Tesla vs. Xeon Phi vs. Radeon
A Compiler Writer's Perspective

Brent Leback, Douglas Miles, *and* **Michael Wolfe**
The Portland Group (PGI)

ABSTRACT: *Today, most CPU+Accelerator systems incorporate NVIDIA GPUs. Intel Xeon Phi and the continued evolution of AMD Radeon GPUs make it likely we will soon see, and want to program, a wider variety of CPU+Accelerator systems. PGI already supports NVIDIA GPUs, and is working to add support for Xeon Phi and AMD Radeon. Here we explore the features common to all three types of accelerators, those unique to each, and the implications for programming models and performance portability from a compiler writer's and application writer's perspective.*

KEYWORDS: Compiler, Accelerator, Multicore

1. Introduction

Today's high performance systems are trending towards using highly parallel accelerators to meet performance goals and power and price limits. The most popular compute accelerators today are NVIDIA GPUs. Intel Xeon Phi coprocessors and AMD Radeon GPUs are competing for that same market, meaning we will soon be programming and tuning for a wider variety of host + accelerator systems.

We want to avoid writing a different program for each type of accelerator. There are at least three current options for writing a single program that targets multiple accelerator types. One is to use a library, which works really well if the library contains all the primitives your application needs. Solutions built on class libraries with managed data structures are really another method to implement libraries, and again work well if the primitives suit your application. The potential downside is that you depend on the library implementer to support each of your targets now and in the future.

A second option is to dive into a low-level, target independent solution such as OpenCL. There are significant upfront costs to refactor your application, as well as additional and continuing costs to tune for each accelerator. OpenCL, in particular, is quite low-level, and requires several levels of tuning to achieve high performance for each device.

A third option is to use a high-level target independent programming model, such as OpenMP or OpenACC directives. OpenMP has had great success for its target domain, shared-memory multiprocessor and multicore systems. OpenACC has shown some initial success for a variety of applications [1, 2, 14, 15, 16, 17]. It is just now being targeted for multiple accelerator types, and our early experience is that it can be used to achieve high performance for a single program across a range of accelerator systems.

Using directives for parallel and accelerator programming divides the programming challenges between the application writer and the compiler. Defining an abstract view of the target architecture used by the compiler allows a program to be optimized for very different targets, and allows programmers to understand how to express a program that can be optimized across accelerators.

PGI has been delivering directive-based Accelerator compilers since 2009. The current compilers implement the OpenACC v1.0 specification on 64-bit and 32-bit Linux, OS/X and Windows operating systems, targeting NVIDIA Tesla, Fermi and Kepler GPUs. PGI has demonstrated initial functionality on AMD Radeon GPUs and APUs, and Intel Xeon Phi co-processors. The PGI compilers are designed to allow a program to be compiled for a single accelerator target, or to be compiled into a single binary that will run on any of multiple accelerator targets depending on which is available at runtime. They are also designed to generate binaries that utilize multiple accelerators at runtime, or even multiple accelerators from different vendors. To make all these cases truly useful, the program must be written in a way that allows the

compiler to optimize for different architectures that have common themes but fundamentally different parallelism structures.

2. Accelerator Architectures Today

Here we present important architectural aspects of today's multi-core CPUs and the three most common accelerators designed to be coupled to those CPUs for technical computing: NVIDIA Kepler GPU [11], Intel Xeon Phi [13] and AMD Radeon GPU [12]. There are other accelerators in current use, such as FPGAs for bioinformatics and data analytics, and other possible accelerator vendors, such as DSP units from Texas Instruments and Tilera, but these have not yet achieved a significant presence in the technical computing market.

2.1 Common Multi-core CPU

For comparison, we present an architectural summary of a common multi-core CPU. An Intel Sandy Bridge processor has up to 8 cores. Each core stores the state of up to 2 threads. Each core can issue up to 5 instructions in a single clock cycle; the 5 instructions issued in a single cycle can be a mix from the two different threads. This is Intel's Hyperthreading, which is an implementation of Simultaneous Multithreading. In addition to scalar instructions, each core has SSE and AVX SIMD registers and instructions. An AVX instruction has 256-bit operands, meaning it can process 8 single-precision floating point operations or 4 double-precision floating point operations with a single instruction. Each core has a 32KB level-1 data cache and 32KB level-1 instruction cache, and a 256KB L2 cache. There is a L3 cache shared across all cores which ranges up to 20MB. The CPU clock rate ranges between 2.3GHz to 4.0GHz (in Turbo mode). The core is deeply pipelined, about 14 stages.

