Manycore-Portable Multidimensional Arrays for Finite Element Computations

H. Carter Edwards
Sandia National Laboratories

July 12, 2012
2012 SIAM Annual Meeting
Minneapolis, Minnesota

SAND2012-5042C
Strategy / Approach

• 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
  – Concise and simple abstractions, API, and runtime
  – C++ with template meta-programming
  – In *spirit* of Intel’s TBB, NVIDIA’s Thrust & CUSP, MS C++AMP, ...

• Data Parallel Operations (parallel_for & parallel_reduce)
  – *Deferred* task parallelism, pipeline parallelism, ...

• Multidimensional Arrays – intuitive for science & engineering
  – “arrays of structs” vs. “structs of arrays” – wrong conversation
  – Abstraction for data placement, locality, mapping
Kokkos Array Abstractions

• Manycore Device – has separate memory space
  – Physically (GPU), Performance (NUMA), Logically (CPU)

• Data Parallel Operations
  – Executed by many threads on the manycore device
  – Performance can be dominated by memory access pattern
    • E.g., NVIDIA coalescing, NUMA regions

• Multidimensional Array
  ➢ Map array data into a manycore device’s memory
    • Parallel partitioning
    • Multi-index computation
  – Data parallel operation + map ⇒ memory access pattern
Kokkos Array Abstraction: Multidimensional Array and its Map

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

• Multi-Index Map
  – Bijective map : multi-index space ↔ array data members
    • \([ 0 .. N0 \) \( \times [ 0 .. N1 \) \( \times [ 0 .. N2 \) \( \times \ldots \leftrightarrow\) memory locations
  – Many 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

• Parallel Partitioning of Data
  – Partition into NP atomic units of parallel work
  – Index space has parallel work dimensions: (NP, N1, N2, …)
  – Limited to 1D for now; deferred 2D+ parallel partitioning

• 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 known on the host and device
• Data members accessible only on the device

```cpp
void my_function( Kokkos::MDArray<double,MapIntoDevice> grad )
{
    // Access data member within code running on the device
    // using standard multi-index notation
    grad( iElem, iSpace, iBases ) = value;

    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);
}
```
Kokkos Array API: Mirrored Arrays and Deep Copy

• Different Devices have Different Maps
  – Need to access array data in Host memory
  – However, remapping array data is expensive

• HostMirror
  – Array in Host memory space using Device’s map
  – No remapping, fast memory copy
  – If Device = Host the mirror can be a view to the same data

```cpp
array_type X = ... ; // device memory and device’s map
array_type::HostMirror X_host = create_mirror( X );
    // host memory with device’s map

deep_copy( X , X_host ); // copy data device <- host
deep_copy( X_host , X ); // copy data host <- device
```
Kokkos Array API: Parallel Functor

• Execute Functors in Parallel on Accelerator Device
  – **Functor**: A user’s C++ class bundling a function + arguments
  – **Dispatch**
    • `parallel_for(NP, functor_object);`
    • `parallel_reduce(NP, functor_object, result);`
  – Called NP times in parallel: `ip \in [0,NP)`
    • `functor_object(ip);` // `parallel_for`
    • `functor_object(ip, result);` // `parallel_reduce`

• NUMA work and data locality affinity
  – Work unit ‘ip’ performed by thread with NUMA-local data
Why we worry about NUMA

A simplified model:
Kokkos Array API:  
Example Parallel Reduce Functor

template< class Device > // template on device  
class CentroidFunctor {  
public:  
    typedef Device device_type ;  
    typedef struct { double coord[3] , mass ; } value_type ;  
    MDArray<double,device_type> m_coord , m_mass ;

    void operator()( int ip , value_type & update ) const  
    {  
        update.mass += m_mass(ip) ;
        update.coord[k] += m_coord(ip,k) * m_mass(ip,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 ; }
};
Finite-Element Mini-Applications Performance Studies

• Single Node Devices
  – Westmere: Xeon 2.93 GHz, 2 cpus X 6 cores x 2 hyperthreads
  – Magny-Cours: Opteron 2.4 GHz, 2 cpus X 8 cores
  – NVIDIA Tesla C2070: 448 cores, 1.2 GHz

• Cray XK6 testbed at Sandia (52 nodes)
  – AMD Opteron Interlagos 2.1 GHz, 16 cores / 2 NUMA regions
  – NVIDIA Tesla M2090: 512 cores, 1.3 GHz
  – Cray Gemini network

• NUMA control via HWLOC
  – http://www.open-mpi.org/projects/hwloc/
Performance-Portable
Finite-Element Mini-Applications

• Explicit Dynamics: computationally intensive
  – Element stress and internal force contributions to nodes
  – Node gather-assemble forces, apply boundary condition, compute acceleration, integrate motion
  – Accelerator device parallel

• Nonlinear Thermal Conduction: memory intensive
  – Newton iteration to solve nonlinear equation
  – Element computation of residual and Jacobian
  – Gather-assemble sparse linear system; CG iterative solver
  – Update nonlinear solution
  – MPI + Accelerator device parallel

• Same finite element kernel source code on all devices
  – Template instantiation inserts device specific array-maps
Explicit Dynamics Mini-Application
Single Node Performance Comparison

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.
Explicit Dynamics Mini-Application
NUMA Performance on Westmere

Element Computation: Impact of NUMA

- With HWLOC : float
- With HWLOC : double
- NO HWLOC : float
- NO HWLOC : double

Node Update : Impact of NUMA

- With HWLOC : float
- With HWLOC : double
- NO HWLOC : float
- NO HWLOC : double

- 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
Nonlinear Thermal Mini-Application Element Computation Performance

- No communication
  - Redundant computations at processor boundaries
- Ideal memory access
  - Coalesced
  - Cache friendly
  - NUMA locality
- Computationally intensive
Nonlinear Thermal Mini-Application Gather-Assemble Performance

- No communication
  - Redundant computations at processor boundaries
- Memory access intensive
- Random access
  - NOT Coalesced
  - NOT Cache friendly
  - Significant cross-NUMA reads
Nonlinear Thermal Mini-Application Gather-Assemble Performance

- Communication intensive
  - Sparse matrix row decomposition
  - Sparse matrix-vector multiply imports portion of column vector
  - Dot-product reductions
- Random access
  - Sparse matrix-vector multiply read of column vector
  - Significant cross-NUMA reads
- To Improve NUMA
  - Minimize cross-NUMA reads
  - Nested domain decomposition among NUMA regions
Conclusion & Plans

• Performance-Portability
  – Data access patterns are critical for performance
  – Data parallel functions on multidimensional arrays
  – Abstract & separate array map: index space ↔ device memory
  – Automatically & transparently insert device-optimal array map
  – Identical finite element code on all devices

• Plans
  – Nested domain decomposition for cross-NUMA kernels
  – Rank 2+ parallel extents, array maps with tiling
  – Intel MIC accelerator device
  – Other dispatch patterns: parallel-scan, heterogeneous functors, ...
  – Other kernel domains: stochastic finite elements, ...