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 alogrithm 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.

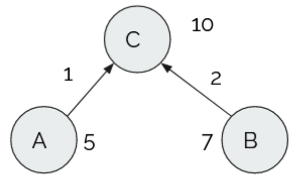

Figure 1: Bellman-Ford: Need for atomic minimum

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
```

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.

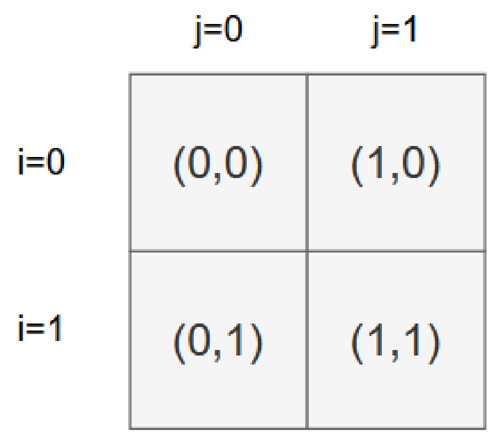

Figure 3: Super Naive: Potential to use shared memory

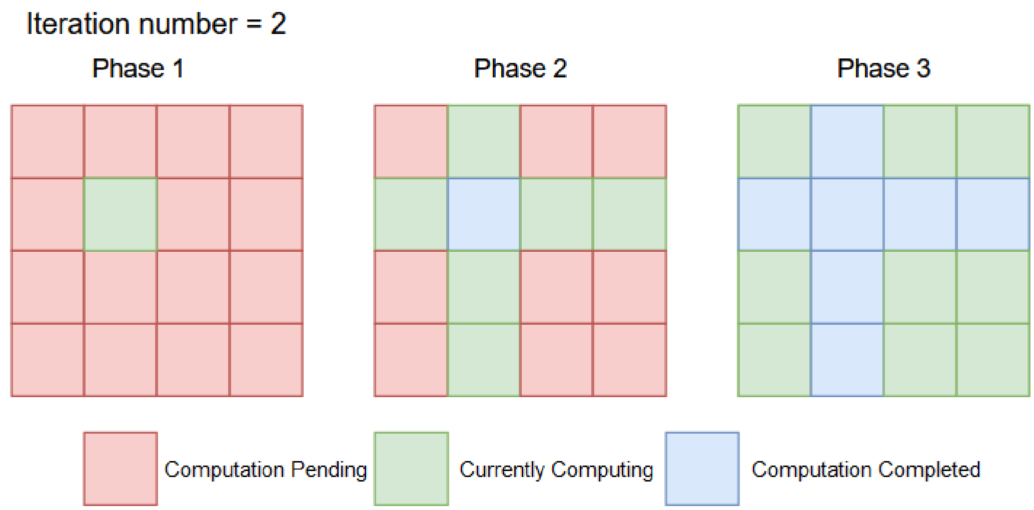

Figure 4: Vanilla GAN (small network)

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

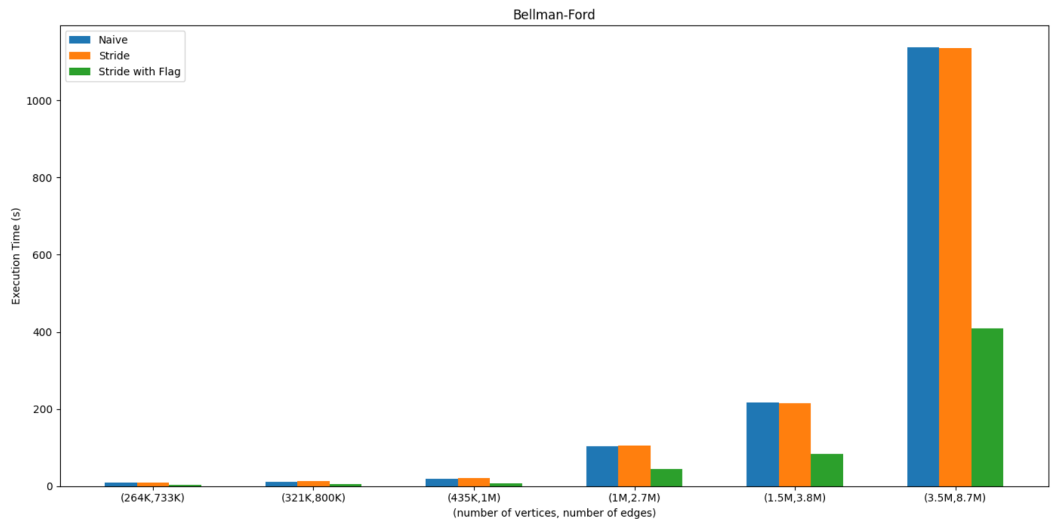

Figure 5: Results: Bellman-Ford

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.

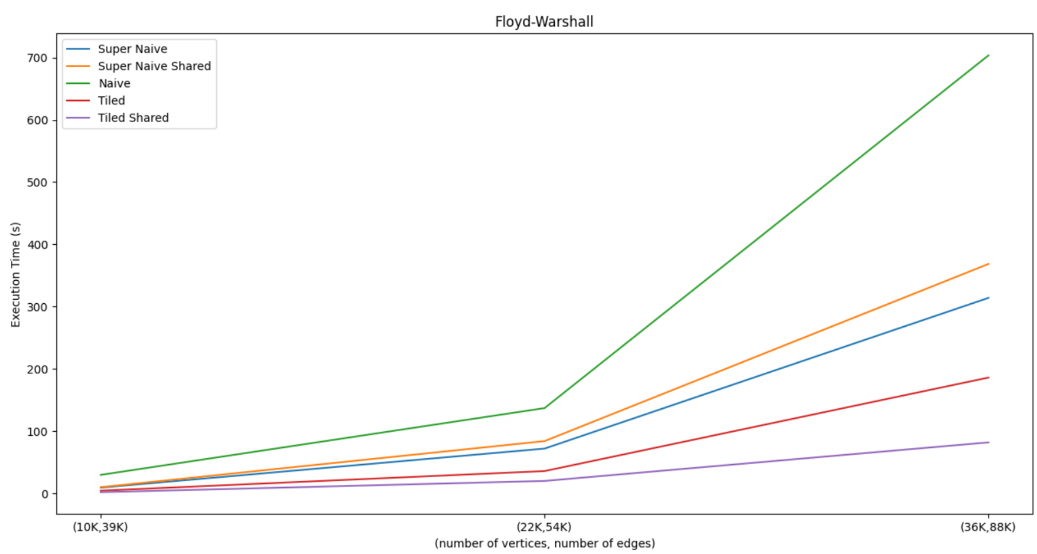

Figure 6: Results of Floyd-Warshall on SNAP's gnutella network dataset[5]

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 agorithm 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

```
Instructions to run

1. Project Structure
  1. Unzip the project
  2. You will see 2 folders in the project. "data" contains all the dataset files.
     sample1.txt represents a graph with 6 nodes and 9 edges. sample2.txt represents
      a graph with 100 nodes and 99000 edges. "output" stores the output files.
     Currently it has a blank dummy files. The output of all the algorithms on just
     one dataset itself is 20GB so they are not included due to size constraints.
     sample files are to be used to confirm the correctness of the algorithms
     because CPU could take hours to run for other files. The output files will be
     overwritten if you rerun the code.
  3. The source code files and CMakeLists.txt will be in the root of the project.

2. Load modules
  2.1 module load ufrc cmake/3.19.1 intel/2018.1.163 cuda/10.0.130

3. Compile
  3.1 mkdir Release
  3.2 cd Release
  3.3 cmake -DCMAKE_BUILD_TYPE=Release ..
  3.4 make

  3 executables named BellmanFord, Dijkstra and FloydWarshall will be created inside
     the Release folder

```

```
4. Bellman Ford

  Command Format
  srun -p gpu --nodes=1 --gpus=geforce:1 --time=00:05:00 --mem=1500 --pty -u BellmanFord algorithm inputFileName source validateOutput outputFormat -i

  algorithm=any integer in the range [0,3] both inclusive 0=CPU, 1=strided, 2=stride with flag
  inputFileName=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

  source = source node in the graph. any number in the range [0, number of nodes-1] both inclusive

  validateOutput=true|false true=compares gpu's output against cpu's output

  outputFormat=none|print|write
  print=prints distance and path info on screen
  write=distance and path is written in a file named bf{algorithm}.txt in "output" folder
  none=doesnt print or write distance and path [use this to time the kernels]

  Run this sample command to make sure everything is set up correctly. It will print output on screen for the input file sample1.txt
  srun -p gpu --nodes=1 --gpus=geforce:1 --time=01:00:00 --mem=1600 --pty -u BellmanFord 1 sample1.txt 0 false print -i

5. Dijkstra

  Command Format
  srun -p gpu --nodes=1 --gpus=geforce:1 --time=01:00:00 --mem=1600 --pty -u Dijkstra algorithm inputFileName validateOutput outputFormat -i

  algorithm=any integer in the range [0,1] both inclusive. 0=CPU, 1=CPU

  inputFileName=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

  validateOutput=true|false true=compares gpu's output against cpu's output

  outputFormat=none|print|write
  print=prints distance and path info on screen
  write=distance and path is written in a file named d{algorithm}.txt in "output" folder
  none=doesnt print or write distance and path [use this to time the kernels]


  Run this sample command to make sure everything is set up correctly. It will print output on screen for the input file sample1.txt
  srun -p gpu --nodes=1 --gpus=geforce:1 --time=01:00:00 --mem=1500 --pty -u Dijkstra 0 sample1.txt false print -i

6. Floyd Warshall
  Command Format
  srun -p gpu --nodes=1 --gpus=geforce:1 --time=01:00:00 --mem=1500 --pty -u FloydWarshall algorithm inputFileName validateOutput outputFormat -i

  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

  inputFileName=name of input file with extension

  validateOutput=true|false true=compares cpu output with gpu output
```

```

outputFormat=none|print|write
print=prints distance and path info on screen
write=distance and path is written in a file named fw{algorithm}.txt in "output"
  folder
none=doesnt print or write distance and path [use this to time the kernels]

sample command
srun -p gpu --nodes=1 --gpus=geforce:1 --time=01:00:00 --mem=25000 --pty -u
  FloydWarshall 1 sample1.txt false print -i



7. utils.py
 The dataset files have already been parsed. There is no need to run these commands
  . They are provided just for documentation purpose.

