Parallel Task Frameworks for FMM

Patrick Atkinson, p.atkinson@bristol.ac.uk

Prof Simon McIntosh-Smith, simonm@cs.bris.ac.uk

University of Bristol

http://uob-hpc.github.io

University of BRISTOL

Motivation for an FMM mini-app

- Currently there's a wide landscape of ***tasking*** programming models
- Many differences in task interface, performance, and supported architectures
- Further, some programming models (e.g. OpenMP) have several different implementations, with large differences in performance
- Difficult to evaluate programmability and performance in this space due to a ***lack of motivating applications***
- Recent addition of GPU-side tasking in Kokkos

See our other mini-apps for heat-diffusion, hydro, particle transport and more:
http://uob-hpc.github.io/projects/

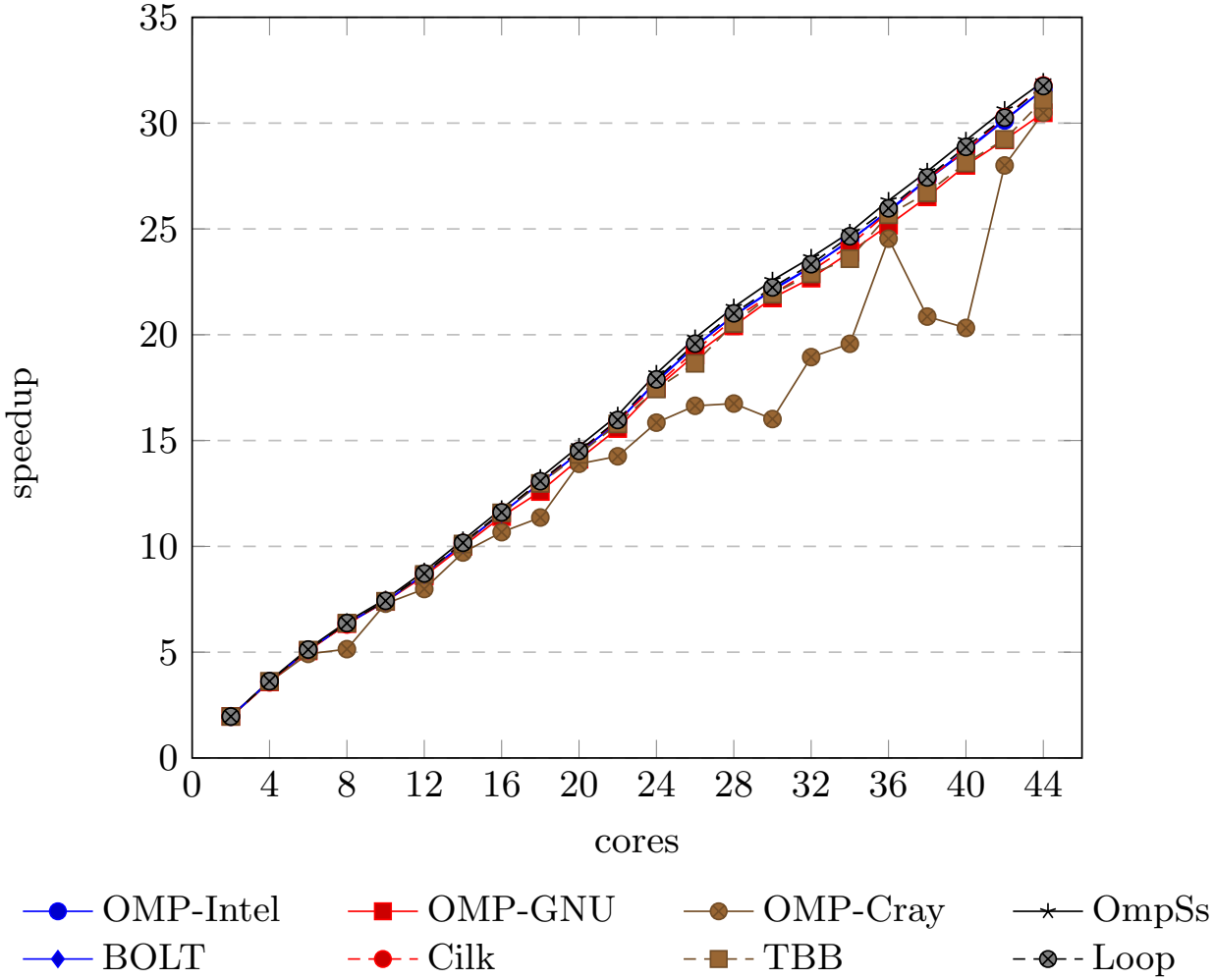
miniFMM

- Introducing a new Fast Multipole Method mini-app: **miniFMM**
- Implementations:
