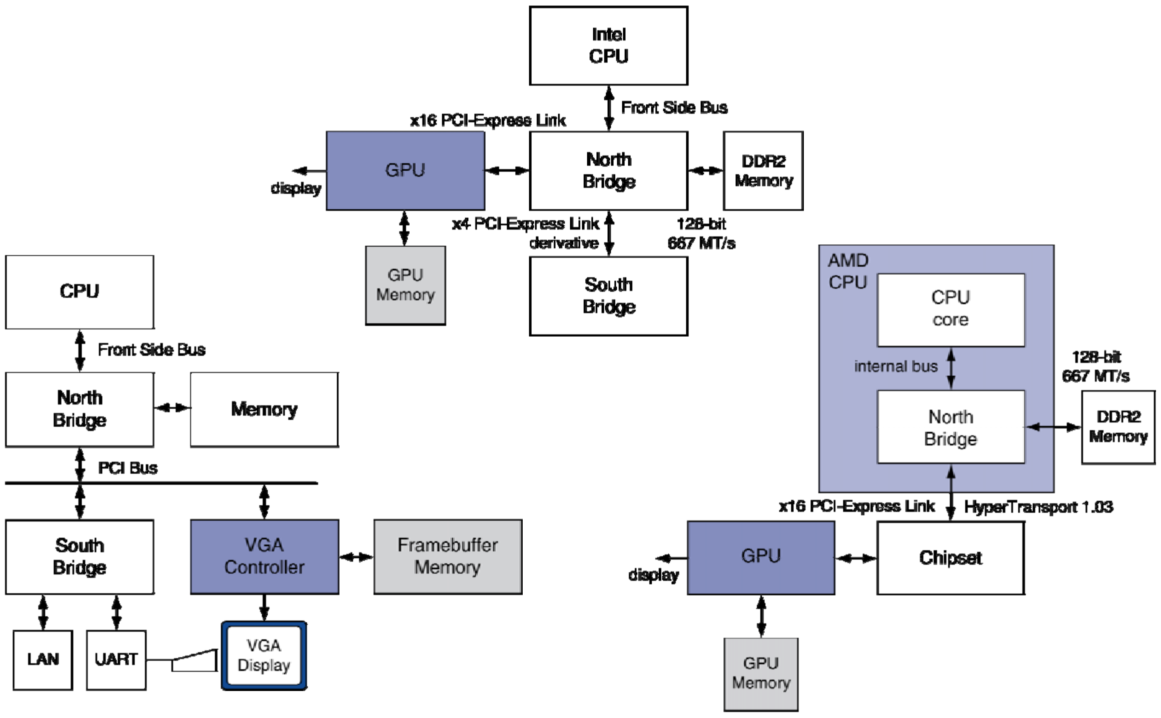
Chapter Six (2/2)

History of GPUs

- Early video cards
  - Frame buffer memory with address generation for video output
- 3D graphics processing
  - Originally high-end computers (e.g., SGI)
  - Moore's Law ⇒ 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

