Photos placed in horizontal position with even amount of white space between photos and header

Photos placed in horizontal position with even amount of white space between photos and header

Sandia National Laboratories

Exceptional service in the national interest

Kokkos update: Memory Spaces, Execution Spaces, Execution Policies, Defaults, and C++11

Carter Edwards and Christian Trott

Trilinos User Group

October 30, 2014

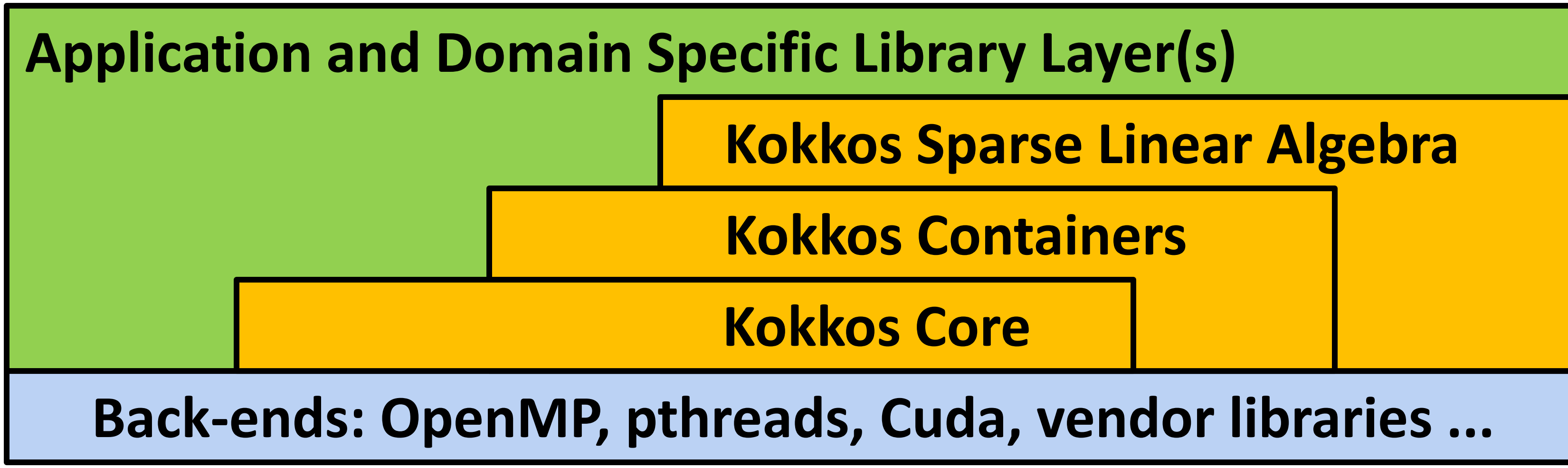
SAND2014-19215 PE

U.S. DEPARTMENT OF ENERGY

NNSA National Nuclear Security Administration

Sandia National Laboratories is a multi-program 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. SAND NO. 2011-XXXXP

