Tampere University COMP.CE.350 Multicore and GPU Programming Project Work Autumn 2023

Topi Leppänen

October 26, 2023

1 Overview of the Project Work

The purpose of this work is to help understand the massive parallelism available in GPUs and multicore processors. The idea is to learn in practice how the parallel hardware can be harnessed by the programmer.

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. Performance analysis between the C version and the OpenCL version running on GPU is also 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.

2 Returning the Exercise

Parts one and two will be returned separately to Moodle. In addition, in the second part of the project work, the working program (either in class TC217 or by a laptop) needs 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 29, 2023**. The returning is done in Moodle and it contains the following contents:

1. A well structured report with:
   - Names, student numbers and email addresses of all the group members
   - Answer all mandatory questions (marked with ?)
   - If no makefiles/project files are used, then write the used compilation command in the report.
   - Optionally, to improve the grade of the project work, you can answer the questions marked with ?.
   - You can also add other analysis and (documented) experiments you found interesting.
2. The source code (canny.c) and all the makefiles of the program.

2.2 The Second Part (OpenCL)

There is a small OpenCL task which is to be completed at the beginning of the Part 2. Deadline for that is **23:59, November 12, 2023**. The deadline for the full second part of the work is **23:59, December 1, 2023**.

The returning of the full 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 topi.leppanen@tuni.fi or during/after the weekly exercises.

2. In addition to the demonstration, submit a report+code (canny.c + canny.cl) 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.

Your project will be graded on the following scale 0-3:

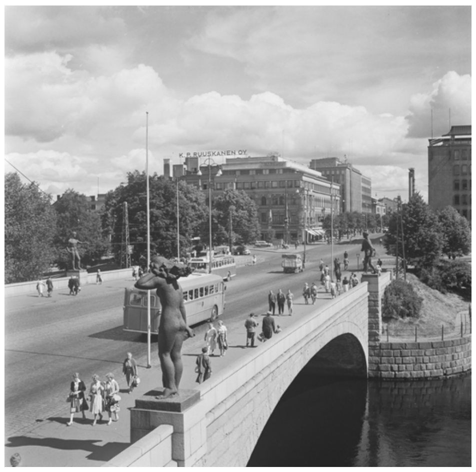
0. Failed, will need to revised according to the feedback
1. Accepted, minimal working implementation, alright answers to the questions.
2. Good, sensible implementation, good answers with not many mistakes (worth 0.25 of a grade to a passed exam)
3. Excellent, fast implementation, in-depth answers, own well-documented experimentation (worth 0.50 of a grade to a passed exam)

During the development stage, CPU implementation of the OpenCL standard such as Intel OpenCL or PoCL (available on many Linux distro package managers) 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 Canny edge detection algorithm, a working C implementation of which is provided. The specification of the algorithm is from OpenVX specification 1.3: Canny Edge Detector. OpenVX is a domain-specific language for computer vision. It is not used for this project work except that the specification of the application is meant to be OpenVX comformant. The project work will be written in C and OpenCL.

The provided C implementation is split into 4 separate functions:


(a) Hämeensilta

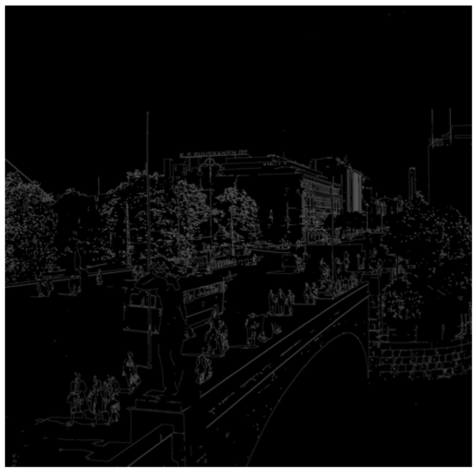

(b) Edges

Figure 1: Output of the Canny Edge Detection algorithm (./canny -B)

1. Sobel, 3x3 window filtering with both horizontal and vertical coefficients
2. Phase and Magnitude, computes gradient direction and magnitude from sobel's output
3. Non-Maximum Suppression, zero-out pixels perpendicular to the edge (to thin out the edges). Also applies double thresholded hysteresis.
4. Edge Tracing, edges are extended to form longer solid lines.

For more detailed description of the algorithm, see the above OpenVX specification and the provided C implementation. You can also use other sources for information about Canny edge detection. However, in case of conflicting information the order of precedence is 1.OpenVX specification, 2.provided implementation, 3.other sources. The provided implementation uses *clamp-to-edge* as a border handling strategy.

There are two test images provided: x.pgm and hameenkatu.pgm. The first one can be useful for debugging broken implementation. The second one is a larger image used for runtime benchmarking.

