Lecture 14:

Transactional Memory

Parallel Computing
Stanford CS149, Fall 2022

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 fined-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

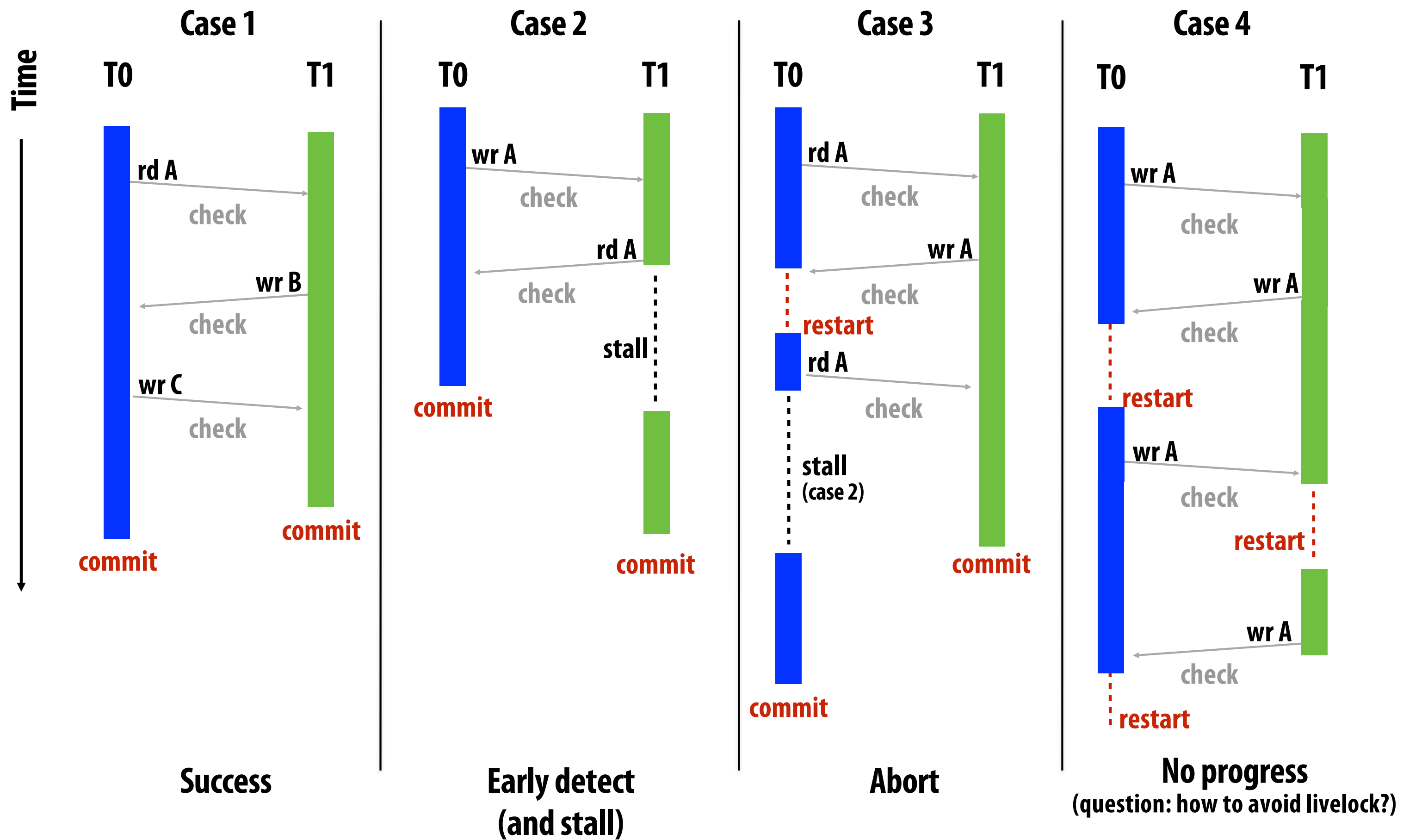
Pessimistic Detection

- 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

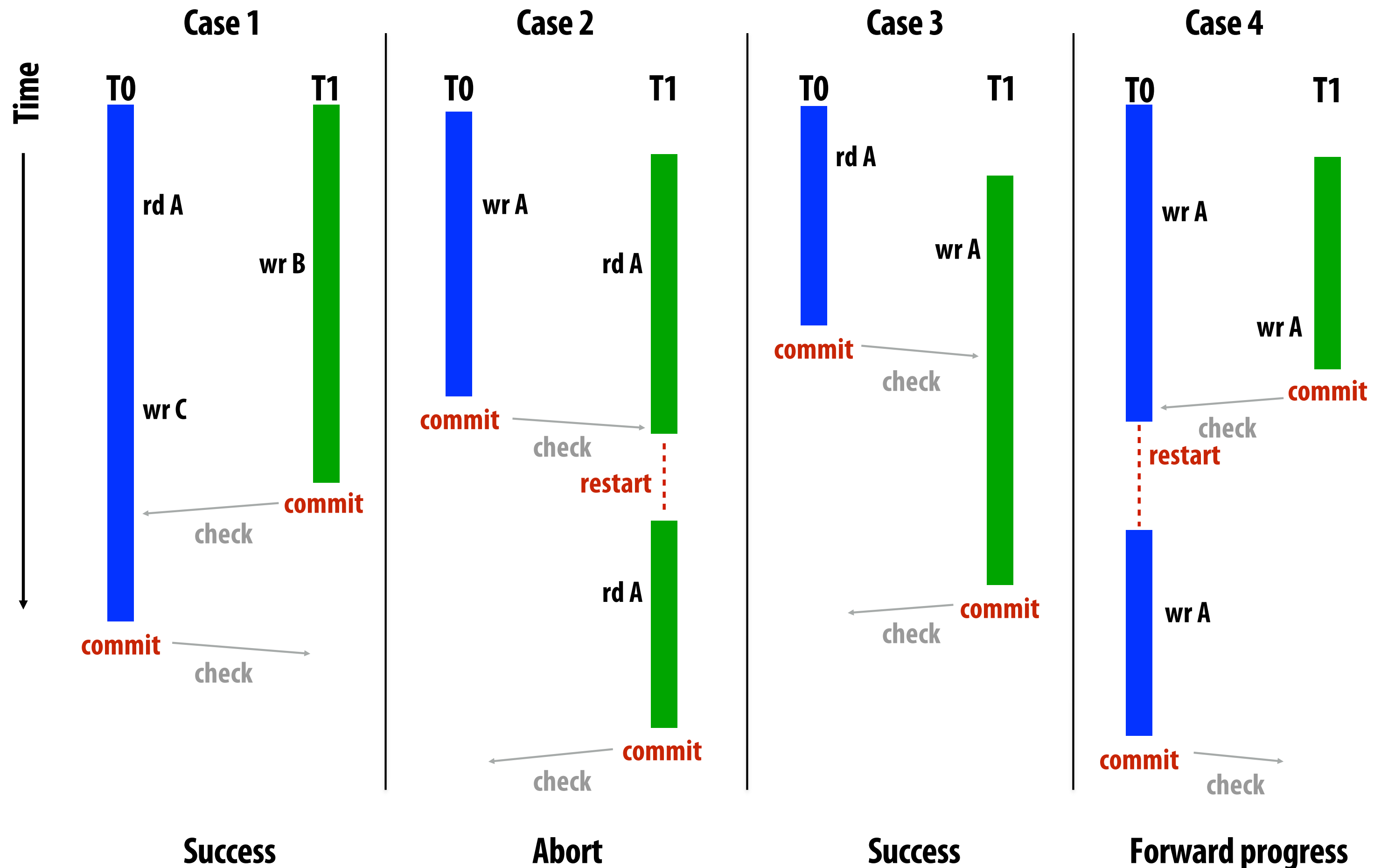
Note: diagrams assume "aggressive" contention manager on writes: writer wins, so other transactions abort)



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



TM implementation space (examples)

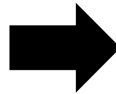
- **Software TM systems**
  - Lazy + optimistic (rd/wr): Sun TL2
  - Lazy + optimistic (rd)/pessimistic (wr): MS OSTM
  - Eager + optimistic (rd)/pessimistic (wr): Intel STM
  - Eager + pessimistic (rd/wr): Intel STM

