ECSE 420: Parallel Computing – Fall 2019
Lab 3 Report: CUDA Musical Instrument Simulation

Group 26
Anna Bieber - 260678856
Walid Chabchoub - 260677008

Date of Submission: Sunday December 1st, 2019
Instructor: Prof. Zeljko Zilic

I. Introduction

The goal of this lab is to implement code for a musical instrument simulation and parallelize it using CUDA. To implement a musical instrument (analogue signals), these signals needed to be estimated into finite elements. The example of a drum is used for this lab, where the drum is estimated to be a square matrix and each element in the matrix is estimated using the provided formula. To be able to simulate the different repercussion we can do multiple iterations while updating the values in the matrices every time.

II. Sequential Simulation

a. Implementation

For the sequential part of the lab, we started out by analyzing the three different type of values that needs to be computed in the grid: center values, border and corner values. The center values are calculated with the formula provided in the lab handout using u1 and u2, then the border and corner values are calculated using the previously calculated center values, where each corner and border has a specific formula (as stated in the handout).

We implemented a hit function where the center, border and corner values are calculated, then we update u1 and u2 to take the value of u and u1 respectively. This method is called in the main function where it is called T number of times, where T is a command value to run the program. For our program we use flattened arrays to calculate u, u1 and u2 where the index is to the one used in previous labs, that is $x + (y \times N)$.

When implementing this program, we had numerous problems. Initially, we wanted to use 2D arrays, however, when passing 2D arrays to another function there were some issues with the array size. This led us to use pointers and 1D arrays, which allowed us to access and calculate the values as we wanted. We also ran into some memory issues, when initializing and copying the values of one array to another. These were easily fixed, and we could move onto testing our program

b. Testing

The testing of our code was simply running the code and compare the output with the expected output in the handout. As mentioned above, we had some incorrect values in the beginning due to incorrectly allocating memories for arrays and pointers. Once this was fixed, the code ran perfectly.

III. Parallel Simulation – one thread per element

a. Implementation

For the first parallel simulation, we had to use one thread per element (node) in the matrix. Thus, for this case we had 16 threads. In the sequential part of the algorithm we used two for loops to iterate over the width and height of our matrix, so for the parallel simulation we needed to figure out how to replace the for loops with threads. This, we had done in previous labs, so we figured out quickly how to do it. We used if statements to assign threads to each element. The inner elements (4 elements in total) were assigned two threads, the corner and border elements were assignment on thread each. The rest of the code remained unchanged as we were already using pointers for the sequential part.

b. Testing

When testing the code, we ran into some issues. The first issue was that the border and corner elements are dependent on the inner elements previously computed. Thus, these must be done in separate methods in order to use threads. We split up the code so that the first function would compute the inner elements of the matrix with 2 threads in total, and then another function would be called to perform the parallel computation of the border and corner elements.
Similar to the previous implementation, we are printing u[N/2][N/2] to the command prompt, however we were unable to get the correct values.

IV. Parallel Simulation – Threads & Blocks

a. Implementation

In a similar fashion to the previous labs, we are traveling through the drums nodes using both block dimensions, coordinates and threads coordinates as well. In fact, we are forming the dimensions of the kernel parameters as being :

$$dim3\ totalThreadsPerBlock(threads\_per\_block, 1, 1)$$

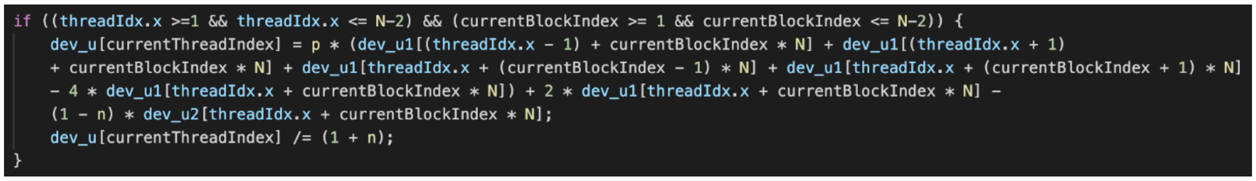
$$dim3\ \text{totalBlocks}(numXBlocks, numYBlocks, 1)$$

Where $numXBlocks = \frac{N^2}{threads\_per\_block}$ and $numYBlocks = 1$, furthermore we have set $threads\_per\_block = 1024$, given that it is the maximum number of threads per block supported.

