Kokkos: Enabling Performance Portability of C++ Applications and Libraries across Manycore Architectures

ORNL / January 29, 2015

SAND2015-0409 O (Unlimited Release)
Increasingly Complex Manycore Architectures

¿ Performance Portable and Future Proof Codes?

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

**Execution Spaces**
- Throughput cores (GPU)
- Latency optimized cores (CPU)
- Processing in memory

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

**Programming models**
- GPU: CUDA-ish
- CPU: OpenMP
- PIM: ??
Vision for Heterogeneous Parallelism

- “MPI + X” Programming Model, separate concerns
  - Inter-node: MPI and domain specific libraries layered on MPI
  - Intra-node: Kokkos and domain specific libraries layered on Kokkos

- Intra-node parallelism concerns: heterogeneity & diversity
  - Execution spaces (CPU, GPU, PIM, ...) have diverse performance requirements
  - Memory spaces have diverse capabilities and performance characteristics
  - Vendors have diverse programming models for optimal utilization of their hardware

- Standardized performance portable programming model?
  - Via vendors’ (slow) negotiations: OpenMP, OpenACC, OpenCL, C++17
  - Vendors’ (biased) solutions: C++AMP, Thrust, CilkPlus, TBB, ArrayFire, ...
  - Researchers’ solutions: HPX, StarPU, Bolt, Charm++, ...

- Necessary condition: address execution & memory space diversity
  - SNL Computing Research Center’s Kokkos (C++ library) solution
  - Engagement with ISO C++ Standard committee to influence C++17
Kokkos: A Layered Collection of C++ Libraries

- Applications and Domain Libraries written in Standard C++
  - *Not* a language extension like OpenMP, OpenACC, OpenCL, CUDA, ...
  - Require C++1998 standard (supported everywhere except IBM’s old xIC)
  - Prefer C++2011 for its concise lambda syntax
    - Vendors are catching up to C++2011 language compliance

- Kokkos implemented with C++ template meta-programming
  - In *spirit* of TBB, Thrust & CUSP, C++AMP, ...

Diagram:
- Application and Domain Specific Library Layer
  - Sparse Linear Algebra (Trilinos)
  - Kokkos Containers
  - Kokkos Core
  - Back-ends: Cuda, OpenMP, pthreads, vendor libraries ...
Performance Portability Challenge:

Best (good) performance requires computations to implement architecture-specific memory access patterns

- **CPUs (and Xeon Phi)**
  - Core-data affinity: consistent NUMA access (first touch)
  - Array alignment for cache-lines and vector units
  - Hyperthreads’ cooperative use of L1 cache

- **GPUs**
  - Thread-data affinity: coalesced access with cache-line alignment
  - Temporal locality and special hardware (texture cache)

- **Array of Structures (AoS) vs. Structure of Arrays (SoA) dilemma**

  - This has been the _wrong_ concern

The right question: Abstractions for Performance Portability?
Kokkos Performance Portability Answer

Integrated mapping of thread parallel computations and multidimensional array data onto manycore spaces

- Kokkos maps users’ parallel computations to threads
  - Standard parallel programming model pattern; e.g., parallel-for
  - Users implement C++ functions or lambdas for their parallel loop bodies
  - Kokkos calls user’s code from architecture’s “hardware” threads

- Kokkos’ multidimensional array data structure *has a twist*
  - Layout mapping: multi-index \((i,j,k,\ldots)\) ↔ memory location
  - Kokkos chooses layout for architecture-specific memory access pattern
  - Layout changes are invisible to user code
    - IF user code honors Kokkos’ simple array API: \(a(i,j,k,\ldots)\)
  - Polymorphic multidimensional array layout

- ... and utilizes special hardware invisibly to users’ code
  - GPU texture cache to speed up read-only random access patterns
  - Atomic operations for thread safety
Projects leveraging Kokkos for MPI+X

Albany: a Trilinos-based finite element application code

Ice Sheet Modeling

Quantum Device Design

Computational Mechanics

Atmosphere Dynamics

Target Application for CAAR Proposal [Salinger et al.]

