Manycore Performance-Portability in Trilinos

H. Carter Edwards / Sandia National Laboratories

Collaborators:
Daniel Sunderland, Vicki Porter, and Michael Heroux / Sandia National Labs
Chris Amsler / Kansas State University
Sam Mish / California State University

August 8-10, 2011

SAND2011-4573C
A Model for Hybrid Parallelism: Layered Separation of Concerns

**Inter-node** (distributed) parallelism
- resource management

**Message Passing**

**Application’s node-local control flow (serial)**

**Trilinos / Kokkos**
- **Intra-node** (manycore) parallelism, data structures, and resource management

**Threads**
- Application’s stateless computational kernels run on each core
- stateless kernels

**Network of computational nodes**

**Computational node with manycore CPUs and/or GPGPU**

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 Thread-Level Parallelism

• Portability with Performance is a Challenge
  – CPU-Multicore and GPGPU-Manycore (e.g., NVIDIA)
  – Language constraints (e.g., CUDA)
  – Performance concerns: memory access patterns dominate

• Data Parallel Computational Kernels
  – Defer task parallelism, pipeline parallelism, ...
  – parallel_for and parallel_reduce semantics

• “Natural” Data Structures
  – Scientific & engineering computations (not just linear algebra)
    • finite element, finite volume, particle, ...
    • matrix-fill computations, matrix-free applications
  – Multidimensional array (a la FORTRAN)
Trilinos’ Kokkos-Array Library

• An API and Library; Not a Compiler
  – C++ template metaprogramming
  – Computational kernels written in subset of C++ (CUDA v3.x)
  – Computing on multidimensional arrays
  – Running on a compute device
    • CPU Multicore, NVIDIA GPGPU, Intel Knights Ferry

• Simple API
  – Very simple C++ class API for multidimensional arrays
  – Very simple “functor” pattern for computational kernels
  – In the spirit of Threaded Building Blocks (TBB) or Thrust
Abstractions

• Manycore Compute Device
  – Provides many threads of execution
  – Owns memory space accessible to and shared by those threads
  – At most one device per process (MPI rank)
    • Choice: for hybrid parallel programming simplicity
    • Two levels: global (MPI) and local (data parallel)

• Multidimensional Array
  – and multivector – a special case not covered in this presentation

• Partitioning and Mapping of Arrays onto a Device

• Data Parallel Computational Kernels
Abstraction: Multidimensional Array

• Homogeneous Collection of Data Members
  – Mathematical, plain-old-data type (for now)
  – Members reside in the memory space of a compute device
  – Members referenced by a multi-index in a multi-index space

• Multi-index (i0, i1, i2, …)
  – Ordered list of indices of a simple integer type
  – Rank – the number of indices

• Multi-index Space
  – Cartesian product of integer ranges
    • Kokkos array: [0..N0) x [0..N1) x [0..N2) x …
    • Abbreviated as: (N0, N1, N2, …)
  – Cardinality = N0 * N1 * N2 * …
Abstraction: Mapping

• Multidimensional Array’s Map
  – Bijective map: multi-index space ↔ array data members
  – \([ 0 .. N0 ) \times [ 0 .. N1 ) \times [ 0 .. N2 ) \times … ↔ array data members

• Two Well-Known Examples
  – Base location + offset into contiguous block of memory
  – FORTRAN: \(( i0 – 1 ) + N0 * (( i1 – 1 ) + N1 * (( i2 – 1 ) + N2 * (…)\))
  – C: \((…(((i0) \times N1 + i1) \times N2 + i2) \times N3 + i3) \times … )

• Key Concept: Choose the Optimal Map for a Device
  – Multiple valid maps; your favorite map is not the only valid map
  – Different devices may have different optimal maps
Abstraction: Parallel Partitioning

• 1D Parallel Partitioning of Data
  – Partition into NP atomic units of parallel work
  – Multidimensional array (and multivector) index space has one parallel work dimension: (NP, N1, N2, …)

• 2D+ Parallel Partitioning of Data – deferred for now
  – Matrices and Grids have multiple parallel work dimensions
  – These are related but different abstractions

• Parallel Work via Computational Kernel
  – Atomic unit of parallel work identified by index: ip ∈ [0..NP)
  – Computational kernel must
    • Update only those array members with index (ip, *, *, …)
    • Not query data being updated by different unit of work
Abstraction: Data Parallel Computational Kernels

- **“Parallel For” Kernel** \( f : (\{\alpha\}, \{X\}) \rightarrow \{Y\} \)
  - \(\{\alpha\}\) are shared parameters
  - \(\{X\}\) and \(\{Y\}\) are sets of partitioned multidimensional arrays
  - \(f\) can be independently applied to each atomic unit of work

- **“Parallel Reduce” Kernel** \( f : (\{\alpha\}, \{X\}) \rightarrow (\{\beta\}, \{Y\}) \)
  - \(\{\beta\}\) are reduction parameters
  - Each atomic unit of work contributes to parameters
    \[ f : (\{\alpha\}, \{X(i_p, \cdots)\}) \rightarrow (\{\beta[i_p]\}, \{Y(i_p, \cdots)\}) \]
  - Contributions are reduced by a *mathematically* commutative and associative function
    \[ f_R : (\{\beta[i_p]\} \forall i_p) \rightarrow \{\beta\} \]
namespace Kokkos {
  template< typename ValueType , class DeviceType >
  class MDArrayView {
public:
    // Query rank and dimensions of multi-index space
    size_type rank() const ;
    size_type dimension( irank ) const ;

