Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

CUDA, Supercomputing for the Masses: Part 1

CUDA lets you work with familiar programming concepts while developing software that can run on a GPU

April 15, 2008
URL:http://www.drdobbs.com/parallel/cuda-supercomputing-for-the-masses-part/207200659

Are you interested in getting orders-of-magnitude performance increases over standard multi-core processors, while programming with a high-level language such as C? And would you like that capability to scale across many devices as well?

Many people (myself included) have achieved this level of performance and scalability on non-trivial problems by using CUDA (short for "Compute Unified Device Architecture") from NVIDIA to program inexpensive multi-threaded GPUs. I purposefully stress "programming" because CUDA is an architecture designed to let you do your work, rather than forcing your work to fit within a limited set of performance libraries. With CUDA, you get to exploit your abilities to design software to achieve best performance on your multi-threaded hardware -- and have fun as well because figuring out the right mapping is captivating, plus the software development environment is both reasonable and straightforward.

This is the first of a series of articles to introduce you to the power of CUDA -- through working code -- and to the thought process to help you map applications onto multi-threaded hardware (such as GPUs) to get big performance increases. Of course, not all problems can be mapped efficiently onto multi-threaded hardware, so part of my thought process will be to distinguish what will and what won't work, plus provide a common-sense idea of what might work "well-enough".

"CUDA programming" and "GPGPU programming" are not the same (although CUDA runs on GPUs). Previously, writing software for a GPU meant programming in the language of the GPU. An acquaintance of mine once described this as a process similar to pulling data out of your elbow to get it to where you could look at it with your eyes. CUDA permits working with familiar programming concepts while developing software that can run on a GPU. It also avoids the performance overhead of graphics layer APIs by compiling your software directly to the hardware (GPU assembly language, for instance), thereby providing great performance.

The choice of CUDA device is up to you. Figures 1 and 2 show the CUDA N-body simulation program running on both a laptop and a discrete GPU based desktop PC.

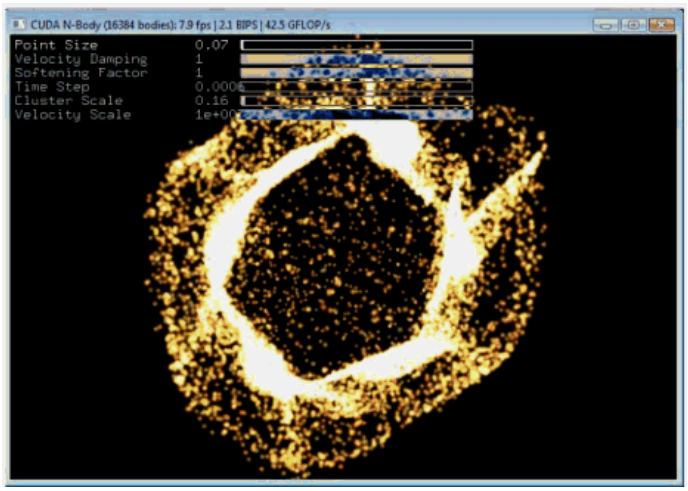

Figure 1: nBody Astrophysics Simulation running on a Quadro FX 570M enabled laptop.

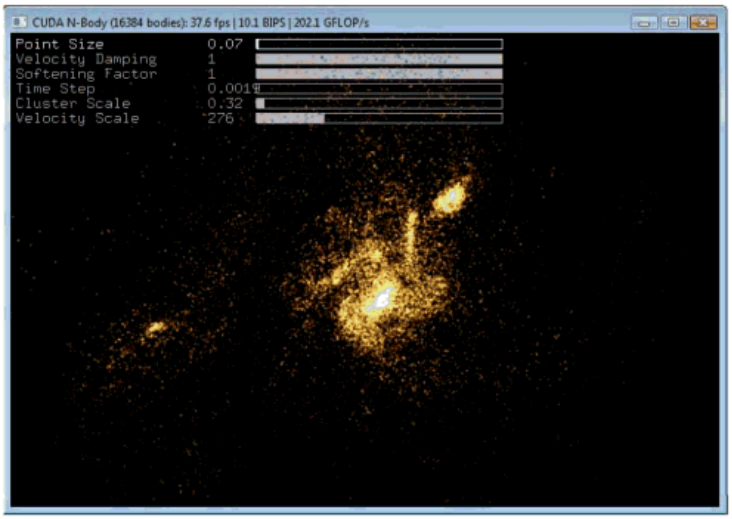

Figure 2: nBody Astrophysics Simulation running on a GeForce 8800 GTS 512MB enabled desktop

Can CUDA really increase application performance by one to two orders of magnitude -- or is all this hype rather than reality?

