MICHAEL D. MCCOOL

achieving high performance by targeting multiple parallelism mechanisms

Michael McCool is an Associate Professor at the University of Waterloo and is also a co-founder and Chief Scientist of RapidMind, Inc., which produces a unified development platform for multicore processors and many-core accelerators. His background includes experience with and research publications in real-time computer graphics, medical imaging, hardware design, parallel computing, compiler design and implementation, and mathematics.

Michael.McCool@rapidmind.com

RECENTLY, PROCESSOR VENDORS HAVE begun increasing performance by adding additional cores, rather than increasing the performance of a single core. The addition of multiple cores augments several other parallel hardware mechanisms already in place on each core. These features create the potential for increased performance, but only if they are properly utilized. Programmers who disregard the underlying design of the hardware in newer processors can actually produce code that runs slower on a multicore processor. In this article, I explain what these design features are. I also discuss the underlying memory models and the impact they have on processing in a multicore context. Finally, I present an example of a method of writing abstract parallel code that allows a development platform to do the heavy lifting of implementing code for different processor architectures.

The trend toward increased hardware parallelism results from several factors, but basically it is not possible to scale clock rate owing to the excessive power required, because power requirements grow nonlinearly with clock rate. It is also simply not cost-effective to use the very large number of transistors that can fit on a modern chip for only one core. There just isn't enough to do, and it takes too long to get signals from one side of the chip to the other.

Several recent multicore processor designs have also used heterogeneous cores, in which some cores are tuned for specific tasks or workloads. In particular, not all processors have to be able to run the operating system and the user interface; they can instead be specialized for high-performance computation. At their targeted workloads, specialized cores can be orders of magnitude more efficient in terms of space and power than general cores. This is the case with the Cell BE processor and with GPUs, the processors found in video accelerator cards, for example. These processors also use a relatively large number of cores (since each core is simpler) and often provide more direct control over on-board memory, which is also a major factor in performance.

There are two major factors to consider when targeting high performance: parallelism and memory access. First, hardware is naturally parallel, and multicore processors make this painfully obvious. However, as we will discuss, there are in fact *many* hardware mechanisms already available besides multiple cores that exploit parallelism, and the performance advantages of these are multiplicative. To get the most out of modern processors and have any hope of running well on future processors a massively parallel approach needs to be considered from the start. Second, memory access can easily become a bottleneck even on single-core processors and the problem is even worse on multicore processors. To achieve even adequate performance on modern processors, programming practices and data structures need to be compatible with the structure of the memory system. Certain naive programming practices that conflict with the memory system can easily drop performance by one or even two orders of magnitude and can make it impossible to scale to a large number of cores.

Parallelism

Modern computer systems and processors actually use several forms of parallelism internally, in addition to multiple cores. To achieve maximum performance, it is often necessary to target several of these forms of parallelism simultaneously. This can be accomplished by designing parallel algorithms in an abstract form first. Once a good parallel algorithm has been designed, the abstract or "latent" parallelism in the algorithm needs to be decomposed and mapped onto the concrete parallelism mechanisms available in the target hardware.

To understand this better, let's review the various forms of parallelism supported in modern processors and summarize how to take advantage of them.

MULTIPLE CORES

The most obvious form of parallelism available in modern processors consists of the multiple cores, of course. Every core is capable of executing independent instruction streams. These cores may or may not share a common memory subsystem. To make use of multiple cores, a workload needs to be decomposed into multiple components and each component run on the various cores. It is desirable to break the workload into equal-sized pieces so that the cores are evenly loaded; otherwise some cores will finish early and have to wait for the slowest core to complete. It may also be necessary to coordinate work among the cores, so that access to shared data is done in the right order. Because the number of cores can vary and is also increasing over time, an approach to decomposing the work that is adaptable to different numbers of cores is desirable. Data parallelism, which drives the decomposition by the structure of the data, is one approach that can accomplish this. An alternative way to achieve parallelism is to use decomposition by task, for instance, mapping different software modules onto different cores, although usually there are a limited number of different tasks available. These two approaches can be combined.

