
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.

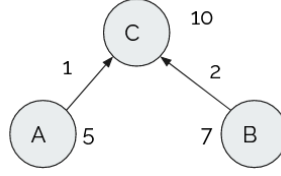


Figure 1: Bellman-Ford: Need for atomic minimum

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

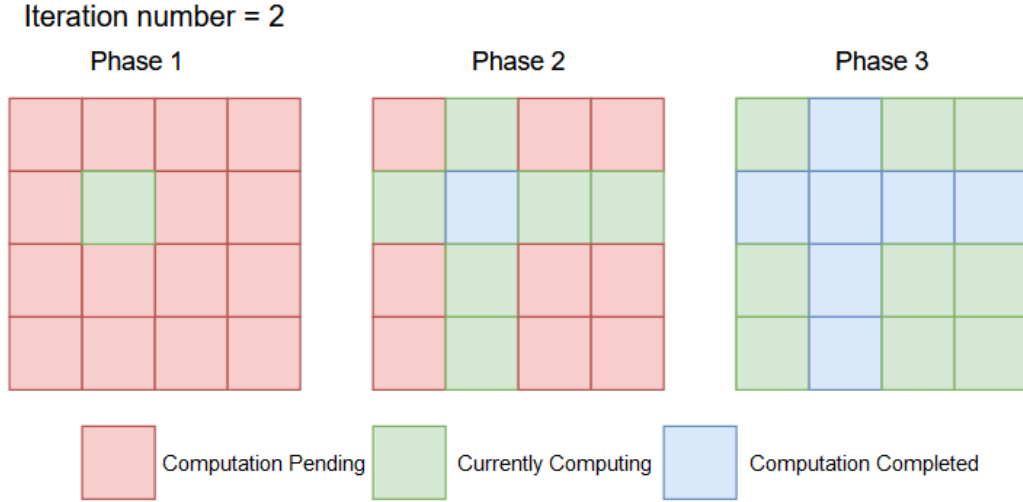


Figure 4: Vanilla GAN (small network)

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

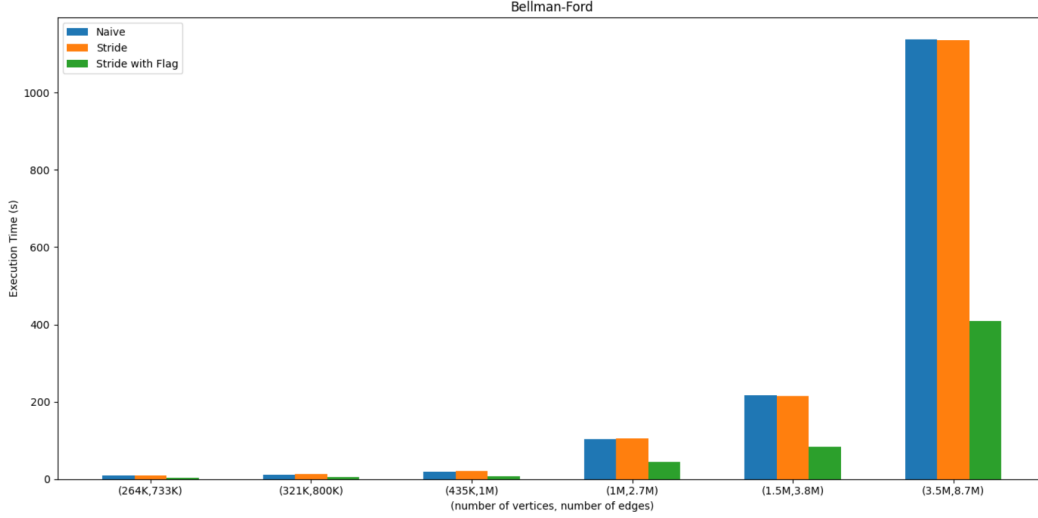


Figure 5: Results: Bellman-Ford

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.

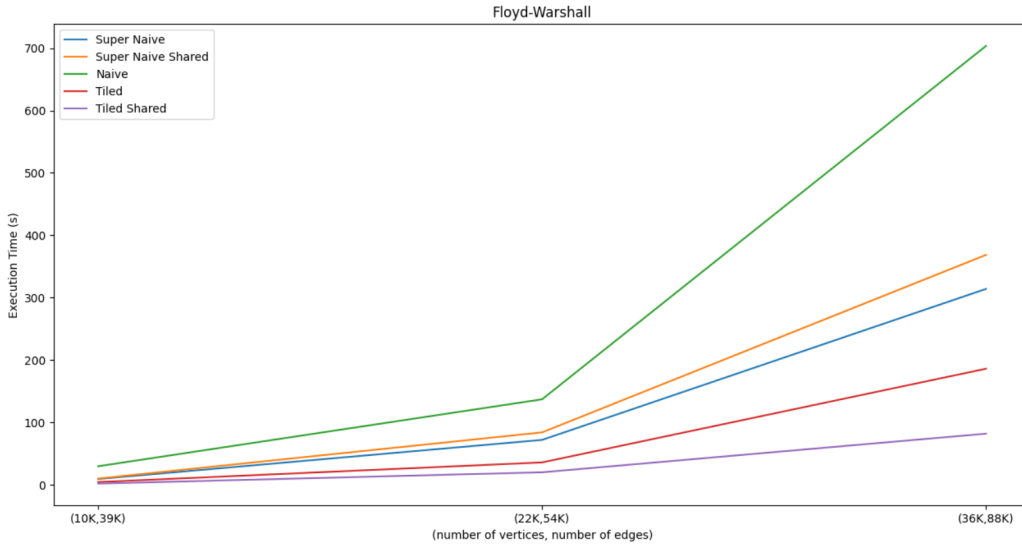


Figure 6: Results of Floyd-Warshall on SNAP’s gnutella network dataset[5]

