Reed Electronics Group

MICROPROCESSOR REPORT

www.MPRonline.com

THE INSIDER'S GUIDE TO MICROPROCESSOR HARDWARE

Parallel Processing With CUDA

Nvidia's High-Performance Computing Platform Uses Massive Multithreading

By Tom R. Halfhill {01/28/08-01}

Parallel processing on multicore processors is the industry's biggest software challenge, but the real problem is there are *too many* solutions—and all require more effort than setting a compiler flag. The dream of push-button serial programming was never fully realized, so it's no surprise that push-button parallel programming is proving even more elusive.

In recent years, *Microprocessor Report* has been analyzing various approaches to parallel processing. Among other technologies, we've examined RapidMind's Multicore Development Platform (see *MPR 11/26/07-01*, "Parallel Processing for the x86"), PeakStream's math libraries for graphics processors (see *MPR 10/2/06-01*, "Number Crunching With GPUs"), Fujitsu's remote procedure calls (see *MPR 8/13/07-01*, "Fujitsu Calls Asynchronously"), Ambric's development-driven CPU architecture (see *MPR 10/10/06-01*, "Ambric's New Parallel Processor"), and Tilera's tiled mesh network (see *MPR 11/5/07-01*, "Tilera's Cores Communicate Better").

Now it is Nvidia's turn for examination. Nvidia's Compute Unified Device Architecture (CUDA) is a software platform for massively parallel high-performance computing on the company's powerful GPUs. Formally introduced in 2006, after a year-long gestation in beta, CUDA is steadily winning customers in scientific and engineering fields. At the same time, Nvidia is redesigning and repositioning its GPUs as versatile devices suitable for much more than electronic games and 3D graphics. Nvidia's Tesla brand denotes products intended for high-performance computing; the Quadro brand is for professional graphics workstations, and the GeForce brand is for Nvidia's traditional consumer graphics market.

For Nvidia, high-performance computing is both an opportunity to sell more chips and insurance against an uncertain future for discrete GPUs. Although Nvidia's GPUs and graphics cards have long been prized by gamers, the graphics market is changing. When AMD acquired ATI in 2006, Nvidia was left standing as the largest independent GPU vendor. Indeed, for all practical purposes, Nvidia is the *only* independent GPU vendor, because other competitors have fallen away over the years. Nvidia's sole-survivor status would be enviable—should the market for discrete GPUs remain stable. However, both AMD and Intel plan to integrate graphics cores in future PC processors. If these integrated processors shrink the consumer market for discrete GPUs, it could hurt Nvidia. On the other hand, many PCs (especially those sold to businesses) already integrate a graphics processor at the system level, so integrating those graphics into the CPU won't come at Nvidia's expense. And serious gamers will crave the higher performance of discrete graphics for some time to come. Nevertheless, Nvidia is wise to diversify.

Hence, CUDA. A few years ago, pioneering programmers discovered that GPUs could be reharnessed for tasks other than graphics. However, their improvised programming model was clumsy, and the programmable pixel shaders on the chips weren't the ideal engines for general-purpose computing. Nvidia has seized upon this opportunity to create a better programming model and to improve the shaders. In fact, for the high-performance computing market, Nvidia now prefers to call the shaders "stream processors" or "thread processors." It's not just marketing hype. Each thread processor in an Nvidia GeForce 8-series GPU

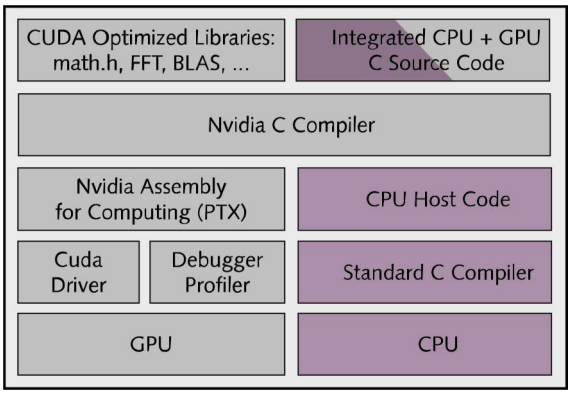

Figure 1. Nvidia's CUDA platform for parallel processing on Nvidia GPUs. Key elements are common C/C++ source code with different compiler forks for CPUs and GPUs; function libraries that simplify programming; and a hardware-abstraction mechanism that hides the details of the GPU architecture from programmers.

can manage 96 concurrent threads, and these processors have their own FPUs, registers, and shared local memory.

