

# An Overview of the RAJA Portability Suite

Approved for public release



**Arturo Vargas**, Rich Hornung (LLNL)  
RAJA/Kokkos Project WBS 2.3.1.18

HPC Best Practices Webinar Series  
March 10, 2021



LLNL-PRES-819903

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



U.S. DEPARTMENT OF  
**ENERGY**

Office of  
Science

# The RAJA Portability Suite provides complementary open-source tools for portable execution and memory management

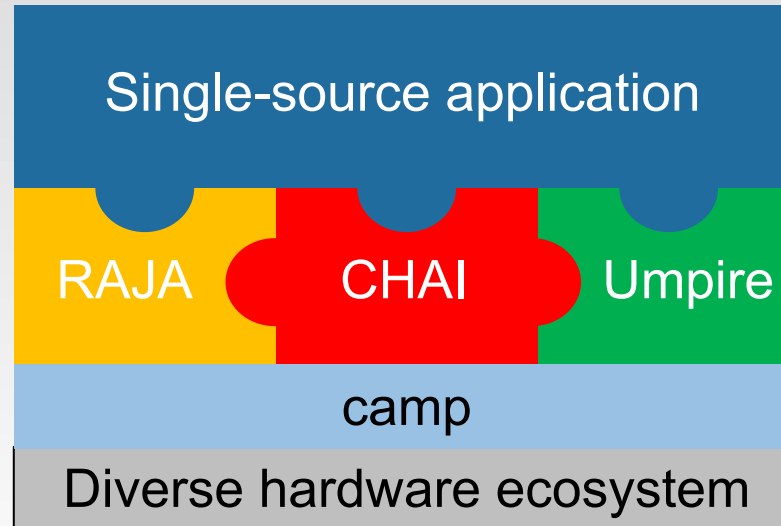


## RAJA: C++ kernel execution abstractions

- Enables apps to target various programming model back-ends while maintaining **single-source** app code

## CHAI: C++ array abstractions

- Automates data copies, giving look and feel of unified memory



<https://github.com/LLNL/RAJA>

<https://github.com/LLNL/CHAI>

<https://github.com/LLNL/Umpire>

<https://github.com/LLNL/camp>



## Umpire: memory API

- Provides high performance memory operations, such as pool allocations. **Native C++, C, Fortran APIs**

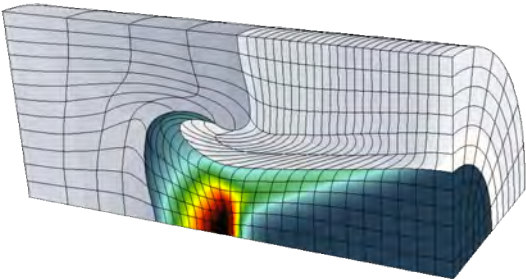


## camp: low-level C++ metaprogramming facilities

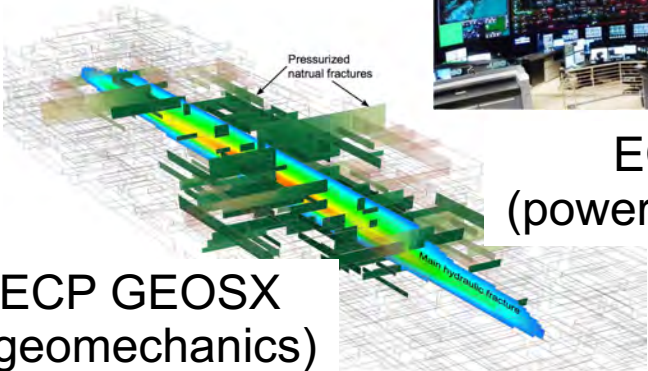
- Focuses on HPC compiler compatibility

# The RAJA Portability Suite insulates applications from many complexities of a diverse hardware ecosystem

## ECP apps using RAJA software tools



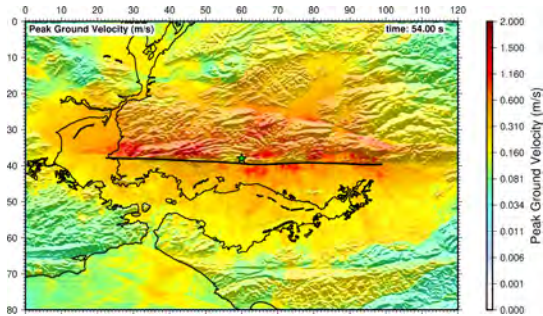
LLNL ECP/ATDM  
(high-order ALE hydro)



ECP GEOSX  
(geomechanics)



ECP ExaSGD  
(power grid optimization)



ECP SW4  
(earthquake modeling)

plus  
others...

## RAJA / Umpire / CHAI



Perlmutter (LBL)  
AMD Milan CPUs +  
NVIDIA Ampere GPUs



Astra (SNL)  
ARM architecture



Sierra (LLNL)  
IBM P9 CPUs + NVIDIA Volta GPUs



Aurora (ANL)  
Intel Xeon CPUs + Xe GPUs



Frontier (ORNL) &  
El Capitan (LLNL)  
AMD CPUs + GPUs



# Most ASC applications plus others at LLNL also rely on the RAJA Portability Suite to run on a wide range of platforms

## Major LLNL ASC Program Applications

	Ares	ALE3D	Kull	MARBL	Ardra	Mercury	Teton	Hydra
Language	C++	C++	C++	C++ & Fortran	C++	C++	Fortran	C++/C
CPU / GPU Execution Model	RAJA	RAJA	RAJA	RAJA + MFEM & OpenMP	RAJA	CUDA & RAJA	OpenMP & CUDA-C (poss. RAJA)	Exploring OpenMP, CUDA, RAJA
Data Transfer	UM + Explicit	CHAI	UM	Explicit	CHAI	UM	Explicit	Explicit, Exploring CHAI
Memory Allocation	Umpire	Umpire	Umpire	Umpire	Umpire	Umpire	Umpire	Explicit, Exploring Umpire

- Integration of these projects into other applications and libraries is ongoing

The LLNL institutional RADIUSS effort promotes and funds integration of these tools into non-ASC applications.



# RAJA supports a variety of loop patterns and parallel constructs

## Simple & complex loop patterns & execution

- Non-perfectly nested loops
- Loop tiling
- Hierarchical parallelism
- Shared and thread local memory

## Multiple execution back-ends

- Sequential
- SIMD (via vector intrinsics, in progress)
- OpenMP (CPU & device offload)
- Intel Threading Building Blocks (partial)
- CUDA
- AMD HIP
- SYCL (in development)

## Loop transformations (without changing app code)

- Change loop iteration patterns, permute loop nest ordering
- Multi-dimensional data views with offsets and index permutations
- Fine-grained GPU thread-block mapping control
- Hierarchical parallelism, asynchronous execution

- Portable reductions, scans, atomic operations, sorts...
- GPU kernel fusing (to reduce impact of GPU launch overhead for small kernels)
- Other work in progress
  - API to encapsulate SIMD/vectorization intrinsics
  - Dynamic plugins to enable tool integration

# A simple example shows how RAJA abstracts kernel execution

## C-style dot product

```
double dot= 0.0;
for (int i = 0; i < N; ++i)
{
    dot += a[i] * b[i];
}

std::cout << "dot = " << dot;
```

**RAJA  
Transformation**

## RAJA-style dot product

```
using EXEC_POL = ...;
using REDUCE_POL = ...;
RAJA::RangeSegment it_space(0, N);
```

Definitions like these  
typically go in header files

```
RAJA::ReduceSum< REDUCE_POL, double > dot(0.0);
RAJA::forall< EXEC_POL >(it_space,
                        RAJA_LAMBDA (int i)
{
    dot += a[i] * b[i];
} );

std::cout << "dot = " << dot.get();
```

In the C-style kernel, all aspects of execution are explicit in the source code; e.g., sequential execution, iteration ordering, etc.

RAJA allows you to change how a kernel runs without changing the source code.

