
Accelerating Shortest Path Algorithms on the GPU using CUDA

Deep Bodra
University of Florida
Gainesville, Florida, USA
d.bodra@ufl.edu

Abstract

There are numerous applications where we would like to find shortest paths in a graph containing millions of vertices and edges. However, the serial versions of the existing fundamental shortest path algorithms take hours to run even on powerful CPUs. This paper presents parallel versions of some of the fundamental shortest path algorithms, optimizes them and reports their performance on the NVIDIA GPU using CUDA API.

1 Introduction

There are problems in many domains that can be represented using a graph and also involves finding the shortest distance between nodes. Some of these domains include road networks, communication networks, social network analysis and Very Large Scale Integration (VLSI chip layout). Even the optimum algorithms don't run fast for large graphs on powerful CPUs.

This paper implements the parallel versions of Bellman-Ford, Dijkstra's and Floyd-Warshall algorithm using the CUDA API and improves the performance and execution time on NVIDIA GPU

2 Parallel Shortest Path Algorithms

Consider a graph $G = (V, E)$ containing $|V|$ vertices and $|E|$ edges.

2.1 Bellman-Ford Algorithm

The serial version of the algorithm consists of $|V| - 1$ iterations. In each iteration, we loop through all the edges and relax them one by one.

The order in which the edges are relaxed is not important for this algorithm and we try to exploit that in the parallel version. However, the next iteration of the algorithm shouldn't start until the current one is finished. So we will have a kernel that will relax the edges for one iteration. The kernel should be launched $|V| - 1$ times.

We represent the graph in Compressed Sparse Row (CSR) format to fit large graphs in the global memory.

2.1.1 Naive

The *bellmanFordRelaxNaive* kernel handles one iteration of the Bellman-Ford algorithm. We assign one thread to each vertex. Each thread loops through all the outgoing edges of the assigned vertex and relaxes them sequentially.

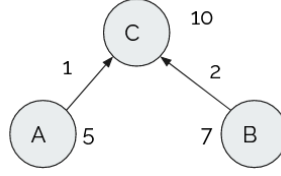


Figure 1: Bellman-Ford: Need for atomic minimum

The current iteration uses the costs from the previous iteration. If we use only one distance array and update that, then some threads will see updated costs in the current iteration itself. To avoid that we maintain two arrays, *prevDistance* and *distance*. We read costs from the *prevDistance* but write out the updated cost to *distance*. After the current iteration is done, we copy the *distance* values into the *prevDistance* array using *bellmanFordUpdateDistanceNaive*.

Since the edges are relaxed in parallel, there may be multiple threads that try to update the cost to the same vertex and this will cause a data race. So we must use *atomicMin* to update the costs. In Figure 1, if threads *A* and *B* read *distance[C]* at the same time and thread *B* writes after thread *A*, then the final value of *distance[C]* will be 9 but the correct value is 6.

The relaxation of edges also involves storing the parent vertex of the end vertex. So, updating *distance* and *parent* arrays for a relaxation should be one single atomic operation. CUDA doesn't have the functionality to perform atomic operations on multiple variables. A workaround for that is to use *bellmanFordParentNaive* kernel that is launched after all iterations are done i.e. we have shortest distances calculated.

To calculate *parent*, We use one thread per vertex and loop through the outgoing edges just as before. If the shortest distance to the end vertex is equal to the addition of the shortest distance to start vertex and the cost of the current edge then the parent of the end vertex is the start vertex.

2.1.2 Stride[4]

The graph size that the naive version can handle is limited by the maximum number of threads a device supports. The strided version is much more scalable. We use the classic grid-stride loop strategy.

The kernel is launched with fewer threads than the total number of vertices. Each thread now handles multiple vertices depending on the grid size and the number of vertices. The optimum number of threads required for a graph can be found by conducting various experiments. The strided version of the kernels are *bellmanFordRelaxStride*, *bellmanFordUpdateDistanceStride* and *bellmanFordParentStride*.

2.1.3 Stride with Flag[1]

In both the naive and the strided version, many threads spend unnecessary time looping through the edges that can't be relaxed. In Bellman-Ford algorithm, outgoing edges from a vertex are relaxed in the current iteration only if the distance to that vertex changed in the previous iteration.

To keep a track of this, we use a flag array of size $|V|$. While copying distance values into *prevDistance*, we set the flag of a vertex to true if the distance to that vertex in the previous iteration is greater than the distance to that vertex in the current iteration.

In the next iteration, the threads relax the outgoing edges of a vertex only if the flag for that vertex is true. The flag is again set to false while relaxing the outgoing edges of this vertex. The kernels for this approach are *bellmanFordRelaxStrideFlag* and *bellmanFordUpdateDistanceStrideFlag*. We can use *bellmanFordParentStride* to find the parent of a vertex.

2.2 Dijkstra's Algorithm

The optimum serial version of Dijkstra's algorithm uses a min-heap. The heap is ordered based on the distance of a node from the source node. The source node is added to the heap and the algorithm runs as long as there are nodes in the heap. A node is popped from the heap and the edges to all it's

```

for k from 1 to |V|
  for i from 1 to |V|
    for j from 1 to |V|
      if dist[i][j] > dist[i][k] + dist[k][j]
        dist[i][j] ← dist[i][k] + dist[k][j]
      end if
    end for
  end for
end for

```

Figure 2: Floyd-Warshall: Serial

neighbors are relaxed. If the distance to any of the neighbors changes, then the neighbor is added back to the heap.

We represent the graph in Adjacency Matrix format.

2.2.1 Naive

This algorithm is for calculating shortest distance from a single source and is not embarrassingly parallel. We use this algorithm for all pairs shortest path i.e. to find shortest path from each vertex to every other vertex. This is done by assigning one vertex to every thread. If we want to use heap, then we need to have one heap per thread and there is not enough shared memory per block for that. Instead we use the *distance* array to find the next vertex to be visited. The kernel *dijkstraNaive* runs dijkstra's in parallel.

To reduce the execution time, we can store the graph in constant memory but it is impractical for large graphs because the constant memory is limited.

2.3 Floyd-Warshall Algorithm

The serial version has $|V|$ iterations. In iteration k , all the edges in the graph are relaxed using k as an intermediate vertex. This algorithm requires the adjacency matrix representation of a graph and is embarrassingly parallel. See Figure 2

2.3.1 Super Naive

The *floydWarshallSuperNaive* kernel will be launched $|V|$ times and we parallelize the two inner loops of Figure 2 (second and third) by launching a 2D grid. We use one thread for every edge in the graph and relax that edge.

2.3.2 Super Naive Shared

For the previous version, in iteration k , every thread in a given row has the same i and k . All these threads access *distance*[i][k]. See Figure 3

We can have the first thread in a row load this value into shared memory. We will need to store one value for each row in shared memory. But this will cause shared memory bank conflicts because threads in different rows access different parts of the shared memory.

To avoid bank conflicts, we use a 1D block so that we store only one value per block and all the threads in the block access the same variable causing shared memory broadcast.

2.3.3 Naive

The graph size handled by the previous versions is limited by the maximum number of threads a device can support. We reduce the number of threads by parallelizing only the second loop and thus having one thread per vertex. The number of threads required drops and is scalable to an extent.

	j=0	j=1
i=0	(0,0)	(1,0)
i=1	(0,1)	(1,1)

Figure 3: Super Naive: Potential to use shared memory

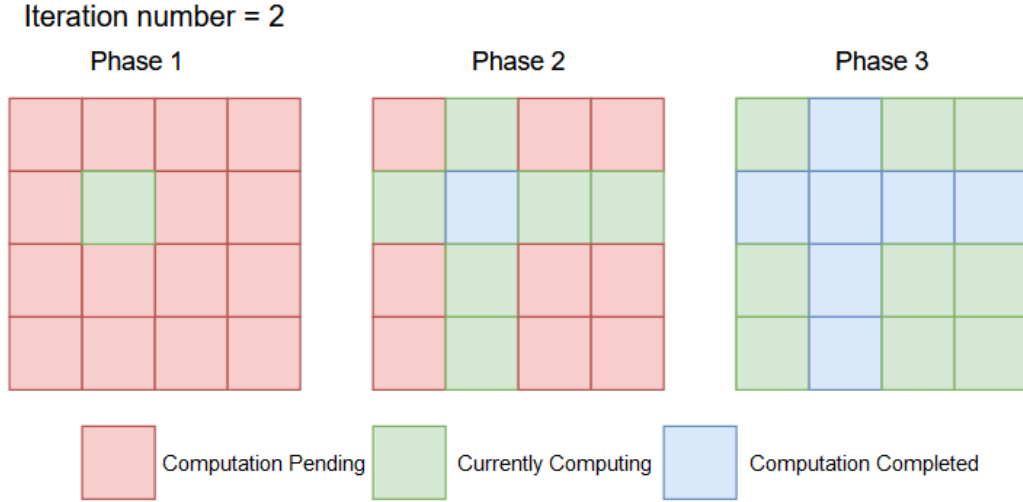


Figure 4: Vanilla GAN (small network)

2.3.4 Tiled[3]

All the previous versions are not efficient for very large graphs. We present a tiled version. This algorithm splits the adjacency matrix into 2D tiles of equal size. In iteration i , the i^{th} tile along the diagonal is considered as the primary tile. Each iteration of the algorithm has 3 phases.

In phase 1, we relax the edges in the primary tile by launching a single tile-sized block.

In phase 2, we relax the edges in the tiles that either share the same row or the same column as that of the primary tile. We update these tiles using vertices of the primary tile as intermediate vertices. For an edge from vertex i to vertex j , we read $distance[i][k]$ from the primary tile (calculated in phase 1) and $distance[k][j]$ from the current tile. The number of blocks launched for this is two times the number of tiles per row. The 1^{st} row of blocks map to the primary row and the 2^{nd} row of blocks map to the primary column.

In phase 3, we relax the edges in the remaining tiles. The row and column of the primary tile is called the primary row and primary column respectively. For an edge from vertex i to vertex j , we read $distance[i][k]$ from the primary column (calculated in phase 2) and $distance[k][j]$ from the primary row (calculated in phase 2). See Figure 4

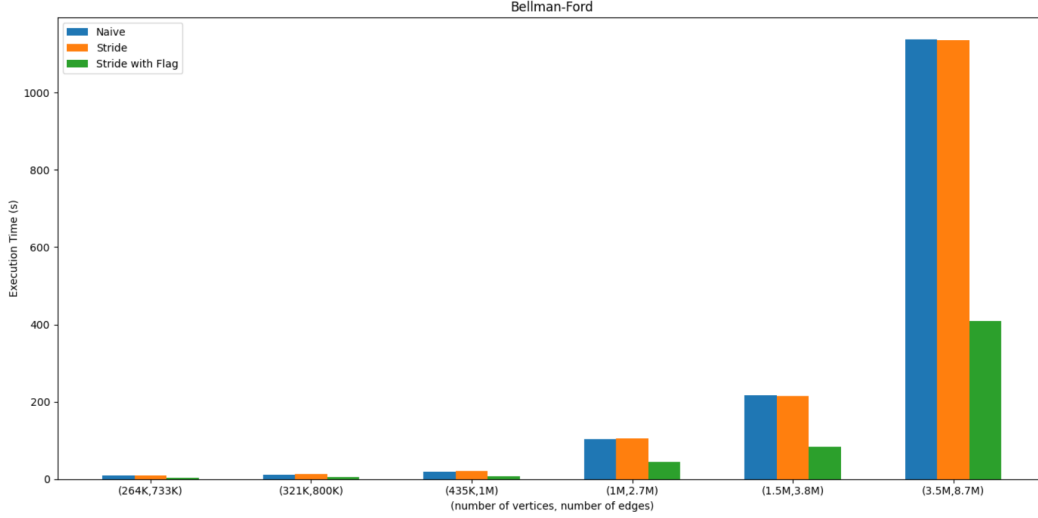


Figure 5: Results: Bellman-Ford

2.3.5 Tiled with Shared Memory

The global memory accesses in the previous approach is a bottleneck for the algorithm. We can make use of shared memory.

For phase 1, we load the primary tile in the shared memory and update it while relaxing the edges. For phase 2, we load the current tile and the primary tile and update the current tile while relaxing the edges. For phase 3, we load the current tile, a tile from the primary row and a tile from the primary column and update the current tile while relaxing the edges. At the end of each phase, the current tile in shared memory is written back to the global memory.

The reads from the global memory into shared memory and writes to the global memory are all coalesced.

3 Experiments and Results

The experiments were performed on HiPerGator using GeForce GPUs. The source code can be found at <https://github.com/deepbodra97/cuda-parallel-shortest-path>

3.1 Bellman-Ford Algorithm

The strided version runs slower than the naive version for small graphs. This is because small graphs need a small CUDA grid. The powerful GPU can easily run all the blocks in parallel. However, for large graphs the overhead of launching large grid slows down the naive version. The strided version runs about a 2000ms faster than the naive version. For graphs larger than this, we may notice a significant difference.

