Architecture 2024

Chapter 6
Multicores, Multiprocessors, and Clusters

Hiroaki Kobayashi

Masayuki Sato

Ver. 01212025

Contents

Introduction (1/2)

- Goal of multiprocessor is to create powerful computers simply by connecting many existing smaller ones.
  - Replace large inefficient processors with many smaller, efficient processors
  - Scalability, availability, power efficiency

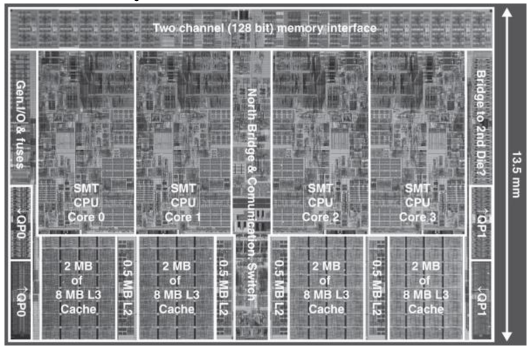
- Job-level (or process-level) parallelism
  - high throughput for independent jobs
- Parallel processing program
  - Single program runs on multiple processors
  - Short turnaround time

Introduction (2/2)

- A cluster is composed of microprocessors housed in many independent servers or PCs
  - Now very popular as search engine, Web servers, email servers and database server in data centers

Cluster System
@Cyberscience Center

- Multicore/Manycore microprocessors
  - Chips with multiple processors (cores)
  - Difficulty in seeking higher clock rates and improved CPI on single processor due to the power problem.
  - Number of cores is expected to double every two years



INTEL Nehalem-EP
Processor with 4-core

Programmers who care about performance must become parallel programmers!

Concurrency vs Parallel Hardware/Software Categorization

		Software	
		Sequential	Concurrent
Hardware	Serial	Matrix Multiply written in C, and compiled for and running on an Intel Pentium 4	Windows Vista Operating System running on an Intel Pentium 4
	Parallel	Matrix Multiply written in C, and compiled for and running on an Intel Xeon E5345 (Clovertown)	Windows Vista Operating System running on an Intel Xeon E5345 (Clovertown)

Challenge: making effective use of parallel hardware for sequential or concurrent software

Difficulty of Creating Parallel Processing Programs

- Parallel software is the problem
  - Need to get significant performance improvement
    - Otherwise, just use a faster uniprocessor, since it's easier!
    - Uniprocessor design technique such as superscalar and out-of-order execution take advantage of ILP, normally without the involvement of the programmer

- Difficulties in design of parallel programs
  - Partitioning (as equally as possible)
  - Coordination/scheduling for load balancing
  - Synchronization and communications overheads

The difficulties get worse as the number of processors increase…

Speedup Challenge

Amdahl's Law says

- Sequential part can limit speedup

- Example: 100 processors, 90 × speedup, how much sequential part should be suppressed?
  - $T_{new} = T_{parallelizable}/100 + T_{sequential}$
  - $\text{Speedup} = \dfrac{1}{(1-F_{parallelizable}) + F_{parallelizable}/100} = 90$
  - Solving: $F_{parallelizable} = 0.999$
- Need sequential part to be 0.1% of original time

Scaling Example (1/2)

- Workload: sum of 10 scalars, and 10×10 (100 scalars) matrix sum
  - Speed up from 10 to 100 processors?

- Single processor: Time = $(10 + 100) \times t_{add}$
- 10 processors
  - Time = $10 \times t_{add} + 100/10 \times t_{add} = 20 \times t_{add}$
  - Speedup = 110/20 = 5.5x faster (55% of potential)
- 100 processors
  - Time = $10 \times t_{add} + 100/100 \times t_{add} = 11 \times t_{add}$
  - Speedup = 110/11 = 10x faster (10% of potential)
- Assumes load can be balanced across processors

Scaling Example (2/2)

- What if matrix size is 100×100?

- Single processor: Time = $(10 + 10000) \times t_{add}$
- 10 processors
  - Time = $10 \times t_{add} + 10000/10 \times t_{add} = 1010 \times t_{add}$
  - Speedup = 10010/1010 = 9.9 (99% of potential)
- 100 processors
  - Time = $10 \times t_{add} + 10000/100 \times t_{add} = 110 \times t_{add}$
  - Speedup = 10010/110 = 91 (91% of potential)
- Assuming load balanced

Strong Scaling vs Weak Scaling

Getting good speed-up on a multiprocessor while keeping the problem size fixed is harder than getting good speed-up by increasing the size of the problem

- Strong scaling: problem size fixed
  - As in example, the same problem size in 10 to 100 processors
- Weak scaling: problem size proportional to number of processors
  - 10 processors, 10 × 10 matrix
    - Time = $10 \times t_{add} + 100/10 \times t_{add} = 20 \times t_{add}$
  - 100 processors, 32 × 32 (nearly 1000 elements) matrix
    - Time = $10 \times t_{add} + 1000/100 \times t_{add} = 20 \times t_{add}$
  - Constant performance in this example

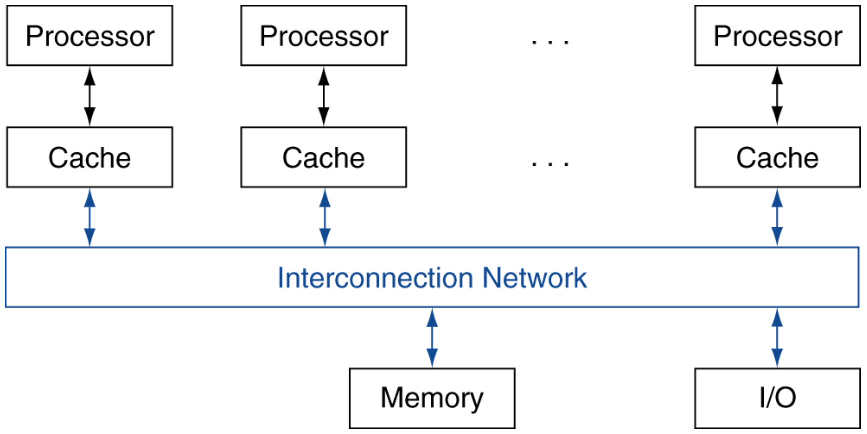
Speed-up Challenge: Balancing Load

- In the previous example, each processor has 1% of the work to achieve the speed-up of 91 on the larger problem with 100 processors.
- If one processor's load is higher than all the rest, show the impact on speed-up. Calculate at 2% and 5%
- If one processor has 2% of 10,000 and other 99 will share the remaining 9800
  - Execution time=Max(9800t/99, 200t/1)+10t=210t
  - Speed-up drops to 10,010t/210t=48
- If one processor has 5% of 10,000 and other 99 will share the remaining 9500
  - Execution time=Max(9500t/99, 500t/1)+10t=510t
  - Speed-up drops to 10,010t/510t=20

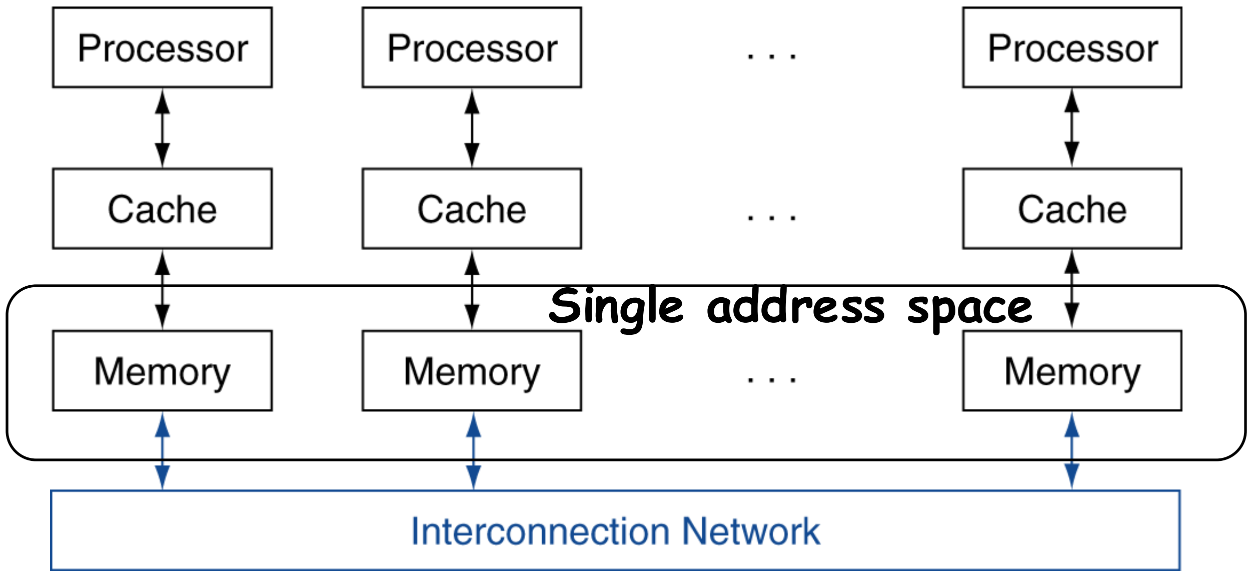
Shared Memory Multiprocessors

- SMP: shared memory multiprocessor
  - Hardware provides single physical address space for all processors
  - Synchronize shared variables using locks
  - Memory access time
    - UMA (uniform) vs. NUMA (nonuniform)

Uniform Memory Access Multiprocessor



Non Uniform Memory Access Multiprocessor



- Lower latency to nearby memory
- Higher scalability
- More efforts for efficient parallel programming

Synchronization and Lock on SMP

- Synchronization: The process of coordinating the behavior of two or more processes, which may be running on different processors.
  - As processors operating in parallel will normally share data, they also need to coordinate when operating on shared data.
- Lock: a synchronization device that allows access to data to only one process at a time.
  - Other processors interested in shared data must wait until the original processor unlocks the variable.

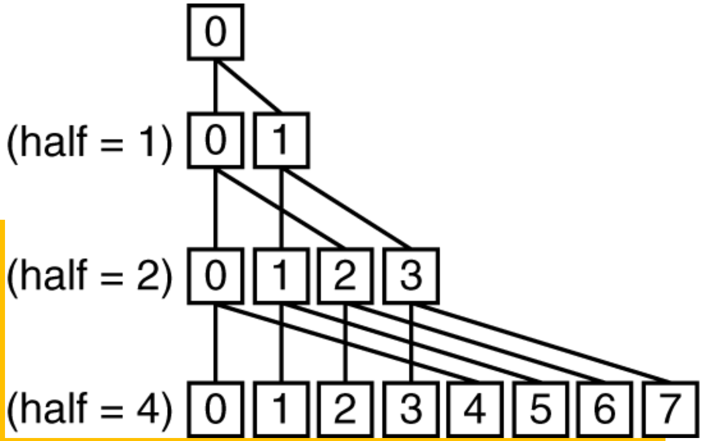
Example: Sum Reduction (1/2)

- Sum 100,000 numbers on 100-processor UMA
  - Each processor has ID: 0 ≤ Pn ≤ 99
  - Partition 1000 numbers per processor
  - Initial summation on each processor

```
sum[Pn] = 0;
for (i = 1000*Pn; i < 1000*(Pn+1); i = i + 1)
    sum[Pn] = sum[Pn] + A[i];
```

- Now need to add these partial sums
  - Reduction: divide and conquer
  - Half the processors add pairs, then quarter, …
  - Need to synchronize between reduction steps

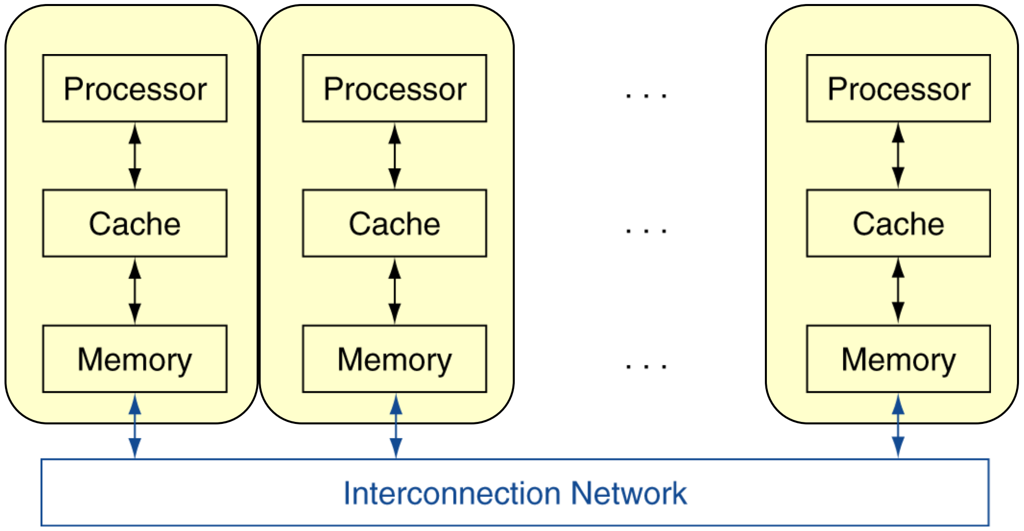
Example: Sum Reduction (2/2)



```
half = 100;
repeat
  synch();
  if (half%2 != 0 && Pn == 0)
    sum[0] = sum[0] + sum[half-1];
    /* Conditional sum needed when half is odd;
       Processor0 gets missing element */
  half = half/2; /* dividing line on who sums */
  if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1);
```

