War Stories : Graph Algorithms in GPUs

Siva Rajamanickam(SNL)
George Slota, Kamesh Madduri (PSU)

FASTMath Meeting
Increasingly Complex Heterogeneous Future; ¿Future Proof Performance Portable Code?

**Memory Spaces**
- Bulk non-volatile (Flash?)
- Standard DDR (DDR4)
- Fast memory (HBM/HMC)
- (Segmented) scratch-pad on die

**Execution Spaces**
- Throughput cores (GPU)
- Latency optimized cores (CPU)
- Processing in memory

**Special Hardware**
- Non caching loads
- Read only cache
- Atomics

**Programming models**
- GPU: CUDA-ish
- CPU: OpenMP
- PIM: ??
Outline

- What is Kokkos (Slides from Kokkos Developers: Carter Edwards, Christian Trott, Dan Sunderland)
  - Layered collection of C++ libraries
  - Thread parallel programming model that managed data access patterns
- Graph Algorithms with OpenMP
- Graph Algorithms with Kokkos
- Conclusion
Kokkos: A Layered Collection of Libraries

- Standard C++, Not a language extension
  - In spirit of Intel’s TBB, NVIDIA’s Thrust & CUSP, MS C++AMP, ...
  - Not a language extension: OpenMP, OpenACC, OpenCL, CUDA

- Uses C++ template meta-programming
  - Currently rely upon C++1998 standard (everywhere except IBM’s xlC)
  - Prefer to require C++2011 for lambda syntax
    - Need CUDA with C++2011 language compliance

Application & Library Domain Layer

- Kokkos Sparse Linear Algebra
- Kokkos Containers
- Kokkos Core

Back-ends: OpenMP, pthreads, Cuda, vendor libraries ...
Kokkos’ Layered Libraries

- **Core**
  - Multidimensional arrays and subarrays in memory spaces
  - `parallel_for`, `parallel_reduce`, `parallel_scan` on execution spaces
  - Atomic operations: compare-and-swap, add, bitwise-or, bitwise-and

- **Containers**
  - `UnorderedMap` – fast lookup and thread scalable insert / delete
  - `Vector` – subset of `std::vector` functionality to ease porting
  - `Compress Row Storage (CRS) graph`
  - `Host mirrored & synchronized device resident arrays`

- **Sparse Linear Algebra**
  - Sparse matrices and linear algebra operations
  - Wrappers for vendors’ libraries
  - Portability layer for Trilinos manycore solvers
Kokkos Core: Managing Data Access

Performance Portability Challenge: Require Device-Dependent Memory Access Patterns

- **CPUs (and Xeon Phi)**
  - Core-data affinity: consistent NUMA access (first touch)
  - Hyperthreads’ cooperative use of L1 cache
  - Alignment for cache-lines and vector units

- **GPUs**
  - Thread-data affinity: coalesced access with cache-line alignment
  - Temporal locality and special hardware (texture cache)

- ¿ “Array of Structures” vs. “Structure of Arrays” ?
  - This is, and has been, the *wrong* question

Right question: Abstractions for Performance Portability?
Kokkos Core: Fundamental Abstractions

- Devices have Execution Space and Memory Spaces
  - Execution spaces: Subset of CPU cores, GPU, ...
  - Memory spaces: host memory, host pinned memory, GPU global memory, GPU shared memory, GPU UVM memory, ...
  - Dispatch *computation to execution space* accessing data in *memory spaces*

- Multidimensional Arrays, *with a twist*
  - Map multi-index (i,j,k,...) ↔ memory location *in a memory space*
  - Map is derived from an array *layout*
  - Choose layout for device-specific memory access pattern
  - Make layout changes transparent to the user code;
  - IF the user code honors the simple API: a(i,j,k,...)

Separates user’s index space from memory layout
Kokkos Core: Multidimensional Array Layout and Access Attributes

- Override device’s default array layout
  ```cpp
class View<double**[3][8], Layout, Device> a("a", N, M);
```
  - E.g., force row-major or column-major
  - Multi-index access is unchanged in user code
  - *Layout* is an extension point for blocking, tiling, etc.

- Example: Tiled layout
  ```cpp
class View<double**, TileLeft<8,8>, Device> b("b", N, M);
```
  - Layout changes are transparent to user code
  - IF the user code honors the `a(i,j,k,...)` API

- Data access attributes – user’s intent
  ```cpp
class View<const double**[3][8], Device, RandomRead> x = a;
```
  - Constant + RandomRead + GPU $\rightarrow$ read through GPU texture cache
  - Transparent to user code
Kokkos Core: Dispatch Data Parallel Functors

‘NW’ units of data parallel work

- **parallel_for( NW , functor )**
  - Call functor( iw ) with iw ∈ [0,NW) and #thread ≤ NW

- **parallel_reduce( NW , functor )**
  - Call functor( iw , value ) which contributes to reduction ‘value’
  - Inter-thread reduction via functor.init(value) & functor.join(value,input)
  - Kokkos manages inter-thread reduction algorithms and scratch space

