Reed Electronics Group

MICROPROCESSOR REPORT

www.MPRonline.com

THE INSIDER'S GUIDE TO MICROPROCESSOR HARDWARE

THE EDITORIAL VIEW

Tools for Multicore Processors

by Tom R. Halfhill {7/28/08-02}

We keep hearing more complaints that it's hard to write software for multicore processors because there aren't enough development tools. Not enough tools? That's like complaining it's hard to buy Chinese products because there aren't enough Wal-Marts.

The real problem with multicore processors is *too many* development tools—and the tools are often difficult to learn and use. What programmers actually seem to want is just one tool. It would work with the most popular CPU architectures, let programmers write serial code in plain-vanilla C, automatically extract parallelism from their code during compilation, and automatically exploit the additional cores in future processors.

Well, sure, we'd all like that. And we would also like an electric car that carries six people, runs 500 miles on a quick charge, and costs $15,000. Maybe it's not impossible, but it's not imminent, either.

Back to reality. We have covered several software-development tools for multicore processors in *Microprocessor Report*, usually while describing the microprocessors they target. These tools fit three categories:

- Tools for a single (usually specialized) CPU architecture.
- Tools that use hardware abstraction to span multiple CPU architectures.
- Tools that require programmers to explicitly manage threading.

Most complaints about software-development tools seem to come from programmers writing mainstream software for PCs and servers. Development tools for embedded processors and high-performance computing (HPC) are more plentiful. That's logical, because multicore processors have a stronger beachhead in the embedded and HPC markets, and the level of integration (cores per chip) is often much higher. Then, too, many embedded and HPC programs are "embarrassingly parallel"—the computer-science buzzword for applications that have so much parallelism, exploiting it is easy to the point of embarrassment.

Tools for Specialized Processors

Let's examine the three categories of software-development tools in turn. First are the tools for specialized CPU architectures. Typically, these tools are created by the same company that invented the architecture. The architecture tends to be the foundation for exotic, often massively parallel processors designed for rather narrow application domains, such as video compression, packet processing, or digital signal processing.

The individual processor cores may be very simple—eight- or 16-bit cores aren't unheard of—but a single chip may have hundreds of them. The inventors of these architectures must create their own development tools, because their companies are usually startups. Third-party toolmakers aren't interested and probably lack the expertise.

Numerous examples we've covered in *MPR* include Ambric, Connex Technology, Elixent, Eutecus, IBM, MathStar, NEC, PicoChip Design, Silicon Hive, Teja Technologies (since acquired by ARC International), Tilera, and XMOS Semiconductor. (For *MPR* article references, see the "For More Information" box.)

Most of these companies are emerging from startup mode, but there are established firms, too. One is NEC, with its Dynamic Reconfigurable Processor. Another is IBM, with its Cell Broadband Engine. Cell BE is a more general-purpose processor than others in this group, but its Synergistic Processor Elements are data engines based on a different architecture than the Power Architecture control processor.

The advantage of these types of devices is a CPU architecture designed for a specific application domain, so they can be very good at what they do. They don't need to run word processors and web browsers. The same engineers who designed the architecture probably also created the software-development tools or had a great deal of input. Indeed, the CPU architecture may be designed to fit the tools, not vice versa. As a result, these processors tend to be highly tuned performance machines.

Of course, they have disadvantages, as well. The development tools usually add proprietary extensions to standard C or C++ and tie the source code to the CPU architecture. The processor, development tools, and application code are tightly bound together, so everything either sinks or swims. If the startup fails and the architecture dies, the source code probably isn't very portable to another architecture. If developers like the processor but hate the tools, there are no alternative tools for that processor. If developers like the tools but hate the processor, there are no alternative processors compatible with the tools.

Tools for Multiple CPU Architectures

Another approach is to create parallel-processing software-development tools that work with multiple CPU architectures—sometimes architectures having almost nothing in common. To insulate programmers from the metal, there's either a run-time hardware-abstraction layer or an interchangeable back-end code generator on the compiler. Unlike the development tools in the first category, tools in this group don't necessarily bind developers to a specific CPU architecture or CPU vendor.

Examples we've covered in *MPR* include Nvidia, PeakStream (since acquired by Google), and RapidMind. In addition, Fujitsu has shown a remote asynchronous procedure-call mechanism that fits somewhere between this category and the next one. Fujitsu's mechanism combines conventional thread programming with procedure calls that can leap across the chasm separating different platforms.

Nvidia's Compute Unified Device Architecture (CUDA) is a software-development platform for massively parallel high-performance computing on the company's powerful GPUs. It was formally introduced in 2006, and Nvidia recently extended CUDA to the Intel x86 architecture. CUDA is free, so it's an attractive way to get started.

PeakStream and RapidMind focus on software-development tools alone. Since the Google acquisition, PeakStream has virtually vanished into the Googleplex, leaving more room in the market for RapidMind. The latest version of the RapidMind Multicore Development Platform works with x86 processors from AMD and Intel; the GPUs from ATI (AMD) and Nvidia; and IBM's Cell BE. Note that those CPU architectures are nearly as disparate as any available.

RapidMind's hardware-abstraction layer hides low-level details from programmers, who write code in C++ extended with new constructs and functions. Once written, the same source code can run on any of the supported CPUs. New generations of those CPUs—even those with additional processor cores—can also run the same code, although modifications will deliver more performance.

Likewise, programmers writing code for the CUDA or PeakStream platforms use C or C++ with special (albeit different) extensions. CUDA departs from RapidMind by using a longer tool chain that compiles the code differently for the various CPUs and GPUs it supports, although the GPU driver provides some hardware abstraction.