MULTIPLE PROCESSORS

When multiple processors are placed in a system, the number of available cores is the sum of the cores in all the processors. It is necessary to auto-

matically distribute the work over all the available cores, even if they are in separate processors. In addition, specific banks of the memory may also be associated with specific processors, and in this case accessing the memory associated with a specific processor will be more efficient from that processor. This property is called Non-Uniform Memory Access (or NUMA for short). For the best performance, it is useful to preferentially assign work units to processors closest to the memory banks where the needed data is located. This can be controlled by using processor affinity, which allows particular threads to be preferentially run on particular cores (although one has to be careful about the core numbering, since the mapping to physical cores and processors varies among vendors).

VECTOR INSTRUCTIONS

Many processors have special vector instructions that can operate on multiple elements of data at once. These are also called Single-Instruction Multiple Data (SIMD) operations. For instance, a processor may be able to apply a single arithmetic operation to 4-tuples of numbers, and that operation will take place in parallel on each element of the 4-tuple. A vector length of four is typical for single-precision floating point but it may be longer or shorter on different processors or for different data types. Examples of such instructions include Altivec instructions on the PowerPC and the SSE instructions on x86 processors. If these special instructions are not used the benefit of this form of parallelism will not be realized. Also, different processors, even within the same "family," may support different instruction set extensions. In particular, there are several generations of SSE instruction set extensions on x86 processors.

Instruction Pipelining

Many operations, in particular floating-point operations, may take multiple clock cycles to complete. The hardware breaks such operations into several stages, like an assembly line. For example, consider a floating-point addition. This is a surprisingly complex operation. Floating-point numbers are represented as in binary scientific notation, with both an exponent and a mantissa. To add two floating-point numbers, it is necessary to (1) compute the difference of the two exponents, (2) shift the mantissa of the smallest value down by this difference to align the "binary" point, (3) add the aligned mantissas, (4) shift the result mantissa so that it is in normalized form (with a leading 1), (5) round the result, and (6) renormalize the result (shifting down by one bit) if the highest bit was rounded up. This process can be implemented with separate hardware units for each step, with one unit feeding its result to the next on every clock pulse. As in an assembly line, several "jobs" (instructions, in this case) can be in the pipeline at the same time, as all the stages can operate in parallel. However, if an instruction depends on a previous result, then that instruction cannot begin until the result of the previous instruction is available. To keep the pipeline operating at maximum efficiency, there must be a large number of independent instructions available. If independent parallel tasks are available, they can be interleaved to avoid dependencies among instructions.

SUPERSCALAR INSTRUCTION ISSUE

Many processors can also start ("issue") multiple instructions in the same clock cycle, as long as they do not depend on each other or use the same

hardware resources. For example, it might be possible to issue an instruction for an integer multiply and a floating-point addition in the same cycle, since they use separate hardware resources (with one using the integer multiplier and the other the floating-point adder).

Some processors will automatically issue multiple instructions simultaneously whenever possible. This is typical of mainstream desktop and server CPUs, which often have two-way or four-way superscalar instruction issue. Long instruction words may also be used to explicitly specify multiple operations at once. The latter approach is called a Very Long Instruction Word (VLIW) architecture. Current ATI/AMD GPUs are examples of the VLIW architecture, in which every core can issue five floating-point operations and one branch operation in every instruction. The Cell BE SPE cores can also be considered to have a VLIW architecture: Each instruction "pair" can issue one four-way SIMD floating-point operation and one integer, branch, or load/store operation in parallel.

As with pipelining, latent parallelism in an algorithm specification can be used to create independent instruction streams to make best use of this hardware feature.

ASYNCHRONOUS MEMORY TRANSFERS

Data can typically be transferred in and out of on-chip memory in parallel with computation, as long as the computation does not depend on the result of the transfer. This can be used to hide the latency of memory transfer. Different processors have different mechanisms for this; on CPUs, cache prefetching instructions are used. Prefetch instructions indicate that the contents of a given memory address in DRAM should be copied into cache in advance of when it will be used. On GPUs and the Cell, DMA transfers must be specified explicitly to move data between on-chip memory and external DRAM. In either case, to exploit this form of parallelism, the need for the data stored in a given memory location must be anticipated.

