GPU Assignment1

Euan Rochester

March 2019

1 Shared implementation Details

1.1 Data types

Both implementations read the ppm file into an array of the `rgb` data structure (see Listing 1) which uses `unsigned char`s for each rgb element as this matches the size of the elements for the ppm file. However for holding the sum of pixels to take the average `long_long_rgb` is used, which uses `long long unsigned int`s for each element, as this gives space to sum 2^{48} `unsigned chars` and so should avoid overflow for most reasonable inputs (anything with $(width * height)/c < 2^{48}$).

In the OpenMP implementation partial global sums are using `long_rgb` allowing for cell sizes up to $\sqrt{2^{24}}$ (4096).

1.2 Error Checking

My implementation does not check that c is less than the image size, or that c is a power of two as it does not cause any errors. If $c > image\_size$ then the whole image is treated as a partial cell. The implementation does check to make sure $c > 0$ and $c < 4096$ as c being less than 0 would likely cause unusually errors, and c being greater than 4096 has the possibility to overflow the `big_rgb` struct.

My implementation checks that exactly $width * height$ pixels have been read from the file, and exits with an error message if more or less have been read, as the file is most likely corrupted in this case.

1.3 Other

As C does not support generics macros have been used for operations on the various `rgb` types, so that e.g. it is not necessary to define function variants for each pair of `rgb` types that might be passed to `rgb_add_assign`.

2 CPU

My CPU implementation is split into a number of separate nested for loops, one for each "stage" of the mosaic blur process. This is relatively fast, and is

simpler to reason about than the approach taken by the OpenMP version, as there are clearer separation of concerns in what each stage does.

The stages are:

Summing Cells are summed into the work buffer, which is sized at $\lceil \frac{image\_size}{c} \rceil$

Cell averages The size of each cell is calculated, and the sum is divided by this to get the average for that cell

Rescaling As the output is required to be the same size as the input the workbuffer is upsampled appropriately for output

3 OpenMP

Initially my OpenMP implementation was identical to my CPU implementation (but used `pragma omp for` on the outer loops), however in OpenMP looping over the rows and columns of the input image, as opposed to the cells requires the use of atomics to be correct, as different threads may execute different parts of the averaging of each cell. Use of atomics in this way made OpenMP significantly slower than CPU (see Table 1).

To rectify this I modified the OpenMP code to loop over the cells instead, this way only one thread will ever work on the output for one cell and so no atomics are needed.[1]

The OpenMP implementation still performs worse than CPU at small cell sizes ($c = 1$ for 2048×2048 images), presumably due to the overhead of launching threads, combined with the small serial portion for computing the global average. However using a secondary buffer to calculate cellwise sums and then average them in one thread for the global sum is still faster in the general case than using atomics and a single global sum counter.

4 Timing

Referenced times are from Linux on an i7 mobile processor using optimisation level -O3. Times on the high performance machines are generally better for OpenMP, but worse for single threaded CPU as would be expected, as the lab machines use Xeon CPUs which have many more somewhat lower performance cores than intel's desktop/mobile CPUs. While timing with `omp_get_wtime` provides some idea of overall timings it is difficult to get an idea of the performance of the algorithm over the complete range of inputs, and of the variance and average of times which are useful in performance analysis as single runs can easily be outliers inn terms of timing due, for example, to other processes using CPU time sporadically. For this reason I used an external tool called hyperfine[1] to

[1] Restructuring the CPU code to the same structure as the OpenMP code does appear to make the CPU version faster, however the gain was relatively minimal, and in this case it seemed better to leave the (subjectively) more readable code in place instead.

Implementation	Time
CPU	30-40ms
OpenMP	10-20ms
Initial OpenMP	≈200ms

Table 1: Approximate execution times on linux using -O3 and a mobile i7 processor at $c = 32$

Listing 1: Shared structures

```
typedef struct rgb{
    unsigned char r;
    unsigned char g;
    unsigned char b;
} rgb;

typedef struct long_rgb{
    unsigned int r;
    unsigned int g;
    unsigned int b;
} long_rgb;

typedef struct long_long_rgb{
    long long unsigned int r;
    long long unsigned int g;
    long long unsigned int b;
} long_long_rgb;
```

run a sweep over a range of input cells and images (as can be seen in Figures 1 and 2). While these benchmarks obviously do contain the time taken to load and save the data as they run against the program as a whole, rather than one function they give the benefit of calculating averages and standard deviations. From the figures we can see that for larger images my OpenMP implementation is almost always better than CPU, however for smaller images there is more overlap in the range of times taken. However the average for OpenMP remains marginally better, even for the smaller images.

