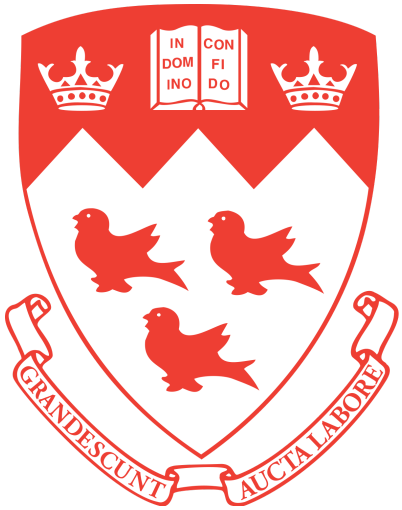
ECSE 420 - Parallel Computing
Fall 2020
Professor Zeljko Zilic

Rebecca Weill - 260804001
Benjamin Szwimer - 260804222
Group 36



Lab 2 - CUDA Convolution and Musical Instrument Simulation

November 2nd 2020

1. Abstract

In the first portion of the lab, the image convolution algorithm that uses simple signalling processing was tested to see the effect of parallelism with CUDA. In order to test the performance of the image convolution algorithm, it was run with a different number of threads and the execution times were recorded. The goal of these tests was to see if having a larger number of threads, will lead to faster execution times (lower runtime) for the algorithm. The image convolution algorithm consists of creating a new image whose pixel values are the weighted sum of the original images' pixels, and its neighbouring pixels. We were required to implement the convolution algorithm for a 3x3 weighted matrix and test it on the 3 images provided to us.

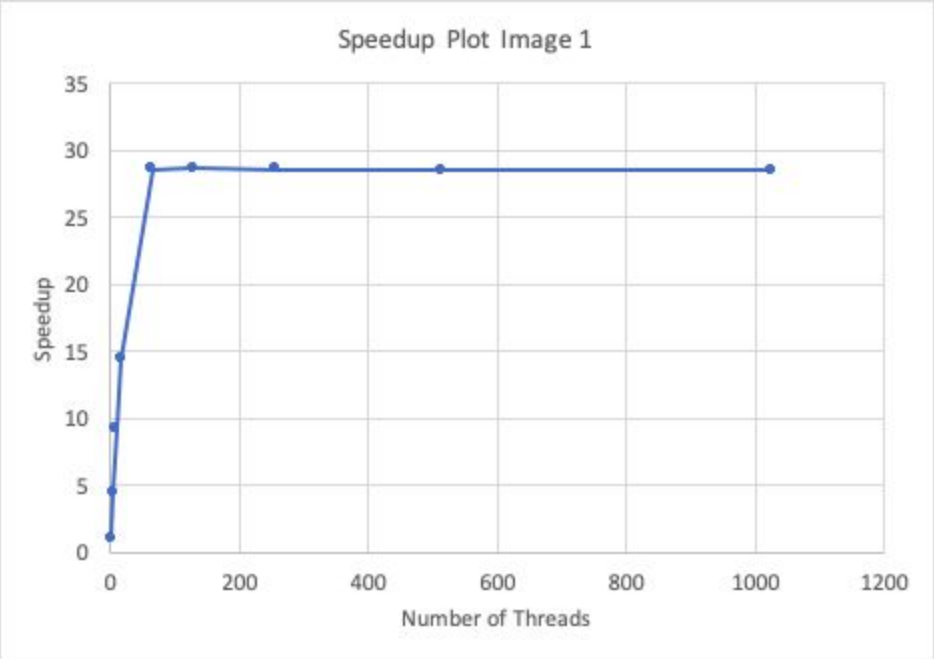
In the second portion of the lab, we were required to synthesise drum sounds using a 2x2 dimensional grid of finite elements. We wrote code that implemented a 4x4 finite element grid sequentially, a 4x4 finite element grid in parallel and finally a 512x512 grid in parallel. In each implementation one had to specify the number of iterations to run the simulation. Specifically for the 512x512 grid, we were required to decompose the grid with different combinations of threads, blocks and finite elements per thread. Each output was recorded and analyzed.

