

Parallel Programming Principle and Practice

Lecture 2 – Parallel Architecture

Outline

单处理器

□ Uniprocessor Parallelism

超标量

- Pipelining, Superscalar, Out-of-order execution
- Vector Processing/SIMD
- Multithreading: including pThreads
- Uniprocessor Memory Systems

□ Multicore Chips

□ Parallel Computer Architecture

- What is Parallel Architecture?
- A Parallel Zoo of Architectures

How to Run App. Faster ?

□ There are 3 ways to improve performance

1. Work Harder
2. Work Smarter
3. Get Help

□ Computer Analogy 对比

1. Use faster hardware: e.g. reduce the time per instruction (clock cycle) 硬件
2. Optimized algorithms and techniques 编译和算法
3. Multiple computers to solve problem: increase no. of instructions executed per clock cycle

Parallel architecture

UNIPROCESSOR PARALLELISM

Parallelism is Everywhere

- Modern Processor Chips have \approx 1 billion transistors
 - Clearly must get them working in parallel
 - Question: how much of this parallelism must programmer understand?

- How do uniprocessor computer architectures extract parallelism?
 - By finding parallelism within instruction stream
 - Called “Instruction Level Parallelism” (ILP)
 - The theory: hide parallelism from programmer

获得

单核架构只
能通过ILP
达到并行

Parallelism is Everywhere

- Goal of Computer Architects until about 2005:
 - Hide Underlying Parallelism from everyone: OS, Compiler, Programmer

- Examples of ILP techniques 实现指令级并行的技术：

- Pipelining: Overlapping individual parts of instructions
- Superscalar execution: Do multiple things at same time
- VLIW: Let compiler specify which operations can run in parallel
- Vector Processing: Specify groups of similar (independent) operations
- Out of Order Execution (OOO): Allow long operations to happen

标量

字 (可
的指)超标量的原理查看
当于一个cycle同时

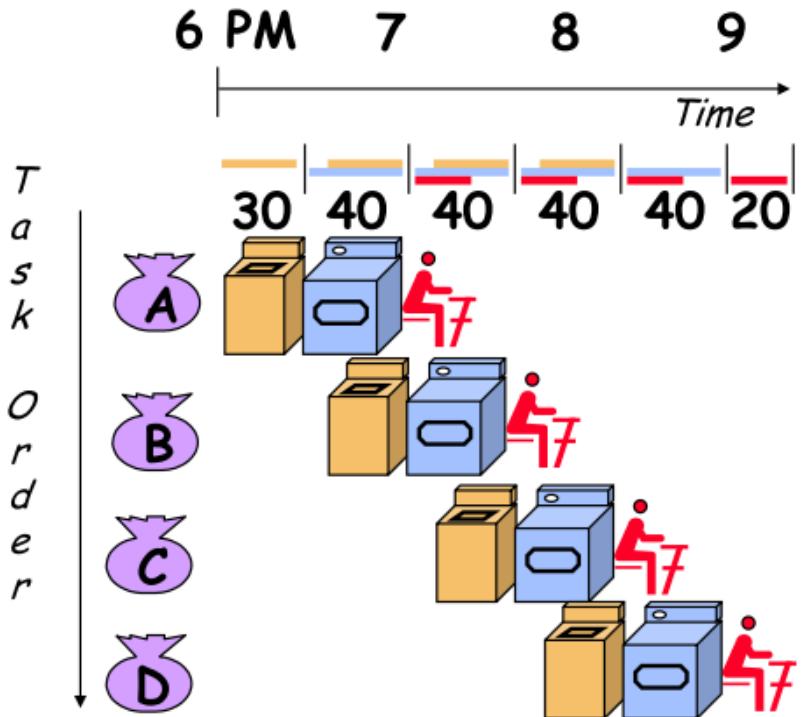
相同的指令

Parallel architecture

PIPELINING, SUPERSCALAR, OUT-OF-ORDER EXECUTION

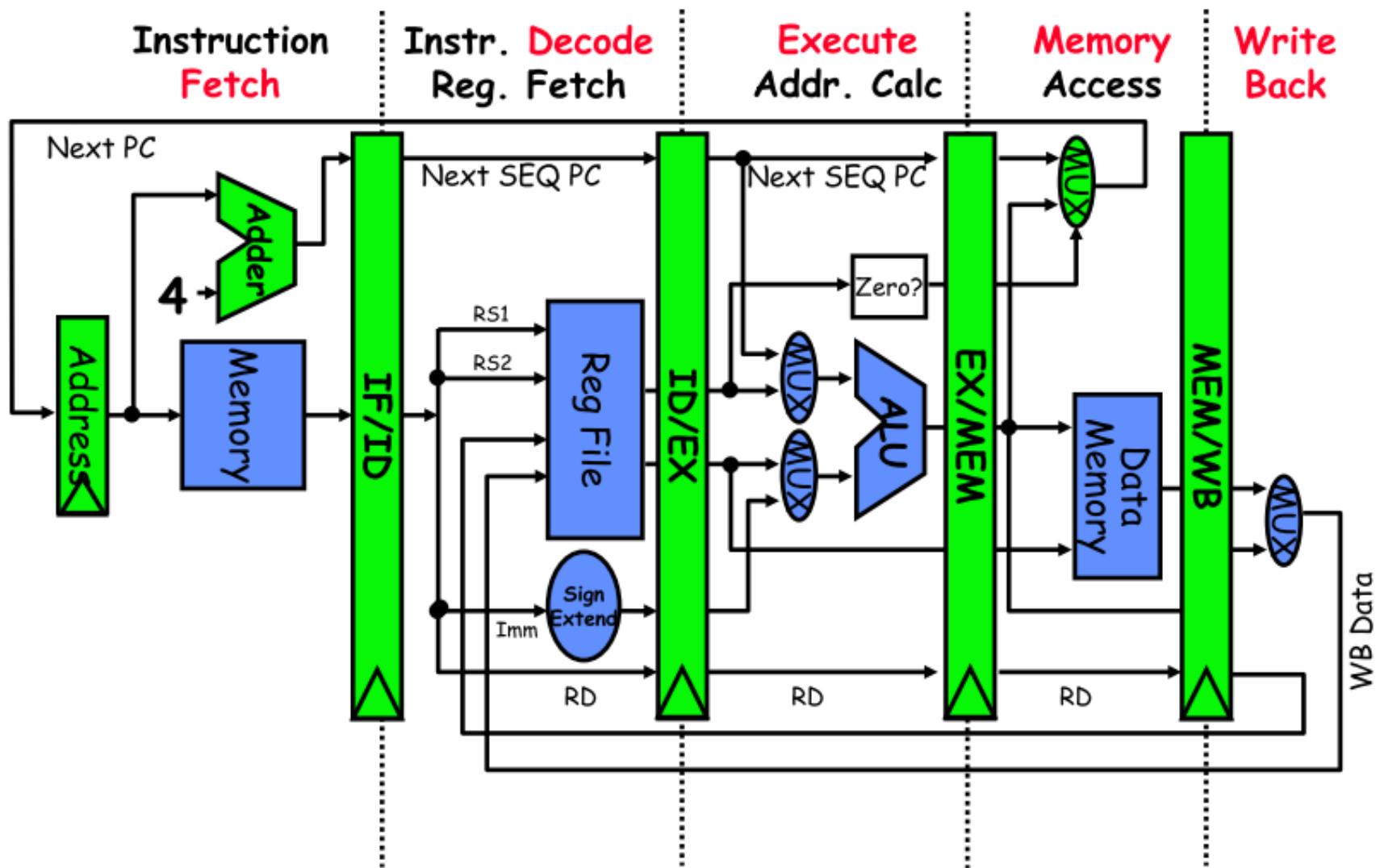
What is Pipelining?

David Patterson's Laundry example: 4 people doing laundry
 wash (30 min) + dry (40 min) + fold (20 min) = 90 min **Latency**

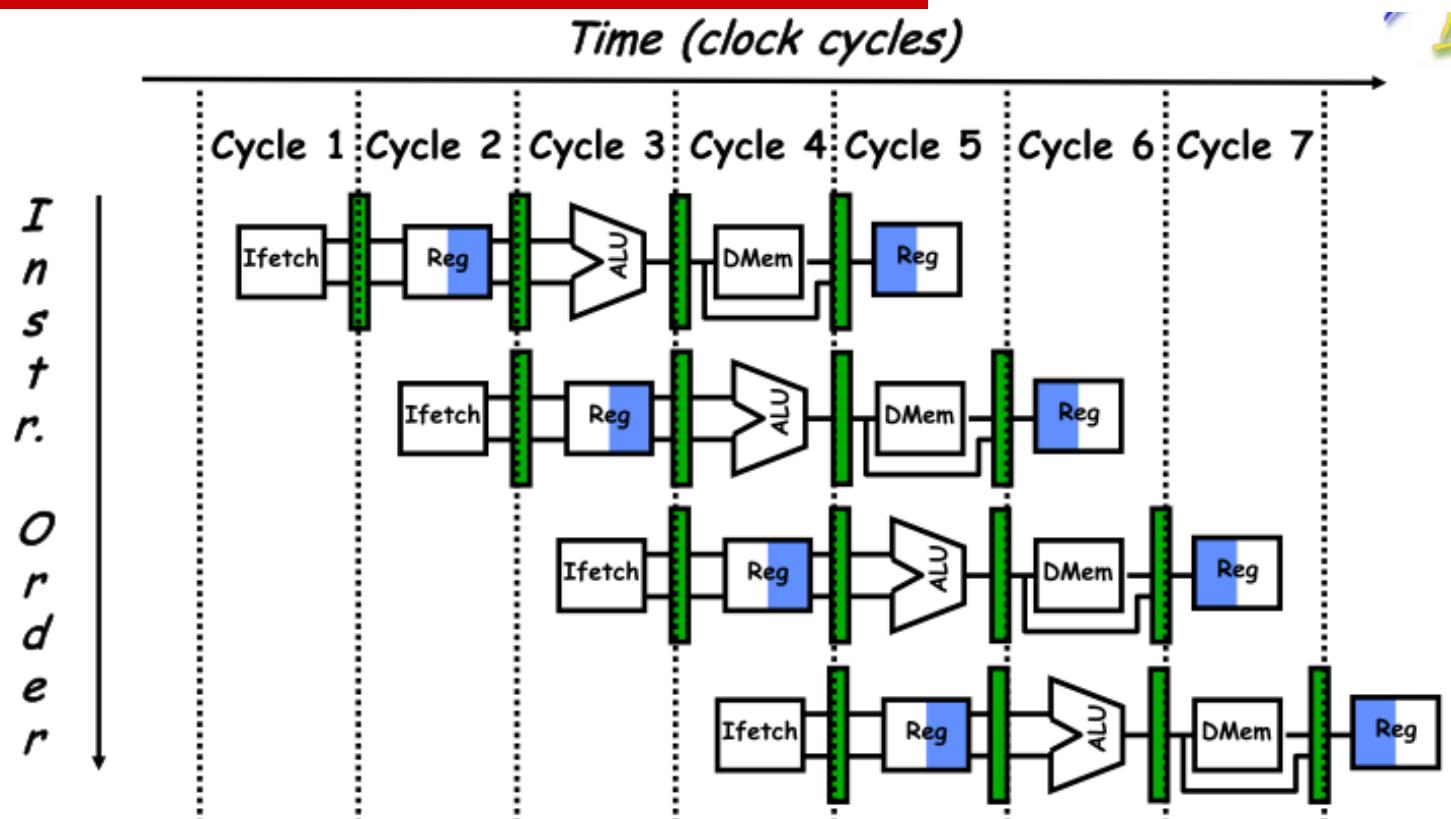


- In this example:
 - Sequential execution takes $4 * 90\text{min} = 6 \text{ hours}$
 - Pipelined execution takes $30+4*40+20 = 3.5 \text{ hours}$
- **Bandwidth** = loads/hour
 - $\text{BW} = 4/6 \text{ l/h w/o pipelining}$
 - $\text{BW} = 4/3.5 \text{ l/h w pipelining}$
 - $\text{BW} <= 1.5 \text{ l/h w pipelining, more total loads}$
- Pipelining helps **bandwidth** but not **latency** (90 min) per job 的时间 : latency
- Bandwidth limited by **slowest pipeline stage**
- Potential speedup = Number of pipe stages

5 Steps of MIPS Pipeline



Visualizing The Pipeline

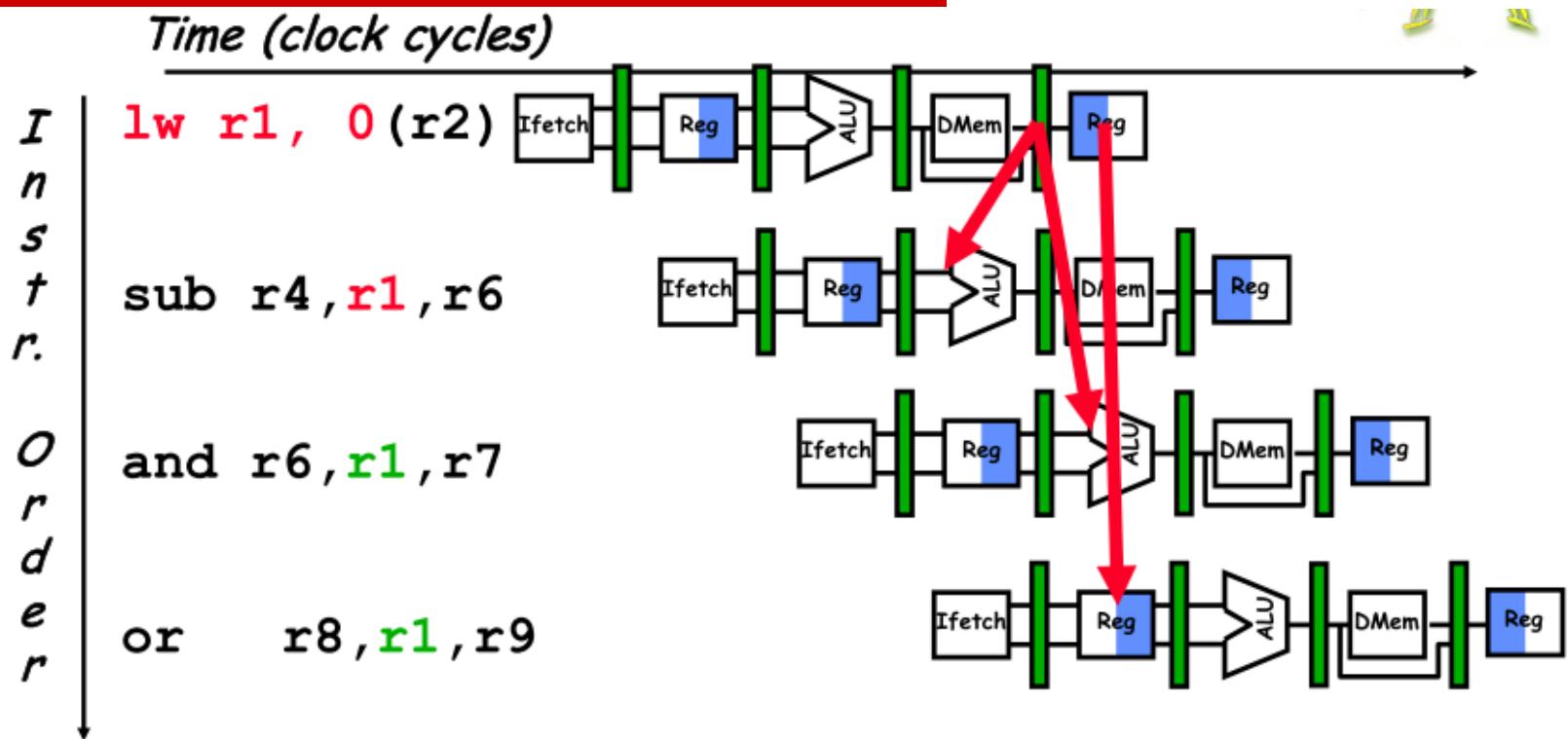


- In ideal case: CPI (cycles/instruction) = 1!
 - On average, put one instruction into pipeline, get one out
- Superscalar: Launch more than one instruction/cycle
 - In ideal case, CPI < 1

Limits to Pipelining

- Overhead prevents arbitrary division** 不能随意划分段
 - Cost of latches (between stages) limits what can do within stage
 - Sets minimum amount of work/stage
- Hazards** prevent next instruction from executing during its designated clock cycle
 - **Structural hazards:** Attempt to use the same hardware to do two different things at once
 - **Data hazards:** Instruction depends on result of prior instruction still in the pipeline
 - **Control hazards:** Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps)
- Superscalar increases occurrence of hazards
 - More conflicting instructions/cycle

Data Hazard: Must go Back in Time?



- Data dependencies between adjacent instructions
 - Must wait (“stall”) for result to be done (No “back in time” exists!)
 - Net result is that CPI > 1

Out-of-Order (OOO) Execution

- Key idea: Allow instructions behind stall to proceed

DIVD F0,F2,F4
ADDD F10,F0,F8
SUBD F12,F8,F14

- Out-of-order execution → out-of-order completion
- Dynamic Scheduling Issues from OOO scheduling

- Must match up results with consumers of instructions
- Precise Interrupts

下半部分这
两行没看太
懂

Instruction	Clock Cycle Number																
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
LD F6,34(R2)	IF	ID	EX	MEM	WB												
LD F2,45(R3)		IF	ID	EX	MEM	WB											
MULTD F0,F2,F4			IF	ID	stall	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	MEM	WB
SUBD F8,F6,F2				IF	ID	A1	A2	MEM	WB								
DIVD F10,F0,F6					IF	stall	D1	D2									
ADDD F6,F8,F2						IF	ID	A1	A2	MEM	WB						

RAW

WAR

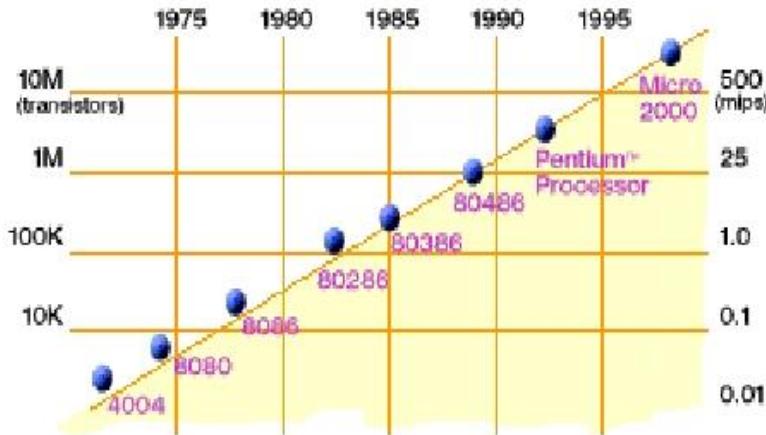
Modern ILP

- Dynamically scheduled, out-of-order execution
 - Current microprocessors fetch 6-8 instructions per cycle
 - Pipelines are 10s of cycles deep → many overlapped instructions in execution at once, although work often discarded 丢弃
- What happens
 - 取 Grab a bunch of instructions, determine all their dependences, eliminate dep's wherever possible, throw them all into the execution unit, let each one move forward as its dependences are resolved
 - Appears as if executed sequentially

Modern ILP (Cont.)

- Dealing with Hazards: May need to *guess!*
 - Called “Speculative Execution” 猜测执行
 - Speculate on Branch results, Dependencies, even Values!
 - If correct, don't need to stall for result → yields performance
 - If not correct, waste time *and power*
 - Must be able to UNDO a result if guess is wrong
 - Problem: accuracy of guesses decreases with number of simultaneous instructions in pipeline
- Huge complexity
 - Complexity of many components scales as n^2 (issue width)
 - Power consumption big problem

Technology Trends: Moore's Law



2X transistors/Chip Every 1.5 years

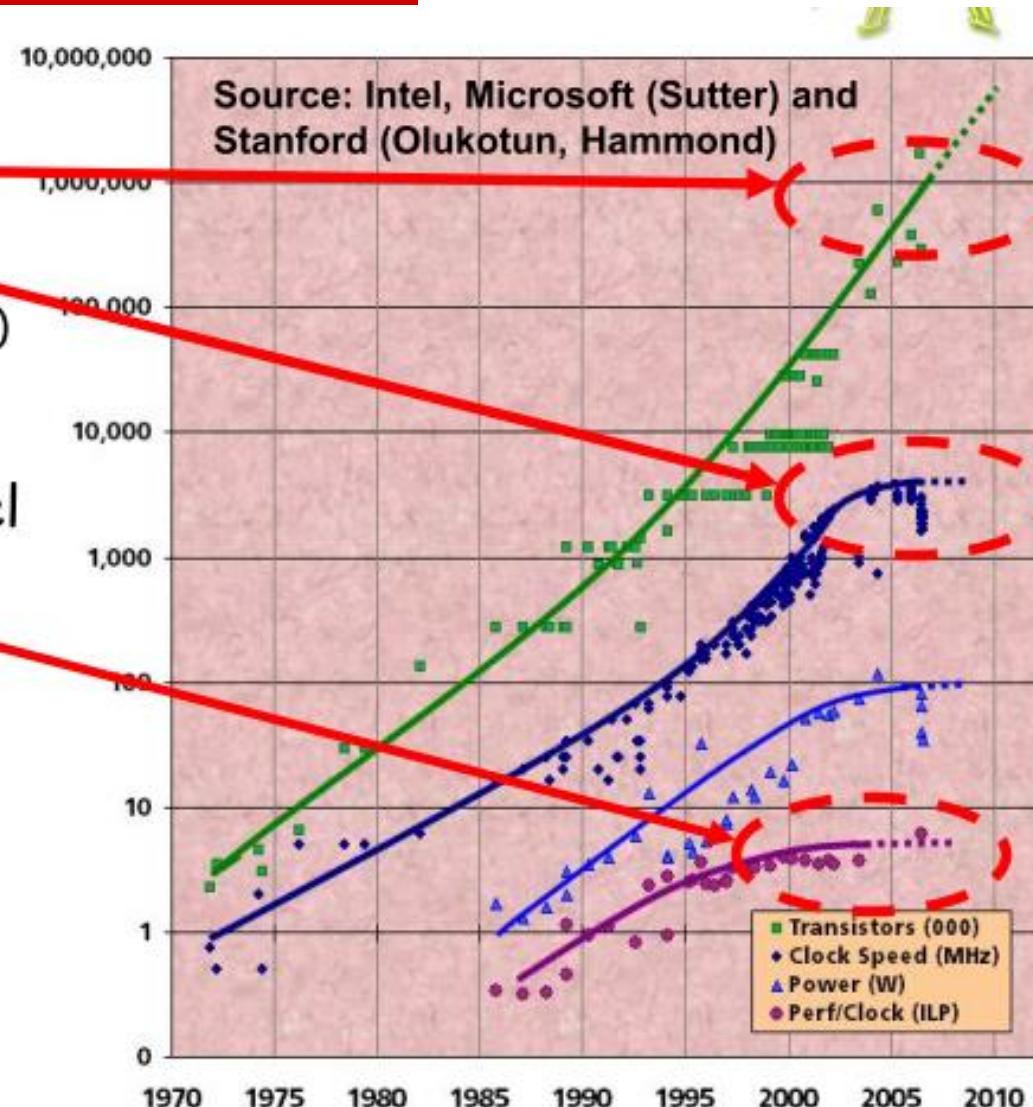
Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18 months.