# RAJA kernel execution has four core concepts

```
using EXEC_POL = ...;
RAJA::RangeSegment it_space(0, N);
① RAJA::forall< EXEC_POL ② >(it_space, ③
                                RAJA_LAMBDA (int i)
                                {
                                  c[i] = a[i] * b[i]; ④
                                } );
```

1. Loop execution template (e.g., 'forall')
2. Loop execution policy type (EXEC\_POL)
3. Loop iteration space (e.g., 'RangeSegment')
4. Loop body (C++ lambda expression)

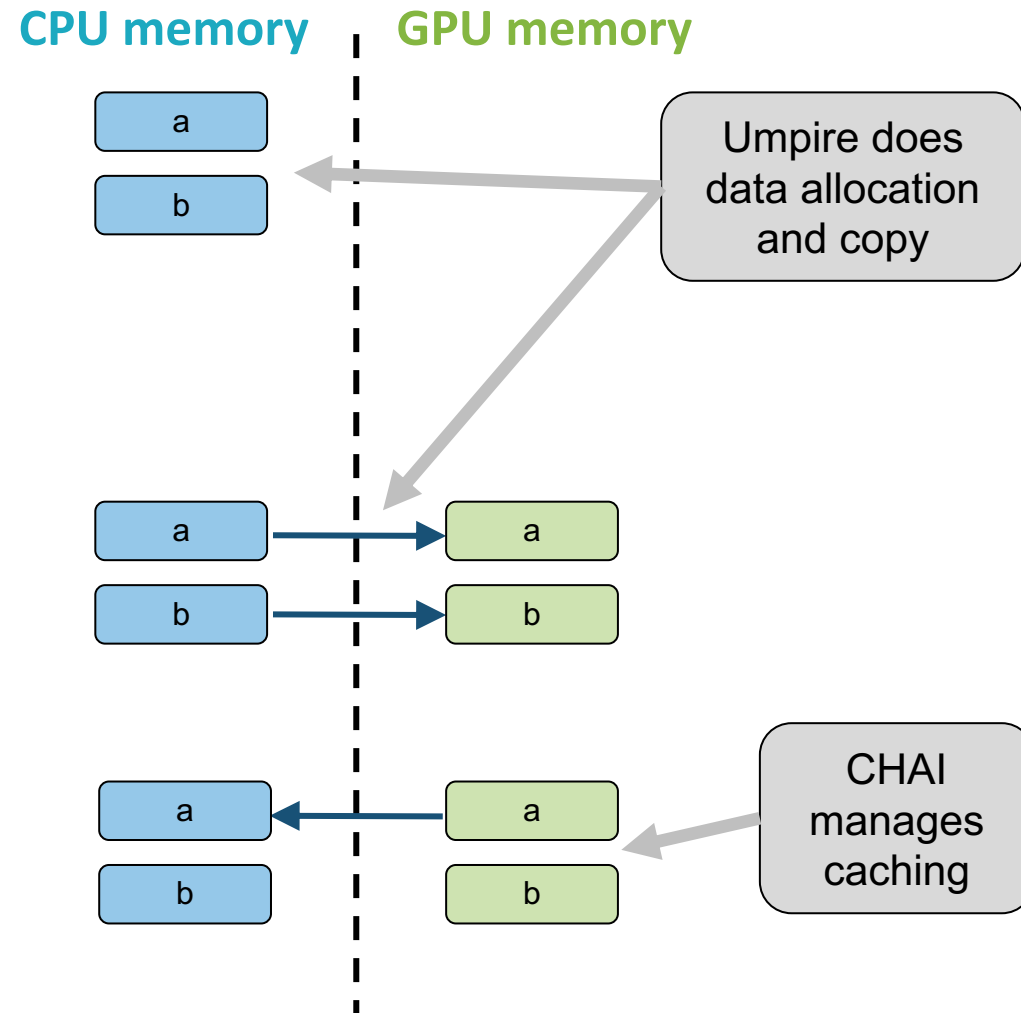
**We'll return to RAJA  
in a bit after we  
introduce Umpire  
and CHAI...**





# CHAI's “managed array” abstraction transfers data automatically at run time as needed to run kernels

```
chai::ManagedArray<float> a(100);  
chai::ManagedArray<const float> b(100);  
  
// ...  
  
RAJA::RangeSegment range(0, 100);  
  
// Run GPU kernel  
RAJA::forall<RAJA::cuda_exec>(range,  
                             RAJA_LAMBDA (int i)  
{ a[i] += b[i]; } );  
  
// Run CPU kernel  
RAJA::forall<RAJA::seq_exec>(range,  
                             RAJA_LAMBDA (int i)  
{ std::cout << "a[i] = " << a[i] << "\n"; } );
```



# CHAI's “managed pointer” simplifies the use of virtual class hierarchies across host and device memory spaces

- `managed_ptr` will make a copy of your object hierarchy in device memory

```
void overlay( Shape* shape, double* mesh_data ) {  
    chai::managed_ptr< Shape > mgd_shape = shape->makeManaged();  
    RAJA::forall< cuda_exec > ( ... {  
        mgd_shape->processData(mesh_data[i]);  
    } );  
    mgd_shape.free();  
}
```

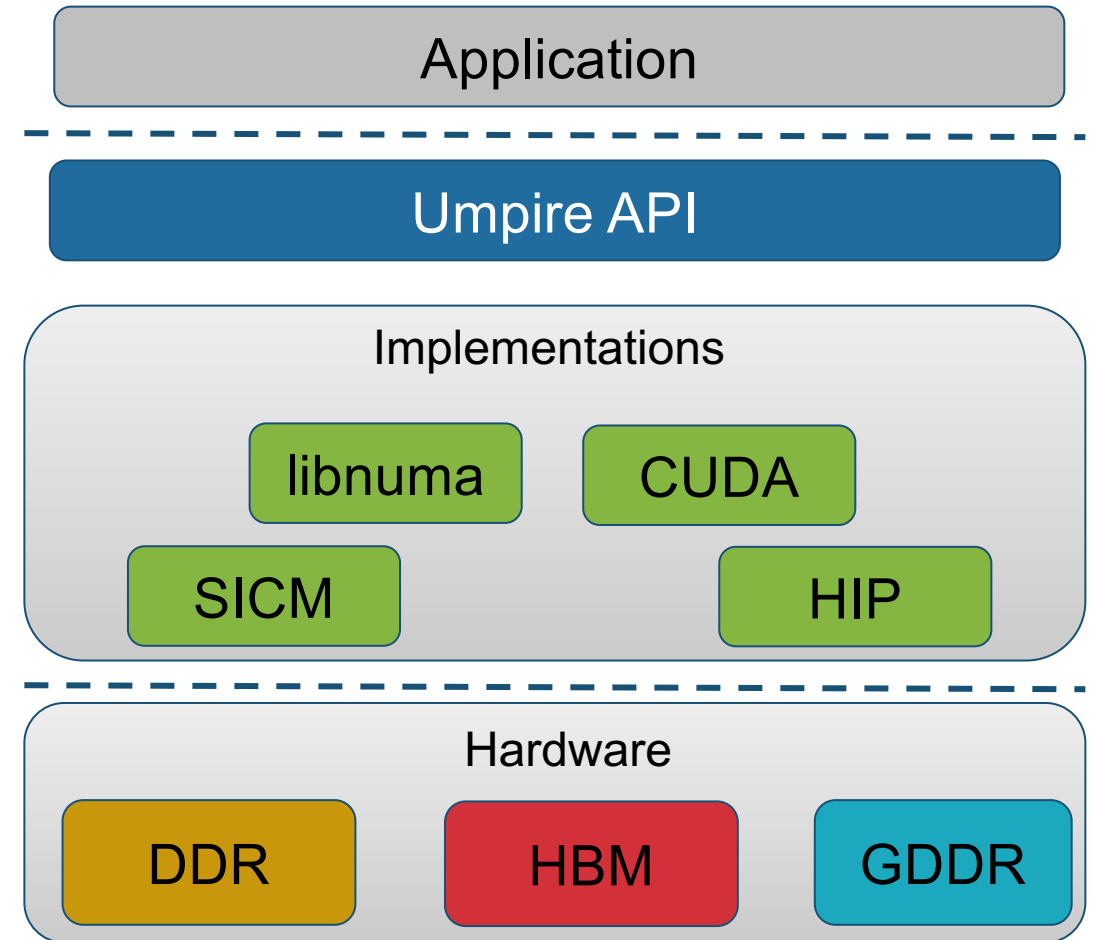
- This requires a method to clone objects and host-device annotations on class constructors

```
chai::managed_ptr< Shape > Sphere::makeManaged( ) { ... }  
__host__ __device__ Sphere::Sphere( ... ) { ... }
```

This mechanism allows you to use C++ virtual class hierarchy code on CPUs and GPUs without a major refactor.

# Umpire provides a unified, portable memory management API

- Allocate, deallocate, copy, move, query
- Memory pools
  - Much faster allocation & deallocation than malloc(), cudaMalloc()...
  - Easily shared between application components
- Introspection for better decision-making
  - Where does data associated with this pointer live?
  - Which allocator was used for this allocation?
  - What is the size of this allocation?
  - How much memory is being used on this resource?