- **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()
```

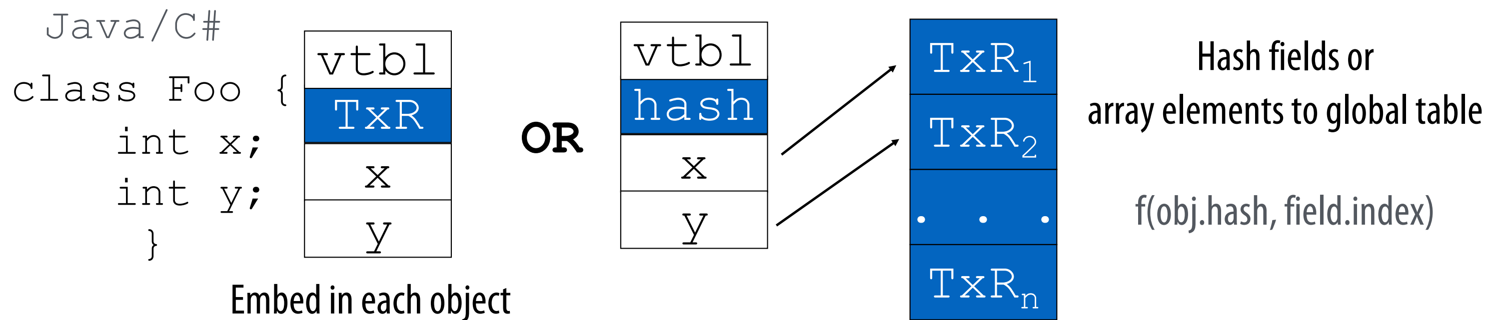
- Software barriers (STM function call) for TM bookkeeping
  - Versioning, read/write-set tracking, commit, . . .
  - Using locks, timestamps, data copying, . . .
- Requires function cloning or dynamic translation
  - Function used inside and outside of transaction

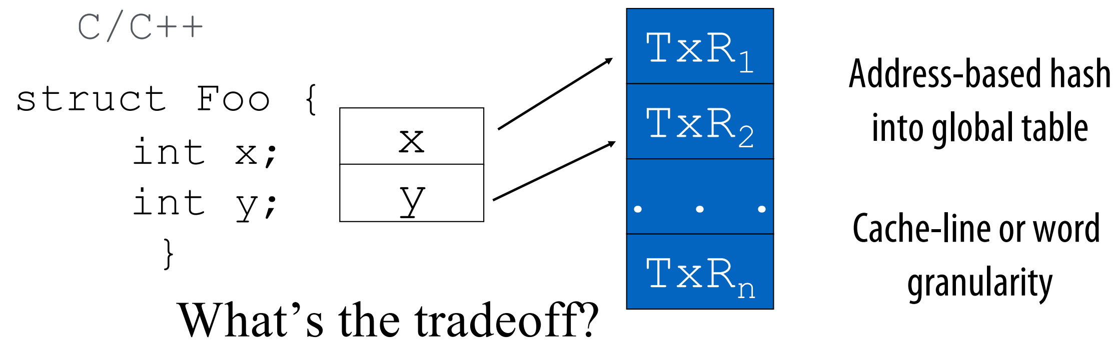
STM Runtime Data Structures

- 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

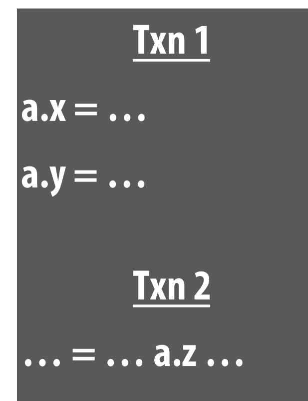




What's the tradeoff?

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



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

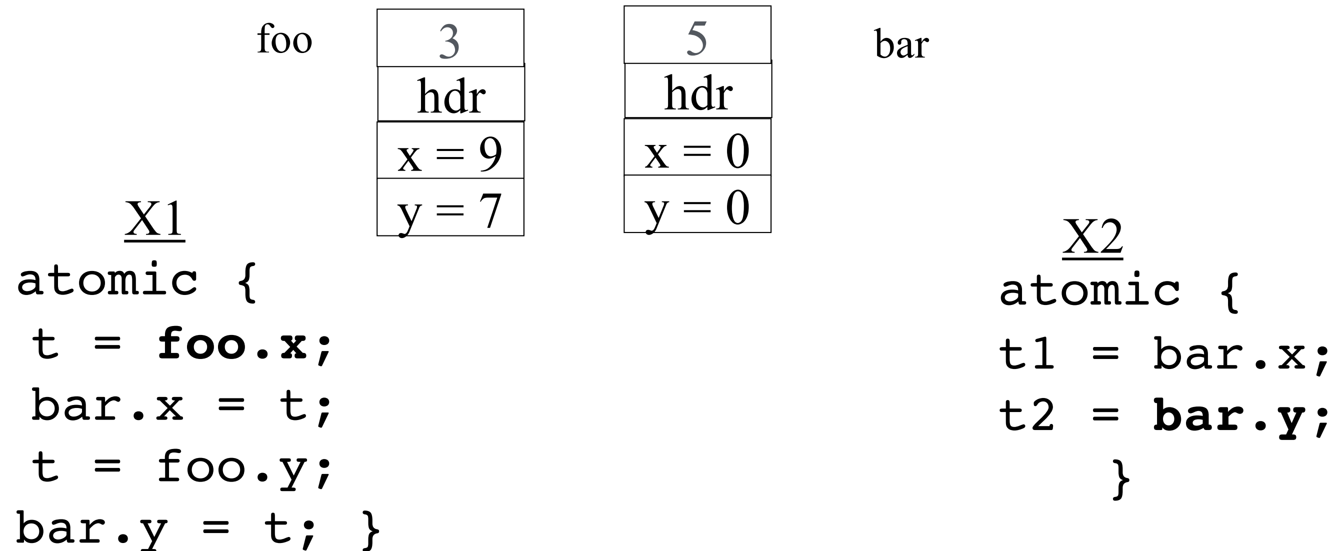
- STM write (pessimistic)
  - Validate data
    - Check if unlocked and data version ≤ local timestamp
  - Acquire lock
  - Insert in write set
  - Create undo log entry
  - Write data in place (eager)

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

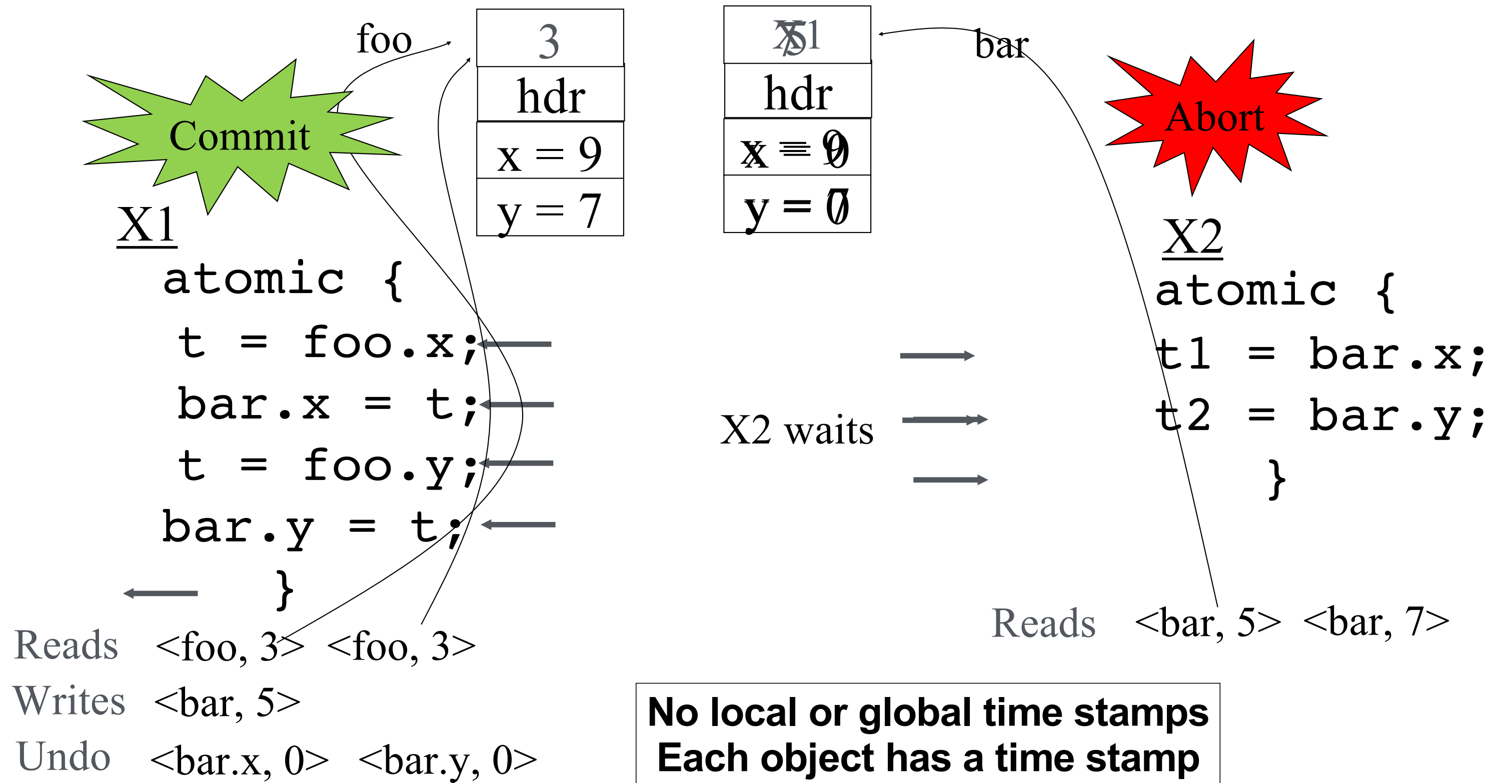
- STM commit
  - Atomically increment global timestamp by 2 (LSb used for write-lock)
  - If preincremented (old) global timestamp > local timestamp, validate read-set
    - Check for recently committed transactions
  - For each item in the write set
    - Release the lock and set version number to global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

STM Example



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

STM Example



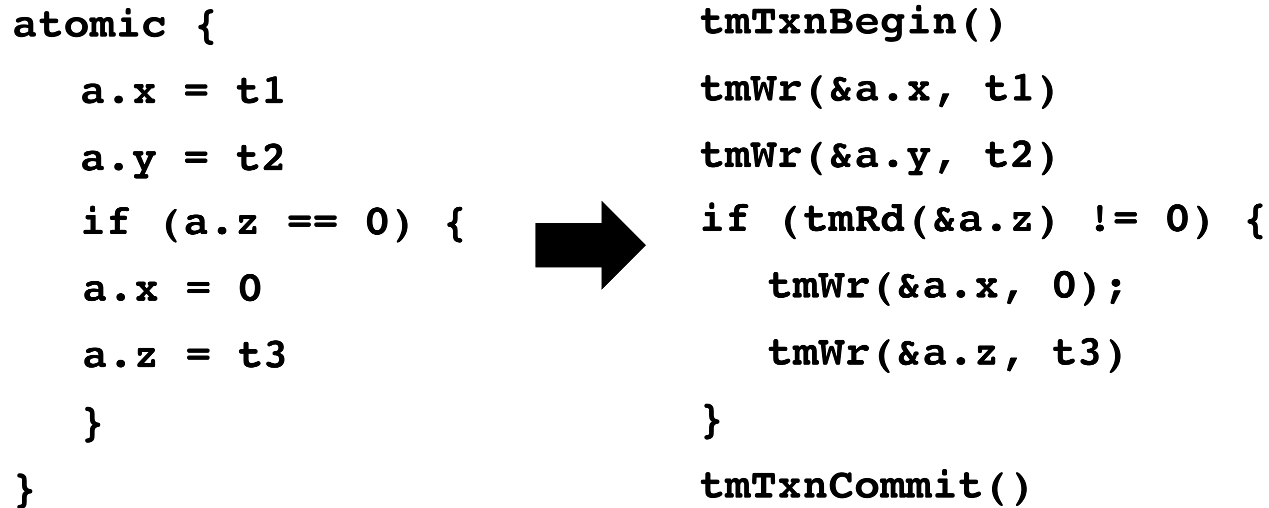
TM Implementation Summary 1

- 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)

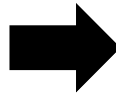
Optimizing Software Transactions



- Monolithic barriers hide redundant logging & locking from the compiler

Optimizing Software Transactions

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

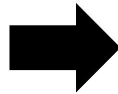


```
txnOpenForWrite(a)
txnLogObjectInt(&a.x, a)
a.x = t1
txnOpenForWrite(a)
txnLogObjectInt(&a.y, a)
a.y = t2
txnOpenForRead(a)
if(a.z != 0) {
 txnOpenForWrite(a)
 txnLogObjectInt(&a.x, a)
 a.x = 0
 txnOpenForWrite(a)
 txnLogObjectInt(&a.z, a)
 a.z = t3
}
```

- Decomposed barriers expose redundancies

Optimizing Software Transactions

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

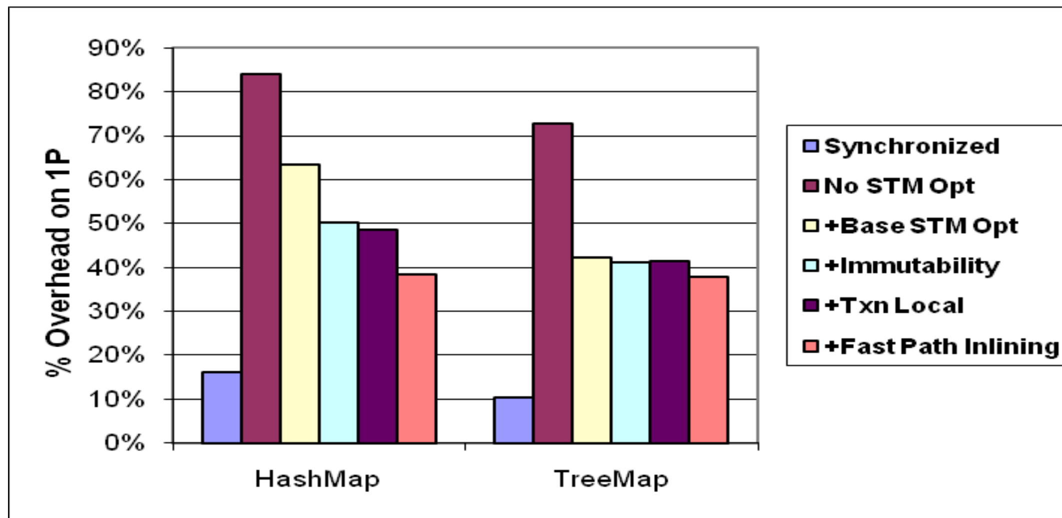


```
txnOpenForWrite(a)
txnLogObjectInt(&a.x, a)
a.x = t1
txnLogObjectInt(&a.y, a)
a.y = t2
if (a.z != 0) {
   a.x = 0
   txnLogObjectInt(&a.z, a)
   a.z = t3
}
```

- Allows compiler to optimize STM code
- Produces fewer & cheaper STM operations

Effect of Compiler Optimizations

- 1 thread overheads over thread-unsafe baseline



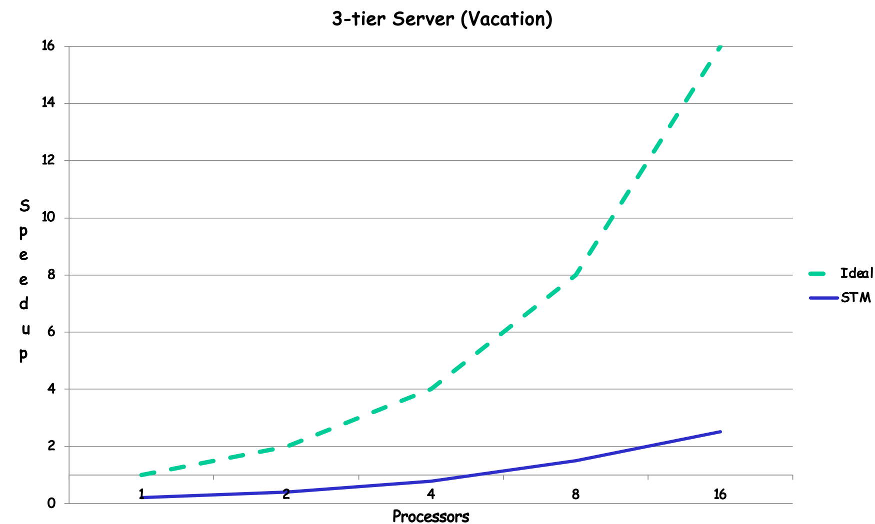
- With compiler optimizations
  - <40% over no concurrency control
  - <30% over lock-based synchronization

STM Question

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

