University of Colorado Boulder

ECEN-5593
Advanced Computer Architecture

Spring 2017

Project Report

Implementation and Analysis of Dijkstra's and Floyd-Warshall's Algorithm: OpenMP and CUDA

Submitted By:

Anirudh Tiwari
Anirudh.Tiwari@colorado.edu

Vishal Vishnani
Vishal.Vishnani@colorado.edu

Index

Introduction

Applications with large computational requirements are the reason for the development of parallel machines. A single compute resource can do only one thing at a given time. Multiple resources can do things simultaneously and provide concurrency. Modern computers and laptops run on parallel architectures with multiple processors and cores. Effective utilization of these machines though requires a depth understanding of their working and the capabilities they incorporate within themselves. For parallel computing, one should be able to understand the hardware and software environments that he or she works in. Parallel software is specifically intended for hardware architectures with multiple cores and threads. A large amount of potential computing power is wasted when serial programs run on these modern, multi-core machines.

As parallel computing is considered as the high end of computing and is being used in modelling the difficult problems in every area of computing, keeping in mind our access to hardware resource of NVIDIA Jetson TX1, we are implementing Dijkstra's Algorithm, i.e. the shortest path algorithm using multiple nodes. For software, we are using *C*, *OpenMp* and *CUDA*. The algorithm is implemented in software and the performance over the three platforms is then analyzed and subsequently, conclusions are drawn.

OpenMP (Open Multi-Processing) is an API (Application Programming Interface). It supports shared platform multi-platform multiprocessing programming in C, CPP and Fortran. It uses a portable, scalable model. It helps provide programmers a simple and flexible interface across a vast range of platforms for developing parallel applications. It is a multi-threading implementation. It is a method of parallelizing in which a master thread forks a given number of slave threads. The system then divides the tasks among them.

CUDA is a platform for parallel computing and a mode for programming by NVIDIA. It increases the computer performance exceptionally by exploiting the capabilities of the Graphical Processing Unit (GPU). CUDA helps in enabling Scattered reads, in which, the code can have read access at arbitrary addresses. Also, CUDA offers a shared memory fast region which can be shared among different threads. It also helps faster memory options to and from GPU.

Dijkstra's Algorithm

Explanation from www.wikipedia.com

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.
The algorithm exists in many variants; Dijkstra's original variant found the shortest path between two nodes, but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree.
For a given source node in the graph, the algorithm finds the shortest path between that node and every other. It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined. For example, if the nodes of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. Hence, the shortest path algorithm is widely used in network routing protocols, most notably IS-IS and Open Shortest Path First (OSPF). It is also employed as a subroutine in other algorithms such as Johnson's.

Floyd-Warshall's Algorithm

Explanation from www.wikipedia.com

Floyd-Warshall is an algorithm for finding shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles). A single execution of the algorithm will find the lengths (summed weights) of the shortest paths between *all* pairs of vertices. Although it does not return details of the paths themselves, it is possible to reconstruct the paths with simple modifications to the algorithm. Versions of the algorithm can also be used for finding the transitive closure of a relation R, or (about the Schulze voting system) widest paths between all pairs of vertices in a weighted graph.

Dijkstra's Algorithm

Implementation

CPU (Serial)

- The serial code for this algorithm was designed to serve a baseline to OpenMP and CUDA implementation and is not meant to be efficient.
- It makes use of two functions in which the first function Shortest_Distance_Node() iterates through the node_shortest_distance() array and finds the next closest node and returns it.

```
/*This function finds the next closest node and returns it*/
int32_t Shortest_Distance_Node(float* Node_Shortest_Dist, uint32_t* Completed_Node){
    uint32_t node_distance=INF_DIST;
    int32_t node=-1;
    uint32_t i;
    for(i=0;i<VERTICES;i++){
        if((Node_Shortest_Dist[i]<node_distance) && (Completed_Node[i]==0)){
            node_distance=Node_Shortest_Dist[i];
            node=i;
        }
    }
    return node;
}
```

Fig1. Pseudocode to find the closest node serially

- The second function checks for the total distance, the distance of the current node from the source plus the distance of the current node to the neighbor. If this distance is lower than the neighbor's current distance, it updates this distance for neighbor's distance to indicate the lower value. It then sets the current node as the precursor of the neighbor and updates that in the parent_node.

```
for(j=0;j<VERTICES;j++){
    uint32_t new_distance=Node_Shortest_Dist[current_node] + Graph[current_node*VERTICES + j];
    if ((Completed_Node[j] != 1) && (Graph[current_node*VERTICES + j] != (float)(0)) && (new_distance < Node_Shortest_Dist[j])){
        Node_Shortest_Dist[j] = new_distance;
        Parent_Node[j] = current_node;
    }
}
```

Fig2. Pseudocode to find the shortest distance serially

OpenMP

- This version is based on the serial code to increase the performance by dividing the code into parallel parts.
- In the first function, the OpenMP implementation parallelizes the serial code by breaking down the for loop between several threads. Each thread finds its closest node and in the end, #pragma omp critical is used so that only one thread will access the region at a given time and we compare the closest node of all the threads and ultimately find the closest node amongst them.

```
#pragma omp parallel private(smallest_dist_thread, closest_node_thread) shared(Node_Shortest_Dist, Completed_Node)
{
    smallest_dist_thread = node_distance;
    closest_node_thread = node;
    #pragma omp barrier       //barrier is used so as to wait for all threads to arrive and start with same data

    #pragma omp for nowait   //nowait is used to remove barrier for "for loop"
    for (i = 0; i < VERTICES; i++) {
        if ((Node_Shortest_Dist[i] < smallest_dist_thread) && (Completed_Node[i] == 0)) {
            smallest_dist_thread = Node_Shortest_Dist[i];
            closest_node_thread = i;
        }
    }
    #pragma omp critical     //omp critical is used so that only one thread can enter this region at a time
    {
        if (smallest_dist_thread < node_distance) {
            node_distance = smallest_dist_thread;
            node = closest_node_thread;
        }
    }
}
```

Fig3. Pseudocode to find the closest node (OpenMP)

- In the second function, each thread will calculate its own total distance, which is the distance of the current node from the source plus the distance of the current node to the neighbor where each thread will read a different neighbor. If this distance is lower than the neighbor's current distance, it will update this distance in the *shared* node_Shortest_Distance array. It then sets the current node as the precursor of the neighbor and updates that in the parent_node.

```
#pragma omp parallel shared(Graph,Node_Shortest_Dist) //shared variables
{
    #pragma omp for private(new_distance,j) //private variables for each thread
    for (j = 0; j < VERTICES; j++) {
        new_distance = Node_Shortest_Dist[current_node] + Graph[current_node*VERTICES + j];
        if ((Completed_Node[j] != 1) && (Graph[current_node*VERTICES + j] != (float)(0)) && (new_distance < Node_Shortest_Dist[j])){
            Node_Shortest_Dist[j] = new_distance;
            Parent_Node[j] = current_node;
        }
    }
    #pragma omp barrier //barrier is used so as to wait for all threads to arrive
}
```

Fig4. Pseudocode to find the shortest distance (OpenMP)

CUDA

- The first function is the same as the serial function to find the closest node. We experimented with a few methods to find the next closest node using multiple threads in parallel, but were unsuccessful due to the race condition between the threads. These threads try to modify the same variables, namely node_distance and node. So, we tried to parallelize the loops by finding the smallest distance for each block and then find the smallest distance among all blocks. But, this still would give incorrect results. Another way we experimented was by launching kernel with one block of one thread, but the overhead for doing this continuously in a for loop was much higher as compared to the serial implementation.
- In the second function, each thread will calculate its own total distance, which is the distance of the current node from the source plus the distance of the current node to the neighbor where each thread will read a different neighbor. If this distance is lower than the neighbor's current distance, it will update this distance in the *shared* node_Shortest_Distance array. It then sets the current node as the precursor of the neighbor and updates that in the parent_node.

```
/*This function calculates the shortest path from source node to all other nodes in parallel*/
__global__ void Shortest_Path_Computation_CUDA(float* Graph, float* Node_Shortest_Dist, int* Parent_Node, int* Completed_Node, int* closest_node){

    Completed_Node[closest_node[0]]=1;
    int tid=blockIdx.x*blockDim.x+threadIdx.x;

    if(tid>VERTICES)
        return;

    int current_node=closest_node[0];
    int new_distance;

    new_distance = Node_Shortest_Dist[current_node] + Graph[current_node*VERTICES + tid];

    if ((Completed_Node[tid] != 1) && (Graph[current_node*VERTICES + tid] != (float)(0)) && (new_distance < Node_Shortest_Dist[tid])){
        //each thread get different j & new_distance
        Node_Shortest_Dist[tid] = new_distance;
        Parent_Node[tid] = current_node;
    }
}
```

