LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-JRNL-789080

Sierra Center of Excellence: Lessons Learned

A. Black, A. Bertsch, J. Dahm, L. Grinberg, S. Kokkioa-Schumacher, E. A. Leon, J. Moreno, R. Neely, R. Pankajakshan, O. Pearce, D. Richards

September 6, 2019

IBM Journal of Research and Development

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

Sierra Center of Excellence: Lessons Learned

Aaron Black, Adam Bertsch, Johann Dahm, Leopold Grinberg, Ian Karlin, Sara Kokkila-Schumacher, Edgar A. León, Rob Neely, Ramesh Pankajakshan, Olga Pearce, David Richards

The introduction of heterogeneous computing via GPUs from the Sierra architecture represented a significant shift in direction for computational science at LLNL, and therefore required significant preparation. Over the last 5 years, the LLNL Center of Excellence has brought employees with specific expertise from IBM and NVIDIA together with LLNL in a concentrated effort to prepare the applications, system software, and tools for the Sierra supercomputer. This article shares the process we applied for the CoE and documents lessons learned during the collaboration, with the hope that others will be able to learn both from our success and intermediate setbacks. We describe what we have found to work for the management of such a collaboration and best practices for algorithms and source code, system configuration and software stack, tools, and application performance.

1. Introduction

Practically from its founding in 1952, Lawrence Livermore National Laboratory (LLNL) has been a pioneer in the use and development of high performance computers. One of the lab's very first large procurements was a Remington-Rand Univac I computer, delivered in 1953. Since that first Univac, LLNL has remained at the forefront of computing not only by operating systems based on the most advanced technology available, but also by adapting our applications in response to periods of rapid change in computer architecture. The arrival of a Cray-1 in 1978 marked the transition to the vector processor era and Livermore codes adopted algorithms to take advantage of vector parallelism. The 90's brought the microprocessor era and a series of massively parallel machines including ASCI Blue Pacific (1998), BlueGene/L (2005), and Sequoia (2012), each of which was the fastest machine in the world at the time. Codes adopted MPI and domain decomposition to exploit those distributed memory architectures. By 2013 it was apparent that another architectural shift was in progress, this time to heterogeneous compute nodes populated with multi-core CPUs and GPU accelerators. Once again, it was time for application codes at LLNL to adapt.

In mid-2014, LLNL entered into a contract with IBM to deploy Sierra, a flagship supercomputer housed at LLNL in support of the National Security mission of the NNSA in partnership with Los Alamos and Sandia National Laboratories. Sierra was ultimately built and accepted for delivery in 2018 on time and on budget, and transitioned to a classified network for programmatic use in early 2019. The 4–5 year lead time offered ample time for co-design of the system hardware and software, as well as an opportunity to prepare our applications for the major shift to GPU-based heterogeneous computing. In support of this shift, a Center of Excellence (CoE) [9, 13] was established early in the contract period as a partnership between IBM, NVIDIA, and the NNSA laboratories to begin the long transition of our application code base and to help ensure Sierra could be put to productive use as soon as possible upon delivery.

The process of transitioning applications and application developers from a CPU-only system to a heterogeneous system is not over, but our experiences can be documented and shared with a wider audience. In this paper, we reflect on the lessons learned in the CoE that can hopefully act as a guide for other institutions. An earlier publication [14] outlined the approach used to organize the Sierra CoE. This paper builds upon that work and focuses more on the technical lessons learned. Shifting to heterogeneous computing is a major undertaking, but we feel strongly that most, if not all, HPC centers are likely to adopt heterogeneous architectures as they offer compelling advantages in performance potential over traditional CPU-only based designs.

We have organized our lessons learned according to the audience to which they apply. Section 3 contains advice that applies most directly to developers who are responsible for porting and optimizing code. Section 4 is directed at developers and users who are responsible for constructing science workflows. Section 5 speaks to users and developers of performance tools, Section 6 is for the team that administers the system, and Section 7 is for the mangers of a CoE. Finally, Section 8 contains advice that doesn't fit neatly into other sections.

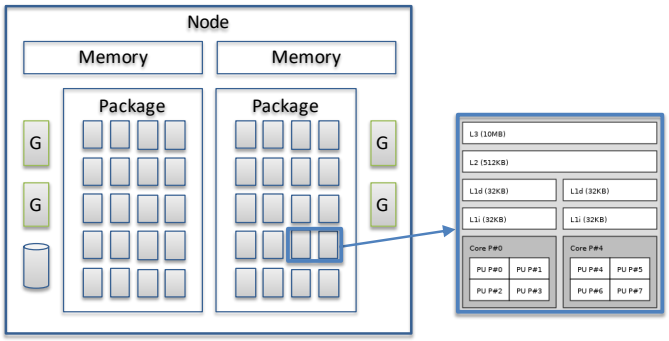

Figure 1 A Sierra compute node with two CPUs and four GPUs. Each CPU has 20 latency-optimized user cores and high-capacity memory. Each GPU has throughput-optimized cores and high-bandwidth memory. As shown on the right-hand side the CPU architecture is SMT-4, which results in 160 user hardware threads.

2. Challenges Moving Production Codes into the Heterogeneous Computing Era

Moving LLNL mission critical applications to Sierra was challenging not merely due architectural change, but also because of the constraints imposed by the day-to-day production demands placed on these applications. Production work needed to continue while the codes were ported. Hence, our codes were required to remain performant and accurate on a wide range of systems including laptops, workstations, and commodity systems even while Sierra support came online.

Many Advanced Simulation and Computing (ASC) applications contain hundreds of thousands to millions of lines of code spread across multiple packages, often written in different programming languages. These codes are long-term projects with 15+ years of development by large teams. As a rule they have outlived the systems for which they were originally designed and most are expected to outlive Sierra. Portability in a single code base is critical to avoid the maintenance costs of supporting machine specific code and is necessary to maintain a connection to concurrent verification and validation efforts. Achieving performance on a wide range of architectures while maintaining portability is a formidable challenge. Therefore, performance portability and methods to achieve it became a large focus of COE effort.

In shifting to heterogeneous hardware code teams needed to deal with new challenges of data placement and movement, along with where to perform compute. Sierra's GPU accelerated nodes contained two types of processors and two memory spaces (see Figure 1) in contrast to homogeneous nodes, which have one of each. GPUs are high throughput devices that require large amounts of parallelism to sustain peak performance. In contrast, CPUs are latency oriented and do not need as much parallelism to run at peak. GPU and CPU memory on Sierra have different properties. GPU memory is limited in capacity with high bandwidth, while CPU memory had large capacity and lower bandwidth. The shift to two processor types and two memory spaces with different trade-offs drove programming decisions and challenges.

