Kokkos: Capabilities Overview

Christian R. Trott [1],

[1]Sandia National Laboratories

Kokkos Support Webinar, March 3, 2017

Sandia National Laboratories is a multi-mission laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

What this is:

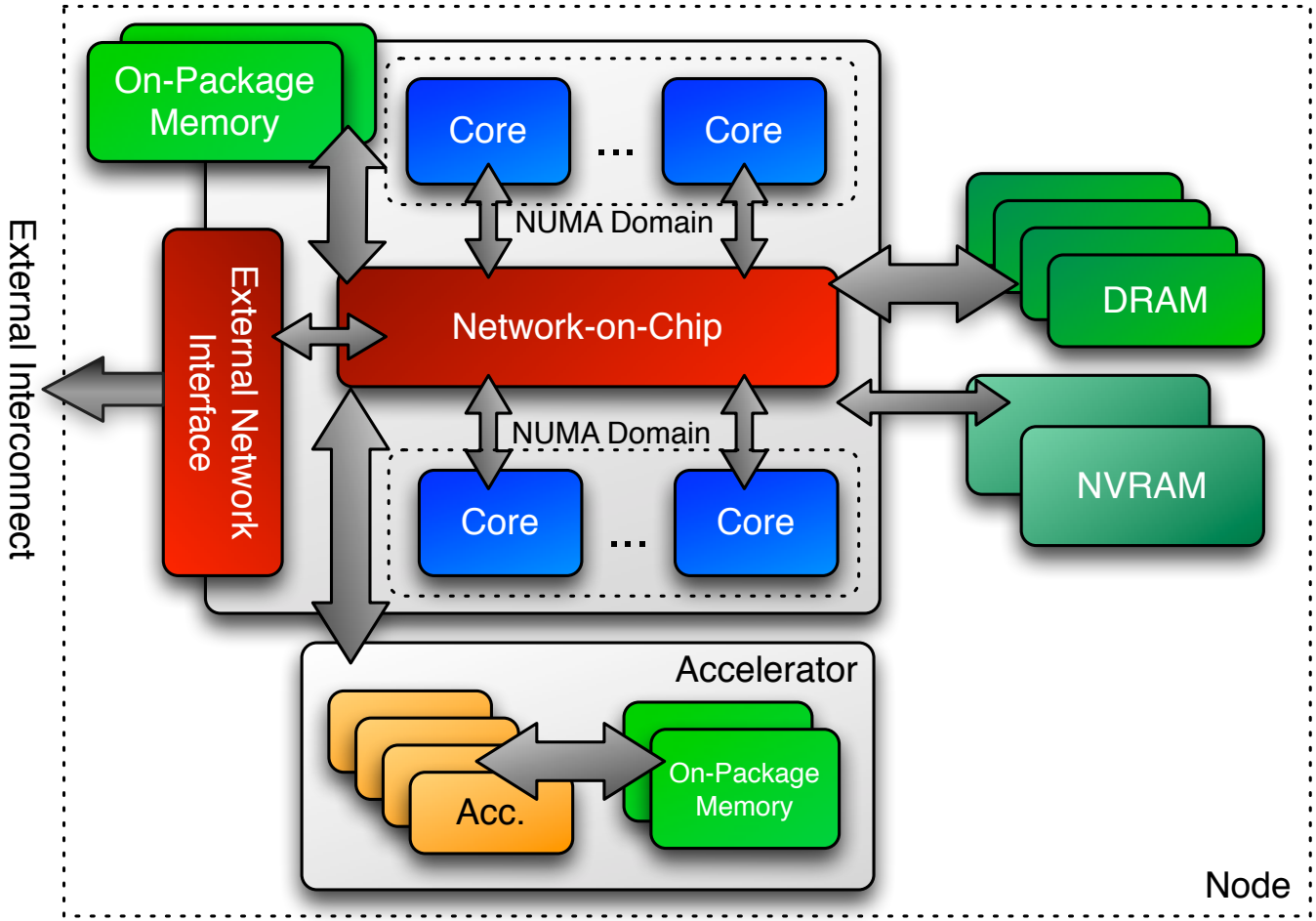
- Overview of Capabilities
- Short characterization of API features and semantics
- Information with the fire hose
- Maybe something you can look at later again ...

