Reed Electronics Group

MICROPROCESSOR REPORT

www.MPRonline.com

THE INSIDER'S GUIDE TO MICROPROCESSOR HARDWARE

PARALLEL PROCESSING FOR THE x86

RapidMind Ports Its Multicore Development Platform to x86 CPUs

By Tom R. Halfhill {11/26/07-01}

The Holy Grail in computer science is a high-level compiler that automatically extracts hidden parallelism from existing source code and efficiently distributes the workloads on the latest multicore processors. Ideally, programmers need not rewrite any code, and the compiler transparently targets microprocessors with any number of cores.

Dream on. Conventional serial code doesn't surrender its hidden parallelism (if, indeed, any exists) without a fight. True, a good vectorizing compiler can find some small-scale data parallelism, assuming the processor has vector-math instructions. Optimizing compilers can find some instruction-level parallelism when targeting processors with superscalar dispatching and other fancy features. And microprocessors with dynamic branch prediction, speculative execution, and out-of-order execution can find a little more parallelism at run time. But none of these techniques fully exploits the rapidly expanding resources of the latest multicore designs.

Due to those limitations, programmers must rewrite at least some of their source code to explicitly expose parallelism to the compiler or the processor. It's not the ideal solution. But for now—and possibly forever—it's the best way to keep multiple processors busy.

RapidMind is one of several parties entering the market for parallel processing. Founded in 2004, RapidMind is a privately funded company based in Ontario, Canada. The RapidMind Multicore Development Platform does require programmers to rewrite the data-intensive portions of their code, and it also requires the target system to run a hardware-abstraction layer between the application program and the microprocessor. In return for those compromises, RapidMind claims big benefits. Some tasks run five to ten times faster, and, in some cases, performance can scale faster than the rising number of processors. In addition, the parallel code is highly portable—programmers needn't rewrite it for each new multicore processor or multiprocessor system.

Previously, RapidMind's platform worked only with IBM's Cell Broadband Engine (Cell BE) and the graphics processors from AMD/ATI and Nvidia. On November 5, RapidMind announced Multicore Development Platform v3.0, which targets the popular multicore x86 processors from AMD and Intel—a big step. This move opens up new opportunities for RapidMind in general-purpose computing, such as desktop publishing. Until now, RapidMind focused mainly on high-performance computing: financial modeling, image processing, data mining, scientific analysis, simulations, broadcast-quality multimedia generation, 3D visualization, transactional databases, and so forth.

Mainstream PC software has comparatively little inherent parallelism, so RapidMind's platform is less useful for general productivity applications. But RapidMind's embrace of the x86 is a boon for heavy-duty number crunching on commodity hardware.

RapidMind's Parallel-Processing Platform

In some ways, RapidMind's approach to parallel processing resembles that of PeakStream, which *Microprocessor Report* covered last year. (See *MPR 10/2/06-01*, "Number Crunching With GPUs.") Both technologies made their debut on the highly parallel math engines of graphics processors; both technologies require programmers to explicitly expose data