Fig5. Pseudocode to find the shortest distance (CUDA)

Results, Observations and Analysis

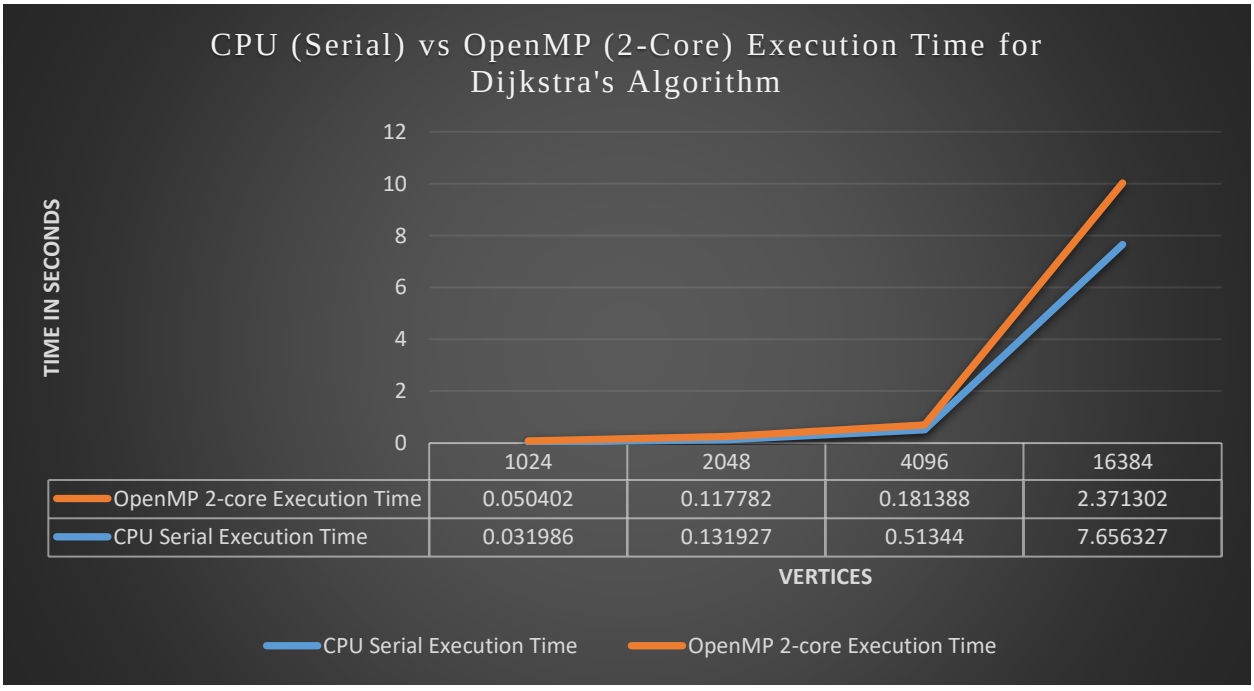
1.1 CPU serial execution time vs OpenMP 2-core execution time – Comparison and Analysis

Vertices	CPU Serial Execution Time (seconds)	OpenMP 2-core Execution Time (seconds)	Speedup
1024	0.031986	0.050402	0.634618
2048	0.131927	0.117782	1.120095
4096	0.51344	0.181388	2.830617
16384	7.656327	2.371302	3.228744

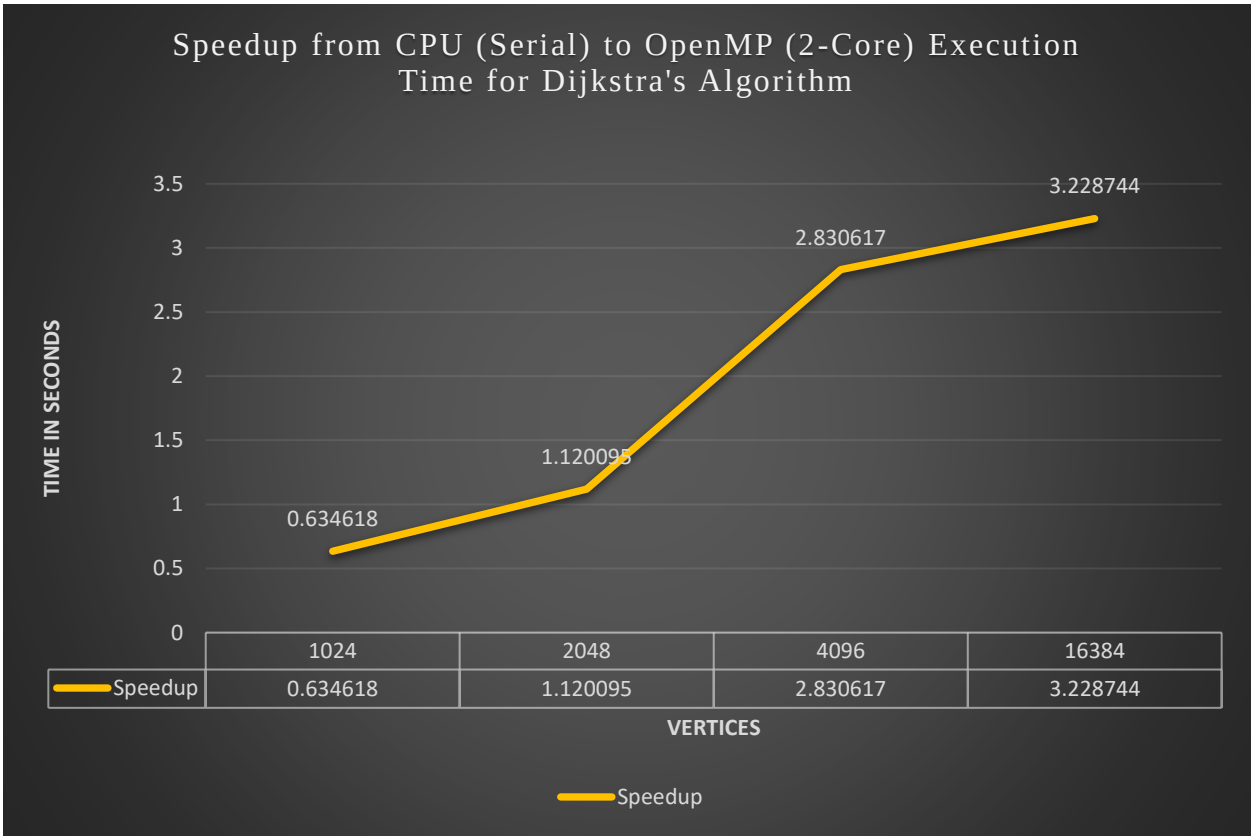
TABLE 1: CPU (Serial) vs OpenMP (2-core) Execution Time

From the table, we observe that for OpenMP 2core implementation of 1024 and 2048 vertices, the speedup is not much. The reason is that, setting-up of the OpenMP threads requires its own time which increases the overall execution time. As the number of vertices increases, the speedup is higher as can been for 4096 and 16384 vertices.

Following graph shows variation in the execution time for CPU and GPU as a function of the number of vertices.



Following graph shows variation in the speedup in execution from CPU to OpenMP as a function of the number of vertices.



