Environmental Considerations When Measuring Relative Performance of Graphics Cards

Daniel Goodman - Oxford eResearch Centre

***Abstract*—In this paper we examine some of the environmental conditions that have to be considered when comparing the performance of GPU's to CPU's. The range of these considerations varies greatly from the differing ages of the hardware used, to the effects of running the GPU code before the CPU code within the same binary. The latter of these has some quite surprising effects on the system as a whole. We then go on to test the different hardware performance at matrix multiplication using both their basic linear algebra libraries and hand coded functions. This is done while respecting the considerations we have described earlier in the paper, and addressing a problem that with the use of the Intel MKL library cannot be argued to be unfair to the CPU.**

I. Introduction

THE advent of CUDA [1] and OpenCL [**?**] as has made the programming of graphics cards far more accessible. This coupled with NVIDIA's push towards HPC through their Tesla ranges, has very much pushed General Purpose Graphical Processing Units (GPGPU's) into the forefront of scientific computing. This has resulted in a large number of studies into the speed up that can be achieved in different applications, and in this paper we look at some of the issues that relate to getting acurate performance meausres, most critically demonstrating how the environment surrounding the CUDA invocations can have a very direct effect on the CPU code around it.. There is of course a huge body of work that has gone before in how to benchmark different computing platforms [2] and we are not aiming to duplicate this. It is hoped that readers will find this insightful and useful both when measuring their own projects and when evaluating other peoples results.

The structure of this paper is a description of some of the issues that we have identified that GPU's raise and that need to be taken into account when evaluating the performance. Then performance results generated when comparing the CUDA Blas libraries and SDK codes against the Intel MKL libraries are discussed before concluding.

II. Issues affecting performance when comparing GPU's and CPU's

a) Age and Specification of hardware: The most basic environmental issue is esuring that the hardware is comparable. This is of course dependant on how you define the hardwares to be equal the best solution is for them both the most powerful currently available, though given the variing release dates of new architecture this is not nessersaraly fair, likewise measures such as cost make it much harder still to choose to comparable pieces of hardware. As an example, if the host computer wass middle of the range 18 months ago, you could expect a 2 - 4 times speed up for properly threaded code simply by replacing it with a top of the range model today. Similarly if experiments with GPU's are performed with a new top end card to see just how fast they go, it is perfectly valid test. However, when reporting *speed up against a CPU*, it must be remembered that the specific CPU is not performing as a fast as CPU's are capable of going. A good demonstration of this is the improvement demonstrated by Intel's latest processors over their last generation, see Figure 1.

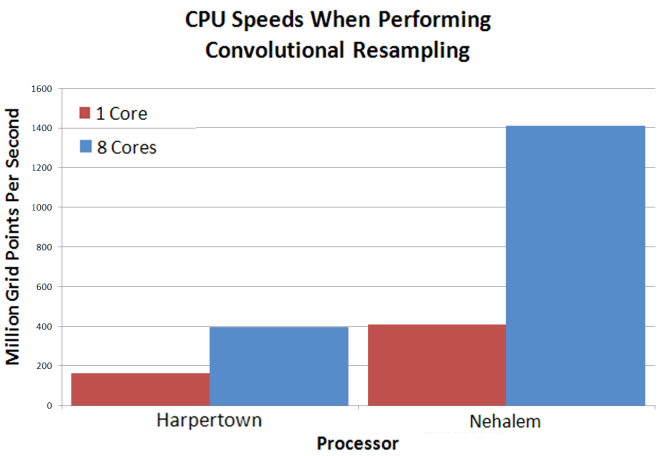

Fig. 1. Performance statistics for computing signal analysis on astronomical data. Measured on a dual processor 2.8GHz Harpertown (November 2007) system and a dual processor 2.8GHz Nehalem (March 2009) system. The performance gain as CPU's adjust to provided better bandwidth to the cores is clearly visible. Data courtesy of Tim Cornwell.

b) Optimisation: When constructing CUDA programs, it is very helpful to have a very basic CPU version which the results can be checked against to ensure that the code is producing correct values. As such this has become standard practice with the CPU program being known as the gold version. Often this gold version is then also the code that the performance of the graphics card is measured against. However, it has to be remembered that the function of this method was to provide results that with a high degree of confidence where believed to be correct. As such they tend to be very naive in their implementation, and generally make no effort to optimise the computation as this would reduce the clarity of the code and distract from the task at hand, which is after all providing a correctness test for code constructed for the graphics card. This does however mean that they are not appropriate for accurate performance testing against the graphics card, as the GPU code will typically be optimised, making effective use of the different types of memory, and the algorithm arranged around carefully chosen parameters and structured memory accesses to ensure optimum efficiency. If a true test is to be performed, a second version of the CPU code is required. This version must have been structured to take full advantage of the cache structures, multiple cores offered by the CPU, and compiled with the optimisation flags set.

