

Chapter 6

Parallel Processors from Client to Cloud and Storage, Bus, and I/O

Tien-Fu Chen

Dept. of Computer Science
National Chiao Tung Univ.

Material source:
RISC-V slides

Introduction

- ❑ **Goal: connecting multiple computers to get higher performance**
 - Multiprocessors
 - Scalability, availability, power efficiency
- ❑ **Task-level (process-level) parallelism**
 - High throughput for independent jobs
- ❑ **Parallel processing program**
 - Single program run on multiple processors
- ❑ **Multicore microprocessors**
 - Chips with multiple processors (cores)

Amdahl's Law

- ❑ Sequential part can limit speedup
- ❑ Example: 100 processors, 90× speedup?

- $T_{\text{new}} = T_{\text{parallelizable}}/100 + T_{\text{sequential}}$

-

$$\text{Speedup} = \frac{1}{(1 - F_{\text{parallelizable}}) + F_{\text{parallelizable}}/100} = 90$$

- Solving: $F_{\text{parallelizable}} = 0.999$

- ❑ Need sequential part to be 0.1% of original time

Scaling Example

- ❑ **Workload: sum of 10 scalars, and 10×10 matrix sum**
 - Speed up from 10 to 100 processors
- ❑ **Single processor: Time = $(10 + 100) \times t_{\text{add}}$**
- ❑ **10 processors**
 - Time = $10 \times t_{\text{add}} + 100/10 \times t_{\text{add}} = 20 \times t_{\text{add}}$
 - Speedup = $110/20 = 5.5$ (55% of potential)
- ❑ **100 processors**
 - Time = $10 \times t_{\text{add}} + 100/100 \times t_{\text{add}} = 11 \times t_{\text{add}}$
 - Speedup = $110/11 = 10$ (10% of potential)
- ❑ **Assumes load can be balanced across processors**

Scaling Example (cont)

- ❑ What if matrix size is 100×100 ?
- ❑ Single processor: Time = $(10 + 10000) \times t_{\text{add}}$
- ❑ 10 processors
 - Time = $10 \times t_{\text{add}} + 10000/10 \times t_{\text{add}} = 1010 \times t_{\text{add}}$
 - Speedup = $10010/1010 = 9.9$ (99% of potential)
- ❑ 100 processors
 - Time = $10 \times t_{\text{add}} + 10000/100 \times t_{\text{add}} = 110 \times t_{\text{add}}$
 - Speedup = $10010/110 = 91$ (91% of potential)
- ❑ Assuming load balanced

Instruction and Data Streams

❑ An alternate classification

		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

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 RISC-V**
 - v0 to v31: 32×64 -element registers, (64-bit elements)
 - **Vector instructions**
 - ❑ fl d. v, fsd. v: load/store vector
 - ❑ fadd. d. v: add vectors of double
 - ❑ fadd. d. vs: add scalar to each element of vector of double
- ❑ **Significantly reduces instruction-fetch bandwidth**

