CPSC6780 Final Project

Implementation of Floyd–Warshall Algorithm for the Shortest Path Problem Using CUDA

Instructor: Dr. Shuangshuang Jin

Team Members: Qiaoyi Yang, Siddhant Gupta

Abstract. In this project, we tried finding the shortest path between two vertices (or nodes) in a graph. Based on our research, algorithms such as the Floyd-Warshall algorithm and different variations of Dijkstra's algorithm are used to find solutions to the shortest path problem. In this project, we used the Floyd-Warshall algorithm because it is perfectly simple and understandable, and it finds the shortest path between all the pairs of vertices in a weighted graph. We implemented our algorithms in Nvidia's GPU using Compute Unified Device Architecture (CUDA). This paper presents a comparative analysis between CPU and GPU based implementation of the Floyd-Warshall algorithm. Our approach achieves a significant acceleration in comparison to the traditional serial algorithm.

Keywords: Shortest path problem, Floyd-Warshall algorithm, GPU, CUDA

1. Introduction

Finding the shortest path between two nodes is a fundamental problem in graph theory. Like the Bellman-Ford algorithm or Dijkstra's algorithm, the Floyd-Warshall algorithm computes the shortest path in a graph. However, Bellman-Ford and Dijkstra are both single-source, shortest-path algorithms. This means they only compute the shortest path from a single source [1]. Floyd-Warshall, on the other hand, computes the shortest distances between every pair of vertices in the input graph. That makes it could solve the problem with more possibilities.

Therefore, in this project, we select the Floyd-Warshall algorithm to find the shortest paths in a weighted graph. Then we implement it in CUDA to seep up the process. CUDA allows us to accelerate C or C++ applications by updating the computationally intensive portions of the code to run on Graphics Processing Units (GPUs). GPUs can supply a high parallel computation power. It has become very popular since the languages involved in their programming have evolved from graphics APIs to general purpose languages. One of the best examples is the CUDA API of NVIDIA. As a consequence of this evolution, the so-called General Purpose Computing on GPU (GPGPU) has consolidated as a very active research area, where many problems that are not directly related to computer graphics are solved using GPUs. The aim of all these GPU-based implementations is to achieve better running times than their CPU-based counterparts. In this project, we integrate the strength of both and present the solution using CUDA based on the Floyd-Warshall algorithm.

2. Floyd-Warshall Algorithm Overview

The Floyd-Warshall algorithm is a classic example of dynamic programming. It breaks the problem down into smaller subproblems, then combines the answers to those subproblems to solve the big, initial problem. The idea is this: either the quickest path from A to C is the quickest path found so far from A to C, or it's the quickest path from A to B plus the quickest path from B to C.

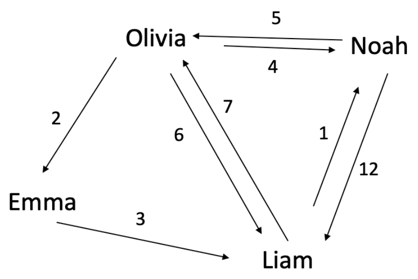

Fig. 1. Floyd-Warshall algorithm Example: 4 friends

Here is an example of how the Floyd-Warshall algorithm works. Imagine there are 4 friends (Fig.1): Olivia, Emma, Liam, and Noah. There are a few roads that connect some of their houses, and the lengths of those roads are known. But Floyd-Warshall can take what is known and provide the optimal route given that information. For example, look at the graph below. It shows paths from one friend to another with corresponding distances.

Floyd-Warshall will tell the optimal distance between each pair of friends. It will clearly tell that the quickest path from Olivia's house to Emma's house is the connecting edge that has a weight of 2. But it will also tell that the quickest way to get from Noah's house to Liam's house is to first go through Olivia's, then from Olivia's to Liam's. This is the power of Floyd-Warshall; no matter what house you're currently in, it will show the fastest way to get to every other house.

