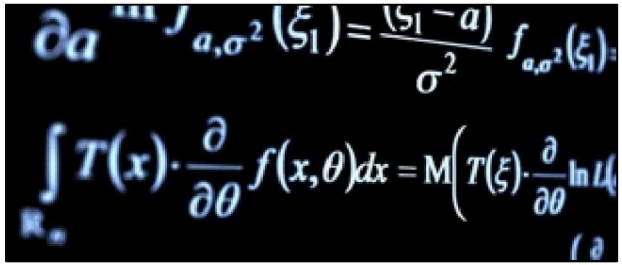
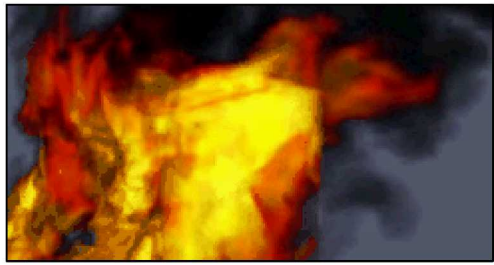
SAND2019-4605PE



$$\int T(x)\cdot\frac{\partial}{\partial\theta}f(x,\theta)dx = \mathrm{M}\left(T(\xi)\cdot\frac{\partial}{\partial\theta}\ln L\right.$$



Kokkos RoadMap

Unclassified Unlimited Release

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

Christian R. Trott, - Center for Computing Research

Sandia National Laboratories/NM

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.

Backend Plan

- OpenMP Update FY19/20:
  - OpenMP compiler/runtime now much better than it used to be
    - => Don’t need to write my own scheduler anymore
  - use OpenMP 4 capabilities for host backend
  - Custom reductions, vectorization, etc.
- CUDA FY19/20
  - 2019 CUDA Graphs support -> see later
- AMD Ready end
  - Existing ROCm backend based on AMD recommendation
    - Buggy compiler prevents compiling of any apps, worked around bugs in unit tests
  - Starting new backend using HIP compiler front end, early FY20 standard capabilities
- Intel GPU (ANL Aurora)
  - Initial capabilities (i.e. enough to do our standard tutorial) end of FY20
  - Full production support FY21

Harmonized Hierarchical Parallelism

```
parallel_for("BigKernel", TeamPolicy<>(N,AUTO,8) KOKKOS_LAMBDA (const team_t& team) {
  parallel_for(TeamVectorRange(team,M), [&] (const int j) {
    // Allowed to call ThreadVectorRange here??
  });
  //...
  parallel_for(TeamThreadRange(team,M), [&](const int j) {
    // Would TeamThreadRange be allowed to be vectorized?
    parallel_for(ThreadVectorRange(team,K), [&] (const int k) {
      //...
    });
    //...
  });
});
```

- What are the semantics of the inner loop?
- What nesting levels am I allowed to call where?

Harmonized Hierarchical Parallelism

- Reusing RangePolicy for “no other nesting”
- Get ”nested executor” for loops which have another nesting level
  - Kind of needs C++17 for template deduction from ctr arguments

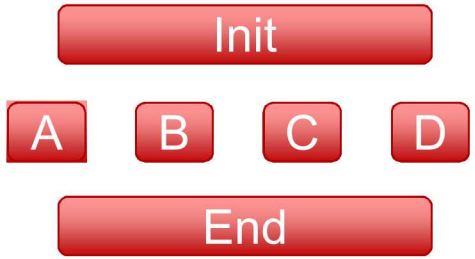
```
parallel_for("BigKernel", TeamPolicy<>(N,AUTO,8) KOKKOS_LAMBDA (const team_t& team) {
  parallel_for(RangePolicy(team,M), [&] (const int j) {
    // Allowed to call ThreadVectorRange here??
  });
  //...
  parallel_for(TeamThreadRange(team,M), [&](const thread_t, const int j) {
    // Would TeamThreadRange be allowed to be vectorized?
    parallel_for(RangePolicy(thread_t,K), [&] (const int k) {
      //...
    });
    //...
  });
});
```