1.2 CPU serial execution time vs OpenMP 4-core execution time – Comparison and Analysis

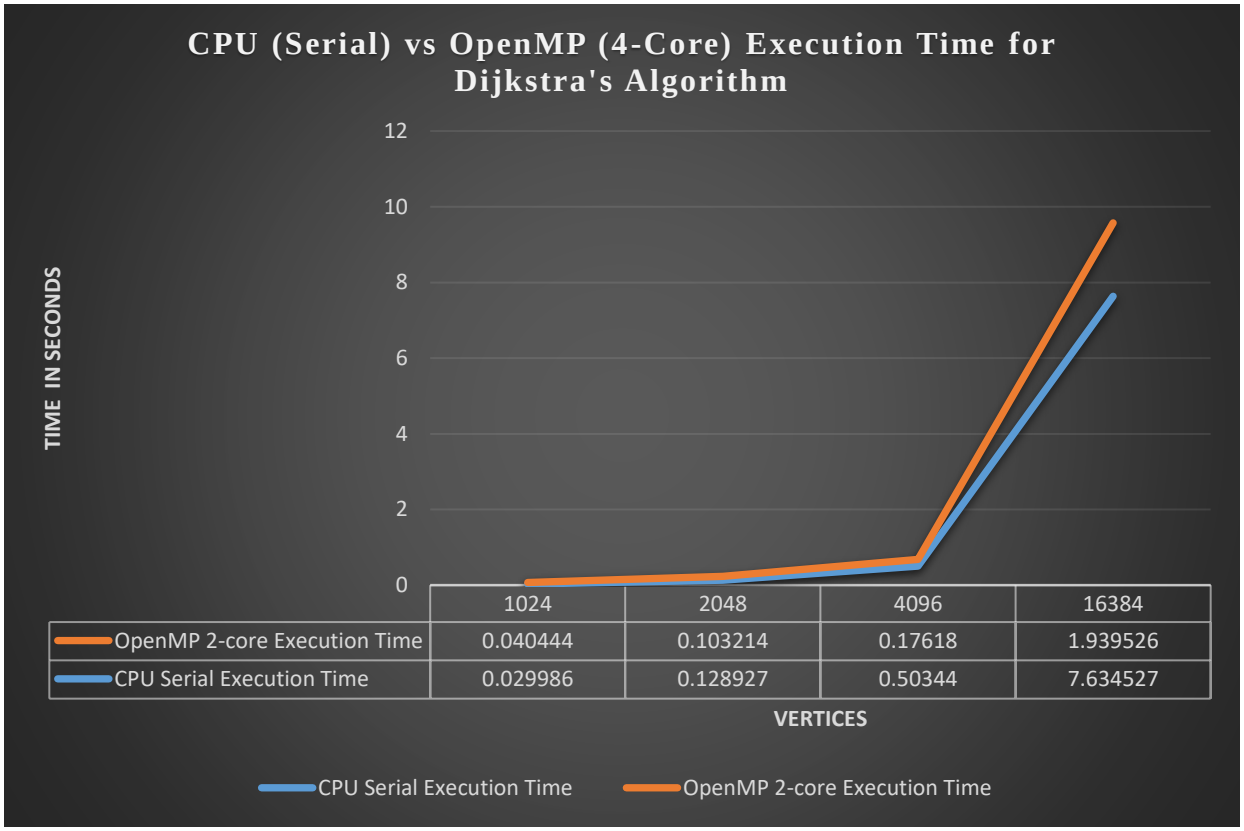
Vertices	CPU Serial Execution Time (seconds)	OpenMP 4-core Execution Time (seconds)	Speedup
1024	0.029986	0.040444	0.554844
2048	0.128927	0.103214	1.249123
4096	0.50344	0.176180	2.857532
16384	7.634527	1.939526	3.936285

TABLE 2: CPU (Serial) vs OpenMP (4-core) Execution Time

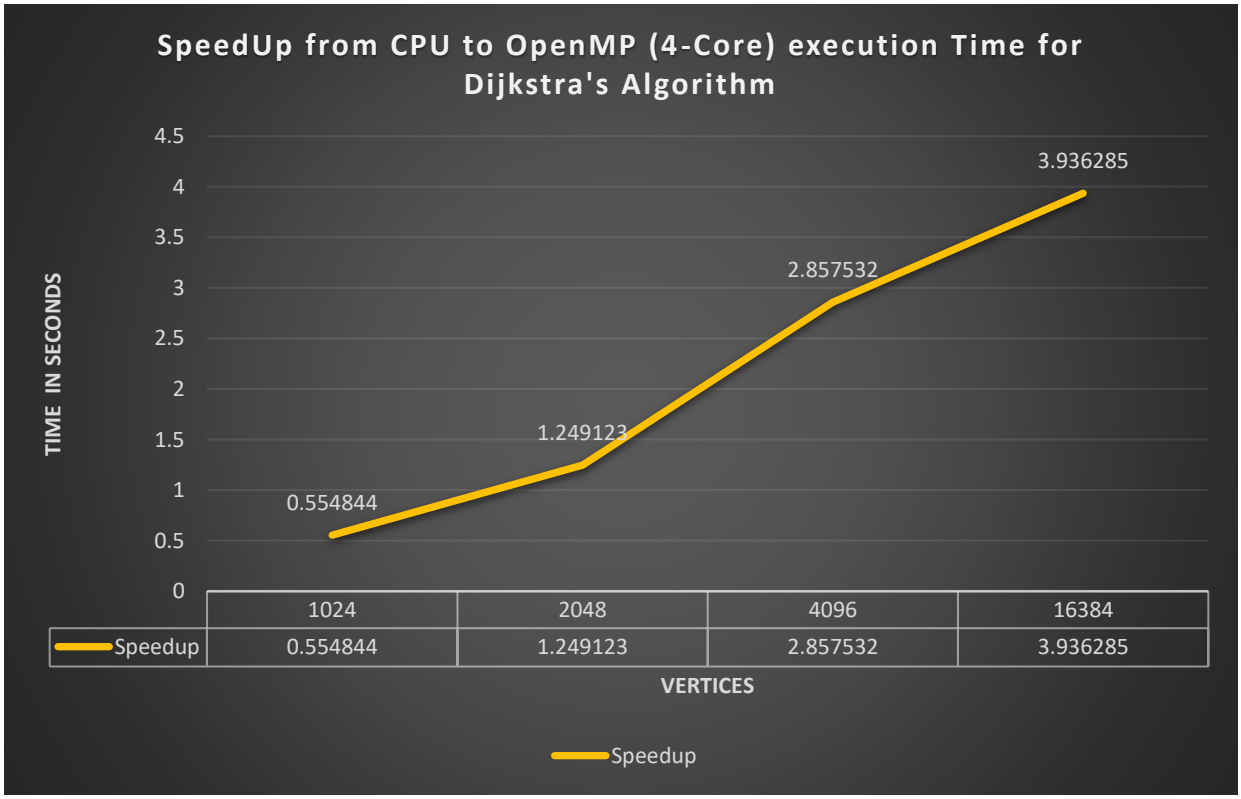
From the table, we observe that for OpenMP 4-core implementation of 1024 and 2048 vertices, the speedup is not much. The reason is that, setting-up of the OpenMP threads requires its own time which increases the overall execution time.

The speedup obtained for OpenMP 4-core is slightly greater than the previous 2-core implementation. The speedup increases to about 4 times for 16384 vertices.

Following graph shows variation in the execution time for CPU and GPU as a function of the number of vertices.



Following graph shows variation in the speedup in execution from CPU to OpenMP as a function of the number of vertices.