The stride with flag version is the best and solves the problem in reasonable time. It is about 2.8 times faster than the naive version. See Figure 5 and see Table 1

3.2 Dijkstra's Algorithm

Even the most efficient serial version of Dijkstra's algorithm runs slower than the naive parallel version. It is better than the serial version but still too slow for real world applications. See Table 2

Table 1: Results of Bellman-Ford on US road networks dataset[2]

Dataset	Number of Nodes	Number of Edges	CPU	Naive	Stride	Stride with Flag
nyc.txt	264,346	733,846	22min	8.58384s	8.77939s	3.91371s
bay.txt	321,270	800,172	-	11.7708s	12.3157s	5.12986s
col.txt	435,666	1,057,066	-	19.6589s	21.1299s	7.40538s
fla.txt	1,070,376	2,712,798	-	102.449s	105.376s	45.2268s
ne.txt	1,524,453	3,897,636	-	216.047s	214.760s	83.5226s
e.txt	3,598,623	8,778,114	-	1137.63s	1135.22s	409.138s

Table 2: Results of Dijkstra’s Algorithm on SNAP’s gnutella network dataset[5]

Dataset	Number of Nodes	Number of Edges	CPU	Naive
gnutella04.txt	10,876	39,994	24min	119.941s

3.3 Floyd-Warshall Algorithm

The super naive version is actually the fastest of all the naive implementations because of the coalesced reads from the global memory but is not practical for very large graphs in terms of scalability. It’s shared memory version does not perform better because of the barrier synchronization.

The tiled version using global memory outperforms all of the naive implementation and runs about 4-6 times faster than the naive version.

The shared memory version is about 2 times faster than the global memory version. It is about 8 times faster than the naive implementation. See Figure 6 and Table 3

4 Future Work

The current implementation of Bellman-Ford has control divergence because different nodes have different number of neighbors. One can try to use ELL representation of the Adjacency Matrix.

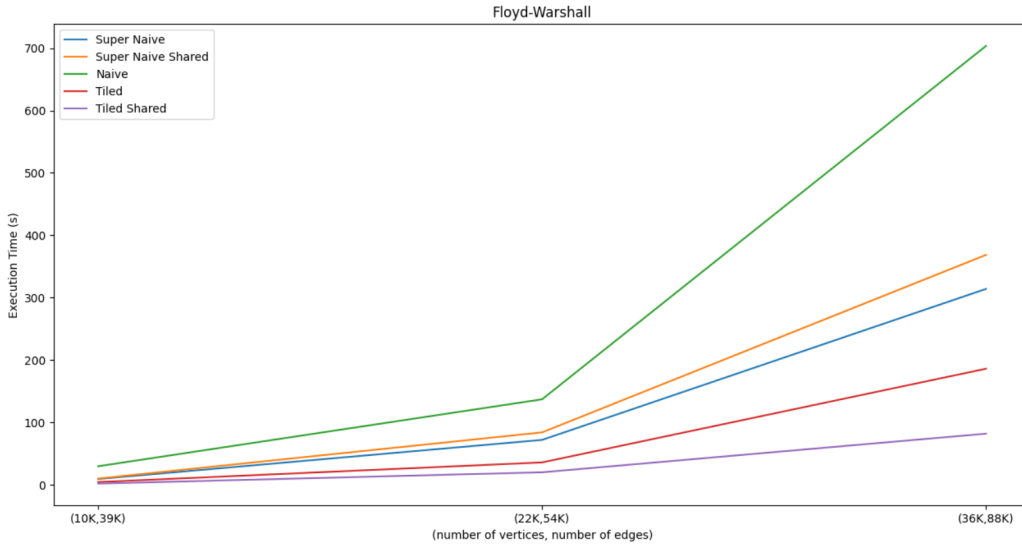


Figure 6: Results of Floyd-Warshall on SNAP’s gnutella network dataset[5]

Table 3: Results of Floyd-Warshall

Dataset	Number of Nodes	Number of Edges	CPU	Naive	Super Naive	Super Naive Shared	Tiled	Tiled with Shared Memory
gnutella04.txt	10,876	39,994	>1hr	30.1521s	9.73591s	10.4189s	4.8403s	2.37704s
gnutella25.txt	22,687	54,705	-	137.369	72.3713s	84.4068s	36.2867s	20.4614s
gnutella30.txt	36,682	88,328	-	703.358s	314.191s	368.685s	186.350	82.3174s

The tiled implementation of Floyd-Warshall algorithm can only handle graphs that fit in the global memory. Staged Load[6] technique can be used to solve graphs that don't fit in the global memory.

References

- [1] Pankhari Agarwal and Maitreyee Dutta. "New Approach of Bellman Ford Algorithm on GPU using Compute Unified Design Architecture (CUDA)". In: *International Journal of Computer Applications* 110 (Jan. 2015), pp. 11–15. DOI: 10.5120/19375-1027.
- [2] DIMACS. *9th DIMACS Implementation Challenge - Shortest Paths*. <http://users.diag.uniroma1.it/challenge9/download.shtml>. June 2010.
- [3] *GH '08: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*. Sarajevo, Bosnia and Herzegovina: Eurographics Association, 2008. ISBN: 9783905674095.
- [4] Mark Harris. *CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops*. 2013. URL: <https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/>.
- [5] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. <http://snap.stanford.edu/data>. June 2014.
- [6] Ben Lund and Justin W Smith. *A Multi-Stage CUDA Kernel for Floyd-Warshall*. 2010. arXiv: 1001.4108 [cs.DC].

Instructions to Run Code

This file is attached with the source code to make it easier to copy commands

```

1 Instructions to run
2
3 1. Project Structure
4   1. Unzip the project
5   2. You will see 2 folders in the project. "data" contains all the dataset files.
      sample1.txt represents a graph with 6 nodes and 9 edges. sample2.txt represents
      a graph with 100 nodes and 99000 edges. "output" stores the output files.
      Currently it has a blank dummy files. The output of all the algorithms on just
      one dataset itself is 20GB so they are not included due to size constraints.
      sample files are to be used to confirm the correctness of the algorithms
      because CPU could take hours to run for other files. The output files will be
      overwritten if you rerun the code.
6   3. The source code files and CMakeLists.txt will be in the root of the project.
7
8 2. Load modules
9   2.1 module load ufrs cmake/3.19.1 intel/2018.1.163 cuda/10.0.130
10
11 3. Compile
12   3.1 mkdir Release
13   3.2 cd Release
14   3.3 cmake -DCMAKE_BUILD_TYPE=Release ..
15   3.4 make
16
17 3 executables named BellmanFord, Dijkstra and FloydWarshall will be created inside
      the Release folder
18

```

```

19 4. Bellman Ford
20
21 Command Format
22 srun -p gpu --nodes=1 --gpus=geforce:1 --time=00:05:00 --mem=1500 --pty -u
    BellmanFord algorithm inputFileNames source validateOutput outputFormat -i
23
24 algorithm=any integer in the range [0,3] both inclusive 0=CPU, 1=strided, 2=stride
    with flag
25 inputFileNames=name of input file with extension (this file should be in "data"
    folder) use any of these files sample1.txt, sample2.txt, nyc.txt, bay.txt, col.
    txt, fla.txt, ne.txt, e.txt
26
27 source = source node in the graph. any number in the range [0, number of nodes-1]
    both inclusive
28
29 validateOutput=true|false true=compares gpu's output against cpu's output
30
31 outputFormat=none|print|write
32 print=prints distance and path info on screen
33 write=distance and path is written in a file named bf{algorithm}.txt in "output"
    folder
34 none=doesn't print or write distance and path [use this to time the kernels]
35
36 Run this sample command to make sure everything is set up correctly. It will print
    output on screen for the input file sample1.txt
37 srun -p gpu --nodes=1 --gpus=geforce:1 --time=01:00:00 --mem=1600 --pty -u
    BellmanFord 1 sample1.txt 0 false print -i
38
39 5. Dijkstra
40
41 Command Format
42 srun -p gpu --nodes=1 --gpus=geforce:1 --time=01:00:00 --mem=1600 --pty -u
    Dijkstra algorithm inputFileNames validateOutput outputFormat -i
43
44 algorithm=any integer in the range [0,1] both inclusive. 0=CPU, 1=CPU
45
46 inputFileNames=name of input file with extension (this file should be in "data"
    folder) use any of these files sample1.txt, sample2.txt, gnutella04.txt,
    gnutella25.txt, gnutella30.txt
47
48 validateOutput=true|false true=compares gpu's output against cpu's output
49
50 outputFormat=none|print|write
51 print=prints distance and path info on screen
52 write=distance and path is written in a file named d{algorithm}.txt in "output"
    folder
53 none=doesn't print or write distance and path [use this to time the kernels]
54
55
56 Run this sample command to make sure everything is set up correctly. It will print
    output on screen for the input file sample1.txt
57 srun -p gpu --nodes=1 --gpus=geforce:1 --time=01:00:00 --mem=1500 --pty -u
    Dijkstra 0 sample1.txt false print -i
58
59 6. Floyd Warshall
60 Command Format
61 srun -p gpu --nodes=1 --gpus=geforce:1 --time=01:00:00 --mem=1500 --pty -u
    FloydWarshall algorithm inputFileNames validateOutput outputFormat -i
62
63 algorithm=any integer in the range [0,5], 0=CPU, 1=Super Naive, 2=Naive, 3=Super
    Naive Shared, 4=Tiled Global, 5=Tiled Shared Memory
64
65 inputFileNames=name of input file with extension
66
67 validateOutput=true|false true=compares cpu output with gpu output

```



```

68
69 outputFormat=none|print|write
70 print=prints distance and path info on screen
71 write=distance and path is written in a file named fw{algorithm}.txt in "output"
   folder
72 none=doesnt print or write distance and path [use this to time the kernels]
73
74 sample command
75 srunk -p gpu --nodes=1 --gpus=geforce:1 --time=01:00:00 --mem=25000 --pty -u
   FloydWarshall 1 sample1.txt false print -i
76
77
78 7. utils.py
79 The dataset files have already been parsed. There is no need to run these commands
   . They are provided just for documentation purpose.
80
81 1. Create a random graph of 100 vertices and store it in a file named sample2.txt
82 python utils.py random 100 ./data/sample2.txt
83
84 2. Parse a file from DIMACS dataset
85 python utils.py parse ./data/nyc.txt
86
87 3. Add random weights in the range [1, 100] to a file from SNAP dataset
88 python utils.py add ./data/gnutella04.txt 1 100
89
90 4. Replace already added weights with new random weights in the range [1, 100] to
   a file generated from command 3
91 python utils.py replace ./data/gnutella04.txt 1 100

```

Listing 1: README.txt

Source Code

```

1 #ifndef UTILS_H
2 #define UTILS_H
3
4 #include <iostream>
5 #include <fstream>
6 #include <string>
7 #include <vector>
8 #include <list>
9 #include <map>
10 #include <queue>
11 #include <sstream>
12
13 #include <cassert>
14
15 #include <limits.h>
16
17 #include <exception>
18
19 #define NUM_ITERATION_WARMUP 10
20 #define INF INT_MAX
21 #define THREADS_PER_BLOCK 1024
22
23 using namespace std;
24
25 int* fileToCostMatrix(string filename, int& numVertex, int& numEdges);
26 struct Graph* fileToAdjacencyList(string filename, struct Graph* costMatrix);
27 void fileToAdjacencyList(string filename, map<int, list<pair<int, int>>>&
   adjacencyList, int& numVertex, int& numEdges);
28
29 void adjacencyListToCSR(map<int, list<pair<int, int>>>& adjacencyList, vector<int>&
   vertices, vector<int>& indices, vector<int>& edges, vector<int>& weights);

```

```

30
31 void APSPInitDistanceParent(int numVertex, int* costMatrix, int* distance, int*
    parent);
32
33 void validateDistanceSSSP(int numVertex, int* exp_distance, int* distance);
34 void validateDistanceAPSP(int numVertex, int* exp_distance, int* distance);
35
36 void printPathSSSP(int numVertex, int* distance, int* parent);
37 void printPathAPSP(int numVertex, int* distance, int* parent);
38
39 void writeOutPathSSSP(string filepath, int numVertex, int* distance, int* parent);
40 void writeOutPathAPSP(string filepath, int numVertex, int* distance, int* parent);
41
42
43 #endif

```

Listing 2: utils.h

```

1 #include "utils.h"
2
3 void splitBySpaceToVector(const string& line, vector<string>& tokens) {
4     stringstream linestream(line);
5     string token;
6     while (linestream >> token) {
7         tokens.push_back(token);
8     }
9 }
10
11 // converts input graph file to cost matrix
12 int* fileToCostMatrix(string filename, int& numVertex, int& numEdges) {
13     cout << "Reading input file" << endl;
14     ifstream file(filename);
15     string line;
16
17     getline(file, line);
18     vector<string> tokens;
19
20     splitBySpaceToVector(line, tokens);
21     numVertex = stoi(tokens[0]), numEdges = stoi(tokens[1]);
22
23     int* costMatrix = (int*)malloc(numVertex * numVertex * sizeof(int));
24     if (costMatrix == NULL) {
25         cout << "Malloc failed" << endl;
26         throw std::exception();
27     }
28     fill(costMatrix, costMatrix + numVertex * numVertex, INF);
29
30     while (getline(file, line)) { // parse input file line by line
31         stringstream linestream(line);
32         vector<string> tokens;
33         string token;
34         while (linestream >> token) {
35             tokens.push_back(token);
36         }
37         int src = stoi(tokens[0]), dest = stoi(tokens[1]), cost = stoi(tokens[2]);
38         costMatrix[src * numVertex + dest] = cost; // update cost matrix
39     }
40     cout << "Finished reading input file" << endl;
41     return costMatrix;
42 }
43
44 // converts input graph file to adjacency list
45 void fileToAdjacencyList(string filename, map<int, list<pair<int, int>>>&
    adjacencyList, int& numVertex, int& numEdges) {
46     cout << "Reading input file" << endl;