In the same vein, often examples of GPU code are presented that would require recompiling of the CUDA code to change the problem parameters. The CPU code on the other hand is generic, and can be changed from run to run. This difference however means that while the GPU compiler can pre-evaluate

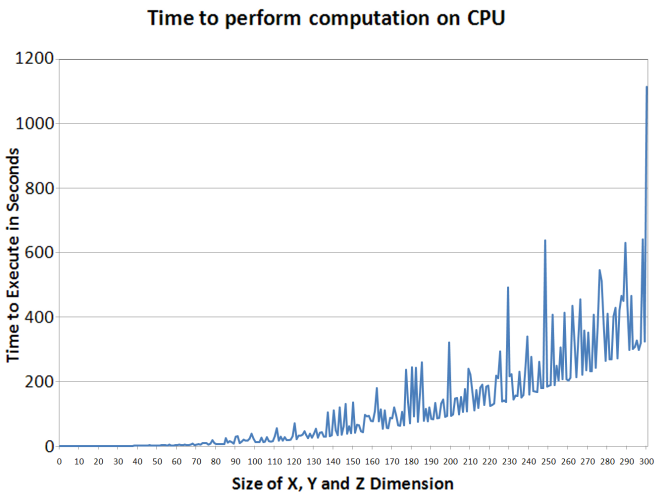

Fig. 2. A graph showing the varying time to execute the CPU Gold code for the Horn-Schunk calculation on different sized datasets.

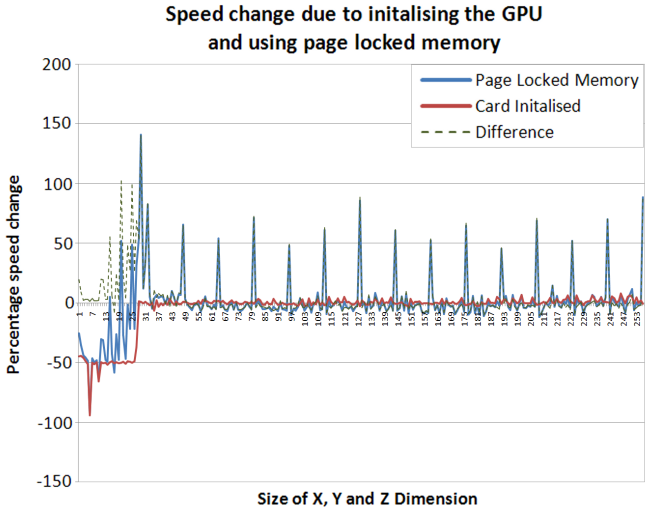

Fig. 3. A graph showing the percentage change in the time to execute the CPU Gold code for the Horn-Schunk [**?**] calculation on different sized datasets when the environmental conditions are changed. The red line shows the effect of initialising a GPU. The initial speed up as a result of this we believe is due to the compiler moving code from further down the computation up to fill the time taken while the card initialises. Once the problem exceeds a certain size this effect disappears, and the change becomes less dramatic, but in this second phase, the initialisation can result in up to an 8% slowdown of the CPU code.

Far more dramatic is the effect of making the memory page locked using a call from the CUDA API. The blue line shows the directly measured changes, and the green dashed line shows the changes with the change caused by initialising the graphics card removed. Aside from the very large slowdowns if data set happens to be an unlucky size, the variation in execution time generally falls into the -10% to +15% range.

parts of the computation, and unwind loops, the CPU compiler does not have this luxury, and as such is unable to produce such high performance code.

