Matrix Processing using CUDA

Francisco Henriques, *Student, FCT-UNL,* Gabriel Gonçalves, *Student, FCT-UNL,*

Abstract—This paper explores different ways to parallelize a given sequential code that applies to any given image two filters: one area filter and one point filter, resorting to the CUDA API. This study was developed within the scope of the course High-Performance Computing at NOVA School of Science and Technology.

Index Terms—Parallel Architectures, Programming Techniques, Parallel Systems, CUDA, LaTeX.

1 Introduction

With this paper we have the intention of analyzing the performance impact of parallelization resorting to the Graphics Processing Unit (GPU).

We will start from a basis of a sequential version of the code that simulates the application of a set of filters to a given image.
From there we will study the concurrency problems that there might exist on the serial version, remove those dependencies and analyze the efficiency of the given parallelization.
The source code was written in the C Language, with the purpose of evaluating the improvements of the performance using the CUDA API.

The source code given consists in the application of two different classes of filters, a point filter, and an area filter. This is done by sequentially iterating through each pixel, and compute the new pixel value that is generated by each filter.

2 CUDA and its uses

Stands for "Compute Unified Device Architecture." CUDA is a parallel computing platform developed by NVIDIA and introduced in 2006. It enables software programs to perform calculations using both the CPU and GPU. By sharing the processing load with the GPU (instead of only using the CPU), CUDA-enabled programs can achieve significant increases in performance. [1]

The developer still programs in the familiar C, C++, Fortran, or an ever expanding list of supported languages, and incorporates extensions of these languages in the form of a few basic keywords. These keywords let the developer express massive amounts of parallelism and direct the compiler to the portion of the application that maps to the GPU. [2]

3 Filters

Image filters are examples of matrix processing that can be taken to the GPU. In this paper we will be studying the improvement of performance that is achieved by resorting to GP-GPU.

3.1 Area Filter

Area filters, consists in considering an area around each pixel to define the final value of that pixel.Each neighbour pixel will has a certain weight in the value of the center pixel depending on the applied filter. We will be considering just the direct neighbors, we get a 3x3 grid of pixels with the original pixel at its center and a Gaussian filter that is used to blur the image.

3.2 Point Filter

Point filters consist in applying a function on the pixel value to generate a new pixel. For the point filter, we pretend to gray the image using the following expression:

$$(r, g, b) = alpha * (c, c, c) + (1 - alpha)(r, g, b)$$

where:

$$c = 0.3r + 0.59g + 0.11b$$

Alpha is a value in [0 .. 1] that defines the color shift to gray scale.

4 Sequential Version

The sequential version iterates through every pixel and applies the aforementioned filters. This is quite slow as it has to iterate twice every pixel as the filters are independent and have to be executed sequentially. Iterating through every pixel on a matrix has a quadratic time complexity.

This implementation will be the basis of our parallelization and will be compared to our final version in the comparison section.

5 Parallel Version

To parallelize the aforementioned sequential version it was required to remove the concurrency problems so that each iteration of the loop would be independent from the other ones. Each piece of the sequential code presented a challenge for the parallelization.

We started with our sequential basis and through CUDA API,implemented a basic parallelized version that launches a set of blocks of pixels so that each thread only computes one pixel. Each thread then calculates which pixel it must

apply the filters. This version is implemented so that consequent threads compute consequent pixels, this is, a block of N threads compute a line of N pixels of the image.

In this section we will discuss possible optimizations to this parallel approach such as: the size of each block of threads, the layout of the block relative to the image, using shared memory between threads within the same block and the use of streams to overlap data transfers.

5.1 Block Size

One of the most important elements of CUDA programming is choosing the right grid and block dimensions for the problem size. It's not always clear which dimensions to choose so we created a script that allowed us to identify what are the effects of grid and block dimensions on execution times.

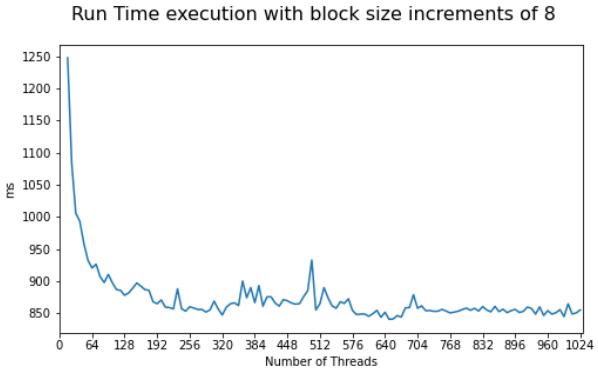

Fig. 1. Plot that correlates the number of threads per block and the runtime

From this results (Figure 1) we can conclude that:

1) Making the number of threads per block a multiple of 32 is the best practice. This is happens because each streaming multiprocessor (SM) has 32 CUDA cores.
2) The performance plateaus on ∼512 threads per block therefore launching more threads than that is redundant

5.2 Grid Layout

Initially we implemented a parallelized version where consequent threads process consequent pixels of the image. However as the area filter is dependent on a central pixel neighbours, usually this were never computed by the same block therefore we implemented a new version where instead of a block of threads compute a "line" of the image, the block would compute a square grid of NxN pixels.