# Umpire interface concepts allow application developers to reason about memory use

- A **Memory Resource** is a kind of memory, with specific performance and accessibility characteristics
- An **Allocation Strategy** decouples how and where allocations are made, allowing complex allocation mechanisms
  - Memory pools, thread-safety layers, specific algorithms for memory allocation, etc.
- An **Allocator** is a lightweight interface for making an allocation and querying it
  - One interface for all resources
- An **Operation** manipulates data in memory through one interface regardless of resource
  - Copy, move, reallocate, memset, etc.
- These concepts are coordinated by a **ResourceManager**
  - Builds allocators based on allocation strategies and available resources, dispatches operations based on pointer locations, etc.

```
auto& rm = umpire::ResourceManager::getInstance();
auto host = rm.getAllocator("HOST");
auto device = rm.getAllocator("DEVICE");

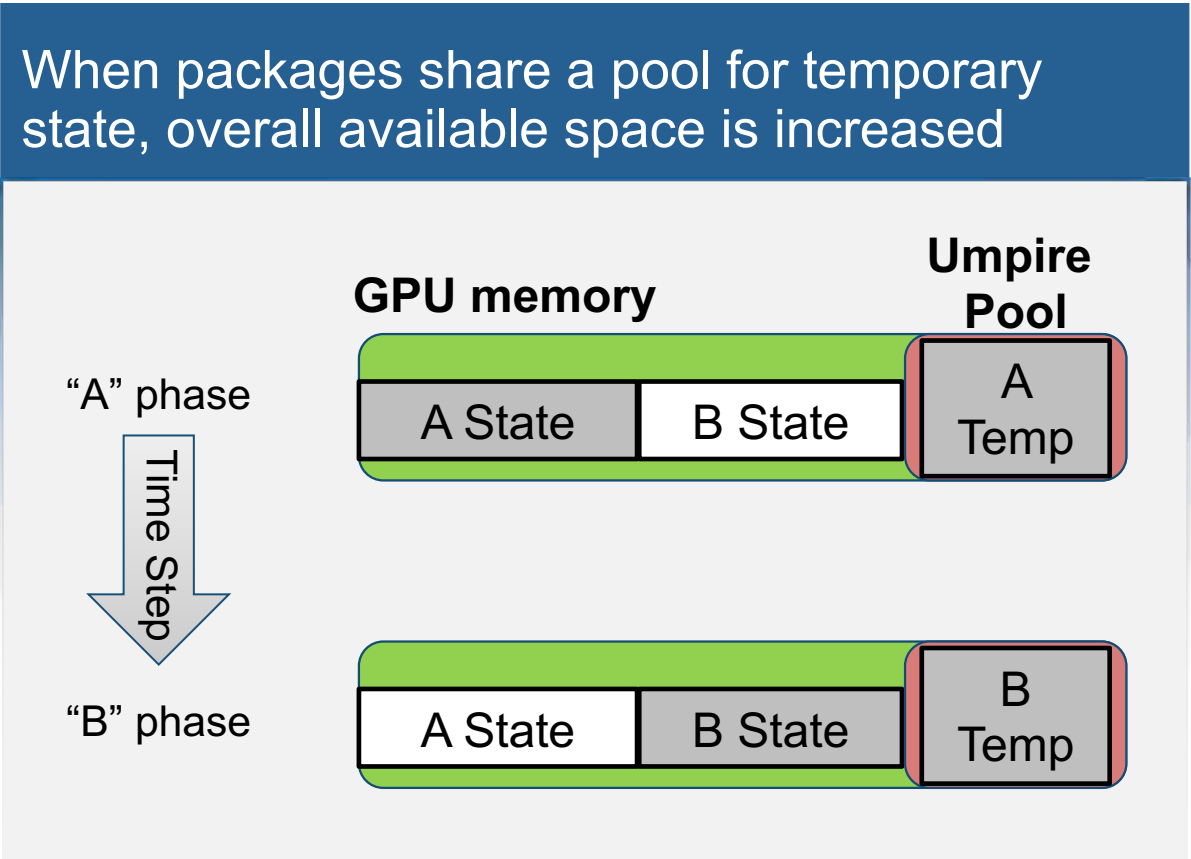
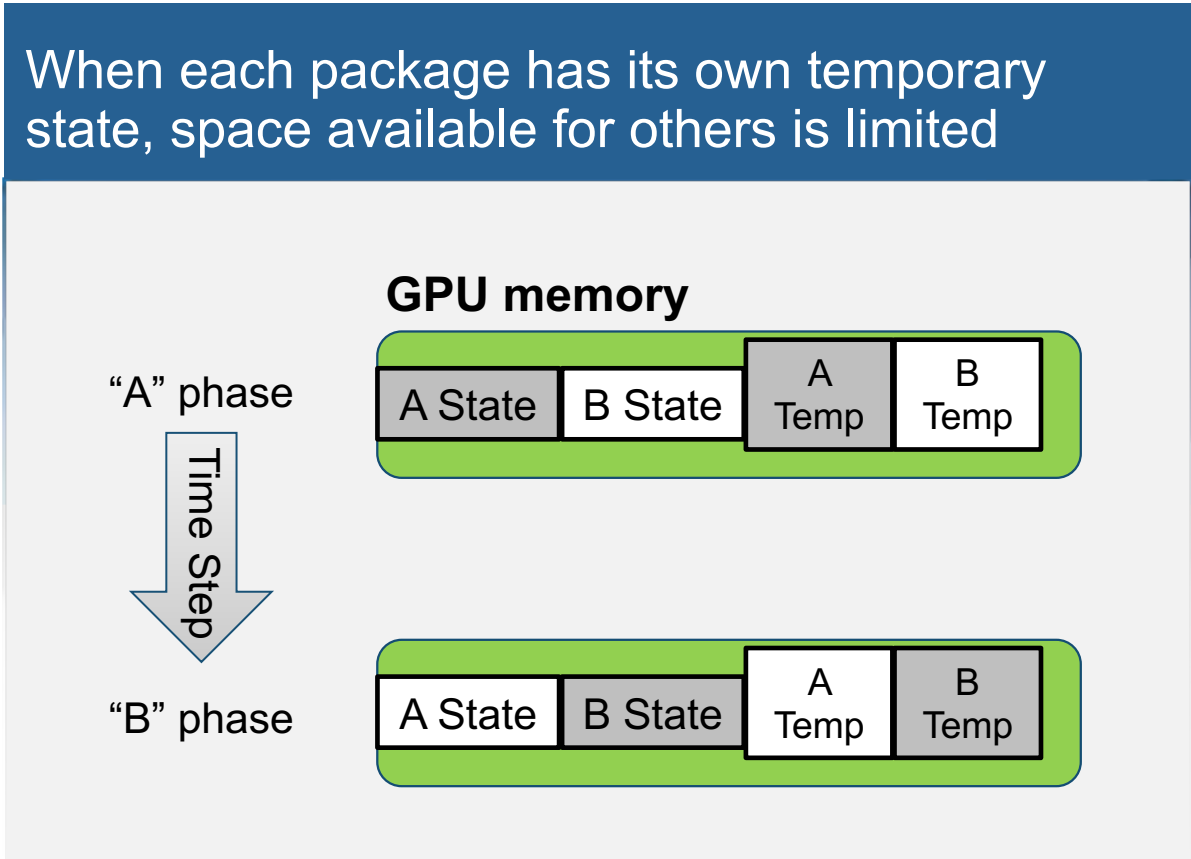
auto device_pool =
    rm.makeAllocator<DynamicPool>("MY_POOL", device);

void* host_data = host.allocate(1024);
void* dev_data = device_pool.allocate(1024);

rm.memset(host_data, 0);
rm.copy(dev_data, host_data);

host.deallocate(host_data);
```

# Sharing GPU memory pools among packages in multiphysics applications enables larger problems to be run





# Umpire provides a variety of memory management capabilities

## Intuitive concepts

- Resources
- Allocators
- Operations

## Supported memory types

- Host (CPU)
- GPU global, constant, (host) pinned
- Unified memory
- Mmapped file memory
- Support for NVIDIA, AMD, and Intel GPU devices available in recent releases

## Features useful in HPC applications

- Various pool allocation strategies (fixed size, dynamic, monotonic, etc.)
- NUMA support
- Memory allocation advice (preferred location, mostly read, etc.)
- Thread safe allocators
- Memory introspection

- Native interfaces for C++, C, and Fortran
- Logging, backtrace, and “replay” capabilities. These are really useful for investigating application performance, finding bugs, etc.

