GPU Computing: More exciting times for HPC!

Paul G. Howard, Ph.D.
Chief Scientist, Microway, Inc.

Introduction

In the never-ending quest in the High Performance Computing world for more and more computing speed, there's a new player in the game: the general purpose graphics processing unit, or GPU. The GPU is the processor used on a graphics card, optimized to perform simple but intensive processing on many pixels simultaneously. Taking a step back from its use in graphics, the GPU can be thought of as a high-performance SPMD (single-process multiple-data) processor capable of extremely high computational throughput, at low cost and with a far higher computational throughput-to-power ratio than is possible with CPUs.

Performance speedups between 10x and 1000x have been reported using GPUs on real-world applications. For most people, a performance improvement of even one order of magnitude does not simply increase productivity. It will allow previously impractical projects to be completed in a timely manner. With three orders of magnitude improvement, a one minute simulation may be viewed real-time at 15 frames per second. GPU computing will change the way you think about your research!

In late 2006, NVIDIA and ATI (acquired by AMD in 2006) began marketing GPU boards as streaming coprocessor boards. Both companies released software to expose the processor to application programmers. Since then the hardware has grown more powerful and the available software has become more high-level and developer-friendly.

Hardware specifications and comparisons

The GPU cards currently on the market are available from NVIDIA and AMD. The flagship product from NVIDIA is the Tesla C1060, and from AMD it's the FireStream 9270. The following table compares the specifications of the two products.

	NVIDIA™ Tesla™ C1060	**AMD™ FireStream™ 9270**
GPU	Tesla T10	ATI Radeon HD 4870 (RV 770)
Transistors	1.4 billion	0.96 billion
Process	65 nm	55 nm
Streaming processor cores	240 (Note 1)	800 (Note 2)
Processor clock frequency	1296 MHz	750 MHz
Theoretical single precision performance	933 Gflops (Note 1)	1200 Gflops
Measured single precision performance (Note 3)	243 Gflops	171 Gflops
Theoretical double precision performance	78 Gflops	240 Gflops (Note 2)

	NVIDIA™ Tesla™ C1060	AMD™ FireStream™ 9270
Floating point	IEEE 754 (Note 4)	IEEE 754 (Note 4)
Memory (Note 5)	4 GB GDDR3 SDRAM	2 GB GDDR5 SDRAM
Memory interface bus width	512 bits	256 bits
Theoretical device memory bandwidth	102 GB/s	108.8 GB/s
Measured device bandwidth	94 GB/s	47 GB/s
System interface	PCI Express (PCIe) x16 Gen 2	PCI Express (PCIe) x16 Gen 2
Theoretical host-to-device b/w	8 GB/s	8 GB/s
Measured host-to-device b/w	6 GB/s	4.6 GB/s
Form factor	4.376" x 10.5", dual slot	4.376" x 9.5", dual slot
Power connectors	Two 6-pin or one 8-pin	Two 6-pin
Power consumption	160 W typical, 188 W peak	160 W typical, <220 W peak
Cooling	Active fan sink	Active fan sink

Note 1: The Tesla C1060 has 30 multiprocessors, each with 8 scalar processor single-precision/integer units, 1 double-precision multiply/add unit, and 2 special function units. Each scalar processor can perform 3 floating point operations (1 multiply/add, 1 multiply, and 1 other function) per clock cycle. The theoretical peak performance is 1296 MHz x 240 cores x 3 operations per clock, or 933 Gflops.

Note 2: The FireStream 9270 has 160 shaders, each with 5 ALUs (arithmetic/logic units), 4 for multiply-add/add/multiply/integer add/dot product and 1 for transcendental functions). The 4 non-transcendental ALUs in a shader can be used together to get 1 double precision ALU. Based on this, the AMD double precision performance is estimated as 1/5 of the single precision performance.

Note 3: Actual performance figures based on a careful implementation of a radio astronomy application from van Nieuwpoort and Romein, "Using Many-Core hardware to Correlate Radio Astronomy Signals", ACM International Conference on Supercomputing, 2009. As in all discussions of HPC performance, the actual performance you obtain depends strongly on the details of your application.

Note 4: Both GPUs provide slightly reduced precision on single precision division and square root.

Note 5: GDDR5 memory uses a bus only half as wide as GDDR3, but sends twice as much data per clock (by using a second double-rate clock for data), so the amount of data transmitted per clock is the same. The wider GDDR3 bus requires more pads and a bigger die.

Software support

General purpose computing with GPUs became feasible when vendors made it possible to program them without using graphics primitives. Higher level control has gradually become available, to the point where now both NVIDIA and AMD offer a tool chain based on a slightly enhanced version of C. The basic programming model is to write ordinary C code that runs on the CPU for sequential operations and for control, with the parallel code segregated into *kernels* (data parallel functions) that run in parallel on the GPU.