The biggest advantage of the software-development tools from Nvidia, PeakStream, and RapidMind is multiplatform flexibility. They support radically different CPU architectures with a recognizable version of C/C++, and they don't require programmers to completely rewrite their code when switching architectures or microarchitectures. In addition, these tools don't tie developers to a specific CPU vendor—particularly a startup vendor whose future is uncertain.

On the downside, these tools do tie developers to the tool vendor. Early adopters of PeakStream's tools probably weren't thrilled when Google swallowed the company. Even if they still receive tech support, what is their long-term outlook?

CUDA is a good bet to stay around, because Nvidia is a substantial company. RapidMind is a startup and appears to be stable. But even if something bad happens to RapidMind, another company would probably snap up RapidMind's computer-science whizzes and their innovative technology. Also, programmers experienced with RapidMind's tools are better prepared to port their source code to a platform like CUDA—or vice versa.

Tools for Explicit Threading

MPR has written little about explicitly threaded software-development tools, because it's the most conventional approach to the problem and has been around for many years. It's not a very flexible approach, and we doubt its viability in the long run. Unfortunately, it's also the programming model that most of today's programmers are most familiar with.

In the context of this discussion, "explicit threading" means writing code to create a thread, protect it against deadlocks, and manage the thread's life cycle. Programming languages requiring this degree of attention include Java and conventional C/C++ (e.g., the Posix pthread API). Implementing threads in this manner is too limited and too prone to bugs. Beyond a relatively small number of threads, the task often becomes unmanageable and unpredictable. (See *MPR 4/30/07-02*, "The Dread of Threads.")

Keep in mind that some development tools we've mentioned in the other two categories can spawn dozens, hundreds, or even thousands of simultaneous threads from only a few lines of high-level code. And whereas PC processors are creeping toward eight cores per chip, *MPR* has written about embedded processors with more than 4,000 cores per chip. Explicitly threaded programming languages can't keep up with tomorrow's (or even today's) manycore and massively parallel processors.

Nevertheless, explicit threading has its place. Some programs are largely serial in nature and either can't use or don't need massive threading. (Word processors and low-level embedded tasks come to mind.) Other applications have lots of inherent parallelism and few or no dependencies among threads to worry about, so spawning new threads is safe and easy. (A process that services numerous visitors to a website is a good example.) Traditional software-development tools can serve those needs well. One example is Intel's Threading Building Blocks, a run-time library for C++. It helps programmers implement multithreading on x86 processors, and it's a multiplatform solution, supporting the Windows, Linux, and Mac OS X operating systems.

The big advantage of these tools and languages is their relative familiarity. Their biggest disadvantage is the difficulty of exploiting the growing number of cores appearing in processors. Barring an unexpected breakthrough that has eluded computer scientists for 50 years, these solutions will have trouble keeping up with the multicore trend over the long term. Software that can't take advantage of more cores will reach a performance plateau that tracks the slow progress in clock frequency.

The Quest for Standards

This editorial barely skims the surface of parallel programming, yet it's clear that software developers have many more choices than they probably realize. To narrow the field, developers can decide which element of a project matters most—the microprocessor or the software-development tools. Either preference will greatly reduce the available options for the other element.

More options are coming. Numerous companies, universities, and consortiums are attacking the problem of parallel programming at various levels. AMD, IBM, Intel, Microsoft, and Sun Microsystems are linking up with university research projects to develop better solutions. Intel and Microsoft are jointly sponsoring a pair of research centers at the University of California at Berkeley and the University of Illinois at Urbana-Champaign, committing $20 million over the next five years. The Multicore Association, an industry consortium, has a new working group devoted to programming practices for multicore processors. The group's work is based partly on technology from Critical-Blue and its Multicore Cascade tools.

I expect most of these projects to settle on language extensions and run-time libraries that standardize parallel programming with C++, much as C++ added object-oriented extensions to procedural C. This solution will be satisfactory for mainstream applications, but it's not a revolutionary breakthrough for top performance in niche applications. As always, the very best results will require special effort with specialized tools—and probably specialized CPU architectures, too. ◇

Tom R. Halfhill

For More Information

- Ambric (see *MPR 10/10/06-01*, "Ambric's New Parallel Processor")
- Connex Technology (see *MPR 1/9/06-01*, "Massively Parallel Digital Video")
- Elixent (see *MPR 6/27/05-02*, "Elixent Improves D-Fabrix")
- Eutecus (see *MPR 2/12/07-01*, "Faster Than a Blink")
- Fujitsu (see *MPR 8/13/07-01*, "Fujitsu Calls Asynchronously")
- IBM Cell Broadband Engine (see *MPR 2/14/05-01*, "Cell Moves Into the Limelight")
- MathStar (see *MPR 7/24/06-02*, "MathStar Challenges FPGAs")
- NEC DRP (see *MPR 11/25/02-04*, "New NEC Array Speeds Data")
- Nvidia CUDA (see *MPR 1/28/08-01*, "Parallel Processing With CUDA")
- PeakStream (see *MPR 10/2/06-01*, "Number Crunching With GPUs")
- PicoChip Design (see *MPR 10/14/03-03*, "PicoChip Makes a Big MAC")
- RapidMind (see *MPR 11/26/07-01*, "Parallel Processing For the x86")
- Silicon Hive (see *MPR 6/20/05-01*, "Busy Bees at Silicon Hive")
- Teja Technologies (see *MPR 4/3/06-02*, "Teja's FPGA Play")
- Tilera (see *MPR 11/5/07-01*, "Tilera's Cores Communicate Better")
- XMOS Semiconductor (see *MPR 8/6/07-01*, "XMOS Redefines Silicon")