Kokkos Evolution:
Task-DAG and Back-ends

DOE COE Performance
Portability
August 22-24, 2017
Denver, CO

H. Carter Edwards

SAND2017-8951 PE
Kokkos*

performance portability for C++ applications

LAMMPS
Albany
Drekar

SPARC
EMPIRE

Applications & Libraries

Trilinos

*ΚΟΚΚΟΣ* Greek: “granule” or “grain”; like grains of sand on a beach
Part 1: Kokkos’ Back-ends

- Map algorithms and arrays to underlying NGP node architecture
  - Productive, performance-portable abstractions / programming model
  - Map onto architecture’s best programming mechanism: CUDA, OpenMP, ...
  - Abstractions and programming mechanisms are evolving

Part 2: Kokkos’ Task-DAG Pattern/Policy

- Previously only data parallel patterns / policies
  - parallel_for, parallel_reduce, parallel_scan patterns over range policy [0..N)
  - Optional hierarchical thread team policy to maximize available parallelism

- New directed acyclic graph of tasks parallel patterns / policies
  - Tasks: Can be heterogeneous collection of parallel computations
  - DAG: Tasks may have acyclic execute-after dependences
  - Dynamic: Tasks can be spawned within executing tasks
FY17-18 evolution of Kokkos’ Back-ends

- **OpenMP for CPU and KNL+**
  - Require OpenMP 4+ for proper granularity of thread-binding
  - Compatibility / interoperability with nested parallel regions
    - Continue optional use of hwloc to choose performant sizes for nesting
  - Leverage matured OpenMP 4+ features
    - Scheduling, loop collapse, customized reductions, ...
  - Strategy for performant AMT / Kokkos / OpenMP interoperability
    - Outcome of collaboration with U-Utah’s “Uintah” framework

- **CUDA 9+ for NVIDIA GPU**
  - Proper collectives and controls provided by CUDA thread groups
    - Address warp divergence bug
  - Sub-block thread teams to improve flexibility of hierarchical parallelism
    - Best realized on Volta architecture
  - Full host-device lambda capability with C++17 capture: [=, *this]
FY17-18 evolution of Kokkos’ Back-ends

- C++11 std::thread for CPU and KNL+
  - Portability to OS/runtime that does not OpenMP (e.g., Windows)
  - Performance comparison with OpenMP
  - Research thread synchronization and collectives, runtimes

- Backend for ARM (CPU)
  - OS/runtime/compiler stack? tbd
  - Best thread parallel mechanism: OpenMP, std::thread, ...? tbd

- ROCm for AMD GPU; developed by AMD
- OpenMP 4.5 offload for GPU
- Clang-CUDA for GPU
Directed Acyclic Graph (DAG) of Tasks

- **Pattern**
  - Parallel execution of computations (tasks)
  - That have “execute after” dependences

- **Policy**
  - Scheduling ready tasks
  - Updating dependences as tasks complete

- **Dynamic and Heterogeneous Task-DAG**
  - Manage tasks’ lifecycle – tasks spawned within executing tasks
  - Manage tasks’ memory – task and workspace allocated/deallocated
  - Each task may be a different function (C++ closure)

- **Static and Homogeneous Work-DAG**
  - Single function, similar to data parallel patterns
  - Predefined DAG of “execute after” work indices: \{ k executes-after \{ i, j, ... \} \}
Task-DAG Motivating Use Cases

1. Multifrontal Cholesky factorization of sparse matrix
   - Frontal matrices require different sizes of workspace (green) for sub-assembly
   - Hybrid task parallelism: tree-parallel & matrix-parallel within supernodes (brown)
   - Dynamic task-dag with memory constraints
   - Matrix computation is internally data parallel
   - Lead: Kyungjoo Kim / SNL

2. Triangle enumeration in social networks, highly irregular graphs
   - Discover triangles within the graph
   - Compute statistics on those triangles
   - Triangles are an intermediate result that do not need to be saved / stored
   - Challenge: memory “high water mark”
   - Lead: Michael Wolf / SNL
Work-DAG Motivating Use Case