Vector vs. Scalar

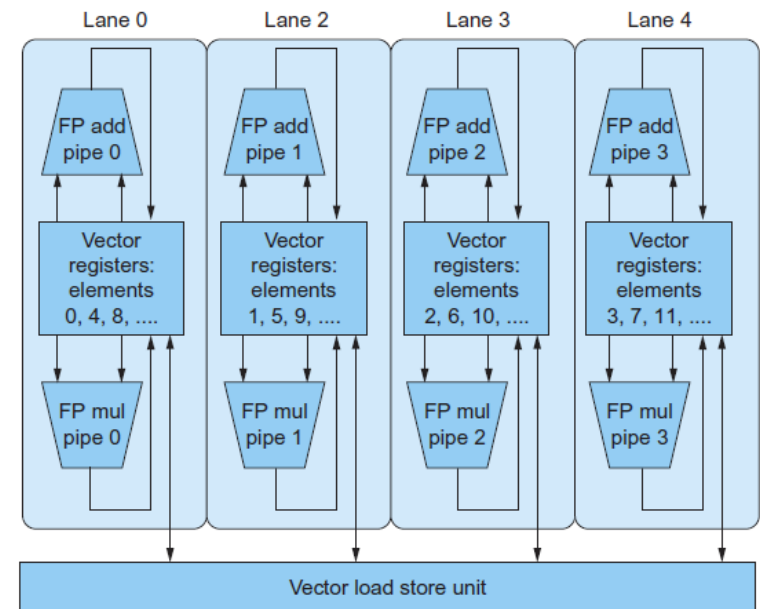
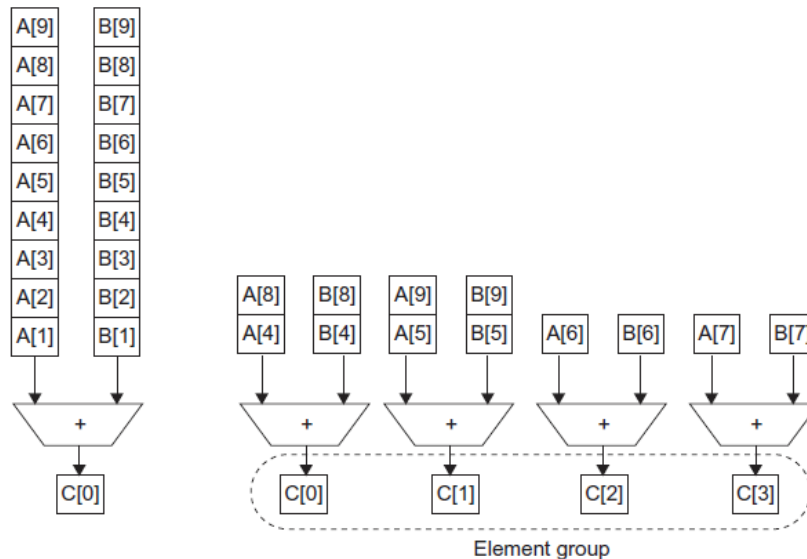
- ❑ **Vector architectures and compilers**
 - Simplify data-parallel programming
 - Explicit statement of absence of loop-carried dependences
 - ❑ Reduced checking in hardware
 - Regular access patterns benefit from interleaved and burst memory
 - Avoid control hazards by avoiding loops
- ❑ **More general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

SIMD

- ❑ Operate elementwise on vectors of data
 - E.g., MMX and SSE instructions in x86
 - ❑ Multiple data elements in 128-bit wide registers
- ❑ 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

Vector vs. Multimedia Extensions

- ❑ Vector instructions have a variable vector width, multimedia extensions have a fixed width
- ❑ Vector instructions support strided access, multimedia extensions do not
- ❑ Vector units can be combination of pipelined and arrayed functional units:

