Photos placed in horizontal position with even amount of white space between photos and header

Photos placed in horizontal position with even amount of white space between photos and header

Sandia National Laboratories

Exceptional service in the national interest

Kokkos Tutorial

A Trilinos package for manycore performance portability

H. Carter Edwards,
Christian Trott, and
Daniel Sunderland

Trilinos User Group (TUG)

November 4, 2013

SAND2013-9404P

U.S. DEPARTMENT OF ENERGY

NNSA National Nuclear Security Administration

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. SAND NO. 2011-XXXXP

Acknowledgements and a little History

- Kokkos (classic) – internal manycore portability layer for Tpetra
  - Chris Baker – primary developer
- Kokkos (array) – package for manycore performance portability
  - Developers: Carter Edwards (PI), Daniel Sunderland, Christian Trott
  - Alpha users: Eric Phipps, Mark Hoemmen, Matt Bettencourt, Eric Cyr
  - Consultants: Mike Heroux, Si Hammond
- Current Funding
  - ASC Computational Systems and Software Environment (CSSE)

    Next-Generation Computing Technologies / Heterogeneous Computing
  - UQ-on-GPU LDRD – support for embedded UQ data types
  - Kokkos/Qthreads LDRD – task/data/vector parallelism (started Oct'13)
- Prior Funding
  - Mantevo LDRD – early concepts and prototypes

Goals: Portable, Performant, and Usable

Sandia National Laboratories

- **Portable to Advanced Manycore Architectures**
  - Multicore CPU, NVidia GPU, Intel Xeon Phi (potential: AMD Fusion)
  - Maximize amount of user (application/library) code that can be compiled without modification and run on these architectures
  - Minimize amount of architecture-specific knowledge that a user is required to have
  - Allow architecture-specific tuning to easily co-exist
  - Only require C++1998 standard compliant
- **Performant**
  - Portable user code performs as well as architecture-specific code
  - ➢ **Thread scalable – not just thread safety (no locking!)**
- **Usable**
  - Small, straight-forward application programmer interface (API)
  - ➢ **Constraint: don't compromise portability and performance**

Collection of Subpackages / Libraries

Sandia National Laboratories

- Core – lowest level portability layer
  - Parallel dispatch and multidimensional arrays for manycore devices
  - Soon to enter Trilinos "Primary Stable" status
- Containers – more sophisticated than core arrays
  - UnorderedMap – fast find and thread scalable insert
    - Very recent R&D success, thread scalable insert is a unique capability
  - Vector – subset of std::vector functionality to ease porting
  - Trilinos "Experimental" status
- LinAlg – primary interface for Tpetra
  - Sparse matrices and linear algebra operations
  - Wrappers to vendors' libraries
  - Trilinos "Experimental" status
- Examples – for this tutorial and beyond
  - Mini-applications / mini-drivers

Outline

- **Core: Fundamental Concepts**
- Core: Views to Arrays
- Core: Views to Arrays – Advanced Features
- Core: Parallel Dispatch of Functors
- Core: Parallel Correctness and Performance
- Core: Device Initialization and Finalization
- Core: Performance Evaluation
- Core: Plans
- Example: Unordered map global-to-local ids
- Example: Finite element integration and nodal summation
- Example: Particle interactions in non-uniform neighborhoods

Core: Fundamental Concepts

Sandia National Laboratories

Diversity of devices and associated performance requirements

- **Performance heavily depends upon device specific requirements for memory access patterns**
  - **Blocking, striding, alignment, tiling, ...**
  - **NUMA core-memory affinity requires first touch and consistent access**
  - **CPU vector units require stride-one access and cache-line alignment**
  - **GPU vector units require coalesced access and cache-line alignment**
- **"Array of Structures" vs. "Structure of Arrays" dilemma**
  - ➢ **This has been the *wrong* question**
- **What abstraction required for performance portability?**
  - **This is the *right* question**
  - **Answer: multidimensional arrays with polymorphic layout**

Core: Fundamental Concepts

Sandia National Laboratories

Two abstractions: (1) Host/Devices

- **Host process dispatches work to manycore device(s)**
  - Host process is the 'main' function
  - Host processes dispatches thread-parallel work to device

    work is computation and data
  - "Device" may be physical (e.g., GPU) or logical:

    partition 16core CPU into 1core "host process" and 15core "device"
- **Host process interacts with MPI, Kokkos does not**
  - "MPI+X" : Kokkos is a potential "X"
  - Kokkos is orthogonal to MPI – devoid MPI datatypes and calls to MPI
- **Multiple memory spaces**
  - Disparate: host main memory vs. GPU on-card memory
  - Integrated: main memory, L3/L2/L1 cache, registers
  - Anticipate increasing complexity of memory architectures

Core: Fundamental Concepts

Two abstractions: (2) Multidimensional Arrays

- **Multidimensional Arrays, *with a twist***
  - Map multi-index (i,j,k,...) ↔ memory location *on the device*
    - Efficient : index computation and memory use
  - Map is derived from an array Layout
  - ➢ Choose Layout for device-specific (optimal) 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

Core: Fundamental Concepts

Implementation and similar work

- **Implemented C++ template meta-programming**
  - Compile-time polymorphism for device back-ends and array layouts
  - C++1998 standard; would be nice to *require* C++2011 for lambdas, ...
- **Similarly motivated libraries:**
  - Intel's TBB: more sophisticated parallel dispatch capabilities,

    CPU only, no data structure abstractions
  - NVIDIA's Thrust: similar simple parallel dispatch capabilities, only vector data structures, no array layout
  - MS C++AMP: close, but uses a proprietary language extension
- **Language extensions: OpenMP, OpenACC, OpenCL, CUDA, Cilk,**
  - Lacking data structure abstractions to manage access patterns

Outline

- Core: Fundamental Concepts
- **Core: Views to Arrays**
- Core: Views to Arrays – Advanced Features
- Core: Parallel Dispatch of Functors
- Core: Parallel Correctness and Performance
- Core: Device Initialization and Finalization
- Core: Performance Evaluation
- Core: Plans