2. Convolution

2a. Testing and Results

Number of threads	Execution Time (ms)	Speedup
1	530.725891	1
4	119.288834	4.449082728
8	58.059776	9.141025467
16	36.989952	14.34783941
64	18.572289	28.57622402
128	18.557823	28.59849946
256	18.583361	28.55919825
512	18.617056	28.50750897
1024	18.611200	28.51647884

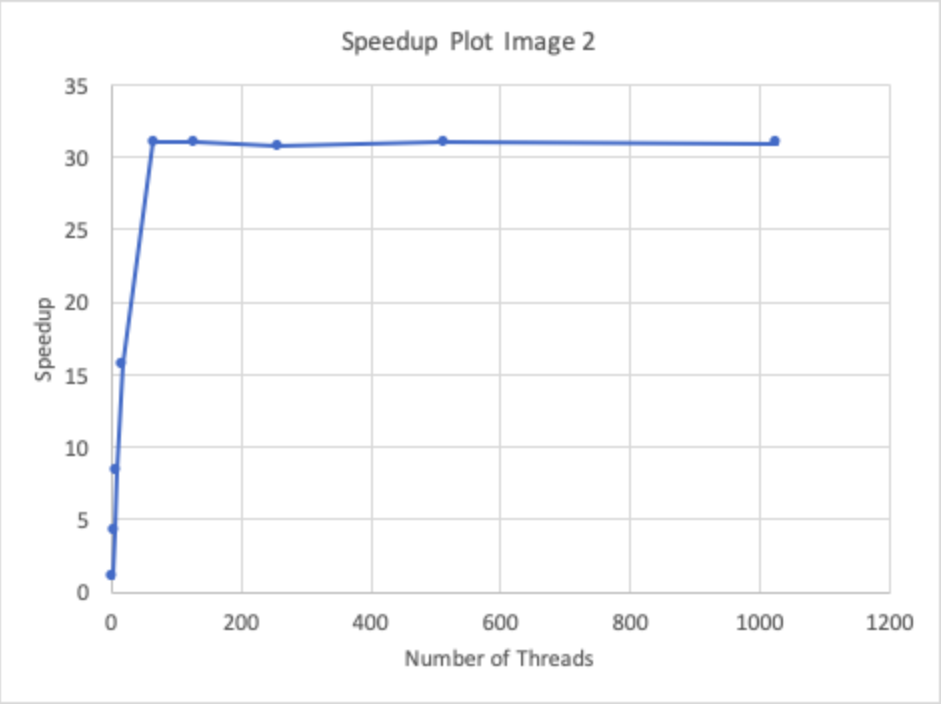
Table 1: Image 1 convolution execution times and speedup



Graph 1: Image 1 speedup plot

Number of threads	Execution Time (ms)	Speedup
1	132.044800	1
4	31.358976	4.210749739
8	15.776768	8.369572272
16	8.414208	15.69307533
64	4.257792	31.01250601
128	4.258816	31.00504929
256	4.293632	30.75363701
512	4.254720	31.03489771
1024	4.268032	30.93809981

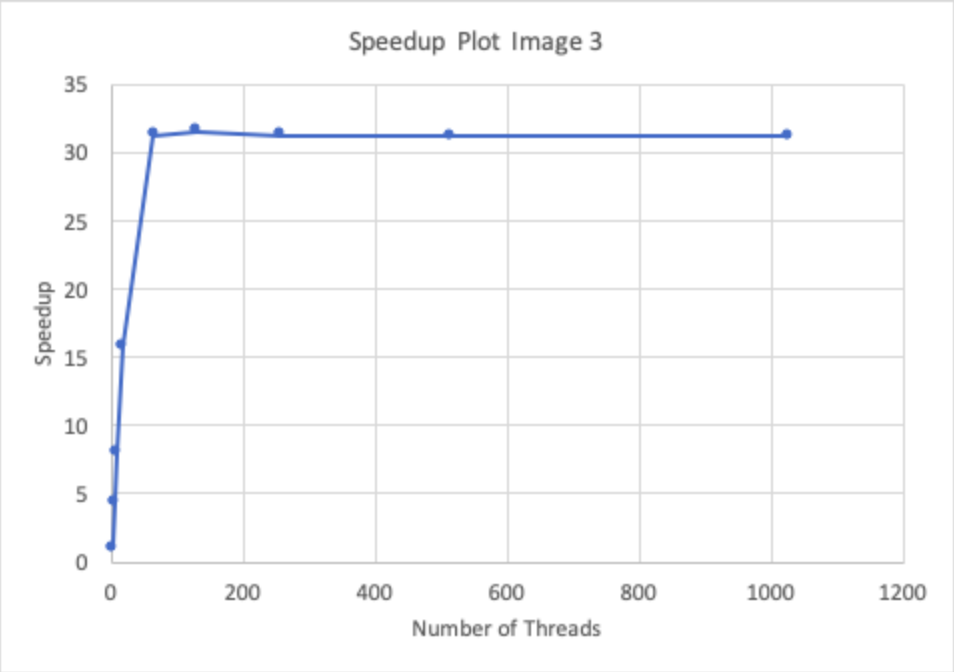
Table 2: Image 2 convolution execution times and speedup



