ECSE420 Lab2 Report

Liang Zhao 260781081
Hongshuo Zhou 260792817

November 1 2020

Group 6

1 Introduction

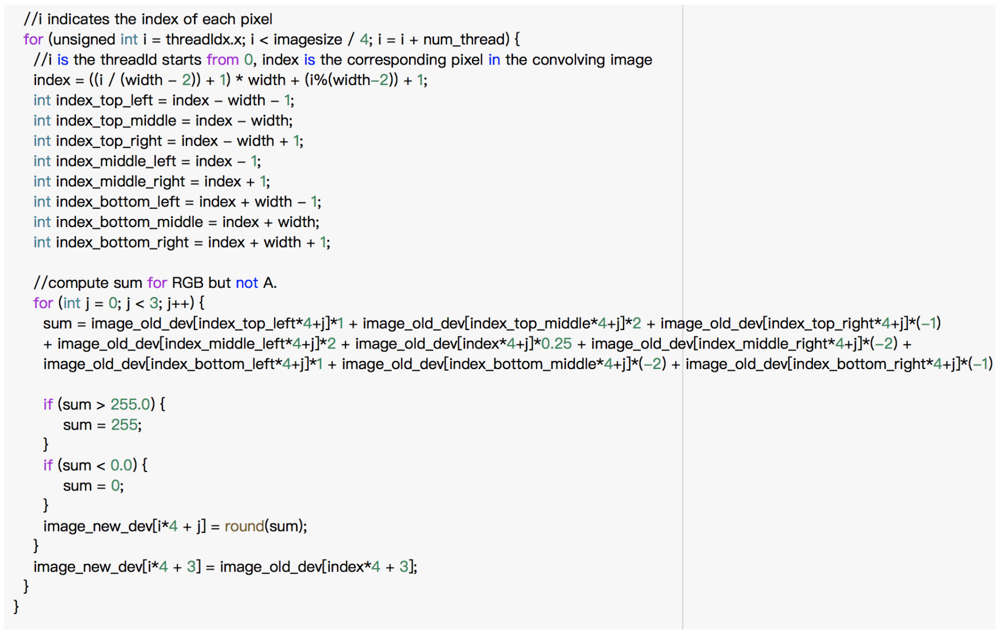
Generally in this lab, we are required to implement a convolution signal processing simulation cuda program and a musical instrument simulation cuda program. We need to write code that simulates image convolution and synthesizing drum sounds using a two-dimensional grid of finite elements.

2 Convolution

2.1 Parallelization Scheme

For this part, we put the code into block: *Convolution.cu.*

Here is our code snippet of our convolution function implementation: The function computed corresponding pixels in matrix, then evaluate the

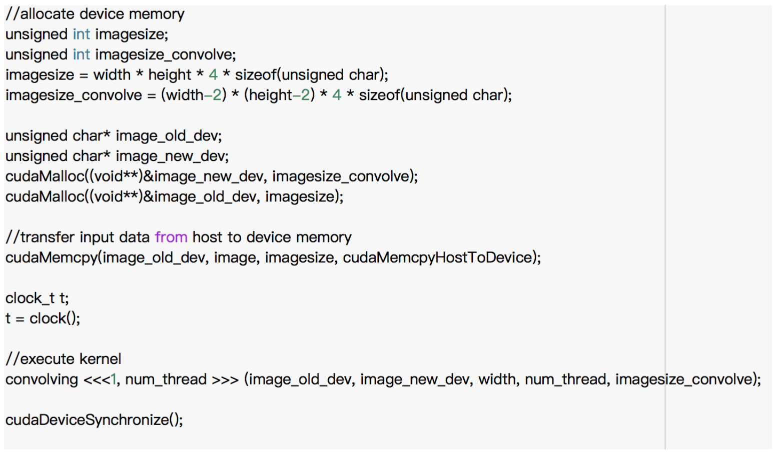


```
  //i indicates the index of each pixel
  for (unsigned int i = threadIdx.x; i < imagesize / 4; i = i + num_thread) {
    //i is the threadId starts from 0, index is the corresponding pixel in the convolving image
    index = ((i / (width − 2)) + 1) * width + (i%(width−2)) + 1;
    int index_top_left = index − width − 1;
    int index_top_middle = index − width;
    int index_top_right = index − width + 1;
    int index_middle_left = index − 1;
    int index_middle_right = index + 1;
    int index_bottom_left = index + width − 1;
    int index_bottom_middle = index + width;
    int index_bottom_right = index + width + 1;

    //compute sum for RGB but not A.
    for (int j = 0; j < 3; j++) {
      sum = image_old_dev[index_top_left*4+j]*1 + image_old_dev[index_top_middle*4+j]*2 + image_old_dev[index_top_right*4+j]*(−1)
      + image_old_dev[index_middle_left*4+j]*2 + image_old_dev[index*4+j]*0.25 + image_old_dev[index_middle_right*4+j]*(−2) +
      image_old_dev[index_bottom_left*4+j]*1 + image_old_dev[index_bottom_middle*4+j]*(−2) + image_old_dev[index_bottom_right*4+j]*(−1)

      if (sum > 255.0) {
          sum = 255;
      }
      if (sum < 0.0) {
          sum = 0;
      }
      image_new_dev[i*4 + j] = round(sum);
    }
    image_new_dev[i*4 + 3] = image_old_dev[index*4 + 3];
  }
}
```

