Lab Project Report[*]

Assignment 4

Christopher Esterhuyse, Mihai-Andrei Spadaru

1 Introduction

For large computational problems, we often care most about runtime and speedup. At first glance it makes sense to optimize the speed of operations to maximize these metrics. For problems of scale, concentrating on increasing *throughput* has the desired effect. GPUs are well-suited to this task, offering an order of magnitude more worker threads than were available in previous reports concerning parallelism with OpenMP and PThreads, but in return requiring the management of an entirely separate physical device. In this report, we seek to achieve meaningful speedups for two problems concerning highly regular data (in both cases, necessitating a sequence of arithmetic operations per element in a large input matrix). Plots shown in sections 2 and 3 are complemented by exhaustive tables in the Appendix.

For its experiments, this work uses one DAS4 node from the University of Delft, equipped with a single Nvidia GeForce GTX480 'Fermi'. The host machine itself has 16 logical cores: 2 sockets, 4 cores per socket, 2 threads per core.

2 Histogram

2.1 The Problem

This section explores a relatively simple program: Given an input image of grayscale pixels (with values in range 0–255), compute the histogram of pixel values. This problem was explored in a previous work. However, this time, the intention is to exploit the parallelism achievable using a GPU in an attempt to achieve higher speedup than before.

[*]This report was submitted for the lab assignment of the 2018 edition of the Universiteit van Amsterdam Programming Multi-Core and Many-Core Systems master course.

2.2 Implementations

In an attempt to find the best possible solution to this problem, we explore two different GPU-centric implementations of the Histogram program. In both versions, each thread visits a single input image matrix cell. The two implementations differ on what they do next:

1. **Global** (G)
   There is a single histogram in the GPU's *global* memory. Each thread directly increments the desired cell using an atomic addition. The global histogram is returned as the result when all workers have finished.

2. **Shared** (S)
   Each thread block is given its own histogram structure, initialized at zero, placed into the GPU's block-local *shared* memory. Threads use an atomic add to increment the appropriate counter of their own shared histogram. When the image has been completely visited, thread workers use atomic adds to *fuse* their local histograms into one global histogram, which is returned as the result.

In the previous report, the *local-data* version of the PThreads Histogram program created a local histogram for each of its 16 worker threads; this allowed workers to incriment histogram counts without locking. There was then a final merge at the end. Using the same approach directly would result in a too-finely grained distribution of local histograms. This would both require too much memory and just move the bottleneck to the histogram-fusing step. S attempts to *adapt* this approach by using a shared histogram *per block* instead. This should dramatically reduce contention in the histogram building step instead of eliminating it; in return, the merge step will remain feasible.

2.3 Performance Results

To understand the performance of the GPU histogram implementations, they were run with various input images with an interesting variety of properties. The hope is to build a more granular intuition for *when* there is speedup/slowdown, rather than just *if* there is speedup/slowdown.

Firstly, images with drastically different *sizes* were considered. Fig. 1 shows performance from runs over input images with sizes[1] in $\{9.025, 90.000, 900.601, 9.000.000\}$[2]. As expected, the overall runtimes of all programs can be seen to scale super-linearly (in fact, quadratically) with an image's side length (ie. linearly with respect to the number of image pixels). The same data is plotted in fig. 2 logarithmically to make the results more distinguishable. It can be seen that while input size does impact performance, the effect is more or less proportional across implementations and steps. Memory transfer time for both G and S are always comparable, and S was significantly faster at solving the problem. For reference, the sequential implementation took longer than either of G and S's runtimes in all circumstances. Note that, while comparable, both measures of GPU 'memory-time' seem to be proportionately higher for smaller problems. The reason for this is twofold: Firstly, compute operations are subject to speedup as a function of cached memory accesses. With smaller problems (but the same cache-line size), there are overall fewer cache-misses; this essentially removes some compute-time for smaller problems. Secondly, the host-device memory copies are performed a constant number of times per run, regardless of the size of the problem. Thus, any accompanying overhead is *constant*, and (proportionately) removes some memory-time for larger problems.

Next, images of different *types*[3] were interesting to understand, as the problem is intended for use on an unspecified domain of possible input images.