2. CPU serial execution time vs GPU execution time – Comparison and Analysis

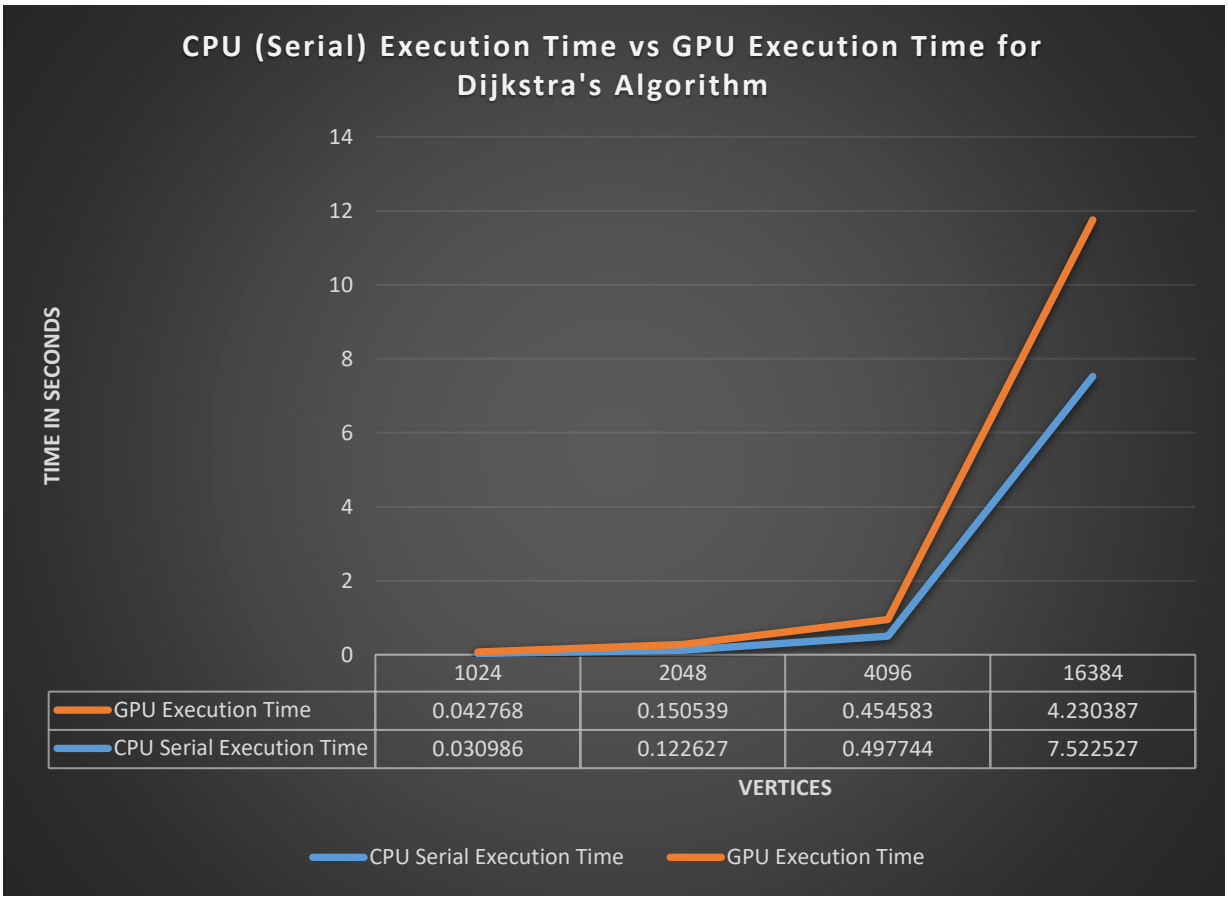
Block Size 256

Vertices	CPU Serial Execution Time (seconds)	GPU Execution Time(seconds)	Total Memory transfer Time	Speedup
1024	0.030986	0.042768	0.001857	0.724514
2048	0.122627	0.150539	0.005895	0.814586
4096	0.497744	0.454583	0.026014	1.094946
16384	7.522527	4.230387	0.343295	1.778212

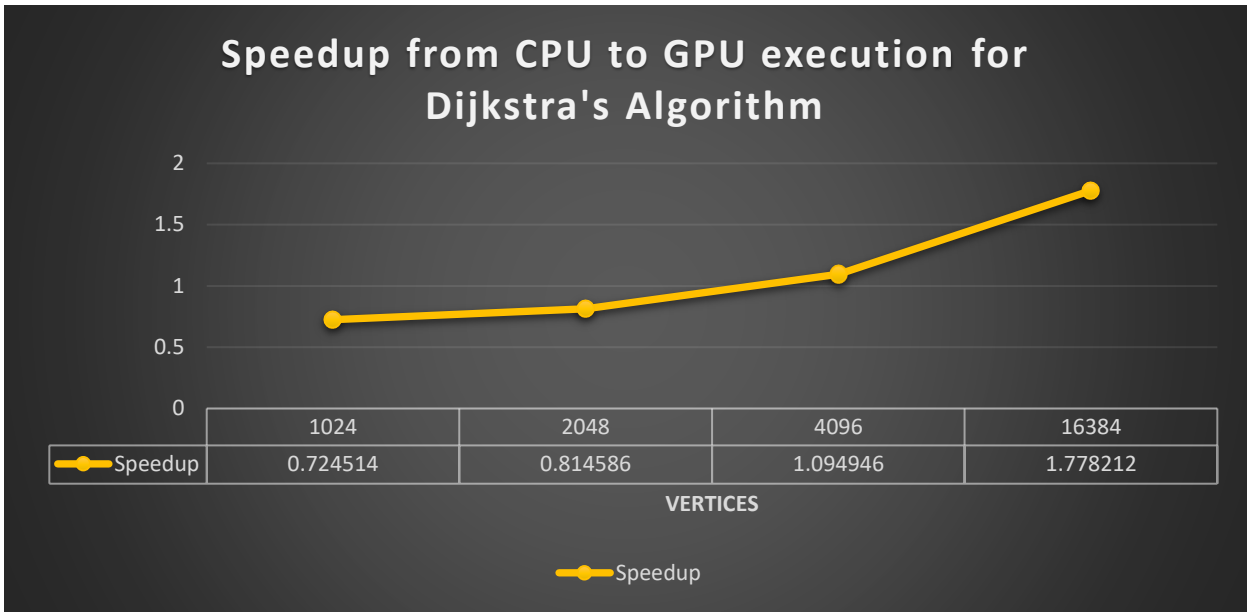
TABLE 3: CPU (Serial) vs GPU Execution Time for Block Size 256

From, the above table we can see that for 1024 and 2048 vertices, the GPU execution is slower than CPU execution. This is because for GPU computation, one function runs serially on CPU and another parallelly on GPU due to which, we need to transfer the shortest distance array from host to device and device to host repeatedly in a for loop. Thus, GPU takes longer time to execute.

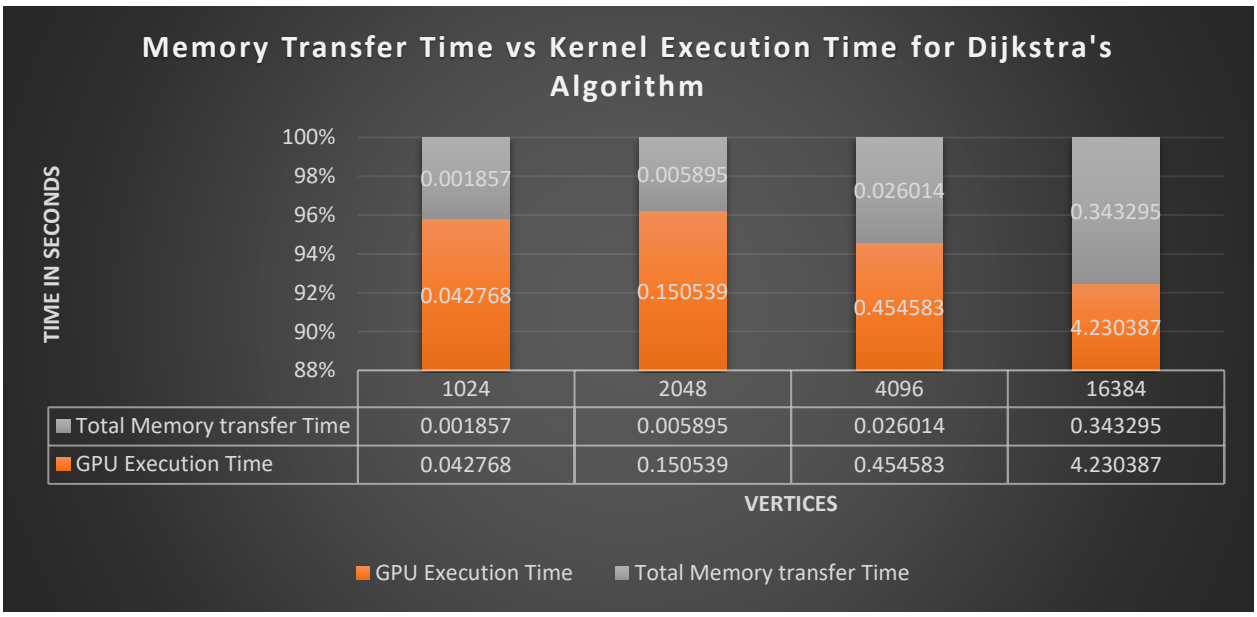
Following graph shows variation in the execution time for CPU and GPU as a function of the number of vertices.



Following graph shows variation in the speedup in execution from CPU to GPU as a function of the number of vertices.



Following graph shows percentage variation in the execution time for Memory Transfer Time and Kernel Execution as a function of the number of vertices.



From the graph, we can observe that the percentage contribution of the GPU execution time to the total execution time decreases while that of the total memory transfer time increases, as the number of vertices increase.

