DA3
All Pairs Shortest Path Problem in CUDA

-CED17I023
J.Veeren Chandrahas

Table of Contents

Problem Statement:

The distance from city *A* to city *B* is given by the value of the element in the row corresponding to *A* and column corresponding to *B*. The route between two cities often passes through other cities in the table. The all pairs shortest path problem is concerned with generating such a table.

The input to the problem may be expressed as a weighted, directed graph, where the vertices are cities and the edges are distances between cities that are directly connected. The weighed, directed graph may be represented by an $N \times N$ *adjacency matrix*, where N is the number of vertices. The value of matrix element i, j is the weight (distance) from vertex i to vertex j if this edge exists. A solution to the all pairs shortest path problem is a table showing the shortest path between all pairs of vertices. The length of the path is given by the sum of the weights on the edges that form the path.

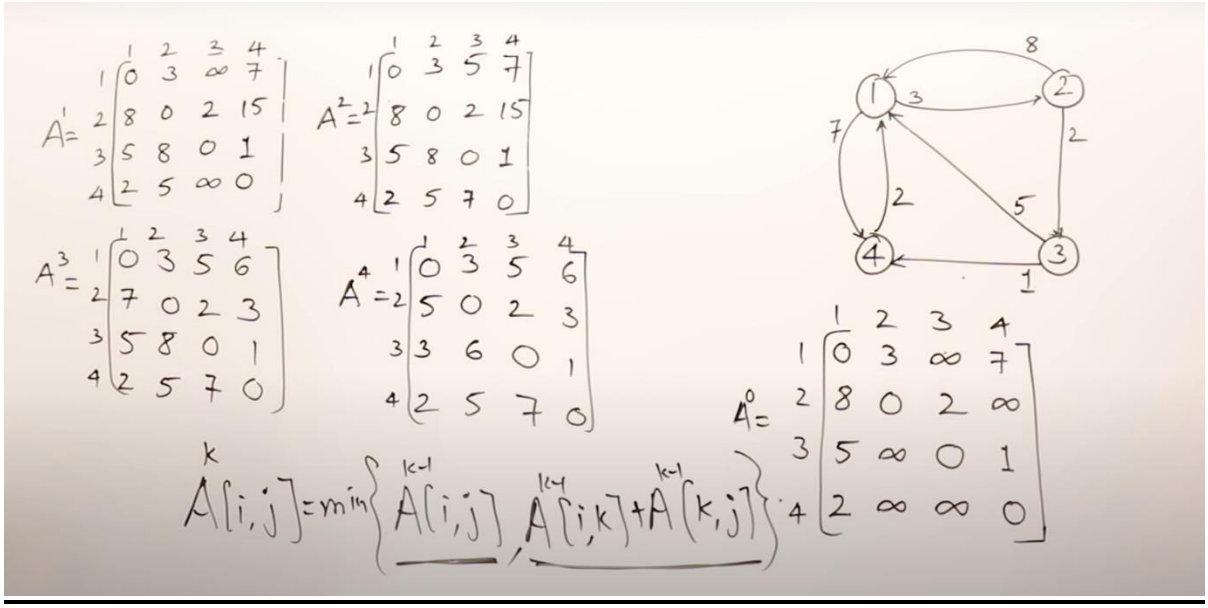
An algorithm to solve the all pairs shortest path problem is Floyd's algorithm. This transforms the input adjacency matrix into a matrix containing the length of the shortest path between all pairs of vertices. A sequential version of Floyd's algorithm may be described by the following pseudocode and has a time complexity of O(N3)

Input Data

	0	1	2	3	4	5
0	0	2	5	-1	-1	-1
1	-1	0	7	1	-1	8
2	-1	-1	0	4	-1	-1
3	-1	-1	-1	0	3	-1
4	-1	-1	2	-1	0	3
5	-1	5	-1	2	4	0

Solution

	0	1	2	3	4	5
0	0	2	5	3	6	9
1	-1	0	6	1	4	7
2	-1	15	0	4	7	10
3	-1	11	5	0	3	6
4	-1	8	2	5	0	3
5	-1	5	6	2	4	0