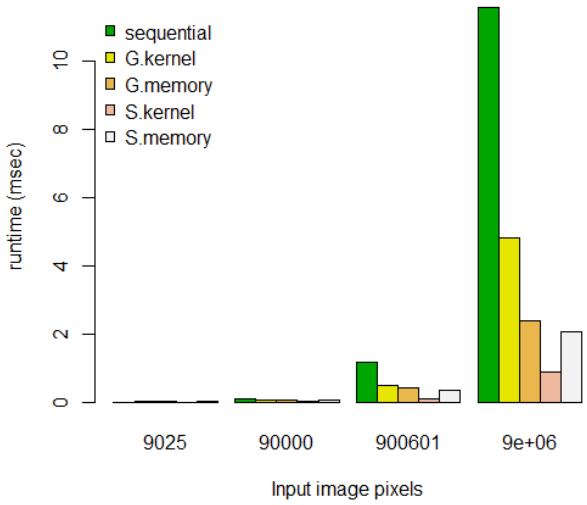

Figure 1: Runtime of the CPU-sequential and both GPU-parallelized implementations of Histogram. The GPU implementations are split into their distinct steps. Each bar group represents runtimes for inputs of different sizes, for random-type images.

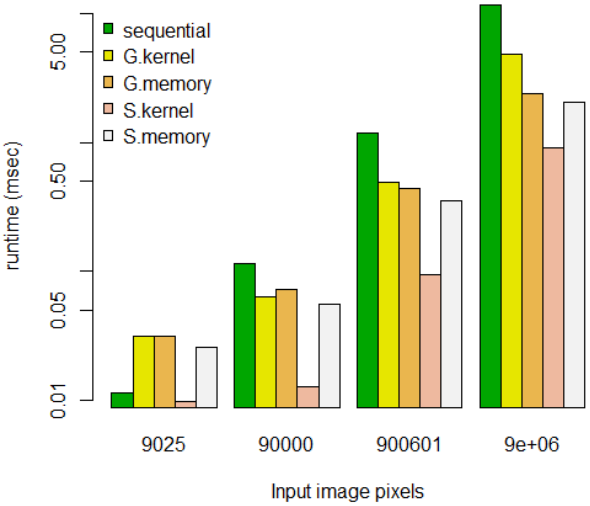

Figure 2: The contents of fig. 1 are plotted here with a logarithmic y-axis.

[1] In the previous report, we determined that the shape of the input image had no impact on performance. Thus, this work considers square images only, for simplicity.

[2] The perfectly round numbers were not all square. So we chose numbers that were close enough.

[3] In the previous report, we considered the image pixel range as its own parameter. Here we assume a fixed range of 256 for the final program; this satisfies the given specifications. This section doubles as a minor test for this metric also, as it no longer warrants a standalone experiment.

A small set image types were chosen to represent the possible domain of input images with properties considered *extreme* enough for interesting behaviors of the program to emerge. The images types are enumerated below:

1. **random**
   Pixel values are independently sampled from a uniform distribution[4] on the interval $[0, 256]$.

2. **h_gradient**
   Pixels values form a linear gradient horizontally. The leftmost column's cells have value 0, and rightmost column's cells have value 255.

3. **v_gradient**
   Pixels values form a linear gradient vertically. The top row's cells have value 0, and bottom row's cells have value 255.

4. **256-striped**
   Pixel values are given by $i \bmod 256$, where i is their column index, starting at 0. This results in vertical stripes, or a 'sawtooth' linear gradient horizontally.

5. **20-striped**
   This is identical to '256-striped', but image colors are calculated using maximum possible value 20 instead. This results in shorter, more frequent stripe bands and a smaller image value range.

The measurements for the implementations are shown in fig. 3 for images of size 3741×3741[5]. The same information is plotted with a logarithmic y-axis in 4, and once again in fig. 5, where the groupings are inverted to facilitate comparison over both *image kind* and *implementation / step*. As observed for the locking PThreads implementations in the previous report, input images that result in contention over the same 'buckets' results in considerable slowdown. With GPU threads per-warp being forced to visit adjacent image cells in perfect lockstep, this effect is exacerbated. For this reason, runs of 'v_gradient' were the slowest. The

[4]The canonical use of something like 'rand() % n' to generate a random number on the interval $[0,n)$ in C does not truly result in uniformly-distributed samples (without bias). However it is close enough for our purposes.

[5]These were the largest images we were able to fit into the GPU's memory.

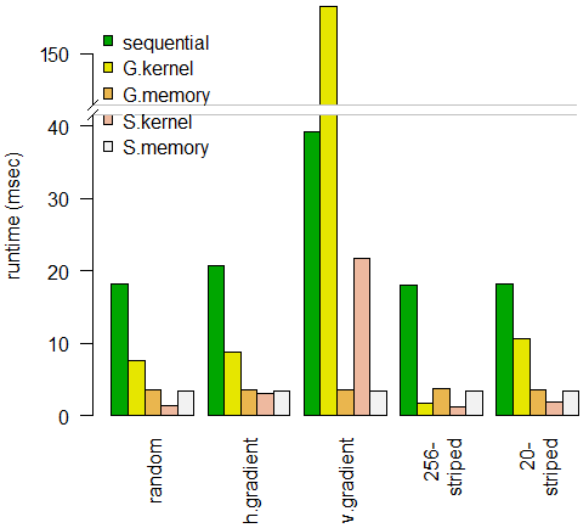

Figure 3: Runtime of the CPU-sequential and both GPU-parallelized implementations of Histogram. Each bar group represents runtimes for inputs of different *kinds*, for images of size 3741×3741.

