A Library for Performance-Portable Multidimensional Array Computations on Manycore Nodes

H. Carter Edwards (presenting) and Daniel Sunderland
Sandia National Laboratories

February 16, 2012
SIAM Conference on Parallel Processing for Scientific Computing
Savannah, Georgia

SAND2012-0294C
Introduction

• Challenge: Manycore Portability **with Performance**
  – Multicore-CPU and manycore-accelerator (e.g., NVIDIA)
  – Diverse memory access patterns, shared memory utilization, ...

• Via a Library, not a language
  – C++ with template meta-programming
  – In the *spirit* of Thrust or Threaded Building Blocks (TBB)
  – Concise and simple API: functions and multidimensional arrays

• Data Parallel Functions
  – Deferred task parallelism, pipeline parallelism, ...
  – Simple parallel_for and parallel_reduce semantics

• Multidimensional Arrays
  – versus “arrays of structs” or “ structs of arrays”
Kokkos Array Abstractions

• Manycore Device
  – Has many threads of execution sharing a memory space
  – Manages a memory space separate from the host process
    • Physically separate (GPU) or logically separate (CPU)
    • or with non-uniform memory access (NUMA)

• Data Parallel Function
  – Created in the host process, executed on the manycore device
  – Performance can be dominated by memory access pattern
    • E.g., NVIDIA coalesced memory access pattern

• Multidimensional Array
  ✓ Map array data into a manycore device’s memory
  – Partition array data for data parallel work
  – Function + parallel partition + map ⇒ memory access pattern
Kokkos Array Abstraction: Multidimensional Array and its Map

• Homogeneous Collection of Data Members
  – Plain-old-data type
  – Members referenced by a multi-index in a multi-index space

• Multidimensional Array Map
  – Bijective map: multi-index space ↔ array data members
    • [ 0 .. N0 ) x [ 0 .. N1 ) x [ 0 .. N2 ) x … ↔ memory locations
  – Multiple valid maps
    • E.g., FORTRAN, ‘C’, space-filling-curve, block-cyclic, …
  – Map for best memory access pattern is device-dependent
  – Transparently introduce the best map at compile-time
    • No alteration of the application’s source code
    • C++ template meta-programming
Kokkos Array Abstraction: Parallel Partitioning

• 1D Parallel Partitioning of Data
  – Partition into NP atomic units of parallel work
  – Index space has one parallel work dimension: (NP, N1, N2, …)
  – Deferred 2D+ partitioning; e.g., matrices and grids

• Parallel Work on Shared Arrays
  – NP atomic units of parallel work: ip ∈ [0 .. NP)
  – Parallel thread-safety:
    • Update only array members with index (ip, *, *, …)
    • Don’t query data being updated by different unit of work

• Example: Finite Element Bases Gradients
  – grad( N-Element, N-Spatial-Dimension, N-Bases-per-Element)
  – Parallel function over elements: compute gradients
Kokkos Array API: Multi-index Space and Data Access

• Index space available on the host and device
• Data members only accessible on the device

template< class Device >
void my_function( Kokkos::MDArray<double,Device> grad )
{
  assert( 3 == grad.rank() ); // Verify index space rank
  size_t nBases = grad.dimension(2); // Query index space dimension
  size_t nSpace = grad.dimension(1);
  size_t nElem = grad.dimension(0);

  // Access data member within code running on the device
  // using standard multi-index notation
  grad( iElem , iSpace , iBases ) = value ;
}
Kokkos Array API:
Shared Ownership View Semantics

• Allocate Array on the Host, Use Array on the Device

```cpp
Kokkos::MDArray<double,Device> grad ; // NULL view
// Allocate array data on the device:
grad = Kokkos::create_mdarray<double,Device>(nElem,nSpace,nBases);
{
    // new shared ownership view to the same data
    Kokkos::MDArray<double,Device> tmp = grad ; // shallow copy
} // tmp is destroyed, data is NOT deallocated
} // grad is last view destroyed, data IS deallocated
```

• Multiple Views to Same Data
  – Last view destroyed (or reassigned) automatically deallocates
  – Assignment is a shallow copy operation
    • The reference is copied, not the allocated data
Kokkos Array API: Mirrored Arrays and Deep Copy

- Different Devices have Different Array Maps
  - Simple memory-to-memory copy yields the wrong map
  - Remapping array data is expensive
  - Need: array in host memory space but with device’s map

Kokkos::MDArray<double,Device>::HostMirror gradHostMirror;

gradHostMirror = create_mirror( grad ); // allocate on the host

deep_copy( grad , gradHostMirror ); // copy data device <- host
deep_copy( gradHostMirror , grad ); // copy data host <- device

- HostMirror – map-compatible array in Host memory
  - Fast, simple memory-to-memory copy with correct map
  - In the same memory space a mirror can be a view
Kokkos Array API: Users’ Parallel-For Function (Functor)