**Returning to RAJA, we'll  
introduce two APIs for  
nested/complex loop  
kernels**



# We will use a matrix multiplication kernel to explore some RAJA features and usage

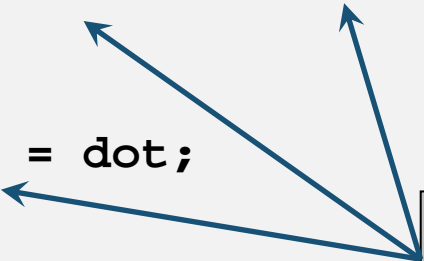
$C = A * B$ , where A, B, C are N x N matrices

C-style  
nested  
for-loops

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

# Nesting RAJA “forall” statements is not a good approach because loops are treated as independent entities

```
forall< exec_policy_row >( row_range, [=](int row) {  
    forall< exec_policy_col >( col_range, [=](int col) {  
  
        double dot = 0.0;  
  
        for (int k = 0; k < N; ++k) {  
            dot += A(row, k) * B(k, col);  
        }  
  
        C(row, col) = dot;  
    } );  
} );
```



Note: RAJA Views simplify multi-dimensional indexing.

- Parallelize row loop?
  - Each thread runs all code in column loop sequentially
- Parallelize column loop?
  - Launch new parallel computation for each row → unwanted synchronization
- Loop interchange and other transformations require source code changes → breaks RAJA encapsulation!

# The RAJA *kernel* API is designed to compose and transform complex parallel kernels, without changing kernel source code

```
using namespace RAJA;
using KERNEL_POL = KernelPolicy<
    statement::For<1, row_policy,
    statement::For<0, col_policy,
    statement::Lambda<0>
    >
    >
    >;
```

Kernel execution policy  
(typically lives in a  
header file)

```
kernel< KERNEL_POL >( make_tuple(col_range, row_range),
    [=](int col, int row) {
```

```
    double dot = 0.0;
    for (int k = 0; k < N; ++k) {
        dot += A(row, k) * B(k, col);
    }
    C(row, col) = dot;
} );
```

Kernel implementation  
(application source code)



# Each loop level has an execution policy and iteration space

```
for(int row = 0; row < N; ++row) {  
    for(int col = 0; col < N; ++col) {  
  
        // row-column dot product  
  
    }  
}
```



```
using EXEC_POL = KernelPolicy<  
    statement::For<1, row_policy,  
        statement::For<0, col_policy,  
            statement::Lambda<0>  
        >  
    >  
>;  
  
kernel< EXEC_POL >(  
    make_tuple(col_range, row_range),  
    [=] (int col, int row) {  
  
        // row-column dot product  
  
    });
```

Integer parameter in each 'For' statement indicates the iteration space tuple item it applies to.

# Kernel transformations are made by altering the execution policy, not the algorithm source code

```
using EXEC_POL = KernelPolicy<
```

```
    statement::For<1, row_policy,  
    statement::For<0, col_policy,
```

```
    ...
```

```
>;
```

Reordering 'For' statements  
changes the loop nest ordering

Outer row loop (1),  
inner col loop (0)

```
using EXEC_POL = KernelPolicy<
```

```
    statement::For<0, col_policy,  
    statement::For<1, row_policy,
```

```
    ...
```

```
>;
```

Outer col loop (0),  
inner row loop (1)

# Lambda statements invoke lambda expressions (loop bodies)

```
for(int row = 0; row < N; ++row) {  
    for(int col = 0; col < N; ++col) {  
  
        double dot = 0.0;  
        for (int k=0; k < N; ++k) {  
            dot += A(row, k)* B(k, col);  
        }  
        C(row, col) = dot;  
    }  
}
```

```
using EXEC_POL = KernelPolicy<  
    statement::For<1, row_policy,  
    statement::For<0, col_policy,
```

```
        statement::Lambda<0>
```

```
    >  
    >  
    >;
```

```
kernel< EXEC_POL >(  
    make_tuple(col_range, row_range),  
    [=] (int col, int row) {
```

```
        double dot = 0.0;  
        for (int k=0; k < N; ++k) {  
            dot += A(row, k)* B(k, col);  
        }  
        C(row, col) = dot;
```

```
    });
```

# The RAJA kernel API offers numerous options to explore execution alternatives and optimization strategies

- Tiling statements to partition loops into tiles
  - Helps ensure data stays in fast memory while it is used (cache or GPU shared memory)
- Portable kernel local memory (CUDA shared memory or stack memory on a CPU)
  - Improved latency for data access, usually compliments tiling policies
- Loop interchange via execution policy change
  - Simplifies exploring different data access patterns for different platforms
- Loop Fission/Fusion
  - Breaking loops into multiple parts or merging loops
- A variety of execution policies to map loop iterates to GPU blocks & threads in different ways

# RAJA also provides a *launch API* which creates a space for writing portable kernels using RAJA loop methods

- **Launch method**

Sets up a kernel execution space for host or device. **Run-time** selected by ExecPlace value

- **Launch Context**

Control flow within a kernel; e.g., thread synchronization

- **Capture types**

- Launch lambda captured by value [=] to make device copies of captured variables
- Loop lambdas captured by reference [&] to enable referencing within loop hierarchies.

```
using RAJA::expt;
```

```
launch< launch_policy >(ExecPlace,  
Resources(Teams(NTeams), Threads(NThreads)))  
[=] RAJA_HOST_DEVICE (LaunchContext ctx) {
```

```
    loop< row_policy >(ctx, row_range, [&] (int row) {  
        loop< col_policy >(ctx, col_range, [&] (int col) {  
  
            double dot = 0.0;  
            for(int k=0; k < N; ++k) {  
                dot += A(row, k)* B(k, col)  
            }  
  
            C(row, col) = dot;  
        } );  
    } );
```

Kernel execution  
space

Experimental  
Release



# The RAJA launch API differs from kernel by encapsulating the loop hierarchy inside an execution space

```
for(int row=0; row < N; ++row) {  
  for(int col=0; col < N; ++col) {
```

```
    double dot = 0.0;  
    for(int k=0; k < N; ++k) {  
      dot += A(row, k)* B(k, col);  
    }  
    C(row, col) = dot;
```

```
  }  
}
```

```
launch< launch_policy >(host_or_device,  
Resources(Teams(NTeams), Threads(NThreads))) [=]  
RAJA_HOST_DEVICE(LaunchContext ctx) {
```

```
  loop<row_policy>(ctx, row_range, [&](int row) {  
    loop<col_policy>(ctx, col_range, [&](int col) {
```

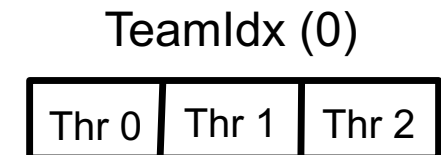
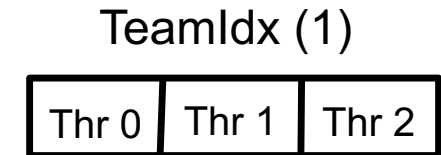
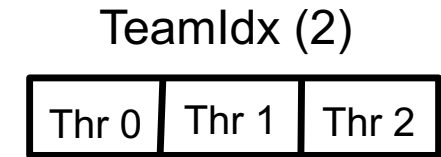
```
      double dot = 0.0;  
      for(int k=0; k < N; ++k) {  
        dot += A(row, k)* B(k, col);  
      }  
      C(row, col) = dot;
```

```
    } );  
  } );
```

```
} );
```

# RAJA launch GPU execution uses a thread team model same as the CUDA/HIP block-thread model

```
launch<launch_policy>(host_or_device,  
                      Resources(Teams(3), Threads(3)))  
[=] RAJA_HOST_DEVICE (LaunchContext ctx) {  
  
    loop<row_policy>(ctx, row_range, [&](int row) {  
        loop<col_policy>(ctx, col_range, [&](int col) {  
  
            // row-column dot product  
        });  
    });  
});
```



RAJA Teams = HIP/CUDA Blocks  
RAJA Threads = HIP/CUDA Threads

Loops can be mapped to CUDA/HIP  
blocks and threads

# Launch and loop methods are templates on both host and device policies for run-time selection of execution back-end

```
using launch_policy =  
LaunchPolicy<host_launch_t,  
             device_launch_t>
```

```
using row_policy =  
LoopPolicy<host_policy,  
           device_policy>;
```

```
using col_policy =  
LoopPolicy<host_policy,  
           device_policy>;
```