CUDA is a fairly new technology but there are already many examples in the literature and on the Internet highlighting significant performance boosts using current commodity GPU hardware. Tables 1 and 2 show summaries posted on the NVIDIA and Beckman Institute websites. At the heart of CUDA is the ability for programmers to keep thousands of threads busy. The current generation of NVIDIA GPUs can efficiently support a very large number of threads, and as a result they can deliver one to two orders of magnitude performance increase in application performance. These graphics processors are widely available to anyone at almost any price point. Newer boards will expand CUDA's capabilities by providing greater memory bandwidth, asynchronous data transfer, atomic operations, and double-precision floating point arithmetic among many hardware improvements. Look for the CUDA software environment to expand as the technology evolves and we eventually lose the distinction between GPUs and "many-core" processors. As developers, we have to anticipate that applications with many thousands of active threads will become common-place and look for CUDA to run on many platforms, including general-purpose processors.

Example Applications	URL	Application Speedup
Seismic Database	http://www.headwave.com	66x to 100x
Mobile Phone Antenna Simulation	http://www.acceleware.com	45x
Molecular Dynamics	http://www.ks.uiuc.edu/Research/vmd	21x to 100x
Neuron Simulation	http://www.evolvedmachines.com	100x
MRI processing	http://bic-test.beckman.uiuc.edu	245x to 415x
Atmospheric Cloud Simulation	http://www.cs.clemson.edu/~jesteel/clouds.html	50x

Table 1: NVIDIA summary from www.nvidia.com/object/IO_43499.html

GPU Performance Results, March 2008

GeForce8800GTX w/ CUDA 1.1, Driver 169.09

Calculation / Algorithm	Algorithm class	Speedup vs. Intel QX6700 CPU
Fluorescence microphotolysis	Iterative matrix / stencil	12x
Pairlist calculation	Particle pair distance test	10x to 11x
Pairlist update	Particle pair distance test	5x to 15x
Molecular dynamics nonbonded force calculation	N-body cutoff force calculations	10x to 20x
Cutoff electron density sum	Particle-grid w/ cutoff	15x to 23x
Cutoff potential summation	Particle-grid w/ cutoff	12x to 21x
Direct Coulomb summation	Particle-grid	44x

Table 2: Beckman Institute table from www.ks.uiuc.edu/Research/vmd/publications/siam2008vmdcuda.pdf

As a scientist at Los Alamos National Laboratory in the 1980s, I had the pleasure of working with the massively parallel 65,536 processor Thinking Machines supercomputers. CUDA has proved to be a natural framework to again start working in a modern massively-parallel (i.e., highly-threaded) environment. Performance is clearly there. One of my production codes, now written in CUDA and running on NVIDIA GPUs, shows both linear scaling and a nearly two orders of magnitude speed increase over a 2.6-Ghz quad-core Opteron system.

CUDA-enabled graphics processors operate as co-processors within the host computer. This means that each GPU is considered to have its own memory and processing elements that are separate from the host computer. To perform useful work, data must be transferred between the memory space of the host computer and CUDA device(s). For this reason, performance results must include IO time to be informative. Colleagues have also referred to these as "Honest Flops" because they more accurately reflect the performance applications will deliver in production.

I claim that a one or two orders of magnitude performance increase over existing technology is a disruptive change that can dramatically alter some aspects of computing. For example, computational tasks that previously would have taken a year can now complete in a few days, hour long computations suddenly become interactive because they be completed in seconds with the new technology, and previously intractable real-time processing tasks now becomes tractable. Finally, lucrative opportunities can present themselves for consultants and engineers with the right skill set and capabilities to write highly-threaded (or massively parallel) software. What about you? How can this type of computing capability benefit your career, applications or real-time processing needs?

