SAND2014-15946PE

Kokkos Path Forward: Spaces, Policies, Defaults, C++11, and Tasks

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

Kokkos Path Forward Review; July 16, 2014

SAND2014-####P (Unlimited Release)

Sandia National Laboratories

Exceptional service in the national interest

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

Execution Space

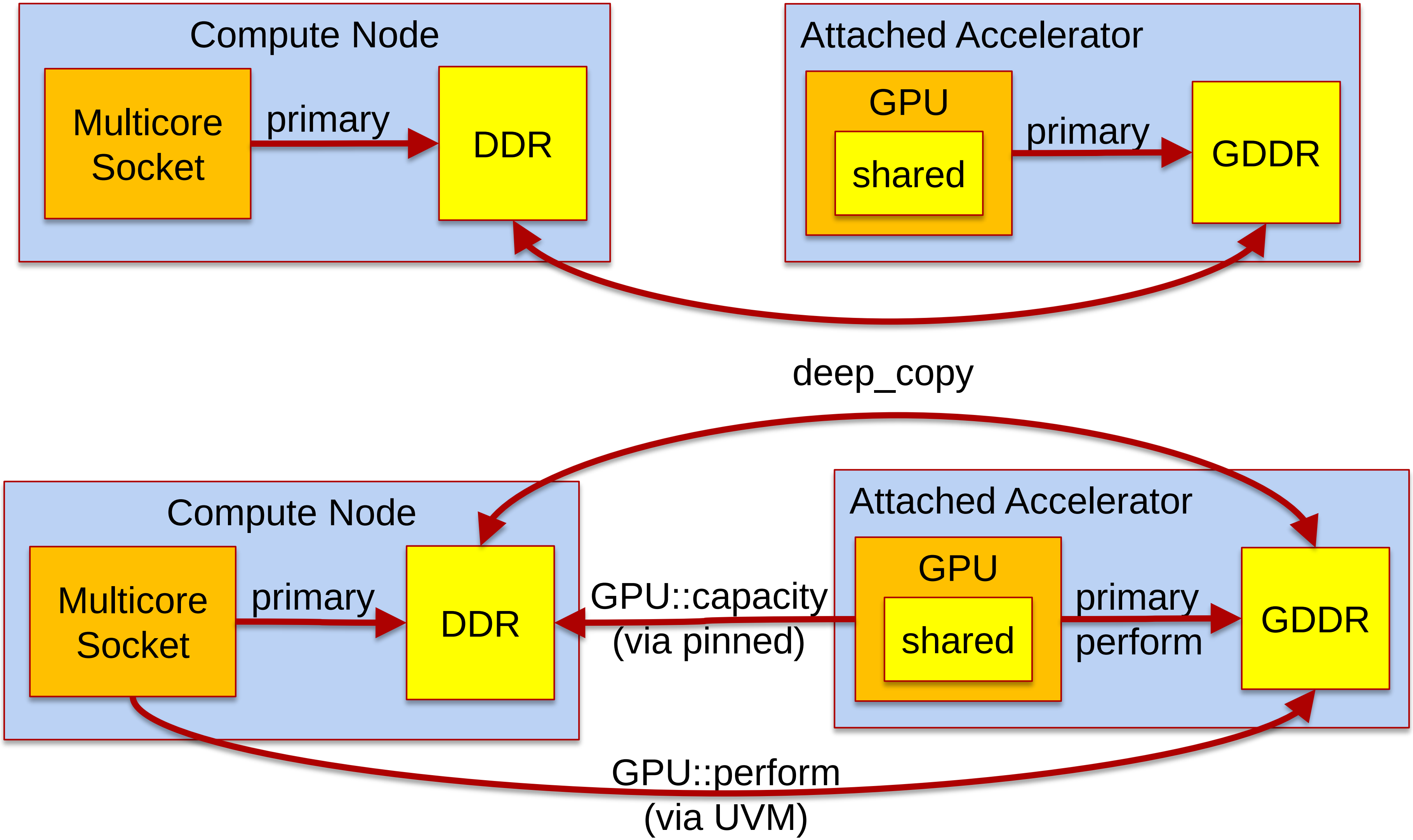
- Execution Space *Instance*
  - Hardware resources (e.g., cores, hyperthreads) in which functions execute
  - Functions may execute concurrently on those resources
  - Concurrently executing functions have coherent view to memory
  - Degree of potential concurrency determined at runtime
  - Number of execution space instances determined at runtime
- Execution Space *Type*
  - Functions compiled to execute on an instance of a specified type
  - Types determined at configure/compile time
- Host Space
  - The main process and its functions execute in the Host Space
  - One type, one instance, and is serial (potential concurrency == 1)