Message-Passing Multiprocessors

- Each processor has private physical address space
- Hardware sends/receives messages between processors
- Multicomputers, clusters, Network of Workstations (NOW)
- Suited for job-level parallelism and applications with little communication
  - Web search, mail servers, file servers…



Loosely-Coupled Clusters

- Network of independent computers
  - Each has private memory and OS
  - Connected using I/O system
    - E.g., Ethernet/switch, Internet
- Suitable for applications with independent tasks
  - Web servers, databases, simulations, …
- High availability, scalable, affordable
- Problems
  - Administration cost (prefer virtual machines)
  - Low interconnect bandwidth
    - c.f. processor/memory bandwidth on an SMP

Sum Reduction on Message-Passing Multiprocessor (1/2)

- Sum 100,000 on 100 processors
- First distribute 1000 numbers to each
  - The do partial sums

```
sum = 0;
for (i = 0; i<1000; i = i + 1)
  sum = sum + AN[i];
```

- Reduction
  - Half the processors send, other half receive and add
  - The quarter send, quarter receive and add, …

Sum Reduction on Message-Passing Multiprocessor (2/2)

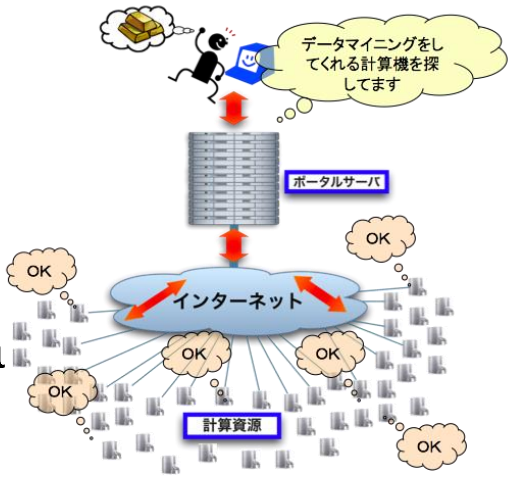
- Given send() and receive() operations

```
limit = 100; half = 100; /* 100 processors */
repeat
  half = (half+1)/2; /* send vs. receive
                        dividing line */
  if (Pn >= half && Pn < limit)
    send(Pn - half, sum);
  if (Pn < (limit/2))
    sum = sum + receive();
  limit = half; /* upper limit of senders */
until (half == 1); /* exit with final sum */
```

  - Send/receive also provide synchronization
  - Assumes send/receive take similar time to addition

Grid Computing

- Separate computers interconnected by long-haul networks and manage them as a single-system image
  - Analogy of power grid
  - Work units farmed out, results sent back
- Can make use of idle time on PCs
  - E.g., SETI@home
  - Aslo named volunteer computing
  - 1+ million computers, 200+ countries, 5.3 Peta-flop/s for BOINC infra. in 2011
  - http://www.planetary.or.jp/setiathome/home_japanese.html



SETI@home
The Search for Extraterrestrial Intelligence

BOINC

Parallelism and Memory Hierarchies: Cache Coherence on a Shared-Memory Parallel Computer

- Suppose two CPU cores share a physical address space
  - Write-through caches

Time step	Event	CPU A’s cache	CPU B’s cache	Memory
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A writes 1 to X	1	0	1

inconsistency

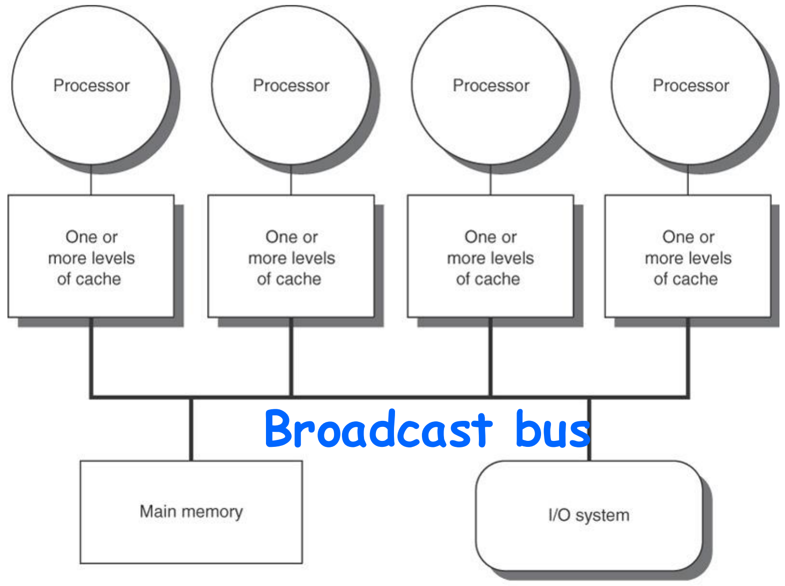
What is Coherence?

- Informally: Reads return most recently written value
- Formally: *A memory system is coherent if*
  - P writes X; P reads X (no intervening writes)
    $\Rightarrow$ read returns written value
  - P_1 writes X; P_2 reads X (sufficiently later)
    $\Rightarrow$ read returns written value
    - c.f. CPU B reading X after step 3 in example
  - P_1 writes X, P_2 writes X
    $\Rightarrow$ all processors see writes in the same order (*serialized*)
    - End up with the same final value for X

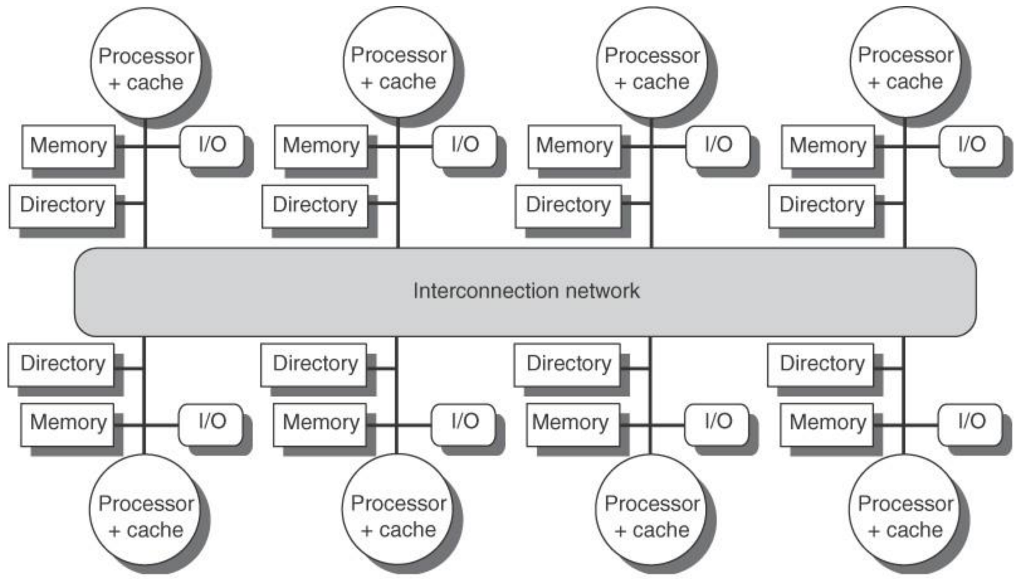
Cache Coherence Protocol

- Operations performed by caches in multiprocessors to ensure coherence
  - Migration of data to local caches
    - Reduces bandwidth for shared memory
  - Replication of read-shared data
    - Reduces contention for access
- Snooping protocols
  - Each cache monitors bus reads/writes
  - Update states of cache entries when modified
- Directory-based protocols
  - Caches and memory record sharing status of blocks in a directory
  - Handling states of cache entries in a distributed manner

Snoop-based and Directory-based Systems


Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Broadcast bus
Main memory
I/O system

Snoop-based system



Directory-based system

Invalidating Snooping Protocols

- Cache gets exclusive access to a block when it is to be written
  - Broadcasts an invalidate message on the bus
  - Subsequent read in another cache misses
    - Owning cache supplies updated value
  - If two processors attempt to write the same data simultaneously, one of them wins the race, causing the other processor's copy to be invalidated.
    - Enforce write serialization.

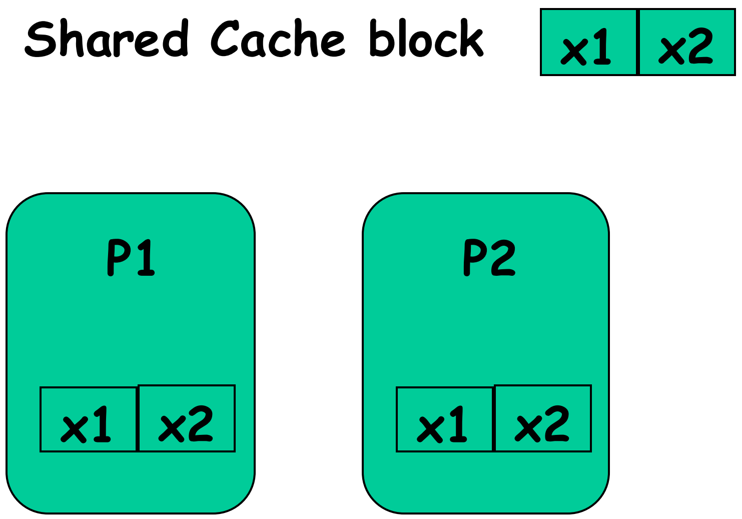
CPU activity	Bus activity	CPU A's cache	CPU B's cache	Memory
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 to X	Invalidate for X	1	✗	0
CPU B read X	Cache miss for X	1	1	1

Coherence Misses

- Coherence Misses: a new class of cache misses due to sharing data on a multiprocessor
- True Sharing
  - A write to the shared data on the cache by one processor causes an invalidate to the entire block that includes the shared data cached on other processors, and a miss occurs when another processor accesses a modified word in that block.
- False Sharing
  - A write to the shared data on the cache by one processor causes an invalidate to the entire block that includes the shared data cached on other processors, and a miss occurs even when another processor accesses a different word but in the same block.
- True and False sharing misses occur because a cache sharing state is given to each cache block.
  - If block size is one word, only true sharing misses occur.

Example of True/False Sharing

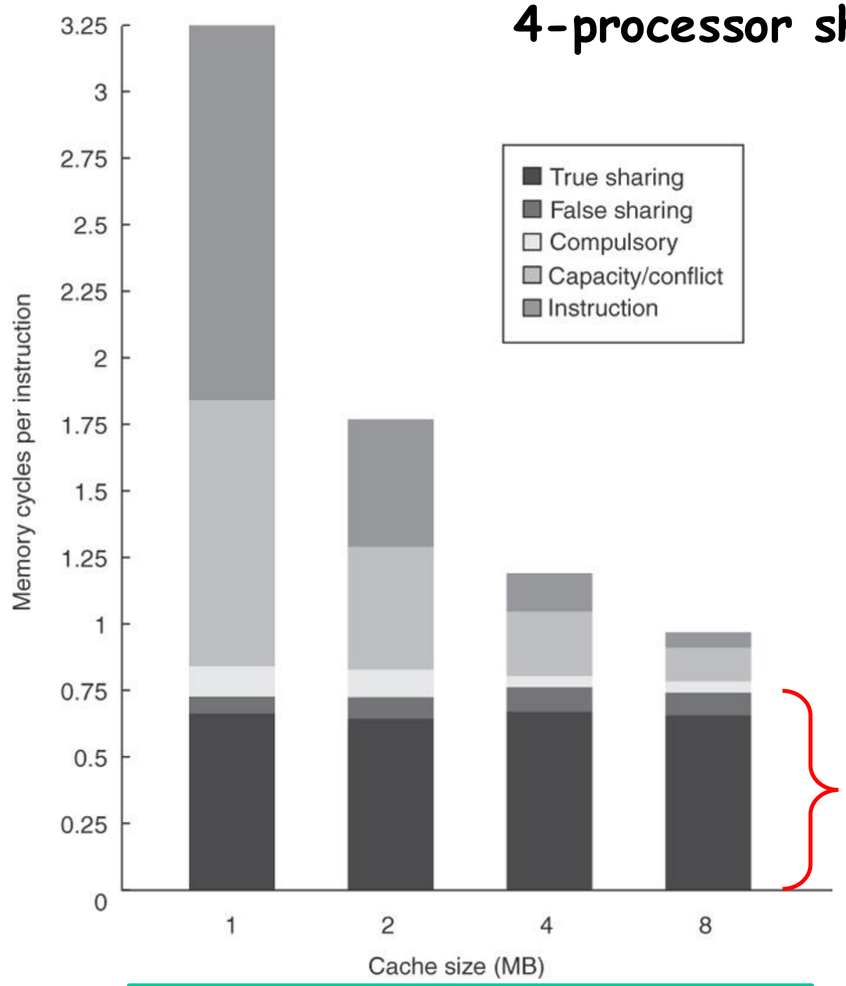
Time	P1	P2
1	Write x1	
2		Read x2
3	Write x1	
4		Write x2
5	Read x2	

