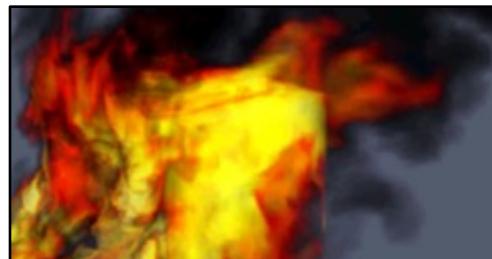




Exceptional service in the national interest



$$\partial a \sim J_{a,\sigma^2}(\xi_1) = \frac{(\xi_1 - a)}{\sigma^2} f_{a,\sigma^2}(\xi_1);$$
$$\int T(x) \cdot \frac{\partial}{\partial \theta} f(x, \theta) dx = M \left(T(\xi) \cdot \frac{\partial}{\partial \theta} \ln L \right)$$



Kokkos and the Future of C++: Parallel Programming for Heterogeneous Architectures

Christian R. Trott, - Center for Computing Research

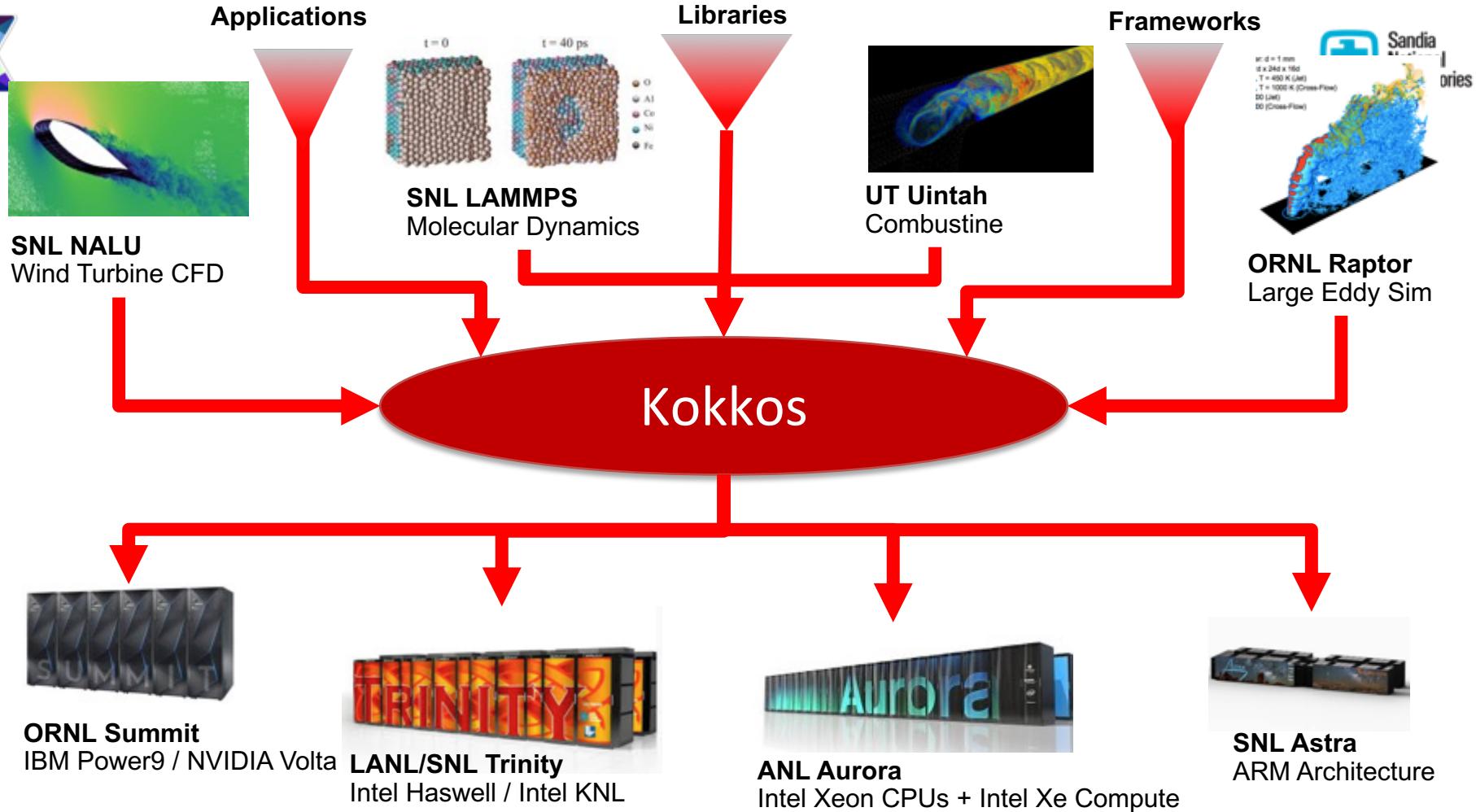
Sandia National Laboratories/NM

Unclassified Unlimited Release

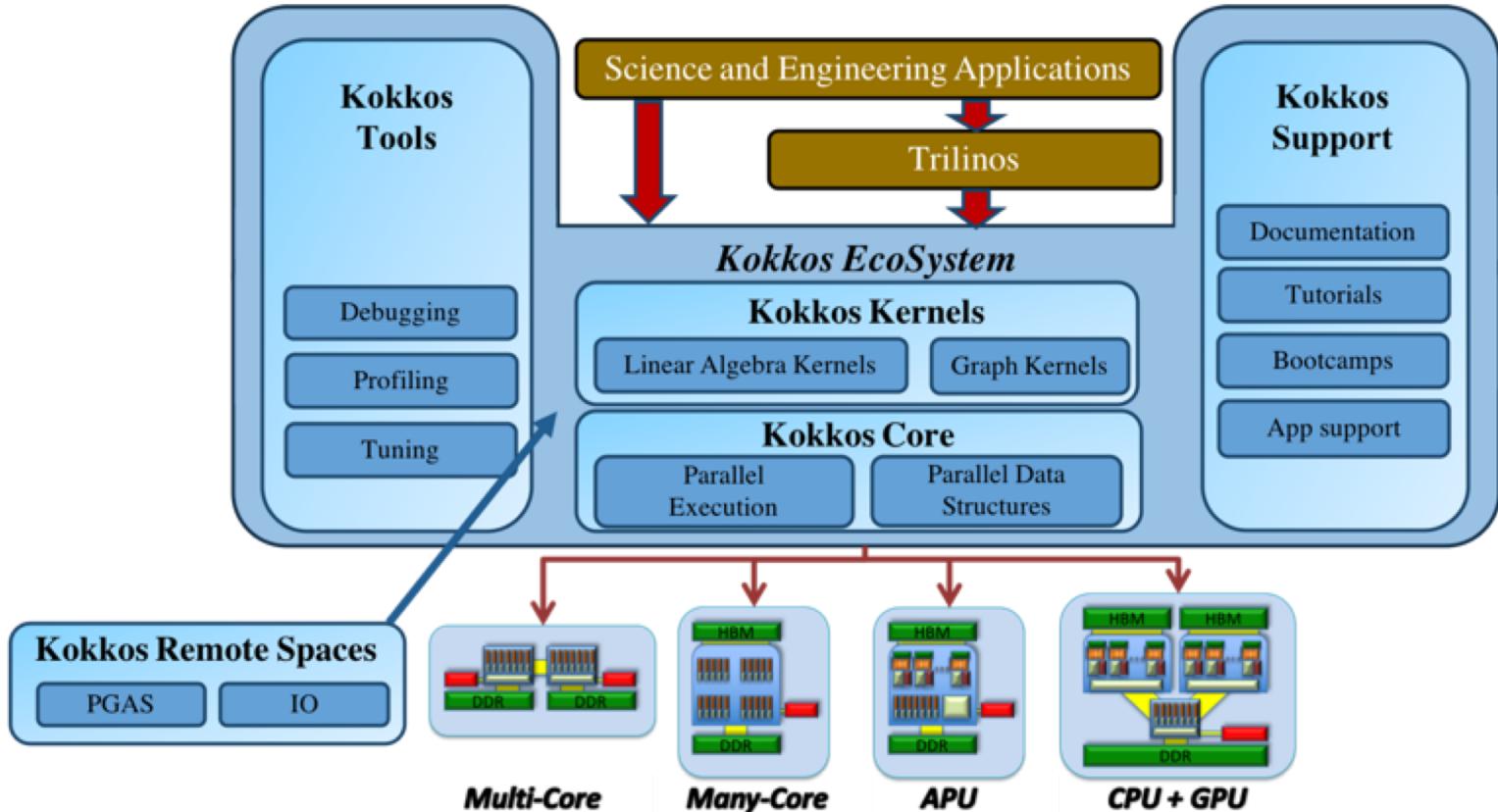
*D. Sunderland, N. Ellingwood, D. Ibanez, J. Miles,
D. Hollman, V. Dang*



Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and 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-NA-0003525.