• C++ Functor : Function + Arguments (References to Data)
  – Template on the Device for portability and map instantiation

```cpp
template< class DeviceType >
class MyFunctor {
public:
    typedef DeviceType device_type;        // Identify device
    void operator()( int iElem ) const {    // Compute for iElem
        m_grad(iElem,iSpace,iBases) = value; // Update my data
    }
    MDArray<double,device_type> m_grad;
    MyFunctor( const MDArray<double,device_type> & grad )
        : m_grad(grad) {} // Shallow copy
};
// Call NElem times
parallel_for( NElem , MyFunctor<Device>( grad ) );
```
Kokkos Array API: Users’ Parallel-Reduce Functor

template< class DeviceType >
class MyCentroid {
public:
    typedef DeviceType device_type;
typedef struct { double coord[3] , mass ; } value_type ;

    void operator()( int ipt , value_type & update ) const
    { update.mass += m_mass(ipt);
        update.coord[k] += m_coord(ipt,k) * m_mass(ipt,k); }

    static void join( volatile value_type & update ,
                      volatile const value_type & input )
    { update.mass += input.mass; update.coord[k] += input.coord[k];}

    static void init( value_type & output )
    { output.mass = 0; output.coord[k] = 0; }

    MDArray<double,device_type> m_coord , m_mass;
    MyCentroid( ... );
};
Kokkos Array API:  
Users’ Parallel-Reduce Functor

// Return result to the Host:  
MyCentroid<Device>::value_type result =  
   parallel_reduce( NPT, MyCentroid<Device>( ... ) );

// OR post-process result on the device:  
parallel_reduce( NPT, MyCentroid<Device>( ... ),  
               MyCentroidFinalize<Device>( ... ) );

template< class Device >
class MyCentroidFinalize {  
   public:
      typedef MyCentroid<Device>::value_type value_type ;  
   // A view to non-array data on the device:  
      Kokkos::Value< value_type , Device > m_result ;  

      void operator( value_type & tmp_result ) const  
      {
         m_result.mass    = tmp_result.mass ;  
         m_result.coord[k] = tmp_result.coord[k] / tmp_result.mass ;  
      }  
};
## Performance Test Case: Parallel_For: Hexahedral Basis Gradient

### Finite Element Kernel
- **Input:** coord(NP,3,8)
- **Output:** grad(NP,3,8)
- 6.6 flops per value access
- **Xeon:** 2 x 6core x 2 HT
- **Opteron:** 2 x 12core
- **NVIDIA C2070** (448 cores)
- Same code on all devices

### vs. Hand-written CUDA
- Hard-coded index map
- **yields 20% performance gain**

---

**Performance of Hexahedral Gradient Kernel:**
Double Precision Gflop/sec vs. Element Count

- NVIDIA via hand-written CUDA
- NVIDIA via Array API
- Xeon using 24 Pthreads
- Opteron using 24 Pthreads
Performance Test Case:
Modified Gram-Schmidt Orthogonalization

• Classical Algorithm
  – sequence of parallel_for and parallel_reduce operations
  – Memory access dominated
  – Xeon: 2 x 6core x 2 HT
  – Opteron: 2 x 12core
  – NVIDIA C2070 (448 cores)
  – Same code on all devices

• Reductions w/Finalize
  – Inner products and norms remain on the device
  – No data returned until algorithm completes
• Explicit Dynamics : computationally intensive
  – Compute element stress and internal force
  – Gather-assemble forces at nodes and compute acceleration
  – Apply boundary conditions at nodes and integrate motion

• Implicit Thermal Conduction : memory access intensive
  – Compute element’s linear system contributions
  – Gather-assemble sparse linear system, solve linear system

• Same code compiled and run on devices:
  – Westmere: Xeon 2.93 GHz, 2 cpu X 12 cores x 2 hyperthreads
  – Magny-Cours: Opteron 2.4 GHz, 2 cpus X 8 cores
  – NVIDIA C2070: 1.2 GHz, 448 cores

• NUMA control (via HWLOC) on multicore-CPUs
Performance-Portable: Explicit Dynamics Mini-Application

Element Computation: Single Prec.

Node Update: Single Prec.

Element Computation: Double Prec.

Node Update: Double Prec.

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy’s National Nuclear Security Administration under contract DE-AC04-94AL85000.
Performance-Portable:
Implicit Thermal Mini-Application

**Element Computation : Single Prec.**

- Westmere-24
- NVIDIA
- Magny-Cours-16

**Linear System Assemble : Single Prec.**

- Westmere-24
- Magny-Cours-16
- NVIDIA

**Element Computation : Double Prec.**

- Westmere-24
- NVIDIA
- Magny-Cours-16

**Linear System Assemble : Double Prec.**

- Westmere-24
- Magny-Cours-16
- NVIDIA

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy’s National Nuclear Security Administration under contract DE-AC04-94AL85000.
**NUMA Effects on Westmere: Explicit Dynamics Mini-Application**

- **NUMA ‘first touch’ on data in both cases**
- **Use HWLOC to explicitly place threads with adjacent data**
  - Adjacent-rank threads have adjacent data
  - Locality: shared core (hyperthreads) and NUMA affinity
Kokkos Array: Conclusion & Plans

• **Achieved Performance-Portability**
  – Data parallel functions on “classical” multidimensional arrays
  – Abstract & separate array map: index space ↔ device memory
  – Automatically & transparently use device-optimal array map
  – Same, unmodified code on Xeon, Opteron, and NVIDIA

• **Plans**
  – Other devices; e.g., Intel MIC
  – Other operations; e.g., parallel-scan, aggregated-functors, …
  – Stochastic finite elements’ polynomial types (in progress)
  – Multi-parallel-index arrays: grids, matrices

• **Available:** [http://trilinos.sandia.gov](http://trilinos.sandia.gov)