CUDA | Software RAID | Roadrunner | Python | GPU Programming

LINUX JOURNAL

Since 1994: The Original Magazine of the Linux Community
NOVEMBER 2008 | ISSUE 175

ROADRUNNER'S COMPUTING POWER

Supercomputing at Your Fingertips with CUDA

REVIEWED:
- NolaPro
- Popcorn Hour A-100

INCREASE PERFORMANCE WITH SOFTWARE RAID

DEBUGGING PROGRAMS ON A CELL PROCESSOR

INTERVIEW WITH CORY DOCTOROW

+ Using Python for Scientific Computing

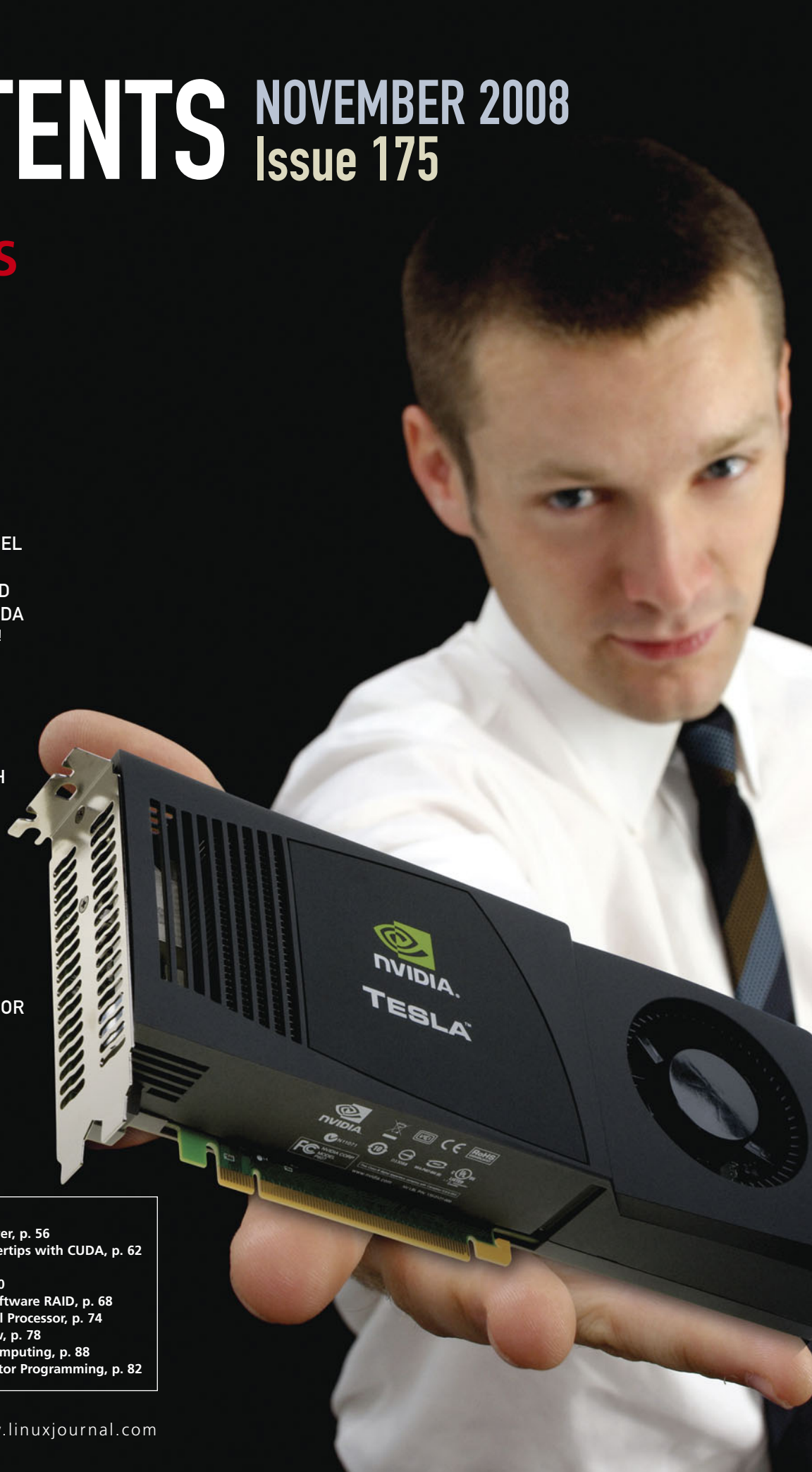
Simplifying GPU and Accelerator Programming