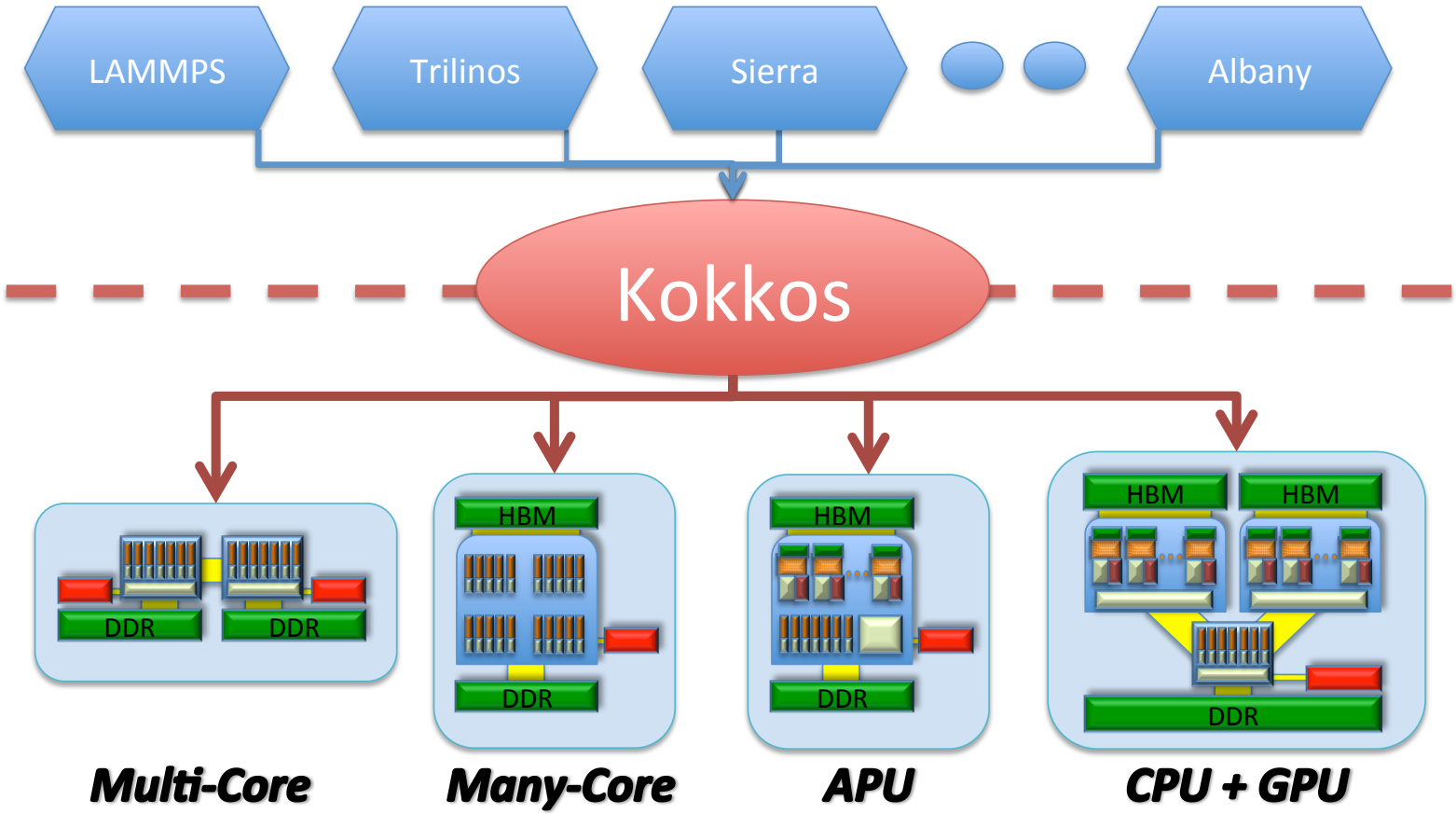
What this is Not:

- Not a tutorial
- Not a class in parallel programming
- Not a class in learning to use Kokkos