Floyd Warshall's Algorithm

Implementation

CPU (Serial)

- It is based on standard $O(N^3)$ algorithm which is three nested for-loops. Therefore, if the number of vertices are 1000, it will take 10^9 iterations of the loop to find the shortest path between all the nodes. This is the reason why the CPU performance is not very efficient and the performance degrades further if the number of vertices increase.

```
/*This function computes shortest distance between nodes serially*/
void Serial_Floyd(int *Graph1,int *Graph_Path){
    int x,y,z;
    for(x=0;x<VERTICES;++x){
        for(y=0;y<VERTICES;++y){
            for(z=0;z<VERTICES;++z){
                        int current_node=y*VERTICES+z;
                        int Node_i=y*VERTICES+x;
                        int Node_j=x*VERTICES+z;
                if(Graph1[current_node]>(Graph1[Node_i]+Graph1[Node_j])){
                    Graph1[current_node]=(Graph1[Node_i]+Graph1[Node_j]);
                    Graph_Path[current_node]=x;
                }
            }
        }
    }
}
```

Fig6. Pseudocode for Floyd's Algorithm (serial)

OpenMP

- In OpenMP, the serial function is parallelized by creating the variables of nested for-loops as private along with the current_node. Thus, each thread will execute in parallel and modify the same shared variable Graph1 and Graph_path for different current_node.

```
/*This function computes shortest distance between nodes parallely*/
void Parallel_Floyd_openMP(int *Graph1,int *Graph_Path){
    int x,y,z;
    int current_node,Node_i,Node_j;
    #pragma omp parallel shared(Graph1,Graph_Path)
    {
    #pragma omp for private(x,y,z,current_node,Node_i,Node_j)
    for(x=0;x<VERTICES;++x){
        for(y=0;y<VERTICES;++y){

                for(z=0;z<VERTICES;++z){

                    current_node=y*VERTICES+z;
                    Node_i=y*VERTICES+x;
                    Node_j=x*VERTICES+z;
                    if(Graph1[current_node]>(Graph1[Node_i]+Graph1[Node_j])){
                        Graph1[current_node]=(Graph1[Node_i]+Graph1[Node_j]);
                        Graph_Path[current_node]=x;
                    }
                }
            }

    }
    #pragma omp barrier
    }
}
```

Fig7. Pseudocode for Floyd's Algorithm (OpenMP)

CUDA

- For this implementation, each thread will calculate its own node_distance and total_distance, which is the distance of a particular node from the source plus the distance between that node and the neighbor, where each thread will read a different neighbor. If this distance is lower than the neighbor's current distance, it will update this distance in the cuda_Graph. It then sets that particular node as the precursor of the neighbor and updates that in the cuda_Path.

```
/*This function computes shortest distance between all nodes parallely*/
__global__ void CUDA_Kernel(int i,int* cuda_Graph, int* cuda_Path){

     int tid=threadIdx.x;
     int gid=blockIdx.x*blockDim.x +threadIdx.x;
     if(gid>=VERTICES){
         return;
     }

int idx=VERTICES*blockIdx.y + gid;
__shared__  int shortest_distance;

if(tid==0){
    shortest_distance=cuda_Graph[VERTICES*blockIdx.y+i];
}

__syncthreads();

    int node_distance=cuda_Graph[i*VERTICES+gid];

    int total_distance=shortest_distance+node_distance;

    if (cuda_Graph[idx]>total_distance){

   cuda_Graph[idx]=total_distance;
   cuda_Path[idx]=i;
   }

}
```

Fig8. Pseudocode for Floyd's Algorithm (CUDA)

Results, Observations and Analysis

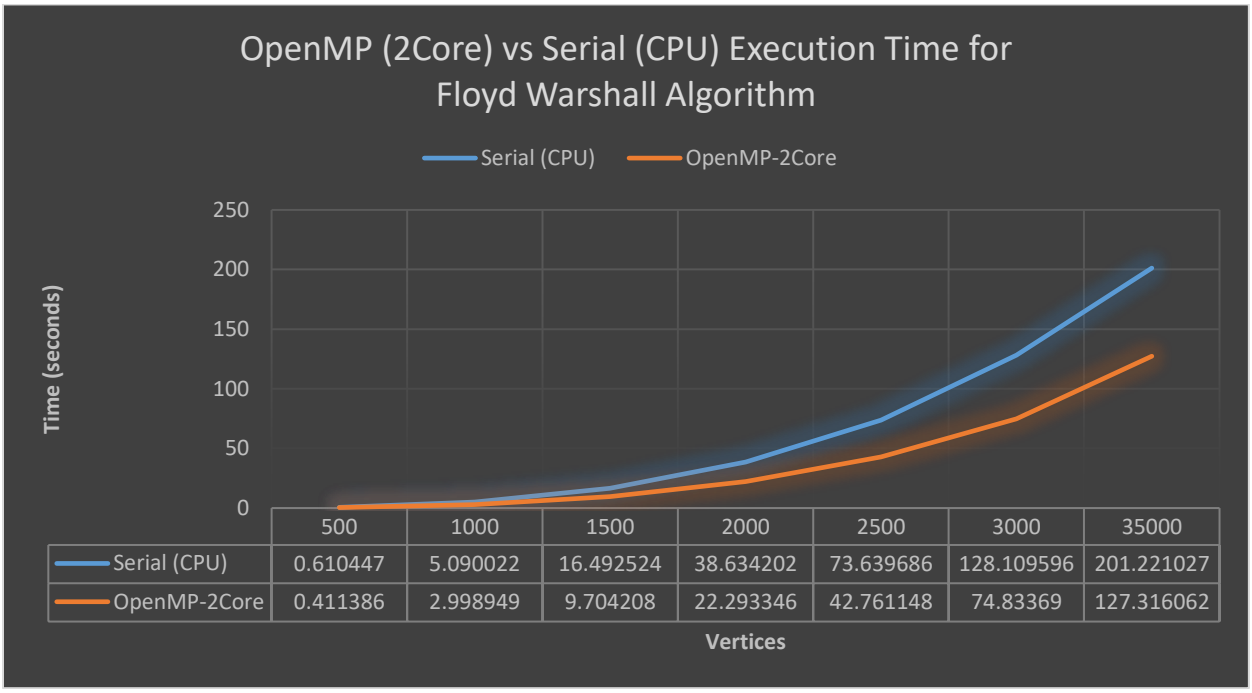
1.1 OpenMP execution time vs CPU serial execution time – Comparison and Analysis

Vertices	Serial (CPU) Execution Time	OpenMP-2Core Execution Time	Speedup
500	0.610447	0.411386	1.483879
1000	5.090022	2.998949	1.697269
1500	16.492524	9.704208	1.699523
2000	38.634202	22.293346	1.732993
2500	73.639686	42.761148	1.722117
3000	128.109596	74.833690	1.711924
3500	201.221027	127.316062	1.580484

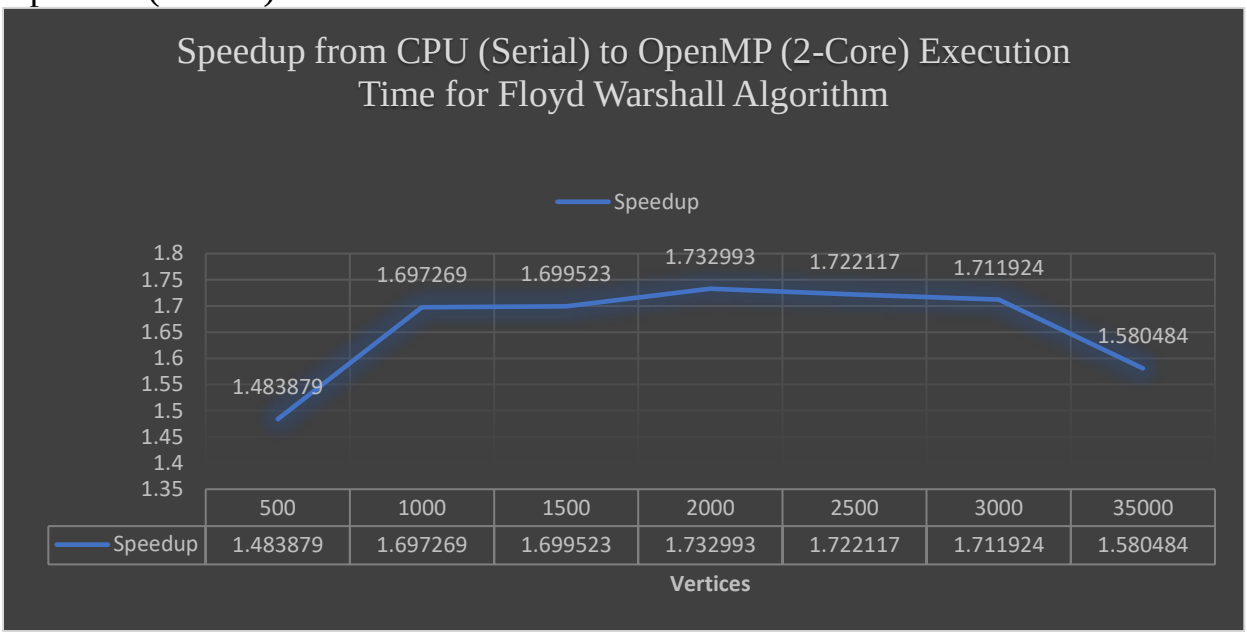
TABLE 4: CPU (Serial) vs OpenMP (2-Core) Execution Time

From the above table, we observe that for OpenMP 2-core implementation the speedup obtained is between 1.5 and 1.75 for vertices between 500 and 3500. We could not experiment with higher vertices values because of limited memory.

Following graph shows variation in the execution time for CPU and OpenMP as a function of the number of vertices.



