Directive-based Programming for Highly-Scalable Nodes

Douglas Miles, Michael Wolfe
PGI / NVIDIA

ABSTRACT: *High-end supercomputers have increased in performance from about 5 TFLOPS to 34 PFLOPS in the past 15 years, a factor of about 7,000. Increased node count accounts for a factor of 20 to 30, and clock rate increases for another factor of 5. Most of the increase, a factor of 50 to 100, is due to increases in single-node performance. We expect this trend to continue with single-node performance increasing faster than node count, and with that performance increase driven by increased parallelism. Building scalable applications for such targets means exploiting as much intra-node parallelism as possible. We discuss coming supercomputer node designs and how to abstract the differences to enable design of portable scalable applications in OpenACC and OpenMP.*

KEYWORDS: Compiler, Accelerator, Multicore, GPGPU, Parallelization, Vectorization

1. Introduction

High-end supercomputers have increased in performance from maximum of about 5 TFlops to 34 PFlops in the past 15 years, according to the Top 500 Supercomputer list [TOP15]. This increase is about a factor of 7,000, and has been driven by 3 factors. Node counts have increased from a range of 500 to 10,000 in the fastest computers 15 years ago, to maximums of over 80,000 today. Another factor is faster processor clocks, from about 400MHz to just under 3GHz today. Together these account for an average factor on the order of about 100X. Another hard-to-measure factor is improvements in microarchitecture, such as deeper pipelines and multiple-instruction issue. The rest of the performance improvement is due to increased parallelism on a node, through SIMD instructions, multicore processors, multiple processor sockets and/or accelerator sockets. Given recent announcements indicating the likely *decrease* in typical processor clock rates [HO16], the push to Exascale computing will be driven by a further increase in on-node parallelism. This is evidenced by the CORAL system designs, which have node counts commensurate with or smaller than current systems while delivering about a 10X performance increase.

The number and type of processors of a node, together with the core count of each processor and SIMD capability of each core, comprise the parallelism profile of that machine. Future machines will have widely different parallelism profiles. To take advantage of these highly parallel nodes without tuning to any single machine requires writing applications so as to expose as much parallelism as possible, without expressing it in a way that over-specifies how it is exploited on different systems.

This requires a mechanism to express parallel programs, as well as compilers or libraries that can map the available software parallelism onto each target system, and profiling and analysis tools to predict and measure the effectiveness of the parallelism in a given application for each target machine. We look at the common directive-oriented programming models for on-node parallel programming, OpenMP [OMP15] and OpenACC [OACC15]. We provide specific descriptions of how we expect to implement certain constructs in these models in PGI compilers targeting GPUs, multi-core CPUs and Xeon Phi, including options we have explored or are exploring. We describe which constructs are necessary and recommended, and those that should be avoided when designing applications for scalable nodes. We also discuss aspects of these new architectures that are not effectively managed by current approaches, and which need further development in the programming models.

2. Node Architecture Features

To motivate this work, we discuss three specific node architectures that represent common current systems and those we expect in the near future. We discuss the hardware features that relate to compute parallelism as well as data management.

2.1 Multi-core CPU Nodes

The first is a CPU node, with one to four multicore CPU sockets, each socket with a memory controller and a subset of the shared memory, and all sockets and cores cooperating with hardware cache coherence. Typical CPUs used in HPC systems today include Intel Xeon and IBM OpenPOWER. We expect ARMv8 CPUs to be used in HPC systems in the near future.

Parallelism across cores is exploited by creating threads in the operating system (POSIX threads), and scheduling one or more threads per core. This allows the same parallelism mechanism to exploit cores both within and across sockets. Synchronization and communication between threads is implemented in the shared memory, using locks (or transactional memory) to protect small critical sections that update shared data structures. This requires no additional processor state or operating system support for each thread, but unfortunately provides no optimized hardware for synchronization. In many current designs, the cores also support 2-8 hardware threads per core, either using simultaneous or temporal multithreading. Often, each core will have a private level-1 and level-2 cache; sometimes, small groups of 2-4 cores share a level-2 cache. Sometimes there is a large level-3 cache as well.

The cores on all CPUs mentioned above have SIMD instructions to perform 2 to 16 operations on SIMD register data in a single instruction. SIMD instructions are an inexpensive mechanism to improve compute throughput at minimal hardware cost. An Intel Broadwell processor with 22 cores and AVX3 instructions has 88 FP64 hardware compute lanes.

