Kokkos: Manycore Programmability and Performance Portability

SIAM Parallel Processing
April 12, 2016

H. Carter Edwards
Christian Trott

SAND2016-2964 C
What is Kokkos?

Kokkos
performance portability for C++ applications

Applications & Libraries

LAMMPS
Albany
Drekar

EMPIRE

SPARC

NABLA (DSL)

Trilinos

Multi-Core

Many-Core

APU

CPU+GPU

Drekar
SPARC

Albany

HBM
DDR

HBM
DDR

HBM
DDR

HBM
DDR

HBM
DDR

HBM
DDR
What is *Kokkos*?

- **ΚΟΚΚΟΣ** (Greek, not an acronym)
  - Translation: “granule” or “grain”; *like grains of sand on a beach*

- **Performance Portable Thread-Parallel Programming Model**
  - E.g., “X” in “MPI+X”; **not** a distributed-memory programming model
  - Application identifies its parallelizable grains of **computations and data**
  - Kokkos maps those computations onto cores *and* that data onto memory

- **Fully Performance Portable C++11 Library Implementation**
  - *Not* a language extension (e.g., OpenMP, OpenACC, OpenCL, ...)
  - **Production** – open source at https://github.com/kokkos/kokkos
  - ✔ *Multicore CPU* - including NUMA architectural concerns
  - ✔ *Intel Xeon Phi (KNC)* – toward DOE’s Trinity (ATS-1) supercomputer
  - ✔ *NVIDIA GPU (Kepler)* – toward DOE’s Sierra (ATS-2) supercomputer
  - ♠ *IBM Power 8* – toward DOE’s Sierra (ATS-2) supercomputer
  - ♠ *AMD Fusion* – back-end in collaboration with AMD via HCC
    - https://bitbucket.org/multicoreware/hcc/wiki/Home
  - ✔ Regularly tested
  - ♠ Ramping up testing
Abstractions: Patterns, Policies, and Spaces

- **Parallel Pattern** of user’s computations
  - `parallel_for`, `parallel_reduce`, `parallel_scan`, task-graph, ...
    *(extensible)*

- **Execution Policy** tells *how* user computation will be executed
  - Static scheduling, dynamic scheduling, thread-teams, ...
    *(extensible)*

- **Execution Space** tells *where* user computations will execute
  - Which cores, numa region, GPU, ...
    *(extensible)*

- **Memory Space** tells *where* user data resides
  - Host memory, GPU memory, high bandwidth memory, ...
    *(extensible)*

- **Layout** (policy) tells *how* user data is laid out in memory
  - Row-major, column-major, array-of-struct, struct-of-array ...
    *(extensible)*

- **Differentiating: Layout and Memory Space**
  - Versus other programming models (OpenMP, OpenACC, ...)
  - Critical for performance portability ...
Layout Abstraction: Multidimensional Array

- Classical (50 years!) data pattern for science & engineering codes
  - Computer languages hard-wire multidimensional array layout mapping
  - Problem: different architectures require different layouts for performance
    - Leads to architecture-specific versions of code to obtain performance
  - E.g., “Array of Structure” ↔ “Structure of Array” redesigns

- Kokkos separates layout from user’s computational code
  - Choose layout for architecture-specific memory access pattern
    - Without modifying user’s computational code
  - Polymorphic layout via C++ template meta-programming (extensible)
    - e.g., Hierarchical Tiling layout (array of structure of array)

- Bonus: easy/transparent use of special data access hardware
  - Atomic operations, GPU texture cache, ... (extensible)
Performance Impact of Data Layout