NVIDIA's software development kit is based on their Compute Unified Device Architecture (CUDA). CUDA is proprietary, and according to NVIDIA it is likely to remain so for rapid adaptability to hardware advances. The key features of CUDA are its thread hierarchy, use of shared device memory, and barrier synchronization. CPU and GPU code can run concurrently.

Programmers typically program in “C for CUDA”, a version of C with extensions for defining kernels and managing device memory. (“C for CUDA” is often mistakenly called “CUDA”. “C for CUDA” is the language; “CUDA” is the architecture.) C for CUDA code compiles to PTX (Parallel Thread Execution), NVIDIA's intermediary assembly language, which is translated at an even lower level to GPU instructions. Programmers have access to PTX when necessary. Device details are left to the runtime system.

NVIDIA supplies the CUDA core library and CUDA runtime for executing CUDA codes, as well as several libraries: CUDPP for some data parallel primitive operations, cuFFT for Fast Fourier Transforms, and cuBLAS for linear algebra.

CUDA and programming in C for CUDA are well-documented, and are already being taught in a number of universities.

AMD provides a similar software stack. They provide Brook+, a C-like language based on the open-source Brook language developed at Stanford. At a lower level, programmers have access to the Compute Abstraction Layer (CAL) and the assembly-language-like Intermediate Language (IL).

AMD also provides the AMD Computation and Math Library (ACML), comprised of BLAS and LAPACK for linear algebra, FFT, math transcendental, and pseudo-random number generation libraries. Only the SGEMM and DGEMM parts of BLAS have been accelerated for GPU processing.

Both NVIDIA and AMD will support the emerging OpenCL language, being created by the Khronos Group. OpenCL will be an OpenGL-compatible language, designed to allow cross-platform development of GPU applications. It is at roughly the same level as C for CUDA and Brook+, but with direct access to lower-level processor functions. The OpenCL 1.0 specification was released in late 2008, but there are no implementations as of early 2009.

AccelerEyes, a third-party software vendor, produces Jacket, a GPU accelerator for MATLAB® that runs on CUDA-enabled GPUs. They also offer the Jacket Graphics Toolbox for visual computing on MATLAB.

Programming

Programming a GPU is not like modifying a sequential program by segmenting your algorithm to make it use several cores in parallel. Instead, in a nutshell, you can get good GPU results by dividing your massively parallel program into lots of threads using whatever cores you have. The number of threads should be much larger than (10 to 100 times) the number of cores. Keep the processors fed, and let the hardware take care of the details. Thus the first step is to use algorithms that expose fine-grained parallelism.

Because GPUs are not like CPUs, you can't effectively program them using CPU-programming rules. On a GPU, arithmetic is cheap, memory accesses are expensive, accessing host memory is prohibitively expensive, and branching is problematic at best (divergent branches run sequentially instead of

concurrently). Thus, it's often better to do extra computations to avoid conditional code, special cases, and memory accesses. Coalesce memory reads and writes as much as possible.

Some of the usual CPU-type optimizations still apply, such as lifting code out of loops, unrolling loops to keep data in registers, and storing constants in constant memory.

To get full benefit of the GPU, you have to understand the underlying architecture. This means knowing efficient data access patterns, including data alignment, and making good use of multiple memory systems (registers, shared memory, global device memory, texture memory, constant memory, and host memory). At this level, profilers are invaluable: use them!

Applications

GPUs are already being used in a number of scientific, engineering, and commercial fields, including molecular modeling, computational finance, computational fluid dynamics, energy exploration, and multimedia processing. In this section we list a number of applications; for many of them we give a brief description of how GPUs are used.

Financial applications

- Black-Scholes option pricing. Simulation of distribution functions that depend on volatility and strike price.
- Lattice methods for option pricing. Dynamic programming.
- Finite difference methods for solving partial differential equations for option pricing.
- Portfolio optimization.
- Credit card fraud. Scanning millions of accounts for suspicious patterns.
- Hedge fund trading. Correlation analysis of historical data.
- Risk analysis. Simulations.

Energy, gas, and oil exploration

- 3D seismic analysis. FFT.
- Geophysical visualization. Filtering and reducing raw seismic data.

Life sciences

- Protein folding (Stanford folding@home project).
- Smith-Waterman sequence alignment. Dynamic programming for approximate alignment.
- MUMmerGPU exact sequence alignment using suffix trees.
- HMMer protein sequence analysis.
- Combinatorial chemistry for drug design.
- Hidden Markov models.
- Protein docking.
- Fluorescence microscopy simulation.
- Neuron simulation.

Health care

- Medical imaging: deformable image registration (lung, prostate).
- MRI processing.

