

ECSE 420 – Parallel Computing – Fall 2020

Lab 2 – CUDA Convolution and Musical Instrument Simulation

Aleksas Murauskas 260718389

Anne-Julie Côté 260803768

Group 16

A. Convolution

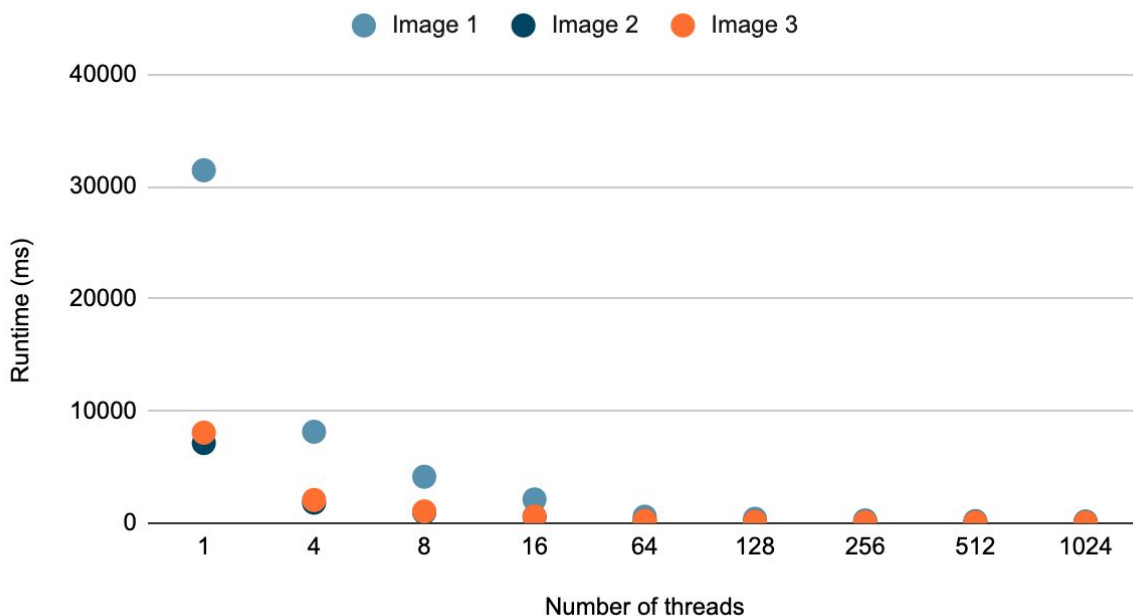
- Measure the runtime when the number of **threads** used is equal to $\{1, 4, 8, 16, 64, 128, 256, 512, 1024\}$.

Below are the runtimes of the parallelization portion of our program on each of the given images. The runtime of the threads

Table 1: Runtime (ms) per number of threads

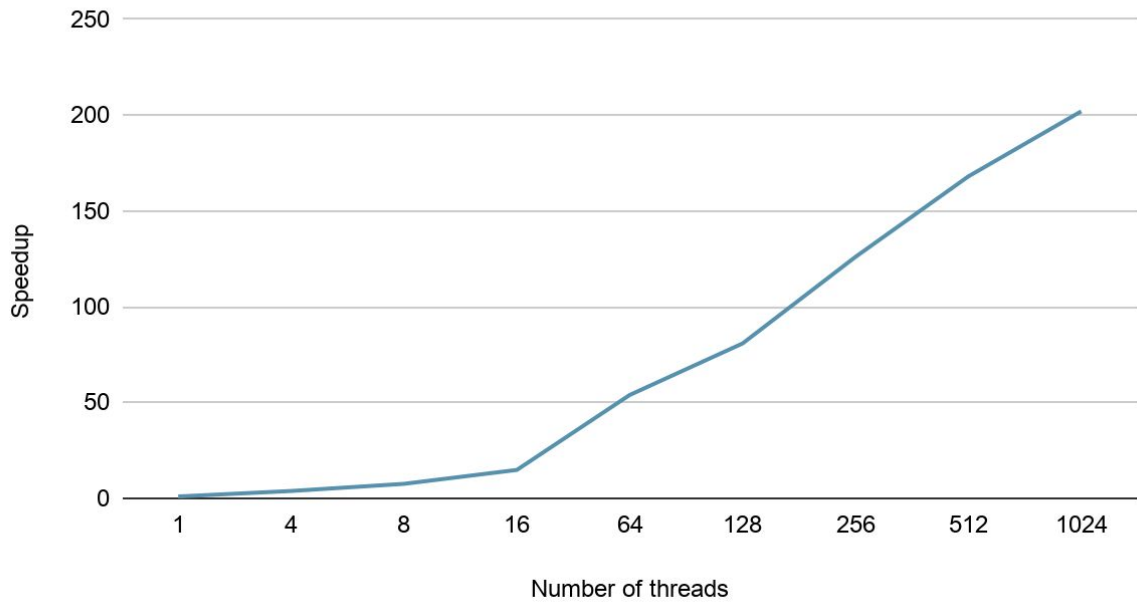
	1	4	8	16	64	128	256	512	1024
Image 1	31457.005859	8148.497070	4141.040527	2117.255371	582.785034	388.962311	249.59167	187.198212	155.717285
Image 2	7146.717285	1825.737183	966.260803	570.175354	202.600479	110.837341	70.274178	47.958782	35.557343
Image 3	8070.236328	2065.504150	1051.275391	601.255920	215.810822	123.846909	77.815773	54.480511	39.366657