As Figure 1 shows, CUDA includes C/C++ software-development tools, function libraries, and a hardware-abstraction mechanism that hides the GPU hardware from developers. Although CUDA requires programmers to write special code for parallel processing, it doesn't require them to explicitly manage threads in the conventional sense, which greatly simplifies the programming model. CUDA development tools work alongside a conventional C/C++ compiler, so programmers can mix GPU code with general-purpose code for the host CPU. For now, CUDA aims at data-intensive applications that need single-precision floating-point math (mostly scientific, engineering, and high-performance computing, as well as consumer photo and video editing). Double-precision floating point is on Nvidia's roadmap for a new GPU later this year.

Hiding the Processors

In one respect, Nvidia starts with an advantage that makes other aspirants to parallel processing envious. Nvidia has always hidden the architectures of its GPUs beneath an application programming interface (API). As a result, application programmers never write directly to the metal. Instead, they call predefined graphics functions in the API.

Hardware abstraction has two benefits. First, it simplifies the high-level programming model, insulating programmers (outside Nvidia) from the complex details of the GPU hardware. Second, hardware abstraction allows Nvidia to change the GPU architecture as often and as radically as desired. As Figure 2 shows, Nvidia's latest GeForce 8 architecture has 128 thread processors, each capable of managing up to 96 concurrent threads, for a maximum of 12,288 threads. Nvidia can completely redesign this architecture in the next generation of GPUs without making the API obsolescent or breaking anyone's application software.

In contrast, most microprocessor architectures are practically carved in stone. They can be extended from one generation to the next, as Intel has done with MMX and SSE in the x86. But removing even one instruction or altering a programmer-visible register will break someone's software. The rigidity of general-purpose CPUs forced companies like RapidMind and PeakStream to create their own abstraction layers for parallel processing on Intel's x86 and IBM's Cell Broadband Engine. (PeakStream was acquired and absorbed by Google last year.) RapidMind has gone even further by porting its virtual platform to ATI and Nvidia GPUs instead of using those vendors' own platforms for high-performance computing.

In the consumer graphics market, Nvidia's virtual platform relies on install-time compilation. When PC users install application software that uses the graphics card, the installer queries the GPU's identity and automatically compiles the APIs for the target architecture. Users never see this compiler (actually, an assembler), except in the form of the installer's progress bar. Install-time translation eliminates the performance penalty suffered by run-time interpreted or just-in-time compiled platforms like Java, yet it allows API calls to execute transparently on any Nvidia GPU. As we'll explain shortly, application code written for the professional graphics and high-performance computing markets is statically compiled by developers, not by the software installer—but the principle of insulating programmers from the GPU architecture is the same.

Although Nvidia designed this technology years ago to make life easier for graphics programmers and for its own GPU architects, hardware abstraction is ideally suited for the new age of parallel processing. Nvidia is free to design processors with any number of cores, any number of threads, any number of registers, and any instruction set. C/C++ source code written today for an Nvidia GPU with 128 thread processors can run without modification on future Nvidia GPUs with additional thread processors, or on Nvidia GPUs with completely different architectures. (However, modifications may be necessary to optimize performance.)

Hardware abstraction is an advantage for software development on parallel processors. In the past, Moore's law enabled CPU architects to add more function units and pipeline stages. Now, architects are adding more processor cores. CUDA's hardware abstraction saves programmers from having to learn a new GPU architecture every couple of years.

CUDA Automatically Manages Threads

CUDA's programming model differs significantly from single-threaded CPU code and even the parallel code that some programmers began writing for GPUs before CUDA. In a single-threaded model, the CPU fetches a single instruction stream that operates serially on the data. A superscalar CPU may route the instruction stream through multiple pipelines, but there's still only one instruction stream, and the degree of instruction parallelism is severely limited by data and