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" contains all the output files.
      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 2. Load modules
8   2.1 module load ufrc cmake/3.19.1 intel/2018.1.163 cuda/10.0.130
9 3. Compile
10  3.1 mkdir Release
11  3.2 cd Release
12  3.3 cmake -DCMAKE_BUILD_TYPE=Release ..
13  3.4 make
14
15 3 executables named BellmanFord, Dijkstra and FloydWarshall will be created inside
      the Release folder
16
17 4. Bellman Ford
18
19 Command Format

```

```

20  srunch -p gpu --nodes=1 --gpus=geforce:1 --time=00:05:00 --mem=1500 --pty -u
    BellmanFord algorithm inputFileNam source validateOutput outputFormat -i
21
22  algorithm=any integer in the range [0,3] both inclusive 0=CPU, 1=strided, 2=stride
    with flag
23  inputFileNam=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
24
25  source = source node in the graph. any number in the range [0, number of nodes-1]
    both inclusive
26
27  validateOutput=true|false true=compares gpu's output against cpu's output
28
29  outputFormat=none|print|write
30  print=prints distance and path info on screen
31  write=distance and path info on screen
32  none=doesn't print or write distance and path [use this to time the kernels]
33
34  Run this sample command to make sure everything is set up correctly. It will print
    output on screen for the input file sample1.txt
35  srunch -p gpu --nodes=1 --gpus=geforce:1 --time=01:00:00 --mem=1600 --pty -u
    BellmanFord 1 sample1.txt 0 false print -i
36
37  5. Dijkstra
38
39  Command Format
40  srunch -p gpu --nodes=1 --gpus=geforce:1 --time=01:00:00 --mem=1600 --pty -u
    Dijkstra algorithm inputFileNam validateOutput outputFormat -i
41
42  algorithm=any integer in the range [0,1] both inclusive. 0=CPU, 1=CPU
43
44  inputFileNam=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
45
46  validateOutput=true|false true=compares gpu's output against cpu's output
47
48  outputFormat=none|print|write
49  print=prints distance and path info on screen
50  write=distance and path info on screen
51  none=doesn't print or write distance and path [use this to time the kernels]
52
53
54  Run this sample command to make sure everything is set up correctly. It will print
    output on screen for the input file sample1.txt
55  srunch -p gpu --nodes=1 --gpus=geforce:1 --time=01:00:00 --mem=1500 --pty -u
    Dijkstra 0 sample1.txt false print -i
56
57  6. Floyd Warshall
58  Command Format
59  srunch -p gpu --nodes=1 --gpus=geforce:1 --time=01:00:00 --mem=1500 --pty -u
    FloydWarshall algorithm inputFileNam validateOutput outputFormat -i
60
61  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
62
63  inputFileNam=name of input file with extension
64
65  validateOutput=true|false true=compares cpu output with gpu output
66
67  outputFormat=none|print|write
68  print=prints distance and path info on screen
69  write=distance and path info on screen
70  none=doesn't print or write distance and path [use this to time the kernels]

```



```

71 sample command
72 srun -p gpu --nodes=1 --gpus=geforce:1 --time=01:00:00 --mem=25000 --pty -u
73   FloydWarshall 1 sample1.txt false print -i
74
75
76 7. utils.py
77 The dataset files have already been parsed. There is no need to run these commands
78   . They are provided just for documentation purpose.
79
80 1. Create a random graph of 100 vertices and store it in a file named sample2.txt
81   python utils.py random 100 ./data/sample2.txt
82
83 2. Parse a file from DIMACS dataset
84   python utils.py parse ./data/nyc.txt
85
86 3. Add random weights in the range [1, 100] to a file from SNAP dataset
87   python utils.py add ./data/gnutella04.txt 1 100
88
89 4. Replace already added weights with new random weights in the range [1, 100] to
   a file generated from command 3
   python utils.py replace ./data/gnutella04.txt 1 100

```

Listing 1: instructions.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>>>&
46     adjacencyList, int& numVertex, int& numEdges) {
47     cout << "Reading input file" << endl;
48     ifstream file(filename);
49     string line;
50
51     getline(file, line);
52     vector<string> tokens;

