Using RAJA for Accelerating LLNL Production Applications on the Sierra Supercomputer

GTC 2018, Silicon Valley

March 26 – 29, 2018

Rich Hornung, Computational Scientist, LLNL
Brian Ryujin, Computer Scientist, LLNL
The Sierra system will be LLNL’s first production GPU-accelerated architecture

Components

**Compute Node**
- 2 IBM POWER9 CPUs
- 4 NVIDIA Volta GPUs
- NVMe-compatible PCIe 1.6 TB SSD
- 256 GiB DDR4
- 16 GiB Globally addressable HBM2 associated with each GPU
- Coherent Shared Memory

**Compute Rack**
- Standard 19”
- Warm water cooling

**Compute System**
- 4320 nodes
- 1.29 PB Memory
- 240 Compute Racks
- 125 PFLOPS
- ~12 MW

**Spectrum Scale File System**
- 154 PB usable storage
- 1.54 TB/s R/W bandwidth

**Mellanox Interconnect**
- Single Plane EDR InfiniBand
- 2 to 1 Tapered Fat Tree

**IBM POWER9**
- Gen2 NVLink

**NVIDIA Volta**
- 7 TFlop/s
- HBM2
- Gen2 NVLink
Advanced architectures are daunting for production, multi-physics applications

- **Large codes**
  - $O(10^5) - O(10^6)$ LOC. Many kernels – $O(10^K)$ – none may dominate runtime

- **Usage diversity**
  - Must run on laptops, commodity clusters, large HPC platforms, ...

- **Long lived**
  - Used daily for decades, across multiple platform generations

- **Continual development**
  - Steady stream of new capabilities; verification & validation is essential

Such apps need *manageable performance portability*:

- Not bound to particular technologies (h/w or s/w)
- Platform-specific concerns (data, execution) insulated from algorithms
- Build and maintain portability without major disruption

RAJA is the path forward for a number of LLNL C++ apps & libraries.
RAJA targets portable loop parallelism while balancing performance and productivity

- **Easy to grasp** for (non-CS) application developers
- Supports **incremental and selective** adoption
- **Easily integrates** with application algorithm and data patterns
  - Loop bodies unchanged in most cases
  - Supports application-specific customizations
- Promotes implementation flexibility via **clean encapsulation**
  - Enables **application parameterization** via types
  - Focus on parallelizing **loop patterns**, not individual loops
  - **Localize modifications** in header files
  - Explore implementation options, systematic tuning

App developers typically wrap RAJA in a layer to match their code’s style.
RAJA is an open source project developed by CS researchers, app developers, and vendors

- RAJA Performance Suite: [https://github.com/LLNL/RAJAPerf](https://github.com/LLNL/RAJAPerf)
- RAJA proxy apps: [https://github.com/LLNL/RAJAProxies](https://github.com/LLNL/RAJAProxies)

RAJA is supported by LLNL programs (ASC and ATDM) and ECP (ST).
RAJA extends the common “parallel-for” idiom for loop execution

With traditional languages and programming models, many aspects of execution are explicit

RAJA encapsulates most execution details

C-style for-loop

```c
double* x; double* y;
double a, sum = 0;
for ( int i = beg; i < end; ++i ) {
    y[i] += a * x[i];
    sum += y[i];
}
```

RAJA-style loop

```cpp
double* x; double* y;
double a;
RAJA::SumReduction<reduce_policy, double> sum(0);
RAJA::RangeSegment range(beg, end);
RAJA::forall<exec_policy>(range, [=](int i) {
    y[i] += a * x[i];
    sum += y[i];
});
```
Users express loop execution using four concepts

using EXEC_POLICY = RAJA::cuda_exec;

RAJA::forall<EXEC_POLICY>( RAJA::RangeSegment(0, N),
  [=] (int i)
  {
    y[i] += a * x[i];
  });

1. **Loop traversal template** (e.g., ‘forall’)
2. **Execution policy** (seq, simd, openmp, cuda, etc.)
3. **Iteration space** (range, index list, index set, etc.)
4. **Loop body** (C++ lambda expression)
RAJA reducer types hide complexity of parallel reduction implementations