We have decided to split the work into 2 different kernels, the first kernel computes the propagation of the wave in the interior nodes but also computes the borders of the drum while the second kernel computes only the corners and uses 4 threads only given that there are only 4 corners in the drum grid.

Inside the main kernel (the one computing the interior nodes and borders), we are defining the variable with which we will be iterating through the 1D flattened array of our grid to be currentThreadIndex = threadIdx.x + currentBlockIndex * blockWidth. Where currentBlockIndex = blockIdx.x + blockIdx.y * gridWidth. In a similar fashion to lab 2 and 3, this has enabled us to naviguate through the grid of the drum nodes.

However, this time, in order to apply the equation to compute the new u, we are checking whether the current threadIdx.x is between 1 and N-2 and similarly if the currentBlockIndex (similar to the y coordinate) is also between 1 and N-2 as shown in the screenshot below.



```
if ((threadIdx.x >=1 && threadIdx.x <= N-2) && (currentBlockIndex >= 1 && currentBlockIndex <= N-2)) {
    dev_u[currentThreadIndex] = p * (dev_u1[(threadIdx.x - 1) + currentBlockIndex * N] + dev_u1[(threadIdx.x + 1)
    + currentBlockIndex * N] + dev_u1[threadIdx.x + (currentBlockIndex - 1) * N] + dev_u1[threadIdx.x + (currentBlockIndex + 1) * N]
    - 4 * dev_u1[threadIdx.x + currentBlockIndex * N]) + 2 * dev_u1[threadIdx.x + currentBlockIndex * N] -
    (1 - n) * dev_u2[threadIdx.x + currentBlockIndex * N];
    dev_u[currentThreadIndex] /= (1 + n);
}
```

b. Testing

Similarly, to the previous section, when testing the code, we encountered issues where the border and corner elements are dependent on the inner elements previously computed. Thus, we decided to separate them in different kernels. Next, another issue that arose was that we were not getting the correct values for u[N/2][N/2], and we believe the issue that, had we had enough time to work on it, is the indexing our 1D array, that is, the way we are accessing the nodes is incorrect.

V. Conclusion

The goal of this experiment was to implement code for musical instrument simulation by first performing sequential processing of this and we showed it was a simple implementation. Next we had to parallelize this process using CUDA with the usage of threads only, then using blocks and threads combined.

This lab has allowed for a broader understanding of the CUDA infrastructure as well as different use cases.

APPENDIX

Appendix A1 – Sequential code

```
#include <stdio.h>

#define N 4
#define n 0.0002
#define p 0.5
#define G 0.75

void hit(float u[N*N], float u1[N * N], float u2[N * N]) {

    for (int i = 1; i <= N / 2; i++) {
        for (int j = 1; j <= N / 2; j++) {
            u[i + j * N] = p * (u1[(i - 1) + j * N] + u1[(i + 1) + j * N] + u1[i + (j - 1) * N] + u1[i + (j + 1) * N] - 4 * u1[i + j * N]) + 2
* u1[i + j * N] - (1 - n) * u2[i + j * N];
            u[i + j * N] /= (1 + n);
        }
    }

    //set borders
    for (int x = 1; x <= N / 2; x++) {
        u[x * N] = G * u[1 + x * N];
        u[N - 1 + x * N] = G * u[N - 2 + x * N];
        u[x + 0 * N] = G * u[x + 1 * N];
        u[x + (N - 1) * N] = G * u[x + (N - 2) * N];
    }

    //set corners
    u[0] = G * u[1];
    u[N - 1] = G * u[N - 2];
    u[N * (N - 1)] = G * u[N * (N - 2)];
    u[(N - 1) + (N - 1) * N] = G * u[(N - 1) + (N - 2) * N];

    float* temp[N * N];

    for (size_t i = 0; i < N * N; i++) {
        temp[i] = 0;
    }

    printf("u[N/2][N/2]: %f \n", u[(N / 2) + (N / 2) * N]);

    memcpy(temp, u2, N * N * sizeof(float));
    memcpy(u2, u1, N * N * sizeof(float));
    memcpy(u1, u, N * N * sizeof(float));
    memcpy(u, temp, N * N * sizeof(float));

    /*
        u2 = u1;
        u1 = u;
        u = temp;
    */

}


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

    if (argc < 1) {
        printf("Missing argument. \n");
        return -1;
    }

    //int T = atoi(argv[1]);

    float u[N * N];
    float u1[N * N];
    float u2[N * N];

    for (size_t i = 0; i < N * N; i++) {
        u[i] = 0;
    }

    for (size_t i = 0; i < N * N; i++) {
        u1[i] = 0;
    }

    for (size_t i = 0; i < N * N; i++) {
        u2[i] = 0;
    }
    u1[(N / 2) + (N / 2) * N] = 1;

    for (int T = 0; T < 10; T++) {
        hit(u, u1, u2);
    }

    return 0;
}
```

