PART 1:

-- original makefile: 10650133us

-- Code creates an GRB array, where every 1/3 of its height

PART 2:

-> These flags are for optimization and offer different levels of peformance improvement.

-> Some of these optimizations rely on undefined behaviour, and when exposed to a code that has not been fully tested, can introduce bugs. Also -Ofast enables optimizations that are not valid for all standard compliant programs, so extra care must be taken.

-> Execution times with flags:

---- with -O flag: 6262940us

---- with -O2 flag: 6383179us

---- with -O3 flag: 805382us

---- with -Ofast flag: 810291us

-> COMPARISON (Original vs -Ofast):

---- ORIGINAL: 10650133us

---- OPTIMIZ.: 810291us

-92.39% over the non optimized code

-> (-Ofast) enables all -O3 optimizations as well as another ones that aren't IEEE compliant such as -ffast-math (which changes x=x\*x\*x\*x; for x\*=x;x\*=x), -fno-rounding-math (uses default rounding mode) and -fallow-store-data-races (the name says it all) among others.

PART 3:

-> The pixels are stored in an array, which in memory are saved next to each other. The original convertGRB2RGBA access first with the 'Y' coordinate. This means that it is accessing the array every Y items, skipping elements that are close in memory and not being optimizied.

-- EXAMPLE (WIDTH=2):

---- access: a[0] -- SKIPS a[1] -- access a[2] -- SKIPS a[3] -- and so on

-> By moving the 'Y' coordinate to the outer loop, we'll be accessing the elements of the array consecutevely without skipping anyone.

-- FOR 100 iterations the time is: 808609us (speed increased in -0.2%)

-- FOR 1000 iterations the time is: 7966568us.

PART 4:

-> The CPU does not access memory one byte at a time, instead accesses memory in 2^n sized chunks of memory. The reason for this is performance, accessing misaligned data will suppose extra CPU operations regarding byte shifting and combining so that the data can be retrieved.

-> In this case 'uchar3' has 3 chars (1 byte each, 3 bytes total). Since its not aligned with any power of 2, the performance of the function is slowed down.

-> Execution times with align(4) on uchar3:

--- 1000 iterations: 1689520us

--- 100 iterations: 175422us

-> According to the compiler docs, the aligned attribute will set a minimum byte alignment for the variable. By adding this attribute, the compiler will add a padding to the uchar3 structure so that it is 4-byte-aligned. In this case, its most likely that it will add one byte at the end of the uchar3 structure.

-> As expected, the CPU time is lower compared with the misaligned code. This is because the CPU can directly get the 4 bytes of data, instead of having to shift them to get 3 bytes.

PART 5:

->After performing the respective tests, we have been able to show that parallelizing the first loop gives better results compared to the second, this is because if we parallelize the second lop, what we are doing is creating and destroying threads in each of the iterations of the first loop, which is not very optimal

->In our case, we have obtained similar performances in the different tests carried out, so we cannot position ourselves on which scheduling is better or worse.

* static: assignment determined at the beginning (does not change while running)
* dynamic: assignment is carried out on-line
* guided: assignment is carried out on-line
* runtime: decision deferred to runtime (run-sched-var/OMP\_SCHEDULE)

#pragma omp parallel for schedule(static, chunk\_size)

#pragma omp parallel for schedule(dynamic, chunk\_size)

#pragma omp parallel for schedule(guided)

#pragma omp parallel for schedule(runtime)

|  |  |
| --- | --- |
| Scheduling (100 ITERATIONS) | Time (us) |
| Static 1 | 694059 |
| Static 5 | 702670 |
| Static 10 | 691784 |
| Dynamic 1 | 716381 |
| Dynamic 5 | 692066 |
| Dynamic 10 | 684185 |
| Guided | 699284 |
| Runtime | 691414 |

|  |  |
| --- | --- |
| Scheduling (1000 ITERATIONS) | Time (us) |
| Static 1 | 7051178 |
| Static 5 | 6946560 |
| Static 10 | 6942459 |
| Dynamic 1 | 7071090 |
| Dynamic 5 | 7065223 |
| Dynamic 10 | 6914113 |
| Guided | 7026037 |
| Runtime | 6807883 |

-> In our case, if we use private variables for width or height it does not work, because the same value will be used in each pixel of the image.

PART 6:

->484945us

->Compared to section 5, the execution of this is done in less time. This is because the necessary threads are created from the beginning, before calling the convertGRB2RGBA function, and they are destroyed at the end of the loop execution. In section 5, what was done was to create and destroy threads in each iteration of the height variable of the image

PART 7:

