ECSE 420: Parallel Computing

Lab 2: CUDA Convolution and Musical Instrument Simulation

In this lab, you will write code for a simple signal processing and for a musical instrument simulation, parallelize it using CUDA, and write a report summarizing your experimental results. We will use PNG images as the test signal for the first part (convolution).

A. Convolution: (30 points)

Convolution is a slightly more complicated operation than rectification and pooling from Lab 0, but it is still highly parallelizable. For each pixel in the input image, a 3x3 convolution computes the corresponding pixel in the output image using a weighted sum of the input pixel and its neighbors. That is:

$$output[i][j] = \sum_{ii=0}^{2}\sum_{jj=0}^{2} input[i+ii-1][j+jj-1]w[ii][jj],$$

$$for\ \ 1 \le i \le m-1, 1 \le j \le n-1$$

where *m* is the number of rows in the input image, and *n* is the number of columns in the input image. Since we are using square weight matrices and the "valid padding" definition of convolution (*1 ≤ i ≤ m – 1, 1 ≤ i ≤ n – 1*), the output image will be of size *m – 2* by *n – 2*. Figure 1 illustrates the convolution operation for a certain weight matrix, *W*. (**Note:** *You don't have to manipulate the Alpha channel*.)

Write the code for convolution. For this lab**, you only need to implement convolution using 3x3 weight matrices for the 3 test images provided**. Before you convert the output to unsigned chars and save the file, you should clamp the output of the convolution between 0 and 255 (i.e., if a value in the output is less than 0, you should set it equal to 0, and if a value is greater than 255, you should set it equal to 255). The header file "wm.h" (on myCourses) contains the weight matrix you should use. **(15 points)**

Please provide the 3 output images after convolution for the given 3 test images. **(15 points)**

The grader will run the following command:

./convolve <name of input png> <name of output png> < # threads>

When the input test image is "Test_1.png", the output of your code should be identical to "Test_1_convolve.png". These are available on myCourses. You can use the "test_equality.c" code provided in Lab 0 to check if two images are identical. The grader should be able to run your code with different numbers of threads, and the output of your code should be correct for any of those thread counts.

