TECHNISCHE UNIVERSITÄT CHEMNITZ

Tasking Meets GPUs: Fighting Deadlocks and Other Monsters

L. Morgenstern[1,2], A. Beckmann[2], I. Kabadshow[2], M. Werner[1]
[1]Operating Systems Group, Chemnitz University of Technology
[2]Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH

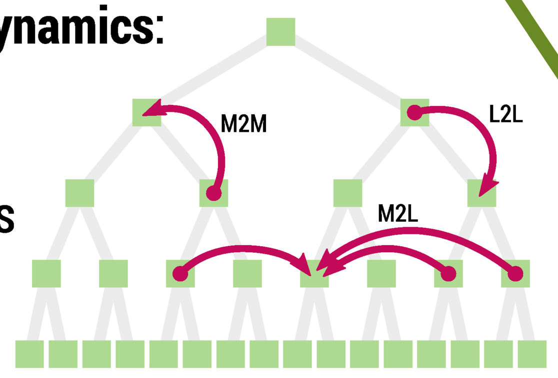
Motivation

Task parallelism is omnipresent these days; whether in data mining or machine learning, for matrix factorization or even molecular dynamics. Despite the success of task parallelism on CPUs, there is currently no performant way to exploit task parallelism of synchronization-critical algorithms on GPUs. Hence, our **goal** is the development of a task-based programming model to exploit fine-grained task parallelism on heterogeneous hardware.

Use Case

Fast Multipole Method for molecular dynamics:

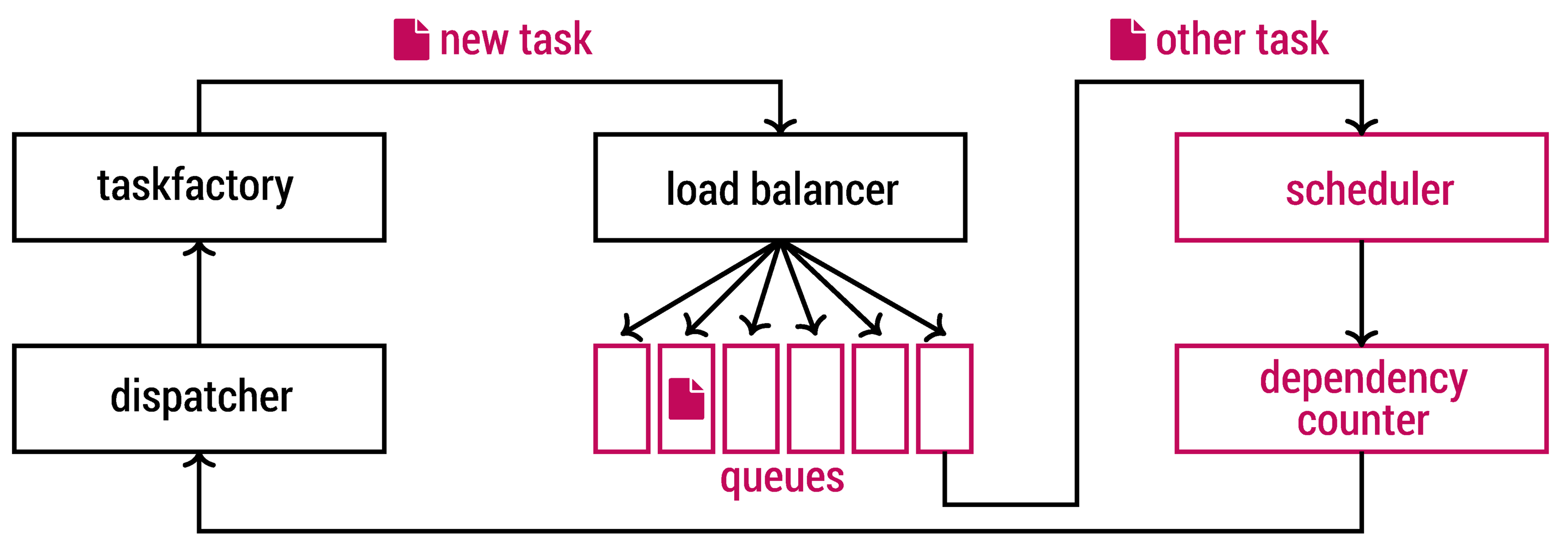
1. Expand particles on lowest level into multipole moments
2. Transfer multipole moments upwards
3. Translate multipole moments into local moments
4. Transfer local moments downwards
5. Translate local moments to particles on lowest level
6. Evaluate near field interactions



Requirements

- **Correctness**
  - Mechanism for mutual exclusion
- **Scalability**
  - Fine-grained task parallelism
  - Dynamic load balancing
- **Portability**
  - Uniform code path for CPU and GPU

Uniform Tasking for CPUs and GPUs...



See [1].