Called “**Moore’s Law**”

Microprocessors have become smaller, denser, and more powerful.

Limiting Forces: Clock Speed and ILP

- Chip density is continuing increase ~2x every 2 years
- Clock speed is not
 - # processors/chip (cores) may double instead
- There is little or no more Instruction Level Parallelism (ILP) to be found
 - Can no longer allow programmer to think in terms of a serial programming model
- Conclusion:
Parallelism must be exposed to software!



Parallel architecture

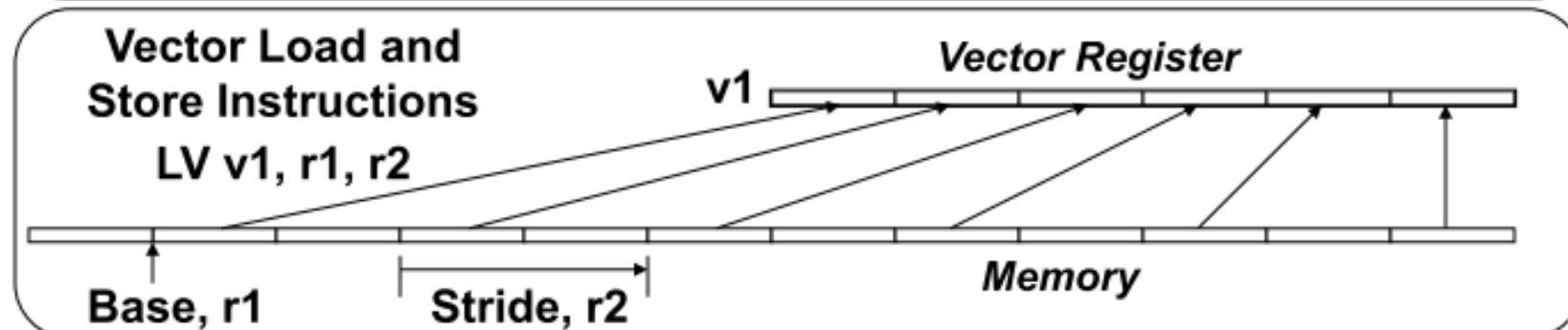
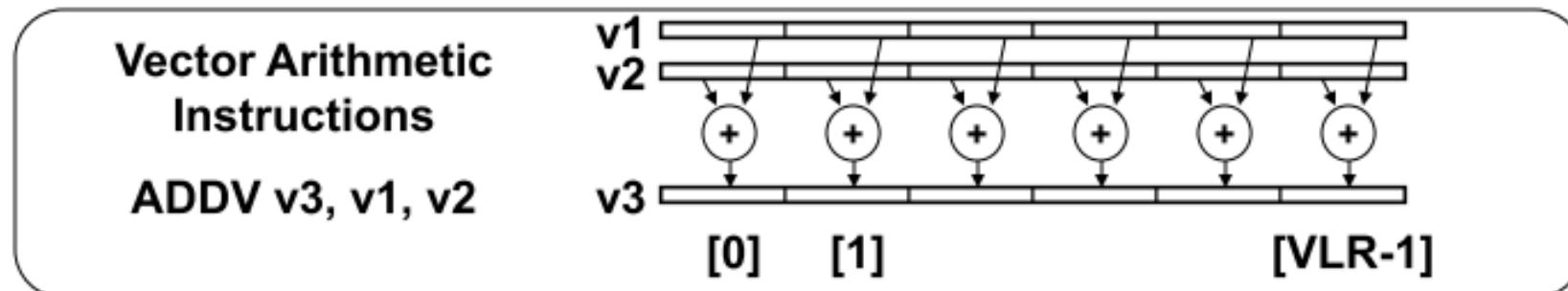
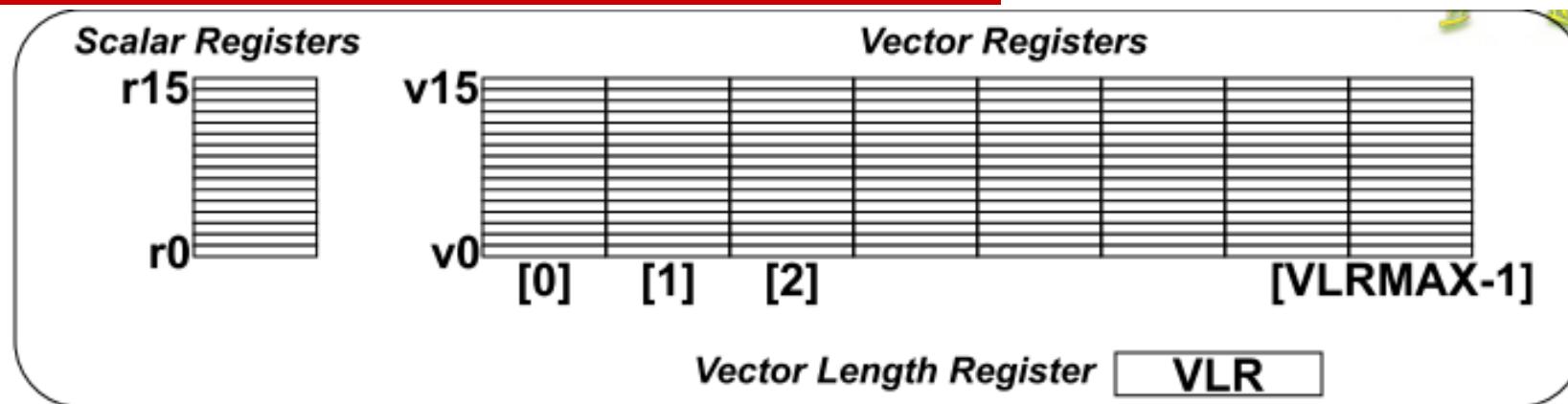
VECTOR PROCESSING/SIMD

Vector Code Example

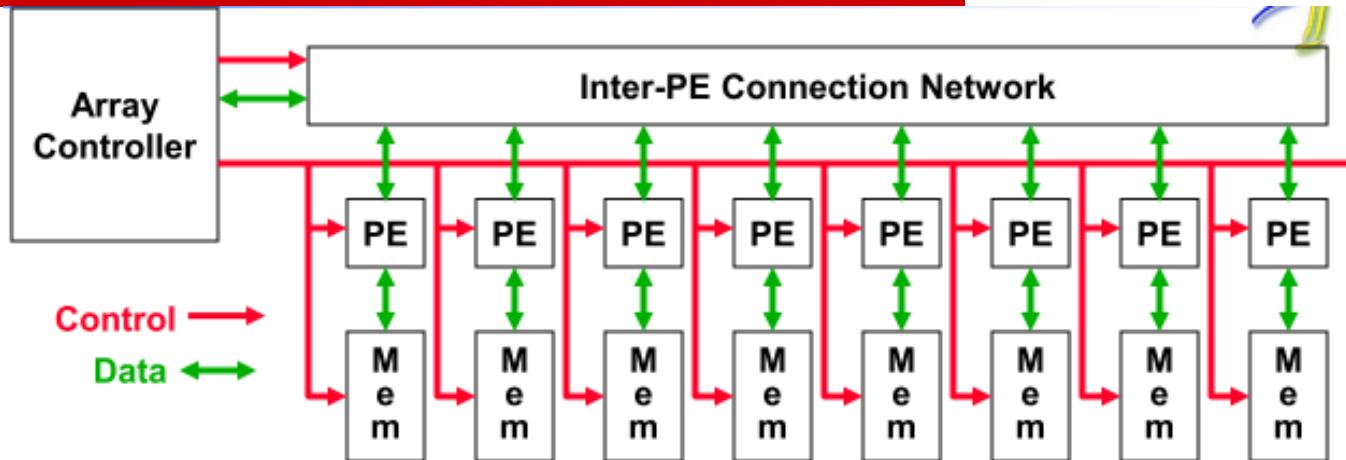
# C code	# Scalar Code	# Vector Code
<pre># C code for (i=0; i<64; i++) C[i] = A[i] + B[i];</pre> <div style="border: 2px solid red; padding: 5px; margin-top: 10px;"> 多个数据用一条指令处理 </div>	<pre># Scalar Code LI R4, 64 loop: L.D F0, 0(R1) L.D F2, 0(R2) ADD.D F4, F2, F0 S.D F4, 0(R3) DADDIU R1, 8 DADDIU R2, 8 DADDIU R3, 8 DSUBIU R4, 1 BNEZ R4, loop</pre>	<pre># Vector Code LI VLR, 64 LV V1, R1 LV V2, R2 ADDV.D V3, V1, V2 SV V3, R3</pre>

- Require programmer (or compiler) to identify parallelism
 - Hardware does not need to re-extract parallelism
- Many multimedia/HPC applications are natural consumers of vector processing

Vector Programming Model

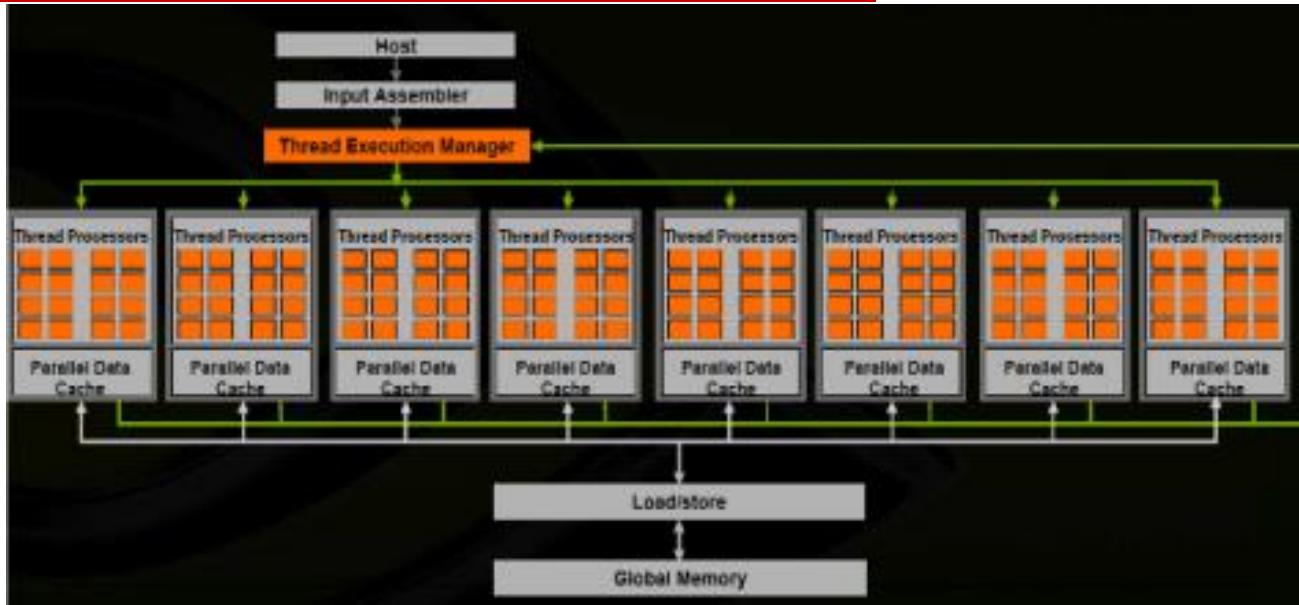


SIMD Architecture



- Single Instruction Multiple Data (SIMD)
- Central controller broadcasts instructions to multiple processing elements (PEs)
 - Only requires one controller for whole array
 - Only requires storage for one copy of program
 - All computations are fully synchronized
- Recent return to popularity
 - GPU (Graphics Processing Units) have SIMD properties
 - However, also multicore behavior, so mix of SIMD and MIMD (more later)
- Dual between Vector and SIMD execution

General-Purpose GPUs (GP-GPUs)



- In 2006, Nvidia introduced GeForce 8800 GPU supporting a new programming language: CUDA
 - Compute Unified Device Architecture
 - OpenCL is a vendor-neutral version of same ideas
- Idea: Take advantage of GPU computational performance and memory bandwidth to **accelerate some kernels** for general-purpose computing
- Attached processor model: Host CPU issues data-parallel kernels to GP-GPU for execution

Parallel architecture

MULTITHREADING: INCLUDING PTHREADS

Thread Level Parallelism (TLP)

- ILP exploits implicit parallel operations within a loop or straight-line code segment 显式的：程序员可以看到
- TLP explicitly represented by the use of multiple threads of execution that are inherently parallel
 - Threads can be on a single processor
 - Or, on multiple processors
- Concurrency vs Parallelism
 - Concurrency is when two tasks can start, run, and complete in overlapping time periods. It doesn't necessarily mean they'll ever both be running at the same instant
 - For instance, multitasking on a single-threaded machine
 - Parallelism is when tasks literally run at the same time, eg. on a multicore processor
- Goal: Use multiple instruction streams to improve
 - Throughput of computers that run many programs
 - Execution time of multi-threaded programs

Common Notions of Thread Creation

- **cobegin/coend**

```
cobegin  
    job1(a1);  
    job2(a2);  
coend
```

- Statements in block may run in parallel
- cobegins may be nested
- Scoped, so you cannot have a missing coend

- **fork/join**

```
tid1 = fork(job1, a1);  
job2(a2);  
join tid1;
```

- Forked procedure runs in parallel
- Wait at join point if it's not finished

- **future**

```
v = future(job1(a1));  
... = ...v...;
```

- Future expression possibly evaluated in parallel
- Attempt to use return value will wait

- Threads expressed in the code may not turn into independent computations

- Only create threads if processors idle
- Example: Thread-stealing runtimes such as cilk

Overview of POSIX Threads

- POSIX: Portable Operating System Interface for UNIX
 - Interface to Operating System utilities
- Pthreads: The POSIX threading interface
 - System calls to create and synchronize threads
 - Should be relatively uniform across UNIX-like OS platforms
 - Originally IEEE POSIX 1003.1c
- Pthreads contain support for
 - Creating parallelism
 - Synchronizing
 - No explicit support for communication, because shared memory is implicit; a pointer to shared data is passed to a thread
 - Only for HEAP! Stacks not shared

Forking POSIX Threads

Signature:

```
int pthread_create(pthread_t *,
                  const pthread_attr_t *,
                  void * (*)(void *),
                  void *);
```

Example call:

```
errcode = pthread_create(&thread_id; &thread_attribute; &thread_fun; &fun_arg);
```

- **thread_id** is the thread id or handle (used to halt, etc.)
- **thread_attribute** various attributes
 - Standard default values obtained by passing a NULL pointer
 - Sample attribute: minimum stack size
- **thread_fun** the function to be run (takes and returns void*)
- **fun_arg** an argument can be passed to **thread_fun** when it starts
- **errcode** will be set nonzero if the create operation fails

Simple Threading Example (pThreads)

```
void* SayHello(void *foo) {  
    printf( "Hello, world!\n" );  
    return NULL;  
}  
  
int main() {  
    pthread_t threads[16];  
    int tn;  
    for(tn=0; tn<16; tn++) {  
        pthread_create(&threads[tn], NULL, SayHello, NULL);  
    }  
    for(tn=0; tn<16 ; tn++) {  
        pthread_join(threads[tn], NULL);  
    }  
    return 0;  
}
```

E.g., compile using gcc -lpthread

Shared Data and Threads

- Variables declared outside of main are shared
- Objects allocated on the **heap** may be shared (if pointer is passed)
- Variables on the **stack** are private: passing pointer to these around to other threads can cause problems
- Often done by creating a large “thread data” struct, which is passed into all threads as argument

char *message = "Hello World!\n";

```
pthread_create(&thread1, NULL,
                print_fun,(void*) message);
```

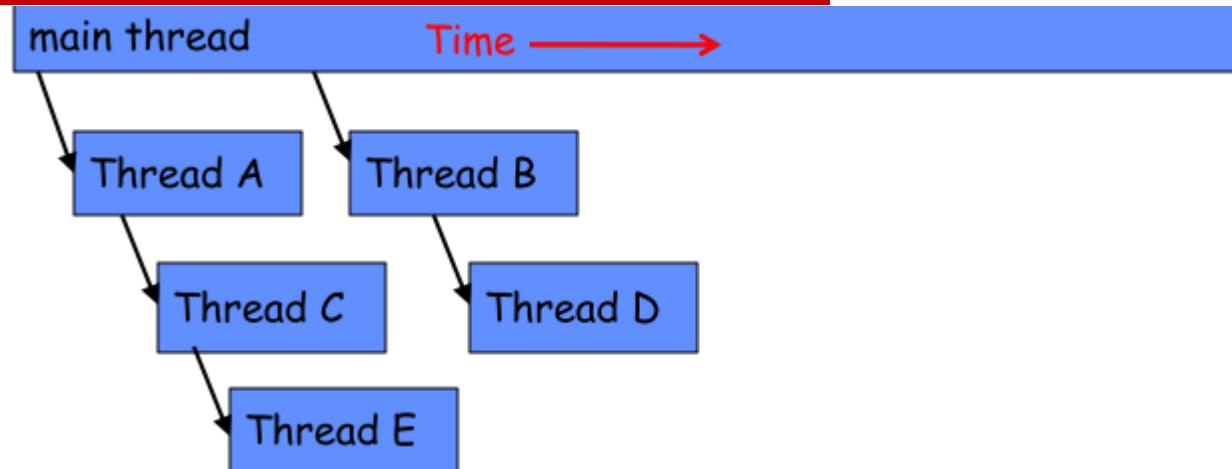
Loop Level Parallelism

- Many application have parallelism in loops

```
double stuff [n][n];  
for (int i = 0; i < n; i++)  
    for (int j = 0; j < n; j++)  
        ... pthread_create (... , update_stuff, ... , &stuff[i][j]);
```

- But overhead of thread creation is nontrivial
 - *update_stuff* should have a significant amount of work
- Common Performance Pitfall: Too many threads
 - The cost of creating a thread is 10's of thousands of cycles on modern architectures
 - Solution: Thread blocking: use a small # of threads, often equal to the number of cores/processors or hardware threads

Thread Scheduling



- Once created, when will a given thread run?
 - It is up to the operating system or hardware, but it will run eventually, even if you have more threads than cores
 - But scheduling may be non-ideal for your application
- Programmer can provide hints or affinity in some cases
 - E.g., create exactly P threads and assign to P cores
- Can provide user-level scheduling for some systems
 - Application-specific tuning based on programming model

Multithreaded Execution

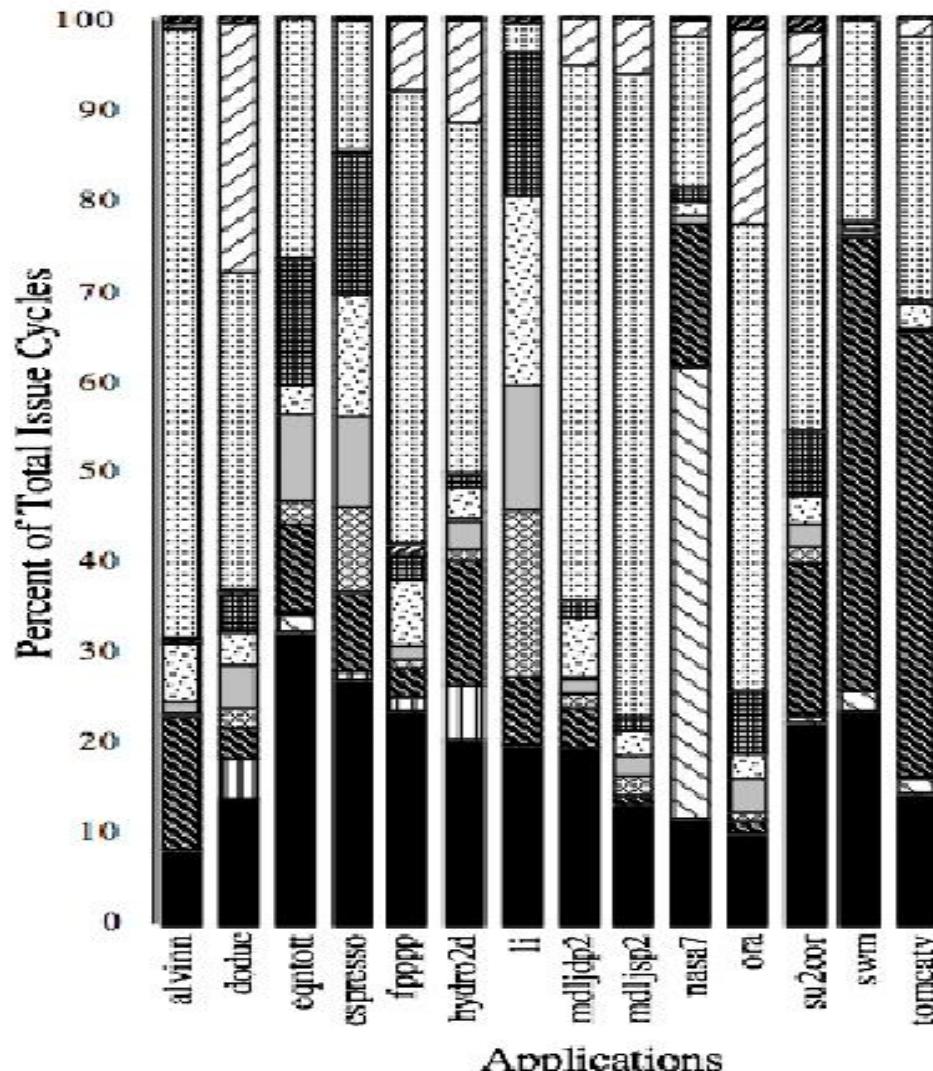
- Multitasking operating system
 - Gives “illusion” that multiple things happen at same time
 - Switches at a coarse-grained time (for instance: 10ms)
- Hardware Multithreading: multiple threads share processor simultaneously (with little OS help)
 - Hardware does switching
 - HW for fast thread switch in small number of cycles
 - much faster than OS switch which is 100s to 1000s of clocks
 - Processor duplicates independent state of each thread
 - e.g., a separate copy of register file, a separate PC, and for running independent programs, a separate page table
 - Memory shared through the virtual memory mechanisms, which already support multiple processes
- When to switch between threads?
 - Alternate instruction per thread (fine grain)
 - When a thread is stalled, perhaps for a cache miss, another thread can be executed (coarse grain)

What about combining ILP and TLP?

- TLP and ILP exploit two different kinds of parallel structure in a program
- Could a processor oriented at ILP benefit from exploiting TLP?
 - functional units are often idle in data path designed for ILP because of either stalls or dependences in the code
 - TLP used as a source of independent instructions that might keep the processor busy during stalls
 - TLP be used to occupy functional units that would otherwise lie idle when insufficient ILP exists
- Called “Simultaneous Multithreading”
 - Intel renamed this “Hyperthreading”

同时多线程/超线程

Quick Recall: Many Resources IDLE!