Since GPUs have less memory capacity than CPUs, developers were forced to choose between constraining problem sizes to fit in the limited space or transfer data between the CPU and GPU memory to run larger problems. Data transfer could be handled manually, left to a runtime, or hybrid approaches could be used. Manual control often resulted in better performance, but at a higher programming cost. Having two processor types each of which, specialize in different computations presented another decision for teams to make. Parallelism sometimes dictated which processor to use, with highly parallel code running on the GPU and less parallel or serial code on the CPU. However, since data motion is expensive, data location would sometimes dictate where to compute. For example, developers might choose to run a kernel with little parallelism on the GPU because the required data was already in GPU memory.

Heterogeneous nodes also introduce new choices and challenges for sending MPI messages. For data already in CPU memory, sending messages from the CPU was best. For data in GPU memory, there are multiple options: copy data manually to the CPU then send a message, use GPU aware MPI to stage data through the CPU, or use GPU aware MPI to send RDMA messages. Each of these present various trade-offs, in code complexity, performance and memory usage. Overall, the switch to heterogeneous computing greatly complicated the search space for best performance and many code teams opted to start with functionality before focusing on performance.

3. Application Modernization

Designing for a heterogeneous CPU/GPU system such as Sierra is difficult: applications need to be written with awareness of inter-node communication and data movement between CPU and GPU devices, all while coordinating program execution across multiple CPU cores and GPU devices. This is a substantial step

beyond what had to be considered on previous systems. In order to address these additional requirements, application code teams found that refactoring and optimization typically progressed through a series of stages:

- Refactor and simplify the code, removing anti-patterns
- Create a mini-app to test algorithms and to engage with others
- Use portable programming models and abstraction frameworks to ensure code works on different platforms
- Focus porting effort with a target problem or use case that exercises the salient features of the application
- Search for additional parallelism in the algorithms
- Manually manage memory for better performance
- Iteratively refactor and apply the steps above until desired performance is reached

3.1. Refactor and simplify code

GPUs are highly threaded devices and therefore require thread-safe code. Some of the code teams at LLNL found that their first step was to eliminate obviously unsafe coding practices, such as global variables and static objects. Codes that use static objects in the device code will not even build properly. GPUs should also avoid anti-patterns such as deeply nested branching, and excessive virtual function calls and pointer chasing. Refactoring and simplifying the code to eliminate these common problems may also help the compiler generate more efficient code for the CPU, and has generated a speed-up before offloading.

Many standard C++ containers and algorithms are also hostile to GPUs. Operations like vector's `push_back()` method are not even well-defined in a parallel execution environment. Other common constructs such as string operations, exceptions, or CPU system calls must typically be removed from any code that intends to target GPU execution. Vendor-supported libraries such as Thrust [3] provide a subset of functionality usually obtained from standard libraries.

It may seem tempting to use custom allocators with standard C++ containers to place data in GPU memory, but their methods, including data initialization, will still execute on the host causing either immediate crashes or frequent data migration—greatly limiting the usefulness of such an approach. Instead, application developers at LLNL have found it useful to eliminate standard C++ containers from their device code and instead implement their own classes with suitable interfaces and well defined parallel semantics.

Developers also need to strike a balance between advanced language features and code simplicity. Compiler writers strive to fully support advanced language standards, but in reality applications that rely on advanced features will need to spend more time working through issues with the vendor/compiler teams. It is often a better idea to keep production device code as simple as possible and use well established language standards.

3.2. The mighty mini-app

Many code teams found it useful, especially when engaging with vendors, to write mini-apps as a playground for testing porting strategies. The small size of a mini-app greatly facilitates rapid prototyping and avoids restrictions and complication of passing the full application between collaborators. Effective mini-apps should include the salient compute regions and general application flow without including the external dependencies or other features that tend to enlarge the code base. When working with mini-apps it is important to remember that the full application may not admit optimizations that seem obvious in the reduced code base. For example, assuming certain variables are constants, or altering a specific loop structure, may improve mini-app performance, but the corresponding transformations may be impossible in the larger application.

Predictably, codes without a mini-app struggled to get support early on, while codes that had openly available mini-apps received fast-track treatment. Mini-apps also played a role in compiler development: advanced code features were exercised more easily in the mini-apps, and frequently facilitated the process of sharing reproducers for issues.

Proxy apps that represent only a specific aspect of a full application have also proven to be extremely useful for trying out different approaches on the heterogeneous architecture to solve problems that did not exist on CPU-only machines. Comb, a proxy app for halo exchange, is one example of such proxy (https://github.com/LLNL/Comb). While the application domains the GPU computes on are large enough to utilize the computational power of the GPU, the application frequently and repeatedly invokes the halo exchange routine, where the boundary information is sent to the logical neighbors. The data to be exchanged

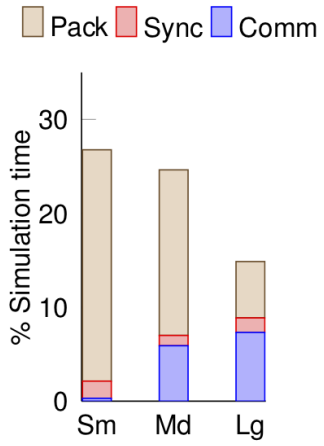

Figure 2 The overhead of a halo exchange in a hydrodymanics application on Sierra, as demonstrated by the Comb proxy application.

is stored in the GPU memory, and needs to be sent to the memory of 26 logically neighboring GPUs, necessitating going off-node on the machine. Prior to being sent, the boundary data needs to be packed into buffers. While the cost of packing was negligible on the CPU-only machines, packing in GPU memory proved to be kernel-launch-overhead bound, as can be seen in Figure 2. Using Comb, we explored packing on the CPU, GPU, and various optimizations such as CUDA Graphs for running the kernels in parallel and thus reducing the kernel launch overhead. We also explored using regular MPI, GPU-aware MPI, GPU-aware GPU direct, and GPU direct async technologies.

3.3. Use portable programming models and abstraction frameworks

Codes with a relatively small number of high intensity kernels can often justify writing platform specific code for each new architecture. Customizing a few hundred or even a few thousand lines of code to a vendor-specific programming model can be the most effective way to extract maximum performance at a reasonable level of effort. However, for the large application codes at LLNL, vendor-specific approaches such as CUDA were almost immediately deemed off-limits due to concerns about portability and future maintainability.