Performance Portability

- **Diverse GPU programming approaches**, e.g. CUDA, HIP, OpenCL, SYCL
- **Our requirements for programming approach**:
  1. Strong subset of C++11
  2. Tasking features
  3. Maturity and sustainability
  4. Portability between GPU vendors
- **Intermediate solution**: Use CUDA and forget about 4. (for now).

Mutual Exclusion

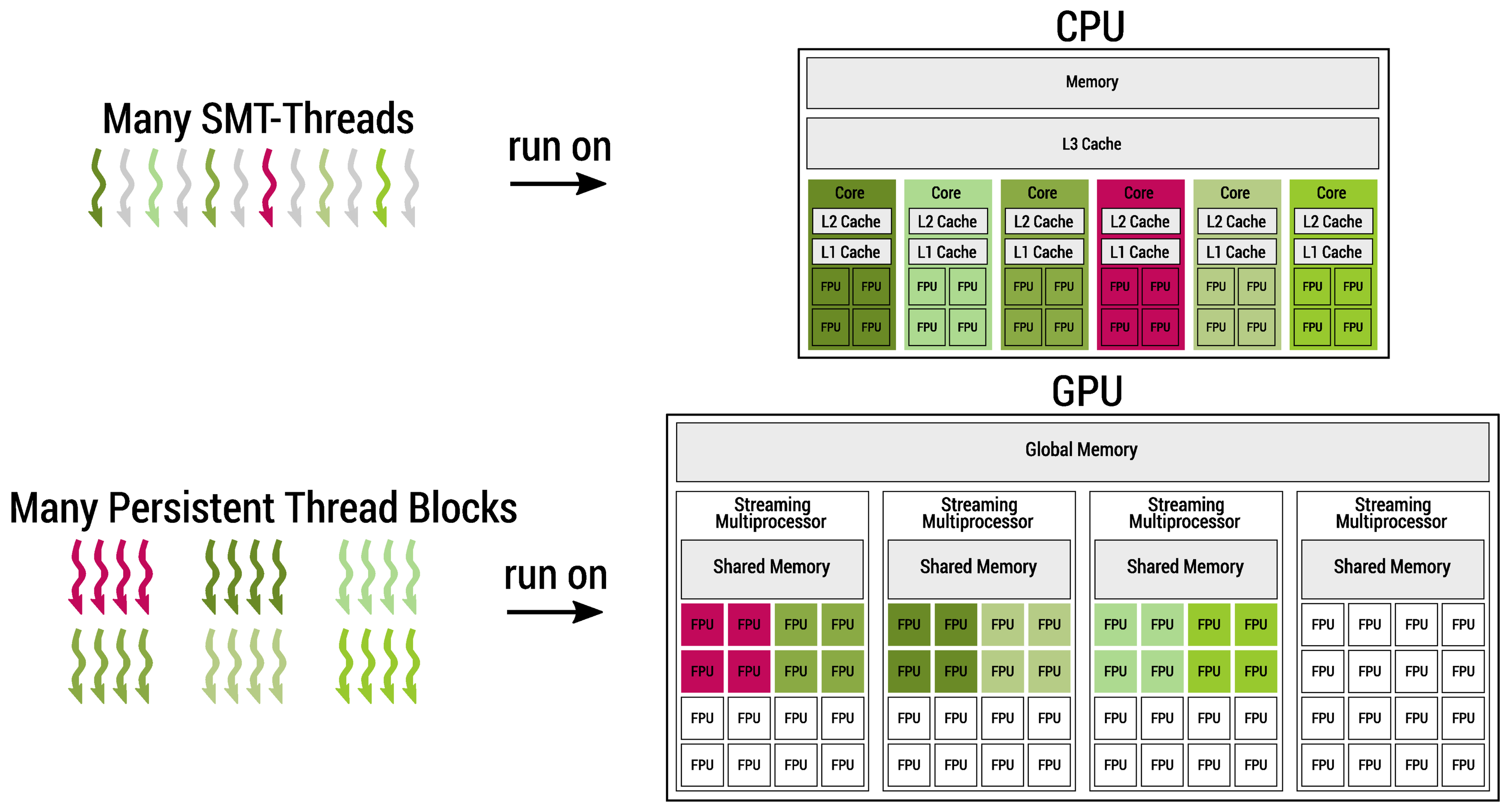
- Problem: CUDA doesn't provide a mutex
- Heavily dependent on use case
- Assumption for our use case: threads in a warp never compete for the same lock
- Implement spin-lock based mutex by means of atomic operations
- Take weak memory cosistency into account by means of memory fencing [2]

```
class Mutex
{
    __inline__ __device__ void lock()
    {
        while (atomicCAS(&mutex, 0, 1) != 0)
        __threadfence();
    };
    __inline__ __device__ void unlock()
    {
        __threadfence();
        atomicExch(&mutex, 0);
    };

    int mutex = 0;
};
```