For an 8-way superscalar



From: Tullsen, Eggers, and Levy, "Simultaneous Multithreading: Maximizing On-chip Parallelism, ISCA, 1995

Simultaneous Multi-threading

One thread, 8 units

Cycle M M FX FX FP FP BR CC

1	Y								Y
2	Y					Y			
3			Y	Y					
4									
5									
6									
7	Y		Y		Y				
8		Y		Y					
9			Y						

Two threads, 8 units

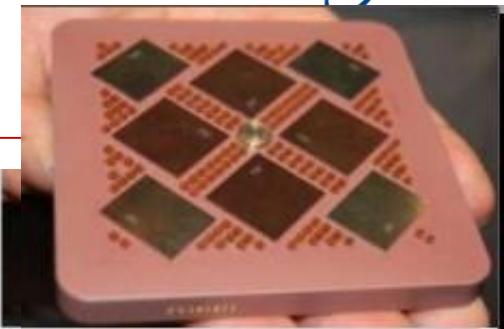
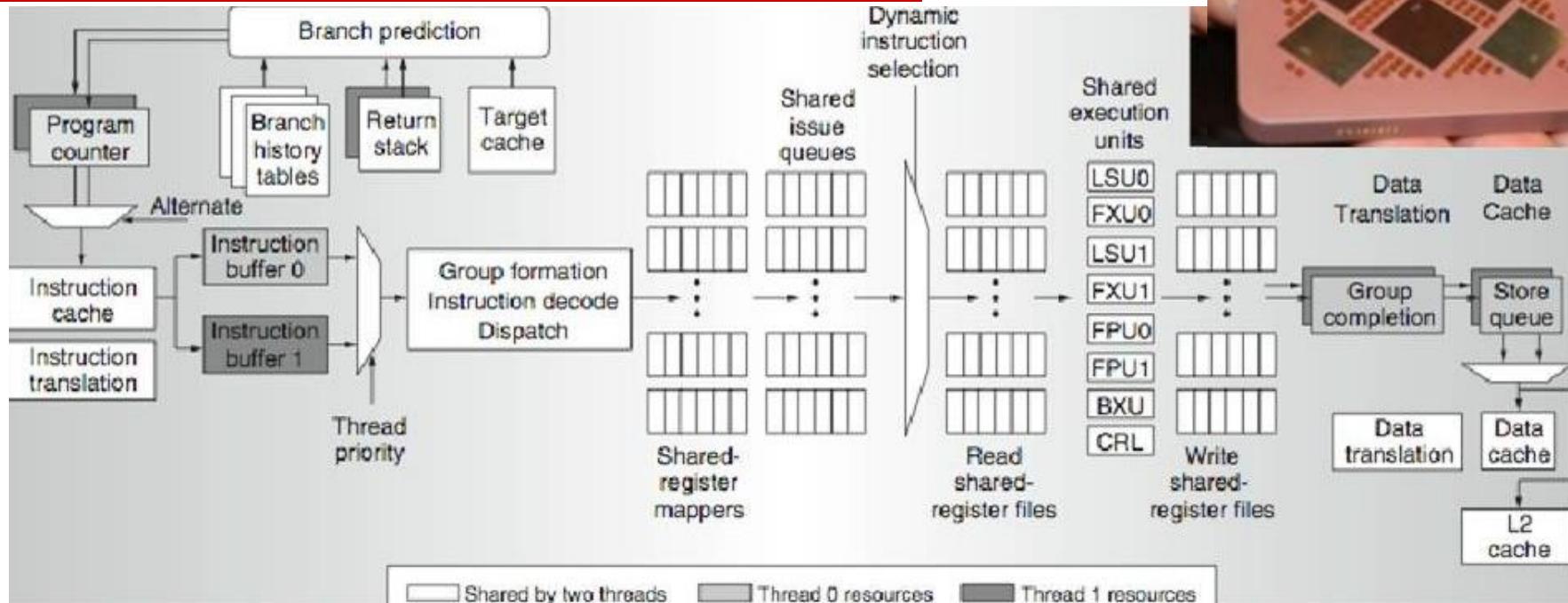
Cycle M M FX FX FP FP BR CC

1	Y		B	B	B				Y
2	Y		Y		B			B	Y
3	B				Y		Y		
4	B							B	
5	B								B
6									
7	Y			B	Y		B		
8		Y			B		Y		
9	B		B		Y			B	

M = Load/Store, FX = Fixed Point, FP = Floating Point, BR = Branch, CC = Condition Codes

Power5 Dataflow

tensorflow的原理



□ Why only two threads?

- With 4, one of the shared resources (physical registers, cache, memory bandwidth) would be prone to bottleneck

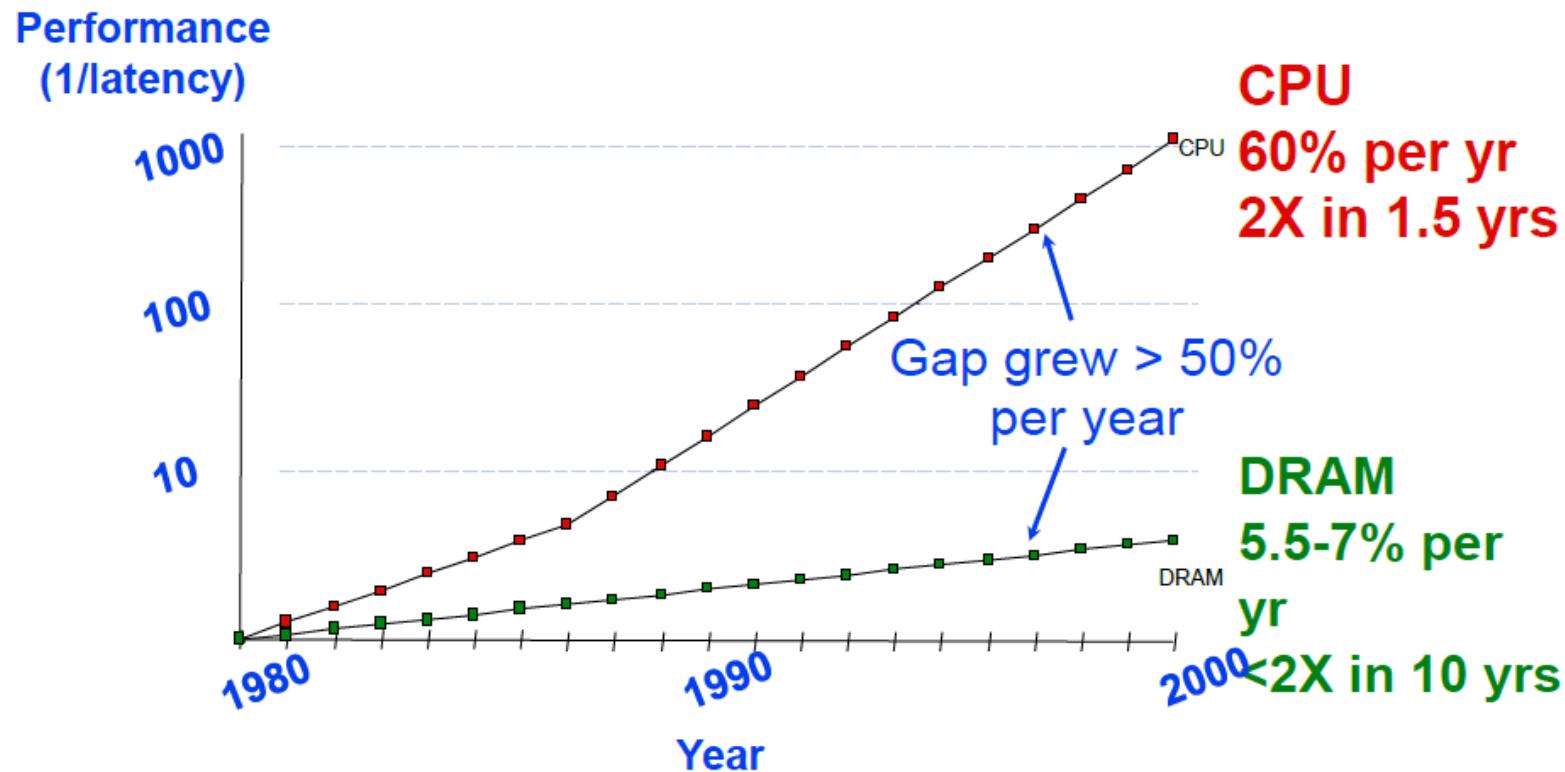
□ Cost

- The Power5 core is about 24% larger than the Power4 core because of the addition of SMT (Simultaneous Multi-threading) support

Parallel architecture

UNIPROCESSOR MEMORY SYSTEMS

Limiting Force: Memory Wall

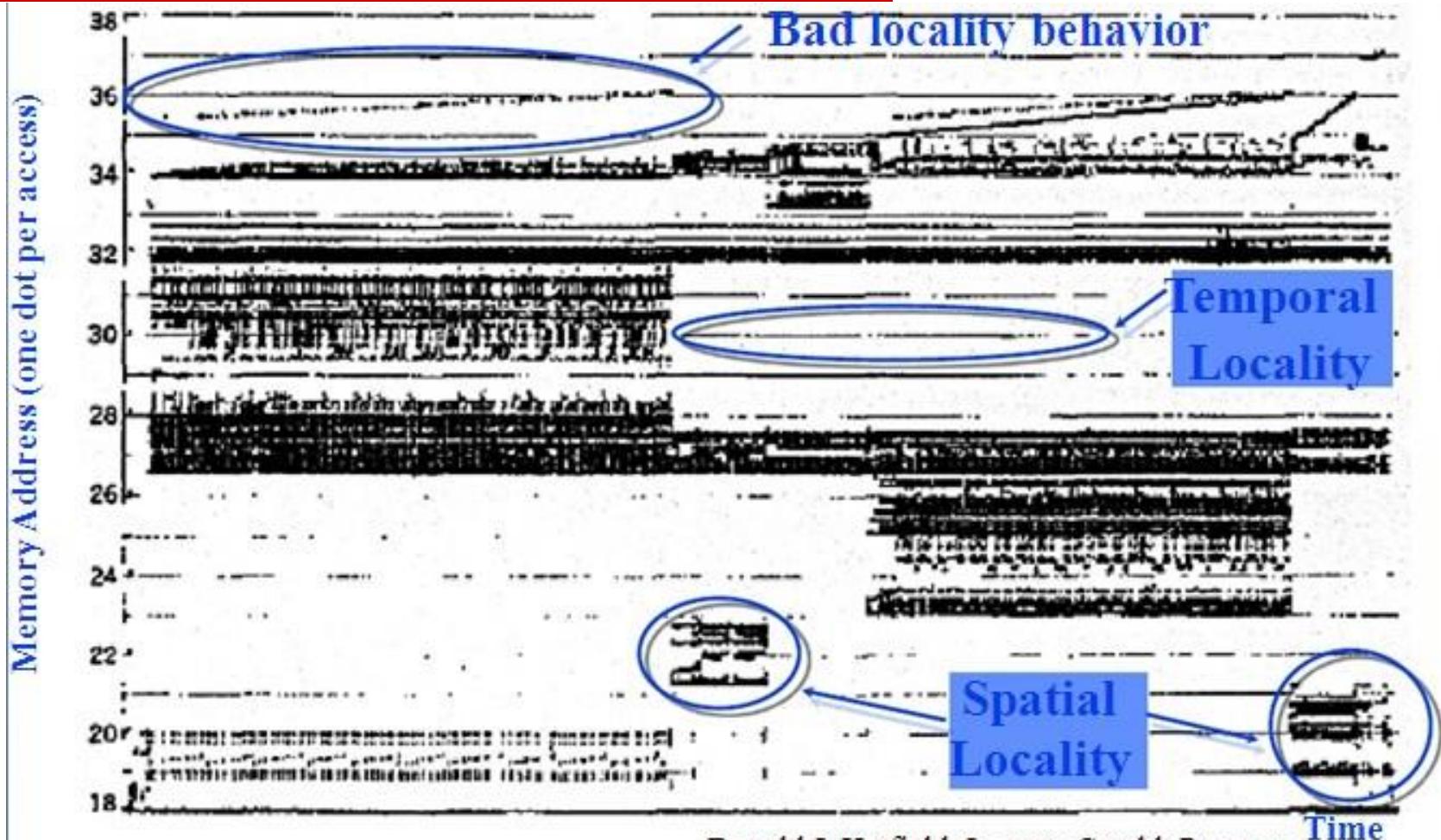


- How do architects address this gap?
 - Put small, fast “cache” memories between CPU and DRAM (Dynamic Random Access Memory).
 - Create a “memory hierarchy” 内存层次结构

Principle of Locality

- Principle of Locality
 - Program access a relatively small portion of the address space at any instant of time
- Two Different Types of Locality
 - *Temporal Locality* (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
 - *Spatial Locality* (Locality in Space): If an item is referenced, items whose addresses are closeby tend to be referenced soon (e.g., straightline code, array access)
- Last 25 years, HW relied on locality for speed

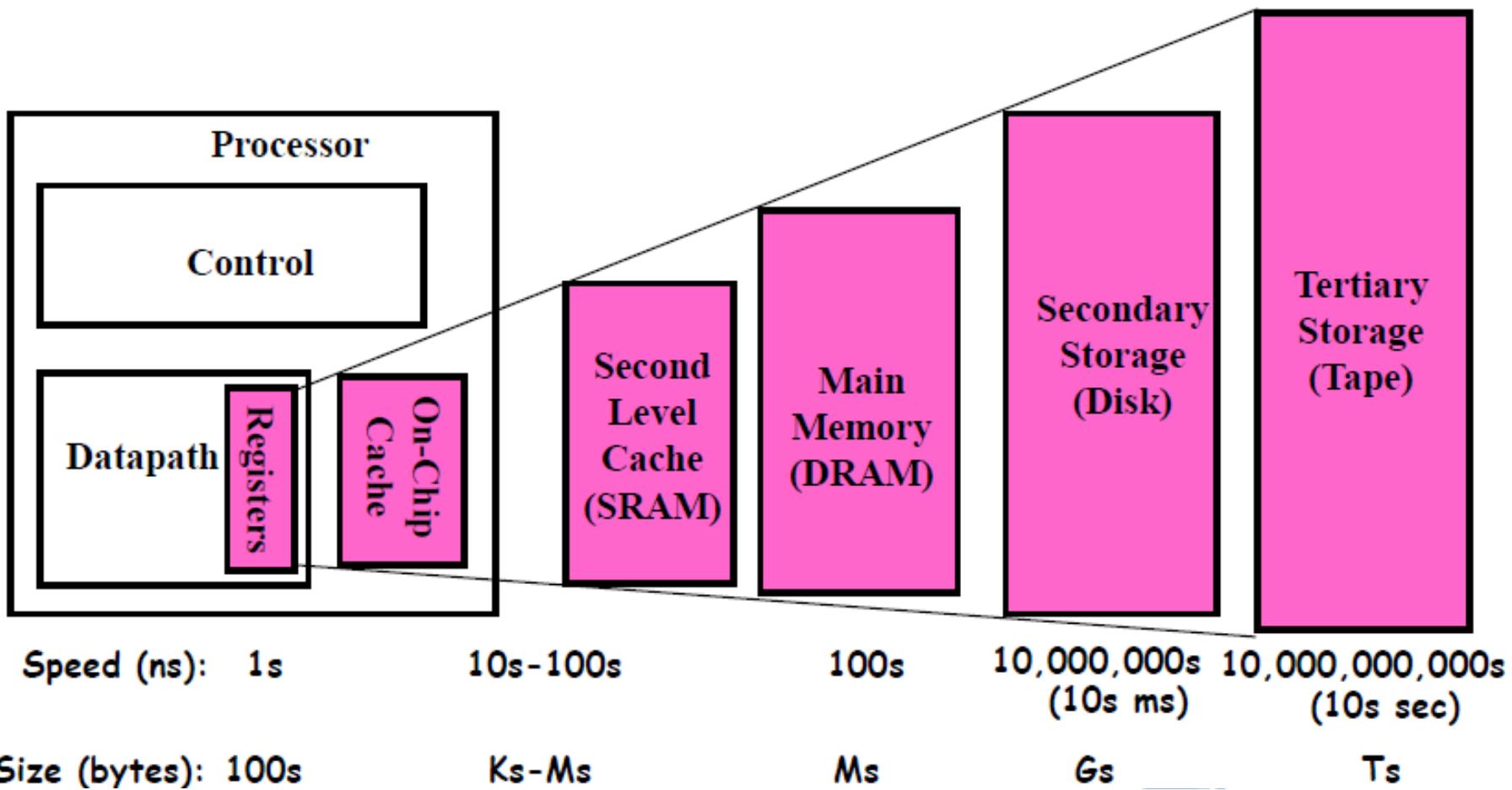
Programs with locality cache well



Donald J. Hatfield, Jeanette Gerald: Program
Restructuring for Virtual Memory. IBM Systems Journal
10(3): 168-192

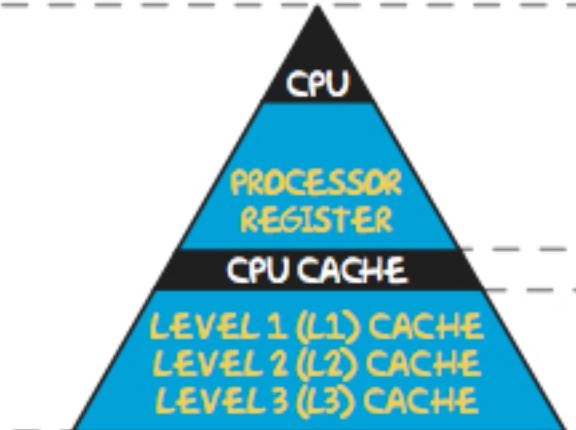
Memory Hierarchy

- Take advantage of the principle of locality to
 - Present as much memory as in the cheapest technology
 - Provide access at speed offered by the fastest technology



Memory Hierarchy

PROCESSOR



SUPER FAST
SUPER EXPENSIVE
TINY CAPACITY

SD-RAM,
DDR-SDRAM, ...

PHYSICAL MEMORY

RANDOM ACCESS
MEMORY (RAM)

FASTER
EXPENSIVE
SMALL CAPACITY

SOLID STATE
DRIVES

SOLID STATE MEMORY

NON-VOLATILE FLASH-BASED MEMORY

FAST
PRICED REASONABLY
AVERAGE CAPACITY

MECHANICAL
HARD DRIVES

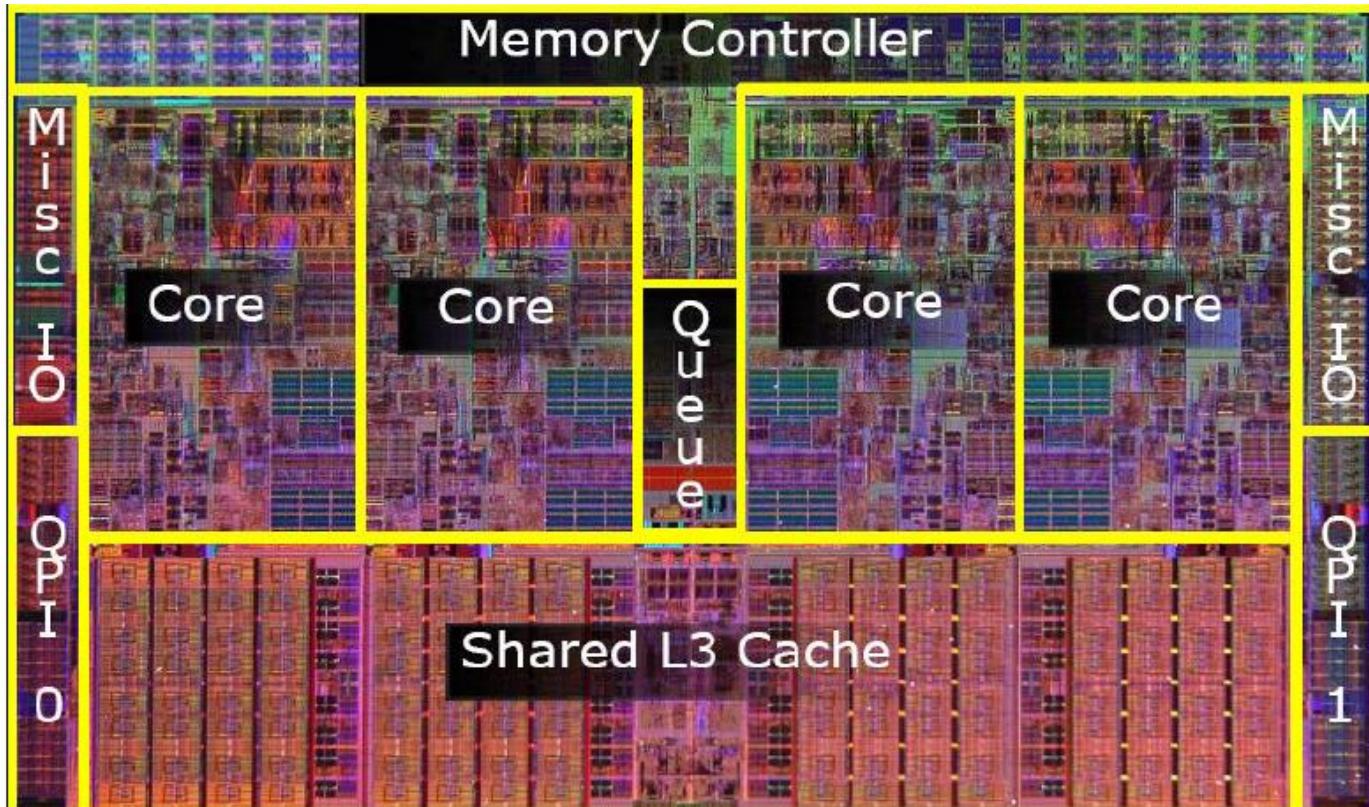
VIRTUAL MEMORY

FILE-BASED MEMORY

AVERAGE SPEED
PRICED REASONABLY
AVERAGE CAPACITY

SLOW
CHEAP
LARGE CAPACITY

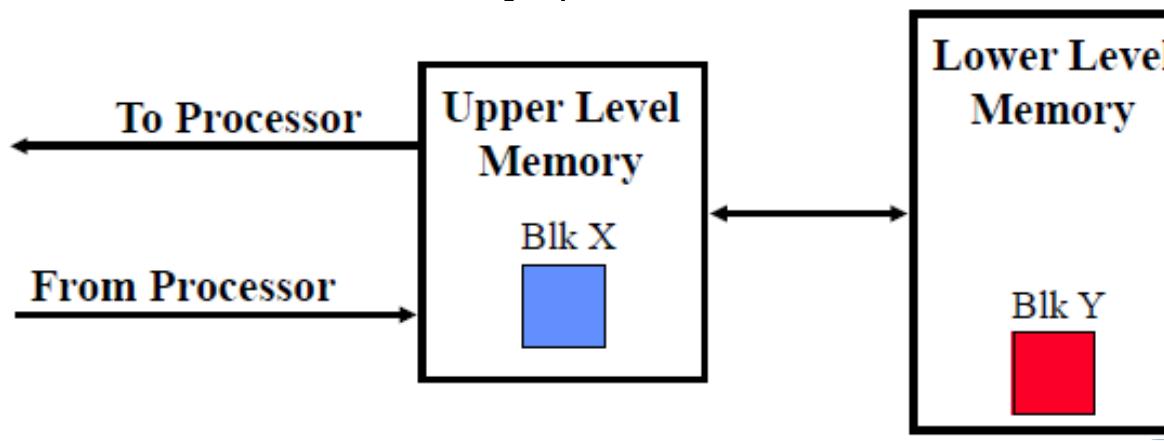
Example of Modern Core: Nehalem



- ON-chip cache resources
 - For each core: L1: 32KB instruction and 32KB data cache, L2: 1MB, L3: 8MB shared among all 4 cores
- Integrated, on-chip memory controller (DDR3)

Memory Hierarchy: Terminology

- Hit: data appears in some blocks in the upper level (example: Block X)
 - Hit Rate: the fraction of memory access found in the upper level
 - Hit Time: Time to access the upper level which consists of RAM access time + Time to determine hit/miss
- Miss: data needs to be retrieve from a block in the lower level (Block Y)
 - Miss Rate = 1 - (Hit Rate)
 - Miss Penalty: Time to replace a block in the upper level + Time to deliver the block the processor
- Hit Time << Miss Penalty (500 instructions on 21264!)



