COMP.CE.350 Project 1

Miika Kulmala

292288

miika.kulmala@tuni.fi

	Sobel3x3 (ms)	phaseAndMagnitude (ms)	nonMaxSupression (ms)	edgeTracing (ms)	Total (ms)
Original	977,658	603,8	514,532	71,627	2167,616
"+ Optimizations"	58,495	443,831	164,395	36,129	702,85
"+ Vectorization"	61,569	455,419	160,413	37,627	715,027
"+ FP relaxation"	58,545	42,282	157,887	32,021	290,736
"+ O3”	44,078	42,05	157,109	38,9	282,21
"+ OpenMP"	16,606	65,26	23,05	25,83	130,7

Table 1: Performance measurements. Running compiled code ./canny -B 10

6.1

Report the runtime averages for each function and the total Canny Edge Detection time (milliseconds)

Format for the following answers :

Sobel3x3 (ms)   phaseAndMagnitude (ms)   nonMaxSupression (ms)   edgeTracing (ms) Total (ms)

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

977,658   603,8   514,532   71,627   2167,616

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

58,495   443,831   164,395   36,129   702,85

3. With the Original C version using also vectorization optimization in the compiler (-ftree-vectorize-mavx2)?

61,569   455,419   160,413   37,627   715,027

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?

None of the loops got vectorized at this point.

After adding -ffast-math flag, loop 2.2 got vectorized.

5. With also the FP relaxation optimization flag (-ffast-math)?.

58,545	42,282	157,887	32,021	290,736

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?

With the given flags fastest run was to have everything on. To get a slightly shorter runtime using flag -O3 instead of -O2 was used. Times:

44,078	42,05	157,109	38,9	282,21

These times are appended to Table 1 on row "+ 03"

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?

The loop 1.2 did not get vectorized. Gcc reported the following:

```
canny.c:54:30: missed: couldn't vectorize loop
canny.c:59:35: missed: not vectorized: not suitable for gather load _3 = *_2;
```

Couldn't find information what this exactly is, but indexing arrays, where the indices are too complex for compiler to understand might cause this.

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

Loop 2.2 got vectorized.

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

Loop 4.6 did not get vectorized. Gcc reported the following:

```
canny.c:251:27: missed: couldn't vectorize loop
canny.c:251:27: missed: not vectorized: control flow in loop.
```

There is no apparent control flow in the loop. Not sure how to fix this.

6.3

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

Loops 1.1, 1.2, 2.1, 2.2, 3.1, 3.2, 4,5 and 4.6 all seem embarrassingly parallel. These could be parallelized as is. As the input is not modified, and output is modified only once per element, there is no risk of data corruption.

Loops 4.1, and 4.2 could be parallelized as long as the tracing_stack_pointer operations are made safe. So shared between threads and atomic.

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?

All of the inner loops 2.2, 3.2, 4.6 will likely not benefit from parallelization, as we can just parallelize the outer loop and have enough entries to occupie all threads provided by the CPU. In other words, the outer loop can be unrolled into image_height amount of threads, and that is surely more than cores/threads our cpu offers.

Similiarly we could parallelize the inner loops, after which the outerloop would not be beneficial for above reason. Inner loop parallelization does bring more overhead though so it is better to parallelize outer loops.

We could also unroll the loops together, so we do not get the overhead of parallelizing inner loops, but also gain the benefit of having everything available for parallelization if we happen to have the hardware to utilize that.

4.1 and 4.2 also have the redunancy between each other as explained above, but on top of that we need to make sure tracing_stack_pointer operations are made thread safe, and that will likely bring more overhead than we would gain in performance parallelizing this, as the loop operation is very simple anyways.

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?

For loop pairs 1.1 / 1.2, 2.1 / 2.2, 3.1 / 3.2, 4.5 / 4.6 the following OpenMP pragma was added.

```
#pragma omp parallel for collapse(2)
```

This tells to parallelize the FOR loops, with collapse(2) unrolling and merging the 2 loops together.

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

Yes. The compiler reports it has vectorized all of the above loops now.

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?

Sobel3x3 (ms)	phaseAndMagnitude (ms)	nonMaxSupression (ms)	edgeTracing (ms)	Total (ms)
16,606	65,26	23,05	25,83	130,7

I did not perform any other coda transformations beyond adding the OpenMP pragmas.

Which loops did you parallelize?

1.1 + 1.2, 2.1 + 2.2, 3.1 + 3.2.

Inner and outer loop of each pair got merged into one and then parallelized.

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

Mostly everything sped up significantly. The function phaseAndMagnitude slowed down a little bit after parallelization. I am guessing this is from the overhead of parallelization, as the operations in this loop are quite simple and might be more efficient to just do sequentially. It also contains more complex instructions: multiplications, divisions and arctan function, so maybe the hardware does not have as many vectorization capable resources for those.

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?

My CPU has 6 cores with 2 threads per core. So 12 threads in total.

The performance did not scale with thread count. The runtime roughly halved, from 290ms to 142ms.

The program still has some sequential parts. The parallelization does bring some overhead, and since the contents of the loops are quite short, the scheduling and whatnot will take significant part of the time. In any case, we still see a huge improvement with multithreading, as the runtime dropped by more than half.

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

I used my own computer.

CPU:

Intel(R) Core(TM) i5-10600K CPU @ 4.10GHz

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

2.5 hours