PROCEEDINGS OF SPIE

SPIEDigitalLibrary.org/conference-proceedings-of-spie

Path forward for softwarization to tackle evolving hardware

Millad Ghane, Sunita Chandrasekaran, Robert Searles, Margaret Cheung, Oscar Hernandez

Millad Ghane, Sunita Chandrasekaran, Robert Searles, Margaret Cheung, Oscar Hernandez, "Path forward for softwarization to tackle evolving hardware," Proc. SPIE 10652, Disruptive Technologies in Information Sciences, 106520O (9 May 2018); doi: 10.1117/12.2304813

SPIE.

Event: SPIE Defense + Security, 2018, Orlando, Florida, United States

Path Forward for Softwarization to Tackle Evolving Hardware

Millad Ghane[a], Sunita Chandrasekaran[b], Robert Searles[b], Margaret Cheung[a,d], and Oscar Hernandez[c]

[a]University of Houston, Texas, USA
[b]University of Delaware, Newark, DE, USA
[d]Center for Theoretical Biological Physics, Rice University, USA
[c]Oak Ridge National Laboratory, Oak Ridge, TN USA

ABSTRACT

Keywords: High Performance Computing, Disruptive Technologies, Softwarization

1. INTRODUCTION

Compute nodes are evolving and becoming increasingly complex but at the same time offering several folds of parallelism. These nodes are heterogeneous at all levels including core, memory demanding smart and sophisticated programming approaches for exploiting various types of parallelism within algorithms. Going forward we will see KNL-type manycore design with vectorization along with conventional cores on a chip, GPU architectures will have more cores with improved shared memory technology between CPU and GPUs. GPUs with half-precision floats are also being introduced especially for deep learning.

Innovation at the node level diversifies High Performance Computing (HPC) systems. The HPC systems tend to differ from each other for a reason, not the least of which being not all scientific applications would benefit from one type of architecture. While the pre-exascale systems such as Summit at the Oak Ridge National Laboratory (ORNL) and Sierra at the Lawrence Livermore National Laboratory (LLNL) are built with IBM processors and NVIDIA's Volta graphic cards. The much-awaited exascale system, Aurora, at Argonne National Lab is expected to comprise of a Knights family massively parallel processor with some Xeon cores on the die as per recent discussions in.[1] Not to forget the world's fastest supercomputer, Sunway Taihulight in China uses Shenwei SW26010 processors while the K computer in Japan uses SPARC64 processors.

Disrupting technologies like neuromorphic and quantum computing add newer opportunities offer computing capabilities beyond today's classical systems. It is quite likely that these novel computing and learning paradigms will not replace HPC (at least not for the next several years) but they could complement HPC by acting as coprocessors thus allowing us to imagine newer applications and newer science that would be impossible with today's systems.

We need better software toolchains to achieve two goals. (a) To meet concurrency demands and massive on-node parallelism (b) To expose maximum parallelism in applications to software toolchains. To address these goals, we need to understand the intricacies of hardware architectures and maximally utilize their rich hardware capabilities. An application developer using legacy codes typically of hundreds to thousands of lines of code cannot be expected to have such in-depth knowledge of the architecture. This begs for high-level software abstractions. Such prescriptive directives can not only expose parallelism in the application but also provide smart hints to the compiler. By using such high-level directives, the application developers can incrementally improve existing code bases without needing to learn low-level languages or architectural details.

The impact of a software toolchain is measured with a combination of performance, portability and productivity. Performance indicates how much of the hardware architectures' peak performance can be achieved using the toolchain. Portability indicates the possibility of reusability of the solutions. Productivity is a measure of the effort of development denoting the time taken from designing to developing solutions for application. These three factors are strongly interconnected in a way that a proposed solution cannot compromise on the performance but still facilitate a portable and productive solution.