Graph 2: Image 2 speedup plot

Number of threads	Execution Time (ms)	Speedup
1	147.310593	1
4	33.720322	4.368599831
8	18.564096	7.935241932
16	9.337856	15.7756334
64	4.717216	31.2282908
128	4.669440	31.54780723
256	4.717568	31.22596071
512	4.719616	31.21241071
1024	4.722400	31.19401004

Table 3: Image 3 convolution execution times and speedup



Graph 3: Image 3 speedup plot

2b. Parallelization Scheme

The parallelization scheme used for the implementation of the convolution algorithm was to create an explicit memory program so that the CPU and GPU don't share the same memory space. Hence, the program data must be copied from the CPU to the GPU, run the processes in parallel on the GPU and return the results back to the CPU after it is done performing the processes in parallel. The way it was implemented in the code was that every convolution matrix, a 3x3 matrix of pixels, was processed on each thread depending on how many threads were defined. Therefore, however many threads are passed in from the command line, the number of blocks was equal to the (width * height / number of threads)+1. This would ensure the proper setup for how many threads were being used on a block, and each parallel thread would run a single convolution (3x3) matrix.

2c. Discussion

As we can see by the speedup values shown in Tables 1, 2 and 3, we reach a plateau for the speedup values at 64 threads for all 3 images. Any number of threads above 64 does not improve the efficiency of the convolution algorithm.

3. Music Synthesis

3a. Testing and Results

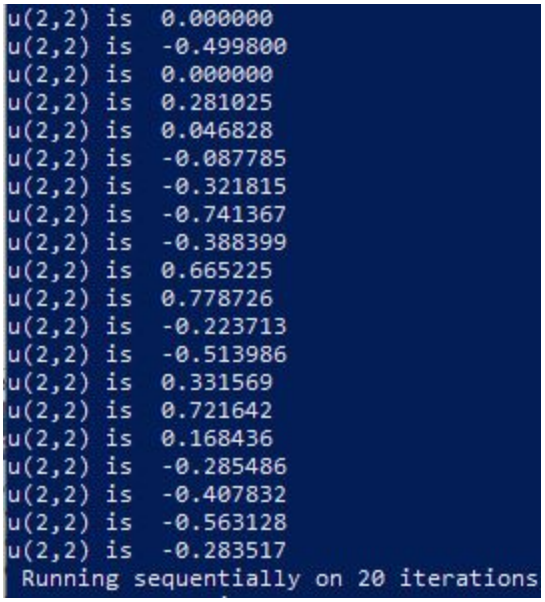


Figure 1: Output on 20 iterations of the sequential processing of a 4x4 grid

Trials	Execution Times (ms)
1	2.603168
2	6.786272
3	6.044000
4	1.563392
5	7.069696
Average	4.8133056

Table 4: Execution times for 5 different trials of sequential processing of a 4x4 grid