```

```

47     ifstream file(filename);
48     string line;
49
50     getline(file, line);
51     vector<string> tokens;
52
53     splitBySpaceToVector(line, tokens);
54     numVertex = stoi(tokens[0]), numEdges = stoi(tokens[1]);
55
56     while (getline(file, line)) { // parse input file line by line
57         tokens.clear();
58         splitBySpaceToVector(line, tokens);
59         int src = stoi(tokens[0]), dest = stoi(tokens[1]), cost = stoi(tokens[2]);
60         adjacencyList[src].push_back(make_pair(dest, cost));
61     }
62     cout << "Finished reading input file" << endl;
63 }
64
65 // converts adjacency list to CSR format
66 void adjacencyListToCSR(map<int, list<pair<int, int>>>& adjacencyList, vector<int>&
vertices, vector<int>& indices, vector<int>& edges, vector<int>& weights) {
67     int index = 0;
68     indices.push_back(index);
69     for (auto uIter = adjacencyList.begin(); uIter != adjacencyList.end(); ++uIter)
70     {
71         int u = uIter->first;
72         vertices.push_back(u);
73         index += uIter->second.size();
74         indices.push_back(index);
75         for (auto vIter = uIter->second.begin(); vIter != uIter->second.end(); ++
vIter) {
76             edges.push_back(vIter->first);
77             weights.push_back(vIter->second);
78         }
79     }
80
81     // initialize distance and parent for floyd warshall
82     void APSPInitDistanceParent(int numVertex, int* costMatrix, int* distance, int*
parent) {
83         cout << "Initializing distance and parent matrices using the cost matrix" <<
endl;
84         for (int i = 0; i < numVertex; i++) {
85             for (int j = 0; j < numVertex; j++) {
86                 if (i == j) {
87                     distance[i * numVertex + j] = 0;
88                     parent[i * numVertex + j] = -1;
89                 }
90                 else if (costMatrix[i * numVertex + j] == INF) {
91                     distance[i * numVertex + j] = INF;
92                     parent[i * numVertex + j] = -1;
93                 }
94                 else {
95                     distance[i * numVertex + j] = costMatrix[i * numVertex + j];
96                     parent[i * numVertex + j] = i;
97                 }
98             }
99         }
100     }
101
102     // compare gpu output with cpu output for single source shortest path
103     void validateDistanceSSSP(int numVertex, int* expDistance, int* distance) {
104         for (int i = 0; i < numVertex; i++) {
105             assert(expDistance[i] == distance[i]);
106         }

```

```

107     cout << "Validation Successful" << endl;
108 }
109
110 // compare gpu output with cpu output for all pairs shortest path
111 void validateDistanceAPSP(int numVertex, int* expDistance, int* distance) {
112     for (int i = 0; i < numVertex; i++) {
113         for (int j = 0; j < numVertex; j++) {
114             assert(expDistance[i * numVertex + j] == distance[i * numVertex + j]);
115         }
116     }
117     cout << "Validation Successful" << endl;
118 }
119
120 // print single source shortest path on screen
121 void printPathSSSP(int numVertex, int* distance, int* parent) {
122     cout << "Node\tCost\tPath" << endl;
123     for (int i = 0; i < numVertex; i++) {
124         if (distance[i] != INF && distance[i] != 0) {
125             cout << i << "\t" << distance[i] << "\t";
126             cout << i;
127
128             int tmp = parent[i];
129             while (tmp != -1)
130             {
131                 cout << "<-" << tmp;
132                 tmp = parent[tmp];
133             }
134         }
135         else {
136             cout << i << "\t" << "NA" << "\t" << "-";
137         }
138         cout << endl;
139     }
140 }
141
142 // write single source shortest path to a file
143 void writeOutPathSSSP(string filepath, int numVertex, int* distance, int* parent) {
144     ofstream out(filepath);
145     out << "Node\tCost\tPath" << endl;
146     for (int i = 0; i < numVertex; i++) {
147         if (distance[i] != INF && distance[i] != 0) {
148             out << i << "\t" << distance[i] << "\t";
149             out << i;
150
151             int tmp = parent[i];
152             while (tmp != -1)
153             {
154                 out << "<-" << tmp;
155                 tmp = parent[tmp];
156             }
157             out << endl;
158         }
159         else {
160             // uncomment this line to output "NA" for paths that don't exist
161             // out << i << "\t" << "NA" << "\t" << "-";
162             // out << endl;
163         }
164     }
165     out.close();
166 }
167
168 // print all pairs source shortest path on screen
169 void printPathAPSP(int numVertex, int* distance, int* parent) {
170     for (int src = 0; src < numVertex; src++) {
171         cout << "Source: " << src << endl;

```

```

172     cout << "Node\tCost\tPath" << endl;
173     for (int dest = 0; dest < numVertex; dest++) {
174         if (distance[src * numVertex + dest] != INF && distance[src * numVertex
+ dest] != 0) {
175             cout << dest << "\t" << distance[src * numVertex + dest] << "\t";
176             cout << dest;
177
178             int tmp = parent[src * numVertex + dest];
179             while (tmp != -1)
180             {
181                 cout << "<-" << tmp;
182                 tmp = parent[src * numVertex + tmp];
183             }
184             cout << endl;
185         }
186         else {
187             // uncomment this line to output "NA" for paths that don't exist
188             // cout << dest << "\t" << "NA" << "\t" << "<";
189             // cout << endl;
190         }
191     }
192     cout << endl;
193 }
194 }
195
196 // write all pairs source shortest path to a file
197 void writeOutPathAPSP(string filepath, int numVertex, int* distance, int* parent) {
198     ofstream out(filepath);
199     for (int src = 0; src < numVertex; src++) {
200         out << "Source: " << src << endl;
201         out << "Node\tCost\tPath" << endl;
202         for (int dest = 0; dest < numVertex; dest++) {
203             if (distance[src * numVertex + dest] != INF && distance[src * numVertex
+ dest] != 0) {
204                 out << dest << "\t" << distance[src * numVertex + dest] << "\t";
205                 out << dest;
206
207                 int tmp = parent[src * numVertex + dest];
208                 while (tmp != -1)
209                 {
210                     out << "<-" << tmp;
211                     tmp = parent[src * numVertex + tmp];
212                 }
213                 out << endl;
214             }
215             else {
216                 // uncomment this line to output "NA" for paths that don't exist
217                 // out << dest << "\t" << "NA" << "\t" << "<";
218                 // out << endl;
219             }
220         }
221         out << endl;
222     }
223     out.close();
224 }

```

Listing 3: utils.cpp

```

1 #ifndef CUDA_CHECK_CUH
2 #define CUDA_CHECK_CUH
3
4 #include "cuda_runtime.h"
5 #include "device_launch_parameters.h"
6
7 #include <stdio>

```

```

8 #include <cassert>
9
10 /* Wrapper to provide error checking for CUDA API calls */
11
12 inline
13 cudaError_t cudaCheck(cudaError_t result) {
14     if (result != cudaSuccess) {
15         fprintf(stderr, "CUDA Runtime Error: %s\n", cudaGetErrorString(result));
16         assert(result == cudaSuccess);
17     }
18     return result;
19 }
20
21 __global__
22 void warmupGpu() {
23     __shared__ int s_tid;
24     int tid = blockIdx.x * blockDim.x + threadIdx.x;
25     if (threadIdx.x == 0) {
26         s_tid = tid;
27     }
28     __syncthreads();
29     tid = s_tid;
30 }
31
32 #endif /*CUDA_CHECK_CUH*/

```

Listing 4: utils.cuh

```

1 #include "cuda_runtime.h"
2 #include "device_launch_parameters.h"
3
4 #include "utils.cuh"
5
6 #include <iostream>
7
8 #include "utils.h"
9
10 using namespace std;
11
12 /*****
13 SERIAL VERSION
14 *****/
15
16 // run bellman ford on cpu
17 void runCpuBellmanFord(int src, int numVertex, int* vertices, int* indices, int*
    edges, int* weights, int* distance, int* parent) {
18     cudaEvent_t start, stop;
19     cudaEventCreate(&start);
20     cudaEventCreate(&stop);
21     float duration;
22     cudaEventRecord(start, 0);
23
24     distance[src] = 0;
25     for (int k = 0; k < numVertex-1; k++) { // a total of numVertex-1 iterations
26         for (int i = 0; i < numVertex; i++) { // loop through all vertices
27             for (int j = indices[i]; j < indices[i + 1]; j++) { // loop through
                neighbors of i
28                 int v = edges[j]; // neighbor j
29                 int w = weights[j]; // cost from i to j
30
31                 if (distance[i] != INF && (distance[i] + w) < distance[v]) { //
                    relax
32                     parent[v] = i;
33                     distance[v] = distance[i] + w;
34                 }
            }
        }
    }

```

```

35     }
36 }
37 }
38
39 cudaEventRecord(stop, 0);
40 cudaEventSynchronize(stop);
41 cudaEventElapsedTime(&duration, start, stop);
42 cout << "Time: " << duration << "ms" << endl;
43 }
44
45 /*****
46 NAIVE VERSION
47 *****/
48
49 // relax
50 __global__
51 void bellmanFordRelaxNaive(int numVertex, int* vertices, int* indices, int* edges,
52     int* weights, int* prevDistance, int* distance, int* parent) {
53     int i = blockIdx.x * blockDim.x + threadIdx.x; // thread i relaxes outgoing
54     edges from vertex i
55     if (i < numVertex) {
56         for (int j = indices[i]; j < indices[i + 1]; j++) { // loop through
57             neighbors of i
58             int v = edges[j]; // neighbor j
59             int w = weights[j]; // cost from i to j
60
61             if (prevDistance[i] != INF && (prevDistance[i] + w) < distance[v]) { //
62                 relax
63                 atomicMin(&distance[v], prevDistance[i] + w); // atomic minimum
64             }
65         }
66     }
67 }
68
69 // copy the updated cost values in prevDistance
70 __global__
71 void bellmanFordUpdateDistanceNaive(int numVertex, int* prevDistance, int* distance)
72 {
73     int i = blockIdx.x * blockDim.x + threadIdx.x; // thread i handles vertex i
74     if (i < numVertex) {
75         prevDistance[i] = distance[i]; // copy distance into prevDistance
76     }
77 }
78
79 // find parents of the vertices
80 __global__
81 void bellmanFordParentNaive(int numVertex, int* vertices, int* indices, int* edges,
82     int* weights, int* distance, int* parent) {
83     int i = blockIdx.x * blockDim.x + threadIdx.x; // thread i checks if it is the
84     parent of any of it's neighbors
85     if (i < numVertex) {
86         for (int j = indices[i]; j < indices[i + 1]; j++) { // loop through
87             neighbors of i
88             int v = edges[j]; // neighbor j
89             int w = weights[j]; // cost from i to j
90
91             if (distance[i] != INF && (distance[i] + w) == distance[v]) {
92                 parent[v] = i;
93             }
94         }
95     }
96 }
97
98 // run naive version of bellmand ford

```

```

91 void runBellmanFordNaive(int src, int numVertex, int* vertices, int* indices, int*
edges, int* weights, int* distance, int* parent) {
92     int* prevDistance = (int*)malloc(numVertex * sizeof(int));
93
94     fill(prevDistance, prevDistance + numVertex, INF); // fill with INF
95
96     prevDistance[src] = 0;
97     distance[src] = 0;
98
99     // time the algorithm
100     cudaEvent_t start, stop;
101     cudaEventCreate(&start);
102     cudaEventCreate(&stop);
103     float duration;
104     cudaEventRecord(start, 0);
105
106     // device pointers
107     int* d_prevDistance;
108     int* d_distance;
109     int* d_parent;
110
111     // allocate memory on device
112     cudaCheck(cudaMalloc((void**)&d_prevDistance, numVertex * sizeof(int)));
113     cudaCheck(cudaMalloc((void**)&d_distance, numVertex * sizeof(int)));
114     cudaCheck(cudaMalloc((void**)&d_parent, numVertex * sizeof(int)));
115
116     // copy from cpu to gpus
117     cudaCheck(cudaMemcpy(d_prevDistance, prevDistance, numVertex * sizeof(int),
cudaMemcpyHostToDevice));
118     cudaCheck(cudaMemcpy(d_distance, distance, numVertex * sizeof(int),
cudaMemcpyHostToDevice));
119     cudaCheck(cudaMemcpy(d_parent, parent, numVertex * sizeof(int),
cudaMemcpyHostToDevice));
120
121     cout << "Calculating shortest distance" << endl;
122     for (int k = 0; k < numVertex - 1; k++) { // numVertex-1 iterations
123         bellmanFordRelaxNaive << <(numVertex - 1) / THREADS_PER_BLOCK + 1,
THREADS_PER_BLOCK >> > (numVertex, vertices, indices, edges, weights,
d_prevDistance, d_distance, d_parent);
124         cudaCheck(cudaGetLastError()); // check if kernel launch failed
125         cudaCheck(cudaDeviceSynchronize()); // wait for kernel to finish
126         bellmanFordUpdateDistanceNaive << <(numVertex - 1) / THREADS_PER_BLOCK + 1,
THREADS_PER_BLOCK >> > (numVertex, d_prevDistance, d_distance);
127         cudaCheck(cudaGetLastError()); // check if kernel launch failed
128         cudaCheck(cudaDeviceSynchronize()); // wait for kernel to finish
129     }
130     cout << "Constructing path" << endl;
131     bellmanFordParentNaive << <(numVertex - 1) / THREADS_PER_BLOCK + 1,
THREADS_PER_BLOCK >> > (numVertex, vertices, indices, edges, weights,
d_distance, d_parent);
132     cudaCheck(cudaGetLastError()); // check if kernel launch failed
133     cudaCheck(cudaDeviceSynchronize()); // wait for kernel to finish
134
135     // copy from gpu to cpu
136     cout << "Copying results to CPU" << endl;
137     cudaCheck(cudaMemcpy(distance, d_distance, numVertex * sizeof(int),
cudaMemcpyDeviceToHost));
138     cudaCheck(cudaMemcpy(parent, d_parent, numVertex * sizeof(int),
cudaMemcpyDeviceToHost));
139
140     cudaCheck(cudaFree(d_prevDistance));
141
142     cudaEventRecord(stop, 0);
143     cudaEventSynchronize(stop);
144     cudaEventElapsedTime(&duration, start, stop);