Memory Spaces

- Memory Space *Types*
  - The *type* of memory is defined with respect to an execution space type
  - Anticipated types, identified by their dominant usage
  - Primary: (default) space with allocable memory (e.g., can malloc/free)
    - Performant : best performing space (e.g., 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*
  - Has relationship with execution space instances (more later)
  - Directly addressable by functions in that execution space
  - Contiguous range of addresses
  - Has bounded capacity

Examples of Execution and Memory Spaces



Default Execution and Memory Spaces for Simple Applications & Libraries

- **Default Execution Space**
  - One type selected at configure/build
  - One instance of that type selected at initialization
  - When an execution space is not specified the default is assumed
- **Execution Space's Default (Primary) Memory Space**
  - Execution space instance has <u>one</u> default allocable memory space instance
  - Allocable memory space has <u>one</u> preferred execution space instance
- **Omission Assumes Default**
  - Omitting an execution space assumes the default
  - Given an execution space, omitting a memory space assumes the default
  - Omitting a memory space assumes the default execution & memory space

Execution / Memory Space Relationships

- (Execution Space , Memory Space , Memory Access Traits)
  - Accessibility : functions can/cannot access memory space
    - E.g., Host functions can never access GPU scratch memory
    - E.g., GPU functions can access Host capacity memory only if it is pinned
    - E.g., Host functions can access GPU performant memory only if it is UVM
  - Readable / Writeable
    - E.g., GPU performant memory using texture cache is read-only
  - Bandwidth : potential rate at which concurrent instructions can read or write
  - Capacity for views to (allocable) data
- Memory Access Traits (extension point) potential examples:
  - read-only, write-only, volatile/atomic, random, streaming, ...
  - Converting between “views” with same space and different traits
  - ➢ Default is simple readable/writeable – no special traits
- Future opportunity
  - Execution space accesses remote memory space (similar to MPI 1-sided)

Views and Defaults

- typedef View< ArrayType , Layout , Space , Traits > view_type ;
  - Omit **Traits** : no special compile-time defined access traits
  - Omit **Space** : default execution space's default memory space
  - Omit **Layout** : allocable memory space's default layout
  - default everything: **View< ArrayType >**

- view_type a(optional_traits , N0 , N1 , ...);
  - **optional_traits** : a collection of optional runtime defined traits
  - label trait : string used in error and warning messages, default none
  - initialize trait : default parallel_for(N0,[=](int i){ a(i,...) = 0 ; })
    - Default uses memory space's preferred execution space with static scheduling
    - Common override is to not initialize after allocating

Execution Policy

- **How** Potentially Concurrent Functions are Executed
  - Where : in what execution space (instance & type)
  - Parallel Work: current capabilities [0..N) or (#teams, #thread/team)
  - Scheduling : currently static scheduling of data parallel work
  - Map work function calls onto resources of the execution space
    - E.g., contiguous spans of [0..N) to a CPU thread for contiguous access pattern
    - E.g., strided subsets of [0..N) to GPU threads for coalesced access pattern
- Compose Pattern & Policy : parallel_for(policy , functor);
  - Policy::execution_space to replace Functor::device_type
  - Allows functor to be a C++11 lambda (more on this later)
- Default Policy and Space for Simple Functors
  - Policy 'size_t N' is [0..N) with static scheduling and default execution space
  - E.g., parallel_for(N , [=](int i) { /* lambda-function body */ });

