Lab 3 report – team 5

All Pairs Shortest Path Problem in CUDA

Summary

Christopher BELINGUIER
Olle GRAHN

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.

```
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. The GPU hardware

- *What are your major observations on memory bandwidths from the program output?*

Our major observations on memory bandwidths is that the device to device bandwidth (26004 MB/s) is much higher than host to device (6106 MB/s) and device to host bandwidth (6538 MB/s).

- *What is the Compute Capability of the GPU ?*

The compute capability of the GPU of our computer is equal to 5.

- *What are the differences between GPUs with different Compute Capabilities ?*

The differences between GPUs with different compute capabilities is the general specifications and available features.

3. Benchmark of the basic kernel

To measure the performance of our algorithm, we performed various tests by varying the block dimension and the size of the initial matrix. To compare the performance with the previous algorithms made with MPI and OpenMP, we have use the same values of N (N=1200,2400,4800). Moreover, we have chosen to use 2D blocks to adapt better to the generated matrix. Finally, to execute our algorithm, we just have to choose the size of the matrix and the size of the block. Indeed, the number of necessary block is calculated directly in our program.

3.1. Total time

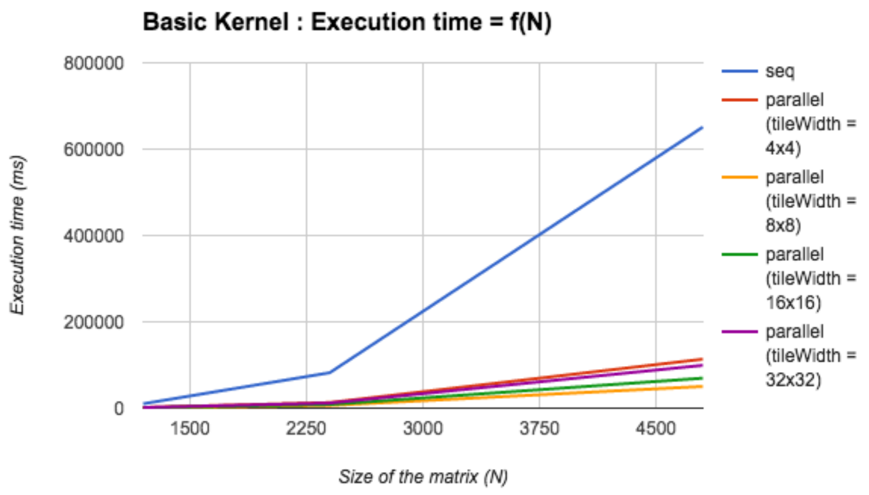
3.1.1. Table

This table summarizes the total execution time (in ms) of sequential and parallel programs based on the block dimension and the size of the initial matrix.

Block size / N	1200	2400	4800
Sequential	10243,34277	82070,78125	651103,9375
Parallel (4x4)	1760,30603	12949,33789	113495,0156
Parallel (8x8)	896,333008	6345,934082	50370,58594
Parallel (16x16)	1186,854004	8491,957031	69482,96875