- **parallel_scan( NW , functor )**
  - Call functor( iw , value , final_flag ) multiple times (possibly)
  - if final_flag == true then ‘value’ is the prefix sum for ‘iw’
  - Inter-thread reduction via functor.init(value) & functor.join(value,input)
  - Kokkos manages inter-thread reduction algorithms and scratch space
Kokkos Core: Dispatch Data Parallel Functors
League of Thread Teams (grid of thread blocks)

- `parallel_for( { #teams, #threads/team }, functor )`
  - Call `functor( teaminfo )`
  - `teaminfo = { #teams, team-id, #threads/team, thread-in-team-id }`

- `parallel_reduce( { #teams, #threads/team }, functor )`
  - Call `functor( teaminfo, value )`

- `parallel_scan( { #teams, #threads/team }, functor )`
  - Call `functor( teaminfo, value, final_flag )`

- A Thread Team has
  - Concurrent execution with intra-team collectives (barrier, reduce, scan)
  - Team-shared scratch memory
  - Exclusive use of CPU and Xeon Phi cores while executing
Outline

- What is Kokkos
- Graph Algorithms with OpenMP
- Graph Algorithms with Kokkos
- Conclusion
Computing Strongly Connected Components

- **Problem:** Given a graph find all the strongly connected components in the graph

- **Multistep method:**
  - Multithreaded with OpenMP
  - Optimized for the best CPU performance, state-of-the-art code.
  - Scales to millions of vertices and billions of edges
  - Data-Parallel code, minimal synchronization
  - Good as a baseline for porting to Kokkos
  - Scales to 16-32 threads

- FASTMath session in PP14 and IPDPS 14.
Multistep Method with OpenMP

- Different Steps of the Algorithm uses different types of parallelism
  - Per-vertex for-loop
  - Level synchronous BFS

![Graph and Speedup vs. Tarjan's plot](image-url)
Outline

- What is Kokkos
- Graph Algorithms with OpenMP
- Graph Algorithms with Kokkos
- Conclusion
Thread Parallel vs Thread Scalable

- Common construct in OpenMP programming
  - Allocate threadlocal data, do parallel work
  - Non- Starter in the GPUs for large arrays

- Count, Allocate, Fill, paradigms
  - Non- Starter for graph algorithms

- Need Algorithms that use tiny threadlocal data and synchronize with global memory
  - Tiny == 16 – 32 edges
  - Expensive, too many synchronizations

- Need Algorithms that use threadteams and shared memory (scratch space) between a team of threads.
Thread Teams in GPUs

- Multiprocessor (up to about 15/GPU)
  - Multiple groups of stream processors (12 × 16)
  - Warps of threads all execute SIMT on single group of stream processors (32 threads/warp, two cycles per instruction)
  - Irregular computation (high degree verts, if/else, etc.) can result in most threads in warp doing NOOPs

- Kokkos Model:
  - Thread team - multiple warps on same multiprocessor, but all still SIMT for GPU
  - Thread league - multiple thread teams
  - Work statically partitioned to teams before parallel code is called
Challenges in ThreadScalable codes

- **Goal:** Fast Kokkos-based ThreadScalable algorithm for CPU/GPU/Phi

- **Challenges:**
  - No persistent thread-local storage
  - Minimize serial portions for GPU/Phi
  - Mitigate effect of high degree vertices, irregular graphs
  - Mitigate algorithmic differences of various architectures

- **Solutions:**
  - Very small static thread-owned arrays
  - No more Tarjan’s, minimize possible per-thread work
  - Implement new algorithmic tweaks, for loadbalancing, to current methods
  - HOPE this doesn’t happen !!!!
Handling Imbalance in GPU threads

- **Chunking:**
  - Transform vertex queue into edge queue
  - Each thread can explore only a few edges and chunks the rest of the edges for later stages

- **Delayed Exploration of High Degree Vertices:**
  - When a single thread in a team encounters a high degree vertex, its exploration is delayed
  - The vertex is placed in shared memory queue, only accessible by thread team (just a template parameter in Kokkos)
  - Once team finishes original work (minus large degree vertices), the team works to explore all delayed vertices via inner loop parallelism
  - On CPU, size of thread teams is usually 1, so this algorithm would default back to standard approach on that architecture
Results - Algorithms

- **Multistep (M):** Simple trimming, Dir. Opt BFS, Coloring until less than 100k vertices remain with single thread exploration on backward stage, Serial Tarjan algorithm.

- **Multistep in Kokkos (MK):** Simple trimming, Dir. Opt BFS with small thread owned queues, coloring with fully parallel forward and backward, no Tarjan

- **GPU in Kokkos (GK):** Simple trimming, Dir. Opt BFS with chunking, coloring with delayed exploration

- **GPU min Memory in Kokkos (GKM):** Only utilize out edges, simple trimming, Forward BFS with chunking and fully bottom-up BFS on backward stage, forward coloring with delayed exploration and fully bottom-up reverse search
Results with Kokkos versions

- **Multistep (M)** is the fastest algorithm in CPU
- **GK** is the fastest algorithm in the GPU
- **Phi** is (a lot) slower
Conclusion

- Kokkos provides the path-forward for refactoring codes to different architectures
  - Handles data layout
  - Portable, ThreadScalable performance

- Algorithmic Challenges:
  - Still different algorithms perform better in different architectures
  - Hard to see a single refactor for algorithms in different architectures

- Kokkos programming is C++ programming not CUDA programming