The control unit for a modern CPU can *issue* instructions even beyond the point where an instruction stalls. For instance, if a memory load instruction misses in the L1 and L2 cache, its result won't be ready for tens or hundreds of cycles until the operand is fetched from main memory. If a subsequent add instruction, say, needs the result of that load instruction, the control unit can still issue the add instruction, which will wait at a *reservation station* until the result from the load becomes available. Meanwhile, instructions beyond that add can be issued and can even execute while the memory load is being processed. Thus, instructions are executed out-of-order relative to their appearance in the program.

The deep, two- or three-level cache hierarchy reduces the average latency for memory operations. If a memory data load instruction hits in the L1 cache, the result will be ready in 4-5 clock cycles. If it misses in the L1 cache but hits in the L2 cache, the result will take 12 cycles. An L3 cache hit will take 35-45 clock cycles, while missing completely in the cache may take 250 or more cycles. There is additional overhead to manage multicore cache coherence for the L1 and L2 caches.

GPU marketing literature often uses the term "GPU core" or equivalent. This is essentially a single precision (32-bit) ALU or equivalent. By this definition, an 8-core Sandy Bridge with AVX instructions has 64 GPU core-equivalents.

2.2 NVIDIA Kepler GPU

An NVIDIA Kepler GPU has up to 15 SMX units, where each SMX unit corresponds roughly to a highly parallel CPU core. The SMX units execute *warp*-instructions, where each warp-instruction is roughly a 32-wide SIMD instruction, with predication. The *warp* is a fundamental unit for NVIDIA GPUs; all operations are processed as warps. The CUDA programming model uses CUDA threads, which are implemented as predicated, smart lanes in a SIMD instruction stream. In this section, we use the term *thread* to mean a sequence of warp-instructions, not a CUDA thread. The SMX units can store the state of up to 64 threads. Each SMX unit can issue instructions from up to four threads in a single cycle, and can issue one or two instructions from each thread. There are obvious structural and data hazards that would limit the actual number of instructions issued in a single cycle. Instructions from a single thread are issued in-order. When a thread stalls while waiting for a result from memory or other slow operation, the SMX control unit will select instructions from some other active thread. The SMX clock is about 1.1 GHz. The SMX is also pipelined, with 8 stages.

Each execution engine of the Kepler SMX has an array of execution units, including 192 single precision GPU cores (for a total of 2880 GPU cores on chip), plus additional units for double precision, memory load/store and transcendental operations. The execution units are organized into several SIMD functional units, with a hardware SIMD width of 16. This means that it takes two cycles to initiate a 32-wide warp instruction on the 16-wide hardware SIMD function unit.

There is a two-level data cache hierarchy. The L1 data cache is 64KB per SMX unit. The L1 data cache is only used for thread-private data. The L1 data cache is actually split between a normal associative data cache and a scratchpad memory, with 25%, 50% or 75% of the 64KB selectively used as scratchpad memory. There is an additional 48KB read-only L1 cache, which is accessed through special instructions. The 1.5MB L2 data cache is shared across all SMX units. The caches are small, relative to the large number of SIMD operations running on all 15 SMX units. The device main memory ranges up to 6GB in current units and 12GB in the near future.

2.3 AMD Radeon GPU

An AMD Radeon GPU has up to 32 compute units, where each compute unit corresponds roughly to a simple CPU core with attached vector units. Each compute unit has four SIMD units and a scalar unit, where the scalar unit is used mainly for control flow. The SIMD units operate on *wavefronts*, which roughly correspond to NVIDIA warps, except wavefronts are 64-wide. The SIMD units are physically 16-wide, so it takes four clock cycles to initiate a 64-wide wavefront instruction. With 32 compute units, each with 4 16-wide SIMD units, the AMD Radeon has up to 2048 single precision GPU cores. In this section, I use the term *thread* to mean a sequence of scalar and wavefront instructions.

A thread is statically assigned to one of the four SIMD units. As with NVIDIA Kepler, the AMD Radeon issues instructions in-order per thread. Each SIMD unit stores the state for up to 10 threads, and uses multithreading to tolerate long latency operations. The compute units operate at about 1GHz.