  - **CPU**: OpenMP, Intel TBB, CILK, Kokkos, OmpSs
  - **GPU**: CUDA, Kokkos
- Uses the **Dual Tree traversal method** – the schedule of node interactions is not known *a priori*, hence this is a **good test case for dynamic task parallelism**
- Small code base to enable testing against a wide variety of parallel programming models
- Open source: https://github.com/UoB-HPC/minifmm

Previous work: CPU results on Broadwell

- Previously miniFMM has been used to explore different tasking programming models on Xeon and Xeon Phi architectures
- Most OpenMP implementations, CILK, TBB, and OmpSs scale well
- Intel runtimes (OpenMP, CILK, TBB) and OmpSs perform best, whilst Cray and GCC lag behind
- Can be explained by measuring time spent within the OpenMP runtime:
  - Intel 2.01%
  - GNU 8.31%
  - Cray 9.13%

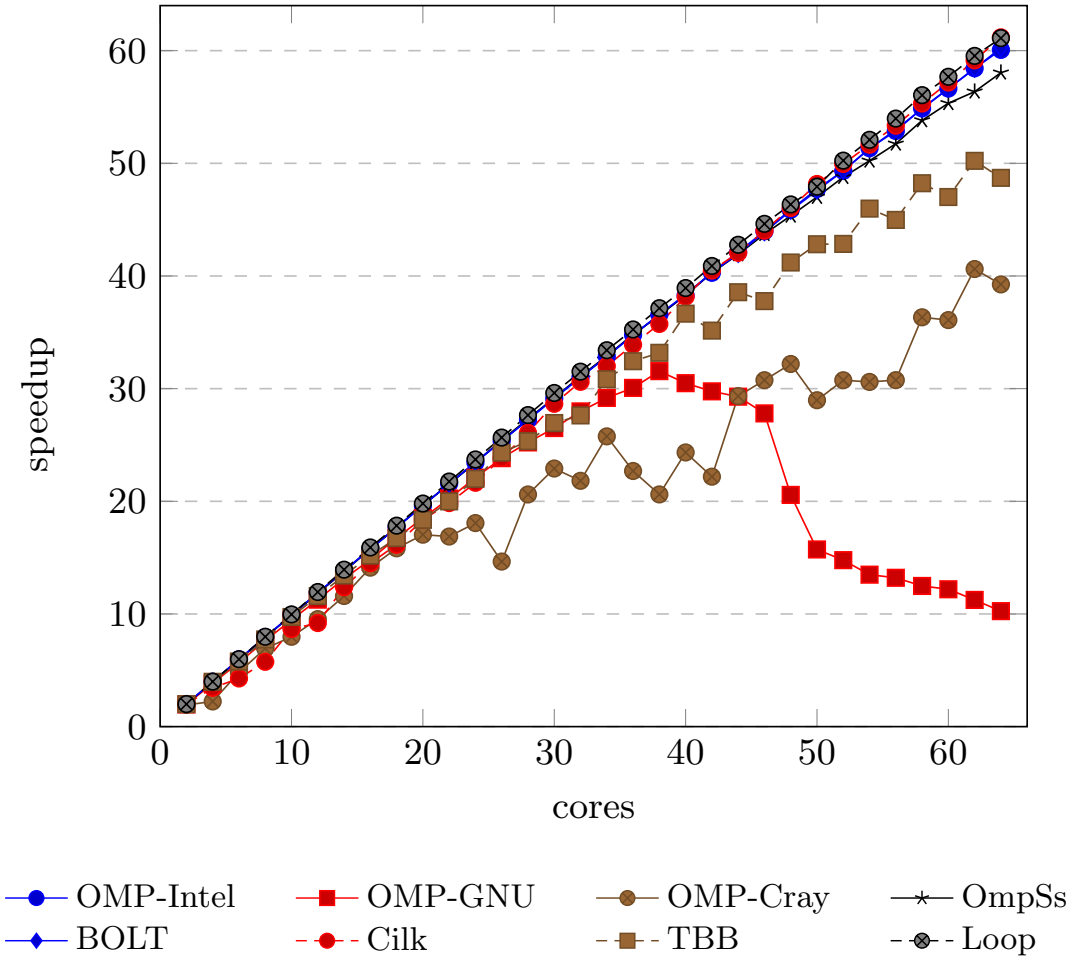
Intel Xeon Broadwell 44 cores, dual-socket, 88 threads



