ECSE420 Parallel Computing

Lab 2 CUDA Convolution and Musical Instrument Simulation

Group 43

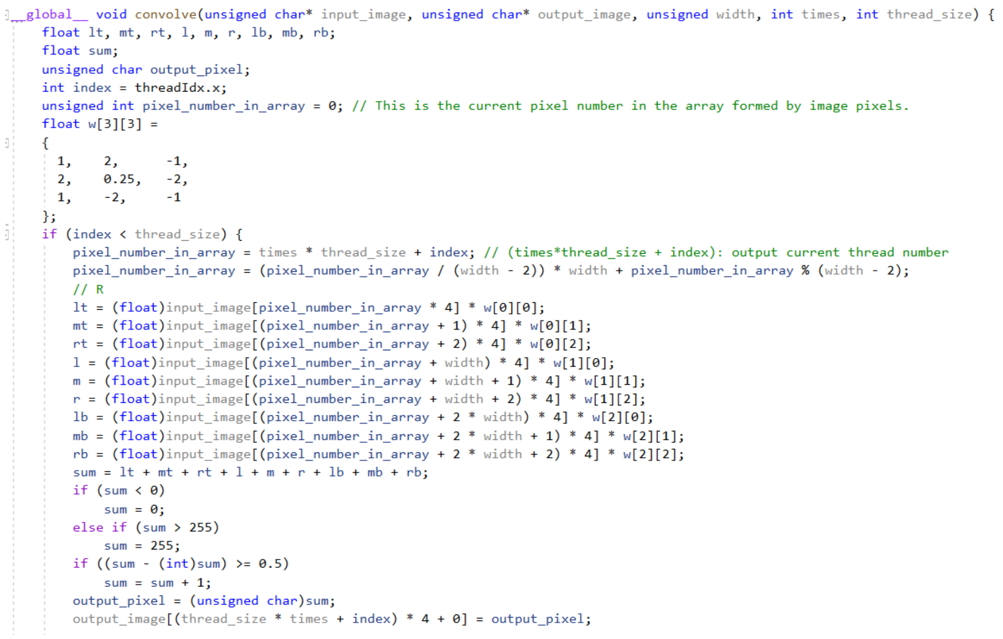
Hanwen Wang 260778557

Mai Zeng 260782174

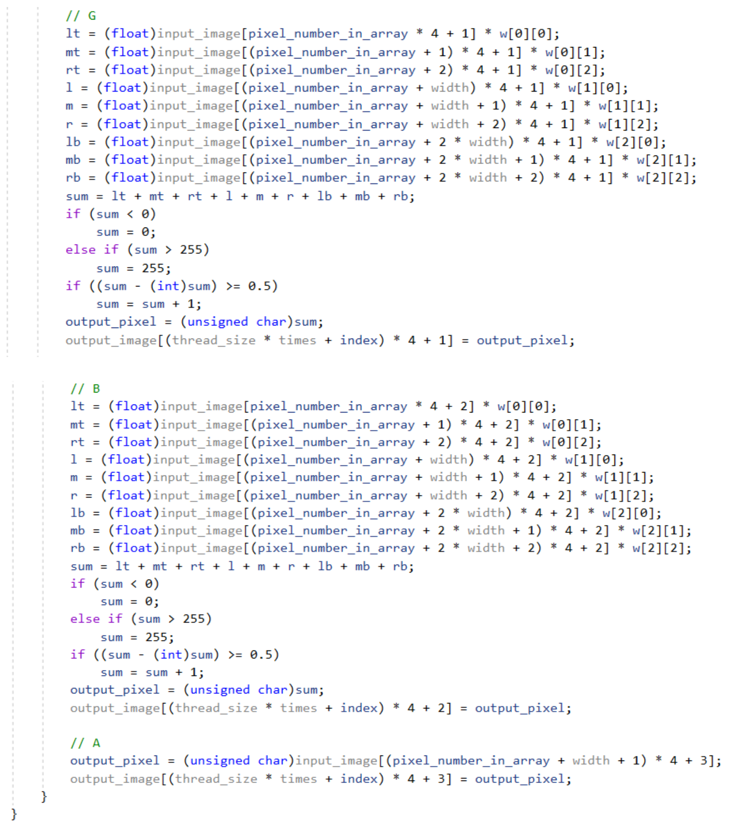
A. Convolution

Code for convolution

In this question, we are asked to implement convolution using a 3x3 weight matrix for the 3 test images provided. Inside a 3x3 matrix, there are 9 elements in total. We multiply each of them to its corresponding position in the image, which we stored in a one-dimensional array. Since the weighted matrix provided has a decimal number inside, we set all the values and sum to float in order to do the calculation. The matrix in the image starts from top left. As a result, when switching to next row, we need to add the value of width of image to the position in the array because that is where that corresponding position in the image is located. After the sum is calculated, we need to make sure it is in range 0 to 255, we set it to 0 or 255 if the value is smaller than 0 or greater than 255. Before we finally store this value into the output image array, we round that value to the nearest integer to reduce error. This is the general idea of convolution. We need to repeat it 3 times because all R, G and B need to follow the same step. The only difference between these three is their position in array. G is behind R by 1 and B is behind G by 1. For the value of A, which is behind B by 1, in each matrix, we let it equal to that of the middle element in the matrix. This will give us the best result compared to that of other elements in the matrix. The snippet of code for convolution logic is shown below.



```
__global__ void convolve(unsigned char* input_image, unsigned char* output_image, unsigned width, int times, int thread_size) {
    float lt, mt, rt, l, m, r, lb, mb, rb;
    float sum;
    unsigned char output_pixel;
    int index = threadIdx.x;
    unsigned int pixel_number_in_array = 0; // This is the current pixel number in the array formed by image pixels.
    float w[3][3] =
    {
      1,    2,      -1,
      2,    0.25,   -2,
      1,    -2,     -1
    };
    if (index < thread_size) {
        pixel_number_in_array = times * thread_size + index; // (times*thread_size + index): output current thread number
        pixel_number_in_array = (pixel_number_in_array / (width - 2)) * width + pixel_number_in_array % (width - 2);
        // R
        lt = (float)input_image[pixel_number_in_array * 4] * w[0][0];
        mt = (float)input_image[(pixel_number_in_array + 1) * 4] * w[0][1];
        rt = (float)input_image[(pixel_number_in_array + 2) * 4] * w[0][2];
        l = (float)input_image[(pixel_number_in_array + width) * 4] * w[1][0];
        m = (float)input_image[(pixel_number_in_array + width + 1) * 4] * w[1][1];
        r = (float)input_image[(pixel_number_in_array + width + 2) * 4] * w[1][2];
        lb = (float)input_image[(pixel_number_in_array + 2 * width) * 4] * w[2][0];
        mb = (float)input_image[(pixel_number_in_array + 2 * width + 1) * 4] * w[2][1];
        rb = (float)input_image[(pixel_number_in_array + 2 * width + 2) * 4] * w[2][2];
        sum = lt + mt + rt + l + m + r + lb + mb + rb;
        if (sum < 0)
            sum = 0;
        else if (sum > 255)
            sum = 255;
        if ((sum - (int)sum) >= 0.5)
            sum = sum + 1;
        output_pixel = (unsigned char)sum;
        output_image[(thread_size * times + index) * 4 + 0] = output_pixel;
```

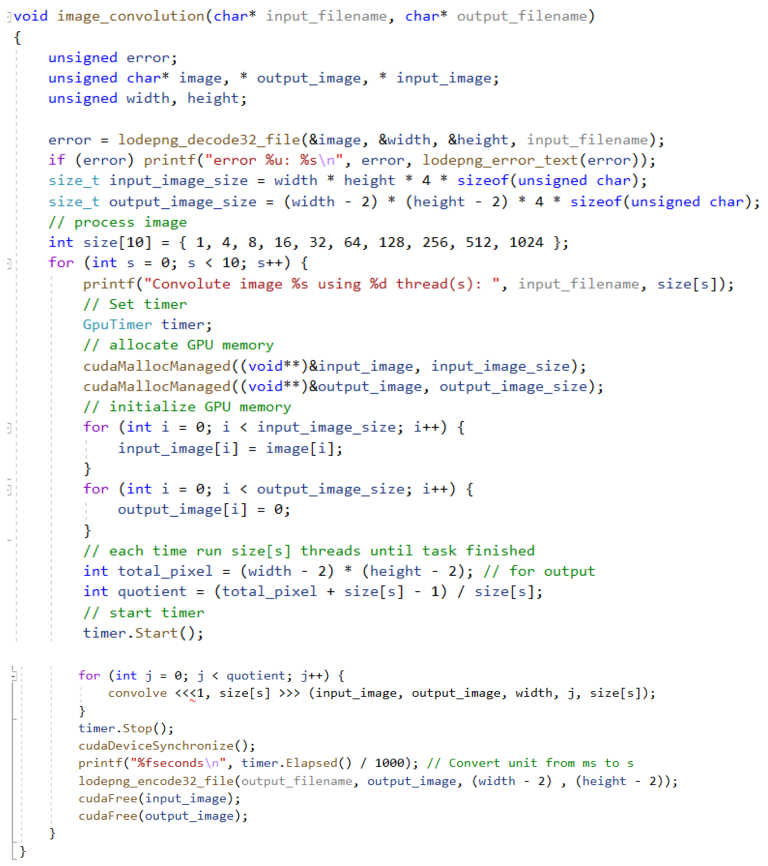


```
        // G
        lt = (float)input_image[pixel_number_in_array * 4 + 1] * w[0][0];
        mt = (float)input_image[(pixel_number_in_array + 1) * 4 + 1] * w[0][1];
        rt = (float)input_image[(pixel_number_in_array + 2) * 4 + 1] * w[0][2];
        l = (float)input_image[(pixel_number_in_array + width) * 4 + 1] * w[1][0];
        m = (float)input_image[(pixel_number_in_array + width + 1) * 4 + 1] * w[1][1];
        r = (float)input_image[(pixel_number_in_array + width + 2) * 4 + 1] * w[1][2];
        lb = (float)input_image[(pixel_number_in_array + 2 * width) * 4 + 1] * w[2][0];
        mb = (float)input_image[(pixel_number_in_array + 2 * width + 1) * 4 + 1] * w[2][1];
        rb = (float)input_image[(pixel_number_in_array + 2 * width + 2) * 4 + 1] * w[2][2];
        sum = lt + mt + rt + l + m + r + lb + mb + rb;
        if (sum < 0)
            sum = 0;
        else if (sum > 255)
            sum = 255;
        if ((sum - (int)sum) >= 0.5)
            sum = sum + 1;
        output_pixel = (unsigned char)sum;
        output_image[(thread_size * times + index) * 4 + 1] = output_pixel;

        // B
        lt = (float)input_image[pixel_number_in_array * 4 + 2] * w[0][0];
        mt = (float)input_image[(pixel_number_in_array + 1) * 4 + 2] * w[0][1];
        rt = (float)input_image[(pixel_number_in_array + 2) * 4 + 2] * w[0][2];
        l = (float)input_image[(pixel_number_in_array + width) * 4 + 2] * w[1][0];
        m = (float)input_image[(pixel_number_in_array + width + 1) * 4 + 2] * w[1][1];
        r = (float)input_image[(pixel_number_in_array + width + 2) * 4 + 2] * w[1][2];
        lb = (float)input_image[(pixel_number_in_array + 2 * width) * 4 + 2] * w[2][0];
        mb = (float)input_image[(pixel_number_in_array + 2 * width + 1) * 4 + 2] * w[2][1];
        rb = (float)input_image[(pixel_number_in_array + 2 * width + 2) * 4 + 2] * w[2][2];
        sum = lt + mt + rt + l + m + r + lb + mb + rb;
        if (sum < 0)
            sum = 0;
        else if (sum > 255)
            sum = 255;
        if ((sum - (int)sum) >= 0.5)
            sum = sum + 1;
        output_pixel = (unsigned char)sum;
        output_image[(thread_size * times + index) * 4 + 2] = output_pixel;

        // A
        output_pixel = (unsigned char)input_image[(pixel_number_in_array + width + 1) * 4 + 3];
        output_image[(thread_size * times + index) * 4 + 3] = output_pixel;
    }
}
```

Figure 1: Screenshot of code of convolution logic

Parallelization scheme

We run this parallel program in unified memory since we use cudaMallocManaged combined with cudaFree. During convolution, we will lose 1 pixel in height on the top, 1 pixel in height on the bottom, 1 pixel in width on the left and 1 pixel in width on the right. As a result, if the input image size is (width * height), the output image size should be (width - 2) * (height - 2). For this assignment, the maximum number of threads running each time is 1024, thus 1 block is enough. In the convolve global method above, we set index = threadIdx.x, and in the image_convolution method shown below, we run the global method using convolution <<<1, size[s]>>>, these mean that we only use 1 block with multiple threads to run this program. In number of threads we set to run each time, each thread is supposed to get 1 pixel in the output image. So we calculate the total number of runs we need first and then we run that many times of convolve to get the convolved image. The snippet of code for parallel and memory settings is shown below.

