Tampere University
COMP.CE.350 Multicore and GPU Programming Lab Work
Autumn 2022

Topi Leppänen

September 30, 2022

1 Overview of the Project Work

The purpose of this work is to help understanding the massive parallelism available by GPUs and manycore processors and to learn how to harness that parallel hardware from the programmer's perspective.

The first part of the exercise work involves

1. Trying different compiler flags and seeing how the performance of the program changes with different compiler flags and how compiler vectorizes a program

2. Adding OpenMP multicore parallelization to a C program.

The Second part involves porting the code into the OpenCL parallel programming standard which can be executed on a GPU. Also performance analysis between the C version and the OpenCL version running on GPU is a part of the project.

Note: OpenMP will be discussed on the lecture 7. After going through it, you are recommended to start working on the part 1. OpenCL will be discussed on the lecture 9. After that, you should have all the prerequisite information you need to begin the second part. The lectures should already be available in Moodle.

2 Returning the Exercise

Parts one and two will be returned separately to Moodle. In addition, in the second part of the lab work, the working program (either in class TC217 or by a laptop) need to be shown to the assistant.

2.1 The First Part (OpenMP)

The first part of the exercise work requires analyzing a C program, checking how the compiler vectorizes it, and adding OpenMP multicore parallelization to it.

Recommended class for the first part of the work is TC217.

The deadline for the first part is **23:59, October 30, 2022**. The returning is done in Moodle and it contains the following contents:

1. A reasonably well structured report with:
   - Names, student numbers and email addresses of all the group members
   - Answers all the questions in this document (marked with **?**)
   - If no makefiles/project files are used, then write the used compilation command in the report.
   - Not mandatory, but to help you get a bonus point, you can also answer to the question marked with **?** You can also add other analysis you found interesting to think about.
2. The source code and all the makefiles of the program.

2.2 The Second Part (OpenCL)

The deadline for the second part of the work is **23:59, December 2, 2022**.

The returning of the second part consists of two parts:

1. A working program needs to be demonstrated to the assistant in class TC217 or using student's own laptop with a separate GPU (Preferably during the allocated Q&A times. You can also arrange a different demonstration time at firstname.lastname@tuni.fi or during/after the weekly exercises.

2. In addition to the demonstration, submit a report+code to Moodle following the same instructions as in part 1 above.

3 Arrangements

Class TC217 contains computers which have Linux with a recent version of GCC compiler and a GPU.

Usage of one's own computer is also allowed. A few groups which have the most excellent answers to the questions and specially fast/well-optimized code may get bonus points. Returning the code which is fastest of all the returnings on the TC217 computers will likely earn you a bonus point.

During the development stage, CPU implementation of the OpenCL standard such as Intel OpenCL (`https://software.intel.com/en-us/intel-opencl`), AMD OpenCL (`http://developer.amd.com/tools-and-sdks/opencl-zone/`) or pocl (`http://portablecl.org/`) (runs on Linux machines) can be used for development, but the final benchmarks of the second part has to be done with a computer with a GPU that has its own discrete memory.

4 The Code Used in the Exercise

The code consists of two parts: A physics engine and a graphics engine. The physics engine updates the locations of satellites orbiting a black hole. (the center of the window). Graphics engine colors all pixels based on 1) the id color of the closest satellite 2) weighted average (based in distance) of all the satellites.

The original C code of the program is given, and contains own functions for the parts.

Only the graphics engine needs to be ported to OpenCL in the second phase of the work. Both physics and graphics engines may(but do not have to) be parallelized with OpenMP on the first part of the work if a working solution is found.

5 How to get the C reference code running?

The original C code should compile in Linux, Windows and MacOS X.

However, the OpenMP support needed for the part 1 can be troublesome on MacOS X, and the vectorization support in Visual studio seems to be lacking. Therefore, Linux is recommended. Linux computers can be found in TC217.

Command line parameter -fopt-info-vec to GCC gives information on what the compiler manages to vectorize.