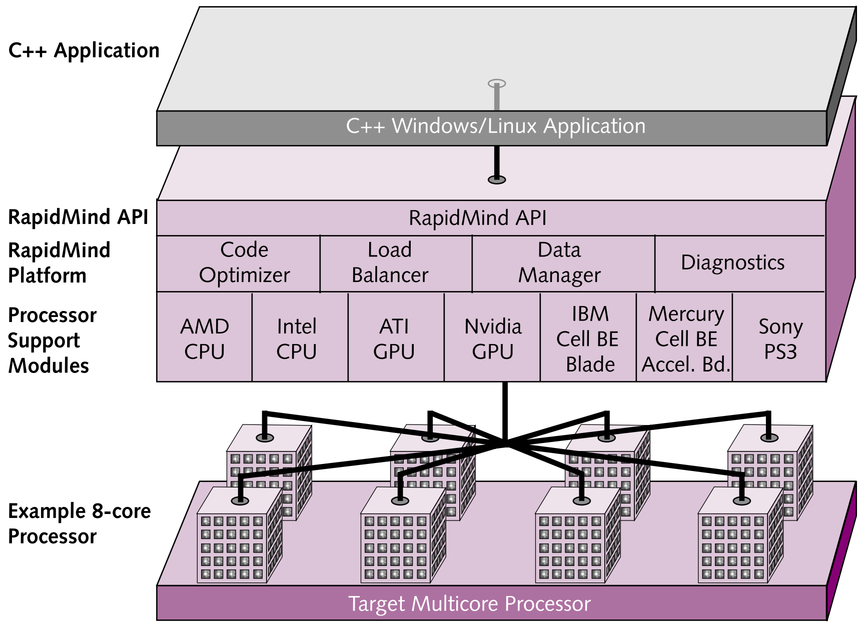

Figure 1. RapidMind's Multicore Development Platform v3.0. A hardware-abstraction layer insulates application code from the underlying processors, automatically handling the load balancing and other low-level functions necessary for parallel processing. At the lowest layer, the processor-support modules are interchangeable software "drivers" that adapt the higher layers to different microprocessor architectures—now including the x86.

parallelism in the source code, but without explicit threading; both technologies use some just-in-time (JIT) compilation to optimize the parallel code; and both technologies provide an application programming interface (API) that largely automates the task of distributing execution among multiple processors or cores. (Last June, Google acquired PeakStream for an undisclosed sum and withdrew its products from the market.)

There are important differences, however. Whereas PeakStream provided a fixed API library of common math functions callable from C++, RapidMind allows programmers to define their own functions in C++. Indeed, user-defined functions are so central to RapidMind's technology that the company has created new C++ constructs for them. As a result, C++ code written for RapidMind's platform differs more visibly from plain-vanilla C++ than PeakStream code does. In both cases, however, programmers expose data parallelism by manipulating specially typed arrays.

Before diving into the source code, it's helpful to understand how RapidMind's Multicore Development Platform works. Figure 1 illustrates the platform's software architecture. A vital point is that the parallel-processing code in the application program runs on an abstraction layer that insulates the code from the hardware. (Conventional serial code runs normally.) Thanks to this virtual-machine layer, the parallel-processing code isn't specific to any particular microprocessor architecture or microarchitecture, and it's not bound by the number of processors or cores. Code written today for a dual- or quad-core processor can run without modification on future designs with many more processors. (RapidMind supports the latest Nvidia GeForce G80 graphics processors that have 128 shaders, which can serve as general-purpose ALUs.)

Figure 2 illustrates the versatility of RapidMind's abstraction. Although programmers must write special C++ code to express data parallelism, once the code is finished, it can run on several different types of parallel-processing systems. These systems can achieve parallelism using multiple discrete processors, a host processor assisted by an accelerator, one or more multicore processors, or multicore processors with homogenous or heterogeneous cores. Depending on the type of application and target hardware, RapidMind's run-time package occupies about 5MB of system memory.

Another key point of RapidMind's technology is that it doesn't use multithreading—at least, not in the common sense. At the hardware level, of course, multiple streams of execution are running on multiple processors. But programmers needn't explicitly define or manage threads when writing their code. It's rather like writing high-level code without worrying how a superscalar processor will divide a single instruction stream into multiple streams for execution in the parallel pipelines.