Graph 1: Runtime over number of threads

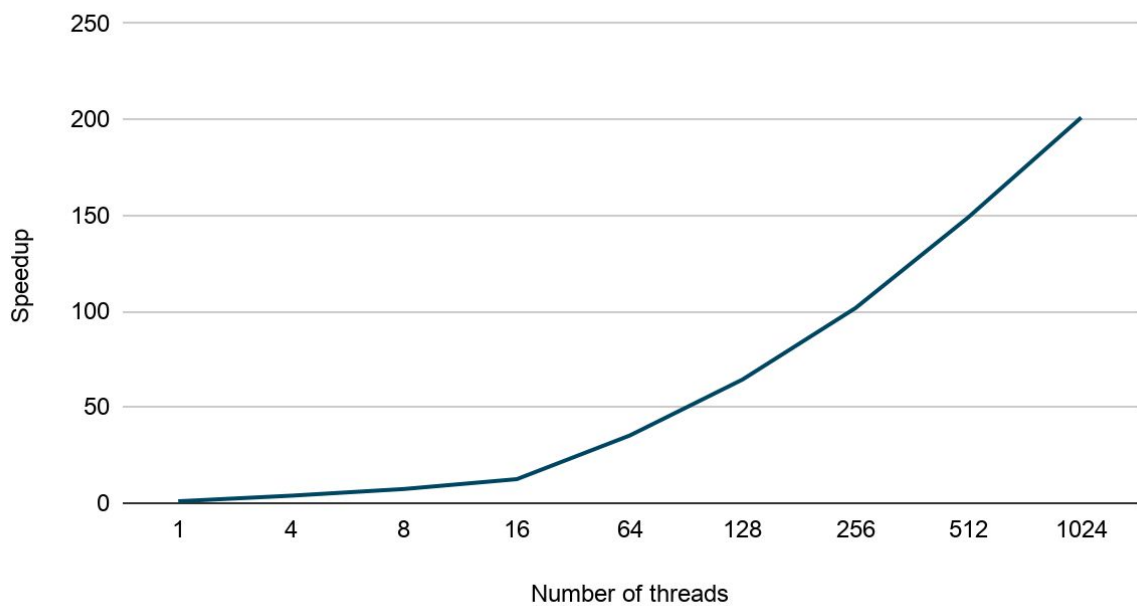


- Plot the speedup as a function of the number of threads for each of the 3 test images provided. (10 points)