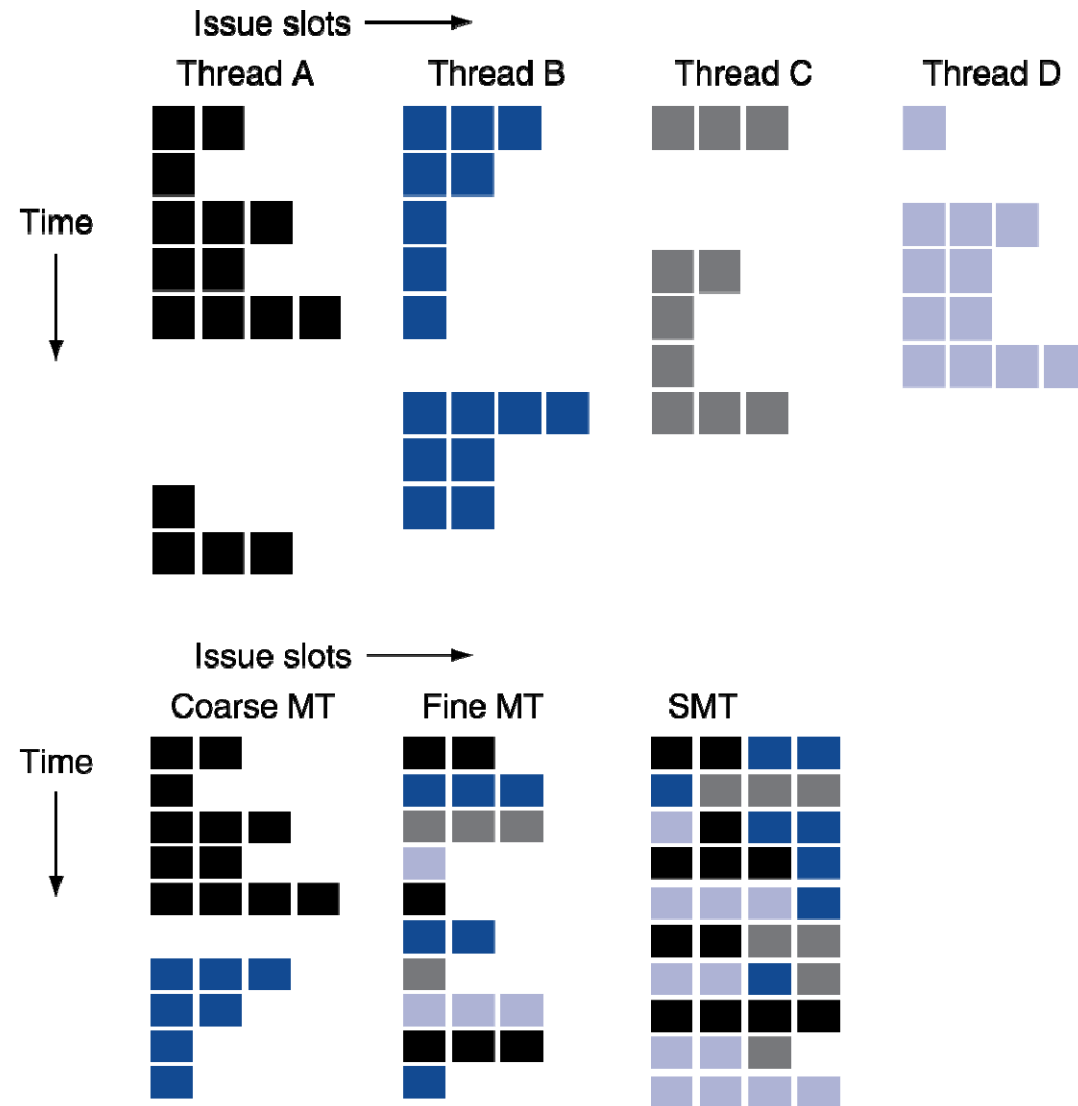


Multithreading

- ❑ **Performing multiple threads of execution in parallel**
 - Replicate registers, PC, etc.
 - Fast switching between threads
- ❑ **Fine-grain multithreading**
 - Switch threads after each cycle
 - Interleave instruction execution
 - If one thread stalls, others are executed
- ❑ **Coarse-grain multithreading**
 - Only switch on long stall (e.g., L2-cache miss)
 - Simplifies hardware, but doesn't hide short stalls (eg, data hazards)

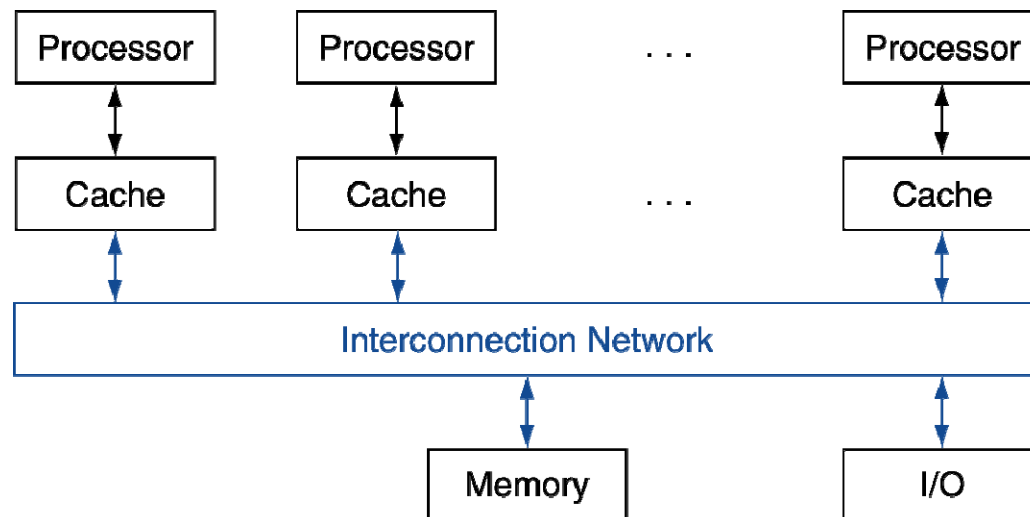
Simultaneous Multithreading

- ❑ In multiple-issue dynamically scheduled processor
 - Schedule instructions from multiple threads
 - Instructions from independent threads execute when function units are available
 - Within threads, dependencies handled by scheduling and register renaming



Shared Memory

- ❑ **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)