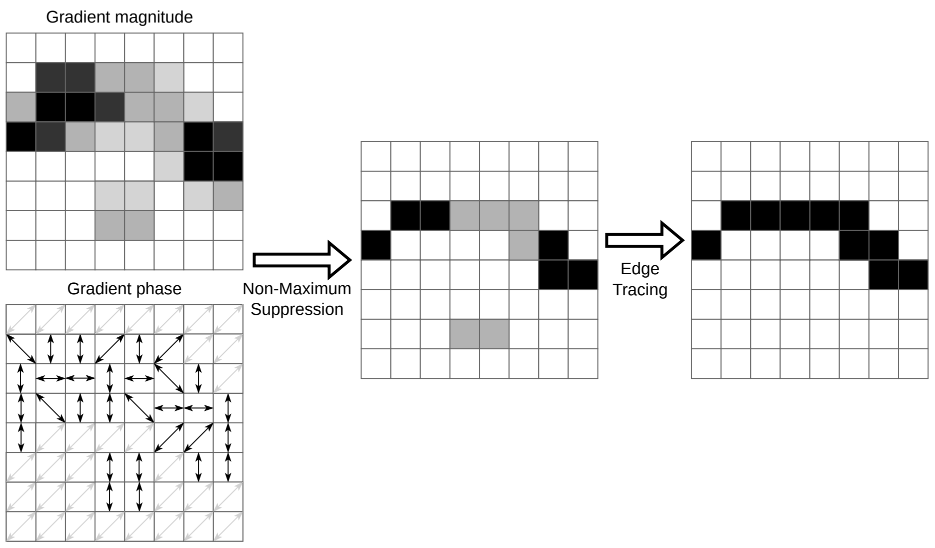


Figure 2: Non-max suppression and edge tracing functions of the Canny algoritm visualized.

5 How to get the C reference code running?

The original C code should compile in Linux, Windows and MacOS X (only tested on Linux).

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 canny util.c canny.c -lm
```

most optimizations, no vectorization

```
gcc -o canny util.c canny.c -lm -O2
```

Also vectorize and show what loops get vectorized:

```
gcc -o canny util.c canny.c -lm -O2 -ftree-vectorize -mavx2
```

```
-fopt-info-vec
```

Also allow math relaxations

```
   gcc -o canny util.c canny.c -lm -O2 -ftree-vectorize -mavx2
-fopt-info-vec -ffast-math
```

Also support OpenMP

```
   gcc -o canny util.c canny.c -lm -O2 -ftree-vectorize -mavx2
-fopt-info-vec -ffast-math -fopenmp
```

Also support OpenCL:

```
   gcc -o canny util.c opencl_util.c canny.c -lm -O2
-ftree-vectorize -mavx2 -fopt-info-vec -ffast-math -fopenmp
-lOpenCL
```

To run the compiled binary:
Runs the application once with the x.pgm image (output written to x_output.pgm). PGM images can be viewed with many common image viewers. For example:

```
   eog image.pgm
```

(From eog preferences, untick *Smooth images when zoomed in* in order to view small images accurately.)

```
   ./canny && eog x_output.pgm
```

To run benchmarking with the large hameenkatu.pgm image

```
   ./canny -B num_of_iterations
```

To run benchmarking with the small x.pgm image

```
   ./canny -b num_of_iterations
```

You can also run the video people.mp4. You need ffmpeg installed for this, which is unfortunately not available on TC217 machines, but is easy to install if you are using your own machine:

```
   ./canny -v
```

5.2 Compiling on MacOS X

No optimizations:

```
   clang -o canny util.c canny.c
```

Full optimizations:

```
   clang -O3 -o canny util.c canny.c
```

With OpenCL libraries:

```
   clang -O3 -o canny util.c opencl_util.c canny.c -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, which you will then include in your report. This is the most important part of your report, so failure to produce a clear table of results will be an automatic fail.

To get the runtime results, run with the provided hameenkatu.pgm image and -B option (./canny -B 10), which runs your code multiple times to get stabler results. Report the runtime averages for each function and the total Canny Edge Detection time (milliseconds)

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 -mavx2)**?**
4. Enable also -fopt-info-vec flag to vectorize and see information about vectorization. Did the compiler tell it managed to vectorize any of the loops (1.1, 1.2, 2.1, 2.2, 3.1, 3.2, 4.1, 4.2, 4.3, 4.4, 4.5 and 4.6) If so, which loops**?** (The TC217 compiler might not be able to autovectorize anything. You might have better success with your own machine and newer compiler version.)
5. With also the FP relaxation optimization flag (-ffast-math)**?**.
6. Do a bit of experimentation to find the fastest possible combination of flags. How fast can you get the program to run with those flags**?**
7. If loop #1.2 DID NOT get vectorized, what is the reason**?** (see -fopt-info-vec-all). How could it be modified to enable vectorization**?**

8. If loop #2.2 DID NOT get vectorized, what is the reason? How could it be modified to enable vectorization?

9. If loop #4.6 DID NOT get vectorized, what is the reason? How could it be modified to enable vectorization?

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 inside all four of the functions described in Section 4. (1.1, 1.2, 2.1, 2.2, 3.1, 3.2, 4.1, 4.2, 4.3, 4.4, 4.5 and 4.6)

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 benchmark option -B 10. What are the average runtimes (milliseconds) in your OpenMP-multi-threaded version of the code? Include these results in the same TABLE that you used in single-core optimizations 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?