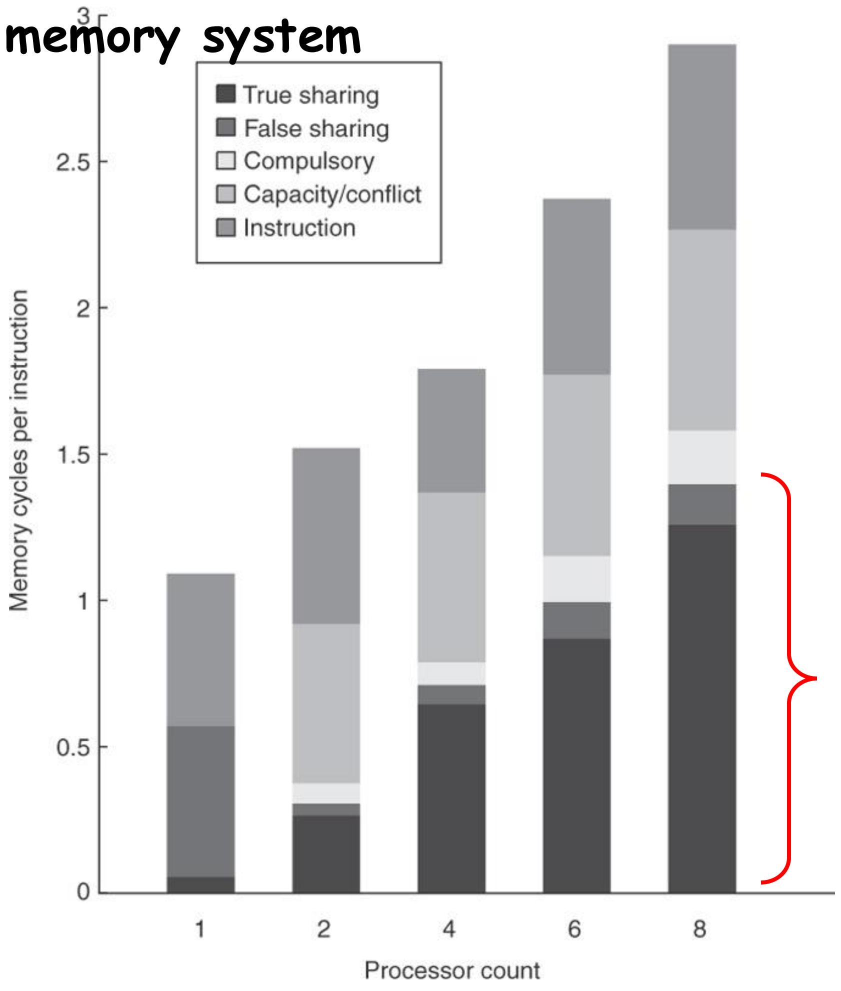


True or false sharing miss??

Contributing Causing of Memory Cycles on SMP

4-processor shared memory system





True/False sharing does not depend on the cache size

True/False sharing increases as increasing parallelism

Memory Consistency (1/2)

- P1

  A = 0;

  …

  A = 1;

  If (B == 0) …

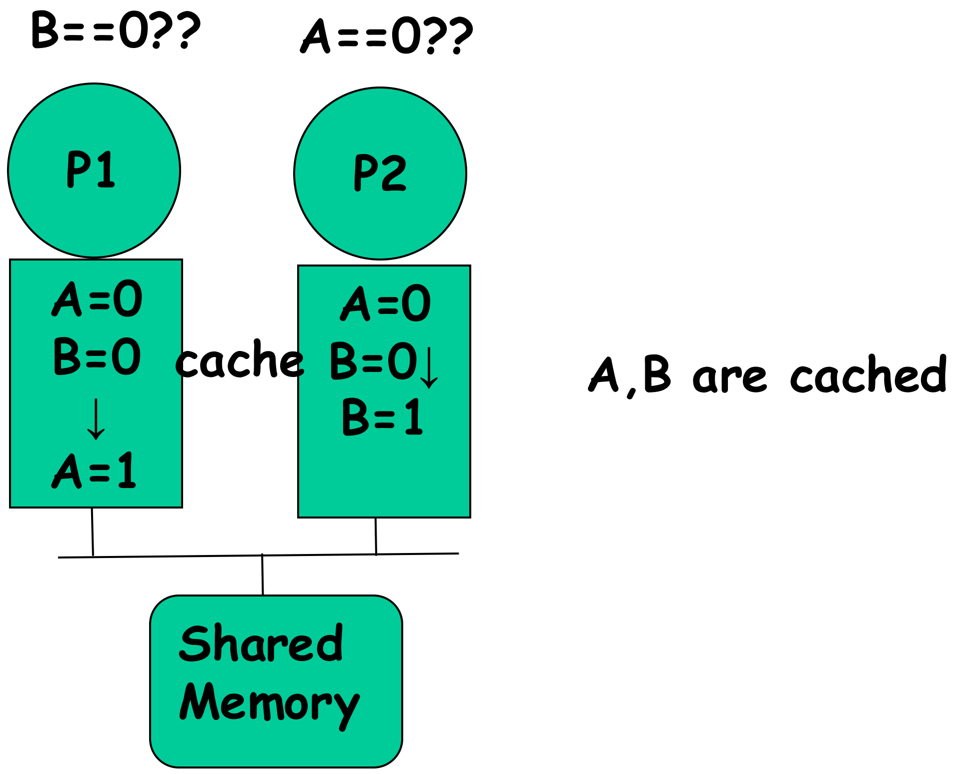
- P2

  B = 0

  …

  B = 1;

  If (A == 0) …



When an updated value by one processor becomes visible to the other??

Memory Consistency

- *Consistency* determines *when* a written value will be returned by a read.
  - *Coherence* defines *what values* can be returned by a read.
  - When are writes seen by other processors
    - "Seen" means a read returns the written value
    - Can't be instantaneously
- Sequential Consistency Model (SCM)
  - The result of any execution is the same as if the memory accesses executed by each processor were kept in order and
  - the accesses among different processors were arbitrarily interleaved.
  - ✓ SCM provides simple programming paradigm, but reduces potential performance
- Relaxed Consistency Model (RCM)
  - Allow reads and writes to complete out of order, but to use synchronization operations to enforce ordering
    - a synchronized program behaves as if the processor were sequentially consistent.
  - ✓ By RCM, the processor can possibly obtain significant performance advantages.

Classifications Based on Instruction and Data Streams

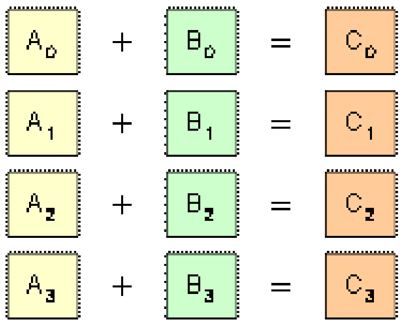
- Flynn's Classification (1966)

		Data Streams	
		Single	Multiple
Instruction Streams	Single	**SISD**: Intel Pentium 4	**SIMD**: SSE instructions of x86
	Multiple	**MISD**: No examples today	**MIMD**: Intel Xeon e5345

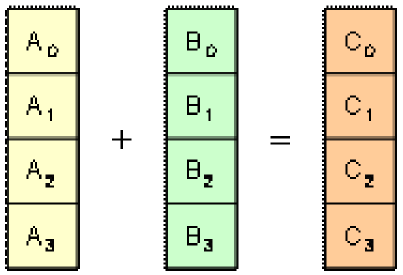
- SPMD: Single Program Multiple Data
  - A parallel program on a MIMD computer
  - Conditional code for different processors

SIMD (1/2)

- Operate element-wise on vectors of data
  - E.g., MMX and SSE instructions in x86
    - Multiple data elements in 128-bit wide registers for
      - four 32-bit single precision floating point number
      - two 64-bit double-precision floating point number
      - Two 64-bit integer
      - Four 32-bit integer
      - Eight 16-bit short integer
      - 16 8-bit bytes or characters

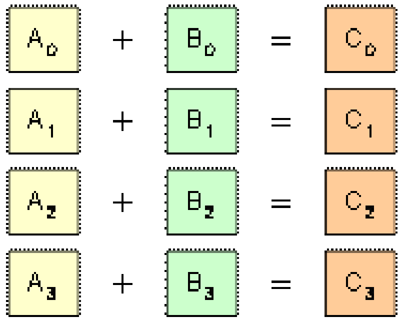


(a) Scalar Operation

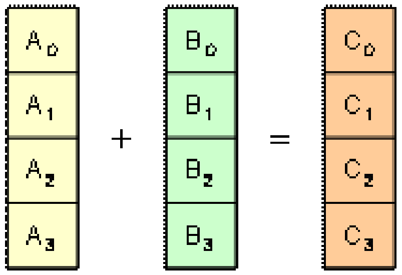


(b) SIMD Operation

SIMD (2/2)



(a) Scalar Operation



(b) SIMD Operation

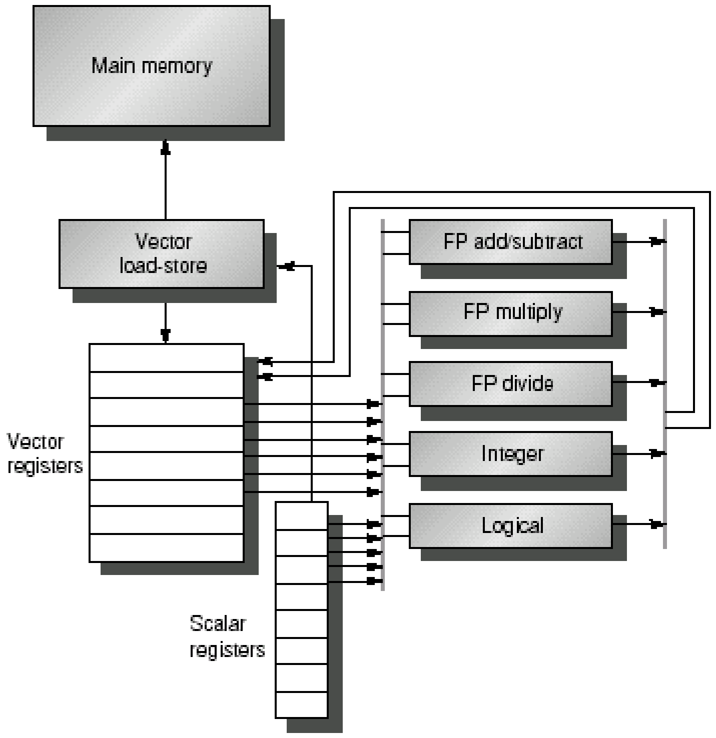
- All processors execute the same instruction at the same time
  - Each with different data address, etc.
- Simplifies synchronization
- Reduced instruction control hardware
- Works best for highly data-parallel applications
- Weakest in *case* or *switch* statement
  - Need a mask operation to select appropriate results

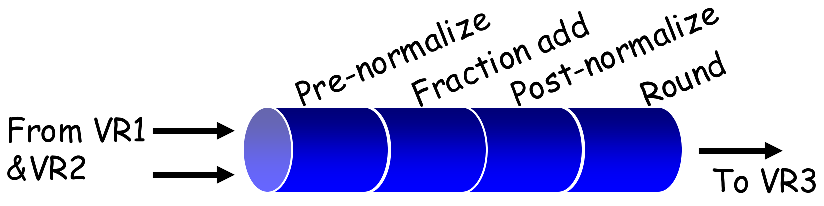
Vector Processors

- Highly pipelined function units
- Stream data from/to vector registers to units
  - Data collected from memory into registers
  - Results stored from registers to memory
- Example: Vector extension to MIPS
  - 32 × 64-element registers (64-bit elements)
  - Vector instructions
    - `lv`, `sv`: load/store vector
    - `addv.d`: add vectors of double
    - `addvs.d`: add scalar to each element of vector of double
- Significantly reduces instruction-fetch bandwidth

Characteristics of Vector Architecture

- Vector registers
  - Each vector register is a fixed-length bank holding a single vector and must have at least two read ports and one write port.
- Vector functional units
  - Each unit is fully pipelined and can start a new operation on every clock cycle. A control unit is needed to detect hazards, both from conflicts for the functional unit and from conflicts for register accesses.
- Vector load-store unit
  - This is a vector memory unit that loads or stores a vector to or from memory. Vector loads and stores are fully pipelined, so that words can be moved between the vector registers and memory with a bandwidth of 1 word per clock cycle, after an initial latency.





