Reed Electronics Group

MICROPROCESSOR REPORT

www.MPRonline.com

THE INSIDER'S GUIDE TO MICROPROCESSOR HARDWARE

AMD's Stream Becomes a River

Parallel-Processing Platform for ATI GPUs Reaches More Systems

By Tom R. Halfhill {12/22/08-01}

In December, AMD started bundling the runtime package for its ATI Stream parallel-processing platform with the latest display driver for ATI graphics processors. As users download this driver, the installed base of Stream-capable systems could swell to more than two million PCs. Before, users had to download and install the free ATI Stream runtime separately.

By making ATI Stream an integral part of its drivers, AMD is strengthening its competitive position against Nvidia, the only other major vendor of PC graphics cards. Nvidia added the runtime for its parallel-processing platform to display drivers in late 2007.

Nvidia's platform—formerly called the Compute Unified Device Architecture, now known simply as CUDA—is the leader in this fast-growing field. Nvidia claims an installed base of 107 million CUDA-capable GPUs, although the actual number of systems running CUDA software is much lower. CUDA has attracted numerous developers (Nvidia claims 25,000) and great interest among programmers who are still kicking the tires. AMD's decision to bundle ATI Stream into its latest driver offers developers another good option.

At the same time, new application software is appearing for both platforms. With relatively little fanfare, massively threaded parallel processing is going mainstream on millions of desktops, servers, and notebooks with discrete graphics. And it's happening first on GPUs, not on the much-heralded multicore CPUs from AMD and Intel.

Moreover, it's coming at a time when many developers still wring their hands over a perceived lack of programming tools for multicore processors. But, as *Microprocessor Report* noted in an editorial last July, plenty of tools are available. The real problem is a lack of industry standards. Although ATI Stream and CUDA are technically similar, they require programmers to use different variations of C, and their tool chains and compiled executables are incompatible with each other's GPUs. What programmers really seem to want is fewer choices that are standardized, not more choices that are proprietary. (See *MPR 7/28/08-02*, "Tools for Multicore Processors.")

OpenCL: Path to Nirvana?

Help is on the way. Both AMD and Nvidia support a new platform called OpenCL (Open Computing Language). As the name implies, OpenCL is an open specification, and it's intended for parallel processing on systems with heterogeneous microprocessor architectures. (Other backers include Apple, ARM, Broadcom, Ericsson, Freescale, IBM, Intel, Imagination Technologies, Motorola, Nokia, RapidMind, Samsung, and Texas Instruments.) The OpenCL technical specification was completed in November and publicly released on December 8.

For programmers working with multicore processors—especially in systems with multicore CPUs and GPUs having completely different architectures—OpenCL could be the long-sought path to nirvana. It defines a common C-language interface for writing data-parallel code. Like ATI Stream and CUDA, it can also exploit higher-level task parallelism. (See the sidebar, "OpenCL Tries to Standardize Parallel Programming.")

But until the OpenCL specification is widely adopted and supported by development tools, programmers still have parallel code to write. Some markets are too competitive for

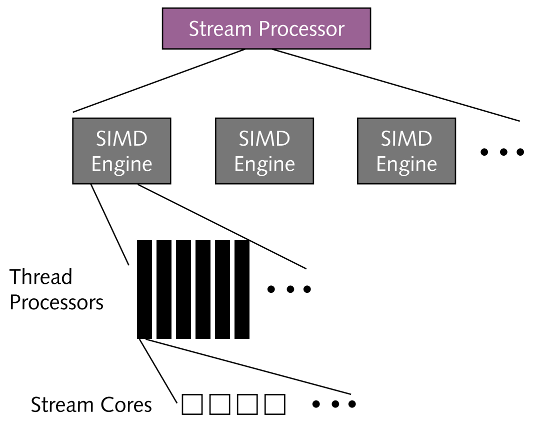


Figure 1. AMD's nomenclature for an ATI GPU varies according to the target application. In the context of general-purpose computing, the GPU is a "stream processor," which is composed of multiple SIMD engines, thread processors, and stream cores. The numbers of these components may vary from one generation of GPU to another. The latest ATI RV770 has 10 SIMD engines, each with 16 thread processors. Each thread processor has five general-purpose stream cores, which are ALU/FPU function units. For graphics processing, stream cores are pixel shaders.

developers to pin all their hopes on a future solution with uncertain prospects. In addition, some programmers may prefer today's solutions, proprietary though they may be. ATI Stream and CUDA work at a somewhat higher level of abstraction than OpenCL does, so they retain some advantages.

Over the past two years, *MPR* has published in-depth articles on Nvidia's CUDA, the RapidMind Multicore Development Platform, and the PeakStream Platform. All are software-development platforms for parallel processing. (PeakStream's products went off the market after Google acquired the company in 2007.) This article will analyze ATI Stream as it exists today, before the advent of OpenCL. To review our previous coverage, see *MPR 1/28/08-01*, "Parallel Processing With CUDA," *MPR 11/26/07-01*, "Parallel Processing for the x86," and *MPR 10/2/06-01*, "Number Crunching With GPUs."

Parallelism for the Masses

Some critics still question whether PC applications have enough inherent parallelism worth exploiting. By now, almost everyone is familiar with the "embarrassingly parallel" opportunities in high-performance computing (HPC) applications, such as financial analysis, energy exploration, pharmaceutical development, and data mining. On the desktop, the only similar tasks seem to be 3D graphics, for which GPUs were invented. But those doubts are evaporating as programmers schooled in sequential coding learn to refactor their algorithms for parallel execution. They are unlocking more parallelism than was first suspected.

To prove the point, AMD offers free parallel-processing software for its latest ATI Catalyst 8.12 display driver. The ATI Avivo Video Converter transcodes standard-definition (SD) or high-definition (HD) digital video among several different formats. Transcoding is brutally slow on a conventional CPU, even one with multiple cores. Avivo, built on the ATI Stream platform, divides the work into hundreds of threads that run concurrently on ATI GPUs, such as Radeon HD 4850 graphics cards. Avivo supports MPEG-1, MPEG-2, MPEG-4/DivX, Windows Media Video (WMV), H.264/AVC, and other formats. It allows users to convert video from camcorders, DVDs, and websites for playback on portable devices.

AMD says Avivo can transcode an hour's worth of video from MPEG-2 1080p HD format to H.264 320 × 240 format (suitable for an Apple Video iPod) in only 12 minutes. The same job takes 3 hours 23 minutes on an Intel Core 2 Duo processor running at 3.0GHz. An enhanced commercial version of Avivo, CyberLink's PowerDirector 7, is scheduled for release in 1Q09. A similar product, called Badaboom, is available from Elemental Technologies for Nvidia's CUDA, except that Badaboom is intended for higher-quality transcoding, especially between high-resolution formats.

