Assignment 6

Multiprocessor Programming 521288S

Miika Sikala 2520562, msikala18@student.oulu.fi

Task: OpenMP implementation of stereo-disparity algorithm (C/C++) and profiling.

Expected result: A working version of the implementation, a brief report (max 2-3 pages), saved output images all together in the form of a compressed folder (.zip file).

The report should contain about the task solved, brief description of your implementation, comparison of profiling information for pthread and OpenMP implementations and the final screenshot of your outputs asked to be displayed under the assignment.

Introduction

The implementation of OpenMP was extremely straightforward. I had to add `-fopenmp`. In Visual Studio, I activated Project Propeties > C/C++ > Language > **Open MP Support**. I also had to change for loop iterators from unsigned to signed integers, as OpenMP requires.

To decide, where to use OpenMP pragmas, I added it for all relevant for loop in Image.cpp and checked the execution times compared to fully sequential execution. It appears that OpenMP is able to optimize where to apply threading and how much since all the functions got significantly faster.

Usage

To change the target device, you can change the `COMPUTE_DEVICE` flag in "Application.hpp". The options are `TARGET_NONE` (no parallelization), `TARGET_PTHREAD` (CPU parallelization using pthread) `TARGET_GPU` (OpenCL on GPU), `TARGET_CPU` (OpenCL on CPU), and `TARGET_OMP` (OpenMP). You must re-compile the program for the change to take effect.

The application takes two to six input parameters, which are the image file names for the left and right images, and optionally some calculation parameters. See the example below, which uses image files `img/im0.png` and `img/im1.png`.

A makefile is provided for compiling with g++.

Compiling using *g++*:

```
make
```

Running:

```
stereo.exe img/im0.png img/im1.png 15 55 8 4

# Arguments: LEFT_IMG RIGHT_IMG [WINDOW_SIZE=9] [MAX_SEARCH_DIST=32]
#            [CROSS_CHECK_THRESHOLD=8] [DOWNSCALE_FACTOR=4]
```

Testing and benchmarking the implementations

The results (intermediate and final) are stored in "img" folder. Figure 2 shows the execution of the program for three different compute targets.

The benchmarking was done using a simple occlusion fill algorithm, which seeks the nearest non-zero pixel in square-like spiral.

Used configuration:

window size: 15
maximum search distance: 55
cross-checking threshold: 8
downscaling factor: 4

Execution times:

Operation	CPU (sequential)	CPU (Pthreads)	CPU (OpenMP)
Load + decode	1.093 s	841.150 ms	859.998 ms
Resize	**4.028 s**	**1.981 s**	**370.580 ms**
Convert to grayscale	**12.412 ms**	**9.687 ms**	**2.110 ms**
Encode + save	124.800 ms	123.382 ms	120.933 ms
Calculate ZNCC	**23.974 s**	**3.424 s**	**3.727 s**
Encode + save	114.879 ms	96.468 ms	94.465 ms
Cross-check	**3.415 ms**	**2.914 ms**	**1.834 ms**
Encode + save	46.402 ms	39.118 ms	39.544 ms
Occlusion fill	**5.028 ms**	**5.951 ms**	**2.355 ms**
Encode + save	43.910 ms	36.552 ms	43.912 ms
Total	**29.446 s**	**6.560 s**	**5.263 s**

As the table shows, the improvement using OpenMP is quite good. The ZNCC calculation takes about the same time as using pthreads (which is not a surprise, since the resulting implementation is the same). However, OpenMP's simple API makes it possible to easily implement it to most places with very minimal changes, giving notable improvement in resizing, grayscaling, occlusion fill, and cross-checking phases. The total execution time is therefore less than using pthreads.

It is to be noted that these benchmarking results were obtained using g++ with optimizations (`-O3` flag), while the previous assignments used g++ without optimizations. The difference is significant, due to a multitude of compiler optimizations.

The table below summarizes CPU and memory usage in different setups. Since threaded implementations utilize all 8 of my cores, the average (and peak) utilization is obviously better. OpenMP has the best since it uses multiple cores in all steps (including pre- and postprocessing). Peak memory usage is the same for all, and the memory consumption profile is very similar for all setups. Figure 1 shows the memory usage profile. If we wanted to decrease the peak memory usage, we could first load the first image, downscale it, and then load the second one.

Statistic	CPU (sequential)	CPU (Pthreads)	CPU (OpenMP)
Average CPU usage	9,7 %	69 %	77 %
Peak CPU usage	12,8 %	100 %	100 %
Memoru usage (peak)	69,1 MB	69,1 MB	69,1 MB

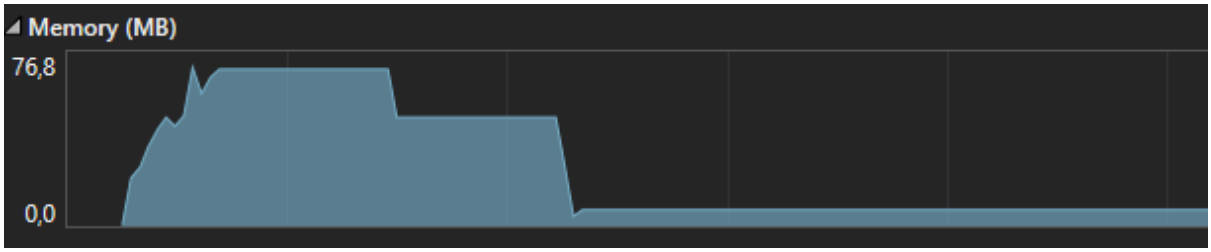

Figure 1. Memory consumption profile (similar for all setups). The peak is after loading both images, but before downscaling.

```
NOTE: The execution times include some printing to console.
Image manipulation is done using OpenMP (CPU).
Loading image... Done.
Decoding image... Done.
Loading image... Done.
Decoding image... Done.
        => time: 839.252 ms
Left image 'img\im0.png', size 2940x2016.
Right image 'img\im1.png', size 2940x2016.
Resizing image... Filtering image using a mask... Done.
Done.
Resizing image... Filtering image using a mask... Done.
Done.
        => time: 360.537 ms
Transforming image to grayscale... Done.
Transforming image to grayscale... Done.
        => time: 1.997 ms
Encoding image... Done.
Saving image... Done.
Encoding image... Done.
Saving image... Done.
        => time: 123.501 ms
Calculating ZNCC... Done.lating ZNCC... 90 % %lating ZNCC... 4 %
Calculating ZNCC... Done.lating ZNCC... 85 % %
        => time: 3.510 s
Encoding image... Done.
Saving image... Done.
Encoding image... Done.
Saving image... Done.
        => time: 93.729 ms
Performing cross-check... Done.
        => time: 1.704 ms
Encoding image... Done.
Saving image... Done.
        => time: 69.817 ms
Performing occlusion fill... Done.
        => time: 3.077 ms
Encoding image... Done.
Saving image... Done.
        => time: 41.354 ms
```

Figure 2. Capture of the output of the program using OpenMP configuration.

Discussion

The current implementation requires that the intermediate calculation results are transferred between the host and compute device memory multiple times. This is due to the modularity of the implementation (easy to make different computations). If we wanted to make the implementation as optimal as possible, we could minimize the number of transfers between the memories. This may be done in future implementations.

Reporting

Task	**Hours**
OpenMP implementation	2 h
OpenMP optimization	1,5 h
Benchmarking & writing the report	2 h
Total	**5,5 h**