

Lecture 15:

TM & Heterogeneous Parallelism and Hardware Specialization

**Parallel Computing
Stanford CS149, Winter 2019**

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 for conflicts)**
 - **Check if unlocked and data version \leq local timestamp**
 - **If not, validate all data in read set for consistency**
 - **Insert in read set**
 - **Return value**

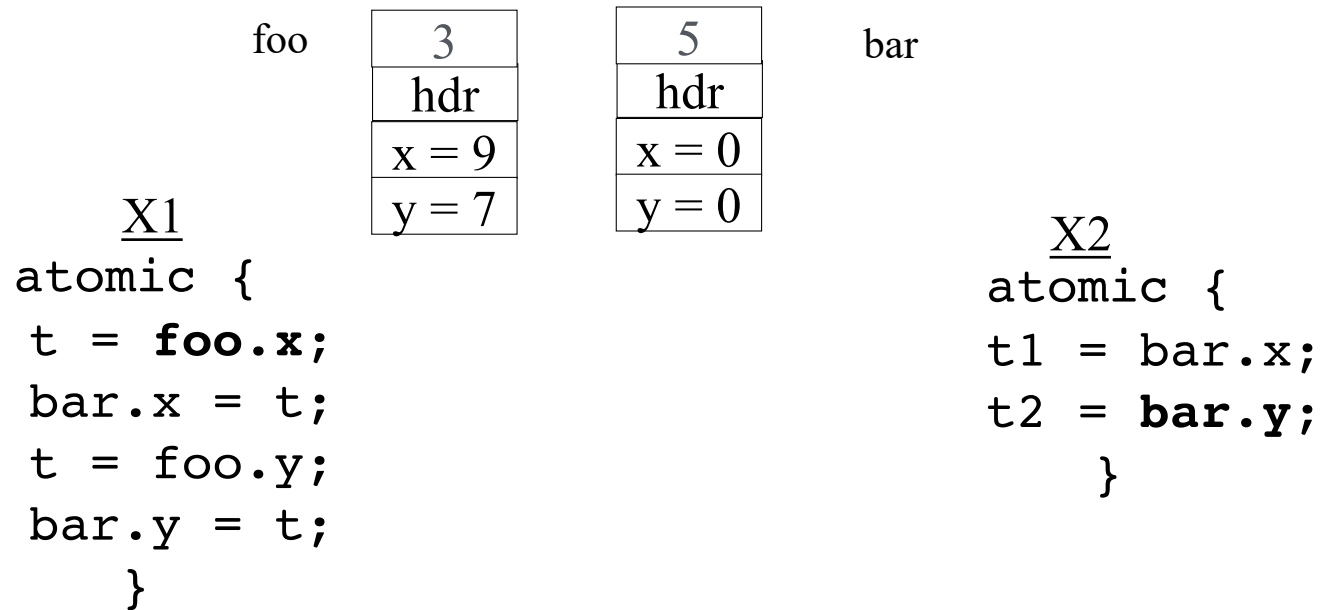
- **STM write (pessimistic)**
 - **Validate data (check for conflicts)**
 - **Check if unlocked and data version \leq local timestamp**
 - **Acquire lock**
 - **Insert in write set**
 - **Create undo log entry**
 - **Write data in place (eager)**

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**

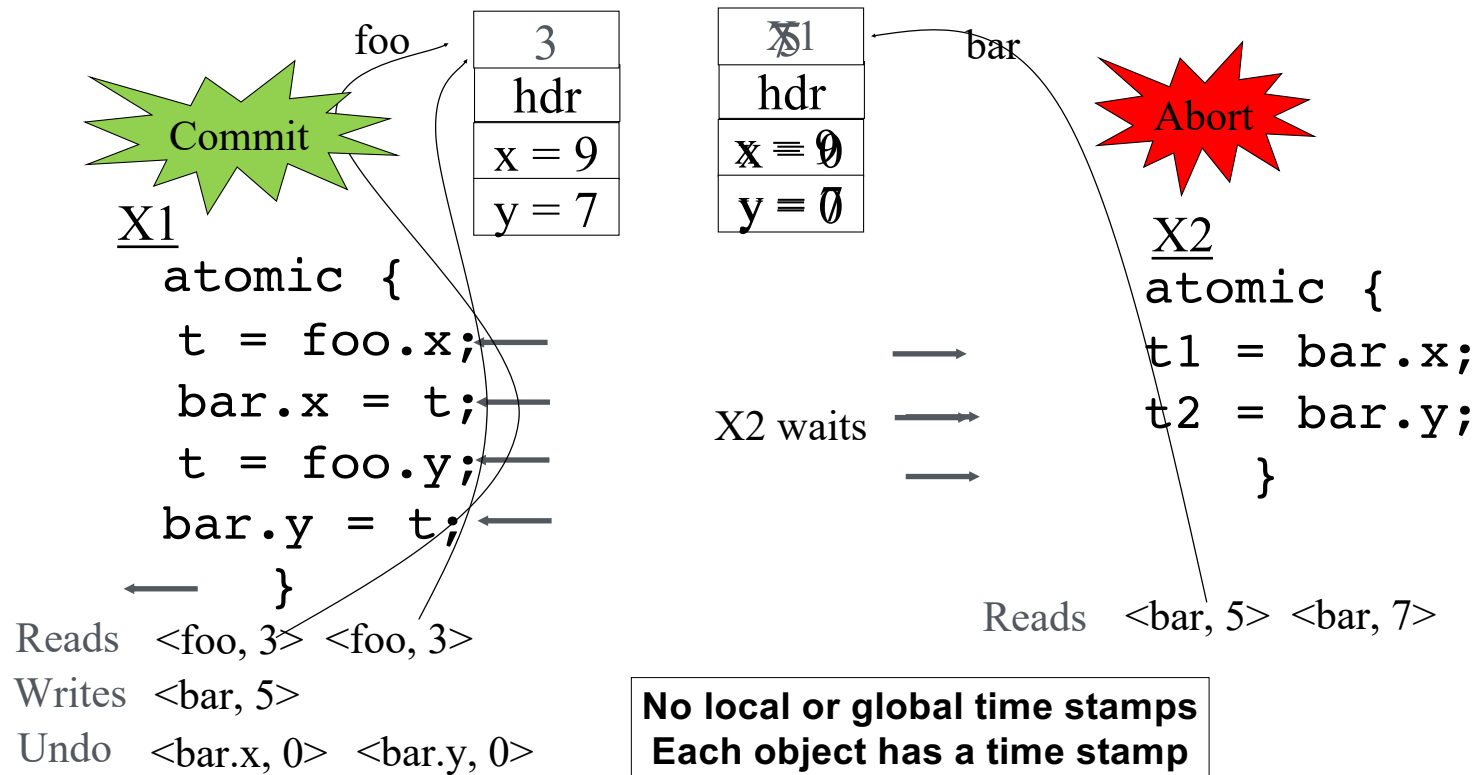
- **STM commit**
 - **Atomically increment global timestamp by 2 (LSb used for write-lock)**
 - **If preincremented (old) global timestamp > local timestamp, validate read-set**
 - **Check for recently committed transactions**
 - **For each item in the write set**
 - **Release the lock and set version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

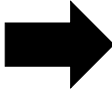


Challenges for STM Systems

- **Overhead of software barriers**
- **Function cloning**
- **Robust contention management**
- **Memory model (strong Vs. weak atomicity)**

Optimizing Software Transactions

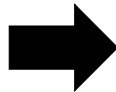
```
atomic {  
    a.x = t1  
    a.y = t2  
    if (a.z == 0) {  
        a.x = 0  
        a.z = t3  
    }  
}  
  
tmTxnBegin()  
tmWr(&a.x, t1)  
tmWr(&a.y, t2)  
if (tmRd(&a.z) != 0) {  
    tmWr(&a.x, 0);  
    tmWr(&a.z, t3)  
}  
tmTxnCommit()
```



- Monolithic barriers hide redundant logging & locking from the compiler

Optimizing Software Transactions

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

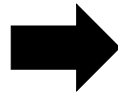


- Decomposed barriers expose redundancies

```
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  
}
```

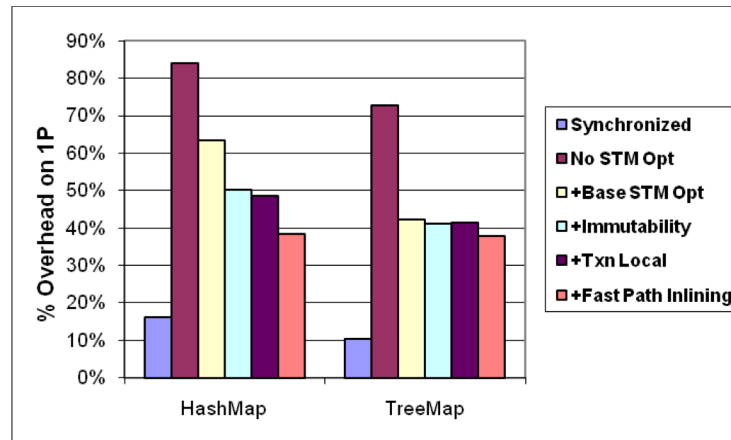
- Allows compiler to optimize STM code
- Produces fewer & cheaper STM operations

Compiler Optimizations for STM

- **Standard compiler optimizations**
 - CSE, PRE, dead-code elimination, ...
 - Assuming IR supports TM, few compiler mods needed
- **STM-specific optimizations**
 - Partial inlining of barrier fast paths
 - Often written in optimized assembly
 - **No barriers for immutable and transaction local data**
- **Impediments to optimizations**
 - Support for nested transactions
 - Dynamically linked STM library
 - Dynamic tuning of STM algorithm

Effect of Compiler Optimizations

- 1 thread overheads over thread-unsafe baseline



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

Function Cloning

- **Problem: need two version of functions**
 - **One with and one without STM instrumentation**
- **Managed languages (Java, C#)**
 - **On demand cloning of methods using JIT**
- **Unmanaged languages (C, C++)**
 - **Allow programmer to mark TM and pure functions**
 - **TM functions should be cloned by compiler**
 - **Pure functions touch only transaction-local data**
 - **No need for clones**
 - **All other functions handled as irrevocable actions**
 - **Some overhead for checks and mode transitions**

STM Efficiency

- **Old question: what is the overhead of STM?**
 - 1.3x – 6x
- **New question: what is the performance of a well engineered system using STM vs. a well engineered system using fine-grained locks?**
 - **Use STM aware data structures**
 - **They don't rely on STM, but play nicely with STM**
 - **Use STM to compose these data structures**
 - **Nathan Bronson**

Transactional Predication: High-Performance Concurrent Sets and Maps for STM

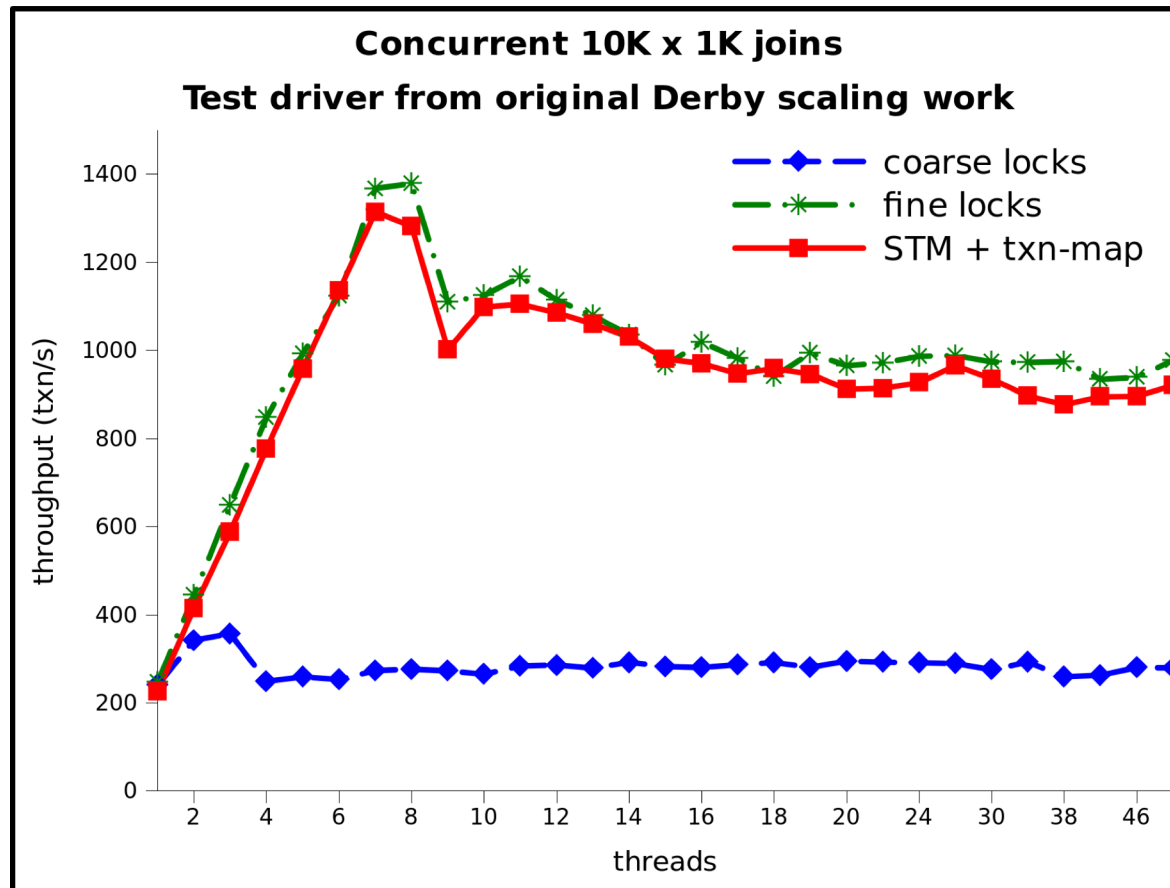
Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun

PODC '10: Proceedings of the 29th Annual ACM Conference on Principles of Distributed Computing, July 2010.

