Kokkos, a Manycore Device
Performance Portability Library
for C++ HPC Applications

H. Carter Edwards, Christian Trott,
Daniel Sunderland

Sandia National Laboratories

GPU TECHNOLOGY CONFERENCE 2014
MARCH 24-27, 2013 | SAN JOSE, CALIFORNIA

SAND2014-2317C (Unlimited Release)
Increasingly Complex Heterogeneous Future; ¿ Future Proof Performance Portable Code?

**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: ??
Outline

- What is Kokkos
  - Layered collection of C++ libraries
  - Thread parallel programming model that managed data access patterns

- Evaluation via mini-applications

- Refactoring legacy libraries and applications
  - CUDA UVM (unified virtual memory) in the critical path!

- Conclusion
Kokkos: A Layered Collection of Libraries

- Standard C++, Not a language extension
  - In spirit of Intel’s TBB, NVIDIA’s Thrust & CUSP, MS C++AMP, ...
  - Not a language extension: OpenMP, OpenACC, OpenCL, CUDA

- Uses C++ template meta-programming
  - Currently rely upon C++1998 standard (everywhere except IBM’s xlC)
  - Prefer to require C++2011 for lambda syntax
    - Need CUDA with C++2011 language compliance

Application & Library Domain Layer

- Kokkos Sparse Linear Algebra
- Kokkos Containers
- Kokkos Core

Back-ends: OpenMP, pthreads, Cuda, vendor libraries ...
Kokkos’ Layered Libraries

- **Core**
  - Multidimensional arrays and subarrays in memory spaces
  - `parallel_for`, `parallel_reduce`, `parallel_scan` on execution spaces
  - Atomic operations: compare-and-swap, add, bitwise-or, bitwise-and

- **Containers**
  - UnorderedMap – fast lookup and thread scalable insert / delete
  - Vector – subset of std::vector functionality to ease porting
  - Compress Row Storage (CRS) graph
  - Host mirrored & synchronized device resident arrays

- **Sparse Linear Algebra**
  - Sparse matrices and linear algebra operations
  - Wrappers for vendors’ libraries
  - Portability layer for Trilinos manycore solvers
Kokkos Core: Managing Data Access

Performance Portability Challenge:
Require Device-Dependent Memory Access Patterns

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

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

- “Array of Structures” vs. “Structure of Arrays”? 
  - This is, and has been, the *wrong* question

Right question: Abstractions for Performance Portability?
Kokkos Core: Fundamental Abstractions

- Devices have Execution Space and Memory Spaces
  - Execution spaces: Subset of CPU cores, GPU, ...
  - Memory spaces: host memory, host pinned memory, GPU global memory, GPU shared memory, GPU UVM memory, ...
  - Dispatch computation to execution space accessing data in memory spaces

- Multidimensional Arrays, with a twist
  - Map multi-index (i,j,k,...) ↔ memory location in a memory space
  - Map is derived from an array layout
  - Choose layout for device-specific memory access pattern
  - Make layout changes transparent to the user code;
  - IF the user code honors the simple API: a(i,j,k,...)

Separates user’s index space from memory layout
Allocate and access multidimensional arrays

```cpp
class View< double ** [3][8] , Device > a("a",N,M);
```

- Dimension [N][M][3][8] ; two runtime, two compile-time
- `a(i,j,k,l)` : access data via multi-index with device-specific map
- Index map inserted at compile-time (C++ template meta programming)

Identical C++ ‘View’ objects used in host and device code

Assertions that ‘a(i,j,k,l)’ access is correct

- Compile-time:
  - Execution space can access memory space (instead of runtime segfault)
  - Array rank == multi-index rank
- Runtime (debug mode)
  - Array bounds checking
  - Uses Cuda ‘assert’ mechanism on GPU
Kokkos Core: Multidimensional Array Layout and Access Attributes

- Override device’s default array layout
  ```
  class View<double**[3][8], Layout, Device> a("a",N,M);
  ```
  - E.g., force row-major or column-major
  - Multi-index access is unchanged in user code
  - *Layout* is an extension point for blocking, tiling, etc.

