find a path due to the nature of the algorithm yet the work carried out in [4] shows it can lead to a number of interesting emergent behaviours.

Acknowledgement

We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Tesla K40 GPU used for this research.

References

1. Ortega-Arranz H, Torres Y, Llanos DR, Gonzalez-Escribano A. A new gpu-based approach to the shortest path problem. *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, IEEE, 2013; 505–511.
2. Johnson T, Rankin J. Parallel agent systems on a gpu for use with simulations and games. *Latest Advances in Information Science and Applications* 2012; .
3. Han TD, Abdelrahman TS. Reducing branch divergence in gpu programs. *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, ACM: New York, NY, USA, 2011; 3:1–3:8.
4. Repenning A. Collaborative diffusion: Programming antiobjects. *Technical Report*, University of Colorado 2006.
5. Hart P, Nilsson N, Raphael B. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transaction On* July 1968; **4**(2):100–107.
6. Dijkstra E. A note on two problems in connexion with graphs. *Numerische Mathematik* 1959; :269–271.
7. Caggianese G, Erra U. Exploiting gpus for multi-agent path planning on grid maps. *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, IEEE, 2012; 482–488.
8. Sanders J, Kandrot E. *Cuda by Example*. Addison Wesley, 2010.
9. NVidia. Cuda toolkit 2013. URL https://developer.nvidia.com/cuda-toolkit, accessed: 17th October 2013.
10. NVidia. Nvidia's next generation cuda compute architecture kepler gk110. *Technical Report*, Nvidia 2013. URL http://www.nvidia.co.uk/object/nvidia-kepler-uk.html.