Lock manager inside Apache's Derby SQL Database

- **Row-level locks**
 - Multiple lock modes
 - Tricky conflict and queue logic
 - Automatic deadlock cycle detection
 - Per-row, per-txn, and per-group operations
- **Using ConcurrentHashMap + fine-grained locks**
 - 2204 non-comment lines of Java
 - 128 lines of discussion to prove that new code is thread safe!
 - Informal proof that the deadlock detector is not itself subject to deadlock!
- **Using STM + HashTrieTxnMaps**
 - 418 non-comment lines of Scala
 - A number of corner cases avoided (races, timeouts, etc.)

Performance Comparison (Read Heavy)

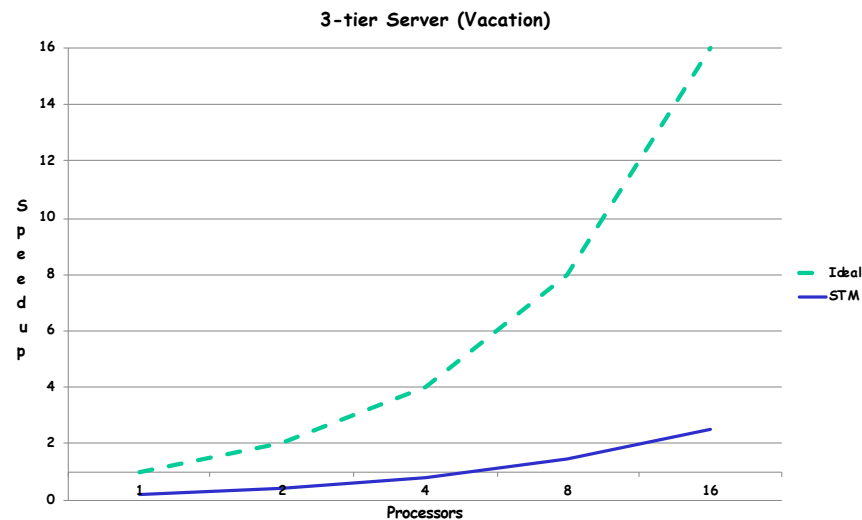


TM Implementation Summary 1

- **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)**

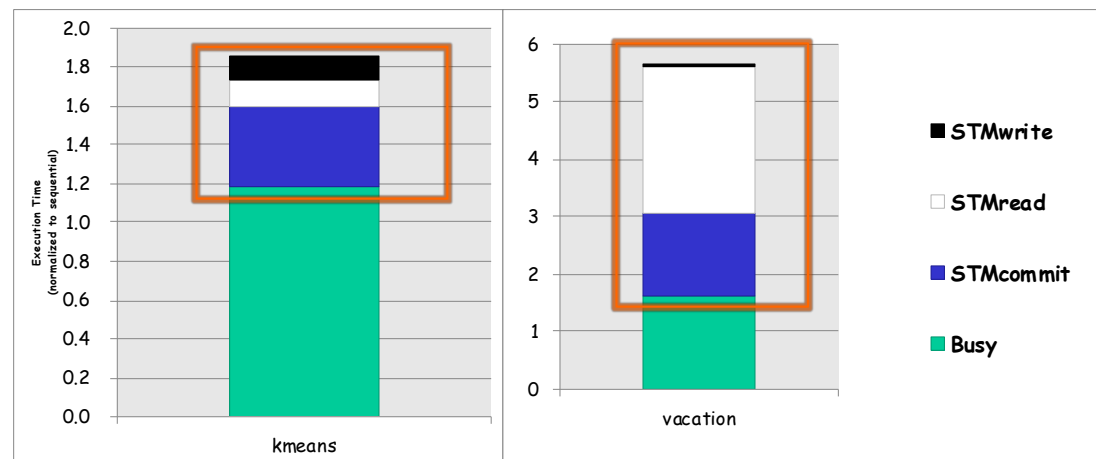
Motivation for Hardware Support



- **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



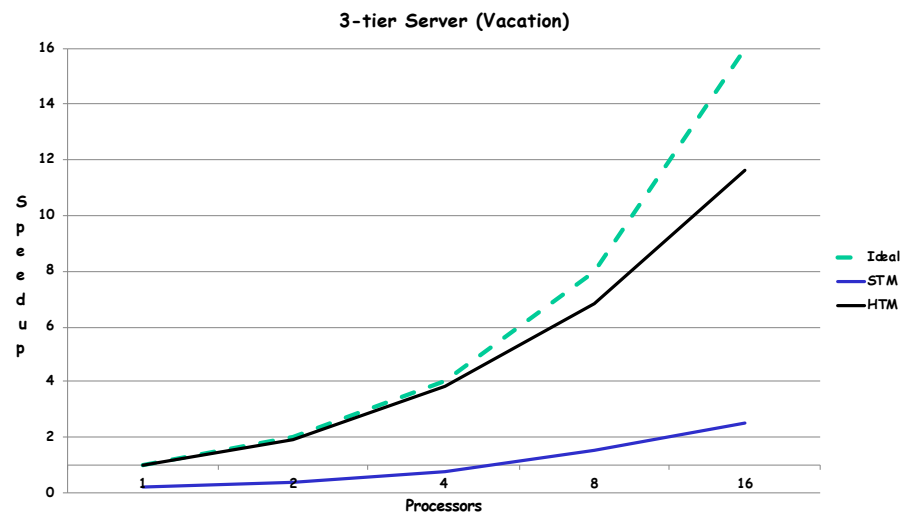
- 1.8x – 5.6x slowdown over sequential
- Most time goes in read barriers & validation
 - Most apps read more data than they write

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

HTM Performance Example



- 2x to 7x over STM performance
 - Within 10% of sequential for one thread
 - Scales efficiently with number of processors
 - Uncommon cases not a performance challenge

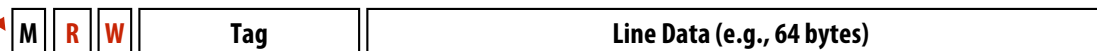
Hardware transactional memory (HTM)

- **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

MESI state bit for line (e.g., M state)

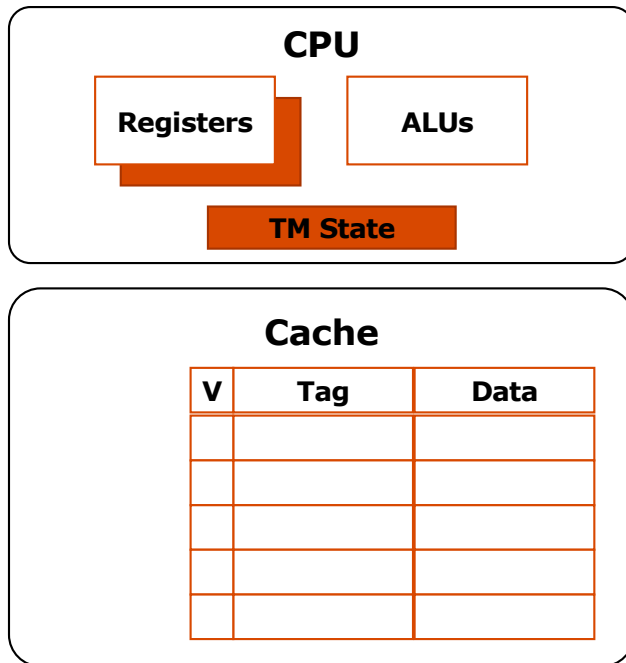


This illustration tracks read and write set at cache line granularity

Bits to track whether line is in read/write set of pending transaction

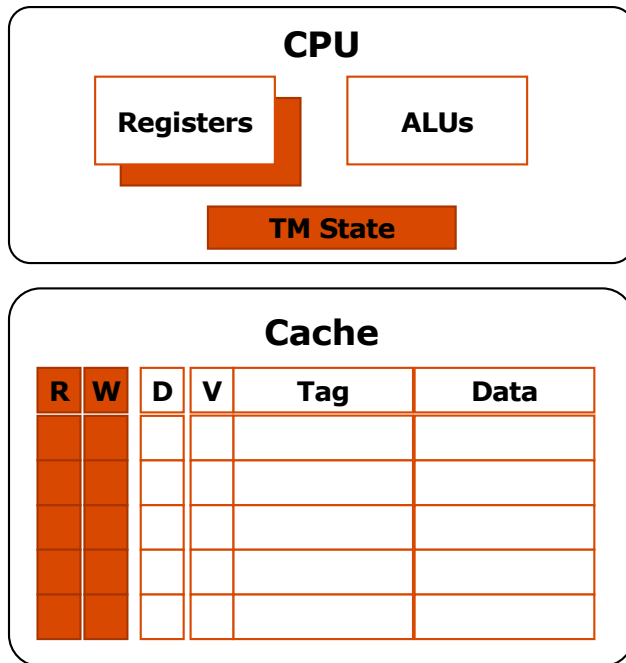
- For eager versioning, need a 2nd cache write for undo log
- **Coherence requests check R/W bits to detect conflicts**
 - Observing shared request to W-word is a read-write conflict
 - Observing exclusive (intent to write) request to R-word is a write-read conflict
 - Observing exclusive (intent to write) request to W-word is a write-write conflict

Example HTM implementation: lazy-optimistic



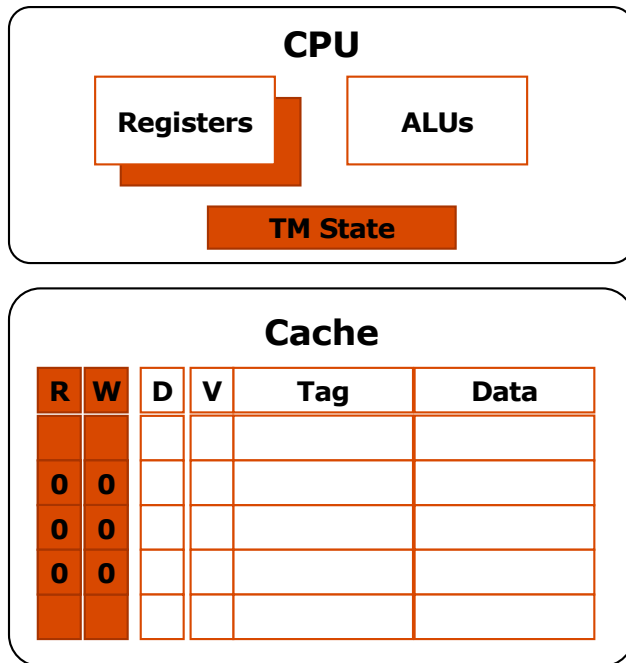
- **CPU changes**
 - Ability to checkpoint register state (available in many CPUs)
 - TM state registers (status, pointers to abort handlers, ...)