More parallel-processing software for GPUs is coming—or is already here. ArcSoft's TotalMedia Theater, scheduled for release in 1Q09, uses ATI Stream to upscale SD video to 1080p HD on PCs. Adobe is accelerating Photoshop Creative Suite 4, After Effects, Flash, and Acrobat Reader. Windows Vista uses GPU acceleration in Picture Viewer, and Microsoft Expression uses it to encode audio and video content for Silverlight, WMV, and WMA. Microsoft PowerPoint 2007 already has an option for GPU acceleration. Additional desktop applications with great potential for parallelism include audio/video editors, video games (especially physics and artificial-intelligence algorithms), and local search engines.

Critics object that many of the aforementioned products use the GPU only, or primarily, for graphics processing, not for general-purpose computing. Although it's "parallel processing," they say, it's no more innovative than the usual graphics processing in games. Their point is a fine one. From our perspective, the bottom line is that a growing number of tasks once executed slowly in sequential code on CPUs are now executing rapidly in parallel code on GPUs. That many of those tasks are graphics oriented is natural. GPUs, first and foremost, are graphics processors. But there's definitely a trend toward delegating other tasks, such as video transcoding, to GPUs. We expect to see many more. (Nvidia says that more than 160 nongraphics applications are running on CUDA.) Without them, a GPU that isn't running a game is little more than a case heater.

Today, parallel processing is no longer a hammer in search of a nail. The proliferation of massively threaded GPUs with general-purpose compute engines is driving a revolution in software development. It's similar to the transition from assembly language to high-level compiled languages in the 1980s—except this time, the big winners are users, not programmers. For possibly the first time in the history of computing, software (independently of hardware) is getting faster, not slower.

The Evolution of ATI Stream

The ATI Stream parallel-processing platform has followed roughly the same evolutionary path as Nvidia's CUDA, although CUDA has evolved a little faster. For both, the most crucial step came three years ago. That's when graphics processors began moving beyond the efficient but rigid fixed-function graphics hardware of earlier GPUs toward logic that is more general purpose and programmable. The dedicated pixel shaders of the past have become flexible compute engines with rudimentary function units and caches. This new "unified shader model" is the opposable thumb of GPUs—it allows them to perform a greater variety of tasks.

Although GPUs remain highly optimized for 3D graphics, they become "stream processors" in their general-purpose (GPGPU) guise. AMD's latest GPU chip is the ATI RV770, which is found on ATI Radeon, FireStream, and FirePro graphics cards. As Figure 1 shows, an AMD stream processor contains several SIMD engines. (The RV770 has 10.) Each SIMD engine has multiple "thread processors." (The RV770 has 16 of them per SIMD engine.) Inside each thread processor is a cluster of "stream cores," which are like function units in CPUs. (The RV770 has five general-purpose stream cores per thread processor, for a total of 800 stream cores on chip.)

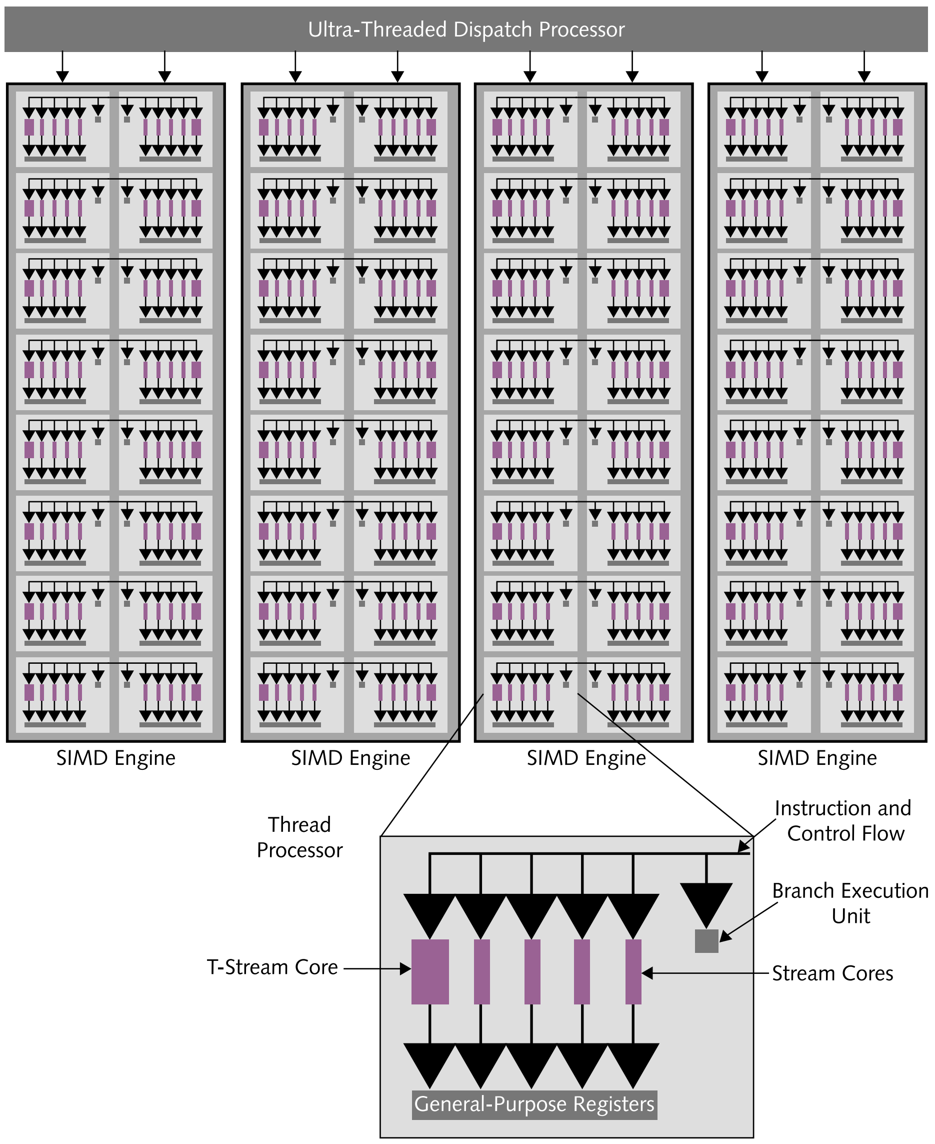


Figure 2. Each SIMD engine in an ATI GPU/stream processor contains multiple thread processors, and each thread processor contains multiple stream cores. Not all stream cores are equal. In the ATI RV770, the "T-stream core" is a beefier function unit, capable of performing transcendental operations as well as 32-bit integer and 32-bit floating-point operations. The other four general-purpose stream cores—known as x, y, z, and w—perform basic 32-bit integer and floating-point operations. A sixth stream core is dedicated to branch instructions. Each thread processor has a local register file shared by the stream cores.