- Host backends supported
  - Sequential/SIMD
  - OpenMP
- Device backends supported
  - CUDA
  - HIP

```
launch< launch_policy >( host_or_device,  
                        Resources(Teams(NTeams), Threads(NThreads)))  
  
    [=] RAJA_HOST_DEVICE (LaunchContext ctx) {  
        loop<row_policy>(ctx, row_range, [&](int row){  
            loop<col_policy>(ctx, row_range, [&](int col){  
                // row-column dot product  
  
            } );  
        } );  
    }  
);
```

# RAJA provides policies for common GPU thread striding patterns, such as CUDA block-stride loops

```
int row = blockIdx.x;
```

```
int col = threadIdx.x;
```

```
for(col; col<N; col += blockDim.x) {  
    // row-column dot product  
}
```

Matrix-Matrix multiplication kernel

```
using row_policy =  
    LoopPolicy< loop_exec, cuda_block_x_direct >;
```

```
using col_policy =  
    LoopPolicy< loop_exec, cuda_thread_x_loop >;
```

...

```
loop<row_policy>(ctx, row_range, [&](int row){  
    loop<col_policy>(ctx, col_range, [&](int col){  
        // row-column dot product  
    });  
});
```

• Runtime for N = 1e4 on NVIDIA V100: 3793 milliseconds

• Runtime for N = 1e4 on NVIDIA V100: 2921 milliseconds

# Global thread ID calculations are simplified with RAJA

```
int row =  
blockIdx.y * blockDim.y + threadIdx.y;
```

```
int col =  
blockIdx.x * blockDim.x + threadIdx.x;
```

```
if(row < N && col < N ){  
    // row-column dot product  
}  
}
```

Bounds checks are not needed  
with RAJA since loop methods  
mask out-of-bounds indices.

Matrix-Matrix multiplication kernel with global threads

```
using row_policy =  
    LoopPolicy<loop_exec, cuda_global_thread_y >;  
  
using col_policy =  
    LoopPolicy<loop_exec, cuda_global_thread_x >;  
  
...  
  
loop<row_policy>(ctx, row_range, [&](int row) {  
    loop<col_policy>(ctx, col_range, [&](int col){  
        // row-column dot product  
    } );  
} );
```

- Runtime for N = 1e4 on NVIDIA V100 : 1297 milliseconds
- Runtime for N = 1e4 on NVIDIA V100 : 1313 milliseconds (within 2%)

# The RAJA launch API provides portable support for device shared memory or host stack memory

```
int by = blockIdx.y;
int bx = blockIdx.x;
. . .
```

```
__shared__ double Cs[BLK_SZ][BLK_SZ];
```

```
Cs[threadIdx.y][threadIdx.x] = 0;
```

```
// Load data tiles into shared memory
```

```
for(int k=0; k < (BLK_SZ+N-1)/BLK_SZ; ++k) {
```

```
    // Tiled matrix-multiply with shared memory
    // Cs[r][s] +=
```

```
    __syncthreads();
```

```
}
```

```
// Write out to global memory
```

- Runtime for  $N = 1e4$  on NVIDIA V100 : 980 milliseconds

```
loop<block_y_pol>(ctx, block_y_range, [&](int by) {
    loop<block_x_pol>(ctx, block_x_range, [&](int bx) {
        . . .
```

```
RAJA_TEAM_SHARED double Cs[BLK_SZ][BLK_SZ];
```

```
loop<thread_y_pol>(ctx, ty_range, [&](int ty) {
    loop<thread_x_pol>(ctx, tx_range, [&](int tx) {
        Cs[ty][tx] = 0.0;
    } );
} );
```

```
// Load data tiles into shared memory
```

```
for(int k=0; k < (BLK_SZ+N-1)/BLK_SZ; ++k) {
```

```
    // Tiled matrix-multiply with shared memory
```

```
    // Cs[r][s] +=
```

```
    ctx.teamSync();
```

```
}
```

```
// Write out to global memory
```

```
} );
```

```
}
```

- Runtime for  $N = 1e4$  on NVIDIA V100 : 1026 milliseconds (within 5%)

**More than a basic loop abstraction layer, RAJA provides other mechanisms to improve application performance**






# RAJA asynchronous execution integrates with CHAI and Umpire

```
chai::ManagedArray<double> a1(N);  
chai::ManagedArray<double> a2(N);
```

```
RAJA::resource::Cuda cuda1;  
RAJA::resource::Cuda cuda2;
```

Resources assigned to different  
CUDA streams passed to RAJA  
execution methods




```
auto event1 = forall<cuda_exec_async>(&cuda1, RangeSegment(0, N),  
                                     [=] RAJA_DEVICE (int i) { a1[i] = ... } );
```

```
auto event2 = forall<cuda_exec_async>(&cuda2, RangeSegment(0, N),  
                                     [=] RAJA_DEVICE (int i) { a2[i] = ... } );
```

```
cuda1.wait_on(&event2);
```

Query or wait on events to  
control synchronization

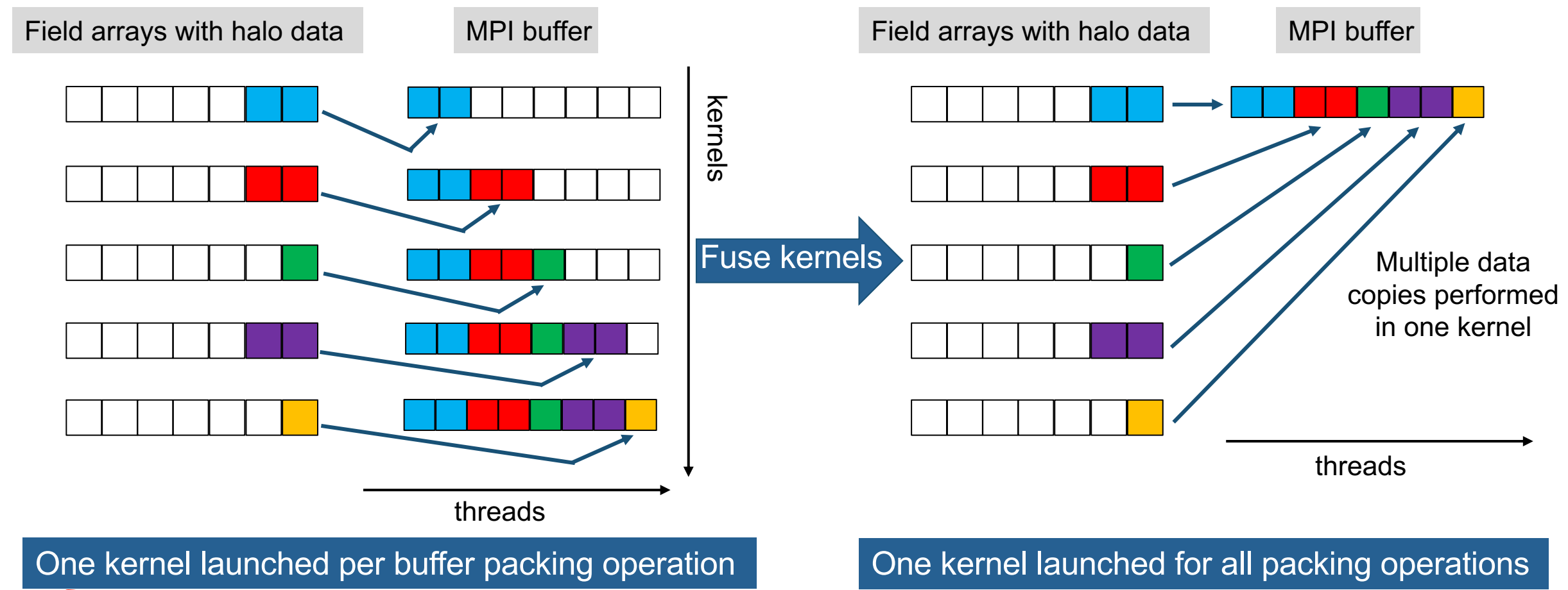


```
forall<cuda_exec_async>(&cuda1, RangeSegment(0, N),  
                       [=] RAJA_DEVICE (int i) { a1[i] *= a2[i]; } );
```

```
forall<seq_exec>(RangeSegment(0, N),  
               [=] (int i) { printf("a1[%d] = %f \n", i, a1[i]); } );
```

# Fusing small GPU kernels into one kernel launch helps alleviate negative impact of launch overhead

Packing/unpacking halo (ghost) data on a GPU into MPI buffers is a key application use case