Example HTM implementation: lazy-optimistic



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

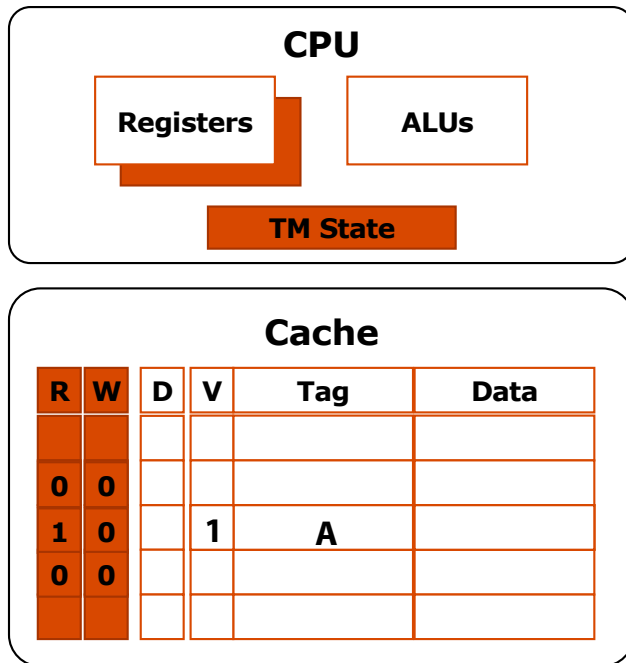
HTM transaction execution



Xbegin ←
Load A
Load B
Store C ← 5
Xcommit

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

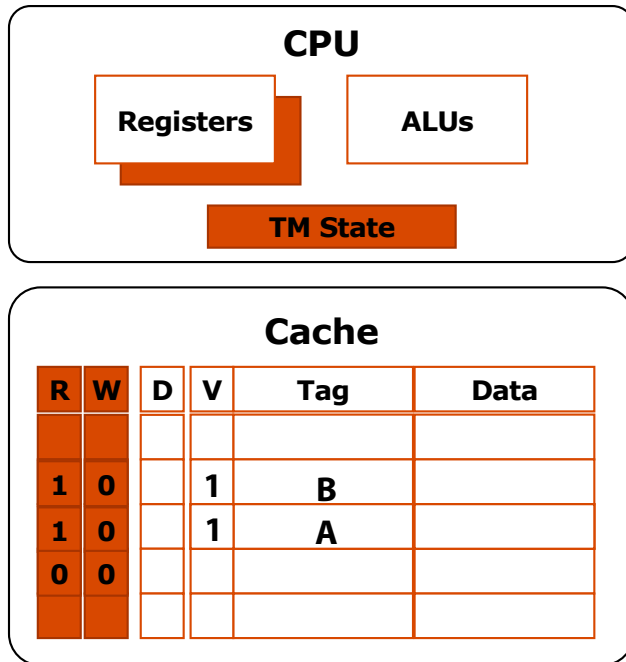
HTM transaction execution



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 ←

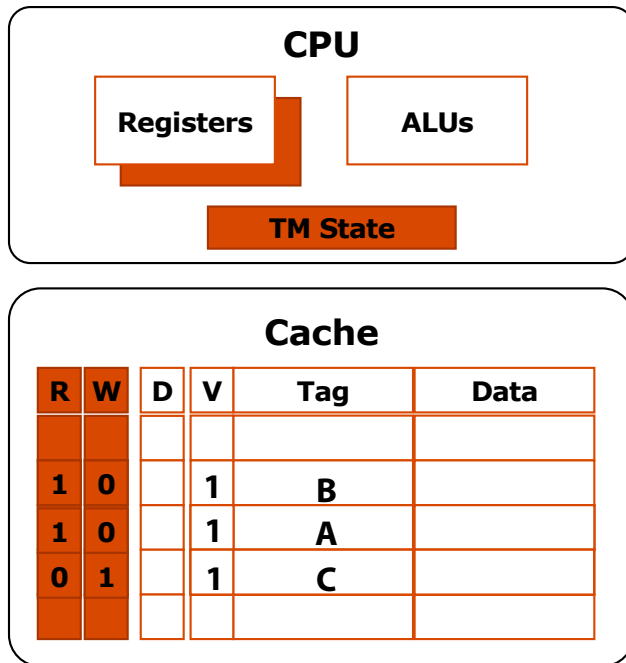
Store C ← 5

Xcommit

■ Load operation

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

HTM transaction execution



Xbegin

Load A

Load B

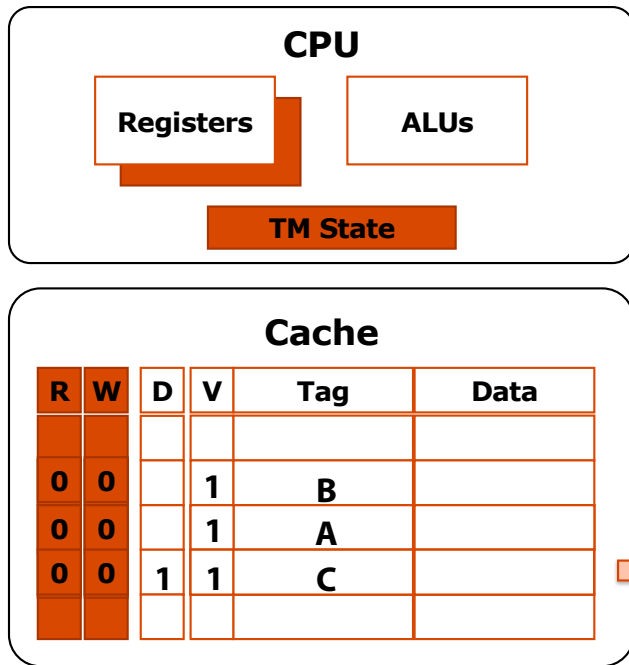
Store C \leftarrow 5 

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



Xbegin
Load A
Load B
Store C \leftarrow 5
Xcommit \leftarrow

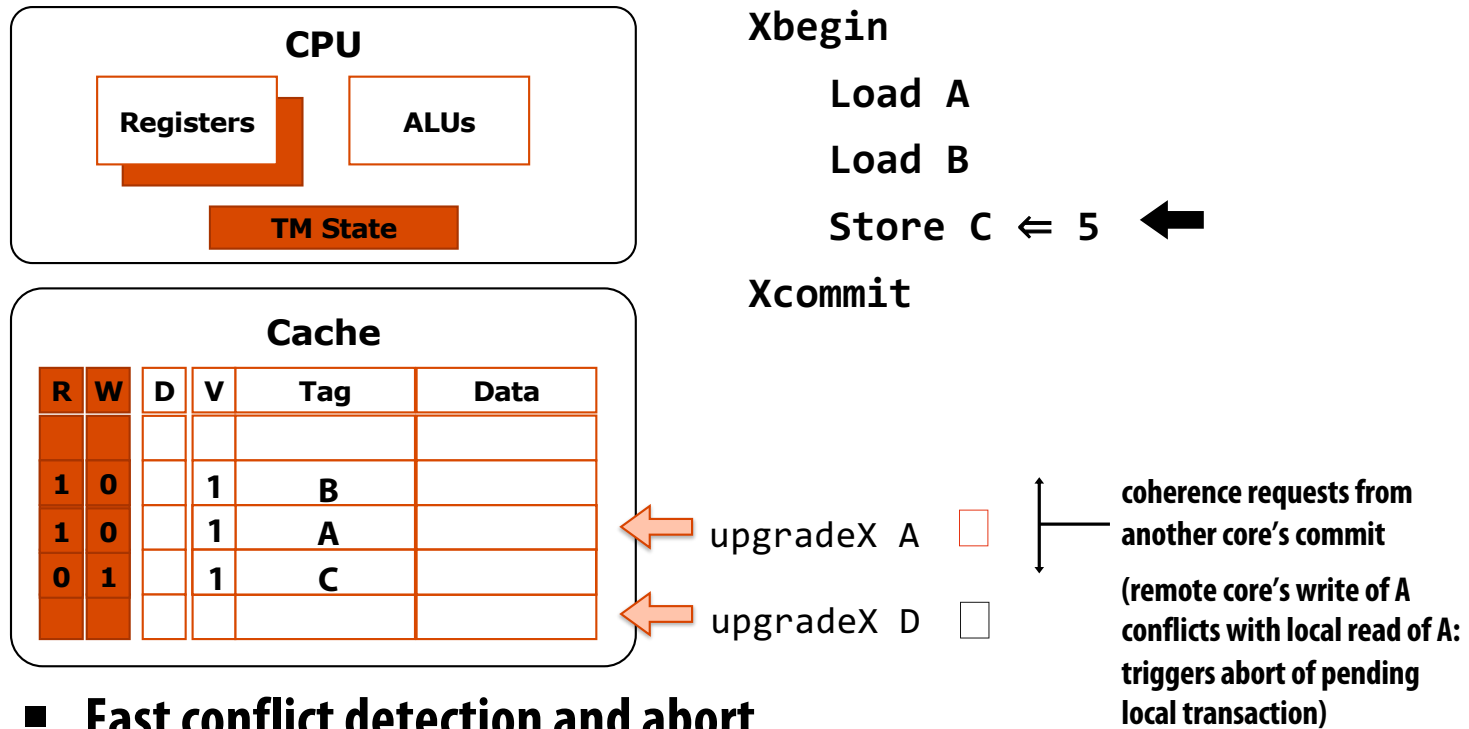
\rightarrow upgradeX C
(result: C is now in dirty state)

■ 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



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

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**
- **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**

I want to begin this part of the 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`.

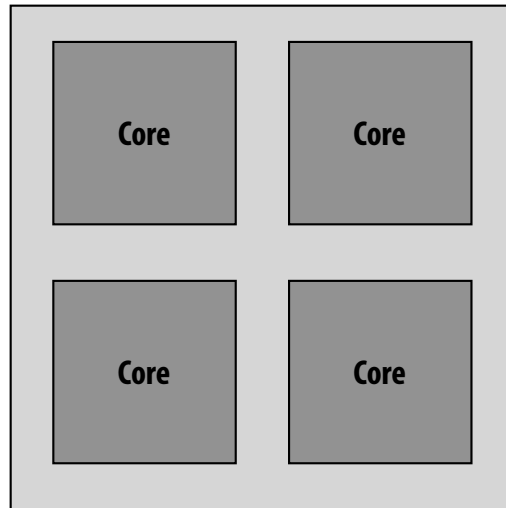
(In other words, a lot of software makes inefficient use of modern CPUs.)

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.



**You need to buy a
new computer...**

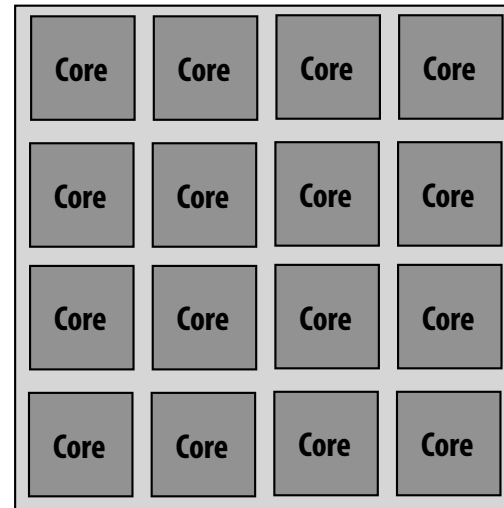
You need to buy a computer system



Processor A

4 cores

Each core has sequential performance P



Processor B

16 cores

Each core has sequential performance $P/2$

All other components of the system are equal.

Which do you pick?

Recall Amdahl's law

$$\text{speedup}(f, n) = \frac{1}{(1 - f) + \frac{f}{n}}$$

f = fraction of program that is parallelizable

n = parallel processors

Assumptions:

Parallelizable work distributes perfectly onto *n* processors of equal capability

Rewrite Amdahl's law in terms of resource limits

$$\text{speedup}(f, n, r) = \frac{1}{\frac{1-f}{\text{perf}(r)} + \frac{f}{\text{perf}(r) \cdot \frac{n}{r}}}$$

Speedup relative to processor with 1 unit of resources, $n=1$

Assume $\text{perf}(1) = 1$

f = fraction of program that is parallelizable

n = total processing resources (e.g., transistors on a chip)

r = resources dedicated to each processing core,

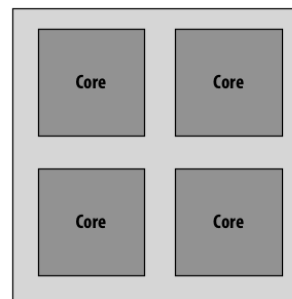
(each of the n/r cores has sequential performance $\text{perf}(r)$)

More general form of Amdahl's Law in terms of f, n, r

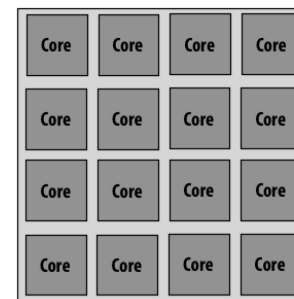
Two examples where $n=16$

$r_A = 4$

$r_B = 1$

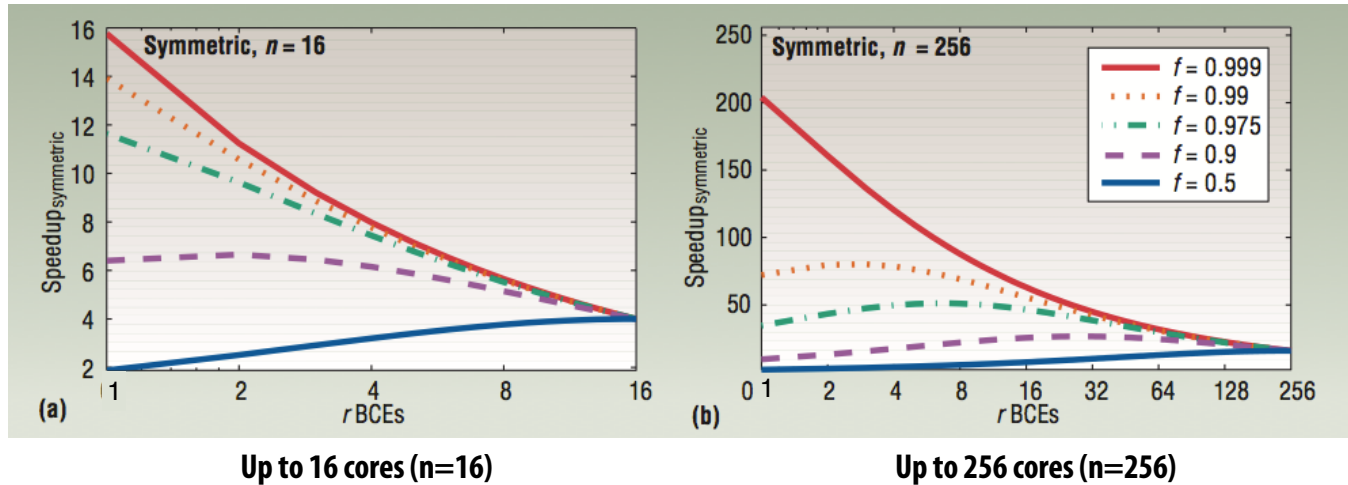


Processor A



Processor B

Speedup (relative to n=1)