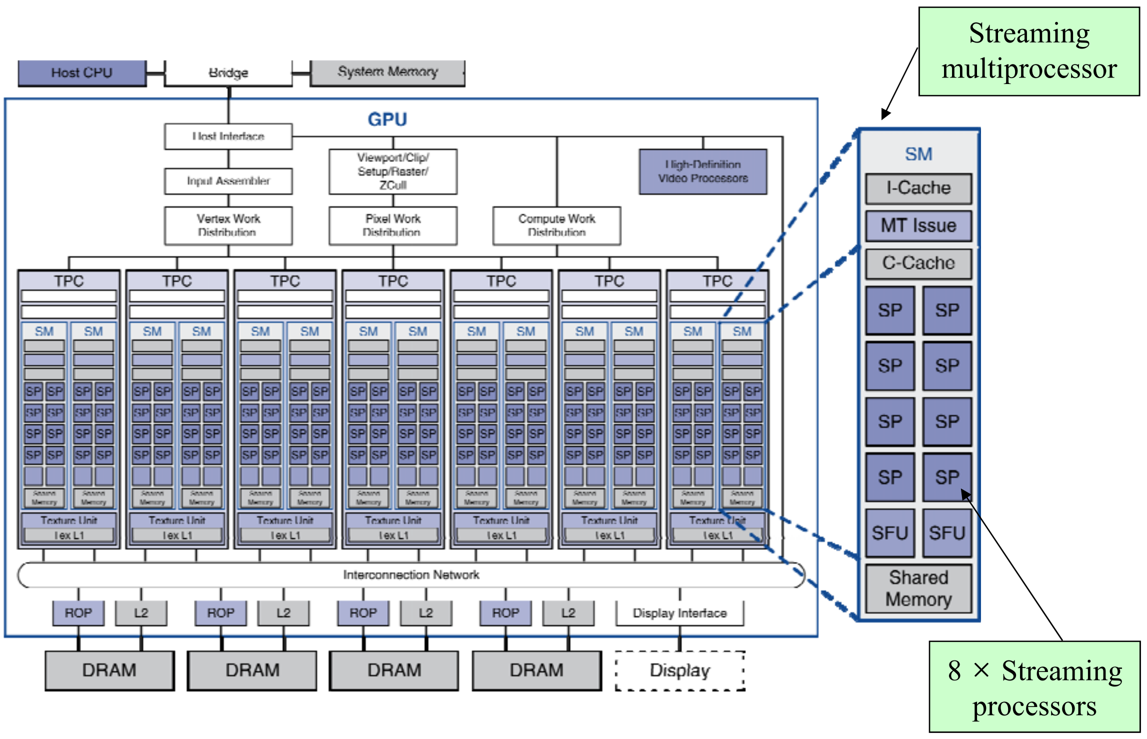
Graphics in the System



GPU Architectures

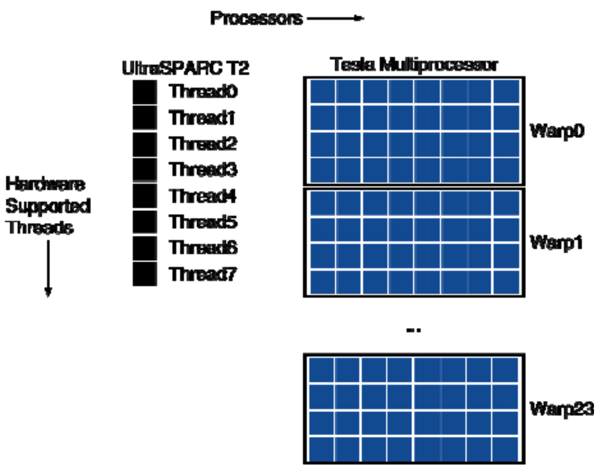
- **Processing is highly data-parallel**
  - **GPUs are highly multithreaded**
  - **Use thread switching to hide memory latency**
    - Less reliance on multi-level caches
  - **Graphics memory is wide and high-bandwidth**
- **Trend toward general purpose GPUs**
  - **Heterogeneous CPU/GPU systems**
  - **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)**
  - **OpenCL**