Admittedly, the new terminology is a little confusing. It helps to visualize thread processors as five-way VLIW (very long instruction word) engines. In VLIW computing, a special compiler bundles multiple assembly-level instructions to maximize utilization of the processor's function units. If a five-way VLIW processor has four ALUs and one FPU, the compiler tries to bundle four integer instructions with a floating-point instruction, so that all five operations in the instruction word can issue in parallel. The same principle applies to an ATI thread processor.

As Figure 2 shows, the five general-purpose stream cores in a thread processor have slightly different capabilities. Four stream cores (known as the x, y, z, and w cores) are simpler function units that can perform 32-bit integer and 32-bit (single-precision) floating-point operations. The fifth general-purpose stream core, known as the "T-stream core," is a little more sophisticated—it can also perform transcendental operations.

Until recently, GPUs from AMD and Nvidia were limited to single-precision floating-point math. It was good enough

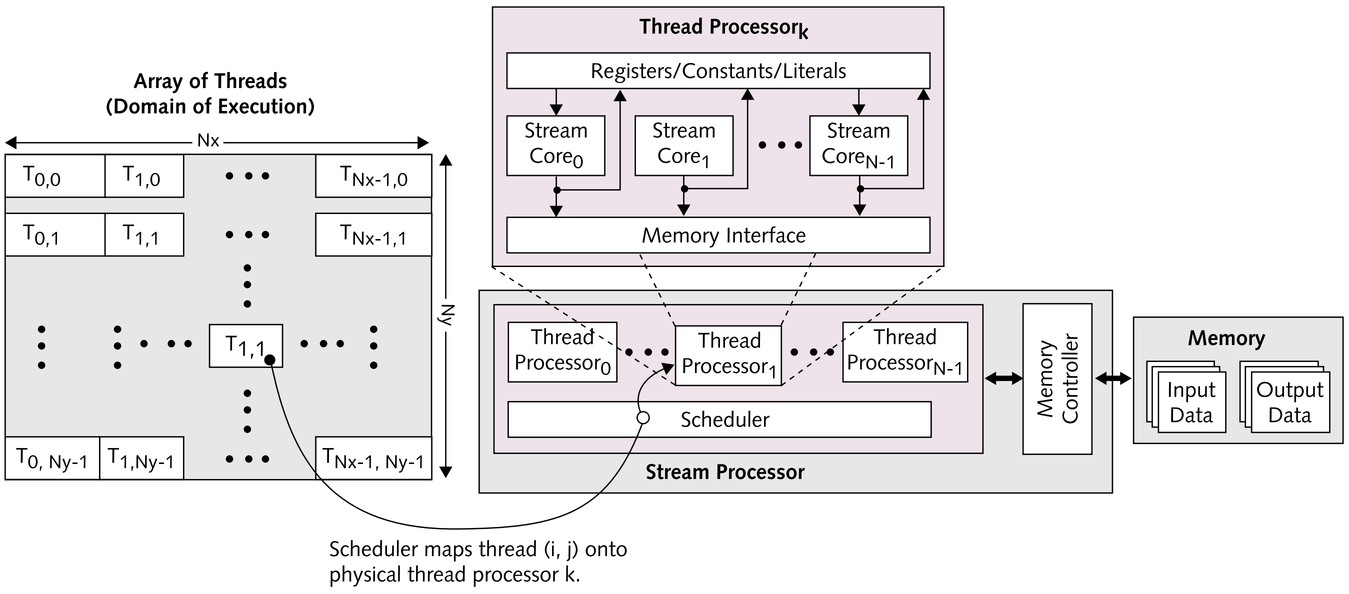


Figure 3. ATI Stream programming model. The square at left represents an array of threads operating on chunks of data in an application program. A scheduler in the stream processor (GPU) maps the threads onto the thread processors diagrammed in Figure 2. The number of threads that can execute in parallel depends on the resources of the processor. The latest ATI RV770 chip has 800 stream cores in its 160 thread processors, so it can execute a maximum of 800 threads in parallel. Additional threads can run concurrently but must be time-sliced.

for 3D graphics, but it impaired their usefulness in HPC applications that demand greater precision. The latest AMD and Nvidia GPUs can handle double-precision (64-bit) floating point, albeit at a slower rate than single precision.

In the case of the ATI RV770, each stream core in a thread processor can execute one single-precision multiply-add (MADD) instruction per clock cycle. Each MADD is two operations, so the peak theoretical throughput is 1.2 teraflops at the maximum clock frequency of 750MHz. (That's 800 stream cores × 2 operations per cycle × 750MHz = 1.2 trillion operations per second.) To execute a double-precision MADD, all four of the simpler stream cores in a thread processor must gang together. (The T-stream core is idle.) Therefore, the peak double-precision throughput is one-fifth the rate of peak single-precision throughput—240 gigaflops vs. 1.2 teraflops, at 750MHz.

ATI Stream Programming Model

The previous overview of the ATI GPU architecture makes the ATI Stream programming model easier to grasp. Although ATI Stream tries to shield programmers from low-level architectural details, some architectural knowledge definitely helps, even when writing high-level code. A common mistake, says Michael Chu, AMD's product manager for ATI Stream software, is to write programs that underutilize the vast threading resources of the processor. The result is disappointment when the threadbare program doesn't run much faster than a single-threaded version.

For programmers, the first step is to analyze the application, breaking down the data into chunks that can execute in parallel as independent threads. Next, programmers write their source code in a special version of C, with extensions for expressing data-level parallelism. Finally, programmers compile the code. Figure 3 illustrates the ATI Stream programming model.

Currently, ATI Stream uses an AMD-enhanced version of the Brook C compiler. Brook was developed at Stanford University for a parallel-computing project launched in the 1990s. AMD's version is called Brook+. Because it's based on the open-source Brook compiler, Brook+ is open source, too. The main difference is the back-end code generator, which AMD has modified for ATI Stream. (More options are coming. In November, the Portland Group—a subsidiary of STMicroelectronics—announced it will adapt its Fortran and C compilers to AMD's x86 and ATI processors.)

Compilation for ATI Stream differs from traditional compilation in two ways. First, Brook+ targets processors with two completely different architectures: the x86-compatible host CPU and the ATI GPU. For the host, Brook+ translates its variation of ANSI C into a standard C++ source file, which must be further compiled into an x86 binary file by another compiler. (Any C++ compiler works.)

For the GPU, Brook+ creates a special intermediate-language (IL) file that resembles high-level assembly language. AMD's IL isn't specific to any particular hardware implementation. It's a generic "assembly language" that can run on many different ATI GPUs. At runtime, a hardware-abstraction layer known as the AMD Compute Abstraction Layer (CAL) translates the IL into executable code for the GPU. Figure 4 is a diagram of the ATI Stream tool chain and runtime module.