In RapidMind's scheme, the abstraction layer automatically divides and distributes a task among multiple processors. (The load balancer seen in Figure 1 is primarily responsible.) By eliminating the need for explicit multithreading, RapidMind simplifies parallel programming and squashes bugs before they breed. The company claims its technology abolishes deadlocked threads and thread-synchronization problems—frequent sources of trouble in conventional

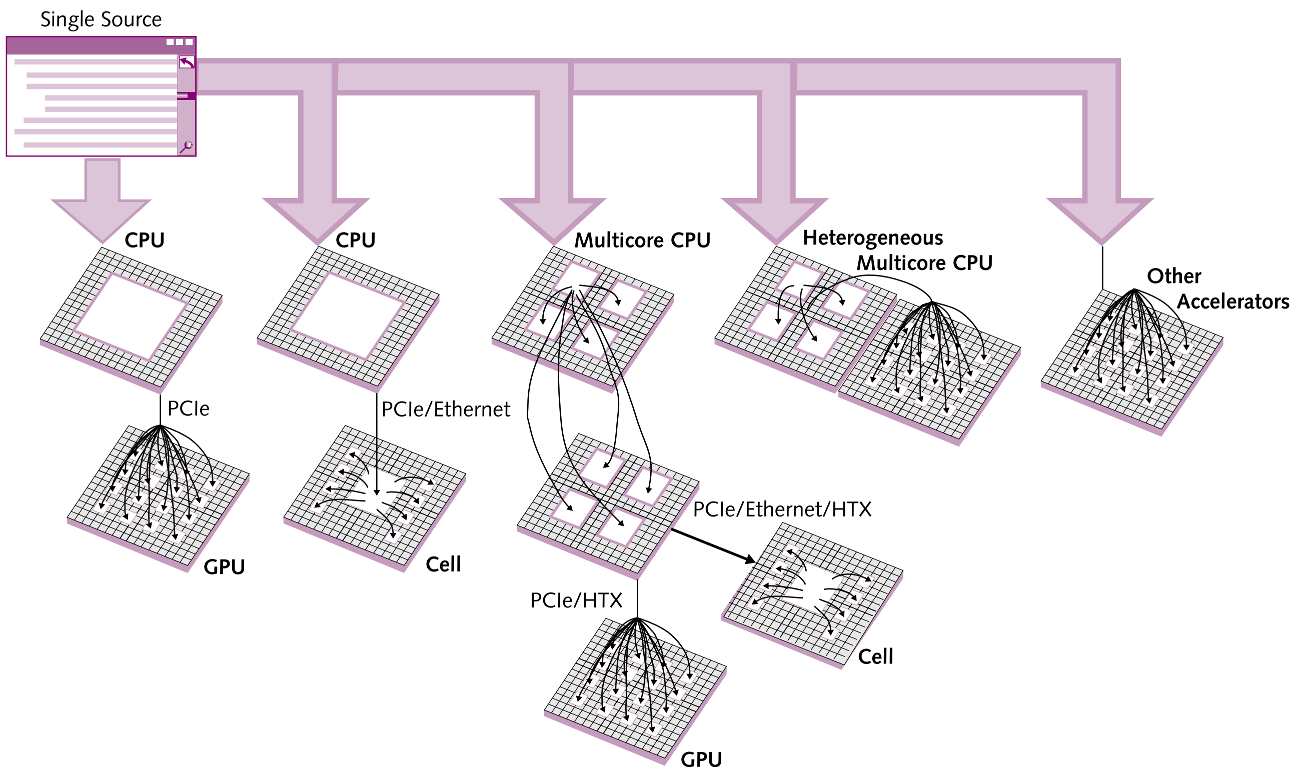

Figure 2. Parallel processing with RapidMind's platform. C++ code written with RapidMind's technology can run on several different kinds of parallel-processing systems, whether they have multiple single-core chips, multicore chips, or a combination of processor types. The processor cores may be homogenous or heterogenous. RapidMind's hardware-abstraction layer shields the application code from these low-level details.

multithreaded code. (See *MPR 4/30/07-02*, "The Dread of Threads.")

C++ Functions Express Parallelism

As mentioned above, one of the prices for parallelism when using RapidMind's technology is that developers must write special C++ code to expose the data parallelism in their software. When modifying existing serial code, the extent of the rewrite depends on the degree of parallelism extracted. If that price seems high, consider that recoding will be necessary for virtually any software that has a significant amount of exploitable parallelism, whether using RapidMind's platform or an alternative.