Core : Views to Arrays

Sandia National Laboratories

View to multidimensional array of “value” type in device memory

- **View< double * * [3][8] , Device > a ;**
  - template class View: A view to an Array on a Device
  - Runtime and compile-time dimensions: example [N][M][3][8]
- “value” type of an array : ~ plain-old-data (pod) type
  - E.g., ‘double’ (in this example), ‘float’, ‘int’, ‘long int’, ...
  - A pure ‘memcpy’ will have the correct result
  - ➢ Does not contain pointers to allocated memory
- ArraySpec template argument = ‘double**[3][8]’ in this example
  - Each ‘*’ denotes a runtime specified dimension
  - Each ‘[#]’ denotes a compile-time specified dimension
  - 0-8 runtime dimensions denoted by ‘*’
  - 0-8 compile-time dimensions denoted by [#]
  - Up to 8 runtime + compile-time dimensions (maximum rank)

Core : Views to Arrays

Sandia National Laboratories

View to multidimensional array of “value” type in device memory

- **View< ArraySpec , Device > a ;**
  - Query dimensions: a.dimension_#() OR a.dimension(#) where # ∈ [0..7]
- **Why runtime + compile-time dimensions? PERFORMANCE!**
  - Array layout computation is faster with compile-time dimensions
  - ➢ If a dimension is known at compile time then specify it
- **Advanced feature: support for aggregate “value” types**
  - Intrinsic “value” type required for optimal array layout
  - but we need ‘complex’ and other aggregate “value” types
  - ... more on this later ...

Core : Views to Arrays

Accessing array data members: a(i0,i1,i2,i3,...)

- Access array data via 'View::operator()'

```
template< typename intType0 , typename intType1 , ... >
ValueType & View::operator()( const intType0 & , const intType1 & , ... );
```

  - Multi-index is mapped according to the array layout
  - Layout chosen to give the best memory access pattern for the device

    Assuming first index is the parallel work index ... more on this later ...

- DO NOT assume a particular array layout (mapping)
  - Might be FORTRAN, might be C, might be something else entirely
  - Chosen at compile-time (C++ template meta-programming)
  - Advanced feature: query the array's layout
  - Advanced feature: override the layout

Core : Views to Arrays

Accessing array data members: a(i0,i1,i2,i3,...)

Sandia National Laboratories

- **Multi-index mapping performance**
  - Heavily used and critical to performance
  - Considerable development effort invested in performance

    Especially so compilers' vectorization can "see through" this operator
  - Completely hidden, non-trivial C++ meta-programming implementation
  - Compile-time dimensions improve multi-index mapping performance
- **Correctness checking: accessible and within bounds**
  - Host not able to access Device memory (and vice-versa)
  - Multi-index bounds checking – in debug mode, and on the GPU

Core : Views to Arrays

Allocation and reference-counting semantics

- View objects are light-weight references to allocated arrays
- Allocate: `View< double * * [3][8] , Device > a("A",N,M);`
  - Dimension [N][M][3][8] ; two runtime, two compile-time
  - "A" is a user supplied label used for error messages; need not be unique
  - Allocated array data resides in the Device's memory space
  - Object 'a' is a reference to allocated array data
- Assign: `View<double**[3][8],Device> b = a ;`
  - Object 'b' is a reference to the same allocated data; a shallow copy
  - By default views to arrays are reference counted
- Destroy: view object goes out-of-scope or is reassigned
  - Last view (via reference counting) deallocates array data

Core : Views to Arrays

Resizing and reallocation

- Given: `View< ArraySpec , Device > a("label",m0,m1,...);`
- Resize: `resize( a , n0 , n1 , ... );`
  - Allocate a new array with "label" and size n0*n1*...
  - Copy corresponding array data from original array to new array
  - Reassign the input View to the new array
  - ➢ All other views to the original array are unchanged
- Reallocate: `realloc( a , n0 , n1 , ... );`
  - De-assign the input View; if last reference then array is deallocated
  - Assign input view to an allocated array with "label" and size n0*n1*...
  - ➢ All other views to the original array are unchanged
  - If no other view to original array this deallocates before allocating, avoids "spike" in allocated memory

Core : Views to Arrays

'const' Views versus 'const' Arrays

- **Constant View: const View< ArraySpec , Device > a(...);**
  - Object 'a' cannot be reassigned
  - Array data can be assigned via parentheses operator
  - Analogous to const pointer to non-const memory
- **Constant Array: View< const ArraySpec , Device> b = a ;**
  - Object 'b' is a reference to the same allocated data; a shallow copy
  - Array data cannot be assigned – parentheses operator returns 'const'
  - Analogous to non-const pointer to const memory
- **Assignment (shallow copy) compatibility**
  - OK : View< const ArraySpec, Device > = View< ArraySpec , Device >
  - ERROR : View< ArraySpec , Device > = View< const ArraySpec , Device >

    this will not compile with "no assignment operator" message

Core : Views to Arrays

Pass view objects by value – they are small and portable

- **Pass view objects by value**

```
typedef View< ArraySpec , Device > my_array_type ;
void my_function( my_array_type A );   // no & or *
struct my_struct { my_array_type A ; }; // no & or *
```

- **Small – designed as references to allocated array data**
  - Pointer to data + array shape (dimensions)
  - Assignment is a fast shallow copy + reference counting (by default)
- **Portable – intended to be passed by value to the device**
  - View object API is portable between Host and Device code
- **Do not pass by reference (or pointer) from Host to Device**
  - The reference / pointer is in the Host memory space
  - Using such a Host pointer on the Device is a memory error

Core : Views to Arrays

Deep copy: Kokkos NEVER has a hidden, expensive deep-copy

- Deep copy array data *only* when explicitly instructed by user
  - `deep_copy( to_array , from_array );`
- Problem: deep copy between different array layouts
  - Same memory space – requires permutation
  - Different memory spaces – also requires allocation of a temporary

    very expensive: allocation + deep copy + permutation + deallocation
- Solution: Mirror the layout in the Host memory space
  - Avoid allocation, permutation, and deallocation

```
View<ArraySpec,Device> a(...);
View<ArraySpec,Device>::HostMirror b = create_mirror( a );
```

  - ‘ ‘b’ has the Device’s array layout but is allocated in the Host space

Core : Views to Arrays

Deep copy: Kokkos NEVER has a hidden, expensive deep-copy

- **Device ↔ Host deep copy pattern:**

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

  - Issue: if 'a' is already in the Host space then allocation of 'a_host' and subsequent deep_copy operations are problably unecessary

- **Avoiding an unnecessary allocation and deep-copy**

```
MyViewType::HostMirror  a_host = create_mirror_view( a );
```