```



```

145     cout << "Time: " << duration << "ms" << endl;
146 }
147
148 /*****
149 STRIDE VERSION
150 *****/
151
152 // relax with stride
153 __global__
154 void bellmanFordRelaxStride(int numVertex, int* vertices, int* indices, int* edges,
155     int* weights, int* prevDistance, int* distance, int* parent) {
156     int tid = blockIdx.x * blockDim.x + threadIdx.x; // thread tid relaxes outgoing
157     edges from vertex tid, tid+stride, tid+2*stride,...
158     int stride = blockDim.x * gridDim.x; // stride length
159
160     for(int i = tid; i < numVertex; i += stride){
161         for (int j = indices[i]; j < indices[i + 1]; j++) {
162             int v = edges[j]; // neighbor j
163             int w = weights[j]; // cost from i to j
164
165             if (prevDistance[i] != INF && (prevDistance[i] + w) < distance[v]) { //
166                 relax
167                     atomicMin(&distance[v], prevDistance[i] + w);
168             }
169         }
170     }
171 }
172
173 // copy the updated cost values in prevDistance with stride
174 __global__
175 void bellmanFordUpdateDistanceStride(int numVertex, int* prevDistance, int* distance
176 ) {
177     int tid = blockIdx.x * blockDim.x + threadIdx.x; // thread tid handles vertex
178     tid, tid+stride, tid+2*stride, ...
179     int stride = blockDim.x * gridDim.x;
180
181     for (int i = tid; i < numVertex; i += stride) {
182         prevDistance[i] = distance[i]; // copy distance into prevDistance
183     }
184 }
185
186 // find parents of the vertices with stride
187 __global__
188 void bellmanFordParentStride(int numVertex, int* vertices, int* indices, int* edges,
189     int* weights, int* distance, int* parent) {
190     int tid = blockIdx.x * blockDim.x + threadIdx.x;
191     int stride = blockDim.x * gridDim.x;
192
193     for (int i = tid; i < numVertex; i += stride) {
194         for (int j = indices[i]; j < indices[i + 1]; j++) { // loop through
195             neighbors of i
196             int v = edges[j]; // neighbor j
197             int w = weights[j]; // cost from i to j
198
199             if (distance[i] != INF && (distance[i] + w) == distance[v]) {
200                 parent[v] = i;
201             }
202         }
203     }
204 }
205
206 // run stride version of bellmand ford
207 void runBellmanFordStride(int src, int numVertex, int* vertices, int* indices, int*
208     edges, int* weights, int* distance, int* parent) {
209     int* prevDistance = (int*)malloc(numVertex * sizeof(int));

```

```

202     fill(prevDistance, prevDistance + numVertex, INF); // fill with INF
203
204
205     prevDistance[src] = 0;
206     distance[src] = 0;
207
208     // time the algorithm
209     cudaEvent_t start, stop;
210     cudaEventCreate(&start);
211     cudaEventCreate(&stop);
212     float duration;
213     cudaEventRecord(start, 0);
214
215     // device pointers
216     int* d_prevDistance;
217     int* d_distance;
218     int* d_parent;
219
220
221     // allocate memory on device
222     cudaCheck(cudaMalloc((void**)&d_prevDistance, numVertex * sizeof(int)));
223     cudaCheck(cudaMalloc((void**)&d_distance, numVertex * sizeof(int)));
224     cudaCheck(cudaMalloc((void**)&d_parent, numVertex * sizeof(int)));
225
226     // copy from cpu to gpu
227     cudaCheck(cudaMemcpy(d_prevDistance, prevDistance, numVertex * sizeof(int),
228         cudaMemcpyHostToDevice));
229     cudaCheck(cudaMemcpy(d_distance, distance, numVertex * sizeof(int),
230         cudaMemcpyHostToDevice));
231     cudaCheck(cudaMemcpy(d_parent, parent, numVertex * sizeof(int),
232         cudaMemcpyHostToDevice));
233
234     int numBlocks = ((numVertex - 1) / THREADS_PER_BLOCK + 1) / 2; // use half the
235     // number of required blocks
236     cout << "Calculating shortest distance" << endl;
237     for (int k = 0; k < numVertex - 1; k++) { // numVertex-1 iterations
238         bellmanFordRelaxStride << <numBlocks, THREADS_PER_BLOCK >> > (numVertex,
239             vertices, indices, edges, weights, d_prevDistance, d_distance, d_parent);
240         cudaCheck(cudaGetLastError()); // check if kernel launch failed
241         cudaCheck(cudaDeviceSynchronize()); // wait for the kernel to finish
242         bellmanFordUpdateDistanceStride << <numBlocks, THREADS_PER_BLOCK >> > (
243             numVertex, d_prevDistance, d_distance);
244         cudaCheck(cudaGetLastError()); // check if kernel launch failed
245         cudaCheck(cudaDeviceSynchronize()); // wait for the kernel to finish
246     }
247     cout << "Constructing path" << endl;
248     bellmanFordParentStride << <numBlocks, THREADS_PER_BLOCK >> > (numVertex,
249         vertices, indices, edges, weights, d_distance, d_parent);
250     cudaCheck(cudaGetLastError()); // check if kernel launch failed
251     cudaCheck(cudaDeviceSynchronize()); // wait for the kernel to finish
252
253     // copy from gpu to cpu
254     cout << "Copying results to CPU" << endl;
255     cudaCheck(cudaMemcpy(distance, d_distance, numVertex * sizeof(int),
256         cudaMemcpyDeviceToHost));
257     cudaCheck(cudaMemcpy(parent, d_parent, numVertex * sizeof(int),
258         cudaMemcpyDeviceToHost));
259
260     cudaCheck(cudaFree(d_prevDistance));
261
262     cudaEventRecord(stop, 0);
263     cudaEventSynchronize(stop);
264     cudaEventElapsedTime(&duration, start, stop);
265     cout << "Time: " << duration << "ms" << endl;
266 }

```

```

258
259
260
261 /*****
262 STRIDE WITH FLAG VERSION
263 *****/
264
265 // relax with stride
266 __global__
267 void bellmanFordRelaxStrideFlag(int numVertex, int* vertices, int* indices, int*
    edges, int* weights, int* prevDistance, int* distance, int* parent, bool* flag)
    {
268     int tid = blockIdx.x * blockDim.x + threadIdx.x; // thread i relaxes outgoing
    edges from vertex i
269     int stride = blockDim.x * gridDim.x;
270
271     for (int i = tid; i < numVertex; i += stride) {
272         if (flag[i]) { // relax outgoing edges of i only if distance to i changed in
    the previous iteration
273             flag[i] = false;
274             for (int j = indices[i]; j < indices[i + 1]; j++) { // loop through
    neighbors of i
275                 int v = edges[j]; // neighbor j
276                 int w = weights[j]; // cost from i to j
277
278                 if (prevDistance[i] != INF && (prevDistance[i] + w) < distance[v]) {
    // relax
279                     atomicMin(&distance[v], prevDistance[i] + w);
280                 }
281             }
282         }
283     }
284 }
285
286 // copy the updated cost values in prevDistance with stride and set flag to true if
    the cost to i was changed in the current iteration
287 __global__
288 void bellmanFordUpdateDistanceStrideFlag(int numVertex, int* prevDistance, int*
    distance, bool* flag) {
289     int tid = blockIdx.x * blockDim.x + threadIdx.x;
290     int stride = blockDim.x * gridDim.x;
291
292     for (int i = tid; i < numVertex; i += stride) {
293         if (prevDistance[i] > distance[i]) {
294             flag[i] = true;
295         }
296         prevDistance[i] = distance[i];
297     }
298 }
299
300 // run stride with flag version of bellmand ford
301 void runBellmanFordStrideFlag(int src, int numVertex, int* vertices, int* indices,
    int* edges, int* weights, int* distance, int* parent) {
302     int* prevDistance = (int*)malloc(numVertex * sizeof(int));
303     bool* flag = (bool*)malloc(numVertex * sizeof(bool));
304
305     fill(prevDistance, prevDistance + numVertex, INF); // fill with INF
306     fill(flag, flag + numVertex, false); // fill with false
307
308     prevDistance[src] = 0;
309     distance[src] = 0;
310     flag[src] = true;
311
312     // time the algorithm
313     cudaEvent_t start, stop;

```

```

314 cudaEventCreate(&start);
315 cudaEventCreate(&stop);
316 float duration;
317 cudaEventRecord(start, 0);
318
319 // device pointers
320 int* d_prevDistance;
321 int* d_distance;
322 int* d_parent;
323 bool* d_flag;
324
325 // allocate memory on gpu
326 cudaCheck(cudaMalloc((void**)&d_prevDistance, numVertex * sizeof(int)));
327 cudaCheck(cudaMalloc((void**)&d_distance, numVertex * sizeof(int)));
328 cudaCheck(cudaMalloc((void**)&d_parent, numVertex * sizeof(int)));
329 cudaCheck(cudaMalloc((void**)&d_flag, numVertex * sizeof(bool)));
330
331 // copy from cpu to gpu
332 cudaCheck(cudaMemcpy(d_prevDistance, prevDistance, numVertex * sizeof(int),
333     cudaMemcpyHostToDevice));
334 cudaCheck(cudaMemcpy(d_distance, distance, numVertex * sizeof(int),
335     cudaMemcpyHostToDevice));
336 cudaCheck(cudaMemcpy(d_parent, parent, numVertex * sizeof(int),
337     cudaMemcpyHostToDevice));
338 cudaCheck(cudaMemcpy(d_flag, flag, numVertex * sizeof(bool),
339     cudaMemcpyHostToDevice));
340
341 cout << "Calculating shortest distance" << endl;
342 int numBlocks = ((numVertex - 1) / THREADS_PER_BLOCK + 1) / 2; // use half the
343     number of required blocks
344 for (int k = 0; k < numVertex - 1; k++) { // numVertex-1 iterations
345     bellmanFordRelaxStrideFlag << <numBlocks, THREADS_PER_BLOCK >> > (numVertex,
346         vertices, indices, edges, weights, d_prevDistance, d_distance, d_parent,
347         d_flag);
348     cudaCheck(cudaGetLastError()); // check if kernel launch failed
349     cudaCheck(cudaDeviceSynchronize()); // wait for the kernel to finish
350     bellmanFordUpdateDistanceStrideFlag << <numBlocks, THREADS_PER_BLOCK >> > (
351         numVertex, d_prevDistance, d_distance, d_flag);
352     cudaCheck(cudaGetLastError()); // check if kernel launch failed
353     cudaCheck(cudaDeviceSynchronize()); // wait for the kernel to finish
354 }
355 cout << "Constructing path" << endl;
356 bellmanFordParentStride << <numBlocks, THREADS_PER_BLOCK >> > (numVertex,
357     vertices, indices, edges, weights, d_distance, d_parent);
358 cudaCheck(cudaGetLastError()); // check if kernel launch failed
359 cudaCheck(cudaDeviceSynchronize()); // wait for the kernel to finish
360
361 // copy from gpu to cpu
362 cout << "Copying results to CPU" << endl;
363 cudaCheck(cudaMemcpy(distance, d_distance, numVertex * sizeof(int),
364     cudaMemcpyDeviceToHost));
365 cudaCheck(cudaMemcpy(parent, d_parent, numVertex * sizeof(int),
366     cudaMemcpyDeviceToHost));
367
368 cudaCheck(cudaFree(d_prevDistance));
369 cudaCheck(cudaFree(d_flag));
370
371 cudaEventRecord(stop, 0);
372 cudaEventSynchronize(stop);
373 cudaEventElapsedTime(&duration, start, stop);
374 cout << "Time: " << duration << "ms" << endl;
375 }
376
377 int main(int argc, char* argv[]) {