SIMULTANEOUS MULTITHREADING (HYPERTHREADING)

Some processors are able to run multiple threads on a single core. These additional threads look as though they are running on two or more "virtual" cores per real, physical core. In many ways, this can be considered an alternative interface to some of the other mechanisms for hardware parallelism already noted. Sometimes these threads are used to generate additional instructions for superscalar issue; sometimes the processor time-slices between the threads or switches between the threads on a memory stall in order to hide latency when data needed by a particular thread needs to be fetched from main memory. It is important to understand that simultaneous multithreading has very different performance characteristics from true multicore threading: It is usually a mechanism for sharing virtualized resources, not for accessing additional resources. It is important, therefore, to understand how processor affinities map threads to both real cores and "virtual" cores. Many times, if the code is carefully scheduled to use pipelining and superscalar issue, and to use prefetching, then multithreading on one core may not add any additional benefit. However, if each thread has a lot of control flow, it can be harder to schedule pipelined and superscalar code explicitly, and in this case multithreading on one core can be beneficial.

Accelerators, which are additional non-CPU co-processors often with their own dedicated memory, such as GPUs, can execute a computation in parallel with the host CPU. If an operation is invoked that targets an accelerator, it is possible to start that operation asynchronously. Control can then be returned to the host program immediately even if the computation on the accelerator is not yet complete. The host process may then continue with additional operations that can execute in parallel with the computation running on the accelerator. However, if the host tries to read the result generated by an accelerator operation still in progress, the host process must wait until the accelerated operation is complete.

Memory

Multicore processors put a high demand on the memory system, and if care is not taken to use the memory system carefully, it can quickly become a bottleneck. The memory system consists of multiple types and forms of memory with different performance characteristics. The most important distinction is between on-chip and off-chip memory. On-chip memory is small but very fast, whereas off-chip memory (typically implemented using DRAM) is high capacity but slow. The number of clock cycles needed to read a data element from memory is called its *latency*. On-chip memory typically have single-digit latencies. Off-chip memory can have hundreds of cycles of latency. Bandwidth is often much higher to on-chip memory as well.

Typically a core can only operate at full speed when operating out of on-chip memory, which has a severely restricted capacity. Therefore on-chip memory is a critical resource and needs to be carefully managed.

Different processors take different approaches to managing on-chip memory. Caches are an automatic approach that makes management of the on-chip memory functionally invisible to the programmer. This is the approach taken by most general-purpose processors. However, the programmer still should take certain steps to make sure the cache performs well. In many cases, more efficiency can be gained if the programmer has direct access to and control of the on-chip memory, since then the use of this critical resource can be adapted to a specific application. This is the approach taken by the Cell BE processor in its specialized high-performance SPU cores: Each SPU core (out of eight total) has 256 kB of dedicated on-chip memory, and data must be explicitly transferred to and from external DRAM.

Cache is a small, fast, usually on-chip memory in which copies of frequently used data are stored temporarily. In fact, there is typically a cache hierarchy, with very small, very fast cache memories right next to the processor that are actually caching data from another, slower and bigger cache lower in the hierarchy. Modern multicore processors can have up to three levels of cache, and data is moved between them automatically in response to the memory access patterns of the running program.

The purpose of cache is to reduce memory access latency *on average*. Reading a data item from off-chip DRAM takes, from the processor's point of view, hundreds of cycles. It will take only a few cycles to read that same data from cache. On every memory access, the processor checks whether a copy of the needed data is in the cache. If it is not, then it must wait until a copy of the

appropriate memory item can be read from a lower, slower level of memory, ultimately from off-chip DRAM. If such *cache misses* happen very infrequently, then on average, the memory access latency is closer to the time to read from the cache than to read from DRAM. Data is also transferred in relatively large blocks (on the order of hundreds of bytes) from DRAM, to amortize the overhead of setting up a memory transaction. A cache miss is only taken on the first access to a block. Later accesses to the same block will find the data already in the cache.