  - If Device uses Host memory then 'a_host' is simply another view of 'a'
  - Call to deep_copy becomes a no-op

Core : Views to Arrays

Recommendation: Dictionary for your View types

```
template< class Device >
struct MyDictionary {
  typedef View< ArraySpec_A , Device > array_A_type ;
  typedef View< ArraySpec_B , Device > array_B_type ;
  typedef View< ArraySpec_C , Device > array_C_type ;
  typedef typename array_A_type::HostMirror array_A_host_type ;
};
```

- Consolidate array type definitions
  - Documentation
  - Consistency
  - Allows single point of change for array spec and array layout

Outline

- Core: Fundamental Concepts
- Core: Views to Arrays
- **Core: Views to Arrays – Advanced Features**
- Core: Parallel Dispatch of Functors
- Core: Parallel Correctness and Performance
- Core: Device Initialization and Finalization
- Core: Performance Evaluation
- Core: Plans

Core : Views to Array – Advanced Features

Optionally specifying a particular array layout

- **View< ArraySpec , Layout , Device >** (optional parameter)
  - Override default layout; e.g., force row-major or column-major
  - ➢ Access via parentheses operator is unchanged in user code
- Standard array layouts for arrays with rank > 1
  - LayoutRight : right-most index is stride-one (~ C ordering)
  - LayoutLeft : left-most index is stride-one (~ FORTRAN ordering)
  - Array dimensions may be padded for cache-line alignment
    - Analogous to 'LDA' matrix parameter in the BLAS
- ➢ *Layout* is an extension point for tiling, blocking, etc.
  - A research-enabling capability
  - Prototype exists for tiled matrices (e.g., MAGMA / PLASMA)

Core : Views to Array – Advanced Features

Specifying behavioral attributes

- **Disable reference counting**

  View< ArraySpec , Device , Unmanaged >

  - Cannot allocate through an unmanaged view
  - Can assign an unmanaged view from a managed view
  - Can assign an unmanaged view from user-provided pointer
  - Dangerous advanced feature unlikely to significantly impact performance

- **Use GPU texture cache to speed up random access**

  View< const ArraySpec , Device, RandomRead >

  - If Device == 'Cuda' then parentheses operator uses GPU texture cache
  - Otherwise no special handling

➢ **An extension point**

Core : Views to Array – Advanced Features

Assignment of compatible views with behavioral attributes

- **Compatible assignment is a shallow copy**

  View< ArraySpec , Device , Attribute > = View< ArraySpec , Device >

  - **Compatible: same ‘ArraySpec’, ‘Device’, and ‘Layout’**
  - **Also OK: ‘const ArraySpec’ = ArraySpec**
  - **Also OK: Different devices using the same memory space**

- **Recommendation**
  - **Initially declare ‘view’ without behavior attributes**
  - **Add behavioral attributes via shallow copy to compatible view**

Core : Views to Array – Advanced Features

Aggregate value types

- Examples of aggregate value types (pod ‘struct’)
  - std::complex
  - Automatic differentiation types
  - Stochastic bases coefficients types
- Memory access pattern for aggregate members
  - Is forced to be an ‘array of structures’
  - Loses coalesced memory access on GPU – degrades performance
- Active research within UQ-on-GPU LDRD
  - View integrates aggregate value types into the array layout
  - Compile-time conversion ‘array of structures’ to ‘structure of arrays’
  - Recover required memory access pattern on GPU

Core : Views to Array – Advanced Features

Aggregate value types

- **Capabilities and Constraints**
  - **"scalar" type must be mappable to an array of an intrinsic type**

    E.g., std::complex<T> ↔ T[2]
  - **For a given View the mapping may have a consistent runtime dimension**

    E.g., View< myType<T> > : myType<T> ↔ T[#]
- **Extension point requires detailed implementation knowledge**
  - **Optimal performance of View::operator()**
  - **Optimal memory access pattern**

    Requires merging the aggregate type's array mapping into the containing View's array layout
- **Path forward to performantly support complex<T>**
  - **... to be done ...**

Core : Views to Array – Advanced Features

Sandia National Laboratories

Querying properties

```
View::device_type  // Device in View< ArraySpec , Device >
View::data_type    // ArraySpec in View< ArraySpec , ... >
View::value_type   // ValueType in View< ValueType***[#][#][#], ... >
View::scalar_type  // For intrinsic ValueType is ValueType
                   // For aggregate ValueType is the mapped intrinsic type
View::const_{}_type       // const added to previous {}_type
View::non_const_{}_type // const removed from previous {}_type

View::array_layout   // Layout type; e.g., LayoutLeft, LayoutRight
View::rank           // total number of dimensions (one added for aggregate)
View::rank_dynamic // number of dynamic dimensions
View::is_managed     // enumerated value if view is reference counted

View::scalar_type * View::ptr_on_device(); // Raw pointer to array data
```

Core : Views to Array – Advanced Features

View ↔ pointer to raw memory

- **Wrapping your memory in a View**
  - **You must specify everything**

    View< ArraySpec, Layout, Device, Unmanaged > a(pointer, N0, N1, ...);
  - **Unmanaged: Kokkos cannot manage your memory**
  - **Device: Your memory must be on this device**
  - **{ ArraySpec , Layout , N0 , N1 , ... }: your memory must have this shape**
- **Interoperability with legacy codes' arrays**
  - **Option 1: Wrap your memory in a View**
  - **Option 2:**
    - **Declare Views with your specified array layout**
    - **Use 'View::ptr_on_device()' to query pointer and pass to legacy code**

Outline

- Core: Fundamental Concepts
- Core: Views to Arrays
- Core: Views to Arrays – Advanced Features
- **Core: Parallel Dispatch of Functors**
- Core: Parallel Correctness and Performance
- Core: Device Initialization and Finalization
- Core: Performance Evaluation
- Core: Plans

Core : Parallel Dispatch of Functors

Dispatch to manycore "Device"

Sandia National Laboratories

- 'Threads' Device : pthreads
  - Pool of threads created once and pinned to cores
  - Hardware detection and core pinning via hardware locality library (hwloc)
  - CPU and Intel Phi
- 'OpenMP' Device : wrapper on OpenMP
  - Attempt to pin to cores via hwloc
  - CPU and Intel Phi
  - Cannot use both 'Threads' and 'OpenMP' – they will compete for cores
- 'Cuda' Device : wrapper on NVidia's CUDA 5.0 (or better)
  - Currently require Fermi (GPU capability 2.0 or better)
  - Eventually require Kepler (GPU capability 3.5 or better)
- Intel Phi used in native mode (no offload)

Core : Parallel Dispatch of Functors

Sandia National Laboratories

Functor: function + calling arguments packaged in a C++ class

- Common to C++ standard algorithms, Intel TBB, NVidia Thrust
- Functor interface requirements for Kokkos

```
template< class Device > // template on the device
struct MyFunctor {
  typedef Device device_type ; // Required: identify the device
  KOKKOS_INLINE_FUNCTION  // Required: macro mapped to device
    void operator()( ... ) const { /* ... */ } // Required: function to call in parallel
  /* ... calling arguments are members of the class ... */
};
```

- Why Functor pattern?
  - Requires only C++1998 standard compliance
  - C++2011 Lambda syntax would be much prettier ...

