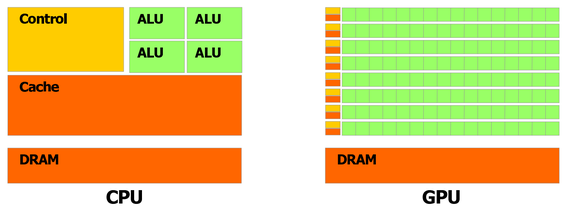
Scientific Programming with Graphical Processing Units (GPUs)

A. Leist, D.P. Playne and K.A. Hawick

Complex Systems & Simulations Group, Computer Science, Institute of Information and Mathematical Sciences, Massey University, Albany

Introduction

Graphical Processing Units (GPUs) [1] have emerged in recent years as an attractive platform for optimising the speed of a number of application paradigms relevant to the games and computer-generated character animation industries [2]. While chip manufacturers have been successful in incorporating ideas in parallel programming into their family of CPUs, inevitably due to the need for backwards compatibility, it has not been trivial to make major advances with complex CPU chips. The concept of the GPU as a separate chip with fewer legacy constraints is perhaps one explanation for the recent dramatic improvements in the use of parallel hardware and ideas at a chip level.



An illustration of the difference between CPU and GPU chip designs. Taken from the CUDA Reference Guide 2.0 [3].

In principle, GPUs can of course be used for many parallel applications in addition to their original intended purpose of graphics processing algorithms. In practice however, it has taken some relatively recent innovations in high-level programming language support and accessibility for the wider applications programming community to consider using GPUs in this way [4, 5, 6]. The term general-purpose GPU programming (GPGPU) [7] has been adopted to describe this rapidly growing applications level use of GPU hardware. Exploiting parallel computing effectively at an applications level remains a challenge for programmers. We focus on two broad application areas - partial differential field equations and graph combinatorics algorithms and explore the detailed coding techniques and performance optimisation techniques offered by the Compute Unified Device Architecture (CUDA) [6] GPU programming language. NVIDIA's CUDA has emerged in recent months as one of the leading programmer tools for exploiting GPUS. It is not the only one [8], and inevitably it builds on many past important ideas, concepts and discoveries in parallel computing.

GPUs

GPU architectures contain many processors which can execute instructions in parallel. These parallel processors provide the GPU with the high-performance parallel computational power. To make use of this parallel computational power, a program executing on the GPU must be decomposed into many parallel threads. Parallel decomposition of computational problems is a widely researched topic and is not discussed in detail here. However, one specific condition of parallel decomposition for GPUs is the number of threads the problem should be split into. As GPUs are capable of executing a large number of threads simultaneously, it is generally advantageous to have as many threads as possible. A large number of threads allows the GPU to maximise the efficiency of execution.

	GTX280	9800GTX	8800GTX
Texture Processing Clusters (TPC)	10	8	8
Streaming Multiprocs (SM) per TPC	3	2	2
32-bit Streaming Procs (SP) per SM	8	8	8
64-bit SPs per SM	1	N/A	N/A
SP Clock (MHz)	1296	1688	1350
Single-Precision Peak Perf. (GFLOP)	933	648	518
Double-Precision Peak Perf. (GFLOP)	78	N/A	N/A
Memory Bandwidth (GB/sec)	141.7	70.4	86.4
Std. Device Memory Config. (MB)	1024	512	768
Registers (32-bit) per SM	16384	8192	8192
Shared Memory per SM (KB)	16	16	16
Total Constant Memory (KB)	64	64	64
Constant Memory Cache per SM (KB)	8	8	8
Texture Memory Cache per SM (KB)	6-8	6-8	6-8
Max. Active Threads per SM	1024	768	768
Max. Total Active Threads per Chip	30720	12288	12288

GPUs contain several types of memory that are designed and optimised for different purposes, but first and foremost for the specific tasks of the graphics pipeline. Using the type of memory best suited to the task can provide significant performance benefits. The memory access patterns used to retrieve/store data from these memory locations is equally important as the type of memory used. Correct access patterns are critical to the performance of the GPU program. CUDA-enabled GPUs provide access to six different types of memory that are stored both on-chip (within a multiprocessor) and on the device.

Global Memory is part of the device memory. It is the largest memory on the GPU and is accessible by every thread in the program. However, it has no caching and has the slowest access time of any memory type - approximately 200-400 clock cycles. Lifetime = application.

Registers are stored on-chip and are the fastest type of memory. The registers are used to store the local variables of a single thread and can only be accessed by that thread. Lifetime = thread.

Local Memory does not represent an actual physical area of memory, instead it is a section of device memory used when the variables of a thread do not fit within the registers available. Lifetime = thread.