5.3 Shared Memory

Shared memory is much faster than local and global memory. In fact, shared memory latency is roughly 100x lower than uncached global memory latency. Shared memory is allocated per thread block, so all threads in the block have access to the same shared memory. Threads can access data in shared memory loaded from global memory by other threads within the same thread block. [3]

In our implementation each thread is responsible for importing the pixel it will compute to shared memory and there are a set of "extra" threads that will import the neighbour pixels that are not a part of that block computation. As we will discuss in the next section, shared memory was what gave us the greatest improvement in performance of all techniques applied.

5.4 Streams

The last mechanism we will test is to use CUDA Streams to overlap data transfers. A stream in CUDA is a sequence of operations that execute on the device in the order in which they are issued by the host code. While operations within a stream are guaranteed to execute in the prescribed order, operations in different streams can be interleaved and, when possible, they can even run concurrently. [4]

It was not clear which dimensions to choose so we created a script that allowed us to identify how many streams to launch (Figure 2).

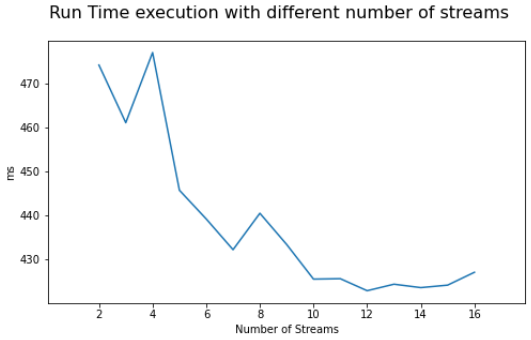

Fig. 2. Plot that correlates the number of streams and the runtime

From this basic plot we concluded that we would achieve the best performance by launching ∼12 streams.

5.5 Pinned Memory

Host (CPU) data allocations are pageable by default. The GPU cannot access data directly from pageable host memory, so when a data transfer from pageable host memory to device memory is invoked, the CUDA driver must first allocate a temporary page-locked, or "pinned", host array, copy the host data to the pinned array, and then transfer the data from the pinned array to device memory, as illustrated below. [5]

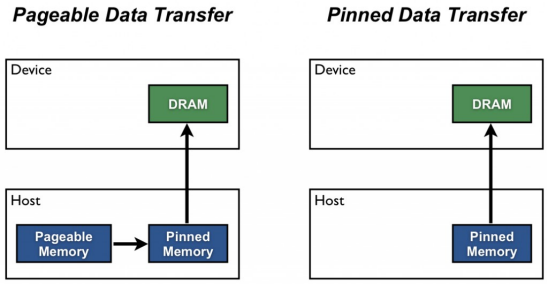

Fig. 3. Ilustration of the process of accessing pageable and pinned memory

Since we're dealing with a situation where, according to nvprof, our program is spending 90% of the time in data transfers between the device and the host, pinned memory is extremely advantageous to use. According to our findings, this simple fix seemed to double memory bandwidth, which massively sped up our image processing.

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	67.96%	2.51853s	5	503.71ms	496.89ms	506.98ms	[CUDA memcpy DtoH]
	24.32%	901.31ms	10	90.131ms	672ns	181.44ms	[CUDA memcpy HtoD]
	4.90%	181.65ms	5	36.329ms	34.730ms	41.413ms	areaFilterTilesSM3(int*, int*, int, int, int*)
	2.82%	104.64ms	5	20.928ms	19.894ms	24.706ms	pointFilter(int*, int const *, int, int, float)
API calls:	71.53%	3.7102s	15	247.35ms	13.637us	573.40ms	cudaMemcpy
	11.21%	581.52ms	9	64.614ms	326.99us	216.34ms	cudaDeviceReset
	8.65%	448.87ms	20	22.443ms	140.16us	91.866ms	cudaMalloc
	7.24%	375.68ms	2	187.84ms	809.68us	374.87ms	cudaHostAlloc
	1.36%	70.668ms	20	3.5334ms	1.0385ms	5.1873ms	cudaFree
	0.00%	168.10us	10	16.809us	4.8970us	32.417us	cudaLaunchKernel
	0.00%	89.407us	101	885ns	104ns	36.416us	cuDeviceGetAttribute
	0.00%	29.502us	5	5.9000us	3.5940us	13.158us	cudaDeviceSynchronize
	0.00%	15.976us	1	15.976us	15.976us	15.976us	cuDeviceGetName
	0.00%	5.3780us	1	5.3780us	5.3780us	5.3780us	cuDeviceGetPCIBusId
	0.00%	1.5040us	3	501ns	183ns	920ns	cuDeviceGetCount
	0.00%	987ns	2	493ns	158ns	829ns	cuDeviceGet
	0.00%	440ns	1	440ns	440ns	440ns	cuDeviceTotalMem