```

```

368
369     if (argc < 6) {
370         cout << "Please provide algorithm, input file, source and validate in the
371             command line argument" << endl;
372         return 0;
373     }
374     string pathDataset("../data/"); // path to dataset
375     string algorithm(argv[1]); // algorithm 0=cpu, 1=naive, 2=stride, 3=stride with
376         flag
377     string pathGraphFile(pathDataset+string(argv[2])); // input file
378     int src = stoi(argv[3]); // source node in the range [0, n-1]
379     string validate(argv[4]); // true=compare output with cpu, false=dont
380     string outputFormat(argv[5]); // none=no output (to time the kernel), print=
381         prints path on screen, write=write output to a file in the directory named
382         output
383
384     int numVertex, numEdges;
385     vector<int> vertices, indices, edges, weights; // for CSR format of a graph
386     map<int, list< pair<int, int > > > adjacencyList; // adjacency list of a graph
387     fileToAdjacencyList(pathGraphFile, adjacencyList, numVertex, numEdges); //
388         convert input file to adjacency list
389     adjacencyListToCSR(adjacencyList, vertices, indices, edges, weights); // convert
390         adjacency list to CSR format
391
392     adjacencyList.clear(); // clear adjacency list
393
394     int* d_vertices;
395     int* d_indices;
396     int* d_edges;
397     int* d_weights;
398
399     if(algorithm != "0"){ // copy data to gpu if needed
400         cudaCheck(cudaMalloc((void**)&d_vertices, numVertex * sizeof(int)));
401         cudaCheck(cudaMalloc((void**)&d_indices, (numVertex + 1) * sizeof(int)));
402         cudaCheck(cudaMalloc((void**)&d_edges, numEdges * sizeof(int)));
403         cudaCheck(cudaMalloc((void**)&d_weights, numEdges * sizeof(int)));
404
405         cudaCheck(cudaMemcpy(d_vertices, vertices.data(), numVertex * sizeof(int),
406             cudaMemcpyHostToDevice));
407         cudaCheck(cudaMemcpy(d_indices, indices.data(), (numVertex + 1) * sizeof(int)
408             ), cudaMemcpyHostToDevice));
409         cudaCheck(cudaMemcpy(d_edges, edges.data(), numEdges * sizeof(int),
410             cudaMemcpyHostToDevice));
411         cudaCheck(cudaMemcpy(d_weights, weights.data(), numEdges * sizeof(int),
412             cudaMemcpyHostToDevice));
413     }
414
415     int* parent = (int*)malloc(numVertex * sizeof(int)); // parent of a vertex for
416         path finding
417     int* distance = (int*)malloc(numVertex * sizeof(int)); // distance from src to a
418         vertex
419
420     fill(distance, distance + numVertex, INF); // fill with INF
421     fill(parent, parent + numVertex, -1); // fill with -1
422
423     if (algorithm == "0") { // cpu version
424         runCpuBellmanFord(src, numVertex, vertices.data(), indices.data(), edges.
425             data(), weights.data(), distance, parent);
426     } else{
427         cout << "Warming up the GPU" << endl;
428         for(int x=0; x<NUM_ITERATION_WARMUP; x++){
429             warmupGpu << < (numVertex - 1) / THREADS_PER_BLOCK + 1,
430             THREADS_PER_BLOCK >> > ();
431             cudaCheck(cudaGetLastError());
432             cudaCheck(cudaDeviceSynchronize());
433         }
434     }

```

```

419     }
420     cout << "GPU is warmed up" << endl;
421
422     if (algorithm == "1") { // naive
423         runBellmanFordNaive(src, numVertex, d_vertices, d_indices, d_edges,
424             d_weights, distance, parent);
425     }
426     else if (algorithm == "2") { // stride
427         runBellmanFordStride(src, numVertex, d_vertices, d_indices, d_edges,
428             d_weights, distance, parent);
429     }
430     else if (algorithm == "3") { // stride with flag
431         runBellmanFordStrideFlag(src, numVertex, d_vertices, d_indices, d_edges,
432             d_weights, distance, parent);
433     }
434     else {
435         cout << "Illegal Algorithm" << endl;
436     }
437
438     if (validate == "true") { // validate gpu output with cpu
439         int* expParent = (int*)malloc(numVertex * sizeof(int)); // expected
440         parent
441         int* expDistance = (int*)malloc(numVertex * sizeof(int)); // expected
442         distance
443         fill(expDistance, expDistance + numVertex, INF); // fill with INF
444         fill(expParent, expParent + numVertex, -1); // fill with -1
445         runCpuBellmanFord(src, numVertex, vertices.data(), indices.data(), edges
446             .data(), weights.data(), distance, expParent); // run on cpu
447         validateDistanceSSSP(numVertex, expDistance, distance); // compare
448         distance with expDistance
449     }
450 }
451
452 // free
453 cudaCheck(cudaFree(d_vertices));
454 cudaCheck(cudaFree(d_indices));
455 cudaCheck(cudaFree(d_edges));
456 cudaCheck(cudaFree(d_weights));
457
458 if (outputFormat == "print") {
459     printPathSSSP(numVertex, distance, parent); // print paths to screen
460 }
461 else if (outputFormat == "write") { // write output to a file named bf{algorithm
462     }.txt in output directory
463     string pathOutputFile(string("../output/bf") + algorithm + string(".txt"));
464     cout << "Writing output to" << pathOutputFile << endl;
465     writeOutPathSSSP(pathOutputFile, numVertex, distance, parent);
466 }
467 else if (outputFormat == "none") { // dont write out path
468 }
469 else {
470     cout << "Illegal output format argument" << endl;
471 }
472 }

```

Listing 5: BellmanFord.cu

```

1 #include <iostream>
2
3 #include "utils.cuh"
4
5 #include "utils.h"
6
7 using namespace std;

```

```

8
9 /*****
10 SERIAL VERSION
11 *****/
12
13 // cpu dijkstra
14 void dijkstra(int src, int numVertex, int* costMatrix, int* distance, int* parent) {
15     priority_queue< pair<int, int>, vector <pair<int, int>>, greater<pair<int, int>>
16     > heap; // heap of pair<distance to node, node>
17     heap.push(make_pair(0, src)); // init heap
18     distance[src * numVertex + src] = 0;
19     while (!heap.empty()) {
20         int u = heap.top().second; // extract min
21         heap.pop();
22
23         for (int v = 0; v < numVertex ; v++) { // loop through neighbors of u
24             int weight = costMatrix[u * numVertex + v]; // cost from u to v
25
26             if (weight != INF && distance[src * numVertex + v] > distance[src *
27             numVertex + u] + weight) { // relax
28                 distance[src * numVertex + v] = distance[src * numVertex + u] +
29                 weight;
30                 parent[src * numVertex + v] = u;
31                 heap.push(make_pair(distance[src * numVertex + v], v)); // add to
32                 heap
33             }
34         }
35     }
36 }
37
38 // run cpu dijkstra for every source
39 void runCpuDijkstra(int numVertex, int* costMatrix, int* distance, int* parent) {
40     // time the algorithm
41     cudaEvent_t start, stop;
42     cudaEventCreate(&start);
43     cudaEventCreate(&stop);
44     float duration;
45     cudaEventRecord(start, 0);
46
47     for (int src = 0; src < numVertex; src++) { // for every source
48         dijkstra(src, numVertex, costMatrix, distance, parent); // call dijkstras
49     }
50
51     cudaEventRecord(stop, 0);
52     cudaEventSynchronize(stop);
53     cudaEventElapsedTime(&duration, start, stop);
54     cout << "Time: " << duration << "ms" << endl;
55 }
56
57 /*****
58 NAIVE VERSION
59 *****/
60
61 // find next node to visit
62 __device__
63 int extractMin(int numVertex, int* distance, bool* visited, int src) {
64     int minNode = -1;
65     int minDistance = INF;
66     for (int i = 0; i < numVertex; i++) {
67         if (!visited[src * numVertex + i] && distance[src * numVertex + i] <
68         minDistance) {
69             minDistance = distance[src * numVertex + i];
70             minNode = i;
71         }
72     }
73 }

```

```

68     return minNode;
69 }
70
71 __global__
72 void dijkstraNaive(int numVertex, int* h_costMatrix, bool* visited, int* distance,
73 int* parent) {
74     int src = blockIdx.x * blockDim.x + threadIdx.x; // thread src calculates
75     // shortest paths from src to every other vertex
76
77     if (src < numVertex) {
78         distance[src * numVertex + src] = 0;
79
80         for (int i = 0; i < numVertex - 1; i++) {
81             int u = extractMin(numVertex, distance, visited, src); // extract min
82             if (u == -1) { // no min node to explore
83                 break;
84             }
85             visited[src * numVertex + u] = true; // mark u as visited
86             for (int v = 0; v < numVertex; v++) { // loop through neighbors of u
87                 if (!visited[src * numVertex + v] && h_costMatrix[u * numVertex + v]
88                     != INF &&
89                     distance[src * numVertex + v] > distance[src * numVertex + u] +
90                     h_costMatrix[u * numVertex + v]) { // relax
91                     parent[src * numVertex + v] = u;
92                     distance[src * numVertex + v] = distance[src * numVertex + u] +
93                     h_costMatrix[u * numVertex + v];
94                 }
95             }
96         }
97     }
98 }
99
100 // run dijkstras on gpu
101 void runGpuDijkstra(int numVertex, int* costMatrix, bool* visited, int* distance,
102 int* parent) {
103     // time the algorithm
104     cudaEvent_t start, stop;
105     cudaEventCreate(&start);
106     cudaEventCreate(&stop);
107     float duration;
108     cudaEventRecord(start, 0);
109
110     // allocate device pointers
111     int* d_costMatrix;
112     int* d_parent;
113     int* d_distance;
114     bool* d_visited;
115
116     // allocate memory on gpu
117     cudaCheck(cudaMalloc((void**)&d_costMatrix, numVertex * numVertex * sizeof(int)));
118     cudaCheck(cudaMalloc((void**)&d_parent, numVertex * numVertex * sizeof(int)));
119     cudaCheck(cudaMalloc((void**)&d_distance, numVertex * numVertex * sizeof(int)));
120     cudaCheck(cudaMalloc((void**)&d_visited, numVertex * numVertex * sizeof(bool)));
121
122     // copy from cpu to gpu
123     cudaCheck(cudaMemcpy(d_costMatrix, costMatrix, numVertex * numVertex * sizeof(
124 int), cudaMemcpyHostToDevice));
125     cudaCheck(cudaMemcpy(d_parent, parent, numVertex * numVertex * sizeof(int),
126 cudaMemcpyHostToDevice));
127     cudaCheck(cudaMemcpy(d_distance, distance, numVertex * numVertex * sizeof(int),
128 cudaMemcpyHostToDevice));
129     cudaCheck(cudaMemcpy(d_visited, visited, numVertex * numVertex * sizeof(bool),
130 cudaMemcpyHostToDevice));

```



```

122     cout << "Kernel is executing" << endl;
123     dijkstraNaive << <(numVertex - 1) / THREADS_PER_BLOCK + 1, THREADS_PER_BLOCK >>
124     > (numVertex, d_costMatrix, d_visited, d_distance, d_parent);
125     cudaCheck(cudaGetLastError()); // check if kernel launch failed
126     cudaCheck(cudaDeviceSynchronize()); // wait for kernel to finish
127
128     // copy from cpu to cpu
129     cudaCheck(cudaMemcpy(distance, d_distance, numVertex * numVertex * sizeof(int),
130     cudaMemcpyDeviceToHost));
131     cudaCheck(cudaMemcpy(parent, d_parent, numVertex * numVertex * sizeof(int),
132     cudaMemcpyDeviceToHost));
133
134     cudaEventRecord(stop, 0);
135     cudaEventSynchronize(stop);
136     cudaEventElapsedTime(&duration, start, stop);
137     cout << "Time: " << duration << "ms" << endl;
138 }
139
140 int main(int argc, char* argv[]) {
141     if (argc < 5) {
142         cout << "Please provide an input file as a command line argument" << endl;
143         return 0;
144     }
145     string pathDataset("../data/"); // path to dataset
146     string algorithm(argv[1]); // algorithm 0=cpu, 1=naive
147     string pathGraphFile(pathDataset + string(argv[2])); // input file
148     string validate(argv[3]); // true=compare output with cpu, false=dont
149     string outputFormat(argv[4]); // none=no output (to time the kernel), print=
150     prints path on screen, write=write output to a file in the directory named
151     output
152
153     int numVertex, numEdges;
154
155     int* h_costMatrix = fileToCostMatrix(pathGraphFile, numVertex, numEdges); //
156     convert input file to adjacency list
157
158     int* h_parent = (int*)malloc(numVertex * numVertex * sizeof(int));
159     int* h_distance = (int*)malloc(numVertex * numVertex * sizeof(int));
160     bool* h_visited = (bool*)malloc(numVertex * numVertex * sizeof(bool));
161
162     fill(h_parent, h_parent + numVertex * numVertex, -1); // fill with -1
163     fill(h_distance, h_distance + numVertex * numVertex, INF); // fill with INF
164     fill(h_visited, h_visited + numVertex * numVertex, false); // fill with false
165
166     if (algorithm == "0") { // cpu version
167         runCpuDijkstra(numVertex, h_costMatrix, h_distance, h_parent);
168     }
169     else if (algorithm == "1") { // naive
170         cout << "Warming up the GPU" << endl;
171         for (int x = 0; x < NUM_ITERATION_WARMUP; x++) {
172             warmupGpu << <(numVertex - 1) / THREADS_PER_BLOCK + 1,
173             THREADS_PER_BLOCK >> > ();
174             cudaCheck(cudaGetLastError());
175             cudaCheck(cudaDeviceSynchronize());
176         }
177         cout << "GPU is warmed up" << endl;
178
179         runGpuDijkstra(numVertex, h_costMatrix, h_visited, h_distance, h_parent);
180         if (validate == "true") {
181             int* expParent = (int*)malloc(numVertex * numVertex * sizeof(int)); //
182             expected parent
183             int* expDistance = (int*)malloc(numVertex * numVertex * sizeof(int)); //
184             expected distance