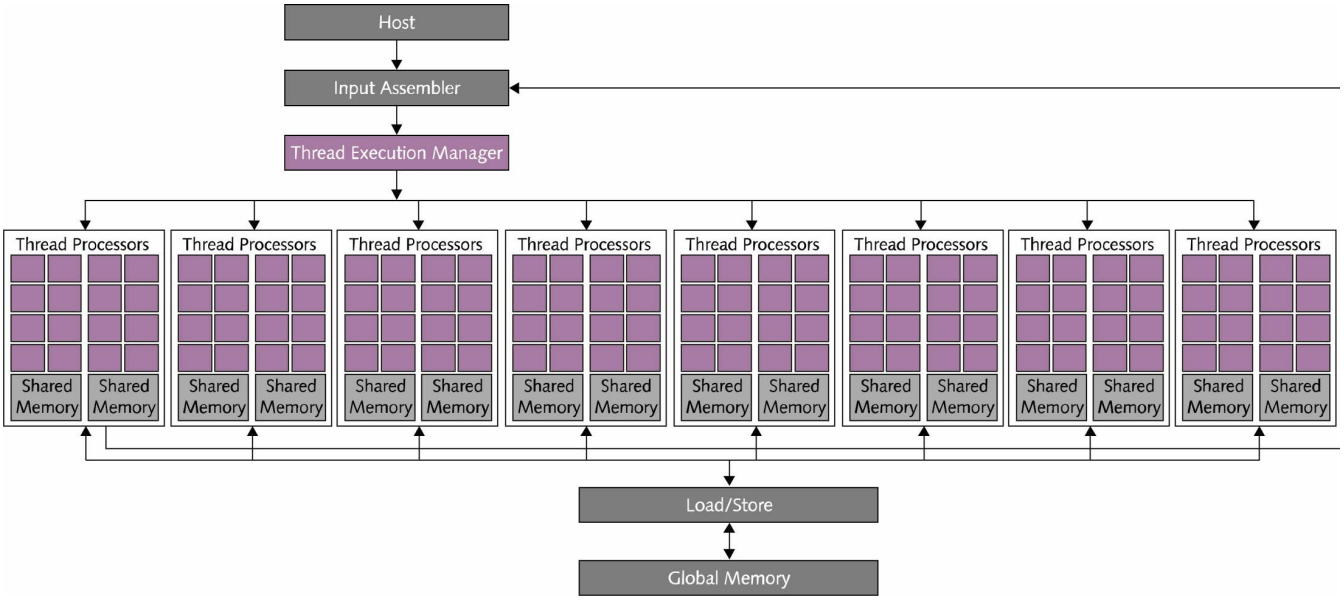

Figure 2. Nvidia GeForce 8 graphics-processor architecture. This particular GeForce 8 GPU has 128 thread processors, also known as stream processors. (Graphics programmers call them "programmable pixel shaders.") Each thread processor has a single-precision FPU and 1,024 registers, 32 bits wide. Each cluster of eight thread processors has 16KB of shared local memory supporting parallel data accesses. A hardware thread-execution manager automatically issues threads to the processors without requiring programmers to write explicitly threaded code. The maximum number of concurrent threads is 12,288. Nvidia GPUs with 128 thread processors include the GeForce 8800 GTX, GeForce 8800 Ultra, and GeForce 8800 GTS 512MB; the Quadra FX 5600; and the Tesla C870, D870, and S870.

resource dependencies. Even the best four-, five-, or six-way superscalar CPUs struggle to average 1.5 instructions per cycle, which is why superscalar designs rarely venture beyond four-way pipelining. Single-instruction multiple-data (SIMD) extensions permit many CPUs to extract some data parallelism from the code, but the practical limit is usually three or four operations per cycle.

Another programming model is general-purpose GPU (GPGPU) processing. This model is relatively new and has gained much attention in recent years. Essentially, developers hungry for high performance began using GPUs as general-purpose processors, although "general-purpose" in this context usually means data-intensive applications in scientific and engineering fields. Programmers use the GPU's pixel shaders as general-purpose single-precision FPUs. GPGPU processing is highly parallel, but it relies heavily on off-chip "video" memory to operate on large data sets. (Video memory, normally used for texture maps and so forth in graphics applications, may store any kind of data in GPGPU applications.) Different threads must interact with each other through off-chip memory. These frequent memory accesses tend to limit performance.

As Figure 3 shows, CUDA takes a third approach. Like the GPGPU model, it's highly parallel. But it divides the data set into smaller chunks stored in on-chip memory, then allows multiple thread processors to share each chunk. Storing the data locally reduces the need to access off-chip memory, thereby improving performance. Occasionally, of course, a thread does need to access off-chip memory, such as when loading the off-chip data it needs into local memory. In the CUDA model, off-chip memory accesses usually don't stall a thread processor. Instead, the stalled thread enters an inactive queue and is replaced by another thread that's ready to execute. When the stalled thread's data becomes available, the thread enters another queue that signals it's ready to go. Groups of threads take turn executing in round-robin fashion, ensuring that each thread gets execution time without delaying other threads.