RAJA::ReduceFoo< \texttt{reduce\_policy}, \texttt{type} > \texttt{foo(in\_value);};

RAJA::forall< \texttt{exec\_policy} > (\ldots \{ 
  \texttt{foo op func(i);} 
\});

type \texttt{reduced\_val} = \texttt{foo.get();}

- A reducer type requires:
  - Reduce policy
  - Reduction value type
  - Initial value

- Updating reduction value (in loop) is simple (+=, min, max)

- After loop, get reduced value via ‘get’ method or type cast

Multiple RAJA reducer objects can be used in a single kernel.
Some notes about C++ lambda expressions...

- A C++ lambda is a **closure** that stores a function with a data environment

  
  ```c++
  [ capture list ] ( param list ) { function body }
  ```

- Capture by-value or by-reference ( [=] vs. [&] )?
  - **Value capture is required** when using CUDA, RAJA reductions, ...

- With nvcc, a lambda passed to a CUDA device function **must have** the “__device__” annotation; e.g.,

  ```c++
  forall< cuda_exec >(range, [=] __device__ (int i) {
    ...
  });
  ```

- Other lambda capture issues require care (global vars, stack arrays)
RAJA iteration space types are used to aggregate, partition, (re)order, ... loop iterates

- A “Segment” defines a set of loop indices to run as a unit
  - Stride-1 range \([\text{beg}, \text{end}]\)
  - Strided range \([\text{beg}, \text{end}, \text{stride}]\)
  - List of indices (indirection)

- An “Index Set” is a container of segments

- All IndexSet segments can be run in a single RAJA traversal

User-defined Segment types can also be used in RAJA traversals
An example of how we use IndexSets...

- Multi-physics codes use indirection arrays (a lot!): unstructured meshes, material regions on a mesh, etc.
  - Indirection impedes performance: index arithmetic, irregular data accesses, etc.

- Consider a real hydrodynamics problem:
  - 16+ million zones (many multi-material)
  - Most loops have “long” stride-1 indexing

- Casting stride-1 ranges as RangeSegments can improve performance (in real codes)

<table>
<thead>
<tr>
<th>Range length</th>
<th>% iterates</th>
</tr>
</thead>
<tbody>
<tr>
<td>16+</td>
<td>84%</td>
</tr>
<tr>
<td>32+</td>
<td>74%</td>
</tr>
<tr>
<td>64+</td>
<td>70%</td>
</tr>
<tr>
<td>128+</td>
<td>69%</td>
</tr>
<tr>
<td>256+</td>
<td>67%</td>
</tr>
<tr>
<td>512+</td>
<td>64%</td>
</tr>
</tbody>
</table>

Index sets can expose SIMD-izable ranges “in place” to compilers. This obviates the need for gather/scatter operations.
RAJA support for complex kernels is being reworked...

- **Application integration revealed new requirements:**
  - More flexible execution policies
  - Capabilities beyond loop nesting, tiling, and collapsing

- **New design/implementation supports:**
  - Simpler expression of CUDA kernel launch parameters
  - Loops not perfectly nested (i.e., intervening code)
  - Shared memory Views ("tiles") for GPU & CPU
  - Thread local (register) variables
  - Loop fusion and other optimizations

Available as “pre-release” now (apps using it). RAJA release coming within a month....
CUDA matrix multiplication kernel to compare with RAJA features for more complex kernels...

```c
__global__ void matMult(int N, double* C, double* A, double* B)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if ( row < N && col < N ) {
        double dot = 0.0;
        for (int k = 0; k < N; ++k) {
            dot += A[N*row + k] * B[N*k + col];
        }
        C[N*row + col] = dot;
    }
}

// Launch kernel...
    dim3 blockdim(BLOCK_SZ, BLOCK_SZ);
    dim3 griddim(N / blockdim.x, N, blockdim.y);
    matMult<<< griddim, blockdim >>>(N, C, A, B);
```

