Kokkos Spaces: Expressing Locality in Heterogeneous Node Architectures

Third Workshop on Programming Abstractions for Data Locality
October 24-26, 2016

H. Carter Edwards

SAND2016-10269 C
Unclassified Unlimited Release (UUR)
What is Kokkos?

Kokkos: performance portability for C++ applications

κόκκος (Greek): “granule” or “grain” ; \textit{like grains of sand on a beach}
Abstractions: Patterns, Policies, and Spaces

- **Parallel Pattern** of user’s computations
  - `parallel_for`, `parallel_reduce`, `parallel_scan`, task-graph, ...
- **Execution Policy** tells *how* user computation will be executed
  - Static scheduling, dynamic scheduling, thread-teams, ...
- **Execution Space** tells *where* user computations will execute
  - Which cores, numa region, GPU, ...
- **Memory Space** tells *where* user data resides
  - Host memory, GPU memory, high bandwidth memory, ...
- **Layout (policy)** tells *how* user data is laid out in memory
  - Row-major, column-major, array-of-struct, struct-of-array ...
- **Differentiating:** Layout and Memory Space
  - Versus other programming models (OpenMP, OpenACC, ...)
  - Critical for performance portability ...
Kokkos currently provides Space *Singletons*, need Space *Instances* to express Locality

- Both Execution and Memory Spaces
- **Singleton space is like using MPI_COMM_WORLD**
  - Single implicit environment, for each type
  - One CPU space, one GPU space
- **Instance spaces are like using MPI_Comm objects**
  - Subsets of the compute node’s set of execution and memory resources
  - Application has *handles* to instances
  - Application specifies which instance to use in a computation
- **Setting up instances is like using MPI_Comm_split ...**
  - How to form subsets of compute node’ resources?
  - Application needs to reason about topology of those resources
Use Cases for Space Instances

- Use multiple GPUs within a single process
- Use multiple CUDA streams on a GPU
  - Enable overlap of deep copy and functor execution
  - Asynchronously launch multiple independent (large) parallel operations
- Use independent groups of CPU cores for asynchronous work
  - Asynchronously launch multiple independent (large) parallel operations
  - Partition cores of manycore node, analogous to MPI_Comm_split
  - Group by NUMA affinity to reduce memory movement
- Required for inter-node asynchronous many-task (AMT)
  - AMT scheduler launches large tasks with internal Kokkos parallelism

Challenge: execution resource management
- Choosing GPU
- Partitioning and selecting CPU cores
- Execution resources <-> memory resource affinity
Current Kokkos Memory Space Singletons

struct MemorySpace {
  typedef MemorySpace memory_space;
  typedef /* preferred */ execution_space;
  void * allocate( size_t ) const;
  void deallocate( void * , size_t ) const;
  MemorySpace(); // access to singleton
};

Kokkos::HostSpace
Kokkos::CudaSpace
Kokkos::CudaUVMSpace
Kokkos::CudaHostPinnedSpace
Kokkos::Experimental::HBWSpace
Current Kokkos Execution Space Singletons

```
struct ExecutionSpace {
  typedef ExecutionSpace execution_space ;
  typedef /* preferred */ memory_space ;
  typedef /* preferred */ array_layout ;
  typedef /* ---------- */ scratch_memory_space ;
  typedef /* preferred */ size_type ;
  static void fence();
  static int is_initialized();
  static void print_configuration( std::ostream & , bool );
  static void finalize();
  static void initialize( /* type dependent arguments */ );
};
```

Kokkos::Serial
Kokkos::Threads
Kokkos::OpenMP
Kokkos::Cuda
Kokkos::Qthread
Using Current Kokkos Space Singletons

```cpp
parallel_pattern( Policy<ExecSpace>(args), functor );
```

- Execute functor on `ExecSpace` according to pattern and policy
- Functor can be a properly declared lambda
- `Pattern` is ‘for’, ‘reduce’, and ‘scan’
- `Policy` is ‘RangePolicy’ and ‘TeamPolicy’
- `args` is `[begin,end)` index range or `(league,team)` size or other properties

```cpp
View< ArraySpec , Space > a(args);
```

- View of a multidimensional array allocated in `Space::memory_space`
- Allocation may execute element constructor in `Space::execution_space`
- Deallocation may execute element destructor in `Space::execution_space`
- `args` is a label for allocation and multidimensional array runtime dimensions
Using TBD Kokkos Space Instances

parallel_pattern( policy(space,args...), functor );

- Execute functor on space instance according to pattern and policy
- If space is omitted then use default
- args are policy properties

View<ArraySpec,SPACE> a(view_alloc(space,args...),dims);

- SPACE type still required to verify accessibility; e.g., CPU can’t access GPU
- space is instance of SPACE type
- If space is omitted then use default
- View of multidimensional array allocated in space.memory_space()
- Allocation may execute element constructor in space.execution_space()
- Deallocation may execute element destructor in space.execution_space()
- args are allocation and construction properties
Using TBD Kokkos Space Instances

deep_copy( exec_space , destination , source );

