INVITED PAPER

Scalable Programming Models for Massively Multicore Processors

Programming systems based fundamentally on massive parallelism and data locality promise to make efficient use of multicore processors.

By Michael D. McCool, *Member IEEE*

ABSTRACT | Including multiple cores on a single chip has become the dominant mechanism for scaling processor performance. Exponential growth in the number of cores on a single processor is expected to lead in a short time to mainstream computers with hundreds of cores. Scalable implementations of parallel algorithms will be necessary in order to achieve improved single-application performance on such processors. In addition, memory access will continue to be an important limiting factor on achieving performance, and heterogeneous systems may make use of cores with varying capabilities and performance characteristics. An appropriate programming model can address scalability and can expose data locality while making it possible to migrate application code between processors with different parallel architectures and variable numbers and kinds of cores. We survey and evaluate a range of multicore processor architectures and programming models with a focus on GPUs and the Cell BE processor. These processors have a large number of cores and are available to consumers today, but the scalable programming models developed for them are also applicable to current and future multicore CPUs.

KEYWORDS | Computer architecture; multicore processors; parallel programming and computation; programming and processing models

Manuscript received June 1, 2007; revised December 12, 2007.
The author is with RapidMind Inc., Waterloo, ON N2J 1P8, Canada, and the David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, ON N2L 3G1, Canada (e-mail: michael.mccool@rapidmind.com; mmccool@uwaterloo.ca).

Digital Object Identifier: 10.1109/JPROC.2008.917731

I. INTRODUCTION

For more than 30 years, most programmers have been using a serial model of computation to design and implement programs. However, digital hardware is naturally parallel. In order to obtain increased performance, compiler and processor designers have struggled to automatically exploit implicit instruction-level parallelism (ILP) in serial code. For example, instructions can be rescheduled, either by the compiler or by the hardware, in order to best exploit pipeline parallelism in functional units or multiple instruction issue in superscalar processors.

However, experience has shown that most serial programs have limited implicit parallelism [1]. If techniques such as speculation [2] and autovectorization mechanisms that can exploit predicated assignment [3], [4] are used, this limit can be raised somewhat, but only at the cost of increased power/performance ratios and hardware complexity. Automatic extraction of parallelism without explicit assistance from the programmer, particularly in hardware but also in software, has reached the point of diminishing returns. Therefore, there is renewed interest in explicitly parallel programming models, languages, runtimes, and platforms.

As processors scale up to billions of transistors, additional levels and forms of parallelism have been added, including short vector [single-instruction multiple-data (SIMD) parallelism within a register] instructions, simultaneous multithreading, very large instruction words (VLIW), and multiple cores. In combination with pipelining and multiple instruction issue, this means that even simple computers can support five to six different kinds of parallelism, without even considering the use of multiple processors or clustering. Traditional central processing units (CPUs) have recently begun adding multiple cores,

but more specialized processors, such as graphics processing units (GPUs) from AMD/ATI and NVIDIA and the Cell BE from IBM, are already using a large number of cores as well as other forms of parallelism. The discussion in this document focuses on programming models and systems developed for GPUs and the Cell BE. This is because while GPUs and the Cell BE are the first processors generally available to consumers with eight or more cores, CPUs are soon to follow. Therefore, the scalable programming models developed for these processors are potentially useful as a basis for generalization to the case of future "many-core" CPUs.

What is a core? We can define a core as a processing element with an independent flow of control. However, cores can vary in computational power and can also have various other kinds of internal parallelism. For instance, an SIMD processor might have hundreds of functional units but only one "core." Because of the many forms that parallelism can take, the most suitable measure of the available parallelism in a processor may be the total number of independent operations that can be issued on every clock cycle rather than the number of cores. By this measure, GPUs have the most available parallelism of any generally available processor: some currently available GPUs have hundreds of pipelined functional units that can each start a new single-precision floating-point operation on every clock. The NVIDIA GeForce 8800 GTX has 128 and the ATI 2900 has 320 such functional units. However, using the definition of cores stated above, these processors have "only" 16 and 4 cores, respectively, as the functional units are grouped into SIMD processing elements. In comparison, the Cell BE (with nine cores and one four-way functional unit in each) has 36 functional units, and recent quad-core CPUs have on the order of 16 to 20 functional units, depending on their microarchitecture.

In order to get the best performance out of processors with such a large number of functional units, an explicit mechanism allowing the programmer to directly express a parallel computation is desirable. It is useful to design such mechanisms around *programming models* [5]. A programming model is an abstract model of computation that is used by the programmer to reason about how a program executes. A *processing* model is similar but describes how a physical machine actually performs computation. Or rather, the processing model is the programming model exposed by the processor vendor and used in its instruction set architecture. The processing model is usually hardware or vendor specific and designed for simplicity and performance rather than convenience. The goal of any programming language implementation is to efficiently translate a computation expressed relative to the programming model used by the programmer into the processing model used by the target hardware.

A programming model is an abstraction and may not explicitly expose every parallelism mechanism available in the target processor. For the purposes of portability and simplicity, it may in fact not be desirable to do so, given the variety of forms parallelism can take and the detailed optimizations that may be required to exploit it efficiently. However, it is far easier to design a system that maps an explicitly parallel algorithm onto a parallel hardware realization, whatever the final mechanisms for expressing that parallelism, than to try to extract parallelism from oblivious serial code. With a parallel programming model, the programmer is encouraged to think in parallel and give the system a large amount of latent parallelism to work with.

In the ideal situation, the programming model expresses the most important aspects of the processing model and the programming language implementation simply automates the more tedious aspects of mapping the desired computation onto the target hardware. The programming language then performs the useful function of suppressing unnecessary detail and by doing so improves programmer productivity.

We argue that the most important aspects of the processing model are those that have the most impact on performance. A programming language implementation should unburden the programmer from dealing with trivia, allowing the programmer to focus on making major policy decisions and on designing an efficient (and correct) algorithm.

In general, in order to allow performance tuning, a programming model should provide a clear cost model. The programmer should be able to manipulate the expression of their program relative to the high-level programming model in order to improve the efficiency of the final implementation.

Unfortunately, the programming models of mainstream programming languages were developed a long time ago when computers were relatively simple. They do not, in general, expose the most important aspect of efficient computation on today's machines: parallelism. In addition to parallelism, there is also the issue of memory access. Poor use of the memory system can degrade the performance of a program by several orders of magnitude. Therefore, in addition to parallelism, a good programming model should also allow the clear expression of memory locality and data movement.

In this paper, we survey a number of parallel programming and processing models. We then discuss some of the available multicore processors in terms of these models. We conclude with a discussion of some recently developed explicitly parallel programming languages, tools, and programming platforms, comparing their programming models with the processing models of the hardware they target. To limit the scope of this paper, we have primarily focused on programming models developed for GPUs and the Cell BE because these are the consumer processors that currently support the most explicit parallelism. CPUs, however, are clearly evolving in the same direction [6]: towards massive parallelism and "many-core" architectures.

II. PROGRAMMING AND PROCESSING MODELS

Programming and processing models are important because these models are what a programmer uses to reason about how a computation actually takes place on a physical computer. Programmers use a programming model, and in particular the costs associated with various operations in the programming model, to reason about how a computer operates and to design efficient algorithms. If a programming model is an accurate reflection of how a computer actually performs a computation in hardware, and if the primary features of the model and the costs as understood by the programmer reflect the most important architectural characteristics of the computer, then it will be possible for a programmer to design efficient algorithms for the computer. If the programming model does not allow manipulation of some important feature of the processing model of the target hardware, or does not provide enough control over the final implementation on the real machine, then the programmer will be unable to optimize their code for the best performance.

Sometimes the programming model used by a programming language is quite different from the processing model of the machine on which it eventually runs. For example, functional languages use term graph rewriting or lambda calculus as their programming model. These models are quite different from the low-level processing models exposed by the instruction set of most computers, which are typically based on an imperative model of computation. Efficient implementation of functional languages on machines (even parallel machines) with an imperative processing model is possible but relatively indirect (see [7] for a detailed description in the case of the term graph rewriting programming model). Even so, if the programmer understands these transformations, then it is still possible for that programmer to write efficient programs. For example, a good functional programmer understands that their compiler will transform tail recursion (a special case of a feature of functional programming models) into iteration (a feature of the underlying machine's imperative processing model). Since iteration is more efficient on the real machine than recursion, good functional programmers will use tail recursion when they are trying to maximize performance.

Understanding nontrivial transformations between programming and processing models can be very difficult for nonexperts, and so generally for high-performance computation we would like the programming and the processing models to be as similar as possible, within the constraints imposed by the desire for portability. In addition to being as similar as possible to the processing models of the real machines it targets, a good programming model must have a few other properties: it should be expressive, simple, and safe [8].

