Migrating to Heterogenous Computing
Lessons Learned in the Sierra & El Capitan Centers of Excellence

David Richards

October 13, 2021
Acknowledgements - This talk builds on the work of many

Thank you to co-authors, code teams, support staff, COE Vendors, and everyone else who has helped to make Sierra a success

Johann Dahm  |  Brian Pudliner  |  Matt Cordery
Aaron Black  |  Adam Kunen     |  David Appelhans
Adam Bertsch |  David Dawson   |  Steve Rennich
Leopold Grinberg  |  Rich Hornung |  Max Katz
Ian Karlin   |  David Beckingsale |  Noah Reddell
Sara Kokkila-Schumacher |  Peter Robinson |  Nick Malaya
Edgar Leon   |  Tom Scogland   |  Judy Hill
Rob Neely    |  Holger Jones   |
Ramesh Pankajkshan |  David Poliakoff |
Olga Pearce  |  Jim Glosli     |
Brian Ryujin |  Bert Still     |
Jason Burmark |  Katie Lewis | And Many Others...
A Center of Excellence is a close partnership between the NNSA and system vendors

- Establish joint work plans, information sharing, and collaboration mechanisms
- Dedicated vendor staff work alongside lab code teams
  - Some staff assigned to work at lab sites
- Labs provide access to our codes
  - Including classified codes for those with security clearances
- Vendors provide non-public information and early access to hardware and software

Forming a Center of Excellence has become a recognized best practice for large DOE system procurements
DOE leadership systems clearly show we have now entered the heterogeneous era.

- **Perlmutter NERSC, 2020**
  - AMD CPU, Nvidia Tesla GPU

- **Frontier ORNL, 2021**
  - AMD CPU, AMD GPU, 1.5 ExaFlop

- **Aurora Argonne, 2022**
  - Intel CPU, Intel Xe GPU, > 1 ExaFlop

- **El Capitan LLNL, 2023**
  - AMD CPU, AMD GPU, > 1.5 ExaFlop
Our switch to GPU-based computing is paying off with big performance increases on Sierra

- **Ares, RT Mixing**: 13x speedup
- **Kull/Teton**: Radiating Sphere, 7x speedup
- **Ardra, Reactor Safety**: 16x speedup
- **ALE3D, Shaped Charge**: 8x speedup
- **SW4, Hayward Fault**: 28x speedup
Our large, integrated multi-physics codes are tailored to mission needs and HPC resources and support a broad range of simulation capabilities

- Millions of lines of code in multiple programming languages
- Scale to $O(1M)$ MPI ranks
- Multiple spatial/temporal scales
- Maintain connection to prior V&V efforts
- Coordinate with 10-60+ libraries

- Long life-time projects
  - 15+ years of development by large teams
  - 10–20+ people, ~50/50 CS/Physicists

- Portable performance
  - Our codes must be fast, reliable, and accurate on multiple systems
  - Laptops, Workstations, Commodity Clusters, Advanced Architectures, Heterogenous Architectures

Migrating these codes to new architectures is a significant challenge
Successful application modernization follows a consistent pattern

1. Refactor and remove anti-patterns
2. Create a mini-app to explore design space
3. Use portable abstractions and frameworks
4. Focus on a specific use case
5. Search for additional parallelism
6. Manually manage memory
7. Iteratively apply the steps above
Proxy apps are extremely useful to explore design and refactoring choices as well as performance bottlenecks.

**Quicksilver tracking loop times**
(weak scaling, lower is better)

- Fat Threads
- Thin Threads

![Graph showing tracking time for Power 8, KNL, Power 8, KNL, and Pascal GPU with Fat Threads and Thin Threads for 1 Node, 2 Nodes, and 4 Nodes.]

- Compiler Bug in Atomics

**Testing thread strategies for Mercury**

- Comb represents the data packing and communication for the halo exchange in Ares.

![Graph showing messaging time breakdown for small (Sm), medium (Md), and large (Lg) job sizes on Sierra.]

Proxy apps are also useful for benchmarking and vendor co-design.
RAJA is our performance-portability solution that uses standard C++ idioms to target multiple back-end programming models

- Decouple loop traversal and iterate (body)
- An iterate is a “task” (aggregate, (re)order, ...)
- IndexSet and execution policy abstractions simplify exploration of implementation/tuning options without disrupting source code
Unified (coherent) memory is helpful, but is not a panacea

No single strategy: multiple paths to success have emerged

- **SW4**: Allow managed memory to handle transfers. Overhead amortized by much re-use between transfers.
- **Ares**: Data transfers are explicit for performance. Managed memory pointers are helpful for libraries and code simplicity.
- **Teton**: All data transfers are explicit.