```

```

177         fill(expDistance, expDistance + numVertex * numVertex, INF); // fill
        with INF
178         fill(expParent, expParent + numVertex * numVertex, -1); // fill with -1
179         runCpuDijkstra(numVertex, h_costMatrix, expDistance, expParent); // run
        on cpu
180         validateDistanceAPSP(numVertex, expDistance, h_distance); // compare
        distance with expDistance
181     }
182 }
183
184 if (outputFormat == "print") {
185     printPathAPSP(numVertex, h_distance, h_parent); // print paths to screen
186 }
187 else if (outputFormat == "write") { // write output to a file named d{algorithm
}.txt in output directory
188     string pathOutputFile(string("../output/d") + algorithm + string(".txt"));
189     cout << "Writing output to" << pathOutputFile << endl;
190     writeOutPathAPSP(pathOutputFile, numVertex, h_distance, h_parent);
191 }
192 else if (outputFormat == "none") { // dont write out path
193 }
194 }
195 else {
196     cout << "Illegal output format argument" << endl;
197 }
198 }

```

Listing 6: Dijkstra.cu

```

1 #include "cuda_runtime.h"
2 #include "device_launch_parameters.h"
3
4 #include "utils.cuh"
5
6 #include <iostream>
7
8 #include "utils.h"
9
10 using namespace std;
11
12 #define TILE_DIM 32
13
14 /*****
15 SERIAL VERSION
16 *****/
17 void runCpuFloydWarshall(int numVertex, int* distance, int* parent) {
18     cudaEvent_t start, stop;
19     cudaEventCreate(&start);
20     cudaEventCreate(&stop);
21     float duration;
22     cudaEventRecord(start, 0);
23
24     cout << "running the algorithm on CPU" << endl;
25     for (int k = 0; k < numVertex; k++) { // choose an intermediate node k
26         for (int i = 0; i < numVertex; i++) { // choose a start node i
27             for (int j = 0; j < numVertex; j++) { // loop through its neighbors
28                 int itoj = i * numVertex + j; // index for i->j
29                 int itok = i * numVertex + k; // index for i->k
30                 int ktoj = k * numVertex + j; // index for k->j
31
32                 // relax i->j using node k
33                 if (distance[itok] != INF && distance[ktoj] != INF && distance[itoj]
> distance[itok] + distance[ktoj]) {
34                     parent[itoj] = k;
35                     distance[itoj] = distance[itok] + distance[ktoj];

```

```

36     }
37     }
38     }
39 }
40
41 // time
42 cudaEventRecord(stop, 0);
43 cudaEventSynchronize(stop);
44 cudaEventElapsedTime(&duration, start, stop);
45 cout << "Time: " << duration << "ms" << endl;
46 }
47
48 /*****
49 SUPER NAIVE VERSION
50 *****/
51 // one thread for each edge
52 __global__
53 void floydWarshallSuperNaive(int numVertex, int k, int* distance, int* parent) {
54     int i = blockIdx.y * blockDim.y + threadIdx.y; // choose a start node i
55     int j = blockIdx.x * blockDim.x + threadIdx.x; // choose a neighbor of i
56     if (i < numVertex && j < numVertex) {
57         int itoj = i * numVertex + j; // index for i->j
58         int itok = i * numVertex + k; // index for i->k
59         int ktoj = k * numVertex + j; // index for k->j
60
61         // relax i->j using node k
62         if (distance[itok] != INF && distance[ktoj] != INF && distance[itoj] >
63             distance[itok] + distance[ktoj]) {
64             parent[itoj] = k;
65             distance[itoj] = distance[itok] + distance[ktoj];
66         }
67     }
68 }
69
70 // runs super naive on gpu
71 void runFloydWarshallSuperNaive(int numVertex, int* distance, int* parent) {
72     cudaEvent_t start, stop;
73     cudaEventCreate(&start);
74     cudaEventCreate(&stop);
75     float duration;
76
77     cudaEventRecord(start, 0);
78
79     int* d_distance;
80     int* d_parent;
81
82     // allocate memory on GPU and copy data from CPU to GPU
83     cout << "allocating data on GPU" << endl;
84     cudaCheck(cudaMalloc((void**)&d_distance, numVertex * numVertex * sizeof(int)));
85     cudaCheck(cudaMalloc((void**)&d_parent, numVertex * numVertex * sizeof(int)));
86
87     cout << "copying data to GPU" << endl;
88     cudaCheck(cudaMemcpy(d_distance, distance, numVertex * numVertex * sizeof(int),
89         cudaMemcpyHostToDevice));
90     cudaCheck(cudaMemcpy(d_parent, parent, numVertex * numVertex * sizeof(int),
91         cudaMemcpyHostToDevice));
92
93     dim3 dimGrid((numVertex - 1) / TILE_DIM + 1, (numVertex - 1) / TILE_DIM + 1);
94     dim3 dimBlock(TILE_DIM, TILE_DIM);
95     // run kernel
96     cout << "Kernel is executing" << endl;
97     for (int k = 0; k < numVertex; k++) {
98         floydWarshallSuperNaive << <dimGrid, dimBlock >> > (numVertex, k, d_distance,
99             d_parent);
100         cudaCheck(cudaGetLastError());

```

```

97         cudaCheck(cudaDeviceSynchronize());
98     }
99
100     // copy results to CPU
101     cout << "copying results to CPU" << endl;
102     cudaCheck(cudaMemcpy(distance, d_distance, numVertex * numVertex * sizeof(int),
103         cudaMemcpyDeviceToHost));
104     cudaCheck(cudaMemcpy(parent, d_parent, numVertex * numVertex * sizeof(int),
105         cudaMemcpyDeviceToHost));
106
107     cudaEventRecord(stop, 0);
108     cudaEventSynchronize(stop);
109     cudaEventElapsedTime(&duration, start, stop);
110     cout << "Time: " << duration << "ms" << endl;
111 }
112
113 /*****
114 SUPER NAIVE SHARED VERSION
115 *****/
116 // one thread per edge but with shared memory
117 __global__
118 void floydWarshallSuperNaiveShared(int numVertex, int k, int* distance, int* parent)
119 {
120     int i = blockIdx.y; // choose a start node i
121     int j = blockIdx.x * blockDim.x + threadIdx.x; // choose a neighbor j of i
122
123     if (j < numVertex) {
124         int itoj = numVertex * i + j; // index for i->j
125         int itok = numVertex * i + k; // index for i->k
126         int ktok = numVertex * k + j; // index for k->j
127
128         __shared__ int dist_itok; // shared variable to store i->k
129         if (threadIdx.x == 0) {
130             dist_itok = distance[itok];
131         }
132         __syncthreads();
133
134         // relax i->j using node k
135         if (dist_itok != INF && distance[ktok] != INF && distance[itoj] > dist_itok
136             + distance[ktok]) {
137             distance[itoj] = dist_itok + distance[ktok];
138             parent[itoj] = k;
139         }
140     }
141 }
142
143 // runs super naive shared on gpu
144 void runFloydWarshallSuperNaiveShared(int numVertex, int* distance, int* parent) {
145     cudaEvent_t start, stop;
146     cudaEventCreate(&start);
147     cudaEventCreate(&stop);
148     float duration;
149
150     cudaEventRecord(start, 0);
151
152     int* d_distance;
153     int* d_parent;
154
155     // allocate memory on GPU and copy data from CPU to GPU
156     cout << "allocating data on GPU" << endl;
157     cudaCheck(cudaMalloc((void**)&d_distance, numVertex * numVertex * sizeof(int)));
158     cudaCheck(cudaMalloc((void**)&d_parent, numVertex * numVertex * sizeof(int)));
159
160     cout << "copying data to GPU" << endl;

```

```

157     cudaCheck(cudaMemcpy(d_distance, distance, numVertex * numVertex * sizeof(int),
158     cudaMemcpyHostToDevice));
159     cudaCheck(cudaMemcpy(d_parent, parent, numVertex * numVertex * sizeof(int),
160     cudaMemcpyHostToDevice));
161
162     dim3 dimGrid((numVertex - 1) / THREADS_PER_BLOCK + 1, numVertex);
163
164     // run kernel
165     cout << "Kernel is executing" << endl;
166     for (int k = 0; k < numVertex; k++) {
167         floydWarshallSuperNaiveShared << <dimGrid, THREADS_PER_BLOCK >> > (numVertex
168         , k, d_distance, d_parent);
169         cudaCheck(cudaGetLastError());
170         cudaCheck(cudaDeviceSynchronize());
171     }
172
173     // copy results to CPU
174     cout << "copying results to CPU" << endl;
175     cudaCheck(cudaMemcpy(distance, d_distance, numVertex * numVertex * sizeof(int),
176     cudaMemcpyDeviceToHost));
177     cudaCheck(cudaMemcpy(parent, d_parent, numVertex * numVertex * sizeof(int),
178     cudaMemcpyDeviceToHost));
179
180     cudaEventRecord(stop, 0);
181     cudaEventSynchronize(stop);
182     cudaEventElapsedTime(&duration, start, stop);
183     cout << "Time: " << duration << "ms" << endl;
184 }
185
186 /*****
187 NAIVE VERSION
188 *****/
189 // one thread per vertex
190 __global__
191 void floydWarshallNaive(int numVertex, int k, int* distance, int* parent) {
192     int i = blockIdx.x * blockDim.x + threadIdx.x; // choose a start node i
193     if (i < numVertex) {
194         for (int j = 0; j < numVertex; j++) { // loop through its neighbors
195             int itoj = i * numVertex + j; // index for i->j
196             int itok = i * numVertex + k; // index for i->k
197             int ktok = k * numVertex + j; // index for k->j
198             // relax i->j using node k
199             if (distance[itok] != INF && distance[ktok] != INF && distance[itoj] >
200             distance[itok] + distance[ktok]) {
201                 parent[itoj] = k;
202                 distance[itoj] = distance[itok] + distance[ktok];
203             }
204         }
205     }
206 }
207
208 // runs naive on gpu
209 void runFloydWarshallNaive(int numVertex, int* distance, int* parent) {
210     cudaEvent_t start, stop;
211     cudaEventCreate(&start);
212     cudaEventCreate(&stop);
213     float duration;
214
215     cudaEventRecord(start, 0);
216
217     int* d_distance;
218     int* d_parent;
219
220     // allocate memory on GPU and copy data from CPU to GPU
221     cout << "allocating data on GPU" << endl;

```

```

216     cudaCheck(cudaMalloc((void**)&d_distance, numVertex * numVertex * sizeof(int)));
217     cudaCheck(cudaMalloc((void**)&d_parent, numVertex * numVertex * sizeof(int)));
218
219     cout << "copying data to GPU" << endl;
220     cudaCheck(cudaMemcpy(d_distance, distance, numVertex * numVertex * sizeof(int),
221         cudaMemcpyHostToDevice));
222     cudaCheck(cudaMemcpy(d_parent, parent, numVertex * numVertex * sizeof(int),
223         cudaMemcpyHostToDevice));
224
225     // run kernel
226     cout << "Kernel is executing" << endl;
227     for (int k = 0; k < numVertex; k++) {
228         floydWarshallNaive << <(numVertex - 1) / THREADS_PER_BLOCK + 1,
229         THREADS_PER_BLOCK >> > (numVertex, k, d_distance, d_parent);
230         cudaCheck(cudaGetLastError());
231         cudaCheck(cudaDeviceSynchronize());
232     }
233
234     // copy results to CPU
235     cout << "copying results to CPU" << endl;
236     cudaCheck(cudaMemcpy(distance, d_distance, numVertex * numVertex * sizeof(int),
237         cudaMemcpyDeviceToHost));
238     cudaCheck(cudaMemcpy(parent, d_parent, numVertex * numVertex * sizeof(int),
239         cudaMemcpyDeviceToHost));
240
241     cudaEventRecord(stop, 0);
242     cudaEventSynchronize(stop);
243     cudaEventElapsedTime(&duration, start, stop);
244     cout << "Time: " << duration << "ms" << endl;
245 }
246
247 /*****
248 TILED VERSION
249 *****/
250 // tiled with global memory
251
252 // phase 1
253 __global__
254 void floydWarshallTiledPhase1(int numVertex, int primary_tile_number, int* distance,
255     int* parent) {
256     int tx = threadIdx.x;
257     int ty = threadIdx.y;
258
259     int i = primary_tile_number * blockDim.y + threadIdx.y; // node i
260     int j = primary_tile_number * blockDim.x + threadIdx.x; // node j
261     if(i < numVertex && j < numVertex){
262         int itoj = i * numVertex + j; // index for i->j
263         for (int k = 0; k < TILE_DIM; k++) { // run floyd warshall in the primary
264             tile
265             if (j-tx+k < numVertex && i-ty+k < numVertex &&
266                 distance[itoj - tx + k] != INF && distance[itoj - ty * numVertex + k
267                 * numVertex] != INF &&
268                 distance[itoj] > distance[itoj - tx + k] + distance[itoj - ty *
269                 numVertex + k * numVertex]) {
270                 distance[itoj] = distance[itoj - tx + k] + distance[itoj - ty *
271                 numVertex + k * numVertex];
272                 parent[itoj] = TILE_DIM * primary_tile_number + k;
273             }
274             // __syncthreads();
275         }
276     }
277 }