- Asynchronous deep_copy only synchronizing on exec_space instance
- E.g., exec_space could be a CUDA stream on a particular GPU

verify_accessible( exec_space , memory_space )

- Can code executing in exec_space access memory in memory_space ?
- Used to catch memory access errors, instead of seg-fault

- **Execution-space / Memory-space relationships**
  - Accessible at all; e.g., CPU cannot access GPU memory without UVM
  - Unloaded bandwidth
  - Unloaded latency
Essential Innovation: Resource Management for Execution and Memory Spaces

- “Software Requirements for ATDM On-Node Resource Management” SAND2016-6357; Olivier, Pedretti, and Brightwell
  - Portable representation of node architecture topology (e.g., hwloc)
    - execution and memory resources
    - and their locality relationships
  - Maintain an inventory of node resources
    - Present, available, assigned
    - Threads pinned to hardware location, blocks of memory with affinity
  - Assigning and Recovering Resource Allocations
    - “User” (e.g., Kokkos) requests and relinquishes resources

- Simple and intuitive abstraction and API
  - Provided by Kokkos to application developers
  - ‘hwloc’ is too low-level of an abstraction, but can leverage ideas
Abstraction: Resource Locality Tree
Borrowing from HWLOC Topology

- **Nodes in the Tree**
  - Locality groups; e.g., socket and numa
  - Architecture; e.g., CPU, GPU, memory
  - Homogeneous execution resource
  - Homogeneous memory resource
Measures

- **Tree coordinate implies qualitative locality measure**
  - \( \text{distance}( (0.0.0.0) , (0.0.0.1) ) < \text{distance}( (0.0.0.0) , (0.0.1.0) ) \)

- **Quantitative measures**
  - \( \text{potential\_execution\_concurrency}( \text{nodeX} ) \); e.g., number of hyperthreads
  - \( \text{potential\_memory\_allocation}( \text{nodeZ} ) \)
  - \( \text{unloaded\_bandwidth}( \text{nodeX} , \text{nodeZ} ) \)
  - \( \text{unloaded\_latency}( \text{nodeX} , \text{nodeZ} ) \)
  - ...

- **Properties**
  - Exclusive assignability; e.g., can request exclusive assignment of core
  - Shared assignability; e.g., can request shared assignment of memory
  - ...

Request / Relinquish / Availability

- **Request {exclusive, shared} assignment of resource**
  - Input: identifier for requestor; e.g., a string
  - Input: number of cores, placement, compact vs. strided
  - Input: if a new control thread
  - Output handle to resource: “execution space instance”

- **Relinquish assigned resource**
  - Reference counted handles

- **Resource availability**
  - All managed resources, assigned and unassigned
  - Resources currently unassigned
  - Resources’ current exclusive and shared assignments
TBD Instance-based Execution Spaces

```cpp
struct ExecutionSpace {
    typedef ExecutionSpace execution_space ;
    typedef /* preferred type */ memory_space ;
    typedef /* preferred */ array_layout ;
    typedef /* -------- */ scratch_memory_space ;
    typedef /* preferred */ size_type ;
    memory_space memory_space_instance() const ;
    void fence() const ;

    static ExecutionSpace request(); // default instance
    static ExecutionSpace request( ...args ); // new instance

    static int is_initialized();
    static void print_configuration( std::ostream & , bool );
    static void finalize();
    static void initialize( init_mode );
};
```

- Relationship of `request` and `init_mode` to resource management?
  - Allow, or not, partitioning of resources into instances
struct MemorySpace {
    typedef MemorySpace memory_space;
    typedef /* preferred type */ execution_space;
    // Already converted to ‘instance’ API:
    void * allocate( size_t ) const;
    void deallocate( void * , size_t ) const;
    // preferred instance:
    execution_space execution_space_instance() const;

    static MemorySpace request(); // default
    static MemorySpace request( ...args ); // specific
};

- **Relationship of request to resource management?**
Resources may be Architecture Specific

- **Memory**
  - Shared resource
  - Allocation and deallocation mechanisms
  - Allocation metrics
    - Amount allocated, number of allocations
    - For the entire resource, for a particular shared assignment

- **Execution**
  - Exclusive use of CPU core resource
  - Shared GPU resource; e.g., multiple CUDA streams on a GPU
  - Closure (function, lambda) execution mechanism
  - Execution metrics
    - Concurrency of entire resource
    - Number of shared assignments
    - Closure execution profiling
Workshop Discussion Questions

- Related Projects / Abstractions?
  - HWLOC
  - ISO/C++ Executors
  - ... ?

- Constructive Critique of Kokkos’ Approach / Abstractions
  - Experience with similar approaches? Lessons learned?
  - Recommendations? Advice?