Appendix A2 – Parallel code (1 thread per node)

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <cuda_runtime_api.h>
#include <cuda.h>
#include <device_functions.h>

#include<string.h>
#include <stdio.h>

#define N 4
#define n 0.0002
#define p 0.5
#define G 0.75

void hitWithCuda(float u[N * N], float u1[N * N], float u2[N * N], int T);

__global__ void hit(float dev_u[N * N], float dev_u1[N * N], float dev_u2[N * N])
{

    int currentIndex = threadIdx.x + N * threadIdx.y;

    if ((threadIdx.x >=1 && threadIdx.x <= 2) && (threadIdx.y >= 1 && threadIdx.y <= 2)) {
        dev_u[currentIndex] = p * (dev_u1[(threadIdx.x - 1) + threadIdx.y * N] + dev_u1[(threadIdx.x + 1) + threadIdx.y * N] + dev_u1[threadIdx.x + (threadIdx.y - 1) * N] + dev_u1[threadIdx.x + (threadIdx.y + 1) * N] - 4 * dev_u1[threadIdx.x + threadIdx.y * N]) + 2 * dev_u1[threadIdx.x + threadIdx.y * N] - (1 - n) * dev_u2[threadIdx.x + threadIdx.y * N];
        dev_u[currentIndex] /= (1 + n);
    }

    //set borders
    if (threadIdx.x == 3 || threadIdx.x == 4) {
        dev_u[threadIdx.x * N] = G * dev_u[1 + threadIdx.x * N];
    }
    if (threadIdx.x == 5 || threadIdx.x == 6) {
        dev_u[N - 1 + threadIdx.x * N] = G * dev_u[N - 2 + threadIdx.x * N];
    }
    if (threadIdx.x == 7 || threadIdx.x == 8) {
        dev_u[threadIdx.x + 0 * N] = G * dev_u[threadIdx.x + 1 * N];
    }
    if (threadIdx.x == 9 || threadIdx.x == 10) {
        dev_u[threadIdx.x + (N - 1) * N] = G * dev_u[threadIdx.x + (N - 2) * N];
    }

    //set corners
    if (threadIdx.x == 11) {
        dev_u[0] = G * dev_u[1];
    }
    if (threadIdx.x == 12) {
        dev_u[N - 1] = G * dev_u[N - 2];
    }
    if (threadIdx.x == 13) {
        dev_u[N * (N - 1)] = G * dev_u[N * (N - 2)];
    }
    if (threadIdx.x == 14) {
        dev_u[(N - 1) + (N - 1) * N] = G * dev_u[(N - 1) + (N - 2) * N];
    }

    __syncthreads();

}


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

    /*if (argc < 1) {
        printf("Missing argument. \n");
        return -1;
    }*/