First, a programming model should be expressive, that is, capable of concisely expressing understandable solutions to problems in a suitable target domain. General-purpose programming models should be able to express efficient solutions to arbitrary computational problems; that is, their target domain should include the largest possible space of possible application areas. Special-purpose programming models are possible and may be useful in some cases, but normally a general-purpose model is desirable. By efficient we mean that the computational complexity (in both space and time) of an implementation generated from a given programming model should not be significantly worse than that of any other implementation with another programming model.

In the context of parallel machines, there are two kinds of time complexity: step complexity and work complexity [9]. Step complexity is the number of parallel operations required to complete a computation given an infinite number of processors. Work complexity is the total number of operations on all processors. An ideal parallel algorithm would have the same work complexity as a serial algorithm solving the same problem. However, it is not possible even in theory to solve certain problems in parallel with the same work complexity as a serial algorithm, even though the step complexity might be lower. A problem may also not have a known solution with a step complexity lower than the time complexity of the best serial algorithm: an example is depth-first labeling of a tree [10]. Such problems are considered *intrinsically serial*. In the context of parallel programming models, an efficient programming model therefore allows the expression of algorithms with an actual work and step complexity that is equivalent to the same measures under any other programming model.

Secondly, a programming model should not be unnecessarily complicated. A complicated programming model can be hard to learn and may make the development process less efficient. We want to reduce unnecessary detail while still exposing the most important costs in the processing model to the programmer. This leads to a natural prioritization of the features that should be exposed in a programming model: by their costs. Basically, the most costly operations in the processing model should be exposed directly to the programmer in the programming model. This lets the programmer make high-level policy decisions that will lead to improved performance.

Thirdly, we want the programming model to be safe, that is, protect the programmer from the most common and expensive mistakes. If it is impossible to even express a flawed program in a programming model, there are obvious benefits in development time, quality, and maintainability. An example in serial programming models is garbage collection. Programming languages that support garbage collection, such as Java, protect programmers from large classes of memory management errors common in

other languages, and therefore remove a significant burden from the programmer.

In the case of parallel programming, synchronizing and coordinating multiple tasks is a common source of errors. In addition, these types of errors are difficult to locate and resolve after the fact, so a programming model that avoids them "by construction" is very desirable. An appropriate programming model can protect programmers from the new classes of errors particular to parallel programming (such as race conditions and deadlock) while still allowing programmers (and in fact encouraging them) to write efficient code.

In the following, we will discuss processing and programming models for *parallel* computers. This actually includes a large class of possible computing devices, including clusters and multicore CPUs. It also includes heterogeneous multicore machines such as the Cell BE and computers with a coprocessor or other accelerator, such as a GPU. We will consider the relationship of processing models to real machines and of programming models to processing models. We will also discuss the prioritization of features in parallel programming models. There are several processing models for parallel computers. The main two categories of processing models are based on task decomposition and data decomposition. We will review these first and their general characteristics, then survey at a general level computer architectures for implementing them. We will then survey a number of programming systems and relate their programming models to the underlying processing models of the machines they target.

A. Task Parallel Processing Models

Task parallelism is based on the idea of decomposing a program into separate tasks and running these tasks at the same time on different processing elements. Task parallelism is sometimes called multiple instruction, multiple data (MIMD) computation [11], [12]. This refers to the fact that separate programs (instruction streams) operate on separate data streams in parallel. The terms SIMD and MIMD were originally introduced to refer to computer architectures. Here we are using them to refer to the *conceptual* architecture of the programming model as understood by the programmer.

In the MIMD model, it is necessary for tasks to communicate and (usually) synchronize with each other. Most commonly, communication happens in one of two ways: through message passing or shared memory regions. The message-passing model [13] can be used on a shared memory computer but is normally used on machines where each processor (or other computational element) has its own memory. Such computers are known as distributed memory machines and may take the form of a cluster of independent computers communicating over a network. However, message passing could also be used between cores on a single chip using an on-chip network.

Synchronization consists of constraints that order or coordinate tasks in time and can also take several forms, both implicit and explicit. In the message-passing model, messages can also be used as explicit synchronization primitives, since two communicating tasks can block until the message is both sent and received. A blocked task cannot make progress until the condition it is waiting for is satisfied. These are called synchronous messages, and in addition to their use as synchronization primitives, they also have the advantage that they use a finite amount of memory. However, asynchronous messages are also possible if a variable-length message queue is used. In the shared memory model, locks supported by the memory system are used as synchronization mechanisms to properly order modifications to shared memory locations. A lock allows only one task at a time to access a shared region of memory. A lock can only be acquired by one task at a time. If another task tries to acquire a lock currently held by a different task, it will block until the lock is released. Barriers are another common form of synchronization. Tasks reaching a barrier block until all tasks in a given set of tasks also arrive at the same barrier. Barriers can be useful to order phases of computation and can be used in both shared and distributed-memory systems. Synchronization can also be implicit; for example, a programming model might automatically order tasks by data dependencies.

Correct use of explicit synchronization mechanisms can often be difficult, and errors in synchronization can lead to subtle and difficult-to-find bugs. For example, in the message passing model, if a task waits for a message to arrive that is never sent, or uses a blocking send to send a message that is never read, it can block forever. With locks, if two tasks acquire locks to the same set of resources in different orders, it is possible for both tasks to simultaneously obtain a lock required by the other, leading both to block forever, a situation called *deadlock*. On the other hand, if shared data are accessed without locks, incorrect behavior can result. For example, if two tasks try to update a value, they have to first read the old value, modify it, and write it back. It is possible for one task to try to update a value only to have its update overwritten by another task simultaneously trying to update the same value. These kinds of timing-dependent bugs, generally known as *race conditions*, can be especially difficult to find since they depend on the exact timing of the execution of two tasks, which may vary from run to run. Such bugs may cause a problem only once in a million runs, making them nearly impossible to detect with normal testing mechanisms. Unfortunately, adding more locks to avoid race conditions is not an optimal solution. Overly conservative use of locks can lead to scalability problems. Specifically, in the worst case, if multiple threads attempt to lock a single resource they can be forced to run in a serial fashion. Techniques have been developed to avoid locks in certain circumstances [14], but application of these techniques can be challenging.

Transactional memory architectures have been proposed to allow scalability of atomic regions without the need for blocking [15], [16], but one question that affects portability is the number of operations that can be contained in a single transaction. It is possible to virtualize transactions so that any number of operations can be supported [17], but hardware support is required to make this efficient.

Finally, even correct task parallel programs can be nondeterministic, that is, they can return different (but correct) results each time they are run. Such nondeterminism can pose a serious problem for automated testing. It is necessary to write another program to prove that the output is correct rather than simply comparing against a single known good solution. In many application areas, determinism is also an end-user requirement, for example, in financial simulation or in computer graphics simulation for film. These are both areas where exact repeatability is important.

The shared memory model that locks are based on has other problems. While being relatively simple for a programmer to understand, shared memory is actually quite difficult to implement in hardware. In other words, shared memory is actually a simplified communication model that hides some important costs.

Most, but not all, massively parallel processors include cache memory for every core [18]. A cache keeps copies of data in faster on-chip memory in an attempt to exploit spatial or temporal locality and to avoid having to access the main memory repeatedly for the same data. In a serial processor, from the processor's point of view, reads and writes to the cache have the same semantics that reads and writes to the main memory would have. However, in a multiprocessor (or multicore) machine with multiple caches, caches can delay writes to main memory. A cache-coherency protocol is needed to maintain a consistent view of the contents of main memory by all processors. For example, if one processor writes to a location in memory but a copy of this location is being kept in the caches of other processors, these copies need to be updated or invalidated. In general, cache coherency protocols can require significant communication between processors and can inhibit scalability. In addition, sometimes the cache coherency communication is spurious. Caches do not hold single words but large blocks of data. A write to part of a block can invalidate the block in other caches, even if the part of the block updated is not actually being read by the other processor. Processors can end up "fighting" over ownership of a single cache block needlessly. This is called *false sharing* and can be avoided with careful data layout that aligns with the cache block boundaries, but this complicates programming and is also machine-dependent.

In order to support cache coherency, processors must send messages to each other behind the scenes to maintain a consistent model of the contents of the shared memory. In fact, at the lowest level, the hardware in large shared memory machines is almost always based on a message passing processing model. As more and more processors and processor cores are added to a system, the overhead of maintaining the illusion of a single shared memory grows. A distributed memory model, such as that used by the message-passing programming model, is more realistic. It is, unfortunately, also harder to program.

Some high-performance processors avoid cache coherency in order to improve scalability, at the cost of a more complex distributed memory processing model, even on a single chip. Eight of the nine cores in the Cell BE, for example, do not support cache at all. They do support fast local memory, but transfers between this memory and the external main memory are explicit. The programmer (or the programming language implementation) is responsible for managing the use of local on-chip memory. While more complex, this approach is potentially more scalable. In GPUs, caches are supported, but there are separate read and write caches, and no guarantees are made that data written will be available immediately in a read from the same location unless specific steps are taken to flush the data from the write cache and invalidate the read cache. Generally, on a GPU, it is best to separate the computation into discrete passes and separate input and output memory locations for each pass. This also has the advantage of avoiding nondeterminism due to read-write hazards.