```

```

271 // phase 2
272 __global__
273 void floydWarshallTiledPhase2(int numVertex, int primary_tile_number, int* distance,
    int* parent) {
274     // exclude primary tile
275     if (blockIdx.x == primary_tile_number) {
276         return;
277     }
278     int tx = threadIdx.x;
279     int ty = threadIdx.y;
280
281     int i, j;
282
283     // 1st row of blocks for row
284     if (blockIdx.y == 0) {
285         i = primary_tile_number * blockDim.y + threadIdx.y;
286         j = blockIdx.x * blockDim.x + threadIdx.x;
287         if (i < numVertex && j < numVertex) {
288             int itoj = i * numVertex + j; // index for i->j
289             // relax edges in current tile using distance[i][k] from primary tile
290             for (int k = 0; k < TILE_DIM; k++) {
291                 if (j-tx+k-blockIdx.x * blockDim.x + primary_tile_number * blockDim.
x < numVertex && i-ty+k < numVertex &&
292                     distance[itoj - tx + k - blockIdx.x * blockDim.x +
primary_tile_number * blockDim.x] != INF &&
293                     distance[itoj - ty * numVertex + k * numVertex] != INF &&
294                     distance[itoj] > distance[itoj - tx + k - blockIdx.x * blockDim.
x + primary_tile_number * blockDim.x]
+ distance[itoj - ty * numVertex + k * numVertex]) {
295
296
297                     distance[itoj] = distance[itoj - tx + k - blockIdx.x * blockDim.
x + primary_tile_number * blockDim.x] + distance[itoj - ty * numVertex + k *
numVertex];
298                     parent[itoj] = TILE_DIM * primary_tile_number + k;
299                 }
300                 // __syncthreads();
301             }
302         }
303     }
304
305     // 2nd row of blocks for columns
306     if (blockIdx.y == 1) {
307         i = blockIdx.x * blockDim.y + threadIdx.y;
308         j = primary_tile_number * blockDim.x + threadIdx.x;
309         if (i < numVertex && j < numVertex) {
310             int itoj = i * numVertex + j; // index for i->j
311             // relax edges in current tile using distance[i][k] from primary tile
312             for (int k = 0; k < TILE_DIM; k++) {
313                 if (j-tx+k < numVertex && i-(ty-k)- (blockIdx.x -
primary_tile_number) * blockDim.x < numVertex &&
314                     distance[itoj - tx + k] != INF &&
315                     distance[itoj - (ty - k) * numVertex - (blockIdx.x -
primary_tile_number) * blockDim.x * numVertex] != INF &&
316                     distance[itoj] > distance[itoj - tx + k]
+ distance[itoj - (ty - k) * numVertex - (blockIdx.x -
primary_tile_number) * blockDim.x * numVertex]) {
317
318
319                     distance[itoj] = distance[itoj - tx + k] + distance[itoj - ty *
numVertex + k * numVertex - (blockIdx.x - primary_tile_number) * blockDim.x *
numVertex];
320                     parent[itoj] = TILE_DIM * primary_tile_number + k;
321                 }
322                 // __syncthreads();
323             }
324         }
325     }

```

```

325     }
326 }
327
328 // phase 3
329 __global__
330 void floydWarshallTiledPhase3(int numVertex, int primary_tile_number, int* distance,
    int* parent) {
331     // exclude primary tile, primary row and primary column
332     if (blockIdx.x == primary_tile_number || blockIdx.y == primary_tile_number) {
333         return;
334     }
335     int tx = threadIdx.x;
336     int ty = threadIdx.y;
337     int i = blockIdx.y * blockDim.y + threadIdx.y;
338     int j = blockIdx.x * blockDim.x + threadIdx.x;
339     if (i < numVertex && j < numVertex) {
340         int itoj = i * numVertex + j; // index for i->j
341         // relax edges in current tile using distance[i][k] from primary column and
342         // distance[k][j] from primary row
343         for (int k = 0; k < TILE_DIM; k++) {
344             if (j-tx+k - blockIdx.x * blockDim.x + primary_tile_number * blockDim.x
345                 < numVertex &&
346                 i-ty+k - (blockIdx.y - primary_tile_number) * blockDim.y < numVertex
347                 &&
348                 distance[itoj - tx + k - blockIdx.x * blockDim.x +
349                     primary_tile_number * blockDim.x] != INF &&
350                 distance[itoj - ty * numVertex + k * numVertex - (blockIdx.y -
351                     primary_tile_number) * blockDim.y * numVertex] != INF &&
352                 distance[itoj] > distance[itoj - (tx - k) - (blockIdx.x -
353                     primary_tile_number) * blockDim.x]
354                     + distance[itoj - (ty - k) * numVertex - (blockIdx.y -
355                     primary_tile_number) * blockDim.y * numVertex]) {
356
357                 distance[itoj] = distance[itoj - tx + k - blockIdx.x * blockDim.x +
358                     primary_tile_number * blockDim.x] + distance[itoj - ty * numVertex + k *
359                     numVertex - (blockIdx.y - primary_tile_number) * blockDim.y * numVertex];
360                 parent[itoj] = TILE_DIM * primary_tile_number + k;
361             }
362         }
363     }
364 }
365
366 // runs tiled version on gpu
367 void runFloydWarshallTiled(int numVertex, int* distance, int* parent) {
368     cudaEvent_t start, stop;
369     cudaEventCreate(&start);
370     cudaEventCreate(&stop);
371     float duration;
372
373     cudaEventRecord(start, 0);
374
375     int* d_distance;
376     int* d_parent;
377
378     // allocate memory on GPU and copy data from CPU to GPU
379     cudaCheck(cudaMalloc((void**)&d_distance, numVertex * numVertex * sizeof(int)));
380     cudaCheck(cudaMalloc((void**)&d_parent, numVertex * numVertex * sizeof(int)));
381
382     cout << "copying data to GPU" << endl;
383     cudaCheck(cudaMemcpy(d_distance, distance, numVertex * numVertex * sizeof(int),
384         cudaMemcpyHostToDevice));
385     cudaCheck(cudaMemcpy(d_parent, parent, numVertex * numVertex * sizeof(int),
386         cudaMemcpyHostToDevice));
387
388     int numDiagonalTiles = (numVertex - 1) / TILE_DIM + 1;

```



```

378
379     dim3 dimGridPhase1(1, 1), dimGridPhase2(numDiagonalTiles, 2), dimGridPhase3(
        numDiagonalTiles, numDiagonalTiles);
380     dim3 dimBlock(TILE_DIM, TILE_DIM);
381
382     cout << "Kernel is executing" << endl;
383     for (int k = 0; k < numDiagonalTiles; k++) {
384         floydWarshallTiledPhase1 << < dimGridPhase1, dimBlock >> > (numVertex, k,
            d_distance, d_parent);
385         cudaCheck(cudaGetLastError());
386         cudaCheck(cudaDeviceSynchronize());
387         floydWarshallTiledPhase2 << < dimGridPhase2, dimBlock >> > (numVertex, k,
            d_distance, d_parent);
388         cudaCheck(cudaGetLastError());
389         cudaCheck(cudaDeviceSynchronize());
390         floydWarshallTiledPhase3 << < dimGridPhase3, dimBlock >> > (numVertex, k,
            d_distance, d_parent);
391         cudaCheck(cudaGetLastError());
392         cudaCheck(cudaDeviceSynchronize());
393     }
394
395     // copy results to CPU
396     cout << "copying results to CPU" << endl;
397     cudaCheck(cudaMemcpy(distance, d_distance, numVertex * numVertex * sizeof(int),
        cudaMemcpyDeviceToHost));
398     cudaCheck(cudaMemcpy(parent, d_parent, numVertex * numVertex * sizeof(int),
        cudaMemcpyDeviceToHost));
399
400     cudaEventRecord(stop, 0);
401     cudaEventSynchronize(stop);
402     cudaEventElapsedTime(&duration, start, stop);
403     cout << "Time: " << duration << "ms" << endl;
404 }
405
406 /*****
407 TILED WIH SHARED MEMORY VERSION
408 *****/
409
410 // phase 1
411 __global__
412 void floydWarshallTiledSharedPhase1(int numVertex, int primary_tile_number, int*
    distance, int* parent) {
413     __shared__ int s_distance[TILE_DIM][TILE_DIM]; // primary tile
414
415     int tx = threadIdx.x;
416     int ty = threadIdx.y;
417
418     int i = TILE_DIM * primary_tile_number + ty;
419     int j = TILE_DIM * primary_tile_number + tx;
420     int itoj = i * numVertex + j;
421
422     int shortestParent;
423     if (i < numVertex && j < numVertex) {
424         s_distance[ty][tx] = distance[itoj];
425         shortestParent = parent[itoj];
426     } else {
427         s_distance[ty][tx] = INF;
428         shortestParent = -1;
429     }
430     __syncthreads();
431
432     #pragma unroll
433     for (int k = 0; k < TILE_DIM; k++) { // run floyd warshall in primary tile
434         __syncthreads();
435         if (s_distance[ty][k] != INF &&

```

```

436         s_distance[k][tx] != INF &&
437         s_distance[ty][tx] > s_distance[ty][k] + s_distance[k][tx]) {
438
439             s_distance[ty][tx] = s_distance[ty][k] + s_distance[k][tx];
440             shortestParent = TILE_DIM * primary_tile_number + k;
441         }
442         __syncthreads();
443     }
444     if (i < numVertex && j < numVertex) {
445         distance[itoj] = s_distance[ty][tx];
446         parent[itoj] = shortestParent;
447     }
448 }
449
450 // phase 2
451 __global__
452 void floydWarshallTiledSharedPhase2(int numVertex, int primary_tile_number, int*
    distance, int* parent) {
453     if (blockIdx.x == primary_tile_number) { // exclude primary tile
454         return;
455     }
456     __shared__ int s_distancePrimaryTile[TILE_DIM][TILE_DIM]; // primary tile
457     __shared__ int s_distanceCurrentTile[TILE_DIM][TILE_DIM]; // current tile
458
459     int i = TILE_DIM * primary_tile_number + threadIdx.y;
460     int j = TILE_DIM * primary_tile_number + threadIdx.x;
461
462     int idxPrimaryTile = i * numVertex + j;
463
464     if (i < numVertex && j < numVertex) {
465         s_distancePrimaryTile[threadIdx.y][threadIdx.x] = distance[idxPrimaryTile];
466     }
467     else {
468         s_distancePrimaryTile[threadIdx.y][threadIdx.x] = INF;
469     }
470     __syncthreads();
471
472     int idxCurrentTile;
473     int shortestDistance;
474     int shortestParent;
475
476     if (blockIdx.y == 0) { // 1st row of blocks for rows
477         i = TILE_DIM * primary_tile_number + threadIdx.y;
478         j = TILE_DIM * blockIdx.x + threadIdx.x;
479         idxCurrentTile = i * numVertex + j;
480
481         if (i < numVertex && j < numVertex) {
482             s_distanceCurrentTile[threadIdx.y][threadIdx.x] = distance[
idxCurrentTile];
483             shortestParent = parent[idxCurrentTile];
484         }
485         else {
486             s_distanceCurrentTile[threadIdx.y][threadIdx.x] = INF;
487             shortestParent = -1;
488         }
489         __syncthreads();
490
491         shortestDistance = s_distanceCurrentTile[threadIdx.y][threadIdx.x];
492
493         // relax edges in current tile using distance[i][k] from primary tile
494         #pragma unroll
495         for (int k = 0; k < TILE_DIM; k++) {
496             int newDistance = s_distancePrimaryTile[threadIdx.y][k] +
s_distanceCurrentTile[k][threadIdx.x];
497             // __syncthreads();

