**Implementing Dijkstra’s algorithm to solve 3D mazes on the GPU**

By: Blake Bandy

***Abstract*- Finding the shortest path between two points is something that is used in many applications in today’s world and is an important tool to have. Our goal is to implement Dijkstra’s algorithm on a GPU to speed up the calculation in solving the shortest path between two points in a 3D maze.**

**Keywords—Dijkstra; GPU; shortest path; 3D maze**

**I. INTRODUCTION**

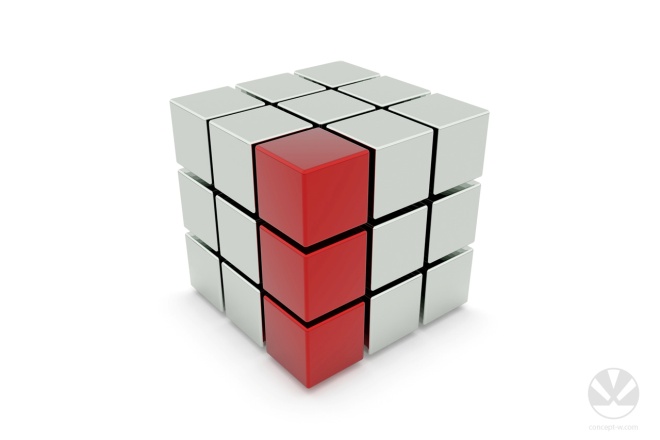
Finding the shortest path between two points in a graph is a vital tool to have in today’s world. It is used in many applications such as car navigation systems [1], traffic simulations [1], spatial databases [1], Internet route planners [1], and graph traversal. The algorithm that we used to solve our 3D maze was Dijkstra’s. This algorithm is a greedy algorithm with a sequential asymptotic time complexity of O(VlogV + E) where V represents the vertices of the graph and E represents the edges of the graph.

This algorithm works by computing the shortest path between all points in the graph. What it does is given a starting point, it looks for the smallest weighted path to take and after finding this, assigns a new weight to the vertex that is next in the shortest path. This algorithm is called greedy because it implements this method for every point of the graph. It is important to note that four our goal, we do not need the shortest weighted path that spans the whole graph because we want to find the shortest path between a source vertex and a target vertex.

This is accomplished by using Dijkstras to find the shortest weighted spanning tree for all points in the graph and then backtracking from the exit vertex to the start/source vertex. This backtracked path is always the shortest path from source to exit due to the nature of the algorithm. In our implementation, we apply a single threaded version of our 3D maze solver in C and then transfer that idea into CUDA to apply this algorithm on the GPU with the goal of achieving a speedup of at least 3-5x the CPU implementation.

**II. The Maze!**

Our implementation of the 3D maze in C is a single threaded version of Dijkstras algorithm that creates a maze with coordinates x, y, and z. These coordinates are input by the user and represent the length, width, and height of the maze. Also, when building this graph, we gave the maze a guaranteed path to the end. The guaranteed path ensures that the maze will have a path to the exit and is non-optimal. This path is starts at the top left corner, coordinate 0,0,0, and ends at coordinate N,N,N. More details described in the image below.



N,N,N

0,0,0

The arrows show the path initialized for the maze. The remaining cubes have randomly opened north, south, east, west walls and floors. This degree of random walls/floors is controlled by user input. If 3 is entered, for example, %30 of walls/floors will be opened. Once the maze is build, we were able to convert it into a graph in order to use a graph searching algorithm. To model the maze as a graph, we made each edge represent an open wall or floor and each cube represent a vertex.

Finally, the last number in the input for the program controls the number of threads to use.

**\***%openWalls should be between [0,10]. 0 = no extra open walls. 10 = 100% open walls.

**Example input: x y z %openWalls threads**

**Example input: 3 20 50 5 512**

**III. CPU Version**

The standard dijkstras algorithm is used for the cpu version. The data structures used are two structs. One is a struct with the definition of vertex, and the other is node. The vertex struct contains attributes such as int \_3dCoordinate[3] which represents the xyz position. Some of the other ones are adjEdges, adjIndex, index, visited, dist, and prev.