www.linuxjournal.com
$5.99US $5.99CAN
11>
0 09281 03102 4

CONTENTS

NOVEMBER 2008
Issue 175

FEATURES



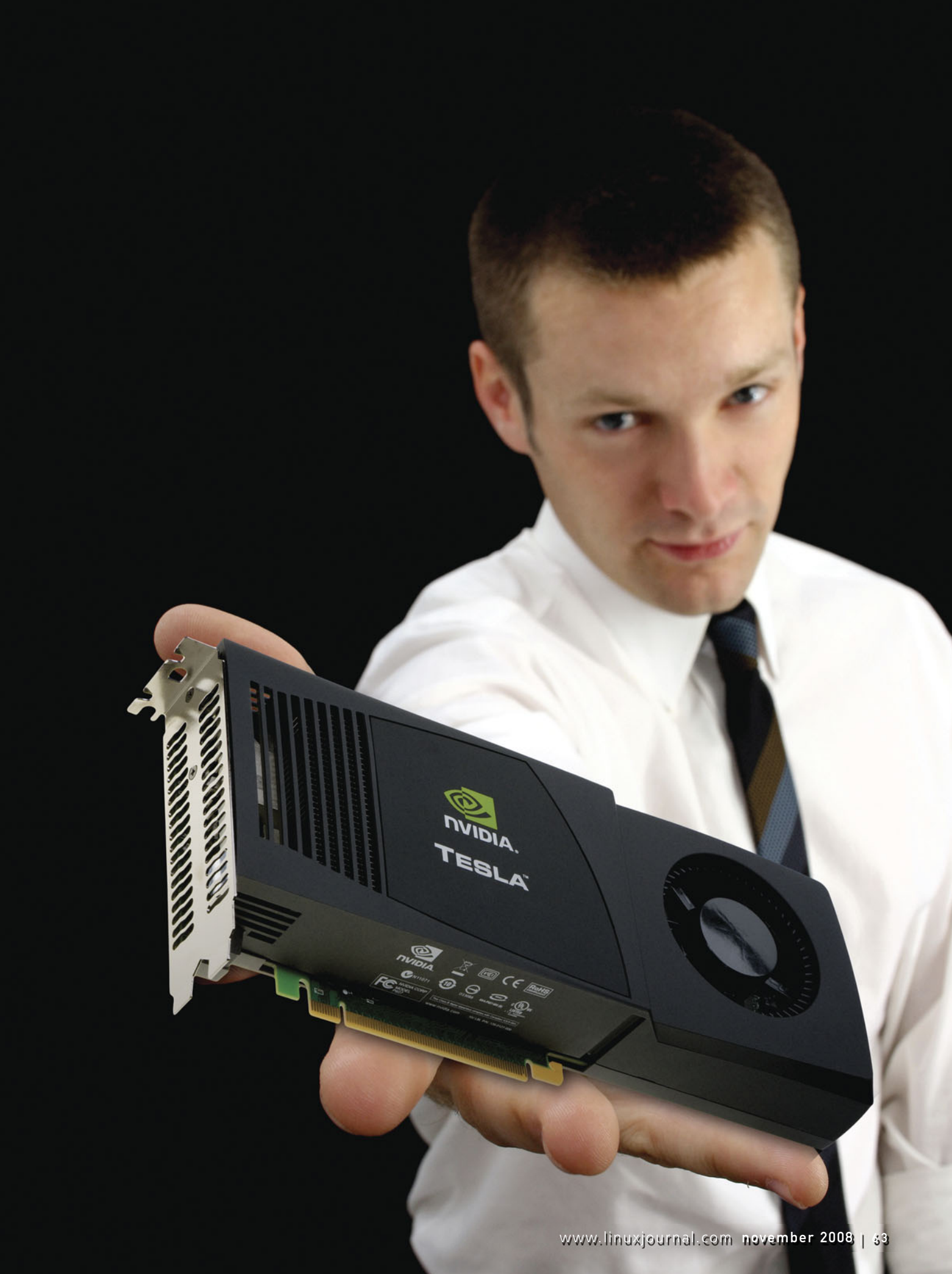
ON THE COVER
- Roadrunner's Computing Power, p. 56
- Supercomputing at Your Fingertips with CUDA, p. 62
- Reviewed: NolaPro, p. 46
- Reviewed: Popcorn Hour, p. 50
- Increase Performance with Software RAID, p. 68
- Debugging Programs on a Cell Processor, p. 74
- Interview with Cory Doctorow, p. 78
- Using Python for Scientific Computing, p. 88
- Simplifying GPU and Accelerator Programming, p. 82

