The Rocky Road To Tasking

March 21, 2019 | Ivo Kabadshow, Laura Morgenstern | Jülich Supercomputing Centre
HPC ≠ HPC

CPU Cycle
Network Latency

ns μs ms s min h

Critical walltime

Requirements for MD
- Strong scalability
- Performance portability
HPC ≠ HPC

- CPU Cycle
- Network Latency
- Deep Learning
- Astrophysics

Critical walltime:
- ns
- μs
- ms
- s
- min
- h

Requirements for MD:
- Strong scalability
- Performance portability
HPC ≠ HPC

- CPU Cycle
- Network Latency
- Game Dev
- Astrophysics
- High Frequency Trading
- Deep Learning

Critical walltime:
- ns
- μs
- ms
- s
- min
- h

Requirements for MD:
- Strong scalability
- Performance portability
HPC ≠ HPC

Critical walltime

CPU Cycle
Network Latency
MD
Game Dev
Deep Learning
Astrophysics

High Frequency Trading

ns μs ms s min h
HPC ≠ HPC

CPU Cycle

Network Latency

High Frequency Trading

MD

Game Dev

Deep Learning

Astrophysics

Critical walltime

ns

μs

ms

s

min

h

Requirements for MD

- Strong scalability
- Performance portability
Our Motivation
Solving Coulomb problem for Molecular Dynamics

Task: Compute all pairwise interactions of $N$ particles

N-body problem: $\mathcal{O}(N^2) \rightarrow \mathcal{O}(N)$ with FMM

Why is that an issue?

- MD targets < 1ms runtime per time step
- MD runs millions or billions of time steps
- not compute-bound, but synchronization bound
- no libraries (like BLAS) to do the heavy lifting

We might have to look under the hood ... and get our hands dirty.
Parallelization Potential

Classical Approach
- Lots of independent parallelism

Algorithmic Complexity
- Classical $O(N^2)$
Fast Multipole Method (FMM)

- Many dependent phases
- Varying amount of parallelism
Coarse-Grained Parallelization

Input

P2M  M2M  M2L  L2L  L2P  P2P  Output

synchronization points

Different amount of available loop-level parallelism within each phase
Some phases contain sub-dependencies
Synchronizations might be problematic
Coarse-Grained Parallelization

- Different amount of available loop-level parallelism within each phase
- Some phases contain sub-dependencies
- Synchronizations might be problematic
FMM Algorithmic Flow

Multipole to multipole (M2M), shifting multipoles upwards

\[ d = 0 \]

\[ 1 \]

\[ 2 \]

\[ 3 \]

\[ 4 \]
FMM Algorithmic Flow

Multipole to multipole (M2M), shifting multipoles upwards

\[ d = 0 \]

\[ 1 \]

\[ 2 \]

\[ 3 \]

\[ 4 \]

Dataflow – Fine-grained Dependencies
FMM Algorithmic Flow

Multipole to local (M2L), translate remote multipoles into local taylor moments

\[ d = 0 \]

\[ 1 \]

\[ 2 \]

\[ 3 \]

\[ 4 \]
FMM Algorithmic Flow

Multipole to local (M2L), translate remote multipoles into local Taylor moments

\[ d = 0 \]

\[ + \]

\[ + + \]

\[ + + + \]

Dataflow – Fine-grained Dependencies
FMM Algorithmic Flow

Local to local (L2L), shifting Taylor moments downwards

\[ d = 0 \]

\[ 1 \]

\[ 2 \]

\[ 3 \]

\[ 4 \]
FMM Algorithmic Flow
Local to local (L2L), shifting Taylor moments downwards

\[ d = 0 \]

Dataflow – Fine-grained Dependencies

\[ \mu \]

\[ \mu \]

\[ \mu \]

\[ \mu \]
CPU Tasking Framework

Diagram:
- Queue
- Scheduler
- ThreadingWrapper
- Thread
- Core

Member of the Helmholtz Association
March 21, 2019
Slide 12
CPU Tasking Framework

Queue
Scheduler
ThreadingWrapper
Thread
Core

Member of the Helmholtz Association
March 21, 2019
CPU Tasking Framework

- Queue
- Scheduler
- ThreadingWrapper
- Thread
- Core
- Dispatcher
- TaskFactory
- LoadBalancer
CPU Tasking Framework
CPU Tasking Framework

Task life-cycle per thread

- TaskFactory
- LoadBalancer
- Queues
- Task execution

- Tasks can be prioritized by task type
- Only ready-to-execute tasks are stored in queue
- Workstealing from other threads is possible
CPU Tasking Framework

Task life-cycle per thread

- Dispatcher
- TaskFactory
- LoadBalancer
- Queues
- Task execution

- Tasks can be prioritized by task type
- Only ready-to-execute tasks are stored in queue
- Workstealing from other threads is possible
CPU Tasking Framework