Parallel (32x32)	1536,629028	11825,89746	99259,76563

3.1.2. 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 block dimension.



3.1.3. Analyse

Thanks to this information, we can see that the execution time grows when the matrix size grow and depends on the block size. For this algorithm and matrix size (N) between 1200 and 4800, use blocks of size 8x8 is the best solution.

3.2. Speedup

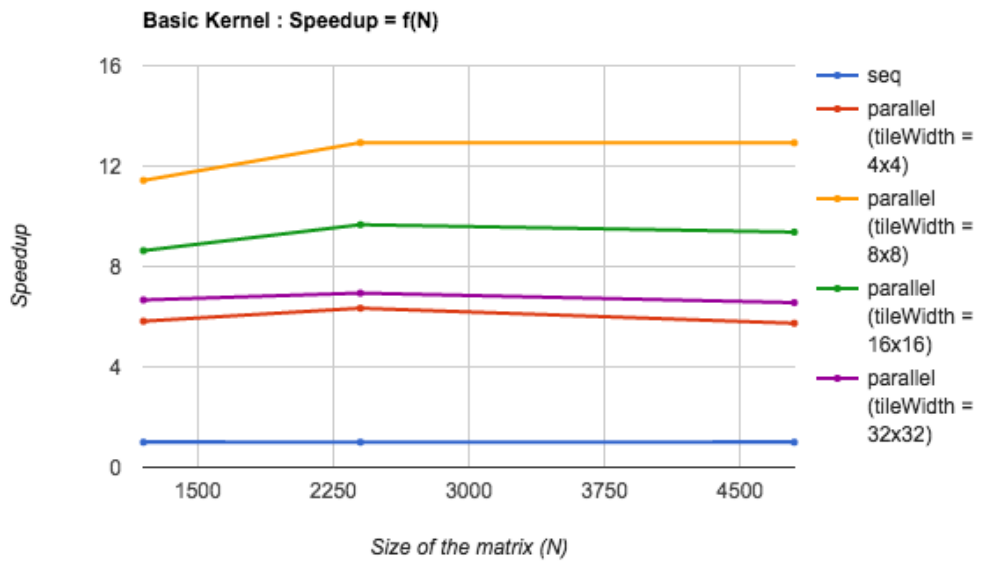
3.2.1. Table

This table summarizes the speedup of parallel programs based on the block dimension and the size of the initial matrix. This allows us to compare the speed of execution of a parallel program with a sequential program. In fact :

Speedup	1200	2400	4800
Sequential	1	1	1
Parallel (4x4)	5,819069297	6,337836107	5,736850503
Parallel (8x8)	11,42805484	12,93281339	12,92627285
Parallel (16x16)	8,630667916	9,664530914	9,370698305
Parallel (32x32)	6,666113022	6,939919911	6,559595758

3.2.2. Graph

This graph shows the speedup of programs depending on the size of the initial matrix, N. Moreover, each curve is associated with a different block dimension.



3.2.3. Analyse

Thanks to this information, we can see that the gain in execution speed does not vary much with the size of the matrix for a given block dimension. Similarly, we see that the speedup not increases when the block dimension increases. Indeed, if the block size is too small (4x4), we have to use too many blocks. Also, if the block size is too large (32x32), there are too many threads per block and the occupancy is too low to have an efficient algorithm.

4. Benchmark of the optimized kernel

To measure the performance of our algorithm, we performed various tests by varying the block dimension and the size of the initial matrix. To compare the performance with the performance of the basic kernel, we have use the same values of N (N=1200,2400,4800) and the same block dimension. In this version, we use coalesced memory accesses and shared memory in order to optimize the performance of the basic kernel.