Kokkos: A Layered Collection of Libraries



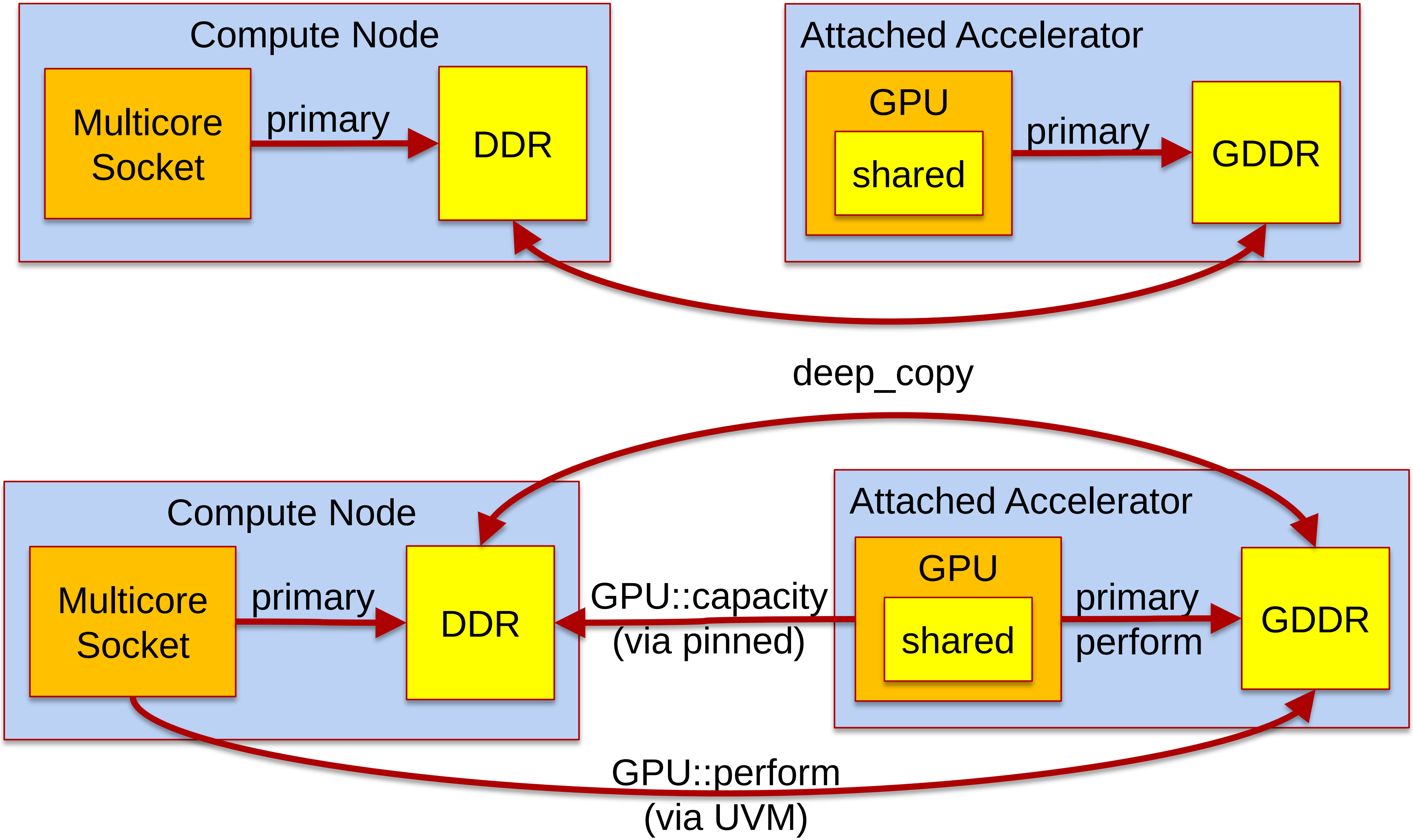
- **C++1998 standard (everyone supports except IBM's xlC)**
- **C++2011 offers concise & convenient lambda syntax**
  - **Vendors catching up to C++11 language compliance**
- **Concern: Can applications move to C++2011 ?**
  - **Can just those applications moving to MPI + X also move to C++2011?**
- **C++2017 working on Kokkos Core -like thread parallel capability**

Kokkos: Spaces and Execution Policies

- Execution Space : where functions execute
  - Encapsulates hardware resources; e.g., cores, hyperthreads, vector units, ...
- Memory Space : where data resides
  - ➢ AND what execution space can access that data
  - Also differentiated by access performance; e.g., latency & bandwidth
- Execution Policy : how (and where) a function is executed
  - Identifies an execution space
  - E.g., data parallel range : concurrently call function(i) for i = 0 .. N-1
  - E.g., task parallel : concurrently call { tasks }
- Compose parallel pattern, execution policy, and functions
  - Patterns: parallel_for, parallel_reduce, parallel_scan, task_parallel, ...
  - User's function is a C++ functor or C++11 lambda

```
parallel_for( Policy<Space>(...), Functor(...) );
```

Examples of Execution and Memory Spaces



Kokkos: Execution Spaces

- Execution Space *Instance*
  - Encapsulate (preferably allocable) hardware execution resources
  - Functions may execute concurrently on those resources
  - Degree of potential concurrency (cores, hyperthreads) determined at runtime
  - Number of execution space instances determined at runtime
- Execution Space *Type* (e.g., CPU, Xeon Phi, GPU)
  - Functions compiled to execute on a type of execution space
  - These types determined at configure/compile time
- Host's Serial Space
  - The main process and its functions execute in the host's Serial Space
  - One type, one instance, and is serial (potential concurrency == 1)
- Execution Space *Default* : one instance of one type
  - Configure/build with one type – it is the default
  - Initialize with one instance – it is the default
  - E.g., Kokkos::Threads, Kokkos::OpenMP, Kokkos::Cuda

