Lecture 16: Transactional Memory II

Parallel Computing
Stanford CS149, Fall 2023
Transactional Memory (TM) Review

- Memory transaction
  - An atomic and isolated sequence of memory accesses
  - Inspired by database transactions

- Atomicity (all or nothing)
  - Upon transaction commit, all memory writes in transaction take effect at once
  - On transaction abort, none of the writes appear to take effect (as if transaction never happened)

- Isolation
  - No other processor can observe writes before transaction commits

- Serializability
  - Transactions appear to commit in a single serial order
  - But the exact order of commits is not guaranteed by semantics of transaction
Advantages (promise) of transactional memory

- Easy to use synchronization construct
  - It is difficult for programmers to get synchronization right
  - Programmer declares need for atomicity, system implements it well
  - Claim: transactions are as easy to use as coarse-grain locks

- Often performs as well as fine-grained locks
  - Provides automatic read-read concurrency and fine-grained concurrency
  - Performance portability: locking scheme for four CPUs may not be the best scheme for 64 CPUs
  - Productivity argument for transactional memory: system support for transactions can achieve 90% of the benefit of expert programming with fine-grained locks, with 10% of the development time

- Failure atomicity and recovery
  - No lost locks when a thread fails
  - Failure recovery = transaction abort + restart

- Composability
  - Safe and scalable composition of software modules
Implementing transactional memory
TM implementation basics

- TM systems must provide atomicity and isolation
  - While maintaining concurrency as much as possible

- Two key implementation questions
  - Data versioning policy: How does the system manage uncommitted (new) and previously committed (old) versions of data for concurrent transactions?
  - Conflict detection policy: how/when does the system determine that two concurrent transactions conflict?
**Data Versioning Policy**

Manage uncommitted (new) and previously committed (old) versions of data for concurrent transactions

1. Eager versioning (undo-log based)
2. Lazy versioning (write-buffer based)
Conflict Detection

- **Must detect and handle conflicts between transactions**
  - Read-write conflict: transaction A reads address X, which was written to by pending (but not yet committed) transaction B
  - Write-write conflict: transactions A and B are both pending, and both write to address X

- **System must track a transaction’s read set and write set**
  - Read-set: addresses read during the transaction
  - Write-set: addresses written during the transaction
Pessimistic Detection

- Check for conflicts (immediately) during loads or stores
  - Philosophy: “I suspect conflicts might happen, so let’s always check to see if one has occurred after each memory operation... if I’m going to have to roll back, might as well do it now to avoid wasted work.”

- “Contention manager” decides to stall or abort transaction when a conflict is detected
  - Various policies to handle common case fast
Pessimistic Detection Examples

Note: diagrams assume “aggressive” contention manager on writes: writer wins, so other transactions abort.

Case 1: Success
- T0: rd A check, wr B check, wr C check, commit
- T1: commit

Case 2: Early detect (and stall)
- T0: wr A check, wr A check, commit
- T1: rd A check, stall

Case 3: Abort
- T0: rd A check, wr A check, wr A check, restart
- T1: rd A check, stall (case 2), restart

Case 4: No progress (question: how to avoid livelock?)
- T0: wr A check, wr A check, restart
- T1: wr A check, restart

Stanford CS149, Fall 2023
Optimistic detection

- **Detect conflicts when a transaction attempts to commit**
  - Intuition: “Let’s hope for the best and sort out all the conflicts only when the transaction tries to commit”
- **On a conflict, give priority to committing transaction**
  - Other transactions may abort later on
Optimistic detection

Case 1
- Time:
  - T0: rd A, wr B, wr C
  - T1: commit
- Success

Case 2
- Time:
  - T0: wr A
  - T1: rd A, commit
- Abort

Case 3
- Time:
  - T0: rd A, wr A
  - T1: commit
- Success

Case 4
- Time:
  - T0: rd A, wr A
  - T1: rd A, wr A, restart
- Forward progress
TM implementation space (examples)

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

- **Hardware TM systems**
  - Lazy + optimistic: Stanford TCC
  - Lazy + pessimistic: MIT LTM, Intel VTM
  - **Eager + pessimistic: Wisconsin LogTM** (easiest with conventional cache coherence)

- **Optimal design remains an open question**
  - May be different for HW, SW, and hybrid
Software Transactional Memory

atomic {
    a.x = t1
    a.y = t2
    if (a.z == 0) {
        a.x = 0
        a.z = t3
    }
}

tmTxnBegin()
    tmWr(&a.x, t1)
    tmWr(&a.y, t2)
    if (tmRd(&a.z) != 0) {
        tmWr(&a.x, 0);
        tmWr(&a.z, t3)
    }
    tmTxnCommit()
STM Runtime Data Structures

- **Transaction descriptor (per-thread)**
  - Used for conflict detection, commit, abort, ...
  - Includes the read set, write set, undo log or write buffer

- **Transaction record (per data)**
  - Pointer-sized record guarding shared data
  - Tracks transactional state of data
    - Shared: accessed by multiple readers
      - Using version number or shared reader lock
    - Exclusive: access by one writer
      - Using writer lock that points to owner
    - BTW: same way that HW cache coherence works
Mapping Data to Transaction Records

Every data item has an associated transaction record

Java/C#  
```
class Foo {
    int x;
    int y;
}
```

C/C++  
```
struct Foo {
    int x;
    int y;
}
```

Embed each object

OR

Hash fields or array elements to global table

```
f(obj.hash, field.index)
```

Address-based hash into global table

Cache-line or word granularity

What’s the tradeoff?
Conflict Detection Granularity

- **Object granularity**
  - Low overhead mapping operation
  - Exposes optimization opportunities
  - False conflicts (e.g. Txn 1 and Txn 2)

- **Element/field granularity (word)**
  - Reduces false conflicts
  - Improves concurrency (e.g. Txn 1 and Txn 2)
  - Increased overhead (time/space)

- **Cache line granularity (multiple words)**
  - Matches hardware TM
  - Reduces storage overhead of transactional records
  - Hard for programmer & compiler to analyze

- **Mix & match per type basis**
  - E.g., element-level for arrays, object-level for non-arrays

```
Txn 1
a.x = ...

Txn 2
... = ... a.z ...
```
An Example STM Algorithm

- Based on Intel’s McRT STM [PPoPP’ 06, PLDI’ 06, CGO’ 07]
  - Eager versioning, optimistic reads, pessimistic writes