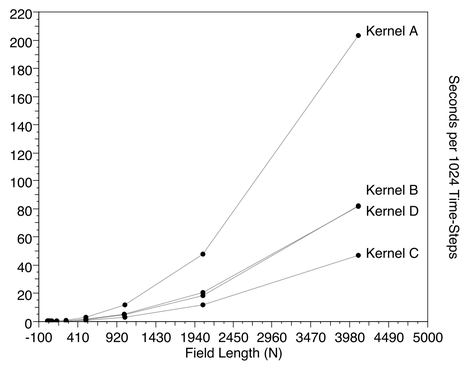
Shared Memory is fast on-chip memory that is shared between all of the threads within a single block. It can be used to allow the threads within a block to co-operate and share information. Lifetime = block.

Texture Memory space is a cached memory region of global memory. Every SM has its own texture memory cache on-chip. Device memory reads are only necessary on cache misses, otherwise a texture fetch only costs a read from the texture cache. Lifetime = application.

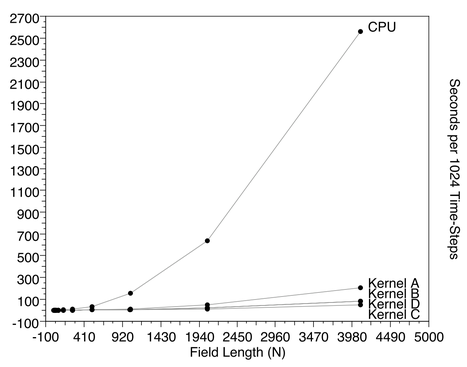
Constant Memory space is a cached, read-only region of device memory. Like the texture cache, every SM has its own constant memory cache on-chip. Device memory reads are only necessary on cache misses, otherwise a constant memory read only costs a read from the constant cache. Lifetime = application.

Simulations

One specific regular-mesh application we investigated used a finite-difference solver for a partial differential field equation such as the Cahn-Hilliard system. Not surprisingly, we found that a parallel geometric decomposition involving a large number of parallel threads that execute independently performed well on the GPUs. However less obviously, the performance was also heavily dependent on how effectively we used the different sorts of memory available on the GPU. We showed that the spatial caching of texture memory was critical to achieving the most dramatic performance improvements. Using the optimal GPU configuration we achieved a 50x speed-up factor as compared to the CPU implementation.

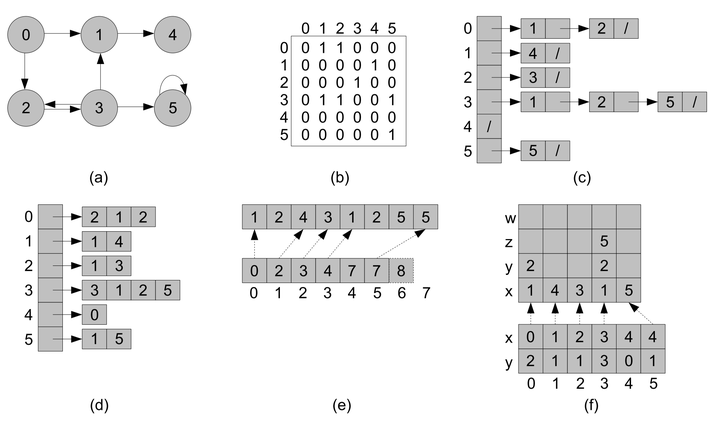


Performance differences when using global memory only (Kernel A), shared memory (Kernel B), texture memory (Kernel C) and texture + shared memory (Kernel D) in our Cahn-Hilliard equation simulation.



A comparison between the GPU implementations and our single-threaded CPU implementation written in C. All GPU programs were executed on an NVIDIA GeForce 8800 GTS operating with an Intel Core2 6600. The CPU program was executed on a 2.66 GHz Intel Xeon processor with 2 GBytes of memory.

We also investigated graph and network applications as typified by computation of the all-pairs, shortest-path metric on various sorts of scale-free and other graph instances. These algorithms are less obviously parallelisable, but we obtained significant performance speed-ups again from arranging the algorithm so that it could best exploit a multi-threaded architecture such as the particular GPUs that we used. The correct approach was less obvious for these problems and we experimented with several different parallelisation techniques. We found that it was possible to achieve a very significant speed-up over a typical CPU by having the application's network data suitably arranged in the GPU's fast and local memory.



Choosing the most suitable data structure for directed graphs on the GPU can yield significant performance gains. **(a)** Visual representation of the sample graph; **(b)** Adjacency-matrix; **(c)** Adjacency-list using an array of linked lists; **(d)** Adjacency-list using an array of arrays; **(e)** Separate arrays for the vertices and arcs. Each vertex points to its first entry in the adjacency-list; **(f)** Like (e), but groups arcs into sets of 4 to utilise hardware specific memory bandwidth optimisations.