Example: NVIDIA Tesla



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**

Coherence Defined

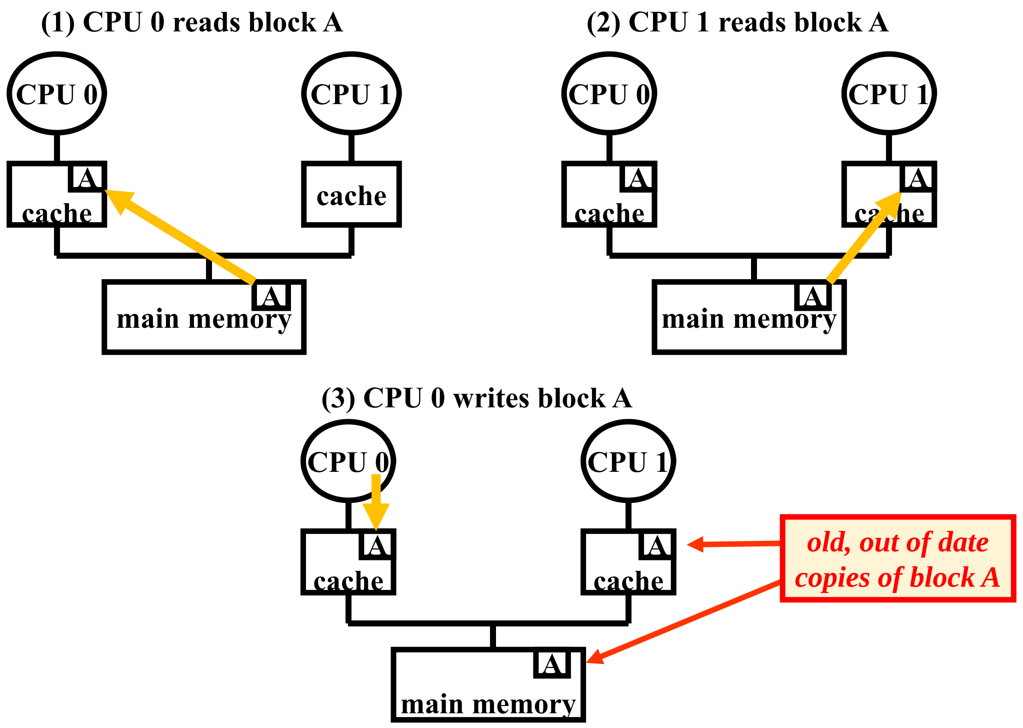
- **Informally: Reads return most recently written value**
- **Formally:**
  - **P writes X; P reads X (no intervening writes)**
    ⇒ read returns written value
  - **P1 writes X; P2 reads X (sufficiently later)**
    ⇒ read returns written value
    - c.f. CPU B reading X after step 3 in example
  - **P1 writes X, P2 writes X**
    ⇒ all processors see writes in the same order
    - End up with the same final value for X

Cache Coherence Protocols

- **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**
- **Directory-based protocols**
  - **Caches and memory record sharing status of blocks in a directory**

Cache Coherence Problem

- **Two writeback caches becoming incoherent**



Cache Coherence Problem

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

Time step	Event	CPU A's cache	CPU B's cache	Memory X
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

Cache coherence protocols

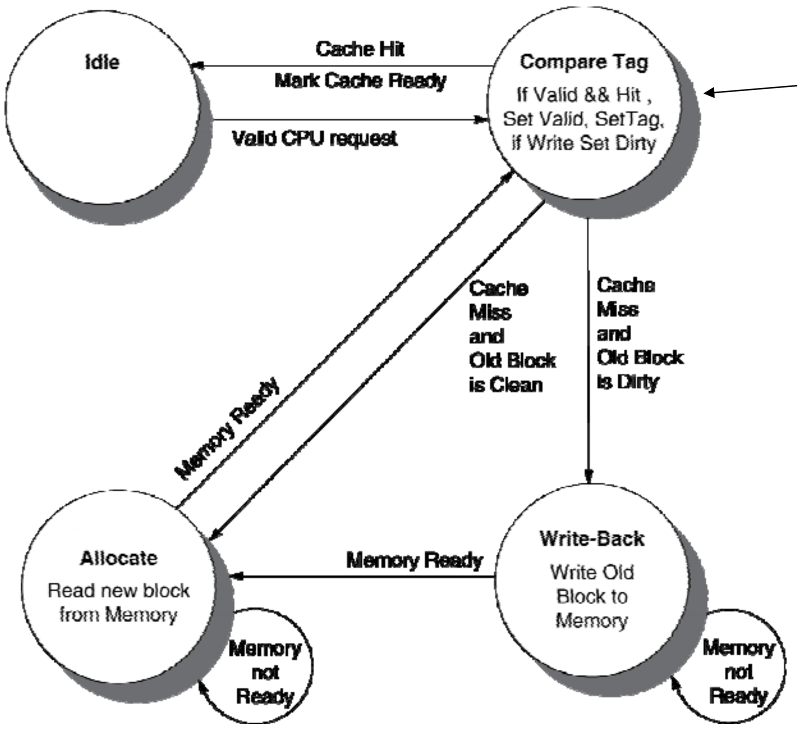
- **Ensures that cached blocks that are written to are observable by all processors**
- **Assigns a *state* field to all cached blocks**
- **Defines actions for performing reads and writes to blocks in each state that ensure cache coherence**
- **Actions are much more complicated than described here in a real machine with a split transaction bus**

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

CPU activity	Bus activity	CPU A's cache	CPU B's cache	Memory X
				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