Kokkos EcoSystem





Kokkos Development Team



- Dedicated team with a number of staff working most of their time on Kokkos
 - Main development team at Sandia in CCR

Kokkos Core:

*C.R. Trott, D. Sunderland, N. Ellingwood, D. Ibanez, J. Miles, D. Hollman, V. Dang, Mikael Simberg, H. Finkel, N. Liber, D. Lebrun-Grandie, B. Turcksin
former: H.C. Edwards, D. Labreche, G. Mackey, S. Bova*

Kokkos Kernels:

S. Rajamanickam, N. Ellingwood, K. Kim, C.R. Trott, V. Dang, L. Berger, J. Wilke, W. McLendon

Kokkos Tools:

S. Hammond, C.R. Trott, D. Ibanez, S. Moore; soon: D. Poliakoff

Kokkos Support:

*C.R. Trott, G. Shipman, G. Lopez, G. Womeldorf,
former: H.C. Edwards, D. Labreche, Fernanda Foertter*



Kokkos Core Capabilities



Concept	Example
Parallel Loops	<code>parallel_for(N, KOKKOS_LAMBDA (int i) { ...BODY... });</code>
Parallel Reduction	<code>parallel_reduce(RangePolicy<ExecSpace>(0,N), KOKKOS_LAMBDA (int i, double& upd) { ...BODY... upd += ... }, Sum<>(result));</code>
Tightly Nested Loops	<code>parallel_for(MDRangePolicy<Rank<3> > ({0,0,0},{N1,N2,N3},{T1,T2,T3}, KOKKOS_LAMBDA (int i, int j, int k) {...BODY...});</code>
Non-Tightly Nested Loops	<code>parallel_for(TeamPolicy<Schedule<Dynamic>>(N, TS), KOKKOS_LAMBDA (Team team) { ... COMMON CODE 1 ... parallel_for(TeamThreadRange(team, M(N)), [&] (int j) { ... INNER BODY... }); ... COMMON CODE 2 ... });</code>
Task Dag	<code>task_spawn(TaskTeam(scheduler , priority), KOKKOS_LAMBDA (Team team) { ... BODY });</code>
Data Allocation	<code>View<double**, Layout, MemSpace> a("A",N,M);</code>
Data Transfer	<code>deep_copy(a,b);</code>
Atomics	<code>atomic_add(&a[i],5.0); View<double*,MemoryTraits<AtomicAccess>> a(); a(i)+=5.0;</code>
Exec Spaces	Serial, Threads, OpenMP, Cuda, HPX (experimental), ROCm (experimental)



