

part 2 report

Group AD

Syed Muhammad Hassan Abidi, 153049316, hassan.abidi@tuni.fi
Jingjing Yang, 154016843, jingjing.yang@tuni.fi

Original performance

on Mac

on Windows

Reflection on the Part 1 feedback

Commands to run OpenMP on Windows

Platform difference

Performance comparison

OpenCL setup

Graphics OpenCL implementation

OpenCL commands and performance

--local argument

on Mac

on Windows

Performance Analysis Questions

Small Window Size Test (80x80)

Improvement

Compile OpenCL function with optimization flags

Add Physics openCL kernel

OpenMP + OpenCL implementation

Final Choice

Further ideas

Feedback

We submitted a zip file containing 4 folders. The folder "Satellites1kernel" is our final submission.

```
//////////  
// aa Physics on CPU with omp, Graphic renderingon GPU kernel aa //  
//  
// Change lines below to make different configuratin //  
// WINDOW_HEIGHT 1024, WINDOW_WIDTH 1920 → 80 80 //  
// Null → "-cl-fast-relaxed-math -cl-mad-enable" //  
// #pragma omp parallel for schedule(static) //  
// //  
// this version is for windows, if run on mac, //
```

```
// please use static_assert instead of _Static_assert      //
```

Original performance

This version is without either OpenMP or OpenCL, just some algorithm optimizations in `parallelGraphicsEngine()` as we explained in part 1 report.

These optimizations improve graphics rendering when running on CPU with multiple threads using OpenMP. However, for the OpenCL kernel, performance is already around 1ms even without these changes.

In `parallelGraphicsEngine()`, the `sqr` 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 cheaper.

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.

We also tried precomputing `const double dtStep = (double)DELTATIME / (double)PHYSICSUPDATESPERFRAME` in `parallelPhysicsEngine()`. This additional rounding still meets the precision standard, but it did not improve performance on either Mac or Windows.

on Mac

```
# 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 -DNDEBUG" && make && ./parallel 1234
```

Case (Mac, 1920*1024,	6-205 frames Average	6-205 frames Average	6-205 frames Total (ms)	Speedup vs Debug

no parallel)	Physics (ms)	Graphics (ms)		
CPU – Debug	70	1829	1903	1.0×
CPU – Release(fast flags)	7	253	263	7.24×
CPU – Release(flags+code optimized)	7	208	217	8.77×

Case (Mac, 80*80, no parallel)	6-205 frames Average Physics (ms)	6-205 frames Average Graphics (ms)	6-205 frames Total (ms)	Speedup vs Debug
CPU – Debug	70	5	76	1.0×
CPU – Release(fast flags)	7	1	8	9.5×
CPU – Release(flags+code optimized)	7	0	8	9.5×

on Windows

```
# Debug build (baseline, single-threaded)
cmake ..
cmake --build . --config Debug
./Debug/parallel.exe 1234
```

```
# Release with optimized flags (best performance, single-threaded)
cmake -DCMAKE_C_FLAGS="/O2 /fp:fast" ..
cmake --build . --config Release
./Release/parallel.exe 1234
```

Case (Windows, 1920*1024, no parallel)	10th frame Physics (ms)	10th frame Graphics (ms)	Total (ms)	Speedup vs Debug
CPU – Debug	164	1254	1420	1.0×
CPU – Release(fast flags)	77	325	403	3.52×
CPU – Release(flags+code optimized)	76	252	329	4.32×

optimized)

Case (Windows, 80*80, no parallel)	10th frame Physics (ms)	10th frame Graphics (ms)	Total (ms)	Speedup vs Debug
CPU – Debug	163	4	169	1.0x
CPU – Release(fast flags)	79	1	80	2.11x
CPU – Release(flags+code optimized)	77	1	78	2.17x

Reflection on the Part 1 feedback

Here is our reflection on the Part 1 feedback: "Your reasoning about the slowdown of the physics satellite loop when multithreaded is correct. However, it should improve if you swap the loops."