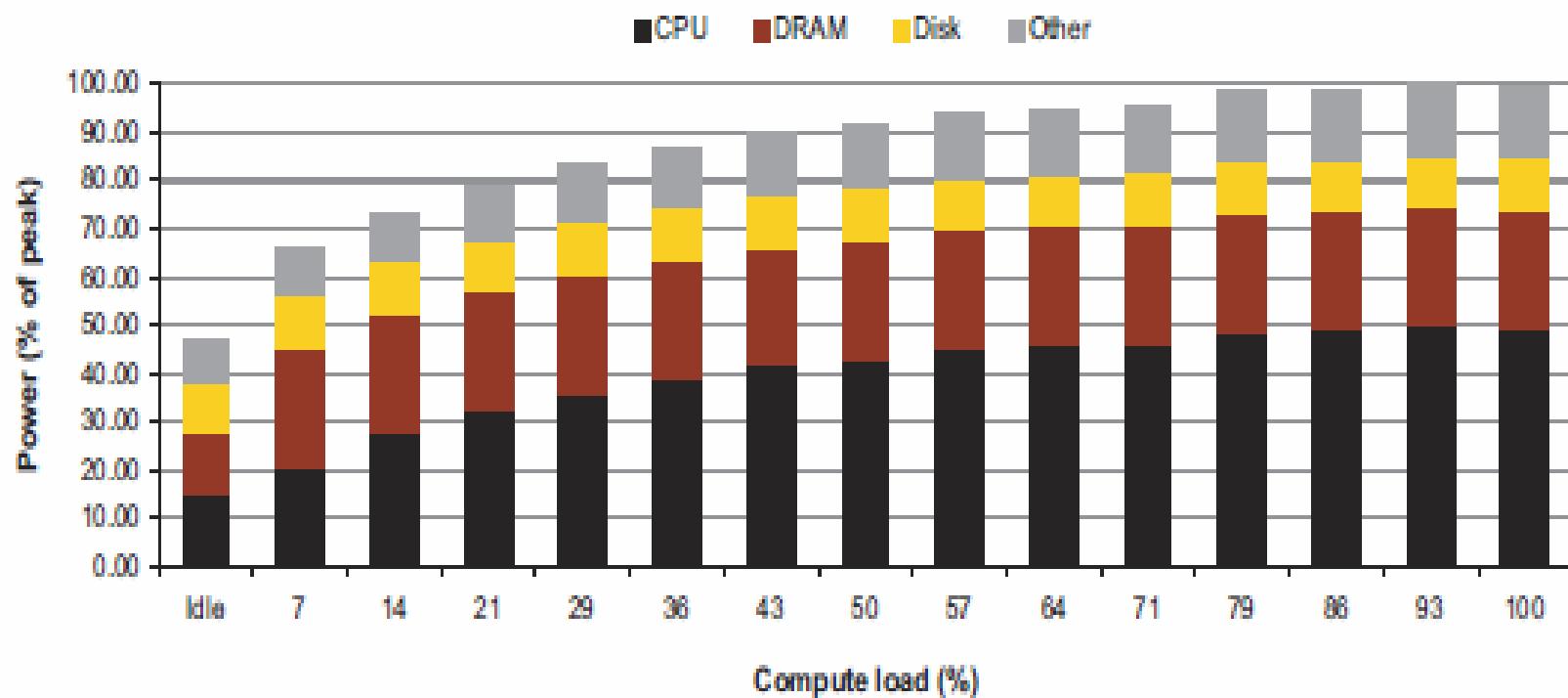
Memory Hierarchy Lessons

- Caches vastly impact performance
 - Cannot consider performance without considering memory hierarchy
- Actual performance of a simple program can be a complicated function of the architecture
 - Slight changes in the architecture or program change the performance significantly
 - To write fast programs, need to consider architecture
 - True on sequential or parallel processor
 - We would like simple models to help us design efficient algorithms

Memory Hierarchy Lessons

- Common technique for improving cache performance, called **blocking** or **tiling**
 - Idea: used divide-and-conquer to define a problem that fits in register/L1-cache/L2-cache
- Autotuning: Deal with complexity through experiments
 - Produce several different versions of code
 - Different algorithms, Blocking Factors, Loop orderings, etc
 - For each architecture, run different versions to see which is fastest
 - Can (in principle) navigate complex design options for optimum

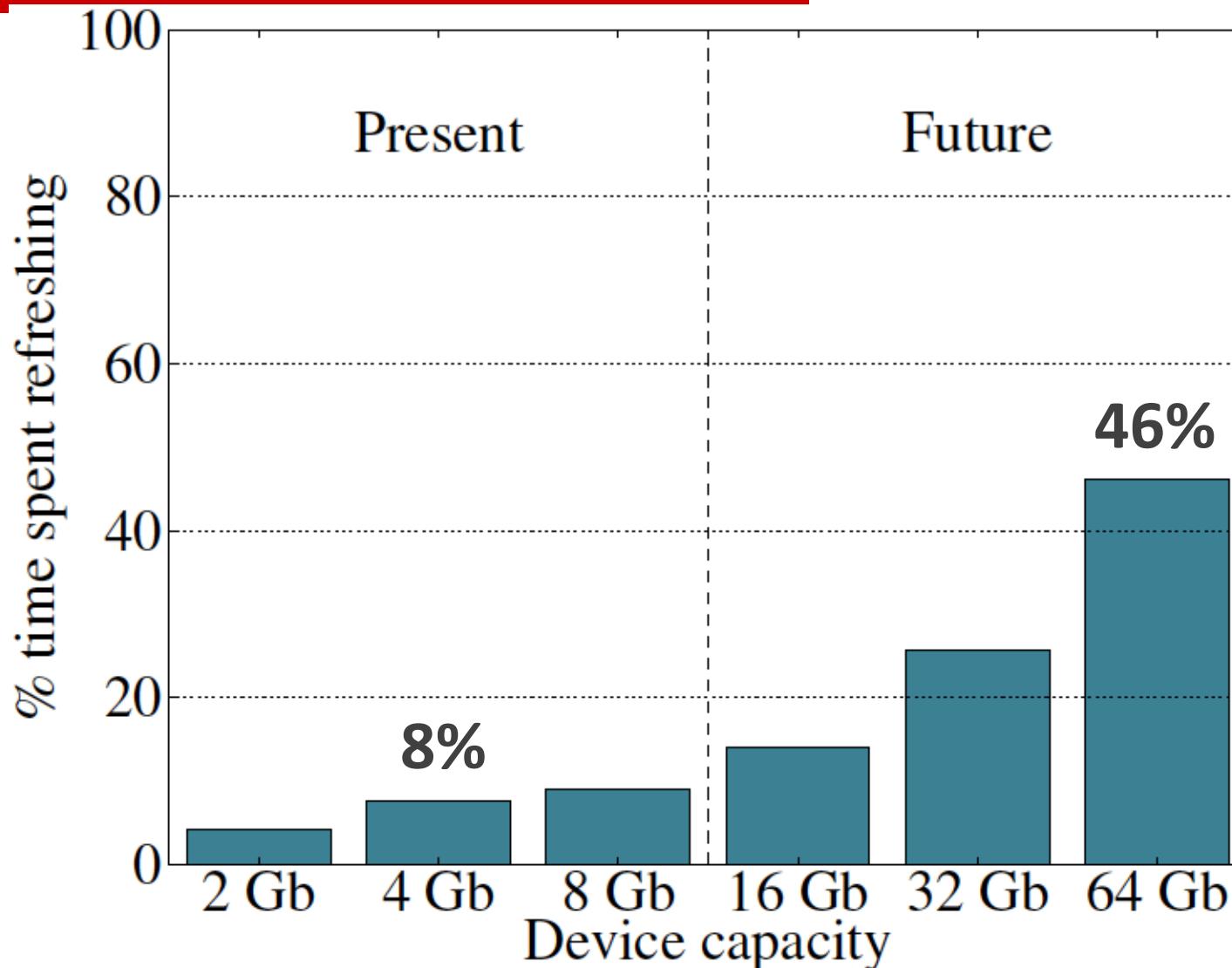
Energy Consumption in Modern Electronic ICT Systems



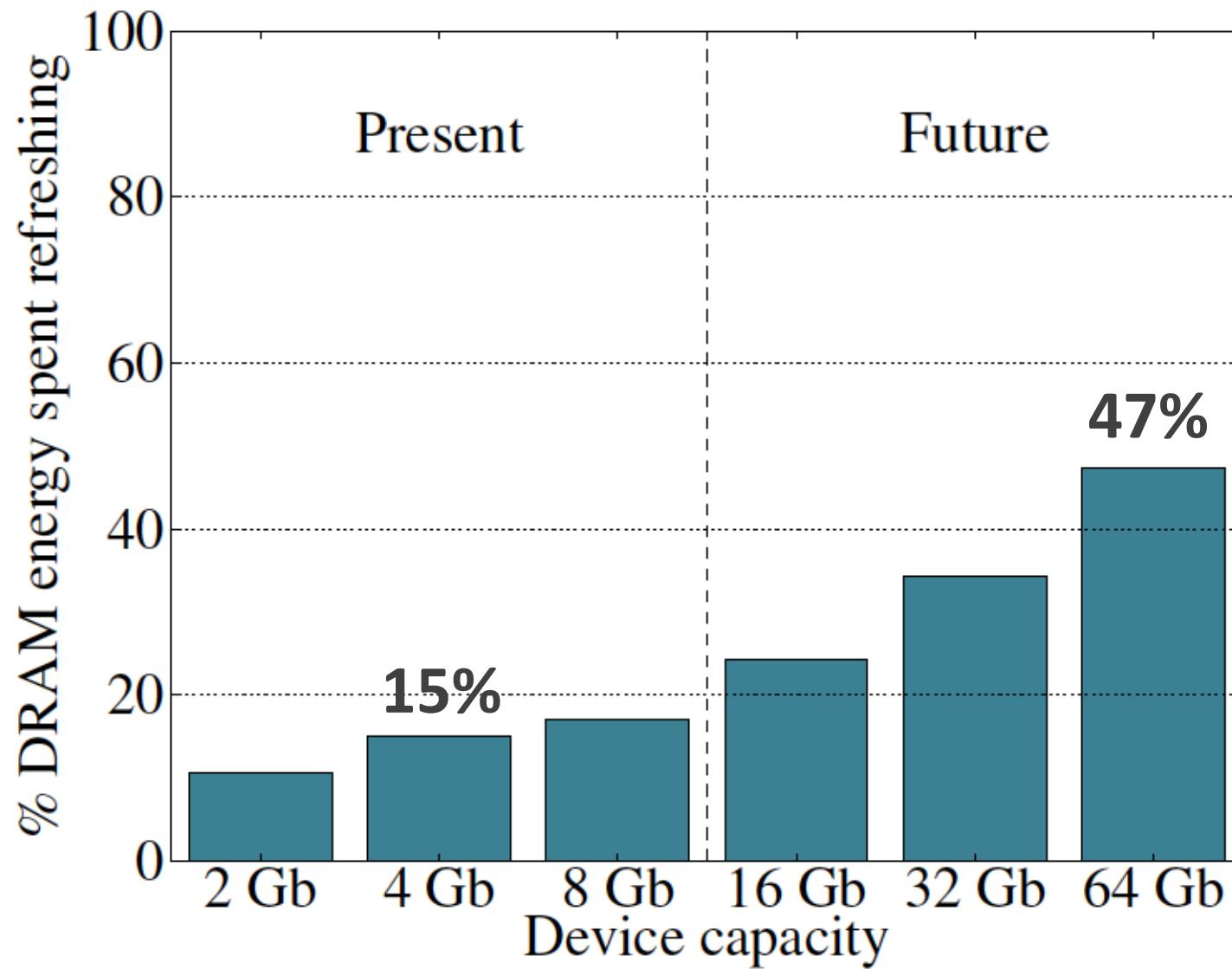
Downsides of DRAM Refresh

- Energy consumption: Each refresh consumes energy
- Performance degradation: DRAM bank unavailable while refreshed 性能下降
- QoS/predictability impact: (Long) pause times during refresh
- Refresh rate limits DRAM capacity scaling

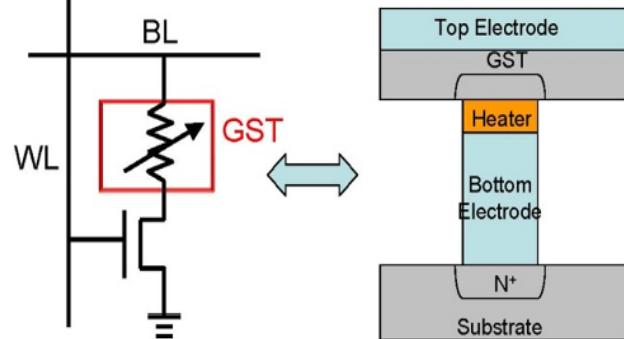
Refresh Overhead: Performance



Refresh Overhead: Energy

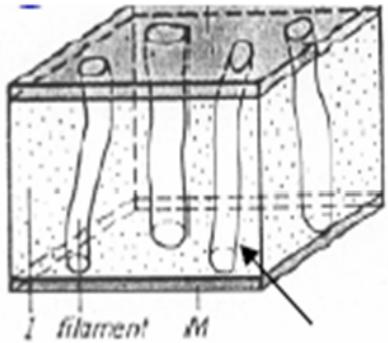


Emerging Non-volatile Memory (NVM) Technologies

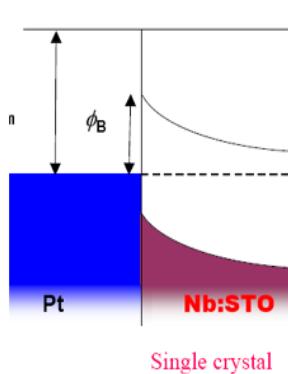


Phase-change RAM (**PCRAM**)

相变存储器

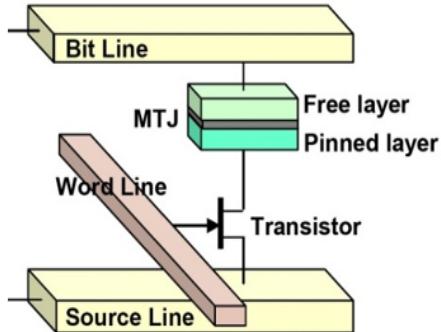


Vacuum

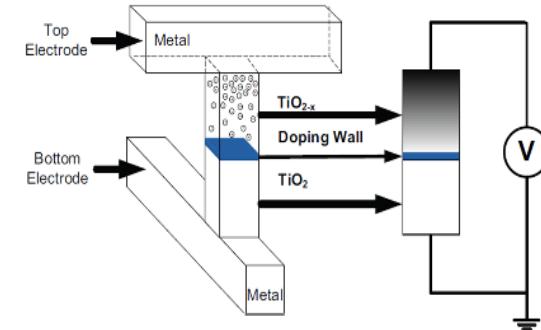


Resistive RAM (**RRAM**)

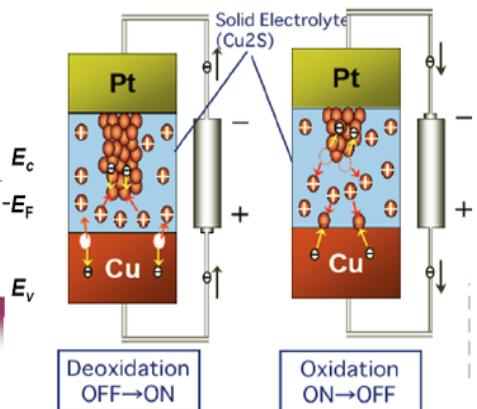
From left to right: Filament-based, Interface-based and PMC



Magnetic RAM (**MRAM**)



Domain wall motion in free layer
Pinned reference layer



Memristor 忆阻器

Top: thin-film device

Bottom: Spintronic material

Emerging Non-volatile Memory (NVM) Technologies

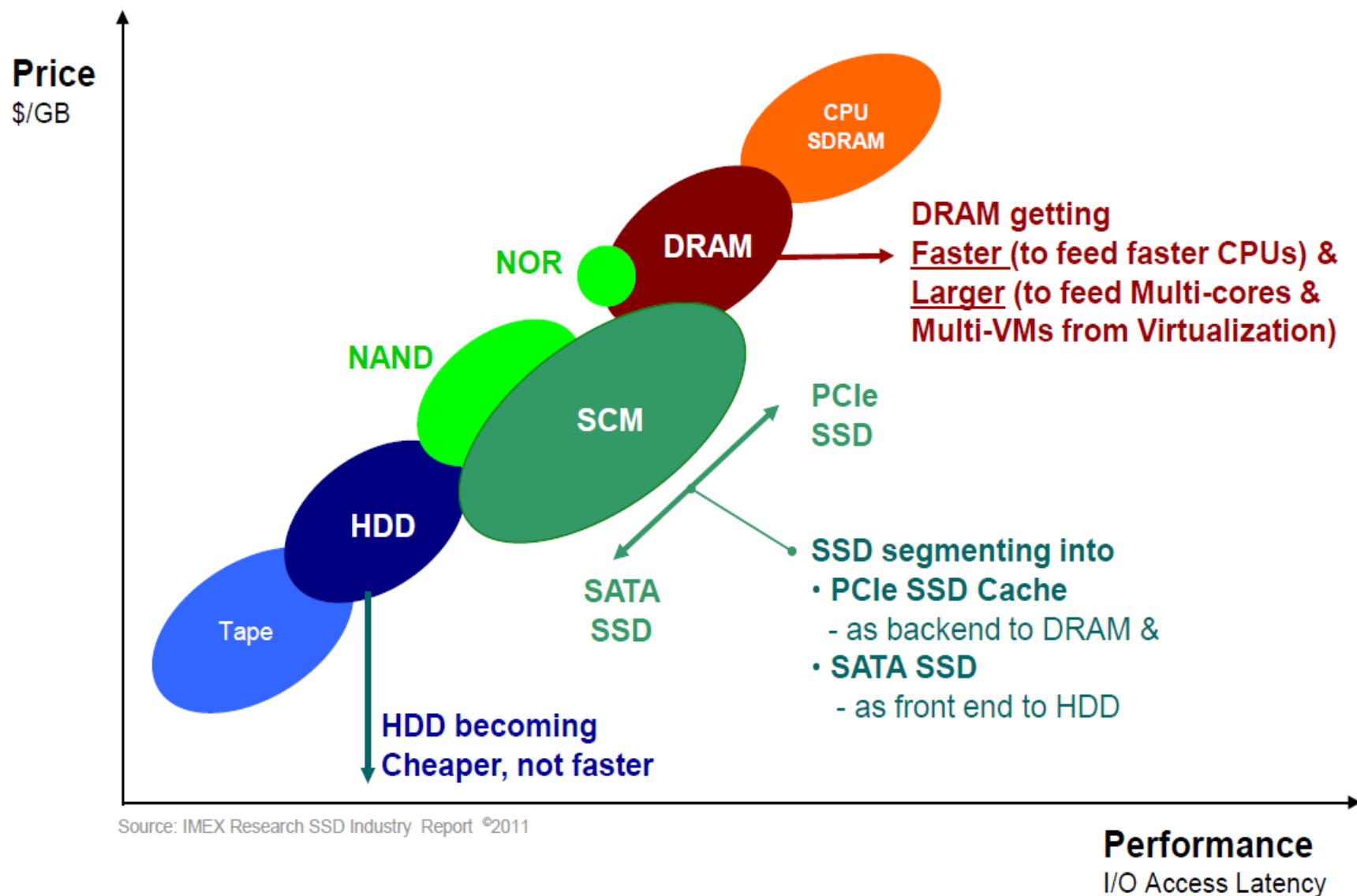
Comparison of Data Storage Technologies

	Memristor	PCM	STT-RAM	DRAM	Flash	HD
Chip area per bit (F^2)	4	8–16	14–64	6–8	4–8	n/a
Energy per bit (pJ) ²	0.1–3	2–100	0.1–1	2–4	10^1 – 10^4	10^6 – 10^7
Read time (ns)	<10	20–70	10–30	10–50	25,000	5 – 8×10^6
Write time (ns)	20–30	50–500	13–95	10–50	200,000	5 – 8×10^6
Retention	>10 years	<10 years	Weeks	<Second	~10 years	~10 years
Endurance (cycles)	$\sim 10^{12}$	10^7 – 10^8	10^{15}	$>10^{17}$	10^3 – 10^6	10^{15} ?
3D capability	Yes	No	No	No	Yes	n/a

New Advances on Memory and Storage

— Storage Class Memory

存储级内存



Intel-Micron 3D XPoint Technology

3D XPoint™ Technology: An Innovative, High-Density Design

Cross Point Structure

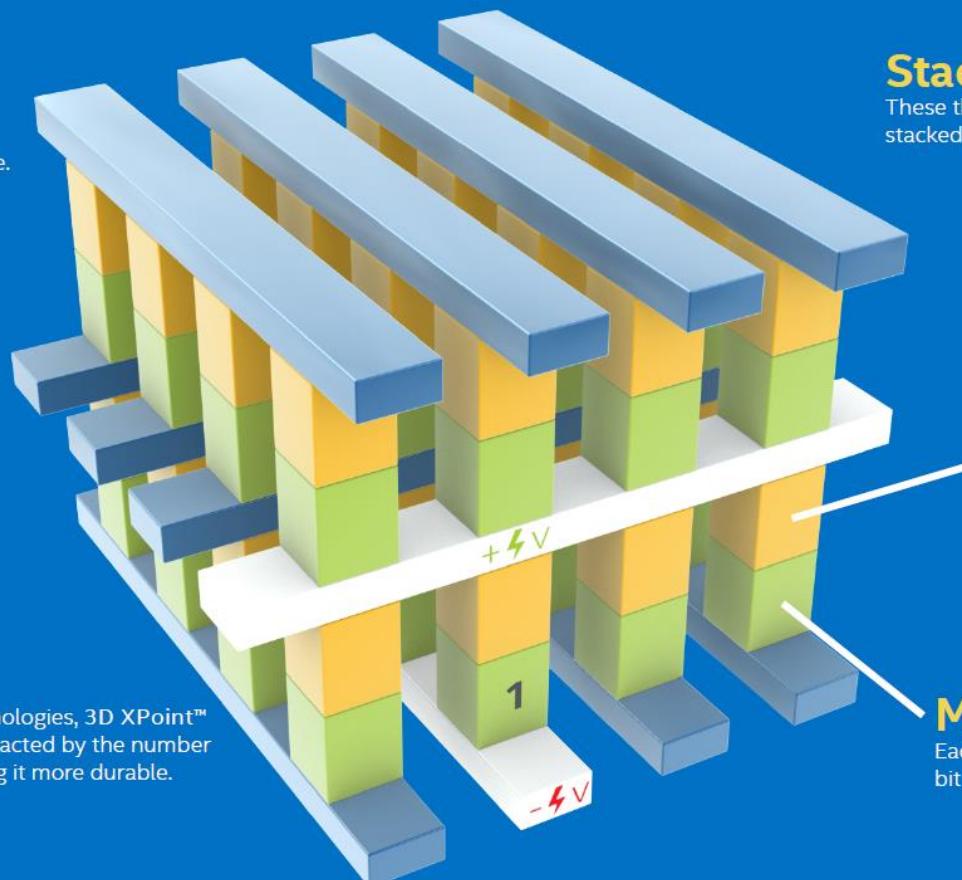
Perpendicular wires connect submicroscopic columns. An individual memory cell can be addressed by selecting its top and bottom wire.