- Molecular dynamics computational kernel in miniMD
- Simple Lennard Jones force model:
  \[ F_i = \sum_{j, r_{ij} < r_{cut}} 6\varepsilon \left[ \left( \frac{\varsigma}{r_{ij}} \right)^7 - 2 \left( \frac{\varsigma}{r_{ij}} \right)^{13} \right] \]
- 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,0..2) - pos(j,0..2); // random read 3 floats
    if (|r_ij| < r_cut) f_i += 6*\epsilon*((s/r_ij)^7 – 2*(s/r_ij)^13)
}
f(i) = f_i;
```

- Test Problem
  - 864k atoms, ~77 neighbors
  - 2D neighbor array
  - Different layouts CPU vs GPU
  - Random read ‘pos’ through GPU texture cache

- Large performance loss with wrong data layout

<table>
<thead>
<tr>
<th></th>
<th>Xeon</th>
<th>Xeon Phi</th>
<th>K20x</th>
</tr>
</thead>
<tbody>
<tr>
<td>correct (with texture)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>correct (without texture)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wrong layout (with texture)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Performance Overhead?

Kokkos is competitive with other programming models

- Regularly performance-test mini-applications on Sandia’s ASC/CSSE test beds
- MiniFE: finite element linear system iterative solver mini-app
  - Compare to versions with architecture-specialized programming models

---

**Diagram:**

MiniFE CG-Solve time for 200 iterations on 200^3 mesh

- K20X
- IvyBridge
- SandyBridge
- XeonPhi B0
- XeonPhi C0
- IBM Power7+

- NVIDIA ELL
- NVIDIA CuSparse
- Kokkos
- OpenMP
- MPI-Only
- OpenCL
- TBB
- Cilk+(1 Socket)
Performance Portability & Future Proofing

Integrated mapping of users’ parallel computations and data through abstractions of patterns, policies, spaces, and layout.

- Versus other thread parallel programming models (mechanisms)
  - OpenMP, OpenACC, OpenCL, ... have parallel execution
  - OpenMP 4 finally has execution spaces; when memory spaces ??
    - All of these neglect data layout mapping
      - Requiring significant code refactoring to change data access patterns
      - Cannot provide performance portability
    - All require language and compiler changes for extension

- Kokkos extensibility “future proofing” wrt evolving architectures
  - Library extensions, not compiler extensions
  - E.g., Intel KNL high bandwidth memory ← just another memory space

- Productivity versus other programming models?
Patterns, Policies, and C++11 Lambdas

- Pattern composed with policy drives the computational body

  ```
  for ( int i = 0 ; i < N ; ++i ) { /* body */ }
  ```

  pattern  policy  body

  ```
  parallel_for ( N, [=]( int i ) { /* body */ } );
  ```

- C++11 lambda implements computational body
  - C++ compiler creates a closure for you: function body + captured data
    - Old school: tedious of writing a C++ class with `operator()( int i )`
  - Kokkos executes your closure according to pattern and policy

- C++17 lambda within a class member function: ` [=,*this]`
  - Fixed defect in C++11: no way to capture `*this` by value

- Data parallel patterns: for, reduce, scan
- Execution policies: range and hierarchical thread team
- Illustrate with the following examples...
Example: Sparse Matrix-Vector Multiply (SPMV)

- **Baseline serial version**
  ```c
  for ( int i = 0 ; i < nrow ; ++i ) {
      for ( int j = irow[i] ; j < irow[i+1] ; ++j )
          y[i] += A[j] * x[jcol[j]] ;
  }
  ```

- **Simple Kokkos parallel version**
  ```c
  parallel_for( nrow , KOKKOS_LAMBDA( int i ) {
      for ( int j = irow[i] ; j < irow[i+1] ; ++j )
          y[i] += A[j] * x[jcol[j]] ;
  });
  ```

- **“nrow” implies a Range execution policy**
  - Call body with i = [0..nrow), call in parallel with no ordering guarantees
  - Call body in the default execution space

- **KOKKOS_LAMBDA for GPU/CUDA portability**
  - **CPU**: #define KOKKOS_LAMBDA [=] /* nothing */
  - **GPU**: #define KOKKOS_LAMBDA [=] __host__ __device__
  - GPU requires CUDA 7.5 and lambda capture-by-value [=]
Example: Dot-product and Prefix-Sum

- **Baseline serial versions**, is the pattern obvious?

```c
double result = 0;
for ( int i = 0 ; i < N ; ++i ) { result += x[i] * y[i]; }

y[i] = 0;
for ( int i = 0 ; i < N ; ++i ) { y[i+i] = y[i] + x[i]; }
```

- **Simple Kokkos parallel versions**

```c
parallel_reduce( N, KOKKOS_LAMBDA( int i, double & tmp ){
    tmp += x[i] * y[i];
}, result );

y[i] = 0;
parallel_scan( N, KOKKOS_LAMBDA( int i, int & tmp, bool final ){
    tmp += x[i];
    if ( final ) y[i+1] = tmp;
});
```

- **Kokkos manages for you:**
  - Thread local temporary variables
  - Inter-thread synchronizations and reductions of thread local temporaries
Example: Sparse Matrix-Vector Multiply (SPMV)

- Explicit Range execution policy version

```cpp
parallel_for( RangePolicy<Space>(0,nrow), KOKKOS_LAMBDA(int i){
    for ( int j = irow[i] ; j < irow[i+1] ; ++j )
        y[i] += A[j] * x[jcol[j] ];
});
```

- Is \([0 .. nrow]\) enough parallelism?
  - With \(O(1000)\)s GPU threads? That nested loop could also be parallel ...

- Hierarchical Thread Team execution policy
  - `TeamPolicy<Space>(LeagueSize,TeamSize)`
  - OpenMP : league of teams of threads
  - CUDA : grid of blocks of threads
  - Threads within a team are concurrent
  - Teams within a league are not concurrent
Example: Sparse Matrix-Vector Multiply (SPMV)