Abstractions improve code performance and developer productivity

- **CHAI**: Smart pointers automate explicit data transfers (Ardra, ALE3D)
- **Umpire**:
  - Unified, portable API to 3rd party memory capabilities
  - Coordinates memory use/introspection among multiple packages
  - Provides memory pools etc. to improve performance

Host-device data transfers must be treated as first class concerns
Umpire is being developed to coordinate complex memory allocations and movement

Assume three packages/libraries A, B, C – each with their own view of the GPU memory resources

**Phase 0**
- Initial state of problem staged in CPU memory

**Phase 1**
- A executes first
- A allocates temporary data in a memory pool (T)
- A’s data is copied to GPU

**Phase 2**
- A’s data copied back to CPU
  - Some shared data remains on the GPU
- B’s data copied to GPU
- Temporary data T deallocated
- B allocates temporary data T’ using the same memory pool
Performance improvement is an iterative process. Each step improves performance but also uncovers the next problem.

Example: Porting of ARES Lagrange hydro capability to a GPU

This result required sustained effort over long time by many people. Vendor partners and COE were critical to this success.
Fortran/OpenMP is not as well supported as C/C++ on GPUs

- Flang/F18 is likely to help with compiler availability
- OpenMP is the only real choice for portable GPU off-load in Fortran. No mechanism for abstraction layers.
- Modern Fortran and features not shared with C/C++ such as shaped arrays or array notation are especially problematic
- Write your Fortran code as much like C as possible if you want it to perform well
- Up-to-date proxies and tests are critical to ensuring compilers will function as desired

The Fortran community is relatively small compared to C++. We should pool effort and spread the overhead/effort.
For El Capitan, we use a 3-stage readiness process to prepare both applications and the system software environment

**Level 1: Basic Build & Run Testing:**
Exposé show-stopping problems in the application code or the software stack
- Uncover any major problems as early as possible. Some work-arounds might be needed
- Build and run on CPU & GPU
- Test basic operation of debuggers and performance tools

**Level 2: Performance Analysis and Comprehensive Tool Tests:**
Establish performance baselines, expand tool testing
- Establish performance baselines on production and early-availability systems
- Characterize application performance and identify bottlenecks
- Utilize additional compilers, debugging, and performance tools

**Level 3: Optimization and Testing at Scale:**
Improve code performance and test tools at scale
- Optimization is almost always an iterative process. Train application team to continue code development and optimization independently of COE staff
- Report/resolve performance problems due to abstraction layers
- Test performance and debugging tools at scales typically used by developers
- Verify that changes do not adversely impact performance on other important platforms

Both applications and system software require testing and preparation on new hardware
Feature matrices help communicate the status of complex application codes with many build options and/or capabilities

<table>
<thead>
<tr>
<th>Feature/Capability</th>
<th>Priority</th>
<th>CPU Build</th>
<th>GPU Build</th>
<th>GPU Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature 1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Library 2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Capability 3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Use Case 4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Feature 5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Capability 6</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We also track work-arounds and known issues that apply to specific applications.

Tracking features & priorities helps focus effort where needed.
Large architecture differences between systems greatly complicates performance comparisons

- What does “10x speedup” mean?
  - Is it useful to say, “my code is 10x faster on Sierra than CTS-1”

- Can we develop ways to characterize the behavior of a code on different platforms in a way that
  - Makes sense
  - Allows natural comparisons

- Idea: use “throughput curves” to characterize performance
  - DOFs vs DOFs/second
  - Curves represent:
    - One specific code and one specific problem of interest
    - One “dimension” in changing DOFs (varying zones)
    - Using all of the resources on a single node
  - Saturation Point
    - Resources are bottlenecked
    - 2x more DOFs = 2x more time

Based on slides by Adam Kunen
Throughput curves help characterize machine performance, and demonstrate the balance of parallelism and memory capacity

- **Throughput curve:**
  - Single node performance
  - DOFs/second
  - From very small problem to OOM

- **Throughput is related to:**
  - Available parallelism
  - Work required to saturate memory bandwidth
  - Work required to saturate compute

- **CPUs (CTS-1):**
  - Modest memory bandwidth and compute
  - Relatively small amount of parallelism
  - Easy to saturate, but lower overall performance

- **GPUs (Sierra):**
  - Very high memory bandwidth and compute
  - Large amount of parallelism
  - Need large amounts of work to saturate, but you get high performance in return

Ardra Single-Node Throughput on Sierra (GPU) vs CTS-1 (CPU)
3D, 48 groups, S8, P4 scattering, varying number of zones

- **27x faster at saturation!**
- **2x more memory per node**

Slide from Adam Kunen
Performance bottlenecks only become clear at saturation, analysis below that point is murky

- Profiling beyond saturation
  - Mostly results in same results
  - Shows bottlenecks

- Profiling under saturation
  - Produces different results
  - This can be really misleading

- Efficiency improvements can move the curve higher

- Algorithmic changes that expose more parallelism can move the curve to the left

---

Slide from Adam Kunen
A smaller fraction of memory needed to achieve peak throughput provides more flexibility

- **Headroom:**
  - Inverse fraction of memory left over once you hit peak throughput
  - How much strong scaling can happen before *on-node performance* suffers