Figure 1: Convolution

RGB result by performing matrix multiplication.

Here is our code snippet of our parallel implementation:
In this part, the code allocate device memory for images and transfer input



```
//allocate device memory
unsigned int imagesize;
unsigned int imagesize_convolve;
imagesize = width * height * 4 * sizeof(unsigned char);
imagesize_convolve = (width-2) * (height-2) * 4 * sizeof(unsigned char);

unsigned char* image_old_dev;
unsigned char* image_new_dev;
cudaMalloc((void**)&image_new_dev, imagesize_convolve);
cudaMalloc((void**)&image_old_dev, imagesize);

//transfer input data from host to device memory
cudaMemcpy(image_old_dev, image, imagesize, cudaMemcpyHostToDevice);

clock_t t;
t = clock();

//execute kernel
convolving <<<1, num_thread >>> (image_old_dev, image_new_dev, width, num_thread, imagesize_convolve);

cudaDeviceSynchronize();
```

Figure 2: Parallel

data from host to device memory. Then execute convolution function to finish the image convolution.

2.2 Test Results and Analysis

$$Speedup = \frac{Told}{Tnew}$$

Told stands for the runtime with only one thread, *Tnew* stands for the runtime under different number of threads. In order to compare runtime performances under different number of threads, we used Speedup to measure it. We ran the algorithm with a fixed number of threads 5 times and take the average using n threads, where n= 1, 4, 8, 16, 64, 128, 256, 512,1024. Below is the speedup Calculation for three test images include the runtime of different threads.

Image 1 Speedup Calculation:
1 thread execution time: 7.394 sec
4 threads = 7.394 / 2.672 = 2.767
8 threads = 7.394/ 1.557 = 4.749
16 threads = 7.394 / 0.875 = 8.450
64 threads = 7.394 /0.350= 21.126
128 threads = 7.394 / 0.275 = 26.815
256 threads = 7.394 / 0.253 = 29.225
512 threads = 7.394 / 0.251 = 29.458
1024 threads = 7.394 / 0.240= 30.808

Image 2 Speedup Calculation:
1 thread execution time: 2.293 sec
4 threads = 2.293 / 0.721 = 3.180
8 threads = 2.293/ 0.494 = 4.641
16 threads = 2.293 / 0.330 = 6.948
64 threads = 2.293/ 0.118= 19.432
128 threads = 2.293 / 0.0744 = 30.820
256 threads = 2.293 / 0.0725 = 31.628
512 threads = 2.293 / 0.0721 = 31.803
1024 threads = 2.293 / 0.0711= 32.250

Image 3 Speedup Calculation:
1 thread execution time: 2.496 sec
4 threads = 2.496 / 0.774 = 3.225
8 threads = 2.496/ 0.487 = 5.125
16 threads = 2.496 / 0.351 = 7.111
64 threads = 2.496/ 0.131= 19.053
128 threads = 2.496 / 0.0824 = 30.291
256 threads = 2.496 / 0.0802 = 31.122
512 threads = 2.496 / 0.0801 = 31.161
1024 threads = 2.496 / 0.0800= 31.200

From those three speedup diagrams, we can observe that the execution time gradually increased as the threads number increased. It is also worth mentioning that, all the three plots shows that the speedup reached its limit at 128 threads, as the speed up stayed steady for 256,512 and 1024 threads. It is also the evidence that our code performed the parallel convolution correctly, as three images have same trend on speedup.

