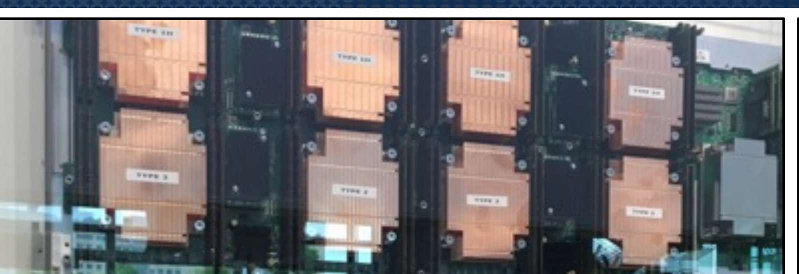
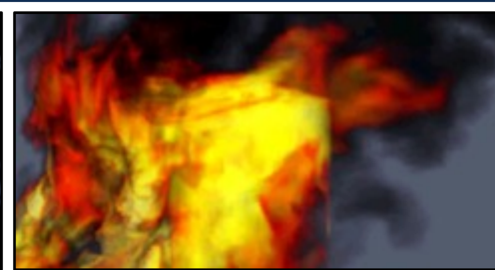
Exceptional service in the national interest





Kokkos: C++ Performance Portability for Production

Unclassified Unlimited Release

D. Sunderland, N. Ellingwood, D. Ibanez, 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.

SAND2019-9040 TR

Cost Of Software

10 LOC / hour ~ 20k LOC / year

- Optimistic estimate: 10% of an application needs to get rewritten for adoption of Shared Memory Parallel Programming Model
- Typical Apps: 300k – 600k Lines
  - Uintah: 500k, QMCPack: 400k, LAMMPS: 600k; QuantumEspresso: 400k
  - Typical App Port thus 2-3 Man-Years
  - Sandia maintains a couple dozen of those
- Large Scientific Libraries
  - E3SM: 1,000k Lines x 10% => 5 Man-Years
  - Trilinos: 4,000k Lines x 10% => 20 Man-Years

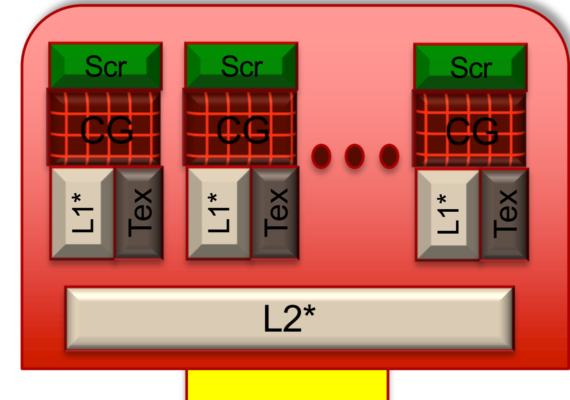
A Vision of the future

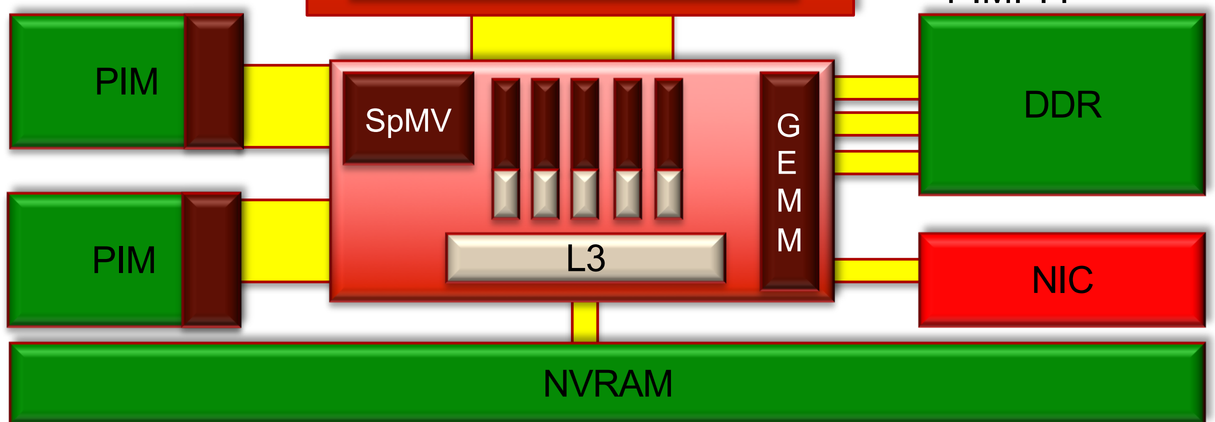
4 Memory Spaces
- Bulk non-volatile (Flash?)
- Standard DDR (DDR4)
- Fast memory (HBM/HMC)
- (Segmented) scratch-pad on die