Lower-Level Programming Is Possible

CAL permits AMD to change the architectures of ATI GPUs without breaking application software. Thanks to this abstraction layer, source code written in Brook+ can run without modification on different ATI GPUs—though it may not run with optimum performance.

Earlier, we compared the thread processors of an ATI RV770 with a five-way VLIW machine. A classic drawback of VLIW is that developers usually must recompile their software for different hardware implementations that have different complements of function units. In our previous VLIW example, the five-way VLIW machine with four ALUs and one FPU is ideal for executing instruction-word bundles having four integer operations and one floating-point operation. If a later generation of the VLIW processor has three ALUs and two FPUs, the previously compiled program either won't run as efficiently or won't run at all.

To some extent, the same principle applies to stream processing on an ATI GPU. Although the abstraction layer in CAL allows ATI Stream programs to run on different ATI GPUs without recompilation, developers may find recompilation desirable to take advantage of the expanded resources in future processors.

When ATI Stream made its debut in 2006, the platform was based on a lower-level abstraction layer known as Close to Metal (CTM). Although CAL has replaced CTM in the latest generation of ATI Stream, adventurous developers can still program the GPU at a lower level by using IL instead of Brook+. AMD doesn't encourage the practice, because IL resembles assembly language, is less portable than Brook+ C, and requires a thorough understanding of the GPU's architecture. As Figure 5 shows, the ATI Stream tool chain includes a utility called the GPU ShaderAnalyzer that allows programmers to view and modify IL code.

Threading Is Partly Automated

Brook+ isn't the long-sought magic compiler that automatically extracts massive parallelism from sequential source code. Programmers must explicitly identify the data-level parallelism when writing their code. Therefore, existing sequential code needs some refactoring for ATI Stream.

However, Brook+ doesn't require programmers to write code that explicitly spawns threads and manages their life cycles in the same way that conventional threading in C, C++, or Java does. Brook+ and CAL handle that drudge work. The CAL runtime package also tries to discover some parallelism that programmers miss. Even so, as we'll explain shortly, programmers shoulder some responsibility for threading in Brook+.

In addition to spawning and managing threads, CAL and the GPU's thread dispatcher automatically map the threads to the arrays of stream cores. This job is very important, because it provides some abstraction between the hardware and software.

For example, the ATI RV770 can execute a maximum of 800 parallel threads. (10 SIMD engines × 16 thread processors × 5 stream cores = 800.) But programmers needn't divide their data into exactly 800 chunks. If there are fewer chunks, some stream cores will be idle. If there are more chunks, the processor's thread scheduler automatically time-slices the threads. This degree of abstraction allows programmers to write code that will spawn virtually any number of threads without worrying whether the program will run on a particular processor.

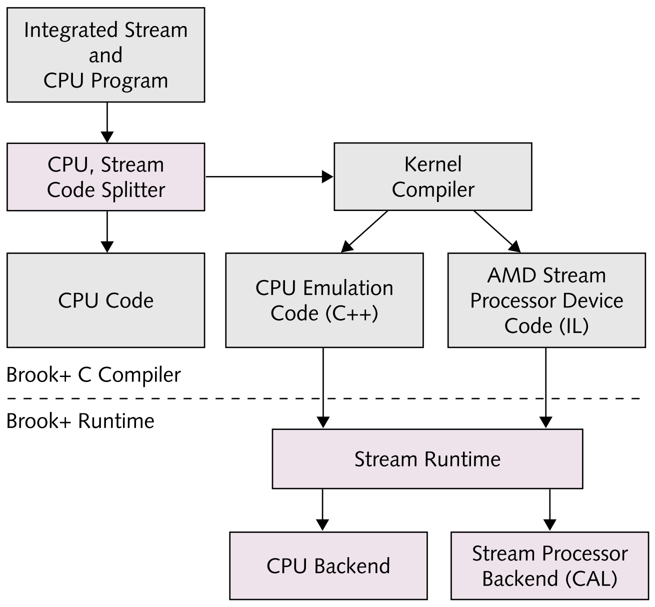
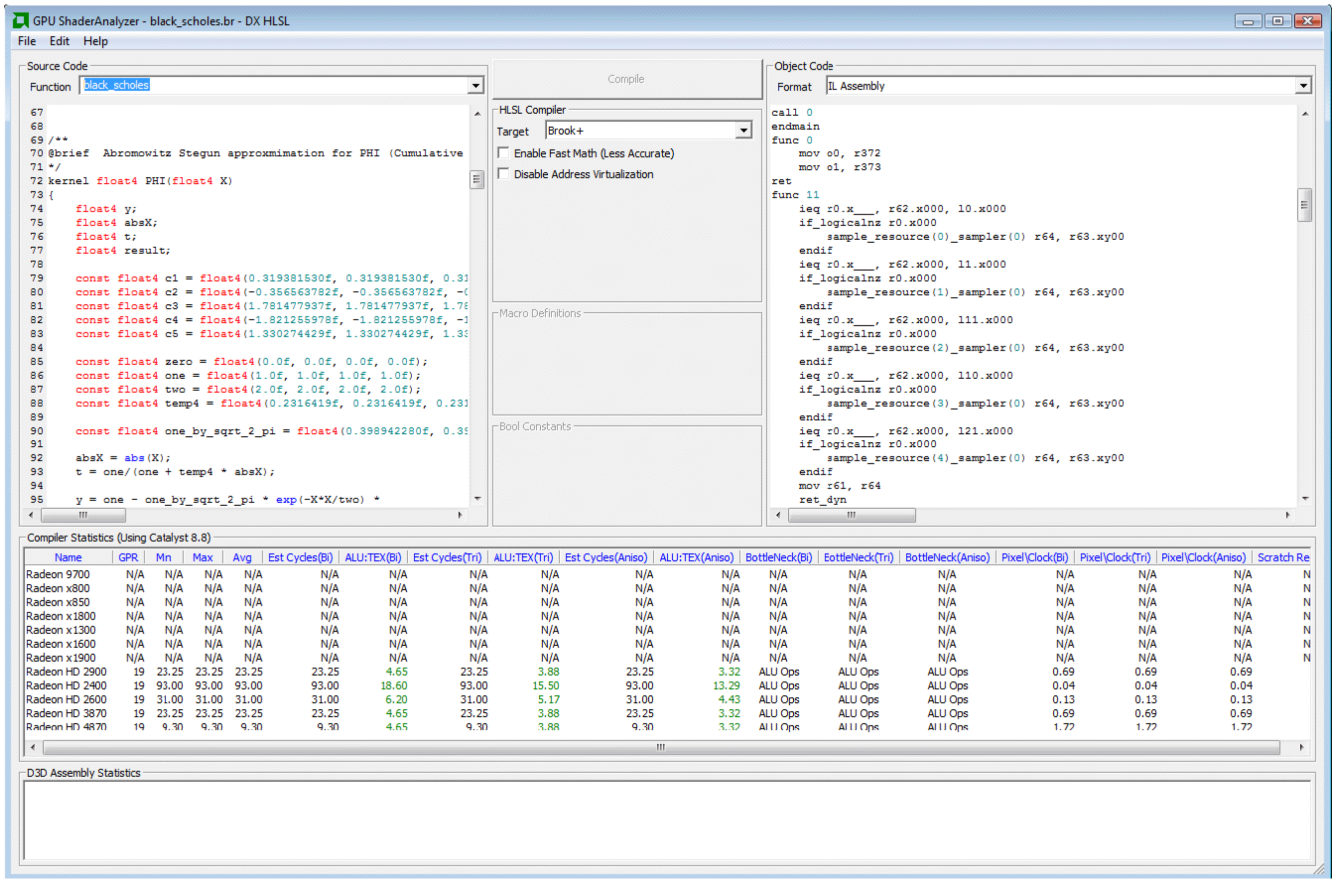