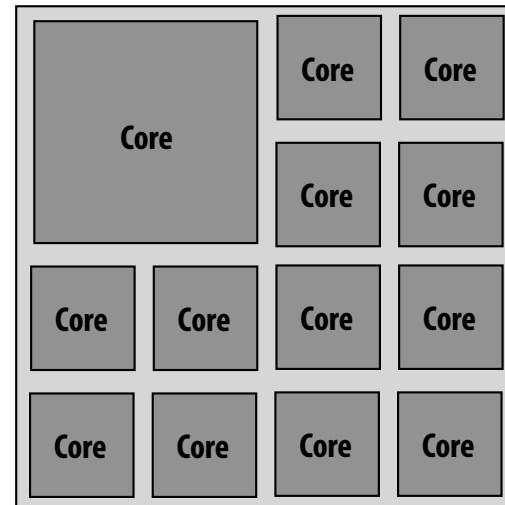


X-axis = r (chip with many small cores to left, fewer “fatter” cores to right)
Each line corresponds to a different workload
Each graph plots performance as resource allocation changes, but total chip resources are kept the same (constant n per graph)

perf(r) modeled as \sqrt{r}

Asymmetric set of processing cores

Example: $n=16$
 One core: $r = 4$
 Other 12 cores: $r = 1$



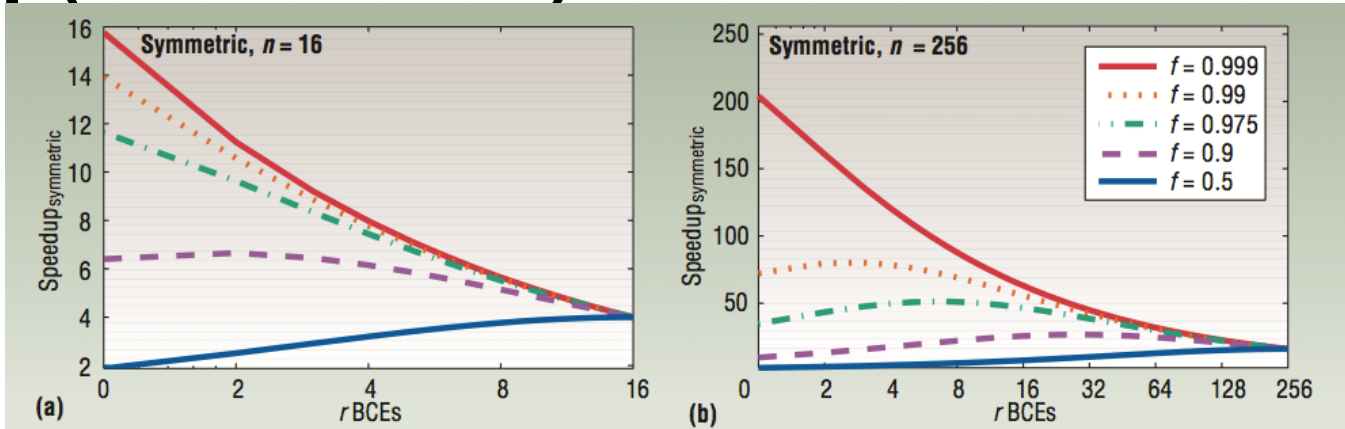
$$\text{speedup}(f, n, r) = \frac{1}{\frac{1-f}{\text{perf}(r)} + \frac{f}{\text{perf}(r) + (n-r)}}$$

(of heterogeneous processor with n resources, relative to uniprocessor with one unit worth of resources, $n=1$)

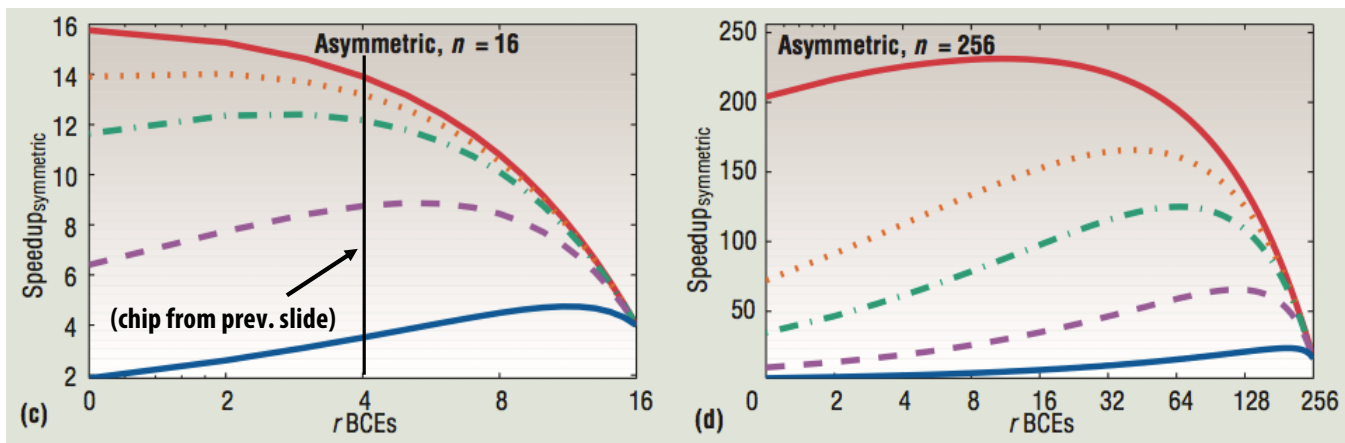
← →
one $\text{perf}(r)$ processor + $(n-r)$ $\text{perf}(1)=1$ processors

Speedup (relative to $n=1$)

[Source: Hill and Marty 08]



X-axis for symmetric architectures gives r for all cores (many small cores to left, few "fat" cores to right)



X-axis for asymmetric architectures gives r for the single "fat" core (assume rest of cores are $r = 1$)

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

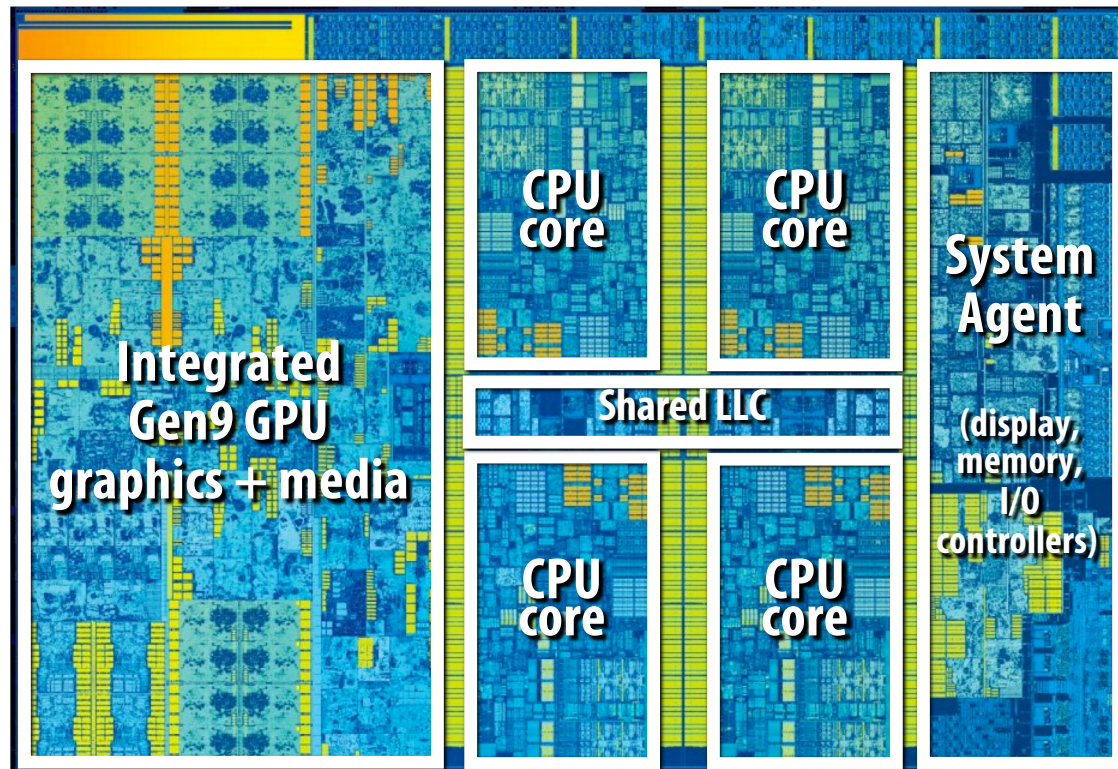
And components with unpredictable access, but those accesses might cache well.

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)

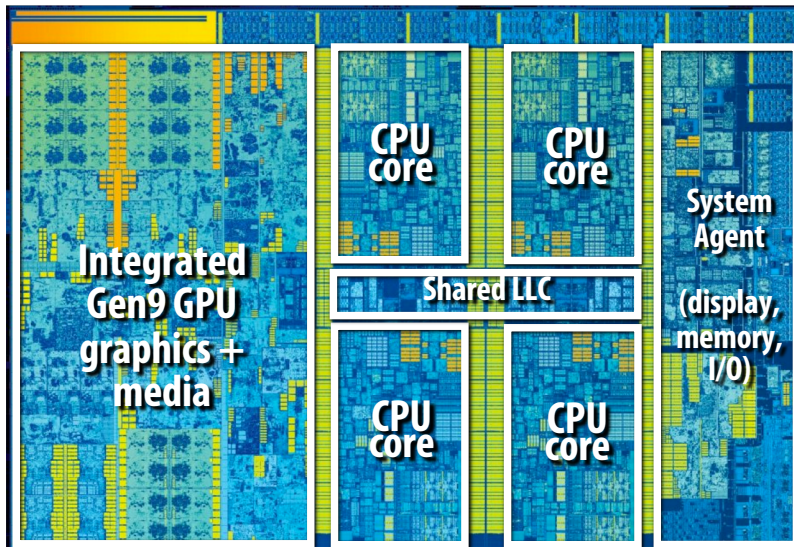
(6th Generation Core i7 architecture)



4 CPU cores + graphics cores + media accelerators

Example: Intel "Skylake" (2015)

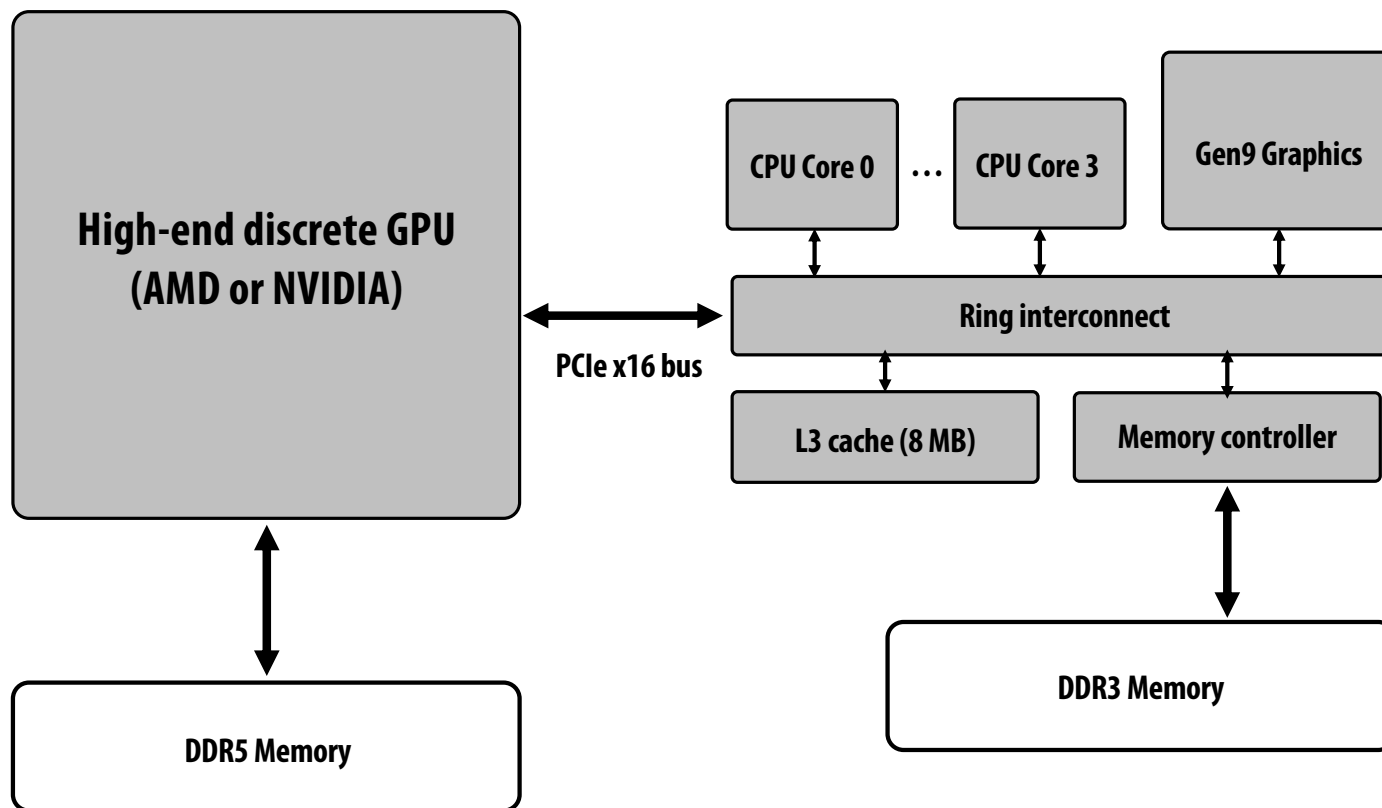
(6th Generation Core i7 architecture)