To find the most effective method, we have perform multiple rounds of code profiling, analysis, code rewriting and testing. In fact, the basic Floyd kernel is bandwidth constrained, that why we have try to decrease the number of global memory transactions in order to improve the performance of the algorithm. We collect all this information thanks to the Nvidia Visual Profiler.

4.1. Total time

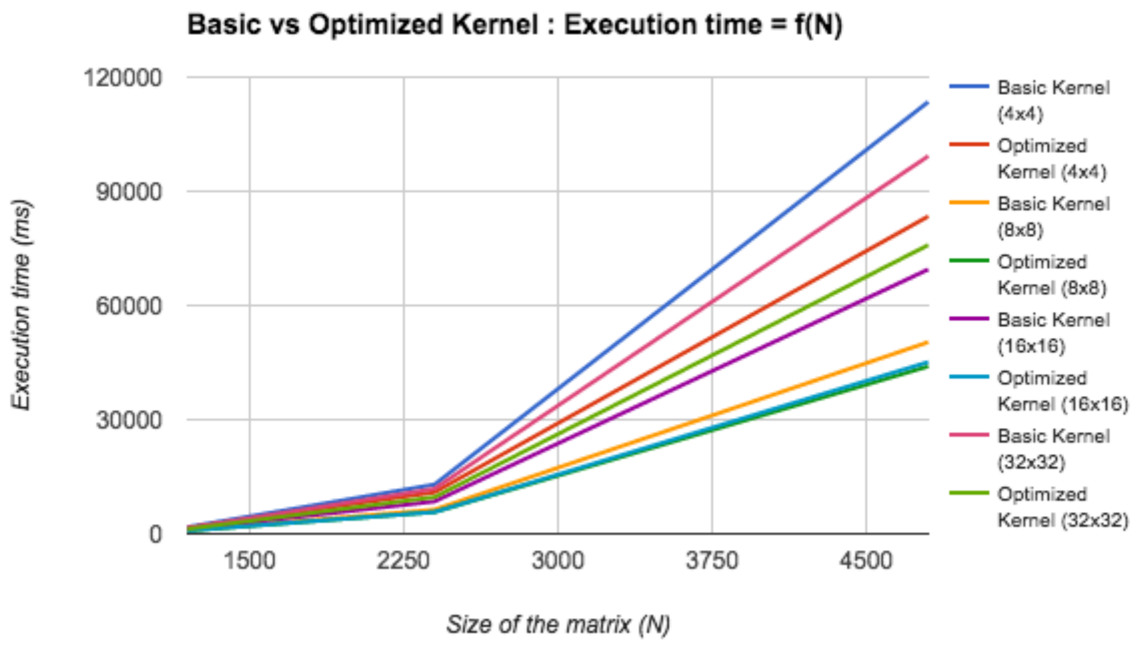
4.1.1. Table

This table summarizes the total execution time (in ms) of sequential and parallel programs with the basic kernel and the optimized kernel, based on the block dimension and the size of the initial matrix.

Execution time (ms)	1200	2400	4800
Sequential	10243,34277	82070,78125	651103,9375
Basic Kernel (4x4)	1760,30603	12949,33789	113495,0156
Optimized Kernel (4x4)	1533,324951	10912,98731	83424,33594
Basic Kernel (8x8)	896,333008	6345,934082	50370,58594
Optimized Kernel (8x8)	805,109009	5693,717773	44004,69531
Basic Kernel (16x16)	1186,854004	8491,957031	69482,96875
Optimized Kernel (16x16)	841,463989	5731,246094	45127,82031
Basic Kernel (32x32)	1536,629028	11825,89746	99259,76563
Optimized Kernel (32x32)	1271,671021	9630,238281	75893,85156

4.1.2. 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 block dimension use in the basic kernel and the optimized kernel.



4.1.3. Analyse

Thanks to this information, we can see that the execution time grows when the matrix size grow and depends on the block size. Moreover, we can see that the use of coalesced memory accesses and shared memory reduced the execution time for all block sizes.

