Manycore Performance Portability through Mapped Multidimensional Arrays

H. Carter Edwards
Sandia National Laboratories

SIAM Computational Science & Engineering 2013
February 24 – March 1, 2013
Boston Massachusetts

SAND2013-1025C (Unlimited Release)
Outline

• Two Fundamental Concepts

• KokkosArray Application Programmer Interface (API)
  – Making it *look* easy to the user

• Performance-Portability Testing
  – Hybrid parallel (MPI+KokkosArray) proxy-applications
  – Cray XK6 with NVIDIA Kepler K20X
  – Cluster of Intel Xeon Phi (MIC) Knights Corner (KNC) cards
    • Pre-production hardware
Challenge: Manycore Portability with Performance

• Multicore CPU
  – Increasing core counts and decreasing global memory / core
  – Hierarchy of shared memory (memory controllers and caches)
  – Resulting in non-uniform memory access (NUMA)
  – Increasing vector unit lengths
    ➢ Memory access patterns critical for best performance

• Manycore GPU
  – Physically separate memory with data-transfer overhead
  – Work-dispatch interaction between host and device
  – Memory controller optimized for thread-gang (warp) based access
    ➢ Memory access patterns critical for acceptable performance

Its all about Memory Access Patterns ...
Challenge: Device-Dependent Memory Access Patterns

• Memory Access Patterns are Critical
  – Correctness – no race conditions
  – Performance – proper placement, blocking, striding, …

• CPU with NUMA and vector units
  – Core-affinity placement (a.k.a. first touch)
  – Blocking for cache
  – Alignment for cache-lines and vector units

• GPU Coalesced Access
  – Alignment for cache-lines

• “Array of Structures” vs. “Structure of Arrays”? 
  ➢ Wrong question

Right question: Abstractions for Performance-Portability?
Programming Model Concept
Just two foundational ideas

• Manycore Device
  – Separate memory space (physically or logically)
  – Dispatch **work**: computation + data on the device

• Classic Multidimensional Arrays, *with a twist*
  – Map multi-index \((i,j,k,...) \leftrightarrow\) 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: \(a(i,j,k,...)\)
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; e.g., OpenMP, OpenACC, OpenCL, CUDA

• Using C++ template meta-programming
  – Device-specialized, polymorphic array layout
  – C++1998 standard (would really 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 : allocate array and access members
  
  ```cpp
  class View< double **[3][8] , Device > a("a",N,M);
  ```
  
  • Dimensioned as [N][M][3][8] (two runtime, two compile-time)
  • Aligned memory allocated in Device memory space
  
  – `a(i,j,k,l)` : access data member via multi-index

  • According to Device-specific array layout

• Advanced : choose your own array layout
  
  ```cpp
  class View<double**[3][8] , Layout , Device> a("a",N,M);
  ```
  
  ➢ Multi-index access API is unchanged for user code
  
  – Override Device’s default layout

  • E.g., force row-major or column-major

  – *Layout* is an extension point for blocking, tiling, etc.
API : View Semantics

• Basic : view (reference counting) semantics
  
  typedef class View<double** Device> MyMatrixType ;
  MyMatrixType a("a",N,M); // allocate array
  MyMatrixType b = a ; // A new 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);
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’
– depth_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); } // function
    typedef DeviceType device_type ; // run on this device
    AXPY( ... ) ... { parallel_for( nwork , *this ); }
    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 \leq nwork)
  – Thread parallel call to ‘operator()(iw)’: iw \in [0,nwork)
  – Access array data with ‘iw’ to avoid race conditions
Parallel Dispatch via Functor

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

• **Why Functor Pattern?**
  – Standard C++ and *Portable*
  – 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 ;  // type of the reduction argument
    void operator()( int iw , value_type & contrib ) const
    { contrib += y(iw) * x(iw); }
    const View<const double*,device_type> x , y ;
    DOT( … ) … { parallel_reduce( nwork , *this, result ); } // ... 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 force memory read & write among threads and
  prevent compiler from optimizing join to a ‘no op’

• Deterministic result
  – Given the same device and # threads
  – Aligned memory insures vectorization produces the same result
Ongoing R&D Within KokkosArray

• Array Layouts
  – Tiled Rank-2 Arrays (e.g., matrices)
  – Blocked, variable-blocked

• Embedded “Scalar-like” Data Types
  – E.g., `View< Type **[3][8], device >`
  – Where `Type` supports
    • Automatic differentiation
    • Stochastic variables

• Multi-Functor Dispatch
  – Heterogeneous functors operating concurrently
Performance-Portability Tests
Same source 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 “neighbor exchange” communication
Performance-Portability Tests
Same source code compiled to devices *

• ‘Curie’ testbed at Sandia
  – Cray XK6 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
Performance Evaluation: Modified Gram-Schmidt Algorithm

- CrayXK6 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
Performance Evaluation on KNC: Modified Gram-Schmidt Algorithm

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

- RW performance at “large enough” problem size
  - ~200 GB/sec “achievable” peak (pre-production hardware)
  - Full threading utilization achieved ~23% of “achievable” peak
  - MPI-per-core achieved ~13% of “achievable” peak

Modified Gram-Schmidt RW Bandwidth / device
Using 4 KNC cards

Vector length (x 16 vectors)

- 224 threads
- 112 threads
- 56 MPI
- 112 MPI
- 56 threads
Performance Evaluation on XK6: Explicit Dynamics ProxyApp

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

- Computationally intense
  - and NO communication

- Hyperthreads:
  - 56x{1-4} MPI processes / card
  - 56x{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
  - 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
  - Again, high random-memory-access intensity
- CG Time / Iteration / Row
  - Sparse-matvec random access and communication
  - Need GPU Direct to speed up communication