3. Floyd-Warshall Algorithm Implementations

First, create a distance table (Fig.2) that shows the distance between every friend. Then, based on the table, list all the paths when no intermediate is allowed, such as: {0,1}， {0,2}， {0,3}， {0, 4}, {1,0}， {1,1}， {1,2}， {1,3}…， {3, 0}， {3, 1}， {3, 2}， {3,3}. After that, calculate the distance for each path and put them into a matrix dis0 of dimension V*V where n is the number of vertices (Fig.3). The row and the column are indexed as i and j respectively. i and j are the vertices of the graph.

Each cell dis0[i][j] is filled with the distance from the ith vertex to the jth vertex. If there is no path from ith vertex to jth vertex, the cell is left as infinity.

	Olivia (0)	Emma (1)	Liam (2)	Noah (3)
Olivia (0)	0	2	6	4
Emma (1)	∞	0	3	∞
Liam (2)	7	∞	0	1
Noah (3)	5	∞	12	0

Fig. 2. Distance Table

$$\text{dis} = \begin{bmatrix} 0 & 2 & 6 & 4 \\ \infty & 0 & 3 & \infty \\ 7 & \infty & 0 & 1 \\ 5 & \infty & 12 & 0 \end{bmatrix}$$

Fig. 3. Matrix dis

For the next step, create a matrix dis0 using matrix dis. Let k be the intermediate vertex in the shortest path from source to destination. dis[i][j] is filled with (dis[i][k] + dis[k][j]) if (dis[i][j] > dis[i][k] + dis[k][j]) [3].

That is, if the direct distance from the source to the destination is greater than the path through the vertex k, then the cell is filled with dis[i][k] + dis[k][j].

In this step, k is vertex 0(Olivia). We calculate the distance from the source vertex to the destination vertex through this vertex k.

$$\text{dis} = \begin{bmatrix} 0 & 2 & 6 & 4 \\ \infty & 0 & 3 & \infty \\ 7 & \infty & 0 & 1 \\ 5 & \infty & 12 & 0 \end{bmatrix} \longrightarrow \text{dis0} = \begin{bmatrix} 0 & 2 & 6 & 4 \\ \infty & 0 & 3 & \infty \\ 7 & \mathbf{9} & 0 & 1 \\ 5 & \mathbf{7} & \mathbf{11} & 0 \end{bmatrix}$$

Fig. 4. Matrix dis0

In matrix dis, the direct distance from vertex 3 to 2 is 12. Once there is an intermediate k, the sum of the distance from vertex 3 to 2 would consist of the distance from vertex 3 to 0 and from vertex 0 to 2, which is [3, 0] + [0, 2] = 5 + 6 = 11. Since 12 > 11, dis1**[3, 2]** is filled with 11. It updated the shortest path from vertex 3 to 2. In the same way, it updated the shortest path for **[2, 1]** and **[3, 1].**

Similarly, matrix dis1(Fig. 5). is created using matrix dis0(Fig. 4). The elements in the second column and the second row are left as they are. In this step, k is the vertex 1(Emma). The remaining steps are the same as in the last step. The shortest path for **[0, 2]** and **[3, 2]** was updated.

$$\text{dis0} = \begin{bmatrix} 0 & 2 & 6 & 4 \\ \infty & 0 & 3 & \infty \\ 7 & 9 & 0 & 1 \\ 5 & 7 & 11 & 0 \end{bmatrix} \longrightarrow \text{dis1} = \begin{bmatrix} 0 & 2 & \mathbf{5} & 4 \\ \infty & 0 & 3 & \infty \\ 7 & 9 & 0 & 1 \\ 5 & 7 & \mathbf{10} & 0 \end{bmatrix}$$

Fig. 5. Matrix dis1

In the same way, have the k be the vertex 2(Liam), then vertex 3(Noah). Let it traverse all the paths so if there are any new paths shorter than the original path, and they will be updated. Matrix dis2(Fig. 6) and dis3(Fig. 7) are also created.