... Requires a Uniform Machine Model

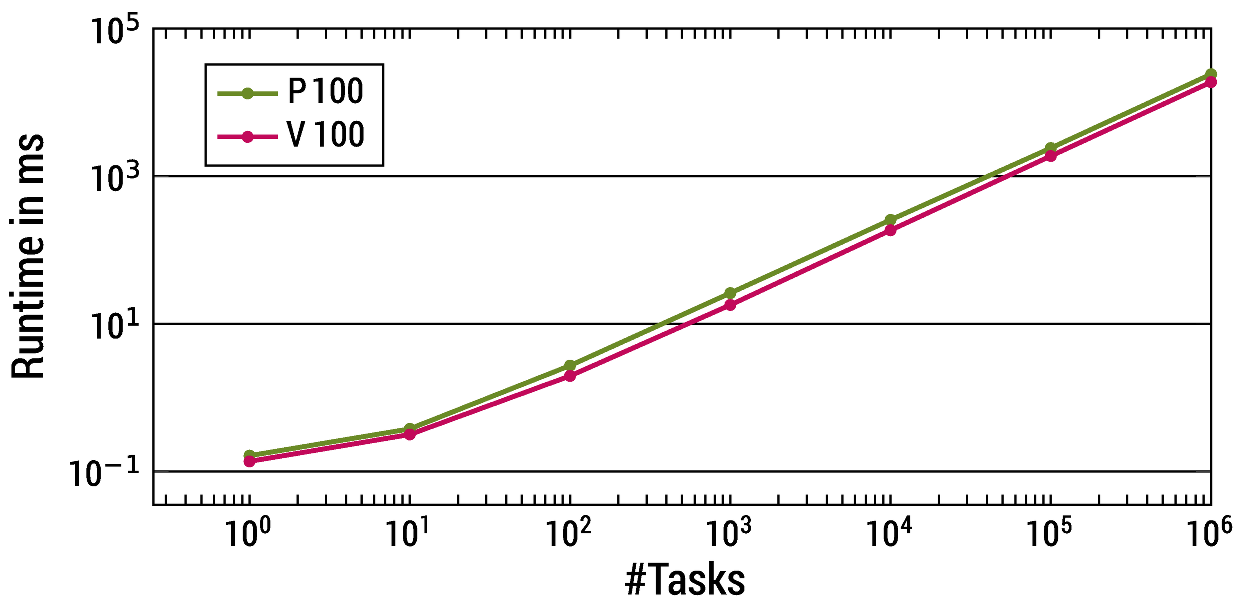
To bridge the gap between CPU and GPU regarding machine and programming model, we view warps as SIMD-units that execute **SIMD-tasks**. Moreover, we use **persistent threads** [3] to emulate the classical CPU threads on the GPU to support fine-grained task-parallelism. Each persistent thread acts as producer and consumer of tasks; meaning that each persistent thread contributes to scheduling and dependency resolution. Currently, all persistent threads access a **global task queue** concurrently. Thread safety is assured via spin-lock based mutexes.



Dynamic Memory Allocation

In our current tasking model dynamic, fine-grained task-parallelism requires plenty of small dynamic memory allocations. Using the built-in CUDA-allocator, this leads to a heavy loss in performance. Due to this, we use **ScatterAlloc** [4] as alternative allocator that supports dynamic memory allocations on massively parallel architectures. Furthermore, we work on an approach that uses static memory allocation only.

First Performance Results



- **Measurements**:
  - 1024 threads per thread block
  - P100 (56 SMs) – 56 thread blocks
  - V100 (80 SMs) – 80 thread blocks
  - Global queue only
  - No work load to measure overhead
- **Conclusion**:
  - Overhead to enqueue/dequeue a task is constant

Thread-block local priority queues

08/2019

Merge task-based FMM and GPU-tasking

11/2019

JÜLICH Forschungszentrum

References

[1] D. Haensel. A C++ based MPI-enabled Tasking Framework to Efficiently Parallelize Fast Multipole Methods for Molecular Dynamics. PhD Thesis, TU Dresden, 2018.
[2] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen and J. Wickerson. GPU Concurrency: Weak Behaviours and Programming Assumptions. SIGPLAN Not. 50,4 (March 2015), pp. 577-591.
[3] K. Gupta, J. A. Stuart and J. D. Owens. A study of Persistent Threads style GPU programming for GPGPU workloads. 2012 Innovative Parallel Computing (InPar), San Jose, CA, 2012, pp. 1-14.
[4] M. Steinberger, M. Kenzel, B. Kainz and D. Schmalstieg. ScatterAlloc: Massively Parallel Dynamic Memory Allocation for the GPU. In Proceedings of inPar 2012, San Jose, USA.