Example: Sum Reduction

❑ Sum 64,000 numbers on 64 processor UMA

- Each processor has ID: $0 \leq P_n \leq 63$
- Partition 1000 numbers per processor
- Initial summation on each processor

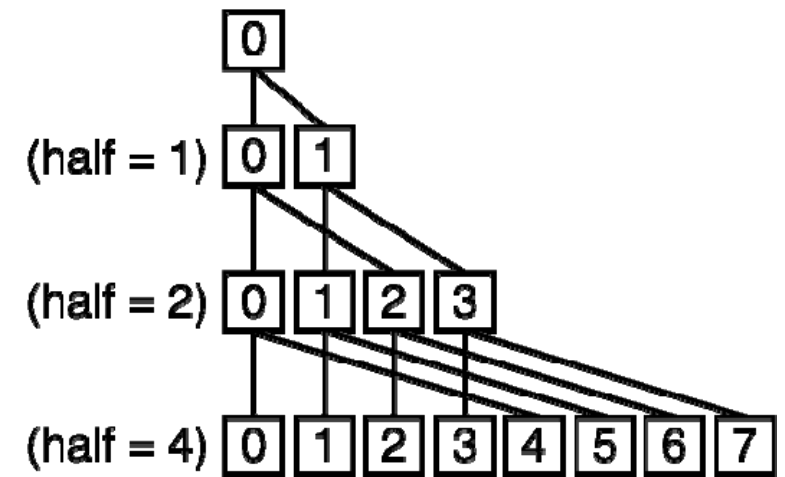
```
sum[Pn] = 0;  
for (i = 1000*Pn;  
     i < 1000*(Pn+1); i += 1)  
    sum[Pn] += A[i];
```

❑ Now need to add these partial sums

- Reduction: divide and conquer
- Half the processors add pairs, then quarter, ...
- Need to synchronize between reduction steps

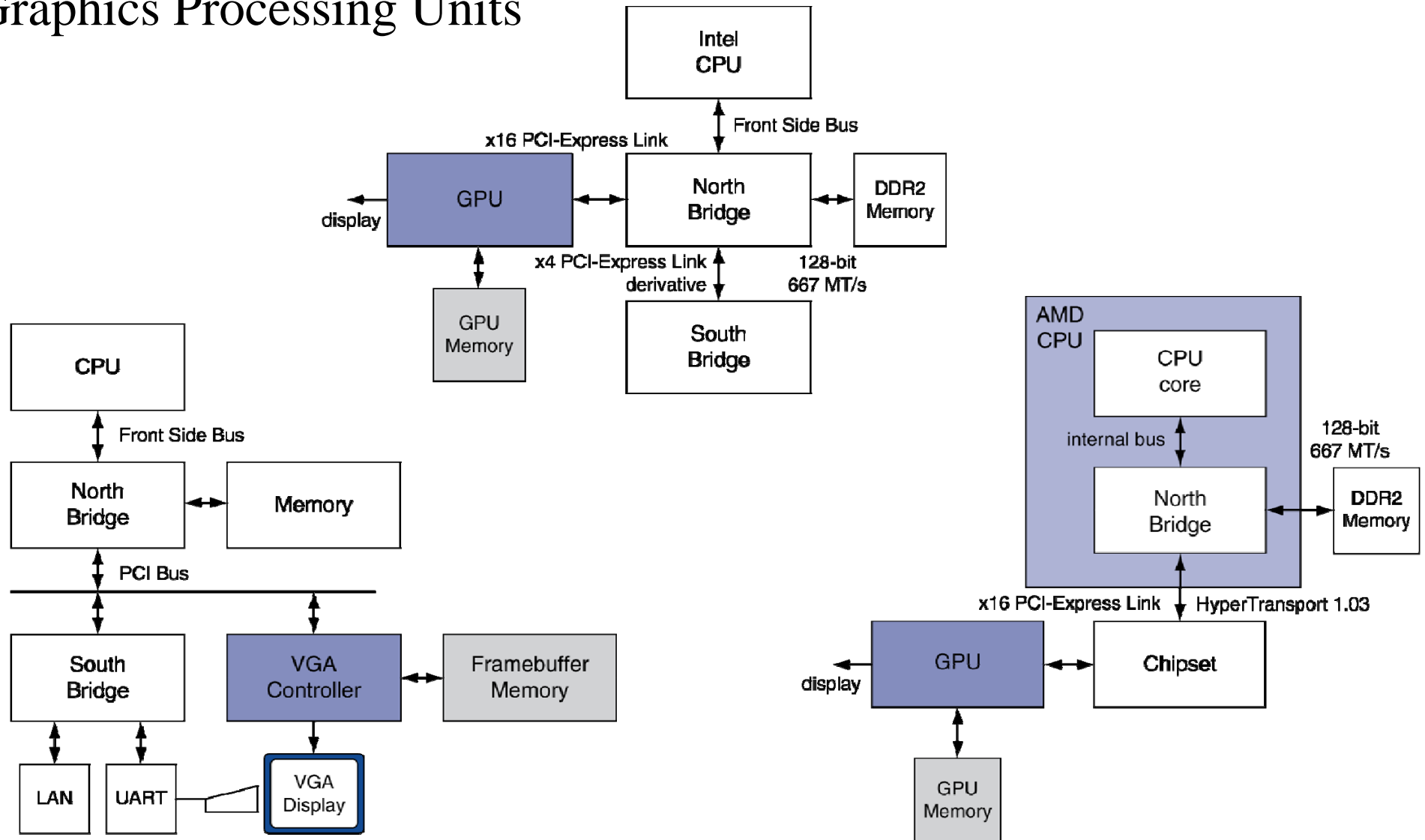
Example: Sum Reduction

```
half = 64;  
do  
    synch();  
    if (half % 2 != 0 && Pn == 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 + half];  
while (half > 1);
```