    // Access data member on the device via its multi-index
    KOKKOS_MACRO_DEVICE_FUNCTION
    ValueType & operator()( iP , i1 , i2 , ... ) const ;
  };
}

• Index space known on the host and on the device
• Data members reside only on the device
  – Data members only accessible on the device
namespace Kokkos {
    template< typename ValueType ,
              class DeviceDest ,
              class DeviceSource >
    void deep_copy(
        const MDArrayView<ValueType,DeviceDest> & dest ,
        const MDArrayView<ValueType,DeviceSource> & source );
}

• “Deep Copy” – Copy Member Data
  – Between arrays on the same device or different devices
  – Between arrays with the same map or different maps
namespace Kokkos {

template< typename ValueType , class DeviceType >
class MDArrayView {
public:
    MDArrayView(); // NULL view
    // New view of same data viewed by RHS (a “shallow copy”) 
    MDArrayView( const MDArrayView & RHS );
    // Clear this view: if the last view then deallocate member data 
    ~MDArrayView();
    // Clear this view and then assign to be a new view of RHS data 
    MDArrayView & operator = ( const MDArrayView & RHS );
};

    // Allocate a multidimensional array 
    template< typename ValueType , class DeviceType >
    MDArrayView< ValueType , DeviceType >
    create_mdarray( NP , N1 , N2 , ... );
}
API Requirements: Users’ Functors

• Functor: work function + work data
  – Work function is called thread-parallel
    • Called NP times on up to NP different threads
  – Work data reside on the compute device
  – Work data are accessed through Views

• Functors are Passed by Value to the Compute Device
  – Functor members are copied
  – Copying a view is ‘shallow’ – the view is copied not the data

• Functors are Compiled for the Compute Device
  – Work function is restricted: CUDA 3.x – a subset of C++
  – NO memory management on the compute device
  – Thread safety – only access ‘ip’ data members
namespace MyNamespace {

template< class DeviceType >
class MyFunctor {

public:
    typedef DeviceType device_type ; // Required to identify device

    KOKKOS_MACRO_DEVICE_FUNCTION
    void operator()( int ip ) const ; // Required work operator

    // Input and output arrays for the operation:
    typedef MDArrayView< myValueType , device_type > myArrayType ;
    const myArrayType myInputA , myInputB , ... ;
    const myArrayType myOutputX , myOutputY , ... ;

    // Constructor copies views ("shallow copy") of input and output
    MyFunctor( const myArrayType & A , ... )
        : myInputA( A ), ... {}
};
}
namespace MyNamespace {

    template< class DeviceType > class MyFunctor {
        public:

            typedef DeviceType device_type ;
            typedef ... value_type ; // Parameter type, could be a “struct”

            // Operator contributes to the update value
            KOKKOS_MACRO_DEVICE_FUNCTION
            void operator()( int ip , value_type & update ) const ;

            // update = reduce_operation( update , input );
            KOKKOS_MACRO_DEVICE_FUNCTION
            static void join( volatile value_type & update ,
                                volatile const value_type & input );

            // Initialize to the “identity” value for the reduce_operation
            KOKKOS_MACRO_DEVICE_FUNCTION
            static void init( value_type & output );

    };
}
Calling Functors on the Device

• Copy Functor to the device and run it

• Call parallel_for Functor NP times:
  – Work function is called thread-parallel
    ```cpp
    Kokkos::parallel_for( NP , MyFunctor( ... ) );
    ```

• Call parallel_reduce Functor NP times:
  – Return single-value parameter result:
    ```cpp
    value = Kokkos::parallel_reduce( NP , MyFunctor( ... ) );
    ```
  – Output multiple-value parameter ‘struct’ result:
    ```cpp
    Kokkos::parallel_reduce( NP , MyFunctor( ... ) , value );
    ```
  – Store the result on the device (single or multiple value):
    ```cpp
    Kokkos::ValueView<value_type,device_type> result ;
    Kokkos::parallel_reduce( NP , MyFunctor( ... ) , result );
    ```
Performance Test Case #1: Parallel_For on Hexahedral Basis Gradient

• Finite Element Kernel
  – Input coordinates (NP,3,8)
  – Output gradients (NP,3,8)
  – Double precision
  – 6.6 flops per value access
  – Xeon: 2 x 6core x 2 HT
  – Opteron: 2 x 12core
  – NVIDIA C2070 (448 cores)

• vs. Hand-written CUDA
  – No in-code index-map
  – Hard-coded memory offsets
  – Within 20% performance
Performance Test Case #2: Modified Gram-Schmidt Orthogonalization

• Classical Algorithm
  – sequence of parallel_for and parallel_reduce operations
  – Double precision
  – \(2 \times N \times M^2\) flops (\(M=32\))
  – Xeon: 2 x 6core x 2 HT
  – Opteron: 2 x 12core
  – NVIDIA C2070 (448 cores)

• Minimize data exchange
  – Launch sequence of functors on the device
  – Leave and use reduction values on the device
Conclusion & Plans

• Performance-portable multidimensional array programming model
  – Demonstrated on Xeon, Opteron, and NVIDIA
  – “Classical” multidimensional array data access interface
  – Templated on the device ⇒ multi-index map
  – Shared-ownership view semantics

• Plans
  – Other devices; e.g., Intel Knights Ferry (done, but under NDA)
  – Evaluate with more complex kernels & mini-applications
  – Automatic differentiation types
  – Uncertainty quantification polynomial types
  – Expand to multi-parallel-index arrays: grids, matrices