There is a 64KB scratchpad memory for each compute unit, shared across all SIMD units. There is an additional 16KB L1 data cache per compute unit, and a 16KB L1 read-only scalar data cache shared across the four SIMD units. A 768KB L2 data cache is shared across all compute units. The device main memory is 3GB in current devices.

2.4 Intel Xeon Phi Coprocessor

The Intel Xeon Phi Coprocessor (IXPC) is designed to appear and operate more like a common multicore processor, though with many more cores. An IXPC has up to 61 cores, where each core implements most of the 64-bit x86 scalar instruction set. Instead of SSE and AVX instructions, an IXPC core has 512-bit vector instructions, which perform 16 single precision operations or 8 double precision operations in a single instruction. Each core has a 32KB L1 instruction cache and 32KB L1 data cache, and a 512KB L2 cache. There is no cache shared across cores. The CPU clock is about 1.1GHz. With 61 cores and 16-wide vector operations, the IXPC has the equivalent of 976 GPU cores.

The control unit can issue two instructions per clock, though instructions are issued in-order and those two instructions must be from the same thread (unlike Intel Hyperthreading). Each core stores the state of up to four threads, using multithreading to tolerate cache misses.

The IXPC is packaged as an IO device, similar to a GPU. It has 8GB memory in current devices. The big advantage to the IXPC is its programmability. In many if not most cases, you really can just recompile your program and run it natively on an IXPC using MPI and/or OpenMP; however, to get the best performance, you will likely have to tune or refactor your application.

2.5 Common Accelerator Architecture Themes

There are several common themes across the Tesla, Radeon and Xeon Phi accelerators.

Separate device memory: The current devices are all packaged as an IO device, with separate memory from the host. This makes memory management key to achieving high performance. Partly this is because the transfer rates across the IO bus are so slow relative to memory speeds, but partly this is because the device memory is not paged and is only a few GB. When a workstation or single cluster node typically has 64GB or more of physical memory, having less memory on the high throughput accelerator presents a challenge. The limited physical memory sizes and relatively slow individual core speeds on today's accelerators also limits their ability to be viewed or used as standalone compute nodes. They are only viable when coupled to and used in concert with very fast mainstream CPUs.

Many cores: Our definition of *core* here is a processor that issues instructions across one or more functional units, and shares resources closely amongst those units. An NVIDIA SMX unit or Radeon Compute unit is a *core*, by this definition. With 15, 32 or 61 cores, your program needs enough parallelism to keep these units busy.

Somewhat unique to each device is how each organizes the cores and functional units within each core. The Radeon GPU has up to 32 compute units where each core has four SIMD units. One could argue that this should be treated as 128 cores. PGI has chosen to treat this like 32 cores, where each core can issue four instructions per clock, one to each SIMD unit. Similarly, PGI treats the Kepler SMX as a single core which can issue many instructions in a single cycle to many functional units.

Multithreading: All three devices use multithreading to tolerate long latency operations, typically memory operations. This is a trade-off between storing more thread state on board instead of implementing a more complex control unit. This means your application will have to expose an additional degree of parallelism to fill the multithreading slots, in addition to filling the cores.

The degree of multithreading differs across the devices. The IXPC has a lower degree (4) than Kepler (64) or Radeon (40), largely because the IXPC depends more on caches to reduce the average memory latency.

Vectors: All three devices have wide vector operations. The physical SIMD parallelism in all three designs happens to be the same (16), though the instruction set for Kepler and Radeon are designed with 32- and 64-wide vectors. Your application must deliver both SIMD and MIMD parallelism to achieve peak performance.

In-order instruction issue: This is really one of several common design issues that highlight the simpler control units on these devices relative to commodity CPU cores. If a goal is higher performance per transistor or per square nanometer, then making the control unit smaller allows the designer to use more transistors for the functional units, registers or caches. However, commodity cores benefit from the more complex control units, so clearly there is value to them. As a result, the cores used in accelerators may prove insufficient on the workloads where a commodity CPU thrives, or on the serial portions of workloads that overall are well-suited to an accelerator.

Memory strides matter: The device memories are optimized for accesses to superwords, a long sequence of adjacent words; a superword essentially corresponds to a cache line. If the data accesses are all stride-1, the application can take advantage of the high hardware memory bandwidth. With indexed accesses or high strides, the memory still responds with 256- or 384- or 512-bit superwords, but only a fraction of that data bandwidth is used.