5.1 Compiling in Linux

no optimizations:

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

most optimizations, no vectorization

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

Also vectorize and show what loops get vectorized:

```
gcc -o parallel parallel.c -std=c99 -lglut -lGL -lm -O2 -ftree-vectorize
-fopt-info-vec
```

Also allow math relaxations

```
gcc -o parallel parallel.c -std=c99 -lglut -lGL -lm -O2 -ftree-vectorize
-fopt-info-vec -ffast-math
```

Also allow AVX2 SIMD instructions

```
gcc -o parallel parallel.c -std=c99 -lglut -lGL -lm -O2 -ftree-vectorize
-fopt-info-vec -ffast-math -mavx2
```

Also support OpenMP

```
gcc -o parallel parallel.c -std=c99 -lglut -lGL -lm -O2 -ftree-vectorize
-fopt-info-vec -ffast-math -mavx2 -fopenmp
```

Also support openCL:

```
gcc -o parallel parallel.c -std=c99 -lglut -lGL -lm -O2 -ftree-vectorize
-fopt-info-vec -ffast-math -mavx2 -fopenmp -lOpenCL
```

To run the compiled binary:

```
./parallel
```

5.2 Compiling on MacOS X

No optimizations:

```
clang -o parallel parallel.c -framework GLUT -framework OpenGL
```

Full optimizations:

```
clang -O3 -o parallel parallel.c -framework GLUT -framework OpenGL
```

With OpenCL libraries:

```
   clang -O3 -o parallel parallel.c -framework GLUT -framework OpenGL
-framework OpenCL
```

OpenMP is unfortunately not trivially working on Macos X, and requires installing custom compiling instead of the one that comes with XCode

6 First part: CPU parallelization

6.1 Benchmarking the original Code and Improving Performance via Compiler Settings

Do at least this part with a computer which has a (non-ancient version of) gcc compiler(Practically any modern Linux distribution is good).

COLLECT ALL YOUR MEASUREMENTS IN A TABLE (e.g. Excel), which you will then include in your report. You can also visualize the results.

Run with default 1024x1024 sized image. What is the proximal averages frametimes of the physics routine("satellite moving"), graphics routine("space coloring"), and total frametime (milliseconds) after the first frames:

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

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

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

   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?

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?.

   Can you explain, what each of the optimization flags you found to give the best performance does?(Note: When googling about those instruction set extensions, leave the "-m" out, for example to see what "-msse" enables, google for just "sse" to see what are sse instructions)

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

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

6.2 Generic algorithm optimization

This part is not mandatory to get passing grade, but can help to get the bonus point.

Lets 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?

6.3 Code analysis for multi-thread parallelization

Analyze the code in the loops in the *ParallelPhysicsEngine* and *Parallel-GraphicsEngine* functions. The loops are called *physics iteration*, *Physics satellite*, *Graphics pixel* and *Graphics satellite*. Use these names in your answers.

Answer the following questions:

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

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?

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?

6.4 OpenMP Parallelization

After answering these questions, use OpenMP pragmas (and maybe perform other related code changes) to parallelize those loops you consider allowed and worth parallelization. Remember to also enable OpenMP from the compiler settings. (-fopenmp in gcc)

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?
Did you perform some extra code transformations or optimizations?

Which loops did you parallelize?

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

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?

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?

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

7 Second part: OpenCL parallelization

Create an OpenCL kernel for the graphics engine routine, and replace C code of the the parallelGraphicsEngine-function with your OpenCL calls. Freely copy-paste (parts of) the original C routine into your OpenCL kernel. Initialization of the OpenCL should be done in the init-function and releasing of the OpenCL objects in the destroy-function.

OpenCL parallelization of the physics engine is not required, but it can help you with the fastest code competition.

Run your OpenCL implementation on a GPU with work group sizes 1x1, 4x4, 8x4, 8x8 and 16x16. (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.

Run with default 1024x1024 sized image. What is the proximal average total frametime (milliseconds) after the first frames for the following cases: Include these results in the same TABLE that you used in Part 1

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?

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.

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

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

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

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

(Bonus: Add some more analytical thoughts on how you would attempt to further improve the total performance of this system?You can also attempt some of these things yourself.)

8 Feedback

Feedback is returned with the second part of the work. If you like to give anonymous feedback of this exercise work you can do that with the course feedback system.

1. What was good in this exercise work?
2. How you would improve this exercise work?
3. What was the most important and/or interesting thing you learned from this exercise work?
4. What was the most difficult thing in this exercise work?
5. Approximately how many hours did it take to complete this exercise part (Part 2)?