Core : Parallel Dispatch of Functors

Sandia National Laboratories

Functor: function + calling arguments packaged in a C++ class

- **Functor object is copied to the device**
  - This includes class member 'calling arguments'
  - View members must be objects

    Not references or pointers to Views (or anything else)
  - View objects are designed to be copied by value from Host to Device
- **Device's threads concurrently call Functor::operator()**
  - Functor::operator() and all functions that it calls
    - Must be compiled for that device
    - Must be marked with KOKKOS_INLINE_FUNCTION
      - Compiling Cuda: "__device__ __host__ inline"
  - A single Functor object is shared among all threads
    - functor::operator() must be 'const'
    - All called member functions must be 'const'

Core : Parallel Dispatch of Functors

parallel_for dispatch with 'nwork' units of work

Sandia National Laboratories

- **Simple example: AXPY (y = a * x + y)**

```
template< class Device >
struct AXPY {
  typedef Device device_type ; // run on this device
  KOKKOS_INLINE_FUNCTION
    void operator()( int iw ) const { Y(iw) += A * X(iw); }
  const double A ;
  const View<const double*,device_type> X ; // View object (not a reference)
  const View<       double*,device_type> Y ;
};
parallel_for( nwork , AXPY<device>( a , x , y ) );
```

  - Thread parallel call to 'operator()(iw)' : iw ∈ [0,nwork)
  - Access array data with 'iw' to avoid thread race conditions

Core : Parallel Dispatch of Functors

Asynchronous dispatch

- **Parallel dispatch initiates asynchronous parallel execution**
  - **'parallel_for' returns before the functor completes**
  - **Device (e.g., Cuda) can have a work queue**
    functor may be placed in queue and not even started
  - **Dispatch creates a temporary internal copy of the functor**
    released when the functor completes
- **Dispatched functors are sequenced**
  - **Previous functor guaranteed to complete before next functor starts**
  - **deep_copy(...) waits for previous functor to complete**
- **Device::fence(); // wait for all functors to complete**
  - **Required when timing the execution of a functor**

Core : Parallel Dispatch of Functors

parallel_reduce dispatch with 'nwork' units of work

- **Simple example: DOT**

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

  - value_type can be a scalar, 'struct', or dynamic array
  - Result is output to the Host

Core : Parallel Dispatch of Functors

parallel_reduce dispatch with 'nwork' units of work

- Initialize and join threads' individual contributions

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

  - Join threads' contrib via 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

Core : Parallel Dispatch of Functors

parallel_reduce dispatch with on-device serial finalization

- **Example: NORM2, just add a final 'sqrt' to the DOT**

```
struct NORM2 {  // ... similar to 'DOT' plus serial finalization
  KOKKOS_INLINE_FUNCTION
    void final( value_type & contrib ) const
      { *result = sqrt( contrib ) ; }          // final serial 'sqrt' on device
   View<double,device_type> result ; // scalar value allocated on device
 };
```

  - If result is needed only on the device, avoid device-host-device copy
  - If final serial computation is needed

Core : Parallel Dispatch of Functors

parallel_scan dispatch with 'nwork' units of work

```
template< class Device >
struct ExclusivePrefixSum {
  typedef  DeviceType   device_type ;
  typedef long int value_type ;  // Require: reduction value type
  KOKKOS_INLINE_FUNCTION
    void operator()( int iw , value_type & contrib , bool final ) const
      {
        contrib += x(iw);
        if ( final ) { y(iw) = contrib ; } // Is scan value IF final pass
      }
  const View<long int *,device_type> x , y ;
  // ... to be continued ...
};
parallel_scan( nwork , ExclusivePrefixSum<device>(x,y) ); }
```

Core : Parallel Dispatch of Functors

parallel_scan dispatch with 'nwork' units of work

- Initialize and join threads' individual contributions
  - Same 'init' and 'join' as the 'parallel_reduce'

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

- Deterministic result ← highly desirable
  - Given the same device and # threads
  - Aligned memory prevents variations from vectorization

Core : Parallel Dispatch of Functors

Thread teams – very new capability and being refined

- Device has teams of threads
  - OpenMP 4.0 vocabulary: team of threads, league of teams
  - # Threads = # threads/team * # teams
  - A team works cooperatively and shares resources; e.g., cache memory

```
template< class Device >
struct MyFunctor {
  KOKKOS_INLINE_FUNCTION void operator()( Device dev , ... ) const ;
  size_t  shmem_size() const ; // Optional request for team-shared memory
};
parallel_{for,reduce,scan}( ParallelWorkRequest , MyFunctor<device>( ... ) );
```

- More complex and more control over performance
- WorkRequest *requests* league and team sizes
  - Actual sizes may be constrained by device's capabilities
  - E.g., maximum team size limited by NUMA, #cores, #hyperthreads

Core : Parallel Dispatch of Functors

Sandia National Laboratories

Why thread teams? Opportunity for Performance Improvements

- **Threads within a team are tightly coupled**
  - E.g., NVidia thread block = team
  - E.g., Intel hyperthreads reside within the same team
  - Teams have synchronization primitives (e.g., barrier)
  - Teams have fast transient team-shared memory
- **Uncooperative teams impede performance**
  - Threads within a team will thrash their shared cache

    Cause eviction of each other's cached memory
  - Intel Phi performs better without hyperthreads *IF* they do not cooperate
  - Intel Phi performs best with cooperating hyperthreads
  - NVidia has dramatic performance loss with uncooperative teams