- FHD-spiral MRI reconstruction.
- Medical device design.

Engineering

- Finite element analysis.
- Crash modeling.
- Cell phone antenna simulation.

Multimedia processing

- Face tracking.
- KLT feature tracking.
- Audio: acoustic simulations using ray-tracing.
- Image analysis.
- Denoising (k nearest neighbors).
- Convolution.
- Discrete wavelet transform.
- Vector signal image processing.

Physics and chemistry

- Atmospheric science. Cloud simulation.
- Ocean and space science.
- Computational electromagnetics and electrodynamics.
- Computational electrostatics. ion placement.
- Computational fluid dynamics. Euler solvers for nonlinear PDEs (Navier-Stokes equations).
- Quantum chemistry. Electron repulsion integrals.
- Astrophysics. N-body simulations.
- Molecular dynamics. Non-bonded force calculations.

Other

- Geographical information systems (GIS).
- Symmetric key cryptography.
- Relational database joins.

High-performance applications

A number of NVIDIA users have reported their performance speedups at NVIDIA's "CUDA Zone" web site. The greatest speedup was reported by Alerstam, Svensson, and Anderssen-Engels in "Monte Carlo simulation of photon migration", *Journal of Biomedical Optics* 13:6, 2008: they achieved a speedup factor of 1080 using an NVIDIA 8800GT GPU.

Here are some of the more impressive reports, all reporting a speedup of two orders of magnitude or more.

- 1080x: Monte Carlo simulation of photon migration
- 675x: stochastic differential equations
- 470x: k nearest neighbor search

- 420x: generalized harmonic analysis
- 340x: power system simulation
- 300x: cone beam computed tomography
- 270x: Particle swarm optimization
- 263x: FHD-spiral MRI reconstruction
- 250x: N-body simulation
- 172x: support vector machine training and classification
- 169x: signal and image reconstruction
- 130x: quantum chemistry two-electron integral evolution
- 120x: 3D particle Boltzmann solver
- 109x: sliding window object detection
- 100x: visualization of volumetric white matter connectivity
- 100x: prestack seismic data interaction
- 100x: folding@home
- 100x: pricing and risk management of exotic financial structures
- 100x: incompressible Navier-Stokes solver

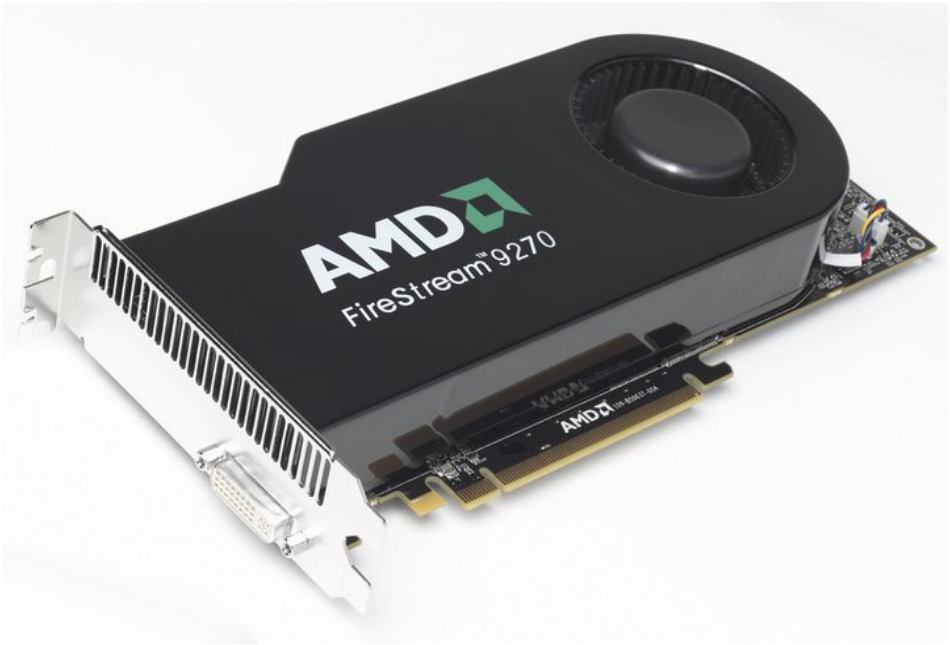
You can see that many of the applications most amenable to GPU speedups involve multidimensional computations, especially simulations.

Questions and answers

What do the GPU boards look like?

Here's the NVIDIA Tesla C1060:

And here's the AMD FireStream 9270:



How is a GPU different from a CPU?

A typical CPU has a few complex arithmetic/logic units (ALUs) and considerable control logic. It has multiple levels of memory on-chip; cache memory takes up a large part of the chip real estate. In a GPU, by contrast, most of the chip area is taken up with a large number of relatively simple ALUs. There is very little control logic, and limited cache.