GPU runtimes for images of kind '256-striped' performed very well; within a warp, each thread is guaranteed to visit a pixel of a different value, resulting in minimal contention. Fig. 4 makes clear just how dramatic this effect is for the kernel-step of both GPU implementations. The time for 'memory' steps remain constant across all images for runs shown here. This is due to memory-copy operations being oblivious to the contents of the image.

This predominance of the memory-copying time is ultimately what costs the GPU implementations their performance potential. Fig. 6 shows the runtime of both implementations for a random-type image and with dimensions 3741×3741. In comparison, the best-performing PThreads implementation (running under ideal circumstances) from our previous work is faster. Fig. 7 shows the relative speedups of these programs measured against the sequential runtime. As can be seen from their decomposition into steps, the actual *work* of the GPU programs (the compute kernels) *do the job* much faster than even the pthreads implementation. However, their real runtimes are unavoidably marred by the time taken to move the image data to the GPU.

The value of using shared-memory histograms

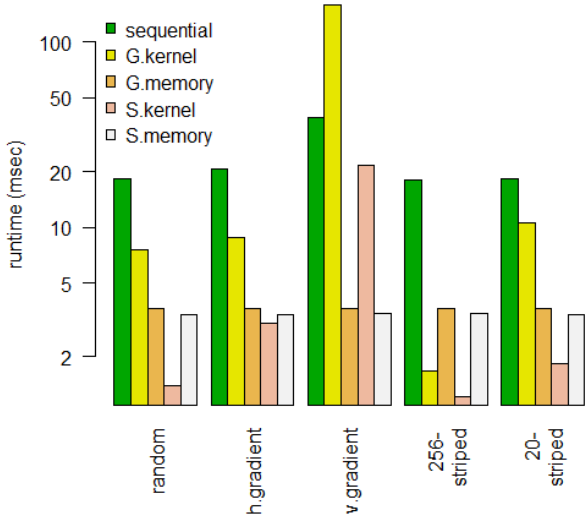

Figure 4: The contents of fig. 3 are plotted here with a logarithmic y-axis.

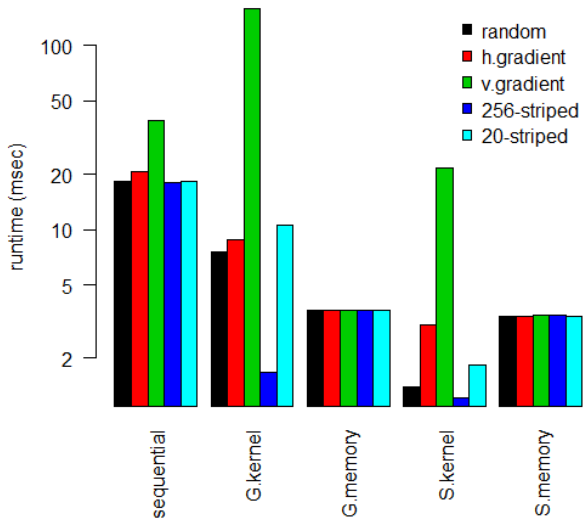

Figure 5: The contents of fig. 4 are plotted here with bar grouping and bars per group inverted.

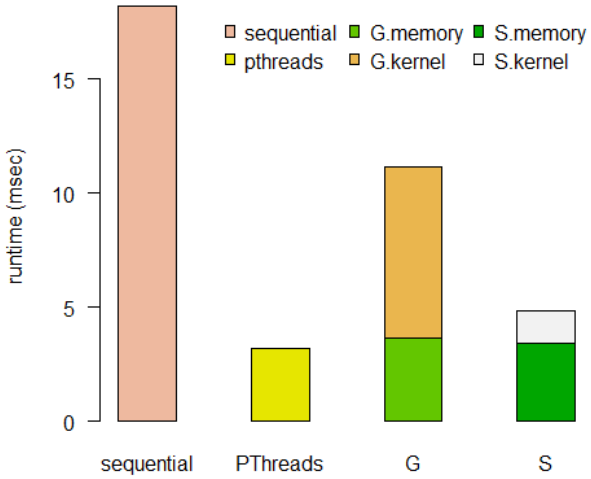

Figure 6: Runtimes of G, S and the fastest PThreads implementation (given 16 threads) plotted alongside the sequential runtime. G and S are shown decomposed into their 'kernel' and 'memory' steps. The input was a random-type image with dimension 3741×3741.

for individual thread blocks in the GPU made itself very clear. Although both mechanisms fundamentally relied on atomic additions to work correctly, S took a page out of the PThreads implementation's book with avoiding contention by localizing the histogram. This high-level approach once again paid off in a completely different context. Rather than the need for atomics in S at all causing its downfall, the completely unrelated memory-copy time ruined the speedup potential of both G and S. However, different problems with more extreme arithmetic intensity (via simply more compute-work or a mechanism to re-use GPU-side data) might succeed where S and G have failed.