    //int T = atoi(argv[1]);

    float u[N * N];
    float u1[N * N];
    float u2[N * N];

    for (size_t i = 0; i < N * N; i++) {
        u[i] = 0;
    }
    for (size_t i = 0; i < N * N; i++) {
        u1[i] = 0;
    }

    for (size_t i = 0; i < N * N; i++) {
        u2[i] = 0;
    }

    u1[(N / 2) + (N / 2) * N] = 1;
    int T = 10;
    hitWithCuda(u, u1, u2, T);

    return 0;
}

void hitWithCuda(float u[N * N], float u1[N * N], float u2[N * N], int T)
{

    float dev_u[N * N];
    float dev_u1[N * N];
    float dev_u2[N * N];

    cudaMalloc((void**)(&dev_u), N * N * sizeof(float));
    cudaMalloc((void**)(&dev_u1), N * N * sizeof(float));
    cudaMalloc((void**)(&dev_u2), N * N * sizeof(float));

    for (int i = 0; i < T; i++) {

        cudaMemcpy(dev_u, u, N * N * sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(dev_u1, u1, N * N * sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(dev_u2, u2, N * N * sizeof(float), cudaMemcpyHostToDevice);

        hit << < 1, 16 >> > (dev_u, dev_u1, dev_u2);
        cudaDeviceSynchronize();
        cudaMemcpy(u, dev_u, N * N * sizeof(float), cudaMemcpyDeviceToHost);
        cudaMemcpy(u1, dev_u1, N * N * sizeof(float), cudaMemcpyDeviceToHost);
        cudaMemcpy(u2, dev_u2, N * N * sizeof(float), cudaMemcpyDeviceToHost);

        float* temp[N * N];

        for (size_t i = 0; i < N * N; i++) {
            temp[i] = 0;
        }

        printf("u[N/2][N/2]: %f \n", u[(N / 2) + (N / 2) * N]);

        memcpy(temp, u2, N * N * sizeof(float));
        memcpy(u2, u1, N * N * sizeof(float));
        memcpy(u1, u, N * N * sizeof(float));
        memcpy(u, temp, N * N * sizeof(float));

    }



    cudaFree(dev_u);
    cudaFree(dev_u1);
    cudaFree(dev_u2);

}
```

Appendix A3 – Parallel code (blocks & threads)

```

#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <cuda_runtime_api.h>
#include <cuda.h>
#include <device_functions.h>

#include<string.h>
#include <stdio.h>

#define N 32
#define n 0.0002
#define p 0.5
#define G 0.75

void hitWithCuda(float u[N * N], float u1[N * N], float u2[N * N], int T);

__global__ void computeCorners(float dev_u[N * N])
{
    //set corners
    if (threadIdx.x == 1) {
        dev_u[0] = G * dev_u[1];
    }
    if (threadIdx.x == 2) {
        dev_u[N - 1] = G * dev_u[N - 2];
    }
    if (threadIdx.x == 3) {
        dev_u[N * (N - 1)] = G * dev_u[N * (N - 2)];
    }
    if (threadIdx.x == 4) {
        dev_u[(N - 1) + (N - 1) * N] = G * dev_u[(N - 1) + (N - 2) * N];
    }
    __syncthreads();

}
```

```
__global__ void hit(float dev_u[N * N], float dev_u1[N * N], float dev_u2[N * N])
{

    // Numbers of blocks in a grid
    unsigned gridWidth = gridDim.x;
    // Fetching index of current block in 1-D (in comparison to the entire grid - equivalent of x + y * width)
    int currentBlockIndex = blockIdx.x + blockIdx.y * gridWidth;

    // Numbers of threads per block
    unsigned blockWidth = blockDim.x;
    // Fetching index of current thread in 1-D inside the current block (in comparison to the entire grid)
    int currentThreadIndex = threadIdx.x + currentBlockIndex * blockWidth;

    if ((threadIdx.x >=1 && threadIdx.x <= N-2) && (currentBlockIndex >= 1 && currentBlockIndex <= N-2)) {
        dev_u[currentThreadIndex] = p * (dev_u1[(threadIdx.x - 1) + currentBlockIndex * N] + dev_u1[(threadIdx.x + 1) + currentBlockIndex * N] + dev_u1[threadIdx.x + (currentBlockIndex - 1) * N] + dev_u1[threadIdx.x + (currentBlockIndex + 1) * N] - 4 * dev_u1[threadIdx.x + currentBlockIndex * N]) + 2 * dev_u1[threadIdx.x + currentBlockIndex * N] - (1 - n) * dev_u2[threadIdx.x + currentBlockIndex * N];
        dev_u[currentThreadIndex] /= (1 + n);
    }