- Example: Tiled layout
  ```
  class View<double**, TileLeft<8,8>, Device> b("b",N,M);
  ```
  - Layout changes are transparent to user code
  - IF the user code honors the *a(i,j,k,...) API*

- Data access attributes – user’s intent
  ```
  class View<const double**[3][8], Device, RandomRead> x = a ;
  ```
  - Constant + RandomRead + GPU → read through GPU texture cache
  - Transparent to user code
Kokkos Core: Deep Copy Array Data

NEVER have a hidden, expensive deep-copy

- Only deep-copy when explicitly instructed by user code
- Avoid expensive permutation of data due to different layouts
  - Mirror the layout in Host memory space

```cpp
typedef class View<...,Device> 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 Device uses host memory *or* if Host can access Device memory space (CUDA unified virtual memory)
- Then ‘a_h’ is simply a view of ‘a’ and deep_copy is a no-op
Kokkos Core: Dispatch Data Parallel Functors
‘NW’ units of data parallel work

- `parallel_for(NW, functor)`
  - Call `functor(iw)` with `iw ∈ [0,NW)` and `#thread ≤ NW`

- `parallel_reduce(NW, functor)`
  - Call `functor(iw, value)` which contributes to reduction ‘value’
  - Inter-thread reduction via `functor.init(value)` & `functor.join(value,input)`
  - Kokkos manages inter-thread reduction algorithms and scratch space

- `parallel_scan(NW, functor)`
  - Call `functor(iw, value, final_flag)` multiple times (possibly)
  - if `final_flag == true` then ‘value’ is the prefix sum for ‘iw’
  - Inter-thread reduction via `functor.init(value)` & `functor.join(value,input)`
  - Kokkos manages inter-thread reduction algorithms and scratch space
Kokkos Core: Dispatch Data Parallel Functors
League of Thread Teams (grid of thread blocks)

- `parallel_for( { #teams , #threads/team } , functor )`
  - Call `functor( teaminfo )`
  - `teaminfo = { #teams, team-id, #threads/team, thread-in-team-id }`

- `parallel_reduce( { #teams , #threads/team } , functor )`
  - Call `functor( teaminfo , value )`

- `parallel_scan( { #teams , #threads/team } , functor )`
  - Call `functor( teaminfo , value , final_flag )`

- A Thread Team has
  - Concurrent execution with intra-team collectives (barrier, reduce, scan)
  - Team-shared scratch memory
  - Exclusive use of CPU and Xeon Phi cores while executing
Kokkos Core: Manage Memory Access Pattern

Compose Parallel Dispatch $\circ$ Array Layout

- Parallel Dispatch
  - Maps calls to functor($iw$) onto threads
  - GPU: $iw = \text{threadIdx} + \text{ blockDim } \times \text{ blockIdx}$
  - CPU: $iw \in [\text{begin, end})_\text{Th}$ ; contiguous partitions among threads

- Multidimensional Array Layout
  - Contract: leading dimension (right most) is parallel work dimension
    - Leading multi-index is ‘$iw$’ : $a( \text{ iw , j,k,l})$
  - Choose array layout for required access pattern
    - Choose AoS for CPU and SoA for GPU

- Fine-tuning
  - E.g., padding dimensions for cache line alignment
Kokkos Containers

- Kokkos::DualView< type, device >
  - Bundling a View and its View::HostMirror into a single class
  - Track which View was most recently updated
  - Synchronize: deep copy from most recently updated view to other view
    - Host $\rightarrow$ device OR device $\rightarrow$ host
  - Capture a common usage pattern into DualView class

- Kokkos::Vector< type, device >
  - Thin layer on rank-one View with “look & feel” of std::vector
  - No dynamic sizing from the device execution space
    - Thread scalability issues
  - Aid porting of code using std::vector
    - That does not dynamically resize within a kernels
Kokkos Containers: Unordered Map

- Thread scalable
  - Lock-free implementation with minimal/essential 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
  - Functor: \{status, index\} = map.insert(key,value);
    - Status = success | existing | failed due to insufficient capacity
    - Reduction on failed-count to resize the map

- Host:
  UnorderedMap<Key,Value,Device> map ;
  do {
    map.rehash( capacity );
    capacity += ( nfailed = parallel_reduce( NW , 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 “Device” Performance Tests
  - NVidia Kepler K40 (Atlas), 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
Outline

- What is Kokkos
- Evaluation via mini-applications
  - MiniMD molecular dynamics
  - MiniFE Conjugate Gradient (CG) iterative solver
  - MiniFENL sparse matrix construction
- Refactoring legacy libraries and applications
  - CUDA UVM (unified virtual memory) in the critical path!
- Conclusion
Solve Newton’s equations for $N$ particles

Simple Lennard Jones force model:

$$F_i = \sum_{j, r_{ij} < r_{cut}} 6 \varepsilon \left[ \left( \frac{\sigma}{r_{ij}} \right)^7 - 2 \left( \frac{\sigma}{r_{ij}} \right)^{13} \right]$$

Use atom neighbor list to avoid $N^2$ 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*e*( (s/r_ij)^7 - 2*(s/r_ij)^13 )
} 

f(i) = f_i;
```