Core : Parallel Dispatch of Functors

Thread teams API: parallel_for, parallel_reduce, parallel_scan

```
template< class Device >
struct MyFunctor {
  KOKKOS_INLINE_FUNCTION void operator()( Device dev , ... ) const
  {
     dev.league_rank();        // Which team within the league
     dev.league_size();        // How many teams in the league
     dev.team_rank();          // Which thread within the team
     dev.team_size();          // How many threads within the team
     dev.team_barrier();       // Synchronize threads within this team
     i = dev.team_scan( n );   // Exclusive scan within this team
     view_type a( dev , N0 , N1 , ... ); // Temp array in team-shared memory
  }
};
```

- Team-shared memory used == MyFunctor::shmem_size()

Outline

- Core: Fundamental Concepts
- Core: Views to Arrays
- Core: Views to Arrays – Advanced Features
- Core: Parallel Dispatch of Functors
- **Core: Parallel Correctness and Performance**
- Core: Device Initialization and Finalization
- Core: Performance Evaluation
- Core: Plans

Core : Parallel Correctness and Performance

Sandia National Laboratories

Avoid thread race conditions

- Parallel dispatch of functor ‘f’ for ‘nwork’ units of work
  - Call f::operator()(iw) where iw ∈ [0,nwork)
  - Calls can be concurrent and in any order
- Don’t have competing updates

```
operator()( int iw ) const { y( iw / 2 ) = ( x(iw) + x(iw+1) ) * 0.5 ; }
```

  - Bad: last thread wins → random result
  - Ugly: concurrent update → corrupted result
- Don’t read what is updated elsewhere

```
operator()( int iw ) const { y(iw+1) = y(iw) + x(iw) ); }
```

  - Bad: last thread wins → cumulative random results
  - Ugly: concurrent update → compounding corrupted results

Core : Parallel Correctness and Performance

Sandia National Laboratories

Parallel reductions to mitigate thread race conditions

- parallel_reduce(nwork , f , & result);

  operator()(int iw , value_type & val) const { val += x(iw) + x(iw) ; }

  - Kokkos orchestrates temporaries, functor calls, and 'join' calls
  - Reduction is thread-safe, deterministic, and O(log(#threads))
- Mapped reduction (scatter-reduce) problem:

  operator()(int iw) const { y(imap(iw)) += x(iw); }

  - Caveat: nondeterministic order → round-off for non-associativity
  - Ugly: concurrent update Y(imap(iw)) → corrupted result
- Mapped reduction solutions:
  - Atomic operations prevent corrupted result

    Still have round-off. Possibly introduce performance bottleneck.
  - Rewrite algorithm as gather-reduce

    Mitigate round-off. Create large temporary array.

Core : Parallel Correctness and Performance

Sandia National Laboratories

Atomic operations with best performance

- Not the C++11 'atomic<T>' functionality and interface
- Three fundamental operations on intrinsic data types
  - 32 and 64 bit integer and floating point types,
  1. old_val = atomic_exchange(address, new_val);
  2. atomic_compare_exchange_strong(address, old_val , new_val);
     - If *address == old_val then exchange
  3. old_val = atomic_fetch_add(address , value);
     - old_val = *address ; *address += value ;
- Likely to have non-deterministic results ← warning!
  - Non-deterministic ordering of atomic operations
  - Floating point addition is NOT associative
- Expect atomics to be at least 2-3x slower that non-atomic

Core : Parallel Correctness and Performance

Sandia National Laboratories

Atomic operations can introduce performance bottleneck

- parallel_for(nwork , Dot<...>(x,y));

```
operator()( int iw ) const { atomic_fetch_add( &val , x(iw) * y(iw) ); }
```

  - Every thread attempts to update the same value
  - Reduction becomes fully serialized: O(#nwork) vs. O(log(#threads))
- Mapped reduction (scatter-reduce):

```
operator()( int iw ) const { atomic_fetch_add( &y(imap(iw)), x(iw)); }
```

  - Update is partially serialized depending upon
    - “Density” of imap(*)
    - Capabilities of atomic units
  - Can be a performant solution given sparse and infrequent updates

Core : Parallel Correctness and Performance

Sandia National Laboratories

Avoid long divergent branches within a thread team

- **Branches impede vector-parallelism and thus performance**

```
void operator()( int iw ) const
{
  if ( condition_A(iw) ) { ... }
  else if ( condition_B(iw) ) { ... }
  else if ( condition_C(iw) ) { ... }
  else { ... }
}
```

  - The entire vector unit (GPU warp) takes every branch
  - Branches to complex: compiler may not be able to vectorize
- **Performant if a team of threads follows the same branch**
  - Different teams can follow different branches
  - Work space iw $\in$ [0,nwork) is partitioned among teams;
    iw and iw+1 are typically in the same team

Core : Parallel Correctness and Performance

Sandia National Laboratories

Avoid redundant access to global memory, use local temporaries

- **Example: Gather finite element's nodal coordinates**

```
void operator()( int ielem ) const
{
  double node_coord[N][3] ;
  for ( int j = 0 ; j < N ; ++j ) {
    const int inode = view_elem_node(ielem,j);
    for ( int k = 0 ; k < 3 ; ++k ) node_coord[j][k] = view_node_coord(j,k);
  }
  /* ... computation uses node_coord ... */
```

- **A performance balancing act**
  - Redundant access to global memory is expensive
  - Local temporaries consumes registers & L1 cache
    threads can compete for registers & thrash each others cache
  - Vendors' diagnostic tools for performance tuning
  - Thread-team algorithms to potentially improve performance

Core : Parallel Correctness and Performance

Sandia National Laboratories

Strided and random access to global memory

- **Parallel read/write of global View data: a(iw,i1,i2,...)**
  - **Leading index is the parallel work index**
  - **Array layout + work↔thread mapping chosen together for optimal memory access pattern**
    - **CPU (and Intel Phi) caching and vectorization**
    - **GPU (e.g., NVidia) warp coalescing**
- **Random read of global View data**
  - **E.g., gathers and tables shared among threads**
  - **View< const ArraySpec , Device , RandomRead >**
    - **Cuda uses texture-fetch capability optimized for random access**

Outline

- Core: Fundamental Concepts
- Core: Views to Arrays
- Core: Views to Arrays – Advanced Features
- Core: Parallel Dispatch of Functors
- Core: Parallel Correctness and Performance
- **Core: Device Initialization and Finalization**
- Core: Performance Evaluation
- Core: Plans