3 Convolution

Data-parallelism is only worthwhile if the workload per unit is worth the overhead of creating the parallel thread to tackle it. In this section we explore the problem 'Convolution', where the amount work per output cell depends on the dimensions of the dimensions of a 'filter' matrix.

Note that unfortunately, the term 'kernel' is used

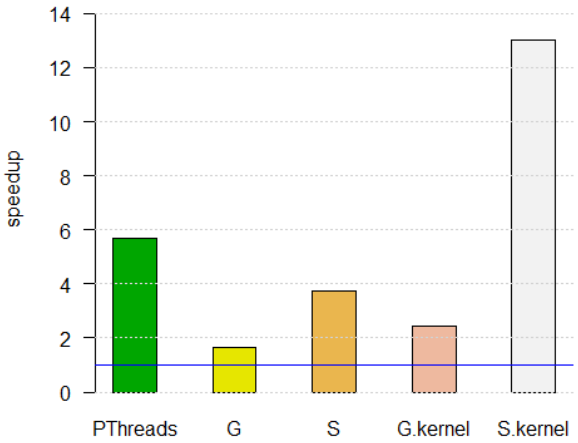

Figure 7: Speedups of all implementations measured relative to the sequential runtime, measured for fig. **??**. 'G.kernel' and 'S.kernel' show the runtime of conceptual versions of G and S respectively that have no cost for the 'memory' step.

to refer to two distinct concepts in the literature, both of which are relevant to this section:

1. A synonym for 'stencil', defining the structure used to compute outputs as functions of position-relative inputs.

2. A computational routine that is assigned as the task for GPU worker threads.

To minimize confusion, this section uses the term 'filter' for case 1.

3.1 The Problem

Given an input matrix of float values, output the matrix after one application of a smoothing filter. The filter itself is expressed in the formulas to follow, wherein:

- $O_{i,j}$ is an element at (i, j) in the output matrix.
- $I_{i,j}$ is an element at (i, j) in the input matrix (AKA the 'image').
- F^w and F^h are compile-time parameters that define the dimensions of the filter matrix F.
- C is the set of 1, 2 or 4 elements at the *center* of F (depending on whether F^w and F^h are odd or even).

$$O_{i,j} = \sum_{m=0}^{2F^w-1} \sum_{n=0}^{2F^h-1} \frac{I_{i+m-F^w,j+n-F^h} \times F_{m,n}}{T}$$

$$T = \sum_{m=0}^{2F^w-1} \sum_{n=0}^{2F^h-1} F_{n,m}$$

$$F_{i,j} = min(\{dist(c, I_{n,m}) : c \in C\})$$

$$dist(I_{i,j}, I_{k,l}) = min(\{|k-i|, |l-j|\})$$

While this filter is well-defined for arbitrary dimensions, the problem given affixes the dimensions at 5×5 (ie. one particular filter). As such, we will consider the behavior of our implementation in general, but ultimately make decisions to optimize performance for the 5×5 filter in particular, which looks like this:

1	1	1	1	1
1	2	2	2	1
1	2	3	2	1
1	2	2	2	1
1	1	1	1	1

3.2 Implementation

The filter matrix F is generated at initialization and remains unchanged until the program is over. The calculation of elements in the output matrix O can be performed in parallel, requiring no looping or blocking[6]. In many ways, this problem is quite similar to the *Simulating heat dissipation on the surface of a cylinder* problem explored in our previous reports. However, in this case, the *arithmetic intensity* of the problem scales with the chosen parameters of dimensions (F^w and F^h) for F. As the GPU is a device with its own memory, I values must be copied over *from* the host at the start, and O values need to be copied *to* the host at the end. The time taken for this operation is significant. Therefore, the GPU is likely to achieve better

[6]Of course, there is an implicit barrier at the end of the kernel's execution preventing the memory copy back until all the threads' tasks on the GPU have been completed.