Scalability of the task decomposition approach is also a concern with the task-parallel programming model. It is difficult to find a large number of separate tasks in a typical program. This limits the number of separate processing elements that can be utilized. If a program is divided into a fixed number of tasks, it will only map efficiently onto machines with that number of processing elements. This is a concern since there is every indication that the number of processor cores in a machine will begin growing exponentially, as single-core performance plateaus.

Finally, tasks in a program are rarely all of the same computational difficulty. Assigning one task to each core can lead to load imbalance. A load imbalance is when one core has more work than another, so it is still running when the other tasks have already finished. This wastes the computational power of the underloaded cores.

B. Data Parallel Processing Models

Data parallelism is based on the idea that operations on collections of data, such as arrays, can themselves be performed in parallel. For example, to add two arrays together, it is possible to decompose the arrays into chunks and add the chunks in parallel on different processors. This approach has several advantages.

First, the number of parallel processes can be matched to the number of cores by simply decomposing the data into an appropriate number of chunks. The processing model is independent of the number of processors. In fact, the computation can be decomposed hierarchically, simultaneously taking advantage of other

forms of parallelism such as vectorization and pipelining but without burdening the programmer with the details.

Secondly, memory bandwidth is a serious bottleneck in modern computers. The data-parallel model, with its emphasis on data layout and management, can address the flow of data as well as the parallelization of computations. For example, data can be padded and aligned to avoid false sharing and other sources of inefficiency. It can also be streamed into caches using prefetching instructions in order to explicitly manage memory access latency. Again, this can be accomplished without burdening the programmer with the details.

Lastly, a data-parallel model is safe and conceptually simple. The necessary coordination and synchronization can be handled by the platform runtime rather than the programmer. Data-parallel programming models can be made deterministic, so it is impossible for the programmer to specify computations that would result in deadlock or suffer from race conditions, eliminating a major source of errors in parallel programs.

However, data-parallel processing models can have their own problems. Some styles of data-parallel processing perform too little work for each memory access and also are limited in expressiveness. There are also challenges in the implementation of control flow in some data-parallel programming models. However, there are several different processing models for data parallelism that attempt to address these problems.

In particular, the stream processing model uses computationally intense kernels rather than single instructions, so more work is done for every memory access. If these kernels include control flow and a suitable set of collective operations is also provided to express communication, there is evidence that the resulting processing model is in fact as powerful as the general task-parallel model, in the sense that large classes of problems can be solved using this model with the same work and step complexities as alternative implementations using task parallel models [9]. However, it is still possible to structure the resulting enhanced data-parallel model so it does not suffer from race conditions or nondeterminism.

In the following, we will review a number of forms of data-parallel processing, starting with the simplest.

SIMD Processing: SIMD parallelism is the simplest form of data parallelism in which a single constant-time operation (an instruction) is applied in parallel to all the elements of a collection of data. SIMD parallelism has many advantages from the standpoint of both hardware implementation and programming. From a hardware standpoint, the control unit for a processor does not need to be replicated, only the datapath, leading to a more efficient hardware implementation. From a programming standpoint, SIMD parallelism is a natural extension of serial programming, in that the control unit is still executing a sequence of operations, only each operation is parallel.

However, SIMD parallelism has an important problem: it has difficulty taking advantage of special cases to avoid work, or in dealing with variable amounts of computation in different parts of a problem. Specifically, it has poor support for the efficient implementation of control flow. To take two examples, a fluid flow simulation might require different and more expensive computations along boundaries, and a ray tracer may have to do more computation in complex parts of an image than in simpler parts.

Suppose a program implements a "common-case" set of code 99% of the time, which takes one unit of effort but 1% of the time, has to execute some "special-case" code that takes 100 units of effort. In a serial program, the average effort would be $1 \times .99 + 100 \times 0.01 = 1.99$ units of effort. Unfortunately, in the SIMD model, both the common case code and the special case code must be executed for every data element, resulting in an average of 101 units of effort. Similar problems arise in iterative algorithms. When looping, the computation for every data element must loop the same number of times, and algorithms cannot take advantage of data-dependent early exits.

In a serial program, branching is used to avoid such special-case work and to provide early exits. The semantics of branching can be emulated with predicated assignment [3], [4], [19]–[21]. However, predication cannot, in general, avoid doing extra work. It is possible to actually avoid work using an SIMD machine using filtering and packing (reordering the elements of an array based on a predicate to eliminate "useless" elements), but the packing operation can be quite expensive [22], [23]. In particular, packing can increase the overall work and step complexities of the implementation of some parallel algorithms compared to alternative approaches.

GPUs are often considered to be SIMD machines (in the past they actually were, to a first approximation), and techniques for emulating control flow have been rediscovered for them [24], [25]. However, current GPUs internally have generalized the SIMD model and are actually single-program multiple-data (SPMD) machines. They now use a tiled approach to SIMD computation, where each tile uses predication internally, but multiple tiles are processed simultaneously in MIMD style.

An example of a commercially available pure SIMD processor is the ClearSpeed accelerator,[1] which uses an array of 96 floating-point units in an SIMD array. Some important problems, such as linear algebra and fast Fourier transforms, can be implemented efficiently on such architectures. There are several cases where the SIMD model is used besides in special-purpose accelerators, so we review these next.

SWAR Parallelism: An important special case of SIMD parallelism is SIMD-within-a-register, sometimes called SWAR [26] or superword [4] parallelism. In this case, a

[1]http://www.clearspeed.com.

short n-tuple of numbers in a register, typically on the order of 128 bits (four single-precision floats), can be operated on in parallel. Many processor instruction sets have been augmented with SIMD instructions, the streaming SIMD extensions on the Intel and AMD processors and the Altivec instructions on the Power processor being examples.

GPUs and the Cell BE processor also support SWAR. In addition to supporting element-wise parallel operations, SWAR instruction sets may also include support for horizontal reduction operations (such as summation of the elements of a register), swizzling, and writemasking. Reduction operations combine all the elements of a tuple and may include summation, dot products, length computations, or logical operations across elements. Swizzling permits the permutation of the elements of a tuple and writemasking allows (possibly predicated) writes to a subset of the elements of an SIMD register.

Vector Processors: A vector processor is an implementation of SIMD parallelism where the data are streamed in and out of a set of (generally variable-length) storage locations. In a vector processor, the coherent access to the collection of arguments is exploited to achieve better memory performance.

Classic vector machines include the Cray-1, which in its day achieved very high peak performance using this mechanism. Vector processing differs from SWAR in that the vectors are often much longer than will fit in a typical register and may in fact be streamed in from off-chip memory. However, SWAR and vector computation are related and proposals have been made to extend the SWAR instructions in modern processors to support larger variable-length vectors [27].

As with other forms of SIMD processing, vector processing can efficiently achieve high peak performance for regular problems but cannot deal efficiently with data-dependent control flow without some enhancements.

In addition to computations, data access operations such as memory reads and writes can be expressed in vector form as "gather" and "scatter" *collective operations*. Collective operations such as reductions and scans are also useful. A reduction combines all the elements of a vector into one, similar to the horizontal operations under SWAR. An example of a reduction is the summation of all the elements of a vector, although in general any associative operation can be used to combine elements. A scan or "parallel prefix" operation is similar to a reduction but outputs a vector instead of a single element; every element of the output contains the partial reduction up to the corresponding position in the input vector. The scan operation is surprisingly useful for building and manipulating data structures. SIMD vector computation combined with a suitable set of collective operations is in fact capable of implementing a range of algorithms [9], and if segmented collectives are provided, nested parallelism can also be supported.

Stream Processing: Traditional vector processing has a problem: although it can take advantage of spatial coherency to stream data in and out of the processor, it still uses memory bandwidth relatively inefficiently. The input array has to be read into the processor and the result written out again, but only a small amount of computation (a single instruction) is done for (on average) every three memory accesses. The ratio between the amount of computation and the memory bandwidth consumed is called the *arithmetic intensity*. It is possible to compute the theoretical peak performance of an algorithm in operations per second by multiplying the arithmetic intensity (operations per memory access) by the memory bandwidth (memory accesses per second), assuming the system is memory-bound (which is often the case). In this case, for a fixed memory bandwidth, increasing the arithmetic intensity will directly increase performance.

Processing speed has been increasing much more rapidly than memory bandwidth. As a result, modern processors now have at least an order of magnitude more computational power than memory bandwidth. They require correspondingly high arithmetic intensity for efficient execution.

Because of its low arithmetic intensity, a "classic" vector processing model is a poor match to modern computer architectures and will not generally be efficient. It does, however, have some good points. In particular, it accesses memory in a very regular and predictable fashion. This is important for obtaining the peak bandwidth from a memory system, and several hardware optimizations are possible in this case.