An important feature of CUDA is that application programmers don't write explicitly threaded code. A hardware thread manager handles threading automatically. Automatic thread management is vital when multithreading scales to thousands of threads—as with Nvidia's GeForce 8 GPUs, which can manage as many as 12,288 concurrent threads. Although these are lightweight threads in the sense that each one operates on a small piece of data, they are fully fledged threads in the conventional sense. Each thread has its own stack, register file, program counter, and local memory. Each thread processor has 1,024 physical registers, 32 bits wide, implemented in SRAM instead of latches. The GPU preserves the state of inactive threads and restores their state when they become active again. Instructions from multiple threads can share a thread processor's instruction pipeline at the same time, and the processors can switch their attention among these threads in a single clock cycle. All this run-time thread management is transparent to the programmer.

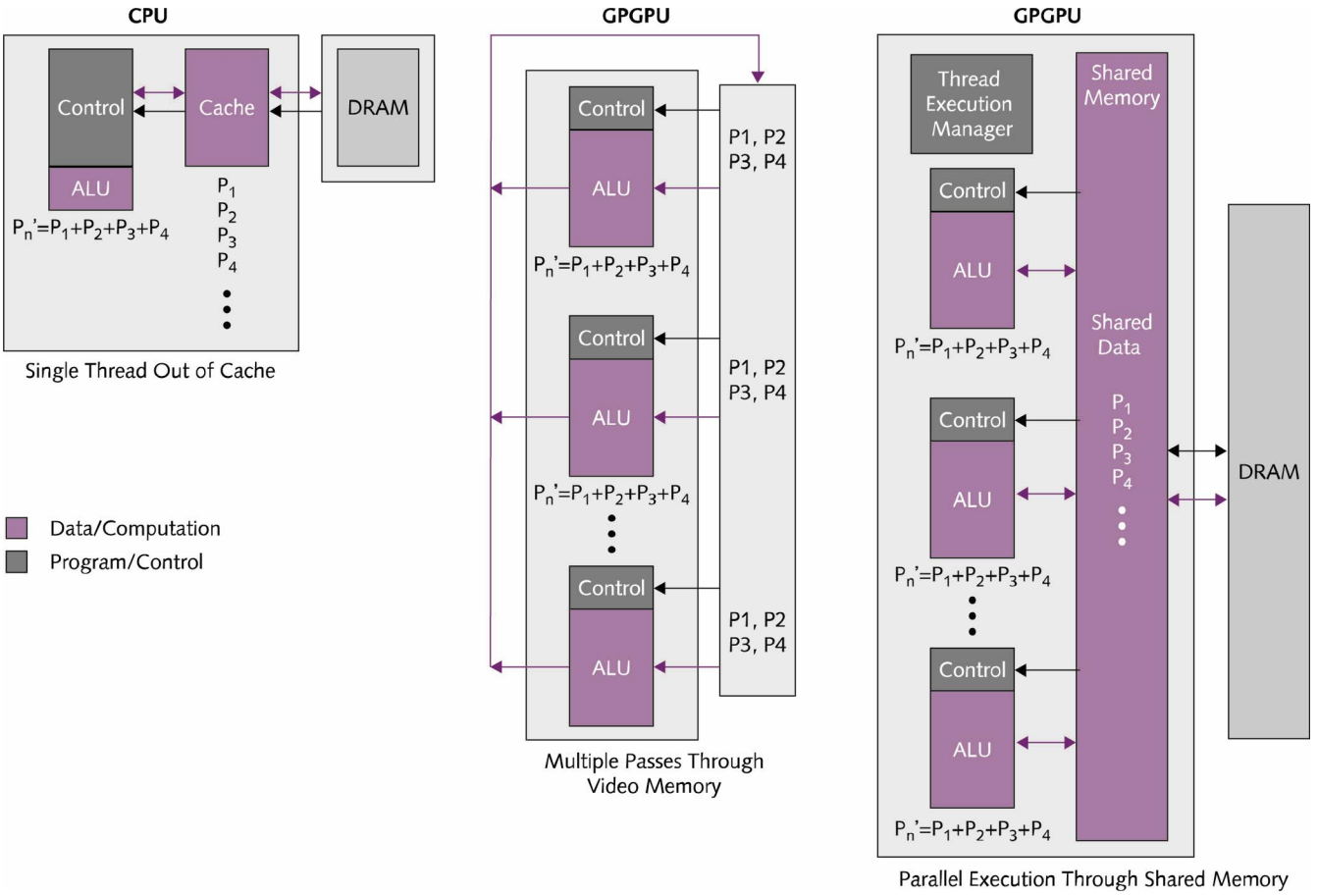

Figure 3. Three different models for high-performance computing. At left, a typical general-purpose CPU executes a single thread of execution by fetching an instruction stream from an instruction cache and operating on data flowing through a data cache from main memory. The middle section of this figure illustrates the GPGPU model, which uses a GPU's programmable pixel shaders for general-purpose processing. It's highly parallel but I/O-bound by frequent accesses to off-chip memory. At right, Nvidia's CUDA model caches smaller chunks of data on chip and shares the data among clusters of thread processors (shaders).