Further author information: (Send correspondence to Sunita Chandrasekaran)
Sunita Chandrasekaran: E-mail: schandra@udel.edu, Telephone: 1 302 831 2714

Disruptive Technologies in Information Sciences, edited by Misty Blowers, Russell D. Hall, Venkateswara R. Dasari
Proc. of SPIE Vol. 10652, 106520O · © 2018 SPIE · CCC code: 0277-786X/18/$18 · doi: 10.1117/12.2304813

2. STATE-OF-THE-ART

Programming these advanced HPC architectures has becoming increasingly important and posing a challenge to define 'X' in the MPI + X programming framework where 'X' is typically what the application developers demand. State-of-the-art shows several programming approaches defined for 'X'. Categorizing them into low-level and high-level programming approaches, some of them include Legion language (Regent),[2] where the programmer is expected to write Legion programs with some awareness of both logical and physical levels, this would require steep learning curve of the programming language that can be quite time consuming for application developers. Similarly, using Concurrent Collections (CnC)[3] will require an application domain expert to identify data and control dependencies in the application and captures them as related collections of computation steps, data items and control tags. Kokkos[4] and RAJA[5] rely on C++11 requiring applications to be rewritten thus not reducing the barrier to entry for scientific developers desiring to target large scale heterogeneous computing systems. Literature survey shows adoption of Kokkos for Molecular Dynamics, 3D unstructured mesh codes, and Tensor Math Library Kernels.

Some of the higher-level approaches include OpenMP,[6,7] a directive-based programming model that comes close to creating a portable software stack expressing parallelism in applications. Similar to OpenMP is OpenACC,[8] another directive-based programming model that offers high-level software abstractions to be used by application developers to port their applications to current HPC systems. Literature survey shows MPI+X-based model (X= OpenMP or OpenACC) has been adopted by legacy codes such as electromagnetics code NekCEM,[9] Community Atmosphere Model - Spectral Element (CAM-SE),[10] Combustion code, S3D[11] and second-order Mazller-Plesset perturbation theory (MP2).[12]

Although there are quite a number of programming frameworks to choose from, in most to all cases, there seems to be a systematic challenge, insofar as it is difficult for compiler teams to develop mature and performant compiler toolchains for processor hardware that is changing so frequently. As a result, application developers often resolve to using multiple programming models within a code. Ideally, they would prefer a single highly performant programming model with a high level abstraction that can target more than just one platform.

3. BACKGROUND

Given advancements in hardware, we need appropriate software abstractions to

- Reduce programmers' burden and improve productivity
- Propose language features at the right level of abstraction
- Achieve performance without compromising on portability
- Improve portability of scientific applications across platforms

We highlight two of our on-going work that focus on the above needs for software abstraction to achieve performance portable solutions. The two major applications are nuclear reactor modeling and molecular dynamics.

3.1 Porting a nuclear modeling miniapplication to extreme scale systems

This work explores parallelization and acceleration of S_n radiation transport[13] algorithms often found in nuclear reactor modeling. The large number of problem dimensions available in this type of transport algorithm provides opportunities to exploit parallelism on multicore and manycore parallel system. This particular algorithm exposes wavefront-based parallel patterns often found in other computational motifs such as dense and spare linear algebra, structured grids as well. To model the physical problem, requires modeling particle flux in all directions. To accomplish this, an execution instance of the algorithm performs a total of eight sweeps, one starting at each corner of the domain. These directions are referred to as octants. The results of all eight octant sweeps are added together to form the final result.

We used OpenACC to parallelize and accelerate this application on heterogeneous systems. We observed that this high-level directive-based programming model could not expose all the five levels of parallelism that

this type of algorithm offers. As a result, we were only able to exploit two levels of parallelism. Nevertheless, we achieved a speedup of 85.06x over the serial version of the code running on an NVIDIA state-of-the-art Volta GPU. This speedup is larger than a low-level programming language CUDA's speedup of 83.72x over the same serial implementation. The porting process of this application on the heterogeneous platform helped us determine that although OpenACC helped achieve comparable speedup to CUDA, the development effort required the programmer to refactor and retune the code to represent the wavefront parallel pattern exhibited by the application. This begs for the creation of software abstractions for such parallel patterns, that is found in other computational motifs as mentioned above.

