Practice 1: Color space conversion

Exercice 1

The code provided to us converts the color spaces of an image with 3 channels to 4 channels. In other words, we convert a 3-channel image to a 4-channel image where the opacity of the 4-channel image will be maximum (255). Once the data structures have been created and the necessary memory has been allocated, we calculate the time required to convert from BRG to RGBA in a number of iterations determined by EXPERIMENT_ITERATIONS. Finally we check if the result is correct or not and the total time required is printed.

Time (us)	21706386	22482205	21475137	21474215	22297426

Figure 1. Table with times in microseconds of program execution with default Makefile values. A total of five samples have been taken.

Exercice 2

With these flag options we control various types of optimization. With one of these optimization indicators we can improve the performance of our program, consequently the compilation code will take a little longer than usual and the level of memory used will grow as we increase the level of optimization.

- -O, the compiler tries to reduce code size and execution time, without performing optimizations that require a large amount of compile time.

- -O2, gcc performs almost all supported optimizations that do not involve a compromise between speed and space. Compared to -O, this option increases both compile time and performance of generated code.

- -O3, it tries to optimize the code heavily for performance. It includes all the optimizations that -O2 includes, and a few more.

- -Ofast, enables a higher level of optimization than -O3. It ignores strict adherence to standards by enabling optimizations that are not valid for all standards-compliant programs.

Although the execution time improves as we increase the optimization option, we must be careful because they are also dangerous. This means that the compilation time will be much slower because it will be using more memory or processor resources. Therefore, the ideal is to find a balance that favors us between compilation and execution time. Also, we must be careful because from -O2 we are likely to lose some packages at compile time.

Flag	-O	-O2	-O3	-Ofast
Time (us)	22109931	21868729	1317703	1321228
Time (us)	21476471	22840424	1326788	1328251
Time (us)	22519134	22535671	1334133	1319486
Time (us)	23855909	23640994	1321837	1320432
Time (us)	23858228	23993680	1328410	1350892

Figure 2. Table with the times in microseconds of each execution of the program with different optimization flags. A total of five samples have been taken for each one.

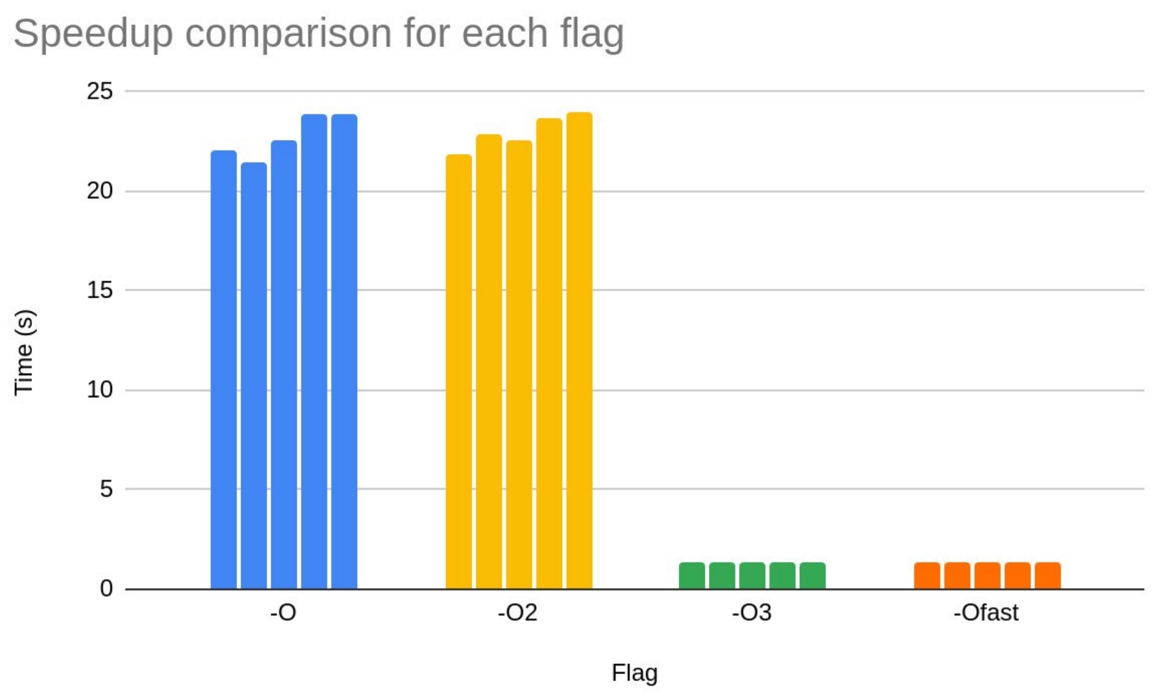

Figure 3. Bar graph showing the time in seconds taken for each execution with its respective optimization flag.

