Parallel Programming Practical
GPU Assignment - CUDA image processing

Laurens Verspeek
VUnetID: 2562508

February 3, 2016

1 Introduction

The goal of this assignment was to learn how to use many-core accelerators, GPUs in this particular case, to parallelize data-intensive code. Three different image processing algorithm are implemented in CUDA:

- grayscale conversion and darkening
- histogram computation
- smoothing

The setup and design overview for each algorithm are explained and the optimisations that were applied to the kernel to achieve a better speedup. The speedups compared to the sequential version (GPU vs. CPU) will be displayed in a speedup graph for each algorithm. The sequential and the parallel algorithms were benchmarked on the DAS-4 system with a Nvidia GTX 480 GPU for the parallel algorithm, which has a theoretical peak performance of 1350 GFLOPS (single precision) and a Memory Bandwidth of 177.4 GB/s. The CPU used for the sequential algorithm is an Intel Xeon E5620 which has a clock rate of 2.40GHz. The input images for each algorithm are RGB images, this means that every color is rendered adding together the three components representing Red, Green and Blue.

2 Grayscale conversion and darkening

The gray value of a pixel is given by weighting the three values of RGB and then summing them together. The formula to do this is: $gray = 0.3 * R + 0.59 * G + 0.11 * B$ To darken this grayscale image, the obtained value for the grey pixel is multiplied by 0.6. The sequential version ot this algorithm simply computes for each pixel, one after the other, the corresponding gray value.

2.1 Parallelization

In the first try space got allocated (3 * width * height * sizeof(unsigned char)) in the global memory of the GPU (so that all the threads can access this variable) the input image is copied to this global variable. Also, space is allocated for the output image (width * height * sizeof(unsigned char)). Now we can run the kernel. In this first version, each thread computes the gray value for just one pixel. The threads were organised in a two-dimensional block structure, as this was easy to see which index for the thread should be used for which pixel, as an image is also two-dimensional.

2.1.1 Optimisation: One-dimensional blocks

The first optimisation that was applied is to not use two-dimensional blocks, but to just use one-dimensional blocks. With the one-dimensional blocks the kernel achieved better speedups. This might be explained due to the fact that there are less operations needed

to calculate the index of the pixel of the image that the threads need to process when using one-dimensional blocks. The optimal number of threads per block is calculated with the CUDA GPU Occupancy Calculator. See the grid-stride optimisation section for more info about the right number of blocks to launch.

2.1.2 Optimisation: Memory coalescing

The next optimisation that was applied is coalescing memory access patterns for the input image. Coalescing is used to mean making sure that threads run simultaneously, try to access memory that is nearby. I first tried to place all the red, green and blue pixel values for 1 pixel next to each other, instead of first all the red values, then all the green values and then all the blue values. This was already showing some better speedups. Even better speedup was achieved when all the red, green and blue values were seperated from each other in three different variables. These three variables are copied to the reserved space in the GPU global memory. The three variables are aligned very nicely with each other, which makes the access to a pixel value (red green and blue) faster.

2.1.3 Optimisation: Grid Stride

The last optimisation that was applied is the use of a grid stride loop. Instead of completely eliminating the loop when parallelizing the computation, a grid-stride loop in the kernel was used, so that every thread calculates not just one pixel, but grid-size pixels at a time. By using a loop with stride equal to the grid size, we ensure that all addressing within warps is unit-stride, so we get maximum memory coalescing, just as in the version where each thread computes one pixel. This also makes the algorithm scalable for very large images. It is useful to launch a number of blocks that is a multiple of the number of multiprocessors on the device (GTX480 has 32 SMs), to balance utilization.

2.2 Performance

In table 1 the results of the benchmarks on DAS-4 are displayed. To visualize this data, the speedup is shown in figure 1. Note that for the speedup graph, only the GPU kernel time is taken into account and compared to the seqential CPU time. The overhead to setup CUDA and to allocate and copy the necessary variables to and from the global memory of the GPU and to free them at the end of the run are excluded. However, these values can be found in the table. Each time entry in the table is the average of three measurements.