Getting started costs nothing and is as easy as downloading CUDA from the CUDA Zone homepage (look for "Get CUDA"). After that, follow the installation instructions for your particular operating system. You don't even need a graphics processor because you can start working right away by using the software emulator to run on your current laptop or workstation. Of course, much better performance will be achieved by running with a CUDA-enabled GPU. Perhaps your computer already has one. Check out the "CUDA-enabled GPUs" link on the CUDA Zone homepage to see. (A CUDA-enabled GPU includes shared on-chip memory and thread management.)

If purchasing a new graphics processor card, I suggest following this article series because I will discuss how various hardware characteristics (such as memory bandwidth, number of registers, atomic operations, and so on) will affect application performance, which will assist you in selecting the appropriate hardware for your application. Also, the CUDA Zone forums provide a wealth of information on all things CUDA, including discussions about what hardware to purchase.

Once installed, the CUDA Toolkit provides a reasonable set of tools for C language application development. This includes:

- The nvcc C compiler

- CUDA FFT and BLAS libraries for the GPU
- A profiler
- An alpha version (as of March 2008) of the gdb debugger for the GPU
- CUDA runtime driver (now also available in the standard NVIDIA GPU driver)
- CUDA programming manual

The nvcc C compiler does most of the work in converting C code into an executable that will run on a GPU or the emulator. Happily, assembly-language programming is not required to achieve high performance. Future articles will discuss working with CUDA from other high-level languages including C++, FORTRAN, and Python. I assume that you're familiar with C/C++. No previous parallel programming or CUDA experience is required. This is consistent with the existing CUDA documentation.

Creating and running a CUDA C language program follows the same workflow as other C programming environments. Explicit build and run instructions for Windows and Linux environments are in the CUDA documentation, but simply stated they are:

1. Create or edit the CUDA program with your favorite editor. Note: CUDA C language programs have the suffix ".cu".
2. Compile the program with nvcc to create the executable. (NVIDIA provides sane makefiles with the examples. Generally all you need to type is "make" to build for a CUDA device or "make emu=1" to build for the emulator.)
3. Run the executable.

Listing One is a simple CUDA program to get you started. It is is nothing more than a program that calls the CUDA API to move data to and from the CUDA device. Nothing new is added that might cause confusion in learning how to use the tools to build and run a CUDA program. In the next article, I will discuss what is going on and start using the CUDA device to perform some work.

Listing One

```
// moveArrays.cu
//
// demonstrates CUDA interface to data allocation on device (GPU)
// and data movement between host (CPU) and device.

#include <stdio.h>
#include <assert.h>
#include <cuda.h>
int main(void)
{
   float *a_h, *b_h;     // pointers to host memory
   float *a_d, *b_d;     // pointers to device memory
   int N = 14;
   int i;
   // allocate arrays on host
   a_h = (float *)malloc(sizeof(float)*N);
   b_h = (float *)malloc(sizeof(float)*N);
   // allocate arrays on device
   cudaMalloc((void **) &a_d, sizeof(float)*N);
   cudaMalloc((void **) &b_d, sizeof(float)*N);
   // initialize host data
   for (i=0; i<N; i++) {
      a_h[i] = 10.f+i;
      b_h[i] = 0.f;
   }
   // send data from host to device: a_h to a_d
   cudaMemcpy(a_d, a_h, sizeof(float)*N, cudaMemcpyHostToDevice);
   // copy data within device: a_d to b_d
   cudaMemcpy(b_d, a_d, sizeof(float)*N, cudaMemcpyDeviceToDevice);
   // retrieve data from device: b_d to b_h
   cudaMemcpy(b_h, b_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
   // check result
   for (i=0; i<N; i++)
      assert(a_h[i] == b_h[i]);
   // cleanup
   free(a_h); free(b_h);
   cudaFree(a_d); cudaFree(b_d);
```

Give it a try and play around with the development tools. A quick note to newbies: You can use `printf` statements to see what is happening on the GPU when running under the emulator (build the executable with `make emu=1`). Also, feel free to try out the alpha version of the debugger.

Rob Farber is a senior scientist at Pacific Northwest National Laboratory. He has worked in massively parallel computing at several national laboratories and as co-founder of several startups. He can be reached at rmfarber@gmail.com.

For More Information

- CUDA, Supercomputing for the Masses: Part 20
- CUDA, Supercomputing for the Masses: Part 19
- CUDA, Supercomputing for the Masses: Part 18
- CUDA, Supercomputing for the Masses: Part 17
- CUDA, Supercomputing for the Masses: Part 16