With CUDA off the table, attention focused on the emerging OpenMP offload model as a potential portable solution. Unfortunately, it was soon determined that the number and variety of pragma statements required to write code capable of running in both CPU and GPU environments overwhelmed developers and also resulted in code that wasn't actually all that portable.

For C++ codes, RAJA quickly gained momentum as a parallel programming framework that satisfied practically all of our requirements. RAJA uses templatized parallel for loops along with lambda capture to separate loop content from execution policy. Mutiple RAJA back-ends are available and allow developers to target CUDA, HIP, or OpenMP target offload on GPUs, as well as threaded (OpenMP) or sequential execution on CPUs without any modification to the loop kernel (or lambda). RAJA is also designed for incremental adoption and interoperates with other GPU and CPU programming models. Developers can add RAJA to as many (or as few) loops as they need, depending on whatever constraints may exist in their development process. Hiding the actual parallel model behind and abstraction layer is clearly a best practice for large science codes written in C++.

Because C and Fortran lack the language features needed to implement strong abstraction layers such as RAJA, other solutions are needed. Many C codes are choosing to convert to C++, specifically to gain access to abstraction frameworks. As for Fortran...

The relatively small community of developers, compilers, and vendor resources devoted to Fortran, compared with C and C++, affected the success of early code porting efforts. LLNL teams found that the compilers, runtime, and tools mostly worked after a short period of bugfixes, as long as they wrote Fortran code using only the set of features common to C and C++. More advanced modern Fortran features, *i.e.*, shaped arrays and array notation, took much longer to mature. Although these features are supported by the compilers, they have not been fully supported in GPU debugging tools because the IBM compilers have to translate Fortran to C before it can be assembled with the NVIDIA ptxas assembler.

Naturally, this led to significantly increased development time and overhead for code teams and vendors using Fortran. For example, a Fortran code team at LLNL spent close to two years collaborating with compiler teams, reporting 60+ issues, before being able to compile the full application using OpenMP 4.5 target offloading. This cost was exacerbated by the difficulty in extracting reproducers from the code, as more complex bugs required increasingly more complex reproducers.

The effect of the smaller community was also felt on the support and maturity of the OpenMP standard for

advanced Fortran features not common to C. Examples include the inability to map allocatables to the GPU, in some cases, and the lack of thread team support in OpenMP workshare for mapping over array operations.

The support for CUDA Fortran was more mature at the beginning of the CoE effort, and successes were made porting code to the GPU using CUDA Fortran by members of the CoE, but these approaches were unfortunately deemed insufficiently portable for the large production codes at LLNL. In addition, the same difficulties were found with tools trying to debug or troubleshoot GPU kernels as compared to the C equivalents.

The Fortran language standard committee has been discussing adding the equivalent of C++ templates and other functionality which may allow developers to implement an abstraction layer to select different kernel implementations. Fortran already has native language support for expressing some parallelism its “do concurrent” loops, “co-arrays”, and array operations syntax. Future work may yield a solution that allows developers to select different kernel backends similar to what the C++ RAJA layer can do. As of yet, however, these are only early discussions and proposals, so the pragma based solutions, OpenMP and OpenACC, are currently the only options for portable code in Fortran.

3.4. Start with a target problem

It is easy to be overwhelmed by the problem of porting a large and complex code to GPUs. There are several ways to approach the task, but our experience suggests that starting with a specific target problem or use case and porting only those parts of the code that are necessary to run that problem is an effective and efficient strategy. Working with a specific test problem will quickly isolate the parts of the code that need to be moved to the GPU from those features that can wait until later. Concentrating on a subset of the code will allow your team to gain experience with the interactions between the GPUs and your coding patterns and data structures. This experience will improve productivity later on when you take on additional features in sections of the code that were initially deferred.

One of our teams decided to start their port by classifying every loop in their code and converting loops to a parallel framework. This had the advantage that all of the apparently similar work was all performed at once. However, after the initial conversion, the team focused on a specific target problem for optimization. In the process of optimizing the code for their target problem they improved their understanding of GPU programming and found that some of the initial conversion work needed to be altered to take advantage of that improved understanding. Working toward a specific problem from the beginning would have avoided the need for this re-work.

3.5. Expose parallelism

The coarse-grained thread- or process-level parallelism typical of codes developed for CPUs is usually insufficient to gain a performance advantage from GPUs; at least thousands or tens of thousands of threads are necessary. Additionally, whereas CPU threads are largely independent and can communicate with each other at any point, GPUs focus on the single instruction multiple thread (SIMT) model, which splits execution into blocks of lightweight threads that execute on streaming multiprocessors composed of GPU cores.

Application developers often need to revisit algorithms, carefully analyzing program dependencies to determine if more parallelism can be obtained by reordering or collapsing loops, reorganizing data structures, or even employing different numerical methods. For kernels with limited compute work, the overhead of kernel launch can dominate performance. Kernel fusion can be an effective technique to lower such overhead, as long as register pressure or local storage requirements do not limit occupancy or force register spills. As long as the problem fits on the device, the key to adding parallelism may be as simple as running a larger problem size.

Algorithms themselves may need to change in order to adapt well to GPUs. For example, high-order finite element codes are switching from matrix-based approaches with less compute for the amount of data loaded from memory to matrix-free algorithms with higher arithmetic intensity. By cleverly writing the GPU kernels, developers can hide the additional computation behind the loads and stores.

Other algorithms are less clearly suited to GPU execution. Algebraic multigrid algorithms are classically difficult to execute solely on a GPU, due to the fact that the setup phase has far less parallelism and much more indirection than the repeated matrix-vector multiplications of the solve phase.

The process of reorganizing code and refactoring algorithms is almost certain to introduce bugs.

Unfortunately, GPU debugging is notoriously difficult because of an almost complete lack of consistency guarantees within kernels. Utilizing an abstraction framework like RAJA can offer significant benefit in this regard. New algorithms can deployed first with a sequential execution policy on the CPU to ensure functional correctness. Next, a CPU-based thread parallel policy can be used to detect many common parallel code bugs such as data access dependencies or race conditions. Code profiling and sanitizer tools developed for CPUs can be extremely helpful in this process, even when they are executed on other platforms.

3.6. Memory management

Applications, especially the large codes at LLNL, have found the page-based CUDA managed memory and address translation service (ATS) available on the Sierra system to be helpful in the process of incrementally porting code while maintaining correctness. However, they have also found that relying solely on this mechanism for host-device transfers fails to meet application performance goals. Most teams have found that they need to insert code to explicitly copy code between CPU and GPU memory. In some cases this takes the form of making (asynchronous) prefetch requests to the CUDA runtime. Other codes use smart pointer abstracts (CHAI) to free developers from manually inserting requests to move memory. Adding explicit memory copies also has the advantage that the application remains portable to systems that do not yet support unified memory.

