KokkosArray: Multidimensional Arrays for Manycore Performance-portability

H. Carter Edwards, Christian Trott, Daniel Sunderland
Sandia National Laboratories

GPU TECHNOLOGY CONFERENCE 2013
MARCH 18-21, 2013 | SAN JOSE, CALIFORNIA

SAND2013-0692C (Unlimited Release)
Outline

- Part 1: KokkosArray Fundamental Concepts and API
  - Making is look easy for the user

- Part 2: Performance-Portability Evaluation
  - “Unit” tests and proxy-applications
  - Cray XK7 with NVIDIA Kepler
  - Intel Knights Corner cluster (pre-production hardware)

- Part 3: Porting MiniMD to KokkosArray
  - Evaluate new ideas and programming models before implementing within the production LAMMPS code
  - Variants for MPI+OpenMP, MPI+OpenCL and MPI+KokkosArray
Performance-Portability Challenge

Device-Dependent Memory Access Patterns

- Correctness: no race conditions
- Performance: proper placement, blocking, striding, ...
- CPUs with NUMA and vector units
  - Core-data affinity: first touch and consistent access
  - Alignment for cache-lines and vector units
- GPU Coalesced Access with cache-line alignment
- “Array of Structures” vs. “Structure of Arrays”?  
  - This is, and has been, the wrong question

Right question: Abstractions for Performance-Portability?
Programming Model Concept

two foundational ideas

• Manycore Device
  – Separate memory spaces (physically or logically)
  – Dispatch work to device: computation + data

• Classic Multidimensional Arrays, with a twist
  – Map multi-index \((i,j,k,...) \leftrightarrow \text{memory location on the device}\)
    • Efficient: computation and memory used
  – Map is derived from a 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: \(a(i,j,k,...)\)

Separate user’s index space from memory layout
KokkosArray Library
Just arrays and parallel dispatch

• Standard C++ Library, 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
  – Compile-time polymorphism for devices and array layouts
  – C++1998 standard; would be nice to *require* C++2011 ...

• KokkosArray is not:
  – A linear algebra library
  – A mesh or grid library
  – A discretization library

Intent: Build such libraries on top of KokkosArray
API: Allocation, Access, and Layout

• Basic: data allocation and access
  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
• Same ‘View’ in both host and device code
• Access Safety
  – Compile-time assertion a(i,j,k,l) is used correctly
    • Assert device code accesses device memory
    • Assert host code accesses host memory
  – Runtime array bounds checking – in debug mode
    • Using Cuda ‘assert’ mechanism on the device
API : Allocation, Access, and Layout

• Advanced : specify array layout
  
  class View<double**[3][8], Layout , Device> a(“a”,N,M);
  
  – Override default layout; 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.

• Advanced : specify memory access attributes
  
  class View<const double**[3][8], Device, RandomRead> x = a ;
  
  – E.g., access ‘x’ data through GPU texture cache
API : View Semantics
(e.g., reference counting)

• Basic : view semantics
  typedef class View<double**,Device> MyMatrixType ;
  MyMatrixType a("a",N,M);  // allocate array
  MyMatrixType b = a ;  // A new light-weight view to the same data
  – Reference counting is internal to avoid cluttering user-code

• Advanced : turn off reference counting
  class View<const double**,Layout,Device,Unmanaged> c = a ;
  – Faster to construct, assign, and destroy; however,
  > User-code assumes responsibility to destroy ‘c’ before ‘a’
  – Can only allocate managed views
API : Deep Copy

NEVER have a hidden, expensive deep-copy

• Only deep-copy when explicitly instructed by user code

• Basic : mirror the layout in Host memory space
  - Avoid transpose or permutation of data: simple, fast deep-copy

```cpp
typedef class View<...,Device> MyViewType;
MyViewType a("a",...);
MyViewType::HostMirror a_host = create_mirror( a );
deep_copy( a, a_host );
deep_copy( a_host, a );
```

