Kokkos, Manycore Device Performance Portability for C++ HPC Applications

H. Carter Edwards and Christian Trott
Sandia National Laboratories

GPU TECHNOLOGY CONFERENCE 2015

MARCH 16-20, 2015 | SAN JOSE, CALIFORNIA

SAND2015-1885C (Unlimited Release)

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

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

What is “Kokkos” ?

- **κόκκος (Greek)**
  - Translation: “granule” or “grain” or “speck”
  - Like grains of salt or sand on a beach
- **Programming Model Abstractions**
  - Identify / encapsulate grains of data and parallelizable operations
  - Aggregate these grains with data structure and parallel patterns
  - Map aggregated grains onto memory and cores / threads
- **An Implementation of the Kokkos Programming Model**
  - Sandia National Laboratories’ open source C++ library

Outline

- Core Abstractions and Capabilities
  - Performance portability challenge: memory access patterns
  - Layered C++ libraries
  - Spaces, policies, and patterns
  - Polymorphic multidimensional array
  - Easy parallel patterns with C++11 lambda
  - Managing memory access patterns
  - Atomic operations
  - Wrap up
- Portable Hierarchical Parallelism
- Initial Scalable Graph Algorithms
- Conclusion

Performance Portability Challenge:

Best (decent) performance requires computations to implement architecture-specific memory access patterns

- **CPUs (and Xeon Phi)**
  - Core-data affinity: consistent NUMA access (first touch)
  - Array alignment for cache-lines and vector units
  - Hyperthreads' cooperative use of L1 cache
- **GPUs**
  - Thread-data affinity: coalesced access with cache-line alignment
  - Temporal locality and special hardware (texture cache)
- **Array of Structures (AoS) vs. Structure of Arrays (SoA) dilemma**
  - i.e., architecture specific data structure layout and access

➢ **This has been the *wrong* concern**

The right concern: Abstractions for Performance Portability?

Kokkos' Performance Portability Answer

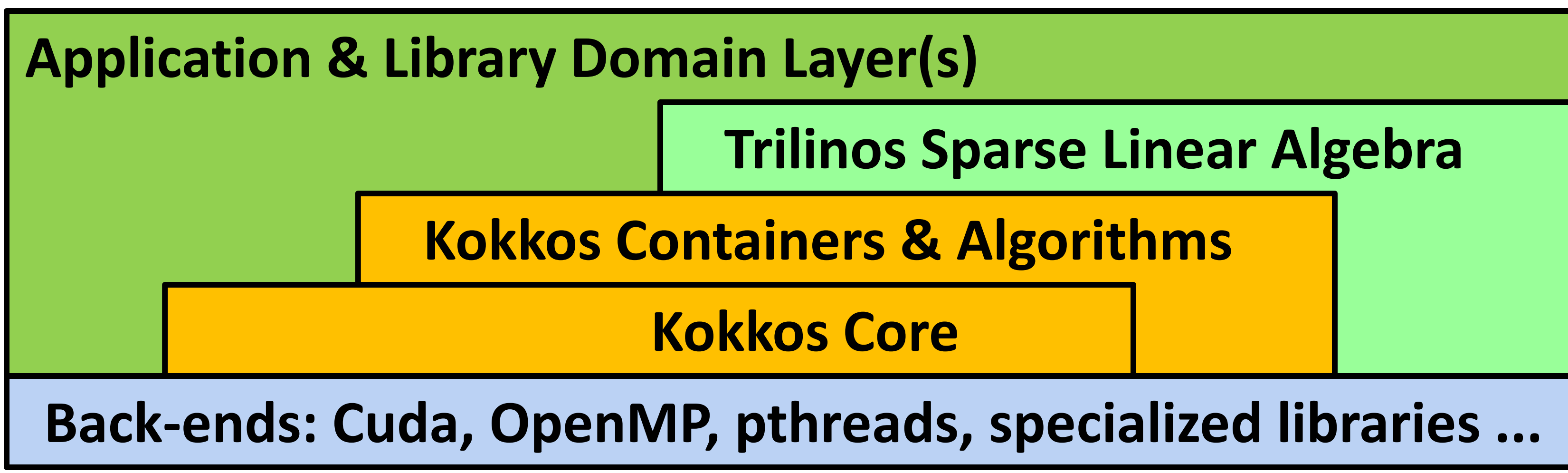
Sandia National Laboratories

Integrated mapping of thread parallel computations and multidimensional array data onto manycore architecture

1. **Map user's parallel computations to threads**
   - Parallel pattern: foreach, reduce, scan, task-dag, ...
   - Parallel loop/task body: C++11 lambda or C++98 functor
2. **Map user's datum to memory**
   - Multidimensional array of datum, ***with a twist***
   - Layout : multi-index (i,j,k,...) ↔ memory location
   - Kokkos *chooses* layout for architecture-specific memory access pattern
   - Polymorphic multidimensional array
3. **Access user datum through special hardware (bonus)**
   - GPU texture cache to speed up read-only random access patterns
   - Atomic operations for thread safety

Layered Collection of C++ Libraries

- **Standard C++, Not a language extension**
  - *Not* a language extension: OpenMP, OpenACC, OpenCL, CUDA
  - In *spirit* of Intel's TBB, NVIDIA's Thrust & CUSP, MS C++AMP, ...
- **Uses C++ template meta-programming**
  - Previously relied upon C++1998 standard
  - Now require C++2011 for lambda functionality
    - Supported by Cuda 7.0; full functionality in Cuda 7.5
  - Participating in ISO/C++ standard committee for core capabilities