 1. Create a random graph of 100 vertices and store it in a file named sample2.txt
 python utils.py random 100 ./data/sample2.txt

 2. Parse a file from DIMACS dataset
 python utils.py parse ./data/nyc.txt

 3. Add random weights in the range [1, 100] to a file from SNAP dataset
 python utils.py add ./data/gnutella04.txt 1 100

 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: README.txt

Source Code

```
#ifndef UTILS_H
#define UTILS_H

#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <list>
#include <map>
#include <queue>
#include <sstream>

#include <cassert>

#include <limits.h>

#include<exception>

#define NUM_ITERATION_WARMUP 10
#define INF INT_MAX
#define THREADS_PER_BLOCK 1024

using namespace std;

int* fileToCostMatrix(string filename, int& numVertex, int& numEdges);
struct Graph* fileToAdjacencyList(string filename, struct Graph* costMatrix);
void fileToAdjacencyList(string filename, map<int, list<pair<int, int>>>&
    adjacencyList, int& numVertex, int& numEdges);

void adjacencyListToCSR(map<int, list<pair<int, int>>>& adjacencyList, vector<int>&
    vertices, vector<int>& indices, vector<int>& edges, vector<int>& weights);
```

```
void APSPInitDistanceParent(int numVertex, int* costMatrix, int* distance, int*
    parent);

void validateDistanceSSSP(int numVertex, int* exp_distance, int* distance);
void validateDistanceAPSP(int numVertex, int* exp_distance, int* distance);

void printPathSSSP(int numVertex, int* distance, int* parent);
void printPathAPSP(int numVertex, int* distance, int* parent);

void writeOutPathSSSP(string filepath, int numVertex, int* distance, int* parent);
void writeOutPathAPSP(string filepath, int numVertex, int* distance, int* parent);


#endif
```

Listing 2: utils.h

```
#include "utils.h"

void splitBySpaceToVector(const string& line, vector<string>& tokens) {
    stringstream linestream(line);
    string token;
    while (linestream >> token) {
        tokens.push_back(token);
    }
}

// converts input graph file to cost matrix
int* fileToCostMatrix(string filename, int& numVertex, int& numEdges) {
    cout << "Reading input file" << endl;
    ifstream file(filename);
    string line;

    getline(file, line);
    vector<string> tokens;

    splitBySpaceToVector(line, tokens);
    numVertex = stoi(tokens[0]), numEdges = stoi(tokens[1]);

    int* costMatrix = (int*)malloc(numVertex * numVertex * sizeof(int));
    if (costMatrix == NULL) {
        cout << "Malloc failed" << endl;
        throw std::exception();
    }
    fill(costMatrix, costMatrix + numVertex * numVertex, INF);

    while (getline(file, line)) { // parse input file line by line
        stringstream linestream(line);
        vector<string> tokens;
        string token;
        while (linestream >> token) {
            tokens.push_back(token);
        }
        int src = stoi(tokens[0]), dest = stoi(tokens[1]), cost = stoi(tokens[2]);
        costMatrix[src * numVertex + dest] = cost; // update cost matrix
    }
    cout << "Finished reading input file" << endl;
    return costMatrix;
}

// converts input graph file to adjacency list
void fileToAdjacencyList(string filename, map<int, list<pair<int, int>>>&
    adjacencyList, int& numVertex, int& numEdges) {
    cout << "Reading input file" << endl;
```

```
    ifstream file(filename);
    string line;

    getline(file, line);
    vector<string> tokens;

    splitBySpaceToVector(line, tokens);
    numVertex = stoi(tokens[0]), numEdges = stoi(tokens[1]);

    while (getline(file, line)) { // parse input file line by line
        tokens.clear();
        splitBySpaceToVector(line, tokens);
        int src = stoi(tokens[0]), dest = stoi(tokens[1]), cost = stoi(tokens[2]);
        adjacencyList[src].push_back(make_pair(dest, cost));
    }
    cout << "Finished reading input file" << endl;
}

// converts adjacency list to CSR format
void adjacencyListToCSR(map<int, list<pair<int, int>>>& adjacencyList, vector<int>& vertices, vector<int>& indices, vector<int>& edges, vector<int>& weights) {
    int index = 0;
    indices.push_back(index);
    for (auto uIter = adjacencyList.begin(); uIter != adjacencyList.end(); ++uIter) {
        int u = uIter->first;
        vertices.push_back(u);
        index += uIter->second.size();
        indices.push_back(index);
        for (auto vIter = uIter->second.begin(); vIter != uIter->second.end(); ++vIter) {
            edges.push_back(vIter->first);
            weights.push_back(vIter->second);
        }
    }
}

// initialize distance and parent for floyd warshall
void APSPInitDistanceParent(int numVertex, int* costMatrix, int* distance, int* parent) {
    cout << "Initializing distance and parent matrices using the cost matrix" << endl;
    for (int i = 0; i < numVertex; i++) {
        for (int j = 0; j < numVertex; j++) {
            if (i == j) {
                distance[i * numVertex + j] = 0;
                parent[i * numVertex + j] = -1;
            }
            else if (costMatrix[i * numVertex + j] == INF) {
                distance[i * numVertex + j] = INF;
                parent[i * numVertex + j] = -1;
            }
            else {
                distance[i * numVertex + j] = costMatrix[i * numVertex + j];
                parent[i * numVertex + j] = i;
            }
        }
    }
}

// compare gpu output with cpu output for single source shortest path
void validateDistanceSSSP(int numVertex, int* expDistance, int* distance) {
    for (int i = 0; i < numVertex; i++) {
        assert(expDistance[i] == distance[i]);
    }
```

```
    cout << "Validation Successful" << endl;
}

// compare gpu output with cpu output for all pairs shortest path
void validateDistanceAPSP(int numVertex, int* expDistance, int* distance) {
    for (int i = 0; i < numVertex; i++) {
        for (int j = 0; j < numVertex; j++) {
            assert(expDistance[i * numVertex + j] == distance[i * numVertex + j]);
        }
    }
    cout << "Validation Successful" << endl;
}

// print single source shortest path on screen
void printPathSSSP(int numVertex, int* distance, int* parent) {
    cout << "Node\tCost\tPath" << endl;
    for (int i = 0; i < numVertex; i++) {
        if (distance[i] != INF && distance[i] != 0) {
            cout << i << "\t" << distance[i] << "\t";
            cout << i;

            int tmp = parent[i];
            while (tmp != -1)
            {
                cout << "<-" << tmp;
                tmp = parent[tmp];
            }
        }
        else {
            cout << i << "\t" << "NA" << "\t" << "-";
        }
        cout << endl;
    }
}

// write single source shortest path to a file
void writeOutPathSSSP(string filepath, int numVertex, int* distance, int* parent) {
    ofstream out(filepath);
    out << "Node\tCost\tPath" << endl;
    for (int i = 0; i < numVertex; i++) {
        if (distance[i] != INF && distance[i] != 0) {
            out << i << "\t" << distance[i] << "\t";
            out << i;

            int tmp = parent[i];
            while (tmp != -1)
            {
                out << "<-" << tmp;
                tmp = parent[tmp];
            }
            out << endl;
        }
        else {
            // uncomment this line to output "NA" for paths that don't exist
            // out << i << "\t" << "NA" << "\t" << "-";
            // out << endl;
        }
    }
    out.close();
}

// print all pairs source shortest path on screen
void printPathAPSP(int numVertex, int* distance, int* parent) {
    for (int src = 0; src < numVertex; src++) {
        cout << "Source: " << src << endl;
```

```
        cout << "Node\tCost\tPath" << endl;
        for (int dest = 0; dest < numVertex; dest++) {
            if (distance[src * numVertex + dest] != INF && distance[src * numVertex
    + dest] != 0) {
                cout << dest << "\t" << distance[src * numVertex + dest] << "\t";
                cout << dest;

                int tmp = parent[src * numVertex + dest];
                while (tmp != -1)
                {
                    cout << "<-" << tmp;
                    tmp = parent[src * numVertex + tmp];
                }
                cout << endl;
            }
            else {
                // uncomment this line to output "NA" for paths that don't exist
                // cout << dest << "\t" << "NA" << "\t" << "-";
                // cout << endl;
            }
        }
        cout << endl;
    }
}

// write all pairs source shortest path to a file
void writeOutPathAPSP(string filepath, int numVertex, int* distance, int* parent) {
    ofstream out(filepath);
    for (int src = 0; src < numVertex; src++) {
        out << "Source: " << src << endl;
        out << "Node\tCost\tPath" << endl;
        for (int dest = 0; dest < numVertex; dest++) {
            if (distance[src * numVertex + dest] != INF && distance[src * numVertex
    + dest] != 0) {
                out << dest << "\t" << distance[src * numVertex + dest] << "\t";
                out << dest;

                int tmp = parent[src * numVertex + dest];
                while (tmp != -1)
                {
                    out << "<-" << tmp;
                    tmp = parent[src * numVertex + tmp];
                }
                out << endl;
            }
            else {
                // uncomment this line to output "NA" for paths that don't exist
                // out << dest << "\t" << "NA" << "\t" << "-";
                // out << endl;
            }
        }
        out << endl;
    }
    out.close();
}
```

Listing 3: utils.cpp

```
#ifndef CUDA_CHECK_CUH
#define CUDA_CHECK_CUH

#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <cstdio>
```

```
#include <cassert>

/*  Wrapper to provide error checking for CUDA API calls */

inline
cudaError_t cudaCheck(cudaError_t result) {
    if (result != cudaSuccess) {
        fprintf(stderr, "CUDA Runtime Error: %s\n", cudaGetErrorString(result));
        assert(result == cudaSuccess);
    }
    return result;
}

__global__
void warmpupGpu() {
    __shared__ int s_tid;
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (threadIdx.x == 0) {
        s_tid = tid;
    }
    __syncthreads();
    tid = s_tid;
}

#endif /*CUDA_CHECK_CUH*/
```