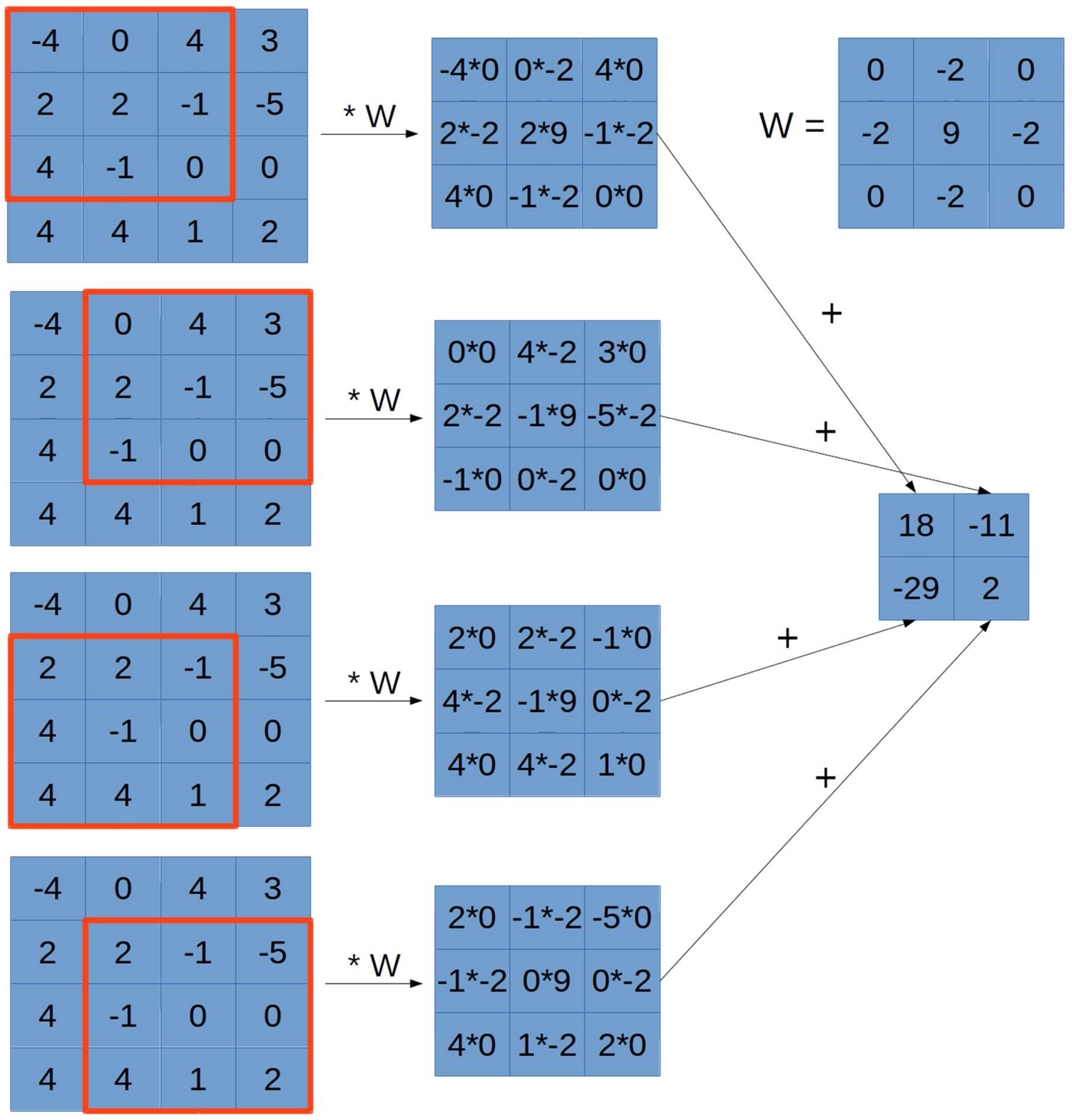


Figure 1: Convolution Example

Report Deliverables: (20 points)

- Measure the runtime when the number of **threads** used is equal to {1, 4, 8, 16, 64, 128, 256, 512, 1024}.
- Plot the speedup as a function of the number of threads for each of the 3 test images provided. **(10 points)**
- Discuss your parallelization scheme and your speedup plots and make reference to the architecture on which you run your experiments. **(10 points)**

B. Finite Element Music Synthesis (30 points)

In a “finite element” model, a complex physical object is modelled as a collection of simple objects (finite elements) that interact with each other and behave according to physical laws. It is possible to simulate electromagnetic fields, calculate stresses in the structure of a building, or synthesize the sounds of a musical instrument using finite elements. In this part of the lab, you will synthesize drum sounds using a two-dimensional grid of finite elements.

The finite element method we will be using is like the ocean simulator examined in the class. In this method, each **interior element** performs the following update at every iteration:

$$u(i,j) = \frac{\rho[u1(i-1,j) + u1(i+1,j) + u1(i,j-1) + u1(i,j+1) - 4u1(i,j)] + 2u1(i,j) - (1-\eta)u2(i,j)}{1+\eta},$$

$$for\ 1 \le i \le N-2, 1 \le j \le N-2$$

where *u* is the displacement of the element at position *(i, j), u1* is the displacement of that element at the previous time step, *u2* is the displacement of that element at the previous previous time step, and *η (0.0002)* and *ρ (0.5)* are constants related to the size of the drum, sampling rate, damping, etc.

Then, we ensure that the **boundary conditions** are met by updating the side elements:

$$u(0,i) := Gu(1,i)$$

$$u(N-1,i) := Gu(N-2,i)$$

$$u(i,0) := Gu(i,1)$$

$$u(i,N-1) := Gu(i,N-2)$$

$$1 \le i \le N-2$$

and then the **corner elements**:

$$u(0,0) := Gu(1,0)$$

$$u(N-1,0) := Gu(N-2,0)$$

$$u(0,N-1) := Gu(0,N-2)$$

$$u(N-1,N-1) := Gu(N-1,N-2)$$

where *G (0.75)* is the boundary gain.

Finally, at the end of the iteration, we set *u2* equal to *u1* and *u1* equal to *u*.

To simulate a hit on the drum, simply add 1 to *u1* at some position (*only once before the first iteration*). To record the sound of the drum, collect the value of *u* at some position for each iteration in an array of length *N*, where *T* is the number of iterations to perform. For this lab, you should use *(N/2, N/2)* as the position on the drum for both the hit and the recording.