The choice ofvalue of parameters for tests such as the size of a kernel in a Gaussian Filter, or the size of a data set is normaly chosen to optimise the performance of the GPU. This generates a maximum value for the potential speed up, but can discise the general case, meaning the performance for a range of different sizes of problem is more insightful. Further to this though, as performance vairies for both GPU and CPU code, using just a single value creates the possibillity that the value being used is really bad for the CPU code the GPU is being compared against? Because the CPU code is often not crafted to the specific architecture that it is built on, but instead is relying on the generic nature of the hardware interface, performance can vary dramatically between different parameter values. To demonstrate this effect, code that was used as the gold function for an implementation of the Horn-Schunk method [3], was executed for data sizes from 10 to 300. The test was repeated 5 times to ensure that variations in timing are as a result of properties of the code and architecture. The results of this can be seen in Figure 2, these clearly show that a small change in the size of the dataset can have a huge change in the time to execute meaning the potential to pick a value where the CPU performs poorly is high and 180 can be seen to be a very bad value for the CPU, but it also corresponds to the largest value that will fit into a 512 MB GPU.

c) Native Data Types: Older GPU's are only able to perform 32bit floating point arithmetic, and newer GPU's perform 64bit floating point arithmetic at a performance price. As such it is standard practice to use 32bit floating point data in GPU applications, and therefore in the CPU code. Furthermore conventional wisdom would say that the CPU code should be faster as a result of this lower accuracy and less data to transfer. However, benchmarking has shown that modern CPU's and libraries are more highly optimised for double precision arithmetic to the extent that it can be up to ten times as fast as single precision arithmetic. So by choosing to work in a data type that suits the GPU, the performance of the CPU code can be harmed.

d) Operating Environment: It is important to ensure that the operating environment for the GPU and CPU tests do not interfear with each other. Our experiments have demonstrated that there is a clearly measurable and consistent change in the performance of the CPU code if it is called from code that has initialised a graphics card beforehand. Furthermore while the use of page locked memory accelerates the transfer of data to and from the graphics card, it can have a negative effect on the performance of CPU code using it. The effects of this can be seen in Figure 3 where the percentage speed change when a GPU is initialised and when page locked memory is used to perform the CPU execution is plotted for a range of problem sizes. Furthermore, if a GPU kernel has actually been executed this can slow the CPU code down even further as shown in Figure 4. As a result of this, if we are to get a true gauge on how much faster the GPU is than the CPU, the CPU code must be executed in an environment that is independent of the environment created to support the GPU. It must also be remembered that as much of the GPU's environment is hidden from the user, this means the executions of separate binaries should be the preferred method to ensure a truly fair test.

Because the execution of GPU code requires the kernel to be loaded into the graphics card, the first run of a computation can be slower than the following ones. As such it can be beneficial to run a warm-up calculation before running the GPU calculation in order to ensure a true maximum performance is recorded. However, if this figure is then to be used to measure against the CPU, it is also necessary to perform a warm up calculation on the CPU to ensure that the caches are fresh.

III. Matrix Multiplication

The performance of matrix multiplication on a Tesla C870, is compared against a dual processor 2.66 Ghz quad core Xeon Harpertown system. On the Tesla card both the CUDA Blas library [4], and the simpler code provided in the CUDA

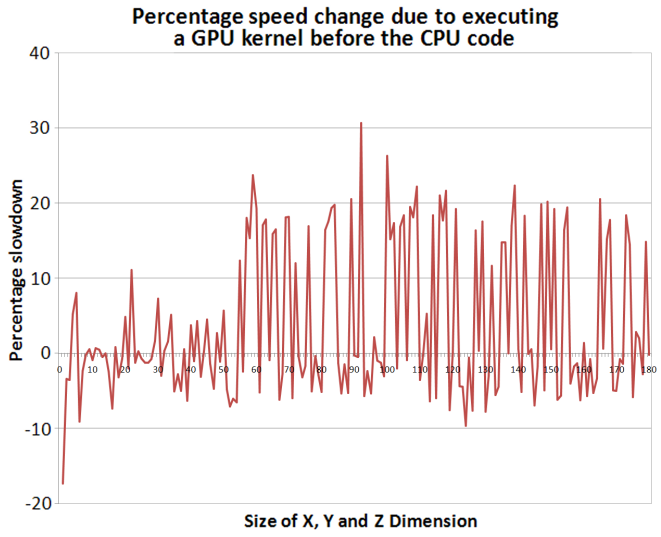

Fig. 4. This graph shows the further change in performance of CPU code if the GPU kernel is also run before the execution of the code. Note that this can make the code take almost a third longer to execute.