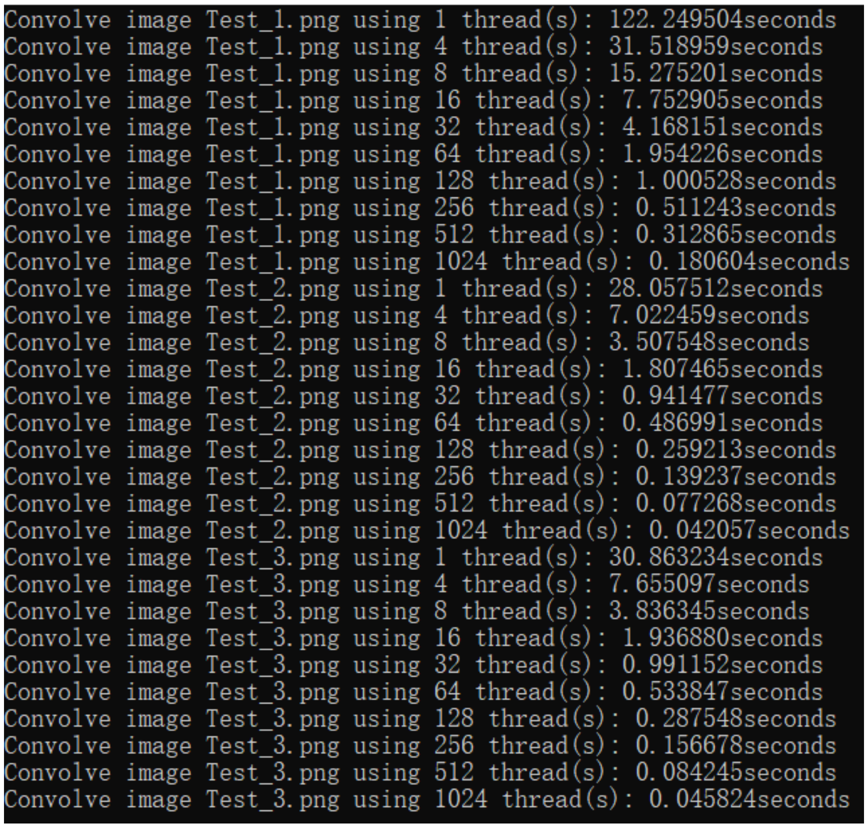


```
void image_convolution(char* input_filename, char* output_filename)
{
    unsigned error;
    unsigned char* image, * output_image, * input_image;
    unsigned width, height;

    error = lodepng_decode32_file(&image, &width, &height, input_filename);
    if (error) printf("error %u: %s\n", error, lodepng_error_text(error));
    size_t input_image_size = width * height * 4 * sizeof(unsigned char);
    size_t output_image_size = (width - 2) * (height - 2) * 4 * sizeof(unsigned char);
    // process image
    int size[10] = { 1, 4, 8, 16, 32, 64, 128, 256, 512, 1024 };
    for (int s = 0; s < 10; s++) {
        printf("Convolute image %s using %d thread(s): ", input_filename, size[s]);
        // Set timer
        GpuTimer timer;
        // allocate GPU memory
        cudaMallocManaged((void**)&input_image, input_image_size);
        cudaMallocManaged((void**)&output_image, output_image_size);
        // initialize GPU memory
        for (int i = 0; i < input_image_size; i++) {
            input_image[i] = image[i];
        }
        for (int i = 0; i < output_image_size; i++) {
            output_image[i] = 0;
        }
        // each time run size[s] threads until task finished
        int total_pixel = (width - 2) * (height - 2); // for output
        int quotient = (total_pixel + size[s] - 1) / size[s];
        // start timer
        timer.Start();

        for (int j = 0; j < quotient; j++) {
            convolve <<<1, size[s] >>> (input_image, output_image, width, j, size[s]);
        }
        timer.Stop();
        cudaDeviceSynchronize();
        printf("%fseconds\n", timer.Elapsed() / 1000); // Convert unit from ms to s
        lodepng_encode32_file(output_filename, output_image, (width - 2) , (height - 2));
        cudaFree(input_image);
        cudaFree(output_image);
    }
}
```

Figure 2: Screenshot of code of image_convolution function

Runtime of different numbers of threads

In image_convolution function, a timer is also set up, which aims to record the running time of convolution under different numbers of threads. We start the timer right before we call the function and stop the timer right after the function is over to ensure its accuracy. Results listed in Microsoft Visual Studio Debug Console is shown in the screenshot below.



```
Convolve image Test_1.png using 1 thread(s): 122.249504seconds
Convolve image Test_1.png using 4 thread(s): 31.518959seconds
Convolve image Test_1.png using 8 thread(s): 15.275201seconds
Convolve image Test_1.png using 16 thread(s): 7.752905seconds
Convolve image Test_1.png using 32 thread(s): 4.168151seconds
Convolve image Test_1.png using 64 thread(s): 1.954226seconds
Convolve image Test_1.png using 128 thread(s): 1.000528seconds
Convolve image Test_1.png using 256 thread(s): 0.511243seconds
Convolve image Test_1.png using 512 thread(s): 0.312865seconds
Convolve image Test_1.png using 1024 thread(s): 0.180604seconds
Convolve image Test_2.png using 1 thread(s): 28.057512seconds
Convolve image Test_2.png using 4 thread(s): 7.022459seconds
Convolve image Test_2.png using 8 thread(s): 3.507548seconds
Convolve image Test_2.png using 16 thread(s): 1.807465seconds
Convolve image Test_2.png using 32 thread(s): 0.941477seconds
Convolve image Test_2.png using 64 thread(s): 0.486991seconds
Convolve image Test_2.png using 128 thread(s): 0.259213seconds
Convolve image Test_2.png using 256 thread(s): 0.139237seconds
Convolve image Test_2.png using 512 thread(s): 0.077268seconds
Convolve image Test_2.png using 1024 thread(s): 0.042057seconds
Convolve image Test_3.png using 1 thread(s): 30.863234seconds
Convolve image Test_3.png using 4 thread(s): 7.655097seconds
Convolve image Test_3.png using 8 thread(s): 3.836345seconds
Convolve image Test_3.png using 16 thread(s): 1.936880seconds
Convolve image Test_3.png using 32 thread(s): 0.991152seconds
Convolve image Test_3.png using 64 thread(s): 0.533847seconds
Convolve image Test_3.png using 128 thread(s): 0.287548seconds
Convolve image Test_3.png using 256 thread(s): 0.156678seconds
Convolve image Test_3.png using 512 thread(s): 0.084245seconds
Convolve image Test_3.png using 1024 thread(s): 0.045824seconds
```

Figure 3: Screenshot of run time under different numbers of threads

Speedup

Speedup is the ratio of the serial runtime 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. Its equation is $S= T_s/T_p$. The speedup graphs for all 3 test images are listed below and all data can be found in “Thread vs Time.xlsx” file.

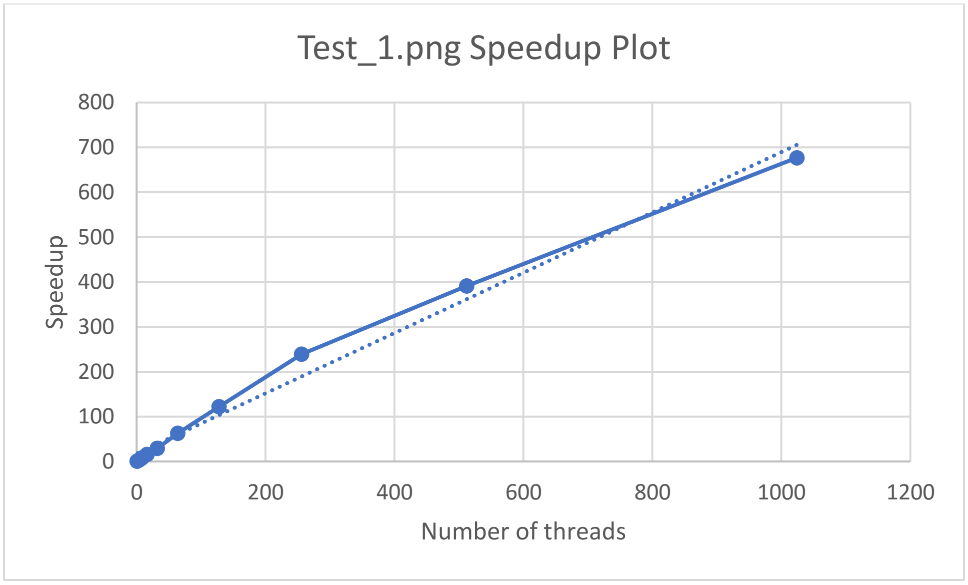


Figure 4: Speedup plot of Test_1.png

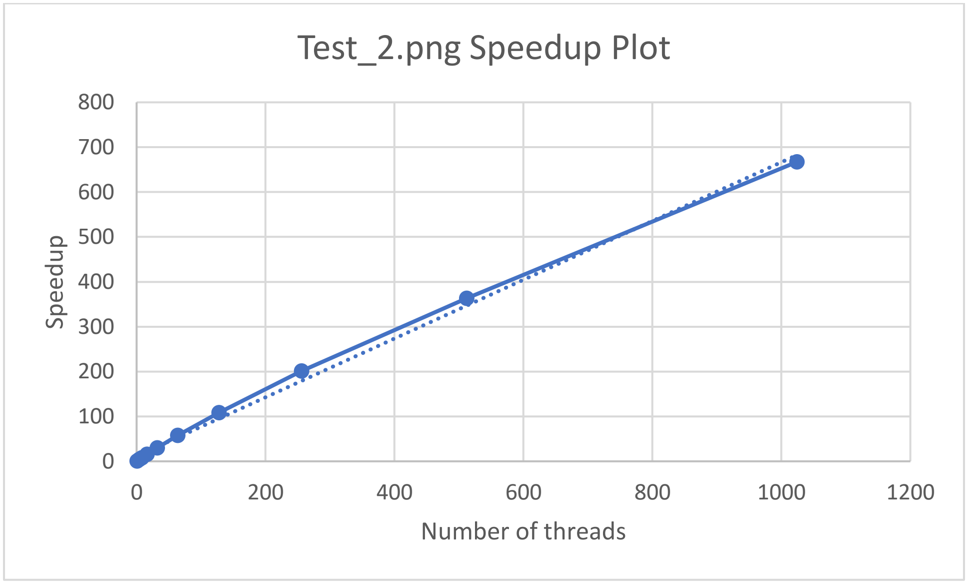


Figure 5: Speedup plot of Test_2.png

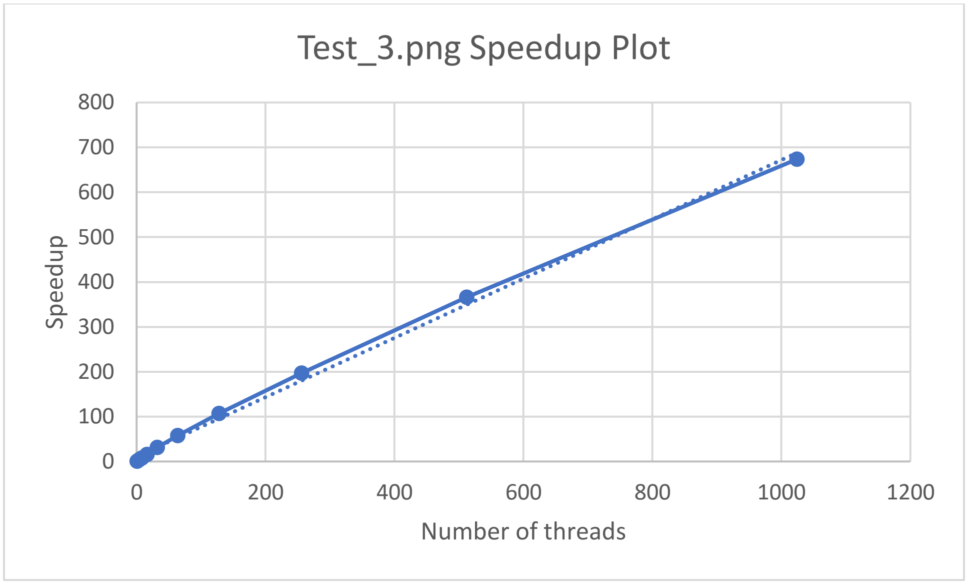
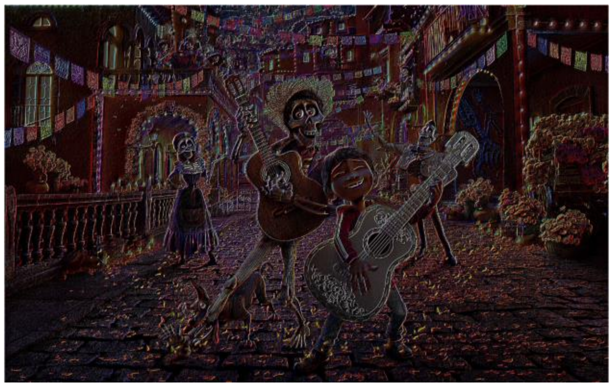
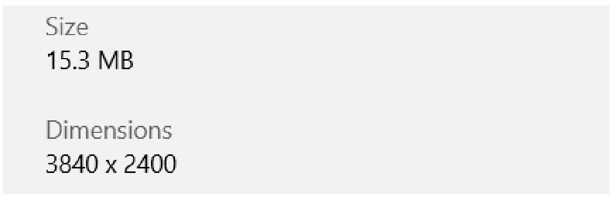