Listing 4: utils.cuh

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include "utils.cuh"

#include <iostream>

#include "utils.h"

using namespace std;

/*********************************************************************
SERIAL VERSION
*********************************************************************/

// run bellman ford on cpu
void runCpuBellmanFord(int src, int numVertex, int* vertices, int* indices, int*
    edges, int* weights, int* distance, int* parent) {
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    float duration;
    cudaEventRecord(start, 0);

    distance[src] = 0;
    for (int k = 0; k < numVertex-1; k++) { // a total of numVertex-1 iterations
        for (int i = 0; i < numVertex; i++) { // loop through all vertices
            for (int j = indices[i]; j < indices[i + 1]; j++) { // loop through
    neighbors of i
                int v = edges[j]; // neighbor j
                int w = weights[j]; // cost from i to j

                if (distance[i] != INF && (distance[i] + w) < distance[v]) { //
    relax
                    parent[v] = i;
                    distance[v] = distance[i] + w;
                }
```

```
            }
        }
    }

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&duration, start, stop);
    cout << "Time: " << duration << "ms" << endl;
}

/**********************************************************************
NAIVE VERSION
**********************************************************************/

// relax
__global__
void bellmanFordRelaxNaive(int numVertex, int* vertices, int* indices, int* edges, int* weights, int* prevDistance, int* distance, int* parent) {
    int i = blockIdx.x * blockDim.x + threadIdx.x; // thread i relaxes outgoing edges from vertex i
    if (i < numVertex) {
        for (int j = indices[i]; j < indices[i + 1]; j++) { // loop through neighbors of i
            int v = edges[j]; // neighbor j
            int w = weights[j]; // cost from i to j

            if (prevDistance[i] != INF && (prevDistance[i] + w) < distance[v]) { // relax
                atomicMin(&distance[v], prevDistance[i] + w); // atomic minimum
            }
        }
    }
}

// copy the updated cost values in prevDistance
__global__
void bellmanFordUpdateDistanceNaive(int numVertex, int* prevDistance, int* distance) {
    int i = blockIdx.x * blockDim.x + threadIdx.x; // thread i handles vertex i
    if (i < numVertex) {
        prevDistance[i] = distance[i]; // copy distance into prevDistance
    }
}

// find parents of the vertices
__global__
void bellmanFordParentNaive(int numVertex, int* vertices, int* indices, int* edges, int* weights, int* distance, int* parent) {
    int i = blockIdx.x * blockDim.x + threadIdx.x; // thread i checks if it is the parent of any of it's neighbors
    if (i < numVertex) {
        for (int j = indices[i]; j < indices[i + 1]; j++) { // loop through neighbors of i
            int v = edges[j]; // neighbor j
            int w = weights[j]; // cost from i to j

            if (distance[i] != INF && (distance[i] + w) == distance[v]) {
                parent[v] = i;
            }
        }
    }
}

// run naive version of bellmand ford
```

```
void runBellmanFordNaive(int src, int numVertex, int* vertices, int* indices, int* edges, int* weights, int* distance, int* parent) {
    int* prevDistance = (int*)malloc(numVertex * sizeof(int));

    fill(prevDistance, prevDistance + numVertex, INF); // fill with INF

    prevDistance[src] = 0;
    distance[src] = 0;

    // time the algorithm
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    float duration;
    cudaEventRecord(start, 0);

    // device pointers
    int* d_prevDistance;
    int* d_distance;
    int* d_parent;

    // allocate memory on device
    cudaCheck(cudaMalloc((void**)&d_prevDistance, numVertex * sizeof(int)));
    cudaCheck(cudaMalloc((void**)&d_distance, numVertex * sizeof(int)));
    cudaCheck(cudaMalloc((void**)&d_parent, numVertex * sizeof(int)));

    // copy from cpu to gpus
    cudaCheck(cudaMemcpy(d_prevDistance, prevDistance, numVertex * sizeof(int), cudaMemcpyHostToDevice));
    cudaCheck(cudaMemcpy(d_distance, distance, numVertex * sizeof(int), cudaMemcpyHostToDevice));
    cudaCheck(cudaMemcpy(d_parent, parent, numVertex * sizeof(int), cudaMemcpyHostToDevice));

    cout << "Calculating shortest distance" << endl;
    for (int k = 0; k < numVertex - 1; k++) { // numVertex-1 iterations
        bellmanFordRelaxNaive << <(numVertex - 1) / THREADS_PER_BLOCK + 1, THREADS_PER_BLOCK >> > (numVertex, vertices, indices, edges, weights, d_prevDistance, d_distance, d_parent);
        cudaCheck(cudaGetLastError()); // check if kernel launch failed
        cudaCheck(cudaDeviceSynchronize()); // wait for kernel to finish
        bellmanFordUpdateDistanceNaive << <(numVertex - 1) / THREADS_PER_BLOCK + 1, THREADS_PER_BLOCK >> > (numVertex, d_prevDistance, d_distance);
        cudaCheck(cudaGetLastError()); // check if kernel launch failed
        cudaCheck(cudaDeviceSynchronize()); // wait for kernel to finis
    }
    cout << "Constructing path" << endl;
    bellmanFordParentNaive << <(numVertex - 1) / THREADS_PER_BLOCK + 1, THREADS_PER_BLOCK >> > (numVertex, vertices, indices, edges, weights, d_distance, d_parent);
    cudaCheck(cudaGetLastError()); // check if kernel launch failed
    cudaCheck(cudaDeviceSynchronize()); // wait for kernel to finis

    // copy from gpu to cpu
    cout << "Copying results to CPU" << endl;
    cudaCheck(cudaMemcpy(distance, d_distance, numVertex * sizeof(int), cudaMemcpyDeviceToHost));
    cudaCheck(cudaMemcpy(parent, d_parent, numVertex * sizeof(int), cudaMemcpyDeviceToHost));

    cudaCheck(cudaFree(d_prevDistance));

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&duration, start, stop);
```

```
    cout << "Time: " << duration << "ms" << endl;
}

/*****************************************************************
STRIDE VERSION
*****************************************************************/

// relax with stride
__global__
void bellmanFordRelaxStride(int numVertex, int* vertices, int* indices, int* edges, int* weights, int* prevDistance, int* distance, int* parent) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x; // thread tid relaxes outgoing edges from vertex tid, tid+stride, tid+2*stride,...
    int stride = blockDim.x * gridDim.x; // stride length

    for(int i = tid; i < numVertex; i += stride){
        for (int j = indices[i]; j < indices[i + 1]; j++) {
            int v = edges[j]; // neighbor j
            int w = weights[j]; // cost from i to j

            if (prevDistance[i] != INF && (prevDistance[i] + w) < distance[v]) { // relax
                atomicMin(&distance[v], prevDistance[i] + w);
            }
        }
    }
}

// copy the updated cost values in prevDistance with stride
__global__
void bellmanFordUpdateDistanceStride(int numVertex, int* prevDistance, int* distance) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x; // thread tid handles vertex tid, tid+stride, tid+2*stride, ...
    int stride = blockDim.x * gridDim.x;

    for (int i = tid; i < numVertex; i += stride) {
        prevDistance[i] = distance[i]; // copy distance into prevDistance
    }
}

// find parents of the vertices with stride
__global__
void bellmanFordParentStride(int numVertex, int* vertices, int* indices, int* edges, int* weights, int* distance, int* parent) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;

    for (int i = tid; i < numVertex; i += stride) {
        for (int j = indices[i]; j < indices[i + 1]; j++) { // loop through neighbors of i
            int v = edges[j]; // neighbor j
            int w = weights[j]; // cost from i to j

            if (distance[i] != INF && (distance[i] + w) == distance[v]) {
                parent[v] = i;
            }
        }
    }
}

// run stride version of bellmand ford
void runBellmanFordStride(int src, int numVertex, int* vertices, int* indices, int* edges, int* weights, int* distance, int* parent) {
    int* prevDistance = (int*)malloc(numVertex * sizeof(int));
```

```
    fill(prevDistance, prevDistance + numVertex, INF); // fill with INF

    prevDistance[src] = 0;
    distance[src] = 0;

    // time the algorithm
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    float duration;
    cudaEventRecord(start, 0);

    // device pointers
    int* d_prevDistance;
    int* d_distance;
    int* d_parent;


    // allocate memory on device
    cudaCheck(cudaMalloc((void**)&d_prevDistance, numVertex * sizeof(int)));
    cudaCheck(cudaMalloc((void**)&d_distance, numVertex * sizeof(int)));
    cudaCheck(cudaMalloc((void**)&d_parent, numVertex * sizeof(int)));

    // copy from cpu to gpu
    cudaCheck(cudaMemcpy(d_prevDistance, prevDistance, numVertex * sizeof(int),
    cudaMemcpyHostToDevice));
    cudaCheck(cudaMemcpy(d_distance, distance, numVertex * sizeof(int),
    cudaMemcpyHostToDevice));
    cudaCheck(cudaMemcpy(d_parent, parent, numVertex * sizeof(int),
    cudaMemcpyHostToDevice));

    int numBlocks = ((numVertex - 1) / THREADS_PER_BLOCK + 1) / 2; // use half the
    number of required blocks
    cout << "Calculating shortest distance" << endl;
    for (int k = 0; k < numVertex - 1; k++) { // numVertex-1 iterations
        bellmanFordRelaxStride << <numBlocks, THREADS_PER_BLOCK >> > (numVertex,
    vertices, indices, edges, weights, d_prevDistance, d_distance, d_parent);
        cudaCheck(cudaGetLastError()); // check if kernel launch failed
        cudaCheck(cudaDeviceSynchronize()); // wait for the kernel to finish
        bellmanFordUpdateDistanceStride << <numBlocks, THREADS_PER_BLOCK >> > (
    numVertex, d_prevDistance, d_distance);
        cudaCheck(cudaGetLastError()); // check if kernel launch failed
        cudaCheck(cudaDeviceSynchronize()); // wait for the kernel to finish
    }
    cout << "Constructing path" << endl;
    bellmanFordParentStride << <numBlocks, THREADS_PER_BLOCK >> > (numVertex,
    vertices, indices, edges, weights, d_distance, d_parent);
    cudaCheck(cudaGetLastError()); // check if kernel launch failed
    cudaCheck(cudaDeviceSynchronize()); // wait for the kernel to finish