Pipelined FP ADD Unit

Example: DAXPY (Y = a x X + Y)

- Conventional MIPS code

```
      l.d   $f0,a($sp)      ;load scalar a
      addiu r4,$s0,#512     ;upper bound of what to load
loop: l.d   $f2,0($s0)      ;load x(i)
      mul.d $f2,$f2,$f0     ;a × x(i)
      l.d   $f4,0($s1)      ;load y(i)
      add.d $f4,$f4,$f2     ;a × x(i) + y(i)
      s.d   $f4,0($s1)      ;store into y(i)
      addiu $s0,$s0,#8      ;increment index to x
      addiu $s1,$s1,#8      ;increment index to y
      subu  $t0,r4,$s0      ;compute bound
      bne   $t0,$zero,loop ;check if done
```

- Vector MIPS code

```
      l.d     $f0,a($sp)    ;load scalar a
      lv      $v1,0($s0)    ;load vector x
      mulvs.d $v2,$v1,$f0   ;vector-scalar multiply
      lv      $v3,0($s1)    ;load vector y
      addv.d  $v4,$v2,$v3   ;add y to product
      sv      $v4,0($s1)    ;store the result
```

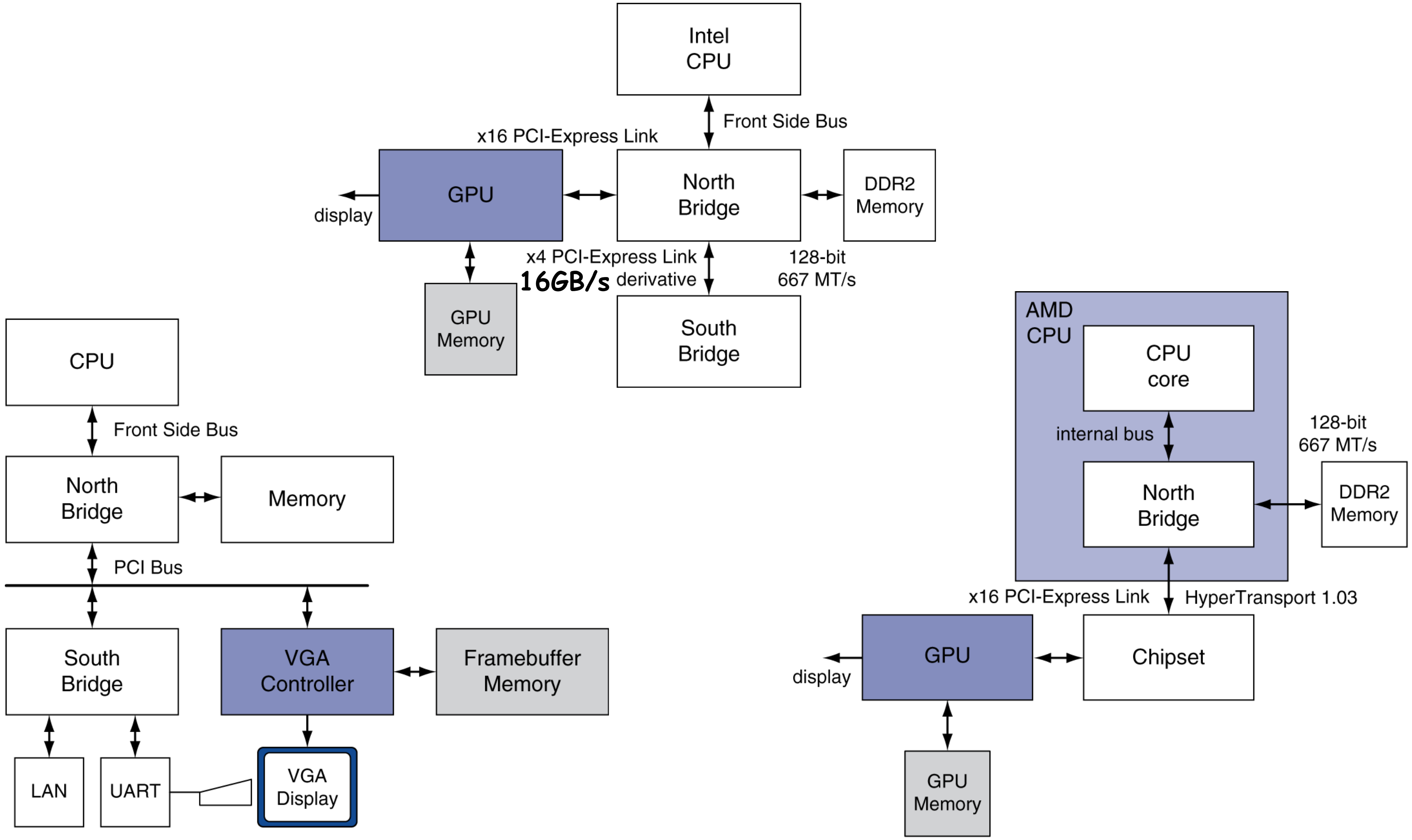
Vector vs. Scalar

- Vector architectures and compilers
  - Simplify data-parallel programming
  - Explicit statement of absence of loop-carried dependences
    - Reduced checking in hardware
  - Regular access patterns benefit from interleaved and burst memory
  - Avoid control hazards by avoiding loops
- More general than ad-hoc media extensions (such as MMX, SSE)
  - Better match with compiler technology

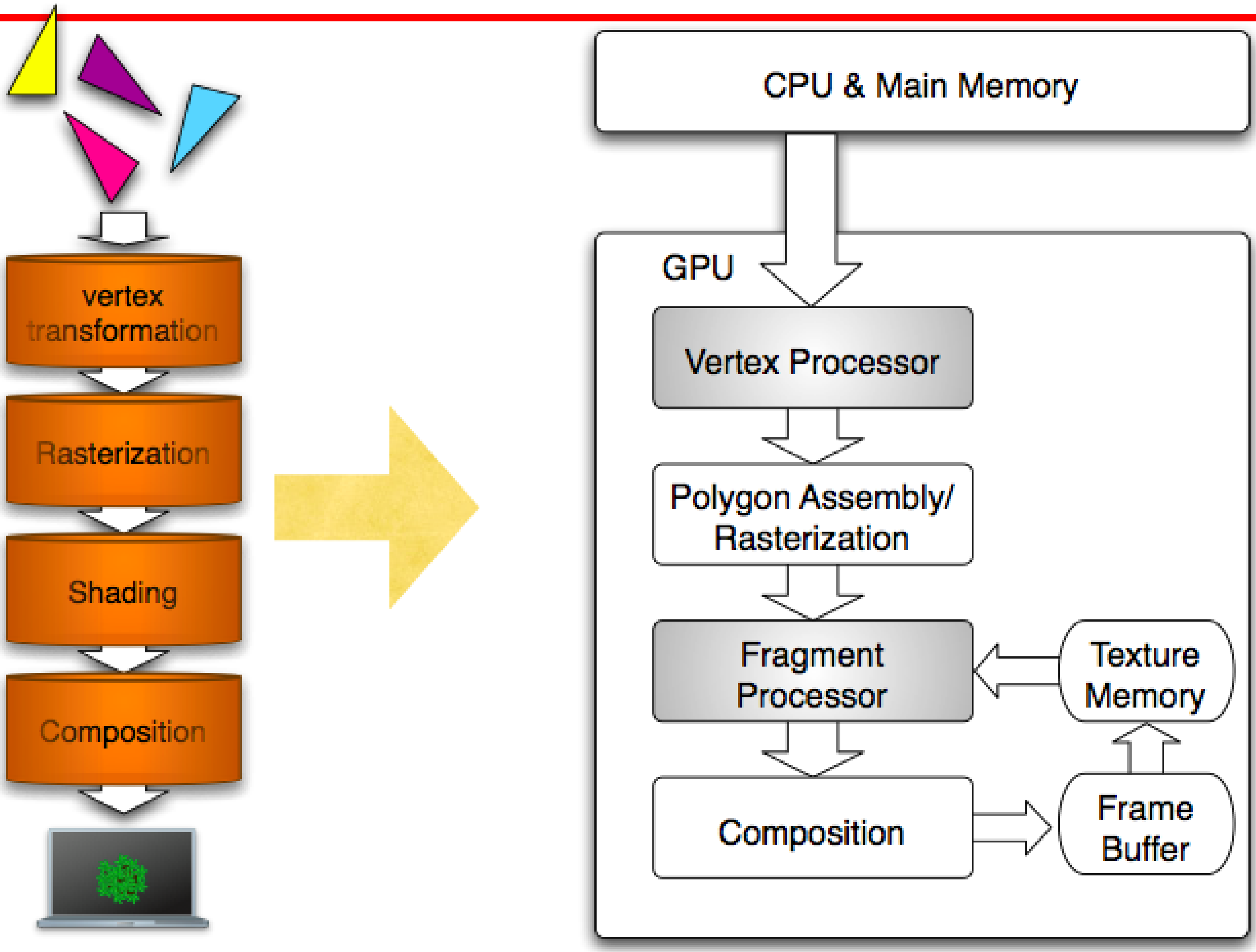
History of GPU

- Early video cards (VGA controller)
  - Frame buffer memory with address generation for video output
- 3D graphics processing
  - Originally high-end computers (e.g., SGI)
  - Moore's Law $\Rightarrow$ lower cost, higher density
  - 3D graphics cards for PCs and game consoles
- Graphics Processing Units
  - Processors oriented to 3D graphics tasks
  - Vertex/pixel processing, shading, texture mapping, rasterization

Graphics in the System



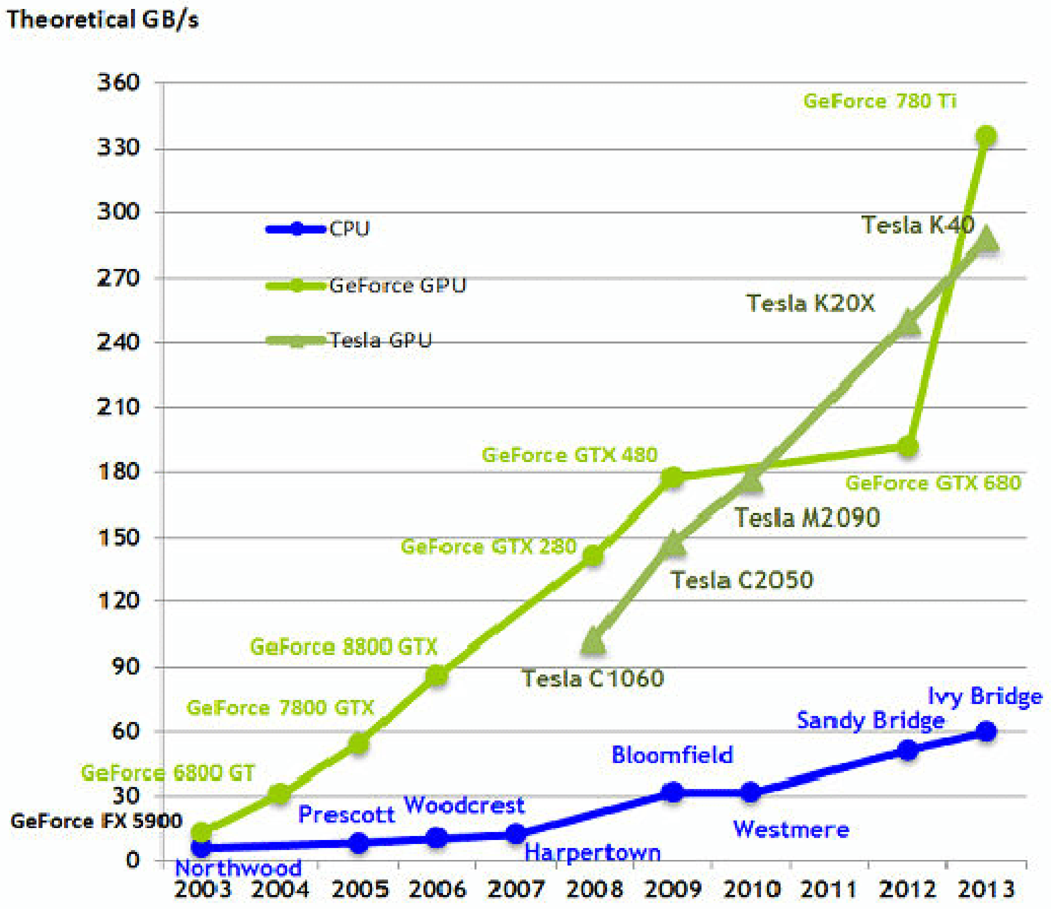
Graphics Rendering Pipeline



GPU Architecture

- Processing is highly data-parallel
  - GPUs are highly multithreaded
  - Use thread switching to hide memory latency
    - Less reliance on multi-level caches
  - Graphics memory is wide and high-bandwidth
- Trend toward general purpose GPUs
  - Heterogeneous CPU/GPU systems
  - CPU for sequential code, GPU for parallel code
- Programming languages/APIs
  - DirectX, OpenGL
  - C for Graphics (Cg), High Level Shader Language (HLSL)
  - Compute Unified Device Architecture (CUDA)