Exercice 3

The creation of the convertGRB2RGBA_2 function can be done in different ways. I will explain two possible ways I have been able to implement to make code execution work faster. The first one, is to invert the loops in order to traverse the image in rows instead of columns. In this way we can improve the location of the data:

- On the one hand, we save memory accesses by quantifying the probability of being visited (memory accesses are more costly than cache accesses).
- On the other hand, we improve the probability of visiting nearby memory addresses. In other words, the next memory address is close to the current one.

I have done another alternative function called convertGRB2RGBA_2_optional using only one for. Next, we will check which one is faster. Since I haven't noticed any big changes with only -Ofast flag i will also show the results for -O3 flag and -O2 flag.

EXPERIMENT_ITERATIONS = 1000

	-ofast	-o3	-o2
convertGRB2RGBA	13140085	13083767	215025714
convertGRB2RGBA_2	13110974	13104944	15108855
convertGRB2RGBA_2_ optional	12959510	13000597	15068897

Figure 4. Time measured in microseconds. New measurements taken for convertGRB2RGBA, they are not reused from previous sections.

The results tell us that no matter what function we use, we obtain very similar results for both -Ofast and -O3. However, things change a lot with the -O2 flag. The two functions that have been created to improve the first one, makes the execution time to change radically, from more than 3 minutes with the original to about 15 seconds with the other two. Moreover, although the improvement is minimal, it takes less time to the function with "one for" than the one with the "2 fors exchanged".

Exercice 4

Uchar3 can give us problems when accessing and saving data, because the way the memory is structured, the most efficient way is to access and segment in power of 2.

Time measured after applying the changes in microseconds. Using the function convertGRB2RGBA_2:

- For 100 iterations and -Ofast: 323968us.
- For 1000 iterations and -Ofast: 3116301us.

Before explaining what _attribute_ and aligned(4) is used for, we should understand how data structure alignment works. This defines how the data is arranged and accessed in computer memory. It consists of data alignment and data structure padding.

- Data alignment means putting the data at a memory offset equal to some multiple of the word size. This increases the system's performance due to the way the CPU handles memory.
- Data structure padding aligns the data, this means inserting if necessary some meaningless bytes (padding) between the end of the last data structure and the start of the next one.

***Note: This information is obtained from the wiki.*

So, gcc provides some functionalities in order to disable this data structure padding. Using the keyword _attribute_ we can specify special properties of variables, function parameters, etc. This keyword is followed by the attribute specification enclosed in double parentheses. The aligned attribute specifies a minimum alignment for the variable or structure field, measured in bytes. For example, in our case we have aligned(4) which causes the compiler to allocate the structure uchar3 on a 4-byte boundary. In simple words, it means that the struct should be aligned to an address that is divisible by 4.

Exercice 5

First of all, comment that worse results have been achieved parallelizing than without parallelizing when using the -Ofast flag. In order to see an improvement in performance using the parallelization of the OpenMP library, we must go down to the -O2 flag, which takes a time of around 9 and a half seconds using parallelization and about 18 seconds without paralyzing for 1000 iterations. Below we can see a table with the different results obtained.

Parallelizing	**-O2**	**-Ofast**
Parallelizing	9310275	5678983
Not parallelizing	18321812	3106793

Figure 5. Time measured in microseconds and 1000 iterations have been used for the experiment . Showing only the times for -O2 and -Ofast flags in order to show the time difference given the use of the flags.

So parallelizing only gives us a better execution time performance with a lower flag, but we lose when using the -Ofast (same happens with the -O3 flag). I personally think that this is because of the cache contention and the danger of using the flags above -O2 (consume too much memory and resources). I think that if a part of the array is accessed by some threads it will be cached several times, in other words it will be copied various times, one for each core. Therefore, if the data has been changed, it will need to check the latest version from others core cache (so I believe that this takes some time, like a time penalty).