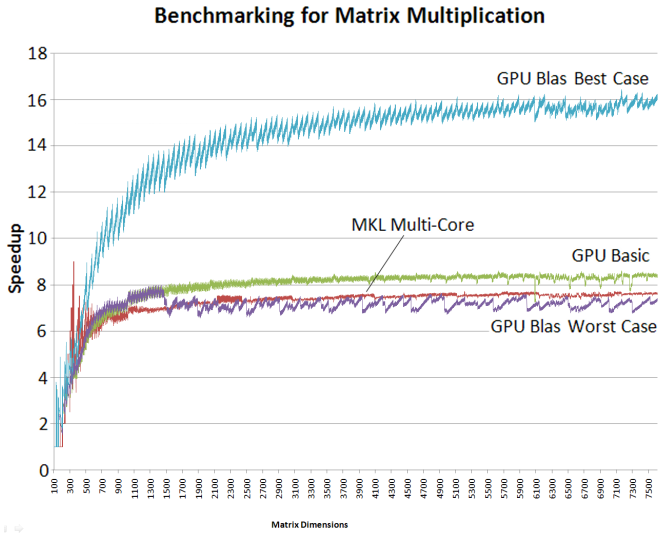

Fig. 5. Speedup for matrix multiplication.

SDK for matrix multiplication are tested. The code from the SDK was included to provide an idea of what can be achieved by a semi skilled user. On the host we user the double precision matrix multiplication routine included in the Intel MKL library [5]. This was used as tests showed it outperformed SGEMM. As a baseline this routine was executed on a single core, and the speedups are reported as improvement against this time. The tests were performed on square matrices, sized between 10 and 7600 in each dimension. The results can be seen in Figure 5.

When measuring the results with CUDA Blas, the performance varied greatly depending on if the size of the problem is equal to a given value modulo 16, 32 and 64. This change in performance was expected due to the benefit of coalesced memory accesses on CUDA devices. As the addition of padding is not always available it was felt that it was worth measuring under both circumstances. As such both the best and worst case scenarios where measured when computing the result for a given data size, to give an idea of the possible spread. This same problem was not observable in the MKL results so they are presented as a single line.

These results show that the MKL code performs slightly better than the worst case scenario for the CUDA Blas routine, which is interestingly worse than the simple version from the CUDA SDK. The CUDA SDK version is able to outperform the MKL version, though it is hard coded to only accept problems of the appropriate sizes it does not have to handle the memory alignment problems that the Blas version is handling. This shows that GPU's have sufficent computing power for the right style of problem to make code written by semi skilled users out perform some of the most highly optermised CPU code. The Best case scenario clearly outperforms all the other versions from problems with dimensions in excess of 500, however given the results in Figure 1 this gap may be closing.

IV. Conclusion

This paper has highlighted a range of environmental factors that have to be taken into account when comparing the relative performance of GPU codes against CPU codes. These include a range of factors that thus far appear not to be taken into consideration such as the surprising effect that initalising the GPU has on the surrounding CPU code.

We have then gone on to demonstrate that using library code provided by the vendors for problems that should parallelise well onto both architectures, the GPU's are able to offer a speedup of approximately 2.1 times over two four core CPU's. On a standard desktop machine that only has a single CPU instead of two, this could increase to approximately 4 times faster. However, this is the number that the majority of users are going to be interested in as they are not going to be hand coding large parts or their work, but are going to be using the libraries that are provided by others. However, it is interesting to note that code crafted by a moderately skilled programmer can be equivalent to or outperform highly optimised Intel code.

Finally, the data used to generate Figure 1 also included timings for an Nvidia GTX 260 card, this achieved 1594 million grid points per second against the Nehalem's 1409 million per second, up from Harpertown's 394 million per second. This shows that the CPU's are catching up again, and for the next few years at least what is truly the fastest architecture should be very interesting indeed.

Acknowledgement

The author would like to thank Professor Mike Giles for all his help and support.

References

[1] *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*, 2nd ed., NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050, June 2008.

[2] R. Damelio, *Basics of Benchmarking*, 1st ed. 270 Madison Avenue, New York, NY, 10016, USA: Productivity Press, August 1995.

[3] B. K. Horn and B. G. Schunck, "Determining optical flow," Massachusetts Institute of Technology, Cambridge, MA, USA, Tech. Rep., 1980.

[4] *CUDA, CUBLAS Library*, 2nd ed., NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050, March 2008.

[5] *Intel Math Kernel Library, Reference Manual*, 10th ed., Intel, 2200 Mission College Blvd, Santa Clara, CA, 95054-1549, USA, August 2008.