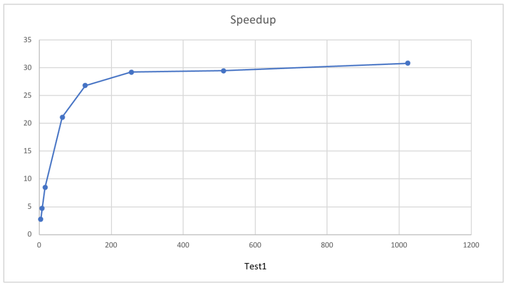


Figure 3: Speedup Test1

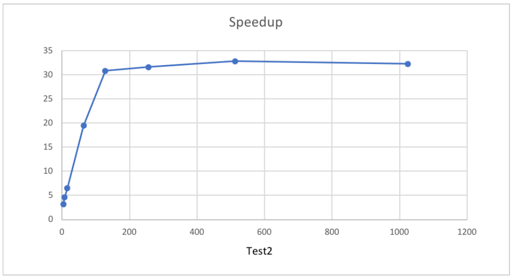


Figure 4: Speedup Test2

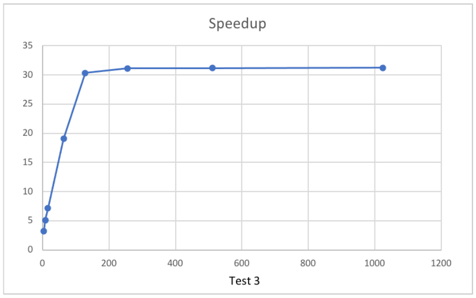
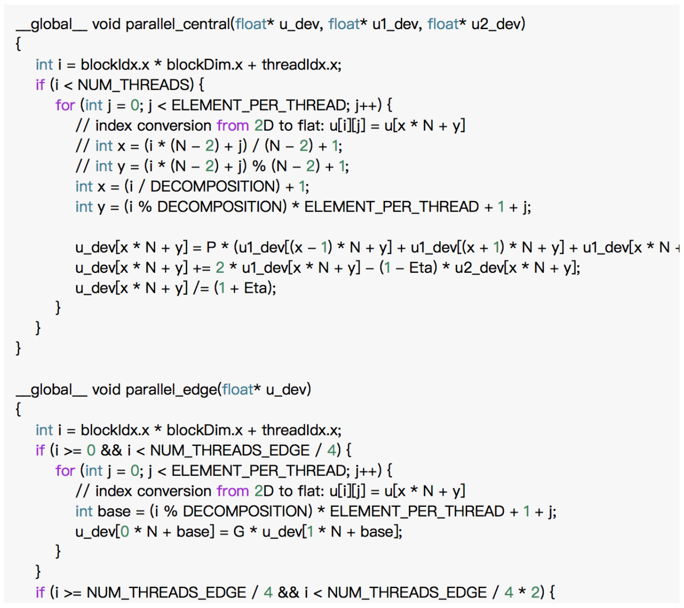