Core : Device Initialization and Finalization

Sandia National Laboratories

Hardware locality (hwloc) for manycore CPU and Xeon Phi

- **Kokkos::hwloc Wraps OpenMPI project's HWLOC library**
  - **Portable query of core topology**
  - **Portable pinning of threads to cores**
- **Capacity = #NUMA * #core/NUMA * #hyperthreads/core**
  - **hwloc::get_available_numa_count()**
  - **hwloc::get_available_cores_per_numa()**
  - **hwloc::get_available_threads_per_core()**

Core : Device Initialization and Finalization

Sandia National Laboratories

Threads and OpenMP devices for manycore CPU and Xeon Phi

```
Device::initialize( team_count , threads_per_team ,
                    use_numa_count = 0, use_cores_per_numa = 0);
```

- Default: use all available NUMA regions and cores
- Each team is assigned a set of cores within a NUMA region
  - Spawn and pin team's threads to these cores
- A team's threads are spread across its cores
  - Team has 4 cores and 4 threads then 1 thread/core
  - Team has 2 cores and 8 threads then 4 threads/core
  - Don't define threads/core > hwloc::core_capacity()
- Device::finalize()
  - Destroy spawned threads

Core : Device Initialization and Finalization

Sandia National Laboratories

Cuda Device

- **Cuda::initialize() OR Cuda::initialize(Cuda::SelectDevice(#))**
  - **Default is device #0**
- **Only one Cuda device per MPI process**
  - **Given two devices on a node use two MPI processes**
  - **Each MPI process on the node should select a different device**
  - **NVidia Kepler devices can be shared (have not tried this)**
- **Query available devices**
  - **std:vector<unsigned> Cuda::detect_device_arch()**
  - **Values match __CUDA_ARCH__ specification**

Outline

- Core: Fundamental Concepts
- Core: Views to Arrays
- Core: Views to Arrays – Advanced Features
- Core: Parallel Dispatch of Functors
- Core: Parallel Correctness and Performance
- Core: Device Initialization and Finalization
- **Core: Performance Evaluation**
- Core: Plans

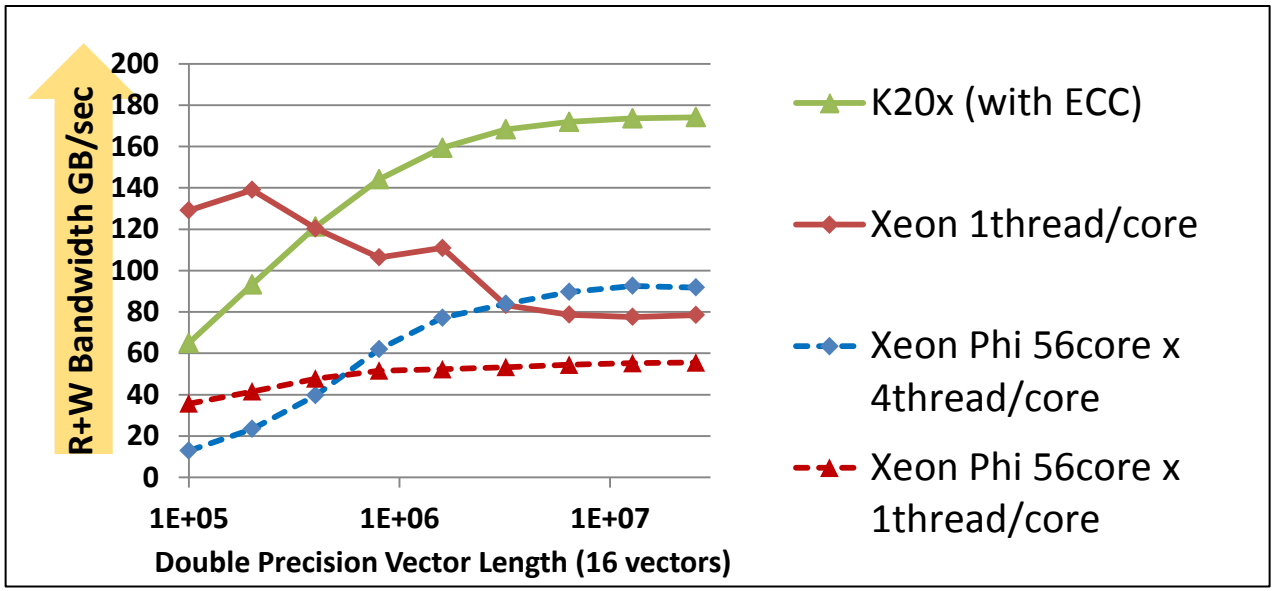
Performance Evaluation

Sandia National Laboratories

- **Using Sandia Computing Research Center Testbed Clusters**
  - **Compton: 32nodes**
    - **2x Intel Xeon E5-2670 (Sandy Bridge), hyperthreading enabled**
    - **2x Intel Xeon Phi 57core (pre-production)**
    - **ICC 13.1.2, Intel MPI 4.1.1.036**
  - **Shannon: 32nodes**
    - **2x Intel Xeon E5-2670, hyperthreading disabled**
    - **2x NVidia K20x**
    - **GCC 4.4.5, Cuda 5.5, MVAPICH2 v1.9 with GPU-Direct**
- **Absolute performance "unit" tests**
  - **Evaluate parallel dispatch/synchronization efficiency**
  - **Evaluate impact of array access patterns and capabilities**
- **Mini-application : Kokkos vs. 'native' implementations**
  - **Evaluate cost of portability**

Performance Test: Modified Gram-Schmidt

Simple stress test for bandwidth and reduction efficiency



Intel Xeon: E5-2670 w/HT
Intel Xeon Phi: 57c @ 1.1GHx
NVidia K20x

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.