To ease the task, RapidMind adds new classes and constructs to C++. Programmers can use Microsoft Visual C++ v7 or v8 with Windows, or GNU C Compiler (GCC) v4 with Linux, after including RapidMind's header file and linking to RapidMind's API library. Although the library provides many predefined functions, programmers will probably write most of their own functions, often calling upon the predefined ones. As with conventional C/C++ functions, the special RapidMind functions can call any other functions, including each other.

A vital part of the RapidMind run-time package is a JIT compiler that converts the special parallel-processing functions into native code at run time. RapidMind's JIT compiler is similar to a Java JIT compiler, with two important exceptions. The programming language is C++, not Java, and portions of the application program written in conventional serial code are statically compiled by the regular C/C++ compiler, as usual. Only the parallel code in the special RapidMind functions is dynamically compiled.

This combination of static and dynamic compilation allows serial code to run conventionally while parallel code adapts at run time. For instance, parallel code written for a dual-core processor will run without modification on a quad-core processor and will take advantage of the additional cores. At run time, the RapidMind abstraction module discovers the additional processing resources and invokes the JIT compiler to automatically generate the appropriate executable code. By swapping the abstraction module for another module, it's possible to run the same parallel code on a different CPU architecture without static recompilation. (The serial portion of the application program written in regular C/C++ must be recompiled, of course.) Moreover, dynamic compilation allows RapidMind to use performance-enhancing techniques applicable only at run time, including optimizations based on system feedback. RapidMind's run-time module has instrumentation for this purpose.

To write the parallel-processing functions, programmers must express an algorithm as operations on specially typed data arrays. To help programmers do this, RapidMind has created the C++ classes Value and Array to provide a familiar interface to C++ programmers. Both classes define objects that are data containers. The Value class is an *N*-tuple containing *N* number of values of type *T*, where *T* is a numeric datatype (signed or unsigned integers of various lengths, floating-point numbers of various lengths, and Booleans). For example, Value1f contains a single-precision floating-point value (like a float in C). Value3f is a three-tuple (triple) single-precision floating-point value—three floats in one container. Programs can manipulate these values using standard C operators and functions, and RapidMind's API library provides additional functions.

RapidMind's Array class defines objects whose elements are Value objects. Unlike a Value object, an Array object is multidimensional (up to three dimensions) and can change size at run time. Standard operators and functions provide the usual random access to array elements. Convenient predefined functions like grid, slice, offset, and stride allow programs to access subarrays in various ways without resorting to troublesome pointer arithmetic.

Writing Parallel-Processing Code

As relatively minor extensions of ordinary datatypes and arrays, RapidMind's Value and Array classes should be quickly grasped by C++ programmers. The key to RapidMind's technology is the Program class, which contains the special parallel-processing functions compiled at run time. These functions execute a sequence of operations, much as a macro does—including operations that can run in parallel on multiple processors. The keywords BEGIN and END define the boundaries of the parallel-processing functions. Here is an example of a simple Program object:

```
Value3f n;
Program p = BEGIN {
   In<Value3f> v;
   Out<Value3f> h;
   h = normalize(v + n);
} END;
```

The first line declares a tuple (n) of three single-precision floating-point values. Note that this nonlocal variable will be accessible by the dynamically compiled Program object, even though it's declared in the statically compiled portion of the program. The second line creates the new Program object (p) and begins defining a function for dynamic compilation and parallel processing. The next two lines declare local variables for input (v) and output (h) arrays, both of the Value3f type.

The fifth line does the real work. It applies a RapidMind predefined function (normalize) to the input array (v) while adding the previously declared tuple (n) to each tuple in the array. This example demonstrates that Program objects can use standard C/C++ operators on Value and Array types. In this case, each addition adds the three floating-point values in the nonlocal tuple (n) to the three floating-point values in each array element. Then, after normalizing the sums, the same line of code stores the result (another Value3f tuple) in the output array (h). The final line of code ends the Program function.

