PMMS 2020: Assignment 4

Thijs Meijerink
2661163
thijs.meijerink@gmail.com

1 Parallelisation of heat dissipation code with CUDA (4.1)

1.1 Introduction

The heat dissipation program is parallelized using CUDA. The target is an NVidia Titan X Pascal on the DAS-5.

1.2 Kernels

Separate kernels are written for the main computation and the copying of the halo cells. The halo kernel copies a single halo cell per thread, and the main kernel computes the value of a single grid cell per thread.

The block size giving the best performance was empirically determined at 128. Also, invoking the kernel in a 2D grid was tried but did not improve performance over the 1D grid.

2 Experiment with your GPU heat dissipation code (4.2)

From all the parallel CPU heat dissipation programs implemented in previous assignments, the block-scheduled OpenMP program performed the best. See figure 1 for the runtime and GFLOP/s of this version, executing 1000 iterations. To compare against these results, the CUDA implementation was executed with the same input image sizes and iteration count. The complete runtime, including the memory transfers, was measured. See figure 2 for the runtime and GFLOP/s results.

We can see that the CUDA program beats the OpenMP program for input images larger than `1000x1000`. This clearly shows the advantage of parallelizing over a very large number of threads when dealing with big problem sizes.

3 Histogram in CUDA (4.3)

3.1 Introduction

Different CUDA programs for calculating the histogram of an image are implemented. As before, the input is through `.pgm` files. In all implementations, each thread reads a single pixel, and updates the corresponding bucket.

3.2 Atomics (kernel A)

In the first version, the bins reside in global device memory, and are incremented using atomic addition.

3.3 Atomics and shared memory (kernel B)

In order to limit contention for the bins in global memory, a separate histogram per thread block is kept in shared memory. Again, the bins are updated using atomics. A `__syncthreads` barrier is places after this operation, and subsequently the sub-histograms are combined, using atomics, to the complete histogram in global memory. The first 256 threads in a block all write one bin to global memory.