# The RAJA API for fusing kernels into one launch is simple to use

## Typical pattern launching many packing kernels

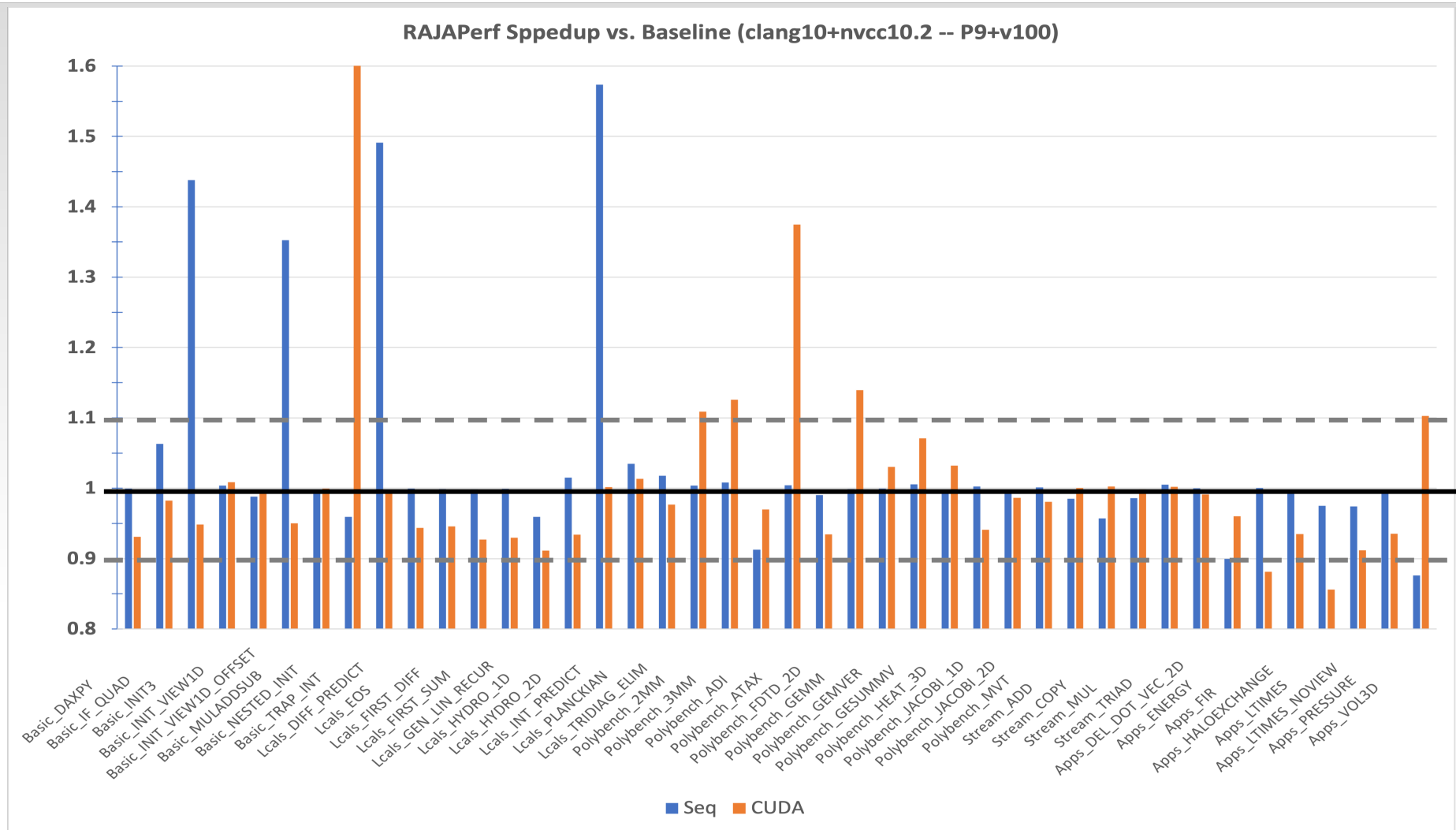
```
for ( neighbor : neighbors ) {
    double* buf = buffers[neighbor];
    for ( f : fields[neighbor] ) {
        int len = f.ghostLen();
        double* ghost_data = f.ghostData();
        forall(Range(0, len), [=](int i){
            buf[ i ] = ghost_data[ i ];
        });
        buf += len;
    }
    send(neighbor);
}
```

This technique is used in production apps at LLNL and yields 5-15% overall runtime reduction in typical problems.

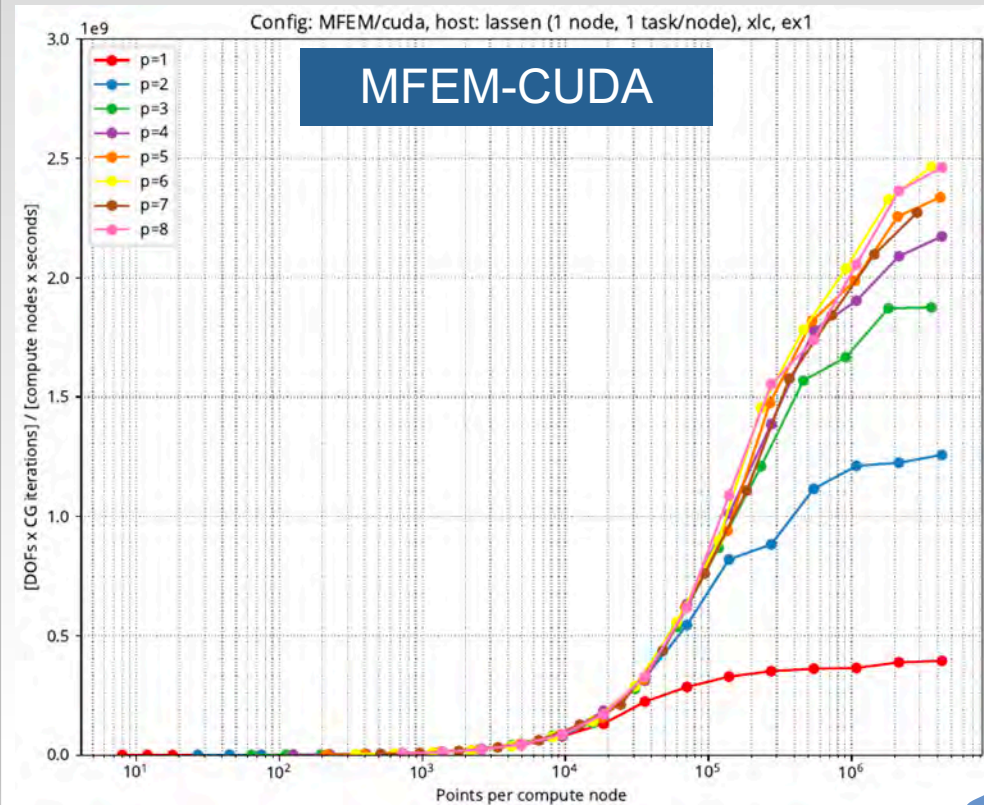
## Fusing the kernels into one GPU launch

```
RAJA::WorkPool< ... > fuser;
for ( neighbor : neighbors ) {
    double* buf = buffers[neighbor];
    for ( f : fields[neighbor] ) {
        int len = f.ghostLen();
        double* ghost_data = f.ghostData();
        fuser.enqueue(Range(0, len), [=](int i){
            buf[ i ] = ghost_data[ i ];
        });
        buf += len;
    }
}
auto workgroup = fuser.instantiate();
workgroup.run();
for ( neighbor : neighbors ) {
    send(neighbor);
}
```

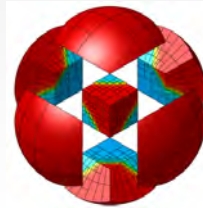
# The RAJA Performance Suite is a useful co-design tool to assess compiler performance and to collaborate with vendors