The stream processing model is a refinement of the vector processing model that improves its arithmetic intensity. Like the vector processing model, it accesses memory in a very regular way, and so can maximize input and output memory bandwidth. However, stream processing also applies a computationally intense kernel to the input and output streams rather than a single operation. The kernel can contain a large number of instructions and can perform many operations for every memory access. Stream processing can achieve high arithmetic intensity, making it a good match to modern architectural constraints.

It should be noted here that it is possible to use a vector programming model but compile it to a stream processing model by automatically fusing vector operations into stream kernels. This is important since modern Fortran, for instance, has support for explicit data-parallel programming in the form of vector operations [28], [29]. However, since arithmetic intensity is an important determinant of performance, exposing the kernel structure to the programmer explicitly is useful. Fortunately, stream kernels can be identified with functions over arrays, in which the computation expressed by the function is replicated and applied to every element of the input arrays (this is sometimes called a *map* operation; in Fortran it is called an *elemental* function). In addition, implementation of stream

kernels as functions provides a convenient place to declare and manipulate local state (which can be mapped to on-chip memory) and to encapsulate imperative control flow.

Memory and Collective Operations: The basic operation in a stream processing model is the application of a kernel to one or more input arrays, generating one or more output arrays. Kernels can contain arbitrary computations, but are not generally allowed to have side effects. In particular, they are not allowed to write to arbitrary memory locations, particularly to locations currently being used as inputs by other kernel instances, since this would have the potential to cause race conditions. By prohibiting side effects, the implementation is free to execute kernels in any order and to hierarchically map them onto multiple parallelism hardware mechanisms in a given processor.

Often implementations of the stream processing model do allow kernels to read from random locations in memory, as long as they are not reading from an array they are also using for output. Such random-access inputs, sometimes called gathers, will generally be less efficient than streaming input. A separate parallel collective gather operation on arrays may also be provided.

However, stream kernel application alone, even with random access reads, does not provide a generally efficient programming model. It is also necessary to provide, at a minimum, the ability to write to a random location in an output array, an operation known as scatter. A collective operation that takes an array of data and an array of addresses can provide this operation, or it can be provided directly from the kernel, along with some suitable rules to disambiguate or avoid collisions. In order to maintain deterministic behavior, the programming model must define a deterministic rule for resolving collisions during scatter and implement it in terms of the mechanisms available on the underlying hardware. For example, writes might be assigned unique priorities, with the highest priority writes always "winning" in case of a collision. If portability is not a major concern, then it is even possible that a different rule might be used on different hardware platforms, as long as it is deterministic and each write is atomic if it succeeds.

If application of a stream kernel to an array can be viewed as the generation of a large number of parallel tasks, then scatter and gather can be seen as a communication and synchronization mechanism between these tasks.

The combination of gather, scatter, scan, and reduction collectives with vector operations has been called the scan-vector model; by extension, we can call the combination of gather, scatter, reduction, and scan collectives with high arithmetic intensity kernels the scan-stream model.

Sometimes the term "stream processing" is used to refer to pipeline parallelism, where the output of one task is sent directly to another in a producer/consumer model. For example, field-programmable gate-array (FPGA) accelerators often make extensive use of this form of parallelism. However, in this paper, the use of "stream programming" refers specifically to high arithmetic intensity array parallelism, as described. In general, in stream processing, no assumption is made on the order in which kernels are executed; specifically, kernels cannot be serially dependent as in pipeline parallelism. However, the combination of array and pipeline "stream" parallelism is potentially interesting since pipeline parallelism allows the expression of certain forms of serial memory access locality not possible under array parallelism.

SIMD Versus SPMD Processing Models: Stream processing can take two forms: SIMD and SPMD. In SIMD processing, the stream kernels are a sequence of data manipulation instructions. In the SPMD stream processing model, the kernels may additionally include control flow (branching). The SPMD processing model is strictly more powerful than the SIMD model, and is similar to bulk-synchronous task parallelism [30], with similar benefits in terms of structuring and predictable performance.

The SIMD model of stream processing is a very simple generalization of vector processing. In the SIMD model of stream processing, each kernel is a simple sequence of instructions, so kernels take a constant amount of time to execute. Data can be read in, the instructions in a kernel executed on state stored in a local memory, and then the results written out again. This processing model has high arithmetic intensity and very predictable performance, but like other SIMD models has severe limits on its applicability.

In an SIMD model, there is no opportunity to do less work in special cases: all instructions must be applied to every data element, without exception. Often it is necessary to do extra work and then throw it away, which eliminates many of the benefits of parallel execution for some applications.

In an SPMD stream processing model, the kernels are more like general programs, and in particular can branch based on computed values in a way that actually avoids work. With branching, kernels can select different sequences of instructions to execute in different circumstances and can reexecute some instructions again and again as needed. This allows SPMD kernels to do less work when possible and more work when necessary.

It is possible to efficiently emulate certain forms of structured task parallelism with the SPMD model. A stream kernel can simply include the code for all tasks but branch as needed to perform the computation for only one task or the other. As mentioned earlier, it is also possible to use scatter and gather operations for communication. However, the SPMD stream processing model has the significant advantage that race conditions and nondeterministic programs are impossible to express, eliminating this class of error by construction.

By simply omitting control flow in a kernel, the SPMD model can be used to express a SIMD computation. However, when control flow is present, load balancing

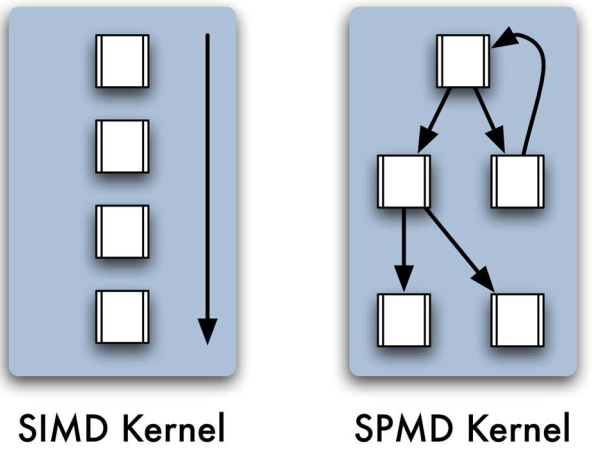

Fig. 1. *A comparison of SIMD and SPMD kernels. SIMD kernels can only execute a fixed sequence of operations. SPMD kernels can select different execution paths based on data-dependent conditions.*

must be considered for efficient execution. SIMD and SPMD kernels are compared in Fig. 1.

Load Balancing: When using the SPMD model for stream processing, some kernels will take less time to execute than others. This leads to a load balancing problem.

A SIMD execution model does not have a load balancing problem, since every kernel takes the same worst case amount of time to execute. However, looking at it another way, load balancing is not needed in the SIMD model only because there is no *opportunity* to avoid unnecessary work. If each kernel did only the necessary work and terminated as early as possible, then other, more useful work could be done instead. This is what is done in the SPMD model. Because of the variable execution times of the kernels, efficient multicore execution under the SPMD model requires a load balancer. The load balancer attempts to assign an equal amount of work to each core, so they all finish at around the same time. In order to make dynamic load balancing efficient, it is useful to have many more kernel invocations than cores.

Fortunately, the data-parallel stream processing model typically provides thousands to millions of kernel invocations. In this case, many kernel invocations can be assigned to each core, and a load balancer can typically achieve a near-uniform load.

C. Architectural Features and Processors

One of the reasons for studying abstract programming models is the desire for portability. If a common programming model can be found that maps a range of important applications efficiently onto a broad range of hardware architectures, then application logic can be developed independently from the selection of the deployment hardware. This would allow a single application to run on different existing architectures as well as allow an application written now to run on future processors.

In this section, therefore, we will survey at a high level some specific multicore processor architectures. We focus on the various forms of parallelism exploited by these architectures in order to have a clear understanding of their processing models.

Multicore Processors: A multicore processor replicates a single core design, duplicating it several times on a single chip. Each core can run a completely separate thread of control, so this is an MIMD processing model.

Multicore processor designs are distinguished from one another by how they manage memory and whether they support intercore communication. In addition, some multicore processors support different kinds of cores on a single chip, with some cores designed for specific tasks. If all the cores are the same, then the design is called homogeneous; otherwise it is heterogeneous.

For example, Intel and AMD multicore processors provide a homogeneous set of cores with both per-core and shared caches. From the programming point of view, multicore Intel and AMD processors are similar to symmetric multiprocessor systems, although with somewhat different performance characteristics.

In contrast, the heterogeneous Cell BE multicore chip (a high-level diagram is shown in Fig. 2) has one general-purpose core and eight other smaller cores specialized for numerically intensive workloads. These smaller cores also have explicitly managed local memory and SWAR parallelism, and can communicate directly with each other via message passing over an on-chip ring network. Intel and AMD CPU cores, of course, also support SWAR, pipelining, instruction co-issue, and other forms of microarchitectural parallelism.

Multithreaded Processors: A multithreaded processor can execute multiple simultaneous threads of control but does not replicate the entire core. Instead, multiple instruction streams are fetched simultaneously, and the processor attempts to schedule the instructions to make best use of the functional units.

