COMP.CE.350 Parallel Computing

Project Report (OpenMP & OpenCL)

Group S

S. No	Name	Student ID	Email Address
01	Sai Poojith Dasari	150165934	poojith.dasari@tuni.fi
02	Sana Javed	50507430	sana.javed@tuni.fi

Contents

6.1. Benchmarking the original Code and Improving Performance via Compiler Settings:

1. **With the Original C Version without any compiler optimization flags?**

Frame time of physics routine (ms)	Frame time of Graphic routine (ms)	Total Frame time (ms)
99	1127	1230

2. **With the Original C version using most compiler optimizations (-02)?**

Frame time of physics routine (ms)	Frame time of Graphic routine (ms)	Total Frame time (ms)
35	328	367

3. **With the Original C version using also vectorization optimization in the compiler(-ftree-vectorize)?**

Frame time of physics routine (ms)	Frame time of Graphic routine (ms)	Total Frame time (ms)
34	339	377

Enable also -fopt-info-vec flag to vectorize and see information about vectorization. Did the compiler tell it managed to vectorize any loops? If so, which loops?

Yes, the compiler provided information about vectorized loops.

Details by Compiler:

```
parallel.c:183:22: optimized: loop vectorized using 16 byte vectors

parallel.c:167:27: optimized: basic block part vectorized using 16 byte vectors

parallel.c:167:27: optimized: basic block part vectorized using 16 byte vectors

parallel.c:167:27: optimized: basic block part vectorized using 16 byte vectors

parallel.c:254:17: optimized: basic block part vectorized using 16 byte vectors
parallel.c:335:24: optimized: basic block part vectorized using 16 byte vectors
parallel.c:394:22: optimized: loop vectorized using 16 byte vectors
parallel.c:378:27: optimized: basic block part vectorized using 16 byte vectors
parallel.c:378:27: optimized: basic block part vectorized using 16 byte vectors
parallel.c:378:27: optimized: basic block part vectorized using 16 byte vectors
parallel.c:534:21: optimized: basic block part vectorized using 16 byte vectors
parallel.c:534:21: optimized: basic block part vectorized using 16 byte vectors
```

Following loops are vectorized by the compiler:

```
for (int i = 0; i < SATELLITE_COUNT; ++i) {
    s[i].position.x = tmpPosition[i].x;
    s[i].position.y = tmpPosition[i].y;
    s[i].velocity.x = tmpVelocity[i].x;
    s[i].velocity.y = tmpVelocity[i].y;
  }
}


 for (int i = 0; i < SATELLITE_COUNT; ++i) {
    satellites[i].position.x = tmpPosition[i].x;
    satellites[i].position.y = tmpPosition[i].y;
    satellites[i].velocity.x = tmpVelocity[i].x;
    satellites[i].velocity.y = tmpVelocity[i].y;
  }


}
```

4. **Experiment with the SIMD instruction set and FP relaxation related optimization flags.(for example -ffast-math, -mavx, -mavx2, -msse4, - mavx512f). You can see gcc man page (man gcc) and find section “x86 options” to see list of supported instruction set extension options. There are a LOT of them so you do not have to test them all.). Which are the compilation flags you found to give the best performance?**
Some of the compilation flags which are tested while experimenting with SIMD flags are listed below in a tabular form.

Flags	**Frame time of physics routine (ms)**	**Frame time of Graphic routine (ms)**	**Total Frame time (ms)**

-ffast-math	23	236	263
-mavx2	21	306	331
-mavx	21	306	331
-msse4	22	328	354
-msse	23	236	263

Details about executed flags:

-ffast-math: This flag does not do anything specific by itself but it is a shorthand to other compiler options.

-msse/msse4 : It performs single mathematical or logical operation on multiple values at once. This is how it boosts the performance of the code. SSE stands for Streaming SIMD Extension.

- mavx2/mavx: These instructions improve performance of an application by processing large chunks of values i.e vectors at the same time. AVX vectors can contain up to 256 bits of data.