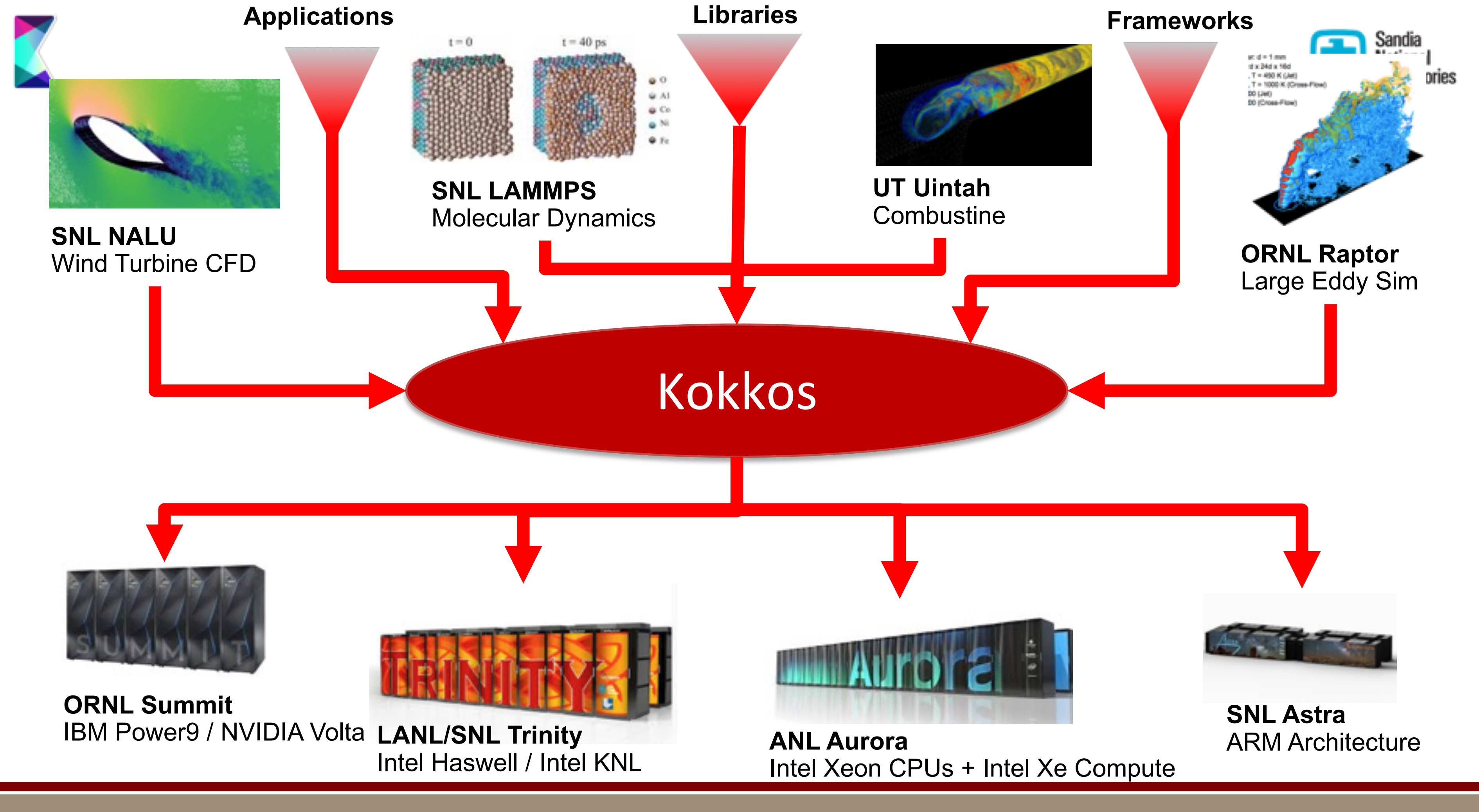
3 Execution Spaces
- Throughput cores (GPU)
- Latency optimized cores (CPU)
- Processing in memory
- SpMV and GEMM accelerator

Special Hardware
- Non caching loads
- Read only cache
- Atomics

3 Programming models??
- GPU: CUDA, HIP, SyCL, OpenMP
- CPU: OpenMP, OpenACC
- PIM: ??







Applications
Libraries
Frameworks
t = 0
t = 40 ps
O
Al
Co
Ni
Fe
SNL NALU
Wind Turbine CFD
SNL LAMMPS
Molecular Dynamics
UT Uintah
Combustine
ORNL Raptor
Large Eddy Sim
Kokkos
ORNL Summit
IBM Power9 / NVIDIA Volta
LANL/SNL Trinity
Intel Haswell / Intel KNL
ANL Aurora
Intel Xeon CPUs + Intel Xe Compute
SNL Astra
ARM Architecture

Outline

- The Kokkos EcoSystem
  - Core, Kernels and Tools
- Capabilities
  - Parallel Dispatch
  - Data structures
  - Algorithms
- Applications
- Future Developments
  - Latency Optimization
  - Remote Spaces
  - C++ Standard

What is Kokkos?