1. Introduction:

The shortest path problem is about finding a path between two nodes in a graph such that the path cost is minimized. One example of this problem could be finding the fastest route from one city to another by car, train or airplane.
The Floyd-Warshall algorithm is an algorithm that solves this problem. It works for weighted graphs with positive or negative weights but not for graphs with negative cycles.
It works by comparing all possible paths between all vertex pairs in the graph. A version of the algorithm implemented in the C language can be seen in the figure below.

```
void ST_APSP(int *mat, const size_t N)
{
    for (int k = 0; k < N; k++)
        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++)
            {
                int i0 = i * N + j;
                int i1 = i * N + k;
                int i2 = k * N + j;
                if (mat[i1] != -1 && mat[i2] != -1)
                {
                    int sum = (mat[i1] + mat[i2]);
                    if (mat[i0] == -1 || sum < mat[i0])
                        mat[i0] = sum;
                }
            }
}
```

The notations used in this figure; "k", "i", "j", "N" and "mat[...]" will be used throughout the report.
For each k all of the current values (mat[i*N,j]) of the matrix is compared to the sum of two other values in the matrix: mat[k*N,j] + mat[i*N,k]. Once the outer loop has run N times all paths between all vertex pairs have been compared.
The purpose of the lab was to parallelize this algorithm with the use CUDA API. This report will show the results of the parallel version that was implemented and explain the design of the parallel algorithm.

2. Results:

To measure the performance of our algorithm, we performed various tests by varying the number of processors and the size of the initial matrix. For simplicity, we have use values of N (N=1000) which are divisible by the numbers of processors (p=1,2,4,8,16,32,64,128,256,512).

2.1. Total time:

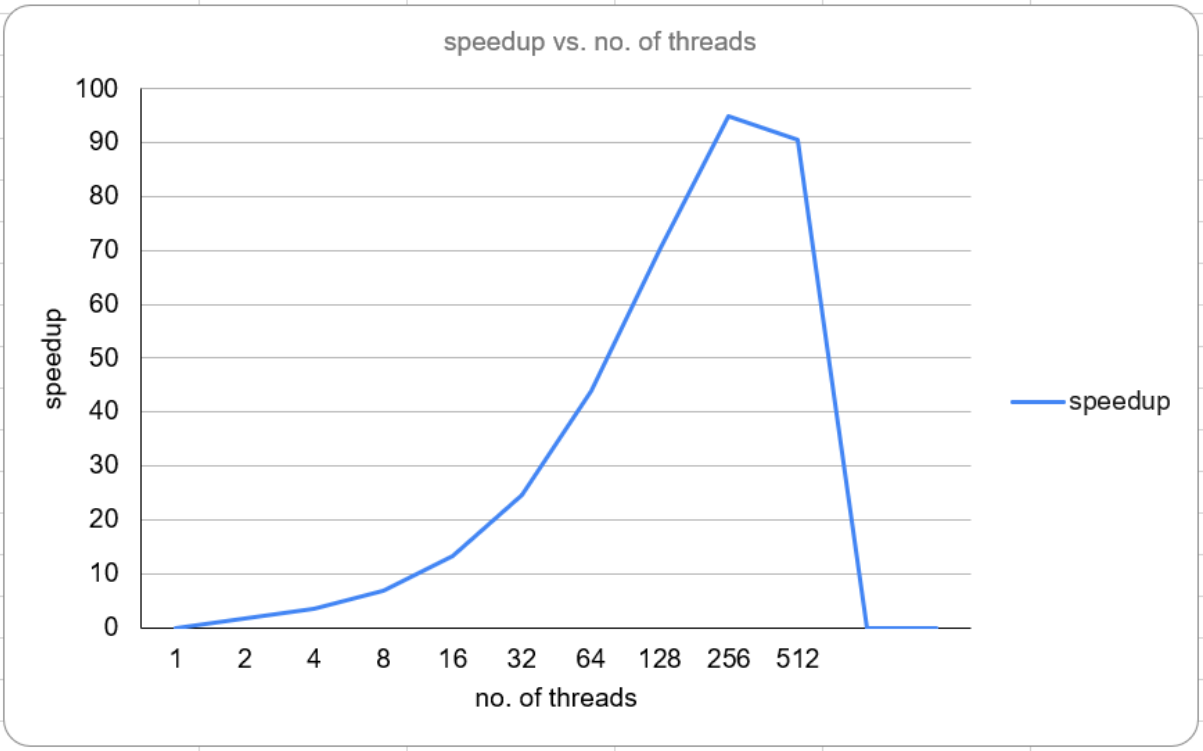
2.1.1. Table:

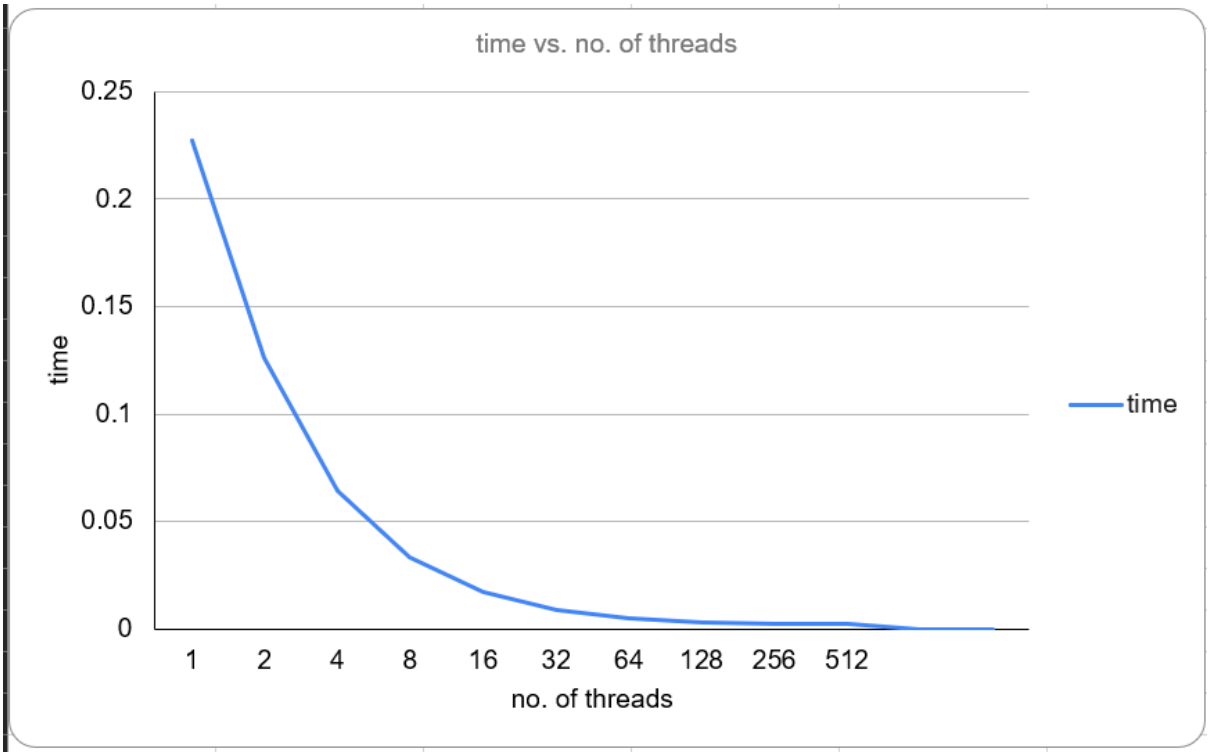
This table summarizes the total execution time of sequential and parallel programs based on the numbers of processors.

no. of threads	time	speedup	parallel fraction
1	0.227349		
2	0.126046	1.80369865	0.891167324
4	0.064306	3.535424377	0.956198033
8	0.033153	6.857569451	0.976200844
16	0.017195	13.22180867	0.985991874
32	0.009221	24.65556881	0.990390928
64	0.005171	43.96615742	0.992767221
128	0.003226	70.47396156	0.993572648
256	0.002392	95.04556856	0.99335904
512	0.002514	90.43317422	0.990877418

2.1.2. Graph:

This graph shows the total execution time of programs depending on the size of the initial matrix, N. Moreover, each curve is associated with a different number of processors.





2.1.3. Analyse:

Thanks to this information, we can see that the execution time fluctuates as the number of processors increases.

2.2. Speedup:

Speed Up is defined as the ratio of the serial run time of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem on p processors.

2.2.1. Table:

This table summarizes the speedup of parallel programs based on the numbers of processors and the size of the initial matrix. This allows us to compare the speed of execution of a parallel program with a sequential program.

Time taken for 1 thread= 0.227349

Time taken for 512 thread= 0.002514

Speedup =T (1)/T (512) = 90.43317422

Parallelization Factor (f) = (1 - T (512)/T (1))/ (1 - (1/512)) = 0.990877418

3. Observations:

First, we implemented a basic version of the kernel. This kernel was implemented without any thought of shared memory or coalesced memory access. The speedup compared to the sequential algorithm was still very high.