How fast was the code with the flags which the give the performance ?

The time is reduced by almost 114ms as compared to original version c vectorized command.

Did you find some compiler flags which cause broken code to be generated, and if, can you think why?

Yes, while testing different compiler flags it is observed that the code broke for -mavx512f, mavx512pf, mavx512er flags. Getting “Illegal instruction (core dumped)” error on running the code with this compiler which means the avx512 is not supported by my CPU.

6.2 Generic algorithm optimization

The command used to compile the code: **gcc -o parallel parallel.c -std=c99 -lglut -lGL -lm -O2 -ftree-vectorize -fopt-info-vec -ffast-math**

Algorithm change: The Pixel weights were calculated twice in the loops at the *parallelGraphicsEngine* function. So, instead of this unnecessary calculation, we can cache the weights and reuse them in the second loop.

Changes for loops are represented here:

```
float weights_cache[SATELLITE_COUNT];

    // First Graphics satellite loop: Find the closest satellite.
    for(int j = 0; j < SATELLITE_COUNT; ++j) {
      floatvector difference = {.x = pixel.x – satellites[j].position.x, .y = pixel.y - satellites[j].position.y};
      float d_diff = difference.x * difference.x + difference.y * difference.y;
      distance = sqrt(d_diff);
      weights_cache[j] = 1.0f / (d_diff * d_diff);
      weights += weights_cache[j];
     }
// Second graphics loop: Calculate the color based on distance to every satellite.
    if (!hitsSatellite) {
      for(int j = 0; j < SATELLITE_COUNT; ++j){
  // floatvector difference = {.x = pixel.x – satellites[j].position.x, .y = pixel.y – satellites[j].position.y};
        //  float dist2 = (difference.x * difference.x + difference.y * difference.y);
        float weight = weights_cache[j];
          }
}
```

These changes save about 60 seconds.

Initially from the above table, we see that when the compiler is run with flag -ffastmath Total framerate : 263ms, satellite moving: 23, space colouring: 236. whereas when it is optimised, the readings are framerate: 205, satellite moving: 22, space colouring : 178.

6.3 Code analysis for multi-thread parallelization

Which of the loops are allowed to be parallelized to multiple threads?

Physics satellite and Graphics satellite can be parallelized safely as they are not dependent on previous values.

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?

Parallelizing Graphics Pixel and Graphics Satellite loops are multi-thread safe but will not benefit from parallelization because there are 1048576 pixels and 64 satellites. Making hundreds and thousands of threads will slow down the process instead of boosting (this is also observed after applying OpenMP pragma to these loops)

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?

It has been observed that there are some same variables i.e., weight and distance in First and Second graphic satellite loops which can be shared by implementing atomic operations to make them thread safe and avoid race conditions.

```
_Atomic float weights = 0.f;

_Atomic float distance = 0;
```

Does your code transformation have any effect on vectorization performed by the compiler?

No, atomic variable does not have any impact on vectorization of loops.

6.4 OpenMP Parallelization

Run with the default 1024x1024 sized image. What is the proximal average total frametime (milliseconds) after the first frames in your OpenMP-multithreaded version of the code?

Frame time of physics routine (ms)	Frame time of Graphic routine (ms)	Total Frame time (ms)
23	41	69

Did you perform some extra code transformations or optimizations?

Yes, when we used the optimized code from 6.2 with our OpenMP-multithreaded version the average timings are:

Frame time of physics routine (ms)	Frame time of Graphic routine (ms)	Total Frame time (ms)
24	29	58

Which loops did you parallelize?

Graphics pixel loop is parallelized.

Did parallelization of some loop break something or cause a slowdown? Try to explain why?

Yes, the parallelization of First Graphics satellite loop failed because OpenMP does not support break. Parallelizing second graphics satellite showed a buggy pixel error message implemented in errorCheck function because probably the pixel size is increased then allowed limit. Physics satellite loop slows down on implementing OpenMP pragma.

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?