- **Neutral Particle Transport via Sweeps**
  - Tycho2 mini-application ([https://github.com/lanl/tycho2](https://github.com/lanl/tycho2))
  - “A neutral particle transport mini-app to study performance of sweeps on unstructured, 3D tetrahedral meshes.”
  - Lead: Kris Garrett / LANL

- **Tycho2 version using Kokkos Work-DAG**
  - All angle sweeps through unstructured mesh in a single DAG
  - Work index: \( K = \text{angle}\_\text{index} \times \text{number}\_\text{elements} + \text{element}\_\text{index} \)
  - Angle sweeps define work “execute after” dependences
  - Running on CPU and KNL - as of July 27, 2017
  - Next steps:
    - Port data structures to Kokkos for performance portability to GPU
    - Performance evaluation and improvements
Hierarchical, Thread Team Parallelism

- Shared functionality with hierarchical data-data parallelism
  - The same kernel (task) executed on ...
  - OpenMP: League of Teams of Threads
  - Cuda: Grid of Blocks of Threads

- Inter-Team Parallelism (data or task)
  - Threads within a team execute concurrently
  - Data: each team executes the same computation
    - Task: each team executes a different task

- Intra-Team Parallelism (data)
  - Nested parallel patterns: for, reduce, scan

- Mapping teams onto hardware
  - CPU: team == hyperthreads sharing L1 cache'
  - GPU: team == warp, for a modest degree of intra-team data parallelism
Dynamic Task DAG Challenges

- **A Dynamic DAG of Heterogeneous Functions (closures)**
  - Map functions onto a single thread or a thread team
  - **Scalable** dynamic allocation / deallocation of tasks
  - **Scalable** and low latency scheduling
  - **Scalable** dynamic creation / completion of execute-after dependences

- **GPU idiosyncrasies / constraints**
  - **Non-blocking tasks**, forced a beneficial “respawn” reconceptualization!
    - Eliminate context switching overhead: stack, registers, ...
  - Heterogeneous function pointers (CPU, GPU)
  - Creating GPU tasks on the host and within tasks executing on the GPU
  - Bounded memory pool and scalable allocation/deallocation
  - Non-coherent L1 caches
Scalable Memory Pool and Task Scheduler

- **Memory Pool**
  - Lock-free and low latency via atomic operations
  - Large chunk of memory allocated in Kokkos memory space
  - From which smaller blocks are allocated and deallocated

- **Task Scheduler**
  - Memory pool for tasks’ dynamic memory
  - Multiple prioritized ready queues
  - Per-task execute-after waiting queues
    - Each queue is a simple linked list of tasks
      - Lock free push/pop via atomic operations
      - Explicitly manage GPU non-coherent L1 cache
        - Problem: dynamic allocation/deallocation across GPU processors not automatically observed by GPU L1 cache hardware
        - Solution: explicitly manage via GPU programmable L1 cache, a.k.a. __shared__
Memory Pool Performance

- **Test Setup**
  - 10Mb pool comprised of 153 x 64k superblocks, min block size 32 bytes
  - Allocations ranging between 32 and 128 bytes; average 80 bytes
  - `parallel_for`: every index allocates; cyclically deallocates & allocates
  - Measure allocate + deallocate operations / second (best of 10 trials)
    - Deallocate much simpler and fewer operations than allocate

- **Test Hardware: Pascal, Broadwell, Knights Landing**
  - Fully subscribe cores
  - Every thread within every warp allocates & deallocates

- **For reference, an “apples to oranges” comparison**
  - CUDA malloc / free on Pascal
  - jemalloc on Knights Landing
Memory Pool Performance