As a simple example of this, suppose a processor has a pipelined floating-point unit with a latency of four cycles. A processor could support four threads of control, and could fetch instructions from them in round-robin order. This strategy would, from the point of view of each thread, completely hide the latency of the functional unit, permitting efficient execution of highly data-dependent code. More sophisticated versions of multithreading may have multiple functional units and may try to schedule multiple instructions drawn from multiple threads to run simultaneously [31].

The advantage of a multithreaded processor over a multicore processor is that fewer resources need to be replicated; however, due to contention for the nonreplicated

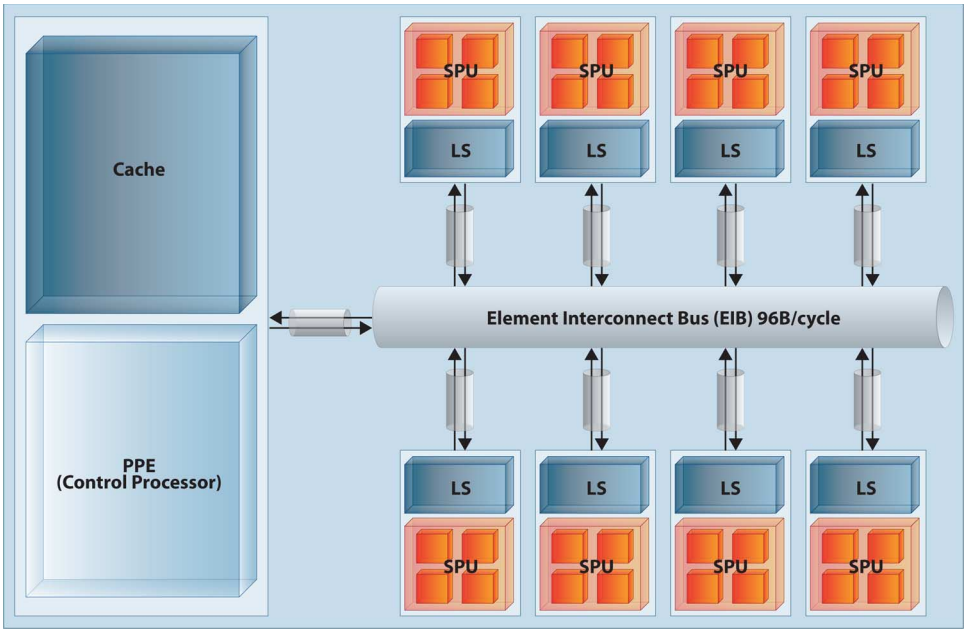

Fig. 2. ***High-level diagram of the IBM Cell BE architecture.***

resources, a multithreaded processor may not be able to achieve as much of a speedup as a multicore processor. Of course, multithreading and multicore architectures can be combined, and they have a similar MIMD processing model.

SPMD Processors: In an MIMD processor, every core or thread in a core can be running a completely different process, and these processes are managed by the operating system, typically. Unfortunately, in an MIMD machine, processes compete for resources, and lack of coordination between processes can result in reduced performance.

The SPMD processing model is like the MIMD model, in that multiple threads of control are permitted. However, it coordinates the various threads of control and treats them as part of a single program or function in order to reduce contention for resources and to optimize data movement.

An SPMD processor includes explicit support for coordinating threads, and GPUs in particular should be considered SPMD processors rather than SIMD or MIMD processors. See Fig. 3 for a high-level architectural diagram of the ATI 2900 (R600) GPU and Fig. 4 for a high-level architectural diagram of the NVIDIA 8800 (G80) GPU.

GPU architectures are designed to support a massive number of threads running simultaneously but are able to suspend threads to hide the latency of certain operations, such as memory reads. In addition, SIMD computation is often exploited over a set of spatially coherent tiles, and on most GPUs, SWAR or VLIW parallelism is also available.

In the NVIDIA G80 architecture, 128 functional units are available but are grouped in processing elements containing eight functional units each. Each processing element has a single thread of control but uses masking to emulate control flow on a number of threads. Processing elements also have access to a small local memory (L1) and a set of read-only caches. Data can be written to a write cache that can also support a limited set of read–modify–write operations. An onboard thread processor keeps track of active and suspended SIMD tiles and schedules tiles for execution on the processor array.

The ATI R600 architecture is similar but does have some differences. First, it is an explicit VLIW architecture, with five-way instruction issue. Also, like the G80, it has a hierarchy of functional units arranged to exploit spatially coherent control flow using SIMD tiles. In the ATI R600, there are four processing elements with 5×16 functional units each. Each processing element has its own flow of control but uses masking to emulate data-dependent control flow at finer granularity. In addition, there is a dispatch processor that can schedule the execution of SIMD tiles and suspend them temporarily when they are waiting for a high-latency operation (such as a memory access) to complete. A number of shared read-only caches (texture units) are supported as well as a number of write caches (render back ends).

Other processors exist that use massive multithreading and an onboard scheduler. For example, the Plurality processor[2] implements a master scheduler in hardware and uses a graph to track data and task dependencies; it parcels out work units as their input data are made available and predecessor tasks complete. Use of a hardware scheduler has the potential for reducing the overhead of

[2]http://www.plurality.com.

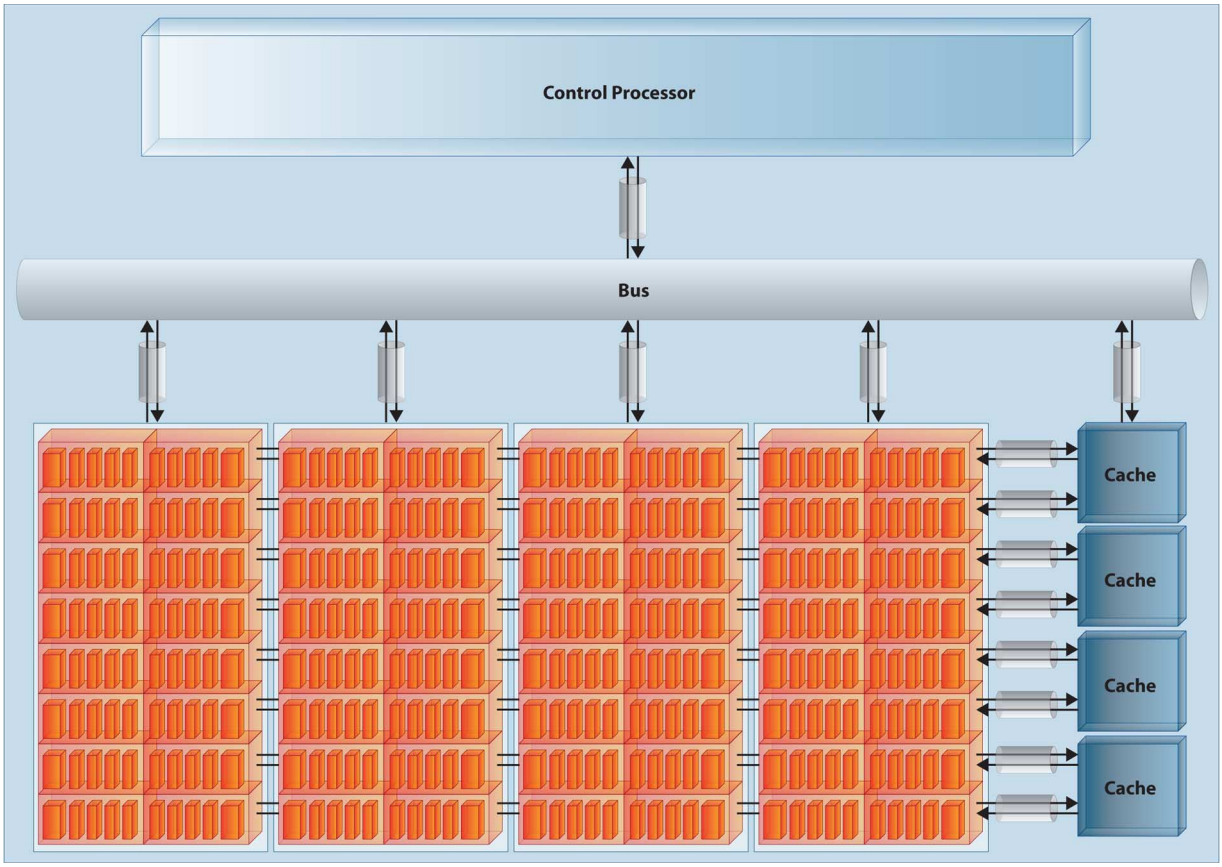

Fig. 3. ***High-level diagram of the AMD/ATI 2900 (R600) architecture.***

scheduling and therefore allowing finer-grained threads to execute more efficiently.