```
atomic{
     obj.f1=42;
}
```

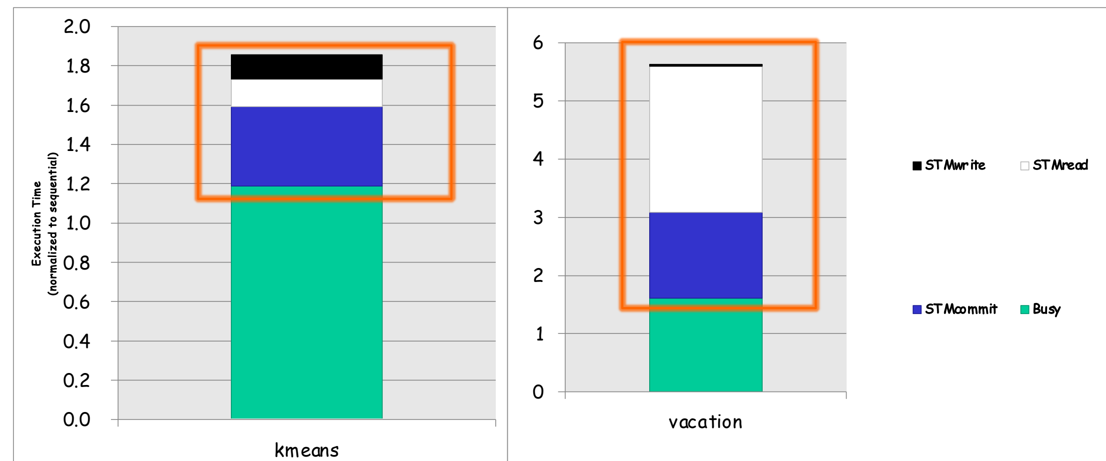
Motivation for Hardware Support



- 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, …

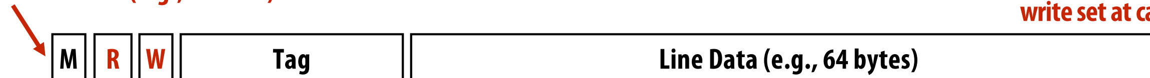
	HTM	STM	HW-STM
Write versioning	HW	SW	SW
Conflict detection	HW	SW	HW

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

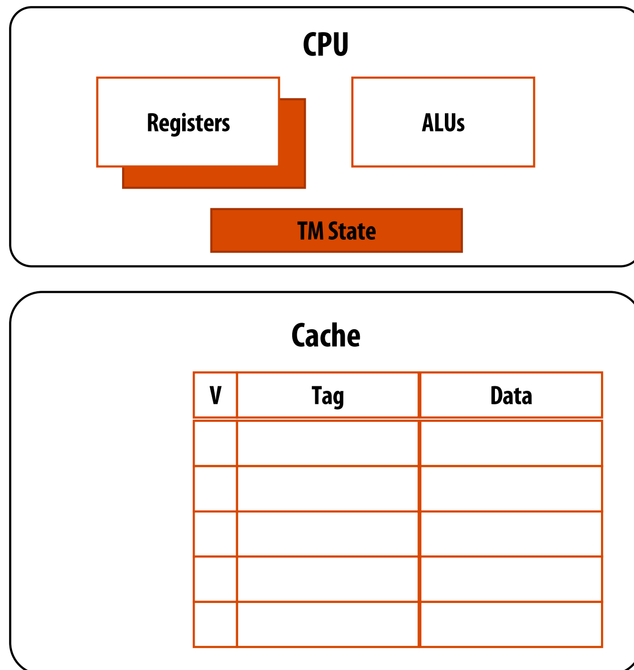
- **Cache lines annotated to track read set and write set**
  - R bit: indicates data read by transaction (set on loads)
  - W bit: indicates data written by transaction (set on stores)
    - R/W bits can be at word or cache-line granularity
  - R/W bits gang-cleared on transaction commit or abort



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

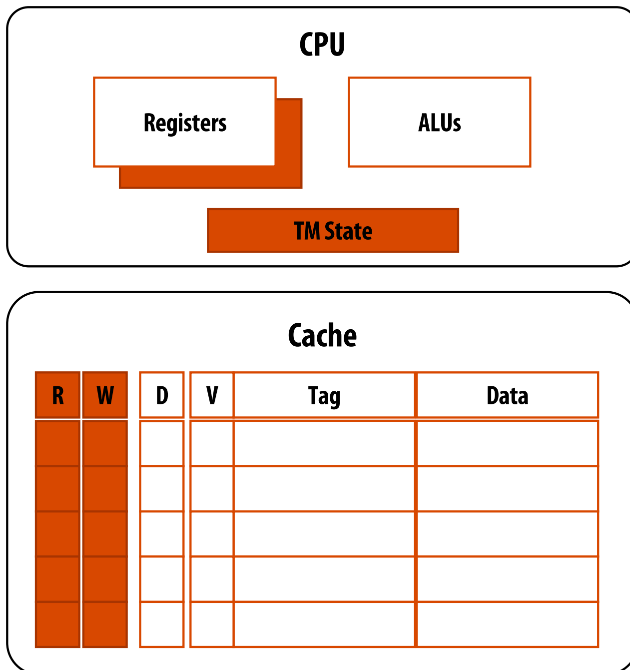
- **Coherence requests check R/W bits to detect conflicts**
  - Observing shared request to W-word is a read-write conflict
  - Observing exclusive (intent to write) request to R-word is a write-read conflict
  - Observing exclusive (intent to write) request to W-word is a write-write conflict

Example HTM implementation: lazy-optimistic



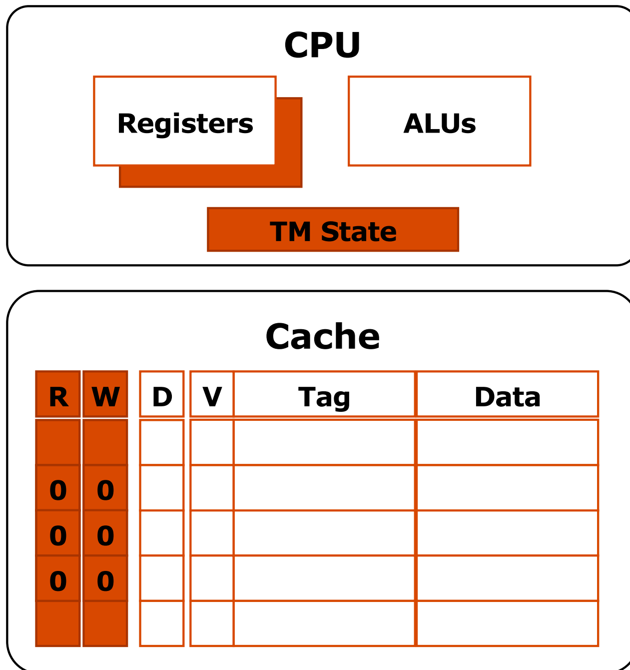
- **CPU changes**
  - Ability to checkpoint register state (available in many CPUs)
  - TM state registers (status, pointers to abort handlers, ...)

Example HTM implementation: lazy-optimistic



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

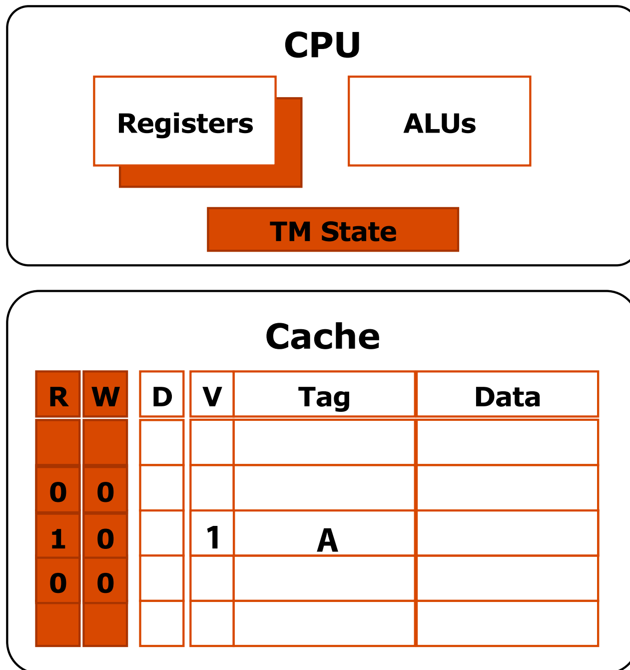
HTM transaction execution




```
Xbegin  ⬅
    Load A
    Load B
    Store C ⇐ 5
Xcommit
```

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

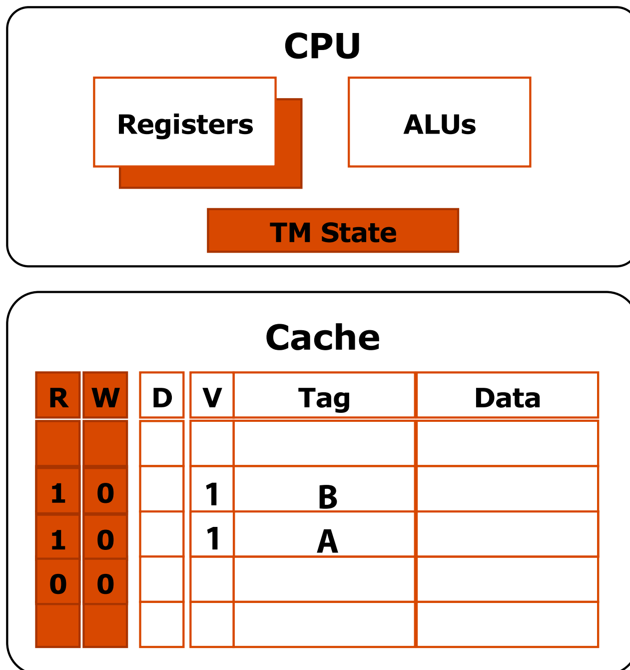
HTM transaction execution




```
Xbegin
    Load A  ⇐
    Load B
    Store C ⇐ 5
Xcommit
```

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

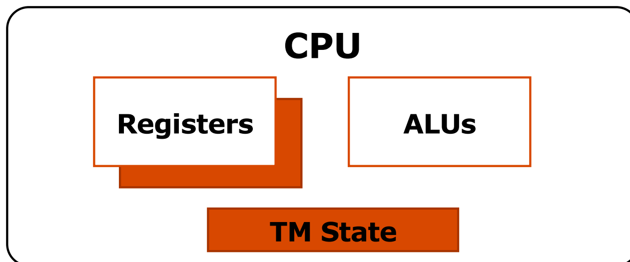
HTM transaction execution