```

```

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>&
67     vertices, vector<int>& indices, vector<int>& edges, vector<int>& weights) {
68     int index = 0;
69     indices.push_back(index);
70     for (auto uIter = adjacencyList.begin(); uIter != adjacencyList.end(); ++uIter)
71     {
72         int u = uIter->first;
73         vertices.push_back(u);
74         index += uIter->second.size();
75         indices.push_back(index);
76         for (auto vIter = uIter->second.begin(); vIter != uIter->second.end(); ++
77             vIter) {
78             edges.push_back(vIter->first);
79             weights.push_back(vIter->second);
80         }
81     }
82 }
83
84 // initialize distance and parent for floyd warshall
85 void APSPInitDistanceParent(int numVertex, int* costMatrix, int* distance, int*
86     parent) {
87     cout << "Initializing distance and parent matrices using the cost matrix" <<
88         endl;
89     for (int i = 0; i < numVertex; i++) {
90         for (int j = 0; j < numVertex; j++) {
91             if (i == j) {
92                 distance[i * numVertex + j] = 0;
93                 parent[i * numVertex + j] = -1;
94             }
95             else if (costMatrix[i * numVertex + j] == INF) {
96                 distance[i * numVertex + j] = INF;
97                 parent[i * numVertex + j] = -1;
98             }
99             else {
100                 distance[i * numVertex + j] = costMatrix[i * numVertex + j];
101                 parent[i * numVertex + j] = i;
102             }
103         }
104     }
105 }
106
107 // compare gpu output with cpu output for single source shortest path
108 void validateDistanceSSSP(int numVertex, int* expDistance, int* distance) {
109     for (int i = 0; i < numVertex; i++) {
110         assert(expDistance[i] == distance[i]);
111     }
112     cout << "Validation Successful" << endl;
113 }
114
115 // compare gpu output with cpu output for all pairs shortest path
116 void validateDistanceAPSP(int numVertex, int* expDistance, int* distance) {
117     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
175 + dest] != 0) {
176                 cout << dest << "\t" << distance[src * numVertex + dest] << "\t";
177                 cout << dest;

```

```

177         int tmp = parent[src * numVertex + dest];
178         while (tmp != -1)
179         {
180             cout << "<-" << tmp;
181             tmp = parent[src * numVertex + tmp];
182         }
183         cout << endl;
184     }
185     else {
186         // uncomment this line to output "NA" for paths that don't exist
187         // cout << dest << "\t" << "NA" << "\t" << "-";
188         // cout << endl;
189     }
190 }
191 cout << endl;
192 }
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
204 + dest] != 0) {
205                 out << dest << "\t" << distance[src * numVertex + dest] << "\t";
206                 out << dest;
207
208                 int tmp = parent[src * numVertex + dest];
209                 while (tmp != -1)
210                 {
211                     out << "<-" << tmp;
212                     tmp = parent[src * numVertex + tmp];
213                 }
214                 out << endl;
215             }
216             else {
217                 // uncomment this line to output "NA" for paths that don't exist
218                 // out << dest << "\t" << "NA" << "\t" << "-";
219                 // out << endl;
220             }
221         }
222         out << endl;
223     }
224     out.close();
225 }

```

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
99 void runBellmanFordNaive(int src, int numVertex, int* vertices, int* indices, int*
100     edges, int* weights, int* distance, int* parent) {
101     int* prevDistance = (int*)malloc(numVertex * sizeof(int));
102
103     fill(prevDistance, prevDistance + numVertex, INF); // fill with INF
104
105     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),
118 cudaMemcpyHostToDevice));
119 cudaCheck(cudaMemcpy(d_distance, distance, numVertex * sizeof(int),
120 cudaMemcpyHostToDevice));
121 cudaCheck(cudaMemcpy(d_parent, parent, numVertex * sizeof(int),
122 cudaMemcpyHostToDevice));
123
124 cout << "Calculating shortest distance" << endl;
125 for (int k = 0; k < numVertex - 1; k++) { // numVertex-1 iterations
126     bellmanFordRelaxNaive << <(numVertex - 1) / THREADS_PER_BLOCK + 1,
127     THREADS_PER_BLOCK >> > (numVertex, vertices, indices, edges, weights,
128     d_prevDistance, d_distance, d_parent);
129     cudaCheck(cudaGetLastError()); // check if kernel launch failed
130     cudaCheck(cudaDeviceSynchronize()); // wait for kernel to finish
131     bellmanFordUpdateDistanceNaive << <(numVertex - 1) / THREADS_PER_BLOCK + 1,
132     THREADS_PER_BLOCK >> > (numVertex, d_prevDistance, d_distance);
133     cudaCheck(cudaGetLastError()); // check if kernel launch failed
134     cudaCheck(cudaDeviceSynchronize()); // wait for kernel to finish
135 }
136 cout << "Constructing path" << endl;
137 bellmanFordParentNaive << <(numVertex - 1) / THREADS_PER_BLOCK + 1,
138 THREADS_PER_BLOCK >> > (numVertex, vertices, indices, edges, weights,
139 d_distance, d_parent);
140 cudaCheck(cudaGetLastError()); // check if kernel launch failed
141 cudaCheck(cudaDeviceSynchronize()); // wait for kernel to finish
142
143 // copy from gpu to cpu
144 cout << "Copying results to CPU" << endl;
145 cudaCheck(cudaMemcpy(distance, d_distance, numVertex * sizeof(int),
146 cudaMemcpyDeviceToHost));
147 cudaCheck(cudaMemcpy(parent, d_parent, numVertex * sizeof(int),
148 cudaMemcpyDeviceToHost));
149
150 cudaCheck(cudaFree(d_prevDistance));
151
152 cudaEventRecord(stop, 0);
153 cudaEventSynchronize(stop);
154 cudaEventElapsedTime(&duration, start, stop);
155 cout << "Time: " << duration << "ms" << endl;
156 }
157
158 /*****
159 STRIDE VERSION
160 *****/

```