- **Simple sequence of vector-reductions and vector-updates**
  - **To orthonormalize 16 vectors**
- **Performance for vectors > L3 cache size**
  - **NVDIA 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**

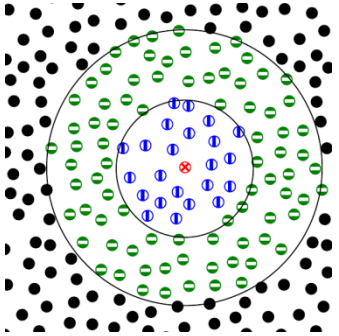
Performance Test: Molecular Dynamics

Lennard Jones force model using atom neighbor list

Sandia National Laboratories

- **Solve Newton's equations for *N* particles**
- **Simple Lennard Jones force model:** $F_i = \sum_{j, r_{ij} < r_{cut}} 6\varepsilon \left[\left(\frac{\varsigma}{r_{ij}} \right)^7 - 2 \left(\frac{\varsigma}{r_{ij}} \right)^{13} \right]$
- **Use atom neighbor list to avoid N^2 computations**

```
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;
```



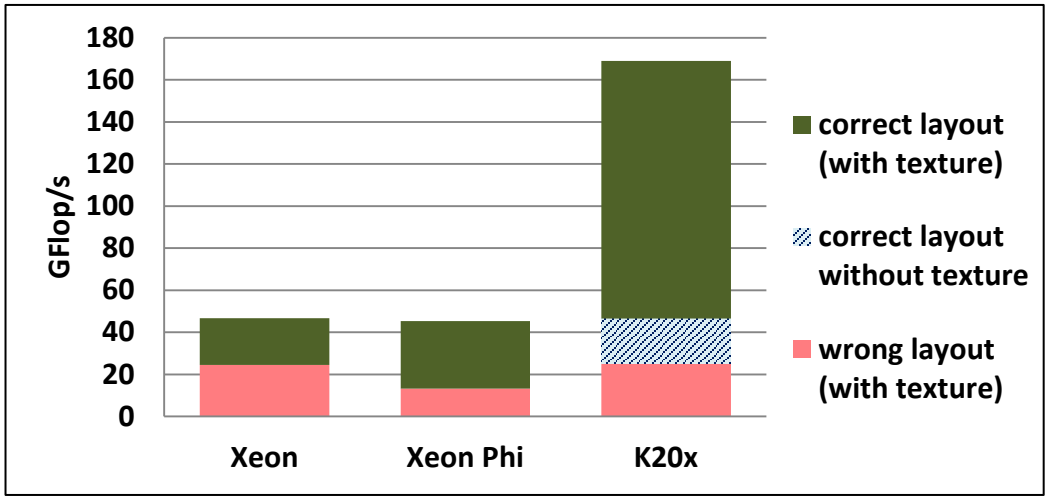
- **Moderately compute bound computational kernel**
- **On average 77 neighbors with 55 inside of the cutoff radius**

Performance Test: Molecular Dynamics

Lennard Jones (LJ) force model using atom neighbor list

Sandia National Laboratories

- **Test Problem (#Atoms = 864k, ~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**



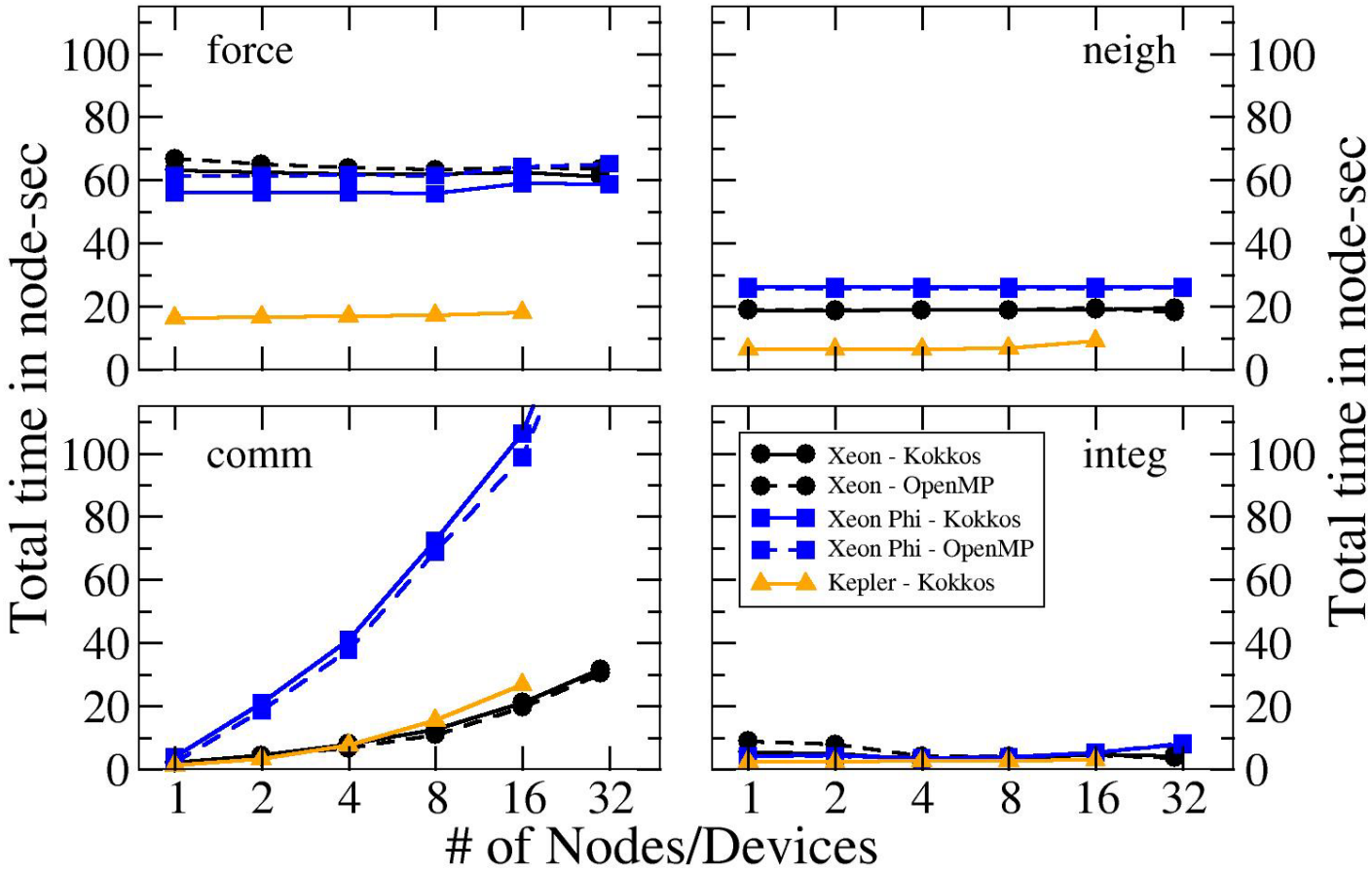
- **Large loss in performance with wrong layout**
  - **Even when using GPU texture fetch**

Intel Xeon: E5-2670 w/HT
Intel Xeon Phi: 57c @ 1.1GHx
NVidia K20x

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.