    // copy from gpu to cpu
    cout << "Copying results to CPU" << endl;
    cudaCheck(cudaMemcpy(distance, d_distance, numVertex * sizeof(int),
    cudaMemcpyDeviceToHost));
    cudaCheck(cudaMemcpy(parent, d_parent, numVertex * sizeof(int),
    cudaMemcpyDeviceToHost));

    cudaCheck(cudaFree(d_prevDistance));

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&duration, start, stop);
    cout << "Time: " << duration << "ms" << endl;
}
```

```



/*********************************************************************
STRIDE WITH FLAG VERSION
*********************************************************************/

// relax with stride
__global__
void bellmanFordRelaxStrideFlag(int numVertex, int* vertices, int* indices, int* edges, int* weights, int* prevDistance, int* distance, int* parent, bool* flag) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x; // thread i relaxes outgoing edges from vertex i
    int stride = blockDim.x * gridDim.x;

    for (int i = tid; i < numVertex; i += stride) {
        if (flag[i]) { // relax outgoing edges of i only if distance to i changed in the previous iteration
            flag[i] = false;
            for (int j = indices[i]; j < indices[i + 1]; j++) { // loop through neighbors of i
                int v = edges[j]; // neighbor j
                int w = weights[j]; // cost from i to j

                if (prevDistance[i] != INF && (prevDistance[i] + w) < distance[v]) { // relax
                    atomicMin(&distance[v], prevDistance[i] + w);
                }
            }
        }
    }
}

// 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
__global__
void bellmanFordUpdateDistanceStrideFlag(int numVertex, int* prevDistance, int* distance, bool* flag) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;

    for (int i = tid; i < numVertex; i += stride) {
        if (prevDistance[i] > distance[i]) {
            flag[i] = true;
        }
        prevDistance[i] = distance[i];
    }
}

// run stride with flag version of bellmand ford
void runBellmanFordStrideFlag(int src, int numVertex, int* vertices, int* indices, int* edges, int* weights, int* distance, int* parent) {
    int* prevDistance = (int*)malloc(numVertex * sizeof(int));
    bool* flag = (bool*)malloc(numVertex * sizeof(bool));

    fill(prevDistance, prevDistance + numVertex, INF); // fill with INF
    fill(flag, flag + numVertex, false); // fill with false

    prevDistance[src] = 0;
    distance[src] = 0;
    flag[src] = true;

    // time the algorithm
    cudaEvent_t start, stop;
```

```
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    float duration;
    cudaEventRecord(start, 0);

    // device pointers
    int* d_prevDistance;
    int* d_distance;
    int* d_parent;
    bool* d_flag;

    // allocate memory on gpu
    cudaCheck(cudaMalloc((void**)&d_prevDistance, numVertex * sizeof(int)));
    cudaCheck(cudaMalloc((void**)&d_distance, numVertex * sizeof(int)));
    cudaCheck(cudaMalloc((void**)&d_parent, numVertex * sizeof(int)));
    cudaCheck(cudaMalloc((void**)&d_flag, numVertex * sizeof(bool)));

    // copy from cpu to cpu
    cudaCheck(cudaMemcpy(d_prevDistance, prevDistance, numVertex * sizeof(int), cudaMemcpyHostToDevice));
    cudaCheck(cudaMemcpy(d_distance, distance, numVertex * sizeof(int), cudaMemcpyHostToDevice));
    cudaCheck(cudaMemcpy(d_parent, parent, numVertex * sizeof(int), cudaMemcpyHostToDevice));
    cudaCheck(cudaMemcpy(d_flag, flag, numVertex * sizeof(bool), cudaMemcpyHostToDevice));

    cout << "Calculating shortest distance" << endl;
    int numBlocks = ((numVertex - 1) / THREADS_PER_BLOCK + 1) / 2; // use half the number of required blocks
    for (int k = 0; k < numVertex - 1; k++) { // numVertex-1 iterations
        bellmanFordRelaxStrideFlag << <numBlocks, THREADS_PER_BLOCK >> > (numVertex, vertices, indices, edges, weights, d_prevDistance, d_distance, d_parent, d_flag);
        cudaCheck(cudaGetLastError()); // check if kernel launch failed
        cudaCheck(cudaDeviceSynchronize()); // wait for the kernel to finish
        bellmanFordUpdateDistanceStrideFlag << <numBlocks, THREADS_PER_BLOCK >> > (numVertex, d_prevDistance, d_distance, d_flag);
        cudaCheck(cudaGetLastError()); // check if kernel launch failed
        cudaCheck(cudaDeviceSynchronize()); // wait for the kernel to finish
    }
    cout << "Constructing path" << endl;
    bellmanFordParentStride << <numBlocks, THREADS_PER_BLOCK >> > (numVertex, vertices, indices, edges, weights, d_distance, d_parent);
    cudaCheck(cudaGetLastError()); // check if kernel launch failed
    cudaCheck(cudaDeviceSynchronize()); // wait for the kernel to finish

    // copy from gpu to cpu
    cout << "Copying results to CPU" << endl;
    cudaCheck(cudaMemcpy(distance, d_distance, numVertex * sizeof(int), cudaMemcpyDeviceToHost));
    cudaCheck(cudaMemcpy(parent, d_parent, numVertex * sizeof(int), cudaMemcpyDeviceToHost));

    cudaCheck(cudaFree(d_prevDistance));
    cudaCheck(cudaFree(d_flag));

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&duration, start, stop);
    cout << "Time: " << duration << "ms" << endl;
}


int main(int argc, char* argv[]) {
```

```

    if (argc < 6) {
        cout << "Please provide algorithm, input file, source and validate in the command line argument" << endl;
        return 0;
    }
    string pathDataset("../data/"); // path to dataset
    string algorithm(argv[1]); // algorithm 0=cpu, 1=naive, 2=stride, 3=stride with flag
    string pathGraphFile(pathDataset+string(argv[2])); // input file
    int src = stoi(argv[3]); // source node in the range [0, n-1]
    string validate(argv[4]); // true=compare output with cpu, false=dont
    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

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

    adjacencyList.clear(); // clear adjacency list

    int* d_vertices;
    int* d_indices;
    int* d_edges;
    int* d_weights;

    if(algorithm != "0"){ // copy data to gpu if needed
        cudaCheck(cudaMalloc((void**)&d_vertices, numVertex * sizeof(int)));
        cudaCheck(cudaMalloc((void**)&d_indices, (numVertex + 1) * sizeof(int)));
        cudaCheck(cudaMalloc((void**)&d_edges, numEdges * sizeof(int)));
        cudaCheck(cudaMalloc((void**)&d_weights, numEdges * sizeof(int)));

        cudaCheck(cudaMemcpy(d_vertices, vertices.data(), numVertex * sizeof(int), cudaMemcpyHostToDevice));
        cudaCheck(cudaMemcpy(d_indices, indices.data(), (numVertex + 1) * sizeof(int), cudaMemcpyHostToDevice));
        cudaCheck(cudaMemcpy(d_edges, edges.data(), numEdges * sizeof(int), cudaMemcpyHostToDevice));
        cudaCheck(cudaMemcpy(d_weights, weights.data(), numEdges * sizeof(int), cudaMemcpyHostToDevice));
    }

    int* parent = (int*)malloc(numVertex * sizeof(int)); // parent of a vertex for path finding
    int* distance = (int*)malloc(numVertex * sizeof(int)); // distance from src to a vertex

    fill(distance, distance + numVertex, INF); // fill with INF
    fill(parent, parent + numVertex, -1); // fill with -1

    if (algorithm == "0") { // cpu version
        runCpuBellmanFord(src, numVertex, vertices.data(), indices.data(), edges.data(), weights.data(), distance, parent);
    } else{
        cout << "Warming up the GPU" << endl;
        for(int x=0; x<NUM_ITERATION_WARMUP; x++){
            warmpupGpu << < (numVertex - 1) / THREADS_PER_BLOCK + 1, THREADS_PER_BLOCK >> > ();
            cudaCheck(cudaGetLastError());
            cudaCheck(cudaDeviceSynchronize());
```

```
        }
        cout << "GPU is warmed up" << endl;

        if (algorithm == "1") { // naive
            runBellmanFordNaive(src, numVertex, d_vertices, d_indices, d_edges,
    d_weights, distance, parent);
        }
        else if (algorithm == "2") { // stride
            runBellmanFordStride(src, numVertex, d_vertices, d_indices, d_edges,
    d_weights, distance, parent);
        }
        else if (algorithm == "3") { // stride with flag
            runBellmanFordStrideFlag(src, numVertex, d_vertices, d_indices, d_edges,
     d_weights, distance, parent);
        }
        else {
            cout << "Illegal Algorithm" << endl;
        }

        if (validate == "true") { // validate gpu output with cpu
            int* expParent = (int*)malloc(numVertex * sizeof(int)); // expected
    parent
            int* expDistance = (int*)malloc(numVertex * sizeof(int)); // expected
    distance
            fill(expDistance, expDistance + numVertex, INF); // fill with INF
            fill(expParent, expParent + numVertex, -1); // fill with -1
            runCpuBellmanFord(src, numVertex, vertices.data(), indices.data(), edges
    .data(), weights.data(), distance, expParent); // run on cpu
            validateDistanceSSSP(numVertex, expDistance, distance); // compare
    distance with expDistance
        }
    }

    // free
    cudaCheck(cudaFree(d_vertices));
    cudaCheck(cudaFree(d_indices));
    cudaCheck(cudaFree(d_edges));
    cudaCheck(cudaFree(d_weights));

    if (outputFormat == "print") {
        printPathSSSP(numVertex, distance, parent); // print paths to screen
    }
    else if (outputFormat == "write") { // write output to a file named bf{algorithm
    }.txt in output directory
        string pathOutputFile(string("../output/bf") + algorithm + string(".txt"));
        cout << "Writing output to" << pathOutputFile << endl;
        writeOutPathSSSP(pathOutputFile, numVertex, distance, parent);
    }
    else if (outputFormat == "none") { // dont write out path