```

152 // relax with stride
153 __global__
154 void bellmanFordRelaxStride(int numVertex, int* vertices, int* indices, int* edges,
    int* weights, int* prevDistance, int* distance, int* parent) {
155     int tid = blockIdx.x * blockDim.x + threadIdx.x; // thread tid relaxes outgoing
    edges from vertex tid, tid+stride, tid+2*stride,...
156     int stride = blockDim.x * gridDim.x; // stride length
157
158     for(int i = tid; i < numVertex; i += stride){
159         for (int j = indices[i]; j < indices[i + 1]; j++) {
160             int v = edges[j]; // neighbor j
161             int w = weights[j]; // cost from i to j
162
163             if (prevDistance[i] != INF && (prevDistance[i] + w) < distance[v]) { //
    relax
164                 atomicMin(&distance[v], prevDistance[i] + w);
165             }
166         }
167     }
168 }
169
170 // copy the updated cost values in prevDistance with stride
171 __global__
172 void bellmanFordUpdateDistanceStride(int numVertex, int* prevDistance, int* distance
    ) {
173     int tid = blockIdx.x * blockDim.x + threadIdx.x; // thread tid handles vertex
    tid, tid+stride, tid+2*stride, ...
174     int stride = blockDim.x * gridDim.x;
175
176     for (int i = tid; i < numVertex; i += stride) {
177         prevDistance[i] = distance[i]; // copy distance into prevDistance
178     }
179 }
180
181 // find parents of the vertices with stride
182 __global__
183 void bellmanFordParentStride(int numVertex, int* vertices, int* indices, int* edges,
    int* weights, int* distance, int* parent) {
184     int tid = blockIdx.x * blockDim.x + threadIdx.x;
185     int stride = blockDim.x * gridDim.x;
186
187     for (int i = tid; i < numVertex; i += stride) {
188         for (int j = indices[i]; j < indices[i + 1]; j++) { // loop through
    neighbors of i
189             int v = edges[j]; // neighbor j
190             int w = weights[j]; // cost from i to j
191
192             if (distance[i] != INF && (distance[i] + w) == distance[v]) {
193                 parent[v] = i;
194             }
195         }
196     }
197 }
198
199 // run stride version of bellmand ford
200 void runBellmanFordStride(int src, int numVertex, int* vertices, int* indices, int*
    edges, int* weights, int* distance, int* parent) {
201     int* prevDistance = (int*)malloc(numVertex * sizeof(int));
202
203     fill(prevDistance, prevDistance + numVertex, INF); // fill with INF
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);
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 }
267
268 /*****
269 STRIDE WITH FLAG VERSION
270 *****/

```

```

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[]) {
378
379     if (argc < 6) {
380         cout << "Please provide algorithm, input file, source and validate in the
381             command line argument" << endl;
382         return 0;
383     }
384     string pathDataset("../data/"); // path to dataset

```

```

374 string algorithm(argv[1]); // algorithm 0=cpu, 1=naive, 2=stride, 3=stride with
    flag
375 string pathGraphFile(pathDataset+string(argv[2])); // input file
376 int src = stoi(argv[3]); // source node in the range [0, n-1]
377 string validate(argv[4]); // true=compare output with cpu, false=dont
378 string outputFormat(argv[5]); // none=no output (to time the kernel), print=
    prints path on screen, write=write output to a file in the directory named
    output

379
380 int numVertex, numEdges;
381 vector<int> vertices, indices, edges, weights; // for CSR format of a graph
382 map<int, list< pair<int, int > > > adjacencyList; // adjacency list of a graph
383 fileToAdjacencyList(pathGraphFile, adjacencyList, numVertex, numEdges); //
    convert input file to adjacency list
384 adjacencyListToCSR(adjacencyList, vertices, indices, edges, weights); // convert
    adjacency list to CSR format

385
386 adjacencyList.clear(); // clear adjacency list
387
388 int* d_vertices;
389 int* d_indices;
390 int* d_edges;
391 int* d_weights;
392
393 if(algorithm != "0"){ // copy data to gpu if needed
394     cudaCheck(cudaMalloc((void**)&d_vertices, numVertex * sizeof(int)));
395     cudaCheck(cudaMalloc((void**)&d_indices, (numVertex + 1) * sizeof(int)));
396     cudaCheck(cudaMalloc((void**)&d_edges, numEdges * sizeof(int)));
397     cudaCheck(cudaMalloc((void**)&d_weights, numEdges * sizeof(int)));
398
399     cudaCheck(cudaMemcpy(d_vertices, vertices.data(), numVertex * sizeof(int),
        cudaMemcpyHostToDevice));
400     cudaCheck(cudaMemcpy(d_indices, indices.data(), (numVertex + 1) * sizeof(int)
        ), cudaMemcpyHostToDevice));
401     cudaCheck(cudaMemcpy(d_edges, edges.data(), numEdges * sizeof(int),
        cudaMemcpyHostToDevice));
402     cudaCheck(cudaMemcpy(d_weights, weights.data(), numEdges * sizeof(int),
        cudaMemcpyHostToDevice));
403 }
404
405 int* parent = (int*)malloc(numVertex * sizeof(int)); // parent of a vertex for
    path finding
406 int* distance = (int*)malloc(numVertex * sizeof(int)); // distance from src to a
    vertex

407
408 fill(distance, distance + numVertex, INF); // fill with INF
409 fill(parent, parent + numVertex, -1); // fill with -1
410
411 if (algorithm == "0") { // cpu version
412     runCpuBellmanFord(src, numVertex, vertices.data(), indices.data(), edges.
        data(), weights.data(), distance, parent);
413 } else{
414     cout << "Warming up the GPU" << endl;
415     for(int x=0; x<NUM_ITERATION_WARMUP; x++){
416         warmupGpu << < (numVertex - 1) / THREADS_PER_BLOCK + 1,
        THREADS_PER_BLOCK >> > ();
417         cudaCheck(cudaGetLastError());
418         cudaCheck(cudaDeviceSynchronize());
419     }
420     cout << "GPU is warmed up" << endl;
421
422     if (algorithm == "1") { // naive
423         runBellmanFordNaive(src, numVertex, d_vertices, d_indices, d_edges,
        d_weights, distance, parent);
424     }

```