Even after optimizing data traffic, some applications encountered performance issues related to the CUDA API calls used to allocate managed memory. Allocating memory through the CUDA API can be 100x (or more) slower than obtaining heap allocations on the system's host memory. As a first step, application teams have found it important to classify memory allocations, so that calls for persistent (host or device) and temporary (device) memory are made through different function calls. To further address this challenges, LLNL is developing an open source library, appropriately named Umpire [2]. This library provides a vendor and system indepent mechanism to request allocators in different memory spaces. It also provides a variety of allocation policies such as pooled memory allocators. With Umpire's pooled allocators, the expensive CUDA API calls can be made only once for a large pool and their cost amortized over many sub-allocations. Umpire also supports a robust introspection functionality, allowing the calling code to query what allocators created pointers, how much memory is left in a specific resource, *etc.* The application can then make execution decisions based on that information.

Large codes can also use Umpire to manage competition for limited GPU memory between multiple packages. For example, a single temporary allocation pool can be shared among multiple packages, each using the corresponding GPU memory while they are active, but freeing the space for other uses before surrendering control to the next package in the execution flow.

Porting Step	Speedup
1. Compile code and run (x86+K80)	0.01
2. Minimize data motion	0.03
3. Minimize dynamic memory allocations	0.44
4. Add asynchronous kernel execution	1.1
5. Optimize individual kernels	1.9
6. Use early access hardware (P8+P100)	5.3
7. Use Sierra hardware	6.9

Table 1 The process of porting an LLNL production code for a specific problem of interest. A speed-up, reported relative to a Haskell node, was not immediate after porting the application. Rather, over a series of steps the performance was improved 500x to obtain a 7x speedup over the CPUs.

3.7. Lather, rinse, repeat

The steps described above are general guidelines—in reality, application porting is a difficult and iterative process. Do not be discouraged when the code is orders of magnitude slower on GPUs the first time it runs. Good performance almost always requires at least a few iterations of code restructuring.

Table 1 shows the progress of porting a large LLNL production code, initially designed and tuned for CPUs, to Sierra. This process required a sustained effort over a long period of time to obtain the desired speed-up.

4. Workflow Strategy

As scientific applications add machine learning and other real-time analytics or visualization into their simulations, workflows are becoming increasingly complex. Heterogeneous architectures and multiple memory resources adds yet another level of complexity to the mix. To achieve optimal performance on systems such as Sierra, application developers need to consider advanced workflow strategies. Key lessons learned from large-scale workflows include:

- Optimize resource allocations at the workflow level

- Use workflow management tools
- Consider the memory hierarchy and data sharing tools when designing a workflow
- Package managers and continuous integration can help ensure the reproducibility of a workflow

4.1. Resource management

Complex workflows may require multiple applications to interact. This can cause different applications to compete for the same resources. In order to avoid over-subscription, job schedulers play a key role in resource management. Additionally, developers must consider which resources, such as CPUs or GPUs, are the right choice for each application and balance the different resource needs for each component of a workflow.

Hierarchical schedulers, such as Flux, or schedulers that allow the creation of child jobs, such as LSF® with jsrun, can overcome the challenges of running complex workflows at scale [1]. In some workflows, co-allocating applications that are data generators with an application that consumes the data can be a critical performance consideration. Other co-allocation challenges include balancing GPU-heavy applications with CPU only codes or file I/O intensive applications with applications that have a low file I/O use. In each of these instances, users need flexibility in the scheduler to manage which types of jobs or applications are co-allocated on the same compute node [7].

Part of the developer's responsibility when developing workflows includes deciding which components of the workflow will benefit from the different available resources. This means that developers must consider how the individual components fit into the full workflow. As an example, porting an application that is part of a workflow to the GPU, does not necessarily improve the performance of the workflow since different applications may have to compete for resources. One of the lessons learned from the CoE was that in coordinated workflows with multiple application interactions, developers should use GPUs for the most compute intensive components, co-allocate low memory codes on nodes with memory intensive codes, and co-allocate file system intensive codes with codes that have minimal use of the file system.

4.2. Workflow management

Workflows with tightly coupled interactions or with thousands of simultaneously running jobs can benefit from dedicated workflow management tools. A key responsibility of workflow managers is to track the available computational resources and tasks that need to be scheduled. This may include using machine learning or other analysis approaches to create priority queues of tasks. Workflow management tools may play an active role in the simulation, in which case it is beneficial to use a workflow management tool that is closely coupled to multiple schedulers [6, 5]. Workflow management tools that are coupled to multiple schedulers also improves the portability of the workflow. Additional responsibilities should include checkpointing or restarting entire workflows—a job which is beyond the scope of the individual tasks' checkpointing and restarting. In ensemble workflows, restarting the whole workflow requires ensuring that each individual component can be restarted and requires checkpoint capabilities at the workflow level to ensure that the entire workflow is capable of restarting from a valid ensemble. One of the lessons learned from the CoE is that workflow checkpointing and restarting needs to be designed into the workflow from the start. Since many things can go wrong during a running simulation, this approach can help reduce wasted compute cycles and will make the workflow more robust against system failures.

4.3. Data management

Sierra offers several different resources for storing data. In order of increasing capacity (and decreasing performance), each node of Sierra provides High Bandwidth Memory (HBM2), DRAM (DDR4), and a PCIe SSD that can be used as a burst buffer or as extended memory. Sierra also has a parallel file system (GPFS) with 154 PB of usable storage. With so many different options for storing data, managing the memory hierarchy can be an important optimization for both single applications or large workflows. As an example from one of the machine-learning enabled workflows at LLNL, machine-learning or analysis codes may be able to take advantage of burst-buffer technology to reduce the impact on GPFS. Optimizing memory use can also impact the required job layouts on a compute node, which means that schedulers and workflow management tools need to allow this type of optimization.

In workflows where multiple applications produce or consume data, there is a need to manage how the data is shared. Traditionally, workflows have used file I/O to share data between the different applications. As storage hierarchies become more complex, and

compute performance further outpaces filesystem performance developers should consider tools such as the IBM Data Broker or similar database software to facilitate data communication between different applications in a workflow [15].