```
Xbegin
    Load A
    Load B  ⟵
    Store C ⇐ 5
Xcommit
```

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

HTM transaction execution



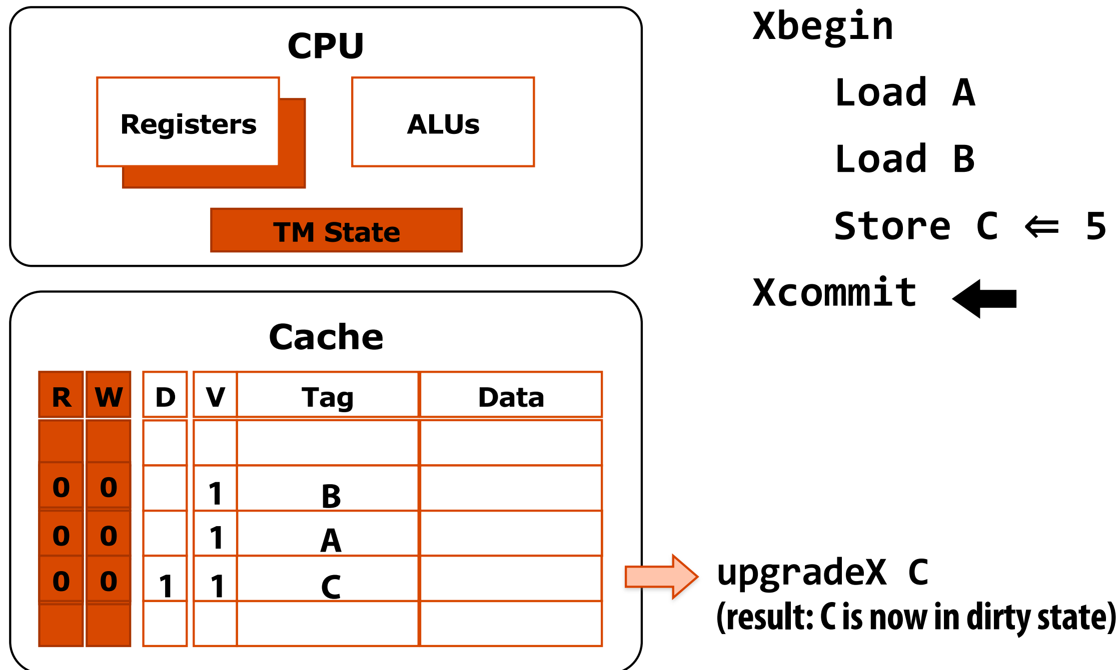
Cache

R	W	D	V	Tag	Data
1	0		1	B	
1	0		1	A	
0	1		1	C	

```
Xbegin
    Load A
    Load B
    Store C ⇐ 5
Xcommit
```

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

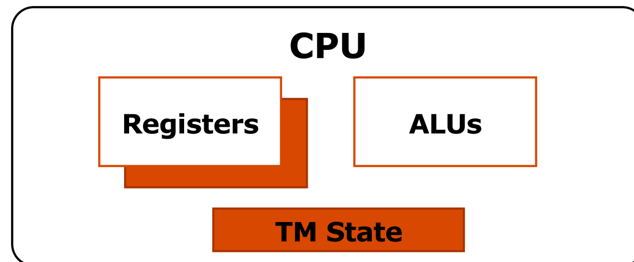
HTM transaction execution: commit

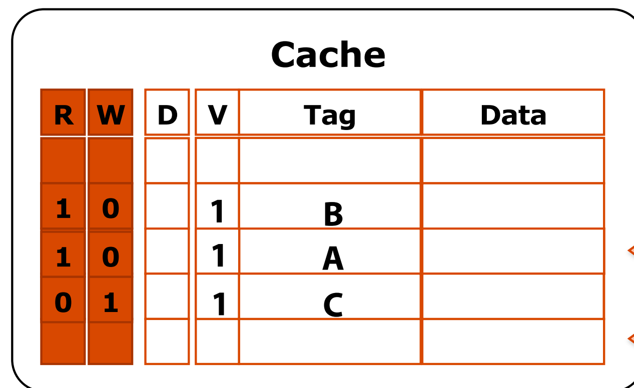


- **Fast two-phase commit**
  - Validate: request RdX access to write set lines (if needed)
  - Commit: gang-reset R and W bits, turns write set data to valid (dirty) data

HTM transaction execution: detect/abort

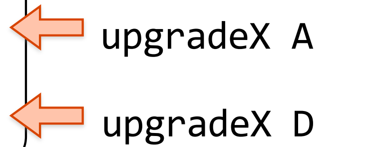
Assume remote processor commits transaction with writes to A and D





R	W	D	V	Tag	Data
1	0		1	B	
1	0		1	A	
0	1		1	C	