```

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

```

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

```

```

73     int src = blockIdx.x * blockDim.x + threadIdx.x; // thread src calculates
       shortest paths from src to every other vertex
74
75     if (src < numVertex) {
76         distance[src * numVertex + src] = 0;
77
78         for (int i = 0; i < numVertex - 1; i++) {
79             int u = extractMin(numVertex, distance, visited, src); // extract min
80             if (u == -1) { // no min node to explore
81                 break;
82             }
83             visited[src * numVertex + u] = true; // mark u as visited
84             for (int v = 0; v < numVertex; v++) { // loop through neighbors of u
85                 if (!visited[src * numVertex + v] && h_costMatrix[u * numVertex + v]
86                     != INF &&
87                     distance[src * numVertex + v] > distance[src * numVertex + u] +
88                     h_costMatrix[u * numVertex + v]) { // relax
89                     parent[src * numVertex + v] = u;
90                     distance[src * numVertex + v] = distance[src * numVertex + u] +
91                     h_costMatrix[u * numVertex + v];
92                 }
93             }
94         }
95     }
96
97     // run dijkstras on gpu
98     void runGpuDijkstra(int numVertex, int* costMatrix, bool* visited, int* distance,
99         int* parent) {
100         // time the algorithm
101         cudaEvent_t start, stop;
102         cudaEventCreate(&start);
103         cudaEventCreate(&stop);
104         float duration;
105         cudaEventRecord(start, 0);
106
107         // allocate device pointers
108         int* d_costMatrix;
109         int* d_parent;
110         int* d_distance;
111         bool* d_visited;
112
113         // allocate memory on gpu
114         cudaCheck(cudaMalloc((void**)&d_costMatrix, numVertex * numVertex * sizeof(int)));
115         cudaCheck(cudaMalloc((void**)&d_parent, numVertex * numVertex * sizeof(int)));
116         cudaCheck(cudaMalloc((void**)&d_distance, numVertex * numVertex * sizeof(int)));
117         cudaCheck(cudaMalloc((void**)&d_visited, numVertex * numVertex * sizeof(bool)));
118
119         // copy from cpu to gpu
120         cudaCheck(cudaMemcpy(d_costMatrix, costMatrix, numVertex * numVertex * sizeof(
121             int), cudaMemcpyHostToDevice));
122         cudaCheck(cudaMemcpy(d_parent, parent, numVertex * numVertex * sizeof(int),
123             cudaMemcpyHostToDevice));
124         cudaCheck(cudaMemcpy(d_distance, distance, numVertex * numVertex * sizeof(int),
125             cudaMemcpyHostToDevice));
126         cudaCheck(cudaMemcpy(d_visited, visited, numVertex * numVertex * sizeof(bool),
127             cudaMemcpyHostToDevice));
128
129         cout << "Kernel is executing" << endl;
130         dijkstraNaive << <(numVertex - 1) / THREADS_PER_BLOCK + 1, THREADS_PER_BLOCK >>
131         > (numVertex, d_costMatrix, d_visited, d_distance, d_parent);
132         cudaCheck(cudaGetLastError()); // check if kernel launch failed
133         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
132 cudaEventRecord(stop, 0);
133 cudaEventSynchronize(stop);
134 cudaEventElapsedTime(&duration, start, stop);
135 cout << "Time: " << duration << "ms" << endl;
136 }
137
138 int main(int argc, char* argv[]) {
139     if (argc < 5) {
140         cout << "Please provide an input file as a command line argument" << endl;
141         return 0;
142     }
143     string pathDataset("../data/"); // path to dataset
144     string algorithm(argv[1]); // algorithm 0=cpu, 1=naive
145     string pathGraphFile(pathDataset + string(argv[2])); // input file
146     string validate(argv[3]); // true=compare output with cpu, false=dont
147     string outputFormat(argv[4]); // none=no output (to time the kernel), print=
148     // prints path on screen, write=write output to a file in the directory named
149     // output
150
151     int numVertex, numEdges;
152
153     int* h_costMatrix = fileToCostMatrix(pathGraphFile, numVertex, numEdges); //
154     // convert input file to adjacency list
155
156     int* h_parent = (int*)malloc(numVertex * numVertex * sizeof(int));
157     int* h_distance = (int*)malloc(numVertex * numVertex * sizeof(int));
158     bool* h_visited = (bool*)malloc(numVertex * numVertex * sizeof(bool));
159
160     fill(h_parent, h_parent + numVertex * numVertex, -1); // fill with -1
161     fill(h_distance, h_distance + numVertex * numVertex, INF); // fill with INF
162     fill(h_visited, h_visited + numVertex * numVertex, false); // fill with false
163
164     if (algorithm == "0") { // cpu version
165         runCpuDijkstra(numVertex, h_costMatrix, h_distance, h_parent);
166     }
167     else if (algorithm == "1") { // naive
168         cout << "Warming up the GPU" << endl;
169         for (int x = 0; x < NUM_ITERATION_WARMUP; x++) {
170             warmupGpu << (numVertex - 1) / THREADS_PER_BLOCK + 1,
171             THREADS_PER_BLOCK >> > ();
172             cudaCheck(cudaGetLastError());
173             cudaCheck(cudaDeviceSynchronize());
174         }
175         cout << "GPU is warmed up" << endl;
176
177         runGpuDijkstra(numVertex, h_costMatrix, h_visited, h_distance, h_parent);
178         if (validate == "true") {
179             int* expParent = (int*)malloc(numVertex * numVertex * sizeof(int)); //
180             expected parent
181             int* expDistance = (int*)malloc(numVertex * numVertex * sizeof(int)); //
182             expected distance
183             fill(expDistance, expDistance + numVertex * numVertex, INF); // fill
184             with INF
185             fill(expParent, expParent + numVertex * numVertex, -1); // fill with -1
186             runCpuDijkstra(numVertex, h_costMatrix, expDistance, expParent); // run
187             on cpu
188             validateDistanceAPSP(numVertex, expDistance, h_distance); // compare
189             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
188     }.txt in output directory
189     string pathOutputFile(string("../output/d") + algorithm + string(".txt"));
190     cout << "Writing output to" << pathOutputFile << endl;
191     writeOutPathAPSP(pathOutputFile, numVertex, h_distance, h_parent);
192 }
193 else if (outputFormat == "none") { // dont write out path
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 ktok = k * numVertex + k; // index for k->k
31
32                 // relax i->j using node k
33                 if (distance[itok] != INF && distance[ktok] != INF && distance[itoj]
34                     > distance[itok] + distance[ktok]) {
35                     parent[itoj] = k;
36                     distance[itoj] = distance[itok] + distance[ktok];
37                 }
38             }
39         }
40     }
41
42     // time
43     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 // runs super naive on gpu
70 void runFloydWarshallSuperNaive(int numVertex, int* distance, int* parent) {
71     cudaEvent_t start, stop;
72     cudaEventCreate(&start);
73     cudaEventCreate(&stop);
74     float duration;
75
76     cudaEventRecord(start, 0);
77
78     int* d_distance;
79     int* d_parent;
80
81     // allocate memory on GPU and copy data from CPU to GPU
82     cout << "allocating data on GPU" << endl;
83     cudaCheck(cudaMalloc((void**)&d_distance, numVertex * numVertex * sizeof(int)));
84     cudaCheck(cudaMalloc((void**)&d_parent, numVertex * numVertex * sizeof(int)));
85
86     cout << "copying data to GPU" << endl;
87     cudaCheck(cudaMemcpy(d_distance, distance, numVertex * numVertex * sizeof(int),
88         cudaMemcpyHostToDevice));
89     cudaCheck(cudaMemcpy(d_parent, parent, numVertex * numVertex * sizeof(int),
90         cudaMemcpyHostToDevice));
91
92     dim3 dimGrid((numVertex - 1) / TILE_DIM + 1, (numVertex - 1) / TILE_DIM + 1);
93     dim3 dimBlock(TILE_DIM, TILE_DIM);
94     // run kernel
95     cout << "Kernel is executing" << endl;
96     for (int k = 0; k < numVertex; k++) {
97         floydWarshallSuperNaive << <dimGrid, dimBlock >> > (numVertex, k, d_distance,
98             d_parent);
99         cudaCheck(cudaGetLastError());
100         cudaCheck(cudaDeviceSynchronize());
101     }
102
103     // copy results to CPU
104     cout << "copying results to CPU" << endl;
105     cudaCheck(cudaMemcpy(distance, d_distance, numVertex * numVertex * sizeof(int),
106         cudaMemcpyDeviceToHost));