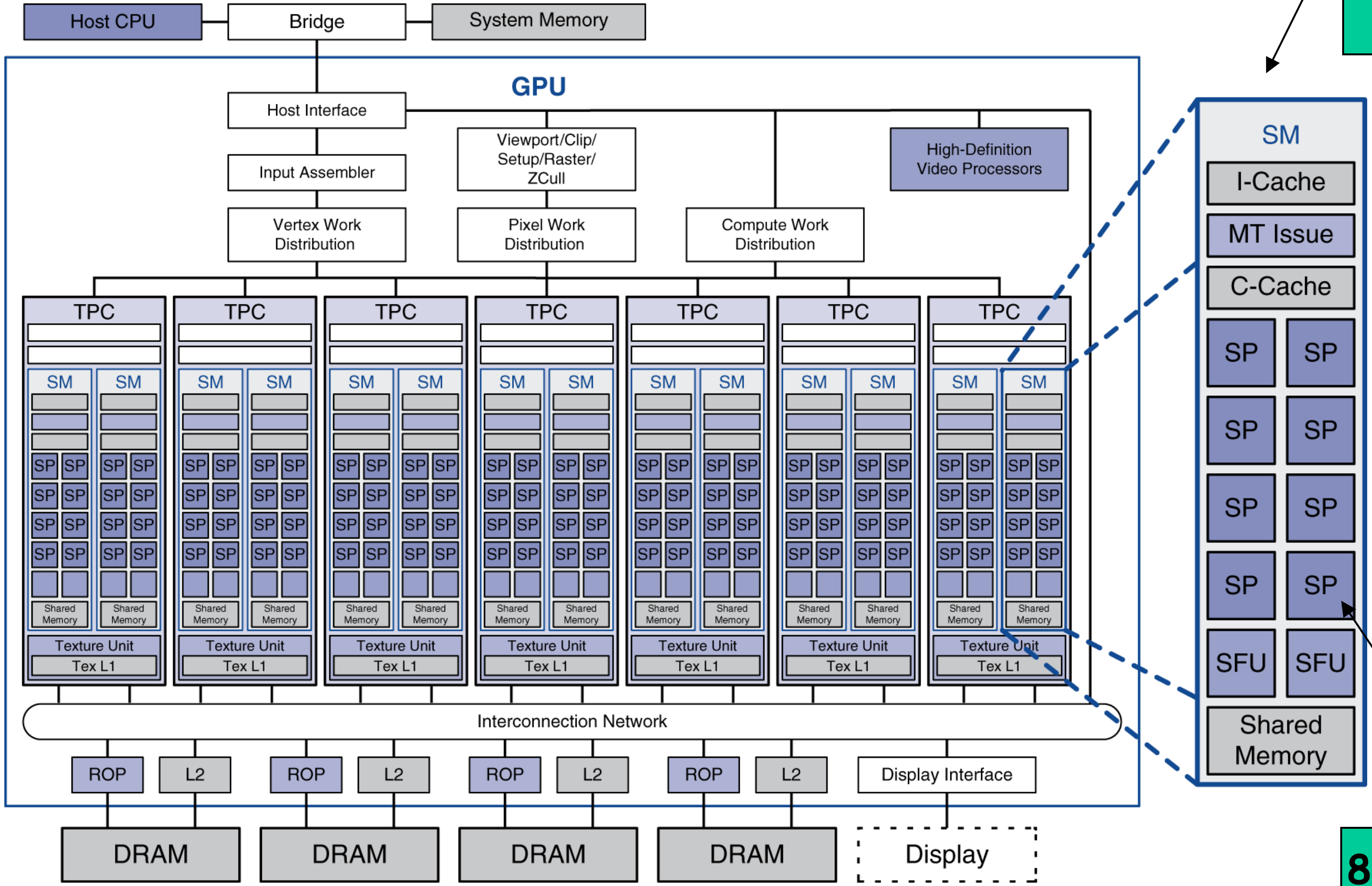
Trend in Performance between CPU and GPU



Thambawita, Vajira & Ragel, Roshan & Elkaduwe, Dhammika. (2014). To Use or Not to Use: Graphics Processing Units for Pattern Matching Algorithms.

Example: NVIDIA Tesla

$345.6\text{Gflop/s} = 16\ \text{MPs} \times 8\text{SPs} \times 2\text{Flop} \times 1.35\text{x}10^9$

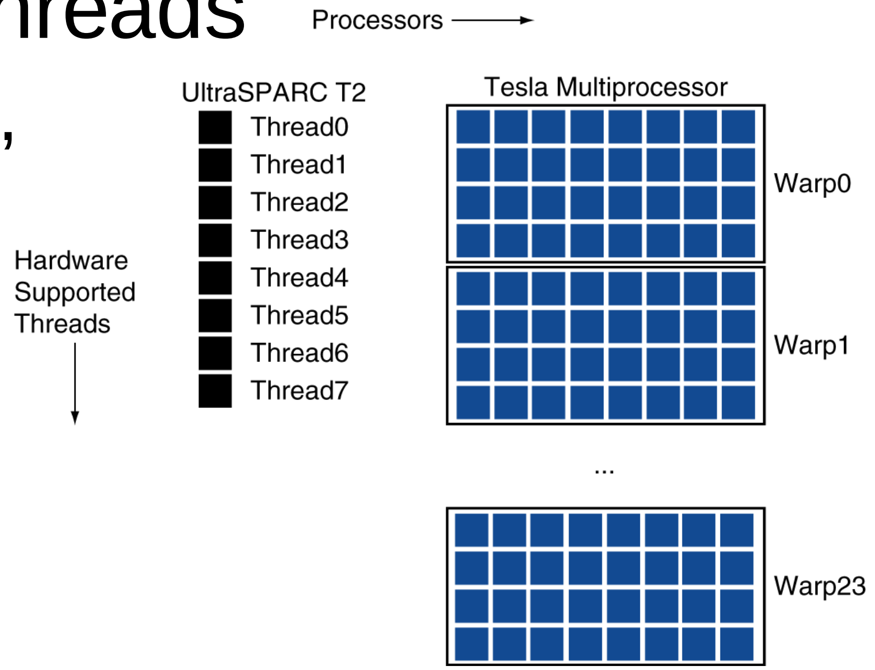


Streaming multiprocessor

8 × Streaming processors

Example: NVIDIA Tesla

- Streaming Processors
  - Single-precision FP and integer units
  - Each SP is fine-grained multithreaded
- Warp: group of 32 threads
  - Executed in parallel, SIMD style
    - 8 SPs × 4 clock cycles
  - Hardware contexts for 24 warps
    - Registers, PCs, …



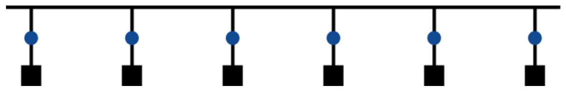
Classifying GPUs

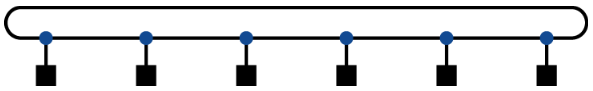
- Don’t fit nicely into SIMD/MIMD model
  - Conditional execution in a thread allows an illusion of MIMD
    - But with performance degredation
    - Need to write general purpose code with care

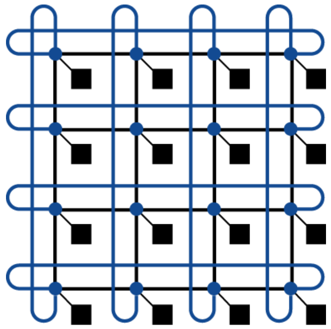
	Static: Discovered at Compile Time	Dynamic: Discovered at Runtime
Instruction-Level Parallelism	VLIW	Superscalar
Data-Level Parallelism	SIMD or Vector	**Tesla Multiprocessor**

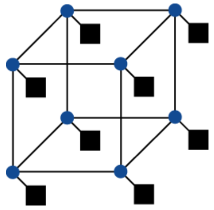
Interconnection Networks

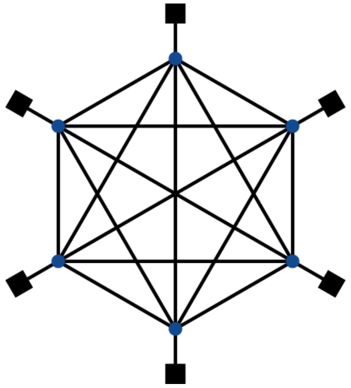
- Network topologies
  - Arrangements of processors, switches, and links


Bus

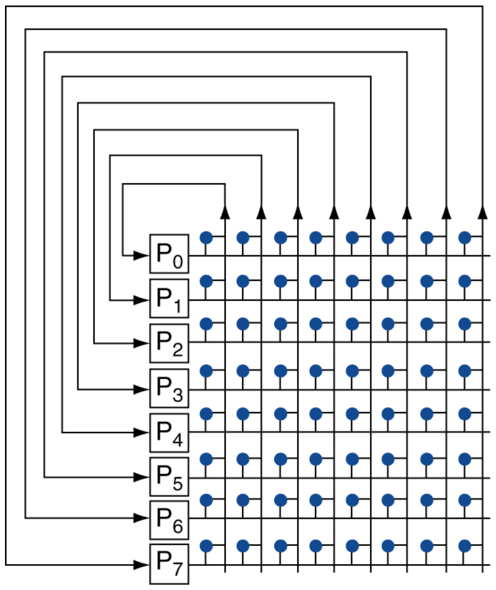

Ring


2D Mesh

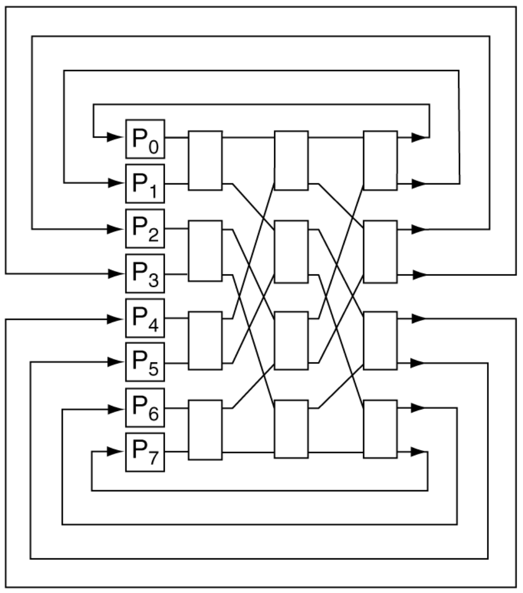

N-cube (N = 3)