// Each thread computes one row-col dot product

// Rows and cols assigned to blocks & threads
One way to write the CUDA mat-mult kernel with RAJA...

```cpp
double* A = ...;
double* B = ...;
double* C = ...;

RAJA::View< double, RAJA::Layout<2> > Aview(A, N, N);
RAJA::View< double, RAJA::Layout<2> > Bview(B, N, N);
RAJA::View< double, RAJA::Layout<2> > Cview(C, N, N);

RAJA::kernel<EXEC_POL>(RAJA::make_tuple(col_range, row_range),
    [=] RAJA_DEVICE (int col, int row) {
        double dot = 0.0;
        for (int k = 0; k < N; ++k) {
            dot += Aview(row, k) * Bview(k, col);
        }
        Cview(row, col) = dot;
    });
```

A View wraps the pointer for each matrix to simplify multi-dimensional indexing.

Lambda body is the same as CUDA kernel body (mod. Views)

RAJA Views and Layouts can be used to do other indexing operations, permutations, etc.
And, the RAJA nested execution policy..

```cpp
using EXEC_POL =
  RAJA::KernelPolicy<
    RAJA::statement::CudaKernel<
      RAJA::statement::For<1, RAJA::cuda_threadblock_exec<BLOCK_SZ>,
      RAJA::statement::For<0, RAJA::cuda_threadblock_exec<BLOCK_SZ>,
      RAJA::statement::Lambda<0>
    >
  >;
```

This policy defines the same kernel launch as the raw CUDA version.

Rows(1) and cols(0) indices assigned to blocks & threads as before
RAJA also provides portable atomics and scans

- **Atomic memory updates** (write, read-modify-write):
  - Arithmetic, min/max, incr/decr, bitwise-logical, replace
  - “built-in” policy for compiler-provided atomics
  - Interface similar to C++ std::atomic also provided

- **Parallel scan** support:
  - Exclusive and inclusive
  - In-place and separate in-out arrays
  - Prefix-sum is default, other ops are supported (min, max, etc.)
  - RAJA CUDA scan support uses CUB internally
Current status of RAJA features for each programming model back-end

<table>
<thead>
<tr>
<th></th>
<th>Seq</th>
<th>SIMD</th>
<th>OpenMP (CPU)</th>
<th>OpenMP (target)</th>
<th>CUDA</th>
<th>TBB</th>
<th>ROCm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single loops</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complex loops</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Segments &amp; Index sets</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Layouts &amp; Views</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reductions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atomics</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scans</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Green = available
- Yellow = in progress
- Red = not available
Case Study: Ares
Ares is a massively parallel, multi-dimensional, multi-physics code

**Physics Capabilities:**
- ALE-AMR Hydrodynamics
- High-order Eulerian Hydrodynamics
- Elastic-Plastic flow
- 3T plasma physics
- High-Explosive modeling
- Diffusion, Sn Radiation
- Particulate flow
- Laser ray-tracing
- Magnetohydrodynamics (MHD)
- Dynamic mixing
- Non-LTE opacities

**Applications:**
- Inertial Confinement Fusion (ICF)
- Pulsed power
- National Ignition Facility Debris
- High-Explosive experiments
Porting a large, existing code comes with considerable challenges

- Ares is 22 years old and ~800k line C/C++ code base
  - Integrates with 60+ libraries in C, C++ and Fortran
  - Has scaled to over 1.5 million MPI processes

- 13 code developers: physicists, mathematicians, engineers and computer scientists

- Ares is used daily on our current supercomputers
  - Cannot break or slow down current functionality
  - Must continue to add new functionality that users request throughout the process

- Code overall has ~5,000 mesh loops with limited hotspots
  - Lagrange hydro problem runs 80+ kernels
  - Grey radiation diffusion problem runs 250+ kernels
  - Arbitrary Lagrangian-Eulerian (ALE) hydro problem runs 450+ kernels

We can only maintain a single code base, but must effectively utilize all HPC platforms
Ares strategy for Sierra

- We tried to adhere to some basic guiding principles
  - Keep strategies relatively simple
  - Leverage existing capabilities and infrastructures
  - Keep concepts familiar to developers

- Overarching approach:
  - Use RAJA to get code to run on the GPU
  - Use Unified Memory to get mesh data onto the GPU
  - 1 MPI process per GPU
  - Keep all data resident on the GPU to avoid data motion

We believe our approach follows our principles, but the devil is in the details...
We encapsulate code concepts in the wrapper layer, which has improved readability in the code.