DOE Machine Announcements



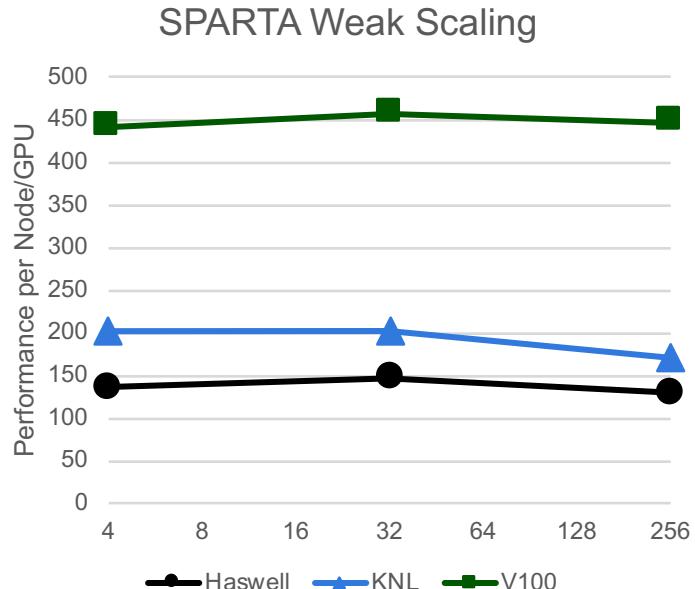
- Now publicly announced that DOE is buying both AMD and Intel GPUs
 - Argonne: Cray with Intel Xeon + Intel Xe Compute
 - ORNL: Cray with AMD CPUs + AMD GPUs
 - NERSC: Cray with AMD CPUs + NVIDIA GPUs
- Have been planning for this eventuality:
 - Kokkos ECP project extended and refocused to include developers at Argonne and Oak Ridge, staffing is in place
 - HIP backend for AMD main development at ORNL
 - The current ROCm backend is based on a compiler which is now deprecated ...
 - Something else for Intel ;-) main development at ANL
 - OpenMPTarget for AMD, Intel and NVIDIA, lead at Sandia



Sparta: Production Simulation at Scale



- Stochastic PArallel Rarefied-gas Time-accurate Analyzer
- A direct simulation Monte Carlo code
- Developers: *Steve Plimpton, Stan Moore, Michael Gallis*
- Only code to have run on all of Trinity
 - 3 Trillion particle simulation using both HSW and KNL partition in a single MPI run
- Benchmarked on 16k GPUs on Sierra
 - Production runs now at 5k GPUs
- Co-Designed Kokkos::ScatterView





Latency Limited Kernels and Asynchronous Execution

- Many applications run into latency limits
 - Targeting 1000 timesteps or solver iterations per second
 - Need to optimize for kernels of 20us and less runtime
 - MiniEM: >3000 Kernel calls per solve => 30k/s to achieve 10 solves/s
- Underlying Programming Models have limits
 - CUDA launch latency 3us (Skylake) to 8us (Power9)
 - Kokkos has additional overhead
 - OpenMP max loop rate about 1us/per loop
- Allocation rate limited
 - CUDA UVM allocation takes up to 200us!

Approaches to Address This

- More asynchronous execution to hide launch latency
 - No API change, improve implementation (i.e. limit fences etc.)
 - May need hints from user to use latency instead of throughput opt path
- Fine Grained Tasking Interface
 - Potentially write big kernels with inner dependencies via tasking
- Execution Space Instances
 - First step support CUDA streams
- Fuse Kernels
 - Real fusion is user level, but maybe help with interfaces
- Kernel Graph Abstraction
 - Exploit CUDA graphs for now
- Coarse Grained Tasking



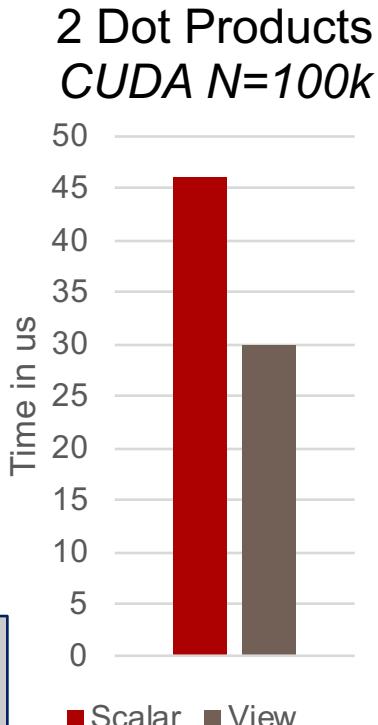
Asynchronicity Semantics



- ParallelReduce/Scan

```
double result;
// parallel_for is always Synchronous
parallel_for("AsynchronousFor",N,F);
// parallel_reduce with Scalar as result is Synchronous
parallel_reduce("SynchronousSum",N,Fr,result);
// parallel_reduce with Reducer constructed from scalar is synchronous
parallel_reduce("SynchronousMax",N,Fr,Max<double>(result));
// parallel_reduce with any type of View as result is asynchronous
Kokkos::View<double,CudaHostPinnedSpace> result_v("R");
parallel_reduce("AsynchronousSum",N,Fr,result_v);
// Even with unmanaged view, and wrapped into Reducer
Kokkos::View<double,HostSpace> result_hv(&result);
parallel_reduce("AsynchronousMax",N,Fr,Max<double>(result_hv));
// Scans without total result argument are asynchronous
parallel_scan("AsynchronousScan",N,Fs);
```

Rule of Thumb: Everything is asynchronous unless reducing into a scalar value!