Cache Controller FSM for a Single Processor



Could partition into separate states to reduce clock cycle time

MESI cache coherence protocol

- Commonly used (or variant thereof) in shared memory multiprocessors
- Idea is to ensure that when a cache wants to write to a cache block that other remote caches *invalidate* their copies first
- Each cache block is in one of four states (2 bits stored with each cache block)
  - Invalid: contents are not valid
  - Shared: other processor caches *may* have the same copy; main memory has the same copy
  - Exclusive: no other processor cache has a copy; main memory has the same copy
  - Modified: no other processor cache has a copy; main memory has an old copy

MESI cache coherence protocol

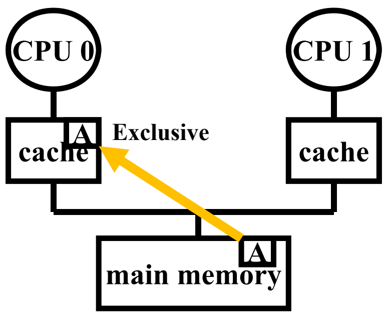
- Cache read actions
  - Hit – local cache actions
    - Read block
  - Hit – remote cache actions
    - None
  - Miss – local cache actions
    - Request block from bus
    - If not in a remote cache, set state to **Exclusive**
    - If also in a remote cache, set state to **Shared**
  - Miss – remote cache actions
    - Look up cache tags to see if the block is present
    - If so, signal the local cache that we have a copy, provide it if it is in state ***Modified***, and change the state of our copy to **Shared**

MESI cache coherence protocol

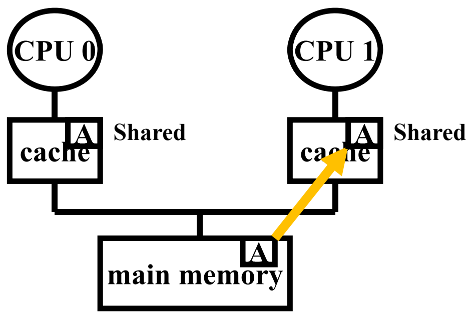
- **Cache write actions**
  - **Hit – local cache actions**
    - Check state of block
    - If Shared, send an ***Invalidation*** bus command to all remote caches
    - Write the block and change the state to **Modified**
  - **Hit – remote cache actions**
    - Upon receipt of an Invalidation command on the bus, look up cache tags to see if the block is present
    - If so, change the state of the block to **Invalid**
  - **Miss – local cache actions**
    - Simultaneously ***request*** block from bus and send an ***Invalidation*** command
    - After block received, write the block and set the state to **Modified**
  - **Miss – remote cache actions**
    - Look up cache tags to see if the block is present
    - If so, signal the local cache that we have a copy, provide it if it is in state ***Modified***, and change the state of our copy to **Invalid**

Cache coherence problem revisited

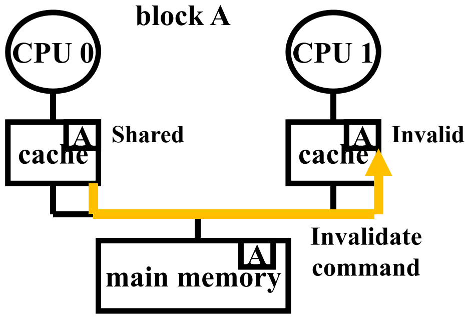
(1) CPU 0 reads block A



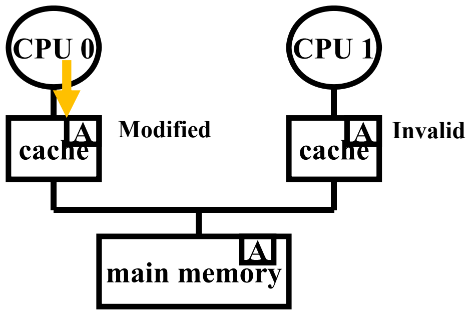
(2) CPU 1 reads block A



(3) CPU 0 cache invalidates remote block A



(4) CPU 0 writes block A



Memory Consistency

- When are writes seen by other processors
  - "Seen" means a read returns the written value
  - Can't be instantaneously
- Assumptions
  - A write completes only when all processors have seen it
  - A processor does not reorder writes with other accesses
- Consequence
  - P writes X then writes Y
    ⇒ all processors that see new Y also see new X
  - Processors can reorder reads, but not writes