- A C++ Programming Model for Performance Portability
  - Implemented as a template library on top of CUDA, OpenMP, ROCm, …
  - Aims to be descriptive not prescriptive
  - Aligns with developments in the C++ standard
- Expanding solution for common needs of modern science/engineering codes
  - Math libraries based on Kokkos
  - Tools which allow inside into Kokkos
- It is Open Source
  - Maintained and developed at https://github.com/kokkos
- It has many users at wide range of institutions.

Kokkos EcoSystem

Science and Engineering Applications

Trilinos

Kokkos Tools

- Debugging
- Profiling
- Tuning

Kokkos EcoSystem

Kokkos Kernels

- Linear Algebra Kernels
- Graph Kernels

Kokkos Core

- Parallel Execution
- Parallel Data Structures

Kokkos Support

- Documentation
- Tutorials
- Bootcamps
- App support

Kokkos Remote Spaces

- PGAS
- IO

HBM
DDR

Multi-Core

Many-Core

APU

CPU + GPU

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. Womeldorff,*
former: ***H.C. Edwards****, D. Labreche, Fernanda Foertter*

Some Kokkos Stats Since 2015

- 18 Releases Since 2016
  - Only 5 since December 2017
- 50 Contributors
  - 17 with more than 10 commits
  - 11 with more than 10k lines touched
- 1345 Issues of which 1134 were resolved
  - 305 bug reports
  - 381 enhancement requests
  - 129 Feature Requests
- 766 pull requests
- 19k messages by 150 members on kokkosteam.slack.com (Started in 2017)

Kokkos Core Abstractions

Kokkos

Data Structures

- Memory Spaces ("Where")
  - HBM, DDR, Non-Volatile, Scratch
- Memory Layouts
  - Row/Column-Major, Tiled, Strided
- Memory Traits ("How")
  - Streaming, Atomic, Restrict

Parallel Execution

- Execution Spaces ("Where")
  - CPU, GPU, Executor Mechanism
- Execution Patterns
  - parallel_for/reduce/scan, task-spawn
- Execution Policies ("How")
  - Range, Team, Task-Graph

Kokkos Kernels

- BLAS, Sparse and Graph Kernels on top of Kokkos and its View abstraction
  - Scalar type agnostic, e.g. works for any types with math operators
  - Layout and Memory Space aware
- Can call vendor libraries when available
- View have all their size and stride information => Interface is simpler

```
// BLAS
int M,N,K,LDA,LDB; double alpha, beta; double *A, *B, *C;
dgemm('N','N',M,N,K,alpha,A,LDA,B,LDB,beta,C,LDC);
```

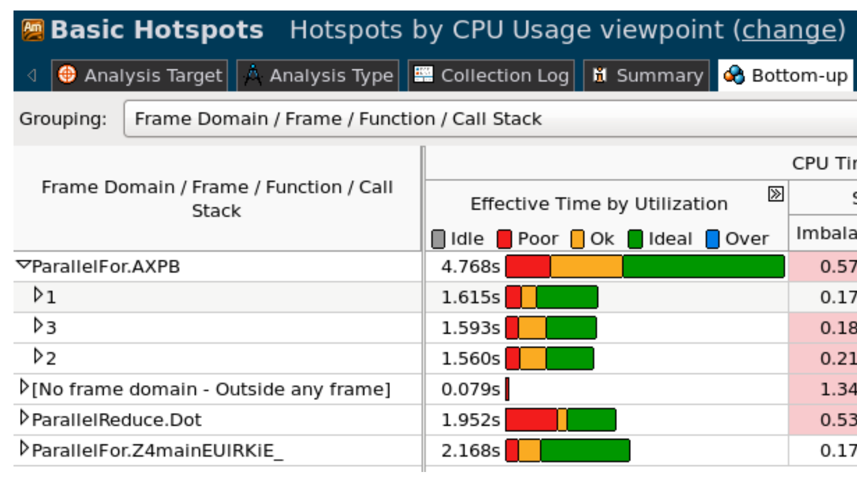
```
// Kokkos Kernels
double alpha, beta; View<double**> A,B,C;
gemm('N','N',alpha,A,B,beta,C);
```

- Interface to call Kokkos Kernels at the teams level (e.g. in each CUDA-Block)

```
parallel_for("NestedBLAS", TeamPolicy<>(N,AUTO), KOKKOS_LAMBDA (const team_handle_t& team_handle) {
  // Allocate A, x and y in scratch memory (e.g. CUDA shared memory)
  // Call BLAS using parallelism in this team (e.g. CUDA block)
  gemv(team_handle,'N',alpha,A,x,beta,y)
});
```

Kokkos-Tools Profiling & Debugging

- Performance tuning requires insight, but tools are different on each platform
- Insight into
- KokkosTools: Provide common set of basic tools + hooks for 3rd party tools
- One common issue abstraction layers obfuscate profiler output
  - Kokkos hooks for passing names on
  - Provide Kernel, Allocation and Region
- No need to recompile
  - Uses runtime hooks
  - Set via env variable