The node struct contains data properties Vertex v, node \*next, node \*prev. These data structures are used to build a doubly linked list that represents the list of vertices in the graph. Here is the pseudo code that we used in our c implementation.

***1 function Dijkstra(Graph, source):***

***2***

***3 dist[source] ← 0 // Distance from source to source***

***4 prev[source] ← undefined // Previous node in optimal path initialization***

***5***

***6 for each vertex v in Graph: // Initialization***

***7 if v ≠ source: // Where v has not yet been removed from Q (unvisited nodes)***

***8 dist[v] ← infinity // Unknown distance function from source to v***

***9 prev[v] ← undefined // Previous node in optimal path from source***

***10 end if***

***11 add v to Q // All nodes initially in Q (unvisited nodes)***

***12 end for***

***13***

***14 while Q is not empty:***

***15 u ← vertex in Q with min dist[u] // Source node in first case***

***16 remove u from Q***

***17***

***18 for each neighbor v of u: // where v is still in Q.***

***19 alt ← dist[u] + length(u, v)***

***20 if alt < dist[v]: // A shorter path to v has been found***

***21 dist[v] ← alt***

***22 prev[v] ← u***

***23 end if***

***24 end for***

***25 end while***

***26***

***27 return dist[], prev[]***

***28***

***29 end function***

[2]

We based our code on what is shown above but with obvious changes due to our data structures. As said above, we use this code to find the shortest weighted path between all vertices, and then backtrack from exit to source vertex.

**IV. The CUDA implementation**

Our code takes the same idea from the C code and implements this using CUDA. At first, we had some trouble trying to apply parallel computing to this algorithm. Our trouble came from the way the algorithm itself works. For the c version, all vertices are iterated through and compared to all its adjacent vertices. As this happens, weights are updated, meaning that there should be a dependency on previous calculations for this algorithm. This dependency makes it difficult to make the algorithm run in parallel.

After looking more into it, we realized that we could use parallelization to discover adjacent vertices. Then, we would use threads mapped to vertices to sequentially compare the current thread/point to all other adjacent points. This is done in parallel. So, the trick to implementing this algorithm in CUDA is to combine sequential code with CUDA. In order to implement this in a fast parallelizable way, we needed to change the data structures a bit. We removed the structure from the cpu version and replaced it with the following arrays.

**Edge array, E:** holds a list of every edge in the graph.

**Start array, S:** holds a list of all starting index positions in the E array for each vertex.

**Degree array, D:** holds the number of adjacent nodes to a vertex[i].

**Update array, U:** holds an updated value weight value for each vertex.

**Cost array, C:** holds calculated cost of a each vertex.

**Mask array, M:** Used to determine whether a vertex may be explored.

S:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 3 | 5 |  |  |  |  |  |

E:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 4 | 6 | 8 | 11 | 15 | 18 | 20 |

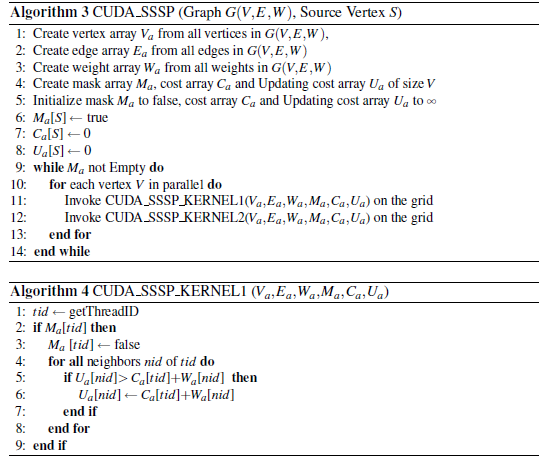
D:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 2 | 3 |  |  |  |  |  |

The above example shows how the S, E and D arrays are used in the gpu. Index 0 in the start array holds the start position for vertex 0. D is similar, but holds the number of vertices that are adjacent to vertex 0. With this data we can see that vertex 0’s adjacent vertices start at element 0 in the E array and continue for 3 elements. So, vertex 1, 4 and 6 are adjacent to vertex 0.