Smaller caches: The cache memories are smaller per core than today's high performance CPUs. The applications where accelerators are used effectively generally operate on very large matrices or data structures, where cache memories are less effective. The trade-off between smaller cache and higher memory bandwidth targets this application space directly.

At PGI, we represent these common elements in an abstracted CPU+Accelerator architecture that looks as follows:

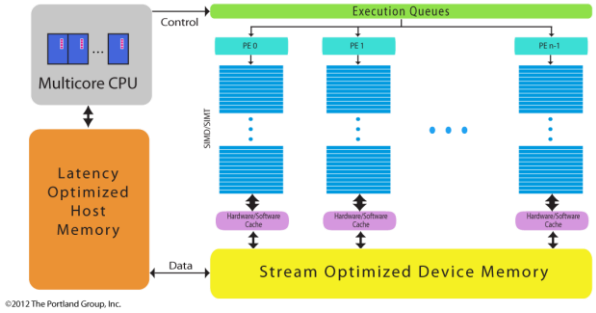

Figure 1: CPU+Accelerator Abstract Machine Architecture

3. Exploiting Accelerator Performance

Here we review and discuss the various means to program for today's accelerators, focusing on how the various important architectural aspects are managed by the programming model. A programming model has several options for exploiting architectural features: hide it, virtualize it, or expose it.

The model can hide a feature by automatically managing it. This is the route taken for register management for classical CPUs, for instance. While the C language has a *register* keyword, modern compilers ignore that when performing register allocation. Hiding a feature relieves the programmer from worrying about it, but also prevents the programmer from optimizing or tuning for it.

The model can virtualize a feature by exposing its presence but virtualizing the details. This is how vectorization is classically handled. The presence of vector or SIMD instructions is explicit, and programmers are expected to write vectorizable inner loops. However, the native vector length, the number of vector registers and the vector instructions themselves are all virtualized by the compiler. Compare this to using SSE intrinsics and how that affects portability to a future architecture (with, say, AVX instructions). Virtualizing a feature often incurs some performance penalty, but also improves productivity and portability.

Finally, a model can expose a feature fully. The classical example is how using MPI exposes the separate compute nodes, distributed memory and network data transfers within a program. The C language exposes the linear data layout, allowing pointer arithmetic to navigate a data structure. Exposing a feature allows the programmer to tune and optimize for it, but also requires the programmer to do the tuning and optimization.

The programming models we will discuss are OpenCL [7], CUDA C and Fortran, Microsoft's C++AMP [3], Intel's offload directives [4] (which are turning into the OpenMP 4.0 *target* directives [5]), and OpenACC [6].

3.1 Memory Management

Memory management is the highest hurdle and biggest obstacle to high performance on today's accelerators. Memory has to be allocated on the device, and data has to be moved between the host and the device. This is essentially identical across all the current devices. All attempts to automatically manage data movement between devices or nodes of a cluster result in great performance for simple stunt cases, but disappointing results in general. The costs of a bad decision are simply too high. For that reason, all of these programming models delegate some aspects of memory management to the programmer.

CUDA C [9] and Fortran [8] require the programmer to allocate device memory and explicitly copy data between host and device. CUDA Fortran allows the use of standard allocate and array assignment statements to manage the device data.

Similarly, OpenCL requires the programmer to allocate buffers and copy data to the buffers. It hides some important details, however, in that it doesn't expose exactly where the buffer lives at various points during the program execution. A compliant implementation may

(and may be required to) move a buffer to host memory, then back to device memory, with no notification to the programmer.

C++AMP uses *views* and *arrays* of data. The programmer can create a *view* of the data that will be processed on the accelerator. As with OpenCL, the actual point at which the memory is allocated on the device and data transferred to or from the device is hidden. Alternatively, the programmer may take full control by creating an explicit new *array*, or create an *array* with data copied from the host.

The Intel offload directives and OpenACC allow the programmer to rely entirely on the compiler for memory management to get started, but offer optional data constructs and clauses to control and optimize when data is allocated on the device and moved between host and device. The language, compiler and runtime cooperate to virtualize the separate memory spaces by using the same names for host and device copies of data. This is important to allow efficient implementations on systems where data movement is not necessary, and to allow the same program to be used on systems with or without an accelerator.

3.2 Parallelism Scheduling

These devices have three levels of compute parallelism: many cores, multithreading on a core, and SIMD execution.