For this algorithm and matrix size (N) between 1200 and 4800, use blocks of size 8x8 is still the best solution. Nevertheless, we can see that the execution time with blocks of size 16x16 has been improved. Indeed, the execution time of the algorithm with 8x8 blocks and 16x16 size are almost similar when we use coalesced memory accesses and shared memory.

4.2. Speedup

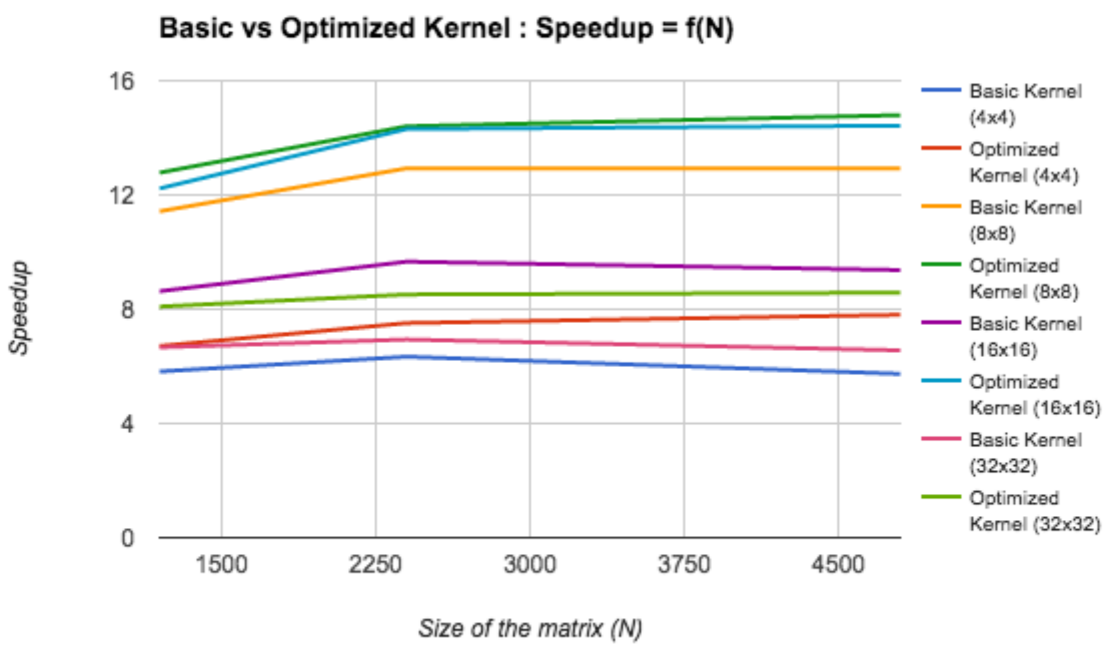
4.2.1. Table

This table summarizes the speedup of parallel programs with the basic kernel and the optimized kernel, based on the block dimension and the size of the initial matrix. This allows us to compare the speed of execution of a parallel program with a sequential program. In fact :

Speedup	1200	2400	4800
Basic Kernel (4x4)	5,819069297	6,337836107	5,736850503
Optimized Kernel (4x4)	6,710586532	7,515002328	7,804724247
Basic Kernel (8x8)	11,42805484	12,93281339	12,92627285
Optimized Kernel (8x8)	12,78026907	14,40379173	14,79623783
Basic Kernel (16x16)	8,630667916	9,664530914	9,370698305
Optimized Kernel (16x16)	12,22810471	14,3094754	14,42799437
Basic Kernel (32x32)	6,666113022	6,939919911	6,559595758
Optimized Kernel (32x32)	8,091329909	8,516001641	8,57913947

4.2.2. Graph

This graph shows the speedup of programs depending on the size of the initial matrix, N. Moreover, each curve is associated with a different block dimension use in the basic kernel and the optimized kernel.



4.2.3. Analyse

Thanks to this information, we can see that the gain in execution speed does not vary much with the size of the matrix for a given block dimension. Similarly, we see that the speedup not always increases when the block dimension increases. Indeed, if the block size is too small (4x4), we have to use too many blocks. Also, if the block size is too large (32x32), there are too many threads per block and the occupancy is too low to have an efficient algorithm.