$$\text{dis1} = \begin{bmatrix} 0 & 2 & 5 & 4 \\ \infty & 0 & 3 & \infty \\ 7 & 9 & 0 & 1 \\ 5 & 7 & 10 & 0 \end{bmatrix} \longrightarrow \text{dis2} = \begin{bmatrix} 0 & 2 & 5 & 4 \\ \mathbf{10} & 0 & 3 & \mathbf{4} \\ 7 & 9 & 0 & 1 \\ 5 & 7 & 10 & 0 \end{bmatrix}$$

Fig. 6. Matrix dis2

$$\text{dis2} = \begin{bmatrix} 0 & 2 & 6 & 4 \\ 10 & 0 & 3 & 4 \\ 7 & 9 & 0 & 1 \\ 5 & 7 & 11 & 0 \end{bmatrix} \longrightarrow \text{dis3} = \begin{bmatrix} 0 & 2 & 5 & 4 \\ \mathbf{9} & 0 & 3 & \infty \\ \mathbf{6} & \mathbf{8} & 0 & 1 \\ 5 & 7 & 10 & 0 \end{bmatrix}$$

Fig. 7. Matrix dis3

The code for this part is:

```
InnerLoops(dis,k)
{
  for (int i = 0; i < |V|; i++)
  {
    for (int j = 0; j < |V|; j++)
    {
      if(dis[i][j]>dis[i][k] + dis[k][j])
        dis[i][j] = dis[i][k] + dis[k][j];
    }
  }
}
FloydWarshall(dis)
{
  for (int k = 0; k < |V|; k++)
  {
    InnerLoops(dis,k);
  }
}
```

This is a classic $O(|V|^3)$ implementation of the FW [4]. Here **dis[i][j]** contains the distance between vertex i and j. We can see from the pseudocode above that for a given value of k the entire **dis** Matrix (**|V|X|V|**) is updated. Reading and writing are happening in this matrix simultaneously, and the actual order of these operations does not matter. This makes it a great candidate for GPU computation. So we will run the inner two loops in the GPU. The inner two loops have $\mathbf{O(|V|^2)}$ complexity and running them on a GPU will yield $\mathbf{O(|V|^2/|P|)}$ complexity where $|P|$ is the number of threads. With a typical GPU, we will get $\mathbf{O(10^3)}$ threads. So when $\mathbf{|V|\sim O(10^3)}$ we will essentially have $\mathbf{O(|V|)}$ *c*omplexity for the inner loops and $O(|V|^2)$ overall complexity.

To make GPU has less branching to do. We have **t=dis[i][k] + dis[k][j]**, if the old **dis[i][j]** was 7, and the new path with k is 9. Then **t<dis[i][j]** is 1, **t>=dis[i][j]** is 0, the updated **dis[i][j]=t*(t<dis[i][j])+dis[i][j]*(t>=dis[i][j]).**

Now we have:

```
InnerLoops(dis,k)
{
  for (int i = 0; i < |V|; i++)
  {
    for (int j = 0; j < |V|; j++)
    {
      t=dis[i][k] + dis[k][j];
      dis[i][j]=t*(t<dis[i][j])+dis[i][j]*(t>=dis[i][j]);
    }
  }
}
FloydWarshall(dis)
{
  for (int k = 0; k < |V|; k++)
  {
    InnerLoops(dis,k);
  }
}
```

4. CUDA Implementations

The architecture of GPU best fits in the data parallel approach. As per analysis, GPU is most suited to algorithms with high arithmetic intensity and regular data access pattern29. Leading GPU developing company NVIDIA introduced Compute Unified Device Architecture to simplify GPU programming by utilizing a high-level application programming interface.

