ELEC-H-473
SIMD: Assignment 3

Tristan VANSPOUWEN, Theofanis RIGAS

May 9, 2017

1 Introduction

For this project, we have implemented multi-threaded C versions of the threshold and Max-Min image filters and we compare their performance to their single-threaded and SIMD counterparts.

1.1 Note on the use of POSIX threads

Our code is compiled on a Window machine (using mingw), yet we opted for POSIX threads (pthread.h) instead of the Windows API (windows.h). This is because for our benchmarks, we needed to use a thread barrier: we create all the threads, use the barrier to block them, start the clock and then have the main thread enter the barrier to release all the other threads. We do this in order to measure the time needed to process the images without taking into account the time needed to instantiate the threads. The Windows thread API does include this functionality, but the migwn implementation is bugged (see `https://github.com/Alexpux/mingw-w64/pull/2`), thus forcing us to use POSIX threads instead.

2 Benchmarks

Both the threshold and the Max-Min filters are applied to a raw 1048576 bytes (about 1Mb) grayscale image. We measure their performance on an Intel Core i5-4210 2.90Ghz CPU. This CPU has two physical cores and uses the Intel Hyper-threading technology (so 4 cores are visible to the OS). We use the `QueryPerfomanceCounter` function to obtain high quality measurements. We start the clock just before the actual processing of the image takes place (so after variable declarations and thread setup) and stop it after the last byte has been processed.

For each filter, we provide benchmarks for the single-threaded C implementation without any compiler optimization (Debug version) and compiled with the -O2 flag (Release version), the multi-threaded equivalent using 2,4 and 8 threads and finally the (single-threaded) SIMD implementation. The benchmarks have been averaged over 10 executions and are expressed in microseconds (*μsecs*) or milliseconds (*msecs*) accordingly:

Table 1: Threshold benchmarks

	C Debug	**C Release**	**SIMD**
Single threaded	2741.2 *μsecs*	1101.3 *μsecs*	454.5 *μsecs*
2 threads	2272.8 *μsecs*	1335.6 *μsecs*	-
4 threads	1889.6 *μsecs*	818.4 *μsecs*	-
8 threads	1905.0 *μsecs*	907.0 *μsecs*	-

Table 2: Max-Min benchmarks

	C Debug	**C Release**	**SIMD**
Single threaded	190.5 *msecs*	103.9 *msecs*	540.5 *μsecs*
2 threads	104.2 *msecs*	29.2 *msecs*	-
4 threads	83.3 *msecs*	25.4 *msecs*	-
8 threads	84.3 *msecs*	25.1 *msecs*	-

3 Remarks

Let us first discuss what we expected to see in terms of performance changes:

- A considerable speedup when comparing the single-threaded and the version using 2 threads: the CPU has two physical cores so processing of the image should be more or less parallelized.
- A smaller but noticeable speedup between the 2 and the 4 threads version: 4 logical cores do not offer the same advantages as 4 physical cores. The additional gain in speed should be around 30% at most.
- No difference between the 4 and the 8 threads version: adding more threads than those who can physically run in parallel on the CPU should not offer any speed advantages.

Our benchmarks largely confirmed our hypothesis:

- In most cases, using threads resulted in a significant improvement in performance compared to the single-threaded version. For example, using 2 threads sped up the optimized (Release) version of the Max-Min filter by 3.5x.
- Using 4 threads instead of 2 also resulted in better performance, offering speed ups between 15-30% as expected. For example, for the Debug version of the threshold filter the performance improved by 20% (from 104.2 *msecs* down to 83.3).
- In both the Release and the Debug versions, for both filters, the 8-thread version is slightly slower than the 4-thread version. As we noted above, this was to be expected given that the CPU has only 4 logical cores. The performance loss could be due to context-switching between the 8 threads.

3.1 Multi-threading vs SIMD

From our benchmarks, it is clear that even if using threads drastically improves the speed of the C implementations, especially when combined with compiler optimizations, the single-threaded SIMD versions remain much faster.

As an experiment, we also benchmarked a multi-threaded version of the SIMD threshold implementation but this resulted in no noticeable increase in speed (see accompanying source code).

3.2 Dividing the data between threads

In order to implement multi-threading, we had to divide the pixels of the image being processed between the worker threads. For the threshold filter we used a straightforward approach and divided the 1D array holding them in 2,4 or 8 equal parts. For the Max-Min filter, our algorithm works with 2D coordinates. We logically split the 1D array in matrices with as many columns as the width of the image and with $\frac{\text{image height}}{\text{number of threads}}$ rows.

3.3 Closing remarks

Finally, we should also note that when looking at the implementations as a whole, the process of spinning up the threads can consume significant time. So, if the size of the processed data is small and the filter simple (like the threshold for example), using multiple threads, even if it speeds up the actual processing of the data, can actually result in a worse overall execution time.