Tampere University

COMP.CE.350 Multicore and GPU Programming Project Work
Autumn 2025
Tapio Nevalainen
November, 2025

Project: Part 1 CPU parallelization:

Group Members:

Syed Muhammad Hassan Abidi, 153049316, hassan.abidi@tuni.fi

Jingjing Yang, 154016843, jingjing.yang@tuni.fi

Table of Results:

Collect all your measurements in a table:

Configuration	**Physics (ms)**	**Graphics (ms)**	**Total (ms)**	**Speedup vs Debug**
Debug (no optimization) **Windows**	**172**	**1281**	**1515**	**1.00x**
Release **(parallel PRIVATE /fp:fast) Windows**	**80**	**343**	**428**	**3.50x**
Debug (no optimization)	**70**	**1829**	**1903**	**1.00x**
Release (-O3 -march=native -ffast-math)	**7**	**253**	**263**	**7.24x**
Release (optimized code structure) (bonus)	**7**	**208**	**217**	**8.77×**
				Speedup vs no OpenMP
OpenMP (1 threads)	**7**	**203**	**212**	**1.02x**
OpenMP (2 threads)	**8**	**104**	**113**	**1.92x**
OpenMP (4 threads)	**7**	**53**	**62**	**3.50x**
OpenMP (6 threads)	**7**	**37**	**46**	**4.72x**
OpenMP (8 threads)	**7**	**30**	**39**	**5.56x**

Only first two options are windows rest all are Mac results.

MacOS:

Compilation Commands

```
# Debug build (baseline, single-threaded)

cmake -DCMAKE_BUILD_TYPE=Debug .. && make && ./parallel 1234


# Release with optimized flags (best performance, single-threaded)

cmake -DCMAKE_BUILD_TYPE=Release .. -DCMAKE_C_FLAGS="-O3 -march=native -
ffast-math" && make && ./parallel 1234
```

6.1 Benchmarking the original Code and Improving Performance via Compiler Settings

1. With the Original C Version without any compiler optimization flags (in Debug-mode)?

Ans: In Debug mode, on Windows without any optimizations the physics frame time is 172ms, graphics is 1281ms, and the total is 1515ms, whereas on Mac-Os the physics frame time is 70ms, graphics is 1829ms, and the total is 1903ms.

2. With the Original C version using most compiler optimizations (in Release-mode)?

Ans: In release mode, on Windows the physics frame time is 119ms, graphics is 559ms, and the total is 680ms, whereas on MacOs the physics frame time is 7ms, graphics is 253ms, and the total is 263ms. but on MacOs the flags **-O3 -march=native -ffast-math** were used. In windows Flags were tested later on in step 4 for comparison reasons.

3. Did the compiler tell it managed to vectorize any of the loops in physicsEngine or graphicsEngine-functions? If the loops of ParallelPhysicsEngine and ParallelGraphicsEngine-functions failed to vectorize, report the reason.

Ans: As mentioned in report questions, Using TC303 Pc and QVEC-report:2, the compiler failed to vectorize and provided the following reasons.
loop not vectorized due to reason '500/1300/1104/1203'
Probable Reasons:
In the loops there are possible function calls which aren't suitable for SIMD (error=1300), Similarly compiler might've detected data dependencies between iterations (error=1104), the error 500 represented that loops contain conditional logic and lastly 1203 representing non uniform write operations on memory. Not being able to vectorize can also be possible due to admin rights on the lab computers as I tried to change some settings but it wouldn't allow.
Logically: The Physics and Graphics loops are using arrays of data with multiple pointer accesses hence achieving no data dependencies between iterations might not be possible and to avoid any data overlaps or corruption compiler wouldn't allow vectorizations.
In MacOs:
The compiler was able to perform vectorization.
3 loops were vectorized but interleaving was not beneficial indicated by the cost model.

- Physics initialization loop (vectorization width: 2, interleaved count: 1)
    - A simple data copy loop (64 iterations, 4 assignments per iteration).

- Physics satellite loop (vectorization width: 2, interleaved count: 1)
  - Updating the satellite’s position and velocity based on gravity is independent.
- Second graphics loop (vectorization width: 4, interleaved count: 1)
  - Computing color contribution of each satellite to the pixel is independent.

2 not vectorized loops:

- Physics update loop
  - “the cost-model indicates that interleaving is not beneficial loops”
  - “the cost-model indicates that vectorization is not beneficial loop”
- “First Graphics satellite loop”
  - “value that could not be identified as reduction is used outside the loop”
  - “could not determine number of loop iterations” (keyword ”break” effect)

4. Experiment with the SIMD instruction set and FP relaxation related optimization flags. Which are the compilation flags you found to give the best performance? Document these well, since you should probably use these ones for the rest of project work.
Windows:

Flags	Physics (ms)	Graphics (ms)	Total (ms)
O2	119	557	719
03	121	565	705
/fp:fast	80	343	428
/arch:AVX2	120	567	691
/Qpar	121	575	699
/favor:INTEL64	120	570	700
/arch:AVX512	-	-	Broke code
/fp:fast/favor:INTEL64	123	569	710

On MacOS:

Compiler Flags	Physics (ms)	Graphics (ms)	Total (ms)	Speed vs Debug
-O0 -g(Debug)	70	1829	1903	1.0×
-O1	14	337	353	5.4×
-O2	14	336	352	5.4×
-O3	14	336	352	5.4×
-O3 -march=native	14	337	353	5.4×
-fno-tree-vectorize	14	339	356	5.3×
-fstrict-aliasing	14	337	354	5.4×
-flto	14	336	353	5.4×
-frounding-math	22	416	440	4.3×
-ffast-math	7	254	263	7.2×
-O3 -ffast-math	7	253	263	7.2×
-O3 -march=native -ffast-math	7	253	263	7.2×
-O3 -march=native -ffast-math -funroll-loops	7	268	278	6.8×

5. Can you explain, what each of the optimization flags you found to give the best performance does?

Windows:

Using windows the best flag found was /fp:fast as it gave the fastest latency of 428ms and also the difference was observed visibly. This is because Fpfast allows aggressive floating point operations reducing unnecessary precision, drastically speeding up the entire process. It has a few cons aswell, although which didn't impact our working and the provided results, the downsides include risk of low accuracy and rounding errors. The other flags we used mentioned in the table as well, didn't provide better results and even the normal release mode without those flags ha worked better or quite close hence only mentioning the best flag Fpfast here.

MacOS:
In MacOS the best flag was ffast-math, which was tried with multiple combinations and individually, but overall ffast-math was the best flag as it provided a latency of 263ms even when tried with combinations only ffast-math combinations gave the best results. What this flag does is it allows relaxed floating point operations, allows compiler to reorder floating point operations and perform whichever ones increase speed first and enables reciprocal approximations. Also has a similar downside in precision as fpfast but works in our scenario.

6. Did you find some compiler flags which cause broken code to be generated, and if so, can you think why?
Windows:
Yes, using the flag **/arch:AVX512,** our code broke and showed unhandled exceptions in parallel c file. The reason was that AVX-512 tells the compiler to use 512 bit vector registers which are present mostly in core I9s or latest gen processors.

MacOs:
No code breakage was seen in any flag usage either independant or inidividually used flags.

6.2 Generic algorithm optimization

Let's still stay inside one thread during this question. Can you find any ways to change the code to either get rid of unnecessary calculations, or allow the compiler to vectorize it better, to make it faster. If yes, what did you do and what is the performance with your optimized version?
Ans:

In parallelGraphicsEngine(), the sqrt function was called twice per pixel for distance comparison, which is computationally expensive. We replaced square root comparisons with squared value comparisons, which are mathematically equivalent but the later one is much simpler.

We also moved constant expressions SATELLITE_RADIUS * SATELLITE_RADIUS and BLACK_HOLE_RADIUS * BLACK_HOLE_RADIUS outside the loop to reduce redundant multiplications.

Another change is that the original "Graphics pixel loop" iterated over all SIZE = WINDOW_WIDTH * WINDOW_HEIGHT pixels as a flattened 1D array. But i % WINDOW_WIDTH and i / WINDOW_WIDTH are expensive operations. We transformed it into a 2D array.
Performance result:

Configuration	Physics (ms)	Graphics (ms)	Total (ms)	Speedup vs Debug
Debug (no optimization)	70	1829	1903	1.0×
Release (-O3 -march=native -ffast-math)	7	253	263	7.24×
Release (-O3 -march=native -ffast-math)(optimized code structure)	7	208	217	8.77×

These algorithm optimizations reduce unnecessary work and arithmetic complexity, resulting in an additional speedup from 7.24× to 8.77×. Graphics rendering improved from 253 ms to 208 ms over the debug baseline.

6.3 Code analysis for multi-thread parallelization

1) **Which of the loops are allowed to be parallelized to multiple threads?**
   Ans: In the "Physics satellite loop," each iteration reads and writes only its own “tmpVelocity” and “tmpPosition”. Each iteration is data independent, so the loop can be safely parallelized. And we need to move the "i" declaration inside the for statement on macOS.
   In the "Graphics pixel loop," each iteration only writes to its own pixels[i] and reads shared satellites[]. Each pixel calculation is independent, so this loop is safe to parallelize. Similarly, we need to move the "i" declaration inside the for statement.
   In the “Physics iteration loop”, each iteration uses positions and velocities computed by the previous iteration, creating a loop-carried dependency. This loop cannot be parallelized.
   The "First Graphics satellite loop" has a break statement that blocks SIMD vectorization, making it unsafe to parallelize.