In figure 1 can be seen we achieve good speedups with almost all images. the smallest image, image00, has the worst speedup, as there are simply not enough pixels in that image to occupy all the 480 cores of the GPU. We also see that the parallel algorithm is scalable, as the more pixels the image has, the better the speedup of that image is. The grid stride loop within each thread makes sure that it can handle large image sizes. With the original method, where each thread computes one pixel, it would be difficult

Image	CPU	GPU:init	GPU:alloc.	GPU:copyDevice	GPU:kernel	GPU:copyHost	GPU:free
image00	0.0032s	0.016227s	0.053260s	0.000450s	0.000099s	0.000232s	0.000108s
image01	0.0363s	0.016743s	0.048721s	0.003125s	0.000284s	0.001873s	0.000289s
image02	0.0445s	0.016732s	0.048980s	0.003152s	0.000343s	0.001971s	0.000341s
image03	0.2274s	0.021737s	0.055181s	0.027364s	0.001485s	0.010645s	0.000389s
image04	0.2009s	0.021165s	0.055146s	0.018427s	0.001080s	0.007344s	0.000346s
image05	0.3477s	0.016635s	0.047889s	0.038121s	0.001833s	0.013749s	0.000413s
image06	0.6861s	0.016611s	0.048394s	0.070199s	0.003415s	0.026631s	0.000432s

Table 1: Times for the sequential and parallel version of grayscale conversion and darkening

to achieve good speedups for large images with a lot of pixels, as there are simply too few threads available in the GPU.

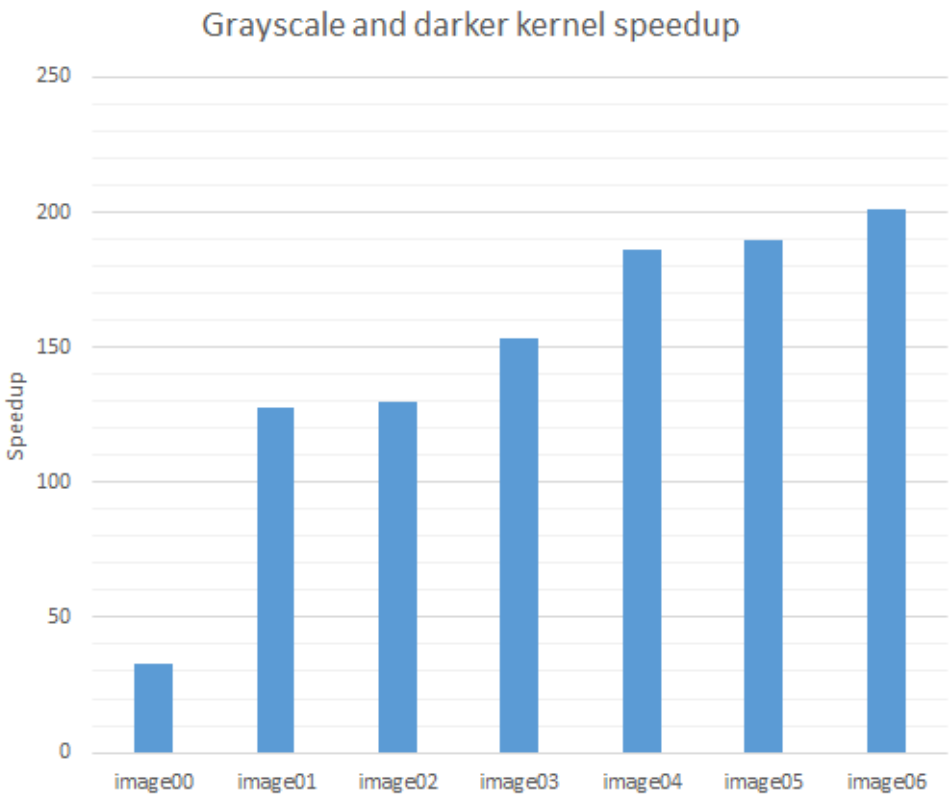

Figure 1: Speedup of the parallel cuda implementation of the grayscale conversion and darkening algorithm compared to the sequential algorithm

3 Histogram computation

The histogram measures how often a value of gray is used in an image. The histogram is computed by simply counting the value of every pixel and increment the corresponding counter (after converting the image to grayscale, which is explained in the previous section). The are 256 possible values of gray. The sequential version ot this algorithm simply computes for each pixel, one after the other, the corresponding gray value and incrementing the corresponding counter.