• Advanced : avoid unnecessary deep-copy
  ```cpp```
  ```cpp
  MyViewType::HostMirror a_host = create_mirror_view( a );
  ```
  - If Device uses host memory then ‘a_host’ is simply a view of ‘a’
  - deep_copy becomes a no-op
API : Parallel Dispatch

parallel_for( nwork , functor )

• Functor : Function + its calling arguments
  template< class DeviceType > // allows for partial-specialization
  struct AXPY {
    void operator()(int iw) const { y(iw) += a * x(iw); } // shared function
    AXPY( ... ) ... { parallel_for( nwork , *this ); } // parallel dispatch
  }
  typedef DeviceType device_type ; // run on this device
  const double a ;
  const View<const double*,device_type> x ;
  const View<double*,device_type> y ;
};

– Functor is shared and called by NP threads (NP ≤ nwork)
– Thread parallel call to ‘operator()(iw)’ : iw ∈ [0,nwork)
– Access array data with ‘iw’ to avoid race conditions
Parallel Dispatch via Functor

• **Thread-Memory Affinity → Data Access Pattern**
  – Assume parallel work index is the array’s leading index
  – CPU : thread ↔ contiguous indices for NUMA
  – CPU : thread ↔ contiguous indices for vectorization
  – GPU : thread ↔ strided indices for coalesced access

• **Why Functor Pattern ?**
  – Standard C++1998 and *Portable* (desirable: C++2011 lambdas)
  – Flexible: as many argument-members as you need

• **Why not Function + Argument List ?**
  ➢ Requires language / compiler extensions
  – Impedes device-specific specializations
API : Parallel Dispatch
parallel_reduce( nwork , functor , result )

• Similar to parallel_for, with *Reduction Argument*