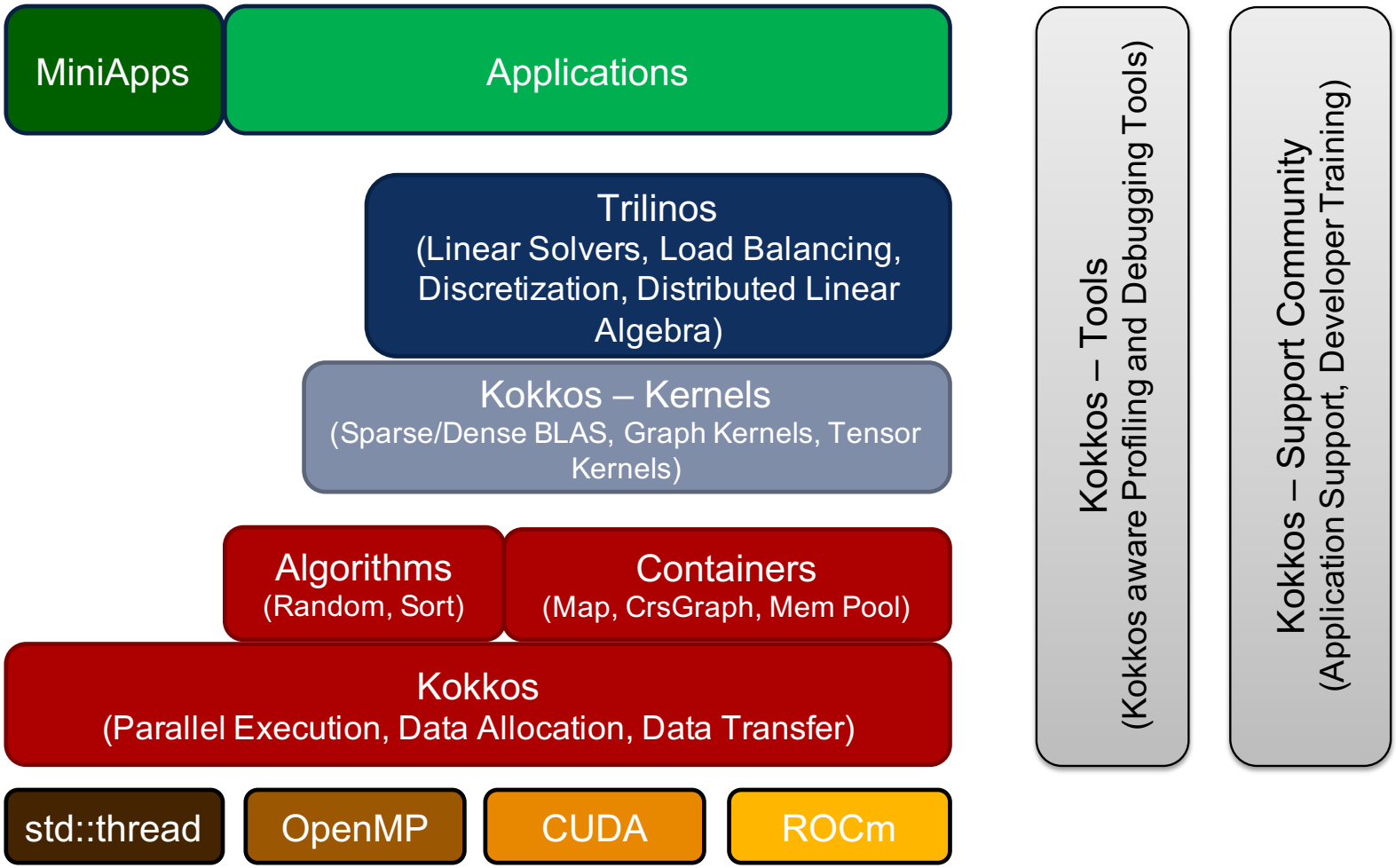
Content

Target machine:




LAMMPS
Trilinos
Sierra
Albany
Kokkos
DDR
DDR
HBM
DDR
HBM
DDR
HBM
HBM
DDR
Multi-Core
Many-Core
APU
CPU + GPU

The Kokkos EcoSystem



The Repositories

- Organization: https://github.com/kokkos
- Kokkos: Programming Model, Algorithms, Containers
- Kokkos-Kernels: Linear Algebra, Graph and Tensor Kernels
- Kokkos-Tools: Profiling and Debugging Tools
- Kokkos-Tutorials: Tutorials with Slides and Exercises
- Kokkos-MiniApps: MiniApps implemented in Kokkos
- Kokkos-Docs: Documentation Webpages etc.