There is a single place to add hooks into our loop constructs:
- Enables a simple way of using host and device simultaneously with the same code
- Enables us to instrument the code for detailed analysis

Separates responsibility between algorithm development and mapping policies and patterns for machine specific optimizations.

Ares implemented a wrapper layer around RAJA

C-style for-loop
```c
double* x; double* y;
double a;
Int* ndx = domain->Zones;
for ( int i = begin; i < end; ++i ) {
    int zone = ndx[i];
    y[zone] += a * x[zone];
}
```

Ares-RAJA-style loop
```c
Ares-RAJA Transformation
double* x; double* y;
double a;
domain_t domain;
for_all_zones<parstream>(domain, [=](int zone) {
    y[zone] += a * x[zone];
});
```

See Olga Pearce’s talk tomorrow at 9:00
RAJA provides us with additional flexibility at little cost to code maintainability and familiarity

- After the port to RAJA, the code still looks very similar
- Over 98% of our loops could be ported in a straightforward manner
- RAJA’s ability to use multiple backends is an invaluable tool
  - Easy to switch between backends (Serial, OpenMP3, CUDA)
  - Serial performance is comparable to non-RAJA code
  - Enables use of CPU analysis tools
    - Debugging: Totalview, DDT, Valgrind, etc.
    - Thread correctness: Archer, thread sanitizer, etc.
- Code benefits directly from performance improvements in RAJA
  - Platform specific optimizations are hidden within RAJA’s primitives
RAJA does not define a memory management strategy, which allows codes to use their own

- Ares has ~5000 malloc calls that are each wrapped by a macro

- We first replaced the macro to use cudaMallocManaged
  - Worked correctly and compatible with non-RAJAfied code and required minimal effort!
  - Performed poorly due to the cost of the allocations and frees

- For performance, we now differentiate between types of memory
  - malloc – CPU control code
  - cudaMallocManaged (UM) – For mesh data (accessed on CPU and GPU)
  - cudaMalloc (cnmem memory pools) – Temporary GPU data

- Switching from a naïve single tier system to a three tier system gave a 14x speedup on current hardware

UM is a great tool for productivity, but is not a silver bullet
The Radiation-Hydrodynamics core of the code is mostly bandwidth bound

To set expectations, we look at effective memory bandwidth of the architectures

For the GPUs, we are only looking at GPU bandwidth

<table>
<thead>
<tr>
<th></th>
<th>CTS-1 (Dual Socket Broadwell)</th>
<th>IBM Early Access (EA) (2x P8 CPU + 4x P100 GPU)</th>
<th>Sierra (2x P9 CPU + 4x V100 GPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Effective Memory Bandwidth per node</td>
<td>130 GB/s</td>
<td>2200 GB/s</td>
<td>3400 GB/s</td>
</tr>
</tbody>
</table>

This is not a perfect measure, but it is a good place to start
Ares + RAJA utilizes both CPU and GPU architectures effectively

<table>
<thead>
<tr>
<th>Resources</th>
<th># of Nodes</th>
<th>Runtime (min)</th>
<th>Relative speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>576 CPU cores</td>
<td>16</td>
<td>909</td>
<td>1</td>
</tr>
<tr>
<td>1152</td>
<td>32</td>
<td>454</td>
<td>2</td>
</tr>
<tr>
<td>2304</td>
<td>64</td>
<td>239</td>
<td>3.8</td>
</tr>
<tr>
<td>4608</td>
<td>128</td>
<td>124</td>
<td>7.3</td>
</tr>
<tr>
<td>32 P100s</td>
<td>8</td>
<td>131</td>
<td>6.9</td>
</tr>
<tr>
<td>64 P100s</td>
<td>16</td>
<td>83</td>
<td>10.9</td>
</tr>
<tr>
<td>32 V100s</td>
<td>8</td>
<td>97</td>
<td>9.4</td>
</tr>
<tr>
<td>64 V100s</td>
<td>16</td>
<td>69</td>
<td>13.2</td>
</tr>
</tbody>
</table>