Massively Parallel Linux Laptops, Workstations and Clusters with CUDA

Use an NVIDIA GPU or a cluster of them to realize massive performance increases.

ROBERT FARBER

NVIDIA's CUDA (Compute Unified Device Architecture) makes programming and using thousands of simultaneous threads straightforward. CUDA turns workstations, clusters—and even laptops—into massively parallel-computing devices. With CUDA, Linux programmers can address real-time programming and computational tasks previously possible only with dedicated devices or supercomputers.


NVIDIA
TESLA

Do you want to use hundreds of processing cores with Linux today? If so, take a look at running CUDA on the NVIDIA graphics processor in your Linux system. CUDA is a way to write C code for various CUDA-enabled graphics processors. I have personally written a C language application that delivers more than 150GF/s (billion floating-point operations per second) on a Linux laptop with CUDA-enabled GPUs. That same application delivers many hundreds of GF/s on a workstation and teraflop performance on a Linux cluster with CUDA-enabled GPUs on each node!

Do you need to move lots of data and process it in real time? Current CUDA-enabled devices use PCI-E 2.0 x16 buses to move data around quickly between system memory and amongst multiple graphics processors at GB/s (billion bytes per second) rates. Data-intensive video games use this bandwidth to run smoothly—even at very high frame rates. That same high throughput can enable some innovative CUDA applications.

One example is the RAID software developed by researchers at the University of Alabama and Sandia National Laboratory that transforms CUDA-enabled GPUs into high-performance RAID accelerators that calculate Reed-Solomon codes in real time for high-throughput disk subsystems (according to "Accelerating Reed-Solomon Coding in RAID Systems with GPUs" by Matthew Curry, Lee Ward, Tony Skjellum and Ron Brightwell, IPDPS 2008). From their abstract, "Performance results show that the GPU can outperform a modern CPU on this problem by an order of magnitude and also confirm that a GPU can be used to support a system with at least three parity disks

Do you want to use hundreds of processing cores with Linux today?

with no performance penalty." I'll bet the new NVIDIA hardware will perform even better. My guess is we will see a CUDA-enhanced Linux md (multiple device or software RAID) driver in the near future. [See Will Reese's article "Increase Performance, Reliability and Capacity with Software RAID" on page 68 in this issue.]

Imagine the freedom of not being locked in to a proprietary RAID controller. If something breaks, simply connect your RAID array to another Linux box to access the data. If that computer does not have an NVIDIA GPU, just use the standard Linux software md driver to access the data. Sure, the performance will be lower, but you still will have immediate access to your data.

Of course, CUDA-enabled GPUs can run multiple applications at the same time by time sharing—just as Linux does. CUDA devices have a very efficient hardware scheduler. It's fun to "wow" people by running a floating-point-intensive program while simultaneously watching one or more graphics-intensive applications render at a high frame rate on the screen.