Commands to run OpenMP on Windows

(macOS commands are in report 1):

```
# Uncommented OpenMP lines in CMakeLists.txt
cmake -DCMAKE_C_FLAGS="/O2 /fp:fast" ..
cmake --build . --config Release

# Run with 1 thread
$env:OMP_NUM_THREADS=1
./Release/parallel 1234

# Run with 2 threads
$env:OMP_NUM_THREADS=2
./Release/parallel 1234

# Run with 4 threads
$env:OMP_NUM_THREADS=4
./Release/parallel 1234

# Run with 6 threads
$env:OMP_NUM_THREADS=6
./Release/parallel 1234
```

```
# Run with 8 threads
$env:OMP_NUM_THREADS=8
./Release/parallel 1234
```

Platform difference

One notable observation: on Mac, declaring the loop counter inside the for statement works fine, but on Windows, we have to declare it outside the statement.

Another thing is that `#pragma omp simd reduction` is not supported by MSVC compiler on Windows, so we commented out the `#pragma omp simd reduction(+:rr, rg, rb)` line, although it improved performance on macOS using Clang.

On Windows, the frame times increase as the program runs. The table below shows data from the **10th** frame.

Performance comparison

The results show that adding the physics satellite loop improved performance on Windows, as shown in the tables below. This inspired us to combine openMP and openCL in later implementation.

However, on macOS, parallelizing the "Graphics pixel loop" alone is fast enough. Adding the physics satellite loop caused a slowdown from around 40 ms to around 140 ms(8 cores).

Thread (Windows, 1920*1024, only graphic openMP)	Physics (ms)	Graphics (ms)	Total (ms)
1	82	250	344
2	77	136	214
4	77	65	143
6	77	50	129
8	77	37	115

Thread (Windows, 1920*1024, Physics openMP & graphic openMP)	Physics (ms)	Graphics (ms)	Total (ms)
1	93	259	354
2	49	131	181
4	25	66	92

6	19	46	66
8	13	53	67

OpenCL setup

OpenCL version is **1.2** on our Mac, while TC303 PC has version 3.0 (we submitted this in preReturn).

```
jingjingyang@dyn-179-28 PreReturnFiles % ls
CMakeLists.txt      VectorAdd.c
Instructions.txt    VectorAdd.cl
jingjingyang@dyn-179-28 PreReturnFiles % cc VectorAdd.c -framework OpenC
L -o VectorAdd
jingjingyang@dyn-179-28 PreReturnFiles % ls
CMakeLists.txt      VectorAdd          VectorAdd.cl
Instructions.txt    VectorAdd.c
jingjingyang@dyn-179-28 PreReturnFiles % ./VectorAdd
Platform 0 information:
  Profile: FULL_PROFILE
  Version: OpenCL 1.2 (Oct 11 2025 00:32:14)
  Name: Apple
  Vendor: Apple
  Extensions: cl_APPLE_SetMemObjectDestructor cl_APPLE_ContextLogging
  Functions cl_APPLE_clut cl_APPLE_query_kernel_names cl_APPLE_gl_sharing cl
_khr_gl_event
```

Using Platform 0.

Device 0 information:

```
  Vendor: Apple
  Name: Apple M1 Pro
  Version: OpenCL 1.2
```

Using Device 0.

Program file is: VectorAdd.cl

38 + 15 = 53 OK

57 + 55 = 112 OK

42 + 25 = 67 OK

49 + 8 = 57 OK

26 + 9 = 35 OK

0 + 7 = 7 OK

```
26 + 21 = 47 OK
32 + 60 = 92 OK
34 + 10 = 44 OK
0 + 10 = 10 OK
7 + 38 = 45 OK
58 + 57 = 115 OK
19 + 8 = 27 OK
17 + 55 = 72 OK
33 + 3 = 36 OK
0 + 20 = 20 OK
```

Graphics OpenCL implementation

First, we enabled OpenCL in the project by uncommenting the relevant lines in

`CMakeLists.txt`.

We used the `vectorAdd` example as a reference. We took the OpenCL setup code and the cleanup code, and organized them into two functions: `ocl_init()` and `ocl_destroy()`.