More Reading Material

- Search for Kokkos here:
- https://cs.sandia.gov : Publications
- www.gputechconf.com/gtcnew/on-demand-gtc.php

Kokkos Core

- Maturity: Mostly Very High, API very stable
- New Features introduced in Experimental Namespace
- Testing Extensive: over 200 configurations per night
- Platforms: X86, XeonPhi, Power8, ARM, NVIDIA (Kepler, Maxwell, Pascal), AMD upcoming
- Compilers: GCC 4.7.2 - 6.1, Intel 14.2 - 17.1, Clang 3.6-4.0 (head), PGI 17.1, IBM 13.1.5, NVCC 7.0-8.0

Kokkos-Kernels

- Under Development
- Kernels used by Trilinos are available (dense, sparse, graph)
- No full BLAS coverage yet
- Performance on sparse mostly superior to vendor libraries

Kokkos-Tools

- Under Development
- Some basic profiling tools available
- Hooks to VTune and Nsight
- Other third party Profiling Tools are developing hooks

Kokkos-Tutorials

- Extensive Full Day Tutorial available for Kokkos Core

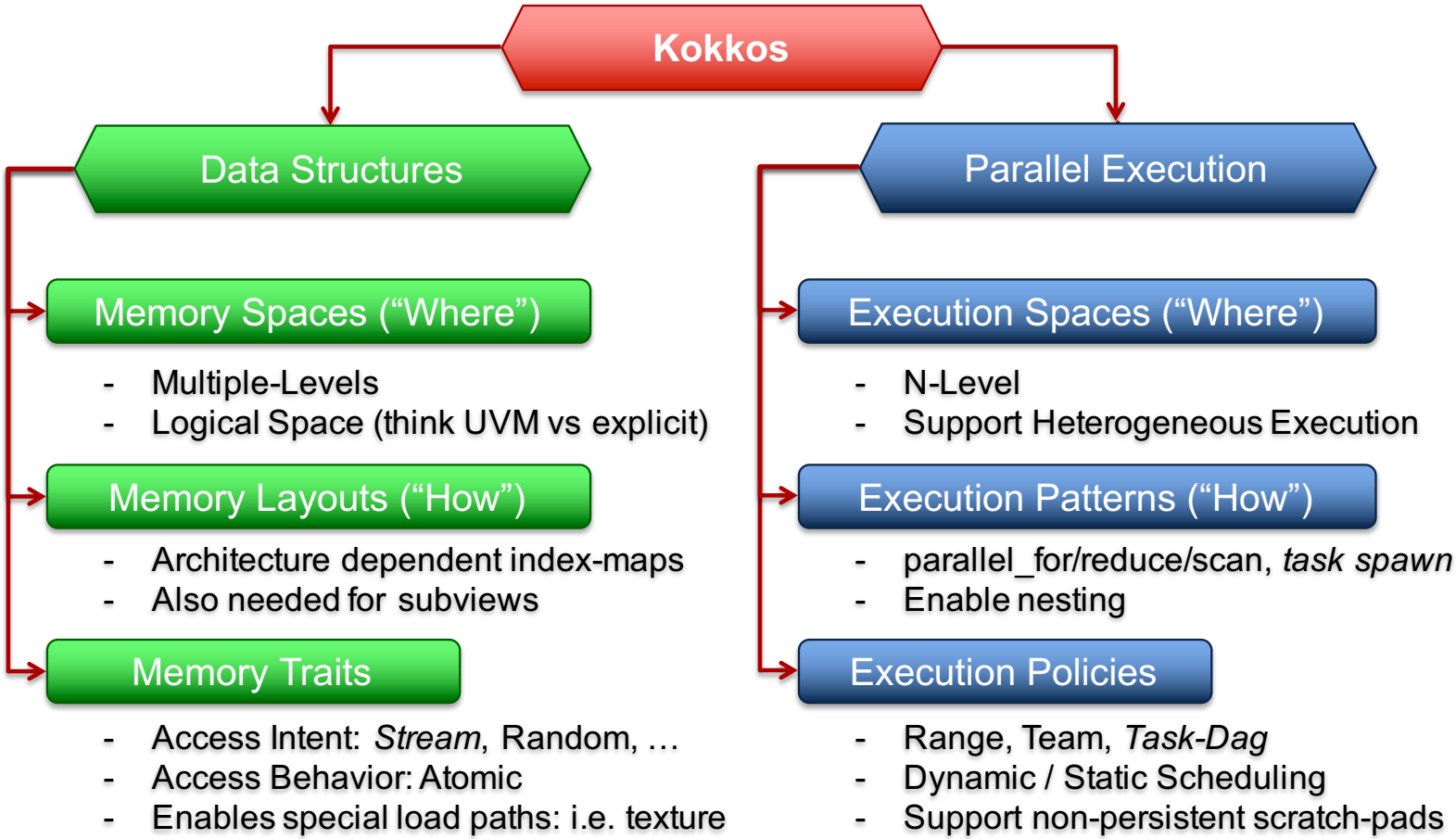
Kokkos-MiniApps

- Just started collecting apps

Kokkos-Docs

- Just started

Programming Model Abstractions



Functors and Lambdas

- Kokkos Patterns call operator() (...) of classes
- Either write these classes explicitly (Functor)
- Or write them implicitly (Lambda)
- Other than in task dispatch functors are treated as const

```
struct Functor {
  //Data Members
  void operator() (const int i) const {
     //Code
  }
};

auto lambda = [=] (const int i) {
 //Code
}
```

Execution Policies Patterns

- `parallel_for`: independent work items
- `parallel_reduce`: work items contribute to a reduction
- `parallel_scan`: pre-fix or post-fix scan algorithm
- `single`: restrict execution to subset of current parallelism
- `host_spawn`: add a task to a task DAG from the host
- `task_spawn`: add a task to a task DAG from within a task
- `respaw`: respawn an active task

Execution Policies

- `RangePolicy`: iterate over cartesian index space
- `TeamPolicy`: each work item is worked on by team of threads, allow nested patterns
- `TeamThreadRange`: policy for nested parallel patterns in TeamPolicy or TaskTeam
- `ThreadVectorRange`: policy for nested parallel patterns
- `TaskSingle`: serial task
- `TaskTeam`: task for a team of threads, allow nested patterns
- `PerTeam`: policy for `single` pattern, execute ones per team
- `PerThread`: policy for `single` pattern, ones per thread