```cpp
template< class DeviceType >
struct DOT {
    typedef DeviceType   device_type ;
    typedef double value_type ;  // reduction value type
    void operator()( int iw , value_type & contrib ) const
        { contrib += y(iw) * x(iw); } // this thread’s contribution
    DOT( … ) … { parallel_reduce( nwork , *this, result ); } // ... to be continued ...
    const View<const double*,device_type> x , y ;
    // ... to be continued ...
};
```

➢ Value type can be a ‘struct’, static array, or dynamic array

– Result is a value or View to a value on the device
API: Parallel Dispatch

parallel_reduce( nwork, functor, result )

• Initialize and join threads’ individual contributions

  struct DOT {
    // ... continued ...
    static void init( value_type & contrib ) { contrib = 0; }
    static void join( volatile value_type & contrib,
                     const volatile value_type & input )
                  { contrib = contrib + input; }
  }

  – Join threads’ contrib via commutative Functor::join
  – ‘volatile’ to prevent compiler from optimizing away the join

• Deterministic result ← highly desirable
  – Given the same device and # threads
  – Aligned memory prevents variations from vectorization
Not Discussed Today

- Hierarchical ThreadPool for NUMA, Intel-KNC
- Tiled Array Layouts
- Embedded Data Types
  - View< Type **[3][8], device >
  - Type can be automatic differentiation, stochastic bases,...
- Plans:
  - Abstracted interface for atomics
  - Blocked & variable blocked layouts
  - Hierarchical task parallelism
    - Task graph of data-parallel functors
    - Integration with task-scheduler (Qthreads)
Outline

• Part 1: KokkosArray Fundamental Concepts and API
  – Making is look easy for the user

• Part 2: Performance-Portability Evaluation
  – “Unit” tests and proxy-applications
  – Cray XK7 with NVIDIA Kepler
  – Intel Knights Corner (KNC): “Results are obtained on pre-production Intel Xeon Phi hardware, performance of final product versions may be different.”

• Part 3: Porting MiniMD to KokkosArray
  – Evaluate new ideas and programming models before implementing within the production LAMMPS code
  – Variants for MPI+OpenMP, MPI+OpenCL and MPI+KokkosArray
Performance-Portability Tests
same code compiled to devices *

• Modified Gram-Schmidt algorithm
  – Sequence of Level-1 BLAS: dot, scale, axpy
  – Limited by memory bandwidth and reduction synchronization

• Explicit dynamics proxy-application
  – Finite element stress and internal forces (computationally intense)
  – Assemble forces to vertices (random access), enforce boundary conditions, and integrate motion
  – “Halo exchange” communication of vertices’ motion

• Nonlinear thermal conduction proxy-application
  – Finite element residual & Jacobian assembled into sparse system
  – Newton iteration w/nested conjugate-gradient (CG) linear solve
    * On GPU using ‘cusparseDcsrmv’ within the CG solve
  – CG iterations have “halo exchange” communication
Performance-Portability Tests

• ‘Curie’ testbed at Sandia
  – Cray XK7 with 50 compute nodes:
    • AMD Opteron 6200 (2x8 cores)
    • NVIDIA K20X
  – GPU Direct capability not available

• ‘Compton’ testbed at Sandia
  – Intel Xeon Phi (MIC) co-processor cards: pre-production hardware
  – Cluster containing 64 Knights Corner (KNC) cards
  – Our KNCs: 57 cores x 4 hyperthreads (reserve one core for OS)
  – Hyperthreading necessary for latency hiding
  – Running in “KNC only” mode – direct inter-card communication
Modified Gram-Schmidt Performance
Limited by bandwidth and reductions

- Performance normalized by # devices
- Cray XK7 compute nodes
  - AMD Opteron 6200 (2x8 cores), ~51 GB/sec theoretical peak
  - NVIDIA K20X, ~250 GB/sec theoretical peak
- RW performance at “large enough” problem size
  - Opteron: achieved ~51% of peak
  - K20X: achieved ~65% of peak
Modified Gram-Schmidt Performance
On Knights Corner (pre-production)

• Hyperthreading
  • Threads-on-hyperthreads improves performance
  • MPI-on-hyperthreads degrades performance

• RW performance at “large enough” problem size
  – Performance normalized by device
  – ~200 GB/sec “achievable” peak (pre-production hardware)
  – Full threading utilization achieved ~23% of “achievable” peak
  – MPI-per-core achieved ~13% of “achievable” peak
Performance Evaluation: Explicit Dynamics ProxyApp

**Explicit Dynamics Element Computation Time / Element**
- 5 Cray XK7 Nodes

**Explicit Dynamics Node Update Time / Node**
- 5 Cray XK7 Nodes

- **Element computation time / element**
  - High computational intensity (operations / memory access)
- **Node update time / node**
  - High random-memory-access intensity
  - Benefit from texture cache? – TBD
Performance Evaluation on KNC: Explicit Dynamics ProxyApp

- Computationally intense
  - and NO communication

- Hyperthreads:
  - $56 \times \{1-4\}$ MPI processes / card
  - $56 \times \{1-4\}$ Threads / card

- Threads consistently outperform MPI processes
  - Using more KNC cards only exacerbates this difference
Performance Evaluation on KNC: Explicit Dynamics ProxyApp

- Threads outperform MPI processes
  - Even with NO communication
- More MPI processes cause large slowdown
  - Processes on hyperthreads competing for memory
- More threads cause slight slowdown
  - Threads on hyperthreads *attempt* to cooperate for memory access
Performance Evaluation on KNC: Explicit Dynamics ProxyApp

- More MPI processes cause drastic slowdown
  - Does not scale!
- More threads cause *slight* slowdown
- Threads significantly outperform MPI processes

- Consider 512 KNC cards
  - 114,688 MPI ranks
  - OR
  - 512 MPI ranks x 224 threads
Performance Evaluation on KNC: Nonlinear Thermal Conduction ProxyApp

- Nonlinear quadratic elem.
  - Compute contributions to residual and Jacobian
  - Computationally intensive
  - No communication

- Threads outperform MPI processes (again)
Performance Evaluation on KNC: Nonlinear Thermal Conduction ProxyApp

- Hyperthreads share core’s L1 cache: NUMA-like effect
  - Sparse mat-vec and matrix-assembly have random access
  - Domain decomposition improves cache utilization for MPI
  - Threads needed a similar domain decomposition / ordering
Performance Evaluation:
Nonlinear Thermal Conduction ProxyApp

- **Assembly Time per Row**
  - Gather-assemble data from element array, low comp. intensity

- **CG Time / Iteration / Row**
  - Dominated by data movement: sparse-matrix-vector multiply gathers vector data inter/intra process (need GPU-direct!)
Performance-Portability Tests

Conclusions

• Performance-portable “unit” tests

• Portable non-trivial proxy-application source code
  – CPU-threaded, GPU, and KNC-threaded
  – Explicit dynamics and nonlinear thermal conduction FEM

• Performance – Intel Phi
  – Compiler does vectorize through KokkosArray API
  – Must use MPI+thread hybrid parallelism (MPI-only will not work)
  – Domain decomposition ordering needed to improve cache utilization

• Performance - GPU
  – Proper coalesced memory access
  – Need GPU direct to reduce inter-node communication
  – Need access to GPU texture cache via portable API

• In progress
Outline

• Part 1: KokkosArray Fundamental Concepts and API
  – Making is *look* easy for the user

• Part 2: Performance-Portability Evaluation
  – “Unit” tests and proxy-applications
  – Cray XK7 with NVIDIA Kepler
  – Intel Knights Corner cluster (pre-production hardware)

• Part 3: Porting MiniMD to KokkosArray
  – Evaluate new ideas and programming models before implementing within the production LAMMPS code
  – Variants for MPI+OpenMP, MPI+OpenCL and MPI+KokkosArray
MiniMD: A mini-app for LAMMPS (lammps.sandia.gov)

- Evaluate new ideas and programming models before implementing within the production LAMMPS code
  - Variants for MPI+OpenMP, MPI+OpenCL and MPI+KokkosArray
- Part of Mantevo Suite (mantevo.org)
- 4k lines of code split into classes:
  - Integrate: main integration loop
  - Force{LJ/EAM}: actual force calculation
  - Neighbor: neighbor list construction
  - Comm: communication between MPI process
  - Thermo: calculates thermo dynamic output
Molecular Dynamics

- Solve Newton’s equations for \( N \) particles

- Force calculation with simple Lennard Jones model:

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

- Loop over particles’ NeighborLists to avoid \( N^2 \) computations