Figure 5: Speedup Test3

3 Finite Element Music Synthesis

3.1 Parallelization Scheme

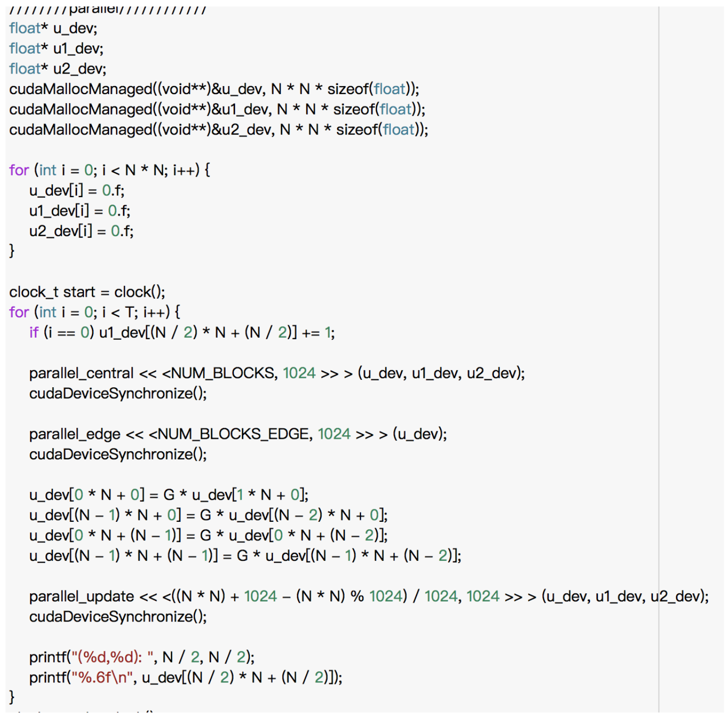
For this part, we wrote the required code into two seperate blocks: *synthesis/sequentially.cu*,*synthesis/parallelly.cu*. The first block performed sequential synthesis for 4*4 grid and the second block performed parallel synthesis for both 4*4 and 512*512 grid.
The sequential part is quite simple, our code updated all of the nodes one by one from given formula. Then output the n[2,2] node. For parallel part, below is a code snippet of our parallel code updating function implementation, due to size limit, only part of the functions is included:
The functions computed central nodes and corner nodes separately , then



```
__global__ void parallel_central(float* u_dev, float* u1_dev, float* u2_dev)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < NUM_THREADS) {
        for (int j = 0; j < ELEMENT_PER_THREAD; j++) {
            // index conversion from 2D to flat: u[i][j] = u[x * N + y]
            // int x = (i * (N - 2) + j) / (N - 2) + 1;
            // int y = (i * (N - 2) + j) % (N - 2) + 1;
            int x = (i / DECOMPOSITION) + 1;
            int y = (i % DECOMPOSITION) * ELEMENT_PER_THREAD + 1 + j;

            u_dev[x * N + y] = P * (u1_dev[(x - 1) * N + y] + u1_dev[(x + 1) * N + y] + u1_dev[x * N +
            u_dev[x * N + y] += 2 * u1_dev[x * N + y] - (1 - Eta) * u2_dev[x * N + y];
            u_dev[x * N + y] /= (1 + Eta);
        }
    }
}

__global__ void parallel_edge(float* u_dev)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= 0 && i < NUM_THREADS_EDGE / 4) {
        for (int j = 0; j < ELEMENT_PER_THREAD; j++) {
            // index conversion from 2D to flat: u[i][j] = u[x * N + y]
            int base = (i % DECOMPOSITION) * ELEMENT_PER_THREAD + 1 + j;
            u_dev[0 * N + base] = G * u_dev[1 * N + base];
        }
    }
    if (i >= NUM_THREADS_EDGE / 4 && i < NUM_THREADS_EDGE / 4 * 2) {
```

Figure 6: Node update

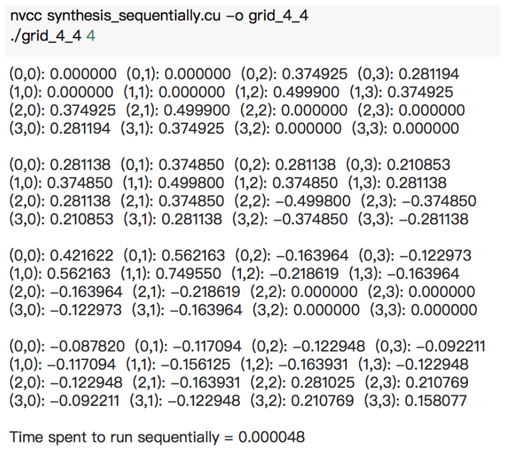
the update function would updated nodes.
Below is a code snippet of our main function implementation, due to size limit, only part of the functions is included:
The functions first allocated the memory for nodes, then paralleled calling two nodes function. Finally called the updating function and output N[2,2].
.



```
////////parallel/////////////
float* u_dev;
float* u1_dev;
float* u2_dev;
cudaMallocManaged((void**)&u_dev, N * N * sizeof(float));
cudaMallocManaged((void**)&u1_dev, N * N * sizeof(float));
cudaMallocManaged((void**)&u2_dev, N * N * sizeof(float));

for (int i = 0; i < N * N; i++) {
    u_dev[i] = 0.f;
    u1_dev[i] = 0.f;
    u2_dev[i] = 0.f;
}

clock_t start = clock();
for (int i = 0; i < T; i++) {
    if (i == 0) u1_dev[(N / 2) * N + (N / 2)] += 1;

    parallel_central << <NUM_BLOCKS, 1024 >> > (u_dev, u1_dev, u2_dev);
    cudaDeviceSynchronize();

    parallel_edge << <NUM_BLOCKS_EDGE, 1024 >> > (u_dev);
    cudaDeviceSynchronize();

    u_dev[0 * N + 0] = G * u_dev[1 * N + 0];
    u_dev[(N – 1) * N + 0] = G * u_dev[(N – 2) * N + 0];
    u_dev[0 * N + (N – 1)] = G * u_dev[0 * N + (N – 2)];
    u_dev[(N – 1) * N + (N – 1)] = G * u_dev[(N – 1) * N + (N – 2)];

    parallel_update << <((N * N) + 1024 – (N * N) % 1024) / 1024, 1024 >> > (u_dev, u1_dev, u2_dev);
    cudaDeviceSynchronize();

    printf("(%d,%d): ", N / 2, N / 2);
    printf("%.6f\n", u_dev[(N / 2) * N + (N / 2)]);
}
```