3.2 Porting proxy application of molecular dynamics to extreme scale systems

This work explores Codesign Molecular Dynamics (CoMD), a proxy application of Molecular Dynamics to further identify programmatic gaps in current programming model. CoMD requires careful partition of data across the host CPU and the device that could be an accelerator such as the GPU. This will also introduce data inconsistency issues that needs to be tackled. Complexity of data transfers increases with the increasing levels of pointer indirections in programs.

This programmatic challenge led us to create a high-level language feature that would only allow copying portions of required data structures to and from the devices. Such a feature is currently identified as a gap within the popular on-node programming models of OpenMP and OpenACC. While there are strategies in place to make the whole data structure available on the device (like Unified Virtual Memory (UVM)), we needed to only make a subset of the structure reachable on the device. The requirement of such a feature is not MD application specific but such a scenario has been observed in other applications too. By applying or proposed OpenACC directives on the CoMD code, we were able to achieve 61% of that of the CUDA performance for one of the compute intensive kernels of CoMD.

4. PROPOSED METHODOLOGY

The porting process with both the applications discussed in the previous section indicated that the performance suffered from the lack of adequate expressibility of parallelism hidden in the applications by the programming model. In this section, we discuss our proposed abstract machine model and a programmatic representation of the model.

4.1 Abstract Machine Model

To facilitate the creation of such software abstractions, we need a better understanding of the hardware architecture. To that end, we need to create an abstract machine model (we will focus on the compute node than the full distributed system) that represents the hierarchies at the computation and memory level. Modern compute nodes have an execution hierarchy. They may be equipped with a single GPU or with multiple GPUs. They may consist of co-located memories that include GPU's local memory and its shared memory or high bandwidth memory. Threads also have characteristics based on location, e.g., thread synchronization across different cores of a node may be impossible or much slower compared to on-core synchronization. Likewise, memories at different levels have different speeds, and accesses to such memory may be affected, depending on the level, by NUMA phenomenon. It should be noted that these concepts are readily applied to the heterogeneous nodes as well as homogeneous ones.

Figure 1 demonstrates an example of our abstract machine model for a system equipped with two NUMA nodes and four connected GPUs. The root node (in this case "System" node) is the parent of its children ("NUMA" and "GPUs" nodes), and in turn, they are parents for their descendants ("CPUx" and "GPUx" nodes). Aside from representing the whole system, the system node contains our unified memory. The relationship chain among nodes helps us to determine that GPU nodes on the *leaves* of the tree can access memories of their parent (and their grandparents), while it is not true vice versa.

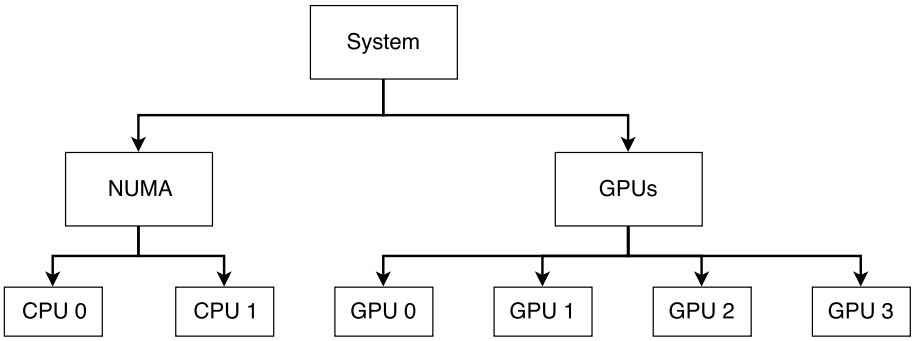

Figure 1. A sample of our model of a heterogeneous system with two NUMA nodes and four connected GPUs.

4.2 Proposed high-level software abstraction

We have developed *Chameleon*: a software abstraction based on directives to represent the hierarchies in our representative machine model. Listing 1 shows the manifestation of our sample model in Figure 1 within a programming language with the help of Chameleon directives. The following three steps are required to represent any models in Chameleon.