CUDA is free, so it's easy to see what I mean. If you already have a CUDA-enabled GPU in your system (see **www.nvidia.com/object/cuda_learn_products.html** for compatible models), simply download CUDA from the NVIDIA Web site (**www.nvidia.com/cuda**), and install it. NVIDIA provides the source code for a number of working examples. These examples are built by simply typing `make`.

Check out the gigaflop performance of your GPU by running one of the floating-point-intensive applications. Start a graphics-intensive application, such as the glxgears (an OpenGL demo application), in another window and re-run the floating-point application to see how well the GPU simultaneously handles both applications. I was really surprised by how well my GPUs handled this workload.

Don't have a CUDA-enabled GPU? No problem, because CUDA comes with an emulator. Of course, the emulator will not provide the same level of performance as a CUDA-enabled GPU, but you still can build and run both the examples and your own applications. Building software for the emulator is as simple as typing `make emu=1`.

The reason the emulator cannot provide the same level of performance—even on high-end multicore SMP systems—is because the NVIDIA GPUs can run hundreds of simultaneous threads. The current NVIDIA Tesla 10-series GPUs and select models of the GeForce 200-series have 240 hardware thread processors, while even the highest-end AMD and Intel processors currently have only four cores per CPU.

I have seen one to two orders of magnitude increase in floating-point performance over general-purpose processors when fully utilizing my NVIDIA graphics processors—while solving real problems. Other people have published similar results. The NVIDIA site provides links to examples where researchers have reported 100x performance increases (**www.nvidia.com/cuda**). A recent paper, dated July 12, 2008, on the NVIDIA CUDA Zone Web page, reports a 270x performance increase by using GPUs for the efficient calculation of sum products.

Happily, these performance levels don't require a big cash outlay. Check out your favorite vendor. The low-end, yet still excellent performing, CUDA-enabled GPUs can be purchased for less than $120.

Why are the prices so good? The simple answer is competition. In the very competitive graphics processor market, both vendors and manufacturers work hard to deliver ever higher levels of performance at various price points to entice customers to buy or upgrade to the latest generation of graphics technology.

So, how can NVIDIA offer hundreds of thread processors while the rest of the industry can deliver only dual- and quad-core processors?

The answer is that NVIDIA designed its processors from the start for massive parallelism. Thread scalability was designed in from the very beginning. Essentially, the NVIDIA designers used a common architectural building block, called a multiprocessor, that can be replicated as many times as required to provide a large number of processing cores (or thread processors) on a GPU board for a given price point. This is, of course, ideal for graphics applications, because more thread processors translate into increased graphics performance (and a more heart-pounding customer experience). Low price-point GPUs can be designed with fewer multiprocessors (and, hence, fewer thread processors) than the higher-priced, high-end models.

CUDA was born after a key realization was made: the GPU thread processors can provide tremendous computing power if

the problem of programming tens to hundreds of them (and potentially thousands) can be solved easily.

A few years ago, pioneering programmers discovered that GPUs could be harnessed for tasks other than graphics—and they got great performance! However, their improvised programming model was clumsy, and the programmable pixel shaders on the chips (the precursors to the thread processors) weren't the ideal engines for general-purpose computing. At that time, writing software for a GPU meant programming in the language of the GPU. A friend once described this as a process similar to pulling data out of your elbow in order to get it to where you could look at it with your eyes.

NVIDIA seized upon this opportunity to create a better programming model and to improve the hardware shaders. As a result, NVIDIA ended up creating the Compute Unified Device Architecture. CUDA and hardware thread processors were born. Now, developers are able work with familiar C and C++ programming concepts while developing software for GPUs—and happily, it works very well in practice. One friend, with more than 20 years' experience programming HPC and massively parallel computers, remarked that he got more work done in one day with CUDA than he did in a year of programming the Cell broadband engine (BE) processor. CUDA also avoids the performance overhead of graphics-layer APIs by compiling your software directly to the hardware (for example, GPU assembly language), which results in excellent performance.