3.1 Parallelization

The same principles for optimal parallelization as in the previous grayscale kernel are applied for this kernel. However, in this algorithm, there are 256 different gray values in the image, which should be counted in a global histogram. See the section *Optimisation: Shared Memory* for the new optimalisation used for this kernel.

3.1.1 Optimisation: One-dimensional blocks

The first optimisation that was applied is to not use two-dimensional blocks, but to just use one-dimensional blocks. Just as with the grayscale kernel, the kernel achieved better speedups with one-dimensional blocks. This might again be explained due to the fact that there are less operations needed to calculate the index of the pixel of the image that the threads need to process when using one-dimensional blocks. The optimal number of threads per block is calculated with the CUDA GPU Occupancy Calculator.

See the grid-stride optimisation section for more info about the right number of blocks to launch.

3.1.2 Optimisation: Memory coalescing

The next optimisation that was applied is coalescing memory access patterns for the input image. Coalescing is used to mean making sure that threads run simultaneously, try to access memory that is nearby. From the previous kernel I already knew that to place all the red, green and blue pixel values for 1 pixel next to each other results in better speedup, but even better speedup is once again achieved by seperating all the red, green and blue values from each other in three different variables. These three variables are copied to the reserved space in the GPU global memory.

3.1.3 Optimisation: Grid Stride

Just as in the grayscale kernel, this kernel uses a grid stride loop. Instead of completely eliminating the loop when parallelizing the computation, a grid-stride loop in the kernel was used, so that every thread calculates not just one pixel, but grid-size pixels at a time. By using a loop with stride equal to the grid size, we ensure that all addressing within warps is unit-stride, so we get maximum memory coalescing, just as in the version where each thread computes one pixel. This also makes the algorithm scalable for very large images. It is useful to launch a number of blocks that is a multiple of the number of multiprocessors on the device (GTX480 has 32 SMs), to balance utilization.

3.1.4 Optimisation: Shared Memory

A new optimisation that is used for this kernel (compared with the grayscale kernel) is the use of shared memory. The counting of the different gray values (256 integers) is a shared memory structure, because the results of each thread should be combined into one histogram. This can be done the simple way, by using global memory and using atomics adds. The problem is that this gives a slow speedup, as each atomic add needs to be serialized. Therefore, shared memory is used to temporary store the counters of the current thread and at the end of the threads computation, the counters stored in the shared memory are added to the global histogram with an atomic add. This results in a better speedup for the kernel (which can be found in the section *Results*).

3.2 Performance

In table 2 the results of the benchmarks on DAS-4 are displayed. To visualize this data, the speedup is shown in figure 2. Note that for the speedup graph, only the GPU kernel time is taken into account and compared to the seqential CPU time. The overhead to setup CUDA and to allocate and copy the necessary variables to and from the global memory of the GPU and to free them at the end of the run are excluded. However, these values can be found in the table. Each time entry in the table is the average of three measurements.

In figure 2 can be seen we achieve good speedups for most images (varying a bit

Image	CPU	GPU:init	GPU:alloc.	GPU:copyDevice	GPU:kernel	GPU:copyHost	GPU:free
image00	0.0028s	0.016655s	0.055300s	0.000502s	0.000093s	0.000244s	0.000162s
image01	0.0282s	0.020544s	0.055546s	0.003227s	0.001198s	0.002344s	0.000375s
image02	0.0389s	0.016716s	0.053736s	0.004651s	0.001778s	0.002537s	0.000535s
image03	0.2659s	0.016297s	0.048314s	0.027600s	0.004848s	0.010649s	0.000422s
image04	0.1971s	0.016650s	0.049669s	0.019760s	0.013299s	0.008552s	0.000525s
image05	0.3550s	0.016629s	0.048772s	0.024360s	0.00449s	0.012913s	0.000425s
image06	0.5183s	0.020986s	0.055207s	0.051332s	0.01021s	0.029231s	0.000576s

Table 2: Times for the sequential and parallel version of the histogram algorithm

per image, possibly due to the variety in gray values). However, the speedups are less good compared with the darker kernel speedups. This can be explained due to the fact that it needs to synchronize all the threads two times (Barrier synchronization, see shared memory optimisation for more information). Although it is not as clear as in the darker kernel speedups due to the more varying speedups for this kernel, we can assume that it is scalable, as it uses the same principles as the darker kernel and for the larger images (image05 and image06) it achieves better speedup than most smaller images. Also, the grid stride loop within each thread makes sure that it can handle large image sizes. With the original method, where each thread computes one pixel, it would be difficult to achieve good speedups for large images with a lot of pixels, as there are simply too few threads available in the GPU.