However, we can see that the use of coalesced memory accesses and shared memory increases the speedup for all block sizes.

5. Method

For each certain iteration (k) of the algorithm the calculations are independent. Calculating the value of mat[i*N, j] will not affect the calculation of mat[i'*N, j'] for a specific k. But the values calculated for the following iterations of k will depend upon the previous iterations. This means that the iterations of k has to be calculated in sequence but the values of the matrix within that iterations can be calculated in sequence.

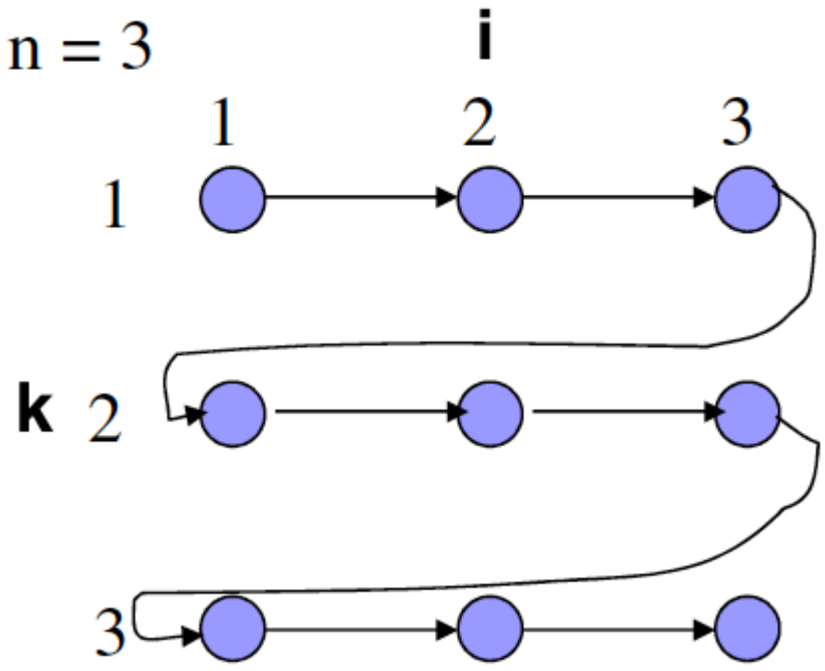


Figure 1 : Iteration-space traversal graph

5.1. Partitioning

Each iteration of k has to be done in sequence but for a specific k it is possible to partition the algorithm into a task for each matrix element. This is possible since there are no loop-carried dependencies in the i and j loops.

5.2. Communication

Since all threads have access to the data that stores the matrix the different tasks or threads do not need to communicate apart from writing their calculated value to the matrix.

However, improvements were made by loading data from global memory into shared memory. This improves performance since shared memory access is much faster than global memory access.

All threads wait for the load into shared memory to be completed. When all of the loading threads are finished this is broadcasted and all threads continue with the calculation.

5.3. Agglomération

Allocating threads on the GPU requires very little overhead. Therefore the tasks were not agglomerated but instead each task was distributed to its own thread.

5.4. Mapping

The load of certain threads is a bit higher than for others. The first row and column of each thread block loads a block sized part of the kth row and kth column into shared memory. When this is done all threads proceed with the APSP calculations.