We copied helper functions like `clErrorString()`, `readSource()`, and `printPlatformInfo()` directly from the `vectorAdd` example. We also added a check to see if the device supports double precision numbers at the end of `printDeviceInfo()`.

```
cl_device_fp_config fp64 = 0;
status = clGetDeviceInfo(deviceIds[r], CL_DEVICE_DOUBLE_FP_CONFIG, sizeof(fp64), &fp64, NULL);
if (status == CL_SUCCESS && fp64 == 0) {
    printf("\tDouble precision: NOT supported\n");
} else if (status == CL_SUCCESS) {
    printf("\tDouble precision: supported\n");
}
```

```
typedef struct {
    int32_t width;           // Window width in pixels
    int32_t height;          // Window height in pixels
    int32_t mouseX;          // Mouse X position (black hole center)
    int32_t mouseY;          // Mouse Y position (black hole center)
    float blackHoleRadius2;  // Black hole radius squared (for fast distance comparison)
    float satelliteRadius2; // Satellite radius squared (for hit detection)
```

```

    int32_t satCount;      // Total number of satellites
} GraphicParams;

static cl_mem      bufSats = NULL;
static size_t      satelliteBytes;

static cl_mem      bufGraphicParams = NULL; // graphic params
static cl_mem      bufPixels = NULL;
static size_t      graphicParamsBytes, pixelBytes;

```

We designed a `GraphicParams` struct to pass rendering parameters efficiently to the OpenCL kernel. This struct contains essential information like satellite count, window dimensions, mouse position, and precomputed squared radii for distance comparisons.

We created three OpenCL memory buffers to transfer data between the host and the kernel:

- `bufSats` - Contains satellite position and velocity data
- `bufGraphicParams` - Contains the `GraphicParams` struct with rendering parameters
- `bufPixels` - Stores the output pixel color data (RGBA format)

The buffer sizes are stored in `satelliteBytes`, `graphicParamsBytes`, and `pixelBytes` respectively. These buffers are allocated once during `ocl_init()` using `clCreateBuffer()` and reused across frames to minimize memory allocation overhead.

We replaced the original C code in the `parallelGraphicsEngine` function with a new function `run_graphics_on_ocl()` that runs the kernel. We kept the same logic as the original code and made sure to check for errors after each OpenCL function call using `status`.

Then, we wrote an OpenCL kernel called `graphics_render()` in a new file named `parallel.cl`. This kernel does the graphics work.

OpenCL commands and performance

"Run your OpenCL implementation on a GPU with work group sizes 1×1, 4×4, 8×4, 8×8 and 16×16. (In case you solved the problem with 1-dimensional work-item range, test the sizes 1, 16, 32, 64, 256.) If some work group size does not work, try the other sizes first before debugging why the one does not work. If other sizes work fine, **just report that the size does not work**, and don't waste too much time trying to get non-working size working."

"What are the average frametimes (milliseconds) for the following cases:

Include these results in the same **TABLE** that you used in Part 1."

--local argument

"Make your code so that you can change the WG size easily afterwards because you might be asked to demonstrate different sizes when showing the code to the assistant."

The code is structured to allow easy WG size changes by passing command-line arguments like --local 256, --local 16, etc.

```
// parse optional --local flag
for (int i = 2; i < argc; ++i) {
    if (!strcmp(argv[i], "--local") && i+1 < argc) {
        localSize = (size_t)atoi(argv[+i]); // 1,16,32,64,256
        printf("Using localSize: %zu\n", localSize);
    }
}
```

on Mac

```
# Build the project in Release mode with optimizations
# -O3: highest optimization level
# -march=native: optimize for the current CPU architecture
# -ffast-math: enable fast floating-point math optimizations
cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_C_FLAGS="-O3 -march=native -ffast-math" .. && make

# Run benchmarks with different work-group sizes
# Seed 1234 ensures reproducible results

# Test with work-group size 1
./parallel 1234 --local 1

# Test with work-group size 16
./parallel 1234 --local 16

# Test with work-group size 32
./parallel 1234 --local 32
```

```
# Test with work-group size 64
./parallel 1234 --local 64
```

```
# Test with work-group size 256
./parallel 1234 --local 256
```