Eventually the cache fills up and blocks have to be replaced when space is needed to handle a new cache miss. If the block to be overwritten has been modified, it needs to be written back to main memory. Also, the hardware needs to select which block to discard. This is done by some simple rule; for instance, the block that has not been accessed for the longest time might be the one replaced.

Unfortunately, certain programming practices can defeat the cache, and cache may also not benefit some applications.

First, if only one element is ever read from every cache block loaded, then the cache is useless. In this case prefetching should be used to hide the memory access latency. Prefetching allows the processor to request a cache block sometime in advance of actually using it.

Second, as noted, data is actually transferred in blocks from main memory. If one element in a block is touched, the whole block is brought into the cache. If other nearby items in the same block are used by the program—a property called *spatial coherence*—then additional cache misses can be avoided. If they are not, then the bandwidth for transferring the rest of the block has been wasted. Therefore, programmers should select algorithms with good spatial coherence. Unfortunately, typical data structures based on pointers between many small memory records are not very good for cache performance. Pointer chasing leads to a lot of jumping around in memory and often results in poor spatial coherence.

Third, the processor has to be able to quickly check if data is in the cache. The hardware structure for this only allows a few locations in the cache to be used to hold copies of a large set of elements in main memory. Typically the locations of the elements in this set are offset by powers of two. If repeated accesses are made to the elements in the same set, they will fight over a very limited set of slots in the cache, a situation called *cache conflict*. The resulting *cache thrashing*, where items repeatedly replace one another, essentially disables the cache and can severely degrade performance.

Finally, if writes are made to data stored in cache, this data needs to be written back to DRAM eventually. Complications can arise if two cores with separate caches write to the same block of memory, or if one tries to write to a block another core is reading from. To maintain the illusion of a single unified memory space, these cores then have to keep track of which processor has the most up-to-date copy of the block. This involves a lot of hidden interprocessor communication, which can degrade performance. Some cache coherency protocols give one core ownership of a block, and only the owner may write to a block. However, if two cores simultaneously try to work on the same block, they can end up fighting over who owns it, with disastrous results for performance. This may occur even if the cores (or processors) are actually trying to modify different locations in the same block, a situation called *false sharing.*

These issues with cache are made more severe by multicore processors. There are additional levels of cache to worry about, and the aforementioned

effects can occur at one or all levels. Issues such as cache coherency and false sharing only arise in systems with multiple cores or processors. With the advent of multicore processors, off-chip memory bandwidth is not likely to grow as rapidly as on-chip computational performance, so off-chip bandwidth is even more likely to be a bottleneck. Finally, if a thread is suspended and restarted on a different core or processor, it will have to reload all its data into the cache on that core, possibly displacing data used by another thread. Yet another form of thrashing can take place between threads if together they need more data than will fit in the cache.

To avoid these issues, several steps need to be taken by the programmer. First, data should be allocated aligned to cache boundaries, and nodes of data structures should be padded if necessary to align to cache boundaries. This may waste some memory space but will avoid false sharing. Also, data structures that have good spatial coherence should be chosen over those with an excessive number of pointers. For example, a B-tree is often better than a simple binary tree, since a B-tree uses large, fixed-size blocks internally (which can be aligned to cache boundaries) and has a shorter number of pointer jumps from the root to its leaves. Finally, offsets between data elements that are a power of two should be avoided if possible. In image processing and matrix operations, for example, power-of-two tile sizes should be avoided by padding row lengths as necessary, because access to elements in adjacent rows may accidentally cause a cache conflict. Unfortunately, exactly what powers of two cause trouble and what alignments are needed vary by processor and the cache structure it uses. Also, avoiding large power-of-two offsets to avoid cache conflicts can be at odds with the desire to align to small powers of two for cache blocking. Some odd multiple of the cache block alignment should be selected.

EXPLICITLY MANAGED MEMORY

Cache is automatic, which is useful for naive code. However, to avoid the many issues that caches raise in multicore systems, some processors have opted for explicitly managed local memory. This is the case with the Cell BE processor, and also to some extent with GPUs (although current NVIDIA GPUs actually have both cache hardware *and* explicitly managed local memory).