References

[1] David Peter. *Hyperfine: A command-line benchmarking tool*. `https://github.com/sharkdp/hyperfine`. 2018–2019.

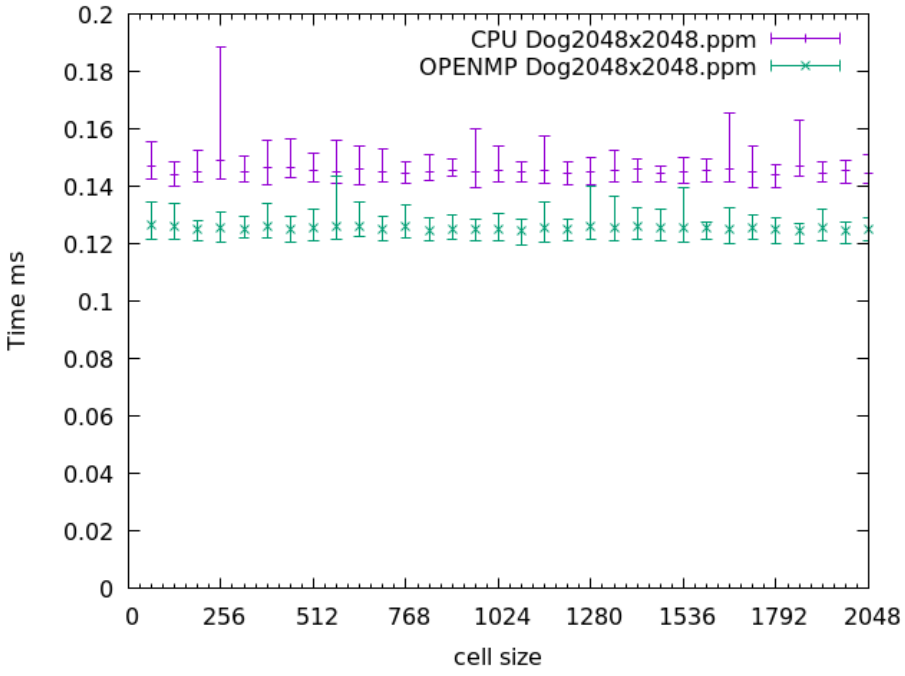

Figure 1: Benchmarking on Dog2048x22048.ppm

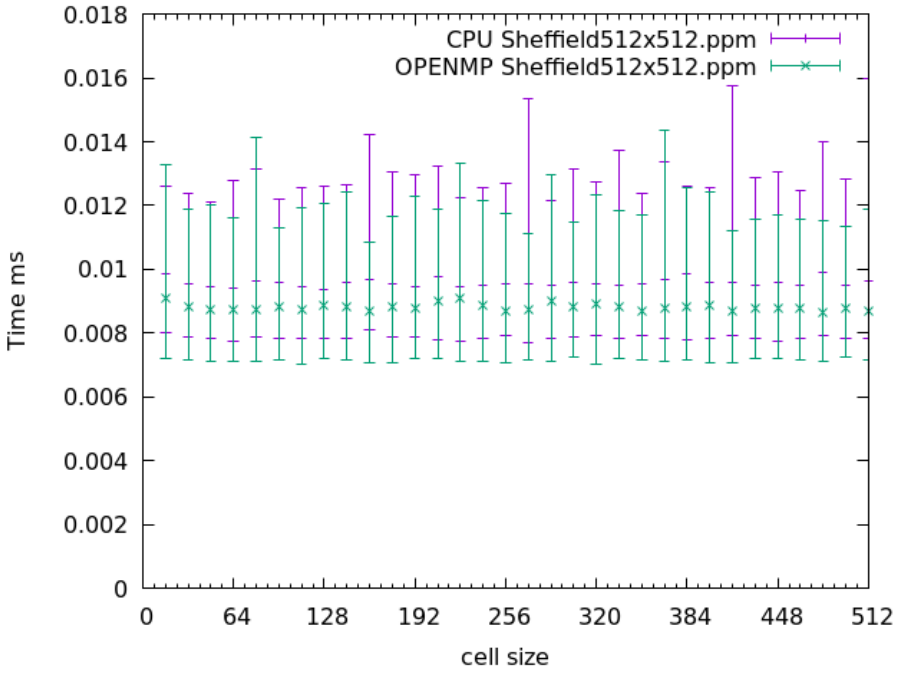

Figure 2: Benchmarking on Sheffield512x512.ppm