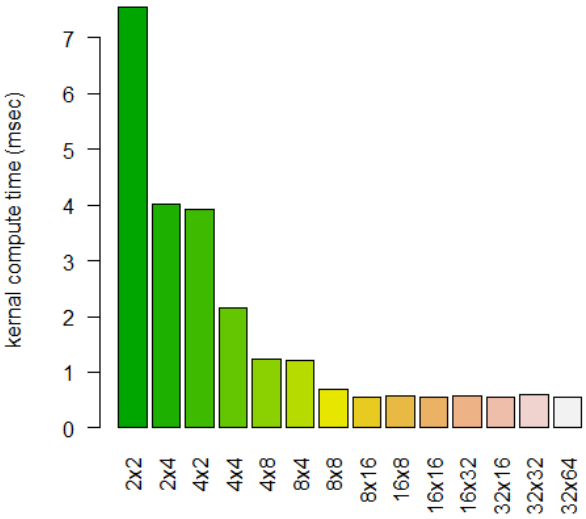

Figure 8: Kernal compute time for CUDA Convolution implementation using various block dimensions, represented as $N \times M$.

speedup from the host-only sequential implementation with problem instances with high arithmetic intensity (ie. using a large filter).

We expect the best utilization of the GPU will be achieved when block dimensions are not too small. Small blocks will lead to under-utilized streaming-multiprocessors ('SM's). To verify this intuition, we conducted a small preliminary experiment with the specified 5×5 filter and a reasonable input image of size 1024×1024. Our implementation uses a 2D grid layout. The results can be seen in fig. 8.

As anticipated, small blocks resulted in long runtimes. Interestingly, there was no significant difference between larger dimensions. We choose the dimension of 32×16 arbitrarily and use it henceforth.

Our implementation makes use of an (intuitively) naïve algorithm which simply concentrates on a very fine kernel granularity for thread workers. As the output is computed from one filter application per cell only, there is no indefinite looping. The kernel is prepared, 'fired' once and the host waits for completion before writing back. Minor design choices optimize this approach:

1. The input image is padded with elements to allow the filter to be applied without having to worry about out-of-bounds reads or requiring conditional checks.

2. Each thread applies the filter once to derive a single output element at a unique position in the output matrix, determined by its computed thread ID.

3. Threads write intermediately-computed results into a local register each, then only write to the output matrix at the end.

4. Threads presumed to be adjacent write to adjacent output cells to make good use of the cache hierarchy.

Both the input image and filter are invariant once initialized for their lifetimes. Best practice dictates that they therefore be declared as CUDA *constants*. Constant data is allocated in a special way. This distinction allows the GPU to perform specialized behavior that leverages its invariant nature to allow for faster access, such as a mechanism to broadcast accesses to threads in a warp all at once.

Firstly, the available constant space of the GTX480 is insufficient to hold I, the entire input image for this problem. Instead, we concentrate on the possibility of using this memory for the filter matrix F.

Counter to our expectations, the use of constant memory for the filter did not lead to consistent speedup. At first glance, there was no rhyme or reason behind whether storing the filter as a constant was beneficial. This warranted further investigation. Fig. 9 shows that for the 5×5 dimension F, the use of constant memory is beneficial for small images, detrimental for moderately-sized ones, and irrelevant for large ones. While seemingly 'unstable' *over* input sizes, the measurements were observed to be highly invariant *per* input size for repeated runs. This would suggest there was some other factor deterministically influencing the runtime of at least one of the implementations as a function of the input dimensions.

This constant-global experiment was repeated for a smaller and larger filter size, shown in figs. 10 and 11 respectively. It was observed that in all cases, the use of constants resulted in the same pattern, and was *less* pronounced when the filter was larger. With larger filters comes more work for both implementations; this would suggest the factor influencing the results is tied to the number of

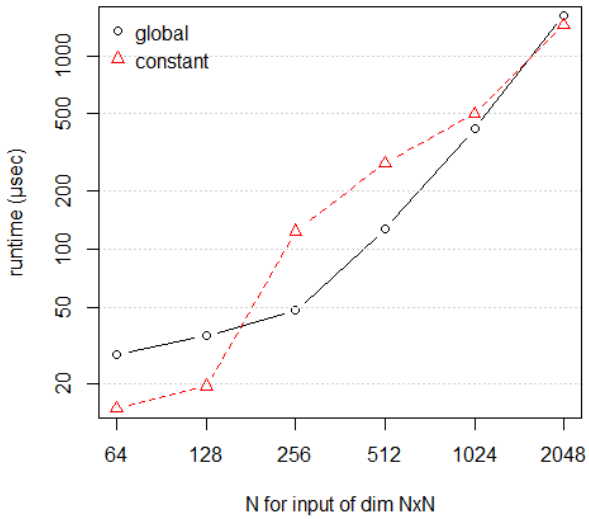

Figure 9: Runtimes for the 'constant' implementation version (storing the filter in the GPU's *constant* memory), and the 'global' version, which stores it in *global* memory. The filter has dimensions 5×5 Note that both axes are logarithmic.