- Based on timestamp for version tracking
  - Global timestamp
    - Incremented when a writing xaction commits
  - Local timestamp per xaction
    - Global timestamp value when xaction last validated

- Transaction record (32-bit)
  - LS bit: 0 if writer-locked, 1 if not locked
  - MS bits
    - Timestamp (version number) of last commit if not locked
    - Pointer to owner xaction if locked
STM Operations

- **STM read (optimistic)**
  - Direct read of memory location (eager)
  - Validate read data
    - Check if unlocked and data version ≤ local timestamp
    - If not, validate all data in read set for consistency
  - Insert in read set
  - Return value

- **STM write (pessimistic)**
  - Validate data
    - Check if unlocked and data version ≤ local timestamp
  - Acquire lock
  - Insert in write set
  - Create undo log entry
  - Write data in place (eager)
STM Operations (cont)

- **Read-set validation**
  - Get global timestamp
  - For each item in the read set
    - If locked by other or data version > local timestamp, abort
  - Set local timestamp to global timestamp from initial step

- **STM commit**
  - Atomically increment global timestamp by 2 (LSb used for write-lock)
  - If preincremented (old) global timestamp > local timestamp, validate read-set
    - Check for recently committed transactions
  - For each item in the write set
    - Release the lock and set version number to global timestamp
STM Example

- X1 copies object foo into object bar
- X2 should read bar as [0,0] or [9,7]
STM Example

Reads <foo, 3> <foo, 3>

Writes <bar, 5>

Undo <bar.x, 0> <bar.y, 0>

No local or global time stamps
Each object has a time stamp

X1

atomic {
    t = foo.x;
    bar.x = t;
    t = foo.y;
    bar.y = t;
}

Reads <foo, 3> <foo, 3>

 aborts

X2

atomic {
    t1 = bar.x;
    t2 = bar.y;
}

X2 waits

X1 hdr

<table>
<thead>
<tr>
<th></th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>hdr</td>
<td>0</td>
</tr>
<tr>
<td>x</td>
<td>9</td>
</tr>
</tbody>
</table>

X2 hdr

<table>
<thead>
<tr>
<th></th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>hdr</td>
<td>0</td>
</tr>
<tr>
<td>x</td>
<td>0</td>
</tr>
</tbody>
</table>

No local or global time stamps
Each object has a time stamp
TM Implementation Summary 1

- **TM implementation**
  - Data versioning: eager or lazy
  - Conflict detection: optimistic or pessimistic
    - Granularity: object, word, cache-line, ...

- **Software TM systems**
  - Compiler adds code for versioning & conflict detection
    - Note: STM barrier = instrumentation code
  - Basic data-structures
    - Transactional descriptor per thread (status, rd/wr set, ...)
    - Transactional record per data (locked/version)
Challenges for STM Systems

▪ Overhead of software barriers

▪ Function cloning

▪ Robust contention management

▪ Memory model (strong Vs. weak atomicity)
Optimizing Software Transactions

```c
atomic {
    a.x = t1
    a.y = t2
    if (a.z == 0) {
        a.x = 0
        a.z = t3
    }
}

tmTxnBegin()
tmWr(&a.x, t1)
tmWr(&a.y, t2)
if (tmRd(&a.z) != 0) {
    tmWr(&a.x, 0);
    tmWr(&a.z, t3)
}
tmTxnCommit()
```

- Monolithic barriers hide redundant logging & locking from the compiler
Optimizing Software Transactions

```c
atomic {
    a.x = t1
    a.y = t2
    if (a.z == 0) {
        a.x = 0
        a.z = t3
    }
}

 Decomposed barriers expose redundancies
```

```c
txnOpenForWrite(a)
txnLogObjectInt(&a.x, a)
a.x = t1
txnOpenForWrite(a)
txnLogObjectInt(&a.y, a)
a.y = t2
 txnOpenForRead(a)
if(a.z != 0) {
    txnOpenForWrite(a)
txnLogObjectInt(&a.x, a)
a.x = 0
txnOpenForWrite(a)
txnLogObjectInt(&a.z, a)
a.z = t3
}
```
Optimizing Software Transactions

```
atomic {
  a.x = t1
  a.y = t2
  if (a.z == 0) {
    a.x = 0
    a.z = t3
  }
}
```

```
 txnOpenForWrite(a)
 txnLogObjectInt(&a.x, a)
 a.x = t1
 txnLogObjectInt(&a.y, a)
 a.y = t2
 if (a.z != 0) {
   a.x = 0
   txnLogObjectInt(&a.z, a)
   a.z = t3
 }
```

- Allows compiler to optimize STM code
- Produces fewer & cheaper STM operations
Effect of Compiler Optimizations

- 1 thread overheads over thread-unsafe baseline

- With compiler optimizations
  - <40% over no concurrency control
  - <30% over lock-based synchronization
STM Question

- Given an optimistic read, pessimistic write, eager versioning STM
- What steps are required to implement the atomic region

```plaintext
atomic{
    obj.f1=42;
}
```
Motivation for Hardware Support

- STM slowdown: 2-8x per thread overhead due to barriers
- Short term issue: demotivates parallel programming
- Long term issue: energy wasteful
- Lack of strong atomicity
- Costly to provide purely in software
Why is STM Slow?

- Measured single-thread STM performance

1.8x – 5.6x slowdown over sequential

Most time goes in read barriers & commit
  - Most apps read more data than they write
Types of Hardware Support

- Hardware-accelerated STM systems (HASTM, SigTM, USTM, …)
  - Start with an STM system & identify key bottlenecks
  - Provide (simple) HW primitives for acceleration, but keep SW barriers

- Hardware-based TM systems (TCC, LTM, VTM, LogTM, …)
  - Versioning & conflict detection directly in HW
  - No SW barriers

- Hybrid TM systems (Sun Rock, …)
  - Combine an HTM with an STM by switching modes when needed
    - Based on xaction characteristics available resources, …

<table>
<thead>
<tr>
<th></th>
<th>HTM</th>
<th>STM</th>
<th>HW-STM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write versioning</td>
<td>HW</td>
<td>SW</td>
<td>SW</td>
</tr>
<tr>
<td>Conflict detection</td>
<td>HW</td>
<td>SW</td>
<td>HW</td>
</tr>
</tbody>
</table>
Hardware transactional memory (HTM)

- Data versioning is implemented in caches
  - Cache the write buffer or the undo log
  - Add new cache line metadata to track transaction read set and write set