Kokkos: *Capabilities*

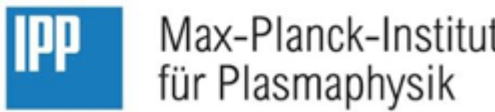
Sandia National Laboratories

Kokkos: *Applications and Users*

Kokkos Based Projects

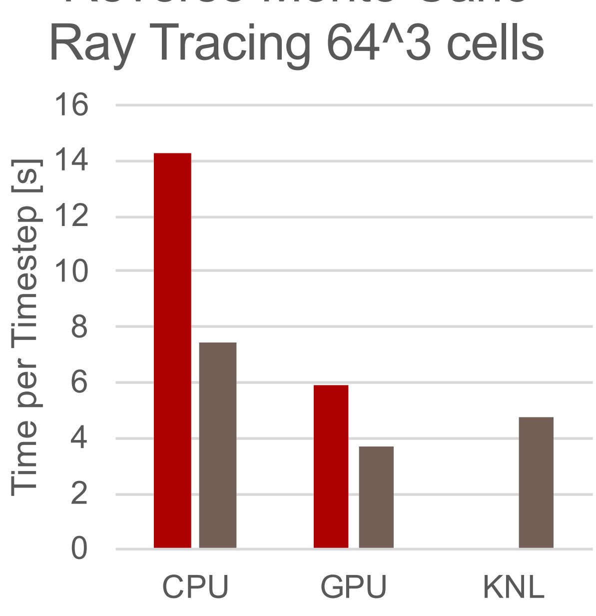
- Production Code Running Real Analysis Today
  - We got about **12** or so.
- Production Code or Library committed to using Kokkos and actively porting
  - Somewhere around **30**
- Packages In Large Collections (e.g. Tpetra, MueLu in Trilinos) committed to using Kokkos and actively porting
  - Somewhere around **50**
- Counting also proxy-apps and projects which are evaluating Kokkos (e.g. projects who attended boot camps and trainings).
  - Estimate **80-120** packages.

Kokkos Users



Uintah

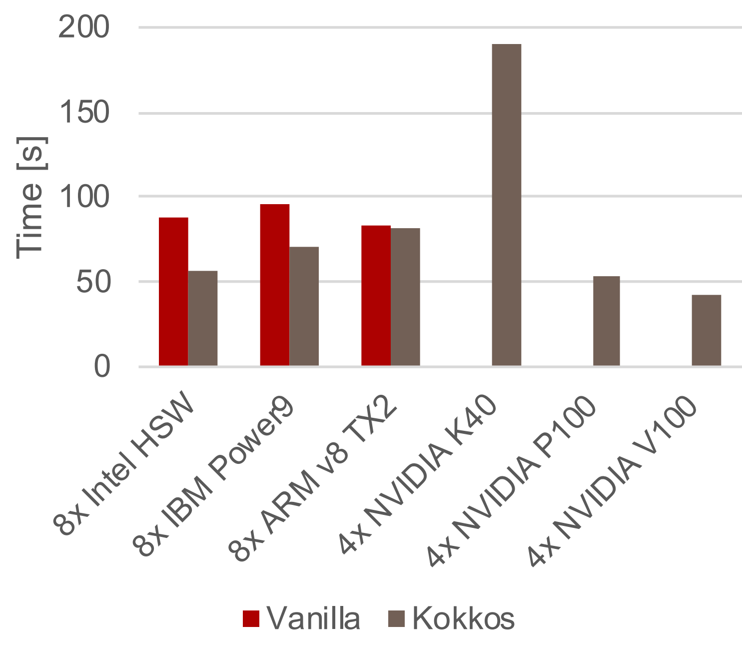
- System wide many task framework from University of Utah led by Martin Berzins
- Multiple applications for combustion/radiation simulation
- Structured AMR Mesh calculations
- Prior code existed for CPUs and GPUs
- Kokkos unifies implementation
- Improved performance due to constraints in Kokkos which encourage better coding practices



Questions: Dan Sunderlan

Questions: Stan Moore

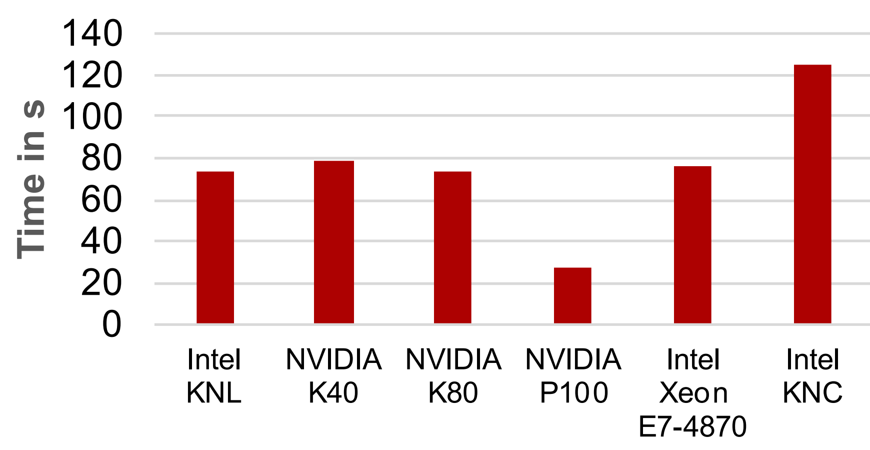
- Widely used Molecular Dynamics Simulations package
- Focused on Material Physics
- Over 500 physics modules
- Kokkos covers growing subset of those
- REAX is an important but very complex potential
  - USER-REAXC (Vanilla) more than 10,000 LOC
  - Kokkos version ~6,000 LOC
  - LJ in comparison: 200LOC
  - Used for shock simulations