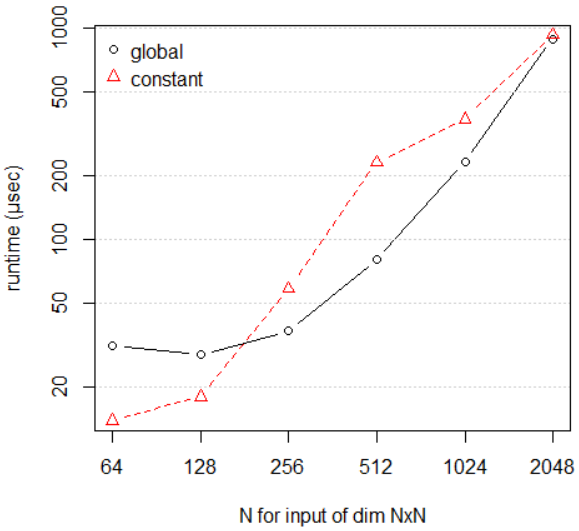

Figure 10: Runtimes for the 'constant' implementation version (storing the filter in the GPU's *constant* memory), and the 'global' version, which stores it in *global* memory. The filter has dimensions 3×3 Note that both axes are logarithmic.

threads rather than to the number of *cells* in the filter. However, the exact cause is not clear.

Runtimes seen thus-far all took less than a hundredth of a second. As the constant and global implementation versions only differed for these very small problem instances, we opted for using the *constant* version henceforth, to adhere to the best practice.

These preliminary experiments were observed to be dominated by time taken to copy memory to-and-from the GPU (as it was in the previous section). For this problem, we sought to observe how the time taken for these steps changes in response to arithmetic intensity, which is a function of F^w and F^h. As can be seen from fig. 12, for small filters the cost of copying memory back and forth from the GPU dwarfs the compute cost. Although increasing filters does increase *both* of these values (due to filters affecting the size of the padding in the input), the compute cost is affected significantly more. For F with dimensions 5×5, the kernel contributes only around 12% of the total runtime.

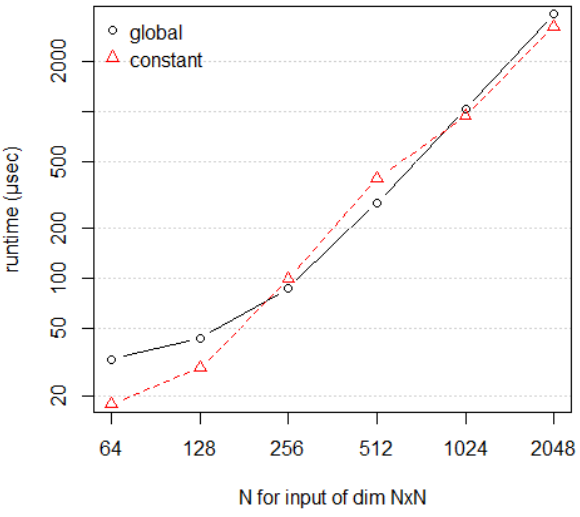

Figure 11: Runtimes for the 'constant' implementation version (storing the filter in the GPU's *constant* memory), and the 'global' version, which stores it in *global* memory. The filter has dimensions 9×9 Note that both axes are logarithmic.

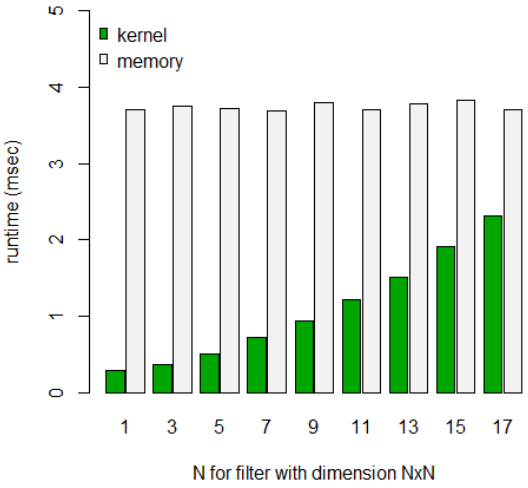

Figure 12: Runtimes of the 'kernel' and 'memory' components for the GPU implementation. Each bar group distinguishes a different dimension of the filter used.

