Name: Do Le Tuan Minh

Student Number: 50366862

Email: le.do@tuni.fi

COMP.CE.350 Multicore and GPU Programming Lab Work Autumn 2021

Part 1

6.1 Benchmarking the original Code and Improving Performance via Compiler Settings

1) Without optimization, the average satellite moving was 154.8 ms, space coloring was 1701.8 ms, total frametime was 1864.8 ms.

(gcc -o parallel parallel.c -std=c99 -lglut -lGL -lm)

2) Using most optimizations, the average satellite moving was 79.2 ms, space coloring was 586.8 ms, total frametime was 671.2 ms.

(gcc -o parallel parallel.c -std=c99 -lglut -lGL -lm -O2)

3) using also vectorization optimization in the compiler(-ftree-vectorize), the average satellite moving was 82.2 ms, space coloring was 589.2 ms, total frametime was 676.6 ms.

(gcc -o parallel parallel.c -std=c99 -lglut -lGL -lm -O2 -ftree-vectorize)

When enabling fopt-info-vec flag, the compiler told:

parallel.c:137:4: note: basic block vectorized

parallel.c:347:4: note: basic block vectorized

line 137 is the first loop in parallelPhysicsEngine() function, line 347 is the first loop in sequentialPhysicsEngine() function.

4) Using SIMD instruction set and FP relaxation related optimization flags, the best performance was combination of -ffast-math and -mavx

ffast-math is a floating point optimization, allowing the reordering of instructions to something which is mathematically the same (ideally) but not exactly the same in floating point, *disables* setting errno after single-instruction math functions, which means avoiding a write to a thread-local variable, no checks for NaN (or zero) are made in place.

avx is Advanced Vector Extensions, using YMM registers to perform SIMD, hold and perform operations of eight 32-bit single-precision floating point numbers or four 64-bit double-precision floating point numbers.

Average performance of the optimization: total frametime: 420.83 ms, satellite moving: 21.7 ms, space coloring: 395 ms.

One optimization caused the broken code to be generated was avc512f. The problem was illegal instruction (core dumped). The problem might be the Compiler targets a CPU that is newer/different than the current CPU and it then chooses to emit instructions that current CPU can't execute.
(gcc -o parallel parallel.c -std=c99 -lglut -lGL -lm -O2 -ftree-vectorize
-fopt-info-vec -ffast-math -mavx)

6.2 Generic algorithm optimization

Unnecessary calculation is in *//Second graphics loop: Calculate the color based on distance to every satellite* section, inside the if (!hitsSatellite), since the difference , dist and weight was already calculated in the *// First Graphics satellite loop: Find the closest satellite.* section. Solution was to created 3 temporary variable of red, green and blue as float, init them as 0.0f, then their value are satellites[j].identifier.<color>*weight. The calculation of these 3 temporary value should be put in a loop(*// First Graphics satellite loop: Find the closest satellite*) to be benefit from parallelization with OpenMP later. finally the final calculation is renderColor.<color> += <color>_temp/weights * 3.0f;

6.3 Code analysis for multi-thread parallelization

All of the loops mention can be parallelized for multi threads. When using parallelized using gcc, only *Physics satellite* was parallelized.

However, the loop *Physics iteration* will not get benefit from being parallelized because its iteration is not used inside the loop, only served as counter, only the iteration of *Physics satellite* was used.

The way to change the code which helps benefit from parallelized is to switch place between *Physics iteration* and *Physics satellite* loop. In this way the *Physics satellite* will nest *Physics iteration* and parallelize *Physics satellite also applies to Physics iteration.*

The performance of the modified code was a little bit worse than the unmodified version, with higher total frametime, significantly higher satellite moving and similar space coloring. The code modification did not have immediate benefit for the code performance. The Code modification will gain benefit when it is parallelized with OpenMp later. (Optimization used in GCC was gcc -o parallel parallel.c -std=c99 -lglut -lGL -lm -O2 -ftree-vectorize -fopt-info-vec -ffast-math -mavx)

6.4 OpenMP Parallelization

Total framerate was 257.8 ms , satellite moving of 81.5 ms, space coloring of 166 ms in average. (Optimization used in GCC was gcc -o parallel parallel.c -std=c99 -lglut -lGL -lm -O2 -ftree-vectorize -fopt-info-vec -ffast-math -mavx -fopenmp)

The code did not have any changes apart from the mention changes in the previous sections and using pragma.

The loops were parallelized with pragma were:

- 2 first loop of each *ParallelPhysicsEngine* and *ParallelGraphicsEngine* functions with omp simd.
- *Physics satellite and Graphics pixel* with omp for
- *First graphic satellite* loop with omp reduction for its 3 += operators

There were no break or slowdown detected.

The performance scale with number of cores. I did not have Linux machine so I used a virtual machine running Ubuntu. The result previously was using default 2 core. Here is the table of average performance contrast with number of cores:

Core num.	Total frametime (ms)	Satellite moving (ms)	Space coloring (ms)
1	507.3	151.17	339.3
2	257.8	81.5	166
3	166.3	56.83	104.5

I used my own computer for this exercise. Its specialization is Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz. Version of GCC is 7.5.0.