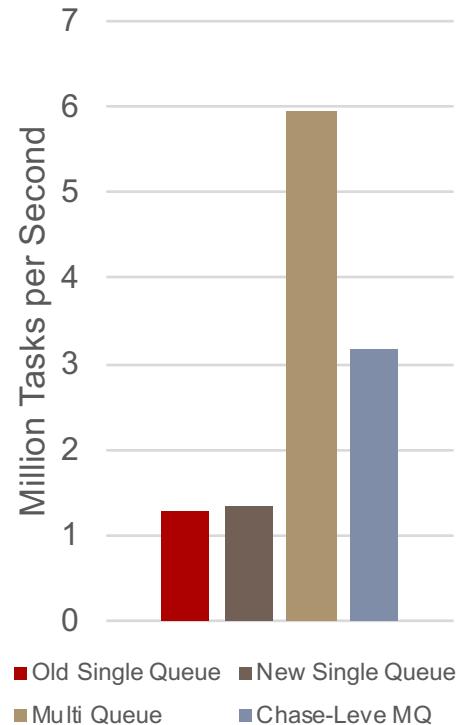
Improved Fine Grained Tasking



- Generalization of TaskScheduler abstraction to allow user to be generic with respect to scheduling strategy and queue
- Implementation of new queues and scheduling strategies:
 - Single shared LIFO Queue (this was the old implementation)
 - Multiple shared LIFO Queues with LIFO work stealing
 - Chase-Lev minimal contention LIFO with tail (FIFO) stealing
 - Potentially more
- Reorganization of Task, Future, TaskQueue data structures to accommodate flexible requirements from the TaskScheduler
 - For instance, some scheduling strategies require additional storage in the Task

Questions: David Hollman

Fibonacci 30 (V100)





CUDA Stream Interop



- Initial step to full coarse grained tasking
 - Discuss in more detail in future directions
- For now: make Kokkos dispatch use user CUDA streams
 - Allows for overlapping kernels: best for large work per iteration, low count

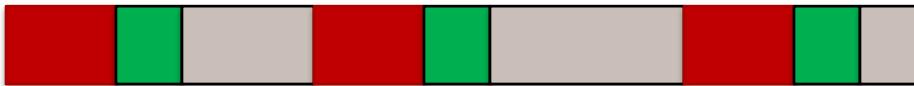
```
// Create two Cuda instances from streams
cudaStream_t stream1,stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
Kokkos::Cuda cuda1(stream1), cuda2(stream2);

// Run two kernels which can overlap
parallel_for("F1",RangePolicy<Kokkos::Cuda>(cuda1,N),F1);
parallel_for("F2",RangePolicy<Kokkos::Cuda>(cuda2,N),F2);
fence();
```



CUDA Graphs

Launch 3 Kernels



CUDA graphs: launch multiple kernels as one



Host Launch 3-10us



Device Grid Setup 1us



Compute Kernel

- CUDA has interface to record Kernel launches, and then dispatch in bulk
- Can resolve dependencies according to streams

```
// Start by initiating stream capture
cudaStreamBeginCapture(stream1);
// Build stream work as usual A<<< ..., stream1 >>>();
cudaEventRecord(e1, stream1); B<<< ..., stream1 >>>();
cudaStreamWaitEvent(stream2, e1); C<<< ..., stream2 >>>();
cudaEventRecord(e2, stream2);
cudaStreamWaitEvent(stream1, e2); D<<< ..., stream1 >>>();
// Now convert the stream to a graph
cudaStreamEndCapture(stream1, &graph);
```

```
cudaGraphInstantiate(&instance, graph);
// Launch executable graph 100 times
for(int i=0; i<100; i++)
    cudaGraphLaunch(instance, stream);
```