Delivering high performance on a multi-core CPU node means scheduling at least one thread per core, and perhaps more than that to populate the hardware threads as well. It requires generation of SIMD instructions wherever possible. To minimize cache coherence traffic, threads assigned to different cores and especially different sockets must be scheduled to work on data that does not live in shared cache lines. That means scheduling high-stride loop iterations to different cores, low-stride loop iterations to threads on the same core, and stride-1 loops to SIMD instructions.

2.2 GPU-Accelerated Nodes

A GPU-accelerated node typically has one or a small number of multi-core CPU host processors, and one to eight GPU accelerators. The most commonly used GPU in HPC systems today is the NVIDIA Tesla. Parallelism on a GPU is exploited by launching a *kernel*, a self-contained sub-program and associated data with multiple levels of parallelism specified in the kernel itself. Current GPUs typically support three levels of parallelism: i) some number of compute engines (NVIDIA Streaming Multiprocessors), ii) within each compute engine are a large number of compute cores (NVIDIA CUDA cores) groups of which execute in lock-step, similar to SIMD mode but with greater flexibility, iii) a high degree of hardware-supported multithreading which allows the compute cores to be heavily oversubscribed as a means to tolerate main memory latency.

GPU hardware implements extremely fast thread creation and shutdown, and fast synchronization between compute cores in the same compute engine. GPU support for synchronization between threads in different compute engines is limited. When a GPU kernel is launched, the host CPU may continue executing, or may wait for the kernel to complete. Current GPUs have a physically and logically separate, relatively small, high bandwidth (HBW) memory. To compute on the GPU, data is moved from the large system memory to HBW memory. Data movement is pure overhead, so must be optimized to minimize its volume and frequency. GPUs run at a much slower clock rate than the fastest CPUs, and have smaller hardware caches. However, with more compute resources and hardware-assisted parallelism support, GPUs can often achieve performance improvements measured in factors over CPUs on highly data parallel applications.

An NVIDIA Pascal P100 GPU [NV16] has 56 compute engines, each with 32 FP64 floating-point units for 1792 FP64 hardware compute lanes. With deep multithreading, a running program can exploit many thousands of simultaneously active parallel operations.

Achieving high performance on a GPU node requires managing data traffic between system and HBW memory carefully, moving all data parallel operations to the GPU, and managing asynchronous CPU/GPU computation. On the GPU itself, high-stride loop iterations can be scheduled across the compute engines and stride-1 loops scheduled to the lock-step compute cores to maximize HBW memory bandwidth. In particular, it is possible to map stride-1 inner loops designed for SIMD-capable CPUs with reasonable efficiency on GPUs. This is important for purposes of creating and supporting programming models that enable performance portability.

2.3 Many Integrated Core (MIC) Nodes

Our third node is exemplified by the Intel Knights Landing (KNL) Xeon Phi MIC processor [SG16], which we consider only as a self-hosted processor (not a co-processor). A KNL processor is in many ways very like a CPU node, but there are many more cores (72), each with support for 4 hardware threads and two 512-bit SIMD units. This gives a KNL processor 1172 hardware FP64 compute lanes. The cores use an on-chip network for distributed cache coherence. To limit or control on-chip network traffic, the application may choose to divide the 72 cores into 4 quadrants so coherence messages only have to traverse the cores in a single quadrant. The KNL shares certain parallelism profile features with a GPU: massive parallelism, caches which are much smaller than on a CPU, and a relatively small attached HBW memory. As on a GPU, moving data between the system memory and HBW memory is pure overhead that must be minimized.

Extracting high performance from a KNL node will require much more parallel work than on a CPU node, and will likely require more programmer effort. There are more cores, so more threads must be created just to populate them. The SIMD instructions are twice as long, and there are two SIMD units per core. The caches are smaller, meaning there will be more cache misses and hardware multithreading will be even more important than on a CPU. This further shrinks the cache available per thread, so scheduling threads that share data to the same core will be even more important. As on a GPU, minimizing data movement between the system and HBW memory will be important. While the KNL can operate out of system memory directly, we expect that maximizing performance will require heavy use of the HBW memory.

2.4 Common Themes

Common themes across all of these no node types are multiple cores or compute engines, SIMD or SIMD-like execution, and multithreading support. The size of each hardware parallelism dimension varies greatly across the different processor and node types, and must be accommodated by the programmer, a compiler and programming model, or some combination of these.