Figure 6: Speedup plot of Test_3.png

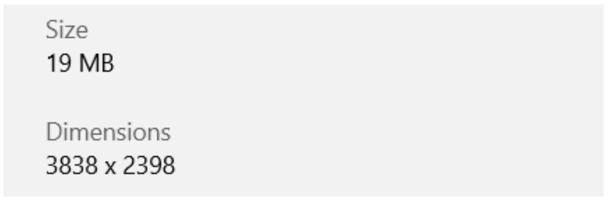
In the figures above, we can see that speedup is increasing almost linearly with the increase in number of threads working in parallel. Also, the speedup values for each image in the same number of threads are quite similar. It is because we are using one block multiple threads method. We need to process the image using $N_{parallel} + N_{serial}$ operations before, then with the threads increasing, now what we need is $\frac{N_{parallel}}{Number\ of\ threads} + N_{serial}$ operations.

Comparison of original image and convolved image

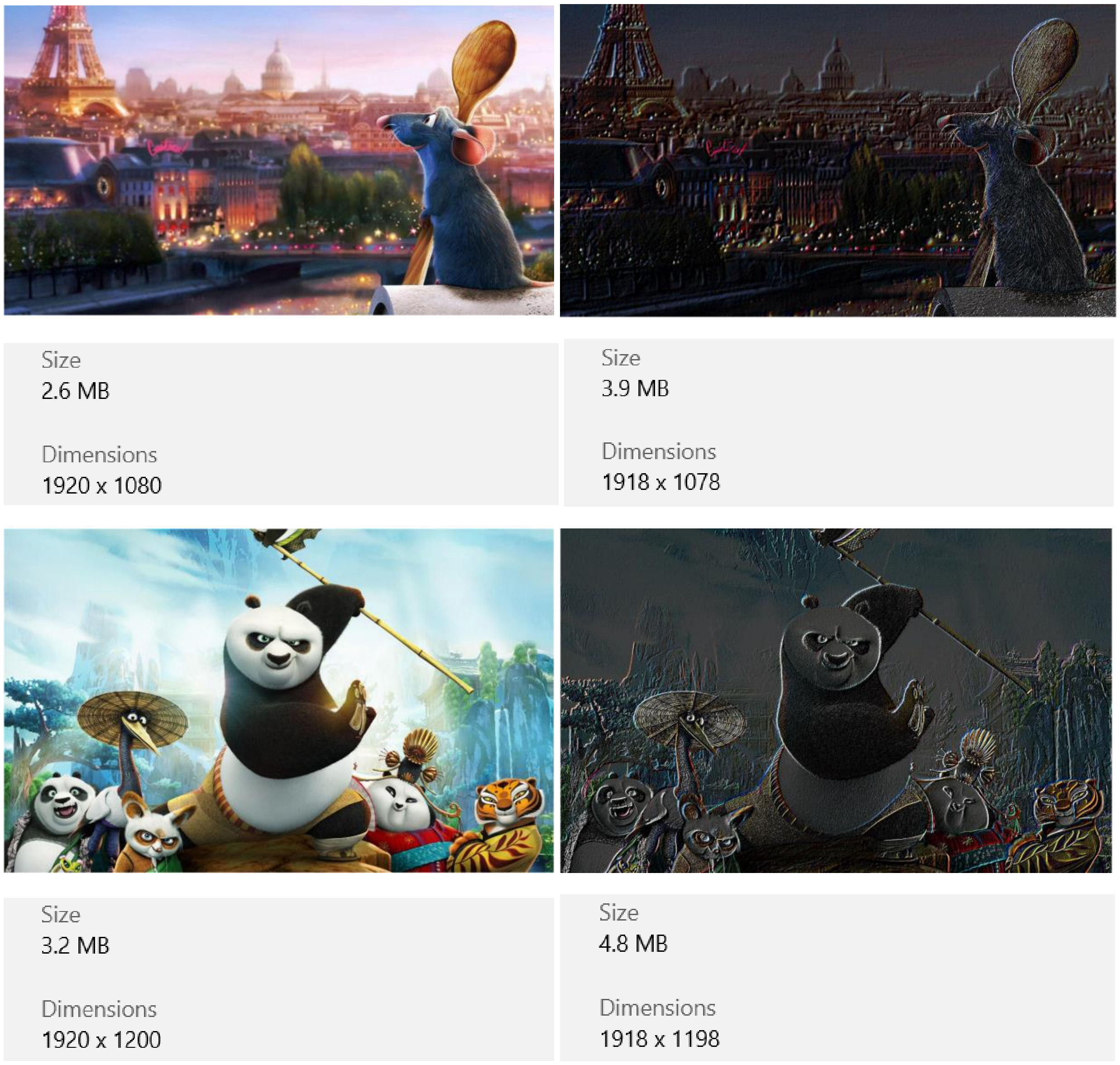



Size
15.3 MB

Dimensions
3840 x 2400


Size
19 MB

Dimensions
3838 x 2398



Size
2.6 MB

Dimensions
1920 x 1080

Size
3.9 MB

Dimensions
1918 x 1078

Size
3.2 MB

Dimensions
1920 x 1200

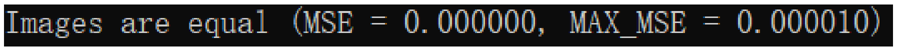
Size
4.8 MB

Dimensions
1918 x 1198

Figure 7: Original image (left) and Convolved image (right)

Test equality

When we use “test_equality.c” to compare convolved image provided and convolved image we get for Test_1.png, it shows that those two images are the same.



```
Images are equal (MSE = 0.000000, MAX_MSE = 0.000010)
```

Figure 8: Screenshot of test_equality.c result

B. Finite Element Music Synthesis

Sequential 4x4

In this question, we are asked to implement a 4 by 4 finite element grid sequentially. First, we are going to calculate the interior element, then we calculate boundaries based on values of interior element, at last we calculate four corners based on values of boundaries. The logic of this function is shown in the figure below. Arrows in the figure represent one value is based on the other.

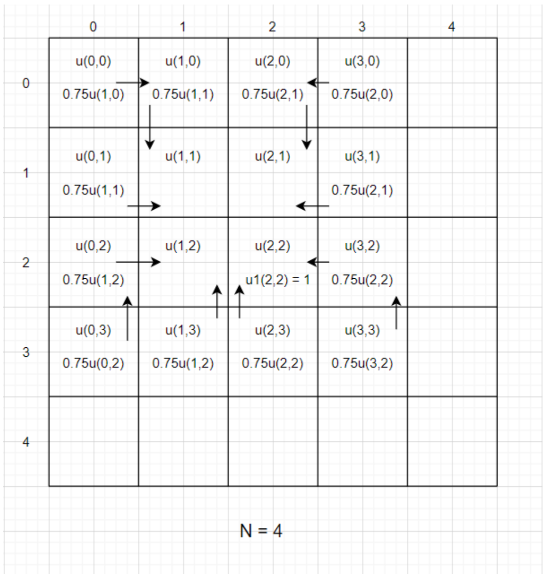
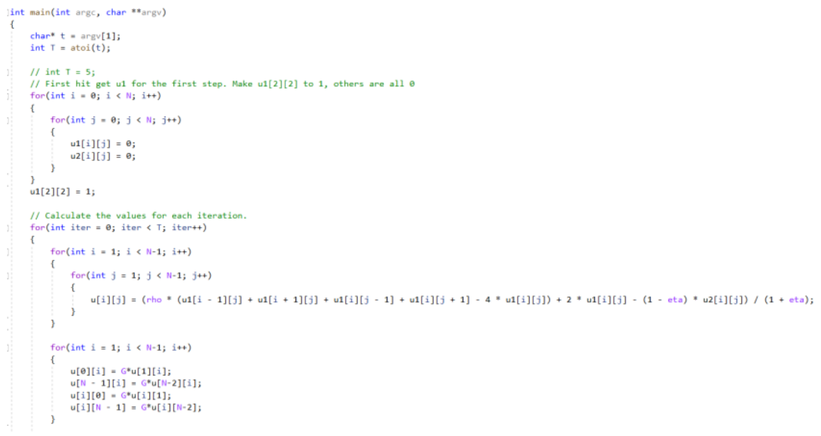
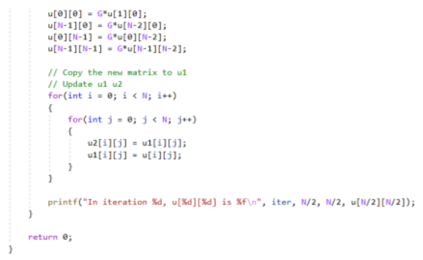


Figure 9: Screenshot of logic of 4 by 4 finite element grid

Since calculating interior element value will depend on values in previous time step and previous previous time step, we set all u1 and u2 to 0 except u1[2][2] to 1 in the first iteration to start the whole process. After each iteration is completed, we update u2 and u1 by setting u2 equals to u1 and u1 equals to current u. What is more, we take the first input string after the executable file name in the command line as T, which is the number of iterations to run the simulation. The snippet of code is shown below and it follows the logic described above with actual values and processes of calculation.