Case (Mac, 1920*1024, only Graphics openCL)	6-205 fames Average Physics (ms)	6-205 fames Average Graphics (ms)	6-205 fames Total (ms)
1 WG	7.32	5.55	14.41
16 WG	7.33	5.59	14.49
32 WG	7.36	5.62	14.41
64 WG	7.40	5.59	14.53
256 WG	7.34	5.67	14.49

on Windows

```
# Build the project in Release mode with optimizations
# Note: -march=native and -ffast-math work with MinGW/GCC on Windows
# If using MSVC, you'll need different flags
cmake -DCMAKE_C_FLAGS="/O2 /fp:fast" ..
cmake --build . --config Release

# Run benchmarks with different work-group sizes
# Seed 1234 ensures reproducible results

cd Release
# Test with work-group size 1
./parallel.exe 1234 --local 1

# Test with work-group size 16
./parallel.exe 1234 --local 16

# Test with work-group size 32
./parallel.exe 1234 --local 32

# Test with work-group size 64
./parallel.exe 1234 --local 64
```

```
# Test with work-group size 256
./parallel.exe 1234 --local 256
```

Case (Windows, 1920*1024, only Graphics openCL)	Physics (ms)	Graphics (ms)	Total (ms)
1 WG	76	24	102
16 WG	77	3	82
32 WG	76	1	79
64 WG	76	2	79
256 WG	78	2	81

Performance Analysis Questions

1. The Original C Version on CPU without optimizations?

on Mac: physics 70ms, graphics 1829ms, total 1903ms

on Windows: physics 164ms, graphics 1254ms, total 1420ms

2. The Original C version on CPU with best optimizations?

on Mac: physics 7ms, graphics 208ms, total 217ms

on Windows (flags + algorithm optimized): physics 76ms, graphics 252ms, total 329ms

3. The OpenCL version on GPU, WG size 1×1?

on Mac: physics 7ms, graphics 5ms, total 14ms

on Windows: physics 76ms, graphics 24ms, total 102ms

4. The OpenCL version on GPU, WG size 4×4?

on Mac: physics 7ms, graphics 5ms, total 14ms

on Windows: physics 77ms, graphics 3ms, total 82ms

5. The OpenCL version on GPU, WG size 8×4?

on Mac: physics 7ms, graphics 5ms, total 14ms

on Windows: physics 76ms, graphics 1ms, total 79ms

6. The OpenCL version on GPU, WG size 8×8?

on Mac: physics 7ms, graphics 5ms, total 14ms

on Windows: physics 76ms, graphics 2ms, total 79ms

7. The OpenCL version on GPU, WG size 16×16?

on Mac: physics 7ms, graphics 5ms, total 14ms

on Windows: physics 78ms, graphics 2ms, total 81ms

8. (Bonus: If some work group size or sizes did not work, try to explain why it did not work?)

All work group sizes worked.

With WINDOW_HEIGHT 1024 and WINDOW_WIDTH 1920, none of the work group sizes (1, 16, 32, 64, 256) require rounding up—we could set `globalWorkSize[0] = totalPixels` directly. However, if the window size changes, this approach isn't safe. Instead, we rounded up the total work to match the local work group size: `globalWorkSize[0] = ((totalPixels + localSize - 1) / localSize) * localSize`. This ensures the total number of tasks is always an integer multiple of the local size, preventing errors from leftover items.

9. Try to analyze the performance with different work group sizes. What might explain the differences?

We tested different work group sizes (1, 16, 32, 64, and 256 workers at a time).

Work group sizes of 16, 32, 64, and 256 performed similarly. On Windows, 32 workers per group achieved the best performance, though the difference was slight. The variations between different sizes were minimal and negligible on both platforms.

For this problem size, the overhead of setting up the GPU and transferring data takes more time than the actual speedup we get from parallel processing. Different work group sizes perform similarly because they all face the same overhead costs. The small differences we see are just normal variation in measurements.

Small Window Size Test (80x80)

"Try reducing the window size to very small, 80×80. The WINDOW WIDTH and WINDOW HEIGHT defines in the beginning of the code control this."