- Conflict detection through cache coherence protocol
  - Coherence lookups detect conflicts between transactions
  - Works with snooping and directory coherence

- Note:
  - Register checkpoint must also be taken at transaction begin (to restore execution context state on abort)
HTM design

- Cache lines annotated to track read set and write set
  - R bit: indicates data read by transaction (set on loads)
  - W bit: indicates data written by transaction (set on stores)
  - R/W bits can be at word or cache-line granularity
  - R/W bits gang-cleared on transaction commit or abort

- For eager versioning, need a 2nd cache write for undo log

- Coherence requests check R/W bits to detect conflicts
  - Observing shared request to W-word is a read-write conflict
  - Observing exclusive (intent to write) request to R-word is a write-read conflict
  - Observing exclusive (intent to write) request to W-word is a write-write conflict
Example HTM implementation: lazy-optimistic

- **CPU changes**
  - Ability to checkpoint register state (available in many CPUs)
  - TM state registers (status, pointers to abort handlers, ...)

![CPU Diagram]

![Cache Diagram]
Example HTM implementation: lazy-optimistic

- **Cache changes**
  - R bit indicates membership to read set
  - W bit indicates membership to write set
HTM transaction execution

- Transaction begin
  - Initialize CPU and cache state
  - Take register checkpoint

Xbegin ←
Load A
Load B
Store C ← 5
Xcommit
HTM transaction execution

- Load operation
  - Serve cache miss if needed
  - Mark data as part of read set

Xbegin
Load A
Load B
Store C ← 5
Xcommit
HTM transaction execution

- Load operation
  - Serve cache miss if needed
  - Mark data as part of read set

Xbegin
Load A
Load B
Store C ← 5
Xcommit
HTM transaction execution

- **Store operation**
  - Service cache miss if needed
  - Mark data as part of write set (note: this is not a load into exclusive state. Why?)
HTM transaction execution: commit

- Fast two-phase commit
  - Validate: request RdX access to write set lines (if needed)
  - Commit: gang-reset R and W bits, turns write set data to valid (dirty) data

Xbegin
Load A
Load B
Store C ← 5
Xcommit

upgrade X C
(result: C is now in dirty state)
HTM transaction execution: detect/abort

Assume remote processor commits transaction with writes to A and D

- Fast conflict detection and abort
  - Check: lookup exclusive requests in the read set and write set
  - Abort: invalidate write set, gang-reset R and W bits, restore to register checkpoint
HTM Performance Example

- 2x to 7x over STM performance
- Within 10% of sequential for one thread
- Scales efficiently with number of processors
Review: Transactional Memory

- **Atomic construct**: declaration that atomic behavior must be preserved by the system
  - Motivating idea: increase simplicity of synchronization without (significantly) sacrificing performance

- **Transactional memory implementation**
  - Many variants have been proposed: SW, HW, SW+HW
  - Implementations differ in:
    - Data versioning policy (eager vs. lazy)
    - Conflict detection policy (pessimistic vs. optimistic)
    - Detection granularity (object, word, cache line)

- **Software TM systems (STM)**
  - Compiler adds code for versioning & conflict detection
    - Note: STM barrier = instrumentation code (e.g. StmRead, StmWrite)
  - Basic data-structures
    - Transactional descriptor per thread (status, rd/wr set, …)
    - Transactional record per data (locked/version)

- **Hardware Transactional Memory (HTM)**
  - Versioned data is kept in caches
  - Conflict detection mechanisms augment coherence protocol
HTM Example: Transactional Coherence and Consistency

- Use TM as the coherence mechanism → all transactions all the time
- Successful transaction commits update memory and all caches in the system

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin T1</td>
<td>Begin T2</td>
<td>Begin T4</td>
</tr>
<tr>
<td>Read A</td>
<td>Read A</td>
<td>Read E</td>
</tr>
<tr>
<td>Write A, 1</td>
<td>Write E, 3</td>
<td>Write B, 6</td>
</tr>
<tr>
<td>Write C, 2</td>
<td>Commit T2</td>
<td>Write C, 7</td>
</tr>
<tr>
<td>Read D</td>
<td>Begin T3</td>
<td>Read F</td>
</tr>
<tr>
<td>Commit T1</td>
<td>Write C, 4</td>
<td>Commit T4</td>
</tr>
<tr>
<td></td>
<td>Read A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write E, 5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commit T3</td>
<td></td>
</tr>
</tbody>
</table>

- Assumptions
  - Lazy and optimistic
  - One “commit” per execution step across all processors
  - When one transaction causes another transaction to abort and re-execute, assume that the transaction “commit” of one transaction can overlap with the “begin” of the re-executing transaction
  - Minimize the number of execution steps
## HTM Example: Transactional Coherence and Consistency

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Begin T1</strong></td>
<td><strong>Begin T2</strong></td>
<td><strong>Begin T4</strong></td>
</tr>
<tr>
<td>Read A</td>
<td>Read A</td>
<td>Read E</td>
</tr>
<tr>
<td>Write A, 1</td>
<td>Write E, 3</td>
<td>Write B, 6</td>
</tr>
<tr>
<td>Commit T1</td>
<td>Commit T2</td>
<td>Write C, 7</td>
</tr>
<tr>
<td>Read D</td>
<td>Begin T3</td>
<td>Read F</td>
</tr>
<tr>
<td>Write C, 2</td>
<td>Write C, 4</td>
<td>Commit T4</td>
</tr>
<tr>
<td>Read A</td>
<td>Read A</td>
<td></td>
</tr>
<tr>
<td>Write E, 5</td>
<td>Write E</td>
<td></td>
</tr>
<tr>
<td>Commit T3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Action</strong></td>
<td><strong>Read set</strong></td>
<td><strong>Write set</strong></td>
</tr>
<tr>
<td>B T1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R A</td>
<td>A:0</td>
<td></td>
</tr>
<tr>
<td>W A, 1</td>
<td>A:0</td>
<td>A:1</td>
</tr>
<tr>
<td>W C, 2</td>
<td>A:0</td>
<td>A:1,C:2</td>
</tr>
</tbody>
</table>
HTM Example: Transactional Coherence and Consistency