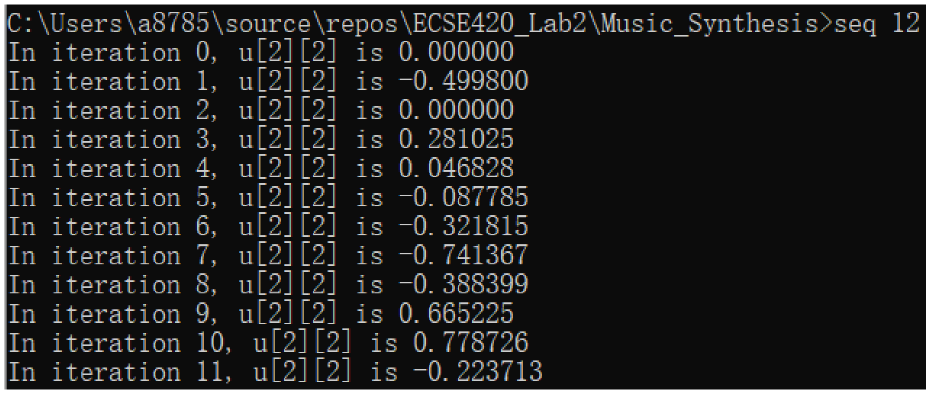
```
int main(int argc, char **argv)
{
    char* t = argv[1];
    int T = atoi(t);

    // int T = 5;
    // First hit get u1 for the first step. Make u1[2][2] to 1, others are all 0
    for(int i = 0; i < N; i++)
    {
        for(int j = 0; j < N; j++)
        {
            u1[i][j] = 0;
            u2[i][j] = 0;
        }
    }
    u1[2][2] = 1;

    // Calculate the values for each iteration.
    for(int iter = 0; iter < T; iter++)
    {
        for(int i = 1; i < N-1; i++)
        {
            for(int j = 1; j < N-1; j++)
            {
                u[i][j] = (rho * (u1[i - 1][j] + u1[i + 1][j] + u1[i][j - 1] + u1[i][j + 1] - 4 * u1[i][j]) + 2 * u1[i][j] - (1 - eta) * u2[i][j]) / (1 + eta);
            }
        }

        for(int i = 1; i < N-1; i++)
        {
            u[0][i] = G*u[1][i];
            u[N - 1][i] = G*u[N-2][i];
            u[i][0] = G*u[i][1];
            u[i][N - 1] = G*u[i][N-2];
        }
```



```
        u[0][0] = G*u[1][0];
        u[N-1][0] = G*u[N-2][0];
        u[0][N-1] = G*u[0][N-2];
        u[N-1][N-1] = G*u[N-1][N-2];

        // Copy the new matrix to u1
        // Update u1 u2
        for(int i = 0; i < N; i++)
        {
            for(int j = 0; j < N; j++)
            {
                u2[i][j] = u1[i][j];
                u1[i][j] = u[i][j];
            }
        }

        printf("In iteration %d, u[%d][%d] is %f\n", iter, N/2, N/2, u[N/2][N/2]);
    }

    return 0;
}
```

Figure 10: Screenshot of main part of sequential code

Values in u[2][2] are shown in the figure below for 12 iterations. We can see the general trend that values are increasing in magnitude. This makes sense because only setting some position in u1 to 1 to start is to simulate a hit on the drum. We can imagine that when we hit on a drum, the drumhead will vibrate up and down increasingly. After it reaches its maximum magnitude, it will vibrate decreasingly until it becomes stationary. Values shown below describe this increasing trend. If we increase the number of iterations, we can also detect the decreasing trend and even the final stationary state.

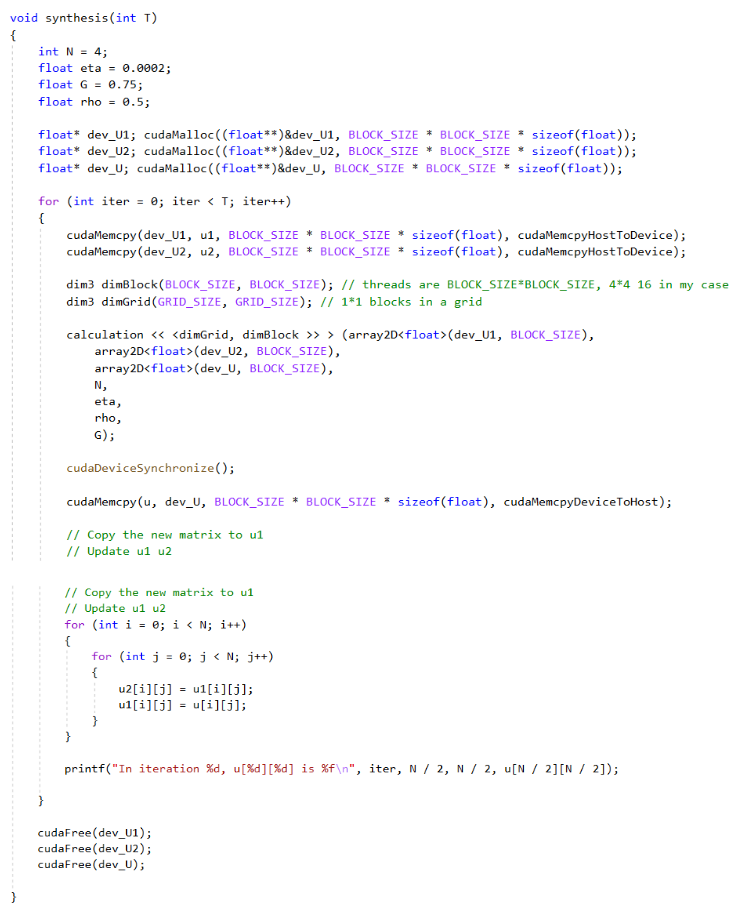


```
C:\Users\a8785\source\repos\ECSE420_Lab2\Music_Synthesis>seq 12
In iteration 0, u[2][2] is 0.000000
In iteration 1, u[2][2] is -0.499800
In iteration 2, u[2][2] is 0.000000
In iteration 3, u[2][2] is 0.281025
In iteration 4, u[2][2] is 0.046828
In iteration 5, u[2][2] is -0.087785
In iteration 6, u[2][2] is -0.321815
In iteration 7, u[2][2] is -0.741367
In iteration 8, u[2][2] is -0.388399
In iteration 9, u[2][2] is 0.665225
In iteration 10, u[2][2] is 0.778726
In iteration 11, u[2][2] is -0.223713
```

Figure 11: Screenshot of values in u[2][2] in each iteration

Parallel 4x4

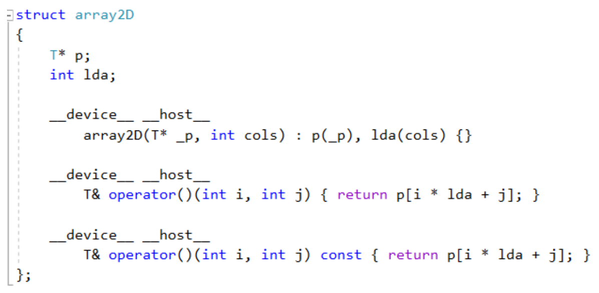
In this question, we are asked to parallelize our code in the former question. We choose to run this code in explicit memory because we use cudaMalloc and cudaMemcpy(cudaMemcpyHostToDevice) with cudaFree and cudaMemcpy(cudaMemcpyDeviceToHost) to update u, u1 and u2 in each iteration. We do not allocate the matrix memory in heap, instead we directly create it on the stack, thus we do not use malloc with free in the function. In order to create a 2D matrix in the GPU, we use dim3 variables to set the Grid and Block dimensions. More details can be found in the later question since that case is a more general case. The snippet of code for memory allocation and calling parallel function is shown below.



```
void synthesis(int T)
{
    int N = 4;
    float eta = 0.0002;
    float G = 0.75;
    float rho = 0.5;

    float* dev_U1; cudaMalloc((float**)&dev_U1, BLOCK_SIZE * BLOCK_SIZE * sizeof(float));
    float* dev_U2; cudaMalloc((float**)&dev_U2, BLOCK_SIZE * BLOCK_SIZE * sizeof(float));
    float* dev_U; cudaMalloc((float**)&dev_U, BLOCK_SIZE * BLOCK_SIZE * sizeof(float));

    for (int iter = 0; iter < T; iter++)
    {
        cudaMemcpy(dev_U1, u1, BLOCK_SIZE * BLOCK_SIZE * sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(dev_U2, u2, BLOCK_SIZE * BLOCK_SIZE * sizeof(float), cudaMemcpyHostToDevice);

        dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE); // threads are BLOCK_SIZE*BLOCK_SIZE, 4*4 16 in my case
        dim3 dimGrid(GRID_SIZE, GRID_SIZE); // 1*1 blocks in a grid

        calculation << <dimGrid, dimBlock >> > (array2D<float>(dev_U1, BLOCK_SIZE),
            array2D<float>(dev_U2, BLOCK_SIZE),
            array2D<float>(dev_U, BLOCK_SIZE),
            N,
            eta,
            rho,
            G);

        cudaDeviceSynchronize();

        cudaMemcpy(u, dev_U, BLOCK_SIZE * BLOCK_SIZE * sizeof(float), cudaMemcpyDeviceToHost);

        // Copy the new matrix to u1
        // Update u1 u2

        // Copy the new matrix to u1
        // Update u1 u2
        for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < N; j++)
            {
                u2[i][j] = u1[i][j];
                u1[i][j] = u[i][j];
            }
        }

        printf("In iteration %d, u[%d][%d] is %f\n", iter, N / 2, N / 2, u[N / 2][N / 2]);

    }

    cudaFree(dev_U1);
    cudaFree(dev_U2);
    cudaFree(dev_U);

}
```

Figure 12: Screenshot of memory allocation parallel code

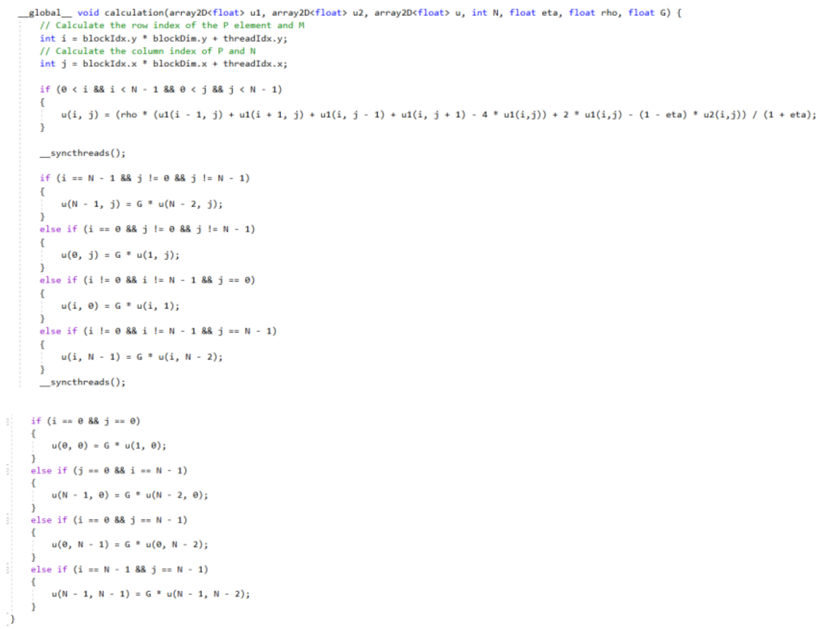
We create a struct in C++ to make the code for processing the 2D array in the GPU function more light weight. The code snippet below shows how we design the struct. In the struct, we have a pointer to point to the original address of the 2D array and a variable lda which is the how many elements are there for each row. Then we have a function to computer where array2D[i,j] is in the address since actually the 2D array is just one 1D array.



```
struct array2D
{
    T* p;
    int lda;

    __device__ __host__
        array2D(T* _p, int cols) : p(_p), lda(cols) {}

    __device__ __host__
        T& operator()(int i, int j) { return p[i * lda + j]; }

    __device__ __host__
        T& operator()(int i, int j) const { return p[i * lda + j]; }
};
```