5.5. Final algorithm

The final algorithm first loads the matrix into the global memory of the GPU. It then runs the k iterations in sequence. For each k iteration one thread is created for each matrix element. The first row and column of each thread block loads a block sized part of the kth row and kth column into shared memory. When this is done all threads proceed with the APSP calculations. Finally the CPU fetches the results from the GPU and loads it back into RAM.

6. Discussion

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.

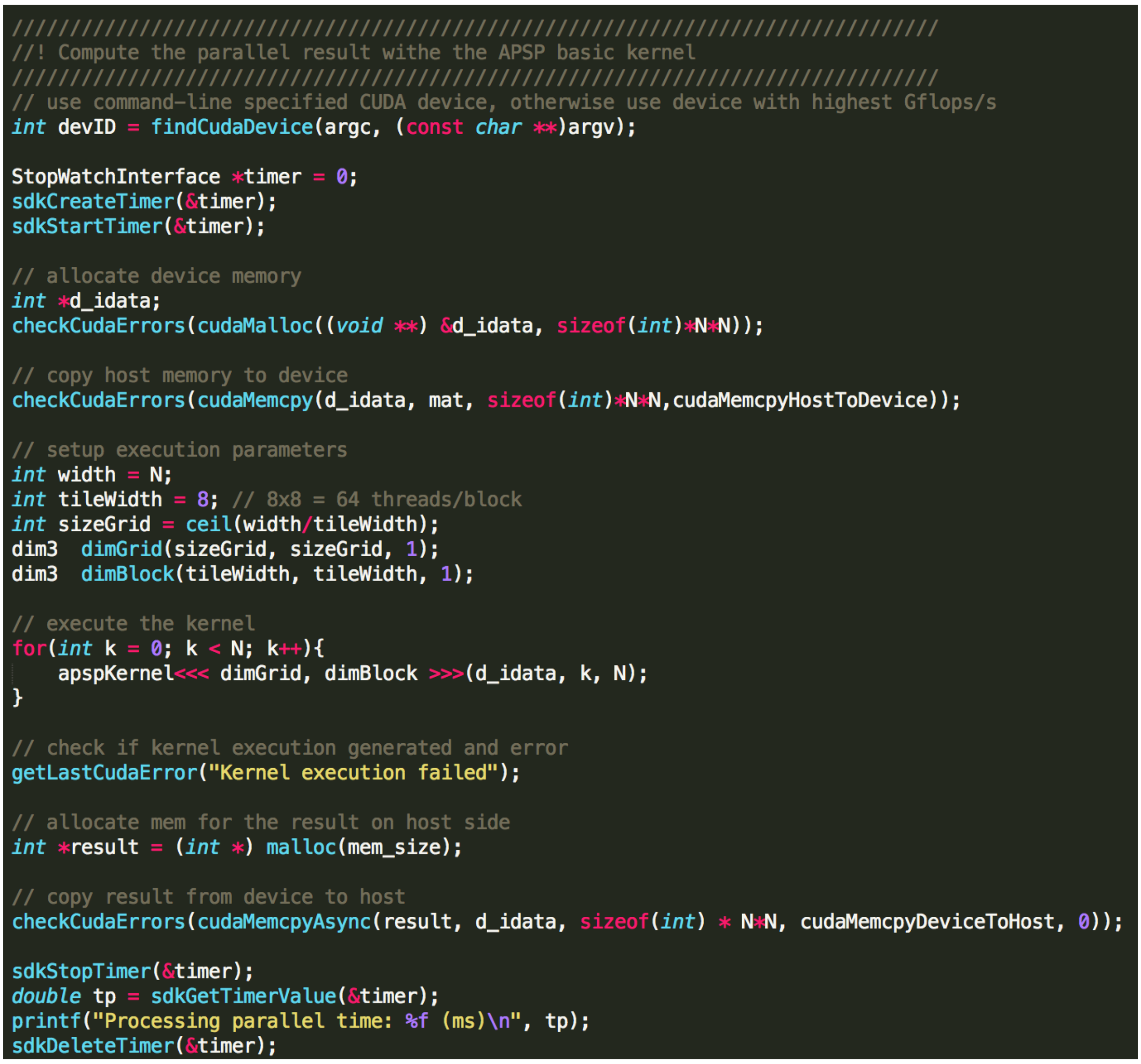
We then proceeded to optimizing the kernel by reducing the number of accesses to global memory by loading data into shared memory. This was done with the use of an if statement that made sure that only the first row and column of each thread block loads a block sized part of the kth row and kth column into shared memory. This improved the speedup of a factor of roughly 1,1-1,2.

We also tried using asynchronous memory copying to the gpu but this had no noticeable impact on performance.

When this was done we tried to find a way to optimize the algorithm in regard to coalesced memory access. We could not come up with a another way of accessing that had better performance than what was already implemented and decided that the memory accesses were optimized to an acceptable degree.

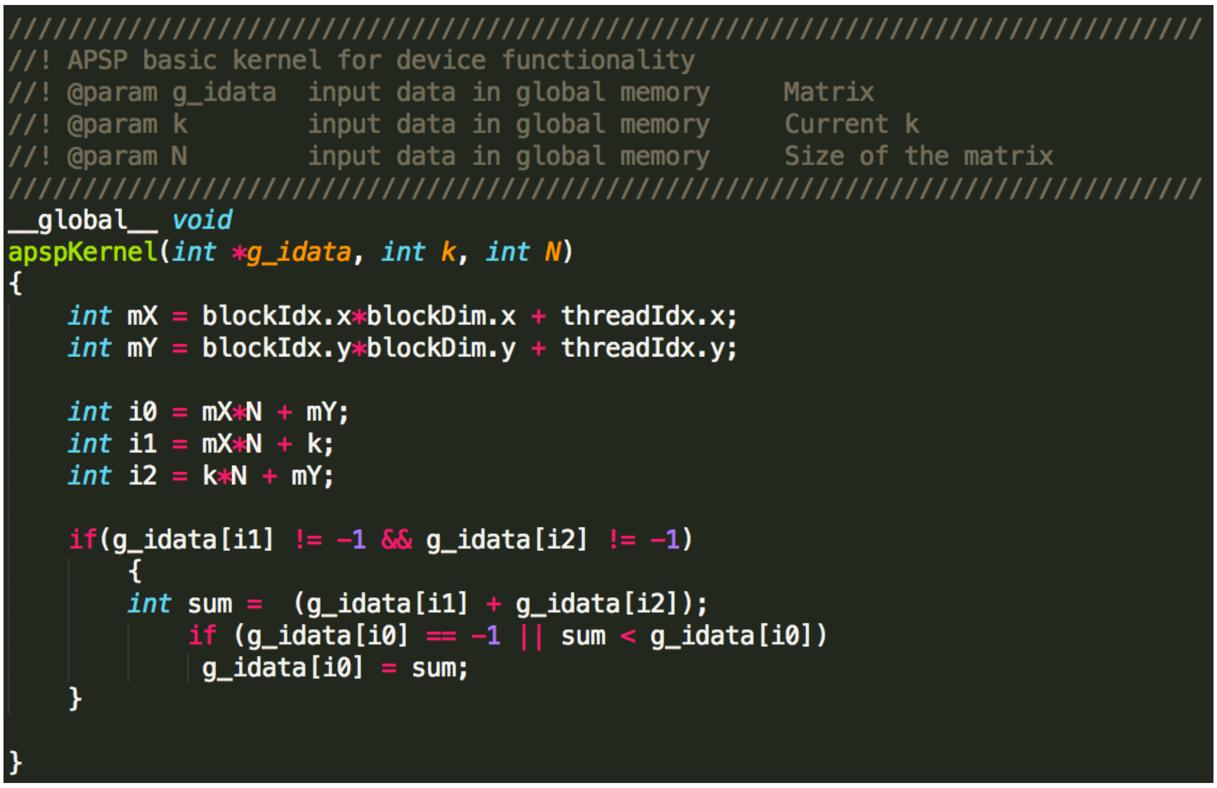
7. Appendix

7.1. Main Basic Kernel



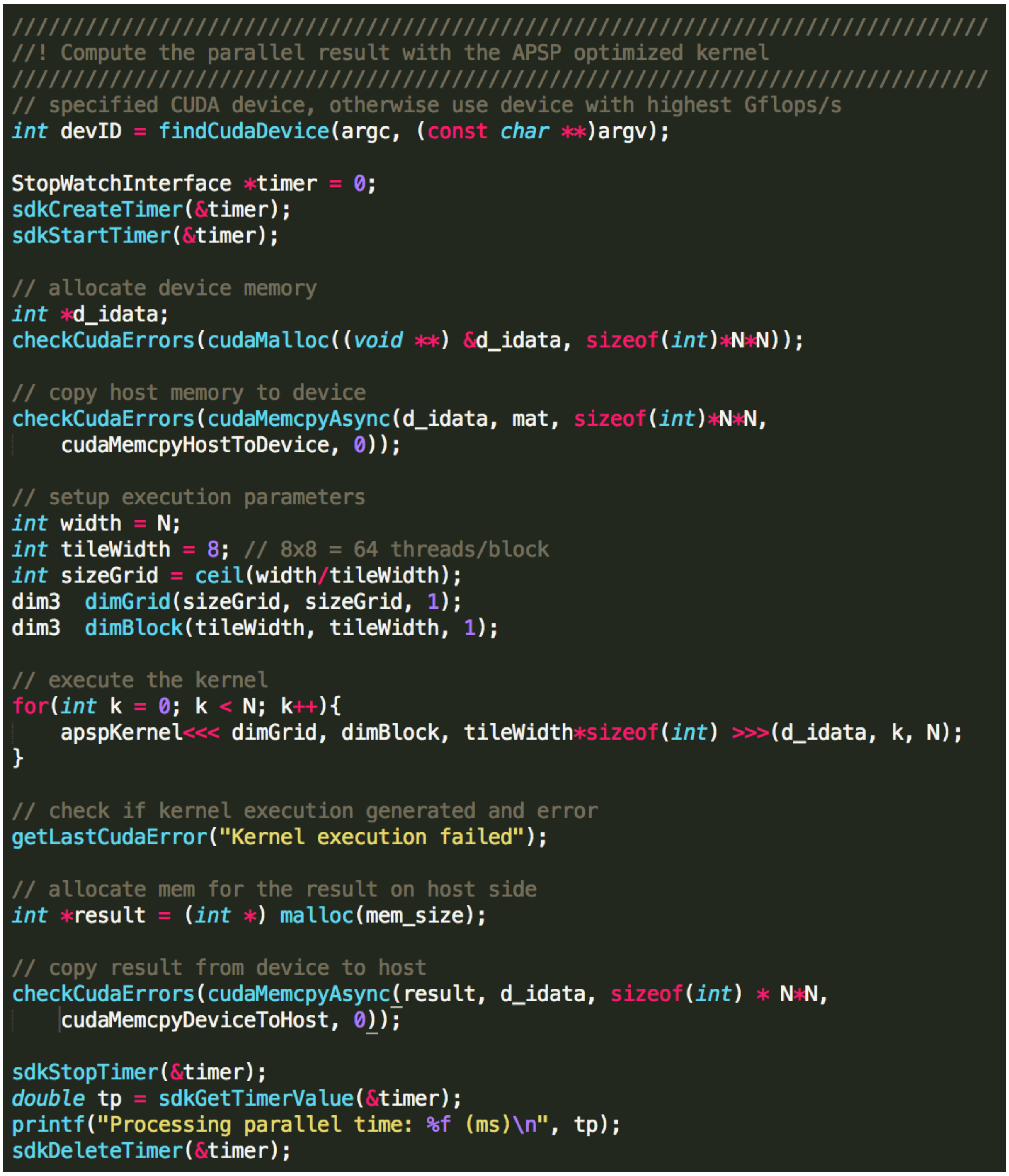
```
////////////////////////////////////////////////////////////////////////////////
//! Compute the parallel result withe the APSP basic kernel
////////////////////////////////////////////////////////////////////////////////
// use command-line specified CUDA device, otherwise use device with highest Gflops/s
int devID = findCudaDevice(argc, (const char **)argv);