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Begin T1</strong></td>
<td><strong>Begin T2</strong></td>
<td><strong>Begin T4</strong></td>
</tr>
<tr>
<td>Read A</td>
<td>Read A</td>
<td>Read E</td>
</tr>
<tr>
<td>Write A, 1</td>
<td>Write E, 3</td>
<td>Write B, 6</td>
</tr>
<tr>
<td>Write C, 2</td>
<td>Commit T2</td>
<td>Write C, 7</td>
</tr>
<tr>
<td>Read D</td>
<td>Begin T3</td>
<td>Read F</td>
</tr>
<tr>
<td>Commit T1</td>
<td>Write C, 4</td>
<td>Commit T4</td>
</tr>
<tr>
<td></td>
<td>Read A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write E, 5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commit T3</td>
<td></td>
</tr>
</tbody>
</table>

**Action**

<table>
<thead>
<tr>
<th>B T1</th>
<th>B T2</th>
<th>B T4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R A</strong></td>
<td><strong>R A</strong></td>
<td><strong>R E</strong></td>
</tr>
<tr>
<td>A:0</td>
<td>A:0</td>
<td>E:0</td>
</tr>
<tr>
<td>W A, 1</td>
<td>W E</td>
<td>W B, 6</td>
</tr>
<tr>
<td>A:0</td>
<td>A:0</td>
<td>E:0</td>
</tr>
<tr>
<td>W C, 2</td>
<td>C T2</td>
<td>B T4</td>
</tr>
<tr>
<td>A:0</td>
<td>A:0</td>
<td>E:3</td>
</tr>
<tr>
<td>R D</td>
<td>B T3</td>
<td>R E</td>
</tr>
<tr>
<td>A:0,D:0</td>
<td>A:1,C:2</td>
<td>E:3</td>
</tr>
<tr>
<td>C T1</td>
<td>W C, 5</td>
<td>W B, 6</td>
</tr>
<tr>
<td>A:0,D:0</td>
<td>A:1,C:2</td>
<td>E:3</td>
</tr>
<tr>
<td>WB, 6</td>
<td>C:5</td>
<td>B:6</td>
</tr>
</tbody>
</table>

Stanford CS149, Fall 2023
### HTM Example: Transactional Coherence and Consistency

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Begin T1</strong></td>
<td><strong>Begin T2</strong></td>
<td><strong>Begin T4</strong></td>
</tr>
<tr>
<td>Read A</td>
<td>Read A</td>
<td>Read E</td>
</tr>
<tr>
<td>Write A, 1</td>
<td>Write E, 3</td>
<td>Write B, 6</td>
</tr>
<tr>
<td>Read D</td>
<td>Commit T2</td>
<td>Write C, 7</td>
</tr>
<tr>
<td>Commit T1</td>
<td>Begin T3</td>
<td>Read F</td>
</tr>
<tr>
<td></td>
<td>Write C, 4</td>
<td>Commit T4</td>
</tr>
<tr>
<td></td>
<td>Read A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write E, 5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Action</strong></th>
<th><strong>Read set</strong></th>
<th><strong>Write set</strong></th>
<th><strong>Action</strong></th>
<th><strong>Read set</strong></th>
<th><strong>Write set</strong></th>
<th><strong>Action</strong></th>
<th><strong>Read set</strong></th>
<th><strong>Write set</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>B T1</td>
<td></td>
<td></td>
<td>B T2</td>
<td></td>
<td></td>
<td>B T4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R A</td>
<td>A:0</td>
<td></td>
<td>R A</td>
<td>A:0</td>
<td></td>
<td>R E</td>
<td>E:0</td>
<td></td>
</tr>
<tr>
<td>W A, 1</td>
<td>A:0</td>
<td>A:1</td>
<td>W E</td>
<td>A:0</td>
<td>E:3</td>
<td>W B, 6</td>
<td>E:0</td>
<td>B:6</td>
</tr>
<tr>
<td>W C, 2</td>
<td>A:0</td>
<td>A:1,C:2</td>
<td>C T2</td>
<td>A:0</td>
<td>E:3</td>
<td>B T4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R D</td>
<td>A:0,D:0</td>
<td>A:1,C:2</td>
<td>B T3</td>
<td></td>
<td></td>
<td>R E</td>
<td>E:3</td>
<td></td>
</tr>
<tr>
<td>C T1</td>
<td>A:0,D:0</td>
<td>A:1,C:2</td>
<td>W C, 5</td>
<td>C:4</td>
<td>W B, 6</td>
<td>E:3</td>
<td>B:6</td>
<td></td>
</tr>
<tr>
<td>R A</td>
<td>A:1</td>
<td>C:5</td>
<td>W C, 7</td>
<td>E:3</td>
<td>B:6,C:7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W E, 6</td>
<td>A:1</td>
<td>C:5,E:6</td>
<td>R F</td>
<td>E:3,F:0</td>
<td>B:6,C:7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A:1</td>
<td>C:5,E:6</td>
<td>C T4</td>
<td>E:3,F:0</td>
<td>B:6,C:7</td>
<td></td>
</tr>
</tbody>
</table>
### HTM Example: Transactional Coherence and Consistency

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>Read set</td>
<td>Write set</td>
<td>Action</td>
</tr>
<tr>
<td>B T1</td>
<td>(A:0)</td>
<td>(A:0)</td>
<td>B T4</td>
</tr>
<tr>
<td>R A</td>
<td>(A:0)</td>
<td>(A:0)</td>
<td>R E</td>
</tr>
<tr>
<td>W A, 1</td>
<td>(A:0), (A:1)</td>
<td>(W E)</td>
<td>(A:0)</td>
</tr>
<tr>
<td>W C, 2</td>
<td>(A:0)</td>
<td>(A:1,C:2)</td>
<td>C T2</td>
</tr>
<tr>
<td>R D</td>
<td>(A:0,D:0), (A:1,C:2)</td>
<td>(B T3)</td>
<td>(R E)</td>
</tr>
<tr>
<td>C T1</td>
<td>(A:0,D:0), (A:1,C:2)</td>
<td>(W C, 5)</td>
<td>(C:5)</td>
</tr>
<tr>
<td>R A</td>
<td>(A:1)</td>
<td>(C:5)</td>
<td>W C, 7</td>
</tr>
<tr>
<td>W E, 6</td>
<td>(A:1)</td>
<td>(C:5,E:6)</td>
<td>R F</td>
</tr>
<tr>
<td>A:1</td>
<td>(C:5,E:6)</td>
<td>C T4</td>
<td>(E:3,F:0)</td>
</tr>
<tr>
<td>C T3</td>
<td>(A:1)</td>
<td>(C:5,E:6)</td>
<td></td>
</tr>
</tbody>
</table>
Hardware transactional memory support in Intel Haswell architecture