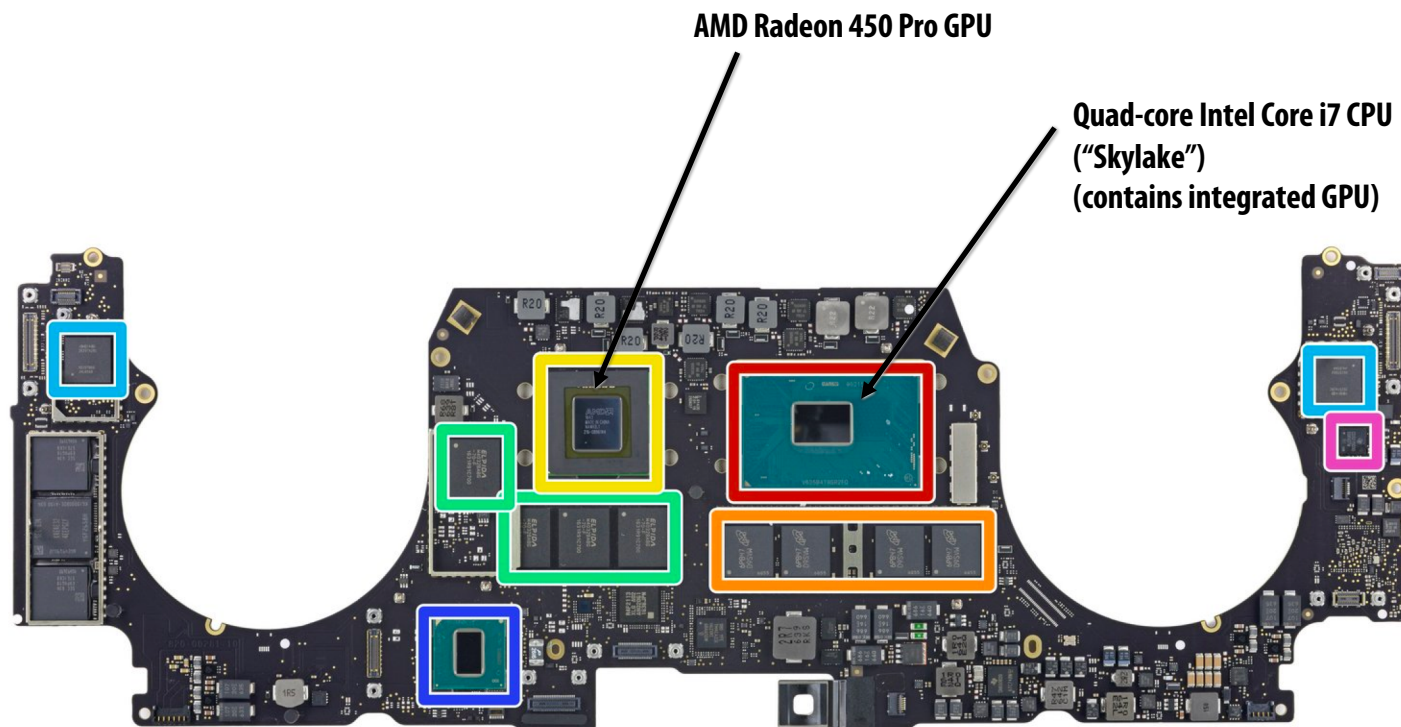
- CPU cores and graphics cores share same memory system
- Also share LLC (L3 cache)
 - Enables, low-latency, high-bandwidth communication between CPU and integrated GPU
- Graphics cores are cache coherent with CPU cores

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

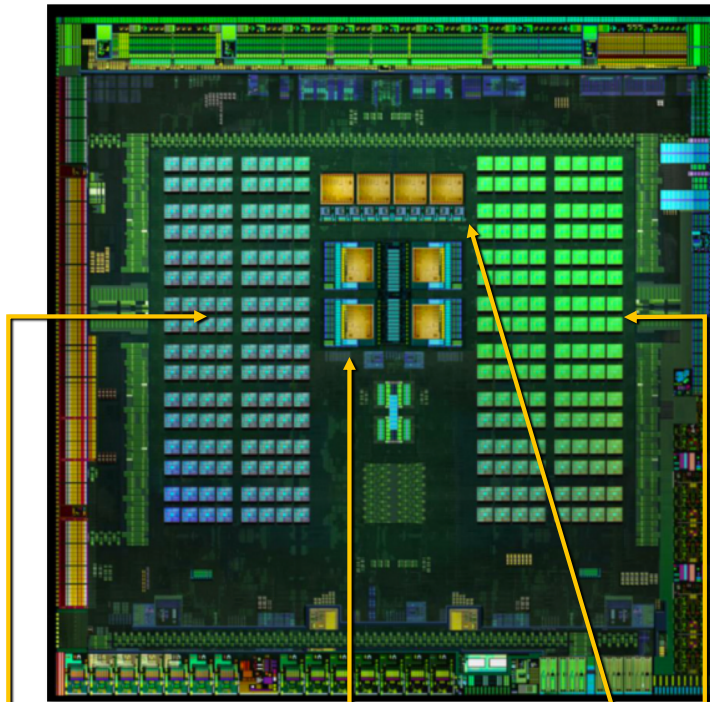


15in Macbook Pro /w Touch Bar (2016) (two GPUs)



From ifixit.com teardown

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)

A11 image credit: TechInsights Inc.'

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



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

Supercomputers use heterogeneous processing

Los Alamos National Laboratory: “Roadrunner”

Fastest US supercomputer in 2008, first to break Petaflop barrier: 1.7 PFLOPS

Unique at the time due to use of two types of processing elements

(IBM’s Cell processor served as “accelerator” to achieve desired compute density)

- 6,480 AMD Opteron dual-core CPUs (12,960 cores)
- 12,970 IBM Cell Processors (1 CPU + 8 accelerator cores per Cell = 116,640 cores)
- 2.4 MWatt (about 2,400 average US homes)



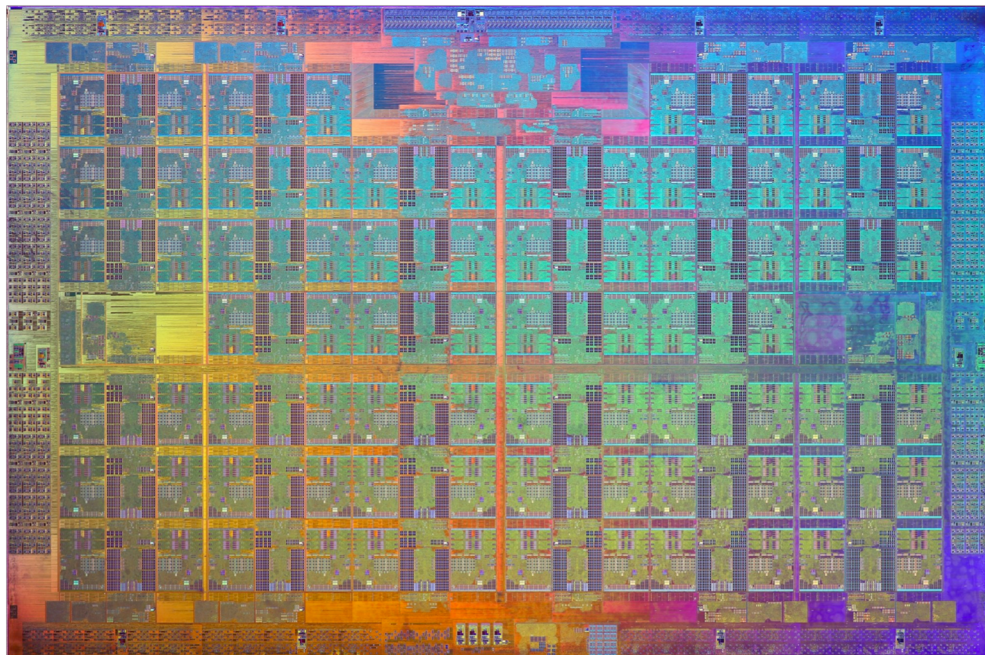
GPU-accelerated supercomputing



**Summit (at Oak Ridge National Lab)
(world's #1 in Fall 2018)
9,216 IBM Power9 22-core CPUs
27,648 NVIDIA V100 GPUs
10 Petabytes DRAM**

Intel Xeon Phi (Knights Landing)

- 72 “simple” x86 cores (1.1 Ghz, derived from Intel Atom)
- 16-wide vector instructions (AVX-512), four threads per core
- Targeted as an accelerator for supercomputing applications



Heterogeneous architectures for supercomputing

Source: Top500.org Fall 2018 rankings

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM DOE/SC/Oak Ridge National Laboratory United States	2,397,824	143,500.0	200,794.9	9,783
2	Sierra - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
3	Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway , NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
4	Tianhe-2A - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 , NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
5	Piz Daint - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 Cray Inc. Swiss National Supercomputing Centre (CSCS) Switzerland	387,872	21,230.0	27,154.3	2,384
6	Trinity - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect , Cray Inc. DOE/NNSA/LANL/SNL United States	979,072	20,158.7	41,461.2	7,578
7	AI Bridging Cloud Infrastructure (ABCI) - PRIMERGY CX2570 M4, Xeon Gold 6148 20C 2.4GHz, NVIDIA Tesla V100 SXM2, Infiniband EDR , Fujitsu National Institute of Advanced Industrial Science and Technology (AIST) Japan	391,680	19,880.0	32,576.6	1,649
8	SuperMUC-NG - ThinkSystem SD530, Xeon Platinum 8174 24C 3.1GHz, Intel Omni-Path , Lenovo Leibniz Rechenzentrum Germany	305,856	19,476.6	26,873.9	

201 Petaflops (peak),
143 Petaflops (effective)
9.7 MWatt
(14.6 GFLOPS/W)

Xeon Phi

GPU

Green500: most energy efficient supercomputers

Efficiency metric: effective GFLOPS per Watt

TOP500			Cores	Rmax (TFlop/s)	Power (kW)	Power Efficiency (GFlops/watts)
Rank	Rank	System				
1	375	Shoubu system B - ZettaScaler-2.2, Xeon D-1571 16C 1.3GHz, Infiniband EDR, PEZY-SC2 , PEZY Computing / Exascaler Inc. Advanced Center for Computing and Communication, RIKEN Japan	953,280	1,063.3	60	17.604
2	374	DGX SaturnV Volta - NVIDIA DGX-1 Volta36 , Xeon E5-2698v4 20C 2.2GHz, Infiniband EDR, NVIDIA Tesla V100 , Nvidia NVIDIA Corporation United States	22,440	1,070.0	97	15.113
3	1	Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100 , Dual-rail Mellanox EDR Infiniband , IBM DOE/SC/Oak Ridge National Laboratory United States	2,397,824	143,500.0	9,783	14.668
4	7	AI Bridging Cloud Infrastructure (ABCI) - PRIMERGY CX2570 M4, Xeon Gold 6148 20C 2.4GHz , NVIDIA Tesla V100 SXM2 , Infiniband EDR , Fujitsu National Institute of Advanced Industrial Science and Technology (AIST) Japan	391,680	19,880.0	1,649	14.423
5	22	TSUBAME3.0 - SGI ICE XA, IP139-SXM2, Xeon E5-2680v4 14C 2.4GHz, Intel Omni-Path, NVIDIA Tesla P100 SXM2 , HPE GSIC Center, Tokyo Institute of Technology Japan	135,828	8,125.0	792	13.704
6	2	Sierra - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100 , Dual-rail Mellanox EDR Infiniband , IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	7,438	12.723

Source: Green500 Fall 2018 rankings

Energy-constrained computing

- **Supercomputers are energy constrained**
 - Due to sheer 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

Energy-constrained computing

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**

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

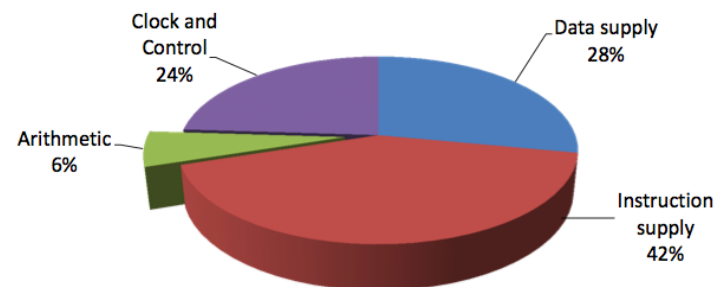
Move data from execution resource to register file

Use decoded operands to control write to register file SRAM

Review question:

How does SIMD execution reduce overhead of certain types of computations?

What properties must these computations have?