Following graph shows variation in the speedup in execution from CPU to OpenMP (2-Core) as a function of the number of vertices.



1.2 OpenMP execution time vs CPU serial execution time – Comparison and Analysis

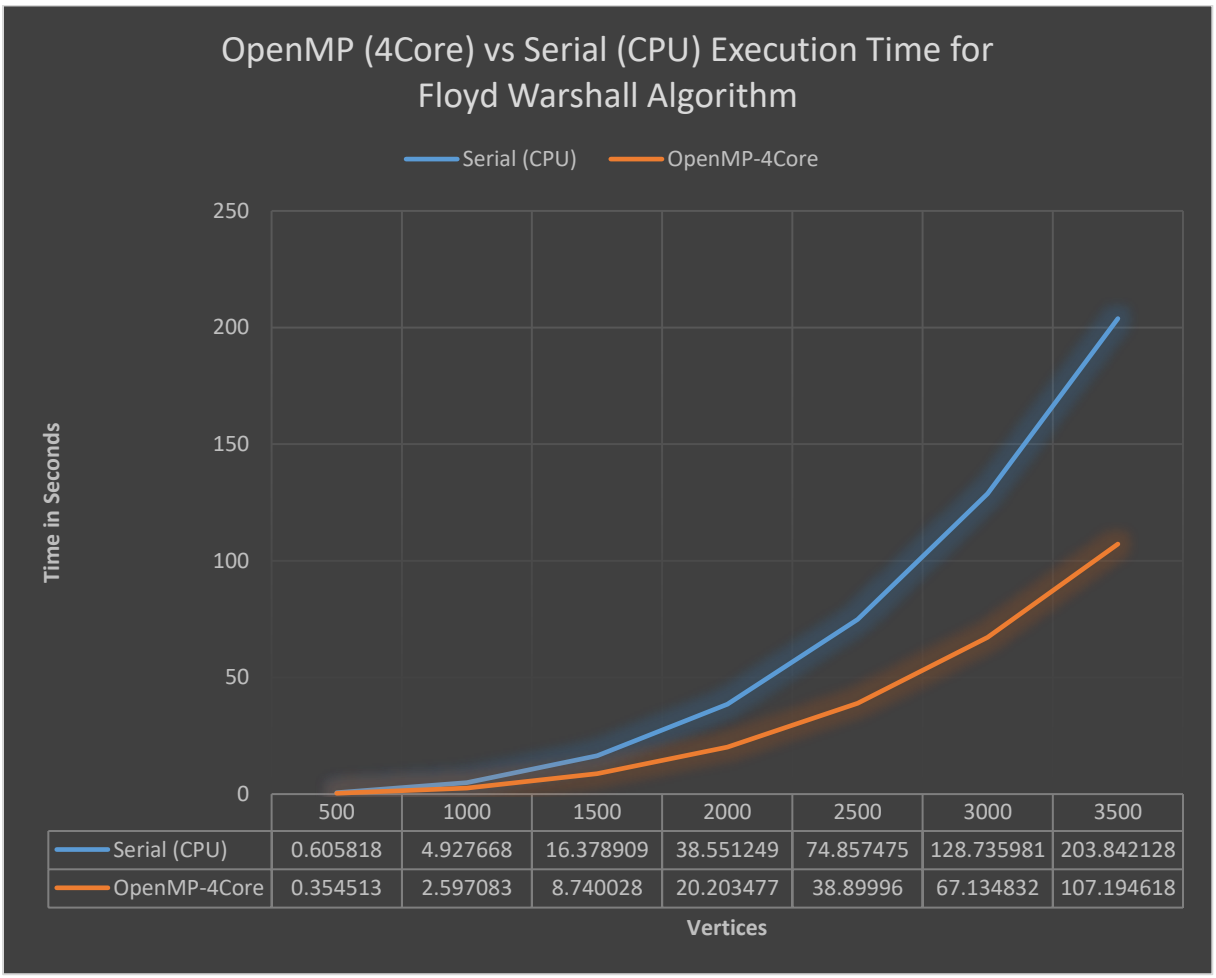
Vertices	Serial (CPU) Execution Time	OpenMP-4Core Execution Time	Speedup
500	0.605818	0.354513	1.708874
1000	4.927668	2.597083	1.897386
1500	16.378909	8.740028	1.874011
2000	38.551249	20.203477	1.908149
2500	74.857475	38.899960	1.924359
3000	128.735981	67.134832	1.917574
3500	203.842128	107.194618	1.901608

TABLE 5: CPU (Serial) vs OpenMP (4-Core) Execution Time

From the above table, we observe that for OpenMP 4-core implementation the speedup obtained is between 1.75 and 2.0 for vertices between 500 and 3500.

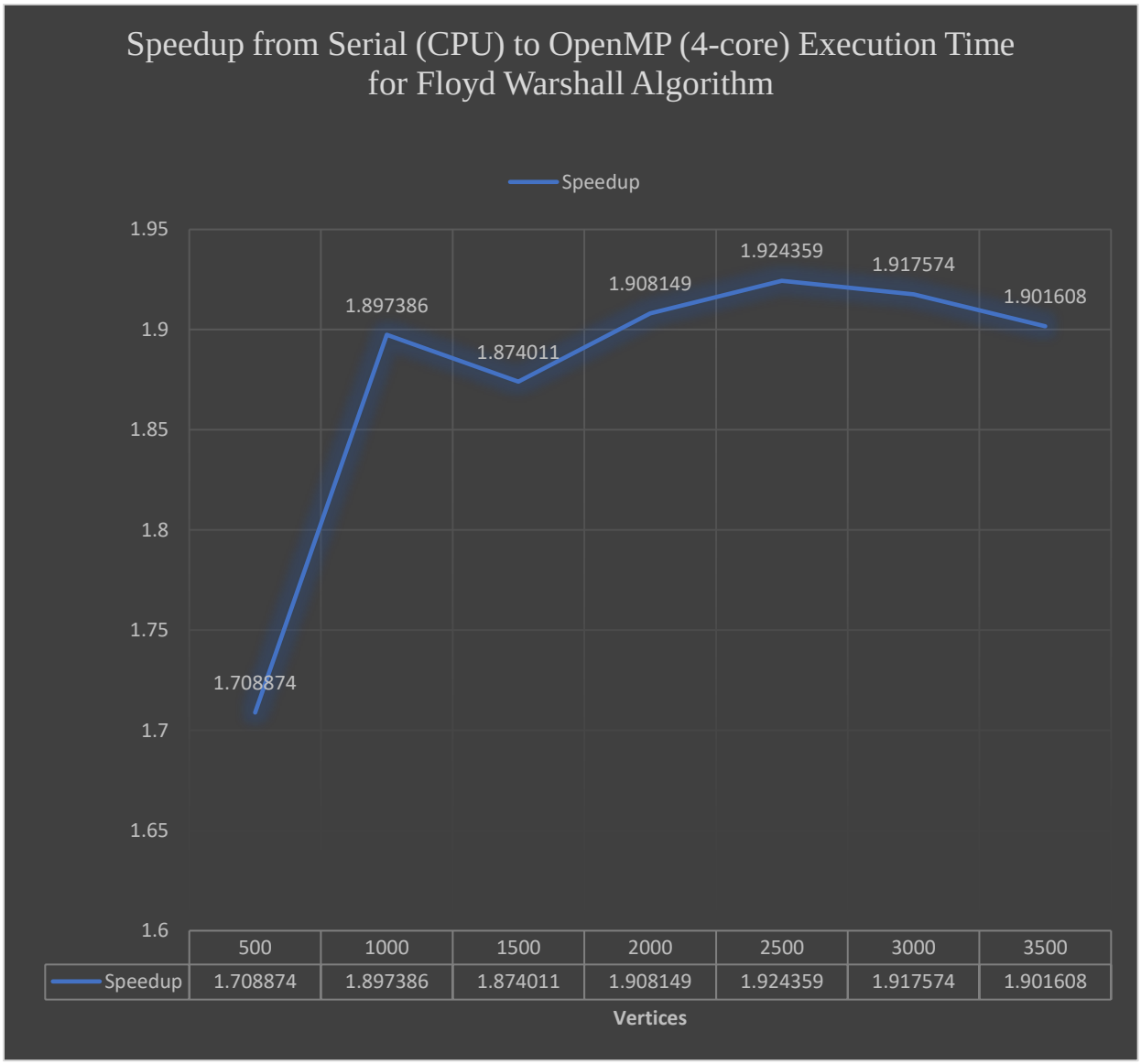
This speedup is slightly better than the speedup obtained for 2-core. We could not experiment with higher vertices values because of limited memory. As the speedup obtained is not significant, highly parallelized GPU computation is required.