```

```

103     cudaCheck(cudaMemcpy(parent, d_parent, numVertex * numVertex * sizeof(int),
104                       cudaMemcpyDeviceToHost));
105
106     cudaEventRecord(stop, 0);
107     cudaEventSynchronize(stop);
108     cudaEventElapsedTime(&duration, start, stop);
109     cout << "Time: " << duration << "ms" << endl;
110 }
111
112 /*****
113  SUPER NAIVE SHARED VERSION
114  *****/
115 // one thread per edge but with shared memory
116 __global__
117 void floydWarshallSuperNaiveShared(int numVertex, int k, int* distance, int* parent)
118 {
119     int i = blockIdx.y; // choose a start node i
120     int j = blockIdx.x * blockDim.x + threadIdx.x; // choose a neighbor j of i
121
122     if (j < numVertex) {
123         int itoj = numVertex * i + j; // index for i->j
124         int itok = numVertex * i + k; // index for i->k
125         int ktok = numVertex * k + j; // index for k->j
126
127         __shared__ int dist_itok; // shared variable to store i->k
128         if (threadIdx.x == 0) {
129             dist_itok = distance[itok];
130         }
131         __syncthreads();
132
133         // relax i->j using node k
134         if (dist_itok != INF && distance[ktok] != INF && distance[itoj] > dist_itok
135             + distance[ktok]) {
136             distance[itoj] = dist_itok + distance[ktok];
137             parent[itoj] = k;
138         }
139     }
140 }
141
142 // runs super naive shared on gpu
143 void runFloydWarshallSuperNaiveShared(int numVertex, int* distance, int* parent) {
144     cudaEvent_t start, stop;
145     cudaEventCreate(&start);
146     cudaEventCreate(&stop);
147     float duration;
148
149     cudaEventRecord(start, 0);
150
151     int* d_distance;
152     int* d_parent;
153
154     // allocate memory on GPU and copy data from CPU to GPU
155     cout << "allocating data on GPU" << endl;
156     cudaCheck(cudaMalloc((void**)&d_distance, numVertex * numVertex * sizeof(int)));
157     cudaCheck(cudaMalloc((void**)&d_parent, numVertex * numVertex * sizeof(int)));
158
159     cout << "copying data to GPU" << endl;
160     cudaCheck(cudaMemcpy(d_distance, distance, numVertex * numVertex * sizeof(int),
161                         cudaMemcpyHostToDevice));
162     cudaCheck(cudaMemcpy(d_parent, parent, numVertex * numVertex * sizeof(int),
163                         cudaMemcpyHostToDevice));
164
165     dim3 dimGrid((numVertex - 1) / THREADS_PER_BLOCK + 1, numVertex);
166
167     // run kernel