In the Cell BE processor, each core gets a dedicated local memory. A separate Memory Flow Controller (MFC) can be programmed to transfer data to and from DRAM to this local memory, and also to and from other local memories on the same chip. These transfers can take place in parallel with computation.

A cache can still be simulated in software on such an architecture. Although slightly slower than a hardware cache, a software cache can be sized and tuned to the properties of the data structure it is caching. In particular, a block size and replacement policy can be chosen that are most suitable for the access patterns and data structures used.

Programming

We have now summarized the main hardware mechanisms available for exploiting parallelism in modern processors and also the properties of the memory system. It should be clear at this point that there is a lot "under the hood."

Unfortunately, programming at this level of detail is very challenging, and consequently it is rarely done. Also, portable software may not be able to exploit a particular hardware feature, such as SIMD instructions, that is not consistently implemented on all hardware targets. As a result, most portable software is relatively inefficient.

The other point worth noting is that threading only targets a few of the levels of parallelism noted, and if not properly managed it can lead to inefficiencies in the memory system. Throwing a large number of threads at a multicore system and letting them fight over resources is unlikely to produce optimal results. Instead, a thread should just be seen as a mechanism for getting access to a single core, and then on that core appropriate steps should be taken to manage the memory and exploit the other forms of parallelism available. Steps should also be taken to avoid moving threads between cores (to avoid cache thrashing) and to keep computations close to the memory banks they are accessing in NUMA systems.

There are now several software development platforms that seek to reduce the complexity of programming multicore systems. The fundamental observation of these systems is that there are actually only two key abstract design principles that need to be targeted: parallelism and data locality. In particular, many *mechanisms* for implementing parallelism in hardware are available, but if a large amount of latent parallelism is available at an abstract level, it is not necessary for a programmer to target each mechanism individually. Instead, it is possible for a semiautomated system to map an abstract, portable programming model to whatever is available. Likewise, if an interface is provided in which the programmer can express an abstract version of data locality, then it can be mapped onto what the physical memory hardware requires.

To make this more concrete, we can look at an example from the RapidMind platform, which does just this. RapidMind is based on three types that can be used within standard C++, using existing compilers: values, arrays, and programs. A value represents a scalar type (e.g., a number or Boolean), arrays manage collections of data, and programs manage code. A sequence of operations on values can be stored in a program, then applied to a collection of data stored in an array.

First, we will declare some one-dimensional arrays to hold the data:

```
Array<1,Value1f> A, B;
```

We won't bother sizing or filling these arrays with data here, although in a real application this would have to be done.

Now we will construct a really simple example program to increment a value:

```
Program p = BEGIN {
        In<Value1f> a;
        Out<Value1f> b;
        b = a + 1.0f;
} END;
```

In a real application, such programs might contain thousands of operations and might include control flow, declarations of temporary variables (including local arrays), random accesses into other arrays, any number of inputs and outputs, and calls to C++ functions and other RapidMind programs. RapidMind programs can be thought of as dynamically constructed functions, for the most part.

Finally, we can apply the program to one of these arrays:

    B = p(A);

This will apply the program to all the elements in A and place the result in B. As it happens, this will execute in parallel.

Applying a function to an array is a very simple way of invoking a parallel computation, conceptually. But what really goes on in the platform to execute this operation efficiently, given everything that we have discussed so far?

Conceptually, the parallelism intrinsic to this example is of the form shown in Figure 1.
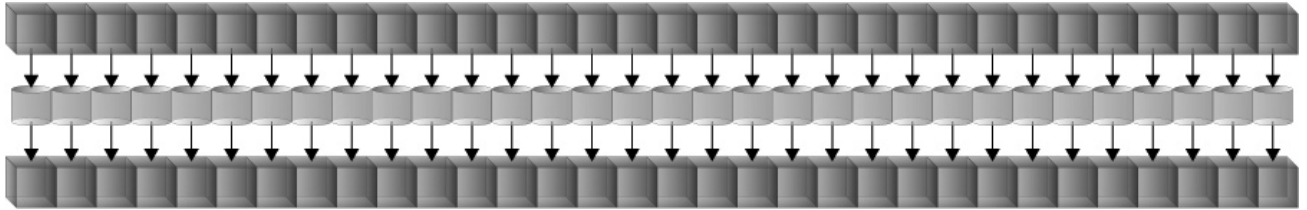
FIGURE 1: THE ABSTRACT PATTERN OF LATENT PARALLELISM SPECIFIED IN THE EXAMPLE