- New instructions for “restricted transactional memory” (RTM)
  - xbegin: takes pointer to “fallback address” in case of abort
    - e.g., fallback to code-path with a spin-lock
  - xend
  - xabort

- Implementation: tracks read and write set in L1 cache

- Processor makes sure all memory operations commit atomically
  - But processor may automatically abort transaction for many reasons (e.g., eviction of line in read or write set will cause a transaction abort)
    - Implementation does not guarantee progress (see fallback address)
  - Intel optimization guide (ch 12) gives guidelines for increasing probability that transactions will not abort
Summary: transactional memory

- **Atomic construct**: declaration that atomic behavior must be preserved by the system
  - Motivating idea: increase simplicity of synchronization without (significantly) sacrificing performance

- **Transactional memory implementation**
  - Many variants have been proposed: SW, HW, SW+HW
  - Implementations differ in:
    - Versioning policy (eager vs. lazy)
    - Conflict detection policy (pessimistic vs. optimistic)
    - Detection granularity (object, word, cache line)

- **Software TM systems**
  - Compiler adds code for versioning & conflict detection
    - Note: STM barrier = instrumentation code
  - Basic data-structures
    - Transactional descriptor per thread (status, rd/wr set, …)
    - Transactional record per data (locked/version)

- **Hardware transactional memory**
  - Versioned data is kept in caches
  - Conflict detection mechanisms built upon coherence protocol
Lecture 16+:
Heterogeneous Parallelism and Hardware Specialization

Parallel Computing
Stanford CS149, Fall 2023
I want to begin this lecture by reminding you…

In assignment 1 we observed that a well-optimized parallel implementation of a compute-bound application is about 40 times faster on my quad-core laptop than the output of single-threaded C code compiled with gcc -O3.

(In other words, a lot of software makes inefficient use of modern CPUs.)

Today we’re going to talk about how inefficient the CPU in that laptop is, even if you are using it as efficiently as possible.
Heterogeneous processing

Observation: most “real world” applications have complex workload characteristics

They have components that can be widely parallelized. And components that are difficult to parallelize.

They have components that are amenable to wide SIMD execution. And components that are not. (divergent control flow)

They have components with predictable data access And components with unpredictable access, but those accesses might cache well.

Idea: the most efficient processor is a heterogeneous mixture of resources (“use the most efficient tool for the job”)
Examples of heterogeneity
Example: Intel “Skylake” (2015)
(6th Generation Core i7 architecture)

4 CPU cores + graphics cores + media accelerators
Example: Intel “Skylake” (2015)
(6th Generation Core i7 architecture)

- CPU cores and graphics cores share same memory system
- Also share LLC (L3 cache)
  - Enables, low-latency, high-bandwidth communication between CPU and integrated GPU
- Graphics cores are cache coherent with CPU cores
More heterogeneity: add discrete GPU

Keep discrete (power hungry) GPU unless needed for graphics-intensive applications
Use integrated, low power graphics for basic graphics/window manager/UI

- High-end discrete GPU (AMD or NVIDIA)
- PCIe x16 bus
- DDR5 Memory
- L3 cache (8 MB)
- Memory controller
- DDR3 Memory
- CPU Core 0
- CPU Core 3
- Gen9 Graphics
- Ring interconnect
Mobile heterogeneous processors

NVIDIA Tegra X1
- Four ARM Cortex A57 CPU cores for applications
- Four low performance (low power) ARM A53 CPU cores
- One Maxwell SMM (256 “CUDA” cores)

Apple A11 Bionic *
- Two “high performance” 64 bit ARM CPU cores
- Four “low performance” ARM CPU cores
- Three “core” Apple-designed GPU
- Image processor
- Neural Engine for DNN acceleration
- Motion processor

* Disclaimer: estimates by TechInsights, not an official Apple reference.

A11 image credit: TechInsights Inc.
GPU-accelerated Supercomputing

Frontier at Oak Ridge National Lab (world’s #1 in Fall 2022)
9,472 AMD 64 core 2 GHz CPUs (606,208 cores)
37,888 Radeon Instinct MI250X GPUs
10 Petabytes DRAM
Power 21 MW
Cost $600M
Energy-constrained computing
Energy (Power x Time)-constrained computing

- Supercomputers are energy constrained
  - Due to shear scale of machine
  - Overall cost to operate (power for machine and for cooling)

- Datacenters are energy constrained
  - Reduce cost of cooling
  - Reduce physical space requirements

- Mobile devices are energy constrained
  - Limited battery life
  - Heat dissipation
Performance and Power

\[ \text{Power} = \frac{\text{Ops}}{\text{second}} \times \frac{\text{Joules}}{\text{Op}} \]

FIXED

Specialization (fixed function) ⇒ better energy efficiency

What is the magnitude of improvement from specialization?
Pursuing highly efficient processing... 
(specializing hardware beyond just parallel CPUs and GPUs)
Efficiency benefits of compute specialization

- Rules of thumb: compared to high-quality C code on CPU...
- Throughput-maximized processor architectures: e.g., GPU cores
  - Approximately 10x improvement in perf/watt
  - Assuming code maps well to wide data-parallel execution and is compute bound

- Fixed-function ASIC (“application-specific integrated circuit”)
  - Can approach 100-1000x or greater improvement in perf/watt
  - Assuming code is compute bound and is not floating-point math

[Source: Chung et al. 2010, Dally 08]
Why is a “general-purpose processor” so inefficient?

Wait... this entire class we've been talking about making efficient use out of multi-core CPUs and GPUs... and now you're telling me these platforms are “inefficient”?
Consider the complexity of executing an instruction on a modern processor...

Read instruction
Decode instruction
Check for dependencies/pipeline hazards
Identify available execution resource
Use decoded operands to control register file SRAM (retrieve data)
Move data from register file to selected execution resource
Perform arithmetic operation
Move data from execution resource to register file
Use decoded operands to control write to register file SRAM

Review question:
How does SIMD execution reduce overhead of certain types of computations?
What properties must these computations have?
Contrast that complexity to the circuit required to actually perform the operation

Example: 8-bit logical OR
H.264 video encoding: fraction of energy consumed by functional units is small (even when using SIMD)

Even after encoding implemented with SIMD instruction

[Hameed et al. ISCA 2010]
Fast Fourier transform (FFT): throughput and energy benefits of specialization

ASIC delivers same performance as one CPU core with ~ 1/1000th the chip area.