Integrated Stream and CPU Program
CPU, Stream Code Splitter
Kernel Compiler
CPU Code
CPU Emulation Code (C++)
AMD Stream Processor Device Code (IL)
Brook+ C Compiler
Brook+ Runtime
Stream Runtime
CPU Backend
Stream Processor Backend (CAL)

Figure 4. Source code written in ATI Stream's flavor of C splits into two forks: code destined for the x86 CPU and code destined for the ATI GPU. For the x86 host processor, AMD's Brook+ compiler creates generic C++ code that any standard C++ compiler can handle. In the other fork, the Brook+ "kernel compiler" generates special intermediate-language (IL) code. At runtime, AMD's Compute Abstraction Layer (CAL) translates IL code into executable code for the GPU. In December, AMD began bundling CAL with the latest display drivers for ATI graphics cards, allowing at least two million PCs to run ATI Stream software.

Usually, it's better to create too many threads than too few threads. If there aren't enough threads to occupy all the stream cores, the program isn't making the most of the GPU's resources. Time-slicing is generally preferable to underutilization. As future generations of processors add more stream cores, the thread scheduler does less time-slicing. Eventually, a future processor may have enough stream cores to completely eliminate the need for time-slicing, allowing all threads to execute simultaneously.

Programmers Learn to 'Think Parallel'

The biggest challenge for programmers is analyzing the application and dividing the data into thread-friendly chunks. This is where programmers accustomed to sequential processing often stumble. Their inclination is to write loops that walk through large datasets, manipulating the data iteratively. But a loop represents a single thread of execution that ignores any parallelism inherent in the data.



Source Code
Function: black_scholes

```
/**
@brief  Abromowitz Stegun approxmimation for PHI (Cumulative
*/
kernel float4 PHI(float4 X)
{
    float4 y;
    float4 absX;
    float4 t;
    float4 result;

    const float4 c1 = float4(0.319381530f, 0.319381530f, 0.31
    const float4 c2 = float4(-0.356563782f, -0.356563782f, -(
    const float4 c3 = float4(1.781477937f, 1.781477937f, 1.78
    const float4 c4 = float4(-1.821255978f, -1.821255978f, -1
    const float4 c5 = float4(1.330274429f, 1.330274429f, 1.33

    const float4 zero = float4(0.0f, 0.0f, 0.0f, 0.0f);
    const float4 one = float4(1.0f, 1.0f, 1.0f, 1.0f);
    const float4 two = float4(2.0f, 2.0f, 2.0f, 2.0f);
    const float4 temp4 = float4(0.2316419f, 0.2316419f, 0.231

    const float4 one_by_sqrt_2_pi = float4(0.398942280f, 0.39

    absX = abs(X);
    t = one/(one + temp4 * absX);

    y = one - one_by_sqrt_2_pi * exp(-X*X/two) *
```

HLSL Compiler
Target: Brook+
Enable Fast Math (Less Accurate)
Disable Address Virtualization

Object Code
Format: IL Assembly

```
call 0
endmain
func 0
    mov o0, r372
    mov o1, r373
ret
func 11
    ieq r0.x___, r62.x000, l0.x000
    if_logicalnz r0.x000
        sample_resource(0)_sampler(0) r64, r63.xy00
    endif
    ieq r0.x___, r62.x000, l1.x000
    if_logicalnz r0.x000
        sample_resource(1)_sampler(0) r64, r63.xy00
    endif
    ieq r0.x___, r62.x000, l11.x000
    if_logicalnz r0.x000
        sample_resource(2)_sampler(0) r64, r63.xy00
    endif
    ieq r0.x___, r62.x000, l10.x000
    if_logicalnz r0.x000
        sample_resource(3)_sampler(0) r64, r63.xy00
    endif
    ieq r0.x___, r62.x000, l21.x000
    if_logicalnz r0.x000
        sample_resource(4)_sampler(0) r64, r63.xy00
    endif
    mov r61, r64
    ret_dyn
```

Compiler Statistics (Using Catalyst 8.8)

Name	GPR	Mn	Max	Avg	Est Cycles(Bi)	ALU:TEX(Bi)	Est Cycles(Tri)	ALU:TEX(Tri)	Est Cycles(Aniso)	ALU:TEX(Aniso)	BottleNeck(Bi)	EottleNeck(Tri)	BottleNeck(Aniso)	Pixel\Clock(Bi)	Pixel\Clock(Tri)	Pixel\Clock(Aniso)	Scratch Re
Radeon 9700	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N
Radeon x800	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N
Radeon x850	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N
Radeon x1800	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N
Radeon x1300	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N
Radeon x1600	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N
Radeon x1900	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N
Radeon HD 2900	19	23.25	23.25	23.25	23.25	4.65	23.25	3.88	23.25	3.32	ALU Ops	ALU Ops	ALU Ops	0.69	0.69	0.69	
Radeon HD 2400	19	93.00	93.00	93.00	93.00	18.60	93.00	15.50	93.00	13.29	ALU Ops	ALU Ops	ALU Ops	0.04	0.04	0.04	
Radeon HD 2600	19	31.00	31.00	31.00	31.00	6.20	31.00	5.17	31.00	4.43	ALU Ops	ALU Ops	ALU Ops	0.13	0.13	0.13	
Radeon HD 3870	19	23.25	23.25	23.25	23.25	4.65	23.25	3.88	23.25	3.32	ALU Ops	ALU Ops	ALU Ops	0.69	0.69	0.69	
Radeon HD 4870	19	9.30	9.30	9.30	9.30	4.65	9.30	3.88	9.30	3.32	ALU Ops	ALU Ops	ALU Ops	1.72	1.72	1.72	

Figure 5. The GPU ShaderAnalyzer utility for ATI Stream. This tool allows programmers to view the special intermediate-language (IL) code generated by the Brook+ compiler. The Brook+ C source code is visible in the left pane, and the corresponding IL code is visible at right. The lower pane displays performance statistics for this code. Programmers can tweak the IL code for better performance, although working at this level is like programming in assembly language and demands thorough knowledge of the complex GPU architecture.