4.2.1 Representation of our abstract machine model

The first step is dedicated to the *definition of location types* with `dtype` clause in Chameleon. Developers have to define types of locations in their systems. Considering a modern supercomputer system, like Titan or Summit from ORNL, the three major (both memory and computation) components of any supercomputing machine are main processor (CPU), accelerators (like GPUs), and main memory. These components are defined in Lines 5-7 of Listing 1. The host is an Intel Skylake with x64 architecture and 4 MB of L3 cache, defined as `host`. The accelerator in our system is that of NVIDIA Volta GPU (with 4 GB of memory) type, defined as `accelerator`. Finally, our main memory is a 16GB unified memory, defined as `node_memory`. These location types will be used when we define our locations in the next step.

The second step is *location definition*. After defining location types, developers have to determine composition of available resources in their system. Lines 10-13 in Listing 1 refer to location definition with the help of `location` clause. Our underlying system has two CPUs of type `host` (Line 12), four GPUs of type `accelerator` (Line 13), and a big main memory for the entire system (Line 10). Developers can also group a subset of resources (mostly similar resources) into one. This ability will allow developers to address the whole subset as one instead of visiting them one by one. For instance, we may run a kernel on all resources of a group or any available resource that is idle. Such cases are prevalent in load-aware runtime libraries. In Chameleon, resource grouping is addressed with `ABSTRACTION` type, which is an internal `location type` to Chameleon. Abstract location types have no physical manifestation in the underlying system, and their purpose is to group many locations into one for organizational purposes. Line 11 denotes two of the abstract locations in our definitions, `NUMA` and `GPUs`.

The last step is devoted to describing the relationship among all locations in our system, *description of hierarchy*. The `hierarchy` clause in Chameleon provides such facilities to developers for laying out the relationship among locations. Figure 1 demonstrates such relationship among locations with arrows. Each location, as implied by arrows, is allowed to have only one parent while the number of children is not limited. Lines 16-18 demonstrates how the `hierarchy` clause in Chameleon is being used.

4.2.2 Memory allocation and kernel execution

Programs should be able to allocate memory and execute instructions on available resources. Memory allocation on Chameleon is performed with the help of `variable` clause. Line 2 in Listing 2 demonstrates how an array of `N` *double-precision* elements is allocated on location *System*. The `allocate` keyword specifies the allocation operation for the array. Based on the concept of hierarchy in Chameleon, such an allocation on location *System* leads to memory space available on all children in the *system*, and consequently, all of their grandchildren.

Listing 1. Manifestation of our abstract model in Figure 1 within C/C++ language

```
1  // Defining the location types
2  //      name:   the user-defined name of the location type
3  //      kind:   the architecture of the location type
4  //      mem:    the available memory on the location
5  #pragma chameleon dtype name(host) kind(x64, Skylake) num_cores(4) mem(4MB)
6  #pragma chameleon dtype name(accelerator) kind(CC3.0, Volta) mem(4GB)
7  #pragma chameleon dtype name(node_memory) kind(Unified_Memory) mem(16GB)
8
9  // Defining location definitions
10 #pragma chameleon location name(System) type(node_memory)
11 #pragma chameleon location name(NUMA, GPUs) type(ABSTRACTION)
12 #pragma chameleon location name(CPU0, CPU1) type(host)
13 #pragma chameleon location name(GPU0, GPU1, GPU2, GPU3) type(accelerator)
14
15 // Defining relationship among parent and children
16 #pragma chameleon hierarchy children(NUMA, GPUs) parent(System)
17 #pragma chameleon hierarchy children(CPU0, CPU1) parent(NUMA)
18 #pragma chameleon hierarchy children(GPU0, GPU1, GPU2, GPU3) parent(GPUs)
```