Fully connected

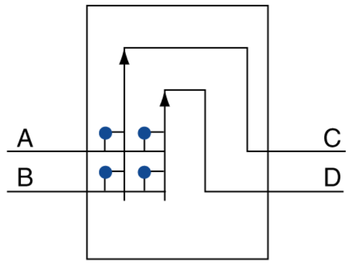
Multi-Stage Networks



a. Crossbar



b. Omega network



c. Omega network switch box

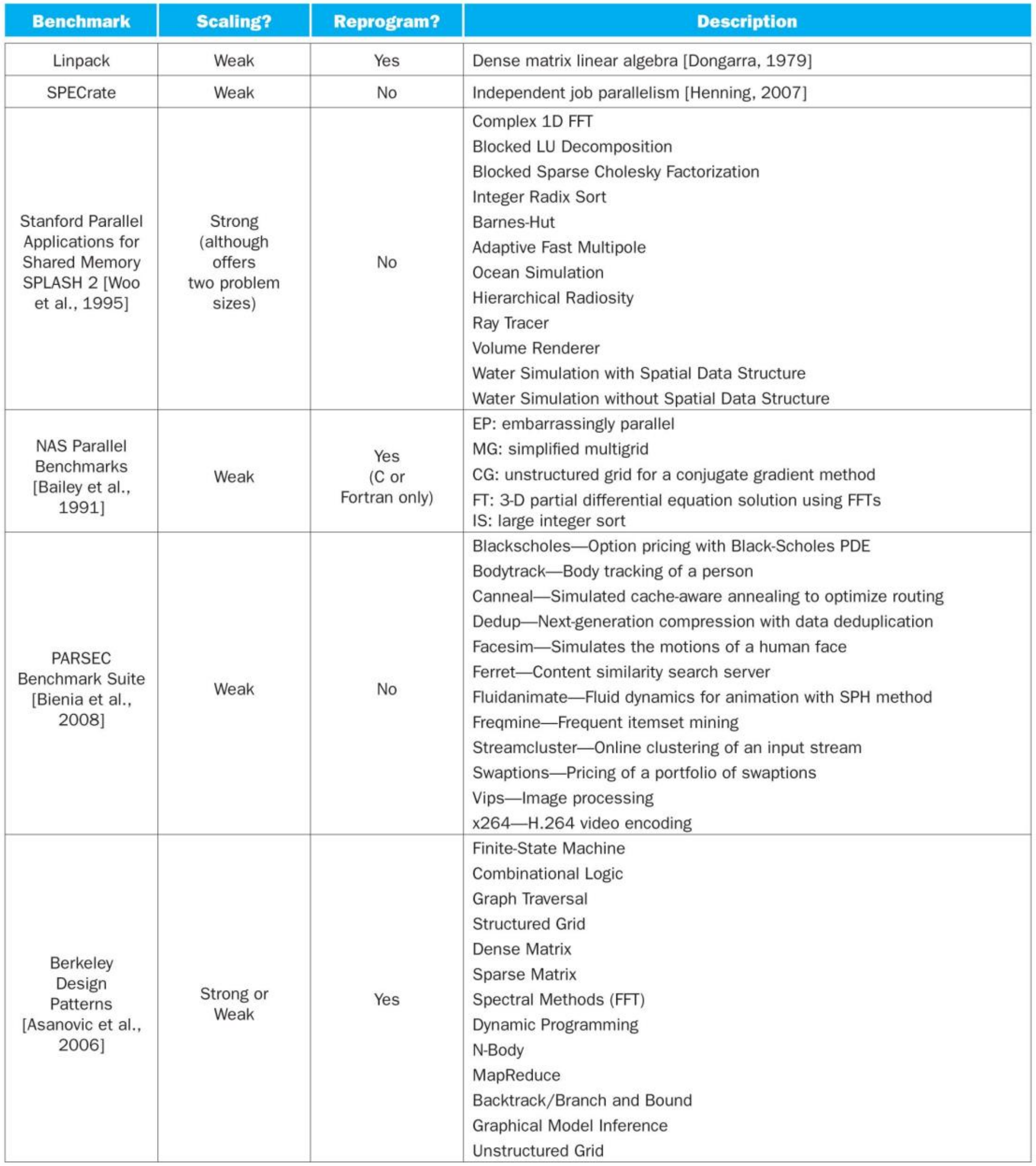
Network Characteristics

- Performance
  - Latency per message (unloaded network)
  - Throughput
    - Link bandwidth
      - Peak data transfer rate per link
    - Total network bandwidth
      - Collective data transfer rate of all links in the network
    - Bisection bandwidth
      - The bandwidth between two equal parts of a multiprocessor.
      - This measure is for a worst case split of the multiprocessor
  - Congestion delays (depending on traffic)
- Cost
- Power
- Routability in silicon

Parallel Benchmarks (1/2)

- Linpack: matrix linear algebra
  - DAXPY routine (double precision ax plus y, y[i] = a*x[i]+y[i])
  - Weak scaling
  - www.top500.org
- SPECrate: parallel run of SPEC CPU programs
  - Job-level parallelism (no communication between them)
  - Throughput metric
  - Weak scaling
- SPLASH: Stanford Parallel Applications for Shared Memory
  - Mix of kernels and applications, strong scaling
- NAS (NASA Advanced Supercomputing) suite
  - computational fluid dynamics kernels
  - Weak scaling
- PARSEC (Princeton Application Repository for Shared Memory Computers) suite
  - Multithreaded applications using Pthreads and OpenMP
  - Weak scaling

Parallel Benchmarks (2/2) – Contents at a glance



Benchmark	Scaling?	Reprogram?	Description
Linpack	Weak	Yes	Dense matrix linear algebra [Dongarra, 1979]
SPECrate	Weak	No	Independent job parallelism [Henning, 2007]
Stanford Parallel Applications for Shared Memory SPLASH 2 [Woo et al., 1995]	Strong (although offers two problem sizes)	No	Complex 1D FFT Blocked LU Decomposition Blocked Sparse Cholesky Factorization Integer Radix Sort Barnes-Hut Adaptive Fast Multipole Ocean Simulation Hierarchical Radiosity Ray Tracer Volume Renderer Water Simulation with Spatial Data Structure Water Simulation without Spatial Data Structure
NAS Parallel Benchmarks [Bailey et al., 1991]	Weak	Yes (C or Fortran only)	EP: embarrassingly parallel MG: simplified multigrid CG: unstructured grid for a conjugate gradient method FT: 3-D partial differential equation solution using FFTs IS: large integer sort
PARSEC Benchmark Suite [Bienia et al., 2008]	Weak	No	Blackscholes—Option pricing with Black-Scholes PDE Bodytrack—Body tracking of a person Canneal—Simulated cache-aware annealing to optimize routing Dedup—Next-generation compression with data deduplication Facesim—Simulates the motions of a human face Ferret—Content similarity search server Fluidanimate—Fluid dynamics for animation with SPH method Freqmine—Frequent itemset mining Streamcluster—Online clustering of an input stream Swaptions—Pricing of a portfolio of swaptions Vips—Image processing x264—H.264 video encoding
Berkeley Design Patterns [Asanovic et al., 2006]	Strong or Weak	Yes	Finite-State Machine Combinational Logic Graph Traversal Structured Grid Dense Matrix Sparse Matrix Spectral Methods (FFT) Dynamic Programming N-Body MapReduce Backtrack/Branch and Bound Graphical Model Inference Unstructured Grid

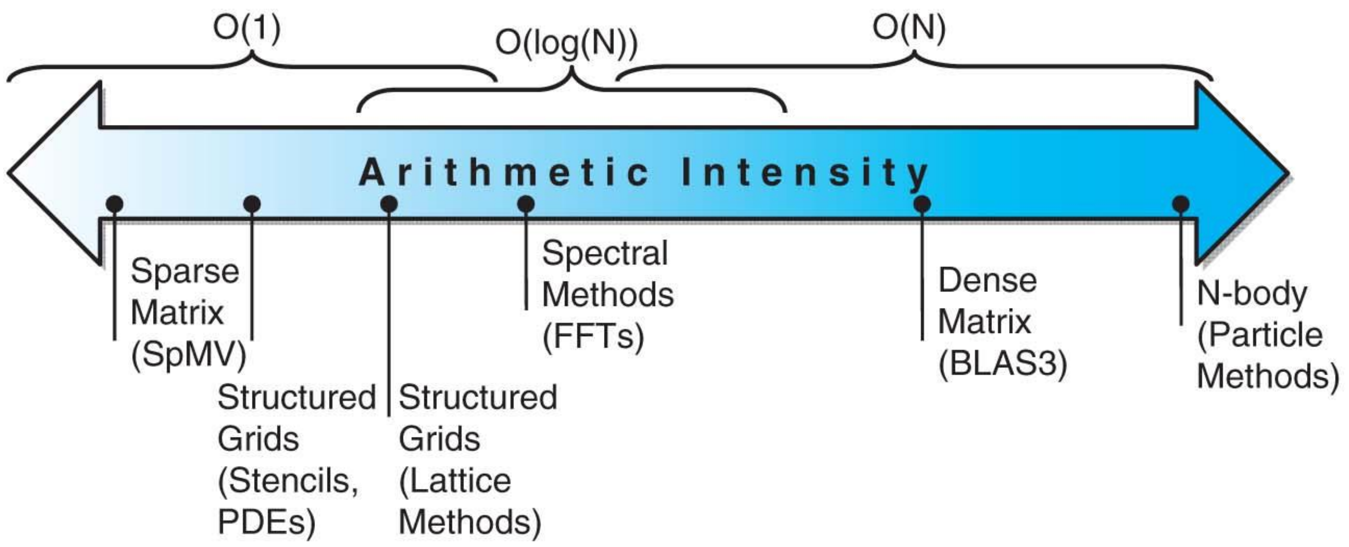
Roofline: A Simple Performance Model (1/3)

- Motivations
  - Increasing diversity in microprocessor design
    - # of cores increasing
    - A wide variety of optimization techniques for different microprocessor designs
  - A simple model that offered insights into the performance different designs is desired
    - Not be perfect, just insightful
    - E.g. 3Cs model for caches
- Factors limiting attainable performance
  - Peak floating point performance (Flop/s) and memory bandwidth (B/s) of a system to run a program
  - Arithmetic Intensity of the program to be executed on the system

Roofline: A Simple Performance Model (2/3)

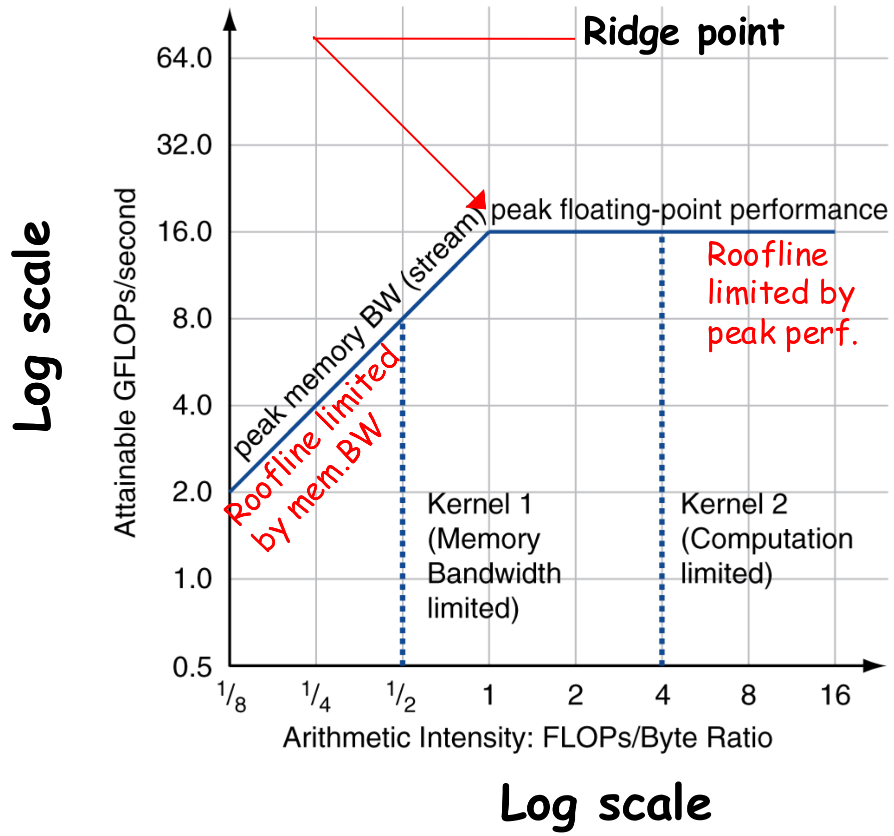
Arithmetic Intensity

- The ratio of floating-point operations in a program to the number of data bytes accessed by a program from main memory



- The demands on the memory system of a program
  - (Peak Flop/s)/(Arithmetic Intensity)=Bytes/Sec ⇦ >? or <? Peak BW

Roofline: A Simple Performance Model (2/3)



A model to tie floating point performance, arithmetic intensity, and memory performance together in a 2D graph

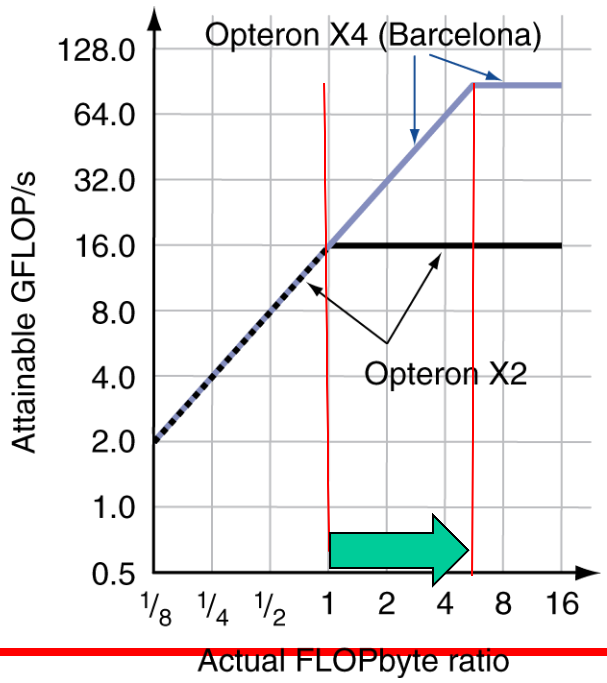
If ridge point is far to the right, only kernels with very high arithmetic intensity can achieve the maximum performance.