3. Dynamic Scalability

A key goal for any modern parallel programming model is to enable writing of programs that perform well on nodes with only a few CPU cores, on fat nodes with multi-socket multicore processors, on KNL MIC processors, or on CPU+GPU processors. The FP64 compute lane counts for these nodes varies from under 100 to over 1700. Multi-GPU nodes can have even much larger numbers of lanes. Mapping a parallel loop efficiently onto the compute lanes of a multicore CPU or KNL requires spreading the iterations across threads, one per core or one per hardware thread, and further across the SIMD lanes in each core or hardware thread. For a GPU, the parallel loop iterations must be spread across the compute engines, and the compute cores within the engines, with enough *slack parallelism* to fill as many multithread slots as possible in order to tolerate memory latency.

For highest performance, a program should exploit all available hardware parallelism. Essentially, the goal is to fill all the compute lanes. Given that different node architectures have significantly different parallelism profiles, either the programmer or a compiler must map program parallelism to hardware parallelism. Program parallelism can either be implicit and descriptive, allowing mappings to a variety of types of hardware, or it can be explicit and prescriptive with the programmer directing exactly how it should be mapped to a given target.

When a program is written with parallelism that can be exploited across multiple hardware dimensions, we say it has *dynamic scalability*. A parallel loop with adjacent memory references may be best mapped across SIMD lanes. A high stride parallel loop may be best mapped across CPU cores, to minimize cache false sharing. A parallel loop that shares data across iterations may be best mapped across the hardware threads of a single core, to maximize cache utilization. If a program exposes more parallelism than the hardware can exploit, it can be dialed back or serialized to ensure efficient execution on targets that don't efficiently support over-subscription. Highly parallel programs can generally be scaled down for efficient execution on a modestly parallel or serial hardware target. It is difficult or impossible to scale a modestly parallel program up to run efficiently on a massively parallel hardware target.

4. Descriptive Parallelism

Implicit in the discussion of dynamic scalability is the ability to remap program parallelism to different hardware parallelism dimensions depending on the target. Here, we discuss how this is done with current and proposed languages: OpenMP, OpenACC, Fortran 2008 and the parallel execution policies being considered for C++17.

In OpenMP, mapping a parallel loop across threads is done with a **parallel do** directive (assuming Fortran). Mapping a single loop across both threads and SIMD lanes on a multicore is done with a **parallel do simd** directive. For a GPU, mapping a parallel loop across the compute engines is done with a **teams distribute** directive, because the compute engines can't efficiently synchronize as required by **parallel** threads (and on some targets can't synchronize at all). Mapping a parallel loop across the compute engines and the compute cores inside the engine could be done with a **teams distribute parallel do** directive, though this may depend on the compiler being used. It is obviously undesirable to write directives in different ways for different targets, or to use **ifdef** preprocessor selection for each parallel directive above each parallel loop.

One option is for the user community to decide on the preferred way to write a parallel loop (**teams distribute parallel do simd**, perhaps) and to push the implementations to agree on how to map the parallelism for each target. This starts to fall apart if there are nested parallel loops that cannot be collapsed. For a GPU target, the programmer may want an outer loop mapped across teams (**teams distribute**) and the inner loop(s) spread across the compute cores within a compute engine (**parallel do**). For a multicore or manycore target, the programmer may want the outer loop mapped across cores or threads (**parallel do**) and the inner loop mapped across the SIMD lanes (**simd**). Such a structure can't be expressed without preprocessing, and requires the programmer to select the mapping for each target.

OpenACC uses a more descriptive approach. The specification of a parallel loop is separated from the selection of the parallelism mapping. The user may select whether to map a loop to **gang**, **worker** or **vector** parallelism dimensions, or may leave this to the compiler. This allows a single program, without preprocessing, to exploit the different parallelism profiles of a multicore, manycore, or GPU processor.

Fortran 2008 [FTN10] added the **do concurrent** construct. The standard declares that the execution of the iterations "may occur in any order," including in parallel, and that "the loop iterations have no interdependencies." It does not specify how the loop must be executed. Rather, it gives properties of the loop that must be true, allowing the implementation to parallelize or otherwise optimize the loop execution. The compiler may vectorize the loop, or software pipeline it, or run it in parallel using multiple hardware cores, or using hardware threads within a core, or any combination of these that is appropriate for the target. The **do concurrent** construct provides the dynamic scalability needed for effectively targeting multiple architectures. Fortran 2015 [FTN15] also adds support for private and shared data annotations on **do concurrent**, but notably does not add support for reductions.