By removing the burden of explicitly managing threads, Nvidia simplifies the programming model and eliminates a whole category of potential bugs. In theory, CUDA makes deadlocks among threads impossible. Deadlocks or "threadlocks" happen when two or more threads block each other from manipulating the same data, creating a standoff in which none of the threads may continue. The danger of deadlocks is that they can lurk undetected in otherwise well-behaved code for years. (See *MPR 4/30/07-02*, "The Dread of Threads.")

Nvidia says CUDA eliminates deadlocks, no matter how many threads are running. A special API call, syncthreads, provides explicit barrier synchronization. Calling syncthreads at the outer level of a section of code invokes a compiler-intrinsic function that translates into a single instruction for the GPU. This barrier instruction blocks threads from operating on data that another thread is using. Although syncthreads does expose some degree of thread management to programmers, it's a relatively minor detail, and Nvidia says it's almost foolproof. A problem might arise were a program to embed a syncthreads barrier within some conditional-branch statements that aren't executed in some cases, but Nvidia strongly discourages using syncthreads in that manner. If programmers follow the rules, Nvidia insists, threadlocks won't happen with CUDA.

RapidMind makes a similar claim for its parallel-processing platform—which, like CUDA, doesn't require programmers to write explicitly multithreaded code. Absent a formal mathematical proof, only time will tell if the CUDA and RapidMind thread managers are as solid as their makers claim.

Developers Must Analyze Data

Although CUDA automates thread management, it doesn't entirely relieve developers from thinking about threads. Developers must analyze their problem to determine how best to divide the data into smaller chunks for distribution among the thread processors. This data layout or "decomposition" does require programmers to exert some manual effort and write some explicit code.

For example, consider an application that has great potential for data parallelism: scanning network packets for malware. Worms, viruses, trojans, root kits, and other malicious programs have telltale binary signatures. An antivirus filter scans the packet for binary sequences matching the bit patterns of known viruses. With a conventional single-threaded CPU, the filter must repeatedly compare elements of one array (the virus signatures) with the elements of another array (the packet). If the contents of either array are too large for the CPU's caches, the program must frequently access off-chip memory, severely impeding performance.

With CUDA, programmers can dedicate a lightweight thread to each virus signature. A block of these threads can run on a cluster of thread processors and operate on the same data (the packet) in shared memory. If much or all the data fits in local memory, the program needn't access off-chip memory. If a thread does need to access off-chip memory, the stalled thread enters the inactive queue and yields to another thread. This process continues until all threads in the block have scanned the packet. Meanwhile, other blocks of threads are simultaneously doing the same thing while running on other clusters of thread processors. An antivirus filter that has a database of thousands of virus signatures can use thousands of threads.

For developers, one challenge of CUDA is analyzing their algorithms and data to find the optimal numbers of threads and blocks that will keep the GPU fully utilized. Factors include the size of the global data set, the maximum amount of local data that blocks of threads can share, the number of thread processors in the GPU, and the sizes of the on-chip local memories. One important limit on the number of concurrent threads—besides the obvious limit of the number of thread processors—is the number of registers each thread requires. The CUDA compiler automatically determines the optimal number of registers for each thread. To reach the maximum possible number of 12,288 threads in a 128-processor GeForce 8, the compiler can't assign more than about 10 registers per thread. Usually, the compiler assigns 20 to 32 registers per thread, which limits the practical degree of concurrency to about 4,000 to 6,500 threads—still, a stupendous number.