Although this Program object calls a predefined function (normalize) in RapidMind's API, it could just as easily call a user-defined function written in plain C++:

```
Value3f normalize(Value3f vv) {
 return vv / length(vv);
}
```

Once a Program object is defined, it can operate on tuples or arrays. For example, assume that a program declares the two-dimensional arrays V and H, both Array objects populated with objects of the type Value3f. The following code applies the operation defined in Program p to all elements of array V (one million elements), returning the results in array H:

```
// array dimension can be 1, 2, or 3
Array<2,Value3f> V(1000,1000);
// initialize V with data here
Array<2,Value3f> H;
H = p(V);
```

Invoking Program p on a Value3f array with one million elements will generate approximately ten million floating-point operations. (For each element in the array, there will be three adds, then a Euclidean-length computation requiring a dot product with three multiplies and two more adds, then a reciprocal square root, and then a multiply.) The results are Value3f tuples.

At run time, the JIT compiler dynamically compiles Program p for the target system, taking into account the number of processors available, their microarchitectures, and other factors. In a homogenous dual-processor system (which could be a dual-core microprocessor or two discrete microprocessors), the RapidMind load balancer might run half of those ten million operations on each processor. In a quad-processor system, the load balancer might run a fourth of those ten million operations on each processor. In practice, the load balancer is smarter than that. It can distribute the workload among heterogeneous processors according to their capabilities, and it may shift some work away from the processor that's running the operating system and the RapidMind run-time package itself.

Dynamic flow control is another important part of this concept. RapidMind has added special keywords for branching (IF, ELSE, ELSEIF, etc.) and looping (FOR/ENDFOR, WHILE/ENDWHILE, DO/UNTIL) that mimic the standard C/C++ keywords for those operations. The new keywords allow dynamically compiled Program objects to make decisions and perform iterative operations independently of flow controls in the statically compiled code.

In effect, RapidMind has created a language within a language for writing programs within a program. Instead of frightening programmers with the unfamiliar syntax of a wholly new language, RapidMind imitates familiar C++ syntax. RapidMind's philosophy is that parallel processing must be easy enough for all programmers to grasp, not just a lab exercise for computer scientists.

The dynamism of the RapidMind platform is vital, because it allows programmers to write parallel code that's independent of the underlying microprocessor architecture and of the system's parallel-processing resources. The source code shown in this example can run on an AMD or Intel x86 with one or more cores, an IBM Cell BE with nine cores, or an ATI or Nvidia graphics processor with dozens of cores. Existing serial code written in regular C/C++ can remain undisturbed; only those portions of existing programs that offer opportunities for data parallelism need rewriting. And once written, the parallel code should be future-proof.

Benchmarking Parallel Performance

RapidMind hasn't yet ported industry-standard benchmark suites to its platform, but the company does offer benchmark results for some problems commonly tackled in high-performance computing. One example is the Black-Scholes model, which stock analysts use to determine the fair market value of an equity. One popular solution to the model is a Monte Carlo analysis. Figure 3 compares serial C++ code for this solution with RapidMind's parallel C++ code. The serial code isn't a straw man—it was written and hand-tuned by experienced Hewlett-Packard programmers. (For more information about these tests, see the joint HP/RapidMind white paper referenced in the "Price & Availability" box.)

Figure 4 shows one result of this benchmark test. The RapidMind code runs nearly eight times faster on an Intel Xeon Core 2 Quad (2.66GHz) than HP's code does when running on one core of the same processor. Usually, programmers struggle to achieve linear performance gains when writing code for multicore processors, but in this case, RapidMind achieved 2x linear performance. This result demonstrates that RapidMind's platform does more than simply distribute a workload among multiple processors—it also applies other optimizations, some of them possible only with the run-time knowledge that comes with dynamic compilation. Of course, it's logical to assume that RapidMind chose this example to show the Multicore Development Platform in the best light. Also, the results don't distinguish between the performance achieved by parallelizing the code and by applying other optimizations. Until better benchmarks are available, *MPR* is skeptical that the results are always this good. Nevertheless, RapidMind's example does show that better-than-linear improvement is possible.