Data-level parallelism requires a different approach. The concept is to divide a large dataset into subsets having no mutual dependencies—operations that must wait for an operation in another subset to finish. At first, the opportunities for parallelism appear elusive. After a few exercises, however, programmers learn to think about problems in a whole new way. Even some applications that seem impervious to parallelism can yield surprising results.

In a recent *MPR* editorial, we described a book-indexing program written in Microsoft QuickBASIC in the 1980s. Although this program was designed to run as a single "thread" of execution on a single computer, it soon became obvious that it was capable of massively parallel processing—albeit in a crude fashion. If the book index contained 400 entries, the indexing program could run on as many as 400 computers simultaneously. Because there were no dependencies among the index words, each instance of the program could "read" the book's text from start to finish, looking for occurrences of the single index entry assigned to it. Later, a human book editor aggregated the results. This example shows that even a seemingly simple program can sometimes hide a surprising amount of parallelism. (See *MPR 3/31/08-01*, "Think Parallel.")

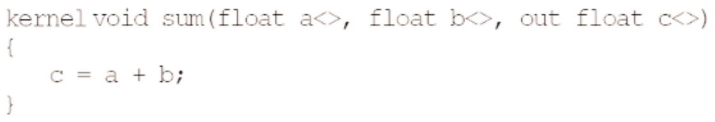
Of course, parallelism makes more sense in some cases than it does in others. Sometimes there isn't enough work in an individual thread to amortize the overhead of spawning and managing a large number of threads. Sometimes there isn't enough I/O bandwidth to fetch the data needed by hundreds or thousands of threads. Sometimes the datasets are too constricted by dependencies to be divided into thread-friendly chunks.

However, this is exactly the kind of analysis programmers must do when evaluating the potential data parallelism in an application. Opportunities for parallelism are not immediately obvious. As this practice becomes an integral part of the computer-science curriculum, it will no longer be the relatively rare skill it is now.

Kernel Functions Run on GPU

Now for a concrete source-code example. In ATI Stream terminology, a "kernel" is a special Brook+ function that uses the GPU to perform operations on data. Ideally, these operations will be parallel SIMD operations that use the multiple stream cores in each thread processor. Brook+ compiles all the code in a kernel for the GPU stream processor.

For example, adding the elements of one array to the elements of another array is an operation that can execute in parallel on many elements at the same time. In an ATI RV770, which has five general-purpose stream cores per thread processor, each thread processor can perform five 32-bit integer or single-precision floating-point adds in parallel. Figure 6 shows the kernel definition.



```
kernel void sum(float a<>, float b<>, out float c<>)
{
    c = a + b;
}
```

Figure 6. This example code defines a kernel function in Brook+ C. This kernel will be compiled for the ATI GPU. The new keyword "kernel" distinguishes between data-parallel functions intended for the GPU and conventional functions intended for the CPU.

The new keyword kernel specifies that Brook+ will compile this function for execution on the GPU, not on the x86 CPU. The familiar keyword void specifies that this function won't return a value to the CPU. The arbitrary name of this kernel is sum(), and its parameter list specifies two input arrays of single-precision floating-point numbers and the same type of output array for the results. Note that sum() will return results to the GPU via the output array, not to the CPU via a conventional return value, which is why the program declares this kernel function void.

Within the curly braces is the heart of this simple kernel. In a single line of code, it adds the two input arrays (a and b) and stores the results in the output array (c). In a conventional C program, this operation requires a loop that steps through the arrays and adds their elements iteratively. However, as explained above, a loop would generate a single thread of execution that wouldn't run any faster on the GPU than it would on the CPU. To leverage the resources of the GPU, the kernel must spawn multiple threads capable of executing in parallel.

That's exactly what this line of code does. Although it appears to be a simple addition, at runtime it will generate numerous threads. If the GPU has enough thread processors and stream cores to handle the number of threads created, then all threads will run in parallel. Otherwise, some threads will run in parallel and others will be time-sliced by the GPU's thread dispatcher.

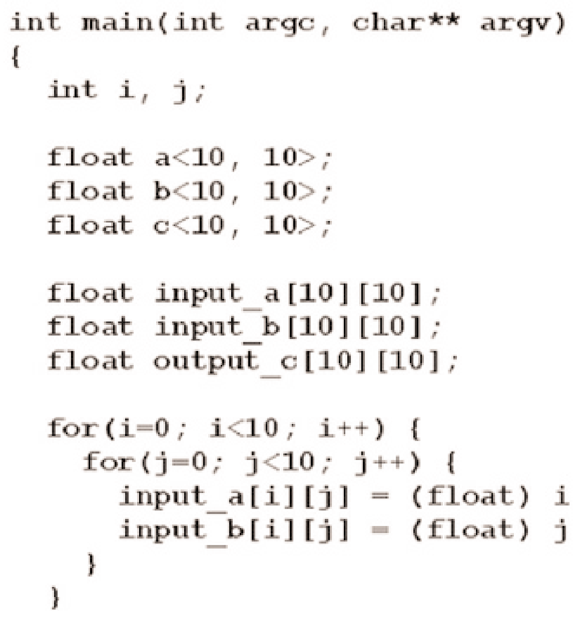
Setup Code Runs on CPU

So far, this example looks ridiculously simple—*too* simple. It doesn't reveal how the kernel determines the number of threads to create. Nor does it show how the programmer divides the data into thread-friendly chunks. These critical operations happen outside the kernel, in setup code that runs on the host CPU. Figure 7 shows the CPU code.

Remember that Brook+ translates all code in main() into standard C++, which another tool will compile for the x86 CPU. First, main() defines two integer variables (i and j) for use as loop counters later. Next, it defines three 10- × 10-element arrays (a, b, and c) of single-precision floating-point types. Note that these array declarations use angle braces (< >) instead of the usual brackets ([ ]). In Brook+, angle braces indicate special arrays called "streams." A stream is a collection of data (usually stored in an array) on which the GPU can perform operations using parallel threads.

Next, main() defines three conventional 10- × 10-element arrays (input_a, input_b, and output_c) using standard C-language brackets. These arrays are the same datatypes (single-precision floating point) as the streams. Data in these arrays will be stored in system memory accessible to the CPU. In contrast, data in stream arrays is stored in the GPU's video memory on the graphics card. In other words, each processor will have a copy of the arrays in its own memory. Why the redundancy? Because an ATI GPU can access video memory at about 100GB/s, whereas references to system memory must traverse the much slower PCI Express bus.