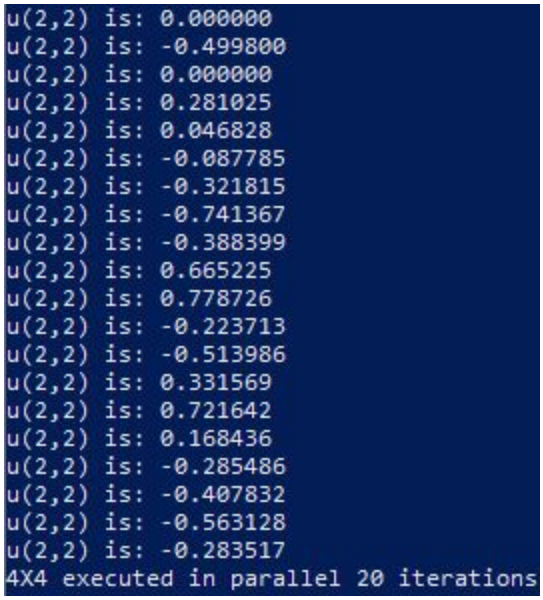


Figure 2: Output on 20 iterations of the parallel processing of a 4x4 grid

Trials	1 Thread Execution Time (ms)
1	6.629376
2	6.635136
3	6.681280
4	6.809696
5	6.033824
Average	6.5578624

Table 5: Execution times for 5 different trials of parallel processing of a 4x4 grid

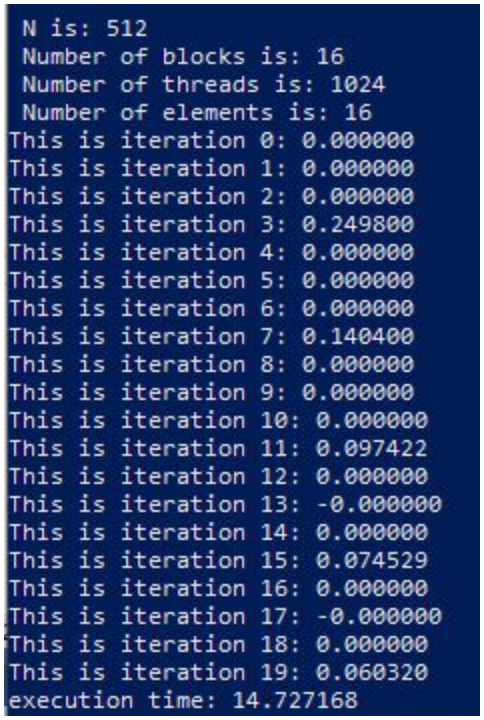
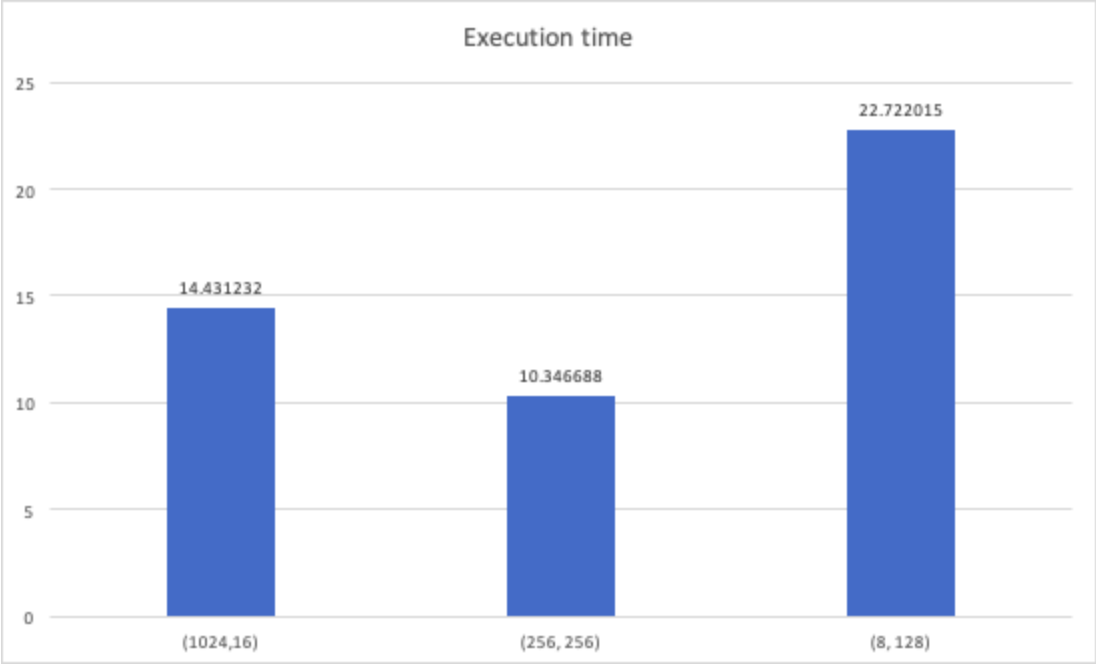