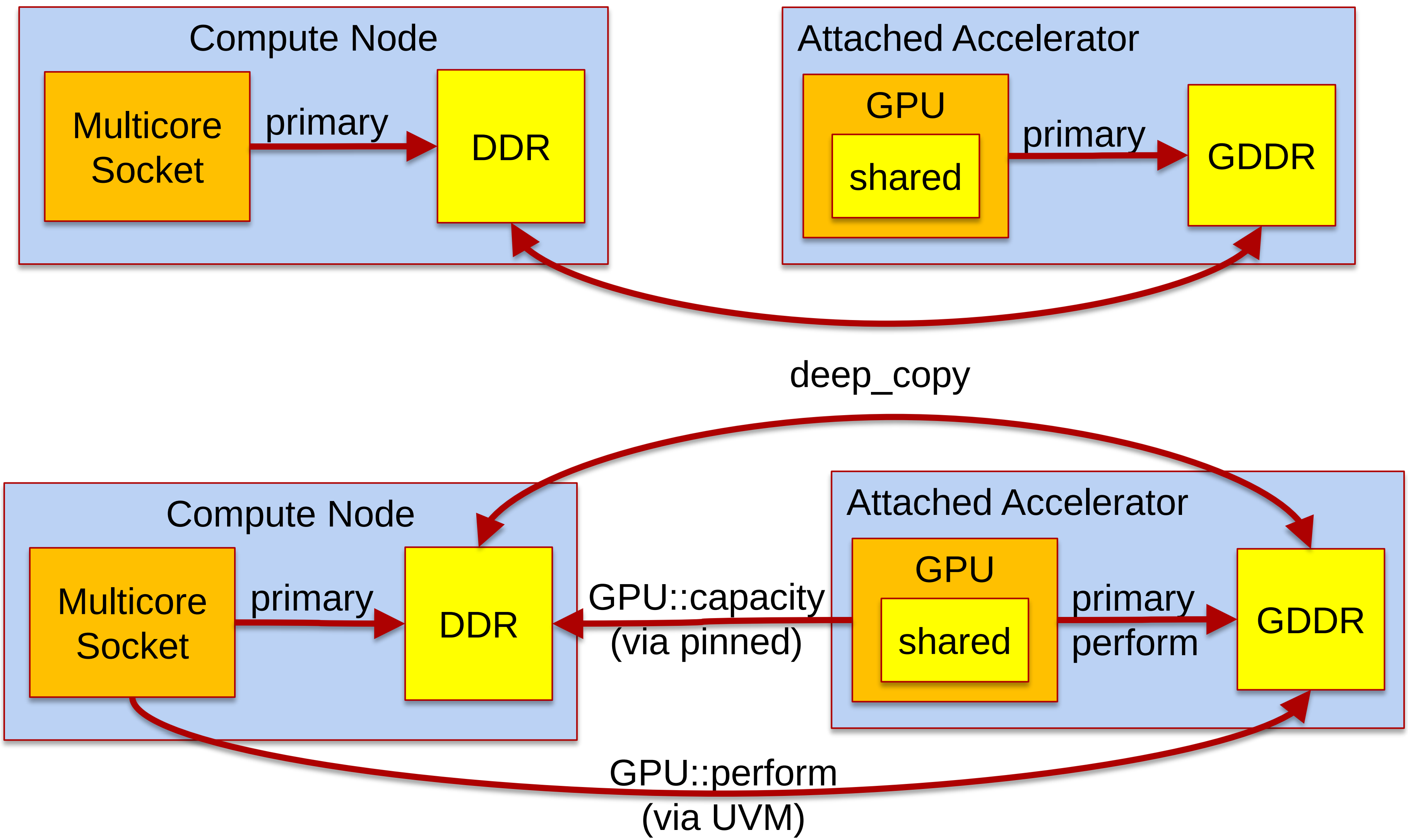
Abstractions: **Spaces, Policies, and Patterns**

- **Memory Space : <u>where</u> data resides**
  - **Differentiated by performance; e.g., size, latency, bandwidth**
- **Execution Space : <u>where</u> functions execute**
  - **Encapsulates hardware resources; e.g., cores, GPU, vector units, ...**
  - **Denote accessible memory spaces**
- **Execution Policy : <u>how</u> (and where) a user function is executed**
  - **E.g., data parallel range : concurrently call function(i) for i = [0..N)**
  - **User's function is a C++ functor or C++11 lambda**
- **Pattern: parallel_for, parallel_reduce, parallel_scan, task-dag, ...**
- **Compose: pattern + execution policy + user function; e.g.,**

```
parallel_pattern( Policy<Space>, Function);
```

  - **Execute *Function* in *Space* according to *pattern* and *Policy***
- **Extensible spaces, policies, and patterns**

Examples of Execution and Memory Spaces

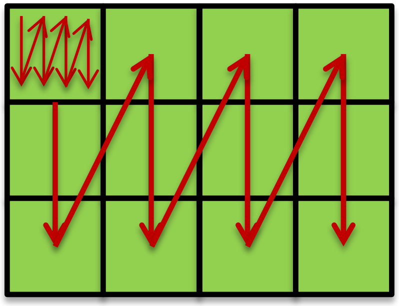


Polymorphic Multidimensional Array <u>View</u>

- View< double**[3][8] , Space > a(“a”,N,M);
  - Allocate array data in memory Space with dimensions [N][M][3][8]
  - ? C++17 improvement to allow View<double[ ][ ][3][8],Space>
- a(i,j,k,l) : User’s access to array datum
  - “Space” accessibility enforced; e.g., GPU code cannot access CPU memory
  - Optional array bounds checking of indices for debugging
- View Semantics: View<double**[3][8],Space> b = a ;
  - A <u>shallow</u> copy: ‘a’ and ‘b’ are *pointers* to the same allocated array data
  - Analogous to C++11 std::shared_ptr
- deep_copy(destination_view , source_view);
  - Copy data from ‘source_view’ to ‘destination_view’
  - ➢ Kokkos policy: never hide an expensive deep copy operation

Polymorphic Multidimensional Array Layout

- Layout mapping : a(i,j,k,l) → memory location
  - Layout is polymorphic, defined at compile time
  - Kokkos chooses default array layout appropriate for “Space”
  - E.g., row-major, column-major, Morton ordering, dimension padding, ...
- User can specify Layout : View< ArrayType, *Layout*, Space >
  - Override Kokkos’ default choice for layout
  - Why? For compatibility with legacy code, algorithmic performance tuning, ...
- Example Tiling Layout
  - View<double**,Tile<8,8>,Space> m(“matrix”,N,N);
  - Tiling layout transparent to user code : m(i,j) unchanged
  - Layout-aware algorithm extracts tile subview



Multidimensional Array Subview & Attributes

- **Array subview of array view (new)**
  - **Y = subview(X , *...ranges_and_indices_argument_list...*);**
  - **View of same data, with the appropriate layout and index map**
  - **Each index argument eliminates a dimension**
  - **Each range [begin,end) argument contracts a dimension**
- **Access intent Attributes**

  View< ArrayType, Layout, Space, *Attributes* >

  - **How user intends to access datum**
  - **Example, View with const and random access intension**
    - **View< double ** , Cuda > a("mymatrix", N, N);**
    - **View< const double **, Cuda, RandomAccess > b = a ;**
    - ➢ **Kokkos implements b(i,j) with GPU texture cache**

Multidimensional Array functionality being considered by ISO/C++ standard committee

- **TBD: add layout polymorphism – a critical capability**
  - To be discussed at May 2015 ISO/C++ meeting
- **TBD: add explicit (compile-time) dimensions**
  - Minor change to core language to allow: T[ ][ ][3][8]
  - Concern: performance loss when restricted to implicit (runtime) dimensions
- **TBD: use simple / intuitive array access API: x(i,j,k,l)**
  - Currently considering : x[{ i , j , k , l }]
  - Concern: performance loss due to intermediate initializer list
- **TBD: add shared pointer (std::shared_ptr) semantics**
  - Currently merely a wrapper on user-managed memory
  - Concern: coordinating management of view and memory lifetime

Easy Parallel Patterns with C++11 and Defaults

parallel_pattern(Policy<Space> , UserFunction)

- **Easy example BLAS-1 AXPY with views**

```
parallel_for( N , KOKKOS_LAMBDA( int i ){ y(i) = a * x(i) + y(i); } );
```

  - Default execution space chosen for Kokkos installation
  - Execution policy "N" => RangePolicy<DefaultSpace>(0,N)
  - #define KOKKOS_LAMBDA [=]   /* non-Cuda */
  - #define KOKKOS_LAMBDA [=]__device__ /* Cuda 7.5 candidate feature */
    - Tell NVIDIA Cuda development team you like and want this in Cuda 7.5 !