MPI+X Performance: MiniMD

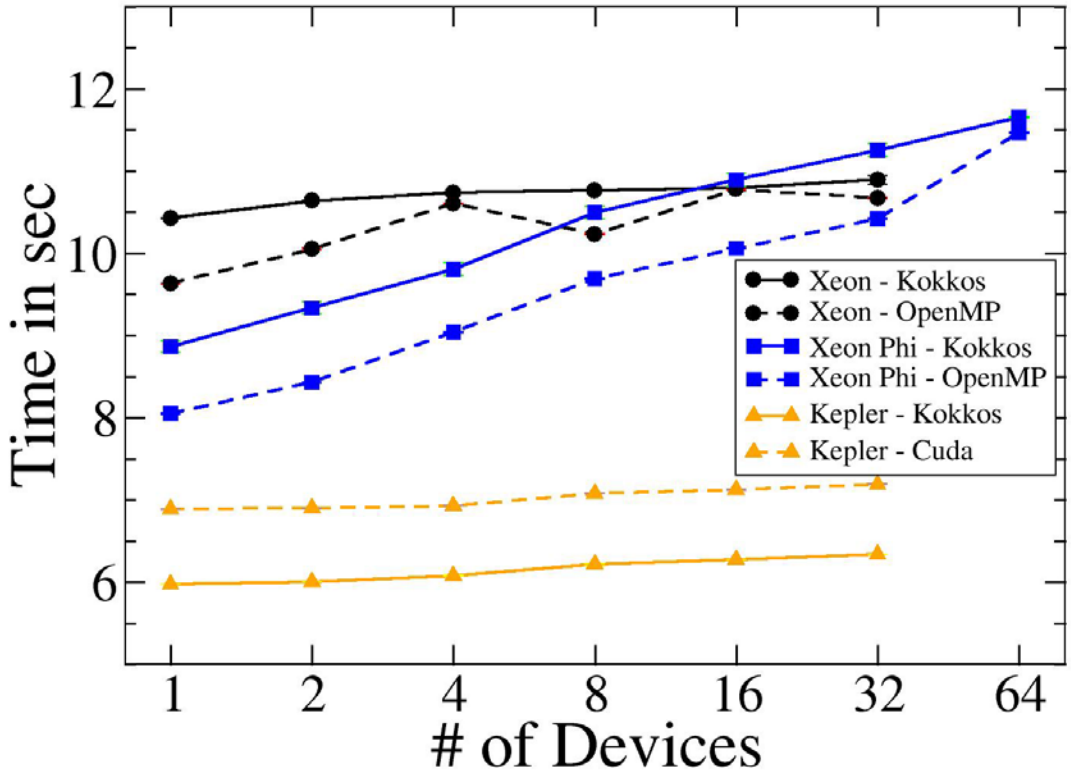
- Comparing X = OpenMPI vs. Kokkos , one MPI process / device
  - Using GPU-direct via MVAPICH2; no native Cuda version to compare
- Strong scaling test: 2,048k atoms, ~77 neighbors/atom



MPI+X Performance Test: MiniFE

Conjugate Gradient Solve of a Finite Element Matrix

- Comparing X = Kokkos, OpenMP, Cuda (GPU-direct via MVAPICH2)
- Weak scaling with one MPI process per device
  - Except on Xeon: OpenMP requires one process/socket due to NUMA
  - 8M elements/device
- Kokkos performance
  - 90% or better of "native"
  - Improvements ongoing



Outline

- Core: Fundamental Concepts
- Core: Views to Arrays
- Core: Views to Arrays – Advanced Features
- Core: Parallel Dispatch of Functors
- Core: Parallel Correctness and Performance
- Core: Device Initialization and Finalization
- Core: Performance Evaluation
- **Core: Plans**

Core : Plans

Research & development

- Mantevo mini-applications (mini-drivers)
- Functor::operator()(Device) interface
  - Portable access to Cuda block & shared memory capabilities
  - Team collectives under development
  - Prototyped with 'Cuda' and 'Threads' devices
- Aggregate scalar types
  - complex, stochastic, automatic differentiation
- Generalize tiled (blocked) layouts
- Task-data-vector unified parallelism: Kokkos/Qthreads LDRD
  - Enhance Kokkos API to parallel dispatch task-graph of functors
  - Enhance Qthreads to schedule functors on teams of threads
  - Views for threaded graph data structures and algorithms
  - Make it all portable and performant (Xeon Phi and GPU)

Core : Plans

Sandia National Laboratories

Incremental migration strategy for C++ applications and libraries

- **Replace array allocations with Views (in Host space)**
  - Specify layout(s) to match existing array layout(s)
  - Extract pointers to allocated array data and use them in legacy code
- **Replace array access with Views**
  - Replace legacy array data structure(s) with View
  - Access data members via View API
- **Replace functions with Functors, run in parallel on Host**
  - Hard part: finding and extracting your functions' hidden states
    - ➢ improve code quality
  - Hard part: finding and fixing remaining thread-unsafe (race) conditions
    - ➢ most easily using atomic operations
- **Set device to 'Cuda' and run on GPU**
  - Hard part: thread scalability, some functors may require redesign

Outline

- Core: Fundamental Concepts
- Core: Views to Arrays
- Core: Views to Arrays – Advanced Features
- Core: Parallel Dispatch of Functors
- Core: Parallel Correctness and Performance
- Core: Device Initialization and Finalization
- Core: Performance Evaluation
- Core: Plans
- **Example: Unordered map global-to-local ids**
- **Example: Finite element integration and nodal summation**
- **Example: Particle interactions in non-uniform neighborhoods**

Example Source Code

In the Trilinos git repository:

- Example: Unordered map global-to-local ids
  - ./packages/kokkos/example/global_2_local_ids/
- Example: Finite element integration and nodal summation
  - ./packages/kokkos/example/feint/
- Example: Particle interactions in non-uniform neighborhoods
  - ./packages/kokkos/example/md_skeleton/
- Configuring ‘cmake’ on testbeds to build examples:
  - ./packages/kokkos/config/configure_compton_cpu.sh
  - ./packages/kokkos/config/configure_compton_mic.sh
  - ./packages/kokkos/config/configure_shannon.sh