Figure 13: Code Snippet for array2D struct

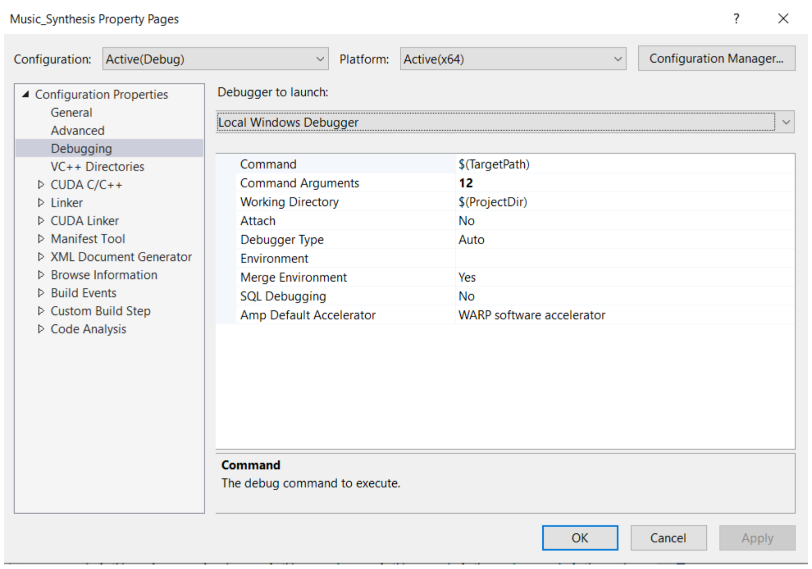
For the logic function, we have multiple blocks with multiple threads because we use blockIdx * blockDim + threadIdx and each thread handles a single node of this grid. Thus we can directly compare values of i and j to values related to N to do the calculation, this condition will vary in the next question because number of finite elements per thread is no longer 1. In this question, we first calculate all the interior elements in parallel, then we use __syncthreads to wait until all threads calculating interior elements are finished. After that, we sequentially move on to calculate boundary elements and __syncthreads is used again to wait until all threads finish calculating. At last, we sequentially move on to calculate corner elements. This is the general logic of this parallel function. Calculation logic is very similar to the sequential part, the main difference is that we calculate interior elements, boundary elements and corner elements in parallel separately and connect them in sequence. We cannot parallelize them all due to the reason we discussed in previous question. We must calculate interior elements first, then boundary elements and at last corner elements. The snippet of code for logic of parallel function is shown below.



```
__global__ void calculation(array2D<float> u1, array2D<float> u2, array2D<float> u, int N, float eta, float rho, float G) {
    // Calculate the row index of the P element and M
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    // Calculate the column index of P and N
    int j = blockIdx.x * blockDim.x + threadIdx.x;

    if (0 < i && i < N - 1 && 0 < j && j < N - 1)
    {
        u(i, j) = (rho * (u1(i - 1, j) + u1(i + 1, j) + u1(i, j - 1) + u1(i, j + 1) - 4 * u1(i,j)) + 2 * u1(i,j) - (1 - eta) * u2(i,j)) / (1 + eta);
    }

    __syncthreads();

    if (i == N - 1 && j != 0 && j != N - 1)
    {
        u(N - 1, j) = G * u(N - 2, j);
    }
    else if (i == 0 && j != 0 && j != N - 1)
    {
        u(0, j) = G * u(1, j);
    }
    else if (i != 0 && i != N - 1 && j == 0)
    {
        u(i, 0) = G * u(i, 1);
    }
    else if (i != 0 && i != N - 1 && j == N - 1)
    {
        u(i, N - 1) = G * u(i, N - 2);
    }
    __syncthreads();

    if (i == 0 && j == 0)
    {
        u(0, 0) = G * u(1, 0);
    }
    else if (j == 0 && i == N - 1)
    {
        u(N - 1, 0) = G * u(N - 2, 0);
    }
    else if (i == 0 && j == N - 1)
    {
        u(0, N - 1) = G * u(0, N - 2);
    }
    else if (i == N - 1 && j == N - 1)
    {
        u(N - 1, N - 1) = G * u(N - 1, N - 2);
    }
}
```

Figure 14: Screenshot of parallel function

Before we launch local windows debugger, we set command arguments in project properties to 12. We take this value as T, which specifies the number of iterations to run the simulation.



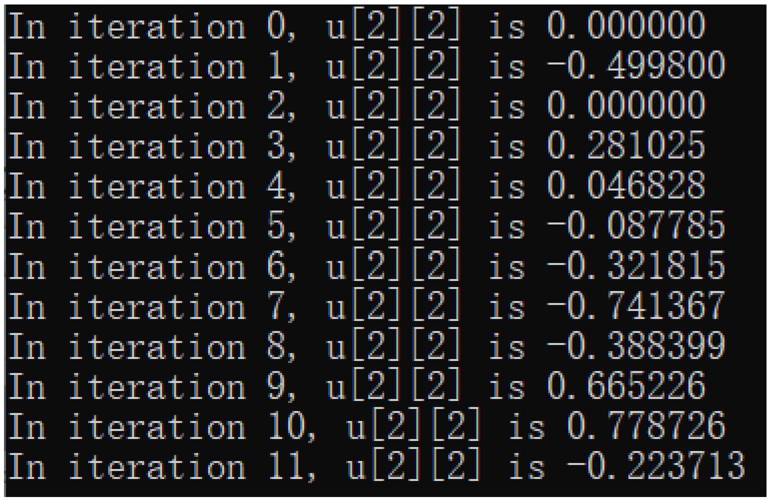
Debugger to launch: Local Windows Debugger

Command	$(TargetPath)
Command Arguments	**12**
Working Directory	$(ProjectDir)
Attach	No
Debugger Type	Auto
Environment	
Merge Environment	Yes
SQL Debugging	No
Amp Default Accelerator	WARP software accelerator

Command
The debug command to execute.

Figure 15: Screenshot of command argument

The results shown in u[2][2] is the same as those in the previous question. This means that our function is correct.

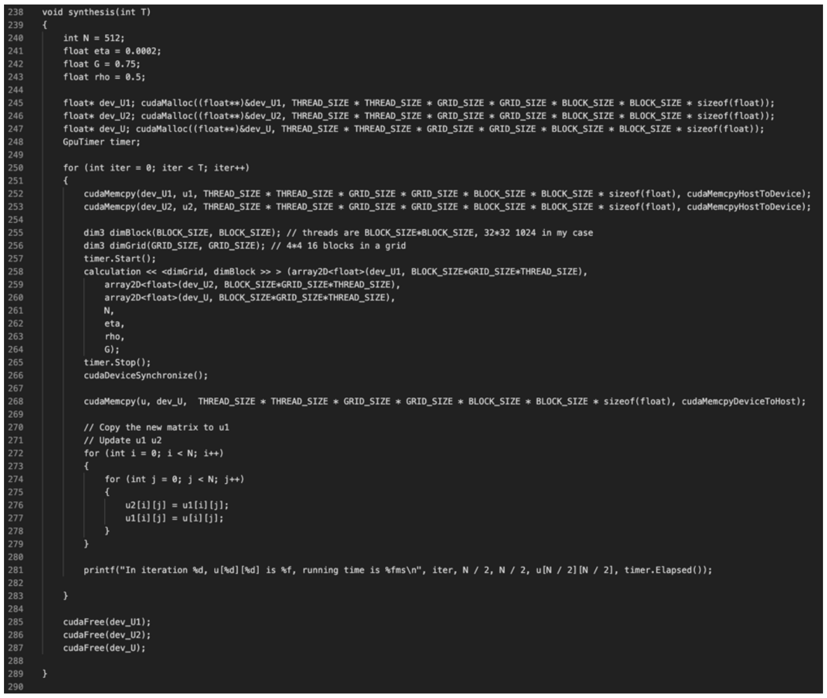


```
In iteration 0, u[2][2] is 0.000000
In iteration 1, u[2][2] is -0.499800
In iteration 2, u[2][2] is 0.000000
In iteration 3, u[2][2] is 0.281025
In iteration 4, u[2][2] is 0.046828
In iteration 5, u[2][2] is -0.087785
In iteration 6, u[2][2] is -0.321815
In iteration 7, u[2][2] is -0.741367
In iteration 8, u[2][2] is -0.388399
In iteration 9, u[2][2] is 0.665226
In iteration 10, u[2][2] is 0.778726
In iteration 11, u[2][2] is -0.223713
```

Figure 16: Screenshot of values in u[2][2] in each iteration

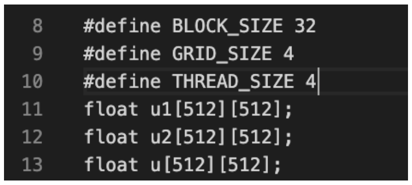
Parallel 512x512

The code snippet below shows how we achieve to allocate the memory for the GPU. We first use cudaMalloc to create 3 size of THREAD_SIZE*THREAD_SIZE*GRID_SIZE*GRID_SIZE*BLOCK_SIZE_BLOCK_SIZE*sizeof(float) memories in the GPU. The values of constant parameters BLOCK_SIZE, GRID_SIZE and THREAD_SIZE is 32, 4 and 4 in the case of 1024 threads 16 blocks and 16 finite elements per thread.



```
void synthesis(int T)
{
    int N = 512;
    float eta = 0.0002;
    float G = 0.75;
    float rho = 0.5;

    float* dev_U1; cudaMalloc((float**)&dev_U1, THREAD_SIZE * THREAD_SIZE * GRID_SIZE * GRID_SIZE * BLOCK_SIZE * BLOCK_SIZE * sizeof(float));
    float* dev_U2; cudaMalloc((float**)&dev_U2, THREAD_SIZE * THREAD_SIZE * GRID_SIZE * GRID_SIZE * BLOCK_SIZE * BLOCK_SIZE * sizeof(float));
    float* dev_U; cudaMalloc((float**)&dev_U, THREAD_SIZE * THREAD_SIZE * GRID_SIZE * GRID_SIZE * BLOCK_SIZE * BLOCK_SIZE * sizeof(float));
    GpuTimer timer;

    for (int iter = 0; iter < T; iter++)
    {
        cudaMemcpy(dev_U1, u1, THREAD_SIZE * THREAD_SIZE * GRID_SIZE * GRID_SIZE * BLOCK_SIZE * BLOCK_SIZE * sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(dev_U2, u2, THREAD_SIZE * THREAD_SIZE * GRID_SIZE * GRID_SIZE * BLOCK_SIZE * BLOCK_SIZE * sizeof(float), cudaMemcpyHostToDevice);

        dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE); // threads are BLOCK_SIZE*BLOCK_SIZE, 32*32 1024 in my case
        dim3 dimGrid(GRID_SIZE, GRID_SIZE); // 4*4 16 blocks in a grid
        timer.Start();
        calculation << <dimGrid, dimBlock >> > (array2D<float>(dev_U1, BLOCK_SIZE*GRID_SIZE*THREAD_SIZE),
            array2D<float>(dev_U2, BLOCK_SIZE*GRID_SIZE*THREAD_SIZE),
            array2D<float>(dev_U, BLOCK_SIZE*GRID_SIZE*THREAD_SIZE),
            N,
            eta,
            rho,
            G);
        timer.Stop();
        cudaDeviceSynchronize();

        cudaMemcpy(u, dev_U,  THREAD_SIZE * THREAD_SIZE * GRID_SIZE * GRID_SIZE * BLOCK_SIZE * BLOCK_SIZE * sizeof(float), cudaMemcpyDeviceToHost);

        // Copy the new matrix to u1
        // Update u1 u2
        for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < N; j++)
            {
                u2[i][j] = u1[i][j];
                u1[i][j] = u[i][j];
            }
        }

        printf("In iteration %d, u[%d][%d] is %f, running time is %fms\n", iter, N / 2, N / 2, u[N / 2][N / 2], timer.Elapsed());

    }

    cudaFree(dev_U1);
    cudaFree(dev_U2);
    cudaFree(dev_U);

}
```

Figure 17: Code Snippet for Memory Allocation

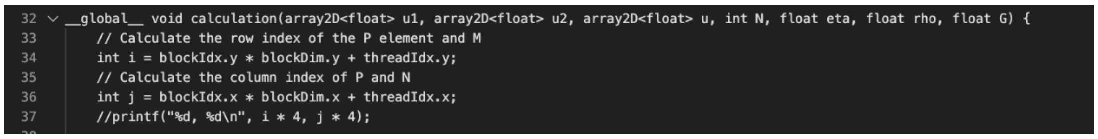


```
#define BLOCK_SIZE 32
#define GRID_SIZE 4
#define THREAD_SIZE 4
float u1[512][512];
float u2[512][512];
float u[512][512];
```

Figure 18: Code Snippet for BLOCK_SIZE GRID_SIZE and THREAD_SIZE

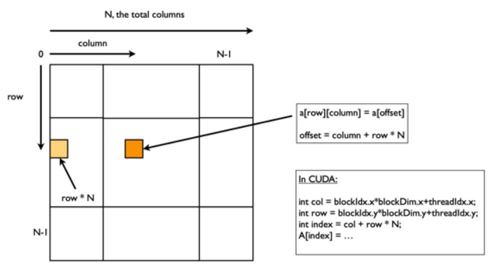
In order to create a 2D matrix in the GPU, we use dim3 variables to set the Grid and Block dimensions. In the first case which is 1024 threads 16 blocks. We need to use dimBlock(32, 32) in order to create 32x32 = 1024 threads per block. We need to use dimGrid(4,4) to create 4x4 blocks.

