C1424548

CM3103: High Performance Computing

Professor David Walker

Programming with CUDA

December 5, 2017

Contents

1 Environment and Methodology

I completed the CUDA code on my windows PC at home, then used WinSCP to transfer the source to the Linux lab computers, and the PuTTY SSH client to remotely access my account. I then compiled the program using NVCC.

Next, I wrote and executed a simple bash script to automatically run the program with different configurations, and collate the results. I set it up to run the program at blur counts 10, 20, 40, 80, and 160, for 100 iterations each, and calculate the mean outputted times for each program section. I ran multiple iterations to achieve higher accuracy, and to reduce noise from external factors.

Finally, I loaded the data into MATLAB and generated some bar charts, which are included in the next section. I chose bar charts because the x-axis is discreet, but still allows you to see patterns over multiple bars. They can also display multiple series of data side-by-side for easy visual comparison.

2 Results

2.1 Original Running-Time

First, I ran the original blur code 100 times and took the mean and standard deviation from the outputted times. The result was $0.04042015s \pm 0.00210662s$. Note that this is the time for 10 blurs using the original code, and does not include the time spent reading/writing files or performing other set-up and clean-up activities.

2.2 CUDA Running-Time

2.2.1 Results Table

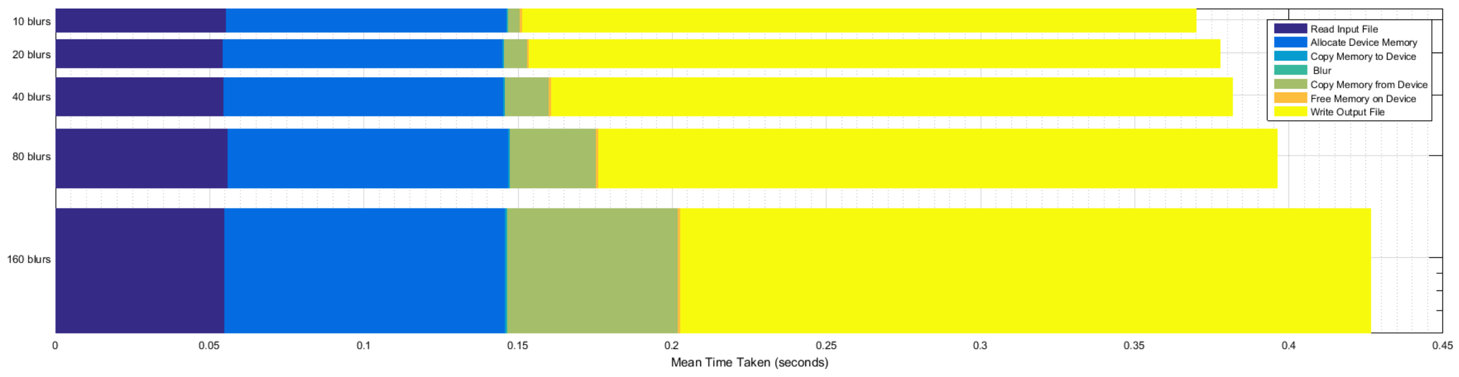
The table below contains all of the timing data I collected from my CUDA program. The first column N shows the number of blurs performed during the execution, and the subsequent columns show the mean time taken for each stage of the program. All time values are given in seconds, with 8 decimal places of precision, and standard deviations are included below each value.

N	Read Input File	Allocate Device Memory	Copy Memory to Device	Blur	Copy Memory from Device	Free Memory on Device	Write Output File
10	0.05528016 ±0.00669045	0.09085131 ±0.03048576	0.00049009 ±0.00001493	0.00003909 ±0.00000246	0.00397467 ±0.00000847	0.00067940 ±0.00000983	0.21866316 ±0.02890739
20	0.05429731 ±0.00171192	0.09068615 ±0.00059459	0.00049010 ±0.00001694	0.00006075 ±0.00000612	0.00739409 ±0.00001855	0.00068116 ±0.00000727	0.22425036 ±0.02010853
40	0.05456403 ±0.00399285	0.09055803 ±0.00162002	0.00049111 ±0.00002075	0.00010774 ±0.00001404	0.01421622 ±0.00002180	0.00068053 ±0.00001804	0.22115894 ±0.07810454
80	0.05581486 ±0.00652362	0.09091106 ±0.00117204	0.00048952 ±0.00001958	0.00019703 ±0.00001895	0.02787756 ±0.00002591	0.00068050 ±0.00001528	0.22026802 ±0.03403625
160	0.05474854 ±0.00204949	0.09095280 ±0.00093145	0.00049183 ±0.00002011	0.00037336 ±0.00001840	0.05521410 ±0.00003633	0.00068553 ±0.00001828	0.22413142 ±0.04829054

2.2.2 Graphs

The bar chart below shows the mean total time taken at each blur count, with the individual times for each section shown within each bar. This chart is useful for comparing the total time taken at each blur count, and gives an idea of how much time is spent on stage of the program.