The next block of code in main() looks more familiar. Two nested loops walk through two of the conventional arrays (input_a and input_b), filling them with dummy data—the iterative values of the loop counters, typecast from integers to floats. A real program would replace this code with a routine that fills the arrays with real-world data, probably fetched from mass storage. Also, a real program could step through the conventional arrays in larger increments or in a



```
int main(int argc, char** argv)
{
  int i, j;

  float a<10, 10>;
  float b<10, 10>;
  float c<10, 10>;

  float input_a[10][10];
  float input_b[10][10];
  float output_c[10][10];

  for(i=0; i<10; i++) {
    for(j=0; j<10; j++) {
      input_a[i][j] = (float) i
      input_b[i][j] = (float) j
    }
  }
```

Figure 7. An example main() function in Brook+ C. This code prepares data for the kernel function in Figure 6, then calls the kernel and stores the results. All code in main() will be compiled for the x86 host processor.

scatter-gather pattern to populate the stream arrays with data in any manner desired.

In the last four lines of code, the real work gets done. Brook+ has a special function called streamRead() that copies the contents of a conventional array into a stream array. In other words, streamRead() copies data from system memory to video memory. In this example, two calls to streamRead() fill the two input streams (a and b) with data from the conventional arrays (input_a and input_b).

Next, the program calls the kernel function sum(), passing along the two input streams and receiving the summed results in the output stream (c). Finally, the program calls a special Brook+ function, streamWrite(), to copy the output stream into the conventional output array (output_c). As the counterpart of streamRead(), streamWrite() copies data from video memory to system memory, where the CPU can use it.

It's SIMD, Not MIMD

At first glance, this example seems to have no code that explicitly declares threads, assigns tasks to threads, and terminates threads when their work is done. But in an abstract way, it does. The statements defining the streams as 100-element arrays will direct the thread dispatcher to launch 100 threads when the program calls the sum() kernel. When sum() adds the arrays together, each addition operation will run as a separate thread on its own stream core. Excluding the overhead of setting up these operations, the GPU will add the 100-element arrays together in one clock cycle.

Technically, the Brook+ thread-programming model is explicit, because programmers determine the number of threads by dividing a dataset into subsets of arrays having no dependencies. It is the programmer's job to make those decisions. In a truly implicit threading model, the compiler or processor automatically finds the parallelism. Even so, Brook+ isn't nearly as explicit as C++ p-threads or Java threads. Conventional threading models require programmers to write code that explicitly declares threads, assigns tasks to threads, and terminates threads. Conventional threading models are difficult to debug and manage beyond a relatively small number of threads, whereas Brook+ makes it easy to create thousands of threads. (See *MPR 4/30/07-02*, "The Dread of Threads.")

Our example creates only 100 threads, leaving 700 of the stream cores in an RV770 twiddling their opposable thumbs. Ideally, the programmer would find a way to keep all 800 stream cores busy. Perhaps, in our example, the limitation is that a larger dataset isn't available. Or, maybe the dataset can't be divided into subsets larger than 100 elements without stalling on data dependencies.

Unfortunately, one method of occupying more stream cores—executing multiple *kernels* in parallel—isn't supported by the ATI Stream platform at this time. ATI Stream and ATI GPUs are based on a SIMD architecture, which applies a single operation to multiple data elements. To run multiple kernels in parallel, the hardware and software must support a MIMD architecture, which allows multiple instruction streams to operate on multiple data elements. That is task-level parallelism.

One exception is that ATI Stream can execute multiple streamRead() functions in parallel. CAL handles the scheduling at runtime, depending on the GPU's resources. Programs can anticipate this capability by calling a CAL function that returns information about the hardware at runtime.

Alternatives Are Remarkably Similar

ATI Stream has much in common with Nvidia's CUDA, the RapidMind Multicore Development Platform, and Intel's software architecture for future Larrabee processors. All rely heavily on data parallelism. All use proprietary variations of standard C or C++ that introduce new statements for expressing parallelism. All have a hardware-abstraction layer that insulates programmers from gory details of the microarchitecture.

Within this group, RapidMind's platform stands out as the most different. Whereas the others support only the vendor's own microprocessor architecture, RapidMind supports multiple architectures—including the GPUs from both AMD and Nvidia, the x86, and IBM's Cell Broadband Engine. Developers pursuing top performance can move their RapidMind software from one processor architecture to another with relatively little effort. RapidMind's runtime module further optimizes the code by automatically applying any techniques for instruction-level parallelism that are available on the target processor. (See *MPR 11/26/07-01*, "Parallel Processing for the x86.")

CUDA and ATI Stream are very much alike, a reflection of their similar target hardware—GPUs supporting the unified shader model. These GPUs have hundreds of simple function units, so data parallelism is their dominant threading model. Some language extensions in ATI Stream and CUDA are virtual clones. For instance, the streamRead() function in ATI Stream that copies data from system memory to video memory is essentially the same as the cudaMemcpy() function in CUDA. Thanks to these similarities, programmers who learn either ATI Stream or CUDA should be able to switch platforms with little trouble. (See *MPR 1/28/08-01*, "Parallel Processing With CUDA.")

Mainly, the choice is between which vendor currently ships the faster GPU, a matter of much contention. The latest ATI GPUs can execute more peak theoretical flops than the latest Nvidia GPUs can. Nvidia claims its GPUs can utilize more parallel threads in real-world HPC applications. *MPR* found only one HPC program written for both ATI Stream and CUDA: Stanford University's Folding@home client, a protein-folding program freely distributed over the Internet to thousands of computers and game consoles. According to the latest statistics from Stanford, both companies' GPUs deliver almost exactly the same floating-point throughput when running this program—about 110 gigaflops. In comparison, the IBM Cell processor in the Sony

OpenCL Tries to Standardize Parallel Programming

OpenCL is distantly related to OpenGL, the widely supported application programming interface (API) for graphics. Actually, OpenCL and OpenGL have little in common except similar names and the goal of insulating software developers from the intricacies of different microprocessor architectures.

Whereas the "GL" in OpenGL stands for Graphics Library, the "CL" in OpenCL stands for Computer Language. The terminology succinctly captures the difference. Instead of merely providing an API library, OpenCL modifies C and C++ for parallel programming. OpenCL also opens the door to other languages, such as Fortran.

Apple, AMD, and Nvidia have been prominent backers of this initiative. Apple plans to support OpenCL in a future release of Mac OS X (Snow Leopard) and played a key role in forming the OpenCL Compute Working Group last June. Apple's "Grand Central" technology for programming multicore processors is partly based on OpenCL.

AMD was among the first to join the OpenCL initiative. Nvidia vice president Neil Trevett is the chairperson of this committee, which is part of the Khronos Group, an industry consortium originally created to standardize multimedia APIs. Khronos announced the OpenCL 1.0 specification on December 8.