Case (Mac, 80*80, only Graphics openCL)	10th frame Physics (ms)	10th frame Graphics (ms)	10th frame Total (ms)
1WG	7.43	0.65	8.29
16 WG	7.41	0.67	8.33
32 WG	7.43	0.67	8.35

64 WG	7.52	0.65	8.40
256 WG	7.43	0.66	8.40

Case (Windows, 80*80, only Graphics openCL)	10th frame Physics (ms)	10th frame Graphics (ms)	10th frame Total (ms)
1 WG	77	0	77
16 WG	78	0	78
32 WG	77	0	77
64 WG	77	0	79
256 WG	76	1	77

1. The Original C Version on CPU without optimizations?

on Mac: physics 70ms, graphics 5ms, total 76ms

on Windows: physics 163ms, graphics 4ms, total 169ms

2. The Original C version on CPU with best optimizations?

on Mac (flags + algorithm optimized): physics 7ms, graphics 0ms (a floating-point number less than 1), total 8ms

on Windows (flags + algorithm optimized): physics 77ms, graphics 1ms, total 78ms

Both are significantly faster than the full-size window due to reduced pixel count ($80 \times 80 = 6,400$ pixels vs default size $1920 \times 1024 = 1,966,080$ pixels).

3. On OpenCL version on GPU, fastest WG size (can be different than with the default image size)?

On Mac, the fastest work group size is 1 for both default and small image sizes.

On Windows, the fastest work group size is 32 for both default and small image sizes.

4. If the OpenCL version running on GPU was slower than C on CPU with this image size, what is the reason for this?

With only 6,400 pixels, the GPU parallelization overhead (memory allocation, data transfer to/from GPU, kernel launch) exceeds the computation savings. GPUs excel with large datasets where parallel computation benefits outweigh the overhead costs. At 80×80 , the CPU can process pixels quickly enough that GPU overhead becomes the bottleneck.

However, we did not find the OpenCL version running slower with small image size. ???

Improvement

As a final (and the most significant) bonus question, you are now free to implement any optimizations to your program. How fast can you get it to run?

Compile OpenCL function with optimization flags

We optimized OpenCL kernels by passing build options to `clBuildProgram`.

```
status = clBuildProgram(  
    program,  
    1,  
    &deviceIds[DEVICE_INDEX],  
    "-cl-fast-relaxed-math -cl-mad-enable",  
    NULL,  
    NULL  
) ;
```

- `"-cl-fast-relaxed-math"` enables faster but less precise floating-point math operations by relaxing IEEE 754 compliance
- `"-cl-mad-enable"` allows the compiler to use fused multiply-add instructions ($a*b+c$ as a single operation), which is faster and can improve performance

However, these flags improve performance on Mac but show no difference on Windows.

Case (Mac, 1920*1024, Graphics openCL + flags)	6-205 frames Average Physics (ms)	6-205 frames Average Graphics (ms)	6-205 frames Total (ms)
1 WG	7,43	5,15	14,15
16 WG	7,37	5,00	13,91
32 WG	7,43	4,94	13,93
64 WG	7,41	5,10	14,07
256 WG	7,32	4,83	13,70

Case (Windows, 1920*1024, Graphics openCL + flags)	10th frame Physics (ms)	10th frame Graphics (ms)	10th frame Total (ms)
1 WG	76	24	102
16 WG	77	3	82
32 WG	76	1	79
64 WG	76	2	79
256 WG	78	2	81

Add Physics openCL kernel

We also tried adding the physics calculations using OpenCL.

OpenCL 1.2 on our Mac does not support the double type, so we could only test on OpenCL 3.0 on the TC303 PC.

The results showed that parallelizing the physics engine using OpenCL improved performance.

The best performance is from 32 local work size. As this size aligns with GPU hardware, maximizing parallel efficiency.

64 and 256 perform worse as they have higher register and memory pressure that reduces occupancy, increased synchronization overhead, and poor alignment with hardware execution units that causes thread idle time.