StopWatchInterface *timer = 0;
sdkCreateTimer(&timer);
sdkStartTimer(&timer);

// allocate device memory
int *d_idata;
checkCudaErrors(cudaMalloc((void **) &d_idata, sizeof(int)*N*N));

// copy host memory to device
checkCudaErrors(cudaMemcpy(d_idata, mat, sizeof(int)*N*N,cudaMemcpyHostToDevice));

// setup execution parameters
int width = N;
int tileWidth = 8; // 8x8 = 64 threads/block
int sizeGrid = ceil(width/tileWidth);
dim3  dimGrid(sizeGrid, sizeGrid, 1);
dim3  dimBlock(tileWidth, tileWidth, 1);

// execute the kernel
for(int k = 0; k < N; k++){
    apspKernel<<< dimGrid, dimBlock >>>(d_idata, k, N);
}

// check if kernel execution generated and error
getLastCudaError("Kernel execution failed");

// allocate mem for the result on host side
int *result = (int *) malloc(mem_size);

// copy result from device to host
checkCudaErrors(cudaMemcpyAsync(result, d_idata, sizeof(int) * N*N, cudaMemcpyDeviceToHost, 0));

sdkStopTimer(&timer);
double tp = sdkGetTimerValue(&timer);
printf("Processing parallel time: %f (ms)\n", tp);
sdkDeleteTimer(&timer);
```

7.2. Basic Kernel



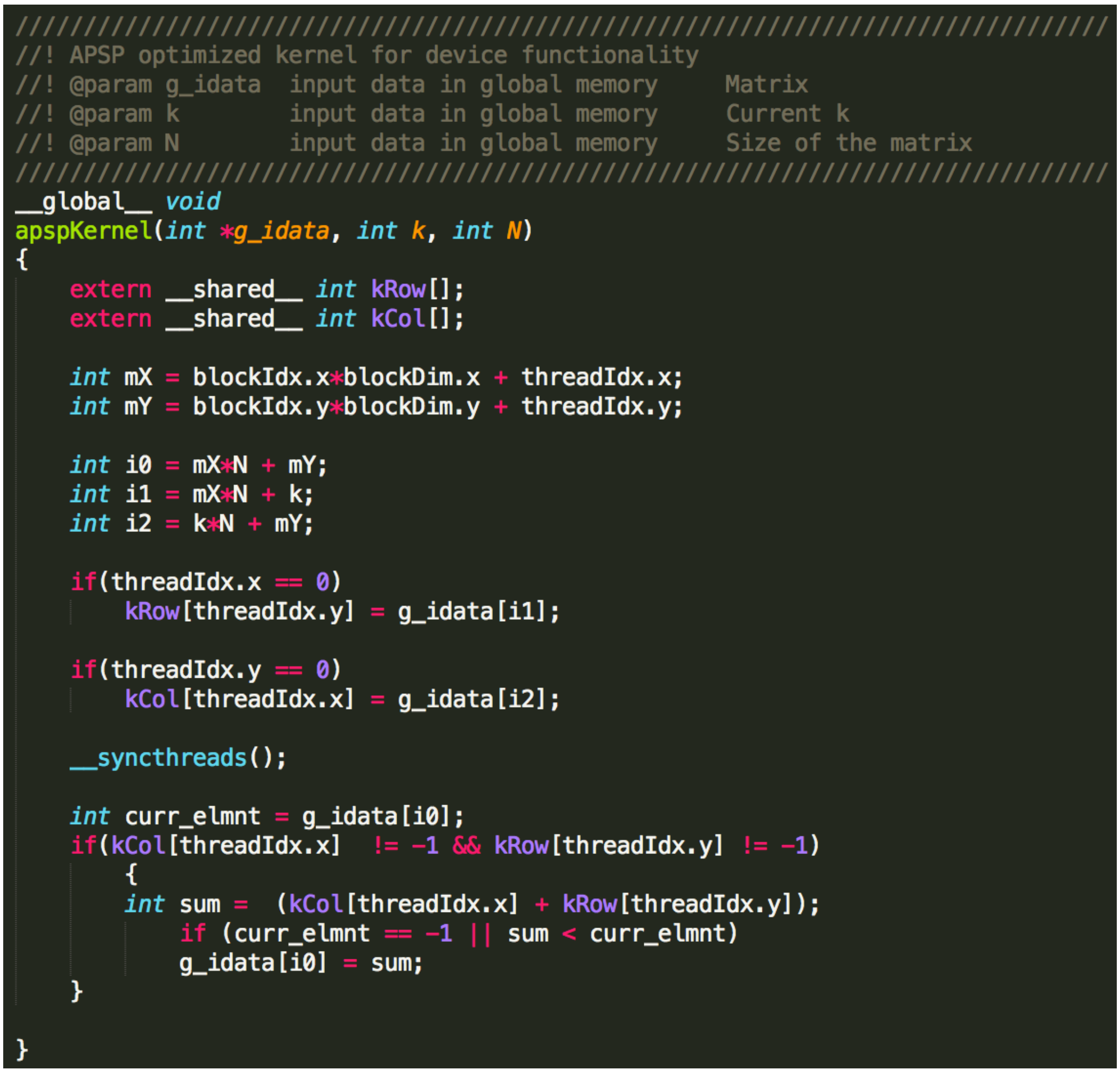
```
////////////////////////////////////////////////////////////////////////////////
//! APSP basic kernel for device functionality
//! @param g_idata  input data in global memory     Matrix
//! @param k        input data in global memory     Current k
//! @param N        input data in global memory     Size of the matrix
////////////////////////////////////////////////////////////////////////////////
__global__ void
apspKernel(int *g_idata, int k, int N)
{
    int mX = blockIdx.x*blockDim.x + threadIdx.x;
    int mY = blockIdx.y*blockDim.y + threadIdx.y;

