Lab 1

Parallel programming

Paula Gómez
2022 - 2023

Useful information for lab1 and lab2

The labs 1 and 2 are both based on a very simple algorithm which consists in transforming the color space of a 4K image.

The goal is to focus on the hardware and compiler related issues that affect the performance of applications, instead of trying to parallelize a complex algorithm.

In lab 1 you will have to parallelize in OpenMP a very simple code in C++, where you will have to work on the concepts like: compiler flags, memory alignment and data locality, and OpenMP parallelization.

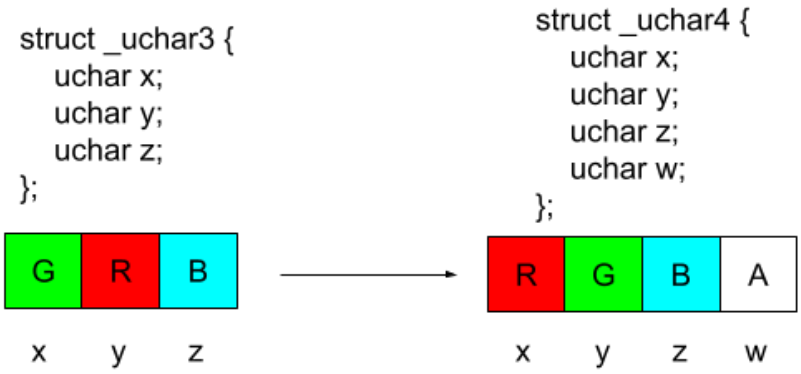
In lab 2, the same code (or slightly modified) will have to be parallelized in CUDA.

In both cases, the goal is to accelerate the transformation of the color space. Therefore, we not only will ask for an implementation, but also a lab report where you should show your performance results with tables and graphs. Some elaboration on why do you think you got the results you got, will be appreciated.

Color space conversion

The color space of an image, is the way in which the pixel information is distributed in memory. In this case, we are going to transform an image in the color space grb (green, red, blue), to an image in the color space rgba (red, green, blue, alpha).

The grb color space is composed by 3 channels, 8bit each. The number of bits per channel is called color depth. So these images have 8bit color depth.



To store the 8bits of a channel, we will be using the type “unsigned char” or “uchar”. To store all the pixel channels, we use structs. “struct uchar3” for grb and “struct uchar4” for rgba.

The “a” channel in rgba is called “alpha channel”, and it is used to specify the level of opacity or transparency of the pixel.

LAB 1: openMP

We recommend that you use any PC at your disposal, with Linux. Alternatively, you can use Google Colab, but there the optimizations seem to not take any effect there, so it makes it more difficult to learn and stay motivated.
Deliver the code with the original **convertGRB2RGBA** function included.

1. Explain **in your own words** what is openMP (definition, use, goals, etc.)
2. Compile and execute the code as it is
    a. Compile it with: make
    b. Execute: ./main
    c. Annotate the time in microseconds “us”
    d. Have a look at the code, and try to understand it.

3. Modify the Makefile, and experiment with the compiler flags -O, -O2, -O3 and -Ofast.
    a. Look for some documentation about these flags on the internet.
    b. Is there any risk when using those flags?
    c. Using -Ofast, compileu: make
    d. Execute: ./main
    e. Annotate the time and compute the Speedup compared to section 1. Give some reasoning about what may be happening, according to the flags documentation.

4. Using -Ofast flag:
    a. Create a convertGRB2RGBA_2 function.
    b. This function has to improve the data locality of the original version. Explain the changes and the obtained speedups.
    c. If the times obtained are lower than 1s, increase EXPERIMENT_ITERATIONS to 1000

5. Keeping the optimizations in section 3:
    a. Which problems may “struct uchar3” pose regarding memory alignment?
    b. Change uchar3 definition by this one:

```
struct _uchar3 {
   uchar x;
   uchar y;
   uchar z;
}__attribute__((aligned (4)));
```

    c. Execute again and measure times.
    d. Google what **__attribute__** is and what does aligned(4) do. Explain in the report what did you understand.

6. Keeping the previous optimizations, create a function “convertGRB2RGBA_3” and parallelize it with OpenMP:
    a. Which loop is better to parallelize?
    b. Which scheduling is the best? Make some tables and/or graphs with results, modifying the size of the image,the number of iterations, and the chunk size. Give some justified reasoning.
    c. What happens if you define the variables as private or shared?

7. Parallelize the loop that iterates over EXPERIMENT_ITERATIONS, executing the function "**convertGRB2RGBA_2**". How does it compare to section 5? Why do youthink it happens?
8. Print the thread id of each thread, only once for each thread, by using the OpenMP functions that provide the id. Use **std::cout** and make sure that the code is thread safe using OpenMP pragmas.
9. **CREATE A FULL REPORT** showing the information asked in each section. If you just deliver the code **YOU WILL NOT PASS** this lab even it is the most optimized code I have ever seen.

The delivery date for LAB1 is March the 22nd.