```

```

163     cout << "Kernel is executing" << endl;
164     for (int k = 0; k < numVertex; k++) {
165         floydWarshallSuperNaiveShared << <dimGrid, THREADS_PER_BLOCK >> > (numVertex
166         , k, d_distance, d_parent);
167         cudaCheck(cudaGetLastError());
168         cudaCheck(cudaDeviceSynchronize());
169     }
170
171     // copy results to CPU
172     cout << "copying results to CPU" << endl;
173     cudaCheck(cudaMemcpy(distance, d_distance, numVertex * numVertex * sizeof(int),
174     cudaMemcpyDeviceToHost));
175     cudaCheck(cudaMemcpy(parent, d_parent, numVertex * numVertex * sizeof(int),
176     cudaMemcpyDeviceToHost));
177
178     cudaEventRecord(stop, 0);
179     cudaEventSynchronize(stop);
180     cudaEventElapsedTime(&duration, start, stop);
181     cout << "Time: " << duration << "ms" << endl;
182 }
183
184 /*****
185 NAIVE VERSION
186 *****/
187 // one thread per vertex
188 __global__
189 void floydWarshallNaive(int numVertex, int k, int* distance, int* parent) {
190     int i = blockIdx.x * blockDim.x + threadIdx.x; // choose a start node i
191     if (i < numVertex) {
192         for (int j = 0; j < numVertex; j++) { // loop through its neighbors
193             int itoj = i * numVertex + j; // index for i->j
194             int itok = i * numVertex + k; // index for i->k
195             int ktoj = k * numVertex + j; // index for k->j
196             // relax i->j using node k
197             if (distance[itok] != INF && distance[ktoj] != INF && distance[itoj] >
198             distance[itok] + distance[ktoj]) {
199                 parent[itoj] = k;
200                 distance[itoj] = distance[itok] + distance[ktoj];
201             }
202         }
203     }
204 }
205
206 // runs naive on gpu
207 void runFloydWarshallNaive(int numVertex, int* distance, int* parent) {
208     cudaEvent_t start, stop;
209     cudaEventCreate(&start);
210     cudaEventCreate(&stop);
211     float duration;
212
213     cudaEventRecord(start, 0);
214
215     int* d_distance;
216     int* d_parent;
217
218     // allocate memory on GPU and copy data from CPU to GPU
219     cout << "allocating data on GPU" << endl;
220     cudaCheck(cudaMalloc((void**)&d_distance, numVertex * numVertex * sizeof(int)));
221     cudaCheck(cudaMalloc((void**)&d_parent, numVertex * numVertex * sizeof(int)));
222
223     cout << "copying data to GPU" << endl;
224     cudaCheck(cudaMemcpy(d_distance, distance, numVertex * numVertex * sizeof(int),
225     cudaMemcpyHostToDevice));
226     cudaCheck(cudaMemcpy(d_parent, parent, numVertex * numVertex * sizeof(int),
227     cudaMemcpyHostToDevice));