CUDA and OpenCL expose the many-core parallelism with a three-level hierarchy. A kernel is written as a scalar thread; many threads are executed in parallel in a rectangular *block* or *workgroup* on a single core; many *blocks* or *workgroups* are executed in parallel in a rectangular *grid* across cores. The number of scalar threads in a *block* or *workgroup*, and the number of *blocks* or *workgroups* are chosen explicitly by the programmer. The fact that consecutive scalar threads are actually executed in SIMD fashion in warps or wavefronts on a GPU is hidden, though tuning guidelines give advice on how to avoid performance cliffs. The fact that some threads in a *block* or *workgroup* may execute in parallel using multithreading on the same hardware, or that multiple *blocks* or *workgroups* may execute in parallel using multithreading on the same hardware is also hidden. The programming model virtualizes the multithreading aspect, using multithreading for large *blocks* or *workgroups*, or many *blocks* or *workgroups*, until the resources are exhausted.

C++AMP does not expose the multiple levels of parallelism, instead presenting a flat parallelism model. The programmer can impose some hierarchy by explicitly using *tiles* in the code, but the mapping to the eventual execution mode is still implicit.

Intel's offload directives use OpenMP parallelism across the cores and threads on a core. The number of threads to launch can be set by the programmer using OpenMP clauses or environment variables, or set by default by the implementation. SIMD execution on a core is implemented with loop vectorization.

OpenACC exposes the three levels of parallelism as *gang*, *worker* and *vector* parallelism. A programmer can use OpenACC loop directives to explicitly control which loop indices are mapped to each level. However, a compiler has some freedom to automatically map or remap parallelism for a target device. For instance, the compiler may choose to automatically vectorize an inner loop. Or, if there is only a single loop, the compiler may remap the programmer-specified *gang* parallelism across gangs, workers and vector lanes, to exploit all the available parallelism. This allows the programmer to worry more about the abstract parallelism, leaving most of the device-specific scheduling details to the compiler.

OpenACC also allows the programmer to specify how much parallelism to instantiate, similar to the *num_threads* clause in OpenMP, with *num_gangs*, *num_workers* and *vector_length* clauses on the parallel directive, or with explicit sizes in the kernels *loop* directives. The best values to use are likely to be dependent on the device, so these should usually be left to the compiler or runtime system.

3.3 Multithreading

All three devices require *oversubscription* to keep the compute units busy; that is, the program must expose extra (slack) parallelism so a compute unit can swap in another active thread when a thread stalls on memory or other long latency operations. This slack parallelism can come from several sources. Here we explore how to create enough parallelism to keep the devices busy.

In CUDA and OpenCL, slack parallelism comes from creating *blocks* or *workgroups* larger than one *warp* or *wavefront*, or from creating more *blocks* or *workgroups* than the number of cores. Whether the larger blocks or workgroups are executed in parallel across SIMD units or time-sliced using multithreading is hidden by the hardware.

C++AMP completely hides how parallelism is scheduled. The programmer simply trusts that if there is enough parallelism, the implementation will manage it well on the target device.

Intel's offload model doesn't expose multithreading explicitly. Multithreading is used as another mechanism to implement OpenMP threads on the device. The programmer must create enough OpenMP threads to populate the cores, then create another factor to take advantage of multithreading for latency tolerance.

OpenACC *worker*-level parallelism is intended to address this issue directly. On the GPUs, iterations of a *worker*-parallel loop will run on the same core, either simultaneously using multiple SIMD units in the same

core, or sharing the same resources using multithreading. As mentioned earlier, an OpenACC implementation can also use longer vectors or more gangs and implement these using multithreading, similar to the way OpenCL or CUDA would.

3.4 SIMD operations

All three current architectures have vector or SIMD operations, but they are exposed differently. The IXPC has a classical core, with scalar and vector instructions. The Kepler is programmed in what NVIDIA calls SIMT fashion, that is, as a number of independent threads that happen to execute in lock-step as a warp; the SIMD implementation is virtualized at the hardware level. The Radeon also has scalar and vector units, but the scalar unit is limited; the Radeon is actually programmed much like the Kepler, with thread-independent branching relegated to the scalar unit.

As mentioned above, CUDA and OpenCL hide the SIMD aspect of the target architecture, except for tuning guidelines. C++AMP also completely hides how the parallelism of the algorithm is mapped onto the parallelism of the hardware.