Fig. 4. Results of running nvprof

6 Comparison

In this section our different approaches to parallelizing the original program by minimizing runtime will be explored and each approach's performance tested in a NVIDIA GeForce GTX 1060. This GPU is composed of 1280 CUDA Cores and 6 GB GDDR5 of memory.

All the future graphs shown were created by averaging the runtime results from 5 runs with a 9000x12000 image (Figure 5).

Fig. 5. rome.ppm

6.1 Baseline Version

As expected, our naive implementation had a massive speed up relatively to the sequential version with a value of about 900 as can be seen in Figure 6 This is due to the use of 108 million threads,as many as pixels, to help with the computation.

This naive version will serve as the basis for our future comparisons with the applied mechanisms to improve performance.

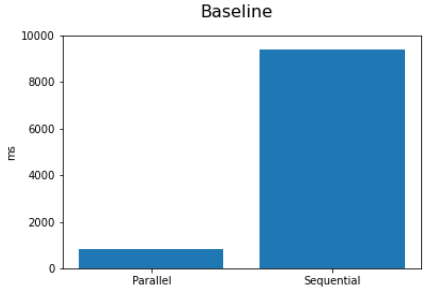

Fig. 6. Comparison between our naive implementation and the base sequential version

6.2 Grid Layout

As seen on Figure 7 , the layout of the grid has little to no impact in the performance of our code. However it made the implementation of shared memory that was discussed in section 5.3 and section 6.3.

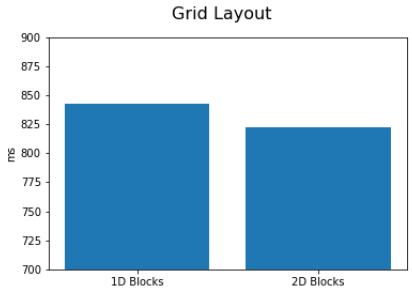

Fig. 7. Plot that correlates the grid layouts and the runtime

6.3 Shared Memory

This mechanism gave us a massive speedup, this was the case because by using shared memory we reduce the amount of times we access global GPU memory to once per pixel. The remaining accesses are made in the shared memory present in every SM. This type of memory allows for much faster reading and writing, making it worthwhile to perform the extra computations to populate it.

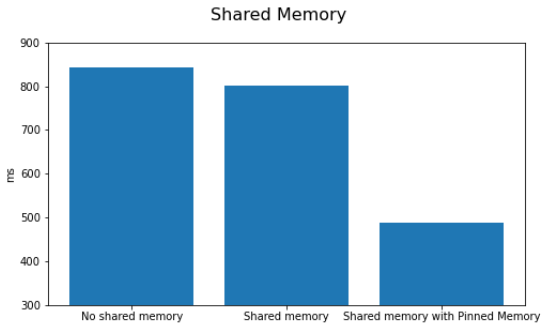

Fig. 8. Plot that correlates the use of shared memory and the runtime

6.4 Streams

Streams allow us to overlap computations with communication by partitioning the image into smaller images and processing each partition in a different stream. As we can see in Figure 9, using streams allows for a slightly faster runtime, however as our kernels are mostly limited by the memory bandwidth of the GPU, the benefit is quite small.

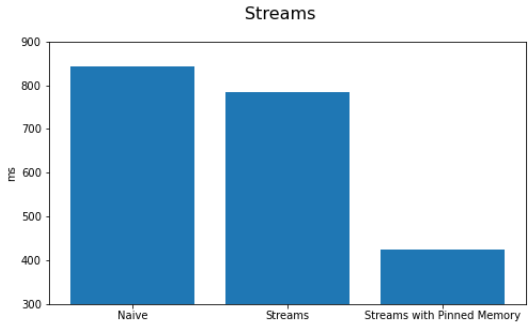

Fig. 9. Plot that compares the runtime of the naive, streams and streams with pinned memory implementations

6.5 Best vs Baseline vs Sequential

As we can see in figure 10, our parallelized code is a lot faster than the initial sequential version. We observed a speedup of 9 in the Naive version and 18 for our best version with all the optimizations.