Non-Volatile

3D XPoint™ Technology is non-volatile—which means your data doesn't go away when your power goes away—making it a great choice for storage.

High Endurance

Unlike other storage memory technologies, 3D XPoint™ Technology is not significantly impacted by the number of write cycles it can endure, making it more durable.



Stackable

These thin layers of memory can be stacked to further boost density.

Selector

Whereas DRAM requires a transistor at each memory cell—making it big and expensive—the amount of voltage sent to each 3D XPoint™ Technology selector enables its memory cell to be written to or read without requiring a transistor.

Memory Cell

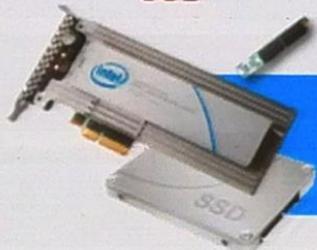
Each memory cell can store a single bit of data.

Intel Non-Volatile Memory Solutions

Intel Non-Volatile Memory Solutions

2014

2D-NAND
SATA/PCIe NVMe
SSD



2015/2016

Intel® Optane™
3D XPoint™ NVM
PCIe NVMe SSD
“Coldstream”



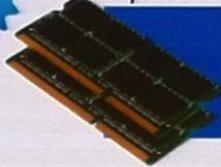
NEW

3D NAND
256Gbit
“Cliffdale”

NEW

2016 / 2017+

Intel® DIMM
3D XPoint™ NVM DIMM
“Apache Pass”



- Highest Read/Write GB/s (seq.)
- Highest IOPS (4K random reads)
- Lowest latency, highest endurance

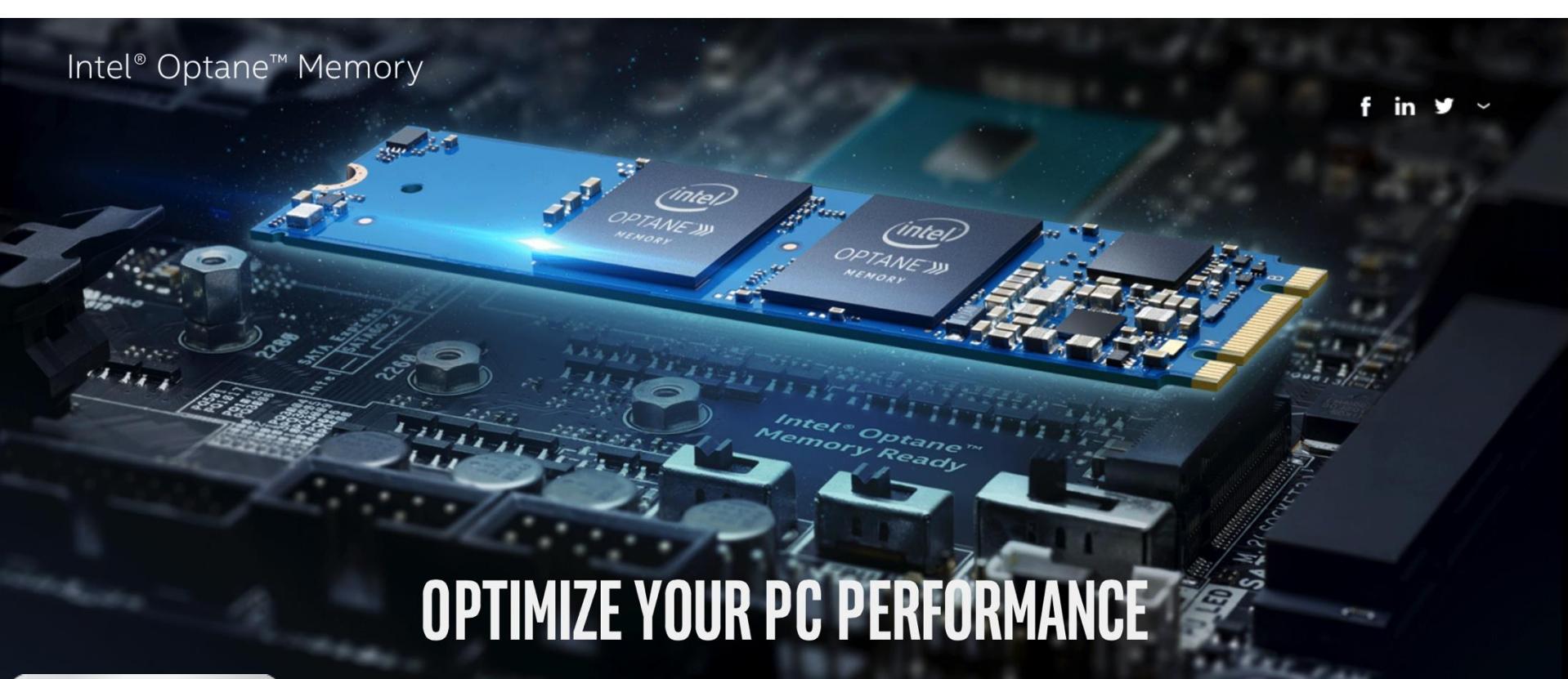
can be used as normal memory (byte addr) or SSD (block I/O)

- Up to 2.5GB/s read/write (seq.) PCIe NVMe, up to 625K IOPS
- Higher capacity & availability, more serviceable than DIMMs
- Higher performance & endurance, lower latency than NAND

- Up to 2.8GB/s read, 1.9GB/s write (seq.)
- Up to 450K IOPS (4K random reads)
- Highest SSD capacity from Intel

All products, dates, features and figures are preliminary and are subject to change without any notice.

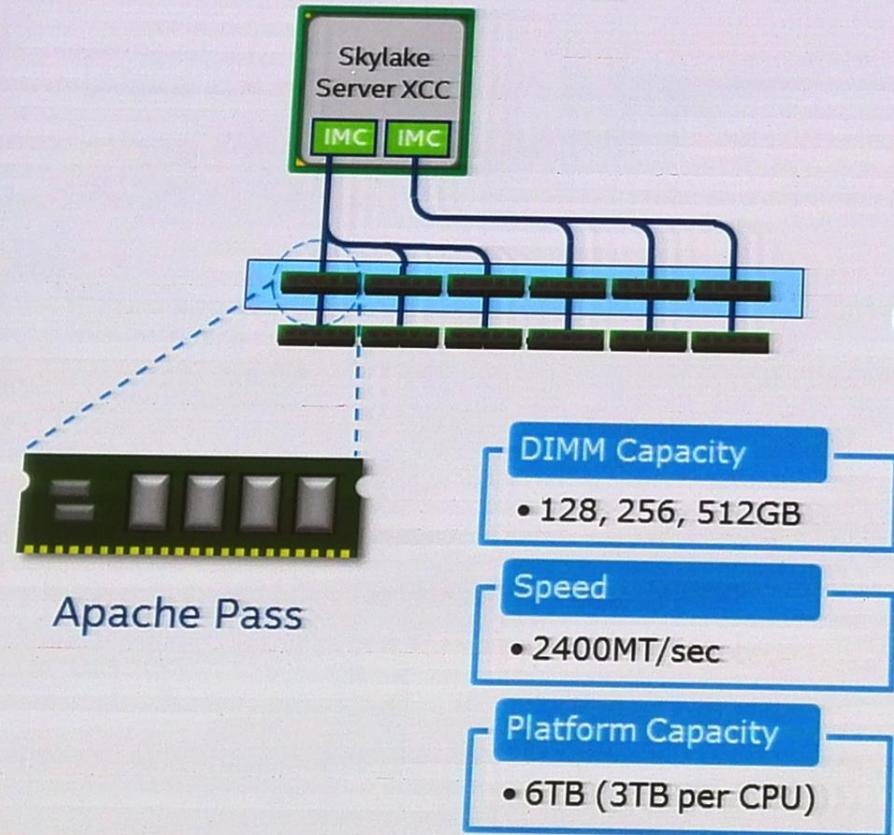
Intel Non-Volatile Memory Solutions



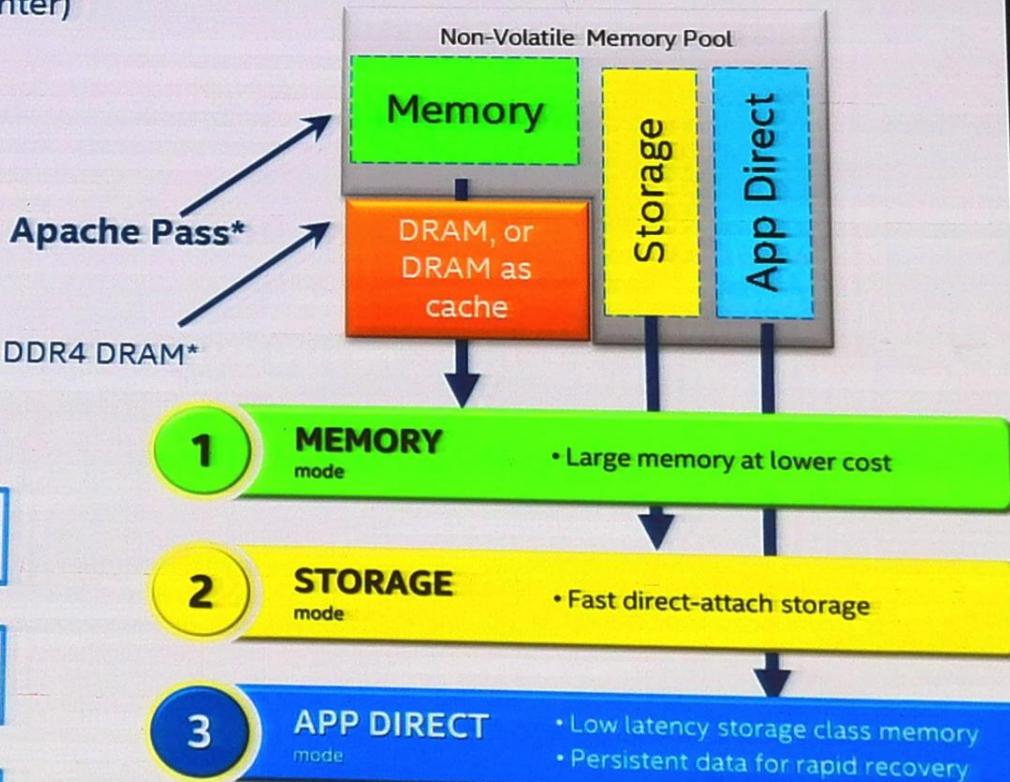
Intel Non-Volatile Memory Solutions

Apache Pass Overview

(3D XPoint™ based Memory Module for the Data Center)

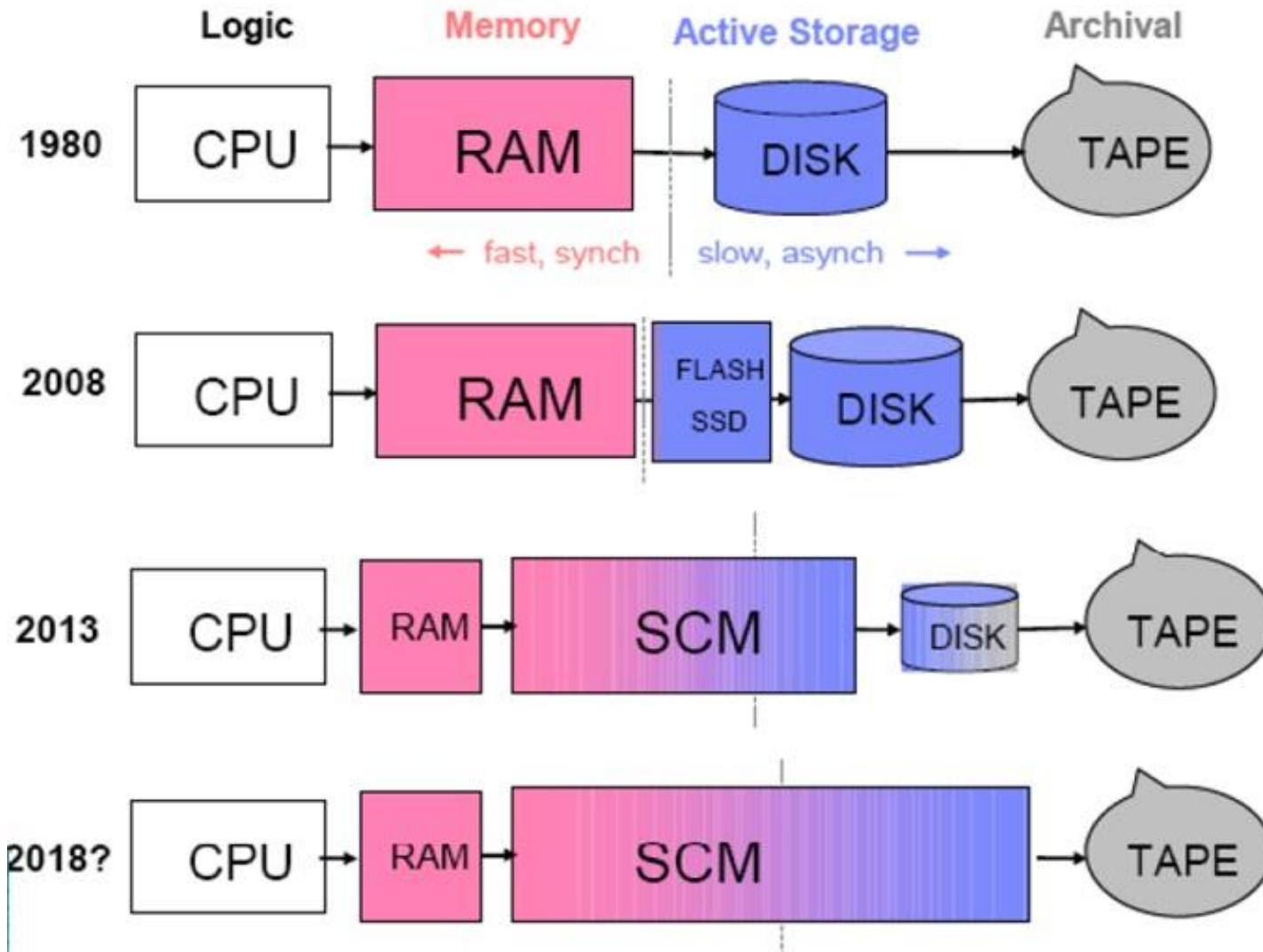


Flexible, Usage Specific Partitions

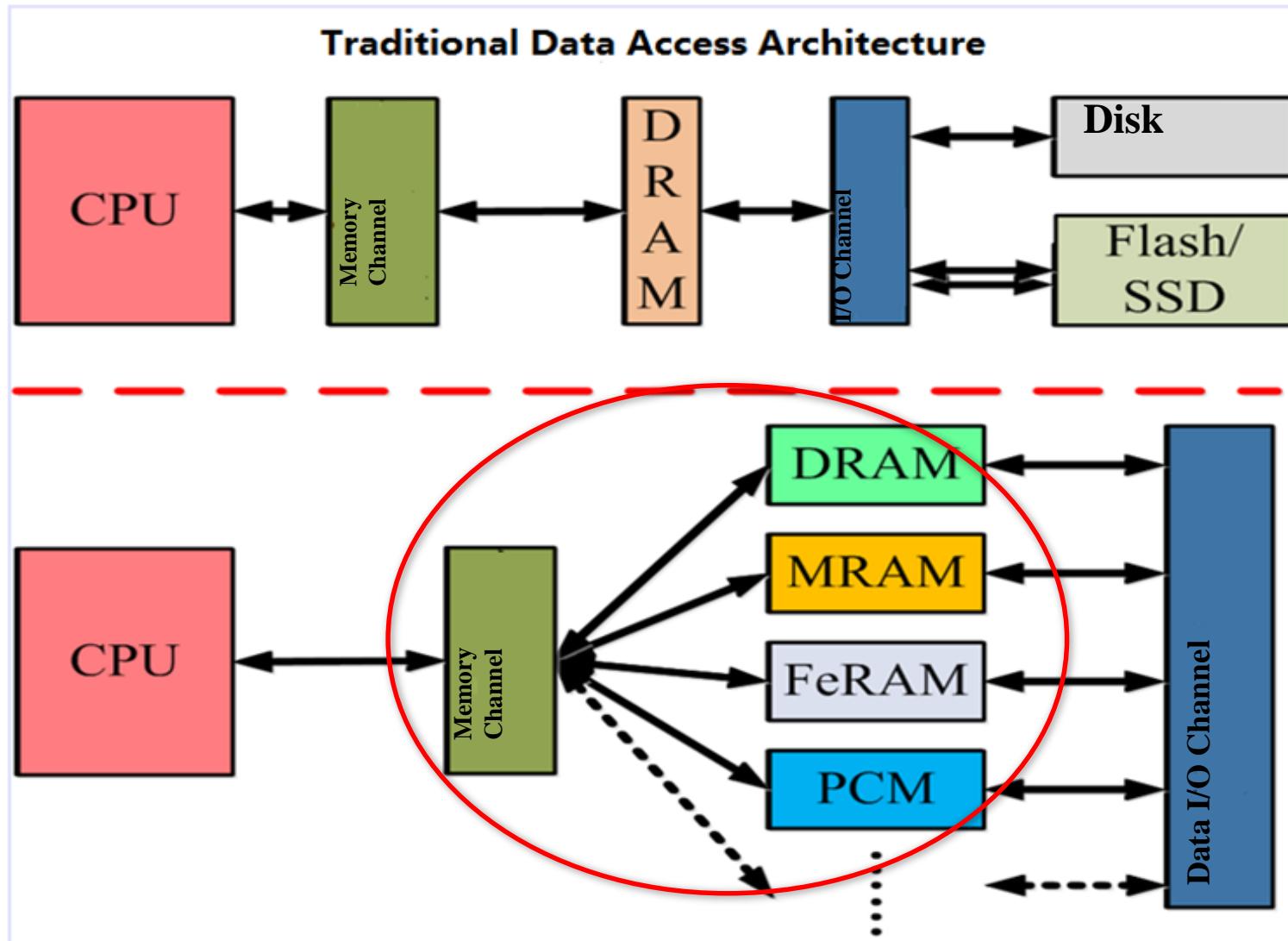


* DIMM population shown as an example only.

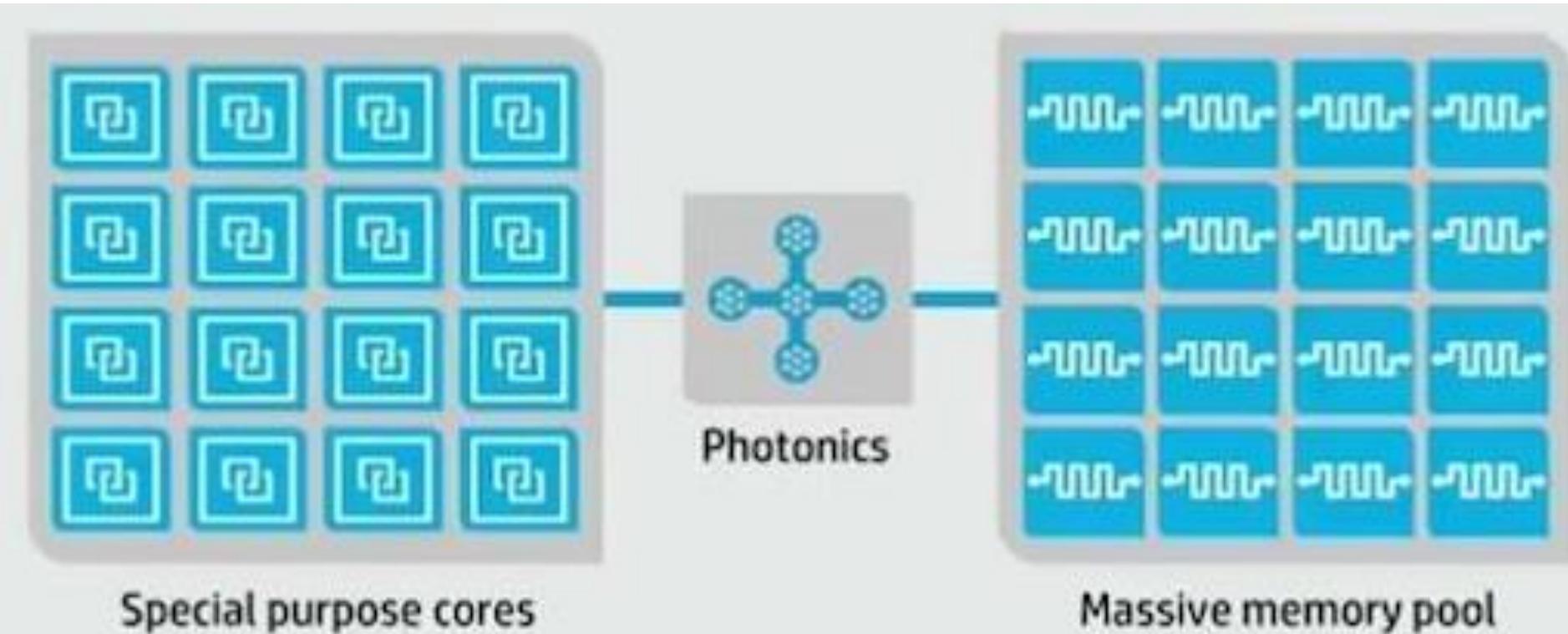
Reconstruction of Virtual Memory Architecture: Break the I/O Bottleneck



New In-Memory Computing Architecture



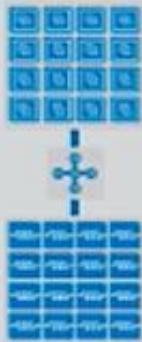
HP: The Machine



The Machine



The Machine – Future



2015

- Memristors begin sampling
- Physical infrastructure of Core prototypes established
- Silicon Scale Integration discipline established across HP and supply chain
- Open Source Machine OS SDK released
- ISV Partner collaborations begin

2017

- Edge devices begin sampling
- Machine OS enters public beta

2019

- Core devices ship
- Machine available as product, service and as a business process transformation

2014

- Memristor media controller, protocols and standards established
- SoC Partners selected for co-development
- Machine OS development begins