Figure 1

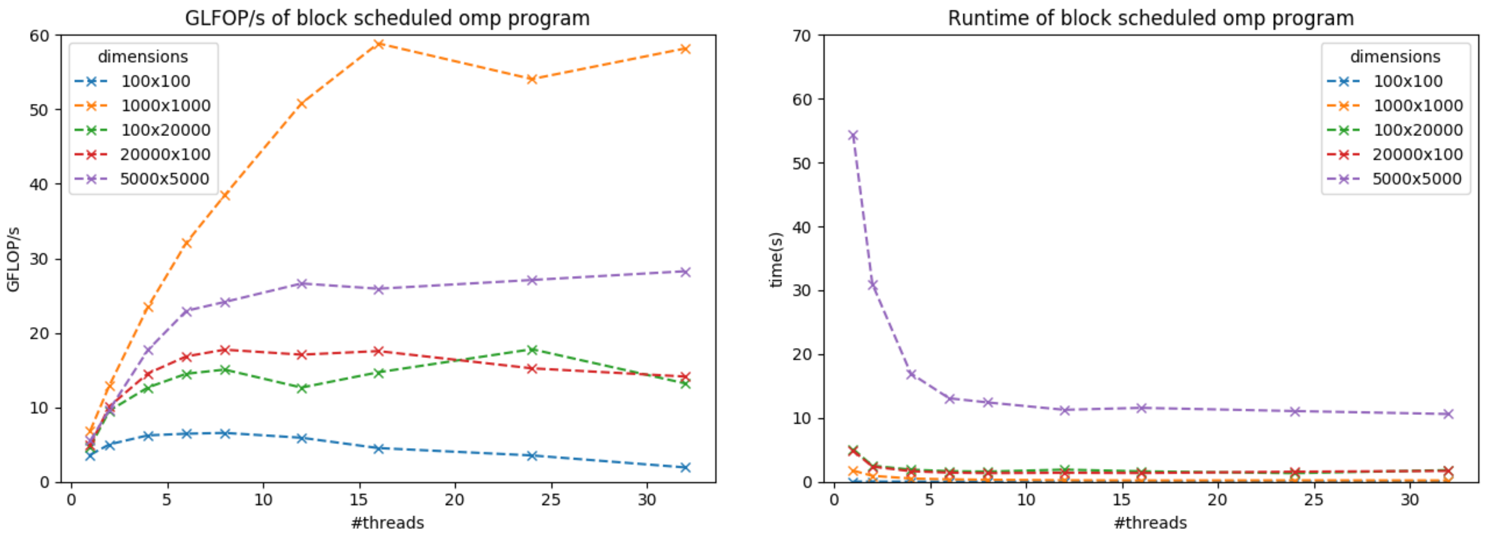
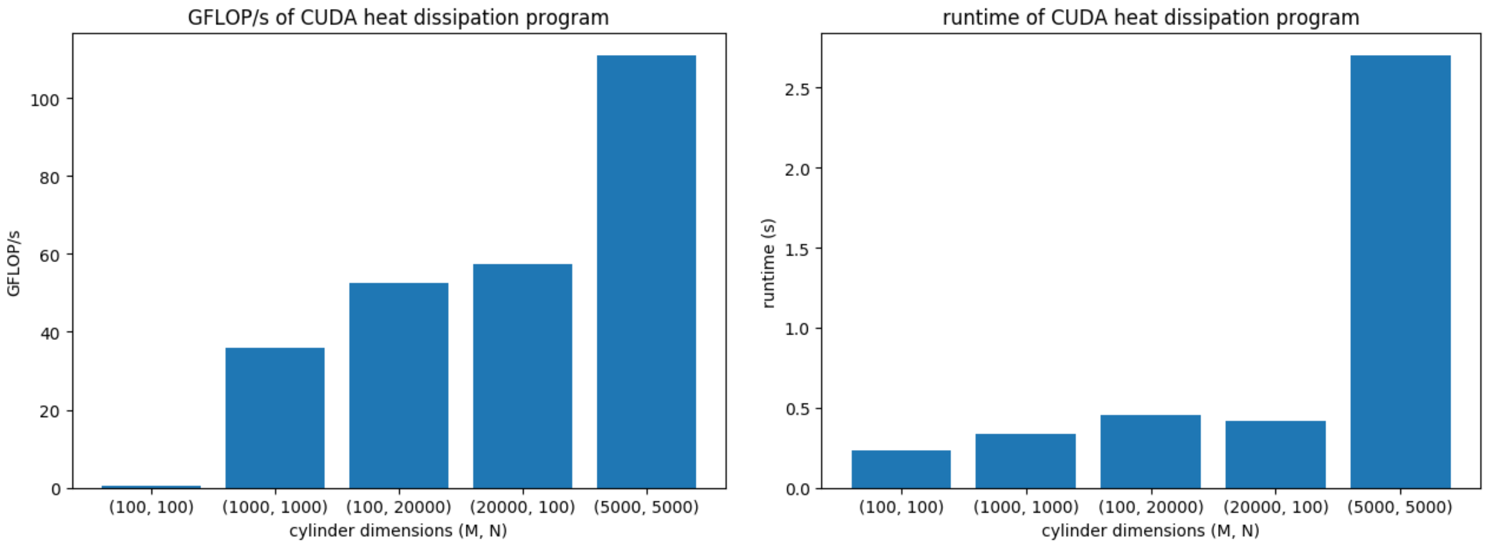


Figure 2



3.4 Hypothesis

It is expected that kernel B performs better than kernel A for two reasons. Firstly, since the sub-histograms in kernel B are shared between fewer threads than the global histogram in kernel A, there will be less contention for the bins. Secondly, updating the bins will be faster since they reside in shared memory.

4 Experiment with your histogram in CUDA (4.4)

The two kernels were used to compute the histogram of the image `areas_2000x2000.pgm`, as generated by the provided makefile. The GPU is a Titan X Maxwell. See the table below for the resulting runtimes. The best CPU histogram implementation from assignment 3, `histo_avoiding_mutual_ex` running on 16 cores, is included for comparison.

program	runtime (kernel)	runtime (memory)
CPU	0.000854s	
kernel A	0.005379s	0.00246
kernel B	0.001016s	0.00177

As can be seen from the table above, kernel B does beat kernel A as expected, however the fastest GPU implementation does not beat the fastest CPU implementation. This can be explained by noting that in the GPU implementation, although it was reduced, bin contention does occur, whereas the CPU program avoids it entirely.

5 Smoothing filter (4.5)

5.1 Introduction

Two-dimensional convolution is implemented in CUDA in order to apply a smoothing filter to an image. The program is tested on an NVidia Titan X Maxwell, on a `10000 x 10000` image. The following sections describe a number of optimizations applied (cumulatively) to the naive CUDA kernel, as well as the resulting performance measurements.

5.2 Optimizations

5.2.1 Using constant memory

The filter is placed in constant memory on the GPU, using the `__constant__` keyword.

5.3 Results

The following table lists the runtime of the kernel in seconds, not including the time used for memory transfers to and from the GPU, which is roughly 0.34 seconds and equal for all versions of the kernel.

Program	Runtime
Sequential	3.44804
Naive	0.20895
Const. mem.	0.05849