    }
    else {
        cout << "Illegal output format argument" << endl;
    }
}
```

Listing 5: BellmanFord.cu

```
#include <iostream>

#include "utils.cuh"

#include "utils.h"

using namespace std;
```

```

/*****************************************************************
SERIAL VERSION
*****************************************************************/

// cpu dijkstra
void dijkstra(int src, int numVertex, int* costMatrix, int* distance, int* parent) {
    priority_queue< pair<int, int>, vector <pair<int, int>>, greater<pair<int, int>> > heap; // heap of pair<distance to node, node>
    heap.push(make_pair(0, src)); // init heap
    distance[src * numVertex + src] = 0;
    while (!heap.empty()) {
        int u = heap.top().second; // extract min
        heap.pop();

        for (int v = 0; v < numVertex ; v++) { // loop through neighbors of u
            int weight = costMatrix[u * numVertex + v]; // cost from u to v

            if (weight != INF && distance[src * numVertex + v] > distance[src * numVertex + u] + weight) { // relax
                distance[src * numVertex + v] = distance[src * numVertex + u] + weight;
                parent[src * numVertex + v] = u;
                heap.push(make_pair(distance[src * numVertex + v], v)); // add to heap
            }
        }
    }
}

// run cpu dijkstra for very source
void runCpuDijkstra(int numVertex, int* costMatrix, int* distance, int* parent) {
    // time the algorithm
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    float duration;
    cudaEventRecord(start, 0);

    for (int src = 0; src < numVertex; src++) { // for every source
        dijkstra(src, numVertex, costMatrix, distance, parent); // call dijkstras
    }

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&duration, start, stop);
    cout << "Time: " << duration << "ms" << endl;
}

/*****************************************************************
NAIVE VERSION
*****************************************************************/

// find next node to visit
__device__
int extractMin(int numVertex, int* distance, bool* visited, int src) {
    int minNode = -1;
    int minDistance = INF;
    for (int i = 0; i < numVertex; i++) {
        if (!visited[src * numVertex + i] && distance[src * numVertex + i] < minDistance) {
            minDistance = distance[src * numVertex + i];
            minNode = i;
        }
    }
```

```
    return minNode;
}

__global__
void dijkstraNaive(int numVertex, int* h_costMatrix, bool* visited, int* distance, int* parent) {
    int src = blockIdx.x * blockDim.x + threadIdx.x; // thread src calculates shortest paths from src to every other vertex

    if (src < numVertex) {
        distance[src * numVertex + src] = 0;

        for (int i = 0; i < numVertex - 1; i++) {
            int u = extractMin(numVertex, distance, visited, src); // extract min
            if (u == -1) { // no min node to explore
                break;
            }
            visited[src * numVertex + u] = true; // mark u as visited
            for (int v = 0; v < numVertex; v++) { // loop through neighbors of u
                if (!visited[src * numVertex + v] && h_costMatrix[u * numVertex + v] != INF &&
                    distance[src * numVertex + v] > distance[src * numVertex + u] + h_costMatrix[u * numVertex + v]) { // relax

                    parent[src * numVertex + v] = u;
                    distance[src * numVertex + v] = distance[src * numVertex + u] + h_costMatrix[u * numVertex + v];
                }
            }
        }
    }
}

// run dijkstras on gpu
void runGpuDijkstra(int numVertex, int* costMatrix, bool* visited, int* distance, int* parent) {
    // time the algorithm
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    float duration;
    cudaEventRecord(start, 0);

    // allocate device pointers
    int* d_costMatrix;
    int* d_parent;
    int* d_distance;
    bool* d_visited;

    // allocate memory on gpu
    cudaCheck(cudaMalloc((void**)&d_costMatrix, numVertex * numVertex * sizeof(int)));
    cudaCheck(cudaMalloc((void**)&d_parent, numVertex * numVertex * sizeof(int)));
    cudaCheck(cudaMalloc((void**)&d_distance, numVertex * numVertex * sizeof(int)));
    cudaCheck(cudaMalloc((void**)&d_visited, numVertex * numVertex * sizeof(bool)));

    // copy from cpu to gpu
    cudaCheck(cudaMemcpy(d_costMatrix, costMatrix, numVertex * numVertex * sizeof(int), cudaMemcpyHostToDevice));
    cudaCheck(cudaMemcpy(d_parent, parent, numVertex * numVertex * sizeof(int), cudaMemcpyHostToDevice));
    cudaCheck(cudaMemcpy(d_distance, distance, numVertex * numVertex * sizeof(int), cudaMemcpyHostToDevice));
    cudaCheck(cudaMemcpy(d_visited, visited, numVertex * numVertex * sizeof(bool), cudaMemcpyHostToDevice));
```

```

    cout << "Kernel is executing" << endl;
    dijkstraNaive << <(numVertex - 1) / THREADS_PER_BLOCK + 1, THREADS_PER_BLOCK >> > (numVertex, d_costMatrix, d_visited, d_distance, d_parent);
    cudaCheck(cudaGetLastError()); // check if kernel launch failed
    cudaCheck(cudaDeviceSynchronize()); // wait for kernel to finish

    // copy from cpu to cpu
    cudaCheck(cudaMemcpy(distance, d_distance, numVertex * numVertex * sizeof(int), cudaMemcpyDeviceToHost));
    cudaCheck(cudaMemcpy(parent, d_parent, numVertex * numVertex * sizeof(int), cudaMemcpyDeviceToHost));

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&duration, start, stop);
    cout << "Time: " << duration << "ms" << endl;
}

int main(int argc, char* argv[]) {
    if (argc < 5) {
        cout << "Please provide an input file as a command line argument" << endl;
        return 0;
    }
    string pathDataset("../data/"); // path to dataset
    string algorithm(argv[1]); // algorithm 0=cpu, 1=naive
    string pathGraphFile(pathDataset + string(argv[2])); // input file
    string validate(argv[3]); // true=compare output with cpu, false=dont
    string outputFormat(argv[4]); // none=no output (to time the kernel), print=prints path on screen, write=write output to a file in the directory named output

    int numVertex, numEdges;

    int* h_costMatrix = fileToCostMatrix(pathGraphFile, numVertex, numEdges); // convert input file to adjacency list

    int* h_parent = (int*)malloc(numVertex * numVertex * sizeof(int));
    int* h_distance = (int*)malloc(numVertex * numVertex * sizeof(int));
    bool* h_visited = (bool*)malloc(numVertex * numVertex * sizeof(bool));

    fill(h_parent, h_parent + numVertex * numVertex, -1); // fill with -1
    fill(h_distance, h_distance + numVertex * numVertex, INF); // fill with INF
    fill(h_visited, h_visited + numVertex * numVertex, false); // fill with false

    if (algorithm == "0") { // cpu version
        runCpuDijkstra(numVertex, h_costMatrix, h_distance, h_parent);
    }
    else if (algorithm == "1") { // naive
        cout << "Warming up the GPU" << endl;
        for (int x = 0; x < NUM_ITERATION_WARMUP; x++) {
            warmpupGpu << < (numVertex - 1) / THREADS_PER_BLOCK + 1, THREADS_PER_BLOCK >> > ();
            cudaCheck(cudaGetLastError());
            cudaCheck(cudaDeviceSynchronize());
        }
        cout << "GPU is warmed up" << endl;

        runGpuDijkstra(numVertex, h_costMatrix, h_visited, h_distance, h_parent);
        if (validate == "true") {
            int* expParent = (int*)malloc(numVertex * numVertex * sizeof(int)); // expected parent
            int* expDistance = (int*)malloc(numVertex * numVertex * sizeof(int)); // expected distance
```

```
            fill(expDistance, expDistance + numVertex * numVertex, INF); // fill with INF
            fill(expParent, expParent + numVertex * numVertex, -1); // fill with -1
            runCpuDijkstra(numVertex, h_costMatrix, expDistance, expParent); // run on cpu
            validateDistanceAPSP(numVertex, expDistance, h_distance); // compare distance with expDistance
        }
    }

    if (outputFormat == "print") {
        printPathAPSP(numVertex, h_distance, h_parent); // print paths to screen
    }
    else if (outputFormat == "write") { // write output to a file named d{algorithm}.txt in output directory
        string pathOutputFile(string("../output/d") + algorithm + string(".txt"));
        cout << "Writing output to" << pathOutputFile << endl;
        writeOutPathAPSP(pathOutputFile, numVertex, h_distance, h_parent);
    }
    else if (outputFormat == "none") { // dont write out path