The important thing is that the semantics of program application provides a large amount of latent parallelism but has *not constrained the order in which these operations can be done or how they can be grouped*. Therefore the code generator and runtime system are free to reorganize them in any way that makes sense. For example, suppose we are targeting a two-core machine with a pipelined floating-point unit, four-way SIMD instructions, and two cores. The platform could then automatically organize this same computation as shown in Figure 2.
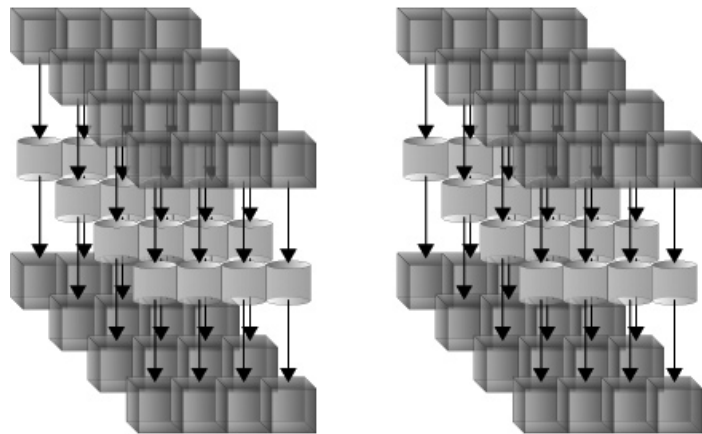
FIGURE 2: A COMBINATION OF CONCRETE PARALLEL MECHANISMS, INCLUDING SIMD INSTRUCTIONS, PIPELINING, AND MULTICORE EXECUTION, THAT COULD BE USED TO EXPLOIT THE LATENT PARALLELISM SPECIFIED IN THE EXAMPLE

Of course, if the hardware target changes, the code might have to be reorganized in a different way. For instance, a target with more cores, or a different SIMD width, might require a different decomposition. However, the code is portable, since the programmer has *not* constrained the computation to any particular ordering or decomposition. The code given here, for example, runs on various flavors of x86 multicore processors, the Cell BE SPUs, and

GPUs without change. Memory optimizations can also be made. The platform will break the work into blocks and prefetch one block into on-chip memory while working on another, work units can be broken into tiles that are suitable for the memory architecture and cache alignment, and arrays can be allocated with appropriate alignments and padding to avoid cache conflicts and false sharing. More complex code would require more complex transformations and management (e.g., control flow inside programs requires load balancing), but the same general principles apply.

Conclusion

Multicore processors are complex, but this complexity is in the form of several mechanisms that all fundamentally depend on two things: parallelism and data locality. It is possible to abstract away the complexity of multicore processors and still achieve high performance if abstractions are chosen that allow the programmer to focus on structuring computations around these two main concepts while not overconstraining the implementation. It is then possible to automatically reorganize the computation to exploit the various parallelism mechanisms available and optimize it for good memory behavior.

ADDITIONAL READING

The Editor suggests additional reading from past *;login:* articles:

[1] "Algorithms for the 21st Century," by Steve Johnson: www.usenix.org/publications/login/2006-10/openpdfs/johnson.pdf.

[2] "Multi-Core Processors Are Here," by Richard McDougall and James Loudon: www.usenix.org/publications/login/2006-10/pdfs/mcdougall.pdf.

[3] "Some Types of Memory Are More Equal Than Others," by Diomedis Spinellis: www.usenix.org/publications/login/2006-04/pdfs/spinellis.pdf.