Following graph shows variation in the execution time for CPU and OpenMP as a function of the number of vertices.



	500	1000	1500	2000	2500	3000	3500
Serial (CPU)	0.605818	4.927668	16.378909	38.551249	74.857475	128.735981	203.842128
OpenMP-4Core	0.354513	2.597083	8.740028	20.203477	38.89996	67.134832	107.194618

Following graph shows variation in the speedup in execution from CPU to OpenMP as a function of the number of vertices.



2. CPU serial execution time vs GPU execution time

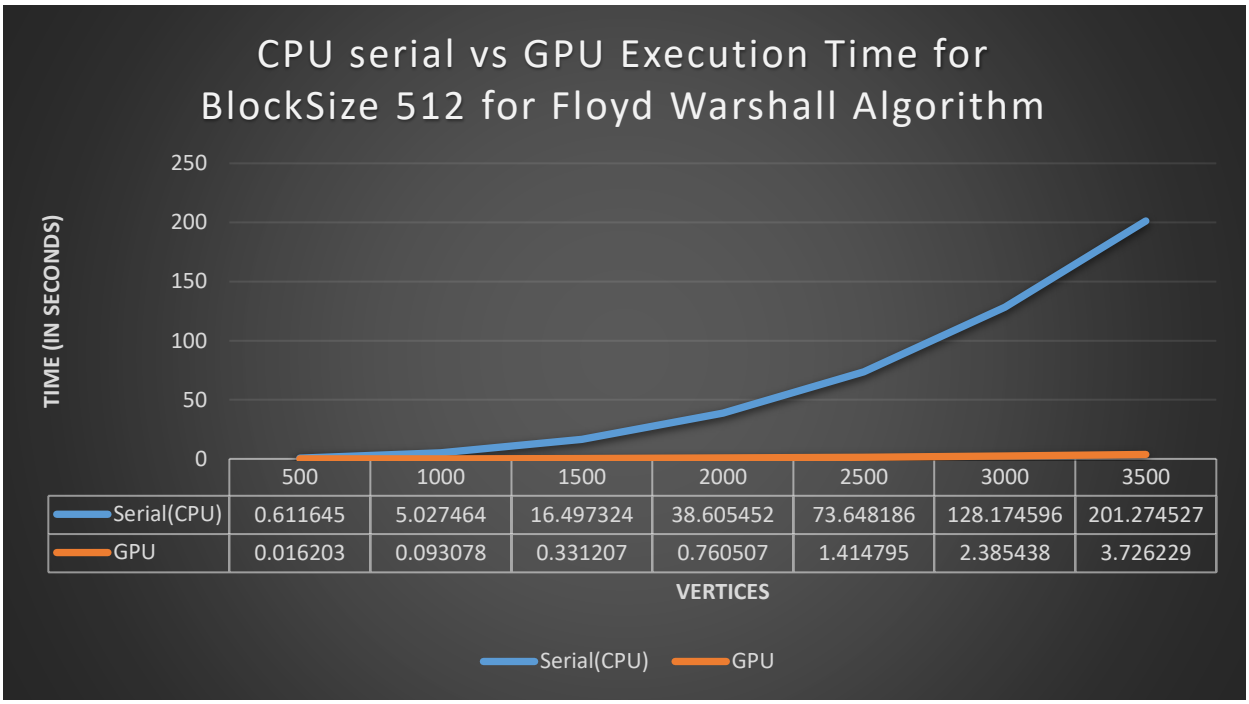
BlockSize-512

Vertices	Serial(CPU) Execution time	GPU Execution Time	Total Memory Transfer Time	Speedup
500	0.611645	0.016203	0.002166	37.7488737
1000	5.027464	0.093078	0.006513	54.0134511
1500	16.497324	0.331207	0.013096	49.8097081
2000	38.605452	0.760507	0.022698	50.7627832
2500	73.648186	1.414795	0.034217	52.0557296
3000	128.174596	2.385438	0.048702	53.7321012
3500	201.274527	3.726229	0.066674	54.0156085

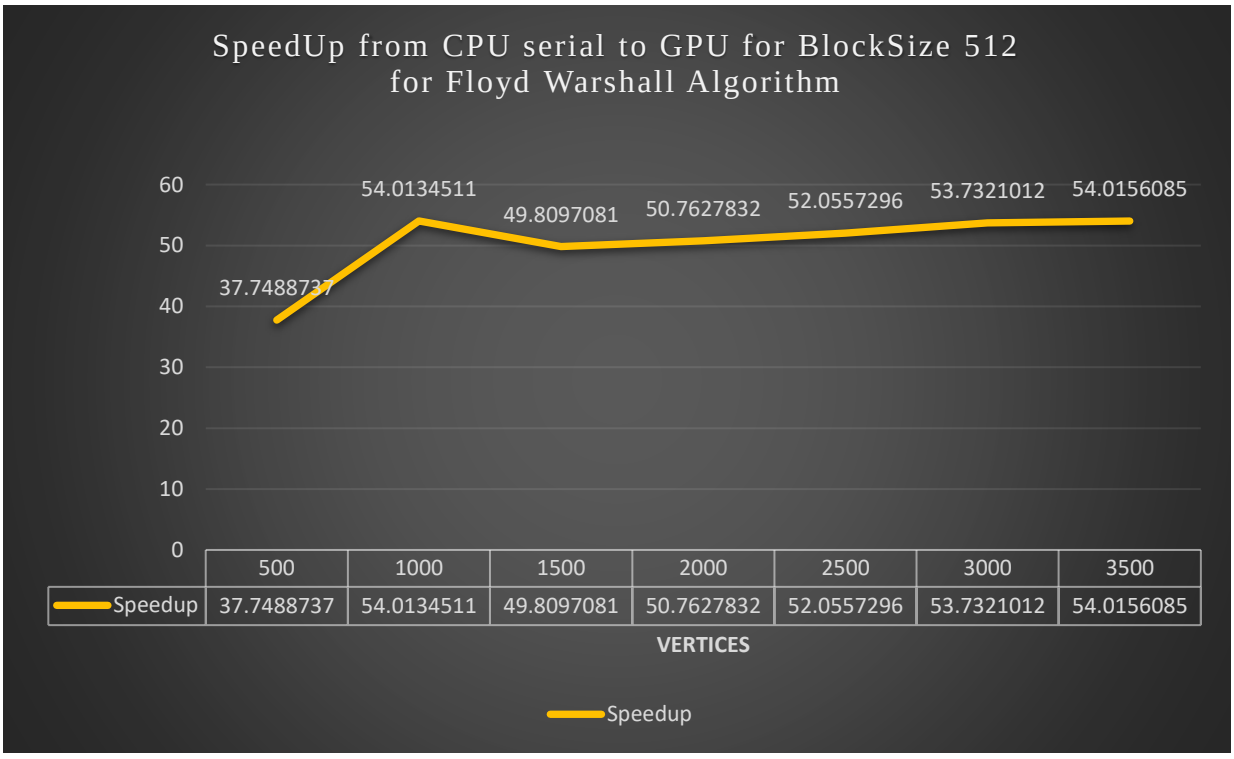
TABLE 6: CPU (Serial) vs GPU Execution Time for Block-Size 512

From the above table, we observe that the speedup from CPU to GPU execution for 512 threads per block, ranges from 40-50. This speedup is much higher than the speedup obtained in OpenMP (4-core) implementation. We could not experiment with higher vertices values because of limited memory.

Following graph shows variation in the execution time for CPU and GPU as a function of the number of vertices.