- **More verbose without lambda and defaults:**

```
struct axpy_functor {
  View<double*,Space> x , y ; double a ;
  KOKKOS_INLINE_FUNCTION
  void operator()( int i ) const { y(i) = a * x(i) + y(i); }
  // ... constructor ...
};
parallel_for( RangePolicy<Space>(0,N) , axpy_functor(a,x,y) );
```

Kokkos Manages Challenging Part of Patterns' Implementation

- **Example: DOT product reduction**

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

  - Challenges: temporary memory and inter-thread reduction operations
    - Cuda shared memory for inter-warp reductions
    - Cuda global memory for inter-block reductions
    - Intra-warp, inter-warp, and inter-block reduction operations

- **User may define reduction type and operations**

```
struct my_reduction_functor {
  typedef ... value_type ;
  KOKKOS_INLINE_FUNCTION void join( value_type&, const value_type&) const ;
  KOKKOS_INLINE_FUNCTION void init( value_type& ) const ;
};
```

  - 'value_type' can be runtime-sized one-dimensional array
  - 'join' and 'init' plugged into inter-thread reduction algorithm

Managing Memory Access Pattern:

Compose Parallel Execution ○ Array Layout

- Map Parallel Execution
  - Maps calls to function(iw) onto threads
  - GPU: iw = threadIdx + blockDim * blockIds
  - CPU: iw $\in$[begin,end)$_{Th}$; contiguous partitions among threads
- Choose Multidimensional Array Layout
  - Leading dimension is parallel work dimension
    - Leading multi-index is ‘iw’ : a(iw , j, k, l)
  - Choose appropriate array layout for space’s architecture
    - E.g., AoS for CPU and SoA for GPU
- Fine-tune Array Layout
  - E.g., padding dimensions for cache line alignment

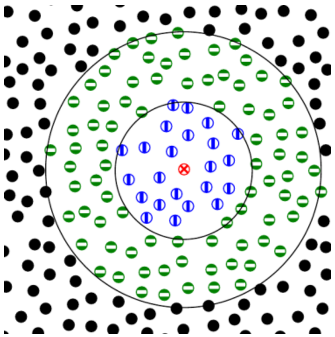
Performance Impact of Access Pattern

- **Molecular dynamics computational kernel in miniMD**
- **Simple Lennard Jones force model:**

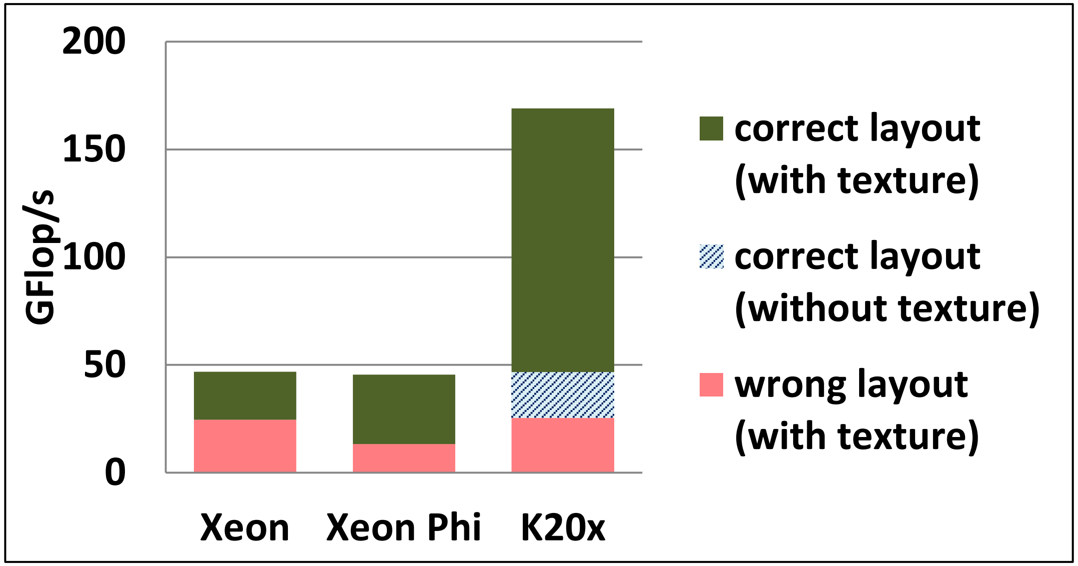
$$F_i = \sum_{j,\, r_{ij} < r_{cut}} 6\varepsilon \left[\left(\frac{\varsigma}{r_{ij}}\right)^7 - 2\left(\frac{\varsigma}{r_{ij}}\right)^{13} \right]$$

- **Atom neighbor list to avoid N^2 computations**

```
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;
```



- **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 array layout**



Atomic operations

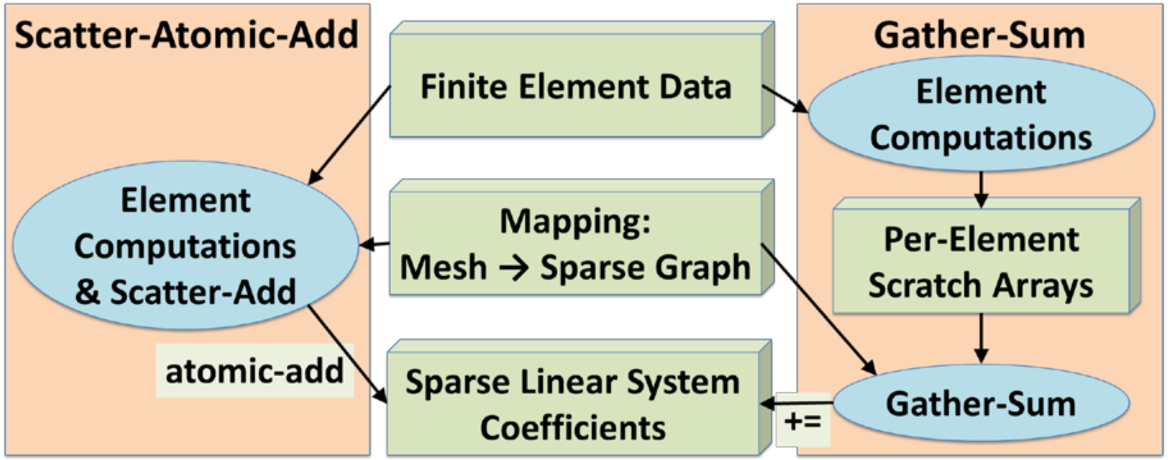
atomic_exchange, atomic_compare_exchange_strong, atomic_fetch_add, atomic_fetch_or, atomic_fetch_and