1. Write code that implements a *4 by 4* finite element grid **sequentially**. The user will provide a single command line argument, *T*, which specifies the number of iterations to run the simulation. Your code should print *u(N/2, N/2)* (which in this case is *u(2, 2)*) to the terminal at each iteration. **(5 points)**

2. Parallelize your implementation in **Part 1** by assigning **each node** of this grid to a single **thread** using CUDA. The user will provide a single command line argument, T, which specifies the number of iterations to run the simulation. Your code should print *u(N/2, N/2)* (which in this case is *u(2, 2)*) to the terminal at each iteration. (The output must match the output from Part 1, otherwise please debug). **(10 points)**

   The grader will run the following commands:

   ```
   ./grid_4_4 <number of iterations>
   ```

3. Using one **thread** per finite element can require an enormous amount of communication and GPU overhead, and even in large clusters there may not be enough hardware available to fully parallelize the computation. It is usually better to use a decomposition that assigns multiple elements to each processor.

   Parallelize your code using such a decomposition (with rows, columns, blocks, etc.) and simulate a *512 by 512* grid. Try to parallelize with **at least 3** *different combinations* of **threads, blocks and finite elements per thread**. Ex – 1024 threads/block and 16 blocks can allow each thread to handle 16 finite elements (Total elements = 512x512 = 1024x16x16). **(15 points)**

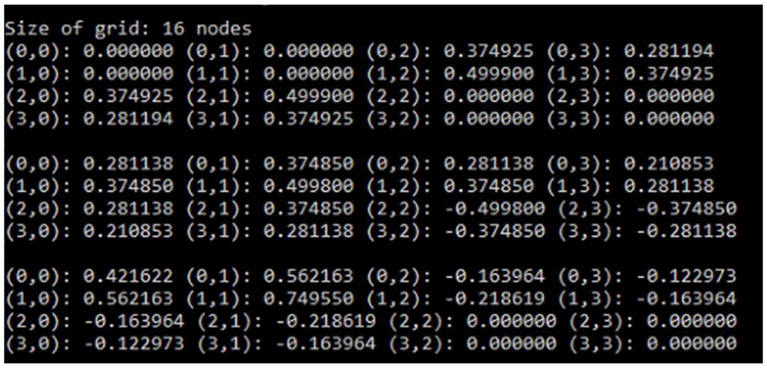
   The grader will run the following commands:

   ```
   ./grid_512_512 <number of iterations>
   ```

Report Deliverables: (20 points)

- Provide your printed outputs for each part (1, 2 & 3). **(5 points)**
- Provide a table comparing the execution times for each combination of threads, blocks and finite elements per thread. **(5 points)**
- Discuss and analyze your parallelization schemes and experimental results. **(10 points)**

To help debug your code, Figure 2: u for first 3 iterationsFigure 2 and Figure 3 shows the entire *u* grid (4x4) for the first three iterations:

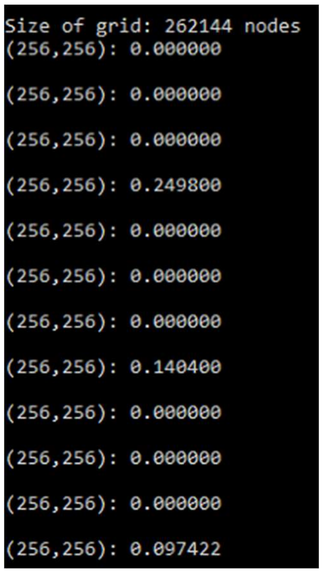


```
Size of grid: 16 nodes
(0,0): 0.000000 (0,1): 0.000000 (0,2): 0.374925 (0,3): 0.281194
(1,0): 0.000000 (1,1): 0.000000 (1,2): 0.499900 (1,3): 0.374925
(2,0): 0.374925 (2,1): 0.499900 (2,2): 0.000000 (2,3): 0.000000
(3,0): 0.281194 (3,1): 0.374925 (3,2): 0.000000 (3,3): 0.000000

(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
```

Figure 2: u for first 3 iterations for 4x4 grid



```
Size of grid: 262144 nodes
(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
```

Figure 3: u(N/2, N/2) for the first 12 iterations for 512x512 grid

Submission Instructions:

Each group should submit a single zip file with the filename *Group_<your group number> _Lab_2.zip*. (Ex – Group_09_Lab_2.zip). The zip file should contain the following:

1. **A lab report** answering all the questions in the lab (please use a PDF format). **(40 points)**
   a. Must be named *Group_<your group number>_Lab_2_Report.pdf*. (Ex – Group_03_Lab_2_Report.pdf).
   b. Must have a cover page (Include Group number, members' names and ID).
   c. Must follow the logical order for the lab discussions and provide all the items asked for as mentioned above.
2. **Your own source** code. Add the whole project solution. **(45 points)**
3. The **output test images** (raw .png) for the grader to validate. **(15 points)**

Late Submission Penalty:

- 5% per day of late submission.
- Lab will be accepted up to 1 week after the deadline (14% max penalty).