Discussion

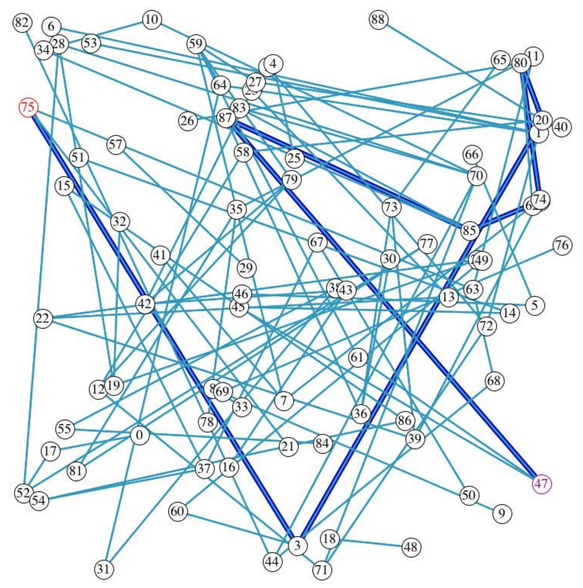
One of the problems in fairly assessing a specific piece of hardware such as a GPU is that the optimal algorithm we might use for an application on it might be appreciably different than the optimal one for a general purpose CPU.

One area of difficulty with the GPU platforms at present is the lack of instrumentation and timing facilities. It is not simple to monitor what the GPU is doing without interfering with the operations.

For the applications we have considered, the precision of floating point is not especially important. Generally, we would expect to solve field equations using the native precision of the CPU or GPU. In most cases that will be 64-bits for modern CPUs, but some GPUs still only offer 32-bit floating point. It is likely that almost all modern GPUs in the future will offer 64-bit or higher however.

An inevitable issue with data-parallel computing is that a real application may not lend itself to a single optimal data layout. Different parts of the application may favour different data layouts. An advantage of the combined CPU+GPU programming approach is that it is relatively easy to program data layout conversions to use the one that is optimal for the particular GPU kernel. Some scientific applications or speed optimised libraries might therefore have several different kernel routines, ostensibly doing the same calculation, but with each optimised for a different data layout.

Although we have approached two applications that are of intrinsic scientific interest for simulations in computational science, we believe these sorts of algorithms are also of strong relevance to games engines. Simulating fields for rending atmospheric or water effects can use a regular field equation approach. Navigating agents in a game or character scenario can also make use of path-finding algorithms. GPUs were originally developed for graphical algorithm optimisation, but it seems clear that they will find increasing uses as parts of simulated environment generation packages, physics engines[9] and other higher level application components of interactive games and animation systems.



A random graph with the shortest path between nodes 47 and 75 highlighted.

More information on our work on GPUs is described in technical note CSTN-065, available at `www.massey.ac.nz/~kahawick/cstn/065`. In that article we describe our ports of a Cahn-Hilliard field equation solver and some selected graph network shortest path finding algorithms to various GPUs and discuss the general utility of GPU architectures to scientific programming applications.

There will continue to be interesting research opportunities on tradeoff analyses to determine what balance of computation is best located on a CPU and its co-GPU(s).

References

[1] K. Fatahalian and M. Houston, "A Closer Look at GPUs," *Communications of the ACM*, vol. 51, pp. 50–57, October 2008.

[2] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. Purcell, "A survey of general-purpose computation on graphics hardware," in *Eurographics 2005, State of the Art Reports*, pp. 21–51, September 2005.

[3] NVIDIA® Corporation, *CUDA™ 2.0 Programming Guide*, 2008. Last accessed November 2008.

[4] W. Langdon and W. Banzhaf, "A SIMD Interpreter for Genetic Programming on GPU Graphics Cards," in *Proc. EuroGP* (M. O'Neill, L. Vanneschi, S. Gustafson, A. E. Alcazar, I. D. Falco, A. D. Cioppa, and E. Tarantino, eds.), vol. LNCS 4971, pp. 73–85, March 2008.

[5] P. Messmer, P. J. Mullowney, and B. E. Granger, "GPULib: GPU computing in high-level languages," *Computing in Science & Engineering*, vol. 10, pp. 70–73, September/October 2008.

[6] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *ACM Queue*, vol. 6, pp. 40–53, March/April 2008.

[7] M. McCool and S. D. Toit, *Metaprogramming GPUs with Sh*. A K Peters, Ltd., 2004.

[8] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: stream computing on graphics hardware," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 777–786, 2004.

[9] D. Conger and A. LaMothe, *Physics Modeling for Game Programmers*. Thompson, 2004.