If ridge point is far to left, almost all any kernel can potentially hit the maximum performance

Attainable GPLOPs/sec
= Min (Peak Memory BW × Arithmetic Intensity, Peak FP Performance)

Comparing Two Generations of Opterons

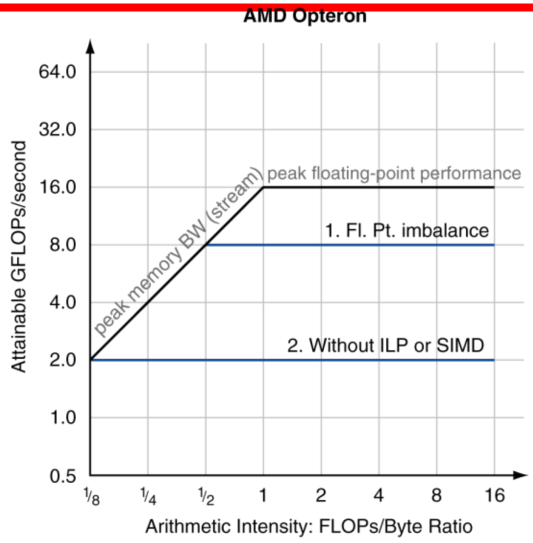
- Example: Opteron X2 vs. Opteron X4
  - X4 has more than 4x peak perf., but same memory BW
    - Opteron X2: 2-core, 1 FP SSE /clock per core, 2.2GHz
    - Opteron X4: 4-core, 2 FP SSEs/clock per core, 2.3GHz
    - Same memory system (socket compatible)

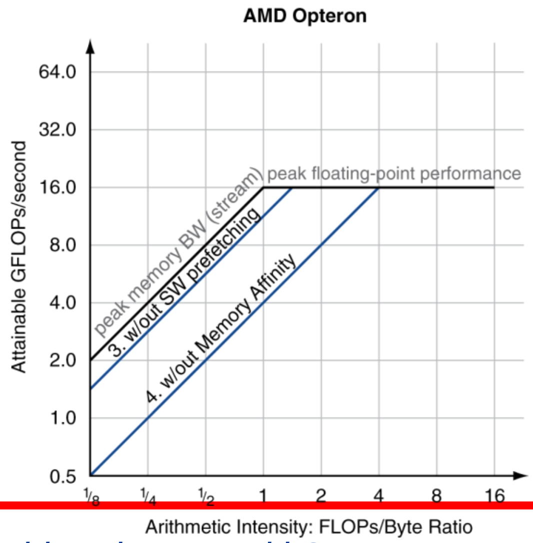


- **To get higher performance on X4 than X2**
  - **Need high arithmetic intensity >1**
  - **Or working set must fit in X4's 2MB L-3 cache**

Optimizing Performance (1/2)

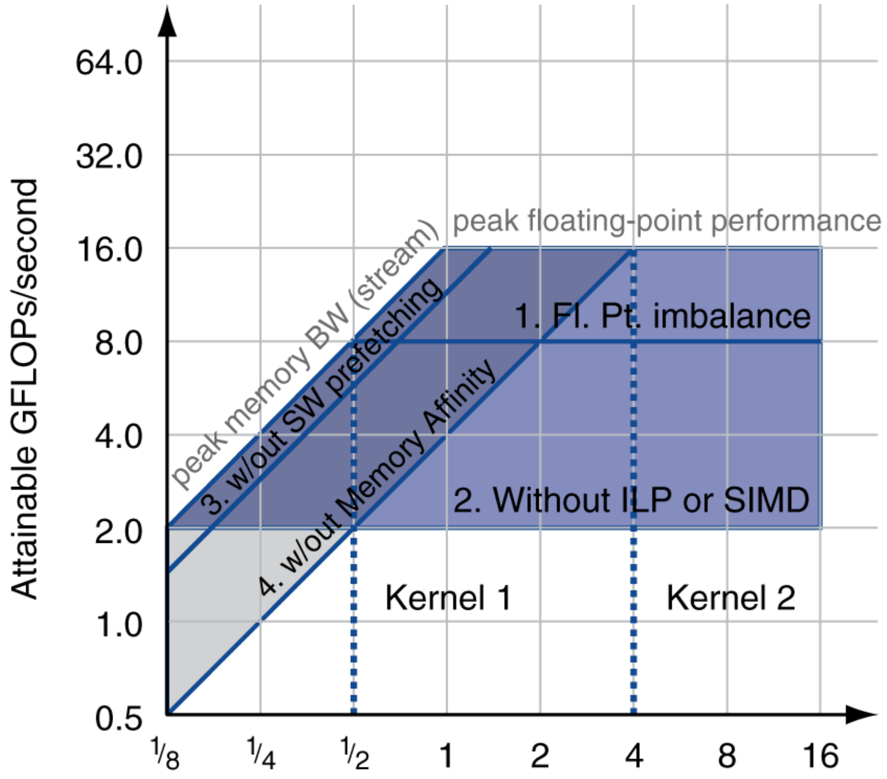
- Optimize FP performance
  - Balance adds & multiplies
    - Fused Mul-Adder and/or equal # of Multipliers and adders
    - Significant fraction of floating point operations, not integers
  - Improve superscalar ILP and use of SIMD instructions
    - E.g., Apply loop unrolling, exploit data level parallelism
- Optimize memory usage
  - Software prefetch
    - Avoid load stalls
  - Memory affinity
    - Enforce localized data to be placed in the local memory, because each chip has its own local memory, even though it is shared with others
    - Avoid non-local data accesses





Optimizing Performance (2/2)

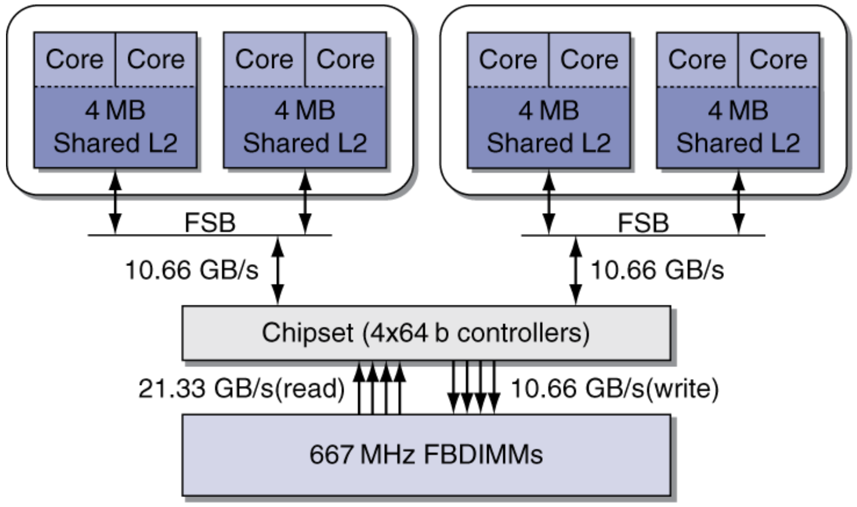
- Choice of optimization depends on arithmetic intensity of code



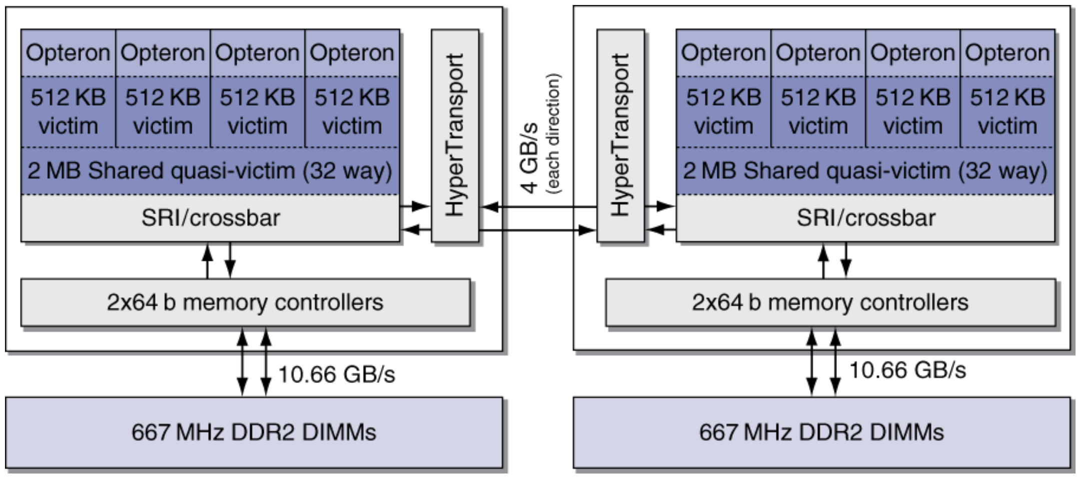
- **Arithmetic intensity is not always fixed**
  - **May scale with problem size**
  - **Caching reduces memory accesses, and increases arithmetic intensity**
- **A larger margin toward the ceiling is a strong motivation to choose its optimization (e.g., 2 and 4)**

The roofline model can help decide which of these optimizations to perform and the order in which to perform them!

Four Example Systems (1/3)

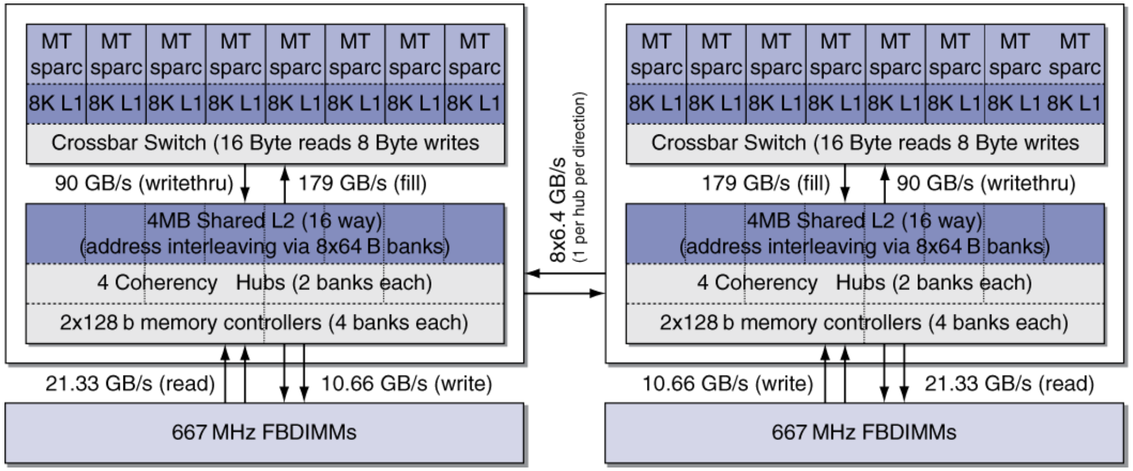


2 × quad-core
Intel Xeon e5345
(Clovertown)
UMA architecture



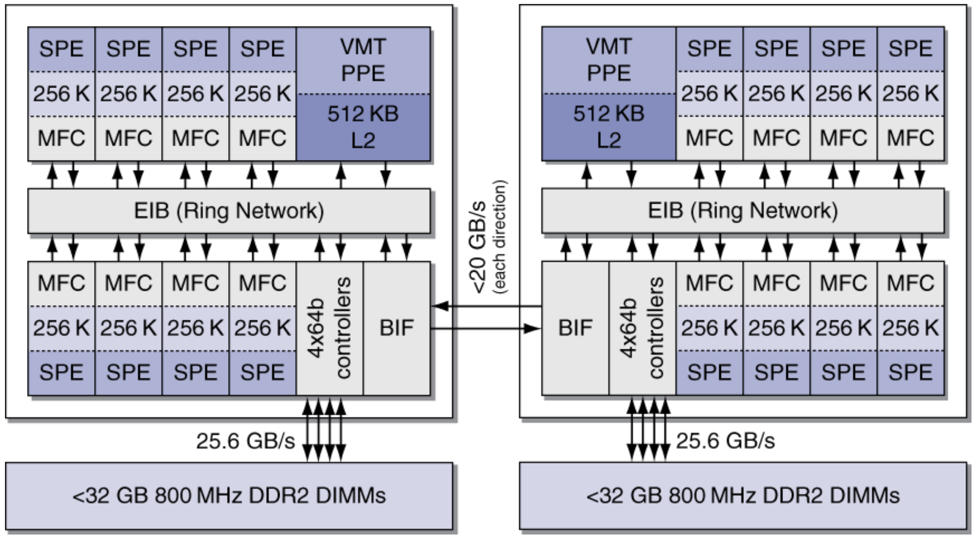
2 × quad-core
AMD Opteron X4 2356
(Barcelona)
NUMA architecture

Four Example Systems (2/3)