```

```

498         if (s_distancePrimaryTile[threadIdx.y][k] != INF &&
499             s_distanceCurrentTile[k][threadIdx.x] != INF &&
500             newDistance < shortestDistance) {
501
502             shortestParent = TILE_DIM * primary_tile_number + k;
503             shortestDistance = newDistance;
504         }
505         __syncthreads();
506     }
507 } else { // 2nd row of blocks for column
508     i = TILE_DIM * blockIdx.x + threadIdx.y;
509     j = TILE_DIM * primary_tile_number + threadIdx.x;
510     idxCurrentTile = i * numVertex + j;
511
512     if (i < numVertex && j < numVertex) {
513         s_distanceCurrentTile[threadIdx.y][threadIdx.x] = distance[
514 idxCurrentTile];
515         shortestParent = parent[idxCurrentTile];
516     }
517     else {
518         s_distanceCurrentTile[threadIdx.y][threadIdx.x] = INF;
519         shortestParent = -1;
520     }
521     __syncthreads();
522     shortestDistance = s_distanceCurrentTile[threadIdx.y][threadIdx.x];
523
524     // relax edges in current tile using distance[i][k] from primary tile
525     #pragma unroll
526     for (int k = 0; k < TILE_DIM; k++) {
527         int newDistance = s_distanceCurrentTile[threadIdx.y][k] +
528 s_distancePrimaryTile[k][threadIdx.x];
529         // __syncthreads();
530         if (s_distancePrimaryTile[k][threadIdx.x] != INF &&
531             s_distanceCurrentTile[threadIdx.y][k] != INF &&
532             newDistance < shortestDistance) {
533
534             shortestParent = TILE_DIM * primary_tile_number + k;
535             shortestDistance = newDistance;
536         }
537         __syncthreads();
538     }
539 }
540 if (i < numVertex && j < numVertex) {
541     distance[idxCurrentTile] = shortestDistance;
542     parent[idxCurrentTile] = shortestParent;
543 }
544 }
545 // phase 3
546 __global__
547 void floydWarshallTiledSharedPhase3(int numVertex, int primary_tile_number, int*
548 distance, int* parent) {
549     // exclude primary tile, primary row and primary column
550     if (blockIdx.x == primary_tile_number || blockIdx.y == primary_tile_number) {
551         return;
552     }
553
554     __shared__ int s_distancePrimaryRow[TILE_DIM][TILE_DIM]; // primary row tile
555     __shared__ int s_distancePrimaryCol[TILE_DIM][TILE_DIM]; // primary column tile
556     __shared__ int s_distanceCurrentTile[TILE_DIM][TILE_DIM]; // current tile
557
558     int i, j;
559
560     i = TILE_DIM * primary_tile_number + threadIdx.y;
561     j = TILE_DIM * blockIdx.x + threadIdx.x;

```

```

560     if (i < numVertex && j < numVertex) {
561         s_distancePrimaryRow[threadIdx.y][threadIdx.x] = distance[i * numVertex + j
562     ];
563     }
564     else {
565         s_distancePrimaryRow[threadIdx.y][threadIdx.x] = INF;
566     }
567
568     i = TILE_DIM * blockIdx.y + threadIdx.y;
569     j = TILE_DIM * primary_tile_number + threadIdx.x;
570     if (i < numVertex && j < numVertex) {
571         s_distancePrimaryCol[threadIdx.y][threadIdx.x] = distance[i * numVertex + j
572     ];
573     }
574     else {
575         s_distancePrimaryCol[threadIdx.y][threadIdx.x] = INF;
576     }
577
578     i = TILE_DIM * blockIdx.y + threadIdx.y;
579     j = TILE_DIM * blockIdx.x + threadIdx.x;
580     int shortestParent;
581     if (i < numVertex && j < numVertex) {
582         s_distanceCurrentTile[threadIdx.y][threadIdx.x] = distance[i * numVertex + j
583     ];
584         shortestParent = parent[i * numVertex + j];
585     }
586     else {
587         s_distanceCurrentTile[threadIdx.y][threadIdx.x] = INF;
588         shortestParent = -1;
589     }
590
591     __syncthreads();
592
593     int shortestDist = s_distanceCurrentTile[threadIdx.y][threadIdx.x];
594     // relax edges in current tile using distance[i][k] from primary column tile and
595     // distance[k][j] from primary row tile
596     #pragma unroll
597     for (int k = 0; k < TILE_DIM; k++) {
598         int newDistance = s_distancePrimaryCol[threadIdx.y][k] +
599         s_distancePrimaryRow[k][threadIdx.x];
600         if (s_distancePrimaryCol[threadIdx.y][k] != INF &&
601             s_distancePrimaryRow[k][threadIdx.x] != INF &&
602             newDistance < shortestDist) {
603             shortestParent = TILE_DIM * primary_tile_number + k;
604             shortestDist = newDistance;
605         }
606     }
607     // __syncthreads();
608     if(i<numVertex && j<numVertex){ // write the tile to global memory
609         distance[i * numVertex + j] = shortestDist;
610         parent[i * numVertex + j] = shortestParent;
611     }
612 }
613
614 // runs tiled with shared memory on gpu
615 void runFloydWarshallTiledShared(int numVertex, int* distance, int* parent) {
616     cudaEvent_t start, stop;
617     cudaEventCreate(&start);
618     cudaEventCreate(&stop);
619     float duration;
620
621     cudaEventRecord(start, 0);
622
623     int* d_distance;

```

```

620 int* d_parent;
621
622 // allocate memory on GPU and copy data from CPU to GPU
623 cout << "allocating data on GPU" << endl;
624 cudaCheck(cudaMalloc((void**)&d_distance, numVertex * numVertex * sizeof(int)));
625 cudaCheck(cudaMalloc((void**)&d_parent, numVertex * numVertex * sizeof(int)));
626
627 cout << "copying data to GPU" << endl;
628 cudaCheck(cudaMemcpy(d_distance, distance, numVertex * numVertex * sizeof(int),
629 cudaMemcpyHostToDevice));
630 cudaCheck(cudaMemcpy(d_parent, parent, numVertex * numVertex * sizeof(int),
631 cudaMemcpyHostToDevice));
632
633 int numDiagonalTiles = (numVertex - 1) / TILE_DIM + 1;
634
635 dim3 dimGridPhase1(1, 1), dimGridPhase2(numDiagonalTiles, 2), dimGridPhase3(
636 numDiagonalTiles, numDiagonalTiles);
637 dim3 dimBlock(TILE_DIM, TILE_DIM);
638
639 cout << "Kernel is executing" << endl;
640 for (int k = 0; k < numDiagonalTiles; k++) {
641     floydWarshallTiledSharedPhase1 <<< dimGridPhase1, dimBlock >>> (numVertex
642 , k, d_distance, d_parent);
643     cudaCheck(cudaGetLastError());
644     cudaCheck(cudaDeviceSynchronize());
645     floydWarshallTiledSharedPhase2 <<< dimGridPhase2, dimBlock >>> (numVertex
646 , k, d_distance, d_parent);
647     cudaCheck(cudaGetLastError());
648     cudaCheck(cudaDeviceSynchronize());
649     floydWarshallTiledSharedPhase3 <<< dimGridPhase3, dimBlock >>> (numVertex
650 , k, d_distance, d_parent);
651     cudaCheck(cudaGetLastError());
652     cudaCheck(cudaDeviceSynchronize());
653 }
654
655 // copy results to CPU
656 cout << "copying results to CPU" << endl;
657 cudaCheck(cudaMemcpy(distance, d_distance, numVertex * numVertex * sizeof(int),
658 cudaMemcpyDeviceToHost));
659 cudaCheck(cudaMemcpy(parent, d_parent, numVertex * numVertex * sizeof(int),
660 cudaMemcpyDeviceToHost));
661
662 cudaEventRecord(stop, 0);
663 cudaEventSynchronize(stop);
664 cudaEventElapsedTime(&duration, start, stop);
665 cout << "Time: " << duration << "ms" << endl;
666 }
667
668 int main(int argc, char* argv[]) {
669
670     if (argc < 5) {
671         cout << "Please provide proper command line arguments" << endl;
672         return 0;
673     }
674     string pathDataset("../data/");
675     string algorithm(argv[1]);
676     string pathGraphFile(pathDataset+string(argv[2]));
677     string validate(argv[3]);
678     string outputFormat(argv[4]);
679
680     int numVertex, numEdges;
681     int* costMatrix = fileToCostMatrix(pathGraphFile, numVertex, numEdges);
682
683     int* parent = (int*)malloc(numVertex * numVertex * sizeof(int));
684     int* distance = (int*)malloc(numVertex * numVertex * sizeof(int));

```

```

677     APSPInitDistanceParent(numVertex, costMatrix, distance, parent);
678
679     if (algorithm == "0") {
680         runCpuFloydWarshall(numVertex, distance, parent);
681     } else {
682         cout << "Warming up the GPU" << endl;
683         for (int x = 0; x < NUM_ITERATION_WARMUP; x++) {
684             warmupGpu << < (numVertex - 1) / THREADS_PER_BLOCK + 1,
685             THREADS_PER_BLOCK >> > ();
686             cudaCheck(cudaGetLastError());
687             cudaCheck(cudaDeviceSynchronize());
688         }
689         cout << "GPU is warmed up" << endl;
690         if (algorithm == "1") {
691             runFloydWarshallSuperNaive(numVertex, distance, parent);
692         } else if (algorithm == "2") {
693             runFloydWarshallNaive(numVertex, distance, parent);
694         } else if (algorithm == "3") {
695             runFloydWarshallSuperNaiveShared(numVertex, distance, parent);
696         } else if (algorithm == "4") {
697             runFloydWarshallTiled(numVertex, distance, parent);
698         } else if (algorithm == "5") {
699             runFloydWarshallTiledShared(numVertex, distance, parent);
700         }
701
702         if (validate == "true") {
703             int* exp_parent = (int*)malloc(numVertex * numVertex * sizeof(int));
704             int* exp_distance = (int*)malloc(numVertex * numVertex * sizeof(int));
705             if (exp_parent == NULL || exp_distance == NULL) {
706                 cout << "Malloc failed" << endl;
707                 return 0;
708             }
709             APSPInitDistanceParent(numVertex, costMatrix, exp_distance, exp_parent);
710             runCpuFloydWarshall(numVertex, exp_distance, exp_parent);
711             validateDistanceAPSP(numVertex, exp_distance, distance);
712         }
713     }
714     //
715     if (outputFormat == "print") {
716         printPathAPSP(numVertex, distance, parent);
717     } else if (outputFormat == "write") {
718         string pathOutputFile(string("../output/fw") + algorithm + string(".txt"));
719         cout << "Writing output to" << pathOutputFile << endl;
720         writeOutPathAPSP(pathOutputFile, numVertex, distance, parent);
721     } else if (outputFormat == "none") {
722     } else {
723         cout << "Illegal output format argument" << endl;
724     }
725 }
726 }

```

Listing 7: FloydWarshall.cu

```

1 import sys
2 import random
3
4 def parseDIMACS(fileName):
5     file = open(fileName, "r")
6     lines = file.readlines()
7
8     for i, line in enumerate(lines):
9         if i==4:
10             tokens = line.split()
11             numVertex, numEdges = tokens[2], tokens[3]

```

```

12     lines[i] = numVertex + " " + numEdges + "\n"
13     elif i>=7:
14         _, src, dest, cost = line.split()
15         lines[i] = str(int(src)-1) + " " + str(int(dest)-1) + " " + cost + "\n"
16     else:
17         lines[i] = ''
18
19     file = open(fileName, "w")
20     file.writelines(lines)
21     file.close()
22
23 def addWeights(fileName, weightMin=1, weightMax=100):
24     file = open(fileName, "r")
25     lines = file.readlines()
26
27     for i, line in enumerate(lines):
28         if i==2:
29             tokens = line.split()
30             numVertex, numEdges = tokens[2], tokens[4]
31             lines[i] = numVertex + " " + numEdges + "\n"
32         elif i>=4:
33             src, dest = line.split()
34             lines[i] = src + " " + dest + " " + str(random.randint(weightMin, weightMax))
35             + "\n"
36         else:
37             lines[i] = ''
38
39     file = open(fileName, "w")
40     file.writelines(lines)
41     file.close()
42
43 def replaceWeights(fileName, weightMin=1, weightMax=100):
44     file = open(fileName, "r")
45     lines = file.readlines()
46
47     for i, line in enumerate(lines):
48         if i:
49             src, dest, cost = line.split()
50             lines[i] = src + " " + dest + " " + str(random.randint(weightMin, weightMax))
51             + "\n"
52
53     file = open(fileName, "w")
54     file.writelines(lines)
55     file.close()
56
57 def createRandomGraph(numVertex, fileName):
58     lines = [str(numVertex)+" "+str(numVertex*numVertex-numVertex)]
59     for i in range(numVertex):
60         for j in range(numVertex):
61             if i != j:
62                 line = str(i) + " " + str(j) + " " + str(random.randint(1, 100)) + "\n"
63                 lines.append(line)
64     file = open(fileName, "w")
65     file.writelines(lines)
66     file.close()
67
68 if __name__ == "__main__":
69     _, action, *rest = sys.argv
70     if action == 'parse':
71         parseDIMACS(rest[0])
72     elif action == 'random':
73         createRandomGraph(int(rest[0]), rest[1])
74     else:
75         fileName, weightMin, weightMax = rest

```

```
75 if action == 'add':  
76     addWeights(fileName, int(weightMin), int(weightMax))  
77 elif action == 'replace':  
78     replaceWeights(fileName, int(weightMin), int(weightMax))
```

Listing 8: utils.py