```c
pos_i = pos[i];
for( jj = 0; jj < num_neigh[i]; jj++) {
    j = neighs[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;
```

- Typically: \( N = 100k \) / compute-node; \#Neighbors = 40

- Sparse memory access moderately compute bound
Data Management

• Data types:

```cpp
typedef View<double*[3],LayoutRight,Device> tvector_2d;
typedef tvector_2d::HostMirror tvector_2d_host;
typedef View<double*[3],LayoutRight,Device,RandomRead>
tvector_2d_rnd; //On GPU use Texture cache
```

• Atom::growarray() --- reallocation function

```cpp
x = (double**) realloc_2d_double_array(x,nmax,3,3*nold);
```

Replaced by:

```cpp
tvector_2d xnew("X",nmax); // allocate new array
deep_copy_grow(xnew,x); // copy old to new
x = xnew; // automatically delete
h_x = KokkosArray::create_mirror_view(x); // create host copy
```

• Atom::upload() / download() --- transfer data between host and device

```cpp
KokkosArray::deep_copy(x,h_x);
KokkosArray::deep_copy(h_x,x);
```

– No-op if h_x and x are the same
Integration (i) – a simple kernel’s API

- Split function looping over variables into: (i) loop body function, (ii) functor calling loop body function, (iii) function submitting functor