Is it easy to program a GPU?

It's different. Many of the optimizations for CPU programming do not apply to GPU programs. To get maximum performance, you have to understand the architecture thoroughly, and be willing to invest time in profiling different potential programming solutions.

If you can use existing libraries (for linear algebra, FFT, or multimedia transcoding, for example), then programming becomes relatively straightforward.

What kind of application will benefit most from a GPU?

GPUs work best on problems with fine-grained parallelism that can provide sufficient work to lots of threads. The most likely candidates for big performance gains are those that involve long-running calculations that can be split up into tens of thousands (or more) of independent threads. Simulations over fine-grained grids and lattices (as in computational fluid dynamics) involving multidimensional spaces, in which each simulation pass requires solving a system of (differential) equations or computing a probability distribution, are ideal candidates. Optimization problems have similar characteristics: trying out lots of independent potential solutions. Visualizations are good candidates too, as are search/filtering/ data reduction problems.

If your application involves heavy use of linear algebra or FFTs, you should be able to use libraries

(BLAS, LAPACK, FFT, etc.) developed by the hardware vendors. Since these are already optimized for the specific architecture, you can get great performance off-the-shelf.

Will I be able to obtain the stated peak performance?

If you are doing standard calculations that can keep the GPU busy with lots of threads and minimal memory access, you can expect performance within range of the stated peak. Performance on a specific application is dependent on a number of factors. Amdahl's law means that only the parallel parts of your code can be sped up at all, and data transfers between the host memory and GPU device memory can be a serious limiting factor. In addition, to obtain peak performance you must understand the details of the hardware architecture, and take the time and effort to map your algorithms to the programming and memory models. So for custom-coded applications, your performance will likely be less than the stated peak performance.

What kind of performance improvement can I expect?

If the bottleneck is memory bandwidth, expect 40-80x. If the bottleneck is calculation, you can expect much more speedup: 200-500x.

Will OpenCL allow cross-platform development?

That's the goal. There will probably be some performance overhead since OpenCL is less closely tied to the underlying hardware architecture than C for CUDA or Brook+.

The OpenCL specification has been released. Implementations for specific hardware platforms are expected in mid-2009.

Can I mix C for CUDA, Brook+, or OpenCL code with MPI code?

Yes. MPI is most suited to applications and algorithms exhibiting coarse-grained parallelism, which are largely independent of and can coexist with the fine-grained parallelism addressed by GPU code. For the fine-grained parts of your application, you will find that GPU computing scales much better.

Can I mix C for CUDA, Brook+, or OpenCL code with OpenMP or POSIX threads code?

Possibly, but if your algorithms can take advantage of the fine-grained parallelism characteristic of OpenMP and POSIX threads on a CPU, they are good candidates for much larger speedups when redirected to a GPU.

Can I add GPUs to my existing cluster, server, or workstation?

Yes. In fact, NVIDIA provides the Tesla S1070 computing system, a 1U system containing four Tesla GPUs, that can connect to any server or compute node with an available PCI Express slot.

You can add a GPU to a workstation if the workstation has a large enough power supply. For smaller power supplies, you can still add a somewhat less powerful but still useful GPU like the CUDA-enabled NVIDIA GeForce (8 series and higher).

Microway engineers can help you to design GPU-enabled configurations compatible with your existing hardware setup.

Can I use more than one GPU in a system?

Most existing software does not take advantage of multiple GPUs, but you can run multiple instances of your application and still keep more than one GPU busy. If you write your own software, you can make full use of all the GPUs in your system by making explicit calls to individual GPUs.

Conclusion

From the hardware point of view, both the Tesla C1060 and the FireStream 9270 offer about 1 Tflops theoretical peak single precision floating point performance. The FireStream 9270 has about triple the double precision performance. Both offer 100 MB/s device memory bandwidth. Each consumes about 160 W, for about 6 Gflops/W peak efficiency. By contrast, CPUs consume about 95 W and provide less than 100 Gflops performance, for about 1 Gflops/W efficiency. Measured computational and memory performance are somewhat lower for both GPUs.

The challenge in GPU computing is programming. Both NVIDIA and AMD offer a software stack featuring a C-like language and access to lower level code. Both have libraries for common computational requirements, such as linear algebra and FFT. The software situation is about to improve even more with the introduction of OpenCL for cross-platform development.

For many applications that can naturally be highly parallelized, the software challenge is not so great. For such applications, the strategy of breaking the problem into millions of tiny parts, feeding the parts to the GPU, and letting the hardware take care of the scheduling and other details has already proven to be effective. The impressive results reported to date, especially for simulations, optimization problems, and multi-dimensional computations, show that GPU processing is already here.