As a bonus question, you are now free to implement any CPU optimizations (incl. advanced multicore and vectorization optimizations) to your program as long as your algorithm produces correct results for the Canny edge detection. How fast can you get your program on a CPU? Please, report your methods.

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

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

7 Second part: OpenCL parallelization

7.1 Pre-return: Introduction to OpenCL

This is a preliminary section of the OpenCL part of this project work. Completing this early is mandatory, since it verifies your OpenCL installation, and introduces you to OpenCL event profiling, which is needed in Part 2.

Common OpenCL programs contain a decent amount of generic code, which is often the same independent of the application to be accelerated (commonly called boilerplate code). In this task, you are introduced to some of this OpenCL boilerplate, and asked to perform small modifications to an example program.

If you are using your own machine, now is the time to install an OpenCL implementation. To avoid having to install anything, you can use the TC217 machines, which already have OpenCL installed.

In this mini-task, you first download the OpenCL University Kit from Moodle, go to example folder lec02_03_code/VectorAdd_C, compile and run the application:

```
make && ./vectoradd
```

1. What does the succesful execution of the application print?

Next, you need to modify the application to print some profiling information. We are interested in the data transfer and kernel execution times.

First, enable profiling in the clCreateCommandQueue-call. Then extract the events from clEnqueueWriteBuffer-, clEnqueueReadBuffer- and clEnqueueNDRangeKernel-calls. Use clGetEventProfilingInfo-calls to get start and end times of the commands. Finally, post-process the timestamps to a format that tells you how long the buffer transfers and kernel executions took.

2. What are kernel execution and buffer transfer times?

Return a short report with answers to just the above 2 questions and the modified vectoradd.c by 23:59, November 12, 2023.

7.2 OpenCL implementation of Canny Edge Detection

Now we will get back to the Canny Edge Detection application from Part 1. You will now implement it with OpenCL and run it on a GPU. You can include the opencl_util.h-header for some OpenCL helper functions. Include opencl_util.c in your compilation (gcc) command to build them. You can also copy some OpenCL boilerplate from the vectoradd-example from previous section. Make sure to add proper error checking after every OpenCL API call.

Create OpenCL kernels for the first three functions, **sobel3x3, phaseAndMagnitude** and **nonMaxSuppression** in a separate file called canny.cl. OpenCL parallelization of the edgeTracing-function is not required, but can be done as an extra task. Freely copy-paste (parts of) the original/optimized C routine into your OpenCL C kernels.

Initialization of the OpenCL (context, command queue, buffers creation) should be done in the init-function and releasing of the OpenCL objects in the destroy-function. These functions are called automatically by the provided code.

Modify the original edgeDetection-function to call OpenCL kernels and to move data from CPU to GPU and back. Since OpenCL commands are asynchronous, timing their execution with CPU timers is not sufficient. Use the OpenCL event profiling API (similarly as in Part 2 Pre-return) to time your kernel execution times and insert them (in ms) to *runtimes*-array. You should also time the buffer transfer times and report them somehow. Include the buffer transfer times in the total time reported for Canny Edge Detection, since they are a fundamental part of this implementation.

You can develop your program into a working state with the synthetic small image x.pgm (./canny).

For final measurements run with benchmark option -B 10. What are the average runtimes (milliseconds) of OpenCL version running on GPU? Include these results in the same TABLE that you used in Part 1. Add a new column for buffer transfer times, or alternatively report them once in text.

After that, do some testing with the smaller input size (100x100). Test benchmarking with the small x.pgm test image. (./canny -b 1000)

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

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

As a final (and the most significant) bonus question, you are now free to implement any optimizations to your program as long as your algorithm produces correct results for the Canny edge detection. How fast can you get your program?

Remember, you can check out the video mode with ./canny -v to see the final speedup of your optimizations (requires ffmpeg).

7.2.1 OpenCL Tips

Any calls to OpenCL APIs can fail *without* any explicit warning or crash. It is recommended practice to add explicit error checking after every OpenCL

API call. Many OpenCL examples you can find on the internet omit these for brevity.

opencl_util-file includes the following helper functions:

1. *read_source(”canny.cl”)* to read external canny.cl file from the working directory.

2. *clErrorString(errCode)* to perform the error checking after every OpenCL API call (e.g. add this after every API call:

   ```
   if(errCode != CL_SUCCESS) printf("%s",
   clErrorString(errCode));
   ```

3. *getStartEndTime(event)* to get the runtime of an event from its start to its end.

We recommend to first complete the project work with just OpenCL Buffer memory objects. This makes the initial conversion from C to OpenCL much easier. However, in this application, the use of OpenCL Image datatypes might be beneficial at some parts. This is because the Image datatypes include explicit border-handling modes, which enable the implementation to use hardware-assisted border handling. (instead of the *idx*-function currently used in the provided code.) However, it can take some effort to convert the application to use Images, so it is optional to use them and only after your OpenCL application already works.

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 and Part 2 pre-return)?