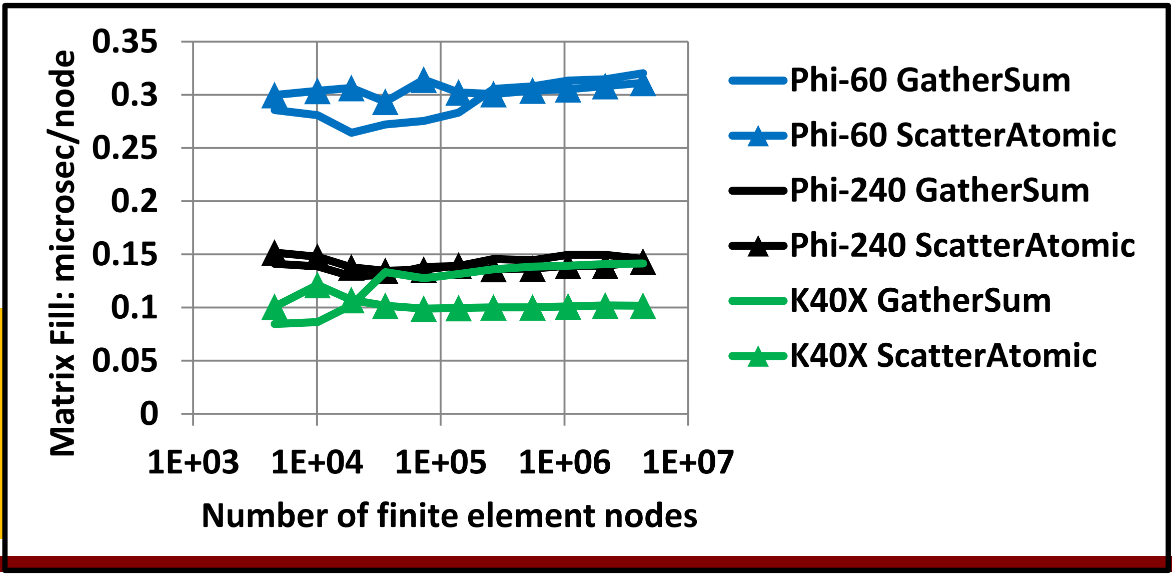
- **Thread-scalability of non-trivial algorithms and data structures**
  - Essential for lock-free implementations
  - Concurrent summations to shared variables
    - E.g., finite element computations summing to shared nodes
  - Updating shared dynamic data structure
    - E.g., append to a shared array or insert into a shared map
- **Portably map to compiler/hardware specific capabilities**
  - GNU and CUDA extensions when available
  - Current: any 32bit or 64bit type, may use CAS-loop implementation
- **ISO/C++ 2011 and 2014 atomics not adequate for HPC**
  - Proposed necessary improvements for C++17

Thread-Scalable Fill of Sparse Linear System

- MiniFENL: Newton iteration of FEM: $x_{n+1} = x_n - J^{-1}(x_n)r(x_n)$
- Fill sparse matrix via Scatter-Atomic-Add or Gather-Sum ?
- Scatter-Atomic-Add
  - + Simpler
  - + Less memory
  - – Slower HW atomic
- Gather-Sum
  - + Bit-wise reproducibility
- Performance win?
  - Scatter-atomic-add
  - ~equal Xeon PHI
  - 40% faster Kepler GPU

✓ Pattern chosen
- Feedback to HW vendors: performant atomics





Core Abstractions and Capabilities (wrap up)

- **Abstractions**
  - **Identify / encapsulate grains of data and parallelizable operations**
  - **Aggregate these grains with data structure and parallel patterns**
  - **Map aggregated grains onto memory and cores / threads**
- **Grains and Patterns**
  - **Parallelizable operation: C++11 lambda or C++98 functor**
  - **Parallel pattern: foreach, reduce, scan, task-dag, ...**
  - **Multidimensional array of datum**
  - **Atomic operations**
- **Extensible Mappings**
  - **Polymorphic multidimensional array : space, layout, access intentions**
  - **Execution policy : where and how to execute**

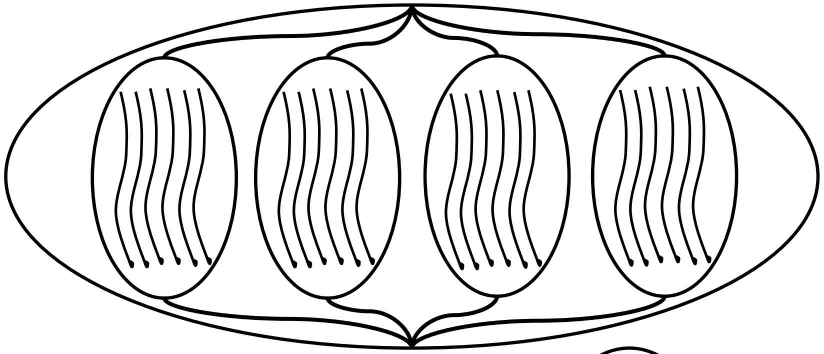
➢ **Next Step : Finer Grain Parallelism with Hierarchical Patterns**
  - **κόκκος : "like grains of sand on a beach" – how fine can we go?**

Outline

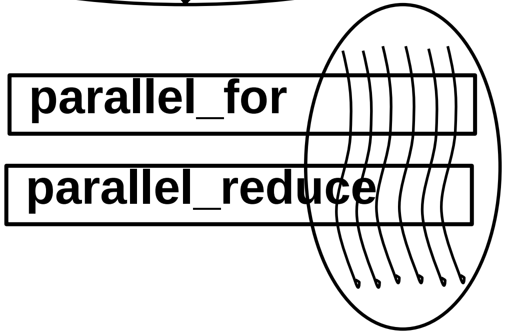
- Core Abstractions and Capabilities
- Portable Hierarchical Parallelism
  - Two-level thread-team execution policy and nested parallel patterns
  - Thread-team shared memory
  - Three-level execution policy
  - Application to molecular dynamics kernels
  - Application to tensor mathematics kernels
- Initial Scalable Graph Algorithms (very new)
- Conclusion

Thread Team Execution Policy

- Expose and map more parallelism
- Vocabulary
  - OpenMP: League of Teams of Threads
  - Cuda: Grid of Blocks of Threads



- Thread Team Functionality
  - Threads <u>within</u> a team execute concurrently
  - Teams do not execute concurrently
  - Nested parallel patterns: foreach, reduce, scan
  - Team-shared scratch memory



- Thread Team Portability : map onto hardware
  - Cuda : team == thread block, possibly a sub-block group of warps
  - Xeon Phi : team == hyperthreads sharing L1 cache
  - CPU : team == thread

Thread Team Example:
Sparse Matrix-Vector Multiplication

- **Traditional serial compressed row storage (CRS) algorithm:**

```
for ( int i = 0 ; i < nrow ; ++i )
  for ( int j = irow(i) ; j < irow(i+1) ; ++j )
    y(i) += A(j) * x( jcol(j) );
```

- **Thread team algorithm, using C++11 lambda**

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

Thread Team Shared Scratch Memory

- **Challenges**
  - **Multiple arrays residing in shared scratch memory**
  - **Arrays may have runtime dimensions**
  - **Arrays' dimensions possibly dependent upon team size**
- **Approach: reuse Kokkos abstractions**
  - **Shared scratch <u>Memory Space</u> of the <u>Execution Space</u>**
  - **Manage array with a <u>View</u> defined on this space**
  - **Thread team executing in the execution space is given an instance of the associated shared scratch memory space**
- **Capability available via user defined functor**
  - **Typically need richer information than C++11 lambda can provide**
  - **... example ...**

Team Shared Scratch Memory Example

```
struct my_functor {
  typedef TeamPolicy<ExecutionSpace>                   Policy ;
  typedef ExecutionSpace::scratch_memory_space         Scratch ;
  typedef View<double**,Scratch,MemoryUnmanaged>       SharedView ;
  SharedView x , y ;
  int nx , ny ;

  KOKKOS_INLINE_FUNCTION
  void operator()( Policy::member_type const & member ) const
  {
    Scratch shmem_space = member.team_shmem();
    x( shmem_space, member.team_size(), nx );
    y( shmem_space, member.team_size(), ny );
    // ... team fill of arrays ...
    member.team_barrier();
    // ... team computations on arrays ...
    member.team_barrier();
  }
  // Query shared memory size before parallel dispatch:
  size_t team_shmem_size( int team_size ) const {
    return SharedView::shmem_size( team_size , nx ) +
           SharedView::shmem_size( team_size , ny );
  }
};
```