Task life-cycle per thread

Dispatcher

Queues

Task execution

Tasks can be prioritized by task type

Only ready-to-execute tasks are stored in queue

Workstealing from other threads is possible
CPU Tasking Framework

Task life-cycle per thread

TaskFactory

Dispatcher

Queues

Task execution

Tasks can be prioritized by task type

Only ready-to-execute tasks are stored in queue

Workstealing from other threads is possible
CPU Tasking Framework

Task life-cycle per thread

- Tasks can be prioritized by task type
- Only ready-to-execute tasks are stored in queue
- Workstealing from other threads is possible
Tasking Without Workstealing

103 680 Particles on 2×Intel Xeon E5-2680 v3 (2×12 cores)
Tasking With Workstealing

103 680 Particles on 2×Intel Xeon E5-2680 v3 (2×12 cores)
GPU Tasking

Goal

- Provide same features as CPU tasking:
  - Static and dynamic load balancing
  - Priority queues
  - Ready-to-execute tasks
GPU Tasking
Uniform Programming Model for CPUs and GPUs

Thread

issues vector instruction

Persistent Warp

issues 32 scalar instructions

Processing Block

Warp Scheduler

FPU

FPU

FPU

FPU
GPU Tasking

Uniform Programming Model for CPUs and GPUs

SMT-Threads

run on

Persistent Thread Block

runs on

Streaming Multiprocessor
GPU Tasking
Uniform Programming Model for CPUs and GPUs

Many SMT-Threads

Many Persistent Thread Blocks

CPU

GPU

Member of the Helmholtz Association

March 21, 2019
GPU Tasking

Uniform Programming Model for CPUs and GPUs

Persistent Thread

issues scalar instruction

Persistent Warp

issues 32 scalar instructions

Persistent Thread Block

runs on

Many Persistent Thread Blocks

run on

FPU

Warp Scheduler

Processing Block

Streaming Multiprocessor

GPU

Member of the Helmholtz Association

March 21, 2019

Slide 17
Pitfalls

Performance Portability

Diverse GPU programming approaches:
- OpenCL
- CUDA
- SYCL

Our requirements:
- Strong subset of C++11
- Portability between GPU vendors
- Tasking features
- Maturity

(Intermediate) Solution

Use CUDA for reasons of performance, specific tasking features and maturity. Take the loss of not being portable out of the box.
Pitfalls

Performance Portability

For performance portability we consider diverse GPU programming approaches:

- OpenCL
- CUDA
- SYCL

Unsatisfying (Intermediate) Solution

Use CUDA for reasons of performance and specific features. Take the loss of not being portable out of the box.
# Pitfalls

## Architectural Differences

### Pitfalls for Load Balancing
- No thread pinning
- No cache coherency

### Pitfalls for Mutual Exclusion
- Weak memory consistency
- Missing forward progress guarantees
Pitfalls
Load Balancing

- No possibility to pin threads to streaming multiprocessors
- No direct access to shared memory of other streaming multiprocessors
- Work stealing requires multi-producer multi-consumer queues → Mechanism for mutual exclusion?
Pitfalls

Mutual Exclusion

- Weak memory consistency
- Warp-synchronous deadlocks due to lock step
- How to prove thread safety?
class Mutex
{
__inline__ __device__ void lock()
{
  while (atomicCAS(&mutex, 0, 1) != 0)
    __threadfence();
}
__inline__ __device__ void unlock()
{
  __threadfence();
  atomicExch(&mutex, 0);
}
int mutex = 0;
};
Very First Evaluation

Conditions

- Tasking with global queue only
- Measurements without work load to determine enqueue and dequeue overhead
- Measurements on P100 with 56 thread blocks with 1024 threads each
- Measurements on V100 with 80 thread blocks with 1024 threads each
First Evaluation

Tasking Overhead on P100 and V100
GPU Tasking

Conclusion

- Fine-grained task parallelism pays off on CPUs
- Developed mapping between CPU and GPU concepts
- (Partly) overcome pitfalls:
  - Lock-based mutual exclusion
  - Reusability of CPU tasking code
  - Architectural differences between CPU and GPU
- Successfully transferred parts of CPU tasking to GPUs
Next Steps

- Analyze and solve performance issues in dependency resolution
- Use memory pool for dynamic allocations
- Implement hierarchical queues
- Transfer priority queue to GPU
- Exploit data-parallelism through warps
- Consider the use of lock-free data structures
- Implement FMM based on GPU tasking
Thank You to Our Sponsor!

NVIDIA Tesla V100 and NVIDIA Tesla P100 where provided by MEGWARE