Intel's offload model uses classical SIMD vectorization within a thread. New directives also allow more control over which loops are vectorized. The SIMD length is (obviously) limited to the 512-bit instruction set, for the current IXPC, but any specification of vector length is not part of the model.

An OpenACC compiler can treat all three machines as cores with scalar and vector units. For Kepler, vector operations are spread over the CUDA threads of a warp or thread block; scalar operations are either executed redundantly by each CUDA thread, or, where redundant execution is invalid or inefficient, executed in a protected region only by CUDA thread zero. Code generation for Radeon works much the same way. The native SIMD length for Kepler is 32 and for Radeon is 64, in both single and double precision. The compiler has the freedom to choose a longer vector length, multiples of 32 or 64, by synchronizing the warps or wavefronts as necessary to preserve vector dependences. Thus the OpenACC programming model for all three devices is really multicore plus vector parallelism. The compiler and runtime manage the differences between the device types.

3.5 Memory Strides

All three devices are quite sensitive to memory strides. The memories are designed to deliver very high bandwidth when a program accesses blocks of contiguous memory.

In OpenCL and CUDA, the programmer must write code so that consecutive OpenCL or CUDA threads access consecutive or contiguous memory locations. Since these threads execute in SIMD fashion, they will issue memory operations for consecutive locations in the same cycle, to deliver the memory performance we want.

In C++AMP, the mapping of parallel loop iterations to hardware parallelism is virtualized. If the index space is one dimensional, the programmer can guess that adjacent iterations will be executed together and should then access consecutive memory locations.

The Intel offload model has no need to deal with strides. The programmer should write the vector or SIMD loops to access memory in stride-1 order.

Similarly, using OpenACC, the programmer should write the vector loops with stride-1 accesses. When mapping a vector loop, the natural mapping will assign consecutive vector iterations to adjacent SIMD lanes, warp threads or wavefront threads. This will result in stride-1 memory accesses, or so-called *coalesced* memory accesses.

3.6 Caching and Scratchpad Memories

All three devices have classical cache memories. For the IXPC, classical cache optimizations should be as effective as for commodity multicore systems, except there is no large shared L3 cache. For the GPUs, the small hardware cache coupled with the amount of parallelism in each core makes it much less likely that the caches will be very effective for temporal locality. These devices have small scratchpad memories, which operate with the same latency as the L1 cache.

In CUDA and OpenCL, the programmer must manage the scratchpad memory explicitly, using CUDA **`__shared__`** or OpenCL **`__local`** memory. The size of this memory is quite small, and it quickly becomes a critical resource that can limit the amount of parallelism that can be created on the core.

C++AMP has no explicit scratchpad memory management, but it does expose a *tile* optimization to take advantage of memory locality. The programmer depends on the implementation or the hardware to actually exploit the locality.

Intel's offload model has no scratchpad memory management, since the target (the IXPC) has no such feature.

OpenACC has a cache directive to allow the programmer to tell the implementation what data has enough reuse to cache locally. An OpenACC compiler can use this directive to store the specified data in the scratchpad memory. The PGI OpenACC compilers also manage the scratchpad memory automatically, by analyzing the program to find temporal or spatial data reuse within and across *workers* and *vector* lanes in the same *gang* to choose to store data in the scratchpad memory.

3.7 Portability

There are three levels of portability. First is language portability, meaning a programmer can use the same language to write a program for different targets, even if the programs must be different. Second is functional portability, meaning a programmer can write one program that will run on different targets, though not all targets will get the best performance. Third is performance portability, meaning a programmer can write one program that gives good performance across many targets. For standard multi-core processors, C, C++ and Fortran do a pretty good job of delivering performance portability, as long as the program is written to allow vectorization and with appropriate memory access patterns. For portability on accelerators, the problems are significantly more difficult.

CUDA provides reasonable portability across NVIDIA devices. Each new architecture model adds new features and extends the programming model, so while old programs will likely run well, retuning or rewriting a program will often give even better performance. PGI has implementations of CUDA C and Fortran to run on an x86 multicore, but there is no pretense that these provide cross-vendor portability, or even performance portability of CUDA source code.

OpenCL is designed to provide language and functionality portability. Research has demonstrated that even across similar devices, like NVIDIA and AMD GPUs, retuning or rewriting a program can have a significant impact on performance [10]. In both CUDA and OpenCL, since the memory management is explicit in the program, data will be allocated and copied even when targeting a device like a multicore which has no split device memory.