- Example

  P1: A=1; B=1; while B=1 do nothing; print A;

  P2: A=0; B=0;

Parallel Benchmarks

- Linpack: matrix linear algebra
- SPECrate: parallel run of SPEC CPU programs
  - Job-level parallelism
- SPLASH: Stanford Parallel Applications for Shared Memory
  - Mix of kernels and applications, strong scaling
- NAS (NASA Advanced Supercomputing) suite
  - computational fluid dynamics kernels
- PARSEC (Princeton Application Repository for Shared Memory Computers) suite
  - Multithreaded applications using Pthreads and OpenMP

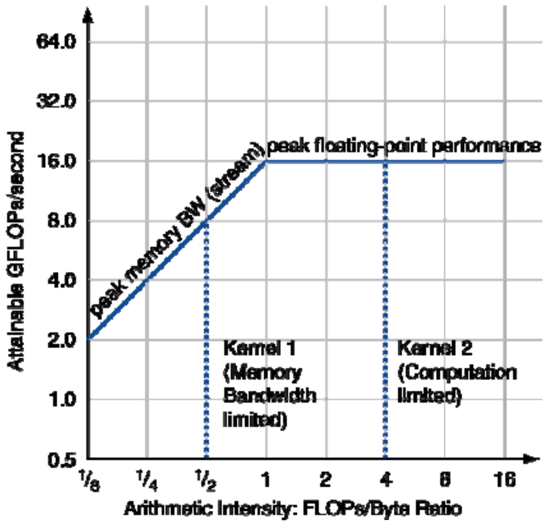
Code or Applications?

- Traditional benchmarks
  - Fixed code and data sets
- Parallel programming is evolving
  - Should algorithms, programming languages, and tools be part of the system?
  - Compare systems, provided they implement a given application
  - E.g., Linpack, Berkeley Design Patterns
- Would foster innovation in approaches to parallelism

Modeling Performance

- Assume performance metric of interest is achievable GFLOPs/sec
  - Measured using computational kernels from Berkeley Design Patterns
- Arithmetic intensity of a kernel
  - FLOPs per byte of memory accessed
- For a given computer, determine
  - Peak GFLOPS (from data sheet)
  - Peak memory bytes/sec (using Stream benchmark)

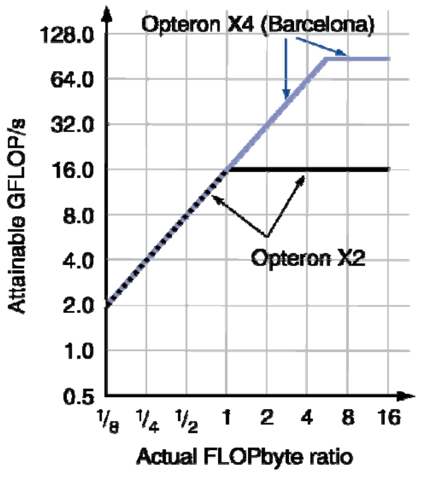
Roofline Diagram



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

Comparing Systems

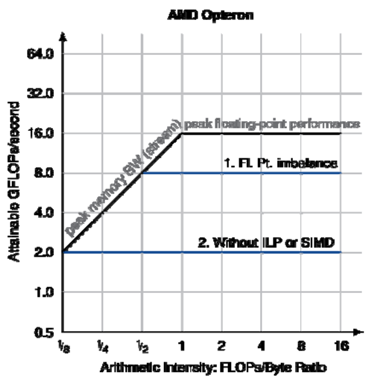
- Example: Opteron X2 vs. Opteron X4
  - 2-core vs. 4-core, 2× FP performance/core, 2.2GHz vs. 2.3GHz
  - Same memory system

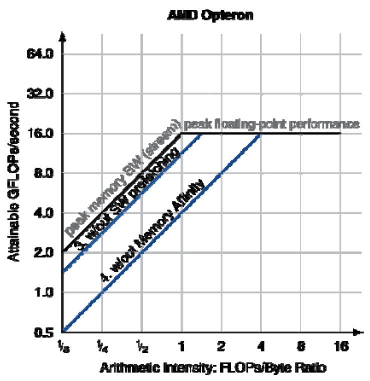


- To get higher performance on X4 than X2
  - Need high arithmetic intensity
  - Or working set must fit in X4's 2MB L-3 cache