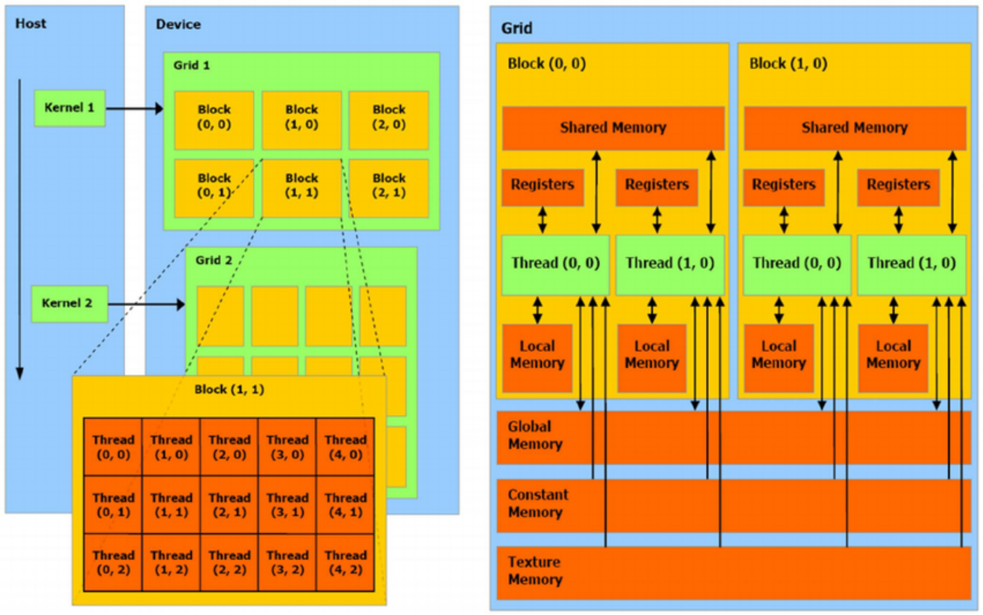

Fig.8. CUDA Programming Architecture

GPUs are collections of multiple streaming processors. In stream processing, a single instruction executes on a data stream in each thread. A CUDA program comprises a host (CPU) code that creates kernel calls and a device (GPU) code that implements the kernel30. The host code is a serial code that runs on the CPU, and the device code runs on GPU in each thread to maximize the GPU thread utilization. From the programmer's view, the CUDA programming model is a collection of threads running in parallel (Fig.8) [2]. Finally, we just convert these functions into CUDA. The code is as below:

```
__global__
void GPUInnerLoops(dis,k)
{
  //calculates unique thread ID in the block
  int t= (blockDim.x*blockDim.y)*threadIdx.z + (threadIdx.y*blockDim.x)+(threadIdx.x);

  //calculates unique block ID in the grid
  int b= (gridDim.x*gridDim.y)*blockIdx.z+(blockIdx.y*gridDim.x) + (blockIdx.x);

  //block size (this is redundant though)
  int T= blockDim.x*blockDim.y*blockDim.z;

  //grid size (this is redundant though)
  int B= gridDim.x*gridDim.y*gridDim.z;

  int t;
 for (int i=b; i<|V|; i+=B)
  {
    for(int j=t; j<|V|; j+=T)
    {
      t=dis[i*V+k]+dis[k*V+j];
      dis[i*V+j]=t*(t<dis[i*V+j])+dis[i*V+j]*(tm>= dis[i*V+j]);
    }
  }
}
FloydWarshall(dis)
{
  for (int k = 0; k < |V|; k++)
  {
    GPUInnerLoops<<<dim3(x,y,z),dim3(a,b,c)>>>(dis,k);
    cudaDeviceSynchronize();
  }
}
```

5. Test Results

We have tested the implementations on the school of computing's Palmetto using an Intel(R) Xeon(R) Gold 6148 2.40 GHz, and an NVIDIA Tesla V100-PCIE-16GB. We set different values of V (1000, 1500, 2000, 2500, 3000, 3500) to test the CPU vs. GPU speedup result (Fig.9). The results show a substantial acceleration of using CUDA on the Floyd-Warshall Algorithm compared to the traditional serial algorithm (Fig.10).