4.4. Reproduciblity approaches

CoE teams frequently involved many different developers and collaborations between different teams. Since one of the goals of the CoE was to help application teams be ready for Sierra on day one, the developers were often working on systems that were undergoing frequent changes (to both hardware and software). These factors influenced several teams to explore different approaches to ensuring reproducibility between developers and across different compute systems. Updating applications to use continuous integration helped to maintain code and was useful for the early detection of issues related to hardware or system level software changes. Package managers, such as Nix or Spack, also proved to be useful approaches for ensuring reproducibility across different developers, platforms, and compilers [8, 4].

5. Tools

For performance tools, our two most significant challenges were profiling GPU code and handling HPC programming models, such as OpenMP and RAJA, that are needed for GPUs. We tackled these challenges with a two-pronged approach, using different tools to tackle each problem. Vendor tools, such as nvprof from NVIDIA, can provide deep technical insights into GPU performance, but don't focus on HPC issues like scalability or programming models. The open-source HPC tools community produces tools that handle scalability and programming-model challenges, but do not have deep insights into the hardware. To push forward the open-source tools we worked with the HPCToolkit team from Rice University to "carve a path through the jungle" and work through the issues that performance tools encounter during system design. This worked well and uncovered numerous issues with DWARF debugging information, performance, and interface design. While HPCToolkit can profile GPU code through the CUPTI interface, providing the same level of GPU-insights as nvprof remains a challenge, and we are continuing to work with NVIDIA to understand both the interfaces and methodologies that are needed for GPU profiling. Efforts are also ongoing to make nvprof more usable in HPC environments. Making nvprof a scalable multi-node tool was not viable, but it is still useful as a single-node tool for focusing on the GPU usage. While issues like proper OpenMP support are still on NVIDIA's roadmap, we made progress with allowing applications to use annotations to name regions of GPU code. The CoE also helped provide feedback to developers improving the NVIDIA profiler, HPC Toolkit, and other system tools to support the address translation service, and others.

The key lessons learned from our experiences with performance tools are mostly a reflection of limitations we encountered (and mostly worked around) during the CoE. These include:

- Performance tools need to capture and display all aspects (CPU, GPU, MPI, and data traffic) of code execution at various scales
- All tools should provide public APIs for accessing the tool functionality and/or data collected
- Offloaded kernels should be easily identifiable and attributable in profiles
- The OpenMP run time should emit errors and warnings when directives are ignored or not executed for any reason

5.1. MPI aware tools

The NVIDIA Visual Profiler (nvvp) is a very powerful interactive profiler that provides both traces and profiles of applications executing on the GPU. It is possible to use nvvp with MPI codes if they can be launched without a MPI launcher (jsrun, mpirun, or mpiexec) but this is increasingly not the case with current MPI implementations and our ability to use nvvp in interactive mode with LLNL applications was severely limited. Developers were able to make progress using a command-line based workaround that involves running multiple experiments and using nvvp to visualize the results. However, this is a laborious process involving complex command-line invocations and is several times slower than using nvvp.

The challenge of collecting and displaying performance data at large scale will not be solved easily. Data sets are large and there is no single obvious way to present the data to developers so that it can be easily understood and lead to actionable insights. The addition of heterogeneous hardware only compounds the problem. Continued co-design with vendors and open-source developers will be needed to deliver tools that are better suited to developer needs.

5.2. Comprehensive profiles and traces

On a heterogeneous node with multiple execution spaces, profiles and traces must account for all the execution time of the application. Using a separate tool for the GPU, CPU and MPI communications leaves significant swaths of time unaccounted for. The problem is exacerbated when the GPU kernels are asynchronous. For SW4 on large problems, this unaccounted time can be as high as 16% of solution time. Such a comprehensive tool needs to be usable at scale since many performance bottlenecks are not easily reproducible at smaller scales.

5.3. APIs for performance tool data

Trace files generated by nvprof contain a wealth of information. Unfortunately, the standard GUI tools to access the traces don't always meet developer needs. Trace data often needs to be processed and correlated to shortlist sections of the code for tuning efforts. For example, we found that sorting kernels based on some combination of GPU pagefaults and runtime enabled developers to locate code sections that would benefit most from prefetching data from CPU to GPU memory. A published API for accessing data from profiling and tracing tools would enable the development of application specific data mining codes and allow the use of metrics that are currently inaccessible.

5.4. Human readable kernel names

The use of lambda expressions in RAJA codes results in very long function names, (approximately 153 characters) with the name of the calling function embedded somewhere in the middle. In routines with multiple lambdas it can be impossible to distinguish between them. OpenMP 4.5 compilers, which have a similar issue with offloaded sections, handle the problem by naming the offload section with a combination of the calling function and line number. A similar approach is needed for uniquely naming lambda expressions used for offloading GPU code. The current workaround is to access and modify the sqlite database generated by nvprof, demangle the function names and do a regexp based search and replace. This approach does not resolve the issue of correlating the lambda expression to line numbers.

5.5. OpenMP 4.5 silent errors

Updates of mapped regions in OpenMP 4.5 can fail silently if there is an error in the original mapping. No warnings or errors are reported by the failed update or mapping. These kinds of bugs are extremely difficult to find since every update has to be laboriously checked by manually computing and comparing the checksum of the data on both the host and device. An error message from the runtime whenever an OpenMP directive is not carried out would go a long way in avoiding these problems.

6. System Deployment Strategy

Administering heterogenous machines has presented some new challenges. While the departure from standard HPC practices is not as substantial for systems management as it is for application programming, a number of techniques have been developed to increase both stability and usability of these systems. The most important lessons learned include:

- Run tests frequently to ensure reliability.
- Consider memory bandwidth when sizing the filesystem.
- Resource management and job scheduling must be closely coupled.
- Resource affinity and binding on heterogeneous nodes can confuse even expert users.

6.1. Test early, test often

Failures on past systems have been dominated by moving parts and power supplies. DRAM and CPUs do have failures though, and they are the next most common failure items. CPU failures are particularly difficult to reliably diagnose. Adding an additional set of both memory components (HBM) and processors (accelerators) increases the problem space for maintaining machine availability. Significant diagnostics have typically occurred at boot time, but we were forced to greatly expand the set of diagnostics that run between jobs in order to maintain the ability to consistently allocate reliable nodes to user jobs. Tests were implemented on Sierra for CPU and GPU performance, correctable and uncorrectable memory errors, parallel filesystem availability, and CUDA library calls.

