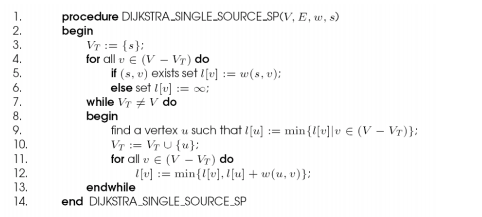
**Assignment 2: Parallel Programming – CUDA & OpenMP**

**ICOM4036 – Programming Languages**

**Jose F. Martinez Rivera**

**Prof. Wilson Rivera Gallego**

1. **Project Description:**

****

**Fig. 1** – Dijkstra’s Algorithm

Implement the parallel form of Dijktra’s Algorithm (Figure 1) in CUDA and OpenMP. After implementing it test CUDA’s parallelization vs OpenMP’s Parallelization.

1. **Implementation**

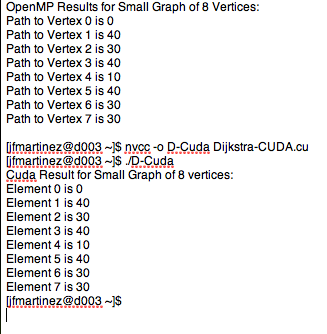
Observing the algorithm and analyzing it one can see that it is a hard algorithm to implement. The first for loop finds the initial paths from the node ‘s’ to all its neighbors. Once we acquire the initial paths we head to the while loop. This loop is in charge of finding the shortest path. It finds a vertex that matches the minimum cost path and it later marks it as visited. The inner for loop then checks the other unvisited vertices and if the weight path from u to v is lower than the current saved path it is updated.

Parallelization:

The first for loop can be parallelized, by giving each processor/thread a vertex v and the root, and each process finds if there’s a path from the root to its own assigned vertex. The way to parallelize the algorithm is by working on the contents of the while loop not the while loop itself. If we parallelize the inside of the while loop we assign each thread a number of vertices divided evenly (by the number of threads) and each thread will be in charge of finding the minimum of it’s own corresponding vertex and replace the cost path of each vertex. After that in the inner for loop it updates the cost of each vertex by the newest found shortest path.

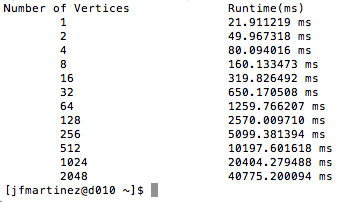
1. **Results**

First Test with a Graph of 8 Vertices and 11 Edges:



Both implementations output the same results.

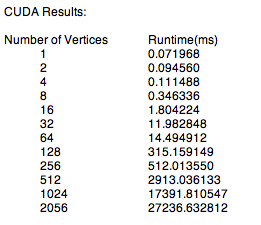
OpenMP Results:

****

**Fig 2. –** OpenMP Running time for N number of vertices.

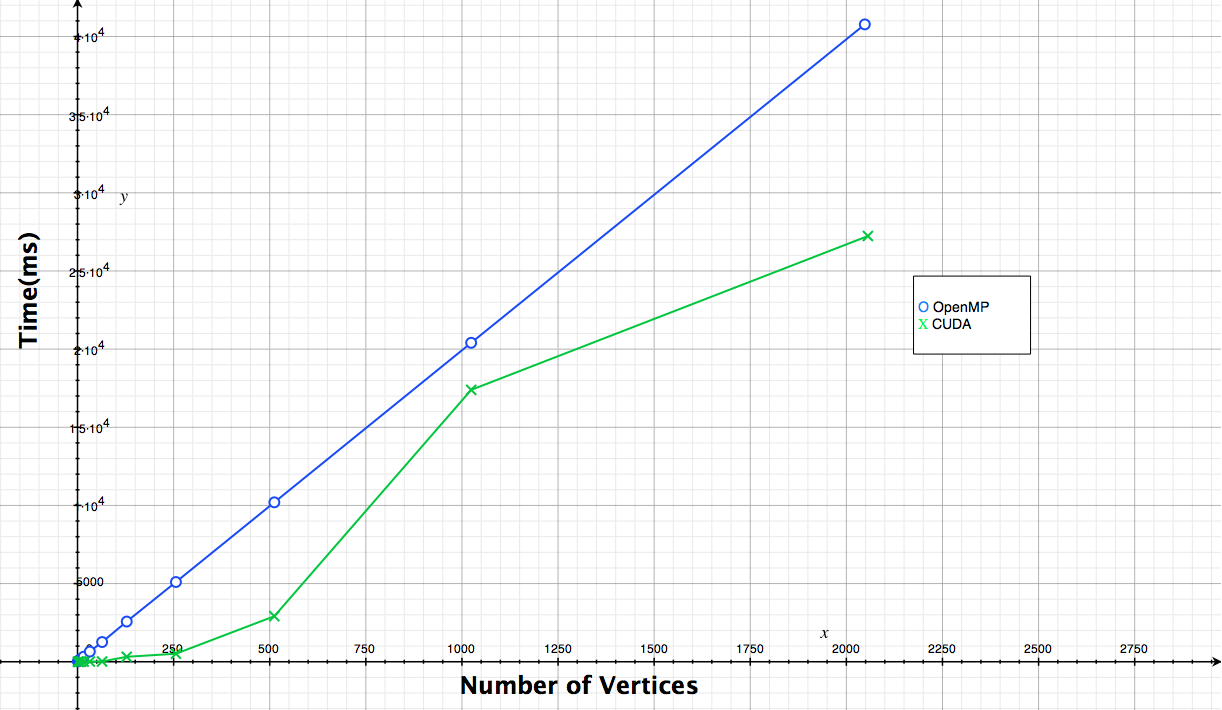
OpenMP’s running time is doubled every time the number of vertices is double. For example for N = 16, t = 319.82 ms vs. N = 32, t= 650.17 ms (close to doubling each value). So there’s a linear relationship between the number of vertices and runtime.

**Cuda Results:**



**Fig. 3 –** Cuda Running time for N Vertices

While running CUDA there isn’t really an easily visible relationship between the number of vertices and the running time. The sudden increases in time could be given due to the fact of the CUDA implementation because there is no dynamic increase of blocks, in other words all the processes are ran inside one block and multiple threads.



**Fig. 4 –** OpenMP vs. CUDA Running Time

Observing Fig. 3 it is obvious that the CUDA implementation runs faster than the OpenMP implementation.

1. **Conclusion**

At least from the implementations made, I can conclude that CUDA is certainly faster than Open MP when parallelizing algorithms. Another note is that this algorithm that I implemented is not as efficient as it could be which means there could be more efficiency or perhaps at some points OpenMP can be faster than CUDA if properly implemented. One thing is certain when parallelizing algorithms it should be for algorithms who have large amounts of data if not a sequential algorithm is more efficient at running small data algorithms.