```cpp
class Integrate {
    public:
        ...
        void initialIntegrate();
        ...
    private:
        double **x, **v, **f;
        int nlocal;
};
```

- Change pointers to Views
- Split out per-item loop body
- Create functor-wrapper

```cpp
class Integrate {
    public:
        ...
        void initialIntegrate();
        ...
    private:
        tvector_2d x,v,f;
        int nlocal;
        KOKKOSARRAY_INLINE_FUNCTION
        void initialIntegrateItem(int &i) const;
        friend class InitialIntegrateFunctor;
    }

struct InitialIntegrateFunctor {
    Integrate c ; // Copy of Integrate object
    KOKKOSARRAY_INLINE_FUNCTION
    void operator()(const int i) const
    {
        c.initialIntegrateItem(i);
    }
};
```
Integration (ii) – a simple kernel’s loop body

- Split function looping over variables into: (i) loop body function, (ii) functor calling loop body function, (iii) function submitting functor

```cpp
void Integrate::initialIntegrate() {
    #pragma omp for
    for(MMD_int i = 0; i < nlocal; i++) {
        v[i*3 + 0] += dtforce * f[i*3 + 0];
        v[i*3 + 1] += dtforce * f[i*3 + 1];
        v[i*3 + 2] += dtforce * f[i*3 + 2];
        x[i*3 + 0] += dt * v[i*3 + 0];
        x[i*3 + 1] += dt * v[i*3 + 1];
        x[i*3 + 2] += dt * v[i*3 + 2];
    }
}
```

```cpp
void Integrate::initialIntegrate() {
    f_initialIntegrate->c = *this;
    KokkosArray::parallel_for(nlocal, *f_initialIntegrate);
}
```

```cpp
KOKKOSARRAY_INLINE_FUNCTION void Integrate::initialIntegrateItem(int &i) const {
    v(i, 0) += dtforce * f(i, 0);
    v(i, 1) += dtforce * f(i, 1);
    v(i, 2) += dtforce * f(i, 2);
    x(i, 0) += dt * v(i, 0);
    x(i, 1) += dt * v(i, 1);
    x(i, 2) += dt * v(i, 2);
}
```
Force Calculation with Conditional Reduction (i)

• Optional energy calculation with force calculation

```c
void ForceLJ::compute_fullneigh() { // Original kernel
    for (int i = 0; i < nlocal; i++) {
        const double xtmp = x[i][0] , ytmp = x[i][1] , ztmp = x[i][2];
        double fix = 0 , fiy = 0 , fiz = 0;
        for (int k = 0; k < numneigh[i]; k++) {
            const int j = neighbors[i][k];
            const double dx = xtmp-x[j][0], dy = ytmp-x[j][1], dz = ztmp-x[j][2];
            const double rsq = dx*dx + dy*dy + dz*dz;
            if (rsq < cutforcesq) {
                const double sr2 = 1.0/rsq;
                const double sr6 = sr2*sr2*sr2;
                const double force = sr6*(sr6-0.5)*sr2;
                fix += dx*force; fiy += dy*force; fiz += dz*force;
            if(evflag) energy += sr6*(sr6-1.0); //conditional reduction
            }
        }
        f[i][0] += fix;  f[i][1] += fiy;  f[i][2] += fiz;
    }
}
```
• New function for inner loop