6.2. New considerations for filesystem size

Sizing of the parallel filesystem on previous systems was accomplished using a metric based on DRAM capacity of the system. The assumptions leading to this metric were that I/O was dominated by defensive, write-heavy I/O and that the size of these defensive checkpoints could be estimated as some fraction of DRAM. Increased memory bandwidth on heterogeneous systems leads to much more frequent saves of scientific data. We also find that novel computing approaches are increasing the prevalence of read-heavy I/O workloads. We are fortunate with the

Sierra system to have a very performant filesystem as well as an experimental burst buffer. Capacity may end up being a challenge over time, unlike on previous advanced technology systems where capacity was always sufficient as a result of procuring enough disk spindles to meet bandwidth needs. Costs of long term storage (tape) are also impacted by the increase in the rate that scientific data is generated. Future advanced technology system procurements will need to size parallel filesystem bandwidth and capacity relative to system memory bandwidth rather than system memory capacity.

6.3. Resource management and job scheduling

Sierra has a separate resource manager and scheduler, a design that was chosen to improve performance and to provide visibility of the heterogeneous nature of the nodes at the resource manager level. Separating these components could also allow a single scheduler to schedule jobs across a collection of clusters while each cluster has its own resource manager. Unfortunately, synchronization between the resource manager and scheduler with respect to availability and state of resources has been a persistent problem with this design on Sierra. Race conditions have proven difficult to eliminate, and error paths sometimes leave persistent lack of agreement between these two processes. We have added monitoring to detect a lack of synchronization which typically requires administrator intervention to correct. Future systems should maximize integration between the resource management and scheduling of the system, ideally utilizing a single piece of software to provide both functions.

Handling resource allocation and affinity at the same time has proven more challenging than we expected with the initial design. It seemed a very natural thing to request resources and the associated binding all as part of the job launch command, but implementation of a syntax that was both intuitive for common cases, but powerful enough to handle edge cases, proved very difficult. This would be particularly challenging if scheduling individual resources rather than whole nodes (ability to divide nodes between users). The most practical solution for Sierra was to schedule jobs in increments of a whole node. We then implemented a series of wrappers to make common launch cases easy.

6.4. Tools for affinity and binding

To obtain the high levels of performance that are needed to deliver breakthroughs in science and engineering, HPC applications must be efficiently mapped to the computing hardware. Unfortunately, when left alone, application users are prone to suboptimal mappings that adversely affect performance. Examples include running multiple OpenMP threads on a single core while other cores on the same socket are idle, OpenMP threads accessing memory in the wrong NUMA domain, or multiple GPU kernels sharing the same GPU while other GPUs are idle. Obtaining optimal mappings is especially challenging on complex heterogeneous architectures like Sierra.

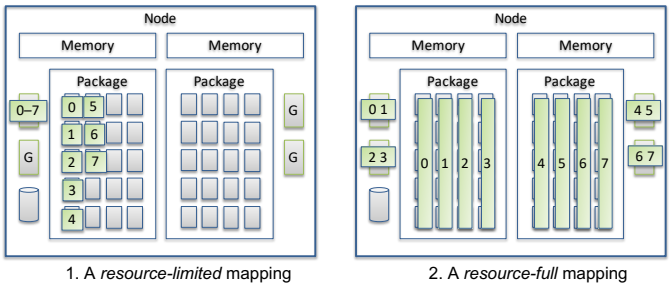

Figure 3 Two application mappings on a Sierra node. The application consists of eight MPI tasks with OpenMP threads and GPU kernels. The green numbered boxes represent MPI tasks or GPU kernels launched by such tasks. On the left mapping, each MPI task is mapped to a single core and all GPU kernels are executed on the first GPU. On the right mapping, each task runs on five cores and executes kernels on a local GPU.

Job affinity and binding on Sierra is provided by the IBM Cluster System Management (CSM) `jsrun` command, which provides a powerful interface to map an application to the hardware. One of the strengths of `jsrun` is the ability to express complex mappings through its parameter-rich interface. At the same time, it is complex and has a high learning curve.

Consider an MPI+OpenMP job with GPU kernels and eight MPI tasks per node. A reasonable mapping distributes the tasks across the CPU cores without crossing NUMA boundaries and launches the GPU kernels on local GPUs (the right panel in Figure 3). To do this with `jsrun` one has to specify:

```
$ jsrun -a 2 -c 10 -g 1 -r 4 \
  -d packed -bpacked:5
```

This means create four *resource sets*, each with two MPI tasks, ten cores, and one GPU; assign tasks into the first resource before using the next resource set (packed); and bind each task to five cores within a resource set. This is not an easy task! A user needs to know the number of cores and GPUs on a compute

node, distribute them accordingly to MPI tasks, bind each task to the corresponding CPUs, and distribute the ranks across the resource sets. Not following the recipe above can have severe consequences on the performance and scalability of an application:

If the user specifies only the number of MPI tasks on the launch command, the tasks are mapped to the first eight cores of the first socket with four OpenMP threads per core. Furthermore, without specifying a particular GPU in the application code, all kernels are run on the first GPU (the left panel in Figure 3). If the code is designed to use as many resources as possible, it will be limited to eight cores and one GPU – as opposed to 40 cores and four GPUs. This example shows how easy it is to limit application performance with even realizing there is a problem.

To address the complexity and the lack of reasonable default settings, LLNL developed a number of tools to complement the CSM launching system. These include `lrun` to launch jobs and `mpibind` to provide affinity [11, 12]. The key concepts of these abstractions distilled from our experience with affinity on heterogeneous systems include:

- Provide a simple and intuitive interface.
- Use the memory hierarchy, as opposed to the compute resources, to guide application mappings.
- Leverage locality of resources.
- Encompass hybrid programming abstractions.
- Assign at most one NUMA domain per MPI task.

Using the LLNL interface, the hypothetical eight rank-per-node MPI+OpenMP job described above is launched with the command:

```
$ lrun -T8
```

(right panel in Figure 3). While the interface is simple and the resulting mapping efficient, this mapping policy does not cover all cases. As such, `jsrun`'s full-feature launcher continues to play an important role in Sierra's runtime system.

7. Management Considerations

By its very nature, a Center of Excellence is designed to bring multiple teams (and in our case multiple organizations) under a common umbrella for the purposes of maximizing training opportunities, sharing of best practices, and pursuing a common goal presenting extensive challenges. The idea that the "whole is greater than the sum of the parts" is a core concept, and as such demands a managed structure that is designed to work across organization and discipline boundaries.