Previous work: CPU results on KNL

- Again, Intel parallel runtimes perform well, with TBB lagging slightly behind
- Good OmpSs performance required changing scheduler to use one task queue per thread, instead of a global queue
- Performance **degrades** >~120 threads using GCC

Intel Xeon Phi Knights Landing, 64 cores, up to 256 threads



Patrick won a “People’s Choice” award for this work at HPCDC

Patrick!

New results using Kokkos

Kokkos can now be used for dynamic task spawning on CPUs and GPUs!

Features of tasks in Kokkos:

- Manually have to allocate memory pool for tasks
- Future-based task dependencies
- Unlike other programming models, Kokkos doesn't rely on **taskwait** constructs
- Instead a task may **respawn** itself with new task dependencies

New results using Kokkos

Kokkos can now be used for dynamic task spawning on CPUs and GPUs!

Features of tasks in Kokkos:

- Manually have to allocate memory pool for tasks
- Future-based task dependencies
- Unlike other programming models, Kokkos doesn't rely on **taskwait** constructs
- Instead a task may **respawn** itself with new task dependencies
- Typically works as follows:
  1. A parent task is spawned and may spawn several tasks

```
class Task {
  Future f1, f2;

  void operator(){


    f1 = Kokkos::task_spawn(Kokkos::TaskSingle(...  , Task(n-1)));
    f2 = Kokkos::task_spawn(Kokkos::TaskSingle(...  , Task(n-2)));


  }
}
```

New results using Kokkos

Kokkos can now be used for dynamic task spawning on CPUs and GPUs!

Features of tasks in Kokkos:

- Manually have to allocate memory pool for tasks
- Future-based task dependencies
- Unlike other programming models, Kokkos doesn't rely on **taskwait** constructs
- Instead a task may **respawn** itself with new task dependencies
- Typically works as follows:
  1. A parent task is spawned and may spawn several tasks
  2. The parent task makes a call to respawn, taking the child task futures as arguments

```
class Task {
  Future f1, f2;

  void operator(){


    f1 = Kokkos::task_spawn(Kokkos::TaskSingle(... , Task(n-1)));
    f2 = Kokkos::task_spawn(Kokkos::TaskSingle(... , Task(n-2)));

    Kokkos::respawn(this, Kokkos::when_all({f1, f2}, 2));
  }
}
```

New results using Kokkos

Kokkos can now be used for dynamic task spawning on CPUs and GPUs!