Parallelize a single loop level

```
parallel_for(N, KOKKOS_LAMBDA (const int i) {
  c(i) = a(i) + b(i);
});
```

Specify Range

```
parallel_for(RangePoliy<>(5,N-5), KOKKOS_LAMBDA (const int i) {
  c(i) = a(i) + b(i);
});
```

Specify where to Execute

```
parallel_for(RangePoliy<HostExecutionSpace>(0,N),
  KOKKOS_LAMBDA (const int i) {
  c(i) = a(i) + b(i);
});
```

Specify Scheduling

```
parallel_for(RangePoliy<Schedule<Dynamic>>(0,N),
  KOKKOS_LAMBDA (const int i) {
  c(i) = a(i) + b(i);
});
```

Default Summation Reduction

```
double result;
parallel_reduce(N, KOKKOS_LAMBDA (const int i, double& tmp) {
  tmp += a(i)*a(i);
},result);
```

In-Build Max Reduction

```
double result;
parallel_reduce(N, KOKKOS_LAMBDA (const int i, double& tmp) {
  tmp = tmp > a(i) ? tmp : a(i);
},Max<double>(result));
```

Other in-builds are: `Sum`, `Prod`, `Max`, `Min`, `LAnd`, `LOr`, `LXor`, `BAnd`, `BOr`, `BXor`, `MaxLoc`, `MinLoc`, `MinMax`, `MinMaxLoc`

Build Your Own Custom Reduction

- One can write custom reducers
- Struct with init and join function
- Some typedefs to specify scalar types

How to allocate data

- Preferred: Managed Kokkos Views (next slide)
- If necessary raw allocations

Raw Allocations

```
double* d = (double*) kokkos_malloc<MemSpace>("Name",num_bytes);
//...
kokkos_realloc(d,new_num_bytes);                                    ;
//...
kokkos_free(d);
```

- `MemSpace` can be omitted (uses default memory space)

View Basics

- Multi Dimensional Arrays (up to 8D)
- Compile Time Rank, Runtime or Compile Time Dimensions
- Memory Layouts: how to map indicies to memory locations
- Memory Space: where to put the data
- Memory Traits: specify access intent
- Reference Counted (unless specified not to)

```
View<double***> a("A",N0,N1,N2); // Three runtime dimensions
View<double*[N1][N2] b("B",N0); // One runtime, two compile time
View<double*, HostSpace> c("C",N0); // Specify memory location
View<const double**, LayoutLeft> d("D",N0); // Specify Layout
View<double***, MemoryTraits<Atomic> > e(a); // Atomic view of a
```

Life Cycle of Views

```
View<double**> a("A",N0,N1); // Create View
// ...
  a(i,j) = foo(...);
// ...
{
  View<const double**> b(a); // Create const view of same data

  a = View<double**>(); // A is now unitialized, b still there
}
// Now the allocation is gone
```