A kernel execution in Chameleon has two aspects to it: code generation and work distribution. The former one refers to the process of generating code for the available computational resources in the system. Currently, Chameleon relies on the compiler to generate codes. With the help of OpenACC directives and the PGI compiler that supports such directives, code generation will be done for both the host and the NVIDIA GPUs simultaneously at the compile time*. Afterwards, with OpenACC APIs, our library can switch to the desired device at the execution time. The latter aspect of kernel execution refers to the distribution of workload among locations. We plan to support many different execution policies in Chameleon, however, at this moment, our runtime library supports kernel execution at a specific location with `at` keyword in `region` clause. Line 5 in Listing 2 shows an example on how to run a kernel on location `GPU0`. In order to introduce all of the utilized pointers to Chameleon, one can use the `variables` keyword. With the help of this keyword, Chameleon can provide correct pointer to the kernel based on the selected device at the run time. The kernel described at Line 5 of Listing 2 will be offloaded to the first GPU in the system as specified by the developer. If we intend to run this kernel on one of the CPUs in the NUMA domain (e.g., `CPU0`), the only modification that this code requires is to replace `GPU0` with `CPU0`.

As we discussed before, the execution policy in Chameleon is not confined to the `at` policy (a unique physical location). We are planning to extend available policies and support other ones like `atany`, `ateach`, and so on. The `atany` policy accepts a subtree within our hierarchical abstract model and tries to issue the kernel on the first available/idle children. On the contrary, the `ateach` policy issues the kernel on all of the resources under the specified subtree. Every physical location has to run the kernel whether they are idle or not.

5. RESULTS AND ANALYSIS

5.1 Experimental setup

All experiments use UHPC cluster.[14] Located at the University of Houston, the UHPC cluster is hosting HPE Apollo XL190r Gen9 compute nodes. They are equipped with a dual Intel Xeon Processor E5-2660 v3 (10 cores) running at 2.6 GHz with 128 GB of memory. An NVIDIA Tesla K80 GPU with 12 GB of GDDR5 is connected through PCI-Express Gen3 to the compute nodes, which is capable of transferring 15.75 GB/sec between main memory and the GPU. We used PGI compiler version 17.5 and CUDA Toolkit 8.0 to build our codes.

*Invoking the PGI compiler with following flags: `-acc -ta=host, tesla`

Listing 2. Memory allocation and kernel execution within Chameleon

```
1  // Defining array X in Location 'System'
2  #pragma chameleon variable allocate(X[0:N]) type(double) location(System)
3
4  // Launching following kernel at the selected location
5  #pragma chameleon region at(GPU0) variables(X)
6  #pragma acc parallel loop collapse(2) independent
7  for( unsigned int i = 1; i < (nRows-1); i++ )
8  {
9      for( unsigned int j = 1; j < (nCols-1); j++ )
10     {
11         // Something to compute on 'X' array
12     }
13 }
14 #pragma chameleon region end
```

5.2 Preliminary Results and Analysis

We conducted a set of experiments to show the performance of our underlying hardware. The experiments we performed on UHPC are based on the following three kernels: stencil, matrix multiplication, and dot product (inner product). We decorated the source code of each application with Chameleon directives. By flipping the target device in Chameleon (`at` keyword in `region` clause), we are able to launch our kernels on any devices we targeted.

Each kernel is represented from the perspective of an application, and they are compiled with PGI compiler and `-acc -ta=tesla,multicore` flags. Such flags, in the OpenACC programming model, guarantee code generation for both NVIDIA GPUs and multicore architectures, respectively. In case we target to run our kernel on the host, the `multicore` mode will execute our code on all available cores of the host. We wrote two versions of the code: serial and Chameleon-based. The serial version, in each case, is the algorithm without parallelism decoration, like OpenMP, OpenACC, PThreads, and so on. The Chameleon-based version is based on our proposed directives in this paper.

We measured the speedup of each application with respect to the serial version. We use Linux timing functions (`gettimeofday()`) to perform the timing measurements and we measured wall clock time of the whole application. We chose to measure the whole lifetime of an application since one of the aspects of Chameleon is memory allocation; this includes the timing of such allocations too. For the serial applications, it includes allocating memory and running the kernel. For Chameleon-based applications, it includes required time for `allocate` clause to finish and then kernel execution by `region` clause.