Features of tasks in Kokkos:

- Manually have to allocate memory pool for tasks
- Future-based task dependencies
- Unlike other programming models, Kokkos doesn't rely on **taskwait** constructs
- Instead a task may **respawn** itself with new task dependencies
- Typically works as follows:
  1. A parent task is spawned and may spawn several tasks
  2. The parent task makes a call to respawn, taking the child task futures as arguments
  3. The parent task will be reinserted into the task queue and can be executed when the child tasks have completed

```
class Task {
  Future f1, f2;

  void operator(){
    if (!f1.is_null() && !f2.is_null()) {
      result = f1.get() + f2.get();
    } else {
      f1 = Kokkos::task_spawn(Kokkos::TaskSingle(... , Task(n-1)));
      f2 = Kokkos::task_spawn(Kokkos::TaskSingle(... , Task(n-2)));

      Kokkos::respawn(this, Kokkos::when_all({f1, f2}, 2));
    }
  }
}
```

Kokkos TaskSingle vs. TaskTeam

- When spawning a task, we can either spawn a **TaskSingle** or a **TaskTeam**
  - A TaskSingle will execute a task on a single thread
  - A TaskTeam will execute a task on a team of threads

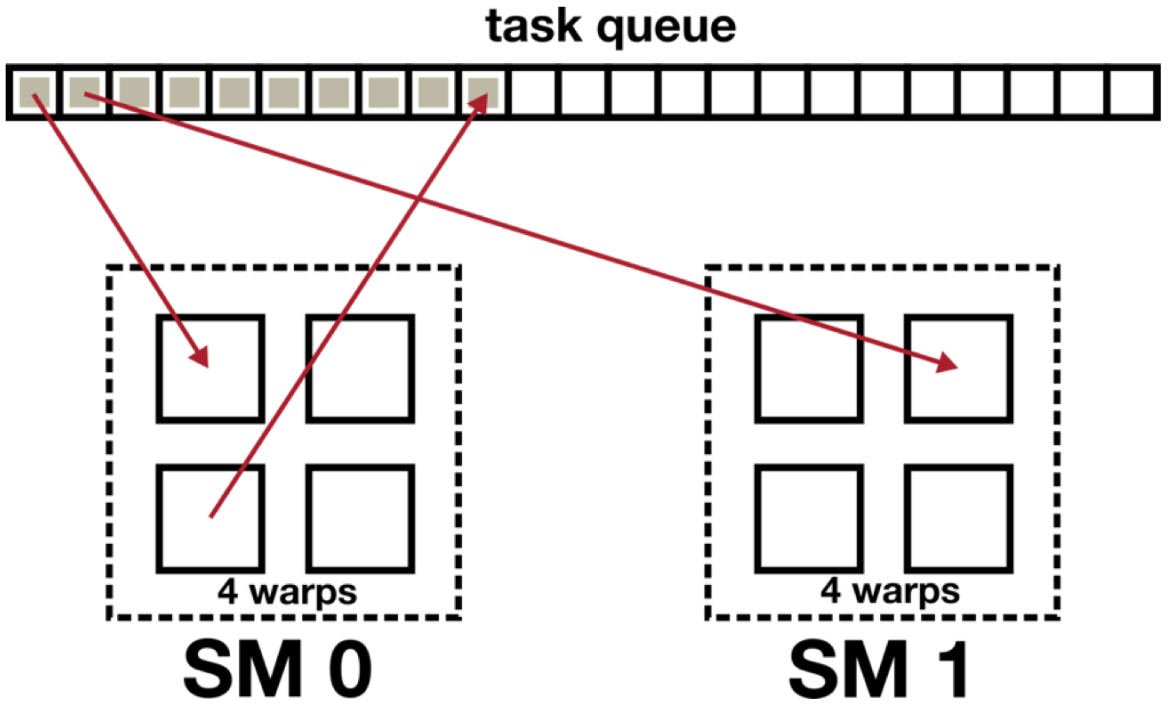
- A **team** will map to:
  - NVIDIA GPU: **a warp**
  - CPU: **a single thread**
  - Xeon Phi: **the hyper-threads of a single core**

```
// Launch a task to be executed by a single thread
Kokkos::task_spawn(Kokkos::TaskSingle(... , Task()));

// Launch a task to be executed by a team of threads
Kokkos::task_spawn(Kokkos::TaskTeam(... , Task()));
```