C++AMP is intended to provide performance portability across devices. It is designed to do this by hiding or virtualizing aspects of the program that a programmer would tune for a particular target.

Intel's offload model makes no attempt to provide portability to other targets. Even when targeting a multicore, the mechanism is to ignore the offload directives entirely, using the embedded OpenMP for parallelism on the device. The successor, the OpenMP 4.0 *target* directives, are promoted to be portable across devices, but it is unlikely to be fully supported except on targets that can implement full OpenMP, such as a many core or DSP.

OpenACC is also intended to provide performance portability across devices, and there is some initial evidence to support this claim. The OpenACC parallelism model is more abstract than any of the others discussed here, except C++AMP, yet allows the programmer to expose different aspects of the parallel execution. An implementation can use this to exploit the different parallel features of a wide variety of target architectures.

4. Accelerator Architectures Tomorrow

Today's devices will be replaced in the next 12-24 months by a new generation, and we can speculate or predict what some of these future devices will look like.

AMD already has APU (Accelerated Processing Units) with a CPU core and GPU on the same die, sharing the same memory interface. Future APUs will allow the GPU to use the same virtual address space as the CPU core, allowing true shared memory. Current CPU memory interfaces do not provide the memory bandwidth demanded for high performance GPUs. Future APUs will have to improve this memory interface, or the on-chip GPU performance will be limited by the available memory bandwidth. This improvement could use new memory technology, such as stacked memory, to give both CPU and GPU very high memory bandwidth, or by using separate but coherent memory controllers for CPU-side and GPU-side memory.

NVIDIA has already announced plans to use stacked memory on future GPU dies. This should improve the memory bandwidth significantly. NVIDIA has also announced plans to allow the GPU to access system memory directly. If the GPU sits on the IO bus, such memory access will be quite slow relative to on-board memory, so this will not be a very broad solution. However, NVIDIA also has plans to package an NVIDIA GPU with ARM cores, which may give it many of the same capabilities as the AMD APU.

While Intel has not published a roadmap, one can imagine a convergence of the Intel Xeon Phi Coprocessor with the Intel Xeon processor line. Current IXPC products sit on the IO bus, but Intel presentations have shown diagrams with future IXPC and Xeon chips as peer processors on a single node.

In all three cases, it is possible or likely that the need to manage memory allocation and movement will diminish somewhat. However, we don't know how the accelerator architecture itself will evolve. The core count may increase, vectors may get longer (or not), other capabilities may be added. The clock speed may vary within a defined envelope as we prove out the most successful accelerator architectures. Other vendors may become viable candidates, such as the Texas Instruments or Tilera products. A successful programming model and implementation must effectively support the important accelerators available today, and be ready to support the accelerators we are likely to see tomorrow.

5. Our Conclusion

NVIDIA Tesla, Intel Xeon Phi and AMD Radeon have many common features that can be leveraged to create an Accelerator programing model that delivers language,

functional and performance portability across devices – separated and relatively small device memories, stream-oriented memory systems, many cores, multi-threading, and SIMD/SIMT. Features we would all like to see in such a model are many:

- Addresses a suitable range of applications
- Allows programming of CPU+Accelerator systems as an integrated platform
- Is built around standard C/C++/Fortran without sacrificing too much performance
- Integrates with the existing, proven, widely-used parallel programming models – MPI, OpenMP
- Allows incremental porting and optimization of existing applications
- Imposes little or no disruption on the standard code/compile/run development cycle
- Is interoperable with low-level Accelerator programming models
- Allows for efficient implementations on each of Tesla, Xeon Phi and Radeon

Admittedly, any programming model that meets most or all of these criteria is likely to short-change certain features of each of the important targets. However, in deciding which models to implement, compiler writers like those at PGI have to maximize those goals across all targets rather than maximizing them for a particular target. Likewise, when deciding which models to support and use, HPC users need to consider the costs associated with re-writing applications for successive generations of hardware and of potentially limiting the diversity of platforms from which they can choose.

To be successful, an Accelerator programming model must be low-level enough to express the important aspects of parallelism outlined above, but high-level and portable enough to allow an implementation to create efficient mappings of these aspects to a variety of accelerator hardware targets.