```cpp
template< int EVFLAG >
double ForceLJ::compute_fullneighItem(int &i) const {
    const double xtmp = x(i,0) , ytmp = x(i,1) , ztmp = x(i,2);
    double fix = 0 , fiy = 0 , fiz = 0;
    double energy = 0 ;
    for (int k = 0; k < numneigh[i]; k++) {
        const int j = neighbors(i,k);
        const double dx = xtmp-x(j,0), dy = ytmp-x(j,1), dz = ztmp-x(j,2);
        const double rsq = dx*dx + dy*dy + dz*dz;
        if (rsq < cutforcesq) {
            const double sr2 = 1.0/rsq;
            const double sr6 = sr2*sr2*sr2;
            const double force = sr6*(sr6-0.5)*sr2;
            fix += dx*force; fiy += dy*force; fiz += dz*force;
            if( EVFLAG ) energy += sr6*(sr6-1.0); //conditional reduction
        }
    }
    f(i,0) += fix;  f(i,1) += fiy;  f(i,2) += fiz;
    return energy ;
}
```

On GPU: Texture Fetch
struct ForceComputeFullneighFunctor { // Functor wrapper
typedef double value_type;
ForceLJ c; // Wrapped force computation class

KOKKOSARRAY_INLINE_FUNCTION
void operator()(int i) const { c.compute_fullneighItem<0>(i); }

KOKKOSARRAY_INLINE_FUNCTION
void operator()( int i, value_type & energy) const
{ energy += c.compute_fullneighItem<1>(i); }

KOKKOSARRAY_INLINE_FUNCTION static void init( ... );
KOKKOSARRAY_INLINE_FUNCTION static void join( ... );
};

void ForceLJ::compute_fullneigh(Atom &atom, Neighbor &neighbor, int me)
{
  f_compute_fullneigh-&>c = *this;

  if(evflag) energy = parallel_reduce(nlocal, *f_compute_fullneigh);
  else parallel_for(nlocal, *f_compute_fullneigh);
}
Neighborlist Build
Specialized Algorithm for Cuda

```cpp
struct NeighborBuildFunctor {
    Neighbor c;
    KOKKOSARRAY_INLINE_FUNCTION void operator()( const int i) const { 
        #if defined( __CUDA_ARCH__ )
            c.build_ItemCuda(i);
        #else
            c.build_Item(i);
        #endif
    }
}

#if defined( __CUDA_ARCH__ )
extern __shared__ double sharedmem[];
__device__ inline void Neighbor::build_ItemCuda(const int & ii) const {
    int ibin = blockIdx.x*gridDim.y+blockIdx.y;
    double* other_x = sharedmem;
    int* other_id = (int*) &other_x[3*blockDim.x];

    int bc = bincount[ibin];
    int i = threadIdx.x < bc ? bins[ibin*atoms_per_bin+threadIdx.x] : -1 ;
    double xtmp = x(i,0);
    other_x[threadIdx.x] = xtmp;
    ....
}
#endif
```
• **SB:** Intel dual Sandy Bridge E5-2670
• **MIC:** KNC with 57 Cores
Normalized Performance
16 nodes (node = 2x8cSB; 1xKNC; 1xK20)

atomsteps / (s/node)

70 M
60 M
50 M
40 M
30 M
20 M
10 M
0

# of atoms/node
256 1 k 4 k 16 k 66 k 262 k 1 M 4 M 17 M

SB (E5-2670, 8c/16HT, 2.6GHz)
KNC (57c/228HT)
K20x

On KNC MPI comm extremely slow fix expected soon
MiniMD
Experience Porting to KokkosArray

• MiniMD is Performance Portable with KokkosArray
  – Equivalent performance to CUDA version
  – Better than OpenMP implementation
  – < 10% performance loss vs. MPI version without threading

• Code complexity slightly increased vs MPI+OpenMP
  – Much less complex than OpenCL or CUDA implementation

• More future-proof than other programming models
  – New device backends through KokkosArray, not production code
  – Simple to change data layout without rewriting production code

• Not presented:
  – Out-of-bounds checking with traceback – in debug build
• KokkosArray available: trilinos.org

• MiniMD available: mantevo.org

• Contacts:
  – Carter Edwards: hcedwar at sandia.gov
  – Christian Trott: crtrott at sandia.gov