Moderately compute bound computational kernel

On average 77 neighbors with 55 inside of the cutoff radius
MiniMD Performance
Lennard Jones (LJ) force model using atom neighbor list

- Test Problem (#Atoms = 864k, ~77 neighbors/atom)
  - Neighbor list array with correct vs. wrong layout
    - Different layout between CPU and GPU
  - Random read of neighbor coordinate via GPU texture fetch

- Large loss in performance with wrong layout
  - Even when using GPU texture fetch
  - Kokkos, by default, selects the correct layout
**MiniFE CG-Solver on Sandia’s Testbeds**

**Kokkos competitive with “native” implementations**

- Finite element mini-app in Mantevo (mantevo.org)
  - CG solve of finite element heat conduction equation
- Numerous programming model variants
  - More than 20 variants in Mantevo repository (eight in release 2.0)
- Evaluating hardware testbeds and programming models

---

**Graph:**

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

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>K20X</td>
<td>4</td>
</tr>
<tr>
<td>IvyBridge</td>
<td>8</td>
</tr>
<tr>
<td>SandyBridge</td>
<td>8</td>
</tr>
<tr>
<td>XeonPhi B0</td>
<td>8</td>
</tr>
<tr>
<td>XeonPhi C0</td>
<td>12</td>
</tr>
<tr>
<td>IBM Power7+</td>
<td>24</td>
</tr>
</tbody>
</table>

**Platforms:**

- NVIDIA ELL
- NVIDIA CuSparse
- Kokkos
- TBB
- OpenMP
- Cilk+(1 Socket)
- MPI-Only
- OpenCL
MiniFENL: Mini driver 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 maps sparse linear system
  - Sparse linear system graph: node-node map
  - Element-graph map for scatter-atomic-add assembly algorithm
    - Graph-element map for gather-sum assembly algorithm

- Compute nonlinear residual and Jacobian
  - Iterate elements to compute per-element residual and Jacobian
    - Scatter-atomic-add values into linear system
      - Save values in gather-sum scratch array
      - Iterate rows, gather data from scratch array, sum into linear system

- Solve linear system for Newton iteration
Scatter-Atomic-Add vs. Gather-Sum

Map: Mesh $\rightarrow$ Sparse Graph

Scatter-Atomic-Add Pattern

Element Computations + Scatter-Add

atomic_add

Sparse Linear System Coefficients

Finite Element Data

Element Computations

very large Scratch Arrays

Gather-Sum

Gather-Sum Pattern
Scatter-Atomic-Add vs. Gather-Sum

- Both are thread-safe and thread-scalable

- Scatter-Atomic-Add
  - Simple implementation
  - Fewer global memory reads and writes
    - Atomic operations much slower than corresponding regular operation
    - Non-deterministic order of additions – floating point round off variability
    - Double precision atomic add is a looped compare-and-swap (CAS)

