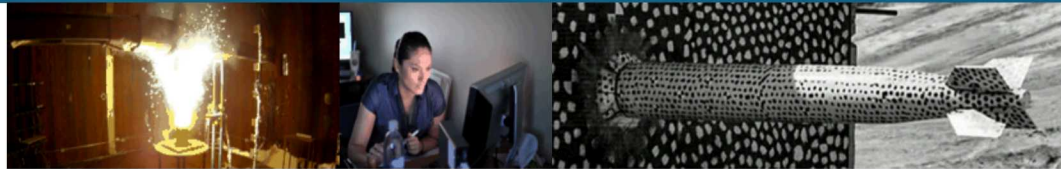


The Kokkos Project



PRESENTED BY

Dan Ibanez



Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

The Kokkos project based at **Sandia** includes collaborators at **Oak Ridge, Los Alamos**, and **Argonne** working to advance portable HPC C++ programming.

The Kokkos project members provide the following products and services:

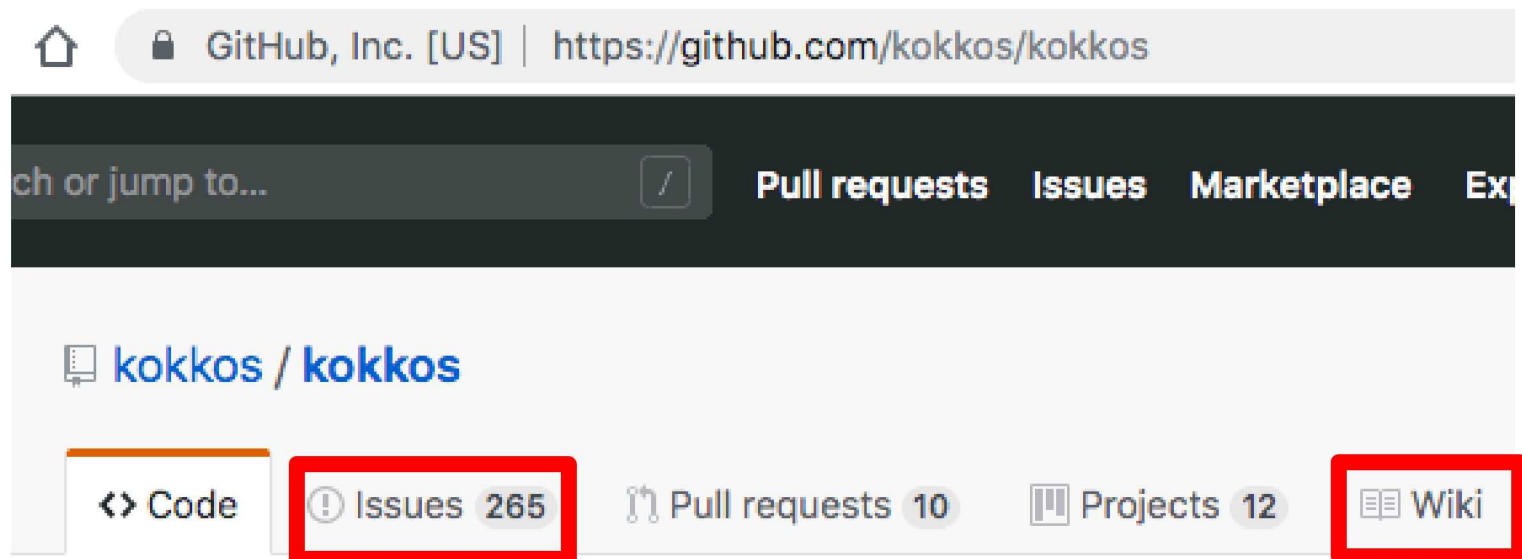
1. The Kokkos ecosystem including the Kokkos library, KokkosKernels, and profiling tools
2. Dedicated support for Kokkos ecosystem users
3. Outreach, training, and tutorial events for DOE software developers to learn portable C++ HPC programming
4. Participate in ISO C++ standardization to ensure C++ continues to serve the interests of the HPC community

3 Goal of the Kokkos library

1. Single-source: There is no architecture-specific syntax or architecture-specific files in user code
2. Portable: The same source code compiles and executes correctly on all platforms
3. Performance: That source code performs close to what can be hand-coded using architecture-specific tools

DOE-relevant “backends”:

1. OpenMP threading: handles multi-core CPUs
2. CUDA: handles NVIDIA GPUs
3. OpenMP target offload: hopes to handle all GPUs, but not vendor favorite (each vendor prefers their vendor-specific thing)
4. HIP: AMD now tells us this is the way to go for their GPUs (e.g. Frontier)
5. ROCm: AMD’s initial GPU backend, which they later dropped
6. We are working on Aurora21 & on track to supporting it