```

```

222
223 // run kernel
224 cout << "Kernel is executing" << endl;
225 for (int k = 0; k < numVertex; k++) {
226     floydWarshallNaive << <(numVertex - 1) / THREADS_PER_BLOCK + 1,
        THREADS_PER_BLOCK >> > (numVertex, k, d_distance, d_parent);
227     cudaCheck(cudaGetLastError());
228     cudaCheck(cudaDeviceSynchronize());
229 }
230
231 // copy results to CPU
232 cout << "copying results to CPU" << endl;
233 cudaCheck(cudaMemcpy(distance, d_distance, numVertex * numVertex * sizeof(int),
        cudaMemcpyDeviceToHost));
234 cudaCheck(cudaMemcpy(parent, d_parent, numVertex * numVertex * sizeof(int),
        cudaMemcpyDeviceToHost));
235
236 cudaEventRecord(stop, 0);
237 cudaEventSynchronize(stop);
238 cudaEventElapsedTime(&duration, start, stop);
239 cout << "Time: " << duration << "ms" << endl;
240 }
241
242
243 /*****
244 TILED VERSION
245 *****/
246 // tiled with global memory
247
248 // phase 1
249 __global__
250 void floydWarshallTiledPhase1(int numVertex, int primary_tile_number, int* distance,
        int* parent) {
251     int tx = threadIdx.x;
252     int ty = threadIdx.y;
253
254     int i = primary_tile_number * blockDim.y + threadIdx.y; // node i
255     int j = primary_tile_number * blockDim.x + threadIdx.x; // node j
256     if(i < numVertex && j < numVertex){
257         int itoj = i * numVertex + j; // index for i->j
258         for (int k = 0; k < TILE_DIM; k++) { // run floyd warshall in the primary
            tile
259             if (j-tx+k < numVertex && i-ty+k < numVertex &&
260                 distance[itoj - tx + k] != INF && distance[itoj - ty * numVertex + k
                * numVertex] != INF &&
261                 distance[itoj] > distance[itoj - tx + k] + distance[itoj - ty *
                numVertex + k * numVertex]) {
262
263                 distance[itoj] = distance[itoj - tx + k] + distance[itoj - ty *
                numVertex + k * numVertex];
264                 parent[itoj] = TILE_DIM * primary_tile_number + k;
265             }
266             // __syncthreads();
267         }
268     }
269 }
270
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                 distance[itoj] = distance[itoj - tx + k - blockIdx.x * blockDim.
x + primary_tile_number * blockDim.x] + distance[itoj - ty * numVertex + k *
numVertex];
297
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                 distance[itoj] = distance[itoj - tx + k] + distance[itoj - ty *
numVertex + k * numVertex - (blockIdx.x - primary_tile_number) * blockDim.x *
numVertex];
319
320                 parent[itoj] = TILE_DIM * primary_tile_number + k;
321             }
322             // __syncthreads();
323         }
324     }
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
367 // runs tiled version on gpu
368 void runFloydWarshallTiled(int numVertex, int* distance, int* parent) {
369     cudaEvent_t start, stop;
370     cudaEventCreate(&start);
371     cudaEventCreate(&stop);
372     float duration;
373
374     cudaEventRecord(start, 0);
375
376     int* d_distance;
377     int* d_parent;
378
379     // allocate memory on GPU and copy data from CPU to GPU
380     cudaCheck(cudaMalloc((void**)&d_distance, numVertex * numVertex * sizeof(int)));
381     cudaCheck(cudaMalloc((void**)&d_parent, numVertex * numVertex * sizeof(int)));
382
383     cout << "copying data to GPU" << endl;
384     cudaCheck(cudaMemcpy(d_distance, distance, numVertex * numVertex * sizeof(int),
385         cudaMemcpyHostToDevice));
386     cudaCheck(cudaMemcpy(d_parent, parent, numVertex * numVertex * sizeof(int),
387         cudaMemcpyHostToDevice));
388
389     int numDiagonalTiles = (numVertex - 1) / TILE_DIM + 1;
390
391     dim3 dimGridPhase1(1, 1), dimGridPhase2(numDiagonalTiles, 2), dimGridPhase3(
392         numDiagonalTiles, numDiagonalTiles);
393     dim3 dimBlock(TILE_DIM, TILE_DIM);
394
395     cout << "Kernel is executing" << endl;
396     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 WITH 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 &&
s_distanceCurrentTile[k][threadIdx.x] != INF &&
499                 newDistance < shortestDistance) {
500
501                 shortestParent = TILE_DIM * primary_tile_number + k;
502                 shortestDistance = newDistance;
503             }
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             shortestParent = TILE_DIM * primary_tile_number + k;
534             shortestDistance = newDistance;
535         }
536     }
537     __syncthreads();
538 }
539 if (i < numVertex && j < numVertex) {
540     distance[idxCurrentTile] = shortestDistance;
541     parent[idxCurrentTile] = shortestParent;
542 }
543 }
544 // phase 3
545 __global__
546 void floydWarshallTiledSharedPhase3(int numVertex, int primary_tile_number, int*
547     distance, int* parent) {
548     // exclude primary tile, primary row and primary column
549     if (blockIdx.x == primary_tile_number || blockIdx.y == primary_tile_number) {
550         return;
551     }
552
553     __shared__ int s_distancePrimaryRow[TILE_DIM][TILE_DIM]; // primary row tile
554     __shared__ int s_distancePrimaryCol[TILE_DIM][TILE_DIM]; // primary column tile
555     __shared__ int s_distanceCurrentTile[TILE_DIM][TILE_DIM]; // current tile
556
557     int i, j;
558
559     i = TILE_DIM * primary_tile_number + threadIdx.y;
560     j = TILE_DIM * blockIdx.x + threadIdx.x;
561     if (i < numVertex && j < numVertex) {
562         s_distancePrimaryRow[threadIdx.y][threadIdx.x] = distance[i * numVertex + j
563     ];
564     }
565     else {
566         s_distancePrimaryRow[threadIdx.y][threadIdx.x] = INF;
567     }
568 }

```

```

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;
624     int* d_parent;
625
626     // allocate memory on GPU and copy data from CPU to GPU
627     cout << "allocating data on GPU" << endl;
628     cudaCheck(cudaMalloc((void**)&d_distance, numVertex * numVertex * sizeof(int)));
629     cudaCheck(cudaMalloc((void**)&d_parent, numVertex * numVertex * sizeof(int)));

```

```

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));
685
686     APSPInitDistanceParent(numVertex, costMatrix, distance, parent);
687
688     if (algorithm == "0") {
689         runCpuFloydWarshall(numVertex, distance, parent);
690     } else {
691         cout << "Warming up the GPU" << endl;

```

```

684     for (int x = 0; x < NUM_ITERATION_WARMUP; x++) {
685         warmupGpu << < (numVertex - 1) / THREADS_PER_BLOCK + 1,
        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
723 } else {
724     cout << "Illegal output format argument" << endl;
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
76         if action == 'add':
77             addWeights(fileName, int(weightMin), int(weightMax))
78         elif action == 'replace':
79             replaceWeights(fileName, int(weightMin), int(weightMax))

```

Listing 8: utils.py