A core set of lessons we would recommend to others include:

- Start with a high-level strategy (but be prepared to deviate)
- Build agile and adaptable plans
- Schedule activities to ensure 2-way engagement
- Invest in collaboration tools

7.1. Establish a High-level Strategy Early

The Sierra CoE was established as an LLNL subcontract to our vendor partners at IBM and NVIDIA, and managed by a CoE steering committee consisting of representatives from each of those organizations. Work was defined through the development of bi-annual *work plans* that outlined the goals for the following six month period. Reports were generated at the end of each period as milestones and used as contract deliverables, but more importantly as a record of work performed that could be shared with other stakeholders and across application teams. We found that six months was an appropriate cadence as it allowed a significant body of work to be performed while forcing regular reflection of priorities. The work plans intentionally included a roadmap for more work than was possible to achieve with given resources—thus building in agility and allowing flexibility in the management team to pursue the most fruitful engagements that arose. Unlike standard contractor relationships, a CoE should be treated as a partnership with mutual co-design benefit; application teams get access to vendor expertise and access to pre-production software, and vendors get insight into the challenges of large-scale application development that will help harden their own offerings prior to public release.

The earliest phases of the Sierra CoE were largely focused on training, workshops, and occasional OpenMP hack-a-thons designed to familiarize CoE staff with the programming models, tools, and basics of GPU computing, and in the case of hack-a-thons, help influence future standards based on early experiences[10]. As discussed earlier, mini-apps proved to be an invaluable tool for coordination, particularly in our environment where applications were not freely distributable due to export control and classification restrictions. As the CoE entered its second and third years, increased attention was paid to algorithmic

restructuring, hack-a-thons, and continued development of the underlying performance-portable abstractions in RAJA, Umpire, CHAI, and Kokkos. Finally, as early access hardware and software came available, the organization of the CoE pivoted to taking the lessons learned in prior years and achieving our ultimate goal of transitioning our large application base.

Applications will always be in various stages of readiness for such a large undertaking. However, we found that most teams followed a similar progression:

1) Train the team in basics of GPU programming
2) Test ideas in a mini-app (if available), often starting with CUDA
3) Transition to use a performance-portable abstraction such as RAJA or OpenMP
4) Ensure your full application kernels are thread-safe, often using CPU-based OpenMP constructs
5) Take lessons learned in prior steps into the full application
6) Profile, develop, repeat

Early in the CoE we mapped out a high level engagement plan that took into consideration the application teams we wanted to prioritize (those we would anticipate using more of the system cycles, or were more "ready"), as well as external events such as planned software releases from our vendor partners, early access systems, and other milestones. This proved a valuable tool in giving the entire suite of application leads an idea of where they fit into a broader schedule, but didn't lock us in to an overly restrictive planning schedule that would've subsequently proven fatally flawed.

7.2. The importance of agility

Each team will necessarily take the approach that works best for them on a timescale that makes sense for them. A reliable solution for the node-level programming model should be established early in the process and the CoE used to maximize leverage across those teams. Having vendor engagement proved invaluable for the Sierra CoE, largely because much of the supporting toolchain was still nascent, required workarounds, and—most importantly—a solid relationship to rapidly address deficiencies in pre-production software. As heterogeneous computing becomes the norm, we anticipate many of these issues will be relegated to something a more traditional vendor-customer relationship can deal with.

Teams will not all operate in lockstep. Application teams with only a few small kernels or a representative mini-app will be able to run ahead, while those dealing with foundational code cleanup and/or thread-safety issues will straggle along behind. Even the best prepared code teams can see their progress derailed by problems in the compiler, debugger, profiler, etc. As such, agility in the organization of your CoE is key to maintaining forward progress even when some aspects of the work don't go as planned.

7.3. 2-way engagement

Aligning work from multiple organizations (*e.g.*, the vendor and lab teams) such that there is sufficient engagement on each side requires planning and, once again, agility. Some of our most inefficient teaming occurred when vendor expertise was assigned to an application team at a time when that team had other priorities and could not provide sufficient mentoring or guidance to vendor staff coming in brand new to the project. Likewise, teams seeking immediate help without prior planning are likely to find vendor personnel are not immediately available resulting in missed opportunities. These issues are not unique to a CoE, but are part of any good project managers role. Recognize that by working with a CoE, you have increased the size of your team and should give them the same consideration in learning the culture and technologies of the team, and the application and algorithmic characteristics that you would offer a new staff hire.

7.4. Collaboration tools

Finally, collaboration tools such as common wikis, repositories, and bug trackers are essential and important tools that should be established early in the process. This can be particularly challenging when proprietary vendor data must be properly protected. Likewise your own applications may not be open source and need protections. Mutual trust in protecting data is presumed based on the CoE partnership, but tools must be able to enforce that trust and pass muster with security organizations.

When personnel are remote, as is often the case, screen-sharing tools typically used for presentations can be utilized as an effective way to collaborate on code development—similar to the team programming principals of agile programming. With a common view of source code editors, debuggers, and profiling tools the communication bandwidth between developer partners is much higher than trading emails, chats, or

phone calls.

8. Other Advice

To wrap up our lessons learned we present a few final suggestions that didn't fit in elsewhere but are too important to omit.

- Organize hack-a-thons
- Build multi-disciplinary teams
- Commit to reducing technical debt

8.1. Organize hack-a-thons

A *hack-a-thon* is a collaboration event that gathers application developers with compiler and tool developers for a concentrated period of time (usually 2–4 days). Hack-a-thons proved to be an extremely effective tool to collectively work through issues, learn from each other in a software co-design environment, and rapidly address issues in either the application implementation or the underlying compiler and system software without the usual long-delayed implement-test-fix cycle typical in collaborative software development.

The Sierra CoE hack-a-thons were largely focused around the development of OpenMP in the IBM compiler stack. OpenMP v4.x target offload was still an emerging standard during the early phases of the Sierra CoE, and hack-a-thons proved to be a high-bandwidth approach to help application developers understand OpenMP usage while simultaneously providing opportunities for the developers of compilers and the OpenMP runtime to react to deficiencies in the standard. In fact, early hack-a-thons in the Sierra CoE had a large impact on the OpenMP 4.0 standard and informed a series of changes to the standard, many of which were realized in the OpenMP 4.5 and 5.0 standards.

8.2. Build multi-disciplinary teams

The process of adapting a large application to a heterogeneous architecture requires a wide variety of skills and experience that is best addressed by a multi-disciplinary team. The subject matter knowledge and computer science expertise required to refactor and optimize codes practically requires teams that include both domain scientists and computer scientists as integral members. Many teams have also adopted more rigorous version tracking and continuous integration testing to ensure code correctness and portability. Effectively implementing those practices requires yet another set of skills and experience. To maximize the effectiveness of teams with mixed skills, we strongly recommend co-locating the team members to the greatest degree possible. In any case, the days when a single person, or small team of domain scientists could develop a large and complex code are fading away in the rear-view mirror.