Generic Execution Space Instances

- Added CUDA stream support as interop, but whatabout a general interface
- Propose e.g. Kokkos::partition(ExecSpace,PartitioningRule) functionality

```
auto instances = partition(DefaultExecutionSpace(),4);

parallel_for(”Init",RangePolicy<>(0,N),functor_init);
parallel_for("A",RangePolicy<>(instances[0],0,N),functor_A);
parallel_for("B",RangePolicy<>(instances[1],0,N),functor_B);
parallel_for("C",RangePolicy<>(instances[2],0,N),functor_C);
parallel_for("D",RangePolicy<>(instances[3],0,N),functor_D);
parallel_for(”End",RangePolicy<>(0,N),functor_end);
```



- Is equal partitioning enough?
- If partitioning is not possible, fail or just return same instance 4 times?

Pipelining Kernels

- Often dependency is only iteration to iteration.
  - Exploiting this provides caching bene fits, e.g.

```
parallel_for("Axpby",N,KOKKOS_LAMBDA (const int i) {
  c(i) += a(i)+b(i);
});
parallel_reduce("Dot",N,KOKKOS_LAMBDA (const int i, double& lsum) {
  lsum += c(i)*c(i);
},sum);
```

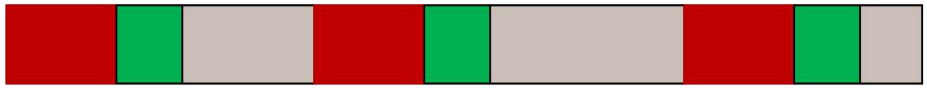
- Pipeline interface promises only iteration to iteration dependency
  - What about reductions, and using reduction result in the next kernel?
  - Mix of RangePolicy/TeamPolicy??

Latency Limited Kernels

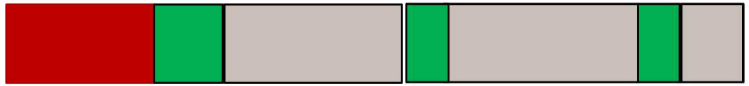
- Many applications run into latency limits
  - Targeting 1000 timesteps or solver iterations per second
  - Need to optimize for kernels of 20us or less runtime
- Underlying Programming Models have limits
  - CUDA launch latency 3us (Skylake) to 12us (Power9)
  - OpenMP max loop rate about 1us/per loop
- Allocation rate limited
  - CUDA UVM allocation takes 200us!
- MPI communication?

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 initating 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);
```

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

Capturing Host Events

- Maybe capture as host_spawn?
  - The captured host lambda must stay valid, e.g. capture comm class as const?

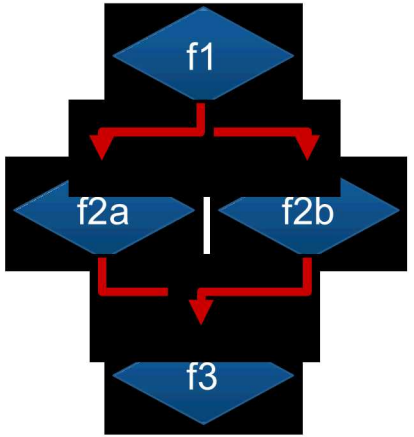
```
Kokkos::View<double> reduce_result("red");
auto graph = Kokkos::capture_kernel_graph(scheduler,[=] () {
  Kokkos::parallel_for("A",N,KOKKOS_LAMBDA(const int i) {...});
  Kokkos::parallel_reduce("A",N,KOKKOS_LAMBDA(const int i, double& r) {...},reduce_result);
  scheduler.spawn(SingleTask, [=] (team_t)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

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?