Thread Team Execution Policy, 3rd Level

- **Add third level of Vector parallelism**
  - **Map algorithm's thread teams onto hardware resources**
  - **Cuda : "thread" == warp, "vector lane" == thread of warp**
  - **Xeon Phi : "thread" == hyperthread, "vector lane" == SSE or AVX lane**
- **Vector parallelism functionality**
  - **Vector lanes execute lock-step concurrently**
  - **Consistent parallel patterns at vector level: foreach, reduce, scan**
  - **New "single" pattern denoting only one vector lane performs operation**
- **Portably covering all levels used in sophisticated Cuda kernels**
- **C++11 lambda necessary for usability**
  - **Vector parallel lambdas nested within team parallel lambdas**
  - **Fortunately Cuda 6.5 supports C++ lambda within device kernels!**

Application to Molecular Dynamics Kernel

Atom Neighbor List Construction

- atom ids stored in a Cartesian grid (XYZ) locality-bin data structure
- atoms sorted by locality -> Non-Team algorithm has good cache efficiency
- using teams and shared memory to improve cache efficiency on GPU
- a team works on a set of neighboring bins, 1 bin per thread in the team

Non-Team Algorithm

```
parFor i in natoms {
  n = 0
  bin_idx = bin_of(i);
  for bin in stencil(bin_idx) {
    for j in bin_atom_ids(bin) {
      if( distance(i,j) < cut )
        neighbor(i,n++) = j;
    }
  }
}
```

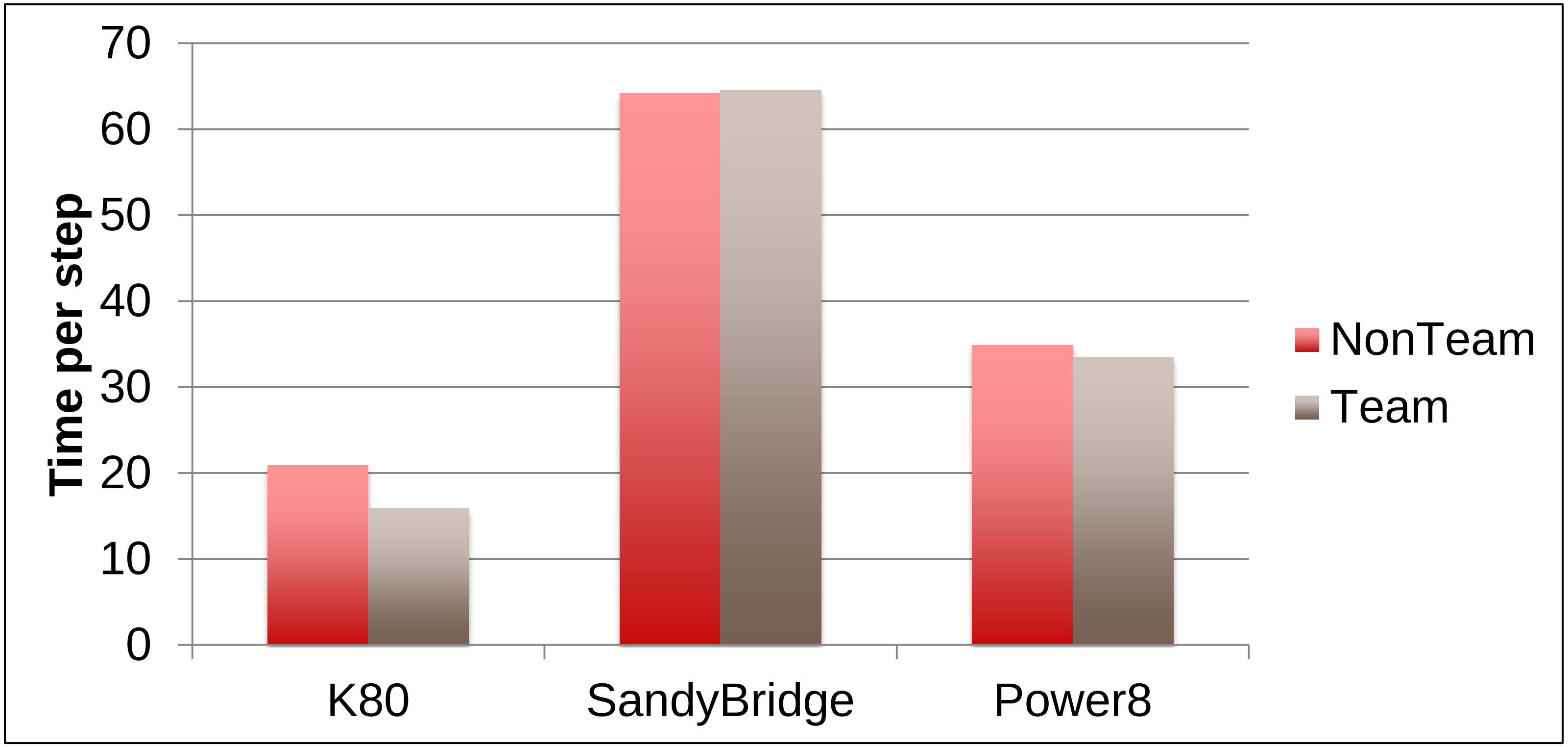
Team Algorithm

```
parForTeam base_bins in bins {
  copy_to_shared(base_bins,shared_base_bins)
  for bin_row in YZ_part_of(base_bins) {
    copy_to_shared(bin_row,shared_bin_row)
    parForTeam i in bin_atom_ids(shared_base_bins) {
      parForVector i in bin_atom_ids(shared_base_bins) {
        for j in bin_atom_ids(shared_bin_row) {
          if( distance(i,j) < cut ) neighbor(i,n++) = j;
        }
      }
    }
  }
}
```

- **Previously a Cuda-specialized implementation**
- **Now a portable implementation**

Performance of a Complete Simulation Step

- Timing data for isolated kernel not available
- Comparing compute nodes of roughly equivalent power
  - 1/2 of K80 (i.e. one of the two GPUs on the board)
  - 2 Sockets of 8 Core Sandy Bridge with 2 wide SMT
  - 2 Sockets of 10 Core Power 8 chips with 8 wide SMT
- CPUs using Team-Size 1
- GPUs using Team-Size 2x32

