Kokkos
Hierarchical Task-Data Parallelism for C++ HPC Applications

GPU Tech. Conference
May 8-11, 2017
San Jose, CA

H. Carter Edwards

SAND2017-4681 C
Applications & Libraries

Kokkos*
performance portability for C++ applications

LAMMPS
Albany
Drekar

EMPIRE
SPARC

Trilinos

Multi-Core
Many-Core
APU
CPU+GPU

*KÓKKOΣ Greek: “granule” or “grain”; like grains of sand on a beach
Dynamic Directed Acyclic Graph (DAG) of Tasks

- **Parallel Pattern**
  - Tasks: *Heterogeneous* collection of parallel computations
  - DAG: Tasks may have acyclic “execute after” dependences
  - Dynamic: New tasks may be created/allocated by executing tasks

- **Task Scheduler Responsibilities**
  - Execute ready tasks
  - Choose from among ready tasks
  - Honor “execute after” dependences
  - Manage tasks’ dynamic lifecycle
Motivating Use Cases

1. Incomplete Level-K Cholesky factorization of sparse matrix
   - Block partitioning into submatrices
   - Given submatrix may/may not exist
   - DAG of submatrix computations
   - Each submatrix 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
     - Problem: memory “high water mark”
   - Lead: Michael Wolf / SNL
Hierarchical 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

- Intra-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
  - Nested parallel patterns: for, reduce, scan

- Mapping teams onto hardware
  - CPU : team == hyperthreads sharing L1 cache
    - Requires low degree of intra-team parallelism
  - Cuda : team == warp
    - Requires modest degree of intra-team parallelism
  - A year ago: team == block, infeasible high degree parallelism
Anatomy and Life-cycle of a Task

- **Anatomy**
  - Is a C++ closure (e.g., functor) of data + function
  - Is referenced by a `Kokkos::future`
  - Executes on a single thread or a thread team
  - May only execute when its dependences are complete (DAG)

- **Life-cycle:**
  -Serial task on a single thread
  -Executing
  -Waiting
  -Constructing
  -Task with internal data parallelism on a thread team
  -Complete
Dynamic Task DAG Challenges

- A DAG of heterogeneous closures
  - Map execution to a single thread or a thread team
  - Manage memory dynamically created and completed tasks
  - Manage DAG with dynamically created and completed dependences

- GPU – executing task cannot block or yield to another task
  - Forced a beneficial reconceptualization! Non-blocking tasks
  - Eliminate context switching overhead: stack, registers, ...

- Portability and Performance
  - 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
  - Scalable DAG management and scheduling
Managing a Non-blocking Task’s Lifecycle

- **Create: allocate and construct**
  - By main process or within another task
  - Allocate from a memory pool
  - Construct internal data
  - Assign DAG dependences

- **Spawn: enqueue to scheduler**
  - Assign DAG dependences
  - Assign priority: high, regular, low

- **Respawn: re-enqueue to scheduler**
  - Replaces waiting or yielding
  - Assign new DAG dependences and/or priority
  - Reconceived wait-for-child-task pattern
    - Create & spawn child task(s)
    - Reassign DAG dependence(s) to new child task(s)
    - Re-spawn to execute again after child task(s) complete
Task Scheduler and Memory Pool

- **Memory Pool**
  - Large chunk of memory allocated in Kokkos memory space
  - Allocate & deallocate small blocks of varying size within a parallel execution
  - Lock free, extremely low overhead
  - Tuning: min-alloc-size <= max-alloc-size <= superblock-size <= total-size

- **Task Scheduler**
  - Uses memory pool for tasks’ memory
  - Ready queues (by priority) and waiting queues
  - Each queue is a simple linked list of tasks
  - A ready queue is a head of a linked list
  - Each task is head of linked list of “execute after” tasks
  - Limit updates to push/pop, implemented with atomic operations
  - “When all” is a non-executing task with list of dependences for data
Memory Pool Performance, as of April’17

- **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, as of April’17

<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>Fill 75%</th>
<th>Fill 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, as of April’17

Test Setup, (silly) Fibonacci task-dag algorithm

- F(k) = F(k-1) + F(k-2)
  - if k >= 2 spawn F(k-1) and F(k-2) then
  - respawn F(k) dependent on completion of when_all( { 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/Dealloc (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>
Conclusion

✓ Initial Dynamic Task-DAG capability
  - Portable: CPU and NVIDIA GPU architectures
  - Directed acyclic graph (DAG) of heterogeneous tasks
  - Dynamic – tasks may create tasks and dependences
  - Hierarchical – thread-team data parallelism within tasks

- Challenges, primarily for GPU portability and performance
  - Non-blocking tasks ➔ respawn instead of wait
  - Memory pool for dynamically allocatable tasks
  - Map task’s thread-team onto GPU warp, modest intra-team parallelism
Ongoing Research & Development

- **In progress / to be resolved**
  - Work around warp divergence / fail-to-reconverge bug w/ CUDA 8 + Pascal
    - Known issue, Nvidia will soon have fix for us
  - Prevents task-team parallelism:
    - one thread per warp atomically pops task from DAG
    - whole warp executes task

- **In progress / to be done**
  - Merge Kokkos ThreadTeam and TaskTeam intra-team parallel capabilities
    - Currently are separate / redundant implementations
  - Performance evaluation & optimization
  - Performance evaluation with applications’ algorithms
    - sparse matrix factorization, social network triangle enumeration/analysis

- ... stay tuned