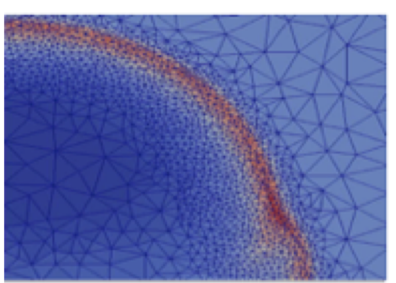


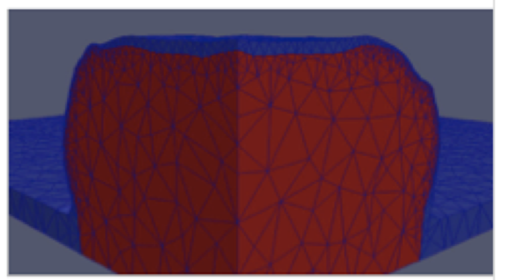
Alexa

- Portably performant shock hydrodynamics application
- Solving multi-material problems for internal Sandia users
- Uses tetrahedral mesh adaptation

Questions: Dan Ibanez





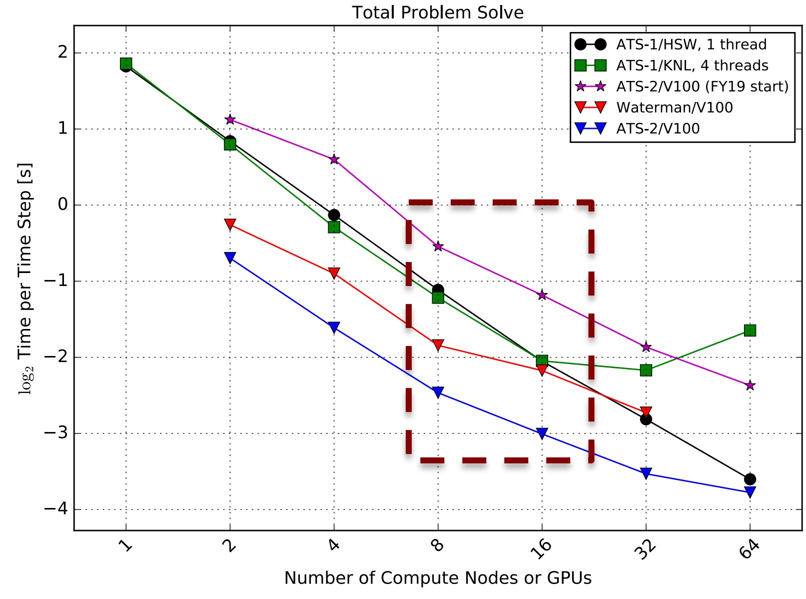


- All operations are Kokkos-parallel
- Test case: metal foil expanding due to resistive heating from electrical current.

SPARC Courtesy of: Micah Howard

Sandia National Laboratories

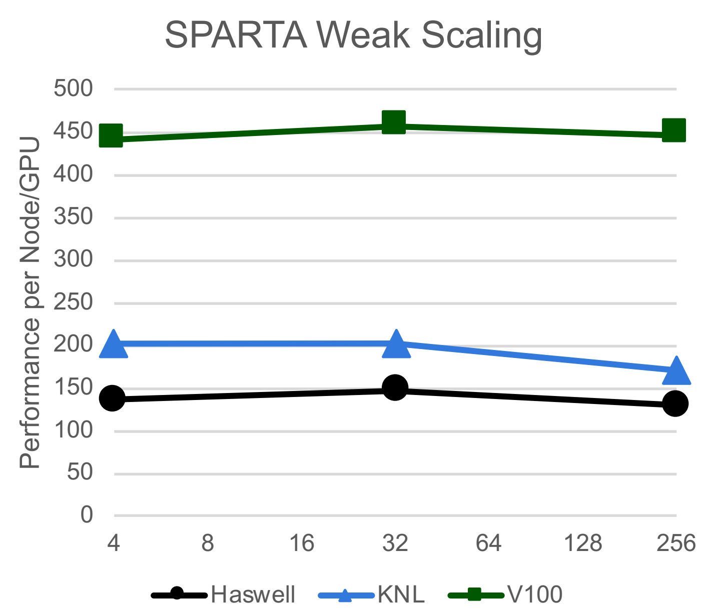
- Goal: solve aerodynamics problems for Sandia (transonic and hypersonic) on ‘leadership’ class supercomputers
- Solves compressible Navier-Stokes equations
- Perfect and reacting gas models
- Laminar and RANS turbulence models -> hybrid RANS-LES
- Primary discretization is cell-centered finite volume
- Research on high-order finite difference and discontinuous Galerkin discretizations
- Structured and unstructured grids



