Aleksander Skraastad

Assignment 7

TDT4200

OpenMP, CUDA and MPI

Autumn 2015

NTNU

Norwegian University of Science and Technology

Table of contents

Chapter 1

Multiple APIs

1.1 OpenMP

The OpenMP section was parallelized by using a single `#pragma omp parallel` directive.

Thread-local sectioning was done by calling `omp_get_thread_num()` and `omp_get_num_threads()`, and assigning senterY thresholds for each thread.

The first line was initiated outside the senterY loop, as it was causing issues.

As each thread writes to its own portion of the `imageOut` vector, no race conditions in writing to the output image will occur.

Initially, I used parallel sections in the main function, before seeing in the recitation slides that we were not supposed to alter the main method area. This should have been stated in the problem description as well, as it caused some extra work.

1.2 CUDA

The CUDA implementation is parallelized using 2D thread blocks of 32x32, yielding the maximum amount of threads per block (1024).

Grid layout is using the standard minimum block amount formula:

```
dim3 dimGrid(
    ceilf((width + 31) / 32),
    ceilf((height + 31) / 32)
);
```

Images are bytecopied directly from the PPMImage and into an unsigned char buffer on the device, where a float conversion kernel prepares the image vector.

Each kernel checks thread index bounds and instantly returns potential threads that go outside bounds.

An offset on the image vector is then calculated from the x and y components of blockIdx, blockDim and threadIdx.

All transformations are performed directly on the GPU, as buffers are already allocated. As soon as enough iterations are complete, the floatvector is run through the finalization kernel before being retrieved from the device and written to file.

After being written to file, kernel invocations for the next iterations are initiated.

1.3 MPI

MPI was parallelized using asynchronous send and receives, accompanied by synchronized waits.

The `MPI_Wait()` walls were used to account for data dependencies between the different iterations of the images.

Each process reads the image file and allocates all buffers, and ranks 1 through 3 write their own output file as soon as they have confirmation the necessary data has been transmitted.

The master rank only transmits to rank 1 and awaits send confirmation before exiting.

Rank 1 sets up a receive, before processing, and sends data to rank 2 before awaiting confirmation of data from master. Rank 1 writes to file on data receive confirmation.

The same approach is used on rank 2 and 3, as with rank 1, except that rank 3 does not send any data. It simply awaits receive confirmation before writing to file and exiting.

1.4 Benchmarks

1.4.1 OpenMP

The OpenMP version (benchmarked on Climb) performed as follows:

- Execution time: 1.38s
- Energy: 6.55j

- EDP: 9.03js

1.4.2 CUDA

```
its-015-07:~/tdt4200/ex07$ time ./newImageIdeaGPU 1

real    0m2.886s
user    0m0.865s
sys     0m0.232s
its-015-07:~/tdt4200/ex07$ time ./newImageIdeaGPU 1

real    0m2.869s
user    0m0.842s
sys     0m0.234s
its-015-07:~/tdt4200/ex07$ time ./newImageIdeaGPU 1

real    0m2.896s
user    0m0.793s
sys     0m0.318s
```

1.4.3 MPI

```
its-015-07:~/tdt4200/ex07$ time make runmpi
mpirun -n 4 ./newImageIdeaMPI

real    0m2.972s
user    0m1.742s
sys     0m0.103s
its-015-07:~/tdt4200/ex07$ time make runmpi
mpirun -n 4 ./newImageIdeaMPI

real    0m3.001s
user    0m1.750s
```

```
sys     0m0.151s
its-015-07:~/tdt4200/ex07$ time make runmpi
mpirun -n 4 ./newImageIdeaMPI

real    0m2.957s
user    0m1.678s
sys     0m0.103s
```