The stencil benchmark is from the SHOC benchmark[15] suite. We included relevant Chameleon clauses in the source code. The matrix multiplication implementation is based on the naive *ijk-strategy*. In order to make the code cache-friendly to both architectures, the partial sum for the innermost loop is recorded in the local variable and then added to the output matrix. Finally, the dot product kernel is a loop that traverses all elements in both vectors and multiplies corresponding elements of the vector with each other. The final value is the result of the reduction clause of OpenACC on those multiplications.

Figure 2 shows the speedup results for our three benchmarks. We tested our applications with big input sizes to utilize all of the available memory resources on our devices. All the experiments are done with double-precision floating-point numbers. The matrix input size for stencil application is a $10,000 \times 10,000$. The multicore and tesla versions of the code (based on the Chameleon) have an advantage over the serial version as one can observe from the Figure 2. The measured speedup for both architectures is 4.7X and 4.5X, respectively. The reason why the GPU version is behind the multicore one is due to the complex access patterns of the stencil kernel. The access pattern is not cache-friendly, and naive cache structure in GPU architecture suffer more from such accesses. On the other hand, since matrix multiplication kernel is more cache-friendly on the GPU, we can observe a

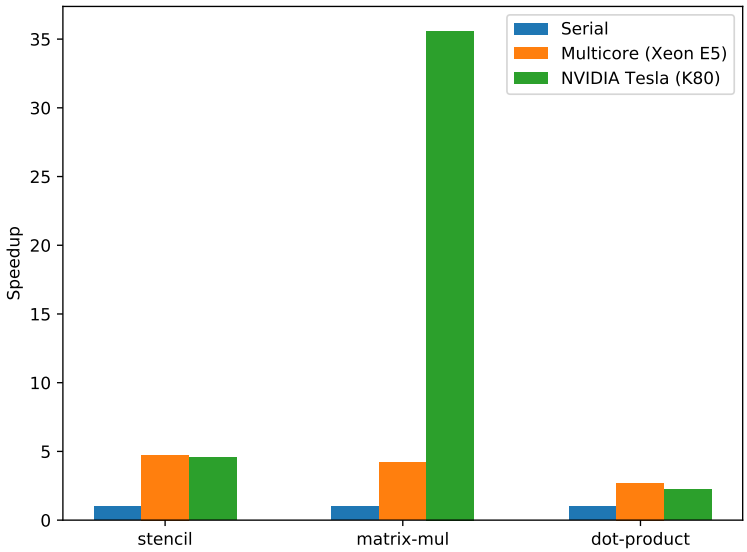

Figure 2. Speedup results

35.6X performance improvement. Every three matrices in our matrix multiplication kernels are a $2,000 \times 2,000$ matrix. And lastly, the dot product application is based on the two vectors with $80,000,000$ elements. Similar to the stencil kernel, the dot product kernel does not benefit from utilizing GPUs, in comparison to the multicore architecture, due to the overhead of data transfer over PCI-Express bus. If the floating-point computation on our data is fairly simple, we will not gain any performance and will suffer from the overhead of communications. Having said that, state-of-the-art architectures, like Volta architecture,[16] have improved significantly and most of the performance inefficiencies have been addressed.

6. CONCLUSION AND FUTURE WORK

This work highlights that the hardware is changing very rapidly and the software is still catching up. Although there are effective programming models and compiler tools for the application developers to use, there are still gaps in these models. These gaps are usually identified by application developers looking for particular software abstraction or features to port their code to target platforms. In the event, they do not find a relevant feature, the developers tend to refactor and retune their application manually in order to achieve the best performance on the hardware system. Such an approach gets hardware to adopt for larger applications that demand major refactoring. To that end, this work proposes a set of software abstractions, Chameleon, to address programmatic challenges in both the computation and at the memory level hierarchies of the systems. We evaluate this newly proposed abstraction using simple test cases. As on-going and near future work, we plan to gather more feedback on our abstractions from the programming model community and refine these abstractions to better cater to the needs of the application developers. We will also be evaluating our abstractions against real-world scientific applications to determine further gaps in the abstraction and relevant solutions to these gaps.