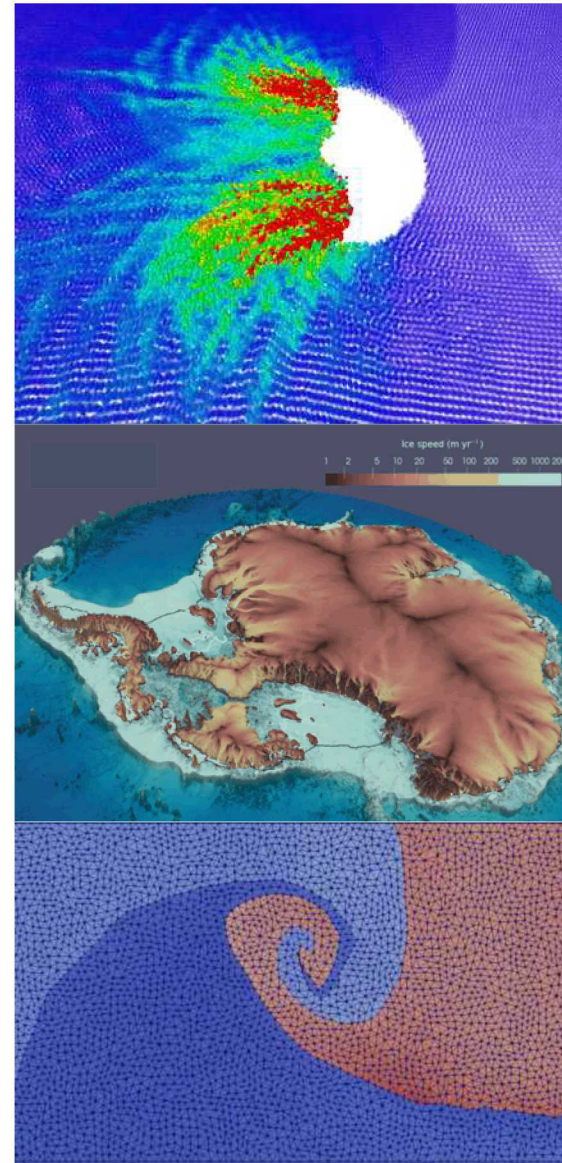


Ask questions,
report possible bugs,
request features

Documentation:
Programming Guide,
API Reference

Some Relevant Sandia Kokkos Simulation Codes

1. **LAMMPS**: Broad range of molecular dynamics capabilities, significant literature impact
2. **MALI/ProSPect**: Simulating arctic land ice erosion to improve our understanding of climate change
3. **LGR**: Demonstrating unstructured mesh adaption on GPUs with applications to fluid-structure interaction and shocks





Portable HPC C++ in 4 steps



7 Step 1: Data-Oriented Design

Possibly the most difficult step in the process! For example, you may need to design your code architecture from the beginning such that **data** is stored in contiguous **arrays** and **algorithms** are expressed as **for-loops**. It may require a serious refactor, but don't be discouraged! Complex data structures and algorithms including unstructured mesh adaptation can fit this **Data-Oriented Design**.

```
std::vector<double> a(100 * 1000);  
for (std::size_t i = 0; i < a.size(); ++i) {  
    a[i] = 1.0;  
}  
  
double sum = 0.0;  
for (std::size_t i = 0; i < a.size(); ++i) {  
    sum += a[i];  
}
```

8 Step 2: Separate loop bodies

C++11 introduces the **lambda**, which allows the compiler to create an object that contains the **variables used** and code executed in a **loop body**.

```
std::vector<double> a(100 * 1000);
auto body1 = [&](std::size_t i) {
    a[i] = 1.0;
};
for (std::size_t i = 0; i < a.size(); ++i) {
    body1(i);
}

double sum = 0.0;
auto body2 = [&](std::size_t i) {
    return a[i];
};
for (std::size_t i = 0; i < a.size(); ++i) {
    sum += body2(i);
}
```


9 Step 3: Apply Kokkos

C++ for-loops expressing certain parallel patterns may now be replaced with the appropriate Kokkos parallel pattern, and data structures may also be replaced.

```
Kokkos::View<double*> a("name", 100 * 1000);
auto body1 = KOKKOS_LAMBDA(std::size_t i) {
    a[i] = 1.0;
};
Kokkos::parallel_for(Kokkos::RangePolicy(0, a.size()), body1);

double sum = 0.0;
auto body2 = KOKKOS_LAMBDA(std::size_t i) {
    return a[i];
};
Kokkos::parallel_reduce(Kokkos::RangePolicy(0, a.size()), body2, sum);
```

Step 4: (Wait a few years... then) Use standard C++

The Kokkos project is working to ensure that standard C++ eventually contains the necessary tools to do portable HPC programming, including 4 team members who are active on the committees of C++ and OpenMP. Here is what our example code might look like in the future:

```
std::vector<double, std::gpu_allocator> a(100 * 1000);  
std::fill(std::execution::gpu, a.begin(), a.end(), 1.0);  
auto sum = std::reduce(std::execution::gpu, a.begin(), a.end());
```