GPUs, Transactional Memory

Ryan Lee

6.823 Fall 2021

Adapted from prior course offerings

GPU: Graphics Processing Unit

- Originally designed as a graphics acceleration engine
- Has evolved into a hardware accelerator for massively parallel applications

- Exploit parallelism to achieve higher throughput, performance
  - Hide latency by massive multi-threading

Why care about GPUs?

» Massive data parallelism in today's popular workloads
  - CNNs, ML
  - Graph analytics

Types of Parallelism

» ILP: Instruction-level parallelism
  - Between independent instructions in a sequential program

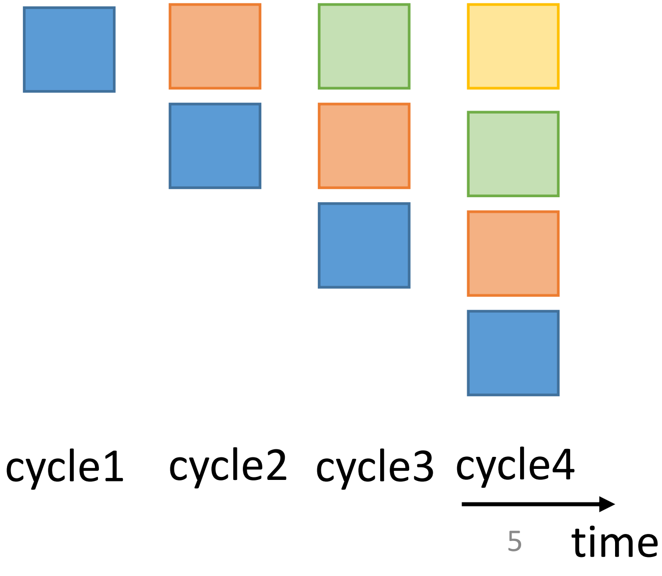
» TLP: Thread-level parallelism
  - Between independent execution contexts (threads)

» DLP: Data-level parallelism
  - Between elements of a vector (say); same operation on multiple elements

How to Utilize Parallelism?

» Horizontal parallelism: More units working in parallel

» Vertical parallelism: Pipelining: Keep units busy when waiting for memory dependences etc.



How to Extract Parallelism?

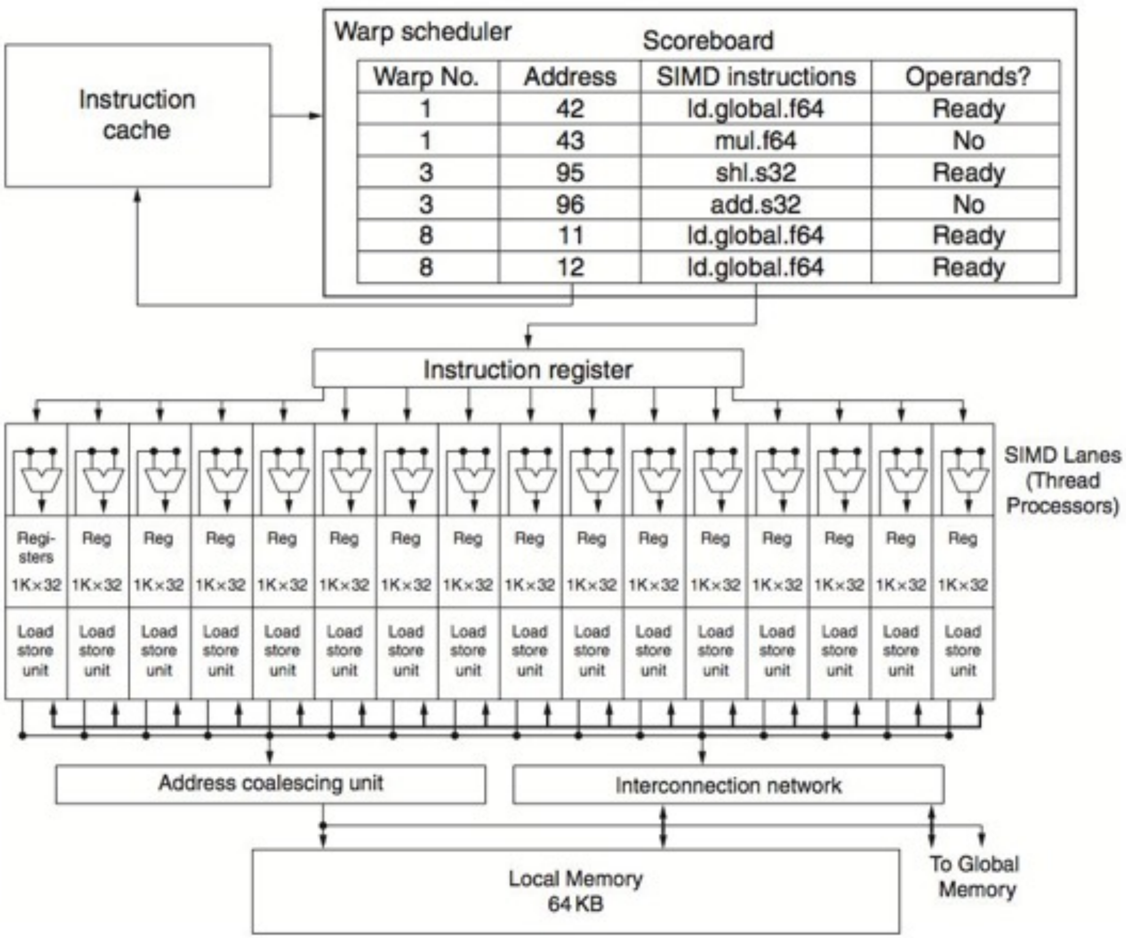
	Horizontal	Vertical
ILP	Superscalar	Pipelining/OoO
TLP	Multi-core	SMT
DLP	SIMD/SIMT/Vector	Temporal SIMT