```
Xbegin
    Load A
    Load B
    Store C ⇐ 5
Xcommit
```

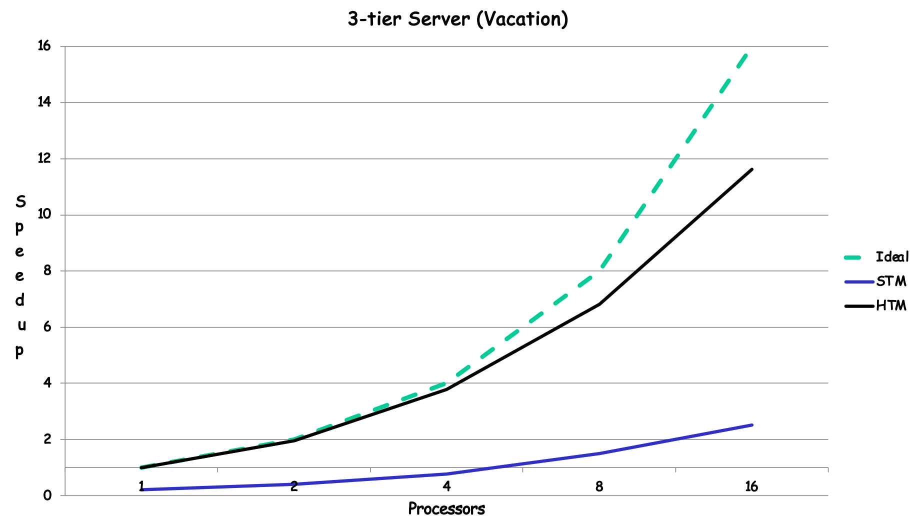


coherence requests from another core's commit

(remote core's write of A conflicts with local read of A: triggers abort of pending local transaction)

- **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

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 14+:

Heterogeneous Parallelism and Hardware Specialization

Parallel Computing
Stanford CS149, Fall 2022

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

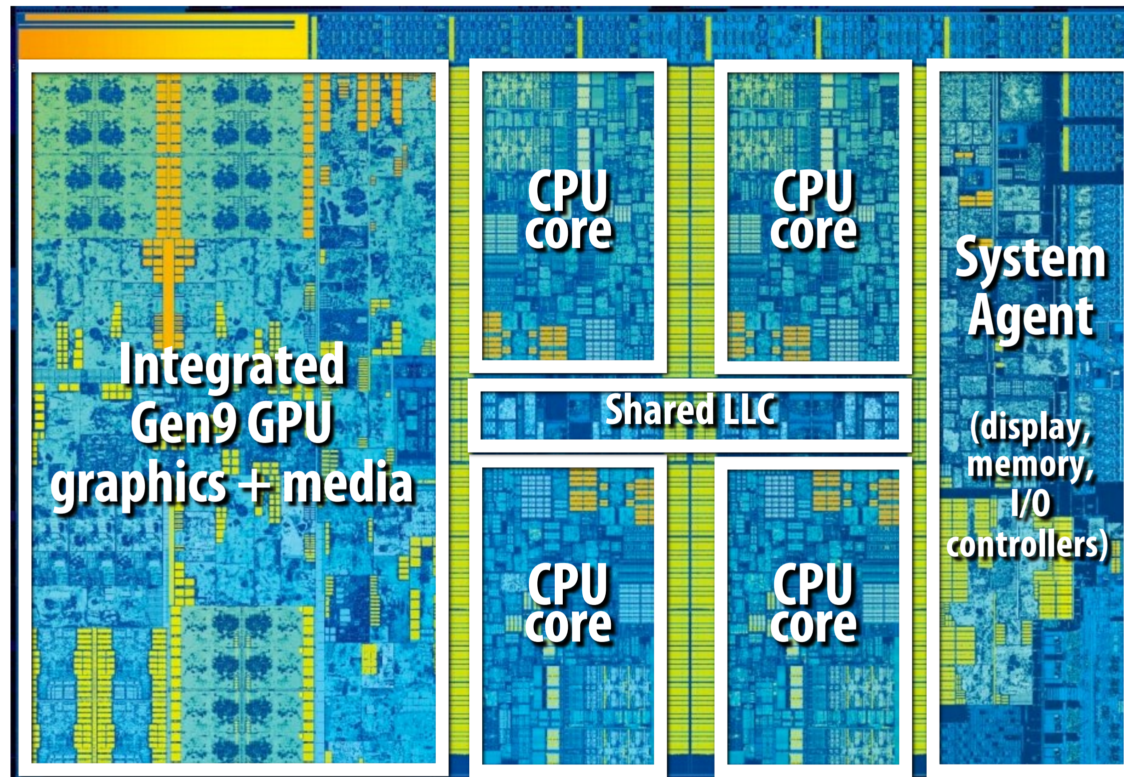
And components with unpredictable access, but those accesses might cache well.

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)

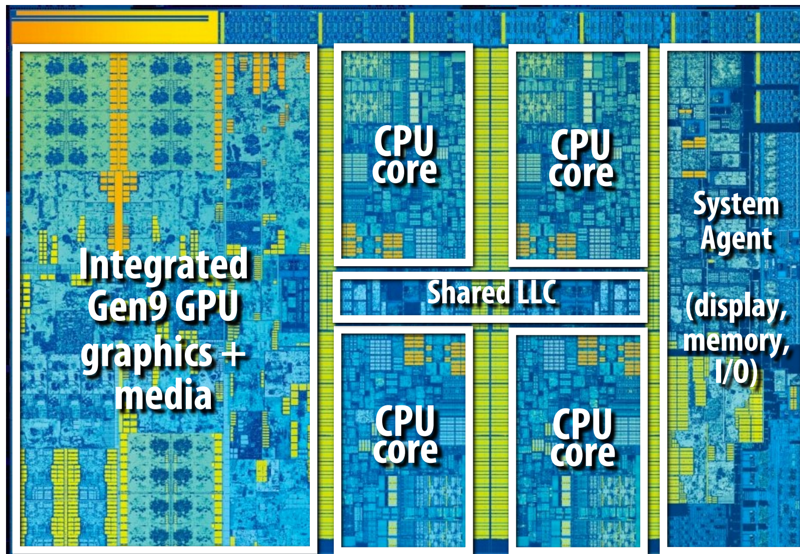
(6th Generation Core i7 architecture)



4 CPU cores + graphics cores + media accelerators

Example: Intel “Skylake" (2015)

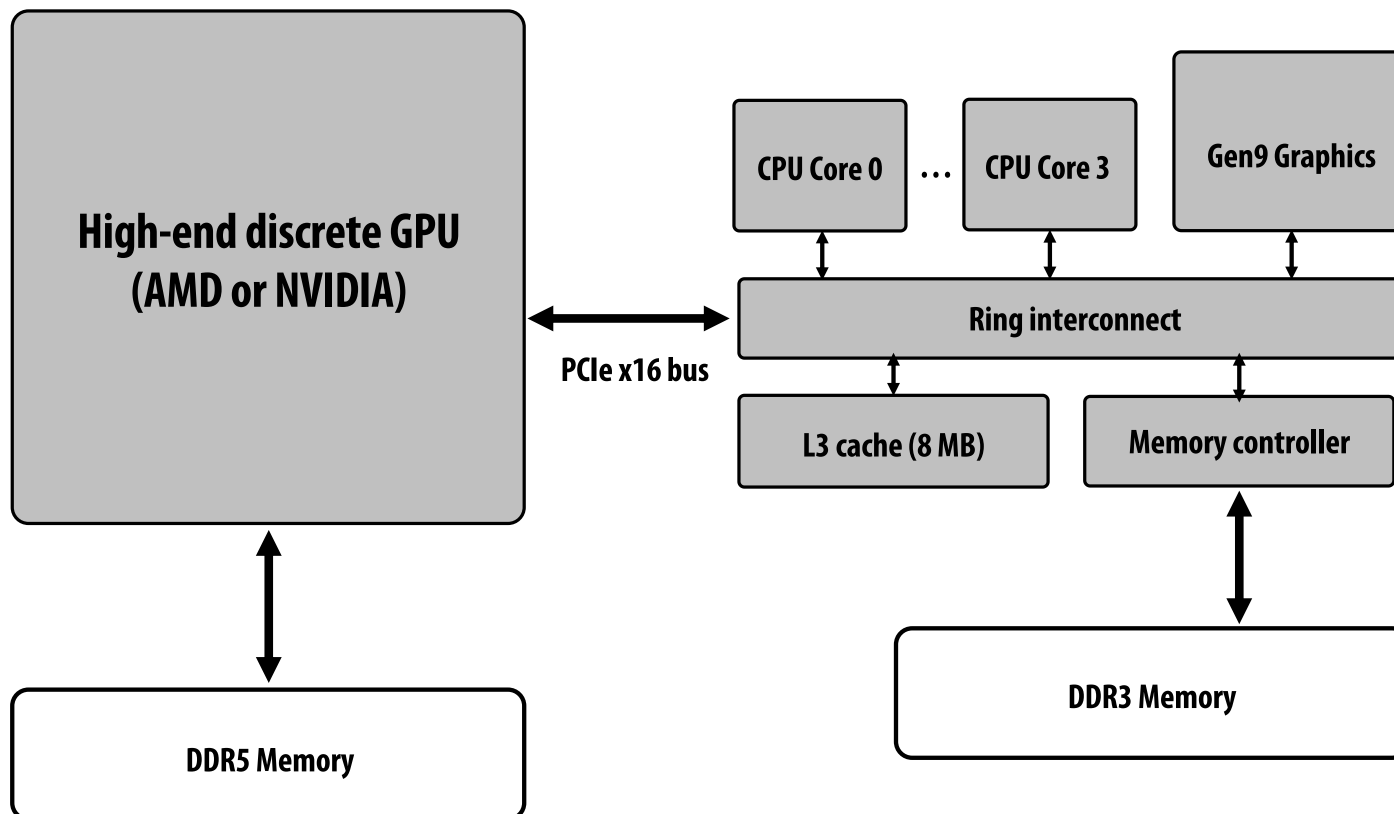
(6th Generation Core i7 architecture)



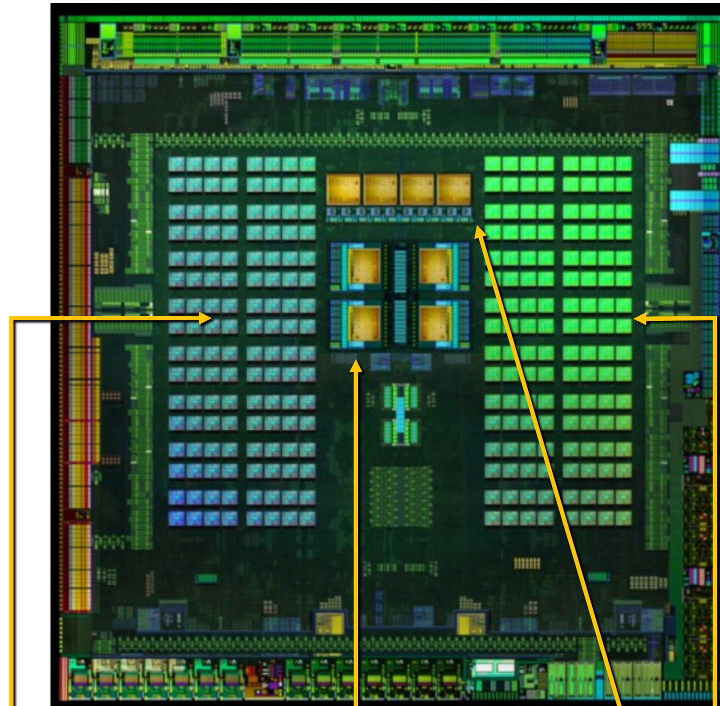
- CPU cores and graphics cores share same memory system
- Also share LLC (L3 cache)
  - Enables, low-latency, high-bandwidth communication between CPU and integrated GPU
- Graphics cores are cache coherent with CPU cores

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



Mobile heterogeneous processors

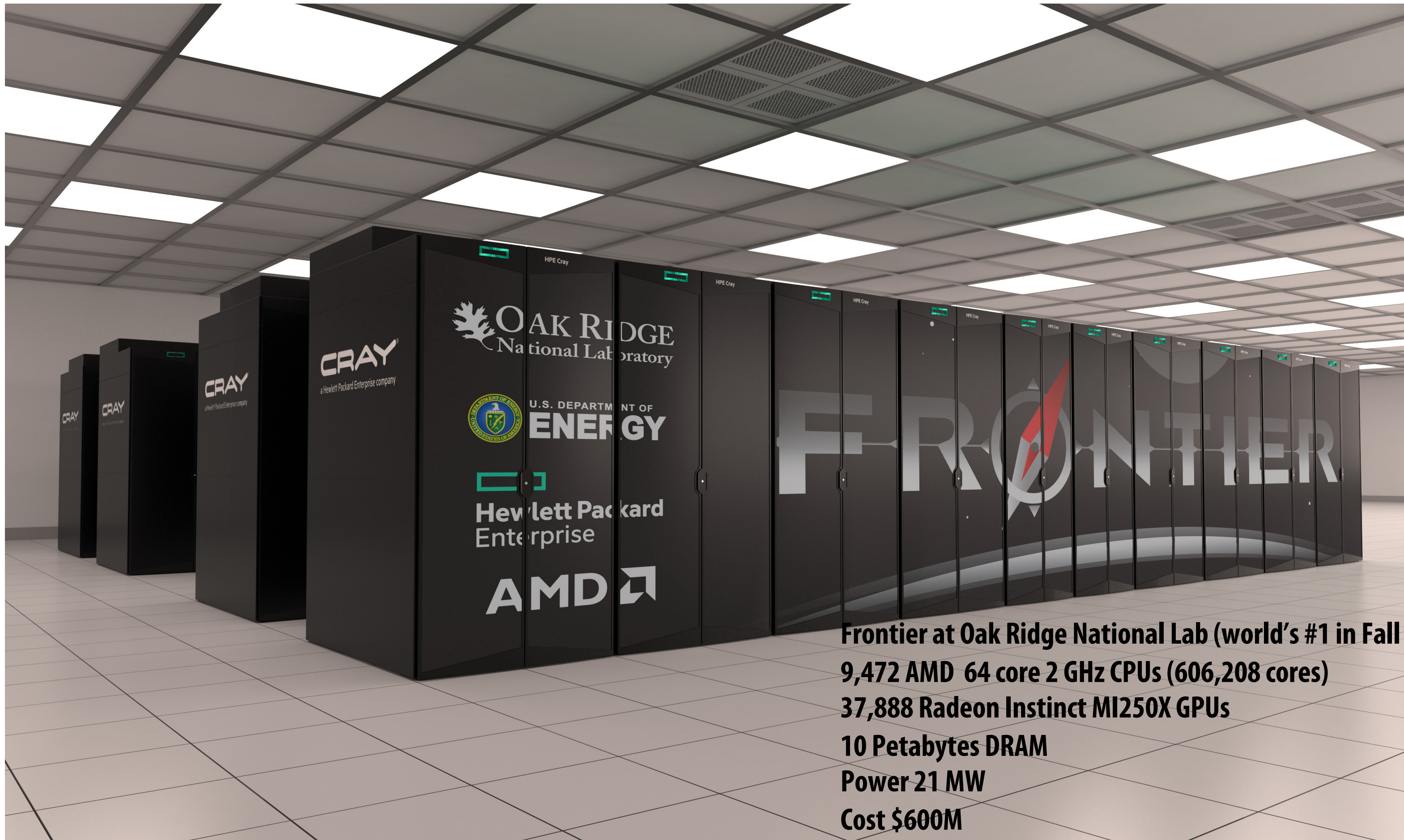


NVIDIA Tegra X1
Four ARM Cortex A57 CPU cores for applications
Four low performance (low power) ARM A53 CPU cores
One Maxwell SMM (256 "CUDA" cores)

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

A11 image credit: TechInsights Inc.'
* Disclaimer: estimates by TechInsights, not an official Apple reference.

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

Performance and Power

Performance | Energy efficiency

$$Power = \frac{Ops}{second} \times \frac{Joules}{Op}$$

FIXED

What is the magnitude of improvement from specialization?

Specialization (fixed function) ⇒ better energy efficiency

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

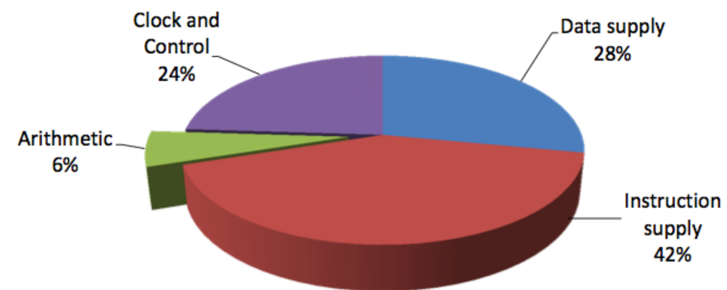
[Source: Chung et al. 2010 , Dally 08]

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 — Address translation, communicate with icache, access icache, etc.
