Grenoble INP - ENSIMAG

MoSIG

GPU Computing

JPEG Encoder GPU Version

Students :
DEMEYERE Hugo
SCIASCIA Matteo

Professor :
PICARD Christophe

Year 2023-2024

Table of content

1. Abstract

The purpose of the assignment is to build a project that could benefit greatly from parallelization, by using CUDA and its functionalities.

In our case, we decided to take the JPEG encoder project we did in our 1st year of engineering school (JPEG Project ENSIMAG 1st year), which converts images from raw format (PGM for gray images or PPM for color images) into compressed format (JPEG), and to parallelize it. More specifically, we use the most widespread JPEG mode, called baseline sequential, which displays the image in a single pass, pixel by pixel.

In order to carry out this assignment, we parallelized the different "parallelizable" algorithms corresponding to specific processing (compression and filtering) steps of the original raw image. In addition to parallelizing the processing steps within a same image data unit (called "block"), we also parallelized the processing across multiple image blocks concurrently. This approach and solution allowed us to encode (compress and filter) several image data blocks simultaneously, which we then wrote sequentially into the output .jpeg file.

Our study compared the performance of GPU and CPU implementations of a JPEG encoder across different image sizes. We found that while the CPU encoder performed better with smaller images due to CUDA overhead, the GPU encoder consistently outperformed as image size increased. This highlights the GPU's advantage with larger datasets, overcoming the CUDA overhead.

Based on our results, it can be concluded that GPU acceleration offers significant performance advantages for image encoding tasks, particularly with larger datasets. Despite initial overhead associated with CUDA utilization, the parallel processing capabilities of the GPU encoder effectively offset this overhead as image size increases. Therefore, leveraging GPU acceleration can greatly enhance the efficiency and speed of image encoding processes, making it a valuable tool for various applications requiring high-throughput image processing.

2. Design Methodology

2.a. Overview of the encoder steps

This section details the successive steps involved in encoding, i.e. converting an image in PGM & PPM format into an image in JPEG format. For more details, click here : JPEG Project ENSIMAG 1st year.

— **Partitioning** : First, the image is partitioned into macroblocks, or MCUs for Minimum Coded Unit. MCUs are usually 8x8, 16x8, 8x16 or 16x16 pixels in size, depending on the sampling factor (compression technique that consists in reducing the number of values, called samples, for certain image components). Each MCU is then reorganized into one or more 8x8 pixel blocks.

— **Compression** : Next, an 8x8 block is compressed. Each block is translated into the frequency domain by discrete cosine transform (DCT). The result of this processing, called a frequency block, is still an 8x8 block, but whose coordinates are no longer pixels, i.e. positions in the spatial domain, but amplitudes at given frequencies.

— **Rearranging** : A small but important step is to rearrange the frequencies through zig-zag (ZZ) reordering to sort the values from the smallest frequenciest to the biggest ones.

— **Filtering** : As the eye is less sensitive to high frequencies, it is easier to filter them with this frequency representation. This filtering step, known as quantization, destroys information in order to improve compression, to the detriment of image quality (hence the importance of the choice of filtering). It is performed block by block using a quantization filter, which may be generic or specific to each image. This way, we obtain statistically more 0's at the end of the vector.

— **Coding and writing** : This vectorized block is then compressed using several lossless encodings in succession : firstly, RLE encoding to exploit repetitions of 0, difference encoding rather than value encoding, and then entropy encoding, known as Huffman encoding, which uses a dictionary specific to the image being processed.

— The above steps are applied to all the blocks making up the image's MCUs. The concatenation of these compressed vectors forms a bitstream which is stored in the JPEG file.

Those described steps can be found in the figure 1, with "RGB to YCbCr" and "DOWN SAMPLER" being part of the partitioning step, "DCT" representing the compression, "D+ZZ" the filtering, and "Variable Length Encoding" the coding step.