Kokkos GPU Task Queue Implementation

- Uses a single CUDA thread-block per SM
- All warps in all thread blocks pull from a single global task queue
- Warp lane #0 will pull tasks from the queue and, depending on the task type, either:
  - Execute a thread team task across the full warp, or
  - Execute a single thread task on lane #0, leaving the remaining threads in the warp idle
- Hence **optimal performance was only achieved through writing warp-aware code**



Warps of 2 SMs placing/acquiring tasks to/from the global task queue

CUDA Shared Memory in Kokkos GPU tasks

- Shared-memory is required for good performance in miniFMM on GPUs
- Data-parallel constructs in Kokkos allow for shared memory for a single team
- Shared-memory support is not yet complete for Task Policy in Kokkos
- Workaround is to declare shared memory statically and index warp-wise

```
Kokkos::View<float*, ...> shmem =
    Kokkos::View<float*, ...>(team.team_shmem(), 256);

shmem[i+lane] = global_array[i];
```

CUDA shared memory in data-parallel Kokkos

```
const int lane = team.team_rank();
const int num_warps = blockDim.z;
const int warp_id = threadIdx.z;

__shared__ float shmem_base[num_warps*256];
float* shmem = shmem_base[warp_id*256];

shmem[i+lane] = global_array[i];
```

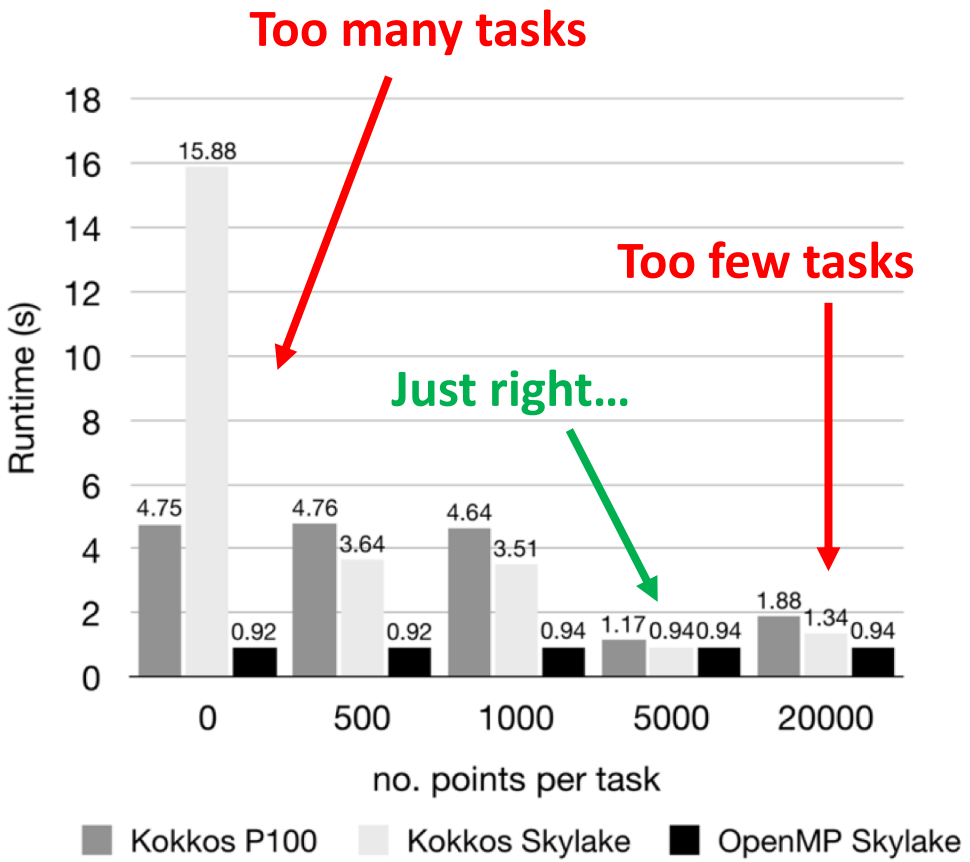
Work-around for shared memory in Kokkos task