4 Sierra nodes (16x V100) equivalent to ~40 Trinity nodes (80x Haswell 16c CPU)

Sparta: Production Simulation at Scale

- **S**tochastic **PA**rallel **R**arefied-gas **T**ime-accurate **A**nalyzer
- 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



Sandia National Laboratories

Kokkos: *Future Developments*

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 …
  - SyCL backend for Intel, main development at ANL
  - OpenMPTarget for AMD, Intel and NVIDIA, lead at Sandia

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

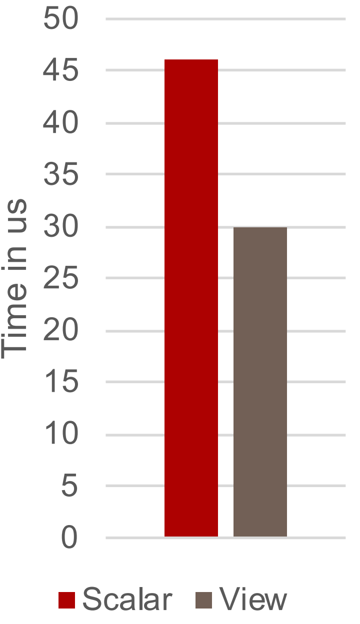
- More asynchronous execution to hide launch latency
  - No API change, improve implementation (i.e. limit fences etc.)
  - May need hints from user to use latency instead of throughput opt path
- Fine Grained Tasking Interface
  - Potentially write big kernels with inner dependencies via tasking
- Execution 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);
```

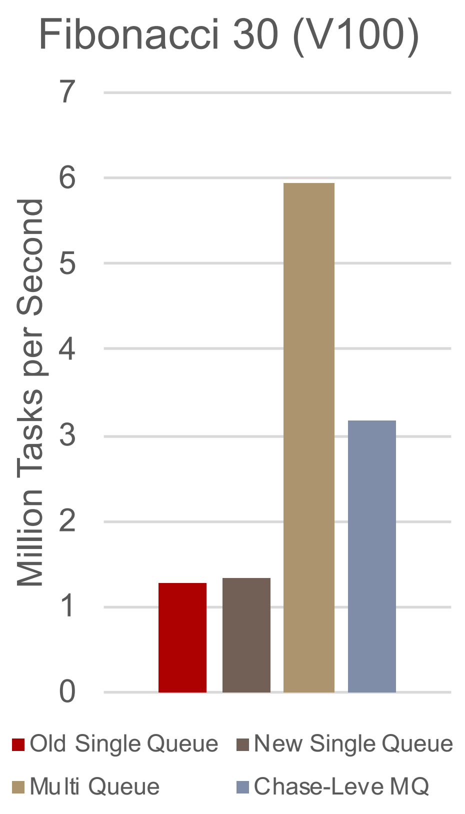
Rule of Thumb: Everything is asynchronous unless reducing into a scalar value!



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



Tasking Example Code

```
template< typename Scheduler >
struct FibonacciTask {
  using sched_type = Scheduler;
  using future_type = BasicFuture< long, Scheduler >;
  future_type fib_m1, fib_m2;
  const long n;

  KOKKOS_INLINE_FUNCTION
  TestFib( const value_type arg_n )
    : fib_m1(), fib_m2(), n( arg_n ) {}

  KOKKOS_INLINE_FUNCTION
  void operator()( typename sched_type::member_type & member, value_type & result ) {
    auto& sched = member.scheduler();
    if ( n < 2 ) { result = n; }
    else if ( !fib_m2.is_null() && !fib_m1.is_null() ) { result = fib_m1.get() + fib_m2.get(); }
    else {
      fib_m2 = task_spawn( TaskSingle( sched, TaskPriority::High ), FibonacciTask( n - 2 ) );
      fib_m1 = task_spawn( TaskSingle( sched ), FibonacciTask( n - 1 ) );