Stream Processors: Stream processing was introduced with the Imagine and Merrimac designs [32]. A stream processor is characterized by hierarchical on-chip explicitly managed memory and scheduled data transfer. A stream processor is similar to an MIMD multicore processor like the Cell BE but has a deeper on-chip memory hierarchy. A stream processor can be considered to be a generalization of a vector processor. Like a vector processor, a stream processor applies an operation to a collection of data. However, in a stream processor, the operation can be an entire function, whereas in a classic vector processor, the operation can only be a single instruction. Compared to a vector processor, a stream processor reduces the need to transfer data to and from the external memory system, since it supports higher on-chip arithmetic intensity.

Memory and Communication Systems: In addition to different forms of parallelism, it is important to consider the different forms that the memory and communication systems can take. Often access to data is the chief bottleneck in a system.

Bus systems that use a common communication fabric to connect memory and processors do not scale to a large number of processors. Scalable processor/memory interconnects typically use a network fabric. The AMD HyperTransport protocol is a point-to-point network that allows a certain number of links to every processor as well as a certain number of memory ports. A large system can connect many processors together into a network. All processors can access all memories; however, depending on where the memory and the processors are located with respect to one another, the access latency may vary. This is known as a nonuniform memory architecture. Several multicore processors also use an on-chip ring network to communicate between processors and possibly memory. However, a single ring network does not scale to a large number of cores, and so for very large systems, a mesh or hypercube topology is usually considered, or some kind of hierarchical interconnect.

An additional issue that arises with memory is that of atomicity. If two processors write to the same location in memory, what will the result be? Often in order to ensure correctness, we have to ensure that an entire sequence of memory reads, computations, and writes complete atomically, which may be complicated by the fact that the

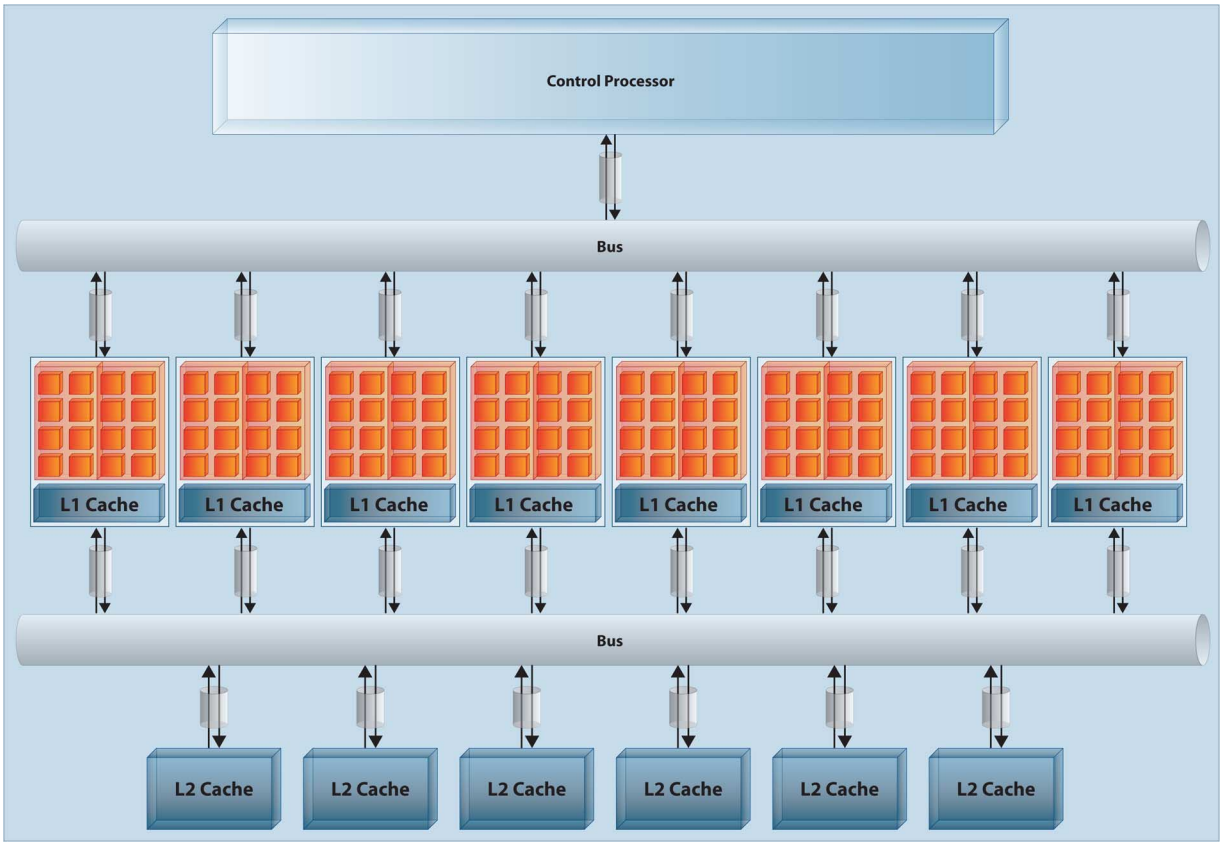

Fig. 4. *High-level diagram of the NVIDIA 8800 (G80) GPU architecture.*

underlying hardware may transfer data in relatively large blocks. Locking mechanisms can be implemented to secure exclusive access to a resource such as a set of memory locations, but these often will not scale well to a large number of cores.

General-purpose CPU designs, for historical reasons, tend to use cache coherency protocols. Implementation of the cache coherency protocol relies on implicit communication among processors.

III. PROGRAMMING MODELS, LANGUAGES, AND PLATFORMS

Now that we have discussed the processing models and outlined the architectural features of some of current multicore processors, we can discuss some programming systems in terms of their programming models and features. We will emphasize systems designed for the GPU and Cell BE based on the stream processing model, since these are relatively recent and have not been reviewed extensively elsewhere, but will also survey some of the other options available.

A. Shading Languages

GPUs were originally designed as fixed-function graphics accelerators, using a fixed pipeline of operations. Most of the chip space devoted to parallel computation in modern GPUs, however, is there to support programmable shading. Graphics APIs support several shading languages, and these shading languages can be used, with some effort, to perform general-purpose computations.

The main shading languages available today are Cg [33], the OpenGL Shading Language (GLSL) [34], and the DirectX High-Level Shading Language (DX HLSL) [35]. The RapidMind platform, discussed later, evolved out of an experimental shading language called Sh [36], [37]. We mention shading languages since historically, these were the first systems to expose programmability on GPUs, and the combination of shading languages and graphics APIs was used for many years to implement algorithms on GPUs.

A general survey on using GPUs for general-purpose computation has been compiled by Owens *et al.* [38].

B. General-Purpose Stream Programming Languages

GPUs can be programmed through graphics APIs and shading languages, but this is not optimal for general-purpose computations. At a high level, graphics APIs use a stream processing model but the mapping between graphics concepts and the elements of this model can sometimes be obscure. For this reason, systems have been

developed that hide the graphics-specific nature of GPUs while making their processing power available for general-purpose computations. In some cases, these systems have also been mapped to additional targets, such as the Cell BE and multicore CPUs.

The Brook system uses an SPMD stream processing model and has been mapped onto both GPUs and the Merrimac stream supercomputer[3] [39]. Recently, its processing model has been extended by Sequoia [40], [41], which adds the concepts of tunables and recursion over the memory hierarchy in order to better exploit memory locality. As noted, there can be many kinds and levels of parallelism and memory in a parallel processor. Sequoia permits code to be structured so that a recursive problem can be conveniently mapped onto such a structure.

The RapidMind platform [42] uses an SPMD stream programming model but generalizes it to multidimensional arrays. The RapidMind array abstraction also encapsulates data strongly, allowing automatic management of remote data stored on accelerators. RapidMind uses an embedded programming model so the kernels can be specified directly in the source code of a controlling C++ program. The RapidMind platform can target the Cell BE, both NVIDIA and ATI GPUs, and multicore CPUs with the same programming model. Via metaprogramming, it also supports tunable, parameterized code. In order to express memory locality, local arrays may be declared and operated on inside other kernels, and arrays may be "diced" into subarrays.

Microsoft Accelerator [43] (targeting GPUs) and the Peakstream platform[4] (targeting ATI GPUs) use a SIMD vector programming model in their interfaces based on operators over arrays. They then attempt to automatically infer arithmetically intense stream kernels. However, as noted, an SIMD processing model can lead to serious inefficiencies, especially on more complex, irregular problems. An SIMD kernel cannot avoid unnecessary computation in special cases. Secondly, these systems infer kernels rather than allowing their direct specification. While this simplifies the interface somewhat, high-performance code requires control of what operations get placed in each kernel.

Ct is another data-parallel programming platform under development by Intel [44] that targets x86 multicore CPUs. Ct supports the SPMD stream programming model and uses an embedded C++ interface similar to the RapidMind platform. Ct also includes an implementation of segmented collectives, which are useful for implementing algorithms using nested parallelism.

There are a couple of other languages that target GPUs or the Cell BE but are designed to be portable, including StreamIt [45] and Scout [46]. These also use variants of the stream processing model. The CellSs system [47] uses annotations to identify parallel constructs in a C program and then maps these computations to the coprocessors in the Cell BE; it uses an SPMD model.