To help developers perform this analysis, Nvidia provides an Excel spreadsheet called the Occupancy Calculator. It considers all these factors (and others) to suggest the best method of decomposing the data. Nvidia says future CUDA development tools will incorporate a more-automated calculator for data layout. In any case, programmers must be intimately familiar with their algorithms and data. Some applications have dependencies that will prevent decomposing the data to the extent described in the antivirus example.

```
Computing y _ ax + y with a serial loop:
void saxpy_serial(int n, float alpha, float *x, float *y)
{
    for(int i = 0; i<n; ++i)
        y[i] = alpha*x[i] + y[i];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);

Computing y _ ax + y in parallel using CUDA:
__global__
void saxpy_parallel(int n, float alpha, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if( i<n )  y[i] = alpha*x[i] + y[i];
}
// Invoke parallel SAXPY kernel (256 threads per block)
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

Figure 4. CUDA programming example in C. The first section of code is a single-threaded loop for a conventional CPU, whereas the second section is rewritten for multithreading on an Nvidia GPU. Both examples do the same thing—they perform a SAXPY operation on two arrays of floating-point numbers (*x* and *y*), using the scalar value *alpha*. CUDA's special extensions, highlighted in color, signal the tool chain to compile some of the code for an Nvidia GPU instead of for the host CPU.

Writing C/C++ Code for CUDA

After a developer has performed this data analysis, the next step is to express the solution in C or C++. For this purpose, CUDA adds a few special extensions and API calls to the language, as Figure 4 shows. The first code snippet in Figure 4 is a single-threaded function for a conventional CPU. The second snippet is a multithreaded implementation of the same function for CUDA. Both functions perform a so-called SAXPY operation ($y ? ax + y$) on a floating-point array. This function (or "kernel" in CUDA-speak) is part of the basic linear algebra subprograms (BLAS) included with CUDA.

One extension, __global__, is a declaration specification ("decl spec") for the C/C++ parser. It indicates that the function saxpy_parallel is a CUDA kernel that should be compiled for an Nvidia GPU, not for a host CPU, and that the kernel is globally accessible to the whole program. Another extension statement, in the last line of the CUDA example, uses three pairs of angle brackets (<<<nblocks, 256>>>) to specify the dimensions of the data grid and its blocks. The first parameter specifies the dimensions of the grid in blocks, and the second parameter specifies the dimensions of the blocks in threads. In this example, each block has 256 threads.

An additional CUDA extension for C, not used in Figure 4, is the __shared__ modifier for variables. It indicates that a variable in local memory is shared among threads in the same thread block. There are also some special API calls, such as cudaMalloc() and cudaFree(), for CUDA-specific memory allocation; and cudaMemcpy() and cuda Memcpy2D(), for copying regions of CPU memory to GPU memory.

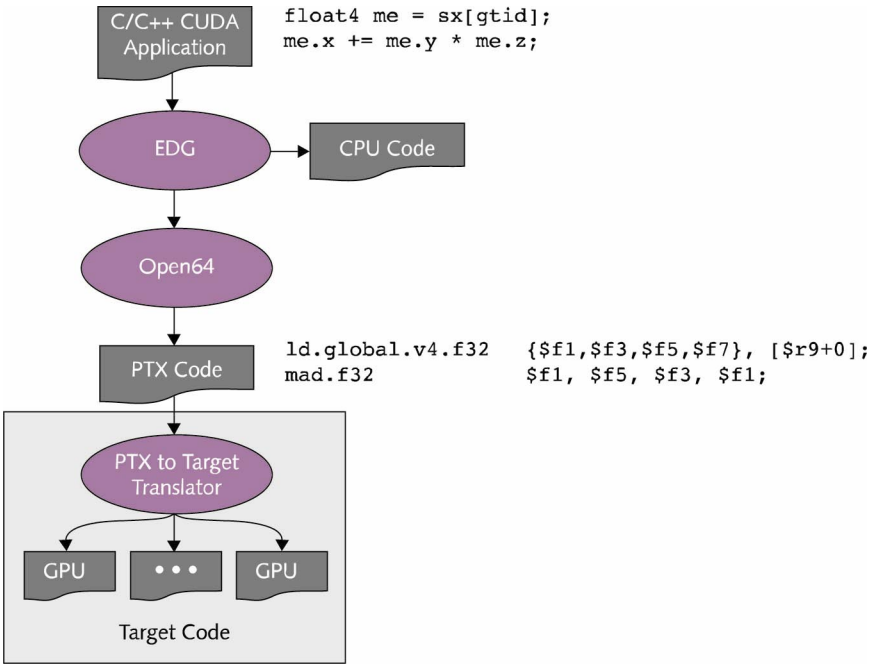

Figure 5. CUDA's compilation process. Source code written for the host CPU follows a fairly traditional path and allows developers to choose their own C/C++ compiler, but preparing the GPU's source code for execution requires additional steps. Among the unusual links in the CUDA tool chain are the EDG preprocessor, which separates the CPU and GPU source code; the Open64 compiler, originally created for Itanium; and Nvidia's PTX-to-Target Translator, which converts Open64's assembly-language output into executable code for specific Nvidia GPUs.

To obtain the best performance from a GeForce 8, Nvidia advises CUDA developers to keep the GPU occupied with about 5,000 concurrent threads. Yes, that's a frightening number. Note, however, that the code in Figure 4 is capable of doing this—without explicitly creating any threads in the conventional sense. At run time, CUDA's hardware thread manager handles the gory details of spawning threads and thread blocks, and it supervises their life cycles.

For developers, the tough work lies in analyzing the application and decomposing the data into the best layout of grids and blocks. Some re-education may be in order. Most programmers (and programming languages) tend to focus on the algorithms that operate on data. That's the normal approach for single-threaded programming. But data parallelism inverts that thinking. Understanding the nature of the data becomes as important—perhaps more important—than crafting the algorithms. Nvidia offers classes and tutorials to help retrain programmers for CUDA.

Notice that the CUDA code in Figure 4 generally adheres to standard C syntax but differs markedly from the corresponding statements in the single-threaded code. The loop in the serial code is completely absent from the parallel code. Instead of operating on the data array iteratively, CUDA performs the SAXPY operations in large blocks of threads running concurrently. These blocks, as mentioned above, represent subdivisions of the whole data set that can be locally shared by multiple thread processors. The grid is the whole data set, configured as a matrix subdivided into blocks.