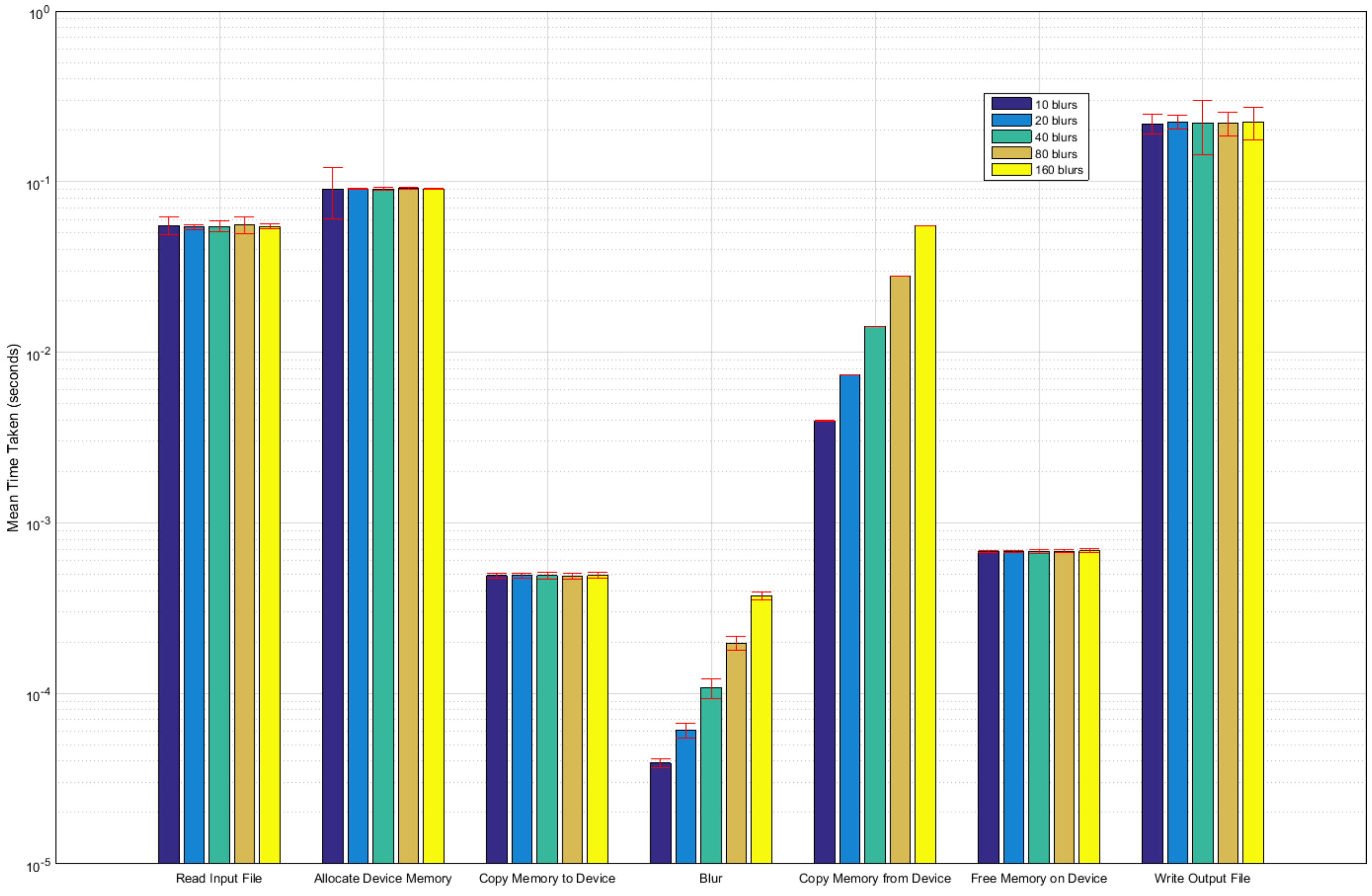
The bars are different widths because the samples are logarithmically spaced, but I kept the axis linear. I did this to ensure linear relationships appeared as straight lines. The alternative would have been making the time axis logarithmic, which would have made it difficult to compare the sizes of different sections within bars.