Figure 5 compares the performance of serial and parallel code on x86 processors to RapidMind's parallel code running on an Nvidia GeForce 8800 GTX graphics card, using the shaders as general-purpose parallel-processing engines. (RapidMind's platform also targets AMD/ATI graphics cards, such as the x1900XT.) The Nvidia G80 graphics processor handily beats a 2.66GHz Intel Xeon Core 2 Quad processor—which is why RapidMind continues to support GPUs for high-performance computing.

To scale performance at a greater-than-linear rate, the RapidMind platform optimizes the software in several ways.

Typical C++ Code

```
for (int i = 0; i < num_experiments; i++) {
  float i1 = float(i + 0.5f) / num_experiments;
  float i2 = bitreverse(i + 1);

  boxmuller_shirley(i1, i2, phi_const[0],
  phi_const[1]);

  s1_const[0] = S_0 * expf(R + SDT * phi_const[0]);
  s1_const[1] = S_0 * expf(R + SDT * phi_const[1]);

  CT = std::max(s1_const[0] - K, 0.0f);
  sumCT = sumCT + CT;
  sumCT2 = sumCT2 + CT*CT;

  CT = std::max(s1_const[1] - K, 0.0f);
  sumCT = sumCT + CT;
  sumCT2 = sumCT2 + CT*CT;
}
```

RapidMind-enabled C++ Code

```
Program blackscholes = BEGIN {
 In<Value1i> i;

 Value2f hammersley = join((i + 0.5f) /
 num_experiments, bitreverse (i + 1));

 Value2f phi = boxmuller_shirley(hammersley);

 Value2f S = S_0 * exp(R + SDT * phi);
 Value2f CT = rapidmind::max(0.0f, S - K);

 Out<Value2f> CT_CT2 = join(sum(CT), dot(CT, CT));
} END;

// ...

Value2f sum_of_CT_and_CT2 = sum<Value2f>(blackscholes
(grid (num_experiments)));
```

Figure 3. Comparison of C++ code before and after rewriting for RapidMind's Multicore Development Platform. These routines solve a Black-Scholes model used by stock traders. Hewlett-Packard wrote the single-threaded code and hand-optimized it for Intel x86 processors. RapidMind wrote the parallel code using new constructs added to C++.

As seen in the code examples above, the special Program functions primarily expose large-scale data parallelism. Programmers can also exploit data parallelism on a smaller scale by using the single-instruction multiple-data (SIMD) instructions commonly found in CPU instruction sets, such as the MMX and SSE extensions in x86 processors. (RapidMind's documentation refers to "SIMD within a register [SWAR] vectorization," which simply means that some SIMD instructions operate on multiple data elements packed into a single register.) Exploiting data parallelism in these ways usually requires programmers to exert manual labor, although a good vectorizing compiler can help. RapidMind says its v3.0 compiler automatically performs some vectorization.

Instruction-level parallelism can be largely automated by the compiler and the microprocessor. An optimizing compiler rearranges instructions to take advantage of superscalar pipelines and other microarchitectural features of the target