- Decode instruction — Translate op to uops, access uop cache, etc.
- 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



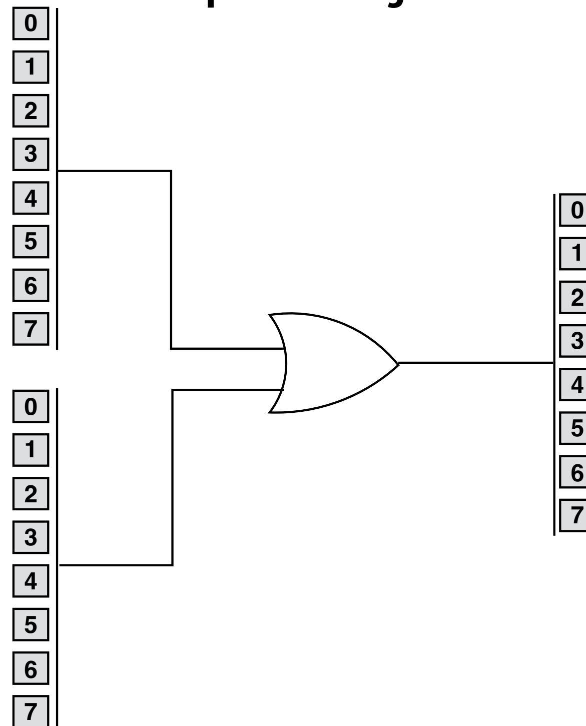
Efficient Embedded Computing [Dally et al. 08]

[Figure credit Eric Chung]

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

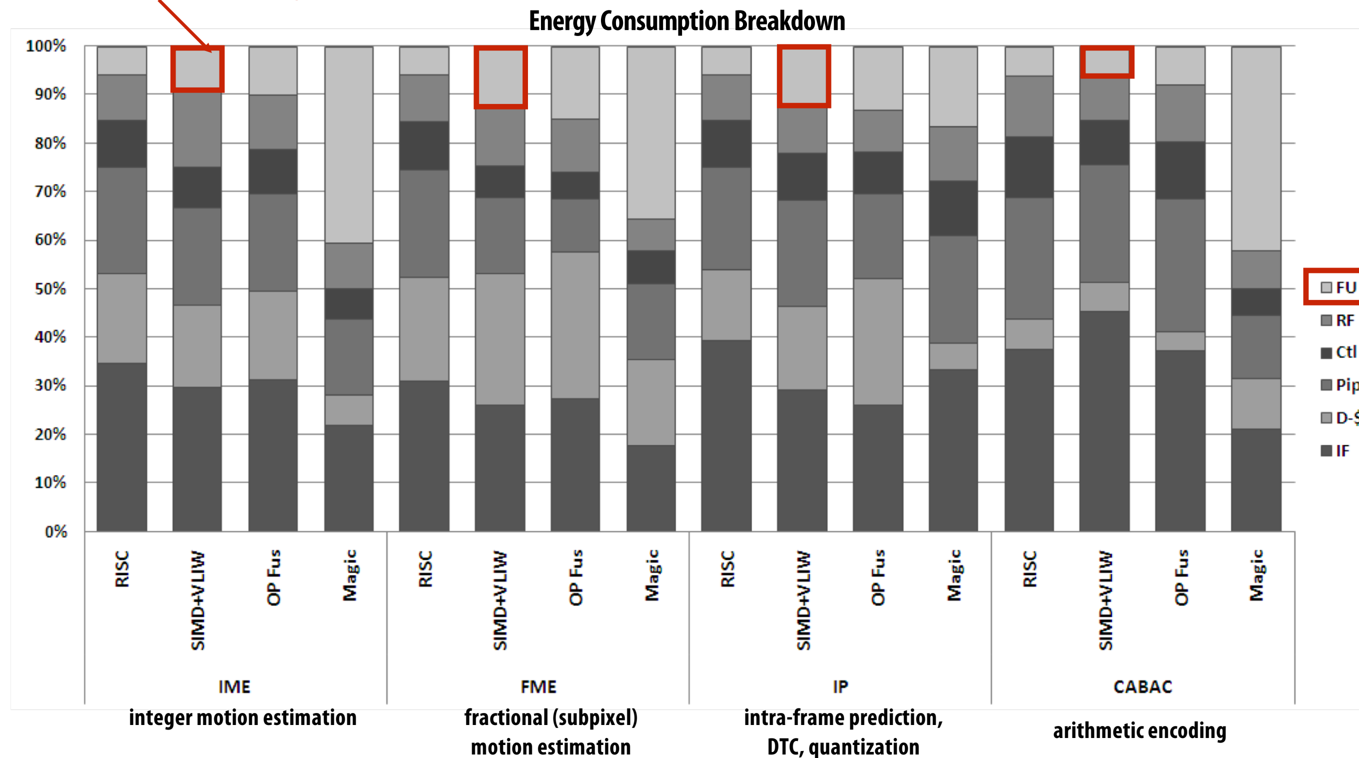
Example: 8-bit logical OR



H.264 video encoding: fraction of energy consumed by functional units is small (even when using SIMD)

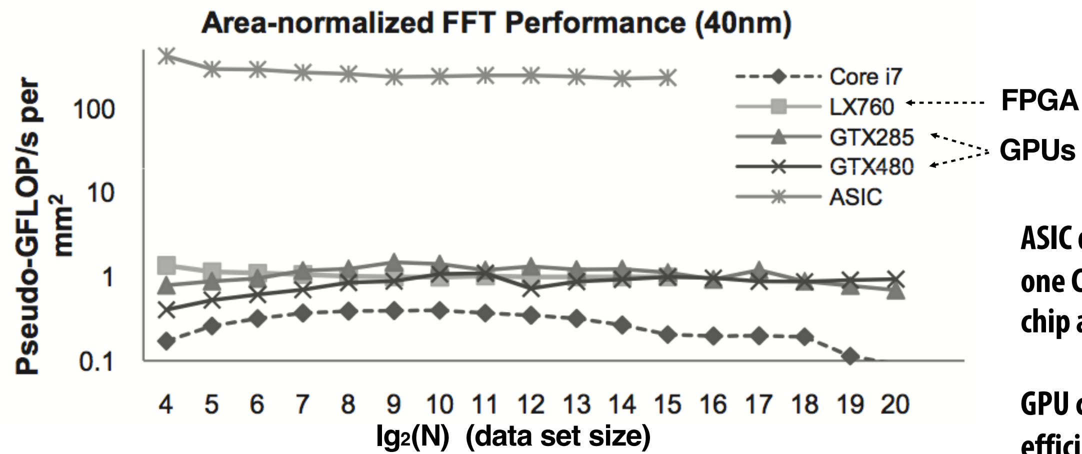
Even after encoding implemented with SIMD instruction

[Hameed et al. ISCA 2010]



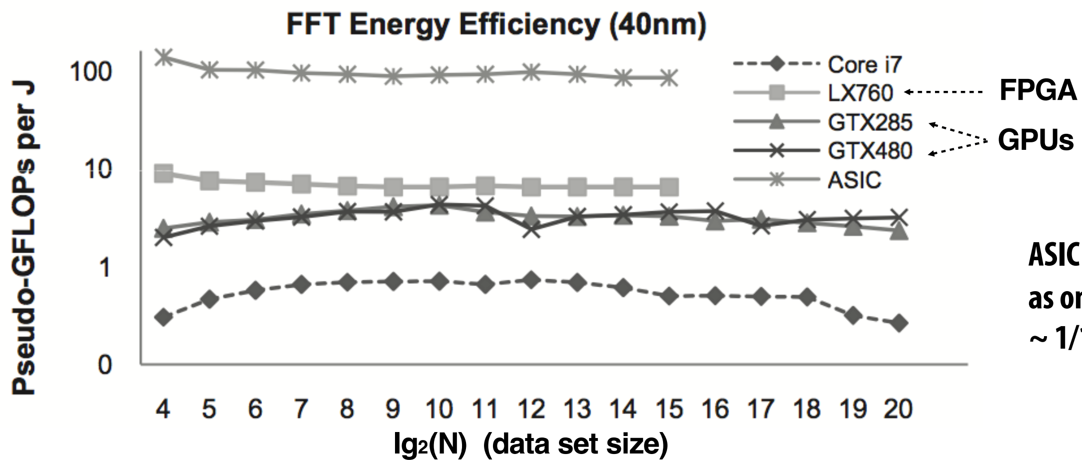
FU = functional units
RF = register fetch
Ctrl = misc pipeline control
Pip = pipeline registers (interstage)
D-$ = data cache
IF = instruction fetch + instruction cache

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.



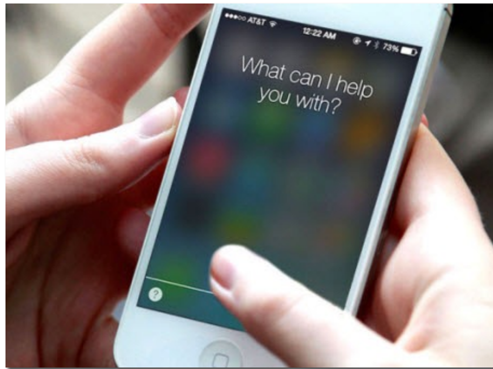
ASIC delivers same performance as one CPU core using only ~ 1/100th the power

[Chung et al. MICRO 2010]

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



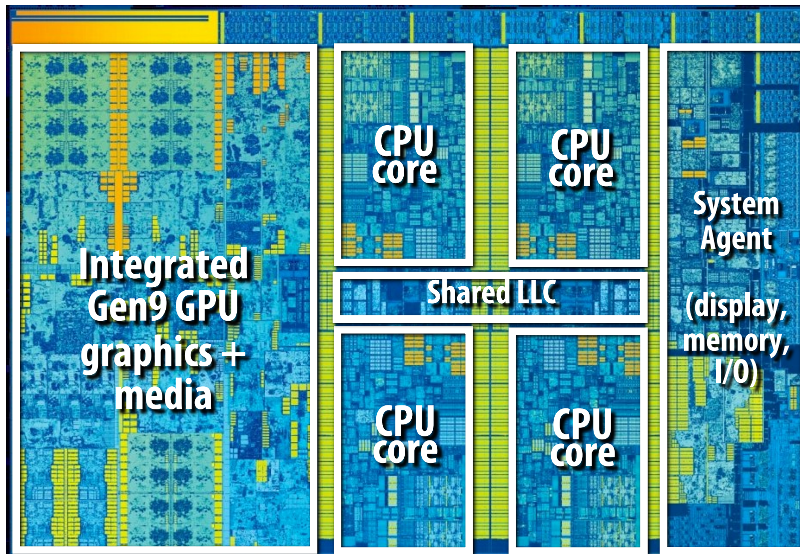
iPhone:
Siri activated by button press or holding phone up to ear

Amazon Echo / Google Home
Always listening

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

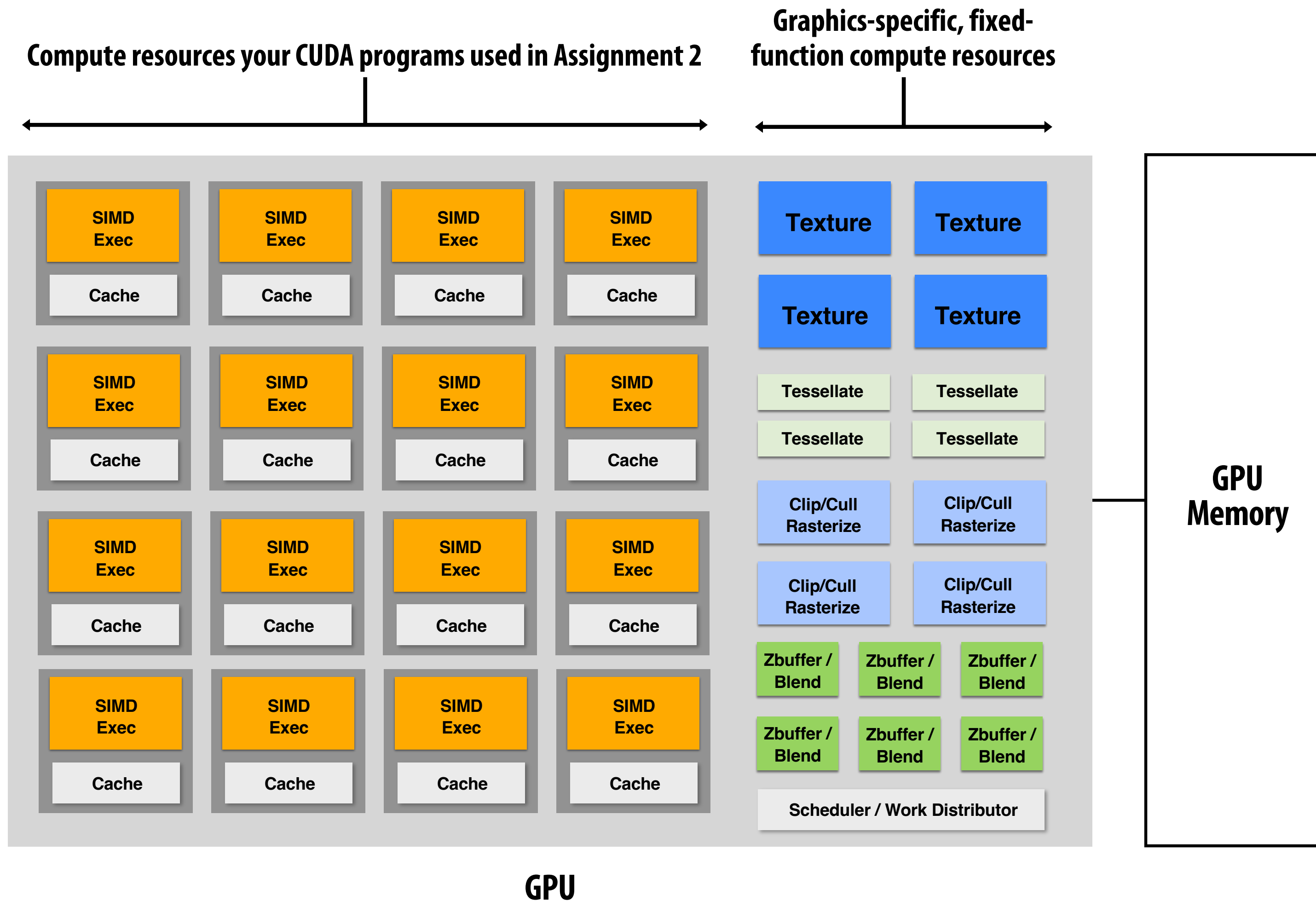
Example: Intel "Skylake" (2015)

(6th Generation Core i7 architecture)



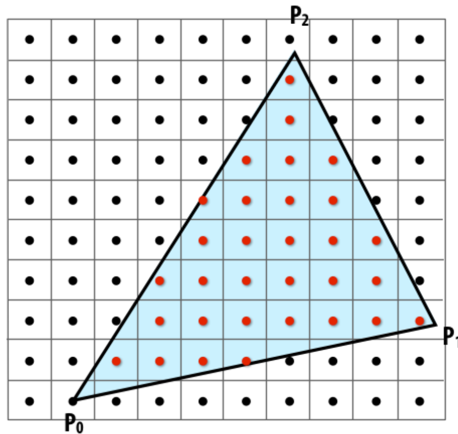
- CPU cores and graphics cores share same memory system
- Also share LLC (L3 cache)
  - Enables, low-latency, high-bandwidth communication between CPU and integrated GPU
- Graphics cores are cache coherent with CPU cores

GPU's are themselves heterogeneous multi-core processors

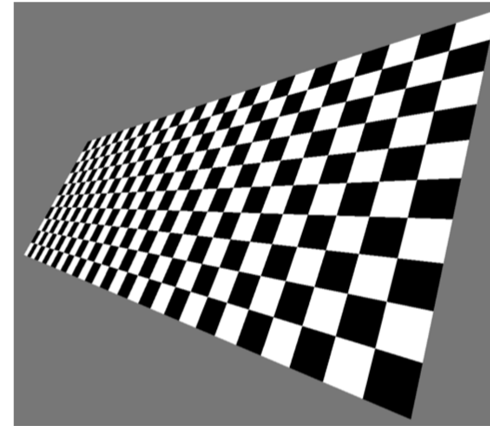


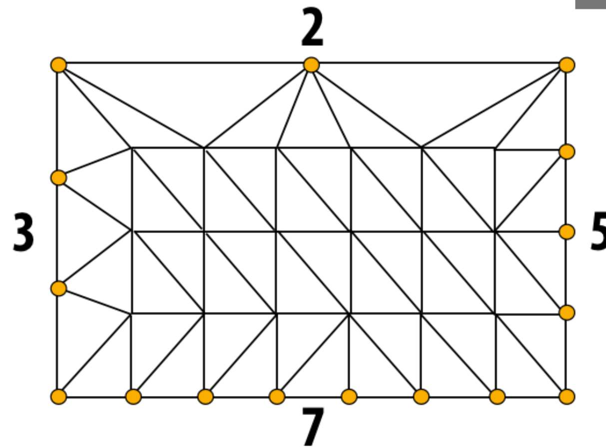
Example graphics tasks performed in fixed-function HW

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

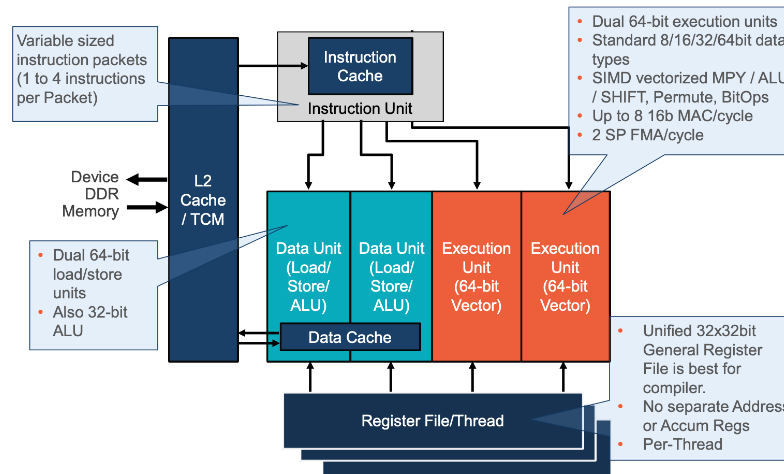
Complex instructions (e.g., SIMD/VLIW): perform many operations per instruction (amortize cost of control)

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)

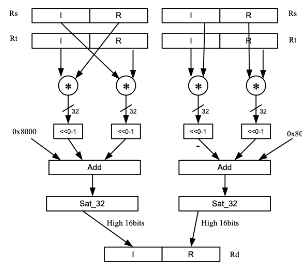
Below: innermost loop of FFT
Hexagon DSP performs 29 "RISC" ops per cycle