The default threads of CPU are 2 but increasing the threads to 4 and 8 or 16 increases the frame time as compared to default threads.

Number of threads	**Frame time of physics routine (ms)**	**Frame time of Graphic routine (ms)**	**Total Frame time (ms)**
4 & 8	22	61	88
16	22	47	73

Did you use TC217 computers or your own computer? If your own computer, what is the brand and model number of your CPU? and version of your compiler?

TC217 lab computer is used for accomplishing the tasks in exercise.

Approximately how many hours did it take to complete this exercise part (Part 1)?

It took us almost 24 hours to complete part 1 of the lab work.

7. OpenCL Parallelization:

1. **The Original C Version on CPU without optimizations?**
2. **The Original C version on CPU with best optimizations?**
3. **The OpenCL version on GPU, WG size 1x1?**
4. **The OpenCL version on GPU, WG size 4x4?**
5. **The OpenCL version on GPU, WG size 8x4?**
6. **The OpenCL version on GPU, WG size 8x8?**
7. **The OpenCL version on GPU, WG size 16x16?**

Optimization Flag	**Frame time of physics routine (ms)**	**Frame time of Graphic routine (ms)**	**Total Frame time (ms)**
Original C Version without any compiler optimization flags	99	1127	1230
Original C version using most compiler optimizations (-02)	35	328	367
Original C version using also	34	339	377

vectorization optimization in the compiler(-ftree-vectorize)			
-ffast-math	23	236	263
-mavx2	21	306	331
-mavx	21	306	331
-msse4	22	328	354
-mavx512f, mavx512pf, mavx512er	Code Broke		
-msse	22	236	263
OpenCL version on GPU, WG size 1x1	10	179	192
OpenCL version on GPU, WG size 4x4	10	12	27
OpenCL version on GPU, WG size 8x4	12	7	24
OpenCL version on GPU, WG size 8x8	12	6	23
OpenCL version on GPU, WG size 16x16	11	6	23

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

Work group is a 1,2- and 3-dimensional set of threads which further comprises of work items and these work items maps to a core in a GPU. To gain the best performance the work group size should match the hardware compute units. The performance depends upon the work item which can be scheduled and tracked in most of the devices the normal work items are 1024 so we got better results with 16* 16 (1024 / 256 = 4) per processor can be used.

Try reducing the window size to very small, 80x80. The WINDOW WIDTH and WINDOW HEIGHT defines in the beginning of the code control this.

1. **On Original C version on CPU with optimizations on (-03 flag)?**

Frame time of physics routine (ms)	**Frame time of Graphic routine (ms)**	**Total Frame time (ms)**
20	1	22

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

Frame time of physics routine (ms)	Frame time of Graphic routine (ms)	Total Frame time (ms)
10	0	16

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

   One of the reasons for a slower OpenCL code can be incorrect group size. On selecting the wrong group size, the hardware resources will not be properly used and code can slow down.

8. Feedback:

1. **What was good in this exercise work?**

We learnt to improve the performance of the code, which we normally don't bother about in practical life. Improving the performance of the software is highly appreciated in Embedded systems as the system should respond to requests quickly.

2. **How you would improve this exercise work?**
   There should be more details of OpenCL in the lectures rather than reading a set of new slides, which increased the workload immensely or there should be more time than 2 weeks.

3. **What was the most important and/or interesting thing you learned from this exercise work?**
   Few of the most important things I learnt through this exercise work are: Optimizing the code to facilitate low latency is very important and OpenCL, OpenMP parallelization and vectorization of loops are important part of that.
   Also, parallelization may not give better performance always, we should parallelize based on the need and the CPU/GPU configuration we have.

4. **What was the most difficult thing in this exercise work?**
   Understanding and implementing OpenCL was a bit tricky but wouldn't say it was difficult. It was a good way of learning OpenCL.

5. **Approximately how many hours did it take to complete this exercise part (Part 2)?**
   It took approximately 15 hours to understand and implement part 2 of the exercise.