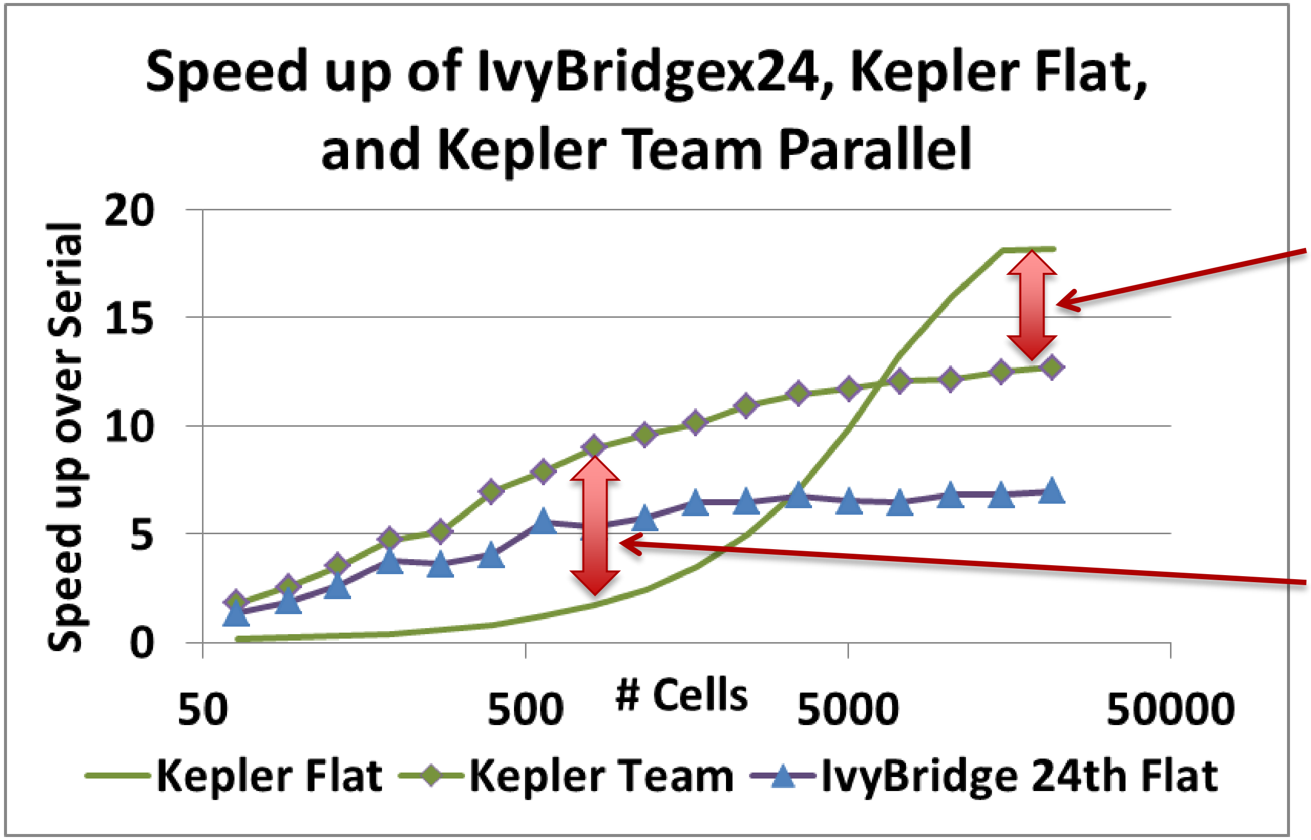


Application to Tensor Math Library Kernels

- **Performed through Harvey Mudd College clinic program**
  - **Advisor/Professor: Jeff Amelang**
  - **Undergraduate team: Brett Collins, Alex Gruver, Ellen Hui, Tyler Marklyn**
- **Project: re-engineer serial kernels to use Kokkos**
  - **Initially using "flat" range policy**
  - **Progressing to thread team policy for appropriate kernels**
  - **Several candidate kernels for team parallelism, results for:**
  - **Multi-matrix multiply :** $\forall(c,d,e)\colon V(c,d,e) = \sum_p A(c,p,d) * B\,(c,p,e)$
- **Thread team**
  - **Outer (league level) parallel_for over dimension 'c'**
  - **Inner (team level) parallel_reduce over summation dimensions p**
  - **Inner (team level) parallel_for over tensor dimensions d, e**

Application to Tensor Math Library Kernels

- **Performance of "multi-matrix multiply" tensor contraction**
  - $\forall(c,d,e): V(c,d,e) = \sum_p A(c,p,d) * B(c,p,e)$
  - **d = e = 6, symmetric tensor**
  - **p = 27 point numerical integration of a hexahedral cell**
  - **c = # cells**



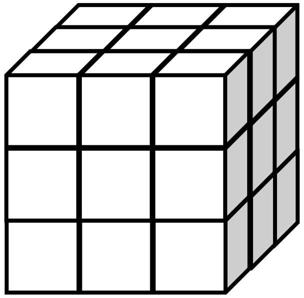
Team-synchronization overhead with nested parallelism

More parallelism available to map

Outline

- Core Abstractions and Capabilities
- Portable Hierarchical Parallel Execution Policies
- **Initial Scalable Graph Algorithms**
  - **Construction of sparse matrix graph from finite element mesh**
  - **Breadth first search of highly variable degree graph**
- Conclusion

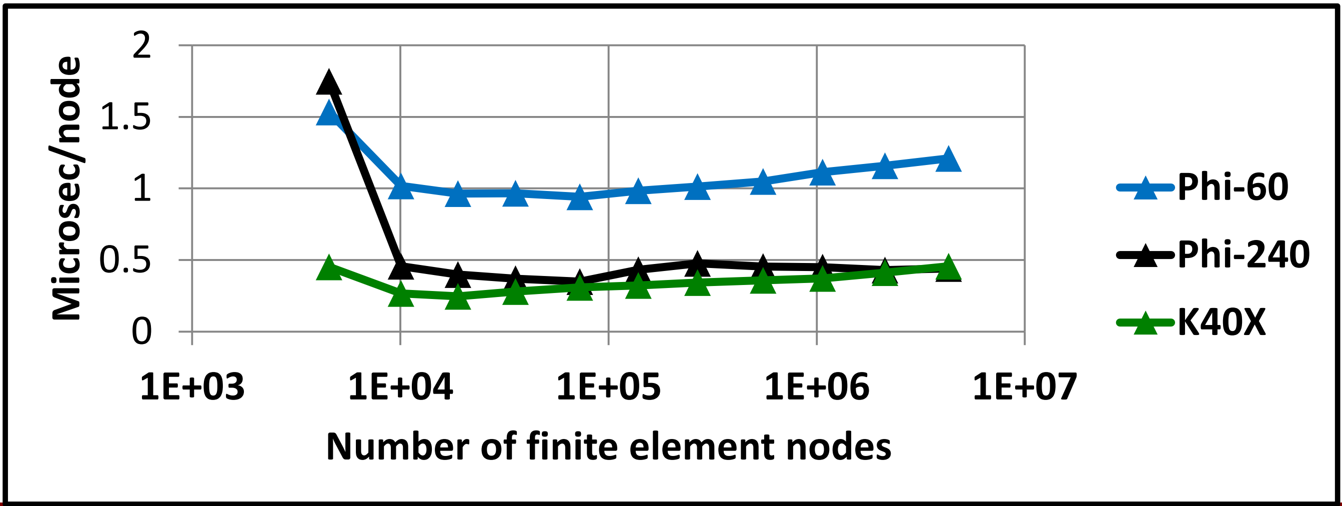
Thread-Scalable Construction of Sparse Matrix Graph from Finite Element Mesh



- **Given Finite Element Mesh Connectivity**
  - { element → { nodes } }
  - View<int*[8],Space> element_node ;
- **Generate node→node graph**
  - Compressed sparse row data structure
  - $\{(\boldsymbol{node}, \boldsymbol{column}(\boldsymbol{j})) : \forall\, \boldsymbol{j} \in [\boldsymbol{irow}(\boldsymbol{node}) \ldots \boldsymbol{irow}(\boldsymbol{node}+\mathbf{1})),\ \forall\, \boldsymbol{node}\}$
  - node = node index, irow = offset array, column(j) = connected node index
- **Challenges**
  - Determine unique node-node entries given redundant entries
    - { element → { nodes } } have shared faces and edges
  - Unknown number of node-node entries
  - Upper bound N^2 is too large to allocate