C++17 [CPP15] adds extensions for parallelism, expressed as execution policies on common C++ *algorithms*. There are about 75 of these algorithms, including sort, finding the maximum element, and so on. For this discussion, the key algorithm is **for_each**, which functions much like a loop, invoking a function or lambda on each value of an index range. Invoking the **for_each** algorithm with a **par** or **par_vec** execution policy will tell the implementation that parallel execution is legal. In particular, specifying the **par_vec** policy allows any execution ordering, much like the Fortran **do concurrent**, including interleaving on a single thread, as with software pipelining. As with **do concurrent**, the **par_vec** policy seems to provide the dynamic scalability needed to effectively target multiple parallelism profiles effectively.

As hardware parallelism and complexity increases, including multiple dimensions of parallelism, more parallelism must be exposed and exploited in software. The software to hardware mapping problem becomes increasingly challenging. Creating efficient mappings requires knowledge of the characteristics of the program as well as the target hardware. For the languages we are considering in this discussion, a compiler processes a program for execution on a specific hardware target. This is a key advantage of compilers and perhaps the strongest argument for pushing the mapping problem as much as possible into a compiler. Another argument is that while programmers can often manually create mappings that are more efficient than a compiler, compilers are tireless. A compiler that delivers mappings with reasonable efficiency on a consistent basis provides a huge boost to programmer productivity.

This is exemplified by the inner workings of vectorizing compilers, which have been common for more than four decades [Co73, LC91]. A vectorizing compiler determines whether to generate vector or SIMD instructions for a given loop by analyzing both legality and profitability. Most compiler literature on this topic focuses on the legality analysis, such as flow graph and data dependence analysis.

However, in some cases SIMD code is slower than the corresponding sequential code. For instance, if the code requires gather or scatter operations but the instruction set has no SIMD gather instruction, simulating the gather operation may be slower than just executing the loop sequentially. Compilers use both compile-time heuristics and dynamic runtime checks to drive alternate code paths and optimize the chances that an optimal code sequence is executed. This profitability analysis for vectorizing compilers is carefully tuned for the target architecture. This technology has been successful enough that it's now relatively uncommon for HPC programmers to manually vectorize loops using SIMD intrinsics.

The same characteristics and opportunities hold true for generating parallel code in general, but the mapping problem is more complex. Not only must the programmer or the compiler decide if parallel code is legal, but also whether it is profitable and how to map it efficiently onto the target hardware parallelism.

5. Bottlenecks to Scaling

To support performance portability, we must promote a programming style that enables applications to scale and perform well across all HPC target architectures. This means training users to write programs that will run well, and training compilers (and compiler implementers) how to compile these programs for high performance. Successful applications tend to live for many years and are used across many hardware targets and generations of HPC systems. As a consequence, programmers should focus on scalable parallel constructs which can be flexibly exploited on different targets and allow expression of enough parallelism to fill the many compute lanes that current and future processors will have.

Specifically, hindrances to scalable parallelism or any bottlenecks in the middle of a parallel construct should be avoided. Synchronization features were intentionally minimized in the design of OpenACC, and are limited to reductions and atomics. These are so commonly used in important applications as to be unavoidable, and modern hardware often adds features to maximize their efficiency.

OpenMP has a rich set of synchronization features, many of which were designed for systems with very modest parallelism in the form of a few SMP processors. These include OpenMP **single**, **master**, **critical**, **ordered** and **barrier** constructs, and use of unstructured locks in general. These OpenMP features should be avoided at all costs in the design and implementation of scalable programs.

The OpenMP **sections** and **tasks** constructs can generate parallelism, but it is hard to create and exploit scalable parallelism with these constructs alone. For **sections** the source code itself literally does not scale, and OpenMP **task**s have an implicit bottleneck in the task queue that limits scalability. These constructs should be used carefully, and only where the code in the section or in the task itself exploits scalable parallelism.

Encoding parallelism limits into your code will inhibit scalability as well. OpenMP clauses like **num_threads**, **num_teams**, and **safelen** should be avoided. Any values used may be optimal for some target and very wrong for others. If your current program needs these features, you should either explore a more scalable algorithm, or look for another dimension of parallelism in your application. The same holds true for use of the OpenACC clauses **num_gangs**, **num_workers** and **vector_length**.