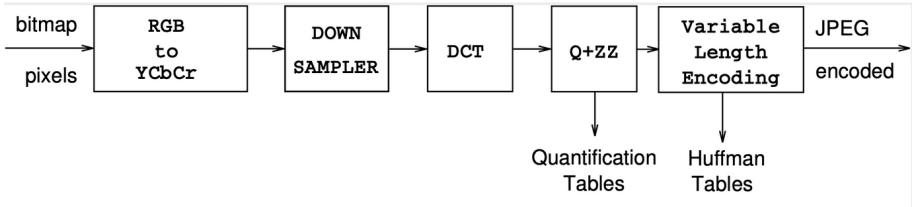

FIGURE 1 – Encoder Pipeline

2.b. Algorithms to be parallelized

In the previous described algorithms (encoder steps), parallelization can be introduced at several places. After having partitioned the original file (.pgm & .ppm) data into MCUs and organized them into 8x8 pixel blocks, the compression (using DCT) and filtering steps (quantization) of each 8x8 pixel block can be parallelized

in 2 ways :

— **Inside of a single block** : The compression and filtering computations can be processed simultaneously.
— **In between the blocks** : Those 2 steps are specific to a single block, and therefore those computations can be done independently from one block to another.

2.c. Limitations

Even though those steps can be parallelized, we have to keep in mind that the coding step has to happen sequentially and cannot be parallelized because it consists in writing to the output .jpeg file. Therefore, we cannot have concurrent writings to the same file and the writing has to happen sequentially in order for the image to be correctly decoded.

2.d. Approaches to the problem

2.d.1. Parallelizing within a single block

At first, we decided to do all those previous steps for each 8x8 block one at a time and parallelize only the computation **inside** of a block and **not in between** the blocks. Therefore, this meant partitioning one 8x8 pixel block, compressing it, filtering it, coding and writing it to the jpeg file, one at a time, and re-doing all those steps for the next block, until the original file's end is reached. The problem with this approach is that with parallelizing only inside of a 8x8 block, the parallelization is done on too few data : we allocated memory on gpu for only a single thread block of 64 threads, and this for every single 8x8 pixel block. Therefore, the allocation on gpu of a single 64-elements block with 1 thread block of 64 threads (1 thread for each element computation) and the communication between the cpu and gpu were definitely not worth it compared to the little parallelization obtained on the compression and filtering steps on a single 64-elements block. This approach was always taking more time than using the cpu only.

2.d.2. Parallelizing a line of blocks

As we saw that our initial approach of a single block was not definitely not performing well enough, we decided to not only parallelize within a single 8x8 block but as well as across the blocks. This meant allocating on gpu an array of **several** 8x8 pixels blocks and and therefore an equivalent number of thread blocks of 64 threads each. In addition to the parallelization happening inside a 64-threads block, several thread blocks would do their computations concurrently with one another.

This solution already showcases greater parallelization by leveraging processing on many 8x8 blocks concurrently, and therefore offers better performances. To give a rough estimate of the observed parallelization, our biggest image (5120 x 4920 pixels) is processed by bunch of $4920/8 = 615$ blocks (of 8x8 pixels each) ; each one

of them is processed simultaneously. Without any arithmetic, we can clearly see that the processing of the entire image is done much faster than by processing each 8x8 pixels block at a time.

2.d.3. Parallelizing the maximum possible number of blocks

Finally, why stop at a single line of 8x8 pixels blocks when we can parallelize on more data ?

On a side note, we are always working with only a single allocation on cpu and on gpu (of the original image partitioned data into 8x8 pixels blocks, which then get processed on gpu and transferred back on cpu) in order to limit the communication and round-trips between the cpu and gpu and optimize as much as possible time performance. Due to this as well as the variability in the maximum amount of data that can be allocated on different computers and operating systems, we opted to parallelize based on the maximum amount of data achievable on both CPU and GPU.

This approach therefore consists in allocating memory for the processing of the whole image if possible and parallelize on **all** the 8x8 blocks in the image. If the allocation fails, we divide by 2 the amount of data to allocate (and therefore to process), and this recursively until the allocation successes. As a remark, we observed that our dataset of images always successfully allocated the whole image, and therefore the parallelization could occur on all the 8x8 blocks simultaneously. This solution brings about even better performances than our 2nd approach as it processes many more blocks than just the ones on a single image line.

This 3rd approach is therefore the one retained in our final version of the project, as it offers the best performances observed so far.

2.e. Master and slave execution and communication

As described above in retained solution, the master (host) is doing the tasks that cannot be parallelized such as the partitioning of the original file data into 8x8 blocks, and later the coding and writing to the output .jpeg file. In between those 2 steps, the master is delegating the processing (compression and filtering) of the partitioned data blocks to the slaves (device threads) as this processing can be parallelized. As stated previously, the master delegates the processing of as many partitioned blocks as possible to the slaves by allocating memory for this partitioned data and copying all this data to the device (GPU) in one go. The slaves consist of as many thread blocks as there are partitioned 8x8 blocks sent to the device, and each thread block is composed of 8x8 threads, with each single thread responsible of the computations for the output of the corresponding input element.