- Gather-Sum
  - Deterministic order of additions – no round off variability
  - Extra scratch arrays for element residuals and Jacobians
  - Additional parallel-for

- Performance comparison – execution time
  - Neglecting the time to pre-compute mapping(s), assuming re-use
  - Cost of atomic-add vs. additional parallel-for for the gather-sum
Performance Comparison: Element+Fill

- ScatterAtomic as good or better without extra scratch memory
- Phi: ScatterAtomicAdd ~equal to GatherSum
  - ~2.1x speed up from 1 to 4 threads/core – hyperthreading
- Kepler: ScatterAtomicAdd ~40% faster than GatherSum
  - Fewer global memory writes and reads
  - Double precision atomic-add via compare-and-swap algorithm
  - Plan to explore element coloring to avoid atomics for scatter-add
Thread Scalable CRS Graph Construction

1. Fill unordered map with elements’ (row-node, column-node)
   - Parallel-for of elements, iterate node-node pairs
   - Successful insert to node-node unordered map denotes a unique entry
   - Column count = count unique entries for each row-node

2. Construct (row-node, column-node) sparse graph
   - Parallel-scan of row-node column counts
     - This is now the CRS row-offset array
   - Allocate CRS column-index array
   - Parallel-for on node-node unordered map to fill CRS column-index array
   - Parallel-for on CRS graph rows to sort each row’s column-indices

   Thread scalable pattern for construction
   a. Parallel count
   b. Allocate
   c. Parallel fill
   d. Parallel post-process
Graph construction is portable and thread scalable

Graph construction 2x-3x longer than one Element+Fill

- Finite element fill computation is
  - Linearized hexahedron finite element for: $-k \Delta T + T^2 = 0$
  - 3D spatial Jacobian with 2x2x2 point numerical integration
Outline

 What is Kokkos

 Evaluation via mini-applications

 Refactoring legacy libraries and applications
  ▪ CUDA UVM (unified virtual memory) in the critical path!
  ▪ From pure MPI parallelism to MPI + Kokkos hybrid parallelism
  ▪ Tpetra: Open-source foundational library for sparse solvers
  ▪ LAMMPS: Molecular dynamics application

 Conclusion
Tpetra: Foundational Layer / Library for Sparse Linear Algebra Solvers

- **Tpetra**: Sandia’s templated C++ library for sparse linear algebra
  - Distributed memory (MPI) vectors, multi-vectors, and sparse matrices
  - Data distribution maps and communication operations
  - Fundamental computations: axpy, dot, norm, matrix-vector multiply, ...
  - Templated on “scalar” type: float, double, automatic differentiation, polynomial chaos, ...

- **Higher level solver libraries built on Tpetra**
  - Preconditioned iterative algorithms
  - Incomplete factorization preconditioners
  - Multigrid solvers

- **Early internal prototype for portable thread-level parallelism**
  - Did not address array layouts or access traits, used raw pointers
  - Limited use / usability outside of internal Tpetra implementation
Tpetra: Foundational Layer / Library for Sparse Linear Algebra Solvers

- **Incremental Porting of Tpetra to (new) Kokkos**
  - Maintain backward internal compatibility during transition
  - Change internal implementation of data structures
    - Kokkos Views with prescribed layout to match existing layout
    - Extract raw pointers for use by existing computational kernels
  - Incrementally refactor kernels to use Kokkos Views

- **Status**
  - Vector, MultiVector, and CrsMatrix data structures using Kokkos Views
  - Basic linear algebra kernels working
  - CUDA, OpenMP, and Pthreads back-ends operational

- **CUDA UVM (unified virtual memory) critical for transition**
  - Sandia’s early access to CUDA 6.0 via Sandia/NVIDIA collaboration
  - Refactoring can neglect deep-copy and maintain correct behavior
  - Allows incremental insertion of deep-copies as needed for performance
CUDA UVM Expedites Refactoring Legacy Code

- **UVM memory space** accessible to all execution spaces
  - Hard to find all points in legacy code where deep copy is needed
  - Start with UVM allocation for all Kokkos View device allocations
  - Hide special UVM allocator within Kokkos’ implementation