    }
    else {
        cout << "Illegal output format argument" << endl;
    }
}
```

Listing 6: Dijkstra.cu

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include "utils.cuh"

#include <iostream>

#include "utils.h"

using namespace std;

#define TILE_DIM 32

/************************************************************************
SERIAL VERSION
************************************************************************/
void runCpuFloydWarshall(int numVertex, int* distance, int* parent) {
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    float duration;
    cudaEventRecord(start, 0);

    cout << "running the algorithm on CPU" << endl;
    for (int k = 0; k < numVertex; k++) { // choose an intermediate node k
        for (int i = 0; i < numVertex; i++) { // choose a start node i
            for (int j = 0; j < numVertex; j++) { // loop through its neighbors
                int itoj = i * numVertex + j; // index for i->j
                int itok = i * numVertex + k; // index for i->k
                int ktoj = k * numVertex + j; // index for k->j

                // relax i->j using node k
                if (distance[itok] != INF && distance[ktoj] != INF && distance[itoj] > distance[itok] + distance[ktoj]) {
                    parent[itoj] = k;
                    distance[itoj] = distance[itok] + distance[ktoj];
```

```
                }
            }
        }
    }

    // time
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&duration, start, stop);
    cout << "Time: " << duration << "ms" << endl;
}

/***********************************************************************
SUPER NAIVE VERSION
***********************************************************************/
// one thread for each edge
__global__
void floydWarshallSuperNaive(int numVertex, int k, int* distance, int* parent) {
    int i = blockIdx.y * blockDim.y + threadIdx.y; // choose a start node i
    int j = blockIdx.x * blockDim.x + threadIdx.x; // choose a neighbor of i
    if (i < numVertex && j < numVertex) {
        int itoj = i * numVertex + j; // index for i->j
        int itok = i * numVertex + k; // index for i->k
        int ktoj = k * numVertex + j; // index for k->j

        // relax i->j using node k
        if (distance[itok] != INF && distance[ktoj] != INF && distance[itoj] >
    distance[itok] + distance[ktoj]) {
            parent[itoj] = k;
            distance[itoj] = distance[itok] + distance[ktoj];
        }
    }
}

// runs super naive on gpu
void runFloydWarshallSuperNaive(int numVertex, int* distance, int* parent) {
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    float duration;

    cudaEventRecord(start, 0);

    int* d_distance;
    int* d_parent;

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

    cout << "copying data to GPU" << endl;
    cudaCheck(cudaMemcpy(d_distance, distance, numVertex * numVertex * sizeof(int),
    cudaMemcpyHostToDevice));
    cudaCheck(cudaMemcpy(d_parent, parent, numVertex * numVertex * sizeof(int),
    cudaMemcpyHostToDevice));

    dim3 dimGrid((numVertex - 1) / TILE_DIM + 1, (numVertex - 1) / TILE_DIM + 1);
    dim3 dimBlock(TILE_DIM, TILE_DIM);
    // run kernel
    cout << "Kernel is executing" << endl;
    for (int k = 0; k < numVertex; k++) {
        floydWarshallSuperNaive << <dimGrid, dimBlock >> > (numVertex, k, d_distance
    , d_parent);
        cudaCheck(cudaGetLastError());
```

```
        cudaCheck(cudaDeviceSynchronize());
    }

    // copy results to CPU
    cout << "copying results to CPU" << endl;
    cudaCheck(cudaMemcpy(distance, d_distance, numVertex * numVertex * sizeof(int), cudaMemcpyDeviceToHost));
    cudaCheck(cudaMemcpy(parent, d_parent, numVertex * numVertex * sizeof(int), cudaMemcpyDeviceToHost));

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&duration, start, stop);
    cout << "Time: " << duration << "ms" << endl;
}

/**********************************************************************
SUPER NAIVE SHARED VERSION
**********************************************************************/
// one thread per edge but with shared memory
__global__
void floydWarshallSuperNaiveShared(int numVertex, int k, int* distance, int* parent) {
    int i = blockIdx.y; // choose a start node i
    int j = blockIdx.x * blockDim.x + threadIdx.x; // choose a neighbor j of i

    if (j < numVertex) {
        int itoj = numVertex * i + j; // index for i->j
        int itok = numVertex * i + k; // index for i->k
        int ktoj = numVertex * k + j; // index for k->j

        __shared__ int dist_itok; // shared variable to store i->k
        if (threadIdx.x == 0) {
            dist_itok = distance[itok];
        }
        __syncthreads();

        // relax i->j using node k
        if (dist_itok != INF && distance[ktoj] != INF && distance[itoj] > dist_itok + distance[ktoj]) {
            distance[itoj] = dist_itok + distance[ktoj];
            parent[itoj] = k;
        }
    }
}

// runs super naive shared on gpu
void runFloydWarshallSuperNaiveShared(int numVertex, int* distance, int* parent) {
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    float duration;

    cudaEventRecord(start, 0);

    int* d_distance;
    int* d_parent;

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

    cout << "copying data to GPU" << endl;
```

```
    cudaCheck(cudaMemcpy(d_distance, distance, numVertex * numVertex * sizeof(int), cudaMemcpyHostToDevice));
    cudaCheck(cudaMemcpy(d_parent, parent, numVertex * numVertex * sizeof(int), cudaMemcpyHostToDevice));

    dim3 dimGrid((numVertex - 1) / THREADS_PER_BLOCK + 1, numVertex);

    // run kernel
    cout << "Kernel is executing" << endl;
    for (int k = 0; k < numVertex; k++) {
        floydWarshallSuperNaiveShared << <dimGrid, THREADS_PER_BLOCK >> > (numVertex, k, d_distance, d_parent);
        cudaCheck(cudaGetLastError());
        cudaCheck(cudaDeviceSynchronize());
    }

    // copy results to CPU
    cout << "copying results to CPU" << endl;
    cudaCheck(cudaMemcpy(distance, d_distance, numVertex * numVertex * sizeof(int), cudaMemcpyDeviceToHost));
    cudaCheck(cudaMemcpy(parent, d_parent, numVertex * numVertex * sizeof(int), cudaMemcpyDeviceToHost));

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&duration, start, stop);
    cout << "Time: " << duration << "ms" << endl;
}

/**********************************************************************
NAIVE VERSION
**********************************************************************/
// one thread per vertex
__global__
void floydWarshallNaive(int numVertex, int k, int* distance, int* parent) {
    int i = blockIdx.x * blockDim.x + threadIdx.x; // choose a start node i
    if (i < numVertex) {
        for (int j = 0; j < numVertex; j++) { // loop through its neighbors
            int itoj = i * numVertex + j; // index for i->j
            int itok = i * numVertex + k; // index for i->k
            int ktoj = k * numVertex + j; //// index for k->j
            // relax i->j using node k
            if (distance[itok] != INF && distance[ktoj] != INF && distance[itoj] > distance[itok] + distance[ktoj]) {
                parent[itoj] = k;
                distance[itoj] = distance[itok] + distance[ktoj];
            }
        }
    }
}

// runs naive on gpu
void runFloydWarshallNaive(int numVertex, int* distance, int* parent) {
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    float duration;

    cudaEventRecord(start, 0);

    int* d_distance;
    int* d_parent;

    // allocate memory on GPU and copy data from CPU to GPU
    cout << "allocating data on GPU" << endl;
```

```
    cudaCheck(cudaMalloc((void**)&d_distance, numVertex * numVertex * sizeof(int)));
    cudaCheck(cudaMalloc((void**)&d_parent, numVertex * numVertex * sizeof(int)));

    cout << "copying data to GPU" << endl;
    cudaCheck(cudaMemcpy(d_distance, distance, numVertex * numVertex * sizeof(int), cudaMemcpyHostToDevice));
    cudaCheck(cudaMemcpy(d_parent, parent, numVertex * numVertex * sizeof(int), cudaMemcpyHostToDevice));

    // run kernel
    cout << "Kernel is executing" << endl;
    for (int k = 0; k < numVertex; k++) {
        floydWarshallNaive << <(numVertex - 1) / THREADS_PER_BLOCK + 1, THREADS_PER_BLOCK >> > (numVertex, k, d_distance, d_parent);
        cudaCheck(cudaGetLastError());
        cudaCheck(cudaDeviceSynchronize());
    }

    // copy results to CPU
    cout << "copying results to CPU" << endl;
    cudaCheck(cudaMemcpy(distance, d_distance, numVertex * numVertex * sizeof(int), cudaMemcpyDeviceToHost));
    cudaCheck(cudaMemcpy(parent, d_parent, numVertex * numVertex * sizeof(int), cudaMemcpyDeviceToHost));

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&duration, start, stop);
    cout << "Time: " << duration << "ms" << endl;
}


/**********************************************************************
TILED VERSION
**********************************************************************/
// tiled with global memory

// phase 1
__global__
void floydWarshallTiledPhase1(int numVertex, int primary_tile_number, int* distance, int* parent) {
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int i = primary_tile_number * blockDim.y + threadIdx.y; // node i
    int j = primary_tile_number * blockDim.x + threadIdx.x; // node j
    if(i<numVertex && j<numVertex){
        int itoj = i * numVertex + j; // index for i->j
        for (int k = 0; k < TILE_DIM; k++) { // run floyd warshall in the primary tile
            if (j-tx+k <numVertex && i-ty+k<numVertex &&
                distance[itoj - tx + k] != INF && distance[itoj - ty * numVertex + k * numVertex] != INF &&
                distance[itoj] > distance[itoj - tx + k] + distance[itoj - ty * numVertex + k * numVertex]) {

                distance[itoj] = distance[itoj - tx + k] + distance[itoj - ty * numVertex + k * numVertex];
                parent[itoj] = TILE_DIM * primary_tile_number + k;
            }
            // __syncthreads();
        }
    }
}