```cpp
parallel_for(
    TeamPolicy<Space>(nrow,AUTO),
    KOKKOS_LAMBDA( TeamPolicy<Space>::member_type member ) {
    const int i = member.league_rank();
    double result = 0;
    parallel_reduce(
        TeamThreadRange(member,irow[i],irow[i+1]),
        [&]( int j , double & tmp ) { tmp += A[j] * x[jcol[j]]; },
        result);
    if ( member.team_rank() == 0 ) y[i] = result;
});
```

- **Outer level of parallel pattern + execution policy**
  - TeamPolicy requires closure (lambda) with ‘member_type’ argument
    - `member` is a handle for thread within team within a league
  - Requires KOKKOS_LAMBDA macro (CPU ➔ GPU)

- **Inner level of parallel pattern + execution policy**
  - TeamThreadRange identifies `member` threads that participate
  - Ordinary (unmarked) C++11 lambda may be used
Data Placement and Layout: Views

- **View< double**[3][8] , Space_opt > a(“a”,N,M);
  - Allocate array data in a memory Space with dimensions [N][M][3][8]
  - View semantics analogous to C++11 std::shared_ptr

- a(i,j,k,l) : User’s access to array datum
  - Multi-index mapping according to layout
  - “Space” accessibility enforced; e.g., GPU code cannot access CPU memory
  - Optional array bounds checking of indices for debugging

- **View< ArrayType , Layout_opt , Space_opt , Attributes_opt >**
  - Explicitly declare array layout instead of letting Kokkos choose
  - Access intent attributes; e.g., atomic, random access (GPU texture cache)

- Array subview of array view
  - b = subview( a , {10,100} , {200,300} , 2 , 3 ); // ranges and indices
  - View of same data, with the appropriate layout and multi-index map

- View-like functionality on-track for C++20
Thread Safety and Atomic Operations

- Some algorithms have inherent thread safety challenges
  - Histogram summing into buckets
  - Finite element assembly of linear system coefficients
  - Scatter-add pattern: \[ A[\text{index}[i]] += f(x[i], y[i], ...) \];

- Strategies for thread safety
  - *Coloring* (partitioning) of work into disjoint subsets avoids conflicts
    - Serial execution across subsets, parallel execution within a subset
    - Performance concerns: reduced parallelism and coloring algorithm overhead
  - *Atomic* operations serialize conflicts
    - Special hardware for “+” of numeric types, perhaps reduced performance
    - Simpler to use than coloring, no loss of parallelism

- Atomics, C++11, and Kokkos
  - C++11 has “hard wired” atomic types with atomic operations
  - Kokkos provides atomic operations on ordinary types
  - C++20 atomic operations for non-atomic types is “in the works”
Other Features (new or in-development)

- Back-ends for new & changing node architectures
  - AMD Fusion with new open source HCC compiler
  - Intel KNL heterogeneous memory (high bandwidth memory)
  - NVIDIA GPU register shuffle for intra-thread team collectives

- Patterns, policies, spaces, layout
  - Dynamic scheduling (work stealing) execution policies
  - Multidimensional range policies (parallel “loop collapse”)
  - Dynamically resizable arrays - thread-scalable within parallel operations
  - Directed acyclic graph (DAG) of “fine grain” tasks execution pattern/policy
  - Tiling layout mapping

- Portable embedded performance instrumentation
  - Selective instrumentation of individual parallel dispatch
    - parallel_for, parallel_reduce, parallel_scan
Conclusion / Takeaways

- **Performance Portability, for C++ Applications**
  - Integrated mapping of applications’ computations *and* data
    - Other programming models fail to map data and limit *performance* portability
  - Future proofing via designed-in extensibility and ongoing R&D
  - Production on Multicore CPU, Intel Xeon Phi, IBM Power 8, and NVIDIA GPU; *AMD Fusion in progress*
    - github.com/kokkos/kokkos

- **Productivity, for C++ Applications**
  - C++11 lambda for simple conversion of ‘for’ loops to ‘parallel_pattern’
  - Reduce and Scan inter-thread complexity managed by Kokkos
  - Hierarchical parallelism using nested patterns can increase parallelism

- **Goal:** ISO/C++ 2020 Standard subsumes Kokkos abstractions

---

**NOTE:** SIAM-PP16, MS81, Friday 4:50pm
Performance and Productivity of Abstract C++ Programming Model