Kokkos Options To Leverage Graphs

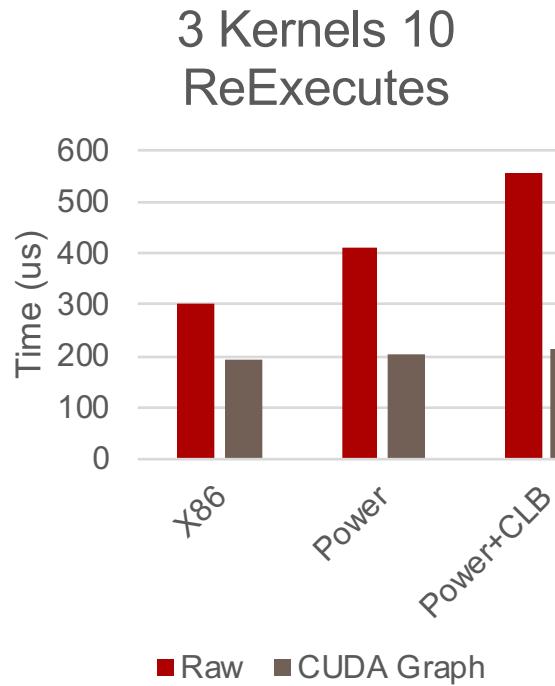


- InterOp option: make the CUDA API capture Kokkos parallel_for etc. correct
- Capture in a coarse grained scope:

```
Kokkos::View<double> reduce_result("red");
auto graph = Kokkos::capture_kernel_graph( [=] () {
    Kokkos::parallel_for("A",N,KOKKOS_LAMBDA(const int i) {...});
    Kokkos::parallel_reduce("A",N,
        KOKKOS_LAMBDA(const int i, double& r) {...},reduce_result);
    Kokkos::parallel_for("A",N,KOKKOS_LAMBDA(const int i) {
        double r = reduce_result();
        ...
    });
});

for(int i=0;i<10;i++) {
    Kokkos::execute_graph(graph);
    graph.fence();
}
```

- Problem: what if I want an MPI call in this loop?

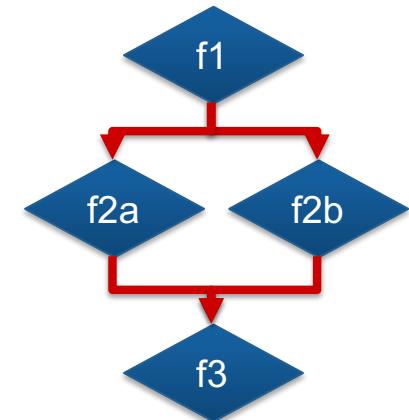


Coarse Grained Tasking

- Somewhat awkward to capture the whole region
- Expressing dependencies indirectly just via ExecSpace instances is suboptimal
 - Make parallel dispatch return “futures” and execution policies consume dependencies instead

```
auto fut_1 = parallel_for( RangePolicy<>("Funct1", 0, N), f1 );
auto fut_2a = parallel_for( RangePolicy<>("Funct2a", fut_1, 0, N), f2a);
auto fut_2b = parallel_for( RangePolicy<>("Funct2b", fut_1, 0, N), f2b);
auto fut_3 = parallel_for( RangePolicy<>("Funct3", all(fut_2a,fut2_b), 0, N), f3);
fence(fut_3);
```

- Could build graph under the hood and submit upon fence?
 - What about eager execution?
 - Insert MPI via host_spawn?