Kokkos integration in progress
Projects leveraging Kokkos for MPI+X and points-of-contact for MPI+X transition effort

- Trilinos / Tpetra: foundational data structures and kernels for sparse linear algebra; Mark Hoemmen, Christian Trott
- Trilinos / Stokhos: accelerating embedded UQ; Eric Phipps
- LAMMPS: molecular dynamics; Christian Trott
- ASCR Multiphysics MHD; Roger Pawlowski and Eric Cyr
- FASTMath SciDAC / Graph Algorithms: Siva Rajamanickam
- Zoltan / Graph Coloring: fast threaded graph coloring to identify independent sets of work for task parallelism; Erik Boman, Siva Rajamanickam
- EMPRESS and miniPIC: particle-in-cell; Matt Bettencourt
- miniAero: CFD finite element mini-application; Ken Franco
- miniContact: contact detection for solid mechanics; Glen Hansen
- SHIFT @ ORNL; Steve Hamilton
- Kokkos SNL/LDRD
  - Directed acyclic graph of internally data parallel (coarse-grain) tasks
  - sparse matrix factorization and graph algorithm mini-applications
Abstractions and Application Programmer Interface
Spaces, Policies, and Patterns

- **Execution Space**: where functions execute
  - Encapsulates hardware resources; e.g., cores, GPU, 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 user function is executed
  - E.g., data parallel range: concurrently call function(i) for i = [0..N)
  - User’s function is a C++ functor or C++11 lambda

- **Pattern**: parallel_for, parallel_reduce, parallel_scan, task, ...

- **Compose**: pattern + execution policy + user function; e.g.,
  ```cpp
  parallel_for( Policy<Space>, Function);
  ```
  - Execute *Function* in *Space* according to parallel_for *pattern* and *Policy*

- Extensible spaces, policies, and patterns (by expert developers)
Examples of Execution and Memory Spaces

- Multicore Socket
- DDR
- Attached Accelerator
- GDDR

- GPU::capacity (via pinned)
- GPU::perform (via UVM)

- primary
- shared

- deep_copy
Multidimensional Array View API (simple)

- `View< double**[3][8] , Space > a("a",N,M);`
  - Allocate array data in memory `Space` with dimensions `[N][M][3][8]`
    - Each * indicates a runtime supplied dimension
    - Proposing C++ standard adjustment to enable `View<double[ ][ ][3][8],Space>`
  - Kokkos chooses array layout appropriate for “Space”

- `a(i,j,k,l)`: User’s access to array data
  - Optional array bounds checking of indices in debug compile
  - “Space” accessibility enforced; e.g., GPU code cannot access CPU memory

- **View Semantics:** `View<double**[3][8],Space> b = a ;`
  - A shallow copy: ‘a’ and ‘b’ are pointers to the same allocated array data
  - Reference counting how many Views to the same data
  - When reference count == 0, automatically deallocates data
**View API: Deep Copy and Host Mirror**

- `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 **layout** in Host’s memory space
    ```cpp
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
  ```cpp
  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
View API (advanced)

- **View<ArrayType,Layout,Space,Attributes>**
  - **ArrayType**: scalar type, # runtime dimensions, compile-time dimensions
  - **Layout**: user can override Kokkos’ choice for layout
  - **Attributes**: user’s access intentions

- **Why manually specify Layout?**
  - Force compatibility with legacy code while incrementally porting
  - Optimize performance with exotic layout
    - **View<double**,Tile<8,8>,Space> m(“matrix”,N,N);**
    - Tiling layout hidden from user code  `m(i,j)`
  - A “plug in” extension point

- **Access intent attributes**
  - Turn off reference counting to wrap an legacy code’s array
  - Indicate const and random access to utilize GPU texture cache
    - **View< const double ***, Cuda, RandomAccess> b = a ;**
  - A “plug in” extension point
Subview : View of a sub-array

B = subview( A , ...range_and_index_argument_list... )

- Challenging capability due to polymorphic array Layout
  - Views are strongly typed: View<ArrayType,Layout,Space,Traits>
  - Compatibility constraints on B’s type, A’s type, and argument list
    - runtime and compile-time dimensions
    - number and type (range or index) of calling arguments
    - array layout
    - ‘const-ness’ and other memory access traits

- C++11 ‘auto’ capability simplifies use
  
  auto B = subview( A , ...range_and_index_argument_list... );

  - Let implementation choose a compatible View type
  - Caution: View’s data may become non-contiguous
Parallel Execution API (simple with C++11)

- AXPY example using C++11 lambda

```c++
parallel_for( N , KOKKOS_LAMBDA( int i )
    { y(i) = alpha * x(i) + y(i); }
);
```

- User functor via C++11 lambda expression
- Default execution space and range policy i = [0..N)
- Kokkos chooses which threads call function with each value of ‘i’

- DOT example using C++11 lambda