REFERENCES

[1] The Next Platform, "Argonne hints at future architecture of aurora exascale system." August 2017 https://www.nextplatform.com/2018/03/19/argonne-hints-at-future-architecture-of-aurora-exascale-system/. (Accessed: 28 March 2018).

[2] Slaughter, E., Lee, W., Treichler, S., Bauer, M., and Aiken, A., "Regent: a high-productivity programming language for hpc with logical regions," in [*2015 SC-International Conference for High Performance Computing, Networking, Storage and Analysis,*], 1–12, IEEE (2015).

[3] Budimlić, Z., Burke, M., Cavé, V., Knobe, K., Lowney, G., Newton, R., Palsberg, J., Peixotto, D., Sarkar, V., and Schlimbach, F., "Concurrent Collections," *Scientific Programming* **18**(3-4), 203–217 (2010).

[4] Edwards, H. C., Trott, C. R., and Sunderland, D., "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing* **74**(12), 3202–3216 (2014).

[5] Hornung, R. and Keasler, J., "The RAJA portability layer: overview and status," tech. rep., Lawrence Livermore National Laboratory (LLNL), Livermore, CA (2014).

[6] Chapman, B., Jost, G., and Van Der Pas, R., [*Using OpenMP: portable shared memory parallel programming*], vol. 10, MIT press (2008).

[7] van der Pas, R., Stotzer, E., and Terboven, C., [*Using OpenMP–The Next Step: Affinity, Accelerators, Tasking, and SIMD*], MIT Press (2017).

[8] Chandrasekaran, S. and Juckeland, G., [*OpenACC for Programmers: Concepts and Strategies*], Addison-Wesley Professional; 1 edition (2017).

[9] Otten, M., Gong, J., Mametjanov, A., Vose, A., Levesque, J., Fischer, P., and Min, M., "An mpi/openacc implementation of a high-order electromagnetics solver with gpudirect communication," *The International Journal of High Performance Computing Applications* **30**(3), 320–334 (2016).

[10] Bland, A. S., Wells, J. C., Messer, O. E., Hernandez, O. R., and Rogers, J. H., "Titan: Early experience with the cray xk6 at oak ridge national laboratory," in [*Proceedings of cray user group conference (CUG 2012)*], 3–4, Cray User Group Stuttgart, Germany (2012).

[11] Levesque, J. M., Sankaran, R., and Grout, R., "Hybridizing s3d into an exascale application using openacc: an approach for moving to multi-petaflops and beyond," in [*Proceedings of the International conference on high performance computing, networking, storage and analysis*], 15, IEEE Computer Society Press (2012).

[12] Kjærgaard, T., Baudin, P., Bykov, D., Eriksen, J. J., Ettenhuber, P., Kristensen, K., Larkin, J., Liakh, D., Pawłowski, F., Vose, A., et al., "Massively parallel and linear-scaling algorithm for second-order møller–plesset perturbation theory applied to the study of supramolecular wires," *Computer Physics Communications* **212**, 152–160 (2017).

[13] Plimpton, S., Hendrickson, B., Burns, S., and McLendon, W., "Parallel algorithms for radiation transport on unstructured grids," in [*Supercomputing, ACM/IEEE 2000 Conference*], 25–25, IEEE (2000).

[14] UHPC. https://uhpc-mri.uh.edu/ (2018). (Accessed: 03 March 2018).

[15] Pino, S., Pollock, L., and Chandrasekaran, S., "Exploring translation of OpenMP to OpenACC 2.5: Lessons Learned," in [*Proceedings of the Seventh International Workshop on Accelerators and Hybrid Exascale Systems (AsHES)*], IEEE Press (2017).

[16] NVIDIA, "NVIDIA Tesla V100 GPU Architecture - White paper." August 2017 http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. (Accessed: 03 March 2018).