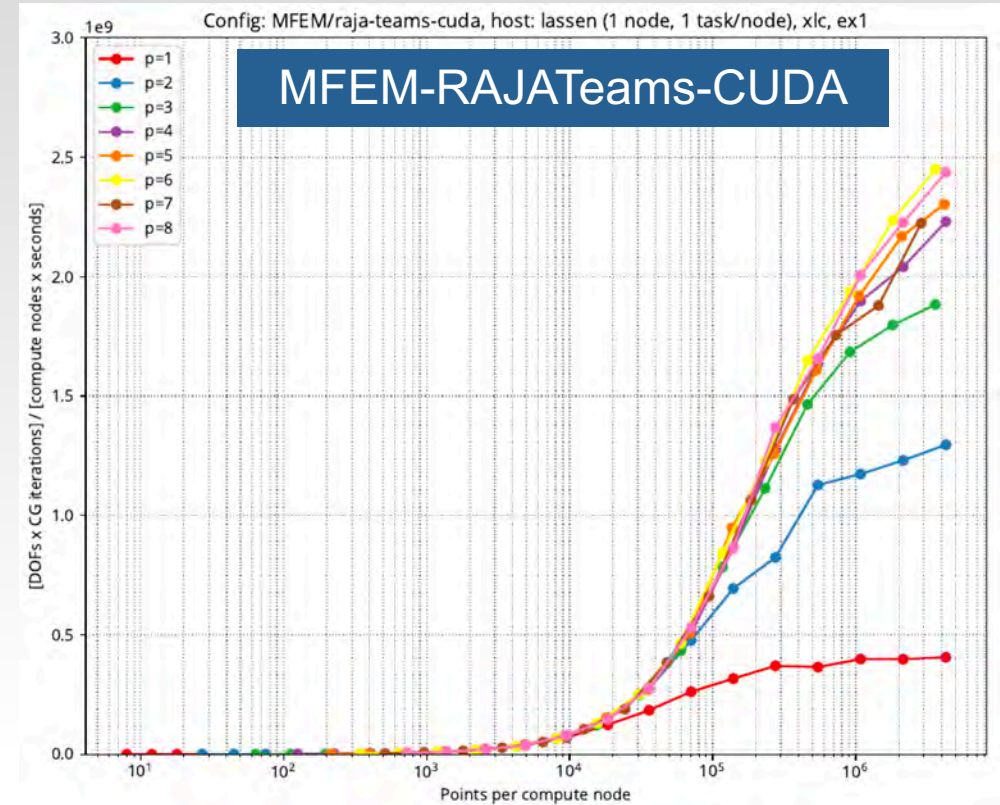
# RAJA “Teams” (described earlier) was co-developed with the LLNL ATDM application (MARBL) team



RAJA performance  
on par with MFEM  
native CUDA  
implementation



**CEED**  
EXASCALE DISCRETIZATIONS

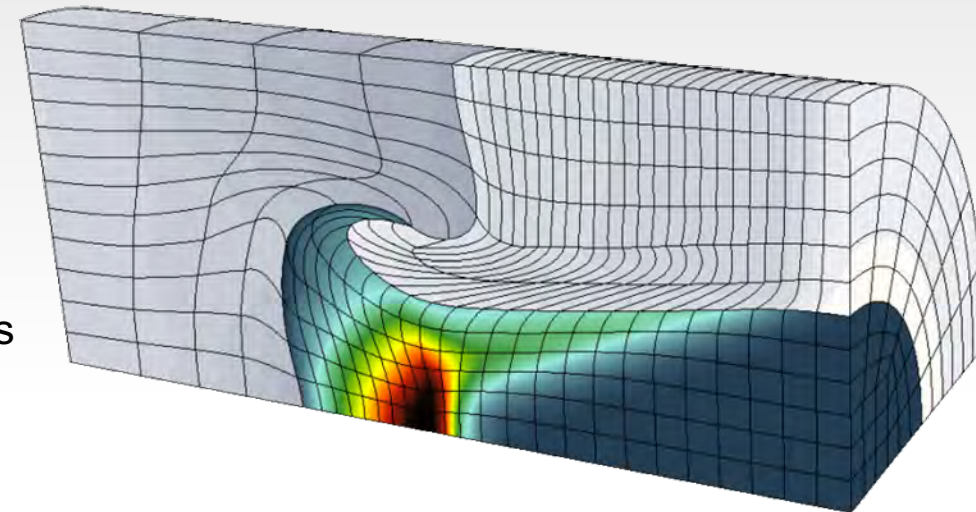


Hierarchical parallelism & shared memory are key performance enablers.



# ECP applications are showing impressive performance on pre-exascale platforms

- **LLNL ATDM application** (high-order ALE hydro simulations) : uses RAJA, Umpire
  - Node-to-node speedup:
    - **15x** : Sierra (2 P9 + 4 V100) vs. CTS-1 Intel Cascade Lake (48 core CPUs)
    - **30x** : Sierra vs. Astra (Cavium ThunderX2 28 core CPUs)
  - Programmatically-relevant simulations scaled to 50% of Sierra (2048 nodes) and 100% of Astra (SNL) (2048 nodes)
    - Documented in ATDM Tri-lab Level 1 milestone report (Dec. 2020)
  - Relies heavily on MFEM library (CEED co-design)
    - Provides RAJA and Umpire execution and memory back-end options
  - RAJA “Teams” capability (discussed earlier) resulted from collaboration between RAJA and LLNL ATDM application team



# ECP applications are showing impressive performance on pre-exascale platforms

- **SW4 application** (high-resolution earthquake simulations) : uses RAJA, Umpire

- Node-to-node speedup:

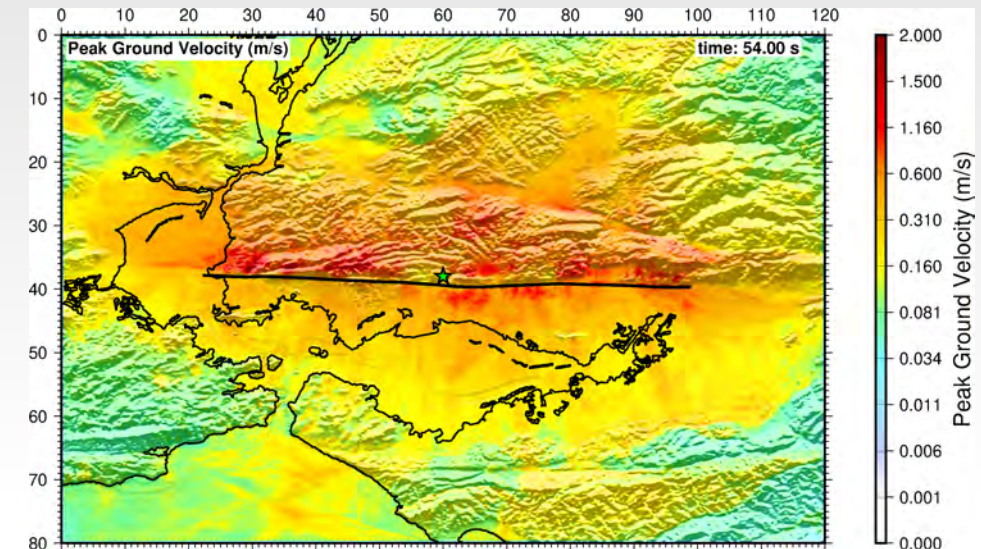
- **16x** : Sierra vs. CTS-1 Intel Cascade Lake

- **32x** : Sierra vs. CTS-1 Intel Broadwell

- Recent paper in *Bulletin of Seismological Society of America* presents highest resolution earthquake simulation studies to date enabled by SW4-RAJA application

- Partial application running on AMD MI-60 GPUs – additional support in HIP compiler needed

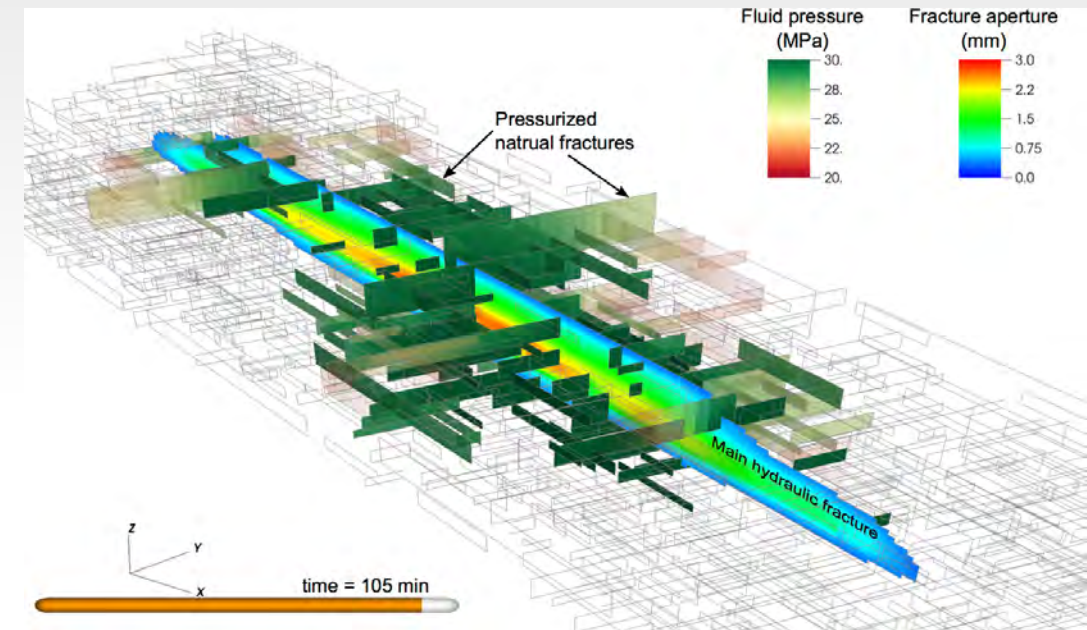
- SW4-lite proxy app running on Intel GPUs (GEN9) – additional support in DPC++ compiler needed for full SW4