```

```
// phase 2
__global__
void floydWarshallTiledPhase2(int numVertex, int primary_tile_number, int* distance, int* parent) {
    // exclude primary tile
    if (blockIdx.x == primary_tile_number) {
        return;
    }
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int i, j;

    // 1st row of blocks for row
    if (blockIdx.y == 0) {
        i = primary_tile_number * blockDim.y + threadIdx.y;
        j = blockIdx.x * blockDim.x + threadIdx.x;
        if (i < numVertex && j < numVertex) {
            int itoj = i * numVertex + j; // index for i->j
            // relax edges in current tile using distance[i][k] from primary tile
            for (int k = 0; k < TILE_DIM; k++) {
                if (j-tx+k-blockIdx.x * blockDim.x + primary_tile_number * blockDim.x < numVertex && i-ty+k < numVertex &&
                    distance[itoj - tx + k - blockIdx.x * blockDim.x + primary_tile_number * blockDim.x] != INF &&
                    distance[itoj - ty * numVertex + k * numVertex] != INF &&
                    distance[itoj] > distance[itoj - tx + k - blockIdx.x * blockDim.x + primary_tile_number * blockDim.x]
                    + distance[itoj - ty * numVertex + k * numVertex]) {

                    distance[itoj] = distance[itoj - tx + k - blockIdx.x * blockDim.x + primary_tile_number * blockDim.x] + distance[itoj - ty * numVertex + k * numVertex];
                    parent[itoj] = TILE_DIM * primary_tile_number + k;
                }
                // __syncthreads();
            }
        }
    }

    // 2nd row of blocks for columns
    if (blockIdx.y == 1) {
        i = blockIdx.x * blockDim.y + threadIdx.y;
        j = primary_tile_number * blockDim.x + threadIdx.x;
        if (i < numVertex && j < numVertex) {
            int itoj = i * numVertex + j; // index for i->j
            // relax edges in current tile using distance[i][k] from primary tile
            for (int k = 0; k < TILE_DIM; k++) {
                if (j-tx+k < numVertex && i-(ty-k)- (blockIdx.x - primary_tile_number) * blockDim.x < numVertex &&
                    distance[itoj - tx + k] != INF &&
                    distance[itoj - (ty - k) * numVertex - (blockIdx.x - primary_tile_number) * blockDim.x * numVertex] != INF &&
                    distance[itoj] > distance[itoj - tx + k]
                    + distance[itoj - (ty - k) * numVertex - (blockIdx.x - primary_tile_number) * blockDim.x * numVertex]) {

                    distance[itoj] = distance[itoj - tx + k] + distance[itoj - ty * numVertex + k * numVertex - (blockIdx.x - primary_tile_number) * blockDim.x * numVertex];
                    parent[itoj] = TILE_DIM * primary_tile_number + k;
                }
                // __syncthreads();
            }
        }
```

```
    }
}

// phase 3
__global__
void floydWarshallTiledPhase3(int numVertex, int primary_tile_number, int* distance, int* parent) {
    // exclude primary tile, primary row and primary column
    if (blockIdx.x == primary_tile_number || blockIdx.y == primary_tile_number) {
        return;
    }
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < numVertex && j < numVertex) {
        int itoj = i * numVertex + j; // index for i->j
        // relax edges in current tile using distance[i][k] from primary column and distance[k][j] from primary row
        for (int k = 0; k < TILE_DIM; k++) {
            if (j-tx+k - blockIdx.x * blockDim.x + primary_tile_number * blockDim.x < numVertex &&
                i-ty+k - (blockIdx.y - primary_tile_number) * blockDim.y < numVertex &&
                distance[itoj - tx + k - blockIdx.x * blockDim.x + primary_tile_number * blockDim.x] != INF &&
                distance[itoj - ty * numVertex + k * numVertex - (blockIdx.y - primary_tile_number) * blockDim.y * numVertex] != INF &&
                distance[itoj] > distance[itoj - (tx - k) - (blockIdx.x - primary_tile_number) * blockDim.x]
                + distance[itoj - (ty - k) * numVertex - (blockIdx.y - primary_tile_number) * blockDim.y * numVertex]) {

                distance[itoj] = distance[itoj - tx + k - blockIdx.x * blockDim.x + primary_tile_number * blockDim.x] + distance[itoj - ty * numVertex + k * numVertex - (blockIdx.y - primary_tile_number) * blockDim.y * numVertex];
                parent[itoj] = TILE_DIM * primary_tile_number + k;
            }
        }
    }
}

// runs tiled version on gpu
void runFloydWarshallTiled(int numVertex, int* distance, int* parent) {
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    float duration;

    cudaEventRecord(start, 0);

    int* d_distance;
    int* d_parent;

    // allocate memory on GPU and copy data from CPU to GPU
    cudaCheck(cudaMalloc((void**)&d_distance, numVertex * numVertex * sizeof(int)));
    cudaCheck(cudaMalloc((void**)&d_parent, numVertex * numVertex * sizeof(int)));

    cout << "copying data to GPU" << endl;
    cudaCheck(cudaMemcpy(d_distance, distance, numVertex * numVertex * sizeof(int), cudaMemcpyHostToDevice));
    cudaCheck(cudaMemcpy(d_parent, parent, numVertex * numVertex * sizeof(int), cudaMemcpyHostToDevice));

    int numDiagonalTiles = (numVertex - 1) / TILE_DIM + 1;
```

```

    dim3 dimGridPhase1(1, 1), dimGridPhase2(numDiagonalTiles, 2), dimGridPhase3(
    numDiagonalTiles, numDiagonalTiles);
    dim3 dimBlock(TILE_DIM, TILE_DIM);

    cout << "Kernel is executing" << endl;
    for (int k = 0; k < numDiagonalTiles; k++) {
        floydWarshallTiledPhase1 << <  dimGridPhase1, dimBlock >> > (numVertex, k,
    d_distance, d_parent);
        cudaCheck(cudaGetLastError());
        cudaCheck(cudaDeviceSynchronize());
        floydWarshallTiledPhase2 << <  dimGridPhase2, dimBlock >> > (numVertex, k,
    d_distance, d_parent);
        cudaCheck(cudaGetLastError());
        cudaCheck(cudaDeviceSynchronize());
        floydWarshallTiledPhase3 << <  dimGridPhase3, dimBlock >> > (numVertex, k,
    d_distance, d_parent);
        cudaCheck(cudaGetLastError());
        cudaCheck(cudaDeviceSynchronize());
    }

    // copy results to CPU
    cout << "copying results to CPU" << endl;
    cudaCheck(cudaMemcpy(distance, d_distance, numVertex * numVertex * sizeof(int),
    cudaMemcpyDeviceToHost));
    cudaCheck(cudaMemcpy(parent, d_parent, numVertex * numVertex * sizeof(int),
    cudaMemcpyDeviceToHost));

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&duration, start, stop);
    cout << "Time: " << duration << "ms" << endl;
}

/*********************************************************************
TILED WIH SHARED MEMORY VERSION
*********************************************************************/

// phase 1
__global__
void floydWarshallTiledSharedPhase1(int numVertex, int primary_tile_number, int*
    distance, int* parent) {
    __shared__ int s_distance[TILE_DIM][TILE_DIM]; // primary tile

    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int i = TILE_DIM * primary_tile_number + ty;
    int j = TILE_DIM * primary_tile_number + tx;
    int itoj = i * numVertex + j;

    int shortestParent;
    if (i < numVertex && j < numVertex) {
        s_distance[ty][tx] = distance[itoj];
        shortestParent = parent[itoj];
    } else {
        s_distance[ty][tx] = INF;
        shortestParent = -1;
    }
    __syncthreads();

    #pragma unroll
    for (int k = 0; k < TILE_DIM; k++) { // run floyd warshall in primary tile
        __syncthreads();
        if (s_distance[ty][k] != INF &&
```

```
            s_distance[k][tx] != INF &&
            s_distance[ty][tx] > s_distance[ty][k] + s_distance[k][tx]) {

            s_distance[ty][tx] = s_distance[ty][k] + s_distance[k][tx];
            shortestParent = TILE_DIM * primary_tile_number + k;
        }
        __syncthreads();
    }
    if (i < numVertex && j < numVertex) {
        distance[itoj] = s_distance[ty][tx];
        parent[itoj] = shortestParent;
    }
}

// phase 2
__global__
void floydWarshallTiledSharedPhase2(int numVertex, int primary_tile_number, int* distance, int* parent) {
    if (blockIdx.x == primary_tile_number) { // exclude primary tile
        return;
    }
    __shared__ int s_distancePrimaryTile[TILE_DIM][TILE_DIM]; // primary tile
    __shared__ int s_distanceCurrentTile[TILE_DIM][TILE_DIM]; // current tile

    int i = TILE_DIM * primary_tile_number + threadIdx.y;
    int j = TILE_DIM * primary_tile_number + threadIdx.x;

    int idxPrimaryTile = i * numVertex + j;

    if (i < numVertex && j < numVertex) {
        s_distancePrimaryTile[threadIdx.y][threadIdx.x] = distance[idxPrimaryTile];
    }
    else {
        s_distancePrimaryTile[threadIdx.y][threadIdx.x] = INF;
    }
    __syncthreads();

    int idxCurrentTile;
    int shortestDistance;
    int shortestParent;

    if (blockIdx.y == 0) { // 1st row of blocks for rows
        i = TILE_DIM * primary_tile_number + threadIdx.y;
        j = TILE_DIM * blockIdx.x + threadIdx.x;
        idxCurrentTile = i * numVertex + j;

        if (i < numVertex && j < numVertex) {
            s_distanceCurrentTile[threadIdx.y][threadIdx.x] = distance[idxCurrentTile];
            shortestParent = parent[idxCurrentTile];
        }
        else {
            s_distanceCurrentTile[threadIdx.y][threadIdx.x] = INF;
            shortestParent = -1;
        }
        __syncthreads();

        shortestDistance = s_distanceCurrentTile[threadIdx.y][threadIdx.x];

        // relax edges in current tile using distance[i][k] from primary tile
        #pragma unroll
        for (int k = 0; k < TILE_DIM; k++) {
            int newDistance = s_distancePrimaryTile[threadIdx.y][k] + s_distanceCurrentTile[k][threadIdx.x];
            // __syncthreads();
```

```
            if (s_distancePrimaryTile[threadIdx.y][k] != INF &&
                s_distanceCurrentTile[k][threadIdx.x] != INF &&
                newDistance < shortestDistance) {

                shortestParent = TILE_DIM * primary_tile_number + k;
                shortestDistance = newDistance;
            }
            __syncthreads();
        }
    } else { // 2nd row of blocks for column
        i = TILE_DIM * blockIdx.x + threadIdx.y;
        j = TILE_DIM * primary_tile_number + threadIdx.x;
        idxCurrentTile = i * numVertex + j;

        if (i < numVertex && j < numVertex) {
            s_distanceCurrentTile[threadIdx.y][threadIdx.x] = distance[idxCurrentTile];
            shortestParent = parent[idxCurrentTile];
        }
        else {
            s_distanceCurrentTile[threadIdx.y][threadIdx.x] = INF;
            shortestParent = -1;
        }
        __syncthreads();
        shortestDistance = s_distanceCurrentTile[threadIdx.y][threadIdx.x];

        // relax edges in current tile using distance[i][k] from primary tile
        #pragma unroll
        for (int k = 0; k < TILE_DIM; k++) {
            int newDistance = s_distanceCurrentTile[threadIdx.y][k] + s_distancePrimaryTile[k][threadIdx.x];
            // __syncthreads();
            if (s_distancePrimaryTile[k][threadIdx.x] != INF &&
                s_distanceCurrentTile[threadIdx.y][k] != INF &&
                newDistance < shortestDistance) {

                shortestParent = TILE_DIM * primary_tile_number + k;
                shortestDistance = newDistance;
            }
            __syncthreads();
        }
    }
    if (i < numVertex && j < numVertex) {
        distance[idxCurrentTile] = shortestDistance;
        parent[idxCurrentTile] = shortestParent;
    }
}

// phase 3
__global__
void floydWarshallTiledSharedPhase3(int numVertex, int primary_tile_number, int* distance, int* parent) {
    // exclude primary tile, primary row and primary column
    if (blockIdx.x == primary_tile_number || blockIdx.y == primary_tile_number) {
        return;
    }

    __shared__ int s_distancePrimaryRow[TILE_DIM][TILE_DIM]; // primary row tile
    __shared__ int s_distancePrimaryCol[TILE_DIM][TILE_DIM]; // primary column tile
    __shared__ int s_distanceCurrentTile[TILE_DIM][TILE_DIM]; // current tile

    int i, j;

    i = TILE_DIM * primary_tile_number + threadIdx.y;
    j = TILE_DIM * blockIdx.x + threadIdx.x;
```

```
    if (i < numVertex && j < numVertex) {
        s_distancePrimaryRow[threadIdx.y][threadIdx.x] = distance[i * numVertex + j];
    }
    else {
        s_distancePrimaryRow[threadIdx.y][threadIdx.x] = INF;
    }


    i = TILE_DIM * blockIdx.y + threadIdx.y;
    j = TILE_DIM * primary_tile_number + threadIdx.x;
    if (i < numVertex && j < numVertex) {
        s_distancePrimaryCol[threadIdx.y][threadIdx.x] = distance[i * numVertex + j];
    }
    else {
        s_distancePrimaryCol[threadIdx.y][threadIdx.x] = INF;
    }

    i = TILE_DIM * blockIdx.y + threadIdx.y;
    j = TILE_DIM * blockIdx.x + threadIdx.x;
    int shortestParent;
    if (i < numVertex && j < numVertex) {
        s_distanceCurrentTile[threadIdx.y][threadIdx.x] = distance[i * numVertex + j];
        shortestParent = parent[i * numVertex + j];
    }
    else {
        s_distanceCurrentTile[threadIdx.y][threadIdx.x] = INF;
        shortestParent = -1;
    }

    __syncthreads();

    int shortestDist = s_distanceCurrentTile[threadIdx.y][threadIdx.x];
    // relax edges in current tile using distance[i][k] from primary column tile and distance[k][j] from primary row tile
    #pragma unroll
    for (int k = 0; k < TILE_DIM; k++) {
        int newDistance = s_distancePrimaryCol[threadIdx.y][k] + s_distancePrimaryRow[k][threadIdx.x];
        if (s_distancePrimaryCol[threadIdx.y][k] != INF &&
            s_distancePrimaryRow[k][threadIdx.x] != INF &&
            newDistance < shortestDist) {
            shortestParent = TILE_DIM * primary_tile_number + k;
            shortestDist = newDistance;
        }
    }
     // __syncthreads();
    if(i<numVertex && j<numVertex){ // write the tile to global memory
        distance[i * numVertex + j] = shortestDist;
        parent[i * numVertex + j] = shortestParent;
    }
}

// runs tiled with shared memory on gpu
void runFloydWarshallTiledShared(int numVertex, int* distance, int* parent) {
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    float duration;

    cudaEventRecord(start, 0);

    int* d_distance;
```

```
    int* d_parent;

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

    cout << "copying data to GPU" << endl;
    cudaCheck(cudaMemcpy(d_distance, distance, numVertex * numVertex * sizeof(int), cudaMemcpyHostToDevice));
    cudaCheck(cudaMemcpy(d_parent, parent, numVertex * numVertex * sizeof(int), cudaMemcpyHostToDevice));

    int numDiagonalTiles = (numVertex - 1) / TILE_DIM + 1;

    dim3 dimGridPhase1(1, 1), dimGridPhase2(numDiagonalTiles, 2), dimGridPhase3(numDiagonalTiles, numDiagonalTiles);
    dim3 dimBlock(TILE_DIM, TILE_DIM);

    cout << "Kernel is executing" << endl;
    for (int k = 0; k < numDiagonalTiles; k++) {
        floydWarshallTiledSharedPhase1 << <  dimGridPhase1, dimBlock >> > (numVertex, k, d_distance, d_parent);
        cudaCheck(cudaGetLastError());
        cudaCheck(cudaDeviceSynchronize());
        floydWarshallTiledSharedPhase2 << <  dimGridPhase2, dimBlock >> > (numVertex, k, d_distance, d_parent);
        cudaCheck(cudaGetLastError());
        cudaCheck(cudaDeviceSynchronize());
        floydWarshallTiledSharedPhase3 << <  dimGridPhase3, dimBlock >> > (numVertex, k, d_distance, d_parent);
        cudaCheck(cudaGetLastError());
        cudaCheck(cudaDeviceSynchronize());
    }

    // copy results to CPU
    cout << "copying results to CPU" << endl;
    cudaCheck(cudaMemcpy(distance, d_distance, numVertex * numVertex * sizeof(int), cudaMemcpyDeviceToHost));
    cudaCheck(cudaMemcpy(parent, d_parent, numVertex * numVertex * sizeof(int), cudaMemcpyDeviceToHost));

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&duration, start, stop);
    cout << "Time: " << duration << "ms" << endl;
}

int main(int argc, char* argv[]) {

    if (argc < 5) {
        cout << "Please provide proper command line arguments" << endl;
        return 0;
    }
    string pathDataset("../data/");
    string algorithm(argv[1]);
    string pathGraphFile(pathDataset+string(argv[2]));
    string validate(argv[3]);
    string outputFormat(argv[4]);

    int numVertex, numEdges;
    int* costMatrix = fileToCostMatrix(pathGraphFile, numVertex, numEdges);

    int* parent = (int*)malloc(numVertex * numVertex * sizeof(int));
    int* distance = (int*)malloc(numVertex * numVertex * sizeof(int));
```

```

    APSPInitDistanceParent(numVertex, costMatrix, distance, parent);

    if (algorithm == "0") {
        runCpuFloydWarshall(numVertex, distance, parent);
    } else{
        cout << "Warming up the GPU" << endl;
        for (int x = 0; x < NUM_ITERATION_WARMUP; x++) {
            warmpupGpu << < (numVertex - 1) / THREADS_PER_BLOCK + 1,
    THREADS_PER_BLOCK >> > ();
            cudaCheck(cudaGetLastError());
            cudaCheck(cudaDeviceSynchronize());
        }
        cout << "GPU is warmed up" << endl;
        if (algorithm == "1") {
            runFloydWarshallSuperNaive(numVertex, distance, parent);
        } else if (algorithm == "2") {
            runFloydWarshallNaive(numVertex, distance, parent);
        } else if (algorithm == "3") {
            runFloydWarshallSuperNaiveShared(numVertex, distance, parent);
        } else if (algorithm == "4") {
            runFloydWarshallTiled(numVertex, distance, parent);
        } else if (algorithm == "5") {
            runFloydWarshallTiledShared(numVertex, distance, parent);
        }

        if (validate == "true") {
            int* exp_parent = (int*)malloc(numVertex * numVertex * sizeof(int));
            int* exp_distance = (int*)malloc(numVertex * numVertex * sizeof(int));
            if (exp_parent == NULL || exp_distance == NULL) {
                cout << "Malloc failed" << endl;
                return 0;
            }
            APSPInitDistanceParent(numVertex, costMatrix, exp_distance, exp_parent);
            runCpuFloydWarshall(numVertex, exp_distance, exp_parent);
            validateDistanceAPSP(numVertex, exp_distance, distance);
        }
    }
    //
    if (outputFormat == "print") {
        printPathAPSP(numVertex, distance, parent);
    } else if (outputFormat == "write") {
        string pathOutputFile(string("../output/fw") + algorithm + string(".txt"));
        cout << "Writing output to" << pathOutputFile << endl;
        writeOutPathAPSP(pathOutputFile, numVertex, distance, parent);
    } else if (outputFormat == "none") {

    } else {
        cout << "Illegal output format argument" << endl;
    }
}
```