Understanding the Data Is Crucial

The exact configuration of the grid and the blocks depends on the nature of the data and on the algorithms operating on the data. In the previous example of an antivirus filter, the block size might depend on the sizes of the virus signatures, while the configuration of the grid might depend on the number of signatures in the database.

Likewise, the saxpy_parallel kernel in Figure 4 walks through data arrays in steps corresponding to the sizes of the blocks and the configuration of the grid. Within each block of data, the GPU performs the necessary operations in parallel. Each individual operation may be a separate thread. There may be hundreds of threads in a block and hundreds of thread blocks, depending on the configuration of the data.

A Tool Chain With Many Links

Compiling a CUDA program is not as straightforward as running a C compiler to convert source code into executable object code. There are additional steps involved, partly because the program targets two different processor architectures (the GPU and a host CPU), and partly because of CUDA's hardware abstraction.

As the CUDA example in Figure 4 showed, the same source file mixes C/C++ code written for both the GPU and the CPU. Special extensions and declarations identify the GPU code. The first step is to separate the source code for each target architecture. For this purpose, Nvidia provides a C++ front-end preprocessor from Edison Design Group (EDG). The EDG preprocessor parses the source code and creates different files for the two architectures.

For the host CPU, EDG creates standard .cpp source files, ready for compilation with either the Microsoft or GNU C/C++ compiler. (Nvidia doesn't provide this compiler.) For Nvidia's graphics processors, EDG creates a different set of .cpp files. Strangely enough, these source files are then compiled using Open64, an open-source C/C++ compiler originally created for Intel's Itanium architecture. Nvidia says it chose Open64 because it's a good parallel-processing compiler, it's easy to customize, and it's free.

Nvidia's implementation of Open64 generates assembly-language output in a format called Parallel Thread eXecution

(PTX) files. PTX files aren't specific to a particular Nvidia GPU. In the consumer graphics market, these are the device-driver files that Nvidia's software installer assembles into GPU-specific executable code at install time. However, in the professional graphics and high-performance computing markets, PTX code usually isn't assembled during installation. Developers in those high-end markets usually prefer to statically assemble the code ahead of time, because they need to test and verify the program to stringent standards before using it for mission-critical applications.

Figure 5 illustrates the entire CUDA compilation process. Although it's certainly more convoluted than traditional source-to-binary compilation, it has several benefits. First, CUDA developers can conveniently mix CPU and GPU code in a single source file, instead of writing separate programs for each architecture. Second, the CUDA chain compiles the CPU and GPU code using different back-end tools, each optimized for its target architecture, instead of forcing a single compiler to do both jobs less well. Third, CUDA developers needn't rewrite their source code for future Nvidia GPUs—retranslating the PTX files is all that's strictly necessary.

Although retranslating the PTX files isn't strictly necessary for forward compatibility, it's desirable for maximum performance and peace of mind. True, CUDA source code is independent of the GPU architecture. Thanks to Nvidia's hardware abstraction and PTX translator, the same source code will run on any future Nvidia GPU. But to some extent, CUDA source code is bound to the GPU's microarchitecture. When developers analyze their data to determine the best layout of grids and blocks, they must consider such factors as the number of thread processors in the GPU and the local-memory configuration. A future (better) Nvidia GPU will presumably have more thread processors, more memory, and other enhancements. These factors will influence the optimal data layout.

So yes, the same source code will run just as well on a future Nvidia GPU, and performance will likely scale in step with the rising number of thread processors. But minor modifications to the source code will probably improve performance. Those modifications will also require careful CUDA developers to retest and requalify their programs. Because Nvidia introduces a new generation of GPUs approximately every 18 months, CUDA developers intent on squeezing every drop of performance from their software will probably follow a similar development schedule.

The Pros and Cons of CUDA