Libraries are one solution, and can be a good solution for applications dominated by widely-used algorithms. CUDA and OpenCL give control over all aspects of Accelerator programming, but are low-level, not very incremental, and come with portability issues and a relatively steep learning curve. C++ AMP may be a good solution for pure Windows programmers, but relies heavily on compiler technology to overcome abstraction, is not standardized or portable, and for HPC programmers offers no Fortran solution. The OpenMP 4.0 *target* and *simd* extensions are attractive on the surface, but come with the Achilles heel of requiring support for all of OpenMP 3.1 on devices which are ill-suited to support it.

In our view, OpenACC meets the criteria for long-term success. It is orthogonal to and interoperable with MPI and OpenMP. It virtualizes into existing languages the concept of an accelerator for a general-purpose CPU where these have potentially separated memories. It exposes and gives the programmer control over the biggest current performance bottleneck – data movement over a PCI bus. It virtualizes the mapping of parallelism from the program onto the hardware, giving both the compiler and the user the freedom to create optimal mappings for specific targets. And finally, it was designed from the outset to be portable and performance portable across CPU and accelerator targets of a canonical form − a MIMD parallel dimension, a SIMD/SIMT parallel dimension, and a memory hierarchy that is exposed to a greater or lesser degree and which must be accommodated by the programmer or compiler.

About the Authors

Brent Leback is an Engineering Manager for PGI. He has worked in various positions over the last 25 years in HPC customer support, math library development, applications engineering and consulting at QTC, Axian, PGI and STMicroelectronics. He can be reached by e-mail at *brent.leback@pgroup.com*.

Douglas Miles is the Director of PGI; prior to joining PGI, he was an applications engineer at Cray Research Superservers and Floating Point Systems. He can be reached by e-mail at *douglas.miles@pgroup.com*.

Michael Wolfe joined PGI as a compiler engineer in 1996; he has worked on parallel compilers for over 35 years. He has published one textbook, *High Performance Compilers for Parallel Computing*, and a number of technical papers. He can be reached by e-mail at *michael.wolfe@pgroup.com*.

References

[1] B. Cloutier, B.K. Muitey, P. Riggez, Dept of Electrical Engineering and Computer Science, University of Michigan, *Performance of FORTRAN and C GPU Extensions for a Benchmark Suite of Fourier Pseudospectral Algorithms*, arXiv:1206.3215v2, August 14, 2012

[2] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey, JARA, RWTH Aachen University, Germany Center for Computing and Communication, *OpenACC — First Experiences with Real-World Applications*, Euro-Par 2012, LNCS 7484, pp. 859–870, 2012.

[3] D. Moth, A *Code-Based Introduction to C++ AMP*, MSDN Magazine, April 2012.

[4] J. Jeffers, J. Reinders, *Intel Xeon Phi Coprocessor High Performance Programming*, Morgan Kaufmann, 2013.

[5] OpenMP ARB, *OpenMP Application Program Interface*, Version 4.0 – RC 2 – March 2013.

[6] OpenACC ARB, *The OpenACC Application Programming Interface*, Version 1.0, November, 2011.

[7] Khronos OpenCL Working Group, *The OpenCL Specification*, Version 1.2, November 2011.

[8] The Portland Group, Inc., *CUDA Fortran Programming Guide and Reference*, March 2011.

[9] NVIDIA Corp., *CUDA C Programming Guide*, October 2012.

[10] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, J. Dongarra, *From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming*, Parallel Computing, August 2012.

[11] NVIDIA Corp., *NVIDIA's Next Generate CUDA Compute Architecture: Kepler GK110*, 2012.

[12] AMD Corp., *Southern Islands Series Instruction Set Architecture*, August 2012.

[13] Intel Corp., *Intel Xeon Phi Coprocessor Instruction Set Architecture Reference Manual*, September, 2012.

[14] A. Herdman and W. Gaudin, *An Accelerated, Distributed Hydro Code with MPI and OpenACC*, Cray Technical Workshop on XK6 Programming, October, 2012

[15] A. Hart, R. Ansaloni, A. Gray, *Porting and scaling OpenACC applications on massively-parallel, GPU-accelerated supercomputers*, The European Physical Journal Special Topics, August 2012.

[16] M. Colgrove, *5x in 5 Hours: Porting a 3D Elastic Wave Simulator to GPUs Using OpenACC*, PGInsider Newsletter, March 2012.

[17] J. Levesque, R. Sankaran, R. Grout, *Hybridizing S3D into an Exascale application using OpenACC: An approach for moving to multi-petaflops and beyond*, SC12, November, 2012.