6. Data Management

There are two aspects of data management in HPC: data structure design and access patterns, and the exposed memory hierarchy or memory pools in current and upcoming systems. We explore each briefly here in relation to programming models and compilers.

6.1 Data Structure Design

The key point for data structure design is that strides matter. All current machines use hardware data caches with a cache line size typically around 64 bytes. This means when the processor loads a word, the surrounding 8 words are loaded into the cache. If those other 7 words are not used before that cache line is evicted, the memory bandwidth used to load the cache line was mostly wasted. When designing a data structure, it should take into account how the data is being accessed. Arrays of structures are great for keeping logically-related data together, but if the data members are not accessed together, the logical relation should be questioned. Multidimensional arrays should be declared so that inner loops run across the low-stride dimension.

SIMD load and store instructions sometimes require data to be contiguous in memory. Some processors have SIMD gather and scatter instructions, but contiguous loads and stores are faster. GPUs may not have SIMD instructions, but the compute cores executing in lock-step will all issue memory instructions simultaneously. These operations will be fastest when the addresses are all contiguous in memory, taking best advantage of memory bandwidth and cache behavior. In that sense, the GPU lock-step compute cores can be treated very like SIMD lanes when designing algorithms and data structure access patterns.

6.2 Exposed Memory Hierarchy

The key characteristics for each level of a memory hierarchy are capacity, latency and bandwidth. Current systems with GPU-accelerated nodes have a large system memory attached to the CPU processor(s) and a smaller, high bandwidth (HBW) memory attached to each GPU. The upcoming KNL nodes have both a large system memory and a much smaller HBW memory attached to the processor. Both of these designs are much like a memory hierarchy, where the system memory has more capacity but the HBW memory has higher bandwidth. However, they are different from current cache or virtual memory hierarchies because an application can or in some cases must control placement or movement of data in the hierarchy. This is a challenge for programmers and compilers, since placement and movement necessarily must take into account the size of the smaller (HBW) memory, and that size may differ across systems and generations of hardware.

Two approaches towards memory placement and movement have been developed. The first is to create new allocation routines (in C or C++) or attributes (Fortran) to allocate memory in a given memory. For a GPU, examples are **cudaMalloc** for CUDA C [NV15] and the **device** attribute in CUDA Fortran [PGI11]. For a KNL node, examples are **hbw_malloc** (built on the more general memkind heap manager) and the **attributes fastmem** directive for Intel Fortran [SO15]. These approaches require programmers to modify programs for each target. If the program data structures do not fit in the HBW memory, the programmer is responsible for swapping blocks of data between HBW and system memory. If the next generation node has a larger (or smaller) HBW memory, the swapping tradeoffs may change.

A second approach is to enhance parallel programming directives with data management clauses. This is the approach taken by OpenACC, and which was later adopted in OpenMP 4+. The data clauses describe whether and how data objects should be moved to the HBW memory for use by the highly parallel compute engine (GPU or KNL). For earlier GPUs, explicit data movement by either the programmer or the compiler runtime support libraries was required for correct execution, since the GPU could not directly access the whole of system memory. The NVIDIA Pascal GPU with updated operating system and driver support will be able to move data automatically between system and HBW memory, much like virtual memory paging support.

In both OpenACC and OpenMP, if the parallel compute device can access system memory directly the data clauses can be ignored. In this case the hardware or system software support manages system memory access. A compiler implementation for a KNL or Pascal GPU may then use the data clauses as hints to the runtime to prefetch data to the HBW, while ignoring those directives entirely on a multicore node. In all of these cases, the data clauses become potential optimizations rather than requirements for correct program execution. This allows the programmer to focus on parallelism first, and then tune the data management as a second step.

7. OpenACC and OpenMP gaps

There are several notable gaps in both OpenACC and OpenMP with respect to future HPC nodes, which we list here briefly.

For accelerated compute nodes, OpenMP currently has a structure for distinguishing which parallel regions should be compiled for the multicore host (normal OpenMP **omp parallel** region) and which should be compiled for the accelerator (**omp target**). OpenACC has no such structure. OpenACC syntax is much simpler as a result, but there is currently no way for the programmer to direct where a given region should execute.