As we alluded at the beginning of this report, the real problem with parallel processing is too many solutions. Finding the best one for a particular application isn't easy. Choosing a processor architecture is only the first step and may be less important than the supplier's product roadmap, the software-development tools, the amount of special programming required, the degree of forward code compatibility, and the anticipated longevity of the vendors.

Price & Availability

Nvidia's Compute Unified Device Architecture (CUDA) is available now for software development with Nvidia graphics processors. CUDA works with all GPUs in Nvidia's Tesla high-performance computing line; with the latest 4600 and 5600 GPUs in Nvidia's Quadro professional-graphics line; and with GeForce-8 GPUs in Nvidia's consumer graphics line. Nvidia says CUDA will be compatible with all future Quadro and GeForce variants. The latest CUDA Software Development Kit (SDK) is version 1.1, which runs on Microsoft Windows XP (x86 and x86-64) and Linux (Fedora 7, RedHat Enterprise Linux 3.x–5.x, SUSE Linux Enterprise Desktop 10-SP1, OpenSUSE 10.1/10.2, and Ubuntu 7.04, all on x86 or x86-64). Developers can download the CUDA SDK from Nvidia's website. Licensing is free.

General information about CUDA:

- *www.nvidia.com/object/cuda_home.html*

List of Nvidia GPUs compatible with CUDA:

- *www.nvidia.com/object/cuda_learn_products.html*

CUDA sample source code:

- *www.nvidia.com/object/cuda_get_samples.html*

Download the CUDA SDK:

- *www.nvidia.com/object/cuda_get.html*

Specifications of Nvidia GeForce 8800 GPUs:

- *www.nvidia.com/page/geforce_8800.html*

Nvidia has obvious business reasons for promoting GPUs in the field of high-performance computing. For Nvidia's prospective customers in this field, GPUs have several attractions. Parallelism isn't new in GPUs, so Nvidia's engineers are experienced with massively parallel processing and are rapidly applying their knowledge to new designs. The PC graphics market largely subsidizes the development and production of these GPUs, so the chips evolve at a fast rate and are relatively inexpensive and plentiful. (Nvidia has already sold 50 million CUDA-capable GPUs.) Nvidia is a big, established company, so it may be a safer bet than a young startup struggling to introduce a new processor architecture and programming model. High-performance computing on GPUs has attracted an enthusiastic following in the academic community as well as within the industry, so there is growing expertise among programmers and alternative approaches to software development.

Downsides are few. GPUs only recently became fully programmable devices—the shaders used to be hardwired—so their programming interfaces and tools are somewhat immature. Single-precision floating point is sufficient for consumer graphics, so GPUs don't yet support double precision. The primary market for GPUs is electronic gaming, so Nvidia must optimize the devices for that application, sometimes at the expense of auxiliary markets. GPUs

are large, power-hungry chips, so cramming many of them into a small space requires special attention to cooling and energy.

When evaluating all these factors, keep in mind that CUDA isn't the only option for software development on Nvidia GPUs. CUDA is merely Nvidia's platform. In some ways, RapidMind's Multicore Development Platform is superior. RapidMind supports multiple processor architectures, including Nvidia's GPUs, ATI's GPUs, IBM's Cell BE, and Intel's and AMD's x86. This flexibility lets developers target the architecture currently delivering the best performance—without rewriting or even recompiling the source code. RapidMind's C++ extensions appear less dependent on the processor's microarchitecture than CUDA's extensions are. However, without independent benchmarks, we can't judge whether RapidMind or CUDA delivers better performance on Nvidia GPUs. Also, RapidMind doesn't publicly disclose the price of its platform, whereas CUDA is a free download from the Nvidia website.

Because of the steep learning curve involved, and the new code that must be written, developers will probably resist making a switch after adopting a particular development platform for their high-performance computing application. Compare the example source code in this article with the example code in our recent RapidMind article—there's nothing in common, even though both platforms essentially do the same thing.

For that reason, some developers would rather wait for a solution to parallel processing that isn't tied to one vendor's proprietary technology. They prefer an architecture-independent, industry-standard solution widely supported by tool vendors. (Indeed, some developers perceive a need for an entirely new programming language.) AMD, IBM, Intel, Microsoft, and other companies are working in the direction of standard parallel-processing extensions to C/C++, but it's far from certain that their efforts will converge anytime soon. Therefore, developers needing high performance right now must choose a nonstandard approach. CUDA is a strong contender, but it's only one option among many. ◇