Optimizing Performance

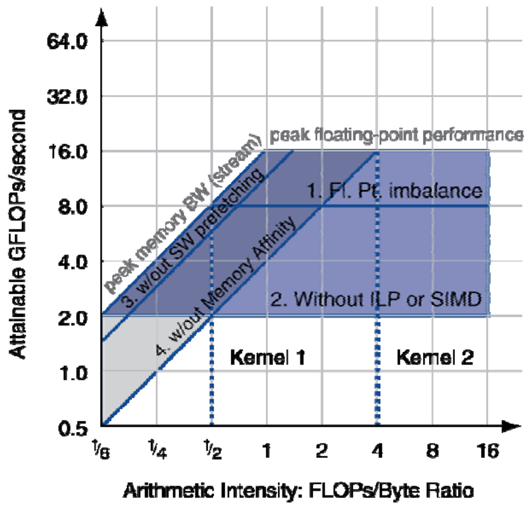
- Optimize FP performance
  - Balance adds & multiplies
  - Improve superscalar ILP and use of SIMD instructions
- Optimize memory usage
  - Software prefetch
    - Avoid load stalls
  - Memory affinity
    - Avoid non-local data accesses





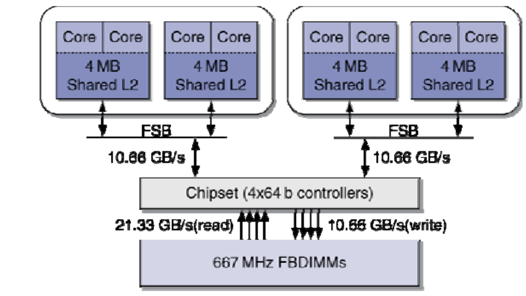
Optimizing Performance

- Choice of optimization depends on arithmetic intensity of code

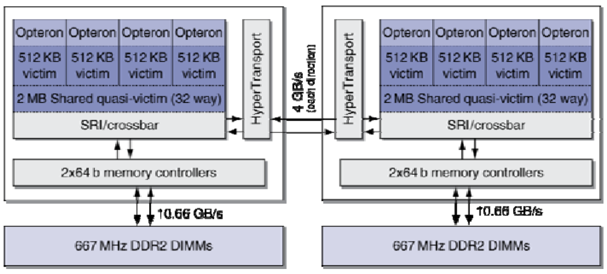


- Arithmetic intensity is not always fixed
  - May scale with problem size
  - Caching reduces memory accesses
    - Increases arithmetic intensity

Four Example Systems

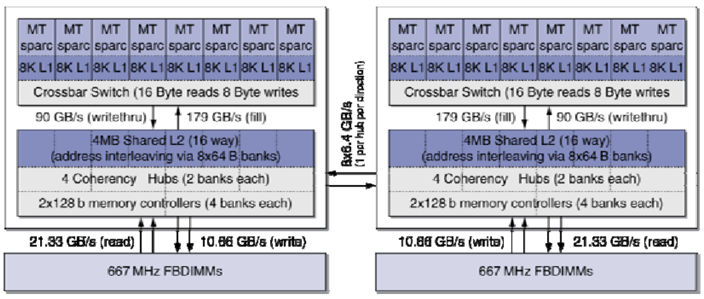


2 × quad-core
Intel Xeon e5345
(Clovertown)



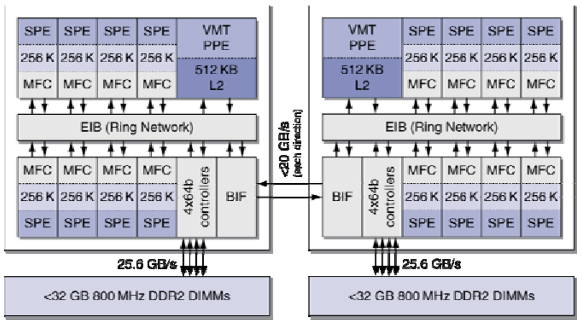
2 × quad-core
AMD Opteron X4 2356
(Barcelona)