The slaves execute the compression and filtering for each 8x8 input block passed to the device concurrently of other blocks. During the processing, within a thread block, the slaves communicate with each other through shared memory in

order to store temporary data that is later needed in the next steps of the processing. We also had some constraints during the compression step as the DCT algorithm had to be done row-wise before being done column-wise, and to meet this constraint, we had to force the threads inside a single thread block to wait for each other, and synchronize in between the row-wise and column-wise computations. The same goes for the quantization (filtering) step, that needs to happen after the compression (DCT) finished ; a thread synchronization is also needed here. We can see that even though those steps can be parallelized, the general processing steps (compression and filtering) have to occur sequentially within a thread block, which adds a bit of complexity.

Once all the processing steps are done for the allocated partitioned data, the processed data is transferred back to the master (host/CPU), which then proceeds to the coding and writing to the output .jpeg file in a sequential manner.

Once all those previous steps are done for the maximum possible number of allocated partitioned blocks, we move to the next partitioned blocks and repeat those steps again.

This way, we minimize the communication and round trips between the master and slaves, which proved to be very heavy and slow operations when testing our initial approach (described in previous subsection). In addition to minimizing the communication between the master and slaves, we decided to allocate and send the greatest amount possible of partitioned data to the device so that we could parallelize as many 8x8 pixels blocks as possible, and obtain the best possible time performances.

2.f. Potential future optimization leads

Here are some potential optimization leads that we could explore were we to further improve our project :

— An optimization we could have explored would be to potentially parallelize the partitioning of the original image into MCUs directly from the gpu instead of partitioning it sequentially in the cpu, as it consists in reading from a file and not writing to it.

— Another optimization that we could have tried doing would be to parallelize within each stage of DCT compression algorithm. Indeed, this algorithm consists of 4 stages, each of which represents several operations (additions/-multiplications) that could be parallelized. For instance, the 1st stage consists of 8 assignments each preceded by 1 addition. An idea would be to create 8 threads, each of which could do 1 of the 8 additions + assignments. This would have allowed us to parallelize at an even finer granularity level.
That being said, the 4 stages must be executed sequentially, which would force us to add synchronization points in order for the threads to wait for each other in between each 2 consecutive stages. Nevertheless, we were thinking that the

cost of having several threads do an operation each, managing them with if statements (to make sure each one of them does the correct operation) and the overhead caused by the synchronization would potentially not be worth the very little parallelization this optimization idea brings. In our project, we preferred to focus on greater parallelization (across blocks), but this could be a potential optimization to explore further were we to improve our project.