In the GPU function code snippet shown below, we use blockIdx.y*blockDim.y + threadIdx.y as row number and blockId.x*blockDim.x + threadIdx.x as column number. Note here this is the column and row number for “thread matrix” since in each thread we have to process 16 finite elements. The details of this implementation is shown in the figure below.



```
__global__ void calculation(array2D<float> u1, array2D<float> u2, array2D<float> u, int N, float eta, float rho, float G) {
    // Calculate the row index of the P element and M
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    // Calculate the column index of P and N
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    //printf("%d, %d\n", i * 4, j * 4);
```

Figure 19: Code Snippet For Find Row and Colum in 2D array



```
a[row][column] = a[offset]

offset = column + row * N
```

```
In CUDA:

int col = blockIdx.x*blockDim.x+threadIdx.x;
int row = blockIdx.y*blockDim.y+threadIdx.y;
int index = col + row * N;
A[index] = ...
```

Figure 20: 2D array in GPU

In order to simulate the drum, we need to take consider the element that we are processing is the interior element, boundary element or corner elements. As shown in the figure below. Since each thread we would be going to handle 16 elements (4x4), we need to use the row number and the column number of the current thread location in the "thread matrix" to compute the exact location of the element in the 2D array.

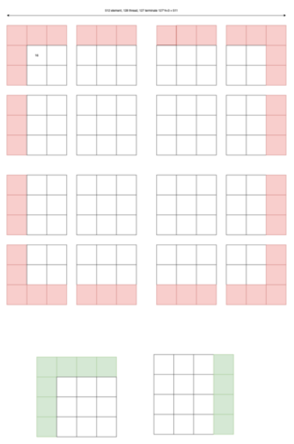
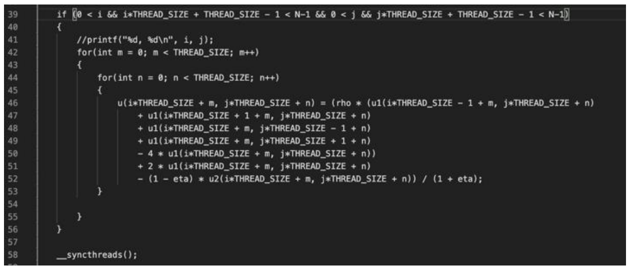


Figure 21: The whole matrix, upper left corner and right boundary visualization.

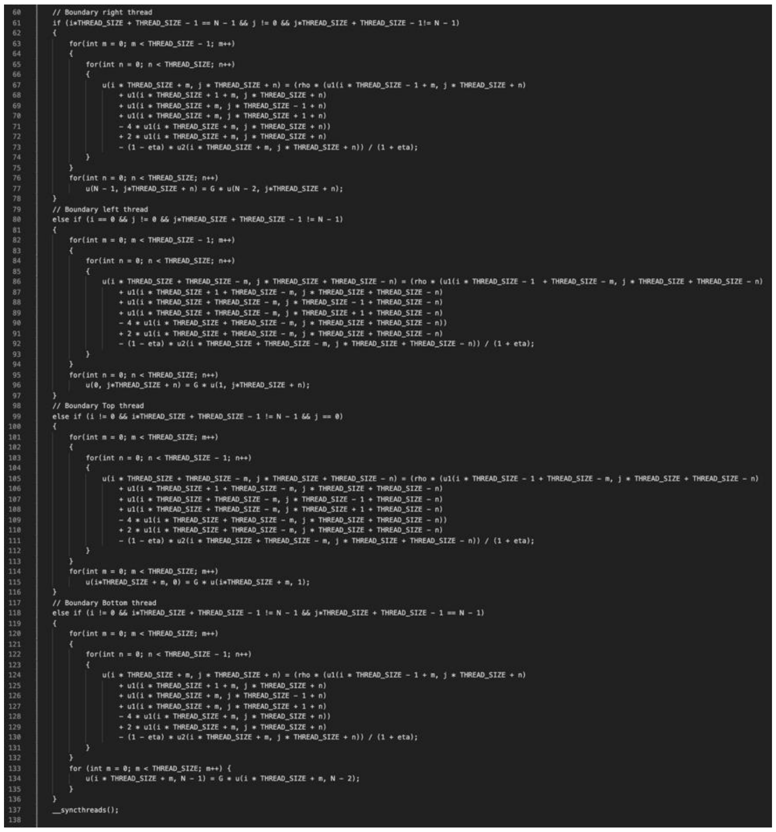
The code snippet below shows how we handle the interior elements. In the last line, we put the __syncthreads() there to set the barrier which means that every threads need to wait at that line in order to continue. The way we handle the interior elements is to create a nested for loop to process the 16 elements. Each for loop would traversal 4 times and two level nested for loop would traversal 4x4=16 elements. In each loop, we need to multiply the row and column number by its thread size, in the first case it is 4. If we look at the figure above, each thread will contain a 4x4 square matrix and that is the 16 finite elements we need to handle for this thread.



```
if (0 < i && i*THREAD_SIZE + THREAD_SIZE - 1 < N-1 && 0 < j && j*THREAD_SIZE + THREAD_SIZE - 1 < N-1)
{
    //printf("%d, %d\n", i, j);
    for(int m = 0; m < THREAD_SIZE; m++)
    {
        for(int n = 0; n < THREAD_SIZE; n++)
        {
            u(i*THREAD_SIZE + m, j*THREAD_SIZE + n) = (rho * (u1(i*THREAD_SIZE - 1 + m, j*THREAD_SIZE + n)
                + u1(i*THREAD_SIZE + 1 + m, j*THREAD_SIZE + n)
                + u1(i*THREAD_SIZE + m, j*THREAD_SIZE - 1 + n)
                + u1(i*THREAD_SIZE + m, j*THREAD_SIZE + 1 + n)
                - 4 * u1(i*THREAD_SIZE + m, j*THREAD_SIZE + n))
                + 2 * u1(i*THREAD_SIZE + m, j*THREAD_SIZE + n)
                - (1 - eta) * u2(i*THREAD_SIZE + m, j*THREAD_SIZE + n)) / (1 + eta);
        }

    }
}

__syncthreads();
```

Figure 22: Code Snippet for Interior Element

The code snippet below shows how we handle the boundary elements. Note here, the boundary thread would handle 16 finite elements as well but there are only 4 of 16 elements are actually the boundary elements. Just like the way we deal with the interior elements, we used nested for loop to handle 12 (3x4) elements which is the elements in the boundary threads but are actually the interior elements then we would use one for loop to handle the 4 boundary elements.



```
// Boundary right thread
if (i*THREAD_SIZE + THREAD_SIZE - 1 == N - 1 && j != 0 && j*THREAD_SIZE + THREAD_SIZE - 1!= N - 1)
{
    for(int m = 0; m < THREAD_SIZE - 1; m++)
    {
        for(int n = 0; n < THREAD_SIZE; n++)
        {
            u(i * THREAD_SIZE + m, j * THREAD_SIZE + n) = (rho * (u1(i * THREAD_SIZE - 1 + m, j * THREAD_SIZE + n)
                + u1(i * THREAD_SIZE + 1 + m, j * THREAD_SIZE + n)
                + u1(i * THREAD_SIZE + m, j * THREAD_SIZE - 1 + n)
                + u1(i * THREAD_SIZE + m, j * THREAD_SIZE + 1 + n)
                - 4 * u1(i * THREAD_SIZE + m, j * THREAD_SIZE + n))
                + 2 * u1(i * THREAD_SIZE + m, j * THREAD_SIZE + n)
                - (1 - eta) * u2(i * THREAD_SIZE + m, j * THREAD_SIZE + n)) / (1 + eta);
        }
    }
    for(int n = 0; n < THREAD_SIZE; n++)
        u(N - 1, j*THREAD_SIZE + n) = G * u(N - 2, j*THREAD_SIZE + n);
}
// Boundary left thread
else if (i == 0 && j != 0 && j*THREAD_SIZE + THREAD_SIZE - 1 != N - 1)
{
    for(int m = 0; m < THREAD_SIZE - 1; m++)
    {
        for(int n = 0; n < THREAD_SIZE; n++)
        {
            u(i * THREAD_SIZE + THREAD_SIZE - m, j * THREAD_SIZE + THREAD_SIZE - n) = (rho * (u1(i * THREAD_SIZE - 1  + THREAD_SIZE - m, j * THREAD_SIZE + THREAD_SIZE - n)
                + u1(i * THREAD_SIZE + 1 + THREAD_SIZE - m, j * THREAD_SIZE + THREAD_SIZE - n)
                + u1(i * THREAD_SIZE + THREAD_SIZE - m, j * THREAD_SIZE - 1 + THREAD_SIZE - n)
                + u1(i * THREAD_SIZE + THREAD_SIZE - m, j * THREAD_SIZE + 1 + THREAD_SIZE - n)
                - 4 * u1(i * THREAD_SIZE + THREAD_SIZE - m, j * THREAD_SIZE + THREAD_SIZE - n))
                + 2 * u1(i * THREAD_SIZE + THREAD_SIZE - m, j * THREAD_SIZE + THREAD_SIZE - n)
                - (1 - eta) * u2(i * THREAD_SIZE + THREAD_SIZE - m, j * THREAD_SIZE + THREAD_SIZE - n)) / (1 + eta);
        }
    }
    for(int n = 0; n < THREAD_SIZE; n++)
        u(0, j*THREAD_SIZE + n) = G * u(1, j*THREAD_SIZE + n);
}
// Boundary Top thread
else if (i != 0 && i*THREAD_SIZE + THREAD_SIZE - 1 != N - 1 && j == 0)
{
    for(int m = 0; m < THREAD_SIZE; m++)
    {
        for(int n = 0; n < THREAD_SIZE - 1; n++)
        {
            u(i * THREAD_SIZE + THREAD_SIZE - m, j * THREAD_SIZE + THREAD_SIZE - n) = (rho * (u1(i * THREAD_SIZE - 1 + THREAD_SIZE - m, j * THREAD_SIZE + THREAD_SIZE - n)
                + u1(i * THREAD_SIZE + 1 + THREAD_SIZE - m, j * THREAD_SIZE + THREAD_SIZE - n)
                + u1(i * THREAD_SIZE + THREAD_SIZE - m, j * THREAD_SIZE - 1 + THREAD_SIZE - n)
                + u1(i * THREAD_SIZE + THREAD_SIZE - m, j * THREAD_SIZE + 1 + THREAD_SIZE - n)
                - 4 * u1(i * THREAD_SIZE + THREAD_SIZE - m, j * THREAD_SIZE + THREAD_SIZE - n))
                + 2 * u1(i * THREAD_SIZE + THREAD_SIZE - m, j * THREAD_SIZE + THREAD_SIZE - n)
                - (1 - eta) * u2(i * THREAD_SIZE + THREAD_SIZE - m, j * THREAD_SIZE + THREAD_SIZE - n)) / (1 + eta);
        }
    }
    for(int m = 0; m < THREAD_SIZE; m++)
        u(i*THREAD_SIZE + m, 0) = G * u(i*THREAD_SIZE + m, 1);
}
// Boundary Bottom thread
else if (i != 0 && i*THREAD_SIZE + THREAD_SIZE - 1 != N - 1 && j*THREAD_SIZE + THREAD_SIZE - 1 == N - 1)
{
    for(int m = 0; m < THREAD_SIZE; m++)
    {
        for(int n = 0; n < THREAD_SIZE - 1; n++)
        {
            u(i * THREAD_SIZE + m, j * THREAD_SIZE + n) = (rho * (u1(i * THREAD_SIZE - 1 + m, j * THREAD_SIZE + n)
                + u1(i * THREAD_SIZE + 1 + m, j * THREAD_SIZE + n)
                + u1(i * THREAD_SIZE + m, j * THREAD_SIZE - 1 + n)
                + u1(i * THREAD_SIZE + m, j * THREAD_SIZE + 1 + n)
                - 4 * u1(i * THREAD_SIZE + m, j * THREAD_SIZE + n))
                + 2 * u1(i * THREAD_SIZE + m, j * THREAD_SIZE + n)
                - (1 - eta) * u2(i * THREAD_SIZE + m, j * THREAD_SIZE + n)) / (1 + eta);
        }
    }
    for (int m = 0; m < THREAD_SIZE; m++) {
        u(i * THREAD_SIZE + m, N - 1) = G * u(i * THREAD_SIZE + m, N - 2);
    }
}
__syncthreads();
```