<table>
<thead>
<tr>
<th></th>
<th>Fill 75%</th>
<th>Fill 95%</th>
<th>Cycle 75%</th>
<th>Cycle 95%</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>blocks:</strong></td>
<td>938,500</td>
<td>1,187,500</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pascal</td>
<td>79 M/s</td>
<td>74 M/s</td>
<td>287 M/s</td>
<td>244 M/s</td>
</tr>
<tr>
<td>Broadwell</td>
<td>13 M/s</td>
<td>13 M/s</td>
<td>46 M/s</td>
<td>49 M/s</td>
</tr>
<tr>
<td>Knights Landing</td>
<td>5.8 M/s</td>
<td>5.8 M/s</td>
<td>40 M/s</td>
<td>43 M/s</td>
</tr>
</tbody>
</table>

**apples to oranges comparison:**

<table>
<thead>
<tr>
<th></th>
<th>Fix 75%</th>
<th>Fix 95%</th>
<th>Cycle 75%</th>
<th>Cycle 95%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pascal using CUDA malloc</td>
<td>3.5 M/s</td>
<td>2.9 M/s</td>
<td>15 M/s</td>
<td>12 M/s</td>
</tr>
<tr>
<td>Knights Landing using jemalloc</td>
<td>379 M/s</td>
<td>4115 M/s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Memory pools have finite size with well-bounded scope
  - Algorithms’ and data structures’ memory pools do not pollute (fragment) each other’s memory
Scheduler Unit Test Performance

- (silly) Fibonacci task-dag algorithm measures overhead
  - \( F(k) = F(k-1) + F(k-2) \)
  - \( F(k) \) cumulatively allocates/deallocates \( N \) tasks >> “high water mark”
  - 1Mb pool comprised of 31 x 32k superblocks, min block size 32 bytes
  - Fully subscribe cores; single thread Fibonacci task consumes entire GPU warp
    - Real algorithms' tasks have modest internal parallelism
  - Measure tasks / second; compare to raw allocate + deallocate performance

<table>
<thead>
<tr>
<th></th>
<th>F(21)</th>
<th>F(23)</th>
<th>Alloc/Dealoc (for comparison)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cumulative tasks:</td>
<td>53131</td>
<td>139102</td>
<td></td>
</tr>
<tr>
<td>Pascal</td>
<td>1.2 M/s</td>
<td>1.3 M/s</td>
<td>144 M/s</td>
</tr>
<tr>
<td>Broadwell</td>
<td>0.98 M/s</td>
<td>1.1 M/s</td>
<td>24 M/s</td>
</tr>
<tr>
<td>Knights Landing</td>
<td>0.30 M/s</td>
<td>0.31 M/s</td>
<td>21 M/s</td>
</tr>
</tbody>
</table>
Tacho’s Sparse Cholesky Factorization

- **Multifrontal algorithm with bounded memory constraint**
  - Kokkos task DAG + Kokkos memory pool for shared scratch memory
  - Task fails allocation => respawn to try again after other tasks deallocate
  - Test setup: scratch memory size = M * sparse matrix supernode size
  - Compare to Intel’s pardiso, sparse matrix N=57k, NNZ=383k, 6662 supernodes

**Graphs**

- **Haswell (2x16x2)**
  - Factorization/minute vs. # threads
  - Peak memory MB vs. # threads

- **Knights Landing (1x68x4)**
  - Factorization/minute vs. # threads
  - Peak memory MB vs. # threads
Summary

- **Initial Task-DAG capability**
  - Portable: CPU and GPU architectures
  - Dynamic DAG of heterogeneous tasks
  - Hierarchical – thread-team data parallelism within tasks
  - Evaluation/improvement underway via sparse matrix factorization mini-app

- **Initial Work-DAG capability**
  - Portable: CPU and GPU architectures
  - Static DAG of work indices for single work function
  - Evaluation/improvement underway via sweep particle transport mini-app

- **Challenges conquered, esp. for GPU portability and performance**
  - Non-blocking (non-waiting) tasks ➔ new respawn pattern
  - Lock free, scalable memory pool and scheduler
  - GPU __shared__ memory to address non-coherent L1 cache