C. Vendor-Specific Languages

Recently, some lower level interfaces have been provided by GPU hardware vendors.

The close-to-the-metal (CTM) interface [48] from AMD/ATI provides a simple, flat model of memory, a way to specify and invoke processing passes, and a way to load assembly language specifications of the computation into each kernel. This interface supports an SPMD stream processing model, since the assembly language supports control flow. Although CTM provides a simple and complete interface to ATI GPUs, its low-level nature makes it more suited as a compilation target than as a general-purpose programming environment. However, several high-performance libraries have been implemented using this interface.

The CUDA system[5] from NVIDIA is a high-level language based on an extension of C in which certain functions are identified using a special syntax as stream functions. Application of these functions to arrays then invokes parallel computations on an NVIDIA GPU. CUDA is based on an SPMD stream processing model but includes extensions that allow the specification of thread blocks that in practice will execute together on a single processor element using SIMD masking. A model of communication is also supported between elements in a thread block. This is implemented using the local memory in each cluster since all elements in a "thread block" actually execute together in SIMD fashion.

D. Other Approaches to Parallel Programming

Data-parallel computation has been used at the basis for several parallel programming languages, including APL [49] and C* [50]. APL identified the usefulness of the scan collective operation, and C* introduced restriction to regions. Both were based on the SIMD (vector) programming model.

The current Fortran standard [28] includes several data-parallel programming constructs and supports a data-parallel vector processing sublanguage. It also permits the definition of "pure" and "elemental" functions, which are similar to stream kernels. Coarray Fortran [51] and high-performance Fortran [27] include a number of other features for organizing the distribution of data.

Fortress is a proposed parallel language that supports the implementation of data-parallel abstractions [52]. Loops are based on generators and are parallel by default; there is also built-in support for reduction. Fortress also

[3]http://www.graphics.stanford.edu/projects/brookgpu/.
[4]http://www.peakstreaminc.com/.
[5]http://www.developer.nvidia.com/cuda/.

supports the concept of regions, which allow the hierarchical decomposition of parallelism; and distributions, for mapping data structures onto regions.

Other proposed portable parallel programming languages, such as UPC [53], [54], unfortunately make assumptions about the memory model that are inconsistent with the split read/write caches in GPUs. Specifically, GPUs separate cached reads from writes and do not currently directly support efficient (cached) fine-grained read–modify–write operations to off-chip memory. Likewise, the Cell BE does not have a hardware cache on its synergistic coprocessors. Implementation of a coherent cache in software on the Cell BE is theoretically possible but would (probably) be too expensive to be useful; it is better thought of as a distributed-memory cluster-on-a-chip. The assumption of a shared memory supporting fine-grained read–modify–write operations is also true of OpenMP, a standard for annotating code to express loop-carried parallelism. However, OpenMP is supported by many current CPU compilers.

The Titanium project [55] consists of a set of extensions to Java that can express many kinds of data parallelism and also includes explicit support for managing regions of arrays as well as partitioned address spaces. Another recent proposal for extending Java is X10 [56], which supports partitioned memory spaces and a data-parallel sublanguage.

Chapel [57] is an entirely new language that permits the expression of both data and task parallelism. Task parallelism is supported with a "cobegin" statement that can execute a set of statements in parallel; they can also be nested. It includes types for domains (sets of locations in arrays) and locales (locations where computations are performed) and also supports atomic regions. Variables support empty/full semantics in addition to their value; this permits the use of variables as message-passing communication channels between threads.

Cilk [58] is an extension of C that supports very lightweight tasks by a generalization of the idea of a function call. It does not emphasize data parallelism but it is interesting to note that stream programming models are also based on functions, and a hybrid between these two approaches is possible. Cilk uses an efficient load-balancing strategy called *work stealing* that minimizes communication between processors.

Programming models based on distributed memory models such as MPI can be mapped onto multicore processors such as the Cell BE that are essentially clusters on a chip, but the limited local memory space available relative to the requirements of existing MPI codes makes this awkward. MPI was primarily designed for large-grain tasks. However, the MicroMPI system [59] presents some modifications to MPI that make a mapping onto the small local memories in the Cell BE feasible and can permit the scheduling and merging of tasks based on their communication dependencies.

Finally, many functional languages permit or have parallel implementations, and some have explicit constructs supporting parallelism. Concurrent Clean [7] and Erlang [60] are especially notable. NESL is a functional language that directly supports the scan-vector model and also supports a powerful form of nested recursion based on segmented collective operations [9], [61]–[63]. However, due to the necessary nontrivial transformation between pure functional languages and the imperative processing models of real machines, as well as the current emphasis on imperative languages in practice, it is unclear what the acceptance of pure functional languages would be. An incremental approach that builds on existing imperative programming models would probably be desirable for many applications. The SPMD stream processing model is a hybrid, using concepts like pure functions borrowed from functional languages at a high level for stream kernels in combination with imperative control flow at the lowest level.

For more information, a survey of data-parallel languages is available [64].

IV. CONCLUSIONS

We have reviewed several programming models, and in particular have discussed programming systems based on the stream processing model.

Several programming systems are now available that make use of some variant of the stream processing model. Implementations of this model with good efficiency have now been demonstrated on GPUs, the Cell BE, and multicore CPUs. This model also has desirable properties in terms of generality and safety. In particular, if implemented in a deterministic fashion, it is not possible to express programs using such a system that have race conditions or deadlock. At the same time, the stream programming model exposes at a high level the most important aspects of the underlying processing models needed to tune performance: parallelism and locality.

Real processors of course contain many levels and types of parallelism. One of the attractive features of the SPMD stream processing model is that the parallelism specified by the programmer can be hierarchically and automatically decomposed over whatever forms of parallelism are present in the hardware.

Future work will likely extend and enhance the SPMD stream processing model. For example, full support for recursion and nested parallelism would be useful, and it would also be desirable to extend its support for fine-grained task parallelism. Extending functions to permit the spawning of nested subtasks would be one way to do this. However, existing programming systems based on stream parallelism can expose the power of a range of massively parallel processors today, and the SPMD stream processing model will likely be a major component of future approaches to scalable multicore programming.■

REFERENCES

[1] D. W. Wall, "Limits of instruction-level parallelism," in *Int. Conf. Architectural Support Program. Lang. Oper. Syst. (ASPLOS)*, 1991, vol. 26, pp. 176–189.

[2] R. S. Chappell, F. Tseng, A. Yoaz, and Y. N. Patt, "Microarchitectural support for precomputation microthreads," in *Int. Symp. Microarchitect.*, Istanbul, Turkey, Nov. 2002.

[3] S. A. Mahlke, "Exploiting instruction-level parallelism in the presence of conditional branches," Ph.D. dissertation, Dept. of Electrical and Computer Engineering, Univ. of Illinois, Jan. 1997.

[4] J. Shin, "Introducing control flow into vectorized code," in *Parallel Architect. Compil. Tech. (PACT)*, Brasov, Romania, Sep. 15–19, 2007.

[5] D. B. Skillicorn and D. Talia, "Models and languages for parallel computation," *ACM Comput. Surv.*, vol. 30, no. 2, pp. 123–169, Jun. 1998.

[6] K. Olukotun and L. Hammond, "The future of microprocessors," *ACM Queue*, pp. 26–34, Sep. 2005.

[7] R. Plasmeijer, M. van Eekelen, M. Pil, and P. Serrarens, "Parallel and distributed programming in concurrent clean," in *Research Directions in Parallel Functional Programming*, K. Hammond and G. Michaelson, Eds. Berlin, Germany: Springer-Verlag, pp. 323–338.

[8] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec.18, 2006.

[9] G. E. Blelloch, *Vector Models for Data-Parallel Computation*. Cambridge, MA: MIT Press, 1990.

[10] A. Gibbons and W. Rytter, *Efficient Parallel Algorithms*. Cambridge, U.K.: Cambridge Univ. Press, 1988.

[11] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Trans. Comput.*, vol. C-21, pp. 948–960, Sep. 1972.

[12] D. Sima, T. Fountain, and P. Karsuk, *Advanced Computer Architectures: A Design Space Approach*. Reading, MA: Addison-Wesley, 1997.

[13] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978.

[14] A. Kejariwal, H. Saito, X. Tian, M. Girkar, W. Li, U. Banerjee, A. Nicolau, and C. D. Polychronopoulos, "Lightweight lock-free synchronization methods for multithreading," in *Proc. ICS 2006*, 2006, pp. 361–371.

[15] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proc. Int. Symp. Comput. Architect.*, 1993, pp. 289–300.

[16] J. R. Larus and R. Rajwar, *Transactional Memory*. San Rafael, CA: Morgan & Claypool, 2006.

[17] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded transactional memory," in *Proc. Int. Symp. High-Perform. Comput. Architect.*, Feb. 2005, pp. 316–327.