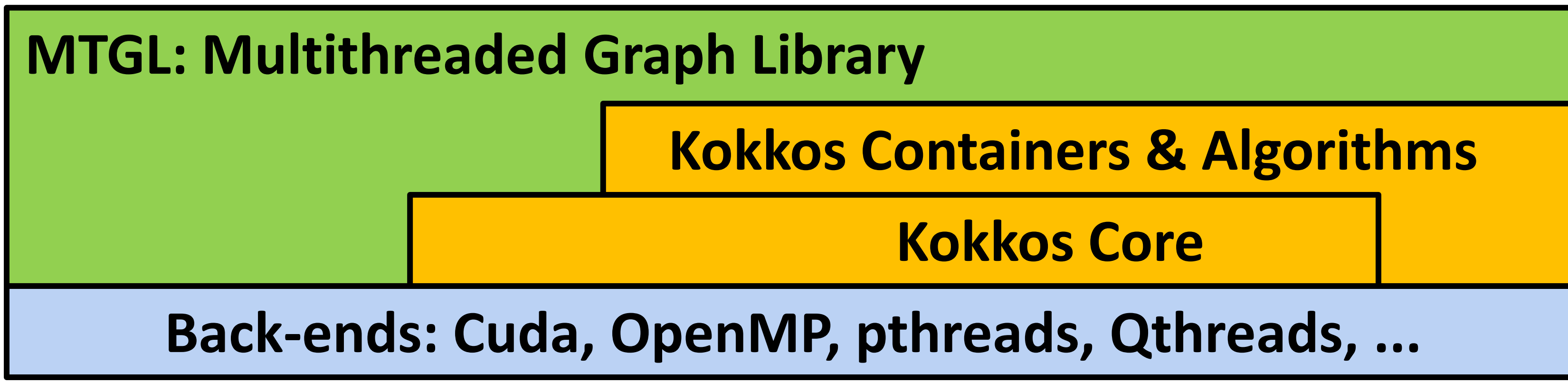
Thread-Scalable Construction of Sparse Matrix Graph from Finite Element Mesh

1. Parallel-for : fill Kokkos lock-free unordered map with node-node pairs
   - { element → { nodes } } : foreach element, foreach pair of nodes
   - Successful insert → atomic increment node's column counts
2. Parallel-scan : sparse matrix rows' column counts generates row offsets
   - Last entry is total count of unique node-node pairs
3. Allocate sparse matrix column-index array
4. Parallel-for : query unordered map to fill sparse matrix column-index array
   - foreach entry in unordered map of node-node pairs
5. Parallel-for : sort rows' column-index subarray



Breadth First Search of Graph with Highly Varied Degree Vertices

- **Porting portions of MTGL to GPU via Kokkos**
  - **MTGL: Sandia's multithreaded graph library**
  - **Internal laboratory directed research & development (LDRD) project**
  - **Sandia collaborators: Jonathan Berry and Greg Mackey**



- **Evaluate suitability of Kokkos and GPU for graph algorithms**
  - **MTGL previously threaded for CPU via Qthreads**
  - **Ease and performance of layering MTGL on Kokkos ?**
  - **Performance of MTGL algorithms on GPU ?**

Breadth First Search of Graph with Vertices of Highly Varying Degree

- **Iterative frontier-advancing algorithm (*conceptually* simple)**
  - **Given a frontier set of vertices**
  - **Foreach edge associated with each vertex in the frontier if edge's other vertex has not been visited, add to next frontier**
- **Challenges for thread-scalability**
  - **Maximizing parallelism in "foreach edge of each frontier vertex"**
  - **Removing load imbalance in "foreach edge of each frontier vertex"**
  - **Set of edges will not fit in GPU memory (set of vertices will fit)**
  - **Concurrent growth of global frontier set**
- **Strategy for thread-scalability**
  - **Manhattan loop collapse* of "foreach edge of each frontier vertex"**
  - **Thread-Team coordinated growth of global frontier set**

*** technique used in Cray and LLVM compilers**

Breadth First Search Algorithm

- **Graph implemented via compressed sparse row (CSR) scheme**
  - $\{(v, edge(j)) : \forall\, j \in [irow(v) \ldots irow(v+1)),\ \forall\, v\}$
  - **v = vertex index, irow = offset array, edge(j) = subarray of paired vertices**
- **Given search result array of vertices : search(*)**
  - **[0..a) = vertex indices accumulated from previous search iteration**
  - **[a..b) = vertex indices of current search frontier**

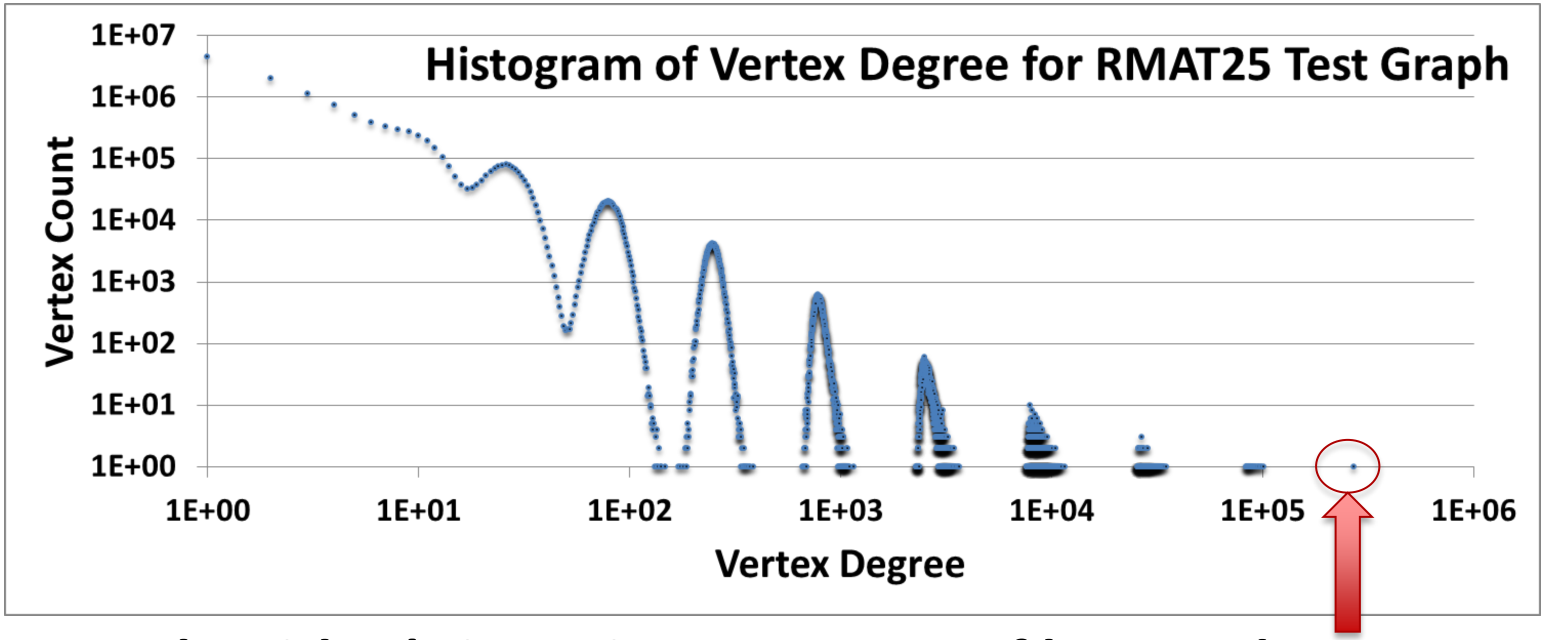
1. **Generate frontier vertex degree offset array 'fscan'**
   - **Frontier sub-array of vertex indices is search([a..b))**
   - **parallel_scan of vertex degrees (irow[v+1] – irow[v]) to generate fscan**
2. **Evaluate search frontier's edges, #edges = fscan(b) – fscan(a)**
   - **parallel_for via TeamPolicy, each team searches range of edges**
   - **Each thread evaluates vertices of collection of edges**
   - **Atomic update to determine if first visit, append thread-local buffer**
   - **Intra-team parallel_scan of local buffers to count team's search result**
   - **Append team's search to global search array, only one atomic update**
3. **Repeat for updated frontier**