As a Linux developer, I especially like the CUDA framework, because I just use my favorite development tools (such as make, vi, emacs and so forth). The only real difference is that nvcc is used to compile the CUDA portions of the code instead of gcc, and the debugger and profiler are different. Optimized libraries also are available, such as math.h, FFT, BLAS and others. A big plus is that CUDA software also can be used within Python, Matlab and other higher-level languages and software.

CUDA and CUDA-enabled devices are best utilized for computational kernels. A kernel is a computationally intensive portion of a program that can be offloaded from the host system to other devices, such as a GPU, to get increased performance.

CUDA is based on a hardware abstraction that allows applications to achieve high performance while simultaneously allowing NVIDIA to adapt the GPU architecture as needed without requiring massive application rewriting. Thus, NVIDIA can incorporate new features and technology into each generation of its products as needed or when the technology becomes cost-effective. I recently upgraded to the 10-series processors and was delighted to see my applications run 2x faster.

What is actually involved in writing CUDA? A number of

excellent resources are available on the Internet to help you. The CUDA Zone forums are an excellent place for all things CUDA, and they have the benefit of letting you post questions and receive answers. I also recommend my *DDJ* columns, "CUDA: Supercomputing for the Masses" on the *Doctor Dobbs* Web site as well as the excellent tutorials at **courses.ece.uiuc.edu/ece498/al1/Syllabus.html**. Many more CUDA tutorials are listed on CUDA Zone.

Writing CUDA is greatly simplified, because CUDA automatically manages threads for the programmer. This wonderful behavior happens as a result of how concurrent threads of execution are expressed in CUDA—coupled with a hardware thread manager that runs on the CUDA-enabled device. In practice, the hardware thread manager performs amazingly well to keep the multiprocessors active and running efficiently even when confronted with non-obvious resource limitations (for example, number of thread processors, memory, memory bandwidth, latency and so on) and other issues.

Instead of creating threads explicitly (as one would do with pthreads, for instance), the developer simply needs to specify an execution configuration when calling the CUDA kernel. The execution configuration provides all the information the hardware thread scheduler needs to queue and asynchronously start CUDA kernels on one or more GPUs.

Syntactically, the call to a CUDA kernel looks like a C language subroutine call—except an execution configuration is added between triple angle brackets (<<< and >>>). The first

Writing CUDA is greatly simplified, because CUDA automatically manages threads for the programmer.

two parameters define the number of threads within a thread block and the number of thread blocks. (The key feature of a thread block is that threads within a thread block can communicate with each other, but not with threads outside the thread block). The total number of simultaneously running threads for a given kernel is the product of those two parameters. Other parameters in the execution configuration define a grid of threads (the 2-D or 3-D topology of the thread blocks) as well as some resource specifications.

Other than the addition of an execution configuration, a CUDA kernel call looks syntactically just like any other C subroutine call, including the passing of parameters and structures. There are no function calls to CUDA, because of the asynchronous nature of the kernel invocations. More information can be found in the tutorials or in "The CUDA Programming Guide" in the documentation.

By changing the execution configuration, the programmer easily can specify a few or many thousands of threads. In fact, the NVIDIA documentation recommends using a large number of threads (on the order of thousands) to future-proof your code and maintain high performance on future generations of GPU products. The hardware scheduler manages all the complexity in keeping as many threads active as is possible. What a wonderful problem not to need to worry about when programming!

From the programmer's point of view, the CUDA kernel acts as if it were contained within the scope of a loop over the total number of threads—except each loop iteration runs in a separate thread of execution. Within each thread, the programmer has all the information needed to distinguish each thread from all other threads (such as thread ID within a thread block, the number of thread blocks and coordinates within the execution configuration grid). With this information, the developer then can program each thread to perform the appropriate work and with the relevant data. Once all the threads have run to completion (which looks to the programmer like the thread reached a return statement), the CUDA kernel exits. However, control returns to the serial portion of the code running on the host computer only after all scheduled CUDA kernels also have run to completion. (Recall that CUDA kernel calls are asynchronous, so many calls can be queued before a result is required from the CUDA devices.) The programmer decides where in the host code to wait for completion.