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

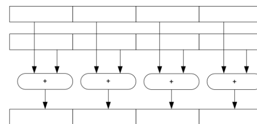
```
{ R17:16 = MEMD(R0++M1)
  MEMD(R6++M1) = R25:24
  R20 = CMPY(R20, R8):<<1:rnd:sat
  R11:10 = VADDH(R11:10, R13:12)
}:endloop0
```

Complex multiply with round and saturation



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

Vector 4x16-bit Add



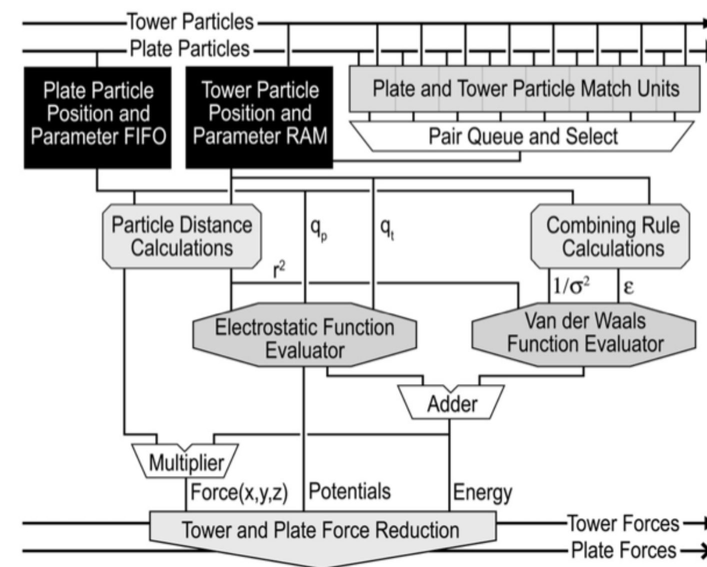
Hexagon DSP is in Google Pixel phone

Anton supercomputer for molecular dynamics

[Developed by DE Shaw Research]

- 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

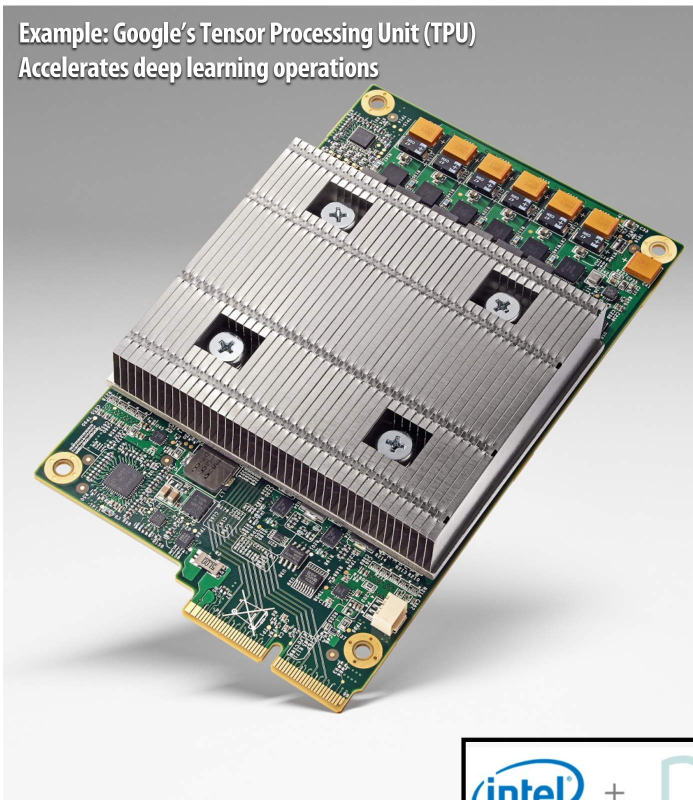
network designed for communication patterns of N-body simulations



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…

- Cambricon: an instruction set architecture for neural networks, Liu et al. ISCA 2016
- EIE: Efficient Inference Engine on Compressed Deep Neural Network, Han et al. ISCA 2016
- Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing, Albericio et al. ISCA 2016
- Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators, Reagen et al. ISCA 2016
- vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design, Rhu et al. MICRO 2016
- Fused-Layer CNN Architectures, Alwani et al. MICRO 2016
- Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Network, Chen et al. ISCA 2016
- PRIME: A Novel Processing-in-memory Architecture for Neural Network Computation in ReRAM-based Main Memory, Chi et al. ISCA 2016
- DNNWEAVER: From High-Level Deep Network Models to FPGA Acceleration, Sharma et al. MICRO 2016

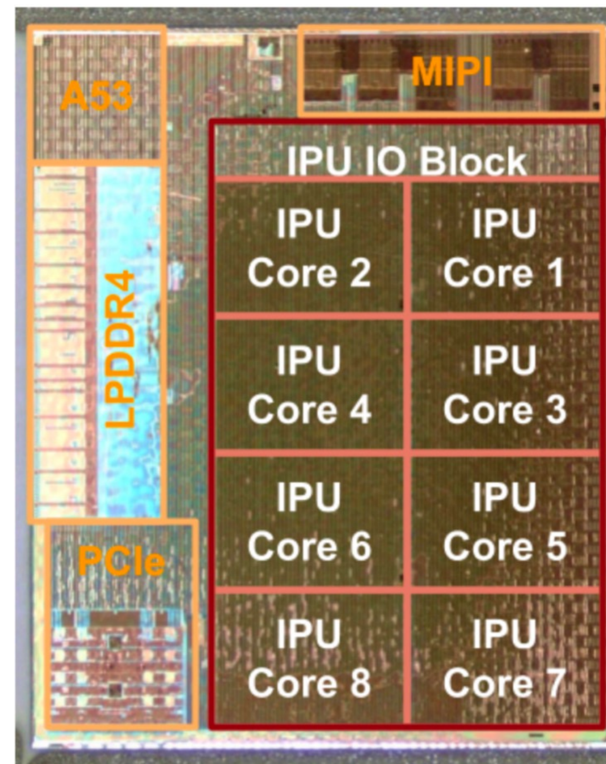


Intel Lake Crest ML accelerator (formerly Nervana)

Example: Google's Pixel Visual Core

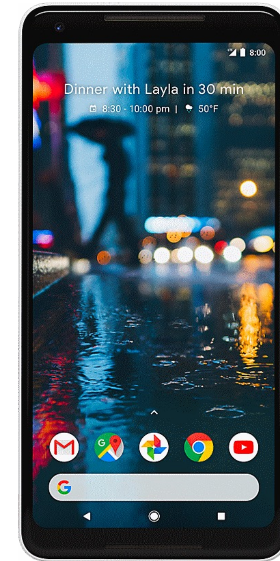
Programmable "image processing unit" (IPU)

- 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)

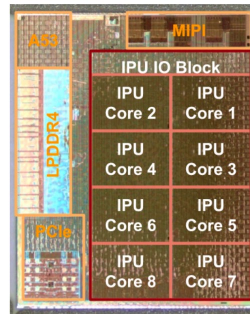


Let's crack open a modern smartphone

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

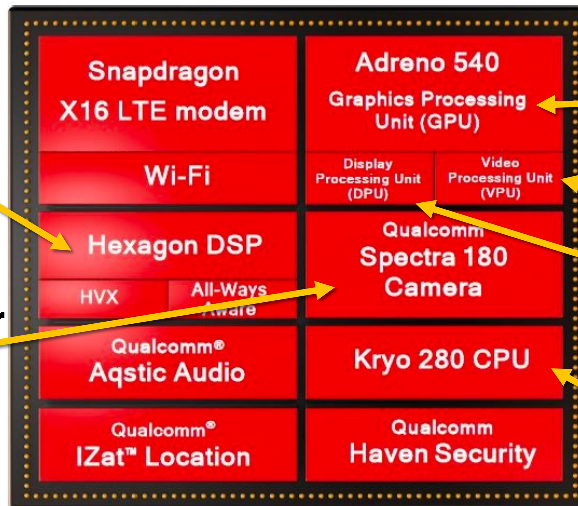


Visual Pixel Core
Programmable image processor and DNN accelerator



"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)

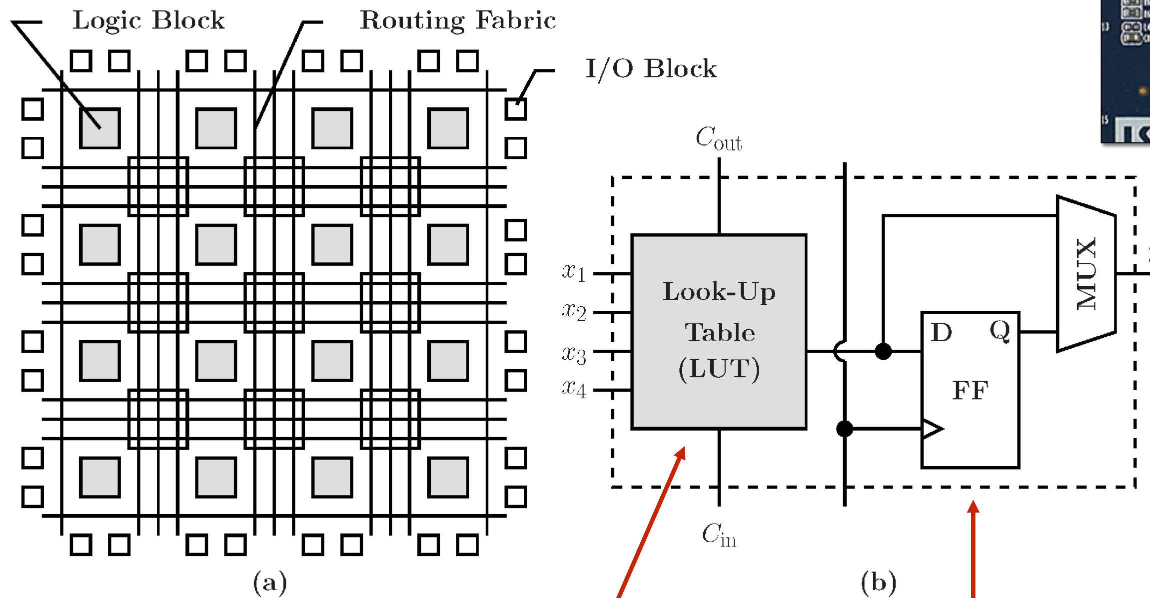
Video encode/decode ASIC

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 FGPA