Listing 7: FloydWarshall.cu

```
import sys
import random

def parseDIMACS(fileName):
  file = open(fileName, "r")
  lines = file.readlines()

  for i, line in enumerate(lines):
    if i==4:
      tokens = line.split()
      numVertex, numEdges = tokens[2], tokens[3]
```

```python
      lines[i] = numVertex + " " + numEdges + "\n"
    elif i>=7:
      _, src, dest, cost = line.split()
      lines[i] = str(int(src)-1) + " " + str(int(dest)-1) + " " + cost + "\n"
    else:
      lines[i] = ''

  file = open(fileName, "w")
  file.writelines(lines)
  file.close()

def addWeights(fileName, weightMin=1, weightMax=100):
  file = open(fileName, "r")
  lines = file.readlines()

  for i, line in enumerate(lines):
    if i==2:
      tokens = line.split()
      numVertex, numEdges = tokens[2], tokens[4]
      lines[i] = numVertex + " " + numEdges + "\n"
    elif i>=4:
      src, dest = line.split()
      lines[i] = src + " " + dest + " " + str(random.randint(weightMin, weightMax)) + "\n"
    else:
      lines[i] = ''

  file = open(fileName, "w")
  file.writelines(lines)
  file.close()

def replaceWeights(fileName, weightMin=1, weightMax=100):
  file = open(fileName, "r")
  lines = file.readlines()

  for i, line in enumerate(lines):
    if i:
      src, dest, cost = line.split()
      lines[i] = src + " " + dest + " " + str(random.randint(weightMin, weightMax)) + "\n"

  file = open(fileName, "w")
  file.writelines(lines)
  file.close()

def createRandomGraph(numVertex, fileName):
  lines = [str(numVertex)+" "+str(numVertex*numVertex-numVertex)]
  for i in range(numVertex):
    for j in range(numVertex):
      if i != j:
        line = str(i) + " " + str(j) + " " + str(random.randint(1, 100)) + "\n"
        lines.append(line)
  file = open(fileName, "w")
  file.writelines(lines)
  file.close()


if __name__ == "__main__":
  _, action, *rest = sys.argv
  if action == 'parse':
    parseDIMACS(rest[0])
  elif action == 'random':
    createRandomGraph(int(rest[0]), rest[1])
  else:
    fileName, weightMin, weightMax = rest
```

```python
    if action == 'add':
      addWeights(fileName, int(weightMin), int(weightMax))
    elif action == 'replace':
      replaceWeights(fileName, int(weightMin), int(weightMax))
```
Listing 8: utils.py