Kokkos: Memory Spaces

- Memory Space *Types* (GDDR, DDR, NVRAM, Scratchpad)
  - The *type* of memory is defined with respect to an execution space type
  - Primary: (default) space with allocable memory (e.g., can malloc/free)
    - Performant : best performing space (e.g., GPU's GDDR)
    - Capacity : largest capacity space (e.g., DDR)
    - Contemporary system: Primary == Performant == Capacity
  - Scratch : non-allocable *and* maximum performance
  - Persistent : usage can persist between process executions (e.g., NVRAM)
- Memory Space *Instance*
  - Accessibility and performance relationship with execution space
  - Directly addressable by functions in that execution space
  - Contiguous range of addresses
- Memory Space *Default*
  - Default execution spaces' primary memory space

Execution / Memory Space Relationship

- **(Execution Space , Memory Space , Memory Access Traits)**
  - **Accessibility : functions can/cannot access memory space**
  - **Readable / Writeable / Allocable**
    - **E.g., GPU performant memory using texture cache is read-only**
  - **Expectations for performance**
  - **Expectations for capacity**

- **Memory Access Traits (extension point)**
  - **examples: read-only, volatile/atomic, random, streaming, ...**
  - **Automatically convert between Kokkos::Views with same space but different memory access traits**
  - ➢ **Default is simple readable/writeable – no special traits**

Kokkos::View, Spaces, and Defaults

- **typedef View< ArrayType , Layout , Space , Traits > view_type ;**
  - **Space** is either memory space or execution space
    - Execution space has a default memory space
    - Memory space has a default execution space
  - Omit **Traits** : no special compile-time defined access traits
  - Omit **Space** : use default execution space
  - Omit **Layout** : use space's default layout
  - **default everything: View< ArrayType >**

- **View< double**[3][8] > : ArrayType == double**[3][8]**
  - Four dimensional array of value type 'double'
  - Dimensions are [N][M][3][8]
  - N and M are runtime defined dimensions

Kokkos::View Construction and Data Access

- **View<double**[3][8], Space> a(*spec*,N,M);**
  - "Spec" for allocating memory or wrapping user-managed memory
  - Allocating memory, spec is
    - ViewAllocate(label = ""), std::string("label"), or "label"
    - ViewAllocateWithoutInitializing(label = "")
    - Dimensions may have hidden padded for memory alignment
    - Label is only used for error and warning messages, need not be unique
    - Allocation, by default, initializes data via 'parallel_for'
  - Wrapping user-managed, spec is a <u>pointer</u> (no label)
    - Dimensions are taken as-is, are never padded for memory alignment
    - <u>Trusting</u> that the user's memory spans the dimensions
- **Data access: a(i,j,k,l)**
  - Array layout deduced from 'Space' or 'Layout' template argument
  - Optional array bounds checking for debugging

Kokkos::View Internal Reference Counting

- View semantics with internal reference counting
  - View<double**[3][8],Space> b = a ; // SHALLOW copy
  - Both 'b' and 'a' reference the same allocated memory
  - Memory deallocated when last referencing view is destroyed
- Wrapped user-managed memory is never reference counted
- View< ... , Traits = MemoryUnmanaged >
  - Do not reference count Views with this trait
  - Cannot allocate non-reference counted views
  - Use cases: temp subview of an allocated view, wrapping user's memory
  - Trusting that temporary subview does not outlive the allocated view
- 'Const-ness' of views and viewed data
  - View<const double **[3][8],Space> c = a ; // OK, view to const array
  - const View<double**[3][8],Space> d = c ; // ERROR, non-const view of const