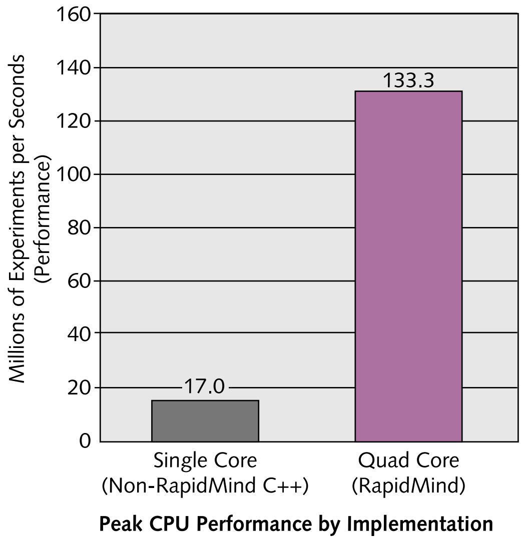

Figure 4. Performance comparison of serial C++ code vs. RapidMind C++ code. These are the benchmark results obtained by running the subroutines in Figure 3. The Y-axis shows the number of Black-Scholes "experiments" (speculative evaluations of stock prices) per second. In this example, RapidMind's performance scales at a greater-than-linear rate.

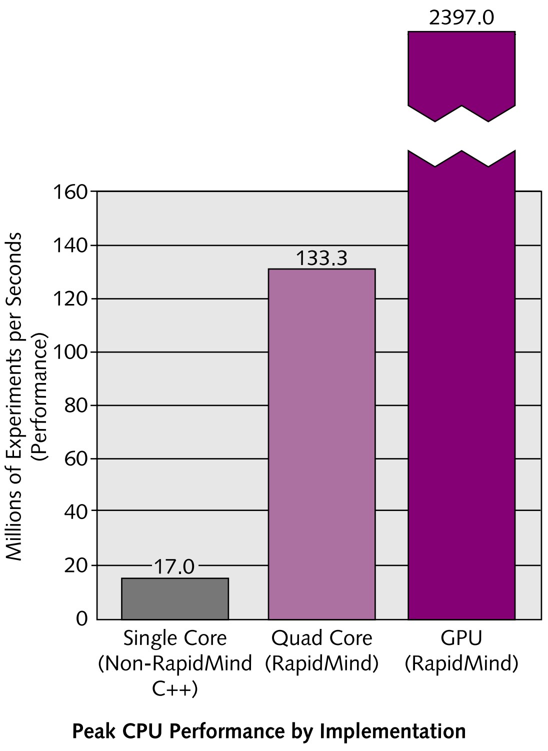

Figure 5. Performance comparison of serial C++ code vs. RapidMind C++ code on x86 processors and an Nvidia GeForce 8800 GTX graphics card. RapidMind harnessed the shaders in Nvidia's G80 graphics processor as general-purpose processing engines, delivering stellar performance in this Black-Scholes benchmark test. The x86 processor was an Intel Xeon Core 2 Quad. As in Figure 4, these are the results of running the subroutines in Figure 3, and the Y-axis shows the number of Black-Scholes "experiments" (speculative evaluations of stock prices) per second.

CPU. Most advanced processors can dynamically predict branches, reorder the instruction stream, and speculatively execute instructions at run time. Some processors support hardware multithreading and/or simultaneous multithreading, using superscalar pipelines and multiple banks of registers. The drawback of instruction-level parallelism is that it's notoriously difficult to find in most software. Rapidly diminishing returns limit the application of these techniques, which is one reason for the trend toward multicore processors.

Despite the limitations of instruction-level parallelism, RapidMind tries to leverage any mechanisms for parallelism it can find. The RapidMind programming model focuses on large-scale data parallelism, but the runtime module optimizes the software for the specific microarchitecture of the target processor, using any techniques available.

A Realistic Approach to Parallelism

RapidMind's Multicore Development Platform isn't the Holy Grail, because it relies on writing or rewriting software to explicitly expose data-level parallelism. The perfect solution would be a really smart compiler that does all the work automagically. But waiting for the perfect solution could keep a software developer out of the loop for a long time. The search for the Holy Grail of compilers might take as long as...well, the search for the Holy Grail.

Parallel processing isn't new, of course. Computer scientists have been wrestling with it since the 1950s. It seems new to the mainstream software-development community only because Moore's law is shrinking multiprocessor systems onto individual chips, at a time when rising power consumption is putting the brakes on clock-frequency scaling. Now, even small embedded systems are becoming multiprocessor systems. Going forward, multicore processors are the only way to continue applying Moore's law to microprocessors. Suddenly everyone, not only computer scientists, worries about writing parallel software.