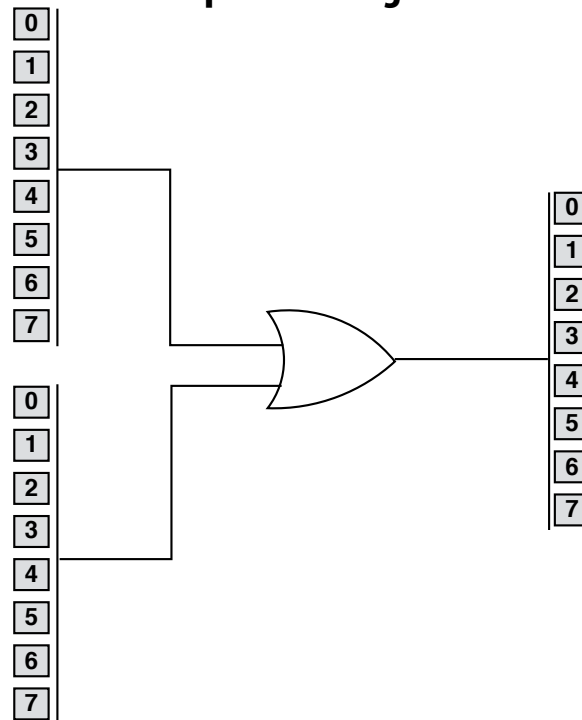


Efficient Embedded Computing [Dally et al. 08]

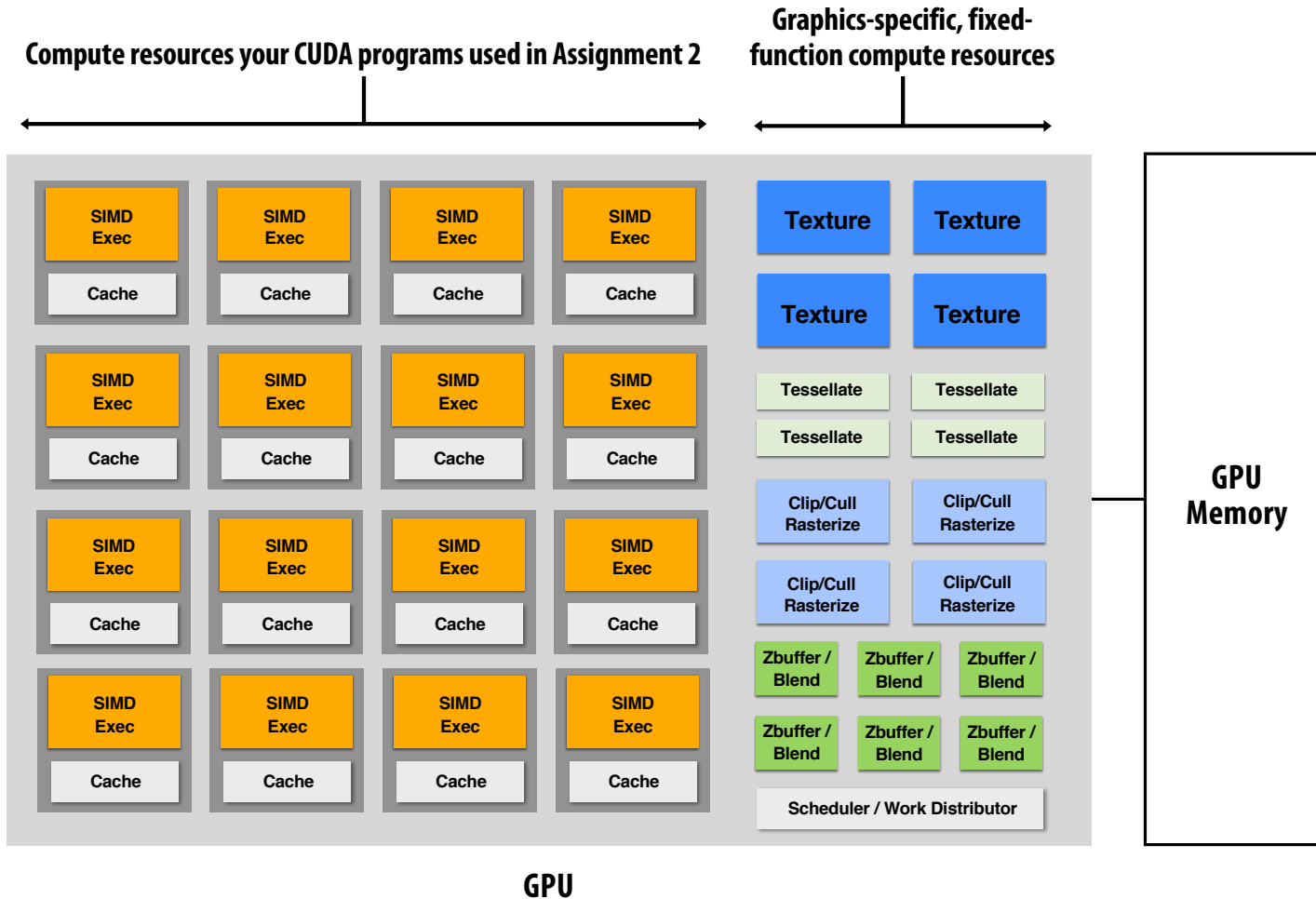
[Figure credit Eric Chung]

Contrast that complexity to the circuit required to actually perform the operation

Example: 8-bit logical OR



GPU's are themselves heterogeneous multi-core processors



Neural Networks: Many Applications

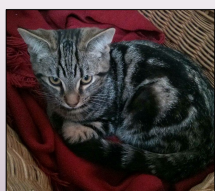
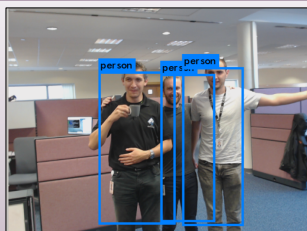
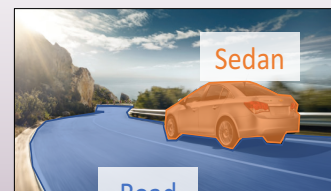


Image Classification



Object Detection



Semantic Segmentation

Computer Vision
CNNs



Speaker
Diarization

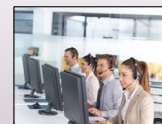


Speech
Recognition

Speech Recognition
RNNs, LSTMs



Translation



Sentiment Analysis

Natural Language Processing
Sequence to sequence



Recommender



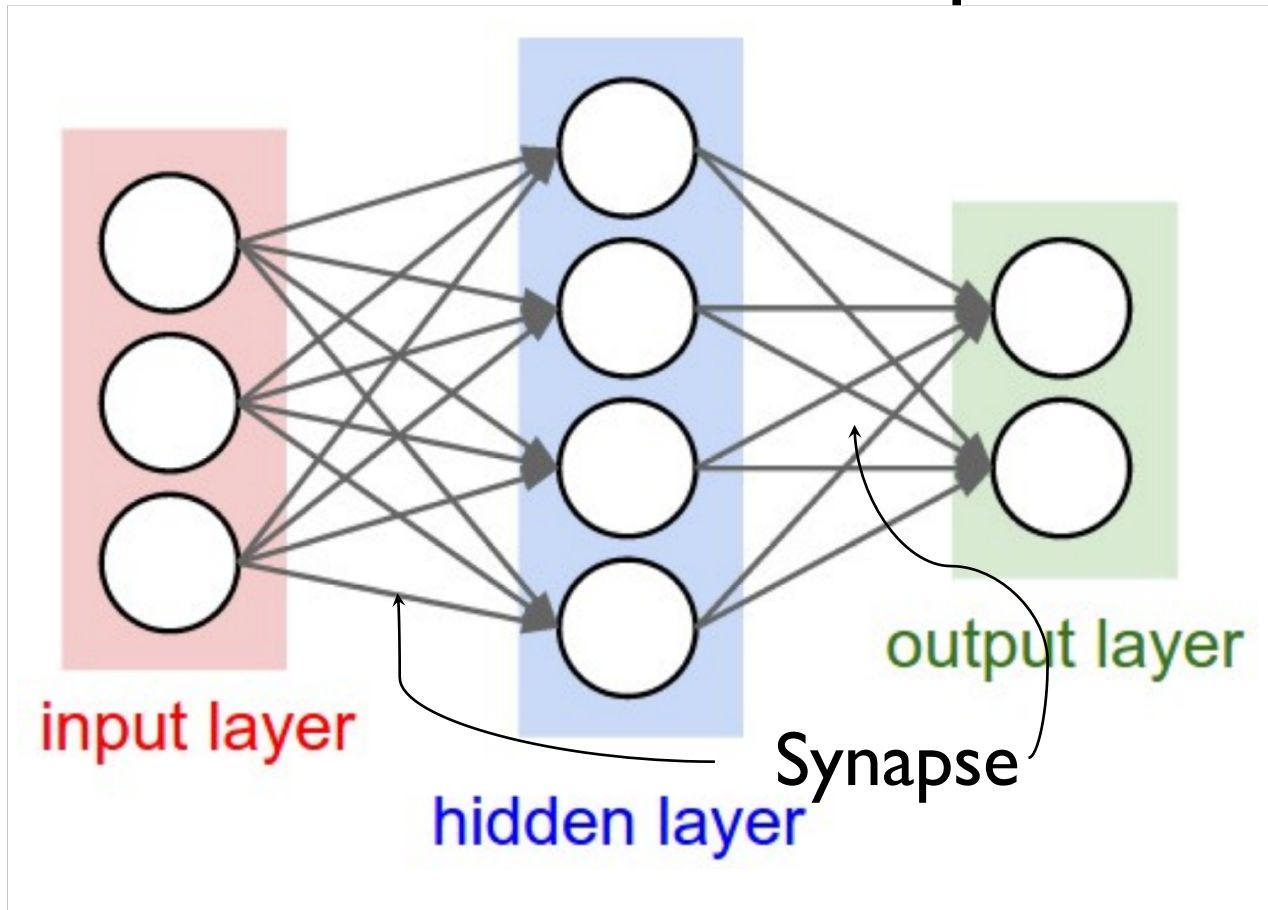
GamePlay

Many more emerging...