Unfortunately, this article cannot delve too deeply into any advanced topics, such as how threads communicate amongst themselves (only within a thread block), how to manage multiple asynchronous kernels (via a streams abstraction), and transferring data to and from the GPU(s), plus many other capabilities that allow CUDA to be a fully functional platform for massively parallel computing.

Although CUDA automates thread management, it doesn't eliminate the need for developers to think about threads. Rather, it helps the programmer focus on the important issues in understanding the data and the computation that must be performed, so the problem can be decomposed into the best data layout on the CUDA device. In this way, the computational power of all those threads can be brought to bear, and those wonderful 100x performance increases over commodity processors can be achieved.

One of the most important performance challenges facing CUDA developers is how to most effectively re-use data within the local multiprocessor memory resources. CUDA-enabled GPUs have a large amount of global memory (many have gigabytes of RAM). This memory is provided so the thread processors will not have to access the system memory of the host Linux system constantly. (Such a requirement would introduce a number of performance and scalability problems.) Even though global memory on the current generation of GPUs has very high bandwidth (more than 100GB/s on the new 10-series GPUs), there is unfortunately just not enough bandwidth to keep 240 hardware thread processors running efficiently without data re-use within the multiprocessors.

The CUDA software and hardware designers have done some wonderful work to hide the performance limitations of global memory, but high performance still requires data re-use within the multiprocessors. Check out the tutorials and CUDA Zone for more information. Pay special attention to the "CUDA Occupancy Calculator", which is an indispensable Excel spreadsheet to use when designing CUDA software and understanding local multiprocessor resource capabilities and restrictions.

CUDA and CUDA-enabled devices are evolving and improving with each new generation. As developers, we really want large amounts of local multiprocessor resources, because it

makes our job much easier and software more efficient. The CUDA-enabled hardware designer, on the other hand, has to focus on the conflicting goal of delivering hardware at a low price point, and unfortunately, large amounts of local multiprocessor memory is expensive. We all agree that inexpensive CUDA hardware is wonderful, so the compromise is to market CUDA-enabled hardware with different capabilities at various price points. The market then selects the best price vs. capability trade-offs.

Letting the market decide is actually a very good solution, because GPU technology is evolving quickly. Each new generation of CUDA-enabled devices is more powerful than the previous generation and contains ever greater numbers of higher-performance components, such as local multiprocessor memory, at the same price points as the previous generation.

Is CUDA appropriate for all problems? No, but it is a valuable tool that will allow you to do things on your computer that you could not do before.

One issue to be aware of is that threads on each multiprocessor execute according to an SIMD (Single Instruction Multiple Data) model. The SIMD model is efficient and cost-effective from a hardware standpoint, but from a software standpoint, it unfortunately serializes conditional operations (for example, both branches of each conditional must be evaluated one after the other). Be aware that conditional operations can have profound effects on the runtime of your kernels. With care, this is generally a manageable problem, but it can be problematic for some issues. Advanced programmers can exploit the MIMD (Multiple Instruction Multiple Data) capability of multiple-thread processors. For more detail on Flynn's taxonomy of computer architectures, I recommend the Wikipedia article **en.wikipedia.org/wiki/Flynn%27s_taxonomy**.

I claim that the one or two orders of magnitude performance increase that GPUs provide over existing technology is a disruptive change that can alter some aspects of computing dramatically. For example, computational tasks that previously would have taken a year now can complete in a few days; hour-long computations suddenly become interactive, because they can be completed in seconds, and previously intractable real-time processing tasks now become possible.

Lucrative opportunities can present themselves to consultants and engineers with the right skill set and capabilities who are able to write highly threaded (and massively parallel) software.

What about you? Can the addition of CUDA to your programming skills benefit your career, applications or real-time processing needs?■

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.