2016

Memristor DIMMs launched
ISV partners collaborations ship as services

2018

- Edge devices ship in volume
- Core Machine POCs at scale
- Machine OS released

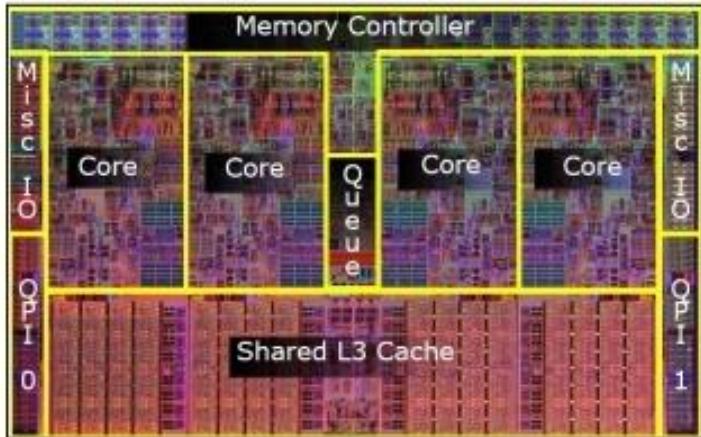
2020

Distributed mesh compute goes mainstream

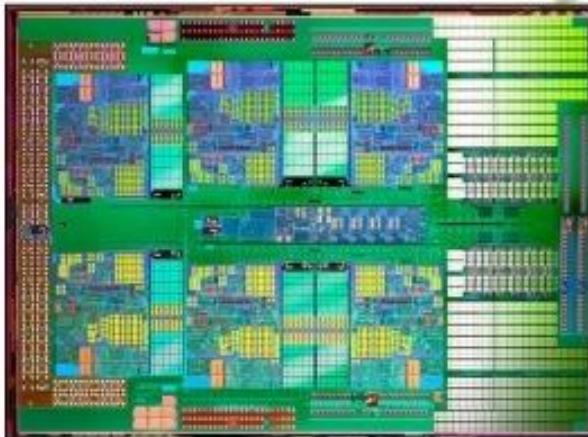
Parallel architecture

MULTICORE CHIPS

Parallel Chip-Scale Processors



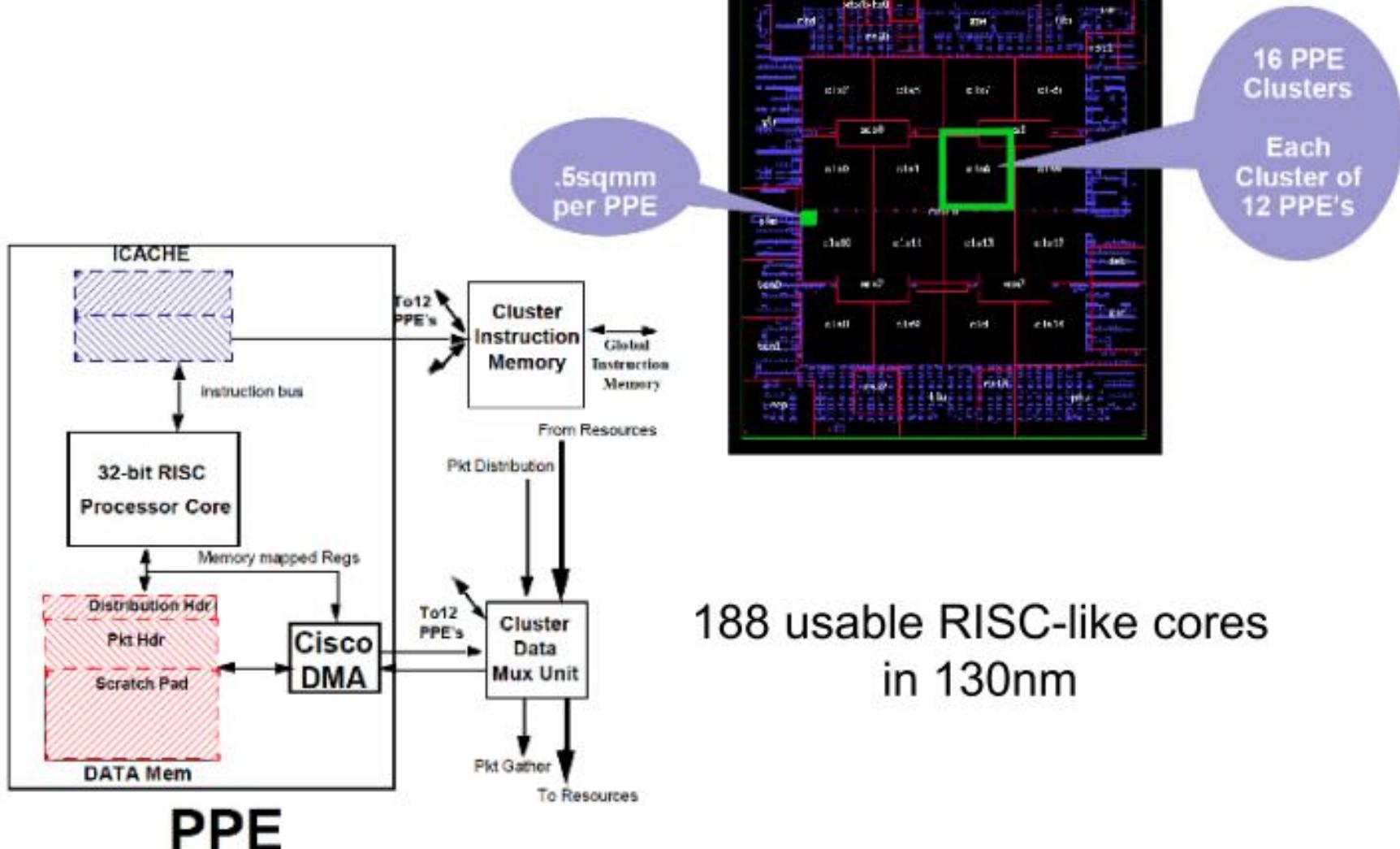
Intel Core 2 Quad: 4 Cores



AMD Opteron: 6 Cores

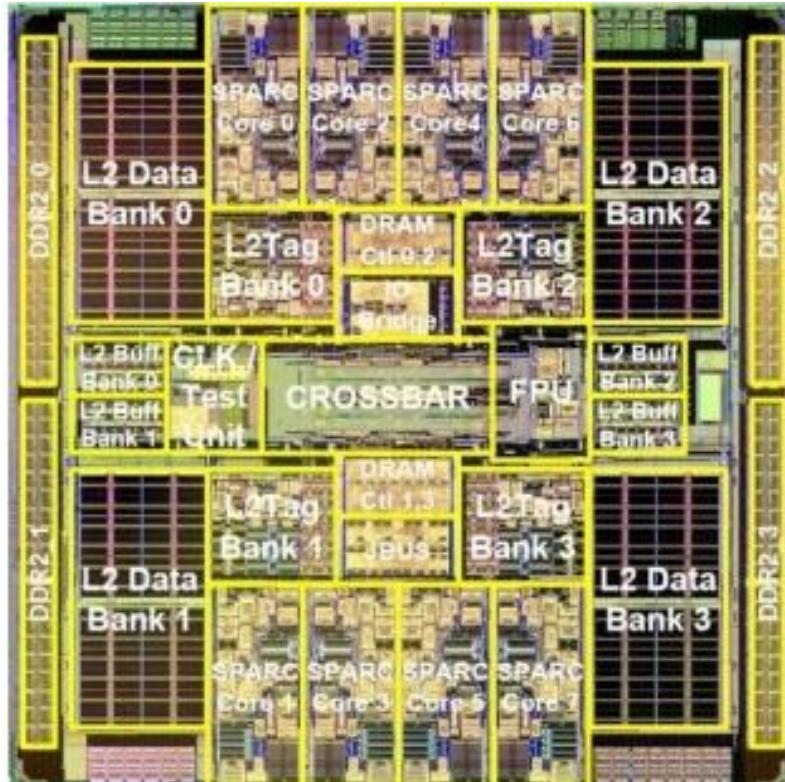
- Multicore processors emerging in general-purpose market due to power limitations in single-core performance scaling
 - Multiple cores connected as cache-coherent SMP
 - Cache-coherent shared memory
- Embedded applications need large amounts of computation
 - Recent trend to build “extreme” parallel processors with dozens to hundreds of parallel processing elements on one die
 - Often connected via on-chip networks, with no cache coherence
 - Examples: 188 core “Metro” chip from CISCO

Cisco CSR-1 Metro Chip



Sun's T1 (“Niagara”)

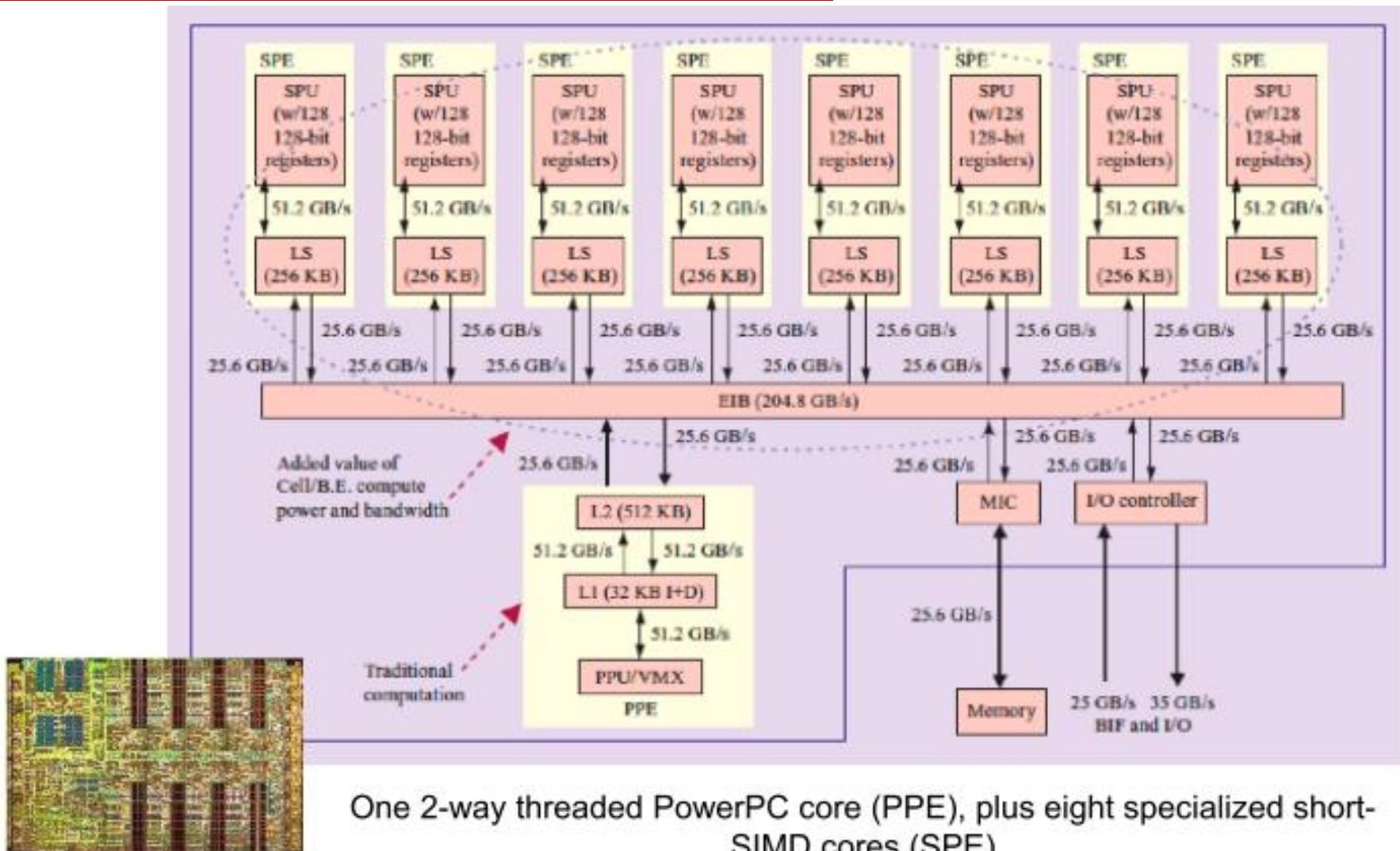
- Highly Threaded
 - 8 Cores
 - 4 Threads/Core
- Target: Commercial server applications
 - High thread level parallelism (TLP)
 - Large numbers of parallel client requests
 - Low instruction level parallelism (ILP)
 - High cache miss rates
 - Many unpredictable branches
 - Frequent load-load dependencies
- Power, cooling, and space are major concerns for data centers
- Metric: Performance/Watt/Sq. Ft.
- Approach: Multicore, Fine-grain multithreading, Simple pipeline, Small L1 caches, Shared L2



Embedded Parallel Processors

- Often embody a mixture of old architectural styles and ideas
- Exposed memory hierarchies and interconnection networks
 - Programmers code to the “metal” to get best cost/power/performance
 - Portability across platforms less important
- Customized synchronization mechanisms
 - Interlocked communication channels (processor blocks on read if data not ready)
 - Barrier signals
 - Specialized atomic operation units
- Many more, simpler cores

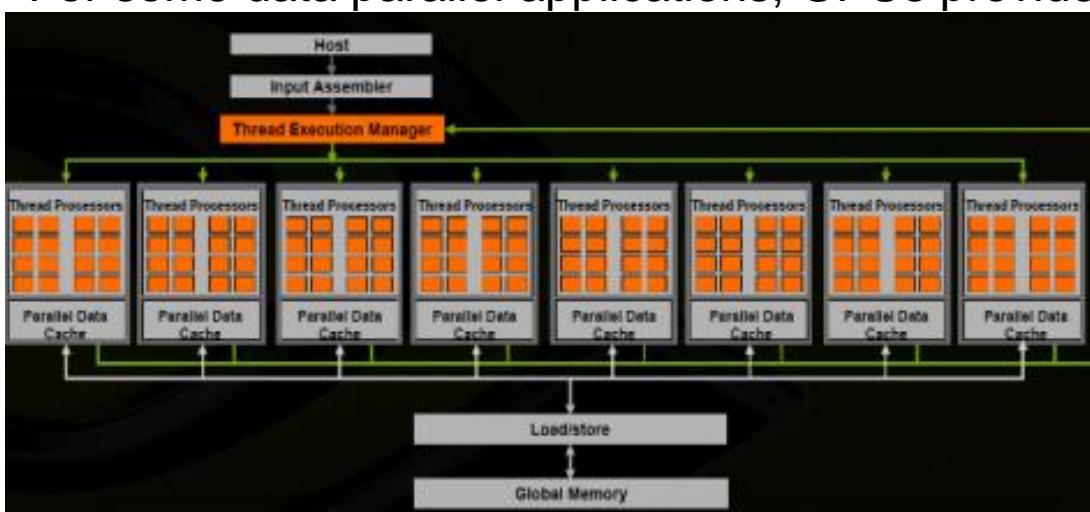
IBM Cell Processor (Playstation-3)



One 2-way threaded PowerPC core (PPE), plus eight specialized short-SIMD cores (SPE)

Nvidia G8800 Graphics Processor

- This is a GPU (Graphics Processor Unit)
 - Available in many desktops
- Example: 16 cores similar to a vector processor with 8 lanes (128 stream processors total)
 - Processes threads in SIMD groups of 32 (a “warp”)
 - Some stripmining done in hardware
- Threads can branch, but loses performance compared to when all threads are running same code
- Complete parallel programming environment (CUDA)
 - A lot of parallel codes have been ported to these GPUs
 - For some data parallel applications, GPUs provide the fastest implementations



Nvidia Fermi GF100 GPU

[Nvidia, 2010]



Nvidia Tesla GPU

Features	Tesla K80 ¹	Tesla K40
GPU	2x Kepler GK210	1 Kepler GK110B
Peak double precision floating point performance	2.91 Tflops (GPU Boost Clocks) 1.87 Tflops (Base Clocks)	1.66 Tflops (GPU Boost Clocks) 1.43 Tflops (Base Clocks)
Peak single precision floating point performance	8.74 Tflops (GPU Boost Clocks) 5.6 Tflops (Base Clocks)	5 Tflops (GPU Boost Clocks) 4.29 Tflops (Base Clocks)
Memory bandwidth (ECC off) ²	480 GB/sec (240 GB/sec per GPU)	288 GB/sec
Memory size (GDDR5)	24 GB (12GB per GPU)	12 GB
CUDA cores	4992 (2496 per GPU)	2880

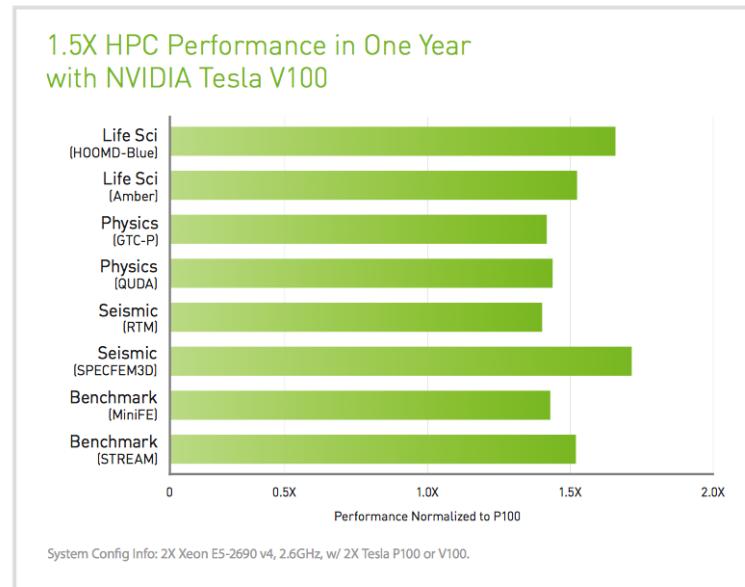
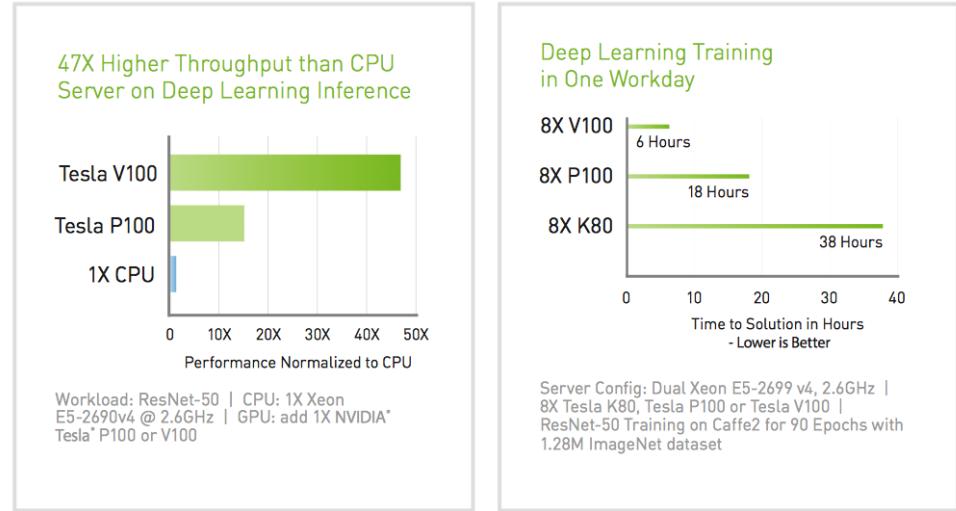
1 Tesla K80 specifications are shown as aggregate of two GPUs.

2 With ECC on, 6.25% of the GPU memory is used for ECC bits. For example, 6 GB total memory yields 5.25 GB of user available memory with ECC on.



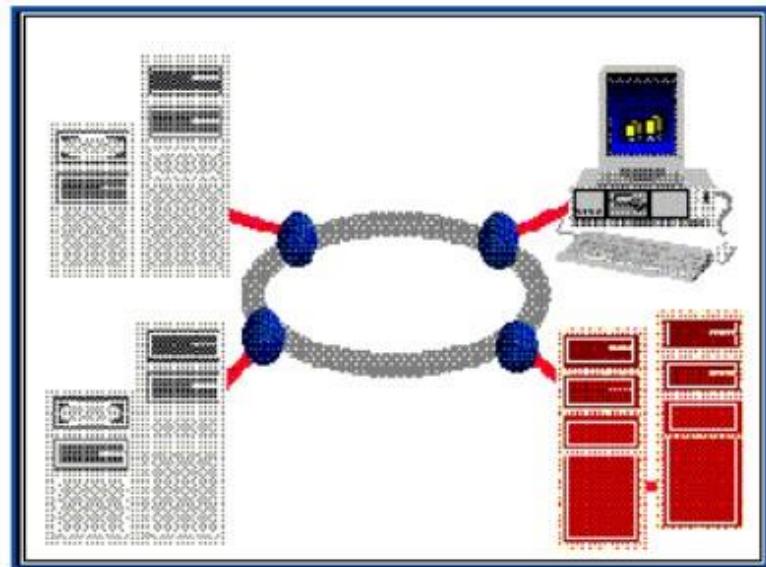
Nvidia Tesla V100 GPU

	Tesla V100 PCIe	Tesla V100 SXM2
NVIDIA Volta		
GPU Architecture		
NVIDIA Tensor Cores	640	
NVIDIA CUDA® Cores	5,120	
Double-Precision Performance	7 TFLOPS	7.8 TFLOPS
Single-Precision Performance	14 TFLOPS	15.7 TFLOPS
Tensor Performance	112 TFLOPS	125 TFLOPS
GPU Memory	16 GB HBM2	
Memory Bandwidth	900 GB/sec	
ECC	Yes	
Interconnect Bandwidth	32 GB/sec	300 GB/sec
System Interface	PCIe Gen3	NVIDIA NVLink
Form Factor	PCIe Full Height/Length	SXM2
Max Power Consumption	250 W	300 W
Thermal Solution	Passive	
Compute APIs	CUDA, DirectCompute, OpenCL™, OpenACC	



The New Wave

- The rate of technological progress for networking is an astounding 10-fold increase every 4 years (77.8% yearly compound rate)



- The emergence of network-centric computing (as opposed to processor-centric) –distributed high performance/throughput computing

Parallel architecture

WHAT IS PARALLEL ARCHITECTURE

What is Parallel Architecture?

- Machines with multiple processors



Intel's Quad Core i7



Apple MacPro



Blacklight at the PSC (4096 cores)



Hadoop cluster at Yahoo!