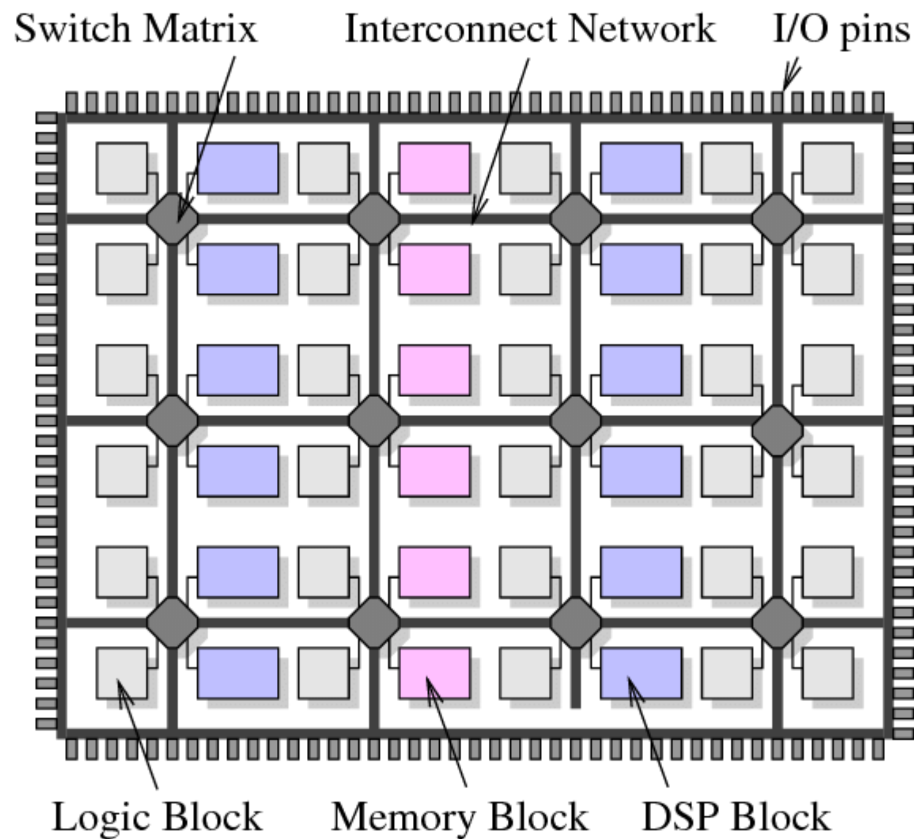


Flip flop (a register)

Programmable lookup table (LUT)

Image credit: Bai et al. 2014

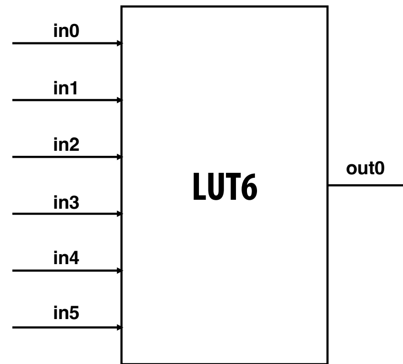
Modern FPGAs



- 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

In	Out
0	0
1	0
2	0
3	0
⋮	⋮
63	1

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

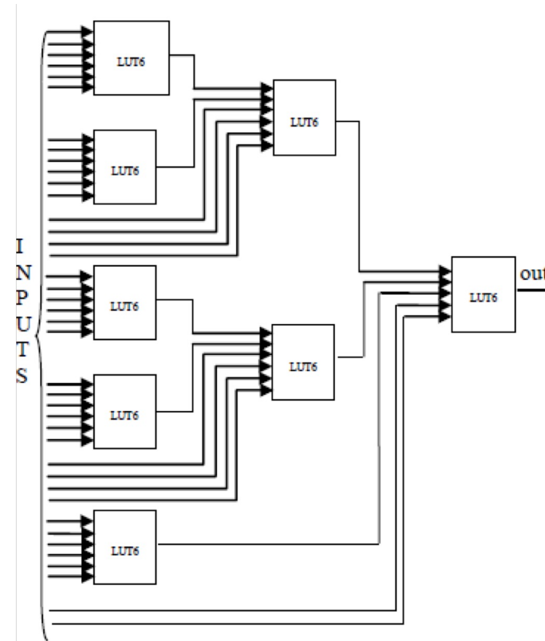


Image credit: [Zia 2013]

Project Catapult

[Putnam et al. ISCA 2014]

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

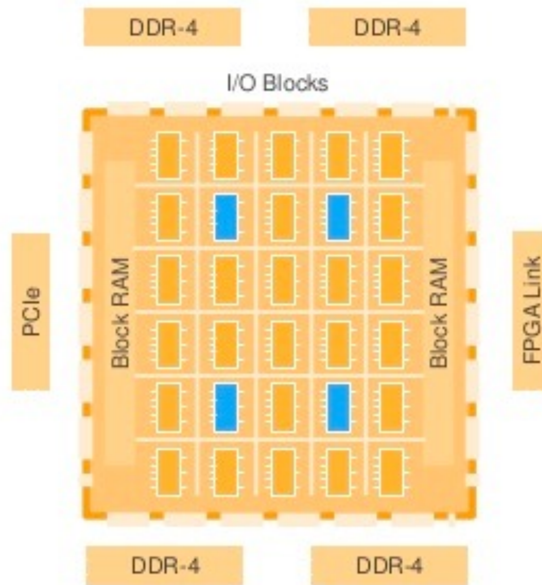
FPGA board

1U server (Dual socket CPU + FPGA connected via PCIe bus)

Amazon F1

- **FPGA's are now available on Amazon cloud services**

What's Inside the F1 FPGA?



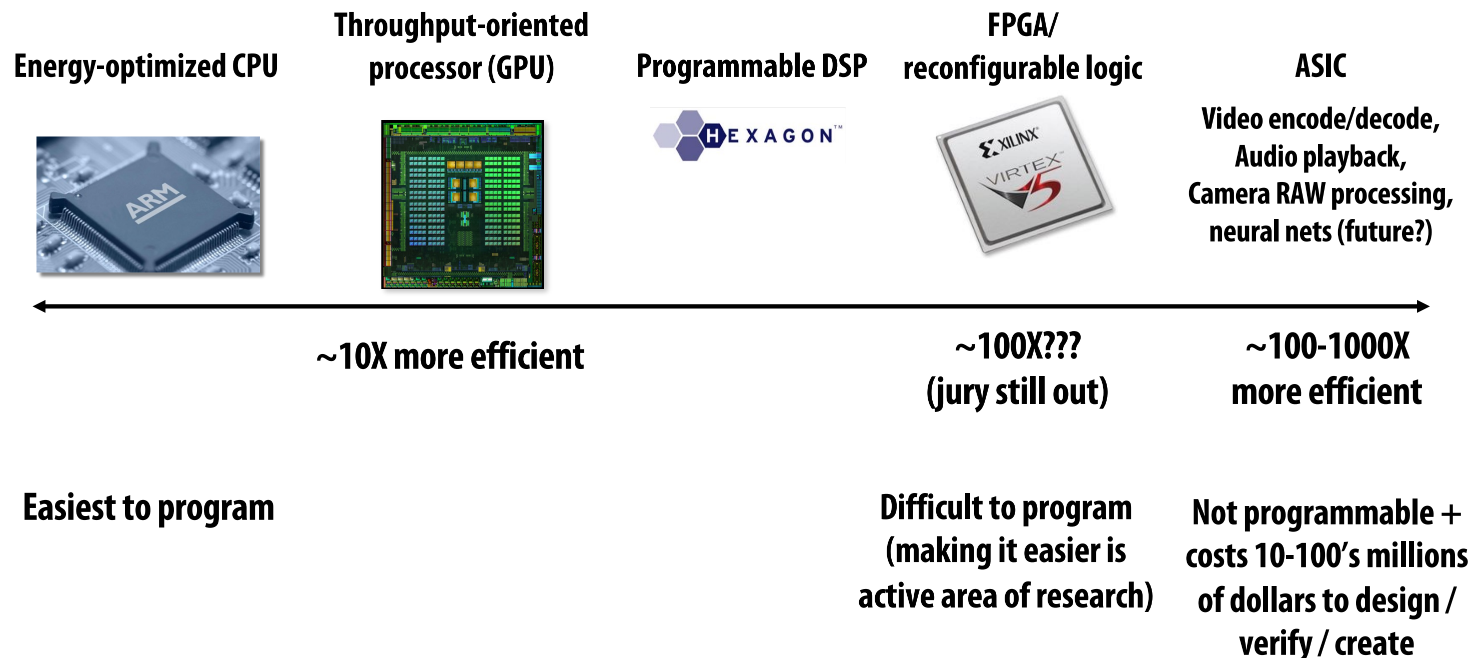
System Logic Block: Each FPGA in F1 provides over 2M of these logic blocks

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



Credit: Pat Hanrahan for this slide design

Challenges of heterogeneous designs:

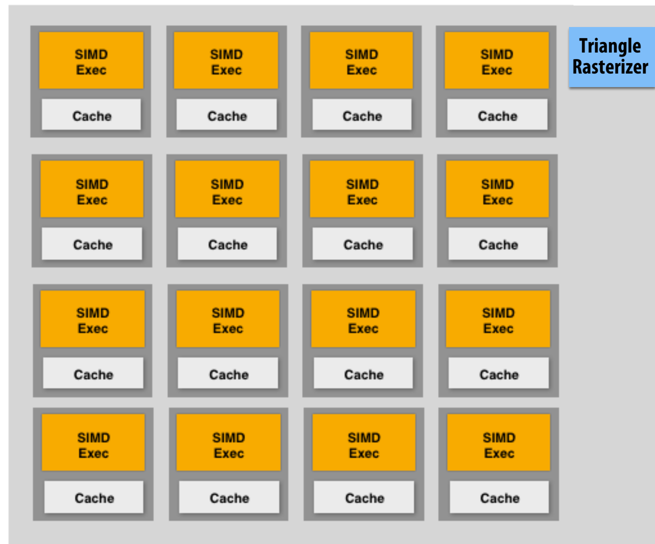
(it’s not easy to realize the potential of specialized, heterogeneous processing)

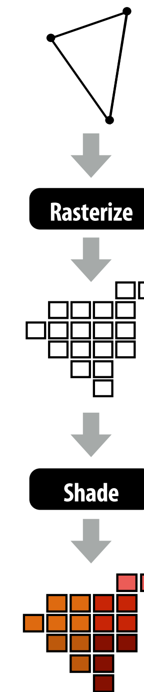
Challenges of heterogeneity

- 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?

Pitfalls of heterogeneous designs

[Molnar 2010]





Consider a two stage graphics pipeline:
Stage 1: rasterize triangles into pixel fragments (using ASIC)
Stage 2: compute color of fragments (on SIMD cores)

Let's say you under-provision the rasterization unit on GPU:
Chose to dedicate 1% of chip area used for rasterizer to achieve throughput T fragments/clock
But really needed throughput of 1.2T to keep the cores busy (should have used 1.2% of chip area for rasterizer)

Now the programmable cores only run at 80% efficiency (99% of chip is idle 20% of the time = same perf as 79% smaller chip!)
So tendency is to be conservative and over-provision fixed-function components (diminishing their advantage)

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

Suggests that recomputing values, rather than storing and reloading them, is a better answer when optimizing code for energy efficiency!

- 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.

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?