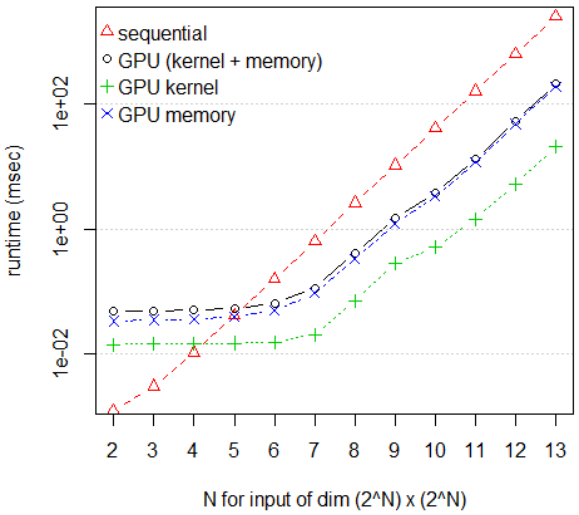

Figure 13: Runtimes plotted over various input image sizes for sequential and GPU implementations. The two steps of the GPU runtime are also shown separately. Note that both axes are logarithmic.

As such, we decided to forgo more complex optimizations typically used for achieving greater GPU speedup such as:

- Heterogeneous computing. Moving a chunk of the work to the CPU to be computed in tandem with the GPU.

- Overlapping computation and communication. Here, the added overhead from splitting up the input in a way to prevent out-of-bounds filter reads, firing multiple kernels and managing the streams is simply not worth the trouble[7].

3.3 Performance Results

Satisfied with our finished implementation, we conducted experiments to determine the performance of our GPU implementation to the sequential CPU-only implementation.

[7]Not worth the trouble in both a programming-effort sense and in a performance-gain sense.

Fig. 13 shows the runtimes of both CPU and GPU versions, along with the times of the GPU version decomposed into 'kernel' (The time taken to compute the output matrix on the GPU) and 'memory' (The time taken to copy memory between host machine and device).

Note that in this plot, both axes are logarithmic. We clearly see that the initialization of the GPU version is non-trivial, and the memory-copy step is the bulk of the runtime for the GPU program. This mirrors the observations made from the preliminary experiments. It can also be confirmed that while the GPU program is certainly much faster for a large image, the use of the GPU does not lower the *complexity class* of the solution, rather it decreases the cost by a constant proportion (the speedup), here observed as a translation downward in the logarithmic plot.

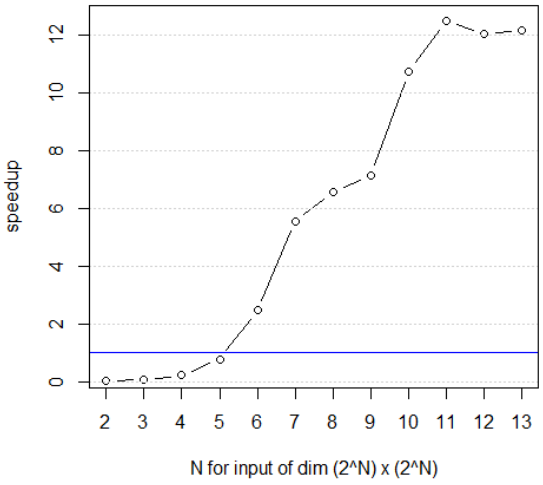

Figure 14: Speedup of the GPU implementation vs. the sequential CPU implementation, with the same data shown as in fig. 13. The horizontal line marks trivial speedup of 1.

The same measurements were used to produce the speedup plot, fig. 14. With larger problems (if they were able to fit on the DAS4 node) we would expect this speedup to stabilize and no longer decrease. The 'steps' for smaller plots are a result of the GPU and CPU implementations performing fundamentally different steps, running on fundamentally different hardware with different caches and so on, as well as the strange behavior we observed as a result of the constant memory before. From this plot we see that somewhere between input images of size 32×32 and 64×64 lies the tipping-point above which the GPU is superior.

4 Conclusion

GPUs offer immense computational power for highly parallel algorithms. With just a single compute node. CUDA offers a *relatively* elegant means of making use of this hardware without deviating too much from the C-like environment that most programmers are accustomed to. By the nature of the fact that programmers turn to GPUs only when performance is important, it is not unreasonable that CUDA requires some investigation on the part of the user. After all, at this level of granularity, attention to hardware specifics is necessary for CPU programming as well.

In the case of both our problems, we observed that while the GPU was most effective and tearing through the work like nobody's business, the memory bandwidth ended up costing the lion's share of the total program runtime; this holds despite the fact that one might consider them both problems of high *arithmetic intensity*, perfect on paper for GPU parallelization. However, these particular problems use their input data very little, requiring just one 'visit' per cell. For other problems without this property, the time investment in moving the work to a powerful GPU would pay off. Fortunately, even with this handicap, in both cases the GPU implementation was still a comfortable order of magnitude faster than the sequential-CPU benchmark.