Restricting Task Spawning for Improved Performance

- **Kokkos maintains a single task queue** – this is a similar problem to that in the GCC OpenMP runtime w.r.t. high task queue contention
- **Volta** has 80 SMs and 4 warp schedulers per SM, thus **320 warps contesting for access to the global queue simultaneously**
- Similarly, KNL could have up to 256 threads contesting the global queue simultaneously
- If we stop spawning tasks after a certain tree depth, we increase the time spent executing each task, ***and*** reduce the total number of tasks – reducing overall queue contention
- Hence ***we need to manually restrict task-spawning to achieve good performance***

Restricting Task Spawning for Improved Performance cont.

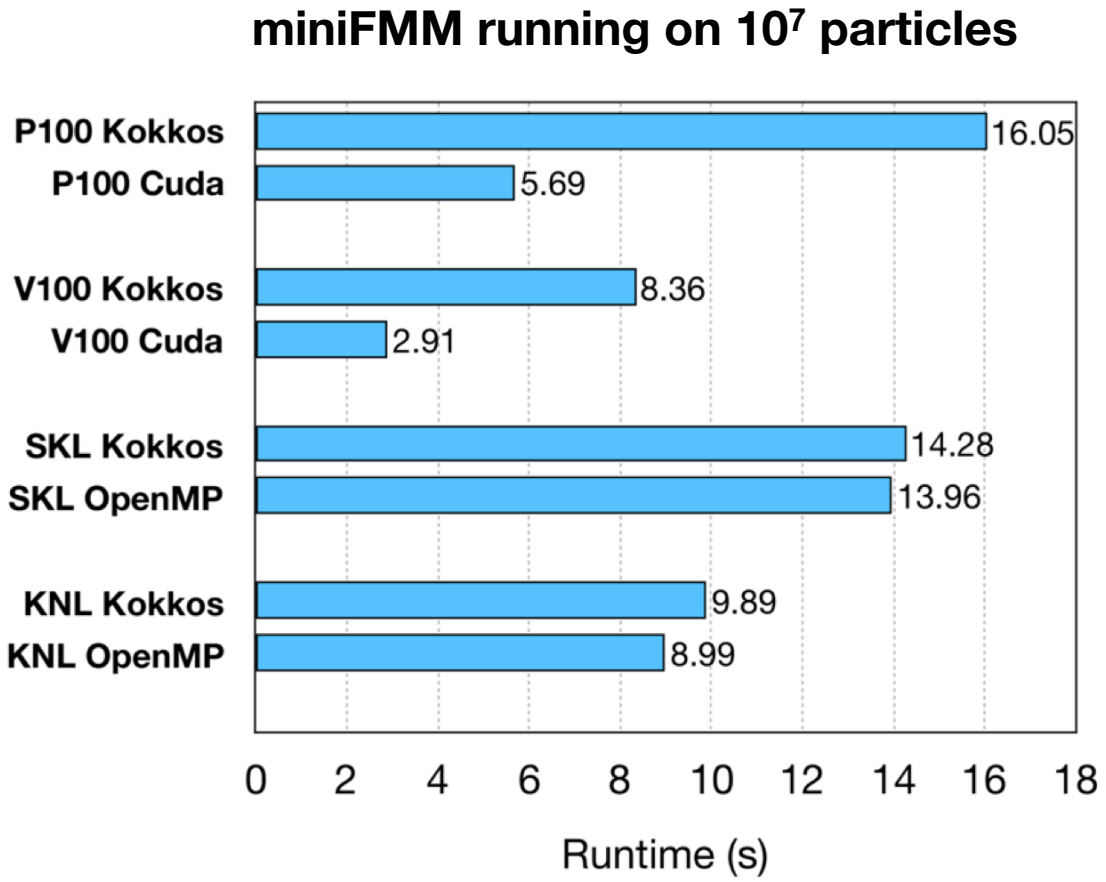
- **If we stop task spawning too low in the tree we create too many tasks for the scheduler**
- **If we stop tasking spawning too high in the tree, we lack parallelism**
- Both CPU and GPU Kokkos runtimes are heavily effected by this cut-off
- The Intel OpenMP runtime isn't affected at all since:
  - It maintains a task queue ***per thread***, which means less contention on a shared resource
  - It performs ***task-stealing***, so it can better handle the lack of parallelism