8.3. Commit to reducing technical debt

Every software team constantly juggles the competing demands of effort levels, feature requests, delivery schedule, and code quality. Without some relief from the constraints of effort, features, and schedule, the additional job of adapting to architectural change will inevitably drive teams to sacrifice code quality. Rather than allowing GPU porting activities to ring up even more charges on the technical debt credit card, teams (and their managers) should commit to taking opportunities to improve code quality as part of the refactoring process. Teams that pay attention to quality as they transition to heterogeneous architectures are likely to find that they emerge with less technical debt rather than more.

9. Center of Excellence Successes

The work done and lessons learned in the COE have produced impressive performance gains for important LLNL applications. Performance gains relative to an identical number of two-socket Broadwell nodes for selected problems on various applications are shown in Table 2.

In reporting this data, we acknowledge that speedup is an imperfect metric for reporting performance. Speedup doesn't consider differences in the optimal problem sizes on different architectures, can easily be inflated by unoptimized baselines, and ignores differences in hardware cost. However, to our knowledge, no perfect metric exists and the speedup numbers presented do provide at least some indication that the CoE did accomplish its goal to have LLNL applications ready to take advantage of Sierra as soon as the machine was delivered.

10. Conclusions

Preparing for Sierra has been a long journey, but we are starting to reap the rewards for our efforts and increased performance is opening doors to previously impossible science. Furthermore, with the size and power of Sierra generating enormous amounts of data, application workflows and data management are quickly becoming critical pieces of a scientist's toolkit. There are also many more challenges to come as we

Code	Science Domain	Problem Description	Programming Environment	Speedup
ALE3D	Hydrodynamics	Shaped Charge	C++, RAJA	8x
Ardra	Deterministic Transport	Reactor Safety	C++, RAJA	16x
Ares	Hydrodynamics	RT Mixing	C++, RAJA	13x
Kull/Teton	Radiation Transport	Radiating Sphere	Fortran, OpenMP	7x
SW4	Seismic modeling	Hayward Fault	C++, RAJA	28x

Table 2 Speedup relative to dual socket Broadwell node of selected applications optimized by the COE.

expand the portfolio of GPU-enabled applications, and add features to applications already ported. Fortunately, the Sierra Center of Excellence has established a strong foundation and body of experience to ensure the success of that future work.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. LLNL-JRNL-789080

References

[1] Dong H. Ahn et al. “Flux: Overcoming Scheduling Challenges for Exascale Workflows”. In: *2018 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*. Dallas, TX, USA: IEEE, Nov. 2018, pp. 10–19. ISBN: 9781728101965. DOI: 10.1109/WORKS.2018.00007. URL: https://ieeexplore.ieee.org/document/8638377/ (visited on 04/02/2019).

[2] David Beckingsale et al. “Umpire: Application-Focused Management and Coordination of Complex Hierarchical Memory”. In: *IBM Journal of Research and Development* 64.2 (2020).

[3] Nathan Bell and Jared Hoberock. “Thrust: A productivity-oriented library for CUDA”. In: *GPU computing gems Jade edition*. Elsevier, 2012, pp. 359–371.

[4] Bruno Bzeznik et al. “Nix As HPC Package Management System”. In: *Proceedings of the Fourth International Workshop on HPC User Support Tools*. HUST’17. Denver, CO, USA: ACM, 2017, 4:1–4:6. ISBN: 978-1-4503-5130-0. DOI: 10.1145/3152493.3152556. URL: http://doi.acm.org/10.1145/3152493.3152556.

[5] Timothy S Carpenter et al. “Capturing Phase Behavior of Ternary Lipid Mixtures with a Refined Martini Coarse-Grained Force Field”. In: *Journal of chemical theory and computation* 14.11 (2018), pp. 6050–6062.

[6] Francesco Di Natale. *Maestro Workflow Conductor*. 2017-07-10. Mar. 2019. URL: https://github.com/LLNL/maestrowf (visited on 03/29/2019).

[7] Francesco Di Natale et al. “A Massively Parallel Infrastructure for Adaptive Multiscale Simulations: Modeling RAS Initiation Pathway for Cancer”. In: *To appear in Supercomputing ’19: The International Conference for High Performance Computing*. SC ’19. 2019. DOI: 10.1145/1122445.1122456.

[8] Todd Gamblin et al. “The Spack Package Manager: Bringing Order to HPC Software Chaos”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’15. Austin, Texas: ACM, 2015, 40:1–40:12. ISBN: 978-1-4503-3723-6. DOI: 10.1145/2807591.2807623. URL: http://doi.acm.org/10.1145/2807591.2807623.

[9] W. Joubert et al. “Accelerated application development: The ORNL Titan experience”. In: *Computers and Electrical Engineering* 46 (2015).

[10] Ian Karlin et al. “Early Experiences Porting Three Applications to OpenMP 4.5”. In: *OpenMP: Memory, Devices, and Tasks: 12th International Workshop on OpenMP (IWOMP 2016)*. Ed. by Naoya Maruyama, Bronis R. de Supinski, and Mohamed Wahib. Cham: Springer International Publishing, Oct. 2016, pp. 281–292. ISBN: 978-3-319-45550-1. DOI: 10.1007/978-3-319-45550-1_20. URL: http://dx.doi.org/10.1007/978-3-319-45550-1_20.

[11] Edgar A. León. “Mapping MPI+X Applications to Multi-GPU Architectures: A Performance-Portable Approach”. In: *GPU Technology Conference*. GTC’18. San Jose, CA, Mar. 2018.

[12] Edgar A. León. “mpibind: A Memory-Centric Affinity Algorithm for Hybrid Applications”. In: *International Symposium on Memory Systems*. MEMSYS’17. Washington, DC: ACM, Oct. 2017.

[13] H. Nam et al. “The Trinity Center of Excellence co-design best practices”. In: *Computing in Science Engineering* 19.05 (2017).

[14] J. R. Neely and B. R. de Supinski. “Application Modernization at LLNL and the Sierra Center of Excellence”. In: *Computing in Science Engineering* 19.5 (2017), pp. 9–18. ISSN: 1521-9615. DOI: 10.1109/MCSE.2017.3421556.

[15] Lars Schneidenbach et al. “Data Broker: A Case for Workflow Enablement Using a Key/Value Approach”. In: The International Symposium on Memory Systems. 2019.