      BasicFuture<void, Scheduler> dep[] = { fib_m1, fib_m2 };
      BasicFuture<void, Scheduler> fib_all = sched.when_all( dep, 2 );


      if ( !fib_m2.is_null() && !fib_m1.is_null() && !fib_all.is_null() ) {
        respawn( this, fib_all, TaskPriority::High );
      } else { Kokkos::abort( "TestFib insufficient memory" ); }
    }
  }
```

Scheduler obtained from arguments: task could be a lambda

Spawn child tasks

Make compound dependency

Respawn task with new deps

If dependencies are not NULL this is respawn

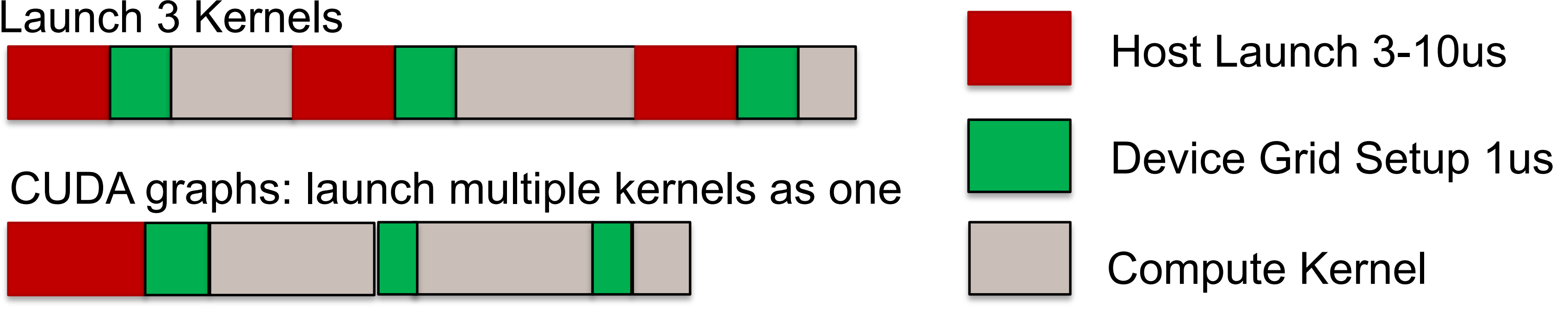
CUDA Stream Interop

- 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



- 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);
```

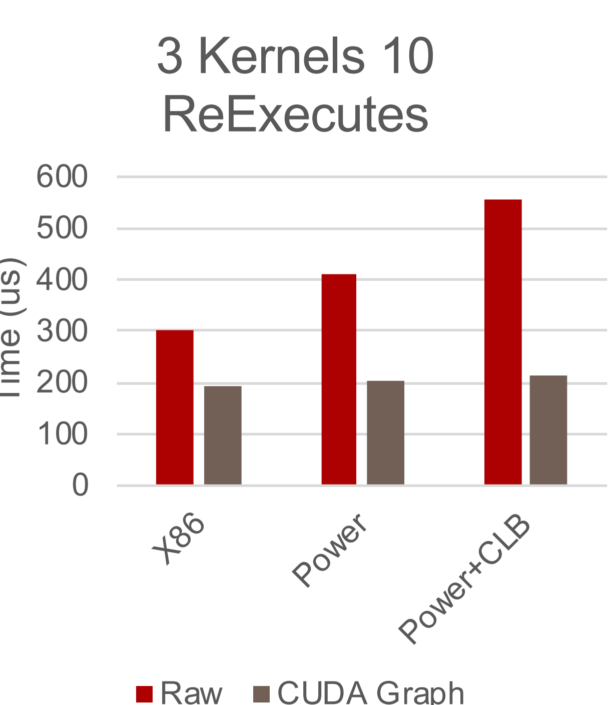
Kokkos Options To Leverage Graphs

- 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);
  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