Figure 23: Code Snippet for Boundary Element

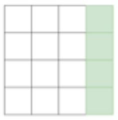
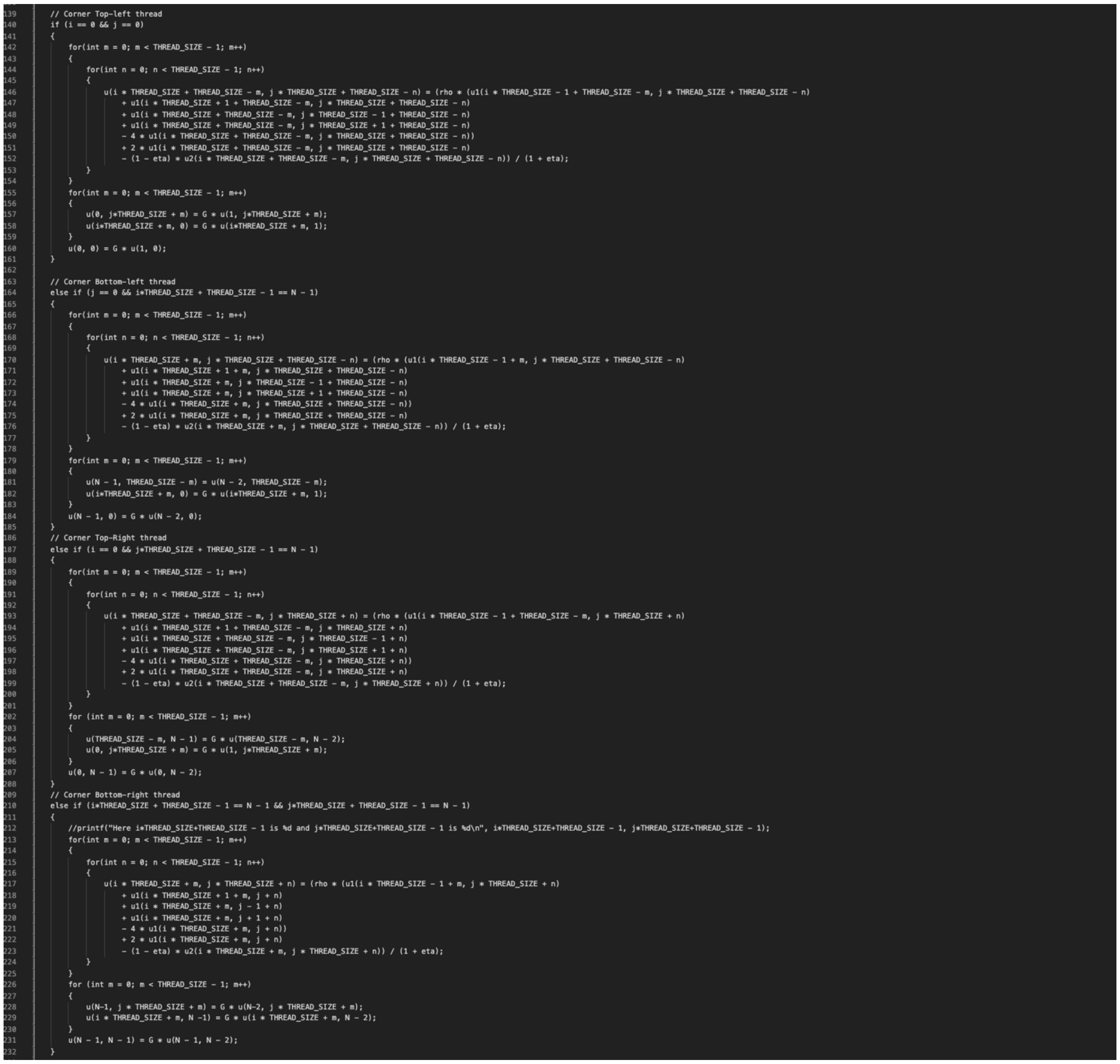


Figure 24: Boundary Thread Visualization (Green is the Boundary element the white square is the Interior Element in the Boundary thread)

The code snippet below shows how we handle the corner elements. Just like the way we did for the boundary case this time, we need to first use a nested 3x3 for loop to deal with the 9 interior elements in the corner thread and then use a for loop to deal with 6 boundary elements and lastly we handle the actually corner element in the 2D array.



```
// Corner Top-left thread
if (i == 0 && j == 0)
{
    for(int m = 0; m < THREAD_SIZE - 1; m++)
    {
        for(int n = 0; n < THREAD_SIZE - 1; n++)
        {
            u(i * THREAD_SIZE + THREAD_SIZE - m, j * THREAD_SIZE + THREAD_SIZE - n) = (rho * (u1(i * THREAD_SIZE - 1 + THREAD_SIZE - m, j * THREAD_SIZE + THREAD_SIZE - n)
                + u1(i * THREAD_SIZE + 1 + THREAD_SIZE - m, j * THREAD_SIZE + THREAD_SIZE - n)
                + u1(i * THREAD_SIZE + THREAD_SIZE - m, j * THREAD_SIZE - 1 + THREAD_SIZE - n)
                + u1(i * THREAD_SIZE + THREAD_SIZE - m, j * THREAD_SIZE + 1 + THREAD_SIZE - n)
                - 4 * u1(i * THREAD_SIZE + THREAD_SIZE - m, j * THREAD_SIZE + THREAD_SIZE - n))
                + 2 * u1(i * THREAD_SIZE + THREAD_SIZE - m, j * THREAD_SIZE + THREAD_SIZE - n)
                - (1 - eta) * u2(i * THREAD_SIZE + THREAD_SIZE - m, j * THREAD_SIZE + THREAD_SIZE - n)) / (1 + eta);
        }
    }
    for(int m = 0; m < THREAD_SIZE - 1; m++)
    {
        u(0, j*THREAD_SIZE + m) = G * u(1, j*THREAD_SIZE + m);
        u(i*THREAD_SIZE + m, 0) = G * u(i*THREAD_SIZE + m, 1);
    }
    u(0, 0) = G * u(1, 0);
}

// Corner Bottom-left thread
else if (j == 0 && i*THREAD_SIZE + THREAD_SIZE - 1 == N - 1)
{
    for(int m = 0; m < THREAD_SIZE - 1; m++)
    {
        for(int n = 0; n < THREAD_SIZE - 1; n++)
        {
            u(i * THREAD_SIZE + m, j * THREAD_SIZE + THREAD_SIZE - n) = (rho * (u1(i * THREAD_SIZE - 1 + m, j * THREAD_SIZE + THREAD_SIZE - n)
                + u1(i * THREAD_SIZE + 1 + m, j * THREAD_SIZE + THREAD_SIZE - n)
                + u1(i * THREAD_SIZE + m, j * THREAD_SIZE - 1 + THREAD_SIZE - n)
                + u1(i * THREAD_SIZE + m, j * THREAD_SIZE + 1 + THREAD_SIZE - n)
                - 4 * u1(i * THREAD_SIZE + m, j * THREAD_SIZE + THREAD_SIZE - n))
                + 2 * u1(i * THREAD_SIZE + m, j * THREAD_SIZE + THREAD_SIZE - n)
                - (1 - eta) * u2(i * THREAD_SIZE + m, j * THREAD_SIZE + THREAD_SIZE - n)) / (1 + eta);
        }
    }
    for(int m = 0; m < THREAD_SIZE - 1; m++)
    {
        u(N - 1, THREAD_SIZE - m) = u(N - 2, THREAD_SIZE - m);
        u(i*THREAD_SIZE + m, 0) = G * u(i*THREAD_SIZE + m, 1);
    }
    u(N - 1, 0) = G * u(N - 2, 0);
}
// Corner Top-Right thread
else if (i == 0 && j*THREAD_SIZE + THREAD_SIZE - 1 == N - 1)
{
    for(int m = 0; m < THREAD_SIZE - 1; m++)
    {
        for(int n = 0; n < THREAD_SIZE - 1; n++)
        {
            u(i * THREAD_SIZE + THREAD_SIZE - m, j * THREAD_SIZE + n) = (rho * (u1(i * THREAD_SIZE - 1 + THREAD_SIZE - m, j * THREAD_SIZE + n)
                + u1(i * THREAD_SIZE + 1 + THREAD_SIZE - m, j * THREAD_SIZE + n)
                + u1(i * THREAD_SIZE + THREAD_SIZE - m, j * THREAD_SIZE - 1 + n)
                + u1(i * THREAD_SIZE + THREAD_SIZE - m, j * THREAD_SIZE + 1 + n)
                - 4 * u1(i * THREAD_SIZE + THREAD_SIZE - m, j * THREAD_SIZE + n))
                + 2 * u1(i * THREAD_SIZE + THREAD_SIZE - m, j * THREAD_SIZE + n)
                - (1 - eta) * u2(i * THREAD_SIZE + THREAD_SIZE - m, j * THREAD_SIZE + n)) / (1 + eta);
        }
    }
    for (int m = 0; m < THREAD_SIZE - 1; m++)
    {
        u(THREAD_SIZE - m, N - 1) = G * u(THREAD_SIZE - m, N - 2);
        u(0, j*THREAD_SIZE + m) = G * u(1, j*THREAD_SIZE + m);
    }
    u(0, N - 1) = G * u(0, N - 2);
}
// Corner Bottom-right thread
else if (i*THREAD_SIZE + THREAD_SIZE - 1 == N - 1 && j*THREAD_SIZE + THREAD_SIZE - 1 == N - 1)
{
    //printf("Here i*THREAD_SIZE+THREAD_SIZE - 1 is %d and j*THREAD_SIZE+THREAD_SIZE - 1 is %d\n", i*THREAD_SIZE+THREAD_SIZE - 1, j*THREAD_SIZE+THREAD_SIZE - 1);
    for(int m = 0; m < THREAD_SIZE - 1; m++)
    {
        for(int n = 0; n < THREAD_SIZE - 1; n++)
        {
            u(i * THREAD_SIZE + m, j * THREAD_SIZE + n) = (rho * (u1(i * THREAD_SIZE - 1 + m, j * THREAD_SIZE + n)
                + u1(i * THREAD_SIZE + 1 + m, j + n)
                + u1(i * THREAD_SIZE + m, j - 1 + n)
                + u1(i * THREAD_SIZE + m, j + 1 + n)
                - 4 * u1(i * THREAD_SIZE + m, j + n))
                + 2 * u1(i * THREAD_SIZE + m, j + n)
                - (1 - eta) * u2(i * THREAD_SIZE + m, j * THREAD_SIZE + n)) / (1 + eta);
        }
    }
    for (int m = 0; m < THREAD_SIZE - 1; m++)
    {
        u(N-1, j * THREAD_SIZE + m) = G * u(N-2, j * THREAD_SIZE + m);
        u(i * THREAD_SIZE + m, N -1) = G * u(i * THREAD_SIZE + m, N - 2);
    }
    u(N - 1, N - 1) = G * u(N - 1, N - 2);
}
```

Figure 25: Code Snippet for Processing Corner thread

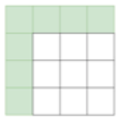
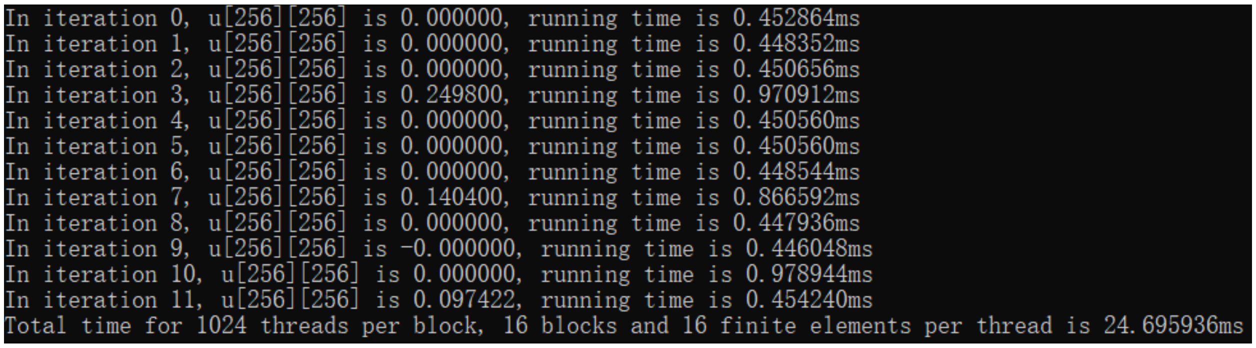


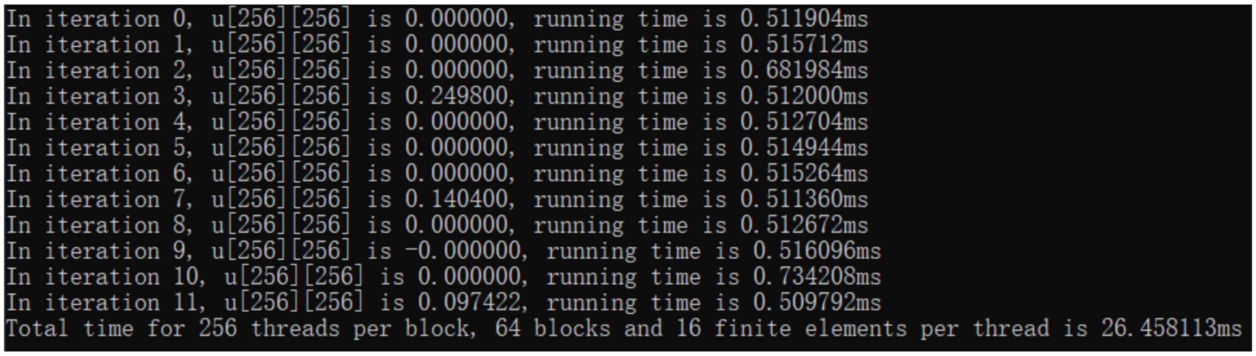
Figure 26: Visualization for Corner Thread (The green squares are boundary elements and the corner element. The white squares are the Interior Elements in the Corner thread.)