Appendix

Table 1: Runtimes (in seconds) for histograms implementations and steps of implementations in section 2. Rows discern between input images sizes. All inputs were of type 'random'.

	sequential	G.kernel	G.memory	S.kernel	S.memory
95×95	1.148668e-05	3.126400e-05	3.137280e-05	9.811200e-06	2.571520e-05
300×300	0.0001150424	0.0000637376	0.0000732864	0.0000128832	0.0000557376
949×949	0.0011848660	0.0004896640	0.0004394752	0.0000942976	0.0003553984
3000×3000	0.0115728200	0.0048270960	0.0024033160	0.0008997632	0.0020611720

Table 2: Runtimes (in seconds) for histograms implementations and steps of implementations in section 2. Rows discern between input images sizes. All inputs had dimension 3741×3741.

	sequential	G.kernel	G.memory	S.kernel	S.memory
random	0.018154520	0.007505976	0.003623570	0.001395772	0.003391280
h_gradient	0.020663300	0.008753978	0.003617928	0.003047582	0.003391198
v_gradient	0.039198860	0.158923600	0.003629298	0.021736940	0.003410366
256-striped	0.018013400	0.001667102	0.003652166	0.001210470	0.003434394
20-striped	0.018139200	0.010527040	0.003611604	0.001825100	0.003377088

Table 3: Runtimes (in seconds) for CPU and GPU implementations for an input of dimension 3741×3741 and type 'random'.

	sequential	pthreads	G.kernel	G.memory	S.kernel	S.memory
runtime (sec)	0.018176360	0.003645324	0.001395876	0.003447922	0.001395876	0.003447922

Table 4: Runtimes (in msec) for GPU Convolution runs using various grid-dimensions. Input image was of dimension 1024×1024

	runtime (msec)
2x2	7.5462267
2x4	4.0209667
4x2	3.9210367
4x4	2.1498167
4x8	1.2241533
8x4	1.2222767
8x8	0.6861857
8x16	0.5509310
16x8	0.5683003
16x16	0.5583223
16x32	0.5698550
32x16	0.5542153
32x32	0.6050937
32x64	0.5541357

Table 5: Runtimes (in μsec) comparing the 'global' and 'shared' implementation versions for Convolution. Columns distinguish N for input image of dimension $N \times N$. All runs used a filter of dimension 3×3.

	2048	1024	512	256	128	64
global	890.6235	232.920	80.19985	36.94533	28.48657	31.45612
constant	931.7490	370.739	231.46625	58.39343	18.07005	13.93094

Table 6: Runtimes (in μsec) comparing the 'global' and 'shared' implementation versions for Convolution. Columns distinguish N for input image of dimension $N \times N$. All runs used a filter of dimension 5×5.

	2048	1024	512	256	128	64
global	1621.127	416.6540	126.3646	48.24145	35.48080	28.65440
constant	1432.880	499.7273	278.3012	123.58375	19.51575	14.96265

Table 7: Runtimes (in μsec) comparing the 'global' and 'shared' implementation versions for Convolution. Columns distinguish N for input image of dimension $N \times N$. All runs used a filter of dimension 9×9.

	2048	1024	512	256	128	64
global	3842.880	1035.240	285.2443	87.6243	43.98237	32.93007
constant	3184.186	940.108	397.7400	99.2799	29.18455	17.75502

Table 8: Runtimes (in msec) of 'constant' for both 'kernel' and 'memory' steps for different choices of N for filter of dimension $N \times N$.

	kernel	memory
1	0.2917775	3.703870
3	0.3670140	3.757100
5	0.5024450	3.714970
7	0.7270100	3.689100
9	0.9376400	3.795950
11	1.2158000	3.698250
13	1.5120100	3.786970
15	1.9069550	3.832700
17	2.3084300	3.700155

Table 9: Runtimes (in msec) for the sequential implementation and both 'kernel' and 'memory' steps for the 'constant' GPU. Rows distinguish between choices of N for input of dimension $N \times N$. For all runs, filters of dimension 5×5 were used.

	sequential	kernel	memory
2^2	1.190403e-03	0.01378105	0.03285500
2^3	2.961125e-03	0.01414855	0.03443265
2^4	1.037272e-02	0.01403142	0.03488025
2^5	4.070952e-02	0.01452630	0.03798115
2^6	1.576175e-01	0.01512255	0.04900743
2^7	6.231773e-01	0.01993915	0.09219500
2^8	2.631730e+00	0.06959873	0.33098650
2^9	1.048785e+01	0.27948050	1.19445500
2^{10}	4.098988e+01	0.50624250	3.31543000
2^{11}	1.628145e+02	1.43530250	11.61635000
2^{12}	6.500835e+02	5.29965250	48.72547500
2^{13}	2.597375e+03	20.81142500	193.12975000