3. Results and Data Analysis

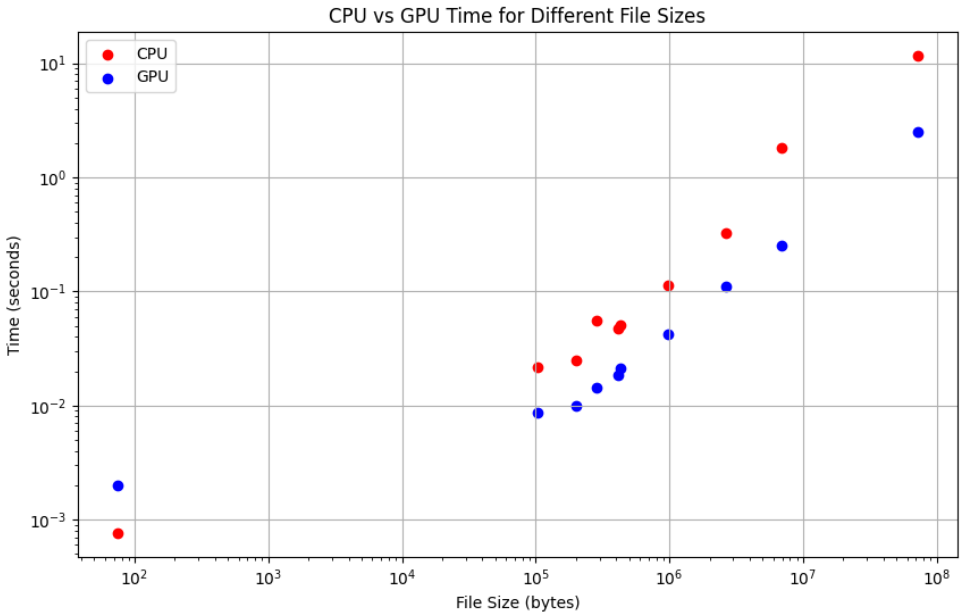

FIGURE 2 – Results comparison between CPU and GPU on the same images. CPU = Only CPU used, no parallelization ; GPU = CPU and GPU used, parallelization using CUDA

As we can see in Figure 2, the GPU version is always better by a small margin compared to the CPU version. The only image where the CPU version was better was the smallest one since the overhead caused by the use of CUDA was bigger than just simply encoding with the CPU on 64 pixels. We can also observe that the bigger the image is, the bigger the margin becomes. This way, the more data there is to parallelize, the more it will overcome the overhead of using CUDA.

Overall, we can also see that if n is the size of the image, the time complexity is still O(n) as we can't fully parallelize the process since we have to write sequentially each block at a time in the outpile jpg file.

Although it's not shown in the graph, it's also important to take into account the memory used in both methods. The CPU version's space complexity is O(1) and as for the GPU version, it's O(n) (depending on much space can be allocated on the machine). For a better comparison, it would have been better to compare both

versions on the same idea since the CPU was focused on a memory efficient encoder than the GPU one where it was focused on a time efficient one.

4. Missing Features

— **Downsampling** : One feature that we didn't have the time to implement is downsampling, which is a compression technique consisting in reducing the number of values "samples" for certain image components. Either way, it would not have been something we could have parallelized because it is part of the partitioning of the original image data but we mention it to indicate that our project version does not support this feature.

5. Conclusion

The primary objective of this assignment was to leverage CUDA and its parallel computing capabilities, which we applied in our case to optimize the JPEG encoding process. Specifically, the goal we set ourselves was to parallelize the various algorithms involved in image compression and filtering. This involved parallelizing both the processing steps within individual image blocks and the processing across multiple image blocks simultaneously, ultimately aiming to enhance the efficiency of the encoding process.

The exercise definitely proved successful in fulfilling its intended purpose. By harnessing the parallel processing power of CUDA, we were able to significantly accelerate the JPEG encoding process. Parallelizing the compression and filtering algorithms not only within a single block but across multiple image blocks concurrently was definitely a game changer ! As a result, the overall encoding process was optimized, leading to improved performance and efficiency compared to the sequential approach.

The reasons behind this success are :

— By parallelizing the processing steps within individual image blocks, we were able to exploit parallelism at a finer granularity, maximizing throughput and reducing processing time.
— Parallelizing processing across multiple image blocks concurrently further enhanced efficiency by enabling simultaneous encoding of multiple data blocks, and leveraging greater GPU resources and optimizing overall throughput.

Based on our results, it's evident that GPU acceleration provides substantial performance benefits for image encoding tasks. Throughout our analysis, the GPU encoder consistently outperformed its CPU counterpart across various image sizes. However, an interesting observation emerged with smaller images, where the CPU encoder exhibited slightly better performance due to the overhead associated with CUDA utilization. Despite this initial setback, as the size of the images increased, the advantage of the GPU encoder became more pronounced. This trend underscores

the effectiveness of GPU parallel processing capabilities, which effectively offset the overhead associated with CUDA utilization, particularly with larger datasets.

Additionally, our study highlighted the importance of warming up the GPU in benchmark tests. We observed that the initial execution on the GPU was significantly slower compared to subsequent runs. This phenomenon underscores the necessity of warming up the GPU before conducting performance evaluations, ensuring more accurate benchmark results.

In general, our findings highlight the significant advantages of leveraging GPU acceleration for image encoding processes. The superior performance observed with the GPU encoder, especially with larger datasets, demonstrates its potential to enhance the efficiency and speed of image encoding tasks. By harnessing the parallel processing power of GPUs, organizations and researchers can achieve faster throughput and improved performance in various image processing applications. Overall, our study emphasizes the importance of considering GPU acceleration as a valuable tool for optimizing image processing workflows and achieving high-performance computing goals.