    //set borders
    if (threadIdx.x > 1 && currentBlockIndex  == 0) {
        // have to fix coordinates below
        dev_u[threadIdx.x * N] = G * dev_u[1 + threadIdx.x * N];
    }
    if (threadIdx.x > 1 && currentBlockIndex == N-1) {
        dev_u[N - 1 + threadIdx.x * N] = G * dev_u[N - 2 + threadIdx.x * N];
    }
    if (threadIdx.x == 0 && currentBlockIndex > 1) {
        dev_u[threadIdx.x + 0 * N] = G * dev_u[threadIdx.x + 1 * N];
    }
    if (threadIdx.x == N-1 && currentBlockIndex > 1) {
        dev_u[threadIdx.x + (N - 1) * N] = G * dev_u[threadIdx.x + (N - 2) * N];
    }

    __syncthreads();

}


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

    /*if (argc < 1) {
        printf("Missing argument. \n");
        return -1;
    }*/
    //int T = atoi(argv[1]);

    float u[N * N];
    float u1[N * N];
    float u2[N * N];

    for (size_t i = 0; i < N * N; i++) {
        u[i] = 0;
    }

    for (size_t i = 0; i < N * N; i++) {
        u1[i] = 0;
    }

    for (size_t i = 0; i < N * N; i++) {
        u2[i] = 0;
    }

    u1[(N / 2) + (N / 2) * N] = 1;
    int T = 10;
    hitWithCuda(u, u1, u2, T);

    return 0;
}

void hitWithCuda(float u[N * N], float u1[N * N], float u2[N * N], int T)
{

    float dev_u[N * N];
    float dev_u1[N * N];
    float dev_u2[N * N];

    int numXBlocks, numYBlocks;
    int threads_per_block = 1024;
    // Computing the numbers of blocks (x-axis & y-axis)
    numXBlocks = N*N / threads_per_block;
    numYBlocks = 1;

    // Creating dim3 structs to contain the dimensionality of our GPU that we will be requiring
    // (Z axis set to 1 for threads and blocks and Y axis set to 1 as well for threads because of hardware possible limitationss)
    dim3 totalThreadsPerBlock(threads_per_block, 1, 1);
    dim3 totalBlocks(numXBlocks, numYBlocks, 1);

    printf("Using %d blocks (%d in x, %d in y), each with %d threads .\n", numXBlocks * numYBlocks, numXBlocks, numYBloc
ks, threads_per_block);

    cudaMalloc((void**)(&dev_u), N * N * sizeof(float));
    cudaMalloc((void**)(&dev_u1), N * N * sizeof(float));
    cudaMalloc((void**)(&dev_u2), N * N * sizeof(float));

    for (int i = 0; i < T; i++) {

        cudaMemcpy(dev_u, u, N * N * sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(dev_u1, u1, N * N * sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(dev_u2, u2, N * N * sizeof(float), cudaMemcpyHostToDevice);

        hit << < totalBlocks, totalThreadsPerBlock >> > (dev_u, dev_u1, dev_u2);
        computeCorners <<< 1, 4 >>> (dev_u);
        cudaDeviceSynchronize();
        cudaMemcpy(u, dev_u, N * N * sizeof(float), cudaMemcpyDeviceToHost);
        cudaMemcpy(u1, dev_u1, N * N * sizeof(float), cudaMemcpyDeviceToHost);
        cudaMemcpy(u2, dev_u2, N * N * sizeof(float), cudaMemcpyDeviceToHost);

        float* temp[N * N];

        for (size_t i = 0; i < N * N; i++) {
            temp[i] = 0;
        }

        printf("u[N/2][N/2]: %f \n", u[(N / 2) + (N / 2) * N]);

        memcpy(temp, u2, N * N * sizeof(float));
        memcpy(u2, u1, N * N * sizeof(float));
        memcpy(u1, u, N * N * sizeof(float));
        memcpy(u, temp, N * N * sizeof(float));

    }


    cudaFree(dev_u);
    cudaFree(dev_u1);
    cudaFree(dev_u2);

}
```