It is arguable that this basic version made use of coalesced memory access to some
degree. The (x,y) process accessed the matrix elements (x,y),(k.y) and (x,k) which is a very structured way to access the memory. There were however a lot of unnecessary accesses to global memory.

4.Appendix:

```
%%cu
#define inf 9999

__global__ void funct(int n, int k, float* x, int* qx) {

    int ix = blockIdx.x * blockDim.x + threadIdx.x;
    int j = ix & (n - 1);
    float temp2 = x[ix - j + k] + x[k * n + j];
    if (x[ix] > temp2) {
        x[ix] = temp2;
        qx[ix] = k;
    }
    if (x[ix] == inf) {
        qx[ix] = -2;
    }
}

__global__ void funct2(int n, int k, float* x, int* qx) {
    int ix = blockIdx.x * blockDim.x + threadIdx.x;
    int j = ix & (n - 1);
    float temp2 = x[ix - j + k] + x[k * n + j];
    if (x[ix] > temp2) {
        x[ix] = temp2;
        qx[ix] = k;
    }
}
```

```
gettimeofday(&third, &tzp2);
////////////////////////////First Mem Copy//////////////////
gettimeofday(&first, &tzp);
cudaMemcpy(dev_x, host_A, n * n * sizeof (float), cudaMemcpyHostToDevice);
cudaMemcpy(dev_qx, host_Q, n * n * sizeof (int), cudaMemcpyHostToDevice);
gettimeofday(&second, &tzp);
if (first.tv_usec > second.tv_usec) {
    second.tv_usec += 1000000;
    second.tv_sec--;
}
lapsed.tv_usec = second.tv_usec - first.tv_usec;
lapsed.tv_sec = second.tv_sec - first.tv_sec;
printf("First Transfer CPU to GPU  Time elapsed: %lu, %lu s\n", lapsed.tv_sec, lapsed.tv_usec);
////////////////////////////////////////////////////GPU Calculation////////////////////////////////
int gputhreads = 1;
bk = (int) (n * n / gputhreads);

if (bk > 0) {
    gputhreads = 1;
} else {
    bk = 1;
    gputhreads = n*n;
}

printf(" \n");
printf("BLOCKS :   %d      GPU THREADS:     %d \n", bk, gputhreads);
printf(" \n");
gettimeofday(&first, &tzp);
funct << <bk, gputhreads>>>(n, k, dev_x, dev_qx);
for (k = 1; k < n; k++) {
    funct2 << <bk, gputhreads>>>(n, k, dev_x, dev_qx);
}
cudaThreadSynchronize();
gettimeofday(&second, &tzp);
if (first.tv_usec > second.tv_usec) {
```

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

    struct timeval first, second, lapsed, third, fourth, lapsed2;
    struct timezone tzp, tzp2;
    float *host_A, *host_D;
    int *host_Q;
    float *dev_x;
    int *dev_qx;
    float *A;
    int *Q;
    float *D;
    float tolerance = 0.001;

    int i, j, bk;
    int k = 0;
    int n = 512; // atoi(argv[1]);

    printf("\n");
    printf("RUNNING WITH %d VERTICES \n", n);
    printf("\n");

    cudaMalloc(&dev_x, n * n * sizeof (float));
    cudaMalloc(&dev_qx, n * n * sizeof (float));

    //CPU arrays
    A = (float *) malloc(n * n * sizeof (float)); //initial matrix A
    D = (float *) malloc(n * n * sizeof (float)); //initinal matrix D
    Q = (int *) malloc(n * n * sizeof (int)); //initinal matrix Q
```

```
srand(time(NULL));
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        if (i == j) {
            A[i * n + j] = 0;
        } else {
            A[i * n + j] = 1200 * (float) rand() / RAND_MAX + 1;
            if (A[i * n + j] > 1000) {
                A[i * n + j] = inf;
            }
        }
    }
}
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        Q[i * n + j] = -1;
    }
}
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        D[i * n + j] = A[i * n + j];
    }

}
for (i = 0; i < n; i++) //copy of A to host_A
{
    for (j = 0; j < n; j++) {
        host_A[i * n + j] = A[i * n + j];
    }
```