Lab 2: CUDA Convolution and Musical Instrument Simulation

ECSE 420 Parallel Computing Fall 2023

5 November 2023

Group 14: Samantha Handal 260983914 & Rebecca Mizrahi 260975001

**Part A**

Measured runtime when the number of threads used is equal to {1, 4, 8, 16, 64, 128, 256, 512, 1024}:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | **Images** | | |
| **Test\_1** | **Test\_2** | **Test\_3** |
| **Threads** | **1** | 509.442 ms | 200.946 ms | 208.742 ms |
| **4** | 203.756 ms | 56.0597 ms | 62.0318 ms |
| **8** | 124.423 ms | 27.9761 ms | 31.131 ms |
| **16** | 62.2968 ms | 14.0145 ms | 15.5676 ms |
| **64** | 29.9352 ms | 6.67011 ms | 7.41987 ms |
| **128** | 29.3284 ms | 6.58253 ms | 7.31075 ms |
| **256** | 29.2579 ms | 6.57315 ms | 7.29699 ms |
| **512** | 29.2828 ms | 6.58442 ms | 7.32179 ms |
| **1024** | 29.6878 ms | 6.69491 ms | 7.46634 ms |

We are using CUDA to parallelize the computation for convolution. CUDA parallel thread execution on a GPU, which is well-suited for tasks like image convolution. To organize threads, the work is divided into blocks, where each block consists of multiple threads. The number of threads in each block is given via input, and the blocks are calculated accordingly to handle all pixels in the image. As such, the kernel handles one RGBA value at a time. For memory management, explicit memory allocation is used for data transfer and allocation between the CPU (host) and GPU (device) to ensure consistent time measurement.

The speedup plots, which can be found in appendix A, show how the execution time varies with the number of threads for different test images. Speedups use the execution time for a single thread as the reference point. For Test\_1.png, the execution time decreases as the number of threads increases, as is expected for parallelized code that is computationally intensive. However, after a certain point, increasing the number of threads provides diminishing returns. In this case, the optimal number of threads appears to be around 64-128. Convolution with Test\_2.png provides similar results to Test\_1.png; the execution time decreases as we increase the number of threads. The speedup is more pronounced in this case. It may be that Test\_2.png is a more complex image to process. Test\_3.png again follows a similar pattern, but with a slightly higher time. It suggests that Test\_3.png requires more computations.

Architecture:

The execution time results are directly influenced by Google Colab GPU's computational power and memory capacity.

Google Colab offers a free T4 GPU for computational tasks, which is part of NVIDIA's Turing family. T4 has 16 Streaming Multiprocessors (SMs) and a total of 2560 CUDA cores. This means it's well-suited to handle thousands of threads in parallel, displaying benefits from increasing the number of threads in grid configuration.

With a memory bandwidth of up to 320 GB/s, data transfer between the global memory and the CUDA cores is rapid, minimizing potential bottlenecks that might arise when fetching pixel data.

Google Colab uses a unified memory model for GPU memory allocation; memory space is shared between the CPU (host) and GPU (device). This simplifies allocation and migration, as data migration between the CPU and GPU is managed automatically. In code that uses explicit memory allocation, for when one wants a more fine-grained control over memory usage, the code bypasses the automatic memory migration provided by the unified memory model. Colab allows you to explicitly allocate memory on the GPU if you choose to do so.

To summarize, these experiments showcase how to harness the parallel processing capabilities of the GPU to significantly accelerate image convolution. The speedup plot highlight the advantages of GPU acceleration for computationally intensive tasks like image processing. They demonstrate how performance scales with the number of threads, helping one find an optimal balance between computational resources and execution time.

**Part B**

Provide your printed outputs for each part (1, 2 & 3). (5 points)

Printed output for part 1, sequential 4x4 finite element grid:

(2, 2): 0.000000

(2, 2): -0.499800

(2, 2): 0.000000

(2, 2): 0.281025

(2, 2): 0.046828

(2, 2): -0.087785

(2, 2): -0.321815

(2, 2): -0.741367

(2, 2): -0.388399

(2, 2): 0.665226

(2, 2): 0.778726

(2, 2): -0.223713

Four iterations of the entire grid are printed in appendix B.1.

Printed output for part 2:

(2, 2): 0.000000

(2, 2): -0.499800

(2, 2): 0.000000

(2, 2): 0.281025

(2, 2): 0.046828

(2, 2): -0.087785

(2, 2): -0.321815

(2, 2): -0.741367

(2, 2): -0.388399

(2, 2): 0.665225

(2, 2): 0.778726

(2, 2): -0.223713

Time Elapsed: 0.497024 ms

Find the full output for parts 1, 2, and 3 in appendix B.

Table comparing the execution times for each combination of threads, blocks, and finite elements per thread:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | Blocks | | | |
| 32 | 128 | 512 | 1024 |
| Threads | 32 | 9.79427 ms | 6.22832 ms | 5.51571 ms | 5.4873 ms |
| 128 | 6.27814 ms | 5.43958 ms | 5.41683 ms | 5.53002 ms |
| 512 | 5.49824 ms | 5.50288 ms | 5.91091 ms | X |
| 1024 | 5.39027 ms | 5.78717 ms | X | X |

In our given code, we experimented with various combinations of thread blocks and threads per block with a 512x512 grid to measure kernel execution time.

Balancing the workload among parallel threads is essential for efficient GPU utilization to minimize computation time. As the experiments demonstrate, the choice of thread configuration has a significant impact on the execution time, and the ideal number of finite elements processed per thread depends on the computation's complexity. Using fewer threads and fewer blocks and balancing more operations per thread led to a longer execution time. It is not fully utilizing the GPU's processing power, as is shown by the slow result of using 32 threads and 32 blocks. On the other hand, the scheme with 1024 threads and 32 blocks, with each thread handling 2 computations has the lowest execution time. While a greater number of threads and blocks result in more efficient parallelized computations and lower time elapsed, our experiment shows that the most efficient combination appears to be a higher thread count with a lower number of blocks. It is also important to note that there is a limit to the increases achieved with higher thread counts or block numbers, after which increasing the number of threads or blocks provides diminishing returns. For example, for a block count of 128, the time elapsed until the kernel synchronizes is faster with 128 threads than 512 or 1024. Similarly, for 512 blocks, it is most efficient to use 128 threads and not 512.

These observations can be summarized with the results of the fastest execution being 32 blocks and 1024 threads, followed closely by 512 blocks and 128 threads, and the slowest execution being 32 threads and 32 blocks, followed by 512 blocks and 512 threads.

In summary, our experiments successfully demonstrated the impact of different parallelization schemes on the execution time of your finite element code. For optimal performance, it's essential to test different schemas, taking into account the computational complexity of the code, and considering the characteristics of the GPU to adapt the configuration accordingly.

**Appendix**

A.1

A graph with a line

Description automatically generated

A.2

A graph with a line and a point

Description automatically generated

A.3

A graph with a line graph

Description automatically generated

A.4

A graph with different colored lines and dots

Description automatically generated

B.1 4x4 grid at 4 iterations, from part 2.1

A screenshot of a computer

Description automatically generated

B.2 Result of part 2.1

A screenshot of a computer program

Description automatically generated

B.3 Result of part 2.2

A screenshot of a computer

Description automatically generated

B.4 Different combinations of threads, blocks and finite elements per thread on a 512x512 grid.

A screen shot of a computer

Description automatically generated