Deep Copy and "Mirror" Semantics

- deep_copy(destination_view , source_view);
  - Copy array data of 'source_view' to array data of 'destination_view'
  - Kokkos policy: never hide an expensive deep copy operation
  - Only deep copy when explicitly instructed by the user
- Avoid expensive permutation of data due to different layouts
  - Mirror the dimensions and <u>layout</u> in Host's memory space

```
typedef class View<...,Space> MyViewType ;
MyViewType a("a",...);
MyViewType::HostMirror a_h = create_mirror( a );
deep_copy( a , a_h ); deep_copy( a_h , a );
```

- Avoid unnecessary deep-copy

```
MyViewType::HostMirror a_h = create_mirror_view( a );
```

  - If Space (might be an execution space) uses Host memory space then 'a_h' is simply a view of 'a' and deep_copy is a no-op

Subview : View of a sub-array

SrcViewType src_view(...);

DstViewType dst_view = subview<DstViewType>(src_view, ...*args*)

- ***...args*** : list of indices or ranges of indices

- **Challenging capability due to polymorphic array Layout**
  - View's are strongly typed: View<ArrayType,Layout,Traits>
  - Compatibility constraints among DstViewType, SrcViewType, ***...args***
    - 'const-ness' and other memory access traits
    - number of dimensions (rank of array)
    - runtime and compile-time dimensions
    - destination layout can accommodate when stride != dimension
  - Performance of deep_copy between subviews
- **Using C++11 'auto' type would help address this challenge**
  - auto dst_view = subview(src_view , ...*args*);
  - Let implementation choose a compatible view type
  - Caution: user will not have a priori knowledge of this type

Execution Policy : how functions are executed

```
pattern( Policy , Function );
```

- Execution policies (an extension point)
  - RangePolicy<Space,ArgTag,IntegerType>(begin , end)
  - TeamPolicy<Space,ArgTag>(#teams , #thread/team)
  - TaskPolicy<...> : experimental for Kokkos/Qthreads LDRD
  - TeamVectorPolicy<...> : experimental for hybrid thread-vector parallel
- Policies have defaults for all template arguments
- Function interface depends upon policy and pattern
  - void operator()(ArgTag , Policy::member_type , ...*args*) const ;
  - void operator()(Policy::member_type , ...*args*) const ; // ArgTag == void
  - RangePolicy::member_type == IntegerType iteration space
  - TeamPolicy::member_type has league-of-teams iteration space
  - *...args* depends upon pattern

Execution Policy : how functions are executed

```
pattern( Policy , Function );
```

- Example with defaults and C++11 lambda (near-future capability)

```
parallel_for( N , KOKKOS_LAMBDA( int i ) { /* function body */ } );
```

  - Integral N “policy” → RangePolicy<DefaultExecutionSpace,void,int>(0,N)
  - Call function in parallel with i = 0 .. N-1
- Example: parallel_for(TeamPolicy< Space > , Functor);
  - void operator()(TeamPolicy<Space>::member_type member) const ;
  - league-of-teams-of-threads
    - member.league_size() == number of teams
    - member.league_rank() == which team is this within the league
    - member.team_size() == number of threads within a team
    - member.team_rank() == which thread is this within this team
  - Threads within a team are guaranteed concurrent, may not be synchronous
  - Intra-team collective operations: member.team_barrier(), member.team_reduce(...), member.team_scan(...)
  - Intra-team shared scratch memory

Parallel Patterns Function Interface

- parallel_for(Policy , F)
  - void F::operator()(Policy::member_type) const ; // no ...*args*
- parallel_reduce(Policy , F)
  - void F::operator()(Policy::member_type , value_type & update) const ;
  - function contributes to reduction through 'update' argument
- parallel_scan(Policy , F)

void F::operator()(Policy::member_type, value_type & update, bool final) const ;

  - Parallel scan is a multi-pass operation
  - Each pass must contribute the exactly the same to 'update'
  - if (final) then 'update' is the parallel prefix sum value
- Inter-thread reduction functions (have defaults)
  - functor::init(value_type & update) const ; // new(& update) value_type();
  - functor::join(volatile value_type & update ,
volatile const value_type & in) const ; // update += in ;

Why ArgTag in Policy< Space , ArgTag >

- **Allow one functor to have multiple parallel work functions**
  - parallel_for(RangePolicy<Space,TagA>(0,N) , my_functor);
    - calls: my_functor::operator()(const TagA & , int i);
  - parallel_for(RangePolicy<Space,TagB>(0,N) , my_functor);
    - calls: my_functor::operator()(const TagB & , int i);
  - "ArgTag" because named member function cannot be used

- **Motivations**
  - Algorithm (class) with multiple parallel passes using the same data
  - Work functions can share member data and member functions
  - Common need in LAMMPS
    - allow LAMMPS to remove clunky "wrapper functor" pattern

TeamVectorPolicy ← highly experimental !