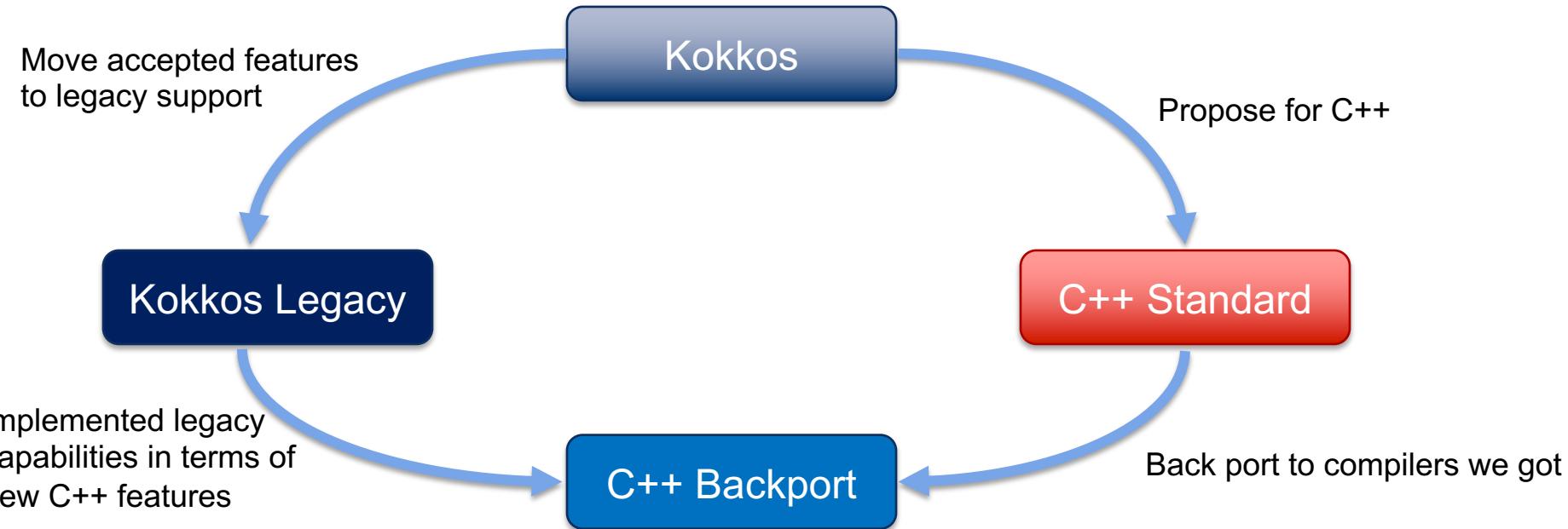




Aligning Kokkos with the C++ Standard



- Long term goal: move capabilities from Kokkos into the ISO standard
 - Concentrate on facilities we really need to optimize with compiler





C++ Atomic Ref



- **atomic_ref<T>** in C++20
 - Provides atomics with all capabilities of atomics in Kokkos
 - Atomic ops on “POD” types with operators
 - Wrap non-atomic object
 - **atomic_ref(a[i])+=5.0;** instead of **atomic_add(&a[i],5.0);**



C++ MDspan



- Provides customization points which allow all things we can do with **Kokkos::View**
- Better design of internals though! => Easier to write custom layouts. ☺
- Also: arbitrary rank (until compiler crashes) and mixed compile/runtime ranks ☺
- More verbose interface though 😞
- We hope will land early in the cycle for C++23 (i.e. early in 2020)
- 4 Template Parameters
 - Scalar Type
 - Extents -> rank and compile dimensions
 - Layout
 - Accessor -> return type of operator, storage handle, and access function

```
View<int**[5], LayoutLeft, MemoryTraits<Atomic>>
=
basic_mdspan<int, extents<dynamic_extent, dynamic_extent, 5>, layout_left, accessor_atomic<int>>
```



C++ MDspan

- How to get MemorySpaces?
 - `accessor_memspace<int,CudaSpace>`
- `mdspan` is non-owning?
 - Derive Kokkos View from `MDspan`
 - store the extra reference count handle
 - Provide allocating constructors
 - Or: use `accessor` with `shared_ptr` as data handle ...
- What about subviews?
 - `subspan` is part of the proposal
- <https://github.com/ORNL/cpp-proposals-pub/tree/master/P0009>



C++ BLAS



- Sandia leads a proposal supported by various parties (including Intel, NVIDIA, AMD and ARM)
- Goals: scalar agnostic, layout aware, support parallelism
- Approach:
 - `Mdspan` (and `mdarray`) as arguments
 - Model after C++ parallel algorithms

```
// y = 3.0 * A * x;  
matrix_vector_product(par, scaled_view(3.0, A), x, y);  
// y = 3.0 * A * x + 2.0 * y;  
matrix_vector_product(par, scaled_view(3.0, A), x, scaled_view(2.0, y), y);  
// y = transpose(A) * x;  
matrix_vector_product(par, transpose_view(A), x, y);
```



Sandia
National
Laboratories