Using the variables as shared or private doesn't help us very much, sometimes I get times below 9 seconds but rarely. The only thing that I can say is that we can't define the variables beside the x and y (the ones we use for looping) as private since the threads need to read the original values of these variables (grb, width and height). Of course this also applies for the rgba since the threads are writing the data on it. However, we can define them as firstprivate in order to get the original values and not a random one. So in short, we can define the x and y variables as private and the other ones as shared or firstprivate, but at least I couldn't see any difference in time execution performance so letting all as default will give us the same result.

We have different types of schedules that we can choose from, those are static, dynamic, guided, auto and runtime.

- Static, the iterations are divided into chunks of chunk size and the chunks are distributed into threads in a circular order. If no chunk size is specified, iterations are divided into chunks of roughly the same size and at most one chunk is distributed to each thread.

- Dynamic, splits the iterations into chunk-sized chunks. Each thread runs a chunk of iterations and then requests another chunk until there are no more chunks available. The default value is one if we don't specify the chunk size.
- Guided, is similar to the dynamic type. Iterations are splitted into chunks, and then each thread runs a chunk of iterations. When the thread has finished, requests another chunk until there are no more chunks available. The size of a chunk is proportional to the number of unassigned iterations divided by the number of the threads.
- Auto, we delegate the decision to the compiler or runtime system to take the scheduling decision.
- Runtime, tells to set the schedule using the environment variable. This environment variable can be set to any other scheduling type (static, dynamic or guide).

After testing each of these configurations we have obtained the following results that are shown in the table below. As we can see in the table, the execution times are very similar and none stands out for being better or worse than the others in this case.

	-Ofast	**-O2**
Static	5738832	8930989
Dynamic	5726252	9187003
Guided	5742622	8869647
Auto	5764021	8917540
Runtime	5819330	8918647

Figure 6. Time measured in microseconds and 1000 iterations have been used for the experiment . Showing only the times for -O2 and -Ofast flags in order to show the time difference given the use of the flags. The number of chunks used is 5 since the same results have been reached regardless of this value.

Exercice 6

Using parallelization in the loop where the convertGRB2RGBA_2 function is called is better because in this way the distribution of the tasks of the threads is more organized and efficient. As we can see in the table, although not by much, the execution times improve for both flags.

-Ofast	**-O2**
4787764	8643574

Figure 7. Time measured in microseconds and 1000 iterations have been used for the experiment . Showing only the times for -O2 and -Ofast flags in order to show the time difference given the use of the flags.

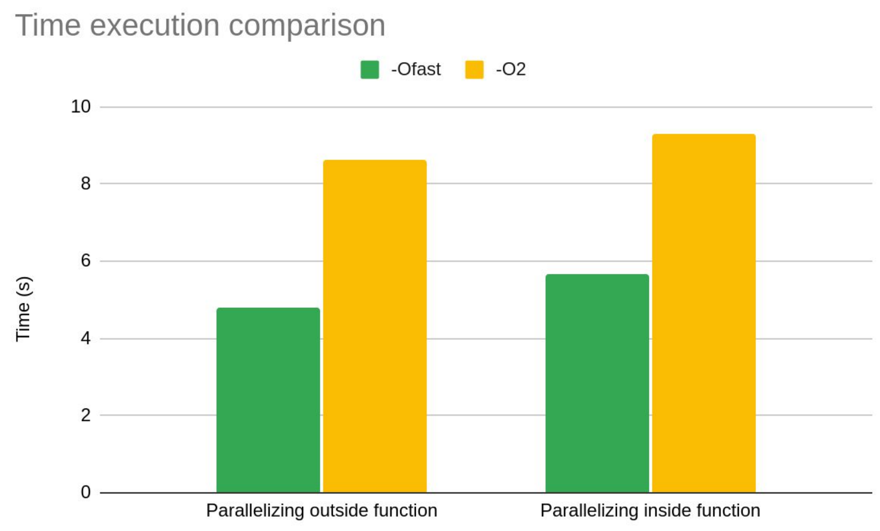

Figure 8. Comparing section 5 where we have parallelized inside the function (right) with the time execution we got in this section (left). We can see that we got an improvement for both flags.

Exercice 7

To print the id of each thread we must define the zone as critical since we want each thread to execute it but only one at a time. The execution time is not affected by this implementation, and in my case a total of four threads are printed. In general, OpenMP uses as many threads as the cores our machine has.