SubViews

```
View<double***,LayoutRight> a("A",N0,N1,N2);
// Use subview function, no need to know layout of result
auto a_i = subview(a,i,pair<int,int>(3,N1-3),ALL);
// Use subview construction: need to know layout, less overhead
View<double*[10],LayoutRight> a_j(a,j,ALL,pair<int,int>(3,13));
```

Atomic Views

- Can support POD of any size (i.e. also larger than 128 bit)
- Utilizes optimal backend implementation where possible
- Does not require an atomic allocation
- All relevant operators supported (arithmetic and logical)

```
View<double*> values("V",K);
// fill vals
View<double*,MemoryTraits<Atomic>> a_vals(values);
parallel_for(N, KOKKOS_LAMBDA (int i) {
  if(foo(N)) a_vals(N%K) += i;
});
```

Free Functions

- Capabilities like Atomic View
- Simply takes pointers and update values

```
int loc = atomic_fetch_add(&counter(),n);
```

How to move data between (Physical) Memory Spaces

- Use explicit deep copies
- Use memory spaces which implicitly transfer data
- Run hardware in cache mode (e.g. Intel KNL)

Explicit data Movement

```
View<int**> a("A",N0,N1); // Create View
// Create Mirror View in HostSpace (if a is on Host, h_a==a)
View<int**>::HostMirror h_a = create_mirror_view(a);
for(int i; i<N0; i++) { // ... fill h_a ... }
deep_copy(a,h_a); // Move data to a
parallel_for(N0, KOKKOS_LAMBDA (int i) { // use a ... });
```

Memory Spaces

```
// Use appropriate memory space: for example CudaUVMSpace
View<int**,CudaUVMSpace> a("A",N0,N1); // Create View
for(int i; i<N0; i++) { // ... fill a on host... }
// Software or hardware will page migrate
parallel_for(N0, KOKKOS_LAMBDA (int i) { // use a ... });
```

Tightly Nested Loops

- Used when nested loops don't have code between nest levels
- Typical application: regular grid based computing
- Exposes Parallelism of all loop levels
- Subdivides loops into tiles, parallel both inter and intra tile
- Specify iteration order over tiles and within tiles

Memory Spaces

```
View<int***> a("A",N0,N1,N2), b("B",N0,N1,N2), c("C",N0,N1,N2);
// ... Fill Views

RangePolicy<Rank<3,Iterate::Left,Iterate::Right>>
  policy({0,0,0},{N0,N1,N2},{4,4,32}); //Start, End, Tiling

parallel_for(policy, KOKKOS_LAMBDA(int i, int j, int k) {
  c(i,j,k) = a(i,j,k) + b(i,j,k);
});
```

Hierarchical Parallelism

- Use to work with multiple threads on work item
- Parallelize Nested Loops and Reductions
- Launch League of Thread Teams
- Threads within a team are concurrently executing
- Nested Patterns with special policies

Example Matrix-Vector Multiplication

```
View<double**,LayoutRight> A("A",N0,N1);
View<double*> y("Y",N0), x("X",N1);
// ... Fill Views
TeamPolicy<> policy(N0,AUTO)
parallel_for(policy, KOKKOS_LAMBDA(TeamPolicy<>::member_type t) {
  int i = t.league_rank(); double y_i;
  parallel_reduce(TeamThreadRange(t,0,N1), [&](int j,double& s) {
    s += A(i,j) * x(j);
  },y_i);
  y(i) = y_i;
});
```

Scratch Space

- Provide Team or Thread local storage
- Scale per work-item scratch by concurrent items in flight
- Explictly cache shared and/or gathered data

```
TeamPolicy<> policy(N1/64,AUTO);
parallel_for(policy.set_scratch_size(Level,PerTeam(Size)),
  KOKKOS_LAMBDA (TeamPolicy<>::member_type t) {
  View<double*,ScratchSpace> s_x(t.team_scratch(),N1);
  parallel_for(TeamThreadRange(t,0,N1), [&] (int i) {
    s_x(i) = x(i); }
  t.team_barrier();
  int i_start = t.league_rank() * 64;
  for(int i=i_start; i<i_start + 64; i++) {
    double y_i;
    parallel_reduce(TeamThreadRange(t,0,N1), [&] (int j, double& s
      s += A(i,j) * s_x(j);
    },y_i);
    y(i) = y_i;
} }
```