Experiment 1: 1024 threads per block, 16 blocks, 16 elements per thread.



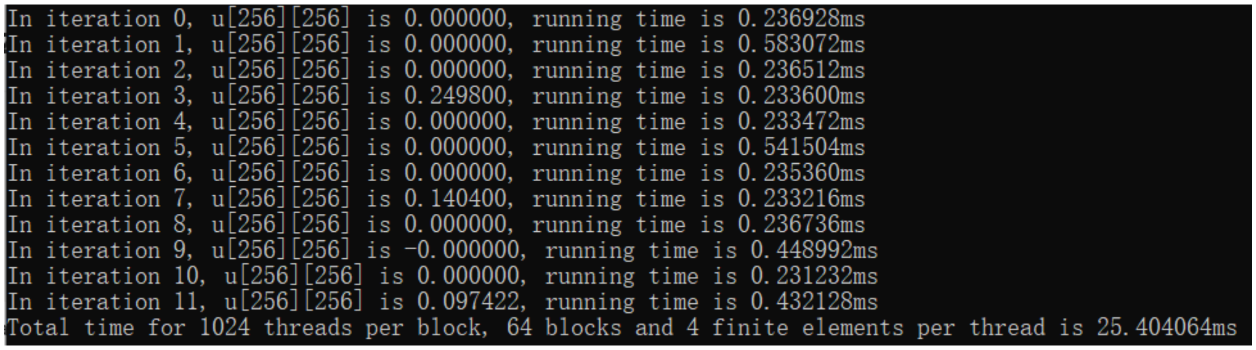
```
In iteration 0, u[256][256] is 0.000000, running time is 0.452864ms
In iteration 1, u[256][256] is 0.000000, running time is 0.448352ms
In iteration 2, u[256][256] is 0.000000, running time is 0.450656ms
In iteration 3, u[256][256] is 0.249800, running time is 0.970912ms
In iteration 4, u[256][256] is 0.000000, running time is 0.450560ms
In iteration 5, u[256][256] is 0.000000, running time is 0.450560ms
In iteration 6, u[256][256] is 0.000000, running time is 0.448544ms
In iteration 7, u[256][256] is 0.140400, running time is 0.866592ms
In iteration 8, u[256][256] is 0.000000, running time is 0.447936ms
In iteration 9, u[256][256] is -0.000000, running time is 0.446048ms
In iteration 10, u[256][256] is 0.000000, running time is 0.978944ms
In iteration 11, u[256][256] is 0.097422, running time is 0.454240ms
Total time for 1024 threads per block, 16 blocks and 16 finite elements per thread is 24.695936ms
```

Experiment 2: 256 threads per block, 64 blocks and 16 finite elements per thread.



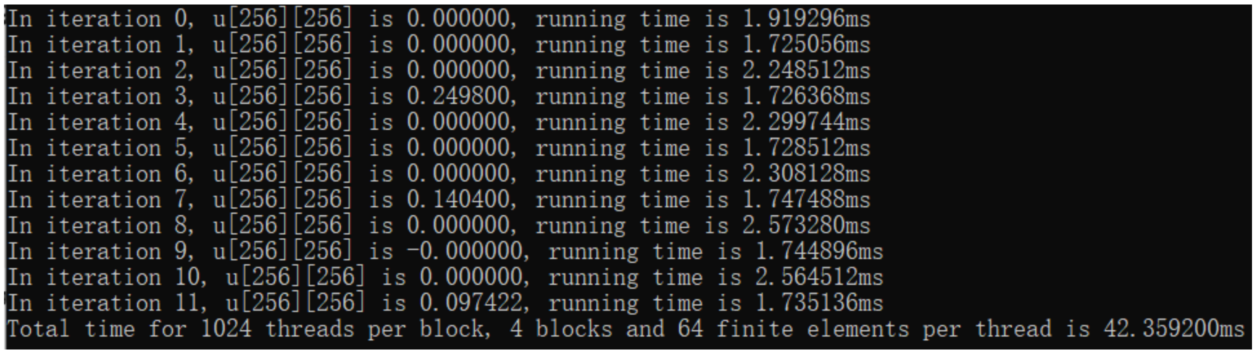
```
In iteration 0, u[256][256] is 0.000000, running time is 0.511904ms
In iteration 1, u[256][256] is 0.000000, running time is 0.515712ms
In iteration 2, u[256][256] is 0.000000, running time is 0.681984ms
In iteration 3, u[256][256] is 0.249800, running time is 0.512000ms
In iteration 4, u[256][256] is 0.000000, running time is 0.512704ms
In iteration 5, u[256][256] is 0.000000, running time is 0.514944ms
In iteration 6, u[256][256] is 0.000000, running time is 0.515264ms
In iteration 7, u[256][256] is 0.140400, running time is 0.511360ms
In iteration 8, u[256][256] is 0.000000, running time is 0.512672ms
In iteration 9, u[256][256] is -0.000000, running time is 0.516096ms
In iteration 10, u[256][256] is 0.000000, running time is 0.734208ms
In iteration 11, u[256][256] is 0.097422, running time is 0.509792ms
Total time for 256 threads per block, 64 blocks and 16 finite elements per thread is 26.458113ms
```

Experiment 3: 1024 threads per block, 64 blocks and 4 finite elements



```
In iteration 0, u[256][256] is 0.000000, running time is 0.236928ms
In iteration 1, u[256][256] is 0.000000, running time is 0.583072ms
In iteration 2, u[256][256] is 0.000000, running time is 0.236512ms
In iteration 3, u[256][256] is 0.249800, running time is 0.233600ms
In iteration 4, u[256][256] is 0.000000, running time is 0.233472ms
In iteration 5, u[256][256] is 0.000000, running time is 0.541504ms
In iteration 6, u[256][256] is 0.000000, running time is 0.235360ms
In iteration 7, u[256][256] is 0.140400, running time is 0.233216ms
In iteration 8, u[256][256] is 0.000000, running time is 0.236736ms
In iteration 9, u[256][256] is -0.000000, running time is 0.448992ms
In iteration 10, u[256][256] is 0.000000, running time is 0.231232ms
In iteration 11, u[256][256] is 0.097422, running time is 0.432128ms
Total time for 1024 threads per block, 64 blocks and 4 finite elements per thread is 25.404064ms
```

Experiment 4: 1024 threads per block, 4 blocks and 64 finite elements



```
In iteration 0, u[256][256] is 0.000000, running time is 1.919296ms
In iteration 1, u[256][256] is 0.000000, running time is 1.725056ms
In iteration 2, u[256][256] is 0.000000, running time is 2.248512ms
In iteration 3, u[256][256] is 0.249800, running time is 1.726368ms
In iteration 4, u[256][256] is 0.000000, running time is 2.299744ms
In iteration 5, u[256][256] is 0.000000, running time is 1.728512ms
In iteration 6, u[256][256] is 0.000000, running time is 2.308128ms
In iteration 7, u[256][256] is 0.140400, running time is 1.747488ms
In iteration 8, u[256][256] is 0.000000, running time is 2.573280ms
In iteration 9, u[256][256] is -0.000000, running time is 1.744896ms
In iteration 10, u[256][256] is 0.000000, running time is 2.564512ms
In iteration 11, u[256][256] is 0.097422, running time is 1.735136ms
Total time for 1024 threads per block, 4 blocks and 64 finite elements per thread is 42.359200ms
```

Experiment Num	Total time used	Max Iteration Time	Min Iteration Time	Average Iteration Time	Total Time used by GPU	Total Time used by CPU
1	24.6	0.97	0.44	0.571	6.859	17.75
2	26.4	0.73	0.50	0.545	6.542	19.85
3	25.4	0.58	0.23	0.323	3.877	21.5
4	42.4	2.57	1.73	2.015	24.289	18.11

Table 1: The results of 4 experiment

We did 4 experiments showing in the figures and the results are shown in the table above. We can see that if we are handling the same number of elements per thread, the value would not change too much. If we increase the threads number and decrease the finite elements per thread, we can see that average time using of each iteration is decreasing very much. If we keep number of threads per block the same and increase finite elements per thread, we can see that the run time increases a lot.

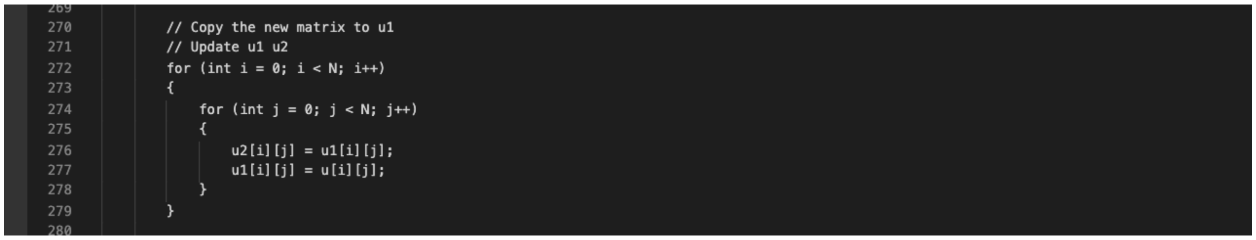
For the average time that each iteration using, we can see that for experiment 4 the average time is the max of 4 values which is 2.015 and the experiment 3 is the minimum time used of all 4 experiments. The average time used for experiment 1 and 2 are almost the same.

The reason is because the way we handle the finite elements are sequential in the GPU.

In the first experiment, we are using 16 blocks each block has 1024 thread and each thread is handling 16 finite elements. There are 1024x16 = 16384 threads in total. In the second experiment we are using 64 blocks each block has 256 threads and each thread would handle 16 finite elements. There are 64x256 = 16384 threads in total which is the same as the first experiment. The average times for these two experiments are almost the same and that is because the parallel part is almost the same for these two cases and both of these cases are dealing with 16 finite elements which is the sequential part in the GPU.

In the 4th experiment, we are using 4 blocks each block has 1024 thread and each thread is handling 64 finite elements. The sequential part of the GPU computation. If we use the average time used by experiment 4 divide by the average time used by experiment 1 then we can see that $\frac{Avg.Time\ Exp\ 4}{Avg.Time\ Exp\ 1} = \frac{2.015}{0.571} \approx 3.703$ and if we use the finite elements handling per thread in experiment 1 divide by the finite elements handling per thread in experiment 4 then we can get $\frac{Elements\ Per\ Thread\ Exp\ 4}{Elements\ Per\ Thread\ Exp\ 1} = \frac{64}{16} = 4$. We can easily see that the speed of computing for each iteration is related to the elements that each iteration is handling.

We can also see that for experiment 1-3 the total time required by the computation is almost the same (24.6ms, 26.4ms, 25.4ms) and this is because after each iteration we need to copy the content from array u to array u1 in order to update array u1. This is the sequential part and it is a 512 loop. This would take the major part of the total time used by this program. As well as the memcopy for copying the elements from CPU to device. If we look at the total time used by CPU they are about the same level between 18-21ms and these times are indicating that the sequential part is actually the overhead and memory copy is actually the most time consuming task in this program.



```
        // Copy the new matrix to u1
        // Update u1 u2
        for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < N; j++)
            {
                u2[i][j] = u1[i][j];
                u1[i][j] = u[i][j];
            }
        }
```

Figure 27: The code snippet for updating U1 and U2