- Three level hierarchy of parallelism: league, team, vector
- Thread of vector *lanes* (experimental)
  - Instructions applied lock-step in each lane
  - Vector collective operations: reduce, single
- Team of threads (current capability)
  - Each thread independently executes instructions in a shared function
  - Team collective operations: barrier, reduce, scan
  - Threads within a team share low-level resources
    - hyperthreads, L1 cache, transient scratch memory, ...
- League of teams of threads (current capability)
  - NO synchronization across teams
- Mapping onto GPU
  - Vector lane = GPU thread
  - Thread = GPU warp
  - Team = GPU block

TeamVectorPolicy ← highly experimental !

- **Example using C++11 lambdas**

```
typedef TeamVectorPolicy<Space>::member_type member_type ;
void operator()( const member_type & member ) const
{
  // team collaboratively performs a parallel_for
  member.team_par_for( N , [&]( const int j ) { // j = 0..N-1
    double sum ;
    // each “thread” performs a reduction in a vector loop
    member.vector_par_reduce( M , [&]( const int k , double & val ){
      val += /* contribute from each lane */ ;
    }, sum );
    // One vector lane of the thread performs an operation
    member.vector_single([&]() { atomic_fetch_add(&global(),sum); }
  });
}
```

Kokkos/Qthread LDRD: Task Parallelism

- TaskPolicy< Space > and Future< type , Space >
  - Task policy object for a group of potentially concurrent tasks

```
TaskPolicy<> manager( ... ); // default Space
Future<type> fa = manager.spawn( functor_a ); // single-thread task
Future<type> fb = manager.spawn( functor_b ); // may be concurrent
```

  - Tasks may be data parallel via data parallel pattern and policy

```
Future<>       fc = manager.foreach(RangePolicy(0,N)).spawn( functor_c );
Future<type> fd = manager.reduce(TeamPolicy(N,M)).spawn( functor_d );
wait( tm ); // Host can wait for all tasks to complete
```

  - Destruction of task manager object waits for concurrent tasks to complete
- Task Manager : TaskPolicy< Space = Qthread >
  - Defines a scope for a collection of potentially concurrent tasks
  - Have configuration options for task management and scheduling
  - Manage resources for scheduling queue

Kokkos/Qthread LDRD: Task Parallelism

- **Tasks may have execution dependences**
  - **Start a task only after other tasks have completed**

```
Future<> array_of_dep[ M ] = { /* futures for other tasks */ };
```

  - **Single threaded task:**

```
Future<> fx = manager.spawn( functor_x , array_of_dep , M );
```

  - **Tasks and their dependences define a directed acyclic graph (dag)**
- **Challenge: A GPU task cannot 'wait' on dependences**
  - **An executing GPU task cannot be suspended – waiting blocks a processor**
  - **Other future light-weight core architecture may not be able to block as well**
  - **A task may spawn nested tasks and need to wait for their completion**
  - **Solution: 'respawn' the task with new dependences**

```
manager.respawn( this , array_of_dep , M );
return ; // 'this' returns to be called after new dependences complete
```

Conclusion : Kokkos Strategy

- **Evolves from "pure research" to "production growth"**
  - **Core abstractions and API stabilizes, as per today's presentation**
  - **Move core of Kokkos from Trilinos to github.com**
- **Tutorial Examples and Mini-Applications using Kokkos**
  - **How to use Kokkos via examples**
  - **How to design and implement thread-scalable algorithms via mini-apps**
- **SON Website: software.sandia.gov/drupal/kokkos**
- **Tpetra and LAMMPS are migrating**
- **Long Term Strategy: C++17 or C++21 instead of Kokkos**
  - **ISO C++ Committee working to incorporate thread parallelism into standard**
  - **I am a voting member on this committee (several week-long mtgs/year)**
  - **Steer Kokkos *and* influence C++ standard → convergence**