GPU Assignment2

Euan Rochester

May 2019

1 Introduction

2 Iterations

2.1 Naive port from openmp

I simply ported the openmp implementation so that each cell ran on it's own cuda thread. I did not expect this to perform well at all as it has diverging flow control within the kernel, and made no use of shared memory, however it performed fairly well. One reason I did not use this kernel for my final code is that it had some issues on non-square and large images that I could not figure out the root cause of to fix.

2.2 Split naive kernel into gather and scatter kernels

I simply split the previous kernel into two separate kernels, a gather kernel which calculates the average for each cell, and a scatter kernel which resizes the image to the correct dimensions. The slight speedup gained here was might be due to lower register use, as warps and registers are reported as the block limit reason by nsight.

2.3 Struct of arrays

Implementing struct of arrays on the implementation from section 2.2 causes an increase in overall runtime, but a significant decrease in cuda runtime (down to 0.15ms). Even when using `cudaMemcpy2D` for efficient strided copies memory management on the host is a significant overhead when using struct of arrays.

2.4 Split horizontal and vertical gather operations

Here I split the gather kernel into first doing a parallel reduction horizontally, and then a parallel reduction vertically within each cell (as illustrated in fig. 4). I expected this to give a speedup as the kernels would now be using shared data in a conflict free manner. However this was not the case. In fact profiling reported that the split kernels would have a maximum occupancy of only 50%, this was due to too small a block size (see figure 2).

2.5 Fused gather and scatter

Here I fused the split gather kernels so that they operated over a full c cell of pixels (see figure 1), and also implemented the scatter kernels using the inverse of the gather method. This gave me $> 90\%$ occupancy as I significantly increased the kernel size to be the size of a full cell. One downside to this approach is that the cell size can't be very large.

2.6 Compromise kernel

I realised that the fused kernel needed quite a large amount of shared memory allocated for large values of c, which meant values of $c > 56$ would not have enough memory on many GPUs. In order to fix this, but keep the performance improvements the larger block size of the fused kernels had I implemented a compromise approach where the linewise implementation was used, but I altered the kernel to work with $blockDim.y > 1$ and calculated the largest even division of c that still would have enough shared memory to be instantiated (see figure 3). A factor involving

the max shared memory per block was hard-coded to speed up developement, but there is likely a better way to calculate this at runtime.

Unfortunately as I did not realise the inadequacies of the fused kernels until fairly late on I did not have much time left to optimise this set of kernels, so it is not the fastest (at least on smaller images), but it is probably the most correct for the brief.

2.7 Streams

I experimented with both the stream and graph apis to try to run the scatter, and averaging code in parallel as neither depends on the other, but could see little or no speed increase with the stream api, and was unable to use the launch profiler with the graph api (there were simply no results displayed), so I focused on optimising the kernels themselves.

3 Timing

Time taken to copy to and from the GPU is included in my timings as this is an unavoidable overhead associated with using CUDA for a task, and so should be taken into account when finding the optimal framework to solve a problem.

4 Other Notes

An additional finding I made is that whether the computer running my code was on battery or mains power had far more effect on the timing of my implementation than any optimization attempt I made did. Running on battery power could increase execution time by at least 400% over mains power.

5 Conclusion

The compromise kernel seems the most promising to follow up on, as it is versatile in terms of size of C, is still relatively fast, and (subjectively) has the cleanest code. While it is not the overall fastest iteration, it is plausible it could be improved perhaps using the stream api to run multiple kernels in parallel, or replacing the $\text{global}_a vgkernelwithacleverlystridedmemcpy, asitisdifficulttogetgo$

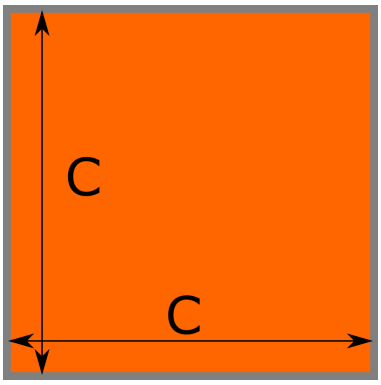

Figure 1: Block dims for fused

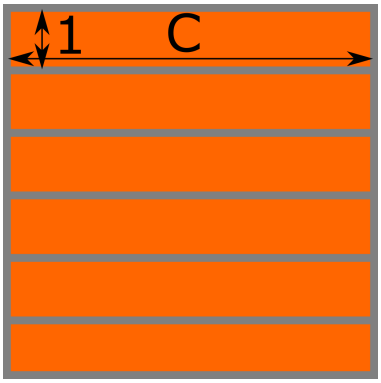

Figure 2: Block dims for original

Iteration	Time	occupancy	reported block limit reason
Naive port from openmp	2ms (0.47 ms in cuda)	39%	warps, registers
Split kernel into gather and scatter	2ms (0.43 ms in cuda)	gather: 21% scatter:22%	warps, registers
Struct of arrays	49ms (0.14 ms in cuda)	gather: 22% scatter:22%	warps, registers
Split row and col gather	8ms (5.55 ms in cuda)		
Fused gather and scatter	2.25 ms (0.31 ms in cuda)	gather: 94% scatter: 95% row_reduction: 95% col_reduction: 92%	warps, blocks
Compromise Kernel	2.64 ms (1.43 ms in cuda)	col_scatter: 90% row_scatter: 95% global_avg: 48%	warps, blocks

Table 1: Performance

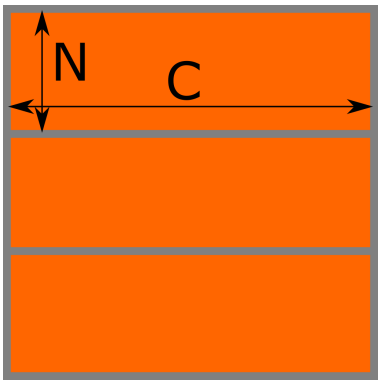

Figure 3: Block dims for compromise

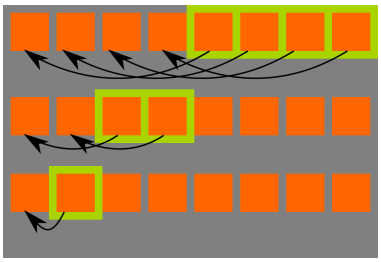

Figure 4: Reduction pattern