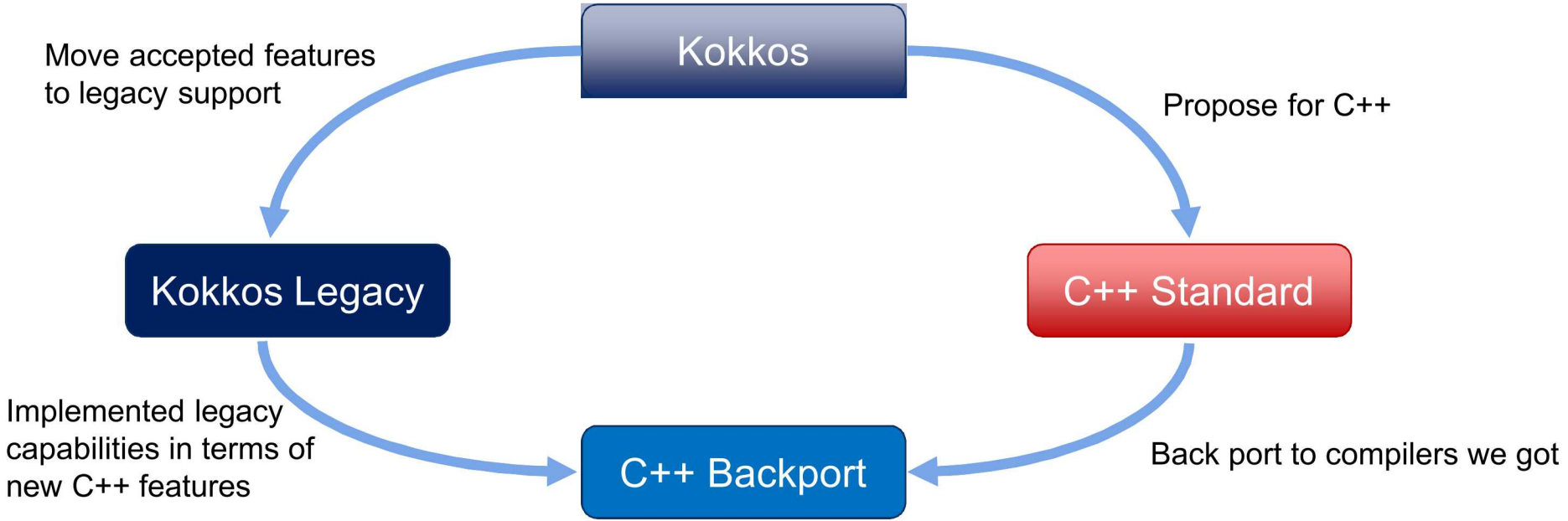
More Generic Properties

- Which properties are valid for which ExecutionPolicies?
  - Dynamic Schedule, index type, ExecutionSpace, …
- How to tell which properties are required, vs hints?
- How do I add a property in a generic context?
- C++ -> require/prefer mechanism
  - May return the same object

```
template<class exec_t>
void foo(exec_t exec) {
  auto exec_dynamic = require(exec,Schedule<Dynamic>());
  parallel_for(exec_dynamic, …);
}
template<class exec_t>
void foo(exec_t exec) {
  auto exec_chunked = prefer(exec,ChunkSize(16));
  parallel_for(exec_chunked, …);
}
```

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 timensions
  - 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

Other things

- Resilience
  - See Jeff's talk from Tuesday
- PGAS support
  - See Christian's talk from Tuesday
- SIMD Support
  - Remember discussion from Tuesday
- Documentation, Documentation, Documentation ….

Timeline Summary

- FY19/20
  - CUDA Graphs Support
  - Initial AMD HIP backend
  - ExecSpace Instances
- FY20
  - Coarse grained tasking
  - Initial Intel GPU backend
  - AtomicRef/MDSpan utilization (via backport)
  - C++14 requirement
- FY21
  - Production AMD and Intel GPU backend