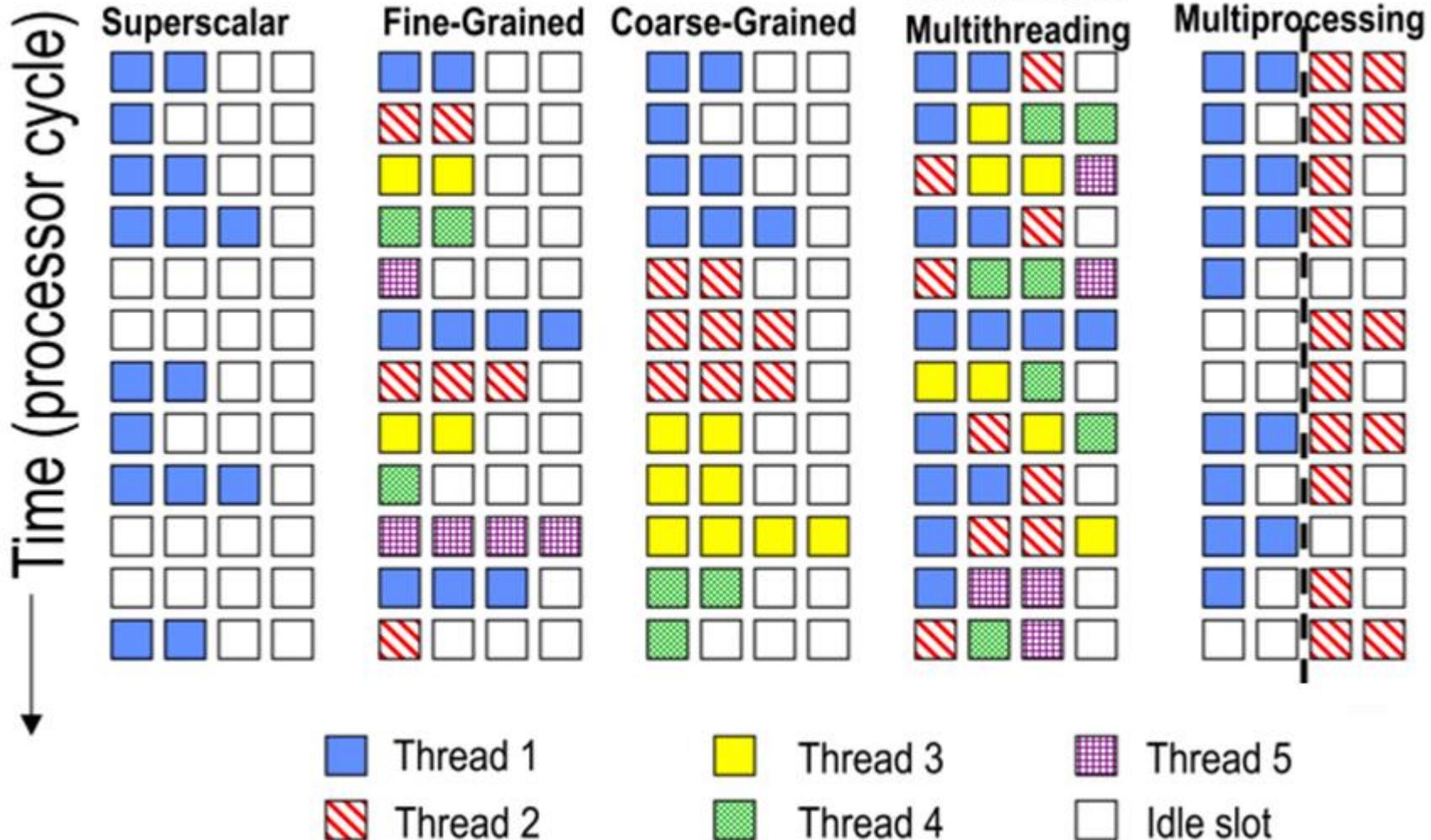
One Definition of Parallel Architecture

A parallel computer is a collection of processing elements that cooperate to solve large problems fast

Some broad issues

- Resource Allocation
 - how large a collection?
 - how powerful are the elements?
 - how much memory?
- Data access, Communication, and Synchronization
 - how do the elements cooperate and communicate?
 - how are data transmitted between processors?
 - what are the abstractions and primitives for cooperation?
- Performance and Scalability
 - how does it all translate into performance?
 - how does it scale?

Types of Parallelism



Parallel architecture

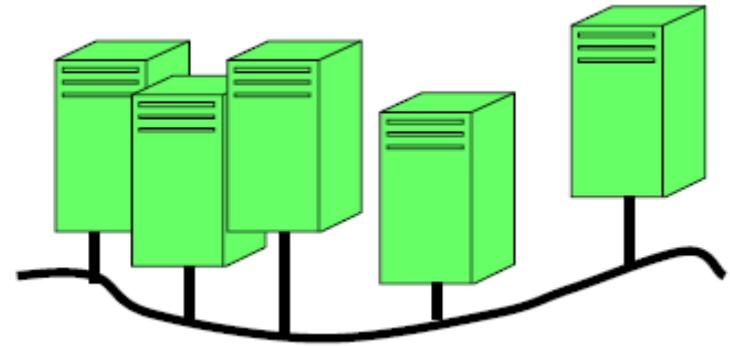
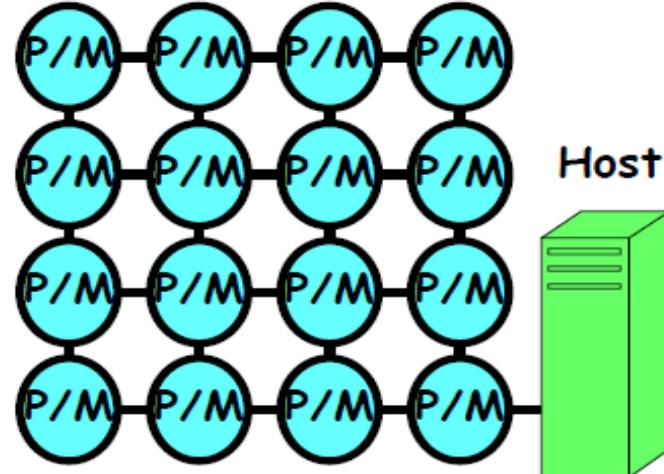
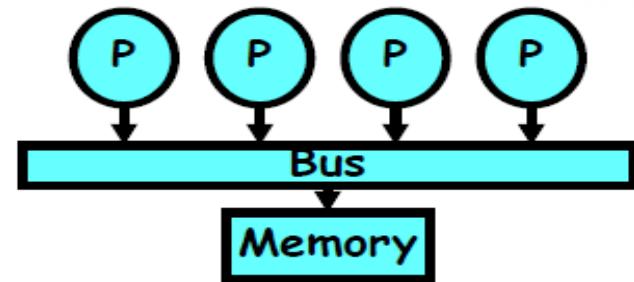
A PARALLEL ZOO OF ARCHITECTURES

MIMD Machines

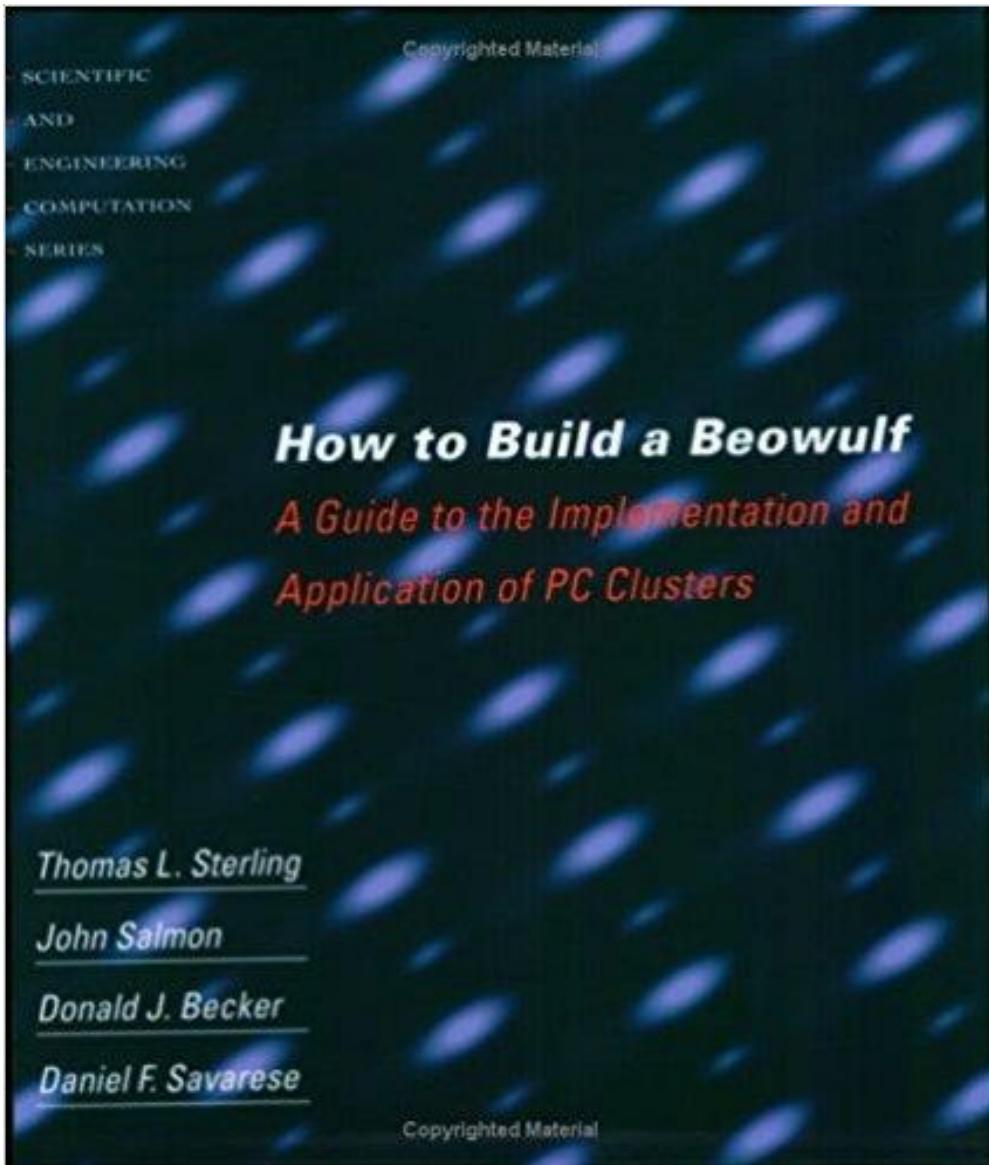
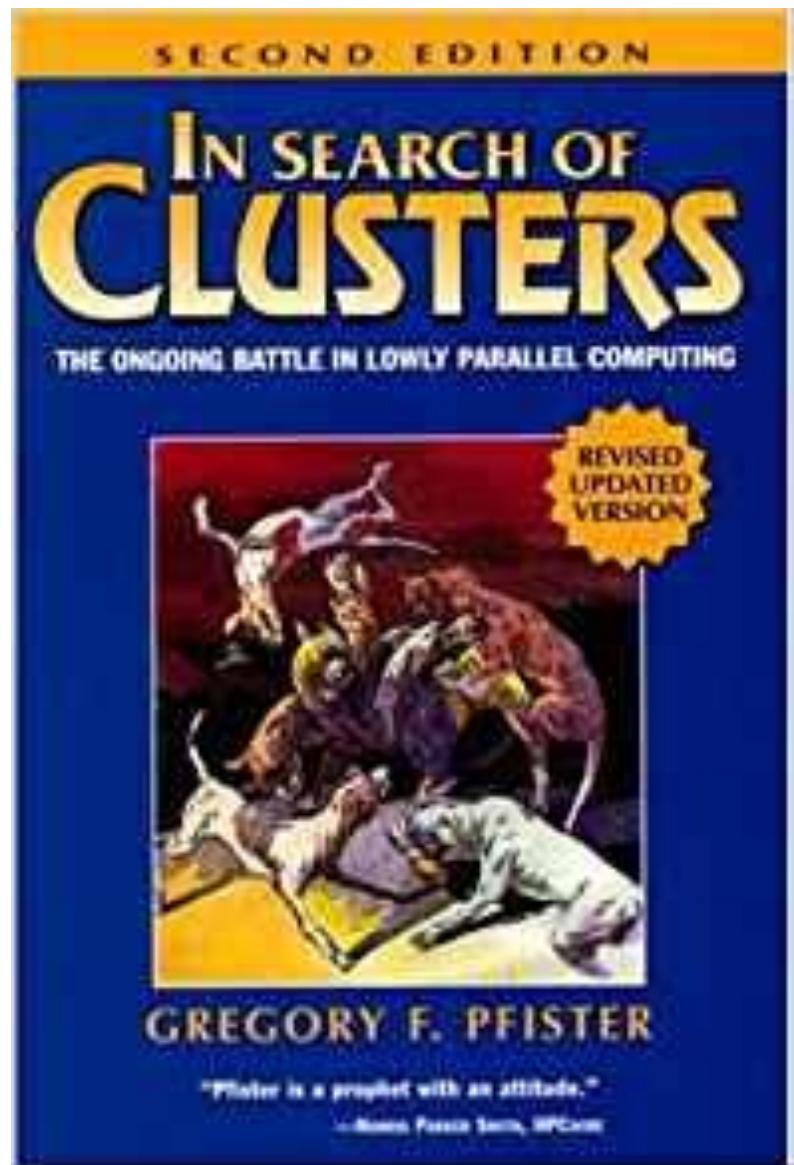
- Multiple Instruction, Multiple Data (MIMD)
 - Multiple independent instruction streams, program counters, etc
 - Called “multiprocessing” instead of “multithreading”
 - Although, each of the multiple processors may be multithreaded
 - When independent instruction streams confined to single chip, becomes a “multicore” processor
- Shared memory: Communication through Memory
 - Option 1: no hardware global cache coherence
 - Option 2: hardware global cache coherence
- Message passing: Communication through Messages
 - Applications send explicit messages between nodes in order to communicate
- For most machines, shared memory built on top of message-passing network
 - Bus-based machines are “exception”

Examples of MIMD Machines

- Symmetric Multiprocessor
 - Multiple processors in box with shared memory communication
 - Current MultiCore chips like this
 - Every processor runs copy of OS
- Non-uniform shared-memory with separate I/O through host
 - Multiple processors
 - Each with local memory
 - general scalable network
 - Extremely light “OS” on node provides simple services
 - Scheduling/synchronization
 - Network-accessible host for I/O
- Cluster
 - Many independent machine connected with general network
 - Communication through messages



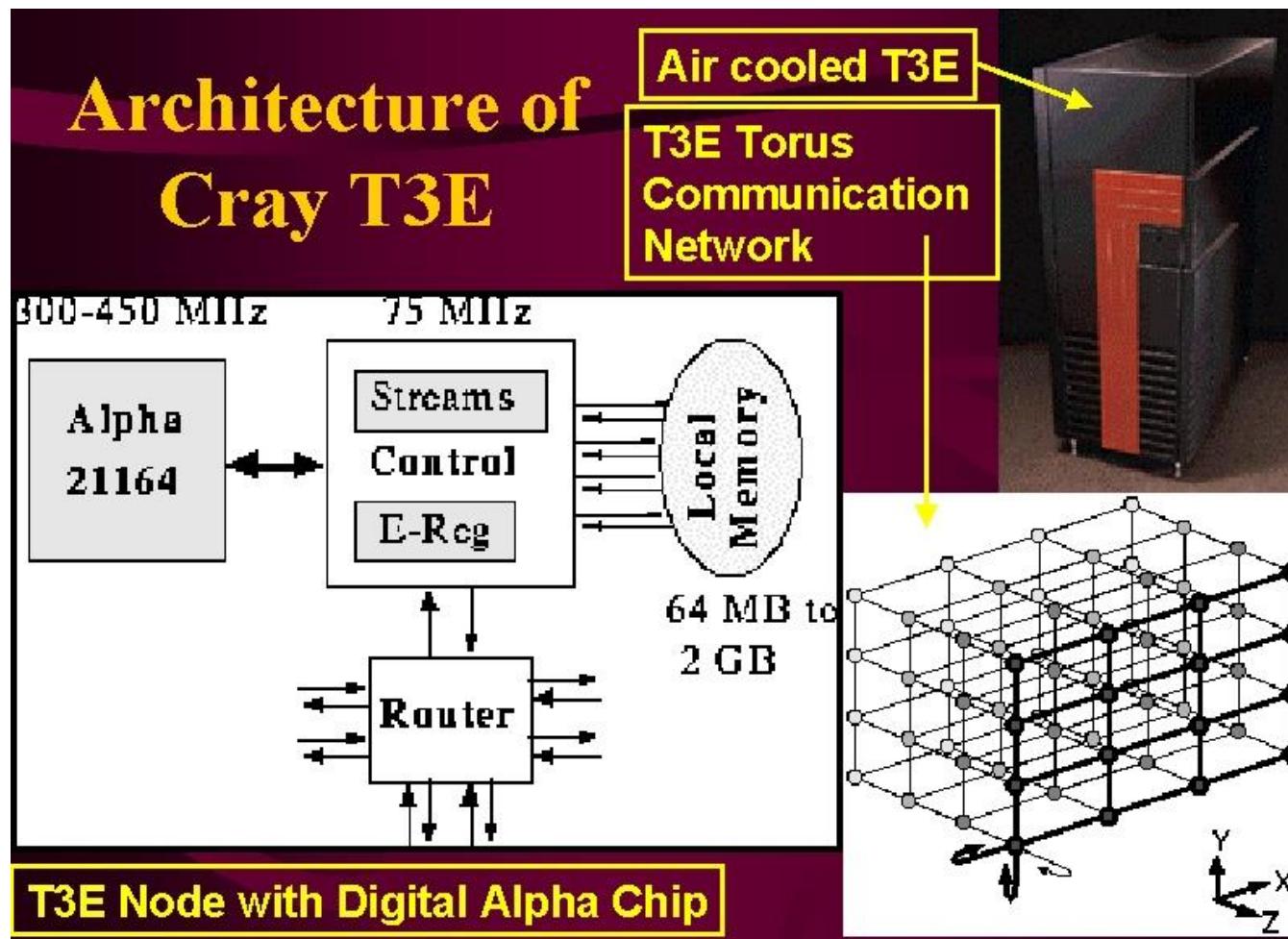
Cluster



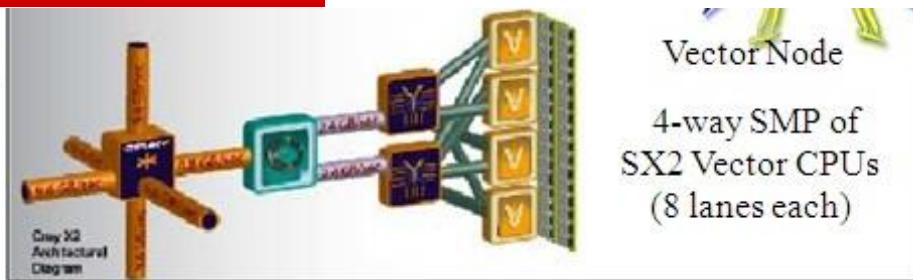
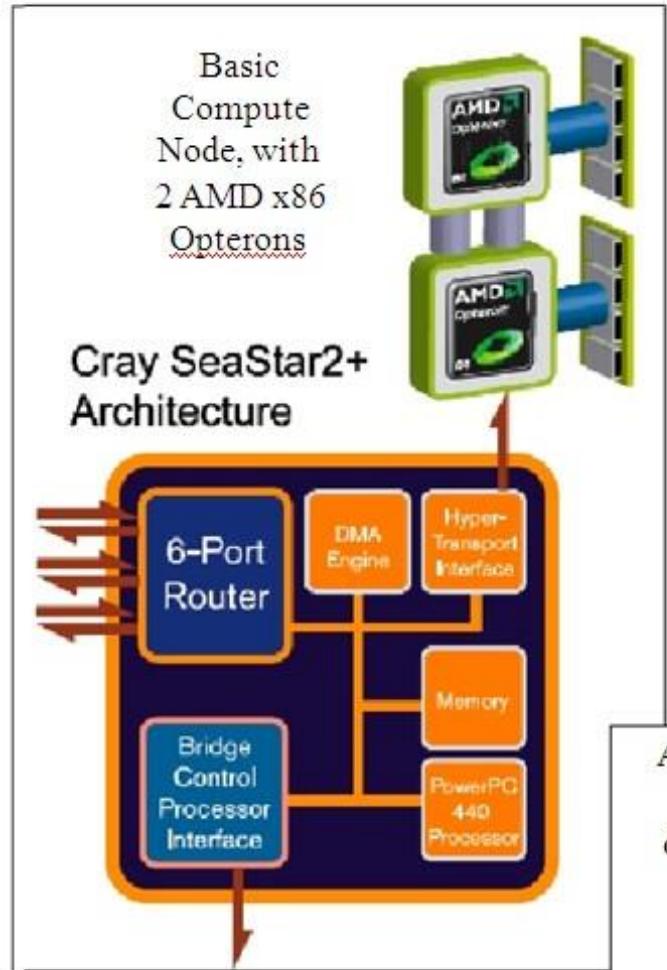
Cray T3E (1996)

Follow-on to earlier T3D (1993) using 21064's

Up to 2,048 675MHz Alpha 21164 processors connected in 3D torus



Cray XT5 (2007)



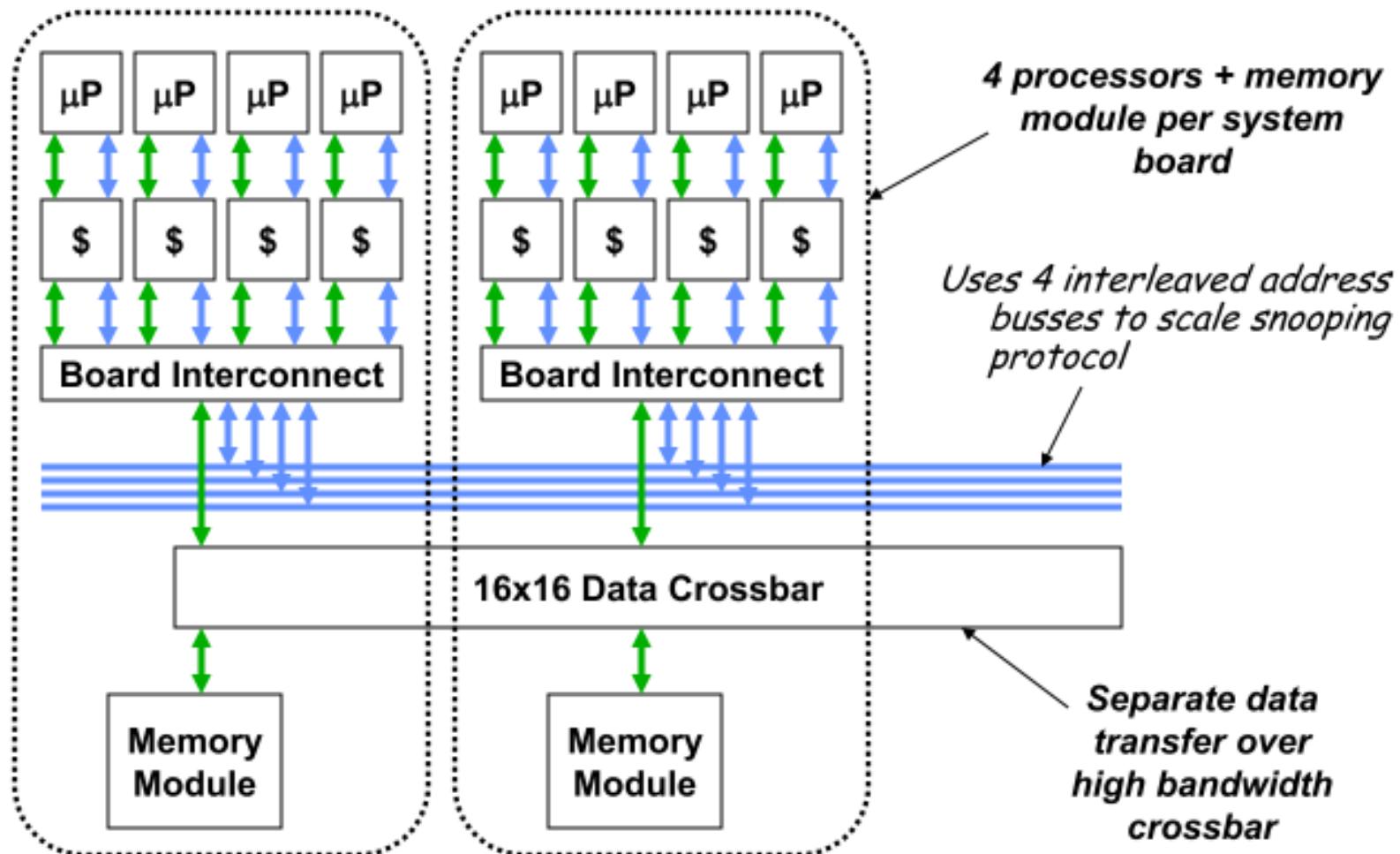
Also, XMT Multithreaded Nodes based on MTA design (128 threads per processor)

