KokkosArray: Multidimensional Arrays for Manycore Performance Portability

H. Carter Edwards and Christian Trott
Sandia National Laboratories

SIAM Annual Meeting
July 10, 2013 | San Diego, California

SAND2013-5312C (Unlimited Release)
Manycore Performance Portability Challenge
Diversity of devices and associated performance requirements

Device Dependent Memory Access Patterns

- Performance heavily depends upon device specific requirements for memory 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**
  - Distinct execution and memory spaces (physical or logical)
  - Dispatch parallel work to device: computation + data

- **Classic Multidimensional Arrays, *with a twist***
  - Map multi-index \((i,j,k,\ldots) \leftrightarrow\) memory location *on the device*
    - Efficient: index computation and memory use
  - 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: \(a(i,j,k,\ldots)\)

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
  ```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
  ```

- 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
    - Capability on the GPU as well
API : Allocation, Access, and Layout

- **View semantics** (shared pointer semantics)
  - Multiple view objects for the same array, shared ownership
  - Last view deallocates array data

- **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 ;
  ```
  - Use special hardware, if available
  - E.g., access ‘x’ data through GPU texture cache
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);
depth_copy(a, a_host); depth_copy(a_host, a);
```

- Advanced: avoid unnecessary deep-copy

```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**

```cpp
template< class DeviceType > // template on device type
struct AXPY {
    void operator()(int iw) const { y(iw) += a * x(iw); } // shared function
    AXPY( ... ) ... { parallel_for( nwork , *this ); }  // parallel dispatch
}
```

```cpp
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**
API : Parallel Dispatch

parallel_reduce( nwork , functor , result )

- Similar to parallel_for, with Reduction Argument
  
  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 ); }
    
    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

```cpp
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
Performance Test: Modified Gram-Schmidt
Simple stress test for bandwidth and reduction efficiency

- Simple sequence of vector-reductions and vector-updates
  - To orthonormalize 16 vectors
- Performance for vectors > L3 cache size
  - NVIDIA K20x: 174 GB/sec = ~78% of theoretical peak
  - Intel Xeon: 78 GB/sec = ~71% of theoretical peak
  - Intel Xeon Phi: 92 GB/sec = ~46% of achievable peak

Results presented here are for pre-production Intel Xeon Phi co-processors (codenamed Knights Corner) and pre-production versions of Intel’s Xeon Phi software stack. Performance and configuration of the co-processors may be different in final production releases.
Performance Test: Molecular Dynamics
Lennard Jones force model using atom neighbor list

- 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
Performance Test: Molecular Dynamics
Lennard Jones (LJ) force model using atom neighbor list

- Test Problem (#Atoms = 864k, #Steps = 100, ~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

Results presented here are for pre-production Intel Xeon Phi co-processors (codenamed Knights Corner) and pre-production versions of Intel's Xeon Phi software stack. Performance and configuration of the co-processors may be different in final production releases.
Comparing X = OpenMPI vs. Kokkos, one MPI process/device
- No native Cuda version to compare
- GPU-direct via MVAPICH2

Strong scaling test
- #Atoms = 864k
- #Steps = 100
- ~77 neighbors/atom

Xeon and Xeon Phi
- Kokkos ≈ OpenMP

Kepler
- Portability!
MPI+X Performance Test: MiniFE
Conjugate Gradient Solve of a Finite Element Matrix

- Comparing X = Kokkos, Cuda, OpenMP
- Weak scaling with one MPI process / device
  - Except OpenMP on Xeon: process/socket due to NUMA
  - GPU-direct via MVAPICH2
- Problem:
  - 3D thermal conduction
  - Compressed row storage
  - 8M elements/device
- Kokkos performance
  - 90% or better of “native”
  - Improvements ongoing
Conclusion
Performance portable manycore programming model

- Solved: “array of structs” vs. “struct of arrays”? 
  - By asking the right question: what abstractions are required? 
  - Answer: multidimensional arrays with device-polymorphic layout 
  - and coordinated parallel dispatch of computational kernels 

- Kokkos C++ library, not a language extension 
  - Performance evaluation “unit tests” and mini-applications 
  - Multicore CPU, NVidia GPU, Intel Xeon Phi coprocessor 
  - 90% or better of device-specialized “native” implementation 

- Plans 
  - Analysis and improvement of back-end implementations 
  - Advanced layouts such as tiling and blocking 
  - Aggregate “scalar” types: automatic differentiation, stochastic variables 
  - Hierarchical task-data parallelism 
  - Higher level libraries: linear algebra, tensors, containers, ... 
    - Map to device optimized libraries via template partial specialization