Others

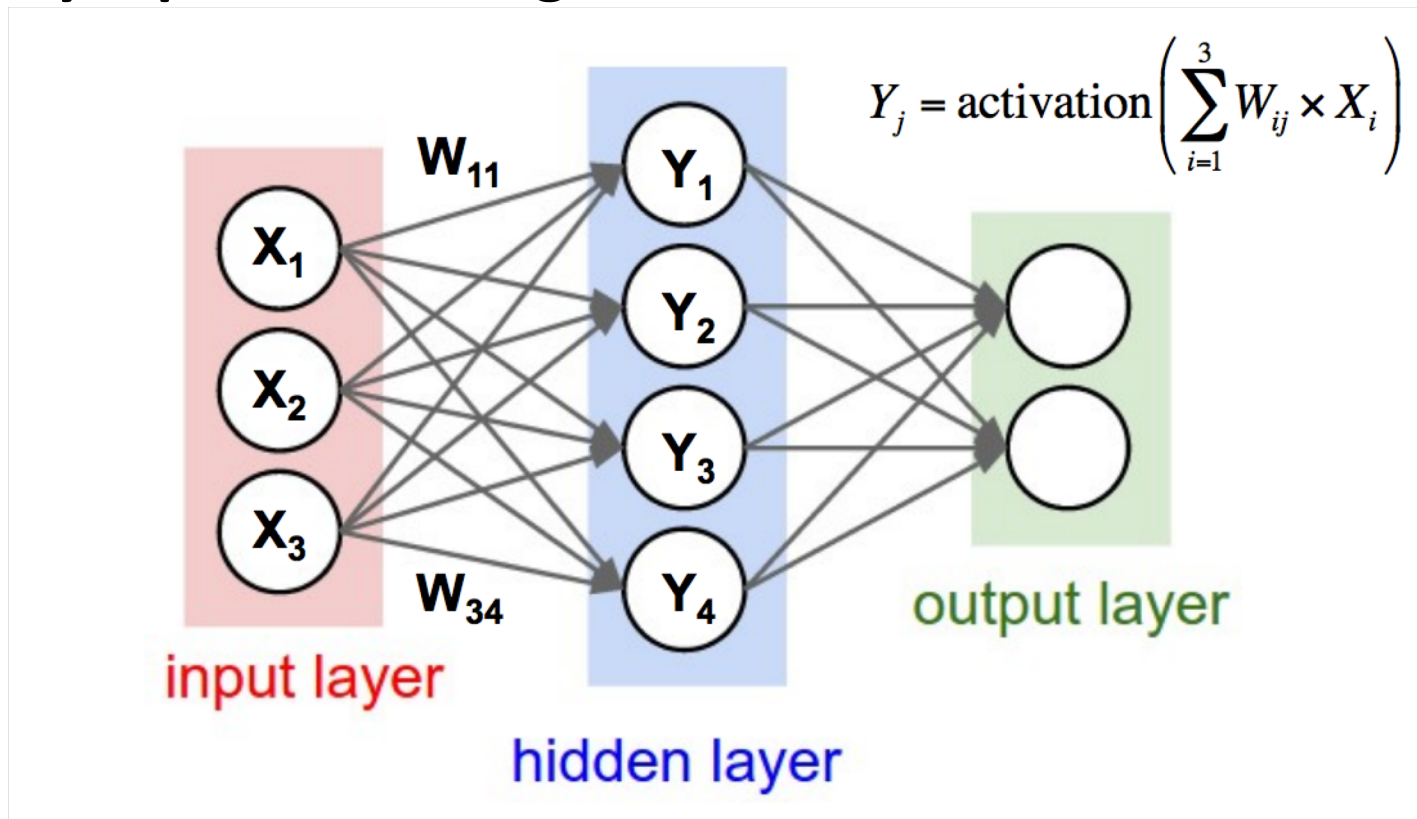
Overview of Deep Neural Networks

- A DNN emulates the behavior of multiple connected neurons



Overview of Deep Neural Networks

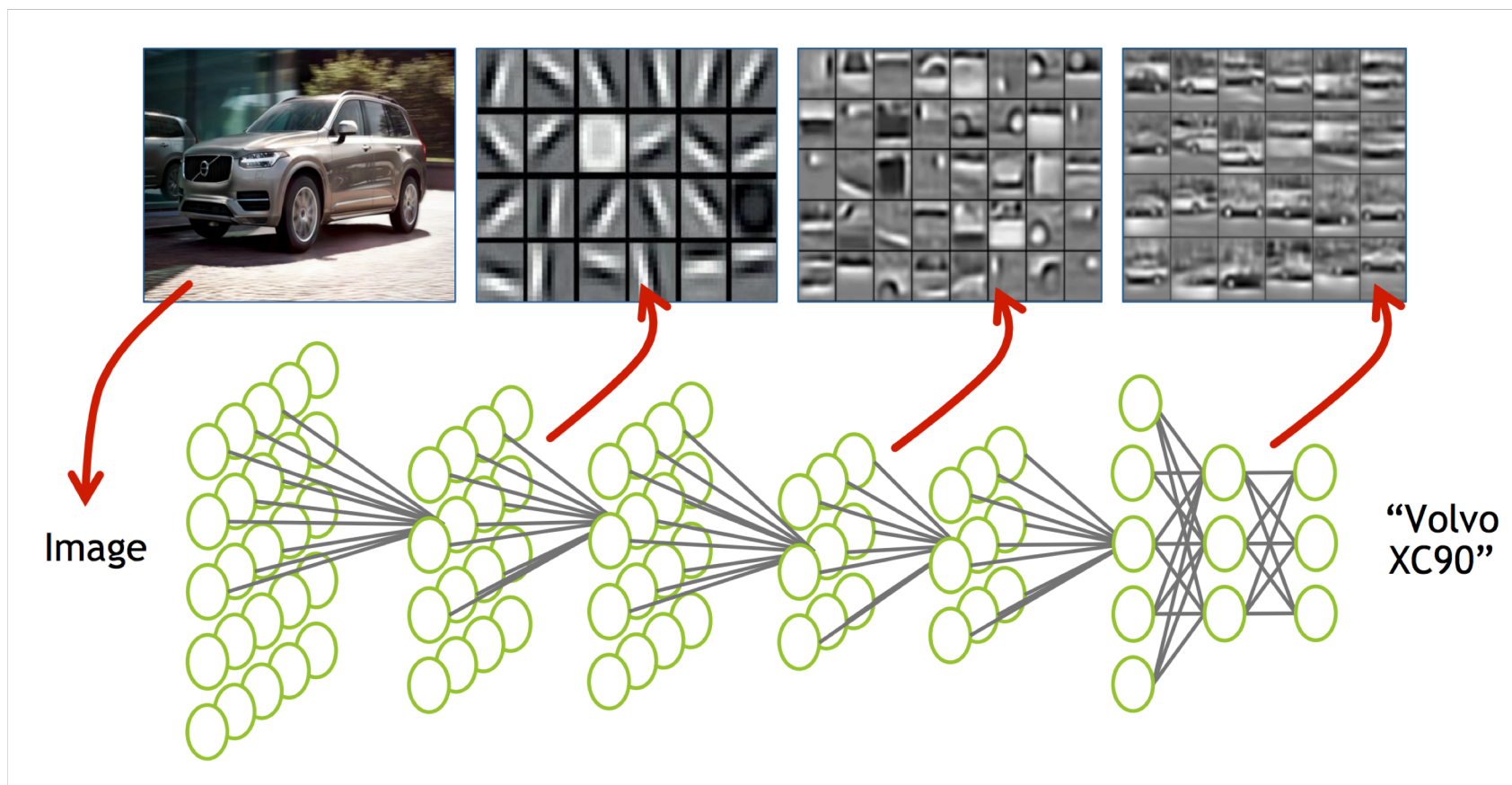
- Each synapse has a weight for neuron activation



Training and Inference

- **Training:**
 - Determine the values of the weights
- **Inference:**
 - Use the values of the weights, inputs, determine the outputs

Example: using DNN to recognize a car

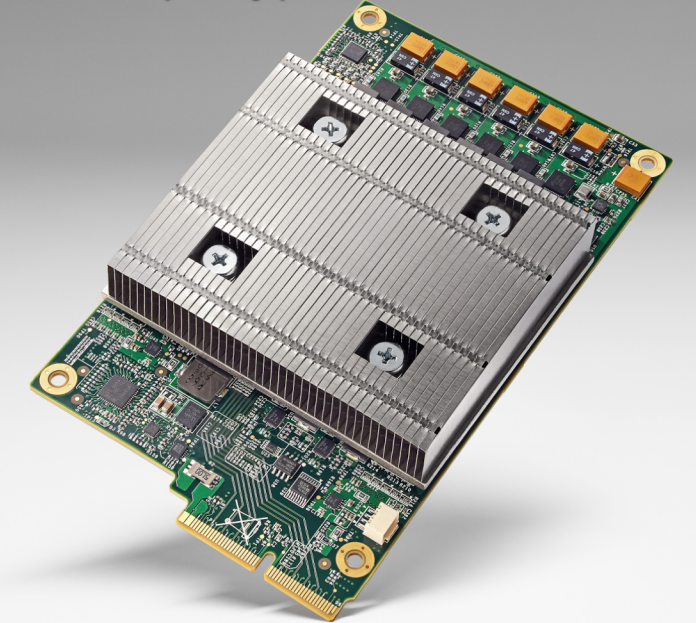


Specialized processors for evaluating deep networks

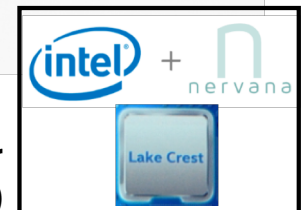
Countless recent papers at top computer architecture research conferences on the topic of ASICs or accelerators for deep learning or evaluating deep networks...

- **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

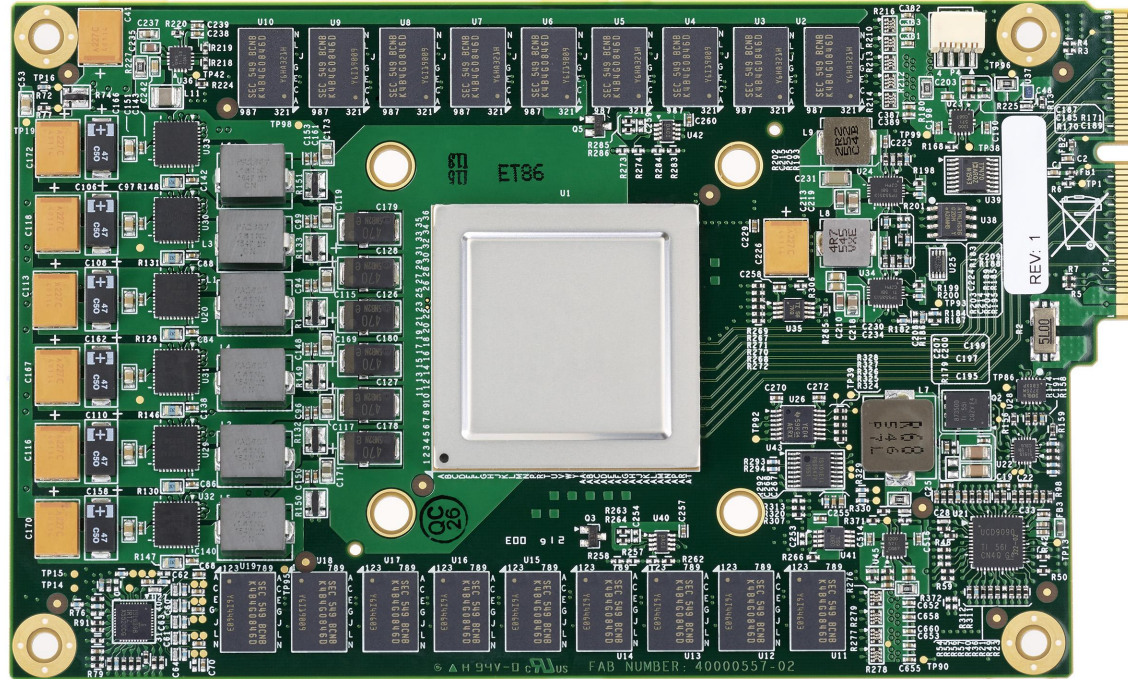
Example: Google's Tensor Processing Unit (TPU)
Accelerates deep learning operations



Intel Lake Crest ML accelerator
(formerly Nervana)



TPU Card

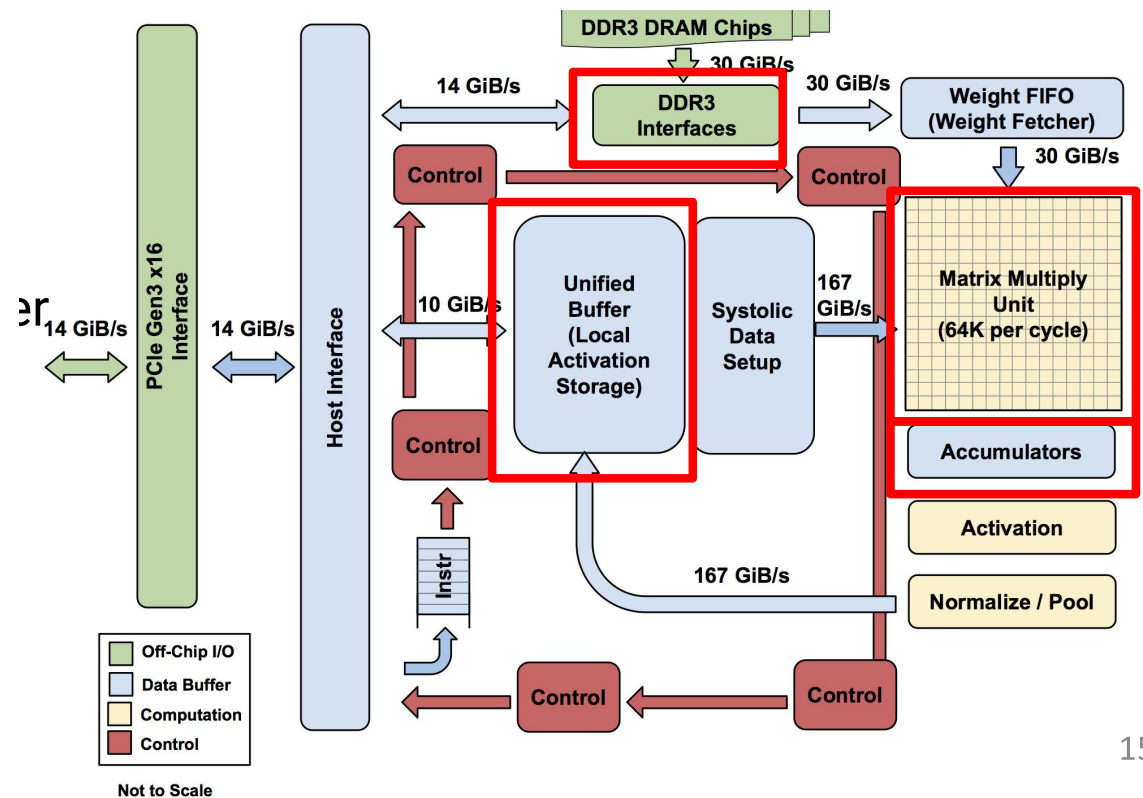


Up to 4 cards per server

8

TPU High-level Chip Architecture

- The Matrix Unit: 65,536 (256x256) 8-bit multiply-accumulate units
 - Systolic array
- 700 MHz clock rate
- Peak: 92T operations/second
 - $65,536 * 2 * 700M$
- 4 MB of on-chip Accumulator memory
- 24 MB of on-chip Unified Buffer (activation memory)
- Two 2133MHz DDR3 DRAM channels
- 8 GB of off-chip weight DRAM memory
- vs GPU and CPU
 - **>25X as many MACs vs GPU**
 - **>100X as many MACs vs CPU**



TPU Programmers View

- **Five key CISC instructions (CPI > 10)**

- Read_Host_Memory

- Write_Host_Memory

- Read_Weights

- MatrixMultiply/Convolve

- Activate (ReLU, Sigmoid, Maxpool, LRN,...)