- **Basics of UVM (without CUDA streams)**
  - Automatic host->device deep copy at kernel dispatch
    - For UVM data updated on the host
  - Automatic device->host deep copy when accessing UVM on the host
    - Per memory page granularity

- **Limitations**
  - Requires compute capability 3.0 or greater (Kepler)
  - Total UVM memory space allocations limited by device memory
  - Host access to UVM data forbidden during kernel execution
    - Enforce by executing with CUDA_LAUNCH_BLOCKING=1
Performance issues identified

- Currently Tpetra with CUDA back-end slower and not scaling
- Due to Tpetra implementation or CUDA/UVM back-end?
Analysis of Tpetra slowdown on CUDA

- Profiling problem using MiniFE with and without UVM
  - Tpetra refactoring relies upon UVM
  - MiniFE quickly modified to use UVM
  - Identified performance issue with kernel launch + UVM

MiniFE without UVM (original) vs. MiniFE with UVM allocations

30us kernel launch overhead

300us kernel launch overhead
Tpetra/MiniFE/Kokkos/UVM – Epilogue

- Early identification of problem leading to fix by NVIDIA
  - Fixed in alpha-driver (#331.56) – soon be publically available
  - Win-win: Tpetra/Kokkos expedited porting + early feedback to NVIDIA
LAMMPS Porting to Kokkos has begun

- LAMMPS molecular dynamics application (lammps.sandia.gov)

Goal
- Enable thread scalability throughout code
- Replace specialized thread-parallel packages
  - Reducing code redundancy by 3x

Leverage algorithmic exploration from miniMD
- MiniMD: molecular dynamics mini-app in Mantevo
- Transfer thread-scalable algorithms from miniMD to LAMMPS

Release with optional use of Kokkos in April 2014
- Implement framework: data management and device management
- All parts of some simple simulations can run on device via Kokkos
LAMMPS Porting to Kokkos early results

- Strong scaling “aggregate compute time” = wall clock * # compute nodes
- Performing as well or better than original non-portable threaded code
LAMMPS Hybrid Parallel Execution Performance

- All kernels compiled for both Host and Device
  - Choose kernels’ execution space at runtime
- Host-device data transfer managed with DualViews
  - Allow legacy code still to run on the host
- Experiment: DeepCopy versus UVM managed data transfers
  - Time integration on CPU (1 or 8 Threads), everything else on GPU
  - 1000 timesteps, 16k atoms, standard LJ force kernel

<table>
<thead>
<tr>
<th>Time Step</th>
<th>Data Transfer</th>
<th># of Dev-&gt;Host</th>
<th>Time Dev-&gt;Host</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeepCopy (8T)</td>
<td>1,870us</td>
<td>340us</td>
<td>2 (2*740kB)</td>
</tr>
<tr>
<td>UVM (1T)</td>
<td>3,820us</td>
<td>*2,290us</td>
<td>~250 (4k pages)</td>
</tr>
<tr>
<td>UVM (8T)</td>
<td>6,620us</td>
<td>*5,090us</td>
<td>~290 (4k pages)</td>
</tr>
</tbody>
</table>

- UVM 4k page transfer latency ~best expected for PCI bus
  - Slow down when Host has more than one idling thread
- Explicit deep copy of large array out-performs per-page UVM
Conclusion

- Kokkos Layered Libraries / Programming Model
  - Data parallel (for, reduce, scan) dispatch to execution spaces
  - Multidimensional arrays with polymorphic layout in memory spaces
  - Parallel dispatch → polymorphic layout → manage data access pattern
  - AoS versus SoA solved with appropriate abstractions using C++ templates
  - UnorderedMap with thread scalable insertion

- Evaluation with Mini-Applications
  - Polymorphic array layout critical for performance portability
  - Kokkos-portable kernels’ performance as good as native implementations
  - Scatter-atomic-add is a performant option for linear system fill
  - CRS graph construction can be thread scalable

- Transition of Legacy Codes
  - Incremental porting necessary and tractable with CUDA UVM
  - Refactored-in deep copy semantics needed for best performance