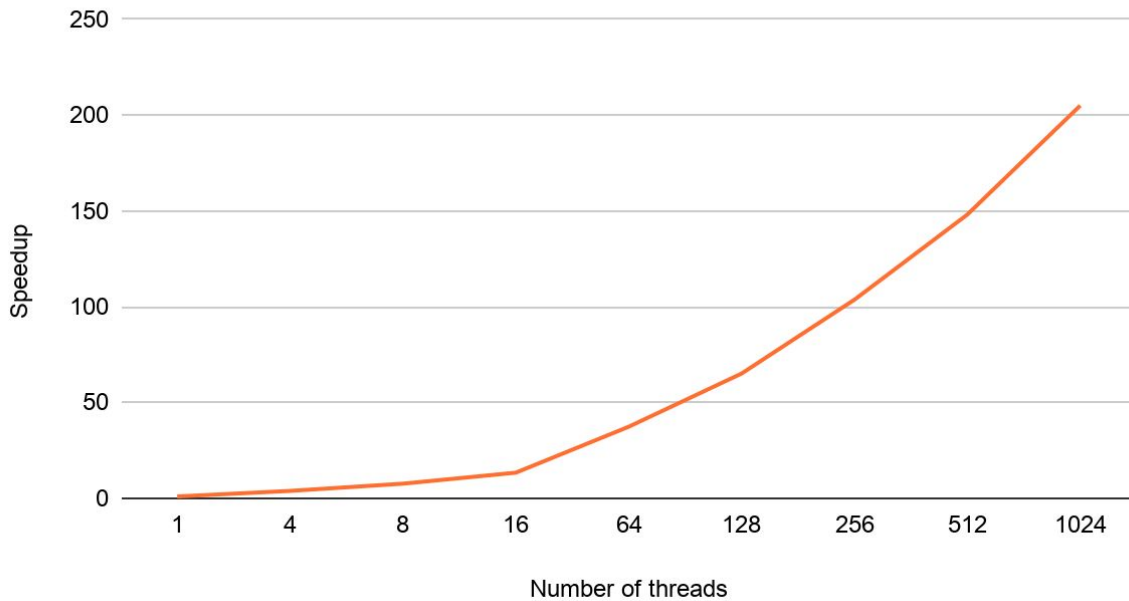
Graph 2: Speedup over number of threads for image 1



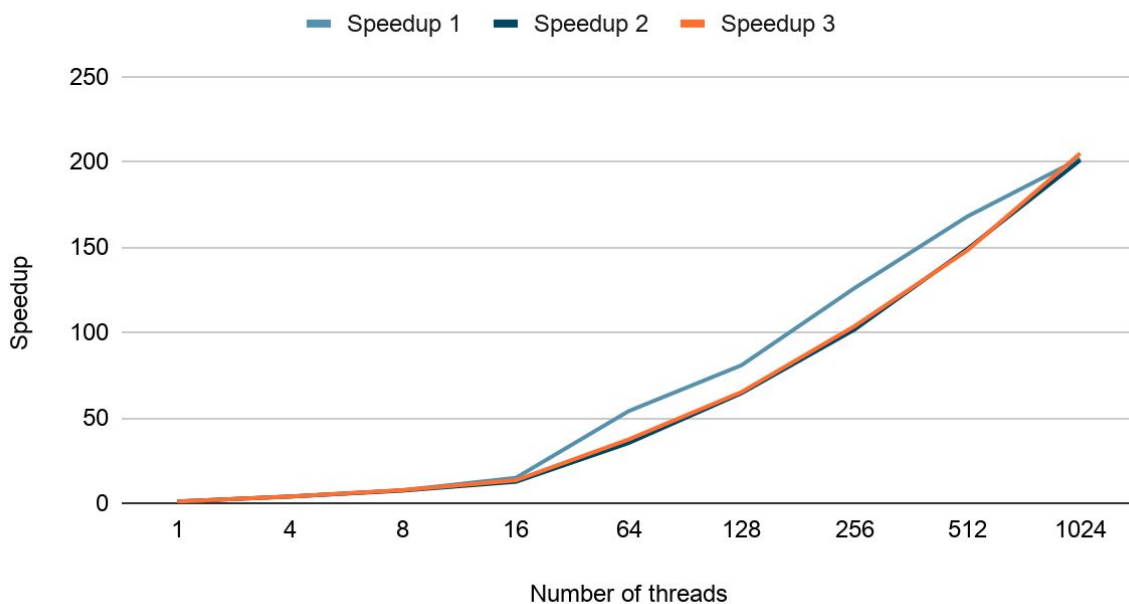
Graph 3: Speedup over number of threads for image 2



Graph 4: Speedup over number of threads for image 3



Graph 5: Speedup over number of threads



- *Discuss your parallelization scheme and your speedup plots and make reference to the architecture on which you run your experiments. (10 points)*

Our programs were developed on Google Colab. The parallel portion of our program is entirely within the method `convolve()`. Convolve requires several fields of input: a pointer to the input and output

images's locations in memory, the height and width of the input image, an array of weights for use in convolution, and the number of threads and a cycle value. When Convolve() is called the thread's index value is first found by multiplying the thread's blockId by the dimensions of the block, plus an offset determined by the threadId. As long as the index is not larger than the number of threads per block, a for loop begins to convolve all of the RGBA values. The cycle value and number of threads are used in order to find the offset in memory that thread should act upon. Within this Loop, another loop occurs to iterate through the values that will be convoluted, as it is a 3x3 matrix, this loop iterates 9 times. A temporary matrix holds the channel values from the input at a location in input memory by offsetting using the offset, width, channel number, and temporary array index.