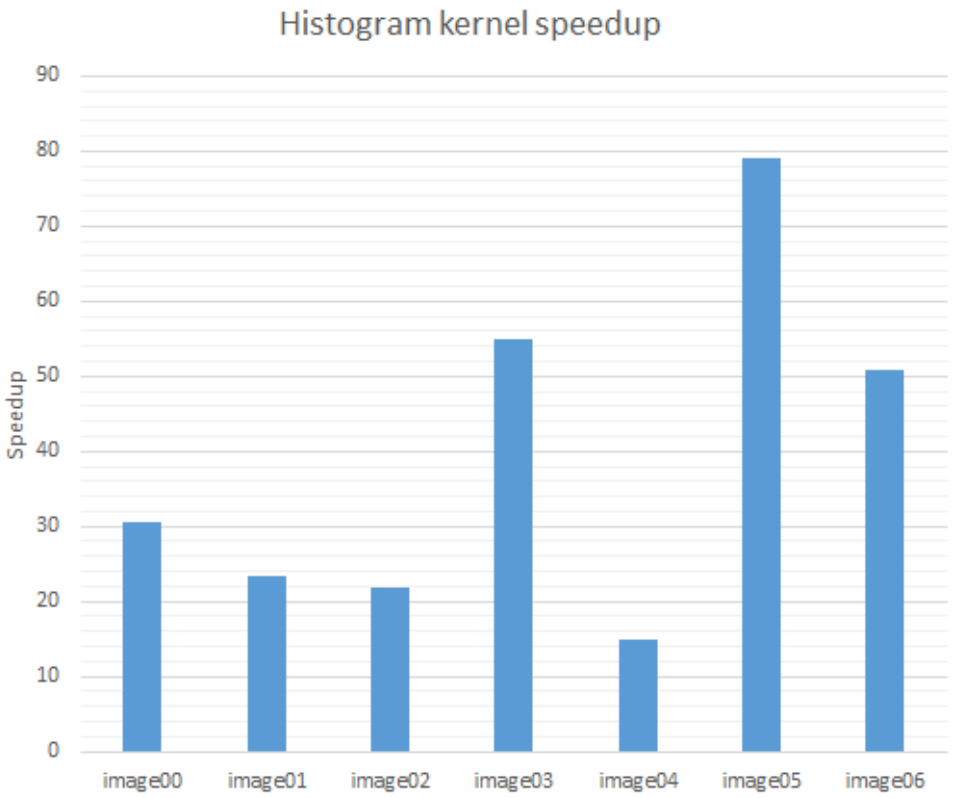

Figure 2: Speedup of the parallel cuda implementation of the histogram algorithm compared to the sequential algorithm

4 Smoothing

Smoothing is the process of removing noise from an image by the means of statistical analysis. To remove the noise, each pixel is replaced by a weighted average of its neighbours. In this way small-scale structures are removed from the image. We are using a triangular smoothing algorithm, i.e. the maximum weight is for the point in the middle and decreases linearly moving far from the center.

4.1 Optimisations

In contrast with the previous two kernel, this kernel achieves better speedups by using two-dimensional blocks for the threads. This is due to the fact that this smoothingr algorithm deals with square areas (filter) of the input image. By using two-dimensional blocks, the parallel algorithm can efficiently share memory between threads and prevent many global memory accesses. The stride used for this kernel is equal to the size of

the spectrum dimension. This was faster than using three-dimensional blocks, where each thread computes one pixel. I also tried to use shared memory, but due to the need of multiple synchronization points, this was slower than the straight forward global memory method, so I eventually stuck with the global memory version.

4.2 Performance

In table 3 the results of the benchmarks on DAS-4 are displayed. To visualize this data, the speedup is shown in figure 3. Note that for the speedup graph, only the GPU kernel time is taken into account and compared to the seqential CPU time. The overhead to setup CUDA and to allocate and copy the necessary variables to and from the global memory of the GPU and to free them at the end of the run are excluded. However, these values can be found in the table. Each time entry in the table is the average of three measurements.