- **Complexity in software**

- **No branches**

- **In-order issue**

- **Software controlled buffers**

- **Software controlled pipeline synchronization**

Three Types of NNs

1. Multilayer Perceptrons

- Each new layer applies nonlinear function F to weighted sum of all outputs from prior layer (“fully connected”) $x_n = F(Wx_{n-1})$

2. Convolutional Neural Network

- Like MLPs, but same weights used on nearby subsets of outputs from prior layer

3. Recurrent NN/“Long Short-Term Memory”

- Each new layer a NL function of weighted sums of past *state* and prior outputs; same weights used across time steps

2016 NN Datacenter Workload

Name	LOC	Layers					Nonlinear function	Weights	TPU Ops / Weight Byte	TPU Batch Size	% Deployed
		FC	Conv	Vector	Pool	Total					
MLP0	0.1k	5				5	ReLU	20M	200	200	61%
MLP1	1k	4				4	ReLU	5M	168	168	
LSTM0	1k	24		34		58	sigmoid, tanh	52M	64	64	29%
LSTM1	1.5k	37		19		56	sigmoid, tanh	34M	96	96	
CNN0	1k		16			16	ReLU	8M	2888	8	5%
CNN1	1k	4	72		13	89	ReLU	100M	1750	32	

Three Contemporary Chips

<i>Processor</i>	<i>mm²</i>	<i>Clock MHz</i>	<i>TDP Watts</i>	<i>Idle Watts</i>	<i>Memory GB/sec</i>	<i>Peak TOPS/chip</i>	
						<i>8b int.</i>	<i>32b FP</i>
CPU: Haswell (18 core)	662	2300	145	41	51	2.6	1.3
GPU: Nvidia K80 (2 / card)	561	560	150	25	160	--	2.8
TPU	<331*	700	75	28	34	91.8	--

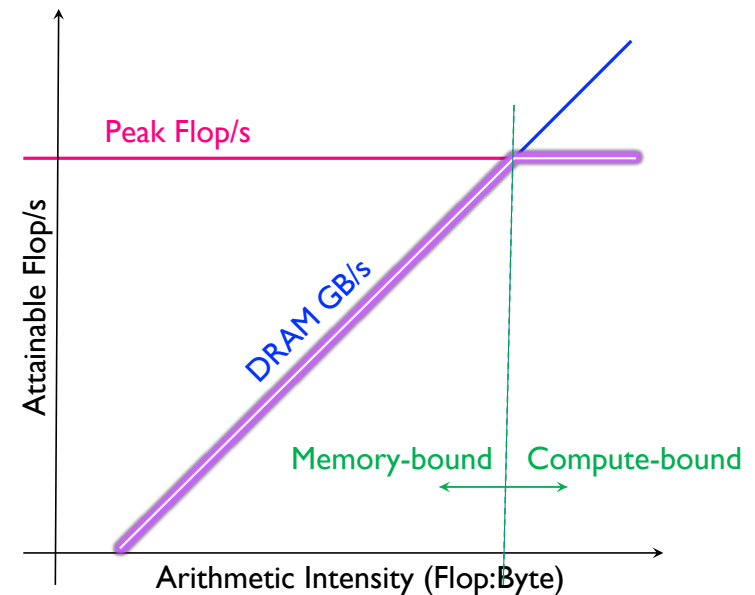
*TPU is less than half die size of the Intel Haswell processor

K80 and TPU in 28 nm process; Haswell fabbed in Intel 22 nm process

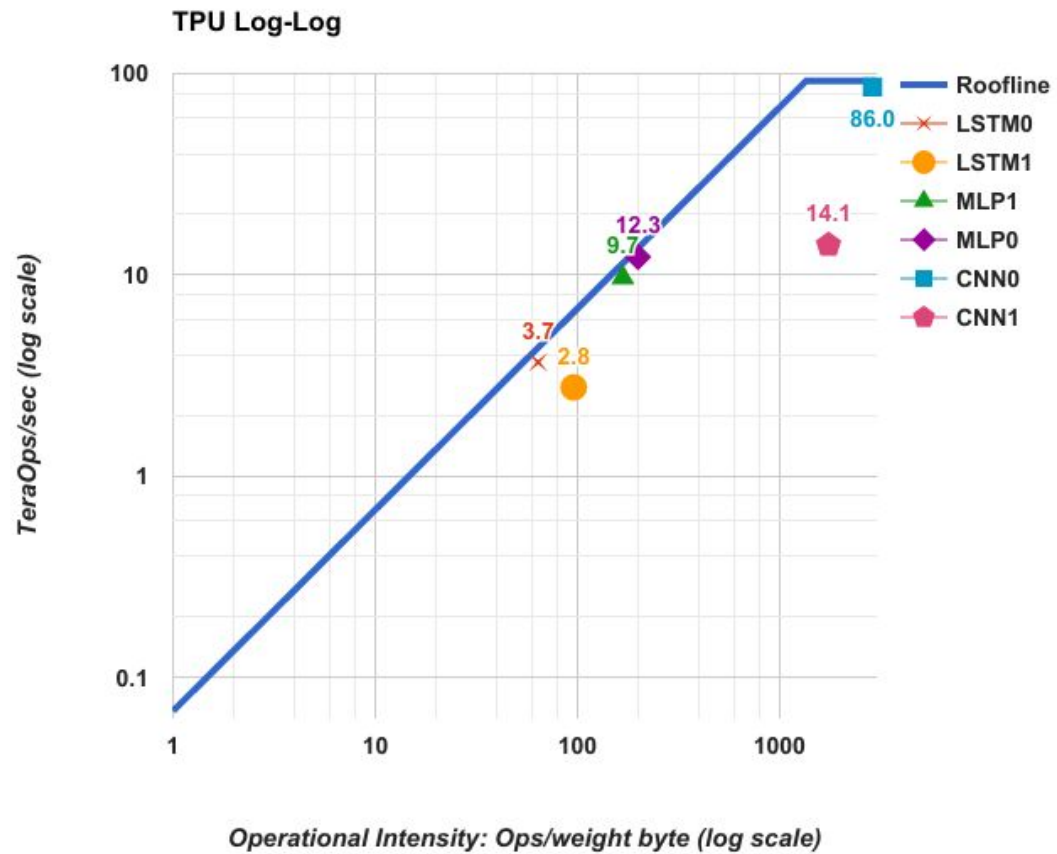
These chips and platforms chosen for comparison because widely deployed in Google data centers

Roofline

- One could hope to always attain peak performance (Flop/s)
- However, finite locality (reuse) and bandwidth limit performance
- Consider idealized processor/caches
- Plot the performance bound using Arithmetic Intensity (AI) as the x-axis...
 - $AI = \text{Flops} / \text{Bytes presented to DRAM}$
 - **Attainable Flop/s = min(peak Flop/s, AI * peak GB/s)**
 - **Log-log scale** makes it easy to doodle, extrapolate performance along Moore's Law, etc...
 - **Kernels with AI less than machine balance are ultimately DRAM**



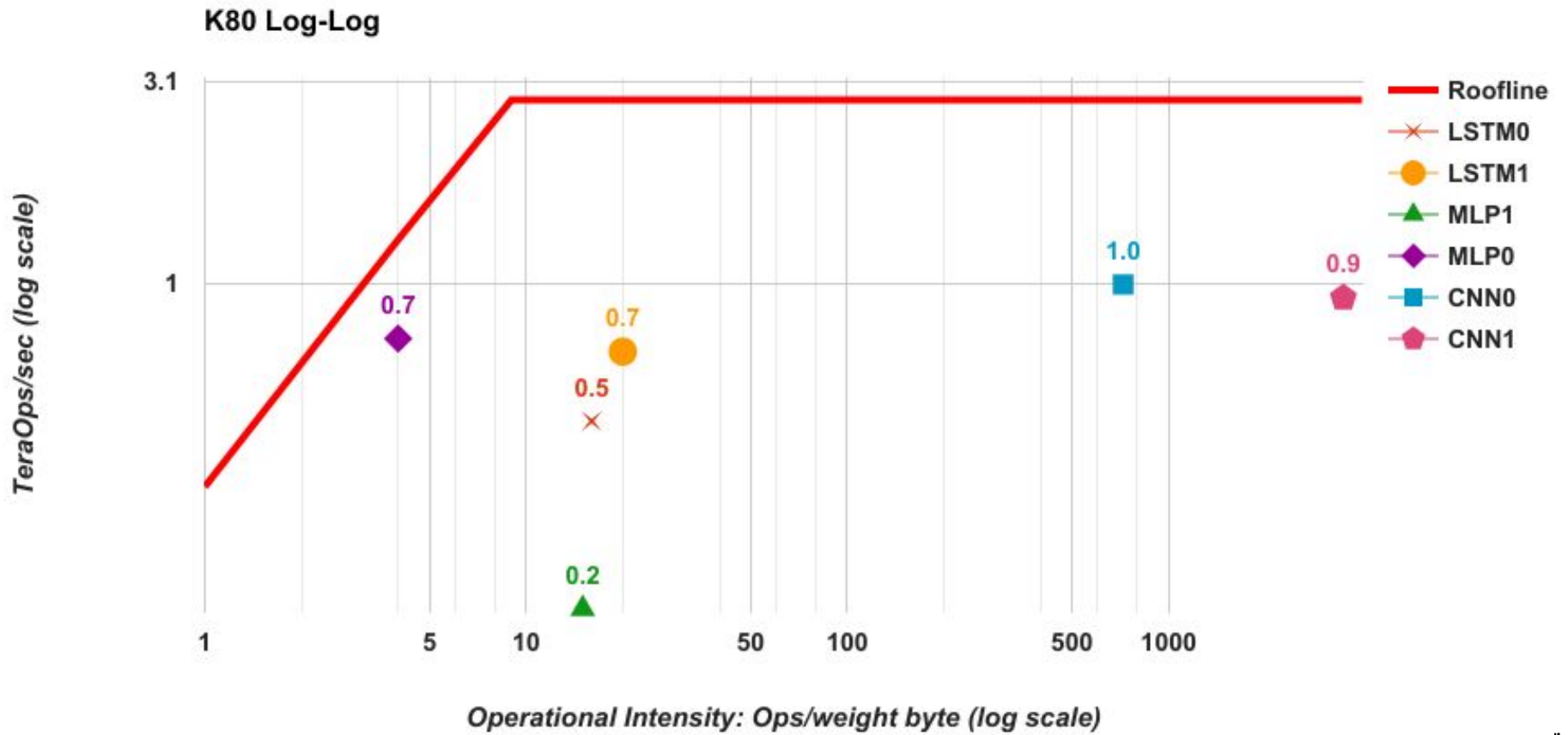
TPU Roofline



CPU (Haswell) Roofline



6 GPU (K80) Roofline



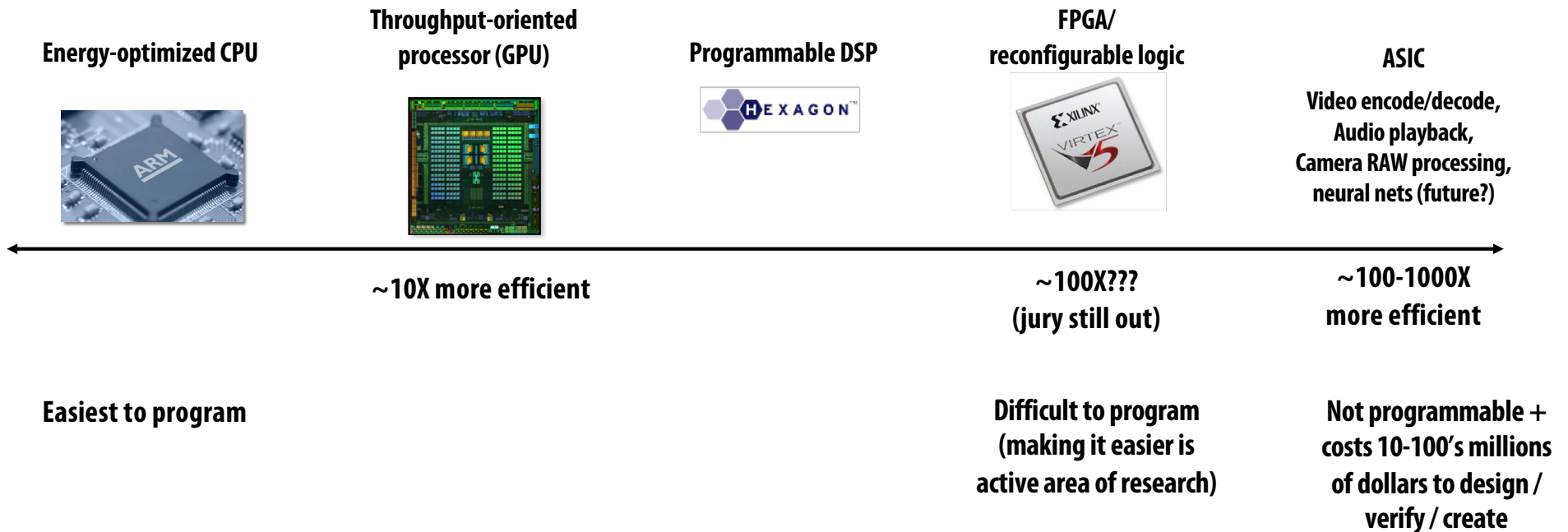
Why Below Rooflines (MLP0)

<i>Type</i>	<i>Batch</i>	<u><i>99th% Response</i></u>	<i>Inf/s (IPS)</i>	<i>% Max IPS</i>
CPU	16	7.2 ms	5,482	42%
CPU	64	21.3 ms	13,194	100%
GPU	16	6.7 ms	13,461	37%
GPU	64	8.3 ms	36,465	100%
TPU	200	7.0 ms	225,000	80%
TPU	250	10.0 ms	280,000	100%

Performance of TPU & GPU Relative to CPU

<i>Type</i>	<i>MLP</i>		<i>LSTM</i>		<i>CNN</i>		<i>Weighted Mean</i>
	<i>0</i>	<i>1</i>	<i>0</i>	<i>1</i>	<i>0</i>	<i>1</i>	
GPU	2.5	0.3	0.4	1.2	1.6	2.7	1.9
TPU	41.0	18.5	3.5	1.2	40.3	71.0	29.2
Ratio	16.7	60.0	8.0	1.0	25.4	26.3	15.3

Summary: choosing the right tool for the job



Credit: Pat Hanrahan for this slide design

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?