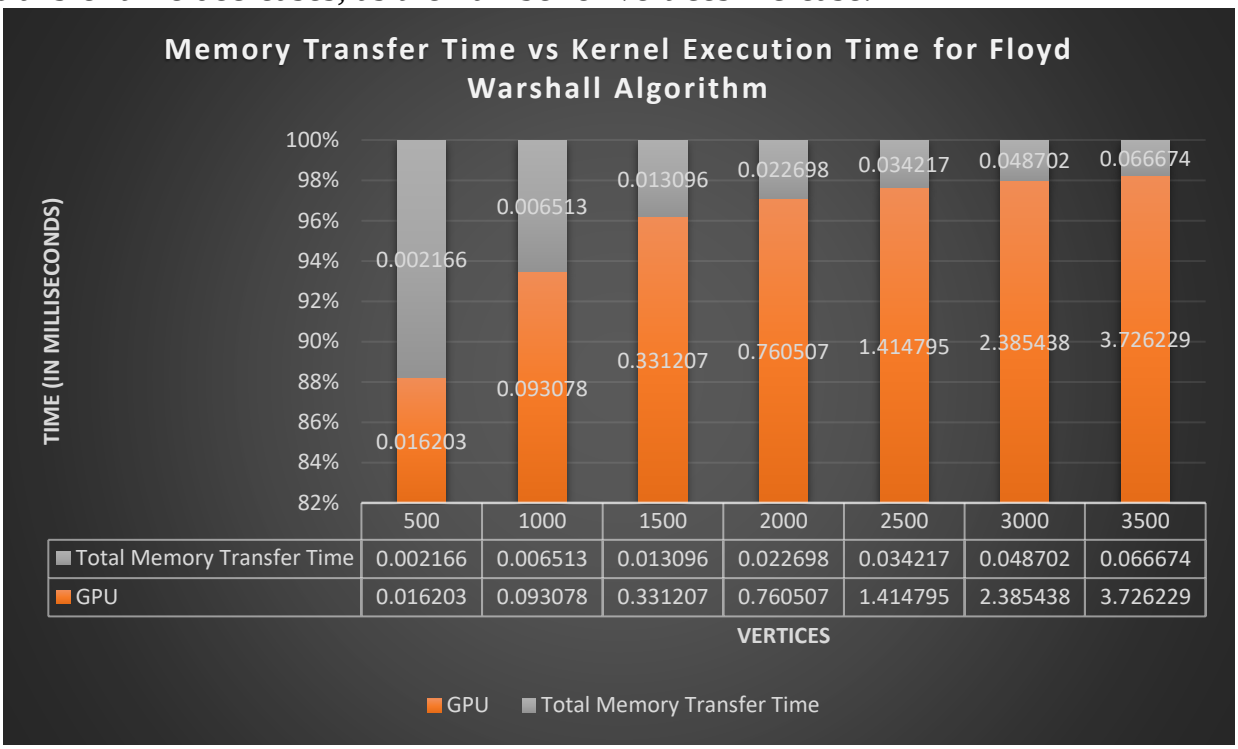


Following graph shows variation in the speedup in execution from CPU to GPU as a function of the number of vertices.



Following graph shows percentage variation in the execution time for Memory Transfer Time and Kernel Execution as a function of the number of vertices.

From the graph, we can observe that the percentage contribution of the GPU execution time to the total execution time increases while that of the total memory transfer time decreases, as the number of vertices increase.



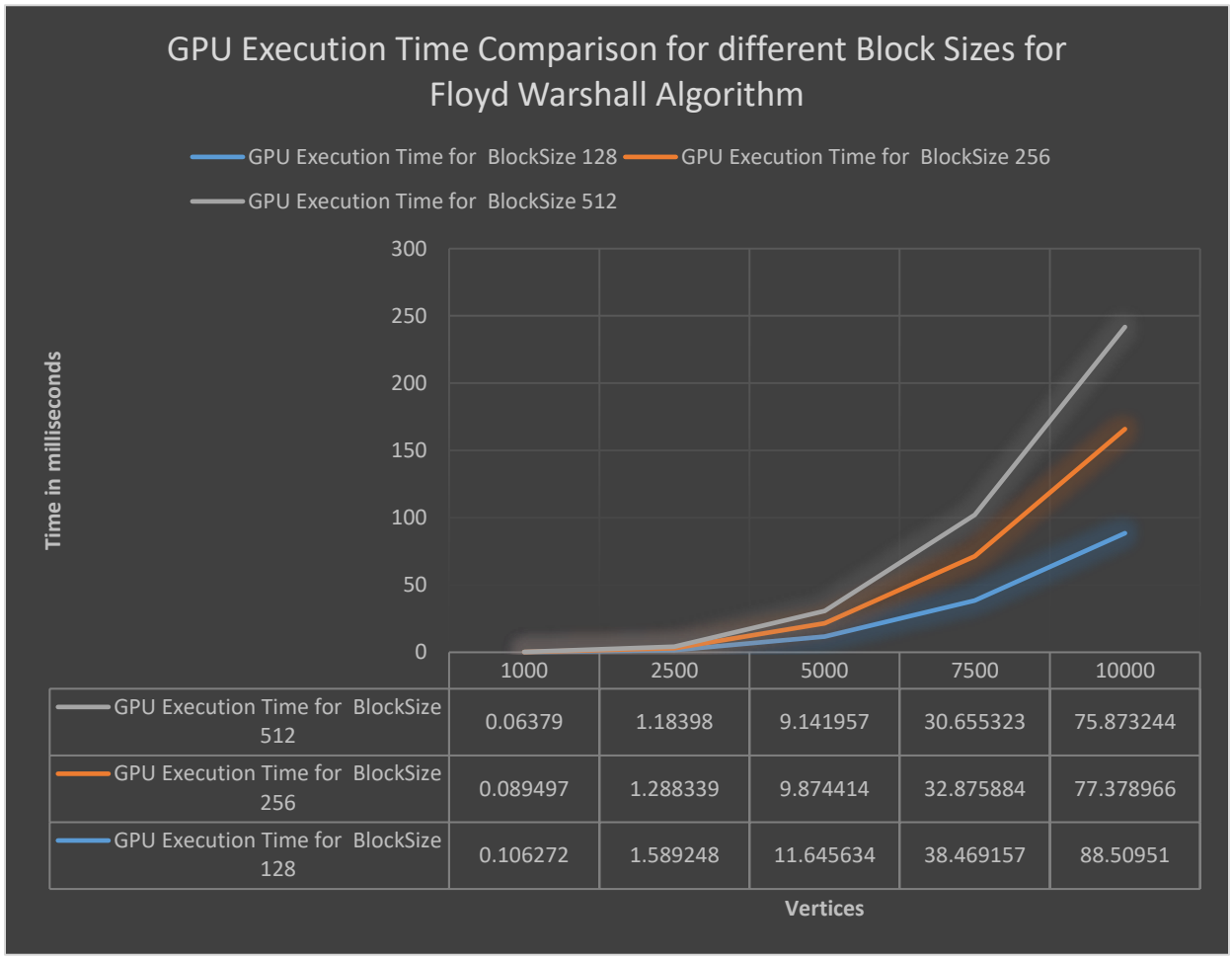
3. Comparison of GPU execution time for different threads per block (Block-Size)

Vertices	GPU Execution Time for BlockSize 128	GPU Execution Time for BlockSize 256	GPU Execution Time for BlockSize 512
1000	0.106272	0.089497	0.063790
2500	1.589248	1.288339	1.183980
5000	11.645634	9.874414	9.141957
7500	38.469157	32.875884	30.655323
10000	88.509510	77.378966	75.873244

TABLE: GPU Execution Time for various Block-Sizes

From the above table, we observe that as we increase the number of blocks from 128 to 512, there is a reduction in the execution time of vertices between 1000 and 10,000.

Following graph shows variation in the execution time for GPU for various Block Sizes as a function of the number of vertices.



	1000	2500	5000	7500	10000
GPU Execution Time for BlockSize 512	0.06379	1.18398	9.141957	30.655323	75.873244
GPU Execution Time for BlockSize 256	0.089497	1.288339	9.874414	32.875884	77.378966
GPU Execution Time for BlockSize 128	0.106272	1.589248	11.645634	38.469157	88.50951

Conclusion

A basic serial version of code was developed for the Dijkstra's algorithm to serve as a base for the codes developed in CUDA and OpenMP. The maximum speedup obtained using OpenMP was four-times as compared to that of the CPU for 16384 vertices. The GPU implementation should have been much faster, but using a serial function and repeated memory transfers increased the execution time. In the end, the maximum speedup obtained using GPU was twice as that of the CPU. So, OpenMP 4-core gave us a better result for this implementation. A fully parallel version of Dijkstra's Algorithm would have given far better results for GPU Computation.

Similarly, a basic serial version of code was developed for the Floyd Warshall's algorithm to serve as a base for the codes developed in CUDA and OpenMP. The maximum speedup obtained using OpenMP was two-times as compared to that of the CPU for 3500 vertices. The GPU implementation of fully parallel Floyd Warshall's Algorithm resulted in speedup of 55 times compared to the serial version of the implementation.

Thus we observe that, a significant increase in speedup is obtained for graph algorithms using CUDA.

References

1. https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
2. http://math.mit.edu/~rothvoss/18.304.3PM/Presentations/1-Melissa.pdf
3. https://developer.nvidia.com/cuda-zone
4. http://www.openmp.org/wp-content/uploads/ntu-vanderpas.pdf
5. https://en.wikipedia.org/wiki/OpenMP
6. https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm
7. http://math.mit.edu/~rothvoss/18.304.1PM/Presentations/1-Chandler-18.304lecture1.pdf
8. https://devtalk.nvidia.com/