GPU cores: ~ 5-7 times more area efficient than CPU cores.

ASIC delivers same performance as one CPU core using only ~ 1/100th the power
Mobile: benefits of increasing efficiency

- Run faster for a fixed period of time
  - Run at higher clock, use more cores (reduce latency of critical task)
  - Do more at once

- Run at a fixed level of performance for longer
  - e.g., video playback, health apps
  - Achieve “always-on” functionality that was previously impossible

iPhone: Siri activated by button press or holding phone up to ear

Amazon Echo / Google Home: Always listening

Google Glass: ~40 min recording per charge (nowhere near “always on”)

Example: Intel “Skylake” (2015)
(6th Generation Core i7 architecture)

- CPU cores and graphics cores share same memory system

- Also share LLC (L3 cache)
  - Enables, low-latency, high-bandwidth communication between CPU and integrated GPU

- Graphics cores are cache coherent with CPU cores
GPU’s are themselves heterogeneous multi-core processors

Compute resources your CUDA programs used in Assignment 2

Graphics-specific, fixed-function compute resources

GPU Memory

GPU

Stanford CS149, Fall 2023
Example graphics tasks performed in fixed-function HW

**Rasterization:**
Determining what pixels a triangle overlaps

**Texture mapping:**
Warping/filtering images to apply detail to surfaces

**Geometric tessellation:**
Computing fine-scale geometry from coarse geometry
Digital signal processors (DSPs)

Programmable processors, but simpler instruction stream control paths
Complex instructions (e.g., SIMD/VLIW): perform many operations per instruction (amortize cost of control)

Example: Qualcomm Hexagon DSP
Used for modem, audio, and (increasingly) image processing on Qualcomm Snapdragon SoC processors

VLIW: “very-long instruction word”
Single instruction specifies multiple different operations to do at once (contrast to SIMD)

Below: innermost loop of FFT
Hexagon DSP performs 29 “RISC” ops per cycle

64-bit Load and 64-bit Store with post-update addressing

Zero-overhead loops
- Dec count
- Compare
- Jump top

Vector 4x16-bit Add

Complex multiply with round and saturation

Hexagon DSP is in Google Pixel phone
Anton supercomputer for molecular dynamics

- Simulates time evolution of proteins
- ASIC for computing particle-particle interactions (512 of them in machine)
- Throughput-oriented subsystem for efficient fast-fourier transforms
- Custom, low-latency communication network designed for communication patterns of N-body simulations

[Developed by DE Shaw Research]
Specialized processors for evaluating deep networks

Countless recent papers at top computer architecture research conferences on the topic of ASICs or accelerators for deep learning or evaluating deep networks...

- EIE: Efficient Inference Engine on Compressed Deep Neural Network, Han et al. ISCA 2016
- Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators, Reagen et al. ISCA 2016
- vHNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design, Rhu et al. MICRO 2016
- Fusion-Layer CNN Architectures, Almans et al. MICRO 2016
- DNNWEAVER: From High-Level Deep Network Models to FPGA Acceleration, Sharma et al. MICRO 2016

Example: Google’s Tensor Processing Unit (TPU) Accelerates deep learning operations.
Example: Google’s Pixel Visual Core

Programmable “image processing unit” (IPU)

- Each core = 16x16 grid of 16 bit multiply-add ALUs
- ~10-20x more efficient than GPU at image processing tasks (Google’s claims at HotChips ’18)
Let’s crack open a modern smartphone

Google Pixel 2 Phone: Qualcomm Snapdragon 835 SoC + Google Visual Pixel Core

Visual Pixel Core
Programmable image processor and DNN accelerator

“Hexagon” Programmable DSP
data-parallel multi-media processing

Image Signal Processor
ASIC for processing camera sensor pixels

Multi-core GPU
(3D graphics, OpenCL data-parallel compute)

Video encode/decode ASIC

Display engine
(compresses pixels for transfer to high-res screen)

Multi-core ARM CPU
4 “big cores” + 4 “little cores”
FPGAs (Field Programmable Gate Arrays)

- Middle ground between an ASIC and a processor
- FPGA chip provides array of logic blocks, connected by interconnect
- Programmer-defined logic implemented directly by FPGA

![FPGA Chip Diagram](Image credit: Bai et al. 2014)
Modern FPGAs

- A lot of area devoted to hard gates
  - Memory blocks (SRAM)
  - DSP blocks (multiplier)
Specifying combinatorial logic as a LUT

- Example: 6-input, 1 output LUT in Xilinx Virtex-7 FPGAs
  - Think of a LUT6 as a 64 element table

Example: 6-input AND

<table>
<thead>
<tr>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>1</td>
</tr>
</tbody>
</table>

40-input AND constructed by chaining outputs of eight LUT6’s (delay = 3)

Image credit: [Zia 2013]
Project Catapult

- Microsoft Research investigation of use of FPGAs to accelerate datacenter workloads
- Demonstrated offload of part of Bing search’s document ranking logic
Amazon F1

- FPGA’s are now available on Amazon cloud services

What’s Inside the F1 FPGA?

- **System Logic Block:** Each FPGA in F1 provides over 2M of these logic blocks.
- **DSP (Math) Block:** Each FPGA in F1 has more than 5000 of these blocks.
- **I/O Blocks:** Used to communicate externally, for example to DDR-4, PCIe, or ring.
- **Block RAM:** Each FPGA in F1 has over 60Mb of internal Block RAM, and over 230Mb of embedded UltraRAM.
Summary: choosing the right tool for the job

Energy-optimized CPU

Throughput-oriented processor (GPU)

Programmable DSP

FPGA/reconfigurable logic

ASIC

Video encode/decode,
Audio playback,
Camera RAW processing,
networks (future?)