2) **Are there loops which are allowed to be parallelized to multiple threads, but which do not benefit from parallelization to multiple threads? If yes, which and why?**
   Ans: The "Physics satellite loop" is not beneficial to parallelize.
   Performance slowdown significantly after parallelizing this loop, from around 40 ms to around 1900 ms(8 cores):

```
  // Physics iteration loop
  #pragma omp parallel
  for(int physicsUpdateIndex = 0;
     physicsUpdateIndex < PHYSICSUPDATESPERFRAME;
    ++physicsUpdateIndex){

     // Physics satellite loop
    #pragma omp for schedule(static)
    for(int i = 0; i < SATELLITE_COUNT; ++i){
      ...
    }
```

Even when we inverted the loop order, performance slowed from around 40 ms to around 140 ms (8 cores).

```
// Physics iteration loop
#pragma omp parallel for schedule(static)
 for(int i = 0; i < SATELLITE_COUNT; ++i)
          { // Physics satellite loop
          for(int physicsUpdateIndex = 0;
          physicsUpdateIndex < PHYSICSUPDATESPERFRAME;
          ++physicsUpdateIndex){
          ... }
          }
```

In the "Physics satellite loop", there are only SATELLITE_COUNT=64 iterations per time step and each iteration does little work. The multi-threading overhead would be greater than the performance benefits, resulting in a slowdown. SIMD vectorization fits these loops better.

3) **Can you transform the code in some way (change the code without affecting the end results) which either allows parallelization of a loop which originally was not parallelizable, or makes a loop which originally was not initially beneficial to parallelize with OpenMP beneficial to parallelize with OpenMP? If yes, explain your code transformation? Does your code transformation have any effect on vectorization performed by the compiler?**

Ans: The "Second graphics loop" can be parallelized using reduction, since each iteration only reads from satellites[j]/satellites[k] and accumulates values in weights/renderColor. To use reduction, we need to declare scalars.

Code changes:

In the second graphics loop we defined new scalars and moved the loop counter variable declaration into the loop header line. To enable SIMD parallelization with reduction, we first declared scalar accumulators by creating separate variables rr, rg, and rb to accumulate color values instead of directly modifying renderColor inside the loop. We then added the OpenMP SIMD directive #pragma omp simd reduction(+:rr,rg,rb) to enable vectorization with reduction operations. Finally, after the loop completes, the accumulated values are applied to renderColor with the division by weights performed once outside the loop.

Effect on vectorization:

The "Second graphics loop" was already vectorized before adding the simd reduction directive, so vectorization remained unchanged. But the way the compiler vectorizes this loop may have changed, because we observed a small speedup after this modification. Although the improvement is minimal, we decided to keep this change.

6.4 OpenMP Parallelization

Compilation Commands

```
# Uncommented OpenMP lines in CMakeLists.txt

 cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_C_FLAGS="-O3 -march=native -ffast-math" .. && make

# Run with 1 thread

OMP_NUM_THREADS=1 ./parallel 1234

# Run with 2 threads

OMP_NUM_THREADS=2 ./parallel 1234

# Run with 4 threads

OMP_NUM_THREADS=4 ./parallel 1234

# Run with 6 threads

OMP_NUM_THREADS=6 ./parallel 1234

# Run with 8 threads

OMP_NUM_THREADS=8 ./parallel 1234
```

1) **What are the average frame times (milliseconds) in your OpenMP-multithreaded version of the code?**

Configuration	**Physics (ms)**	**Graphics (ms)**	**Total (ms)**	**Speedup vs no OpenMP**
Release (no OpenMP)	**7**	**208**	**217**	**1.00x**
1 thread	**7**	**203**	**212**	**1.02x**
2 threads	**8**	**104**	**113**	**1.92x**
4 threads	**7**	**53**	**62**	**3.50x**
6 threads	**7**	**37**	**46**	**4.72x**
8 threads	**7**	**30**	**39**	**5.56x**

2) **Did you perform some extra code transformations or optimizations?**
Ans: Extra optimizations: Used schedule(static) to reduce scheduling overhead.

3) **Which loops did you parallelize?**
Ans: We parallelized the "Graphics pixel loop" and "Second graphics loop" after testing multiple loops.
The "Graphics pixel loop" uses task parallelism (OpenMP parallel for), while the "Second graphics loop" uses data-level parallelism (OpenMP simd reduction).

4) **Did parallelization of some loop break something or cause a slowdown? Try to explain why?**
Ans: All parallel runs executed without issues and passed error checks.
Parallelizing the "Physics satellite loop" caused major slowdowns, so we did not include it in our final code version.
In the "Physics satellite loop", there are only SATELLITE_COUNT=64 iterations per time step and each iteration does little work. The multi-threading overhead would be greater than the performance benefits, resulting in a slowdown.

5) **Did the performance scale with the amount of CPU cores or native CPU threads (if you have an AMD Ryzen, or Intel core i7 or i3 CPU, cores may be multi-threaded and can execute two threads simultaneously)? If not, why?**
Ans: Performance scaled well up to 8 threads but not linearly (see the table above column Speedup vs no OpenMP).
This is due to synchronization and thread management overhead.

6) **Did you use TC303 computers or your own computer? If your own computer, what is the brand and model number of your CPU? and version of your compiler?**
Ans: We tested the non-OpenMP version on two devices: one on our own MacBook M1 Pro (compiler: AppleClang 15.0.0) and another on a Windows PC in TC303. All open Mp parts were done only on Mac OS.

7) **Approximately how many hours did it take to complete this exercise part (Part 1)?**
Ans: Around 30 hours, including understanding the code structure, setting up the environment, testing various build configurations, studying vectorization options, learning OpenMP parallelization, collecting data, and writing the report.