Skylake: Intel Xeon Skylake 56 core dual-socket

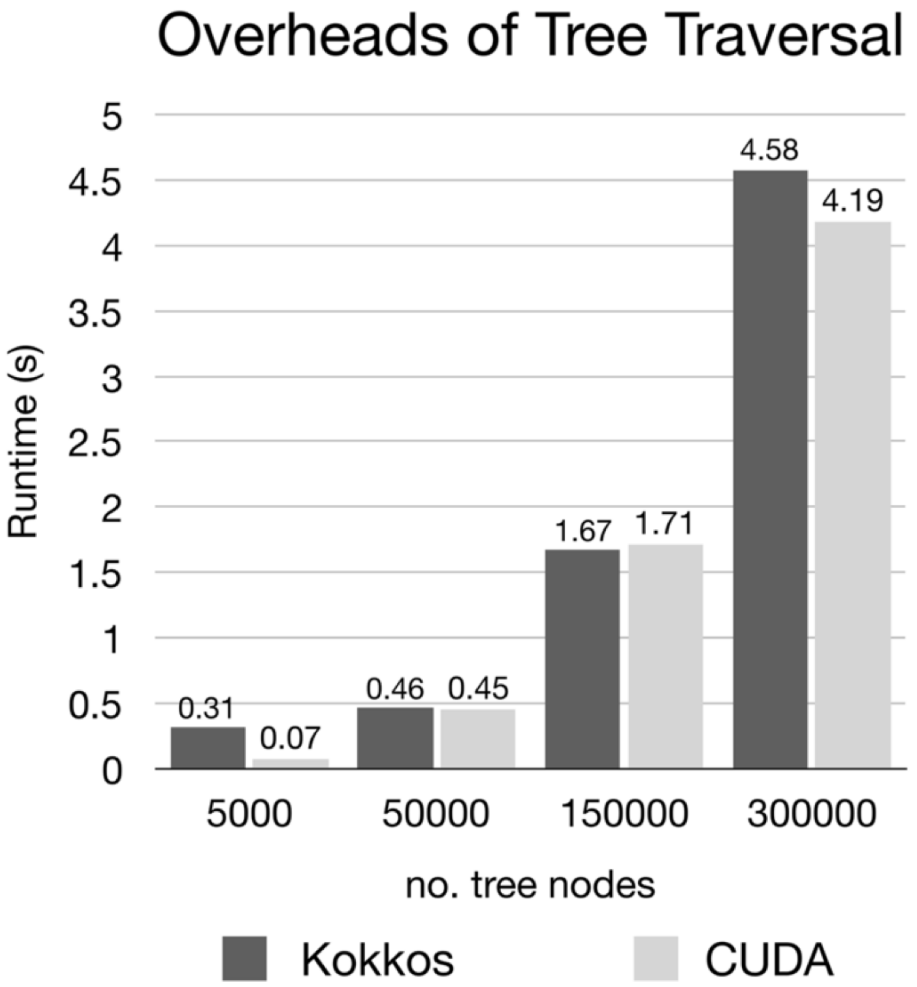
Results of miniFMM on GPUs and CPUs

- CUDA version of miniFMM finds lists of node-node interactions on the host, then transfers to the GPU. The GPU then iterates over interaction lists
- The Kokkos GPU tasking version is ~2.8x slower than CUDA, whilst the Kokkos CPU version is competitive with OpenMP
- However, Kokkos GPU tasks are new; miniFMM is one of the first applications to make use of them
- Volta is typically 2x faster than Pascal, due to its increased SM count and much higher shared-memory bandwidth



Reasons for the Performance Difference between CUDA and Kokkos

- High register pressure: ~200 registers per thread for Kokkos task vs. ~80 for kernels in the CUDA version
- Overhead of the tree traversal in each version is very similar, so the overall performance difference is due to performance of the computational kernels, ***not*** the traversal
- Some team constructs are not yet implemented in Kokkos, which could lead to better performance
- Kokkos only runs with 1 thread-block per SM with 128 threads per block – this could be another performance limiting factor



Summary

- FMM is a great application for exploring task-parallel programming models

- Overall task performance on the CPU is mostly good for FMM, with some problems at high thread counts

- Kokkos is increasingly important because it:
  - Targets both CPU and GPU architectures with (mostly) portable code
  - Supports dynamic task spawning on GPUs
  - Achieves reasonable performance - if you know what you're doing

Publications

Mini-apps including TeaLeaf, CloverLeaf, miniFMM, and SNAP:

http://uob-hpc.github.io/

On the performance of parallel tasking runtimes for an irregular fast multipole method application
Atkinson, Patrick and McIntosh-Smith, Simon, *International Workshop on OpenMP, 2017*

Assessing the performance portability of modern parallel programming models using TeaLeaf
Martineau, Matt, McIntosh-Smith, Simon, and Gaudin, Wayne, *Concurrency and Computation: Practice and Experience, 2017*

Many-core Acceleration of a Discrete Ordinates Transport Mini-app at Extreme Scale
Deakin, Tom, McIntosh-Smith, Simon N, and Gaudin, Wayne, *ISC High Performance, 2016*

The Productivity, Portability and Performance of OpenMP 4.5 for Scientific Applications Targeting Intel CPUs, IBM CPUs, and NVIDIA GPUs
Martineau, Matt and McIntosh-Smith, Simon, *International Workshop on OpenMP, 2017*

Extra slides

Differences Between CPU and GPU Implementations

- Structure of the tree traversal code can be identical if using TaskTeams
- Computational code might have to be written specific to architecture – e.g. if using shared memory on GPUs etc.
- Here, P2P and M2L kernels are written to utilise up to 32 threads, in the case we're executing on a GPU

```
class Task {
  node *target, *source;

  void operator(auto& team) {

    // calculate distance between nodes

    if (can_approximate(source, target)) {
      m2l(target, source);
    } else if (is_leaf(target) && is_leaf(source)) {
      p2p(target, source);
    } else {
      if (target->radius > source->radius) {
        for (child c : target->children) {
          Kokkos::task_spawn(
              Kokkos::TaskTeam, Task(c, source));
        }
      } else {
        for (child : source->children) {
          Kokkos::task_spawn(
              Kokkos::TaskTeam, Task(target, c));
        }
      }
    }
  }
}
```

Kokkos Memory Pool

- In contrast to other programming models, Kokkos requires user to manually allocate memory for tasks through the Kokkos memory pool class
- A memory pool is created by the programmer and associated with an instance of a task scheduler
- When `task_spawn` is called, the task's closure will be allocated from the memory pool
- If the allocations fails, due to exceeding memory-pool size, we will need to restart the computation
- This can be particularly problematic on GPUs as the host will need to expand the memory pool and restart the computation – this cannot be done on the device

```
// Total Memory Capacity of the memory pool
size_t MemoryCapacity = 16384;

// Specify the block properties of the memory pool
enum { MinBlockSize   =   64 };
enum { MaxBlockSize   = 1024 };
enum { SuperBlockSize = 4096 };

// Create task scheduler with memory pool
// of given parameters
sched_type root_sched( memory_space()
         , MemoryCapacity
         , MinBlockSize
         , MaxBlockSize
         , SuperBlockSize);

// Spawn the root task
Kokkos::Future f = Kokkos::host_spawn(
  Kokkos::TaskTeam(root_sched), Task(...));
```