```c++
parallel_reduce( N , KOKKOS_LAMBDA( int i , double & value )
    { value += x(i) * y(i); }
    , result );
```

- Kokkos manages thread-local temporaries
- Kokkos manages scalable inter-thread reduction
Parallel Execution API

paralll_pattern( Policy<Space>, Function )

- User Function: C++11 Lambda or C++ Functor
  - Use a functor with non-trivial function bodies
    ```cpp
    struct UserAXPY {
      double alpha ; View<double*,Space> x , y ; // “calling arguments”
    KOKKOS_INLINE_FUNCTION
      void operator()( int i ) const { y(i) = alpha * x(i) + y(i); } // function
    };
    ```

- Multi-function User Functor capability
  - When more than one function shares the same “calling arguments”
  - Migration to Kokkos: incrementally add parallel functions to existing classes

- Execution Policy: flexibility & extensibility
  - RangePolicy : i = [0..N)
  - TeamPolicy : two-level parallelism with team collectives and shared memory
    - Portable access to Cuda thread grid, block, and shared memory
  - TaskPolicy : experimental using SNL’s Qthreads runtime
Atomic operations

atomic_exchange, atomic_compare_exchange_strong, atomic_fetch_add, atomic_fetch_or, atomic_fetch_and

- Thread-scalability of non-trivial algorithms and data structures
  - Essential for lock-free implementations
  - Concurrent summations to shared variables
    - E.g., finite element computations summing to shared nodes
  - Updating shared dynamic data structure
    - E.g., append to a shared array or insert into a shared map
- Portably map to compiler/hardware specific capabilities
  - GNU and CUDA extensions when available
  - Current: any 32bit or 64bit type, may use CAS-loop implementation
  - Future: any data type via “sharded lock” pattern
- ISO/C++ 2011 and 2014 standards not adequate for HPC
  - Proposal in for 2017 standard to address this gap
Performance Evaluation
Evaluate Performance Impact of Array Layout

- Molecular dynamics computational kernel in miniMD
- Simple Lennard Jones force model:
  \[ F_i = \sum_{j, r_{ij} < r_{cut}} 6\epsilon \left[ \left( \frac{\sigma}{r_{ij}} \right)^7 - 2 \left( \frac{\sigma}{r_{ij}} \right)^{13} \right] \]
- Atom neighbor list to avoid N² computations