# ECP applications are showing impressive performance on pre-exascale platforms

- **GEOSX application** (subsurface solid mechanics simulations) : uses RAJA, Umpire, CHAI
  - Node-to-node speedup
    - **14x** : Lassen (Sierra arch) vs. CTS-1 Intel Cascade Lake
  - Initial studies show good weak scaling up to 64 nodes on Lassen (256 V100s)
  - Team is working on scaling to 1000s of GPUs on Summit



# ECP applications are showing impressive performance on pre-exascale platforms

- **ExaSGD application** (power grid optimization): uses RAJA, Umpire
  - Adopted RAJA & Umpire ~8 months ago
  - Key kernels using RAJA are running at **near peak memory BW** on Summit with little system-specific tuning
  - Parts of code running on Tulip (Frontier EA system) with good performance



## **Our application porting perspectives are based on production experiences and constraints:**

- **Large integrated code bases**
- **Codes must run well on a diversity of platforms always**
- **Codes live for decades so must be viable across multiple platform generations**
- **Under continual development, while continuously in production use**



# Experience shows that the RAJA Portability Suite enables a diverse set of portable, high performance applications

- **Insulates applications from technology disruption**
  - Does not inhibit using new or platform-specific tools
- **Insulates apps from variability** in programming models and architectures
- **Facilitates application flexibility** by promoting clean encapsulation
- ***RAJA-app codesign has led to desirable outcomes***
  - **Easy to leverage** features and/or optimizations developed for another application
  - **Easy to grasp** for all application developers
  - **Easy to integrate** with existing applications
  - **Easy to adopt incrementally**





# Porting an application isn't free – it requires a good plan

- Develop a plan that is **agreeable to all developers** on the team – implementation and ownership
  - **Meaning of manageable portability** (tolerance for disruption) depends on size and complexity of application
  - A **memory management strategy** is as important as a strategy to manage execution
  - Plan for iterative, incremental development → modify code, evaluate performance & cost of change, etc. → repeat...
- Assess algorithm structures and data access patterns
  - Think about commonality across algorithms and loop patterns – focus on individual kernels only when necessary
  - Keep **code and data access simple** in kernels – C++ STL containers are not amenable to GPUs
  - Look for opportunities for changes that will yield benefits, and which are **manageable and maintainable**
- Strive to **maintain a familiar look and feel of the code**
  - Consider a **code-specific wrapper layer** (using templates, macros, etc.)
    - How much of an abstraction layer, such as RAJA, do you want to expose in the application code?
    - How do you sustain SME developer productivity and enable platform-specific optimizations?
    - Add instrumentation for performance analysis
  - Convert kernels to use new parallel patterns (e.g., scans) **only when needed for desirable performance**

# Establishing performance expectations is critical

- Performance expectations should be based on **analysis before you start porting**
  - What are performance limitations in current code? – memory B/W? compute bound? ...
  - For example, if application is B/W bound, set expectations by comparing effective node B/W between architectures
- **First port code, then analyze performance, then optimize as needed**
- **Continuously monitor performance** while making code changes
  - Best done as **part of CI process** to track on a per commit basis
  - Performance should not degrade on current platforms and should improve over time on new systems
  - **Keep data resident on devices** (GPUs) – avoid host-device transfers as much as possible
- **Focus optimization effort** on performance critical code sections
  - Expose as much fine-grained parallelism as is reasonable – how much code disruption can the team tolerate?
  - Start with 1 MPI rank per GPU – explore more complex approaches later if there is potential benefit
- **Don't get frustrated when initial results are not what's expected. It's an iterative process!**
  - Production ASC apps at LLNL have been working at porting to GPUs for 5+ years – much progress has been made, but much work remains....

# Typically, each optimization step improves performance and reveals the next problem to solve

## GPUs have performance overheads not seen on CPUs

- **Kernel launch overhead:** try to hide with asynchronous kernel launches
- **Data transfer between memory spaces:** avoid or overlap with other work
- **Memory allocation on GPUs is much more expensive than CPUs:** memory pools are a must!

## Optimization requires coordinating many parts

- **Libraries have different porting strategies/timelines:** Un-ported parts lead to costly CPU/GPU data transfers
- **GPU memory is a scarce shared resource:** sharing memory pools can help



# The RAJA Portability Suite is on track to be ready for the next generation of platforms, including exascale

Machine	RAJA	CHAI	Umpire
Perlmutter	CUDA support actively used in production on Sierra We continue to investigate and improve performance		
Frontier & El Capitan	HIP support available in RAJA v0.11.0 (1/2020) Developed with AMD	Avail. CHAI v1.2.0 (8/2019) Developed w/ AMD	Avail. Umpire v1.0.0 (8/2019) Developed w/ AMD
Aurora	SYCL back-end development is a collaboration with ANL supported by ECP Currently, filling feature gaps and improving performance w/ Intel		Avail. Umpire v4.0.0 (9/2020) Developed w/ ANL

- Our open-source efforts have seen significant contributions from code teams, vendors, and other external collaborators
  - 38+ RAJA contributors, core project team has 8 people
    - Up from 20 contributors last year
  - Leveraging vendor interactions to support new hardware (IBM, NVIDIA, AMD, Intel, Cray)
- Tutorials at ECP meetings, ATPESC, and academic conferences

Project	Unique Monthly Visitors
RAJA	234
CHAI	48
Umpire	102

# Why use portability solutions like RAJA, Umpire, CHAI, etc?

- It depends. **What does “performance portability” means for your project?**
  - If you can afford to develop and maintain platform-specific code, you may prefer that option and may not need a portability abstraction
  - If your application is large or if programming model, hardware architecture, and optimization expertise is sparse on your team, a portability solution can provide your team with a variety of benefits
- Portability solutions enable you to write ***single-source code*** that runs on a diversity of platforms
  - You may still need to write some platform-specific code to better optimize some kernels
  - Fortunately, a general abstraction approach may be good enough for most of your code
- Benefits of a portability solution include the following:
  - (Most of) the cost of developing and maintaining platform-specific code is eliminated
  - It's straightforward to get running on new hardware architectures
  - It's easier to separate software development concerns on your project – optimization work can be done by experts under the abstraction layer, while application code looks familiar to SME application developers
  - You will leverage the expertise and effort of others who contribute to the portability library (features and optimizations); improving your code performance can be as simple as using an new version of a library

# User documentation, tutorials, and other code repos associated with the RAJA Portability Suite are available

- **RAJA User Guide:** getting started info, details about features & usage, tutorial materials ([readthedocs.org/projects/raja](https://readthedocs.org/projects/raja))
- **RAJA Project Template:** shows how to use RAJA in an app using CMake or make ([github.com/LLNL/RAJA-project-template](https://github.com/LLNL/RAJA-project-template))
- **RAJA Performance Suite:** collection of kernels to assess compilers & RAJA performance. Used by us, vendors, for DOE platform procurements, etc. ([github.com/LLNL/RAJAPerf](https://github.com/LLNL/RAJAPerf))
- **RAJA Proxy Apps:** proxy apps using RAJA, CHAI, Umpire ([github.com/LLNL/RAJAProxies](https://github.com/LLNL/RAJAProxies))
- **Umpire User Guide:** getting started info, details about features & usage, tutorial materials ([readthedocs.org/projects/umpire](https://readthedocs.org/projects/umpire))
- **Umpire Interactive Tutorial:** interactive user tutorial using Jupyter notebooks ([github.com/LLNL/umpire-interactive-tutorial](https://github.com/LLNL/umpire-interactive-tutorial))
- **CARE:** Collection of **CHAI And RAJA Externsions** that are useful to application developers to help write portable code ([github.com/LLNL/CARE](https://github.com/LLNL/CARE))

The RAJA Performance Suite and Proxy Apps are good sources of examples for RAJA usage.



## 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.