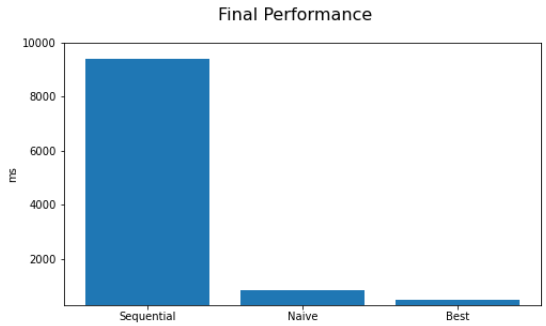

Fig. 10. Plot that compares the given sequential version against our first and best implementations

In figure 11 we can see that the understanding of the underlying concepts for GPU parallelism is essential. This understanding allowed us to double the speed of our program and properly take advantage of the GPU.

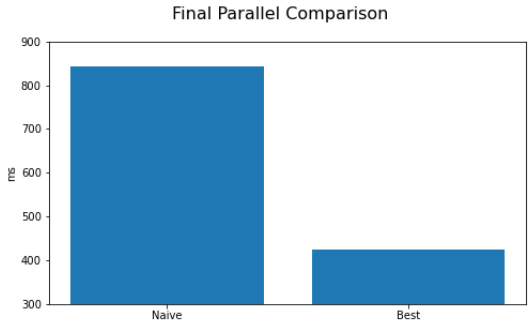

Fig. 11. Plot that compares our first version to our best one using shared memory, streams, tiling and pinned memory

7 Technical Observation

In our implementation, we decided to not compute the edge cases for different streams. This means that our best version will not return the same exact image as the sequential version does since we considered it an unnecessary computation. In real applications padding algorithms are used or borders are not processed at all to avoid algorithmic complexity and improve performance. The submitted version uses the function "sharedMemoryTilesStreamsRestrictPinned", which due to the usage of streams not only returns a very slightly different image but also can´t process very small images due to not being able to divide it into the 12 streams we settled on. For very slighly worse performance and complete correctness the function "sharedMemoryTilesRestrictPinned" should be used instead. This version has the same optimizations but uses one stream instead of 12.

8 Conclusion

In conclusion, small modifications in the implementation can greatly influence the performance or cause serious problems in the program. Being able to analyze the data, using profilers, and removing dependencies that may alter the flow and output of a program in order to parallelize the program is crucial in the work of a computer engineer.

Having an API like CUDA also facilitates the work a lot as we are not required to implement some of the tools that it provides. Allowing developers to focus on what is needed to optimize and not on the intricacies of launching and managing threads and thread pools.

In the end, we can conclude that even though a single GPU thread might be less efficient in optimizing our code, as a GPU has many times more threads than a conventional CPU we can achieve greater speedups and therefore much smaller runtimes.

In this specific case, our speedup wasn't higher due to the small number of computations done in the kernel, making the memory bandwidth a huge bottleneck in performance, with no way to optimize it further. If we did not consider initial load times our speedup would have certainly been much higher.

References

[1] Techterms.com. 2021. CUDA (Compute Unified Device Architecture) Definition. [online] Available at: https://techterms.com/definition/cuda [Accessed 4 December 2021]

[2] The Official NVIDIA Blog. 2021. What Is CUDA — NVIDIA Official Blog. [online] Available at: https://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/ [Accessed 4 December 2021].

[3] Harris, M., 2021. Using Shared Memory in CUDA C/C++ — NVIDIA Developer Blog. [online] NVIDIA Developer Blog. Available at: https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/ [Accessed 5 December 2021].

[4] Harris, M., 2021. How to Overlap Data Transfers in CUDA C/C++ — NVIDIA Developer Blog. [online] NVIDIA Developer Blog. Available at: https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/ [Accessed 5 December 2021].

[5] Harris, M., 2021. How to Optimize Data Transfers in CUDA C/C++ — NVIDIA Developer Blog. [online] NVIDIA Developer Blog. Available at: https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/ [Accessed 7 December 2021].

ACKNOWLEDGMENTS

The authors wish to thank the professor of the High-Performance Computing Course for answering our questions and providing valuable observations that greatly helped the development of this report. We would also like to thank Zowie Haugaard (https://github.com/zowiehi) for helping us convert png images into ppm p3 image format.

Francisco Henriques Student at NOVA School of Science and Technology, Computer Science Master's degree

Gabriel Gonçalves Student at NOVA School of Science and Technology, Computer Science Master's degree