Graphics in the System

Graphics Processing Units

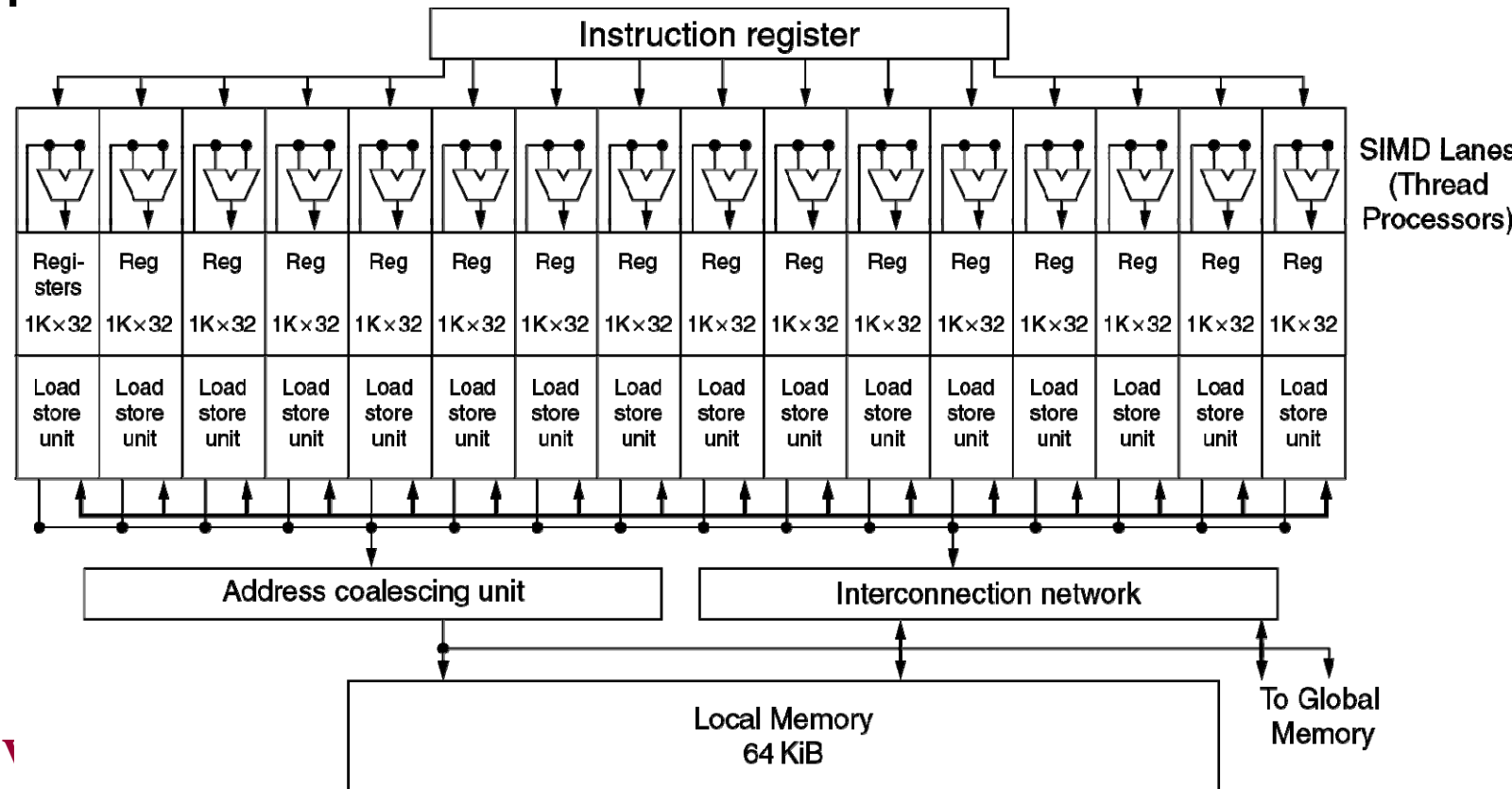


GPU Architectures

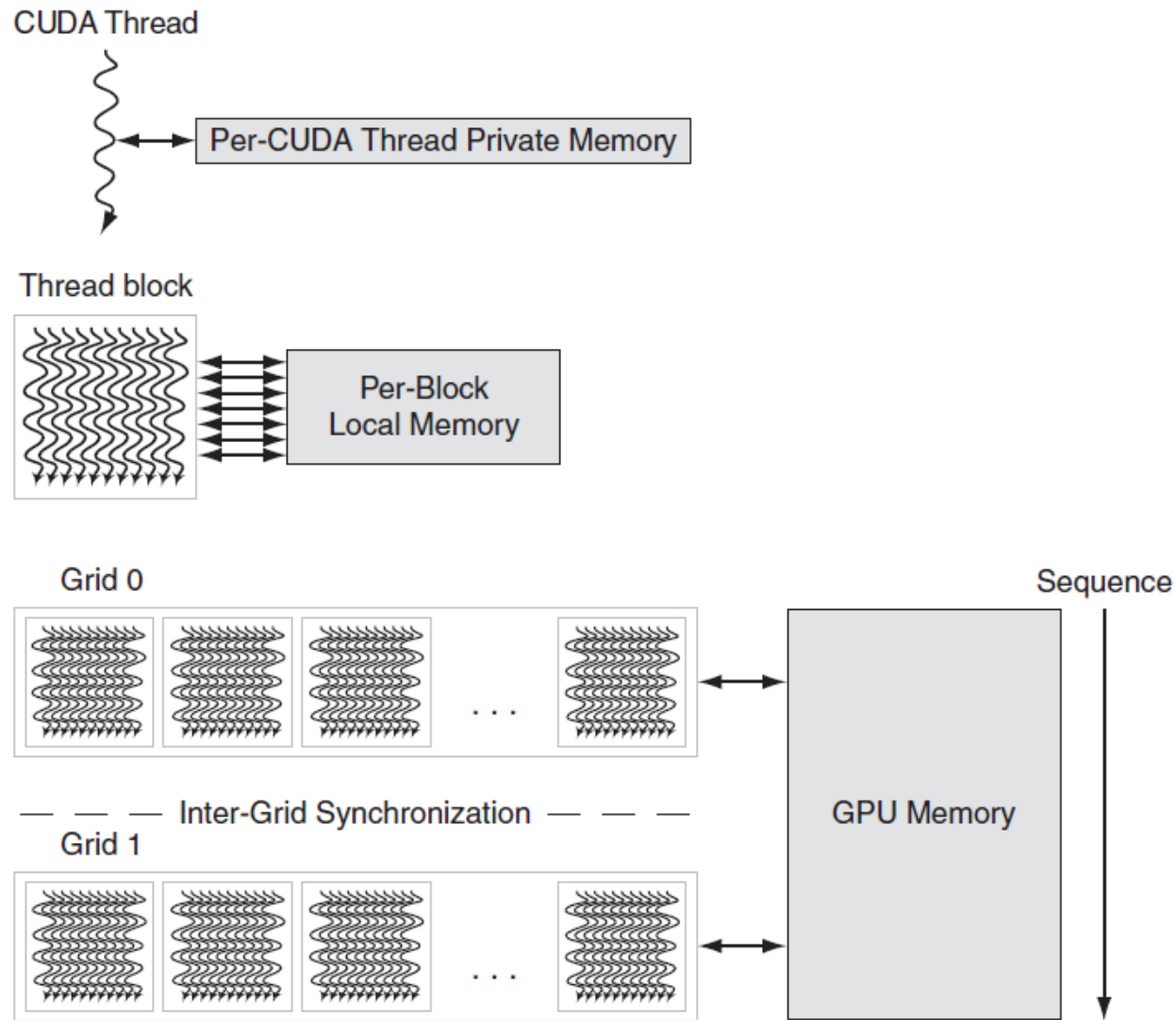
- ❑ **Processing is highly data-parallel**
 - GPUs are highly multithreaded
 - Use thread switching to hide memory latency
 - ❑ Less reliance on multi-level caches
 - Graphics memory is wide and high-bandwidth
- ❑ **Trend toward general purpose GPUs**
 - Heterogeneous CPU/GPU systems
 - CPU for sequential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