- **Example:**
  - 8x headroom = 1/8 memory being used
  - Could fit an 8x larger problem in same resource
  - Could strong scale an 8x larger problem by 8x

- A larger headroom is more flexible:
  - More strong scaling potential
  - Smaller problems can be run efficiently

---

**Ardra Single-Node Throughput on Sierra (GPU) and CTS-1 (CPU)**

3D, 48 groups, S8, P4 scattering, varying number of zones

- 8x headroom
- 770x headroom

Slide from Adam Kunen
Four key lessons learned from large scale workflows

- Optimize resource allocations at the workflow level
  - Match workflow elements to hardware
  - Allocate data generators close to corresponding data consumers

- Use workflow management tools
  - Matching the available resources to ready tasks requires dedicated management software
  - Checkpointing a workflow can be harder than you think

- Consider the memory hierarchy and data sharing tools when designing a workflow
  - File I/O is not adequate to coordinate complex workflows

- Package managers and continuous integration can help ensure the reproducibility of a workflow

Complex workflows including machine learning, and real-time analytics or visualization are placing new demands on facilities
The El Capitan I/O system will support modern workflows with near-node local storage called Rabbits

- Each Rabbit will have 18 SSDs (16+2 spare) and 1 AMD EPYC CPU
- User tools can run in containers on the Rabbit CPU, e.g., SCR, UnifyFS, analytics tools, etc.
- Rabbits are connected via the system interconnect and PCIe connection to local compute nodes
  - User tools can communicate between Rabbits
  - User tools can also communicate between compute nodes and Rabbits
Sierra Center of Excellence work plans featured built-in flexibility

- Codes and libraries “phased in” over time
  - Multi-year plan overlaid with hardware and compiler availability to guide work plans

- Work plans were intentionally overcommitted to allow agility

- Earliest work focused on training and proxy apps

- After year one – pivoted to real applications and greater team engagement with vendor help
Effective collaboration doesn’t happen by accident

- Schedule activities to ensure two-way engagement
  - Trying to force interactions when priorities don’t align is a recipe for failure

- Be prepared to deviate from your plan
  - Things will go wrong
  - Opportunities will arise

- Invest in collaboration and software engineering tools
  - A common set of repos and communication tools will enhance productivity
  - Multi-site, secure tools can be hard to find
  - Avoid fragmentation of information

- Build multidisciplinary teams
  - Co-locate teams as much as possible.
Don’t expect sunshine, lollipops, and rainbows

- Usual new system pains: MPI, scheduler, compilers. These take time to resolve
- There is constant tension between system stability and bleeding-edge system software
- Some apps/algorithms aren’t there yet
  - Monte Carlo, Multi-grid setup phase
- Proprietary tool suites don’t handle all of our use cases
  - Vendor tools don’t always play nicely with HPC, scaling, MPI, etc.
  - Open source tools provide choices for debuggers, performance tools, etc.
- Interoperability of parallel models (OMP, CUDA, ...), compliers, memory handling, across libraries are still difficult
RADIUSS supports an LLNL-developed open source software stack for rapid and enduring HPC application development

Common Development Policies and Tooling
- "Harden" for broad adoption
- Ease developer movement between projects

Training and Documentation
- Encourage external adoption and contributions
- Not often a funded activity under research or program-specific projects

Leverage Programmatic Investments
- Programs provide base long-term support and development
- External outreach and community development

Integrate into LLNL Applications
- Provide path to both advanced architectures and software agility
- Long-term support and minor feature development

Product Categories
- Build Tools
- Physics and Math Libraries
- Application CS Infrastructure
- Data Management and Visualization
- Portable Execution & Memory Mgmt
- Performance Tools and Workflows

Foundational Concepts
- Continuous Integration, Release Management, Integrated Testing, Deployment, Outreach

Software is core infrastructure to the laboratory, similar to institutional HPC platforms and laboratory space. Sustained software investments are core to the mission of LLNL and our continued HPC leadership.
Migrating to heterogeneous architectures has taken years of hard work, but the results are worth it

- Many codes are seeing speedups of 10x or more
  - It is possible to incrementally refactor a large production code

- Code refactoring has reduced technical debt
  - But this takes commitment

- Increased performance is opening doors to previously impossible science

- Lessons learned and ecosystem improvements blaze a trail for others to follow
  - Future efforts should be easier/faster due to improvements in supporting software
References

- Sierra Center of Excellence: Lessons learned, IBMJRD vol 64, issue 3/4, 2:1-2:14
  — DOI: 10.1147/JRD.2019.2961069

- The Importance of Kernels for Performance Portability, Tom Scogland, 2020 P3HPC Forum
  — https://www.youtube.com/watch?v=6OypNNucCrY

Previous HPC Best Practices Webinars
  https://ideas-productivity.org/events/hpc-best-practices-webinars/

- An Overview of the RAJA Portability Suite (#50)
- Best Practices for Using Proxy Applications as Benchmarks (#39)
- Introduction to Kokkos (#37)
- Quantitatively Assessing Performance Portability with Roofline (#25)