input offset = given offset +width*(temp index/3)+(temp index-3*(temp index/j)*4+channel number;

Once that value is found, temp is updated by multiplying its current value by the weight array's value at the same index. Afterward, the value in the temp array is added to a sum value which will be held in. At the end of each channel's loop, the value is checked to see if it is within the correct range of pixel values (0,255), if out of these bounds, the sum is set to the closer of the two values. Lastly this sum is stored in the output.

Convolve is called within a for loop that iterates until it has iterated more times than the total number of pixels divided by the number of threads per block. The iterations are tracked with a variable cycle that iterates each loop. it is used to both terminate the loop and pass the current cycle number to the threads to assist in finding the location in image memory they will act upon.

Our parallelization scheme seems to be efficient as shown in graph 5, as regardless of image file size, the speedup of inputs are the same. Image 1 was significantly larger than inputs 2 and 3, yet it's speed up equation follows the same slope as the final two inputs, showing our parallelization is efficient in regard to image size.

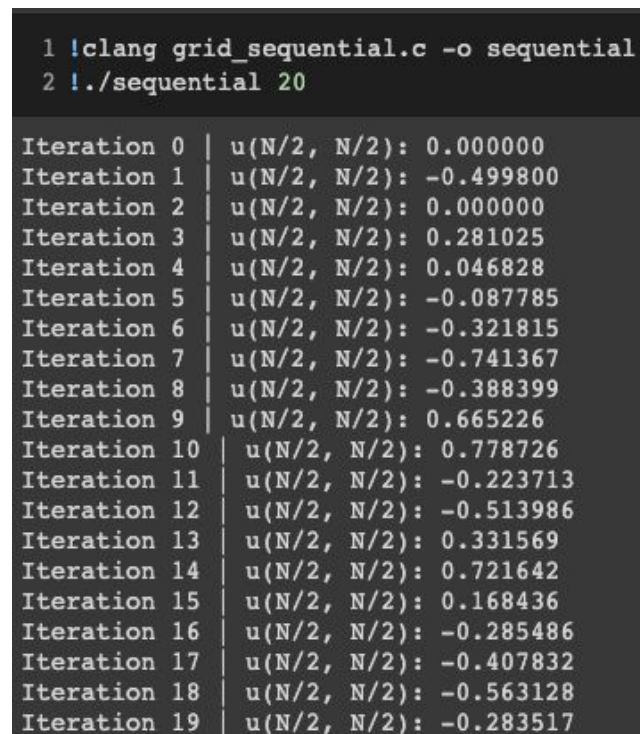
B. Finite Element Music Synthesis

- Provide your printed outputs for each part (1, 2 & 3).

Part 1:

In part 1, we performed music synthesis sequentially. The following output was printed to the terminal.

```
Iteration 0 | u(N/2, N/2): 0.000000
Iteration 1 | u(N/2, N/2): -0.499800
Iteration 2 | u(N/2, N/2): 0.000000
Iteration 3 | u(N/2, N/2): 0.281025
Iteration 4 | u(N/2, N/2): 0.046828
Iteration 5 | u(N/2, N/2): -0.087785
Iteration 6 | u(N/2, N/2): -0.321815
Iteration 7 | u(N/2, N/2): -0.741367
Iteration 8 | u(N/2, N/2): -0.388399
Iteration 9 | u(N/2, N/2): 0.665226
Iteration 10 | u(N/2, N/2): 0.778726
Iteration 11 | u(N/2, N/2): -0.223713
Iteration 12 | u(N/2, N/2): -0.513986
Iteration 13 | u(N/2, N/2): 0.331569
Iteration 14 | u(N/2, N/2): 0.721642
Iteration 15 | u(N/2, N/2): 0.168436
Iteration 16 | u(N/2, N/2): -0.285486
Iteration 17 | u(N/2, N/2): -0.407832
Iteration 18 | u(N/2, N/2): -0.563128
Iteration 19 | u(N/2, N/2): -0.283517
```