GPUs focus on TLP, DLP

Key Concepts

- SIMT: Single-instruction multiple-thread
  - Multiple instruction streams of scalar instructions
- Warps: A set of threads executing the same instruction (grouped dynamically by the hardware)
  - Essentially a SIMD operation formed in hardware
- SM: Streaming multi-processor
- Branch divergence: Masking

Streaming Multiprocessor



Example:

» 16 physical lanes

» Tens of warps with 32 threads per warp

» Warp scheduler issues SIMD instruction, when all threads ready

A Snapshot of Challenges

- Warp scheduling
  - Which warp to pick for running?
    Issues: Prioritize intra-warp locality, inter-warp locality, memory coalescing
- Divergence
- Memory access patterns
  - Coalescing: Grouping memory requests from multiple warps
  - Scatter/Gather optimization
- Memory bandwidth, interconnect bandwidth
- Power
- Programming model (and ease of programming)

Many more…

Transactional Memory

- » Parallel programming is hard
  - Keeping track of multiple events happening simultaneously is difficult
- » Data parallelism vs Task parallelism
- » Key shortcoming today: Lack of effective mechanisms for abstraction and composition

Transactional Memory

» Idea: No locks, only shared data
  Idea: Optimistic (speculative) concurrency
  - Execute critical section speculatively
  - Abort on conflicts

“Better to ask for forgiveness, than to ask for permission”

Transactional Programming

```
void deposit(account, amount) {
  lock(account);
    int t = bank.get(account);
    t = t + amount;
    bank.put(account, t);
  unlock(account);
}
```

```
void deposit(account, amount) {
  atomic {
    int t = bank.get(account);
    t = t + amount;
    bank.put(account, t);
  }
}
```

Transactional Memory

» Atomicity (all or nothing)
  - At commit, all memory writes take effect at once
  - On abort, none of the writes appear to take effect

» Isolation
  - No other code can observe writes before commit

» Serializability
  - Transactions seem to commit in a single serial order
  - The exact order is not guaranteed

Transactional Memory: Advantages

1. Ease of use (declarative)
2. Composability
3. Expected performance of fine-grained locking

Composability

```
void transfer(A, B, amount) {
  lock(A) {
  lock(B) {
    withdraw(A, amount);
    deposit(B, amount);
  }
  }
}
```

```
void transfer(B, A, amount) {
  lock(B) {
  lock(A) {
    withdraw(B, amount);
    deposit(A, amount);
  }
  }
}
```

1. Fine grained locking → Can lead to deadlock
2. Need some global locking discipline now

Composability

```
void transfer(A, B, amount) {
  atomic {
    withdraw(A, amount);
    deposit(B, amount);
  }
}
```

```
void transfer(B, A, amount) {
  atomic {
    withdraw(B, amount);
    deposit(A, amount);
  }
}
```

Transactional Memory Taxonomy

» Data Versioning
  - Eager
  - Lazy

» Conflict Detection
  - Pessimistic
  - Optimistic

Data Management Policy

How to manage the "tentative work" that a transaction does

1. Eager versioning (undo-log based)
   - Update memory location directly
   - Maintain undo info in a log
   - Fast commits
   - Slow aborts
2. Lazy versioning (write-buffer based)
   - Buffer data until commit in a write buffer
   - Update actual memory locations at commit
   - Fast aborts
   - Slow commits

Conflict Detection Policy

How to ensure isolation between transactions

1. Pessimistic detection
   Check for conflicts during loads or stores

2. Optimistic detection
   Detect conflicts when a transaction attempts to commit

TM Implementation Space Examples

» Hardware TM systems
  - Lazy + optimistic: Stanford TCC
  - Lazy + pessimistic: Intel VTM
  - Eager + pessimistic: Wisconsin LogTM

» Software TM systems
  - Lazy + optimistic (rd/wr): Sun TL2
  - Lazy + optimistic (rd)/pessimistic (wr): MS OSTM
  - Eager + optimistic (rd)/pessimistic (wr): Intel STM

A Snapshot of Challenges

- When is TM an appropriate programming abstraction?
  - Shared memory data structures that are difficult to scale with traditional locking (or have too complex fine-grained locking solutions)?
- Interactions with non-transactional code, nested transactions
- Hardware trade-offs
  - Memory system, frequency of aborts vs cost, communication overhead etc.
- Deadlock, livelock, memory consistency

Thank You!