When a compute node has multiple accelerators, both OpenMP and OpenACC have inconvenient and difficult methods for sharing data and work across devices. OpenACC has an API routine to set the *current device*. OpenMP has a **device** clause, which takes an integer device number. We have experimented with both methods and find both of them hard to use. With current GPUs, where data placement is necessary for correct execution, having to continually select on which device to run, or to carry or compute an extra variable for the device ID, is very error prone. The situation may improve with devices that share the same virtual memory as the CPU, but false sharing on virtual memory pages may generate excessive memory traffic between devices.

On accelerated compute nodes, the multicore CPU itself may be considered and targeted as a compute device. A node with a single GPU could be treated as having two compute devices: the GPU and the multicore CPU. Neither OpenMP nor OpenACC include a good method to spread data and work across heterogeneous devices like this.

Threadprivate data is an OpenMP feature that allows parallel execution across procedures that access global data, such as Fortran **common** and **module** variables or C/C++ **extern** variables. OpenACC has no such concept. OpenMP requires that global **threadprivate** data be created for each thread. With highly parallel compute nodes, the number of threads and the overhead of **threadprivate** data will increase.

8. Conclusions

Achieving high performance requires a collaboration of effort between the programmer, the language designer, and the language implementer. Current and future highly parallel compute nodes require a much higher degree of parallelism in applications. The application writer must expose as much parallelism as possible, and drive toward more parallel and scalable algorithms. Application parallelism should be expressed descriptively, so that it can be effectively mapped onto different hardware parallelism profiles. The compiler and runtime must be tuned to exploit parallelism efficiently. It is far easier for a compiler and runtime to scale parallelism down to a level appropriate for the target, than for an application writer to modernize a program with more parallelism when a new machine is targeted.

SIMD parallelism has been and will continue to be important as distinguished from multicore parallelism. Effective SIMD parallelism requires contiguous (stride-1) memory references and that the parallel compute lanes mostly take the same branches (convergent execution). Current and future processors depend on a high degree of SIMD parallelism for performance, and data structures and algorithms should be designed to take advantage of this.

Many modern processor nodes include a high bandwidth memory attached to the highly parallel compute device. This adds a new dimension to data management, allocating data in or moving data to and from the HBW memory. With appropriate programming model support, use of this performance-critical resource can be considered as an optimization rather than as a requirement for correct program execution.

About the Authors

Doug Miles is director of PGI compilers & tools at NVIDIA; prior to joining PGI and later NVIDIA, 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 optimizing and parallel compilers for over 40 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

[BA15] C. Bertolli et al, *Integrating GPU Support for OpenMP Offloading Directives into Clang*, LLVM-HPC2015, Austin, TX, November 2015.

[CO73] W. Cohagan, *Vector Optimization for the ASC*, Seventh Annual Princeton Conf. on Information Sciences and Systems, Princeton, NJ, March 1973.

[CPP15] ISO/IEC, *Programming Languages–Technical Specification for C++ Extensions for Parallelism*, TS 19570, July 2015.

[FTN10] ISO/IEC JTC 1/SC 22/WG 5, *Information technology–Programming languages–Fortran*, 2010.

[FTN16] ISO/J3 WG5, *F2015 Working Document*, January 2016.

[HO16] W. Holt, *Moore's Law: A Path Forward*, Plenary presentation, ISSCC 2016, February 2016.

[LC91] D. Levine, D. Callahan, J. Dongarra, *A Comparative Study of Automatic Vectorizing Compilers*, Parallel Computing, 17:10-11, pp. 1223-1244, December 1991.

[NV15] NVIDIA Corp., *CUDA C Programming Guide*, September 2015.

[NV16] NVIDIA Corp., *NVIDIA Tesla P100*, 2016.

[OACC15] OpenACC Architecture Review Board, *OpenACC Application Programming Interface Version 2.5*, www.openacc.org, October 2015.

[OMP15] OpenMP Architecture Review Board, *OpenMP Application Programming Interface Version 4.5*, www.openmp.org, November 2015.

[PGI11] The Portland Group, *CUDA Fortran Programming Guide and Reference*, March 2011.

[SG16] A. Sodani et al, Knights Landing: Second-Generation Intel Xeon Phi Product, IEEE Micro, 16:2, pp. 34-46, March, 2016.

[SO15] A. Sodani, Knights Landing (KNL): 2nd Generation Intel Xeon Phi Processor, presentation at Hot Chips 2015, Cupertino, California, August 2015.

[TOP15] TOP 500 Supercomputer list, www.top500.org, accessed December, 2015.