    int i0 = mX*N + mY;
    int i1 = mX*N + k;
    int i2 = k*N + mY;

    if(g_idata[i1] != -1 && g_idata[i2] != -1)
        {
        int sum =  (g_idata[i1] + g_idata[i2]);
            if (g_idata[i0] == -1 || sum < g_idata[i0])
             g_idata[i0] = sum;
    }

}
```

7.3. Main Optimized Kernel



```
////////////////////////////////////////////////////////////////////////////////
//! Compute the parallel result with the APSP optimized kernel
////////////////////////////////////////////////////////////////////////////////
// specified CUDA device, otherwise use device with highest Gflops/s
int devID = findCudaDevice(argc, (const char **)argv);

StopWatchInterface *timer = 0;
sdkCreateTimer(&timer);
sdkStartTimer(&timer);

// allocate device memory
int *d_idata;
checkCudaErrors(cudaMalloc((void **) &d_idata, sizeof(int)*N*N));

// copy host memory to device
checkCudaErrors(cudaMemcpyAsync(d_idata, mat, sizeof(int)*N*N,
    cudaMemcpyHostToDevice, 0));

// setup execution parameters
int width = N;
int tileWidth = 8; // 8x8 = 64 threads/block
int sizeGrid = ceil(width/tileWidth);
dim3  dimGrid(sizeGrid, sizeGrid, 1);
dim3  dimBlock(tileWidth, tileWidth, 1);

// execute the kernel
for(int k = 0; k < N; k++){
    apspKernel<<< dimGrid, dimBlock, tileWidth*sizeof(int) >>>(d_idata, k, N);
}

// check if kernel execution generated and error
getLastCudaError("Kernel execution failed");

// allocate mem for the result on host side
int *result = (int *) malloc(mem_size);

// copy result from device to host
checkCudaErrors(cudaMemcpyAsync(result, d_idata, sizeof(int) * N*N,
    cudaMemcpyDeviceToHost, 0));

sdkStopTimer(&timer);
double tp = sdkGetTimerValue(&timer);
printf("Processing parallel time: %f (ms)\n", tp);
sdkDeleteTimer(&timer);
```

7.4. Optimized Kernel



```
////////////////////////////////////////////////////////////////////////////////
//! APSP optimized kernel for device functionality
//! @param g_idata  input data in global memory     Matrix
//! @param k        input data in global memory     Current k
//! @param N        input data in global memory     Size of the matrix
////////////////////////////////////////////////////////////////////////////////
__global__ void
apspKernel(int *g_idata, int k, int N)
{
    extern __shared__ int kRow[];
    extern __shared__ int kCol[];

    int mX = blockIdx.x*blockDim.x + threadIdx.x;
    int mY = blockIdx.y*blockDim.y + threadIdx.y;

    int i0 = mX*N + mY;
    int i1 = mX*N + k;
    int i2 = k*N + mY;

    if(threadIdx.x == 0)
        kRow[threadIdx.y] = g_idata[i1];

    if(threadIdx.y == 0)
        kCol[threadIdx.x] = g_idata[i2];

    __syncthreads();

    int curr_elmnt = g_idata[i0];
    if(kCol[threadIdx.x]  != -1 && kRow[threadIdx.y] != -1)
        {
        int sum =  (kCol[threadIdx.x] + kRow[threadIdx.y]);
            if (curr_elmnt == -1 || sum < curr_elmnt)
            g_idata[i0] = sum;
    }

}
```