[18] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis, "Comparing memory systems for chip multiprocessors," in *Proc. Int. Symp. Comput. Architect. (ISCA)*, San Diego, CA, Jun. 2007.

[19] P. Hudak and E. Mohr, "Graphinators and the duality of SIMD and MIMD," in *ACM Conf. Lisp Functional Program.*, Jul. 1988, pp. 224–234.

[20] M.-Y. Wu and W. Shu, "MIMD programs on SIMD architectures," in *Proc. IEEE 6th Symp. Frontiers Massively Parallel Comput. (FRONTIERS '96)*, Washington, DC, 1996, p. 162.

[21] P. Sanders, "Efficient emulation of MIMD behavior on SIMD machines," Fakultat für Informatik, Univ. Karlsruhe, 1995, Technical Rep.

[22] T. S. Popa, "Compiling data dependent control flow on SIMD GPUs," M.Math thesis, School of Computer Science, Univ. of Waterloo, Waterloo, ON, Canada, 2004.

[23] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for GPU computing," in *Proc. Graph. Hardware 2007*, Aug. 2007, pp. 97–106.

[24] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan, "Ray tracing on programmable graphics hardware," in *ACM Trans. Graph. (Proc. SIGGRAPH)*, Jul. 2002, vol. 21, no. 3, pp. 703–712.

[25] M. Harris and I. Buck, "GPU flow control idioms," in *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, M. Pharr, Ed. Reading, MA: Addison-Wesley, 2005, pp. 547–555.

[26] R. J. Fisher and H. G. Dietz, *Compiling for SIMD Within a Register*. Berlin, Germany: Springer-Verlag, 1998, pp. 290–304.

[27] J. Gebis and D. Patterson, "Embracing and extending 20th-century instruction set architectures," *IEEE Comput.*, vol. 40, no. 4, pp. 68–75, Apr. 2007.

[28] High Performance Fortran Forum, *High performance Fortran language specification*, May 1993.

[29] Fortran 2003 Standard, ISO/IEC FCD 1539-1, JTC1/SC22/WG5.

[30] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.

[31] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proc. Int. Symp. Comput. Architect.*, Jun. 1995, pp. 392–403.

[32] A. Das, W. J. Dally, and P. Mattson, "Compiling for stream processing," in *Proc. ACM Parallel Architect. Compil. Tech. (PACT 2006)*, Seattle, WA, Sep. 16–20, 2006, pp. 33–42.

[33] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, "Cg: A system for programming graphics hardware in a C-like language," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 896–907, 2003.

[34] R. J. Rost, *OpenGL Shading Language*, 2nd ed. Reading, MA: Addison-Wesley, 2006.

[35] C. Peeper and J. L. Mitchell, "Introduction to the DirectX 9 high level shading language," in *Shader X2*, W. R. Engel, Ed. Plano, TX: Wordware, 2003.

[36] M. D. McCool, Z. Qin, and T. S. Popa, "Shader metaprogramming," in *Proc. Graph. Hardware*, Sep. 2002, pp. 57–68.

[37] M. McCool and S. Du Toit, *Metaprogramming GPUs With Sh*. Wellesley, MA: AK Peters, 2004.

[38] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," in *Proc. Comput. Graph. Forum (EG STAR Rep.)*, 2007, vol. 26, pp. 80–113.

[39] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream computing on graphics hardware," in *ACM Trans. Graph. (Proc. SIGGRAPH)*, Aug. 2004, vol. 23, no. 3.

[40] K. Fatahalian, T. J. Knight, M. Erezand, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: Programming the memory hierarchy," in *ACM/IEEE Conf. Supercomput.*, 2006.

[41] T. J. Knight, J. Y. Park, M. Ren, M. Houston, M. Erez, K. Fatahalian, A. Aiken, W. J. Dally, and P. Hanrahan, "Compilation for explicitly managed memory hierarchies," in *Proc. ACM SIGPLAN Symp. Principles Practice Parallel Program. (PPoPP)*, Mar. 2007, pp. 226–236.

[42] M. D. McCool, "Data-parallel programming on the cell BE and the GPU using the RapidMind development platform," in *Proc. GSPx Multicore Applicat. Conf.*, Oct.–Nov. 2006.

[43] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: Using data parallelism to program GPUs for general-purpose uses," in *Proc. ACM Conf. Architect. Support Program. Lang. Oper. Syst.*, Oct. 2006, Microsoft Tech. Rep. 2005-184.

[44] A. Ghuloum, E. Sprangle, and J. Fang *Flexible parallel programming for tera-scale architectures with Ct*, Apr. 26, 2007, Intel White Paper.

[45] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A language for streaming applications," in *CC '02: Proc. 11th Int. Conf. Compiler Construct.*, London, U.K., 2002, pp. 179–196.

[46] P. S. McCormick, J. Inman, J. P. Ahrens, C. Hansen, and G. Roth, "Scout: A hardware-accelerated system for quantitatively driven visualization and analysis," in *Proc. IEEE Visualization '04*, Washington, DC, 2004, pp. 171–178.

[47] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "CellSs: A programming model for the cell BE architecture," in *ACM/IEEE Conf. Supercomput.*, Nov. 11–17, 2006.

[48] *CTM guide: Technical reference manual*, AMD, 2006.

[49] K. E. Iverson, *A Programming Language*. New York: Wiley, 1962.

[50] J. R. Rose and G. L. Steele, Jr., "C*: An extended C language for data parallel programming," in *Proc. 2nd Int. Conf. Supercomput.*, San Francisco, CA, May 1987, vol. 2, pp. 2–16.

[51] R. W. Numrich and J. Reid, "Co-array Fortran for parallel programming," *ACM SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, 1998.

[52] G. Steele, "A growable language (fortress)," in *ACM SIGPLAN Int. Conf. Object-Oriented Syst., Program., Lang., Applicat. (OOPSLA)*, Oct. 25, 2006, (talk).

[53] UPC Consortium, *UPC language specification version 1.2*, 2005.

[54] A. Kamil, J. Su, and K. Yelick, "Towards a sequentially consistent memory model for PGAS languages," in *Proc. ACM/IEEE Conf. Supercomput.*, Nov. 11–17, 2006.

[55] P. Hilfinger, D. Bonachea, K. Datta, D. Gay, S. Graham, B. Liblit, G. Pike, J. Su, and K. Yelick, *Titanium language reference manual version 2.19*, U.C. Berkeley Tech Rep. UCB/EECS-2005-15, 2005.

[56] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar, "X10: An Object-oriented approach to non-uniform clustered computing," in *ACM SIGPLAN Int.*

Conf. Object-Oriented Syst., Program., Lang., Applicat. (OOPSLA), 2005.

[57] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the chapel language," *Int. J. High Perform. Comput. Applicat.*, vol. 21, no. 3, pp. 291–312, Aug. 2007.

[58] K. H. Randall, "Cilk: Efficient Multithreaded Computing," Ph. D. dissertation, Dept. of Electrical Engineering and Computer Science, Massachusetts Inst. of Technology, Jun. 1998.

[59] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani, "MPI microtasks for programming the cell broadband engine processor," *IBM Syst. J.*, vol. 45, no. 1, pp. 85–102, Mar. 2006.

[60] J. Armstrong, R. Virding, C. Wikström, and M. Williams, *Concurrent Programming in Erlang*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1996.

[61] G. W. Sabot, *The Paralation Model: Architecture-Independent Parallel Programming*. Cambridge, MA: MIT Press, 1988.

[62] G. E. Blelloch, "Programming parallel algorithms," in *Proc. Dartmouth Inst. Adv. Grad. Study Parallel Comput. Symp.*, D. B. Johnson, F. Makedon, and P. Metaxas, Eds., 1992, pp. 11–18.

[63] G. E. Blelloch and J. Greiner, "A provable time and space efficient implementation of NESL," in *ACM SIGPLAN Int. Conf. Functional Program.*, May 1996, pp. 213–225.

[64] J. Sipelstein and G. E. Blelloch, "Collection-oriented languages," *Proc. IEEE*, vol. 79, pp. 504–523, Apr. 1991.

ABOUT THE AUTHOR

Michael D. McCool (Member, IEEE) received the B.A.Sc. degree in computer engineering and the M.Sc. and Ph.D. degrees in computer science from the University of Waterloo, Waterloo, ON, Canada in 1989, 1991, and 1995, respectively.

He is an Associate Professor in the David R. Cheriton School of Computer Science, University of Waterloo. He is Cofounder of RapidMind Inc., which develops a general-purpose programming platform that targets high-performance parallel machines and multicore processors including multicore CPUs, the Cell BE, and GPUs. His current research efforts are targeted at enabling high-performance parallel applications by the development of advanced programming technologies. His research interests include interval analysis, Monte Carlo and quasi-Monte Carlo numerical methods, optimization, simulation, sampling, cellular automata, real-time computer graphics, vision, image processing, hardware design, programming languages, and software development platforms.

Prof. McCool received the Sir Sandford Medal from the University of Waterloo.