In figure 3 can be seen we achieve good speedups with all images. In contrast with the darker kernel speedups, the smallest image, image00, also has a good speedup. This can be explained by the fact that there are more computations that needs to be done for each pixel compared to the darkel and histogram kernel, so it is still 'heavy' enough to occupy all the 480 cores of the GPU, even with small images. We also see that the parallel algorithm is scalable, as the speedup is also good for larger images.

Image	**CPU**	**GPU:init**	**GPU:alloc.**	**GPU:copyDevice**	**GPU:kernel**	**GPU:copyHost**	**GPU:free**
image00	0.0902s	0.020840s	0.055622s	0.000504s	0.000751s	0.000637s	0.000217s
image01	1.0087s	0.021287s	0.054817s	0.003058s	0.007085s	0.003680s	0.000310s
image02	1.2637s	0.016604s	0.053503s	0.004274s	0.009926s	0.005227s	0.000300s
image03	6.4974s	0.020254s	0.055184s	0.027848s	0.066680s	0.029919s	0.000342s
image04	7.0454s	0.016757s	0.048363s	0.012201s	0.047494s	0.019511s	0.000303s
image05	11.5709s	0.016880s	0.053643s	0.037899s	0.087618s	0.045454s	0.000433s
image06	26.1040s	0.016539s	0.047918s	0.069986s	0.172938s	0.077570s	0.000427s

Table 3: Times for the sequential and parallel version of the smoothing algorithm

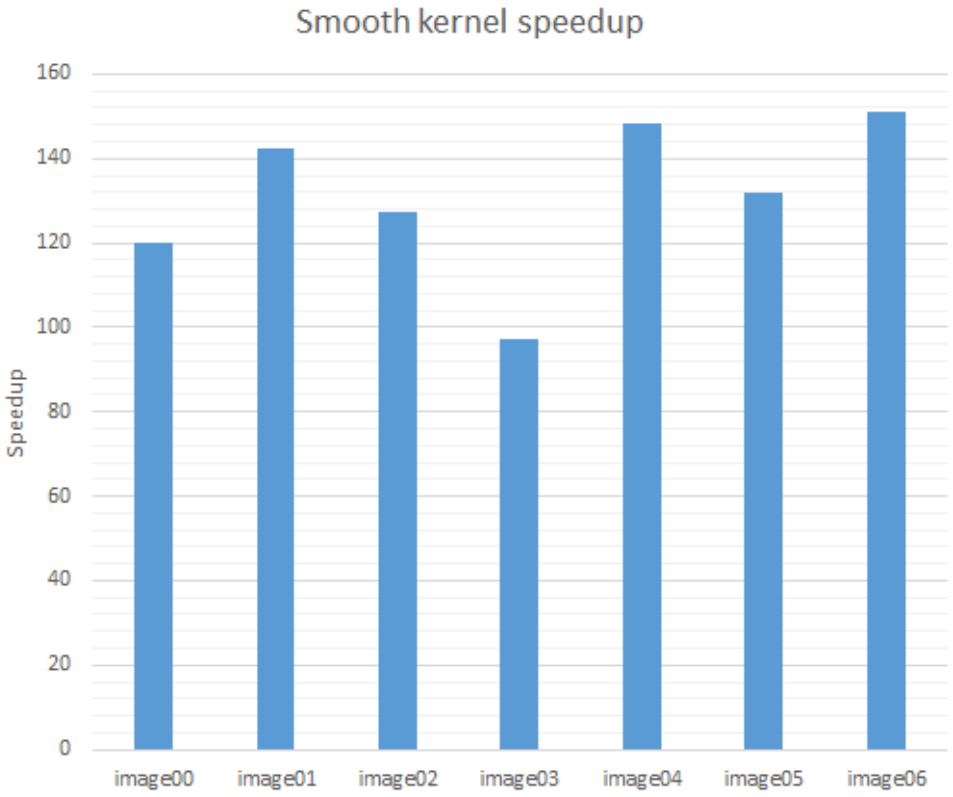

Figure 3: Speedup of the parallel cuda implementation of the smoothing algorithm compared to the sequential algorithm