Execution Policies, Patterns, and Defaults

- **Patterns: parallel_for, parallel_reduce, parallel_scan**
- **parallel_*pattern*(policy , functor);**
  - **Execute on policy's execution space according to policy's scheduling**
  - **functor API requirements defined by pattern and policy**
  - **functor API omissions have defaults**
- **parallel_reduce functor API requirements and defaults**
  - **functor::init(value_type & update); // { new(& update) value_type(); }**
  - **functor::join(volatile value_type & update ,**
    volatile const value_type & in) const ; // { update += in ; }
  - **functor::final(value_type & update) const ; // {;}**
- **Dot product becomes simple with C++11 lambda and defaults**

```
double dot( View<double*> x , View<double*> y ) {
  double d = 0 ;
  parallel_reduce( x.dimension_0() , [=](int i, double & v) { v += x(i) * y(i); } , d );
  return d ;
}
```

Execution Policy

Sandia National Laboratories

- Policy calls functor’s work function in parallel
  - PolicyType<ExecSpace>::index_type // data parallel work index type

    void FunctorType::operator()(PolicyType<...>::index_type) const ;
- Range policy example
  - parallel_for(Range<ExecSpace>(0,N) , functor);

    void FunctorType::operator()(integer_type i) const ;
- Thread team policy example
  - parallel_for(Team<ExecSpace>(#teams,thread/team) , functor);

    void FunctorType::operator()(Team<ExecSpace>::index_type team) const ;
  - Replaces “device” interface
- Extension point for new policies
  - Multi-indices [0..M)x[0..N), index sets, ...
  - Static partitioning with chunk bounds, work stealing, ...

Execution Policy, multi-function Functors

- Allow functors to have multiple parallel work functions
  - typedef PolicyType< ExecSpace , TagType > p_type ;
  - parallel_*pattern*(p_type(...) , functor);

    void FunctorType::operator()(const TagType &, p_type::index_type) const ;
  - Parallel work functions differentiated by 'TagType'
    - TagType used instead of method name
- Motivations
  - Algorithm with multiple parallel passes using the same data
    - miniFENL sparse matrix graph construction from FEM connectivity
  - Common need in LAMMPS, allow LAMMPS to remove "wrapper functors"
- Examples:
  - parallel_for(Range<ExecSpace,TagType>(0,N) , functor);
  - parallel_for(Team<ExecSpace,TagType>(#teams,thread/team) , functor);

Execution Policy for Task Parallelism

- Kokkos/Qthreads LDRD
- TaskManager< ExecSpace > execution policy
  - Policy object shared by potentially concurrent tasks

```
TaskManager<...> tm( exec_space , ... );
Future<> fa = spawn( tm , task_functor_a ); // single-thread task
Future<> fb = spawn( tm , task_functor_b );
```

  - Tasks may be data parallel

```
Future<> fc = spawn_for( tm.range(0..N) , functor_c );
Future<value_type> fd = spawn_reduce( tm.team(N,M) , functor_d );
wait( tm ); // wait for all tasks to complete
```

  - Destruction of task manager object waits for concurrent tasks to complete
- Task Managers
  - Define a scope for a collection of potentially concurrent tasks
  - Have configuration options for task management and scheduling
  - Manage resources for scheduling queue

Execution Policy for Task Parallelism

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

    `Future<> array_of_dep[ M ] = { /* future for other specified tasks */ };`
  - **Single threaded task:**

    `Future<> fx = spawn( tm.depend(M,array_of_dep) , task_functor_x );`
  - **Data parallel task:**

    `spawn_for( tm.depend(M,array_of_dep).range(0..N) , task_functor_y );`
  - **Tasks and dependences define a directed acyclic graph (dag)**

- **At most one active task manager on an execution space**
  - **Well-defined scope and lifetime for collection of potentially current tasks**
  - **Don't consume resources when not in use**