Breadth First Search Algorithm

- **Maximizing parallelism**
  - **Manhattan loop collapse facilitates parallelizing over edges, not vertices**
  - **Removes load imbalance concerns for highly variable degree vertices**
- **Minimizing synchronization**
  - **Thread local buffer for accumulating search result**
  - **Intra-team parallel scan of thread local buffer sizes for team result size**
  - **Team's single atomic update of global search array**
- **Place arrays in appropriate memory spaces via Kokkos::View**
  - **Vertex arrays in GPU memory: irow(*), search(*), fscan(*)**
  - **Edge array in Host-Pinned memory: edge(*)**
- **Performance evaluation of portable implementation**
  - **Scalability for graphs with highly variable degree vertices**
  - **CPU vs. GPU**
  - **Edge array in GPU vs. Host-Pinned**

Breadth First Search Performance Testing

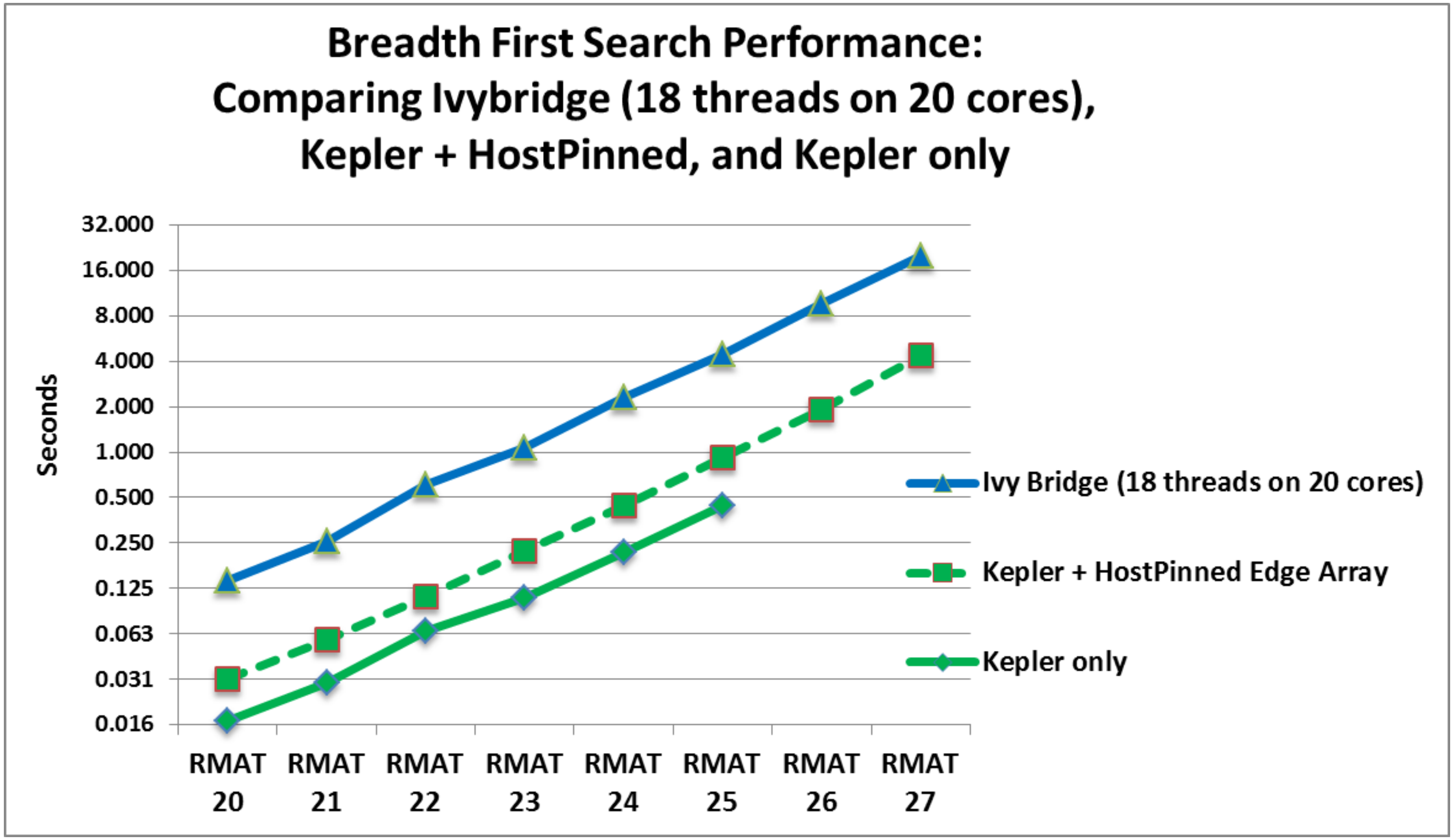
- **Sequence of generated test graphs**
  - **Doubling #vertices and #edges**
  - **Edges eventually cannot fit in GPU memory**
  - **Similar vertex degree histograms for all generated graphs**



- **Start algorithm's iteration on vertex of largest degree**

Breadth First Search Performance Testing

- Good scalability on Kepler
  - Teams stream through edge array with coalesced access pattern
  - Almost 2x performance drop reading edge array from Host Pinned memory
  - Enables processing of large graphs where edges cannot fit in GPU memory



Summary : Concepts and Abstractions

- κόκκος : “like grains of sand on a beach”
  - Identify / encapsulate grains of data and parallelizable operations
  - Aggregate these grains with data structure and parallel patterns
  - Map aggregated grains onto memory and cores / threads
- Mapping
  - User functions, execution spaces, parallel patterns, execution polices
  - Polymorphic multidimensional array, memory spaces, layout, access intent
  - Atomic operations
- Hierarchical Parallel Patterns
  - Maximizing opportunity (grains) for parallelism

Conclusion

- **Kokkos enables performance portability**
  - **parallel_pattern(ExecutionPolicy<ExecutionSpace> , UserFunction)**
  - **Polymorphic multidimensional arrays solves the array-of-structs versus struct-of-arrays dilemma**
  - **Atomic operations**
  - **Engaging with ISO/C++ Standard to advocate for these capabilities**
- **Pure library approach using C++ template meta-programming**
  - **Significantly simplified when UserFunction is a C++11 lambda**
  - **Cuda 7.5 candidate feature for device lambda : [=]__device__**
    - **Tell NVIDIA you like and want this!**
- **Thread team execution policy for hierarchical parallelism**
  - **Portable abstraction for Cuda grids, blocks, warps, and shared memory**
- **Early R&D for application to graph algorithms**