Sierra promises access to an incredible amount of computational power for our scientists

RT Mixing Layer in a Convergent Geometry
- $4\pi$, 191.1M zones
- 14,500 cycles
- ALE Hydrodynamics
- Dynamic Species
Ares + RAJA utilizes both CPU and GPU architectures effectively

Sierra promises access to an incredible amount of computational power for our scientists

Bandwidth vs. Runtime

- Broadwell Ideal
- Broadwell
- P100
- V100

RT Mixing Layer in a Convergent Geometry
- $4\pi$, 191.1M zones
- 14,500 cycles
- ALE Hydrodynamics
- Dynamic Species
We are beginning to tap SIERRA’s resources to run high fidelity calculations

<table>
<thead>
<tr>
<th>Resources</th>
<th># of Nodes</th>
<th>Runtime (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>256 V100 GPUs</td>
<td>64</td>
<td>213</td>
</tr>
</tbody>
</table>

RT Mixing Layer in a Convergent Geometry
- $4\pi$, 1.52B zones
- 29,375 cycles
- ALE Hydrodynamics
- Dynamic Species

Sierra will make high fidelity calculations like these routine instead of heroic
High fidelity multi-physics calculations are essential for model validation and development

<table>
<thead>
<tr>
<th>Resources</th>
<th># of Nodes</th>
<th>Runtime (min)</th>
<th>Relative speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>576 CPU cores</td>
<td>16</td>
<td>755</td>
<td>1</td>
</tr>
<tr>
<td>1152</td>
<td>32</td>
<td>377</td>
<td>2</td>
</tr>
<tr>
<td>2304</td>
<td>64</td>
<td>194</td>
<td>3.88</td>
</tr>
<tr>
<td>4608</td>
<td>128</td>
<td>109</td>
<td>6.92</td>
</tr>
<tr>
<td>48 P100s</td>
<td>12</td>
<td>77</td>
<td>9.76</td>
</tr>
<tr>
<td>64 P100s</td>
<td>16</td>
<td>62</td>
<td>12.23</td>
</tr>
</tbody>
</table>

Reacting RT Mixing Layer
- 4π, 191.1M zones
- 750 cycles
- ALE Hydrodynamics
- Dynamic Species
- Grey Radiation Diffusion
- Thermonuclear Burn

Sierra will improve our understanding of physics
High fidelity multi-physics calculations are essential for model validation and development.

Reacting RT Mixing Layer
- $4\pi$, 191.1M zones
- 750 cycles
- ALE Hydrodynamics
- Dynamic Species
- Grey Radiation Diffusion
- Thermonuclear Burn

Sierra will improve our understanding of physics.
RAJA has allowed us to make steady progress porting to GPUs while supporting our users.
RAJA has enabled Ares to make consistent progress on porting to GPUs
- Over 98% of the loops port cleanly
- Code remains readable and familiar to all developers

GPU performance tracks CPU performance when given an equal amount of memory bandwidth

Multiple RAJA backends allows us to use different programming models, including tools developed for them to help port the code
- e.g. OpenMP + Archer for thread correctness
- This is an effective technique only because we still have a single code base

Sierra will allow us to run higher fidelity simulations, which will improve our scientific understanding of physical phenomena
Acknowledgements

- RAJA Team
  - Rich Hornung
  - Jeff Keasler
  - Holger Jones
  - Adam Kunen
  - Tom Scogland
  - David Beckingsale
  - Will Killian
  - Arturo Vargas

- Ares Team
  - Brian Ryujin
  - Brian Pudliner
  - Jason Burmark
  - Mike Collette
  - George Zagaris
  - Olga Pearce
  - Brandon Morgan
  - Burl Hall

- Ascent Team
  - Cyrus Harrison
  - Matt Larsen
Contact information and links

- Rich Hornung: hornung1@llnl.gov
- Brian Ryujin: ryujin1@llnl.gov
- RAJA: https://github.com/LLNL/RAJA
- Ascent: https://github.com/Alpine-DAV/ascent