RapidMind takes a broad approach to parallelism that seizes upon every available technique. It's not surprising that the biggest gains require the most labor. It's hard to beat the brain of a human programmer who is intimately familiar with the workings of an algorithm. Some alternative solutions require programmers to learn a completely new programming language. Although a new language, once learned, may be more efficient than C++, RapidMind recognizes the inertia of the software-development community. By extending familiar concepts in C++, RapidMind lowers the learning curve and reduces the anxiety of adopting something new.

Hardware abstraction is a critical part of RapidMind's technology. It allows parallel code written today for a dual-core processor to run tomorrow on a quad-core processor—

RapidMind Wins HPCwire Awards at SC07 Conference

In mid-November, RapidMind won two awards at the 2007 International Conference for High-Performance Computing, Networking, Storage, and Analysis (SC07) in Reno, Nevada. RapidMind won the *HPCwire* Editors' Choice Award for the Most Significant New HPC Software Product and was named one of the Top Five Companies to Watch in 2008.

HPCwire is a popular online publication covering high-performance computing. The HPCwire Readers' and Editors' Choice Awards are determined by online polling and by a selection process involving HPCwire editors and industry luminaries. For more information about the awards, see:

- *www.hpcwire.com/hpc/1885812.html*

and someday on an eight-core processor and beyond. That's huge, because a programming model that binds source code to the number of cores imposes built-in obsolescence. Java, too, relies on hardware abstraction, but writing parallel code in Java requires programmers to manually create and manage threads. Performance depends on the number of threads and how effectively the Java virtual machine maps threads to cores. RapidMind's platform lifts the burdens (and dangers) of threading from the programmer's shoulders.

One potential drawback of using RapidMind's technology is the unexpected longevity of program code. Once written, debugged, and proven in the field, code tends to live a long time—often much longer than the original developers intended. Although the Y2K scare of the late 1990s was largely overblown, it provided a surprising windfall of post-retirement income for Cobol programmers. Code written for RapidMind's platform will remain viable for only as long as RapidMind maintains the platform. If RapidMind disappears next year, what will customers do? A similar question haunts PeakStream's early adopters, after that company's acquisition by Google.

Price & Availability

RapidMind's Multicore Development Platform v3.0 is available now for x86 processors. It also supports IBM's Cell BE and graphics processors from AMD/ATI and Nvidia, including AMD's new FireStream 9170. The RapidMind platform includes a linkable function library, enhancements to C++, and a run-time module for the target microprocessor architecture. Pricing is based on the number of processors or sockets on which customers deploy the platform, but RapidMind doesn't publicly disclose specific prices. For more information, follow these links:

General information:

- *www.rapidmind.com*

White paper, "Data-Parallel Programming on the Cell BE and the GPU using the RapidMind Development Platform," by Dr. Michael D. McCool, cofounder and chief scientist:

- *www.rapidmind.com/pdfs/WPdprm.pdf*

Joint white paper by Hewlett-Packard and RapidMind, "Performance Evaluation of GPUs using the RapidMind Development Platform":

- *www.rapidmind.net/case-gpu.php*

Other case studies:

- *www.rapidmind.net/case-studies.php*

One solution is for RapidMind to make its platform as solidly entrenched as Java. But as a small startup, RapidMind is nowhere near as influential as Sun Microsystems. Another solution might be an open-source version of the platform, perhaps placed in escrow, should the need arise.

Naturally, early adoption of any technology has risks. Sometimes, the advantages outweigh the risks, even if the advantages are short-lived. Judged on its technical merits, the RapidMind platform definitely has advantages. It's a modern, intelligent model for parallel processing. That model will probably survive in one form or another, regardless of RapidMind's fortunes. ◇