Tasking

- Build dynamic Task Graph
- Tasks can spawn new tasks
- Tasks can respawn themselves in order to wait for child tasks
- Tasks can be single threaded, or use thread teams

```
struct Task {
  TaskScheduler<> scheduler; long val; Future<long> child_val;
  Task(sched_type sched, long v):scheduler(sched),val(v) {}
  void operator() (sched_type::member_type& h, long& result) {
    if(child_val.is_null()) {
      if(v>1) child_val = task_spawn(sched, v-1);
      else result = 1;
      if(!child_val) respawn( this , child_val);
    } else { result = val * child_val.get(); }
  }
};
TaskScheduler<> sched( HostSpace() , MemCap );
Future<long> f=host_spawn( TaskSingle(sched), Task(sched,10));
Kokkos::wait( sched );
```

Unordered Map: Optimized Insertion

- Optimized for Parallel Insertion
- No Dynamic Growth inside parallel kernel
- Data allocation is sparse

```
typedef double t_key;         // KeyType used for hashing
typedef SomeStruct t_value; // ValueType for the things we store
UnorderedMap<t_key,t_value> map(InitialSizeGuess);
int failed_inserts = 1; int n = 0;
while(failed_inserts) {
  parallel_reduce(N, KOKKOS_LAMBDA (int i, int& my_failed) {
    if( is_inside(values(i))) {
      auto result = map.insert(values(i).x,values(i));
      if(result.failed()) my_failed++;
    }
  },failed_inserts);
  if(failed_inserts>0)
    map.rehash(1.2*(InitialSizeGuess+failed_inserts);
}
for(int i=0; i<map.capacity(); i++)
  if(map.valid_at(i)) filtered_vals(n++) = map.value_at(i);
```

DynRankView: Runtime Rank View

- Some Overhead compared to View
- Used to avoid templating for common functions

```
DynRankView<double> a("A",N0,N1,N2); // create rank 3 view
DynRankView<double, LayoutLeft> b("B",N0,N1); // rank 2 view
```

DualView: Utility for unstructured sychronization

- Utility to keep track of where data was changed
- Synchronize Function uses tracking information

```
typedef Kokkos::DualView<double**> t_view;
t_view a("A",N0,N1);
host_foo(a);
a.modify<t_view::t_host::memory_space>();
... // Other things with a
a.sync<t_view::t_dev::memory_space>();
parallel_for(N, KOKKOS_LAMDBA (int i) {
  // use a on device
});
```

Sorting

- Currently only bin based sorting O(N) algorithm
- Optimal for homogeneously distributed data
- Custom Binning: InBuilt 1D and 3D

```
typedef View<double*> t_view;
t_view s("S",N); // size of something
View<SomeValueType*> vals("V",N);
// fill s and vals
BinOp1D<t_view> binop(N/20,min,max); // Templated on KeyViewType
// Takes view of keys and binop as input
BinSort<t_view,BinOp1D<t_view> > sorter(s,binop);

sorter.create_permute_vector(); // Creates a permute vector
// Reorder views according to permutation vector
sorter.sort(s); sorter.sort(vals);

auto a = foo(); // Create some 1D view
sort(a); // Short cut to sort single view
```

Random Numbers:

- High Quality (equal or better Mersenne Twister)
- Statistics good within thread, and accross
- Threads can pull random numbers independently

```
// Create Random Number Generator Pool
Random_XorShift64_Pool<> rand_pool64(5374857);
View<double**[4]> vals("V",N,M);
parallel_for(N, KOKKOS_LAMBDA (int i) {
  // Get a generator from the pool (potentially expansive)
  auto gen = rand_pool64.get_state();
  for(int j = 0; j<M; j++) {
    vals(i,j,0) = gen.drand(); // [0,1)
    vals(i,j,1) = gen.drand(10.0); // [0,10)
    vals(i,j,2) = gen.drand(-1.0,1.0); // [-1,1)
    vals(i,j,3) = gen.normal(eps,sig); // Normal dist
  }
});
// Convenience
fill_random(vals,rand_pool64,-1.0,1.0);
```