Vertex	CPU- Time (Seconds)	GPU- Time (Seconds)	Speedup(x)
1,000	5	0.014	363
1,500	19	0.050	379
2,000	45	0.110	409
2,500	87	0.204	426
3,000	151	0.340	444
3,500	239	0.551	434

Fig. 9 Results for Using CUDA on the Floyd-Warshall Algorithm

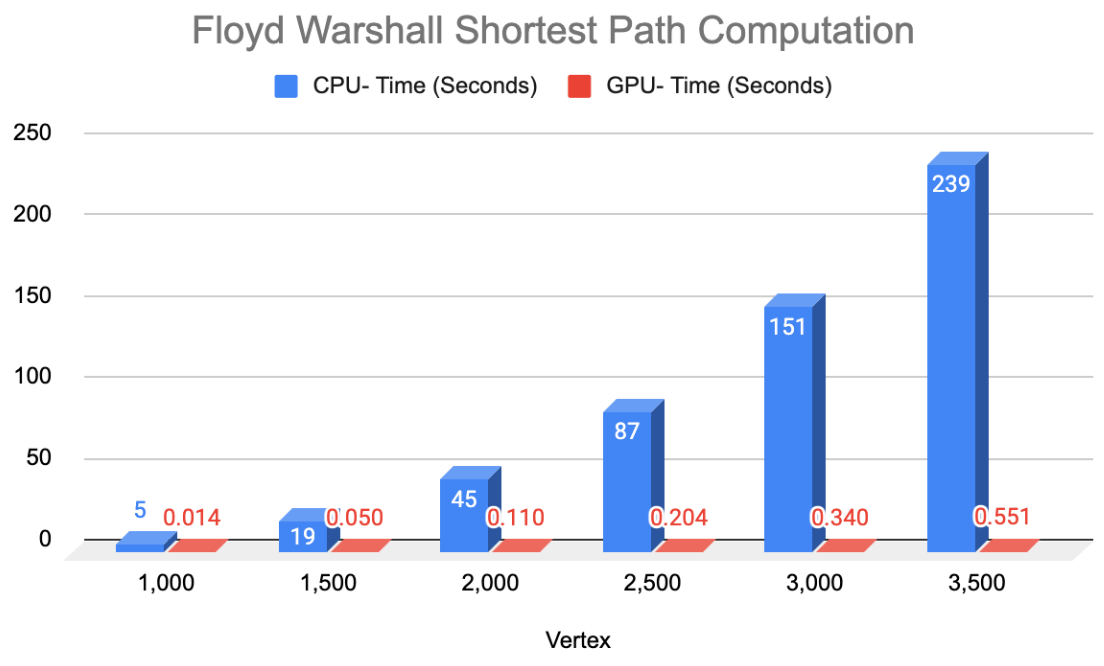

Fig.10. Comparation of CPU Time vs. GPU Time for Floyd-Warshall Shortest Path Computation

6. Conclusions

In this paper, we have shown the solutions of that using Floyd-Warshall Algorithm with CUDA for the shortest path problem. Floyd-Warshall is extremely useful in networking, similar to solutions to the shortest path problem. However, it is more effective at managing multiple stops on the route because it can calculate the shortest paths between all relevant nodes. One run of Floyd-Warshall can give all the information about a static network to optimize most types of paths. It can perfectly combine with CUDA, considerably speeding up finding the shortest path between all the pairs of vertices in a weighted graph.

References

[1]. Martín, P.J., Torres, R., Gavilanes, A. (2009). CUDA Solutions for the SSSP Problem. In: Allen, G., Nabrzyski, J., Seidel, E., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds) Computational Science – ICCS 2009. Lecture Notes in Computer Science, vol 5544. Springer, Berlin, Heidelberg.

[2]. Singh, A.P., & Singh, D.P. (2015). Implementation of K-shortest Path Algorithm in GPU Using CUDA. *Procedia Computer Science*, 48, 5-13.

[3]. *Floyd-Warshall algorithm*. Programiz. (n.d.). Retrieved December 2, 2022, from https://www.programiz.com/dsa/floyd-warshall-algorithm

[4]. Wang, Y. (n.d.). *Floyd-傻子也能看懂的弗洛伊德算法*. 开发者的网上家园. Retrieved December 2, 2022, from https://www.cnblogs.com/wangyuliang/p/9216365.html