OpenCL enjoys such broad support that it seems almost guaranteed to alter the direction of software development for parallel processing. Members include 3DLabs, Activision Blizzard, AMD, Apple, ARM, Barco, Broadcom, Codeplay, Electronic Arts, Ericsson, Freescale, HI, IBM, Intel, Imagination Technologies, Kestrel Institute, Motorola, Movidia, Nokia, Nvidia, QNX, RapidMind, Samsung, Seaweed, Takumi, Texas Instruments, and Umeå University (Sweden). The most notable absence is Microsoft, which is building parallel-processing technology on the foundation of its DirectX graphics API.

Unlike OpenGL, which is specific to graphics, OpenCL aims to standardize general-purpose parallel programming for any application. It is especially suitable for heterogeneous systems—those having two or more microprocessor architectures. That includes PCs, which have x86-compatible CPUs and discrete or integrated GPUs. Many cellphones and other embedded systems also fit this category. OpenCL looks ideal for AMD's future Fusion processors, which will integrate x86-compatible CPU cores with an ATI GPU on the same chip.

Like ATI Stream, Nvidia's CUDA, and the RapidMind Multicore Development Platform, OpenCL adds new statements to C for expressing data-level parallelism. Essentially, these statements perform the same functions as similar statements in the ATI Stream, CUDA, and RapidMind versions of C. OpenCL builds on C99, an ISO specification of ANSI C issued in 1999. The threading model for data parallelism in OpenCL closely resembles the models in ATI Stream, CUDA, and RapidMind. Threading is largely implicit, but OpenCL allows programmers to manage threads more explicitly, if they wish.

In addition to data parallelism, OpenCL supports task-level parallelism. Unlike ATI Stream, it can concurrently execute multiple kernels on multiple CPUs, GPUs, or systems with mixed architectures. Because OpenCL is conceptually similar to RapidMind's solution, that company is adopting OpenCL for its multiarchitecture platform. Nvidia is also embracing the specification and announced on December 8 that OpenCL programs will run on CUDA.

Although today's ATI Stream platform isn't built on OpenCL, AMD doesn't regard it (or RapidMind, for that matter) as an unfriendly competitor. As a member of the OpenCL group, AMD worked with other companies to develop the standard and announced on December 8 that it is adopting the specification. AMD plans to integrate an OpenCL-compatible compiler and runtime package with a new version of the ATI Stream software development kit (SDK). This SDK, version 1.4, is scheduled for release in 1H09. It will allow OpenCL programs to run on ATI GPUs as handily as ATI Stream programs do.

Beyond that, AMD could modify its Brook+ version of C to incorporate the parallel-programming extensions of OpenCL. This option would provide an easy migration path for Brook+ programmers. Further down the road, AMD could offer its own OpenCL tool chain or leave that exercise to third parties. AMD wants to sell chips, not tools. If OpenCL catches on, the ATI Stream platform as it exists today could become redundant. But an industry standard for parallel programming would please AMD—and millions of software developers.

PlayStation 3 delivers about 28 gigaflops with Folding@home, and the x86 processors in Windows PCs average less than 1 gigaflops.

Another parallel processor coming down the pike is Intel's Larrabee, a manycore x86 chip with enhanced 16-lane SIMD engines. Judging from public disclosures, Intel's approach will be much like ATI Stream and CUDA. Larrabee will use its wide SIMD resources to exploit massively threaded data parallelism on multiple levels. Although Intel is using different terminology to describe these techniques—subdividing threads into finer components known as "strands" and "fibers"—the concepts are the same.

One important difference is that Larrabee is capable of greater MIMD-style task parallelism than existing GPUs are. Each Larrabee x86 core has four-way hardware multithreading in addition to a 16-lane SIMD engine. Those high-level "threads" are actually process-level tasks, so they can be completely different programs—even an operating system. As an x86 processor, Larrabee doesn't need a separate host processor, as a GPU does. Larrabee can run any

Price & Availability

AMD's ATI Stream platform for ATI GPUs is available on AMD's website. These software-development tools are free. For more information, see:

www.amd.com/stream

Starting in December, display drivers for recent ATI GPUs include the ATI Stream runtime package. These drivers can be downloaded for free. For more information, see:

http://ati.amd.com/support/driver.html

x86-compatible operating system and act as the master processor for its own SIMD slaves. (See *MPR 9/29/08-01*, "Intel's Larrabee Redefines GPUs.")

AMD hopes to introduce similar microprocessors in the future. AMD's Fusion project aims to integrate the GPU and CPU on one chip. This should greatly improve performance, because the GPU and CPU will no longer need to exchange data over the system's PCI Express bus. Instead, a wide on-chip bus will let them transfer data almost instantaneously, and perhaps share system memory, too. Unfortunately for AMD, the Fusion project has been repeatedly delayed. In November, AMD said the first Fusion processors won't begin production until 32nm fabrication is available—2011 at the soonest.

It's Good Enough for Now

Another alternative to ATI Stream that this article hasn't discussed is microprocessors designed specifically for general-purpose parallel processing. *MPR* has covered several of these advanced architectures. Their big advantage is that parallel processing is an inherent feature from the start, not a retrofit. The same architects who design the processor usually play a major role in designing the software tools, so the hardware and software are tightly integrated. Indeed, sometimes the tools were created first, and the CPU architecture was designed to suit the tools.

The biggest drawback of these architectures is that they require a commitment to a single vendor for both the hardware and software—and the vendor is often a startup with an uncertain future. Nevertheless, these visionary solutions are compelling. For three examples, see *MPR 10/10/06-01*, "Ambric's New Parallel Processor," *MPR 7/24/06-02*, "MathStar Challenges FPGAs," and *MPR 11/5/07-01*, "Tilera's Cores Communicate Better." Note, however, that MathStar has ceased operations and Ambric is currently looking for a buyer. Times are tough for startups.

Of course, these days, even the futures of long-established companies like AMD and Nvidia are not as certain as they used to be. Intel's arrival in this market with Larrabee in 2009 could shake things up still further. Nevertheless, the reality is that ATI and Nvidia GPUs have huge installed bases right now. There's nothing unproven about their hardware or software.

It's possible that ATI Stream will be a temporary bridge to better development tools and programming languages, either from third-party vendors like RapidMind or from industry consortiums like the OpenCL group. At this point, it's unclear how soon (or whether) a truly industrywide standard for parallel programming will emerge.

Meanwhile, programmers have jobs to do and code to write. GPUs, in their new guise as "stream processors," have tremendous potential. If an application lends itself to data parallelism and is important enough to justify the development effort, there's little reason to wait for the perfect solution to appear someday. Even if the code needs rewriting later, it will be useful now, and the learning process will be valuable—especially if OpenCL catches on, because it's so similar.

Finally, consider the possibility that parallel code may not enjoy the historic longevity of sequential code. COBOL lives forever, but programs written for any parallel processor in any language may need rewriting more frequently. Parallelism is a paradigm shift, and paradigm shifts change the rules. ◇