Profiling Hooks

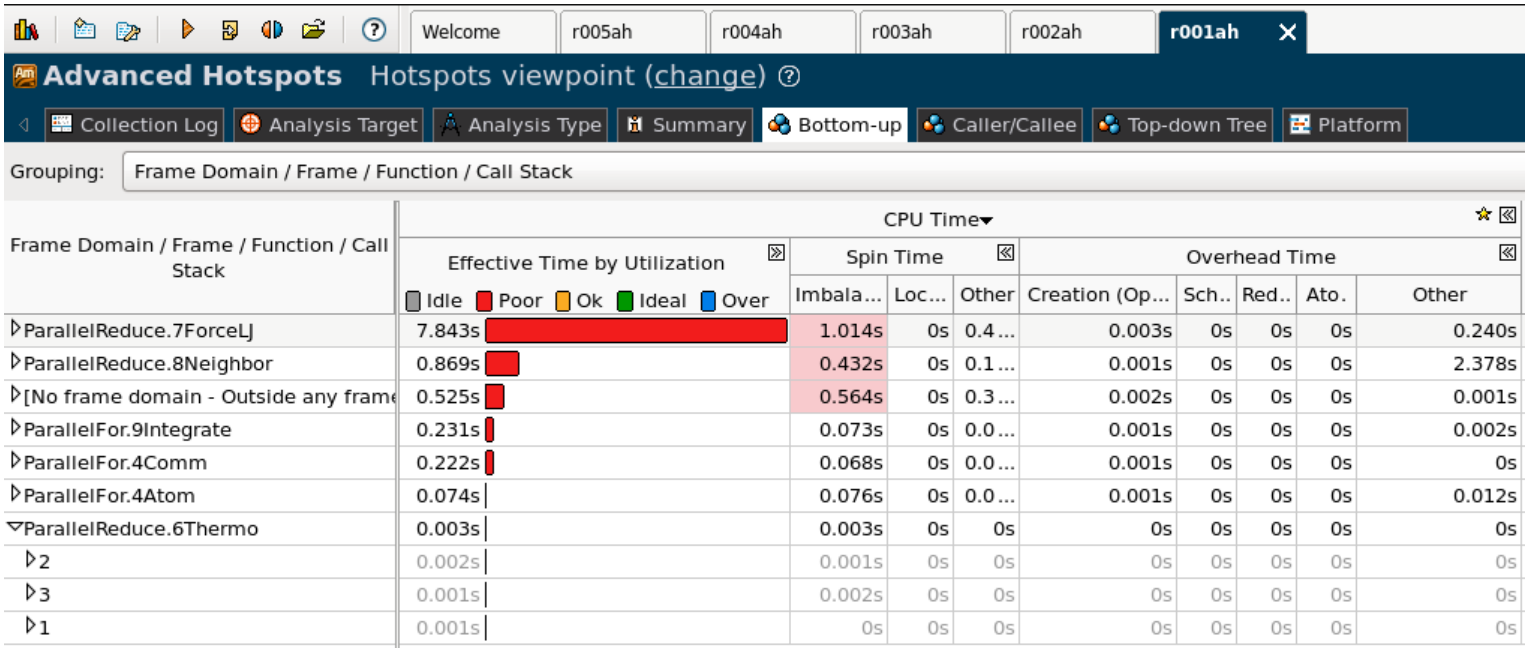
- By default Instrumentation of Kokkos codes is on
- Low cost (pointer comparison) if not used
- At runtime tool specified by env variables
- Custom Kernel Names will show up

Example Simple Kernel Timer

```
export KOKKOS_PROFILE_LIBRARY={PATH_TO_TOOLS}/kp_kernel_timer.so
./run_code
{PATH_TO_TOOLS}/kp_reader HOSTNAME_PROCESS.dat
     AXPB  9.61008  50  0.19220  63.870  59.312
      Dot  5.43792  51  0.10663  36.130  33.551
-----------------------------------------------------------
Summary:
Total Execution Time (incl. Kokkos + Non-Kokkos: 16.20268 s
Total Time in Kokkos kernels:                    15.04626 s
   -> Time outside Kokkos kernels:                1.15642 s
   -> Percentage in Kokkos kernels:              92.86 %
Total Calls to Kokkos Kernels:                   101
```

Example VTune Connector

- Vtune has issues understanding the template layers: use connector to provide information
- Kernels will be marked as regions (domains/frames) in vtune
- Use to filter in and/or limit profiling to specific kernels



KokkosKernels

- Provide Linear Algebra, Tensor and Graph Kernels for Apps
- Prioritization based on application needs
- Interface takes Kokkos Views:
  - no explicit specification of dimensions and strides necessary
  - Memory space aware
- Currently full device kernels
- Future: ThreadTeam and single Thread Kernels
- Future: Complete interface to vendor libraries
- Future Maybe: C-Interface

This is just starting: help to get complete coverage is highly welcome