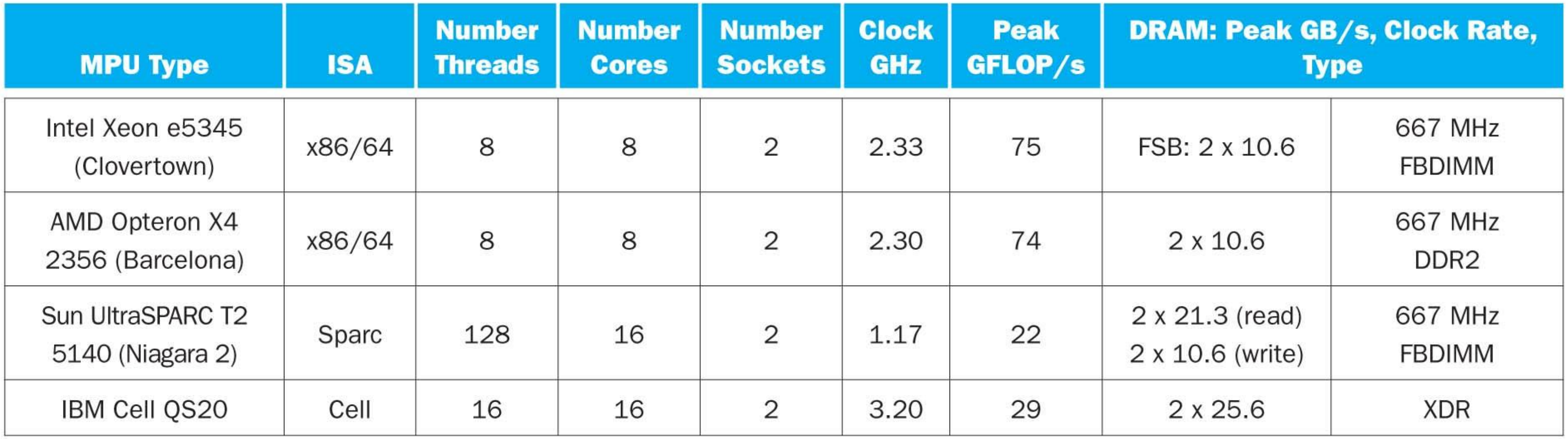
MT sparc
8K L1
Crossbar Switch (16 Byte reads 8 Byte writes
90 GB/s (writethru)
179 GB/s (fill)
4MB Shared L2 (16 way) (address interleaving via 8x64 B banks)
4 Coherency Hubs (2 banks each)
2x128 b memory controllers (4 banks each)
8x6.4 GB/s (1 per hub per direction)
21.33 GB/s (read)
10.66 GB/s (write)
667 MHz FBDIMMs

2 × oct-core
Sun UltraSPARC
T2 5140 (Niagara 2)
NUMA architecture



2 × oct-core
IBM Cell QS20

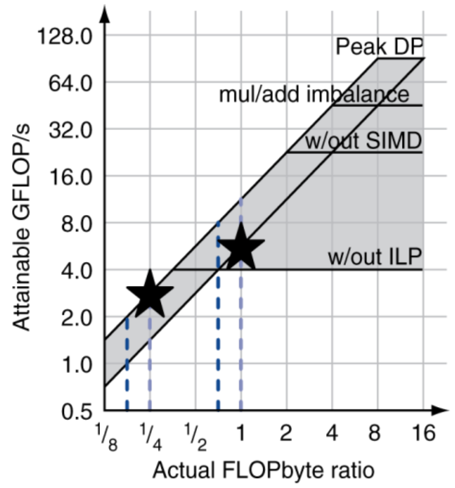
Four Example Systems (3/3)



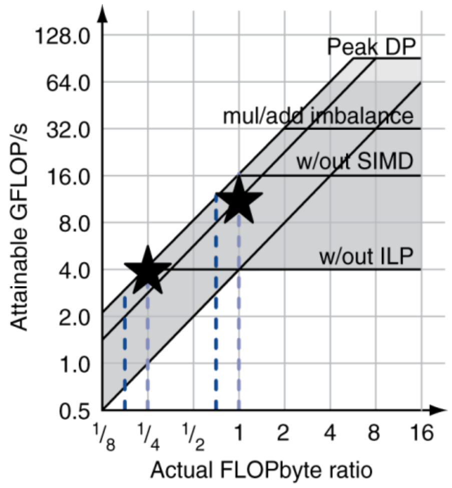
MPU Type	ISA	Number Threads	Number Cores	Number Sockets	Clock GHz	Peak GFLOP/s	DRAM: Peak GB/s, Clock Rate, Type	
Intel Xeon e5345 (Clovertown)	x86/64	8	8	2	2.33	75	FSB: 2 x 10.6	667 MHz FBDIMM
AMD Opteron X4 2356 (Barcelona)	x86/64	8	8	2	2.30	74	2 x 10.6	667 MHz DDR2
Sun UltraSPARC T2 5140 (Niagara 2)	Sparc	128	16	2	1.17	22	2 x 21.3 (read) 2 x 10.6 (write)	667 MHz FBDIMM
IBM Cell QS20	Cell	16	16	2	3.20	29	2 x 25.6	XDR

Rooflines of Four Systems

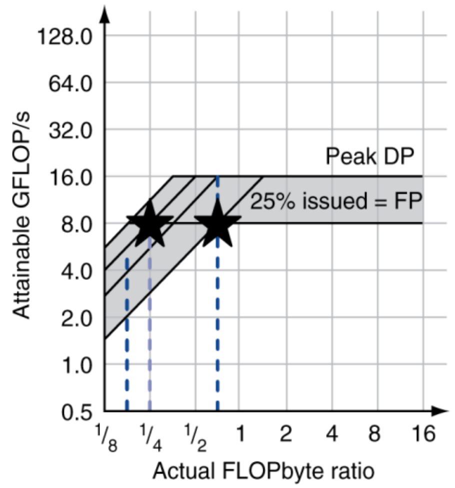
- Kernels
  - SpMV (left)
  - LBHMD (right)
- Some optimizations change arithmetic intensity
- x86 systems have higher peak GFLOPs
  - But harder to achieve, given memory bandwidth



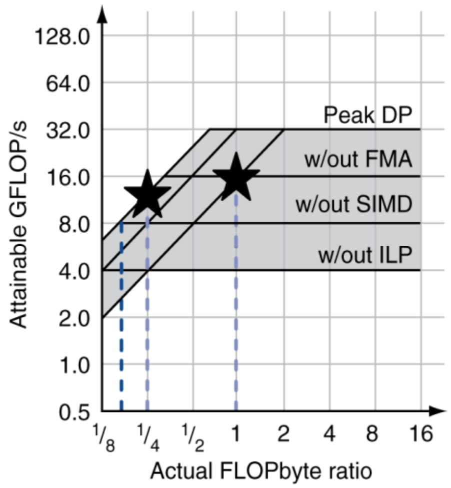
a. Intel Xeon e5345 (Clovertown)



b. AMD Opteron X4 2356 (Barcelona)



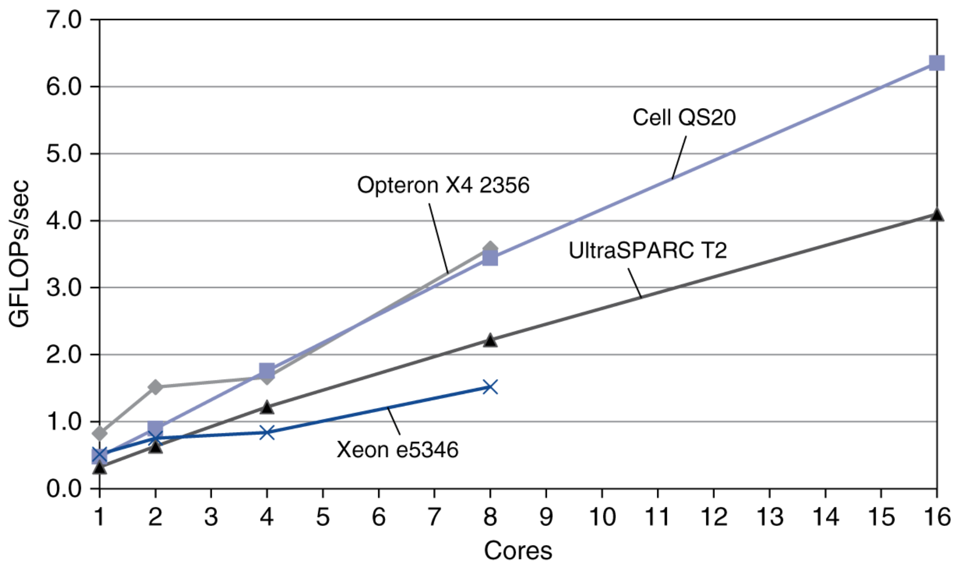
c. Sun UltraSPARC T2 5140 (Niagara 2)



d. IBM Cell QS20

Performance on Sparse Matrix-Vector Multiply

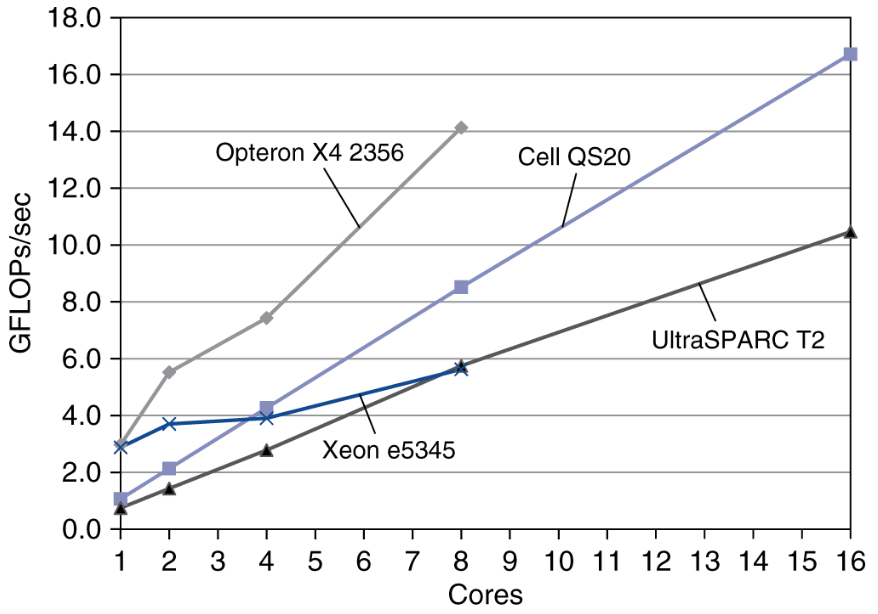
- Sparse matrix/vector multiply: y=A*x, A: sparse matrix
  - Irregular memory accesses, memory bound
- Arithmetic intensity
  - 0.166 before register usage optimization, 0.25 after



- Xeon vs. Opteron
  - Similar peak FLOPS, but Opteron is 2x than xeon
    - Xeon limited by shared FSBs and chipset
- UltraSPARC/Cell vs. x86
  - 20 – 30 vs. 75 peak GFLOPs
  - More simple cores with higher memory bandwidth outperform two X86 processors

Performance of LBMHD on the Four Multicores

- Lattice-Boltzmann Magneto-Hydrodynamics(LBMHD) is popular for computational fluid dynamics, which is a structured grid code over time steps)
  - Each point: 75 FP read/write and 1300 FP ops
- Arithmetic intensity
  - 0.70 before cache optimization of “no-allocation on write miss” for Xeon, X4, T2, 1.07 for Xeon, X4, Cell (no-cache) after



- Opteron vs. UltraSPARC
  - More powerful cores achieve higher Gflop/s at arithmetic intensity of 1.07
- Xeon vs. others
  - Still suffers from memory bottlenecks

Achieving Performance

- Compare naïve vs. optimized code
  - If naïve code performs well, it's easier to write high performance code for the system

System	Kernel	Naïve GFLOPs/sec	Optimized GFLOPs/sec	Naïve as % of optimized
Intel Xeon	SpMV	1.0	1.5	64%
	LBMHD	4.6	5.6	82%
AMD Opteron X4	SpMV	1.4	3.6	38%
	LBMHD	7.1	14.1	50%
Sun UltraSPARC T2	SpMV	3.5	4.1	86%
	LBMHD	9.7	10.5	93%
IBM Cell QS20	SpMV	Naïve code not feasible	6.4	0%
	LBMHD		16.7	0%

Fallacies and Pitfalls

Fallacies

- Amdahl's Law doesn't apply to parallel computers
  - Since we can achieve linear speedup
  - But only on applications with weak scaling
- Peak performance tracks observed performance
  - Marketers like this approach!
  - But compare Xeon with others in example
  - Need to be aware of bottlenecks

Pitfalls

- Not developing the software to take account of a multiprocessor architecture
  - Example: using a single lock for a shared composite resource
    - Serializes accesses, even if they could be done in parallel
    - Use finer-granularity locking

Concluding Remarks

- Goal: higher performance by using multiple processors
- Difficulties
  - Developing parallel software
  - Devising appropriate architectures
- Many reasons for optimism
  - Changing software and application environment
  - Chip-level multiprocessors with lower latency, higher bandwidth interconnect
- An ongoing challenge for computer architects!