Case (Windows, 1920*1024, Physics openCL + Graphics openCL)	10th frame Physics (ms)	10th frame Graphics (ms)	10th frame Total (ms)
1 WG	58	20	80
16 WG	55	2	59
32 WG	54	1	56
64 WG	58	2	61
256 WG	58	2	61

OpenMP + OpenCL implementation

```
# Build the project in Release mode with optimizations
# Note: -march=native and -ffast-math work with MinGW/GCC on Windows
# If using MSVC, you'll need different flags
cmake -DCMAKE_C_FLAGS="/O2 /fp:fast" ..
```

```

cmake --build . --config Release

# Run benchmarks with different work-group sizes
# Seed 1234 ensures reproducible results

cd Release

$env:OMP_NUM_THREADS=8

# Test with work-group size 1
./parallel.exe 1234 --local 1

# Test with work-group size 16
./parallel.exe 1234 --local 16

# Test with work-group size 32
./parallel.exe 1234 --local 32

# Test with work-group size 64
./parallel.exe 1234 --local 64

# Test with work-group size 256
./parallel.exe 1234 --local 256

```

Case (Windows, 1920*1024, Physics openMP + Graphics openCL)	first 10 frames Average Physics (ms)	first 10 frames Average Graphics (ms)	first 10 frames Average Total (ms)
8 threads, 1 WG	14	21	36
8 threads, 16 WG	16	2	20
8 threads, 32 WG	15	1	17
8 threads, 64 WG	16	1	18
8 threads, 256 WG	14	1	17

Final Choice

On Mac M1 Pro, the best performance uses "[-cl-fast-relaxed-math -cl-mad-enable](#)" flags and runs only the graphics function on the GPU kernel with a work size of 1(16/32/64/256 also good).

Case (Mac, 1920*1024)	6-205 frames Average Physics (ms)	6-205 frames Average Graphics (ms)	6-205 frames Total (ms)	Speedup vs Debug
CPU – Debug	70	1829	1903	1.0x
CPU – Release(flags+code optimized)	7	208	217	8.77x
GPU – 1 WG(only Graphics openCL)	7.32	5.55	14.41	132.1x
GPU – 16 WG(only Graphics openCL)	7.33	5.59	14.49	131.4x
GPU – 32 WG(only Graphics openCL)	7.36	5.62	14.41	132.1x
GPU – 64 WG(only Graphics openCL)	7.40	5.59	14.53	131.0x
GPU – 256 WG(only Graphics openCL)	7.34	5.67	14.49	131.4x

Our final choice is to run the physics function with 8-thread OpenMP and the graphics function with OpenCL at a local work size of 256(32/64 also good) on TC303 Windows.

Case (Windows, 1920*1024)	10th frame Physics (ms)	10th frame Graphics (ms)	Total (ms)	Speedup vs Debug
CPU – Debug	164	1254	1420	1.0x
CPU – Release(flags+code optimized)	76	252	329	4.32x
8 threads(Physics openMP), GPU – 1 WG(Graphics openCL)	14	21	36	39.4x
8 threads(Physics openMP), GPU – 16 WG(Graphics openCL)	16	2	20	71.0x
8 threads(Physics openMP),	15	1	17	83.5x

GPU – 32 WG(Graphics openCL)				
8 threads(Physics openMP), GPU – 64 WG(Graphics openCL)	16	1	18	78.9×
8 threads(Physics openMP), GPU – 256 WG(Graphics openCL)	14	1	17	83.5×

Further ideas

(Bonus: Add some more thoughts on how you would attempt to further improve the total performance of this system?)

- Use OpenCL local memory to cache frequently accessed data
- Minimize data transfers between CPU and GPU by keeping data on GPU between frames
- Use asynchronous memory transfers to overlap computation and communication
- Optimize memory access patterns to ensure coalesced reads/writes
- Profile to identify remaining CPU bottlenecks and optimize those sections

Feedback

1. What was good in this exercise work?

The exercise provided excellent hands-on experience with OpenCL and GPU programming. The structured progression from CPU to GPU implementation helped understand parallelization benefits and tradeoffs.

2. How you would improve this exercise work?

The reference implementation is very helpful. More guidance like a checklist on debugging OpenCL kernel errors would be preferred.

3. What was the most important and/or interesting thing you learned from this exercise work?

Understanding the overhead costs of GPU parallelization and when GPU acceleration is beneficial vs. when CPU execution is more efficient.

4. What was the most difficult thing in this exercise work?

We tested on two different operating systems. The commands differ between them, and the parallelization performance varies significantly across the two platforms.

5. Approximately how many hours did it take to complete the project work Part 2?

About 40 hours. Testing various combinations of configurations and creating performance tables was time-consuming.