Four Example Systems



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

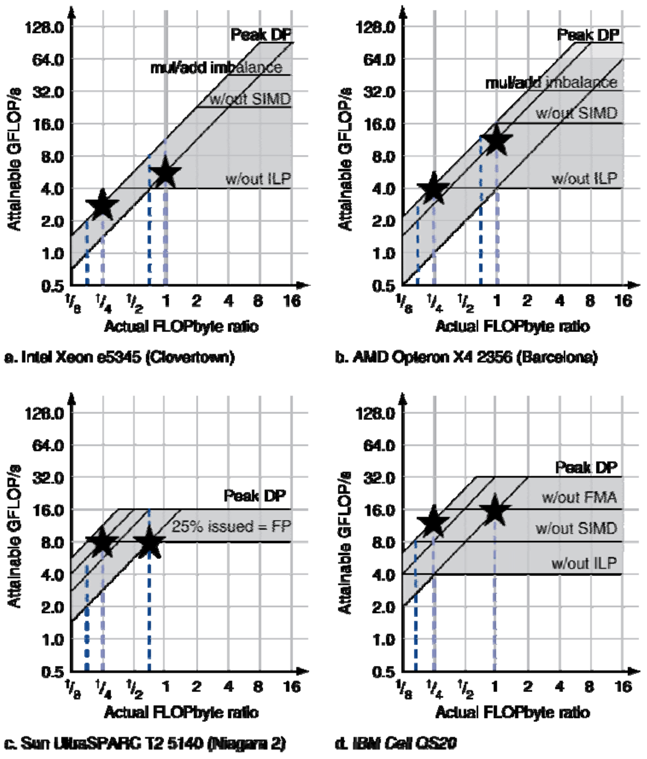
2 × oct-core
Sun UltraSPARC
T2 5140 (Niagara 2)



2 × oct-core
IBM Cell QS20

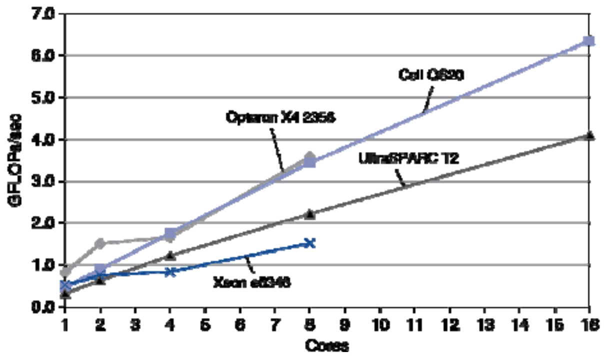
And Their Rooflines

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



Performance on SpMV

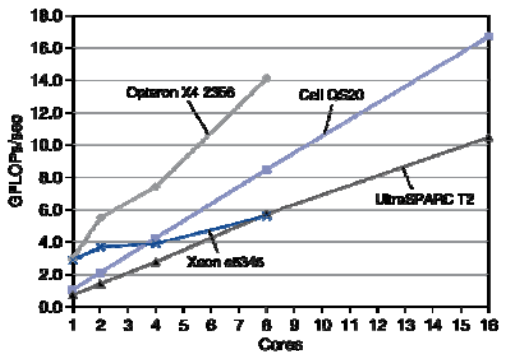
- Sparse matrix/vector multiply
  - Irregular memory accesses, memory bound
- Arithmetic intensity
  - 0.166 before memory optimization, 0.25 after



- Xeon vs. Opteron
  - Similar peak FLOPS
  - Xeon limited by shared FSBs and chipset
- UltraSPARC/Cell vs. x86
  - 20 – 30 vs. 75 peak GFLOPs
  - More cores and memory bandwidth

Performance on LBMHD

- Fluid dynamics: structured grid over time steps
  - Each point: 75 FP read/write, 1300 FP ops
- Arithmetic intensity
  - 0.70 before optimization, 1.07 after



- Opteron vs. UltraSPARC
  - More powerful cores, not limited by memory bandwidth
- 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 LBMHD	1.0 4.6	1.5 5.6	64% 82%
AMD Opteron X4	SpMV LBMHD	1.4 7.1	3.6 14.1	38% 50%
Sun UltraSPARC T2	SpMV LBMHD	3.5 9.7	4.1 10.5	86% 93%
IBM Cell QS20	SpMV LBMHD	Naïve code not feasible	6.4 16.7	0% 0%

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!