The next bar chart shows the timing data, grouped by program stage, and displayed on a logarithmic y-axis. On this chart, you can clearly see how the time taken for each stage of the program varies with the number of blurs.

I used a logarithmic y-axis because of the differences in magnitude between the times taken for different parts of the execution. In the previous chart, the linear time axis meant some of the sections were almost invisibly small. It also matches the logarithmic x-axis, meaning straight lines represent linear relationships.

The standard deviation of the values is also shown. It is represented by the red I on each bar.



3 Analysis

First of all I will compare the performance of the CUDA code and the original code provided in the brief. The timings outputted by the original code represent only the time taken to perform 10 blurs. The equivalent value in the CUDA timing data is the time taken between allocating device memory and freeing device memory (inclusive) for 10 blurs. This includes the time spent blurring, while encompassing all of the work the would not be required in a standard program such as copying memory to the device.

Original Code:	$0.04042015s \pm 0.00210662s$
CUDA Equivalent:	$0.09603456s \pm 0.01363365s$

The obvious thing to point out here is that the CUDA code is actually slower than the sequential code. Almost the entirety of the time taken by the CUDA application was spent allocating memory on the device. In fact, if we exclude memory allocation, the time comes right down to $0.00518325s \pm 0.00000997$. It is clear the CUDA is speeding up the actual blurring considerably, but the overhead of memory allocation is outweighing any potential performance gains.

One way I could speed up the CUDA program would be changing the datatype of the arrays allocated in the device memory. Currently multiple large arrays of ints are used to store colour information for each pixel. Each colour channel should only have a range of 0-255, much smaller than the range of an int. A lot of memory could be saved by using unsigned chars instead of integers in the arrays. I might have done this had I known that memory allocation would have such a big impact on performance.

Overall, overheads like memory allocation should be dwarfed by the performance increases if the task is sufficiently intensive. I suspect that at much higher blur counts for instance, these fixed overheads would become insignificant, however for tasks on such a small time-scale they have a huge impact.

Next I will look at the time-per-blur for the CUDA program at each blur count. On the right is a table of these times.

Blurs	**Time**	**Time/Blur**
10	0.00397467s	0.00039747s
20	0.00739409s	0.00036970s
40	0.01421622s	0.00035541s
80	0.02787756s	0.00034847s
160	0.05521410s	0.00034509s

The data shows that the time-per-blur decreases slightly with more blurs. The decrease is so marginal, that I suspect the actual kernel call takes a constant time, and instead there is a small fixed overhead from calculating things like block and grid dimensions on the host. Over more blurs, this overhead remains fixed, giving it a smaller impact relative to the overall time taken, and leading to a slight speed up per-blur.

Another factor to consider is the small amount of time spent swapping the pointers to the input and output arrays in the device memory. This operation happens between each blur, meaning it happens a total of $n - 1$ times for every n blurs. This would lead to a slight increase in time-per blur with more blurs, as the 1 swap-free blur becomes less and less significant. This factor is clearly outweighed by the effects of fixed overheads described in the previous paragraph.

The final thing to look at is the linear relationship between blur count and copying memory from the device. On the second graph, it is clear to see that as blur count increases, time spent on the *Copy Memory from Device* section increases linearly with it. This is surprising as the amount of memory being copied does not change.

I theorise that this occurs because the host never actually has to wait for the device to finish executing kernel calls until it tries to copy the memory back. If this was true, most of the time spent computing blurs would actually fall within the *Copy Memory from Device* stage. This would mean that time in the blur stage would be almost entirely spent making different amounts of kernel calls, and would not actually reflect the time taken to perform the blur computations. This seems reasonable, as the blur stage has the lowest mean time taken of all the stages, even at 160 blurs.

4 Conclusions

It is clear that there are large performance gains to be made by utilising graphics hardware for parallel computation. You must be careful however of the large potential overheads of communicating with a physically separate computation unit. CUDA is a very powerful and easy to use interface. It definitely demands a focus on optimisation to get the most out of it, but this is to be expected with any high-performance platform.

I have greatly enjoyed writing CUDA code for the first time during this coursework. I always learn a lot from working with different paradigms and this was no exception. I think that the commercial availability and widespread use of graphics cards makes CUDA a very powerful technology, and it is likely that I will use CUDA or a similar technology again in the future.

This task was also a good C exercise for me. I have not written much C code in the past, and I am beginning to become more confident with things like pointers and c-style arrays. Overall this task has been a great learning experience.