```c
pos_i = pos(i);
for( jj = 0; jj < num_neighbors(i); jj++) {
  j = neighbors(i,jj);
  r_ij = pos_i - pos(j); //random read 3 floats
  if (|r_ij| < r_cut) f_i += 6*epsilon*((s/r_ij)^7 - 2*(s/r_ij)^13)
}
f(i) = f_i;
```

- Test Problem
  - 864k atoms, ~77 neighbors
  - 2D neighbor array
  - Different layouts CPU vs GPU
  - Random read ‘pos’ through GPU texture cache
  - Large performance loss with wrong array layout

![Graph showing performance comparison between correct and wrong array layouts for different processors](image)
Evaluate Performance Overhead of Abstraction

Kokkos competitive with native programming models

- MiniFE: finite element linear system iterative solver mini-app
- Compare to versions specialized for programming models
- Running on hardware testbeds

---

**MiniFE CG-Solve time for 200 iterations on 200^3 mesh**

- K20X
- IvyBridge
- SandyBridge
- XeonPhi B0
- XeonPhi C0
- IBM Power7+

- NVIDIA ELL
- NVIDIA CuSparse
- Kokkos
- TBB
- OpenMP
- Cilk+(1 Socket)
- MPI-Only
- OpenCL
Kokkos’ Unordered Map (a.k.a. Hash Map)

- **Thread scalable fill**
  - Lock-free implementation with minimal use of atomics
  - API deviates from C++11 unordered map
    - No on-the-fly allocation / reallocation
    - Index-based instead of iterator-based

- **Insert (fill) within a parallel reduce functor**
  - **Within the functor:** \{status, index\} = map.insert(key,value);
    - Status = success | existing | failed due to insufficient capacity
    - Parallel reduce the failed-count to resize the map
  - **Algorithmic use:**
    UnorderedMap<Key,Value,Space> map;
    do {
      map.rehash( capacity );
      capacity += ( nfailed = parallel_reduce( N , functor ) );
    } while( nfailed ); // should iterate at most twice
Unordered Map Performance Evaluation

- Parallel-for insert to 88% full with 16x redundant inserts
  - NW = number attempts to insert = Capacity * 88% * 16
  - Near – contiguous work indices [iw,iw+16) insert same keys
  - Far – strided work indices insert same keys

- Single Accelerator Performance Tests
  - NVidia Kepler K40X, 12Gbytes
  - Intel Xeon Phi (Knights Corner) COES2, 61 cores, 1.2 GHz, 16Gbytes
    - Limit use to 60 cores, 4 hyperthreads/core

- K40X dramatically better performance
- Xeon Phi implementation optimized using explicit non-caching prefetch
- Theory: due to cache coherency protocols and atomics’ performance
MiniFENL Proxy Application

- Solve nonlinear finite element problem via Newton iteration
  - Focus on construction and fill of sparse linear system
  - Thread safe, thread scalable, and performant algorithms
  - Evaluate thread-parallel capabilities and programming models
- Construct sparse linear system graph and coefficient arrays
  - Map finite element mesh connectivity to degree of freedom graph
  - Thread-scalable algorithm for graph construction
- Compute nonlinear residual and Jacobian
  - Thread-parallel finite element residual and Jacobian
  - Atomic-add to fill element coefficients into linear system
    - Atomic-add for thread safety, performance?
- Solve linear system for Newton iteration
Thread-Scalable Fill of Sparse Linear System

- MiniFENL: Newton iteration of FEM: $x_{n+1} = x_n - J^{-1}(x_n)r(x_n)$
- Fill sparse matrix via Scatter-Atomic-Add or Gather-Sum?

**Scatter-Atomic-Add**
- Simpler
- Less memory
- Slower HW atomic

**Gather-Sum**
- Bit-wise reproducibility

**Performance win?**
- Scatter-atomic-add
- ~equal Xeon PHI
- 40% faster Kepler GPU

- Pattern chosen

- Feedback to HW vendors: performant atomics
Thread-Scalable Sparse Matrix Construction

- MiniFENL: Construct sparse matrix graph from FEM connectivity
- Thread scalable algorithm for constructing a data structure
  1. Parallel-for: fill Kokkos lock-free unordered map with FEM node-node pairs
  2. Parallel-scan: sparse matrix rows’ column counts into row offsets
  3. Parallel-for: query unordered map to fill sparse matrix column-index array
  4. Parallel-for: sort rows’ column-index subarray

Pattern and tools generally applicable to construction and dynamic modification of data structures
Porting in Progress: Trilinos Suite

- Trilinos: SNL’s suite of equation solver libraries (and others)
  - Previously MPI-only parallel
  - Incremental refactoring to MPI+Kokkos parallel
- Tpetra: Trilinos’ core parallel sparse linear algebra library
  - Vectors, multi-vectors, sparse matrices, parallel data distribution maps
  - Fundamental operations: axpy, dot, matrix-vector multiply, ...
  - Templated on “scalar” type: float, double, automatic differentiation (AD), embedded uncertainty quantification (UQ), ...
- Port Tpetra to MPI+Kokkos, other libraries follow
  - On schedule to complete in Spring 2015
  - Use of NVIDIA’s unified virtual memory (UVM) expedited porting effort
- Embedded UQ already Kokkos-enabled through SNL/LDRD
  - Greater computational intensity leads to significant speed-ups compared to non-embedded UQ sampling algorithms
Porting in Progress: LAMMPS

- LAMMPS: molecular dynamics application
  - Fully MPI-only parallel with some (prototype) thread-parallel user packages
    - Architecture specific with redundantly implemented physics
  - Incrementally refactoring to MPI+Kokkos parallel
    - Goal: collapse redundantly implemented physics into “core” code base
  - MPI+Kokkos performing as well or better than thread-parallel user packages
Takeaways: MPI + Kokkos for hybrid parallel

- Performance portability across diverse manycore architectures
  - Compose: pattern + policy + function + polymorphic array layout
  - to obtain architecture-appropriate memory access patterns
  - AoS versus SoA dilemma is a non-issue, with the right abstractions
  - Extensibility of patterns, policies, spaces, and array layout abstractions
    => future proofing versus architectural evolution?

- Negligible performance overhead versus native implementation

- R&D now addressing more challenging algorithms
  - Dynamic data structures
  - Task-dag and hybrid task-data parallelism
  - Graph analytics algorithms

- Transition of legacy codes in progress

- Kokkos to be available via GitHub in FY15/Q3