~10X more efficient

~100-1000X more efficient

Easiest to program

Difficult to program (making it easier is active area of research)

Not programmable + costs 10-100's millions of dollars to design / verify / create

Credit: Pat Hanrahan for this slide design
Challenges of heterogeneous designs:

(it’s not easy to realize the potential of specialized, heterogeneous processing)
Challenges of heterogeneity

- Heterogeneous system: preferred processor for each task
- Challenge to software developer: how to map application onto a heterogeneous collection of resources?
  - Challenge: “Pick the right tool for the job”: design algorithms that decompose into components that each map well to different processing components of the machine
  - The scheduling problem is more complex on a heterogeneous system
- Challenge for hardware designer: what is the right mixture of resources?
  - Too few throughput oriented resources (lower peak throughput for parallel workloads)
  - Too few sequential processing resources (limited by sequential part of workload)
  - How much chip area should be dedicated to a specific function, like video?
Reducing energy consumption idea 1: use specialized processing (use the right processor for the job)

Reducing energy consumption idea 2: move less data
Data movement has high energy cost

- Rule of thumb in mobile system design: always seek to reduce amount of data transferred from memory
  - Earlier in class we discussed minimizing communication to reduce stalls (poor performance). Now, we wish to reduce communication to reduce energy consumption

- “Ballpark” numbers [Sources: Bill Dally (NVIDIA), Tom Olson (ARM)]
  - Integer op: ~ 1 pJ *
  - Floating point op: ~20 pJ *
  - Reading 64 bits from small local SRAM (1mm away on chip): ~ 26 pJ
  - Reading 64 bits from low power mobile DRAM (LPDDR): ~1200 pJ

- Implications
  - Reading 10 GB/sec from memory: ~1.6 watts
  - Entire power budget for mobile GPU: ~1 watt (remember phone is also running CPU, display, radios, etc.)
  - iPhone 6 battery: ~7 watt-hours (note: my Macbook Pro laptop: 99 watt-hour battery)
  - Exploiting locality matters!!!

* Cost to just perform the logical operation, not counting overhead of instruction decode, load data from registers, etc.
Moving data is costly!

Data movement limits performance
Many processing elements...
= higher overall rate of memory requests
= need for more memory bandwidth
(result: bandwidth-limited execution)

Data movement has high energy cost
~ 0.9 pJ for a 32-bit floating-point math op *
~ 5 pJ for a local SRAM (on chip) data access
~ 640 pJ to load 32 bits from LPDDR memory

* Source: [Han, ICLR 2016], 45 nm CMOS assumption
Accessing DRAM
(a basic tutorial on how DRAM works)
The memory system

- DRAM
- 64 bit memory bus
- Memory Controller
  - sends commands to DRAM
- Last-level cache (LLC)
  - issues memory requests to memory controller
- Core
  - issues loads and store instructions
- CPU
DRAM array

1 transistor + capacitor per “bit”  
(Recall: a capacitor stores charge)
DRAM operation (load one byte)

We want to read this byte

1. Precharge: ready bit lines (~10 ns)
2. Row activation (~10 ns)
3. Column selection
4. Transfer data onto bus (~10 ns)

(transfer row)

Data pins (8 bits)
Row buffer (2 Kbits)
2 Kbits per row
DRAM array

Estimated latencies are in units of memory clocks: DDR3-1600 (Kayvon’s laptop)
Load next byte from (already active) row

Lower latency operation: can skip precharge and row activation steps

2 Kbits per row

~ 10 ns

1. Column selection
2. Transfer data onto bus

Data pins (8 bits)

(to memory controller...)

Row buffer (2 Kbits)
DRAM access latency is not fixed

- Best case latency: read from active row
  - Column access time (CAS)

- Worst case latency: bit lines not ready, read from new row
  - Precharge (PRE) + row activate (RAS) + column access (CAS)

  Precharge readies bit lines and writes row buffer contents back into DRAM array (read was destructive)

- Question 1: when to execute precharge?
  - After each column access?
  - Only when new row is accessed?

- Question 2: how to handle latency of DRAM access?
Problem: low pin utilization due to latency of access

Data pins in use only a small fraction of time
(red = data pins busy)

This is bad since they are the scarcest resource!
DRAM burst mode

Idea: amortize latency over larger transfers

Each DRAM command describes bulk transfer
Bits placed on output pins in consecutive clocks

Data pins (8 bits)
DRAM chip consists of multiple banks

- All banks share same pins (only one transfer at a time)
- Banks allow for pipelining of memory requests
  - Precharge/activate rows/send column address to one bank while transferring data from another
  - Achieves high data pin utilization

![Diagram of DRAM chip with multiple banks and signal timing](image)
Organize multiple chips into a DIMM

Example: Eight DRAM chips (64-bit memory bus)
Note: DIMM appears as a single, higher capacity, wider interface DRAM module to the memory controller. Higher aggregate bandwidth, but minimum transfer granularity is now 64 bits.
Reading one 64-byte (512 bit) cache line (the wrong way)

Assume: consecutive physical addresses mapped to same row of same chip
Memory controller converts physical address to DRAM bank, row, column

![Diagram showing the process of reading one 64-byte cache line]
Reading one 64-byte (512 bit) cache line (the wrong way)

- All data for cache line serviced by the same chip
- Bytes sent consecutively over same pins

```
Memory controller
```

```
Last-level cache (LLC)
```

```
CPU
```

```
Request line /w physical address X
```

```
Read bank B, row R, column 0
```

```
64 bit memory bus
```

```
bits 8:15
```

Reading one 64-byte (512 bit) cache line (the wrong way)

All data for cache line serviced by the same chip
Bytes sent consecutively over same pins
Reading one 64-byte (512 bit) cache line

Memory controller converts physical address to DRAM bank, row, column
Here: physical addresses are interleaved across DRAM chips at byte granularity
DRAM chips transmit first 64 bits in parallel
Reading one 64-byte (512 bit) cache line

DRAM controller requests data from new column *
DRAM chips transmit next 64 bits in parallel

* Recall modern DRAM’s support burst mode transfer of multiple consecutive columns, which would be used here
Memory controller is a memory request scheduler

- Receives load/store requests from LLC
- Conflicting scheduling goals
  - Maximize throughput, minimize latency, minimize energy consumption
  - Common scheduling policy: FR-FCFS (first-ready, first-come-first-serve)
    - Service requests to currently open row first (maximize row locality)
    - Service requests to other rows in FIFO order
  - Controller may coalesce multiple small requests into large contiguous requests (to take advantage of DRAM “burst modes”)

![Diagram of memory controller with request queues for banks 0, 1, 2, 3]
Dual-channel memory system

- Increase throughput by adding memory channels (effectively widen bus)
- Below: each channel can issue independent commands
  - Different row/column is read in each channel
  - Simpler setup: use single controller to drive same command to multiple channels
Example: DDR4 memory

Processor: Intel® Core™ i7-7700K Processor (in Myth cluster)