These various arrays are used by starting at the starting vertex, and in parallel, looking for all adjacent vertices to that specific point and updating the weights of all found adjacent vertices. Once a node is found, its cost is updated and mask is set to 1 so that on the next iteration it’s adjacent vertices can be updated. The cycle is then repeated until all vertices have been found and explored. The key to making this work in parallel is by using the M array. This array represents all vertices that have been discovered, but not explored.

The indexes of this array represent the vertex number. Initially, all values in the array are set to 0 with the first index set to 1 because that is our start index. When a vertex is discovered, its index in the M array is updated from 0 to 1 meaning it was found and needs to be explored. This is why in our main function, we have a while loop that goes until a value called mSum is 0. This value represents the sum of all values in the M array. We use this because if this value is greater than 0, that means we still have points that have been discovered, but not explored. We then update the cost arrays to reflect the current cost of each discovered vertex. The pseudo code our implementation Is based on is below.



[2].

**V. Modifications**

We followed the pseudo code to a degree, but were able to make some improvements and devised some methods in order to make it work properly.

Firstly, the M array. On the cpu side, a while loop controls all kernel calls and determines when the loop should finish. The pseudo code says while M not empty do. To handle this we designed a kernel function to add all elements of M and return the value to the variable mSum. Using this we are able to quickly determine when the while loop should exit instead of using a costly cpu search.

Another modification we made was the to the number of kernel calls. We removed the second call from the pseudo code and added it to our first kernel with an extra syncthreads() call for a minor speed up.

Finally, we removed the U\_d[i] = C\_d[i]; line from the second kernel call in the pseudo code. We found that it was unnecessary and does not affect the end result.

By nature our algorithm uses coalesced memory. Any vertex with mask = 1 are able to access their required data from GPU memory at the same time depending on their thread id.

**VI. Data Analysis**

After we got our implementation running, we decided to plot the CPU version against the GPU version to see the difference in speed between the two. There are two graphs that will be shown below. The first graph shows tests that we ran using a various inputs for our program. The input for our program takes 4 values, the x, y, z coordinates and the random value, this number should be between [0, 10]. Using a controlled test, we kept the x and y coordinate at an input of 3 for both for all of our tests. The only coordinate we modified was the z coordinate. The random input we used was 4 and was kept at 4 for all tests. The second graph shows a vertical axis that represents time like the first graph, but the horizontal axis represents the amount of edges. Both of these graphs are CPU vs GPU.

**VII. Conclusion**

From the data we have gathered, it is we have found that using the GPU gave us a very good speedup. At vertices of 1620 and below, the CPU was the better performer. Once vertices are 1800 or more, the GPU became the winner in all cases, especially once the amount of points climbed to the thousands.

This GPU version is being tested against a single threaded version linear search Dijkstra’s algorithm. It is clear that a multithreaded version of Dijkstra’s algorithm or a perfectly optimized CPU version that would have been a more fair comparison, but may run into problems that the GPU version doesn’t have. The main problem for our CPU version is the linear searching, which is quite slow. The GPU doesn’t have this searching shortfall and once the input is large enough the GPU version will greatly outperform our CPU version. This is a test that we may pursue to see how the GPU really stacks up against a CPU when the code is more optimized for both pieces of hardware.

**VIII. References**

**[1]** ***Hector Ortega-Arranz, Yuri Torres, Diego R. Llanos, and Arturo Gonzalez-Escribano, “A new GPU-based Approach to the Shortest Path Problem”, [Online]. Available:*** [***http://www.infor.uva.es/~diego/docs/ortega13b.pdf***](http://www.infor.uva.es/~diego/docs/ortega13b.pdf)

**[2] *Dijkstra’s algorithm Pseudocode,[Online]. Available:*** [***https://en.wikipedia.org/wiki/Dijkstra%27s\_algorithm***](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm)

**[3] Pawan Harish and P.J. Narayanan, “*Accelerating large graph algorithms on the GPU using CUDA”, [Online]. Available:*** [***http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.102.4206&rep=rep1&type=pdf***](http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.102.4206&rep=rep1&type=pdf)