Figure 7: Main

3.2 Test Results and Analysis

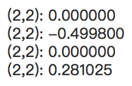


```
nvcc synthesis_sequentially.cu –o grid_4_4
./grid_4_4 4
```

```
(0,0): 0.000000  (0,1): 0.000000  (0,2): 0.374925  (0,3): 0.281194
(1,0): 0.000000  (1,1): 0.000000  (1,2): 0.499900  (1,3): 0.374925
(2,0): 0.374925  (2,1): 0.499900  (2,2): 0.000000  (2,3): 0.000000
(3,0): 0.281194  (3,1): 0.374925  (3,2): 0.000000  (3,3): 0.000000

(0,0): 0.281138  (0,1): 0.374850  (0,2): 0.281138  (0,3): 0.210853
(1,0): 0.374850  (1,1): 0.499800  (1,2): 0.374850  (1,3): 0.281138
(2,0): 0.281138  (2,1): 0.374850  (2,2): –0.499800  (2,3): –0.374850
(3,0): 0.210853  (3,1): 0.281138  (3,2): –0.374850  (3,3): –0.281138

(0,0): 0.421622  (0,1): 0.562163  (0,2): –0.163964  (0,3): –0.122973
(1,0): 0.562163  (1,1): 0.749550  (1,2): –0.218619  (1,3): –0.163964
(2,0): –0.163964  (2,1): –0.218619  (2,2): 0.000000  (2,3): 0.000000
(3,0): –0.122973  (3,1): –0.163964  (3,2): 0.000000  (3,3): 0.000000

(0,0): –0.087820  (0,1): –0.117094  (0,2): –0.122948  (0,3): –0.092211
(1,0): –0.117094  (1,1): –0.156125  (1,2): –0.163931  (1,3): –0.122948
(2,0): –0.122948  (2,1): –0.163931  (2,2): 0.281025  (2,3): 0.210769
(3,0): –0.092211  (3,1): –0.122948  (3,2): 0.210769  (3,3): 0.158077

Time spent to run sequentially = 0.000048
```

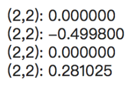
Figure 8: Grid 4*4

Figure 8 shows the nodes for 4*4 grid, iteration time is 4. It can be observed that the result perfectly matched the provided result in lab manual pdf.



```
(2,2): 0.000000
(2,2): -0.499800
(2,2): 0.000000
(2,2): 0.281025
```

Figure 9: Grid 4*4 N[2,2] sequential

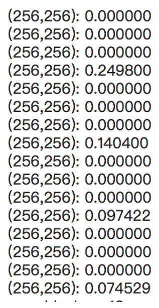


```
(2,2): 0.000000
(2,2): -0.499800
(2,2): 0.000000
(2,2): 0.281025
```

Figure 10: Grid 4*4 N[2,2] parallel

Figure 9 shows the N[2,2] output for sequential 4*4 grid simulation,iteration time is 4.

Figure 10 shows the N[2,2] output for parallel 4*4 grid simulation,iteration time is 4.



```
(256,256): 0.000000
(256,256): 0.000000
(256,256): 0.000000
(256,256): 0.249800
(256,256): 0.000000
(256,256): 0.000000
(256,256): 0.000000
(256,256): 0.140400
(256,256): 0.000000
(256,256): 0.000000
(256,256): 0.000000
(256,256): 0.097422
(256,256): 0.000000
(256,256): 0.000000
(256,256): 0.000000
(256,256): 0.074529
```

Figure 11: Grid 512*512 N[2,2] parallel

Figure 11 shows the N[2,2] output for parallel 512*512 grid simulation,iteration time is 16. It can be observed that the result perfectly matched the provided result in lab manual pdf.

Threads per block	Blocks	Elements per thread	Execution times(s)
1024	16	16	0.010615
512	256	2	0.007125
8	256	128	0.045718

Table 1: Execution Time

Table 1 compared the execution times for three combination of threads, blocks and finite elements per thread. It can be observed that the execution time increased if elements per thread increased. We can conclude that more elements per thread, slower the execution, fewer elements per thread, faster the execution.