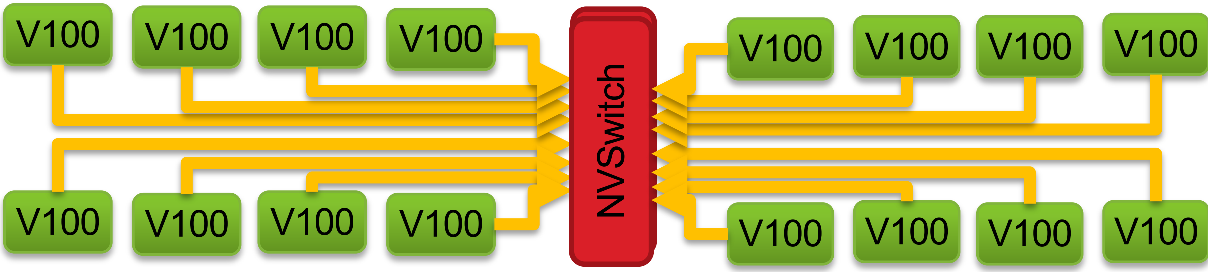
```
auto fut_1 = parallel_for( RangePolicy<>(“Funct1”, 0, N), f1 );
auto fut_2a = parallel_for( RangePolicy<>(“Funct2a”, fut_1,0, N), f2a);
auto fut_2b = parallel_for( RangePolicy<>(“Funct2b”, fut_1,0, N), f2b);
auto fut_3 = parallel_for( RangePolicy<>(“Funct3”, all(fut_2a,fut2_b),0, N), f3);
fence(fut_3);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

f1
f2a
f2b
f3

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

Kokkos Remote Spaces: PGAS Support

- PGAS Models may become more viable for HPC with both changes in network architectures and the emergence of “super-node” architectures
  - Example DGX2
  - First “super-node”
  - 300GB/s per GPU link

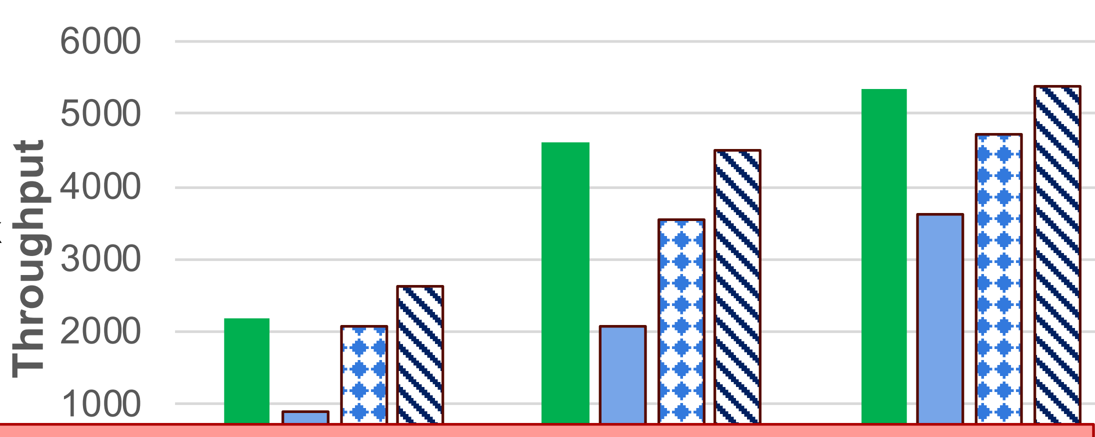


- Idea: Add new memory spaces which return data handles with shmem semantics to Kokkos View
  - `View<double**[3], LayoutLeft, NVShmemSpace> a(“A”,N,M);`
  - Operator `a(i,j,k)` returns:

```
template<>
struct NVShmemElement<double> {
   NVShmemElement(int pe_, double* ptr_):pe(pe_),ptr(ptr_) {}
   int pe; double* ptr;
   void operator = (double val) { shmem_double_p(ptr,val,pe); }
};
```

PGAS Performance Evaluation: miniFE

- Test Problem: CG-Solve
  - Using the miniFE problem N^3
  - Compare to optimized CUDA
  - MPI version is using overlapping
  - DGX2 4 GPU workstation
  - Dominated by SpMV (Sparse Matrix Vector Multiply)
  - Make Vector distributed, and store global indicies in Matrix
- 2 Variants



Warning: I don’t think this is a viable thing in the next couple years for most of our apps!!

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

Kokkos

Move accepted features to legacy support

Propose for C++

Kokkos Legacy

C++ Standard

Implemented legacy capabilities in terms of new C++ features

C++ Backport

Back port to compilers we got

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
- 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);
```

How To Expose Special Function Units?

Libraries!

- Easy to use for applications
- Connect with memory info
  - Is the data accessible and the correct layout?
- KokkosKernels has interface with all necessary information
  - Matrix in main GPU memory
  - RHS vector created on the fly in scratch memory
  - LHS vector in Host accessible memory

```
View<double**,CudaSpace> A = /*...*/;
View<double*,CudaHostPinnedSpace> y = /*...*/;
View<double*,Cuda::scratch_memory_space> x = /*...*/;
gemv(y,A,x); /* Execute in Cuda Space since it can access all data. */
```

Key Things to Help Compilers/Runtimes

- Encode information at compile time (as part of the type system)
  - Where does data life.
  - How do you access it.
  - Properties of algorithms.
- Be descriptive – not prescriptive
  - Say what you want to happen and give properties (see above)
  - Let the compiler/runtime figure out how to use that info
- Provide graceful fallbacks and defaults
- Make it possible to provide incrementally more information

That’s Great But I Don’t Trust TPLs

- Good News! We are working on contributing to the C++ standard!
- Executors for heterogeneous environements (C++23)
  - Control where and how stuff executes
  - Property mechanism to provide more information
  - Hierarchical executors for supporting hierarchical hardware (C++26)
- MDSpan for multi-dimensional arrays with accessors (C++23)
  - Templated on scalar, extents, layout and accessor

```
basic_mdspan<double,extents<dynamic_extent,8>,layout_left,basic_accessor<double>>
```

  - Extent accessors to provide typesafe info about storage place

```
basic_mdspan<double,extents<8,4>,layout_right,memspace_accessor<double,HBM>>
```

- BLAS support in the works: allows SpMV or GEMM accelerator support (C++23)

Summary

- ***Production Quality:*** Extensive Testing and wide usage enables good user experience
- ***Multi-Institution Developer Team:*** 4 National Labs + Swiss National Supercomputing Center support Kokkos directly
- ***Growing Userbase:*** More than 100 projects using Kokkos, many codes available online
- ***Not just the Programming Model:*** Tools and math library integration provide the basis for complex projects

Sandia
National
Laboratories