Figure 3: Output on 20 iterations of the parallel processing of a 512x512 grid

# of threads	# of blocks	# of elements/thread	Execution time (ms)
1024	16	16	14.431232
256	256	4	10.346688
8	128	256	22.722015

Table 6: Results for 512x512 grid on 20 iterations



Graph 4: Visual graph of results for 512x512 grid on 20 iterations (execution time in ms)

3b. Parallelization Scheme

For the parallelization scheme of the parallel implementation of a 4x4 grid, each node was associated to a single thread using CUDA. Unified memory was used in order for both the CPU and GPU to share memory space. Each thread took into account whether or not that node was considered part of the interior of the 4X4 grid, the corners and the boundaries as well.

Finally, for the parallelization scheme of the parallel implementation of a 512x512 grid, we used the decomposition by number of threads, number of blocks and number of elements per thread idea described in the lab. This was done to increase run time efficiency since it allowed for the assignment of multiple elements to each thread and minimize the large amounts of computation and GPU overhead endured when clustering into large groups instead of splitting it up into smaller groups like we did by decomposition.

3c. Discussion

As we can see in Figure 1 and Figure 2, we get the same results for the sequential and parallel implementation of the 4x4 grid. Of course, this confirms the correctness of our solution since both should return the same results since they are computing the same thing but one is done sequentially and the other in parallel. For the sequential implementation of the 4x4 grid, everything is computed in the order, one node after each other. Hence, one computation at a time. Moreover, the parallel implementation of the 4x4 grid, we assigned the computations of each node to a single thread. As shown by Table 4 and Table 5 we see that the execution time for the sequential implementation is smaller than the execution time for the parallel implementation. This is due to the fact that for a smaller grid (4x4), sequential is faster than

parallel because it does not waste time allocating threads to blocks or terminating threads. In addition, it does not have to load libraries that the parallel implementation needs. Especially in this case, where we are only using 1 thread per node, the parallelization actually takes more time than sequential.

In Table 6 and Graph 4, we showcase the different computation times associated with different combinations of number of threads, number of blocks and the number of elements per thread. In Graph 1 we see that when there are 8 threads and 128 blocks, we get the highest execution time. In this case there are 256 elements assigned to each thread. In comparison to the lowest computational speed which goes to the case where there are 256 threads and 256 blocks, the elements assigned per thread will only be 4. Since there are less elements assigned per thread, it can compute the result much faster than in the previous case where there were 256 elements assigned per thread.

Due to the parallelization scheme used in the produced code, this behaviour was expected. If there are more threads and blocks, there will be less strain on the elements per thread, and therefore lead to faster execution times. We know this since we can compute everything by using the following formula.

$$(512 \times 512)/(\#\text{ of threads} \times \#\text{ of blocks}) = \#\text{ of elements/thread}$$

As the denominator gets larger, the result on the right gets smaller. As previously mentioned if we reduce the # of elements per thread, the results will be calculated more efficiently and produce a smaller execution time. We achieve parallelization when every block is used optimally. Hence using the largest combination of threads and blocks will lead to lower computational times since the threads won't be overwhelmed with elements to compute.

4. Conclusion

In conclusion in the first part of the lab, we saw that the speedup plateaued at 64 threads for every image. This means that at 64 threads, the convolution algorithm has reached its best performance. Anything above 64 threads will do nothing to the runtime of the convolution algorithm.

In the second part of the lab, we saw that the sequential implementation of the 4x4 grid led to slightly better performance times than the parallel implementation of the 4x4 grid. This is because the sequential program does not need to create and allocate threads to blocks, terminate said threads and load libraries in order to function. It works simply by computing the program line by line and producing the result. Moreover, for the parallel implementation of the 512x512 grid, the fastest execution time was when there were 256 threads, 256 blocks and 4 elements per thread. This led to the discovery that if there are less elements assigned per thread, the runtime will be faster.