**DDR4 2400**
- 64-bit memory bus x 1.2GHz x 2 transfers per clock* = 19.2GB/s per channel
- 2 channels = 38.4 GB/sec
- ~13 nanosecond CAS

**Memory system details from Intel’s site:**

<table>
<thead>
<tr>
<th>Memory Specifications</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Memory Size (dependent on memory type)</td>
<td>64 GB</td>
</tr>
<tr>
<td>Memory Types</td>
<td>DDR4-2133/2400, DDR3L-1333/1600 @ 1.35V</td>
</tr>
<tr>
<td>Max # of Memory Channels</td>
<td>2</td>
</tr>
<tr>
<td>ECC Memory Supported †</td>
<td>No</td>
</tr>
</tbody>
</table>

* DDR stands for “double data rate”

DRAM summary

- DRAM access latency can depend on many low-level factors
  - Discussed today:
    - State of DRAM chip: row hit/miss? is recharge necessary?
    - Buffering/reordering of requests in memory controller

- Significant amount of complexity in a modern multi-core processor has moved into the design of memory controller
  - Responsible for scheduling ten’s to hundreds of outstanding memory requests
  - Responsible for mapping physical addresses to the geometry of DRAMs
  - Area of active computer architecture research
Modern architecture challenge: improving memory performance:

Decrease distance data must move by locating memory closer to processors

(Enables shorter, but wider interfaces)
Increase bandwidth, reduce power by chip stacking

Enabling technology: 3D stacking of DRAM chips
- DRAMs connected via through-silicon-vias (TSVs) that run through the chips
- TSVs provide highly parallel connection between logic layer and DRAMs
- Base layer of stack “logic layer” is memory controller, manages requests from processor
- Silicon “interposer” serves as high-bandwidth interconnect between DRAM stack and processor

Technologies:
Micron/Intel Hybrid Memory Cube (HBC)
High-bandwidth memory (HBM) - 1024 bit interface to stack

Image credit: AMD
### HBM Advantages

- More Bandwidth
- High Power Efficiency
- Small Form Factor

<table>
<thead>
<tr>
<th></th>
<th>DDR4</th>
<th>LPDDR4(X)</th>
<th>GDDR6</th>
<th>HBM2</th>
<th>HBM2E (JEDEC)</th>
<th>HBM3 (TBD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data rate</td>
<td>3200Mbps</td>
<td>3200Mbps (up to 4266 Mbps)</td>
<td>14Gbps (up to 16Gbps)</td>
<td>2.4Gbps</td>
<td>2.8Gbps</td>
<td>&gt;3.2Gbps (TBD)</td>
</tr>
<tr>
<td>Pin count</td>
<td>x4/x8/x16</td>
<td>x16/ch (2ch per die)</td>
<td>x16/x32</td>
<td>x1024</td>
<td>x1024</td>
<td>x1024</td>
</tr>
<tr>
<td>Bandwidth</td>
<td>5.4GB/s</td>
<td>12.8(17)GB/s</td>
<td>56GB/s</td>
<td>307GB/s</td>
<td>358GB/s</td>
<td>&gt;500GB/s</td>
</tr>
<tr>
<td>Density (per package)</td>
<td>4Gb/8Gb</td>
<td>8Gb/16Gb/24Gb/32Gb</td>
<td>8Gb/16Gb</td>
<td>4GB/8GB</td>
<td>8GB/16GB</td>
<td>8GB/16GB/24GB (TBD)</td>
</tr>
</tbody>
</table>
GPUs are adopting HBM technologies

**AMD Radeon Fury GPU (2015)**
4096-bit interface: 4 HBM chips x 1024 bit interface per chip
512 GB/sec BW

**NVIDIA P100 GPU (2016)**
4096-bit interface: 4 HBM2 chips x 1024 bit interface per chip
720 GB/sec peak BW
4 x 4 GB = 16 GB capacity

**NVIDIA H100 GPU (2022)**
6144-bit interface: 6 HBM3 stacks x 1024 bit interface per stack
3.2 TB/sec peak BW
80 GB capacity
Xeon Phi (Knights Landing) MCDRAM

- 16 GB in package stacked DRAM
- Can be treated as a 16 GB last level cache
- Or as a 16 GB separate address space ("flat mode")
- Intel’s claims:
  - ~ same latency at DDR4
  - ~5x bandwidth of DDR4
  - ~5x less energy cost per bit transferred

```c
float* foo = hbw_malloc(sizeof(float) * 1024);
```
Summary: the memory bottleneck is being addressed in many ways

- By the application programmer
  - Schedule computation to maximize locality (minimize required data movement)

- By new hardware architectures
  - Intelligent DRAM request scheduling
  - Bringing data closer to processor (deep cache hierarchies, 3D stacking)
  - Increase bandwidth (wider memory systems)
  - Ongoing research in locating limited forms of computation “in” or near memory
  - Ongoing research in hardware accelerated compression (not discussed today)

- General principles
  - Locate data storage near processor
  - Move computation to data storage
  - Data compression (trade-off extra computation for less data transfer)
Three trends in energy-optimized computing

- **Compute less!**
  - Computing costs energy: parallel algorithms that do more work than sequential counterparts may not be desirable even if they run faster

- **Specialize compute units:**
  - Heterogeneous processors: CPU-like cores + throughput-optimized cores (GPU-like cores)
  - Fixed-function units: audio processing, “movement sensor processing” video decode/encode, image processing/computer vision?
  - Specialized instructions: expanding set of AVX vector instructions, new instructions for accelerating AES encryption (AES-NI)
  - Programmable soft logic: FPGAs

- **Reduce bandwidth requirements**
  - Exploit locality (restructure algorithms to reuse on-chip data as much as possible)
  - Aggressive use of compression: perform extra computation to compress application data before transferring to memory (likely to see fixed-function HW to reduce overhead of general data compression/decompression)
Summary: heterogeneous processing for efficiency

- Heterogeneous parallel processing: use a mixture of computing resources that fit mixture of needs of target applications
  - Latency-optimized sequential cores, throughput-optimized parallel cores, domain-specialized fixed-function processors
  - Examples exist throughout modern computing: mobile processors, servers, supercomputers

- Traditional rule of thumb in "good system design" is to design simple, general-purpose components
  - This is not the case in emerging systems (optimized for perf/watt)
  - Today: want collection of components that meet perf requirement AND minimize energy use

- Challenge of using these resources effectively is pushed up to the programmer
  - Current CS research challenge: how to write efficient, portable programs for emerging heterogeneous architectures?