A screenshot of a terminal window showing the compilation and execution of a program. The first two lines are commands: '1 clang grid_sequential.c -o sequential' and '2 !./sequential 20'. The subsequent lines show the program's output, which is a list of 20 iterations, each with a value for u(N/2, N/2). The values are: 0.000000, -0.499800, 0.000000, 0.281025, 0.046828, -0.087785, -0.321815, -0.741367, -0.388399, 0.665226, 0.778726, -0.223713, -0.513986, 0.331569, 0.721642, 0.168436, -0.285486, -0.407832, -0.563128, and -0.283517.

```
1 clang grid_sequential.c -o sequential
2 !./sequential 20

Iteration 0 | u(N/2, N/2): 0.000000
Iteration 1 | u(N/2, N/2): -0.499800
Iteration 2 | u(N/2, N/2): 0.000000
Iteration 3 | u(N/2, N/2): 0.281025
Iteration 4 | u(N/2, N/2): 0.046828
Iteration 5 | u(N/2, N/2): -0.087785
Iteration 6 | u(N/2, N/2): -0.321815
Iteration 7 | u(N/2, N/2): -0.741367
Iteration 8 | u(N/2, N/2): -0.388399
Iteration 9 | u(N/2, N/2): 0.665226
Iteration 10 | u(N/2, N/2): 0.778726
Iteration 11 | u(N/2, N/2): -0.223713
Iteration 12 | u(N/2, N/2): -0.513986
Iteration 13 | u(N/2, N/2): 0.331569
Iteration 14 | u(N/2, N/2): 0.721642
Iteration 15 | u(N/2, N/2): 0.168436
Iteration 16 | u(N/2, N/2): -0.285486
Iteration 17 | u(N/2, N/2): -0.407832
Iteration 18 | u(N/2, N/2): -0.563128
Iteration 19 | u(N/2, N/2): -0.283517
```

Figure 1: Output for part 1

Part 2:

In part 2, the music synthesis was parallelized using an approach of one thread per block.

```
Iteration 0 | u(N/2, N/2): 0.000000
Iteration 1 | u(N/2, N/2): -0.499800
Iteration 2 | u(N/2, N/2): 0.000000
Iteration 3 | u(N/2, N/2): 0.281025
Iteration 4 | u(N/2, N/2): 0.046828
Iteration 5 | u(N/2, N/2): -0.087785
Iteration 6 | u(N/2, N/2): -0.321815
Iteration 7 | u(N/2, N/2): -0.741367
Iteration 8 | u(N/2, N/2): -0.388399
Iteration 9 | u(N/2, N/2): 0.665225
Iteration 10 | u(N/2, N/2): 0.778726
Iteration 11 | u(N/2, N/2): -0.223713
Iteration 12 | u(N/2, N/2): -0.513986
Iteration 13 | u(N/2, N/2): 0.331569
Iteration 14 | u(N/2, N/2): 0.721642
Iteration 15 | u(N/2, N/2): 0.168436
Iteration 16 | u(N/2, N/2): -0.285486
Iteration 17 | u(N/2, N/2): -0.407832
Iteration 18 | u(N/2, N/2): -0.563128
Iteration 19 | u(N/2, N/2): -0.283517
```

```
1 !nvcc grid_parallel.cu -o grid_4_4
2 !./grid_4_4 20

Iteration 0 | u(N/2, N/2): 0.000000
Iteration 1 | u(N/2, N/2): -0.499800
Iteration 2 | u(N/2, N/2): 0.000000
Iteration 3 | u(N/2, N/2): 0.281025
Iteration 4 | u(N/2, N/2): 0.046828
Iteration 5 | u(N/2, N/2): -0.087785
Iteration 6 | u(N/2, N/2): -0.321815
Iteration 7 | u(N/2, N/2): -0.741367
Iteration 8 | u(N/2, N/2): -0.388399
Iteration 9 | u(N/2, N/2): 0.665225
Iteration 10 | u(N/2, N/2): 0.778726
Iteration 11 | u(N/2, N/2): -0.223713
Iteration 12 | u(N/2, N/2): -0.513986
Iteration 13 | u(N/2, N/2): 0.331569
Iteration 14 | u(N/2, N/2): 0.721642
Iteration 15 | u(N/2, N/2): 0.168436
Iteration 16 | u(N/2, N/2): -0.285486
Iteration 17 | u(N/2, N/2): -0.407832
Iteration 18 | u(N/2, N/2): -0.563128
Iteration 19 | u(N/2, N/2): -0.283517
```

Figure 2: Output for part 2

Part 3:

In part 3, we implemented a parallelization of a 512 by 512 finite element grid, with varying numbers of blocks and threads per block. The output was the same no matter the number of threads and blocks.

```
Iteration 0 | u(N/2, N/2): 0.000000
Iteration 1 | u(N/2, N/2): 0.000000
Iteration 2 | u(N/2, N/2): 0.000000
Iteration 3 | u(N/2, N/2): 0.249800
Iteration 4 | u(N/2, N/2): 0.000000
Iteration 5 | u(N/2, N/2): 0.000000
Iteration 6 | u(N/2, N/2): 0.000000
Iteration 7 | u(N/2, N/2): 0.140400
Iteration 8 | u(N/2, N/2): 0.000000
Iteration 9 | u(N/2, N/2): 0.000000
Iteration 10 | u(N/2, N/2): 0.000000
Iteration 11 | u(N/2, N/2): 0.097422
Iteration 12 | u(N/2, N/2): 0.000000
Iteration 13 | u(N/2, N/2): 0.000000
Iteration 14 | u(N/2, N/2): 0.000000
Iteration 15 | u(N/2, N/2): 0.074529
Iteration 16 | u(N/2, N/2): 0.000000
Iteration 17 | u(N/2, N/2): 0.000000
Iteration 18 | u(N/2, N/2): 0.000000
Iteration 19 | u(N/2, N/2): 0.060321
```

```
1 !nvcc parallel_decomp.cu -o grid_512_512
2 !./grid_512_512 20

Iteration 0 | u(N/2, N/2): 0.000000
Iteration 1 | u(N/2, N/2): 0.000000
Iteration 2 | u(N/2, N/2): 0.000000
Iteration 3 | u(N/2, N/2): 0.249800
Iteration 4 | u(N/2, N/2): 0.000000
Iteration 5 | u(N/2, N/2): 0.000000
Iteration 6 | u(N/2, N/2): 0.000000
Iteration 7 | u(N/2, N/2): 0.140400
Iteration 8 | u(N/2, N/2): 0.000000
Iteration 9 | u(N/2, N/2): 0.000000
Iteration 10 | u(N/2, N/2): 0.000000
Iteration 11 | u(N/2, N/2): 0.097422
Iteration 12 | u(N/2, N/2): 0.000000
Iteration 13 | u(N/2, N/2): 0.000000
Iteration 14 | u(N/2, N/2): 0.000000
Iteration 15 | u(N/2, N/2): 0.074529
Iteration 16 | u(N/2, N/2): 0.000000
Iteration 17 | u(N/2, N/2): 0.000000
Iteration 18 | u(N/2, N/2): 0.000000
Iteration 19 | u(N/2, N/2): 0.060321
```

Figure 3: Output for part 3

- Provide a table comparing the execution times for each combination of threads, blocks and finite elements per thread. (5 points)

Table 2: Execution time for 20 iterations with varying number of threads and blocks

Number of blocks	Number of threads per block	Finite elements per thread	Execution time (ms)
8	1024	32	74.897888
16	16	1024	127.313309
32	1024	8	66.670464
32	64	128	78.417885
64	32	128	79.470558
128	1024	2	61.571487

- Discuss and analyze your parallelization schemes and experimental results. (10 points)

To parallelize the simulation of the hit of a drum, we started with an approach of 1 thread per block for a 4 by 4 grid (part 2). In this part, we used unified memory in CUDA. We implemented three different methods to deal with the different cases: the corners, the borders or the interior of the grid. The execution time for this part was very small (around 9 ms), because the grid is very small.

For part 3, we had to simulate a 512 by 512 grid with a varying number of threads per block to optimize (i.e. minimize) execution time. We changed the approach for this part and we implemented only one kernel function instead of three, in order to minimize overhead and not waste computing power on dividing the number of threads and blocks between different cases on the CPU before calling the device. We used unified memory with CUDA again, and we generalized the different cases (corner, side or interior) to make the function more efficient. From testing the results and using different parameters for parallelization, we think that our parallelization scheme is efficient.

We tried different combinations of number of blocks, threads per block and finite elements per thread and calculated execution time for each, as we can see in Table 2 above. We draw the conclusion that in general, the less finite elements need to be calculated by each thread, the lower execution time will be. The execution with 8 blocks, 16 threads per block and 1024 finite elements per thread took over twice as long as the execution with 128 blocks, 1024 threads per block and 2 finite elements per thread (127.3 ms vs 61.57 ms). It is better to assign more elements to each block in order to minimize the communication overhead, than to have a few blocks each running a few threads, but each thread executing several elements.