Processor plugs into



Sun Starfire UE10000 (1997)

Up to 64-way SMP using bus-based snooping protocol

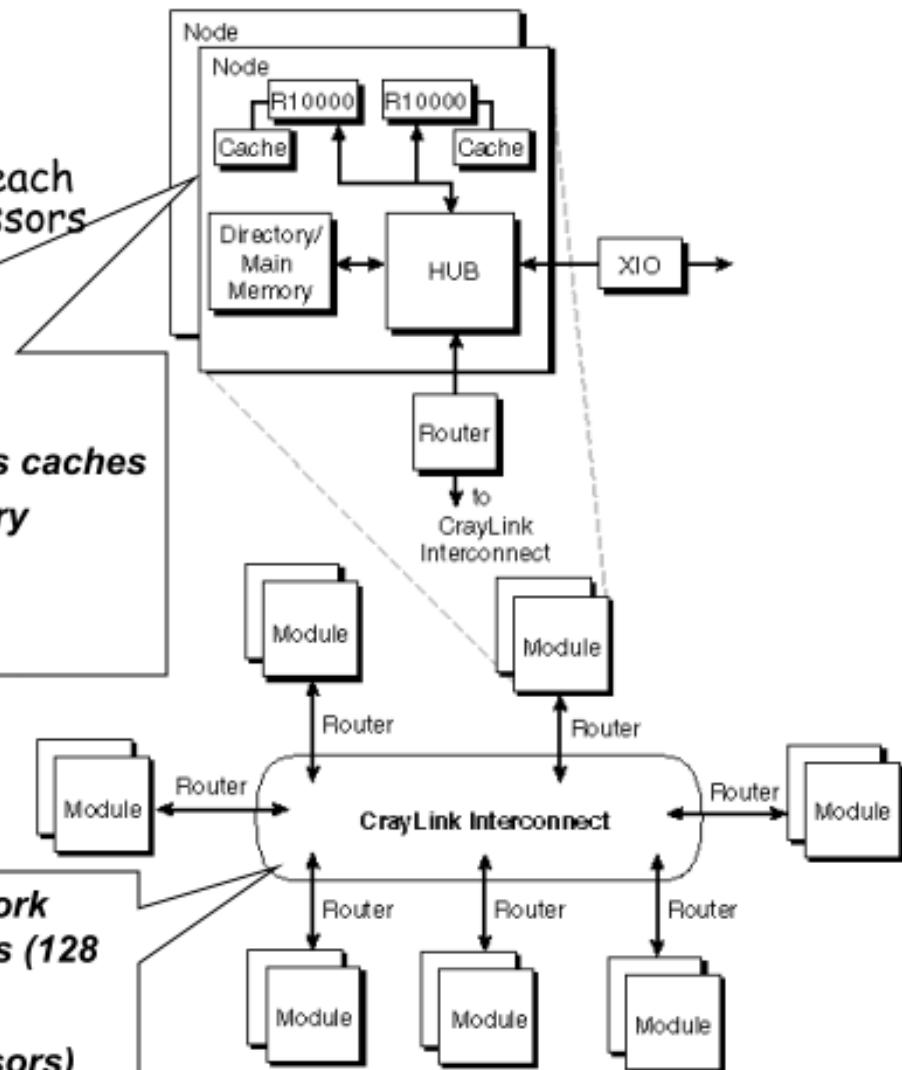


SGI Origin 2000 (1996)

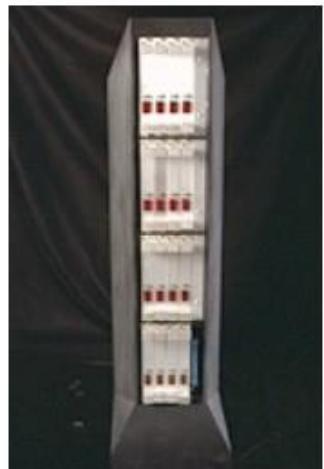
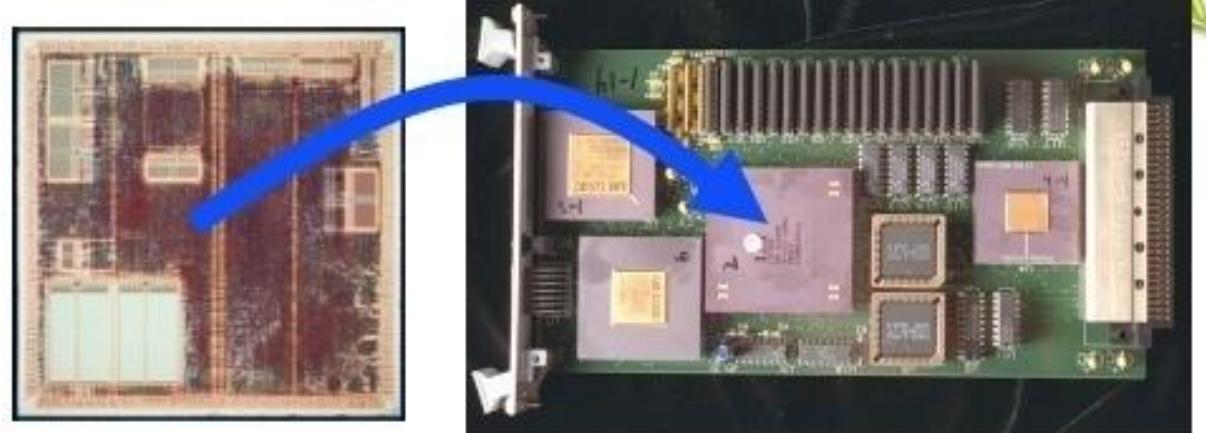
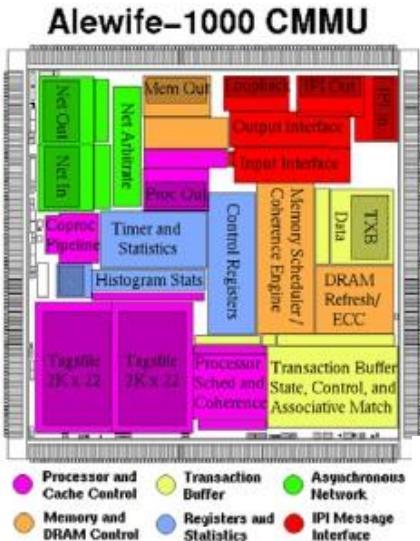
- Large-Scale Distributed Directory SMP
 - Scales from 2 to 512 nodes
 - Direct-mapped directory with each bit standing for multiple processors
 - Not highly scalable beyond this

Node contains:

- Two MIPS R10000 processors plus caches
- Memory module including directory
- Connection to global network
- Connection to I/O



MIT Alewife Multiprocessor: SM & MP

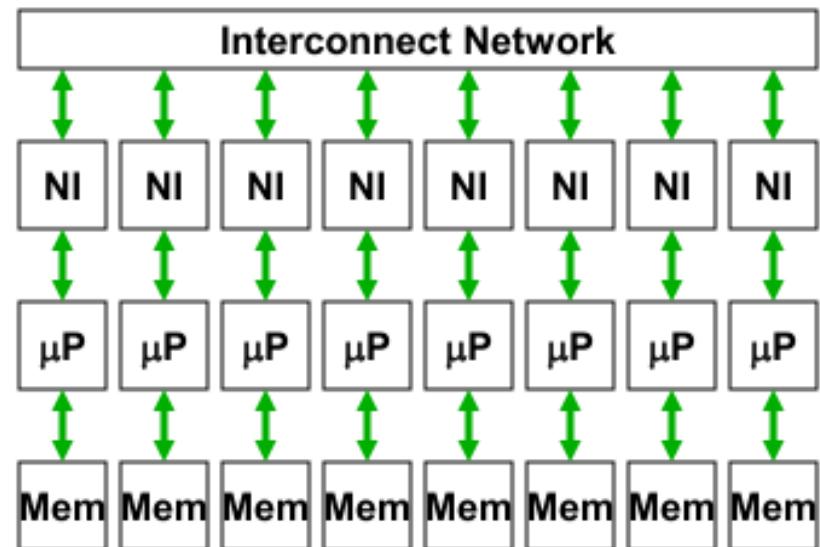


- Cache-coherence Shared Memory
 - Partially in Software!
 - Sequential Consistency
 - LimitLESS cache coherence for better scalability
- User-level Message-Passing
 - Fast, atomic launch of messages
 - Active messages
 - User-level interrupts
- Rapid Context-Switching
 - Course-grained multithreading
- Single Full/Empty bit per word for synchronization
 - Can build locks, barriers, other higher-level constructs

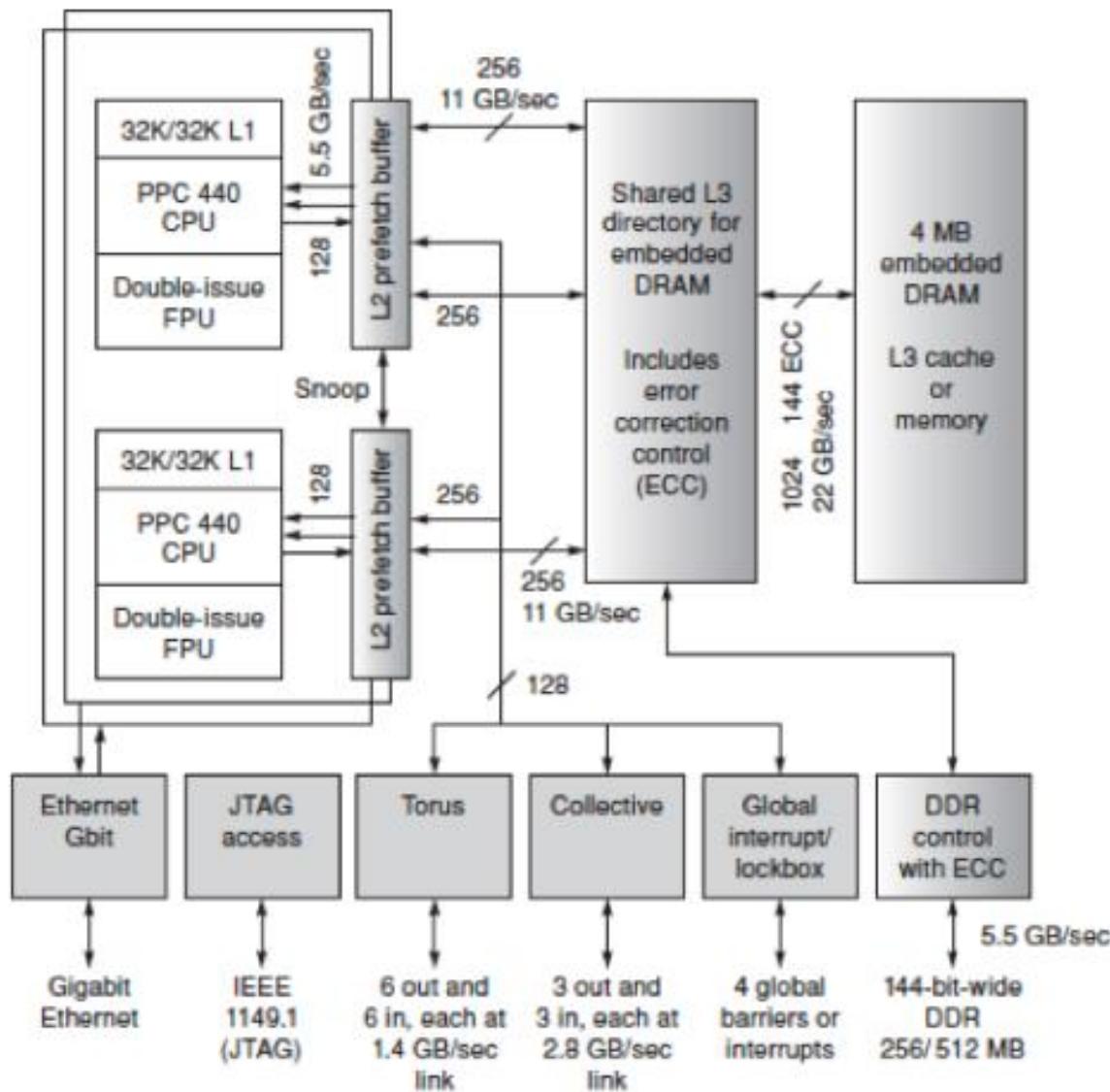
(Massively Parallel Processors)

- Initial Research Projects
 - Caltech Cosmic Cube (early 1980s) using custom Mosaic processors
 - J-Machine (early 1990s) MIT
- Commercial Microprocessors including MPP Support
 - Transputer (1985)
 - nCube-1(1986) /nCube-2 (1990)
- Standard Microprocessors + Network Interfaces
 - Intel Paragon/i860 (1991)
 - TMC CM-5/SPARC (1992)
 - Meiko CS-2/SPARC (1993)
 - IBM SP-1/POWER (1993)
- MPP Vector Supers
 - Fujitsu VPP500 (1994)

*Designs scale to 100s-10,000s
of nodes*



IBM Blue Gene/L Processor



BG/L 64K Processor System

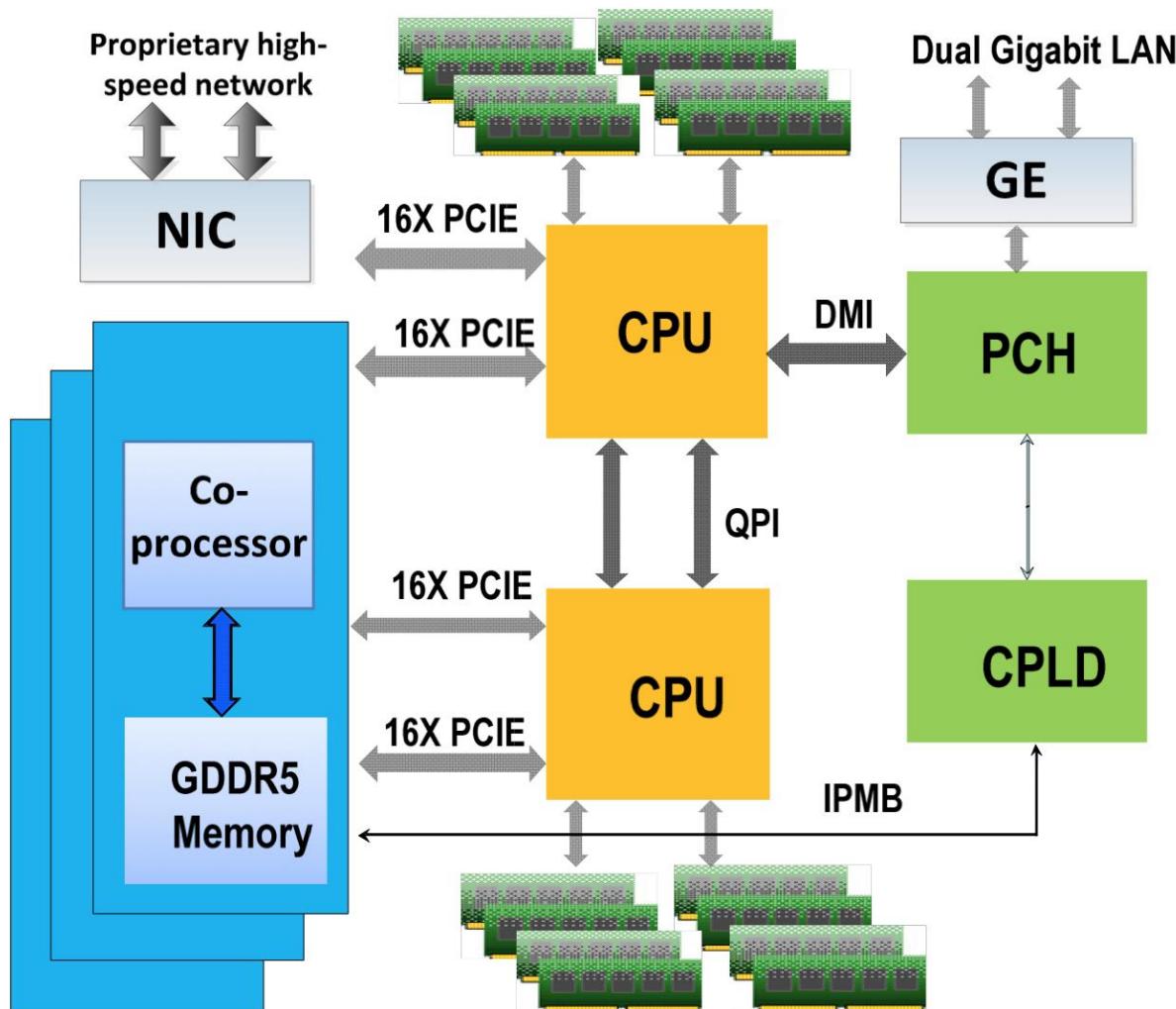
- Peak Performance 360TFLOPS
- Power Consumption 1.4 MW



50th TOP500 on November 2017

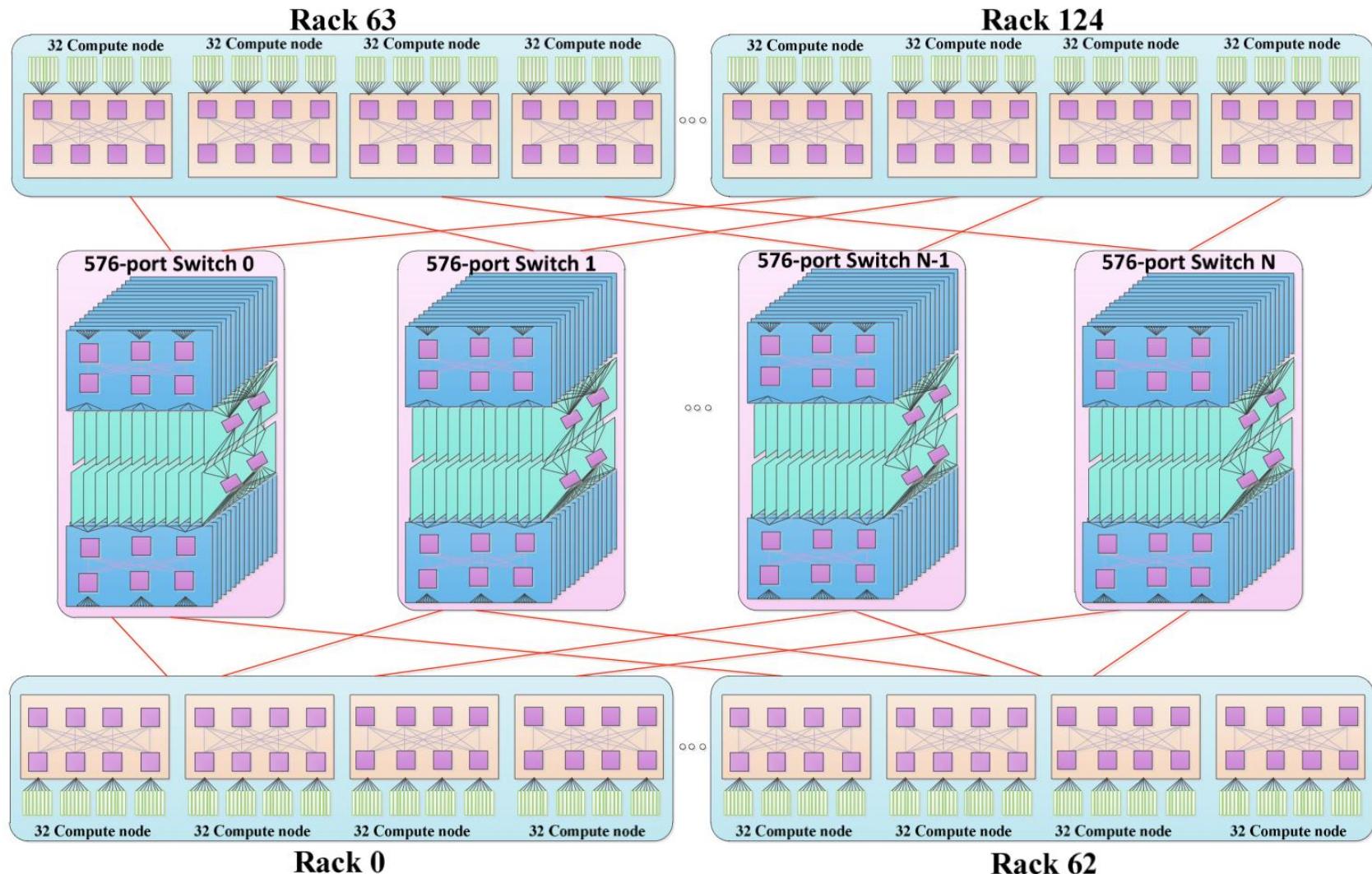
Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway , NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
2	Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P , NUDT National Super Computer Center in Guangzhou China	3,120,000	33,862.7	54,902.4	17,808
3	Piz Daint - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 , Cray Inc. Swiss National Supercomputing Centre (CSCS) Switzerland	361,760	19,590.0	25,326.3	2,272
4	Gyoukou - ZettaScaler-2.2 HPC system, Xeon D-1571 16C 1.3GHz, Infiniband EDR, PEZY-SC2 700Mhz , ExaScaler Japan Agency for Marine-Earth Science and Technology Japan	19,860,000	19,135.8	28,192.0	1,350
5	Titan - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x , Cray Inc. DOE/SC/Oak Ridge National Laboratory United States	560,640	17,590.0	27,112.5	8,209
6	Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom , IBM DOE/NNSA/LLNL United States	1,572,864	17,173.2	20,132.7	7,890

Tianhe-2



Tianhe-2	
Sponsors	863 Program
Location	Guangzhou, China
Architecture	Intel Xeon E5, Xeon Phi
Power	17.6 MW (24 MW with cooling)
Operating system	Kylin Linux ^[1]
Memory	1,375 TiB (1,000 TiB CPU and 375 TiB Coprocessor) ^[1]
Storage	12.4 PB
Speed	33.86 PFLOPS
Cost	2.4 billion Yuan (390 million USD) ^[2]
Purpose	Research and education

Tianhe-2



References

- The content expressed in this chapter is come from
 - berkeley university open course
(<http://parlab.eecs.berkeley.edu/2010bootcampagenda>)
 - Carnegie Mellon University's public course, Parallel Computer Architecture and Programming, (CS 418)
(<http://www.cs.cmu.edu/afs/cs/academic/class/15418-s11/public/lectures/>)
 - Livermore Computing Center's training materials, Introduction to Parallel Computing
(https://computing.llnl.gov/tutorials/parallel_comp/)