Example: NVIDIA Fermi

- ❑ SIMD Processor: 16 SIMD lanes
- ❑ SIMD instruction
 - Operates on 32 element wide threads
 - Dynamically scheduled on 16-wide processor over 2 cycles
- ❑ 32K x 32-bit registers spread across lanes
 - 64 registers per thread context



GPU Memory Structures



Putting GPUs into Perspective

Feature	Multicore with SIMD	GPU
SIMD processors	4 to 8	8 to 16
SIMD lanes/processor	2 to 4	8 to 16
Multithreading hardware support for SIMD threads	2 to 4	16 to 32
Typical ratio of single precision to double-precision performance	2:1	2:1
Largest cache size	8 MB	0.75 MB
Size of memory address	64-bit	64-bit
Size of main memory	8 GB to 256 GB	4 GB to 6 GB
Memory protection at level of page	Yes	Yes
Demand paging	Yes	No
Integrated scalar processor/SIMD processor	Yes	No
Cache coherent	Yes	No

Guide to GPU Terms

Type	More descriptive name	Closest old term outside of GPUs	Official CUDA/ NVIDIA GPU term	Book definition
Program abstractions	Vectorizable Loop	Vectorizable Loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more Thread Blocks (bodies of vectorized loop) that can execute in parallel.
	Body of Vectorized Loop	Body of a (Strip-Mined) Vectorized Loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. They can communicate via Local Memory.
	Sequence of SIMD Lane Operations	One iteration of a Scalar Loop	CUDA Thread	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask and predicate register.
Machine object	A Thread of SIMD Instructions	Thread of Vector Instructions	Warp	A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results stored depending on a per-element mask.
	SIMD Instruction	Vector Instruction	PTX Instruction	A single SIMD instruction executed across SIMD Lanes.
Processing hardware	Multithreaded SIMD Processor	(Multithreaded) Vector Processor	Streaming Multiprocessor	A multithreaded SIMD Processor executes threads of SIMD instructions, independent of other SIMD Processors.
	Thread Block Scheduler	Scalar Processor	Giga Thread Engine	Assigns multiple Thread Blocks (bodies of vectorized loop) to multithreaded SIMD Processors.
	SIMD Thread Scheduler	Thread scheduler in a Multithreaded CPU	Warp Scheduler	Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution.
	SIMD Lane	Vector lane	Thread Processor	A SIMD Lane executes the operations in a thread of SIMD instructions on a single element. Results stored depending on mask.
Memory hardware	GPU Memory	Main Memory	Global Memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU.
	Local Memory	Local Memory	Shared Memory	Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors.
	SIMD Lane Registers	Vector Lane Registers	Thread Processor Registers	Registers in a single SIMD Lane allocated across a full thread block (body of vectorized loop).

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 (Again)

- ❑ **Sum 64,000 on 64 processors**
- ❑ **First distribute 1000 numbers to each**

- **The do partial sums**

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

- ❑ **Reduction**

- **Half the processors send, other half receive and add**
 - **The quarter send, quarter receive and add, ...**

Sum Reduction (Again)

□ Given send() and receive() operations

```
limit = 64; half = 64; /* 64 processors */
do
    half = (half+1)/2; /* send vs. receive
                        dividing line */
    if (Pn >= half && Pn < limit)
        send(Pn - half, sum);
    if (Pn < (limit/2))
        sum += receive();
    limit = half; /* upper limit of senders */
while (half > 1); /* exit with final sum */
```

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

i7-960 vs. NVIDIA Tesla 280/480

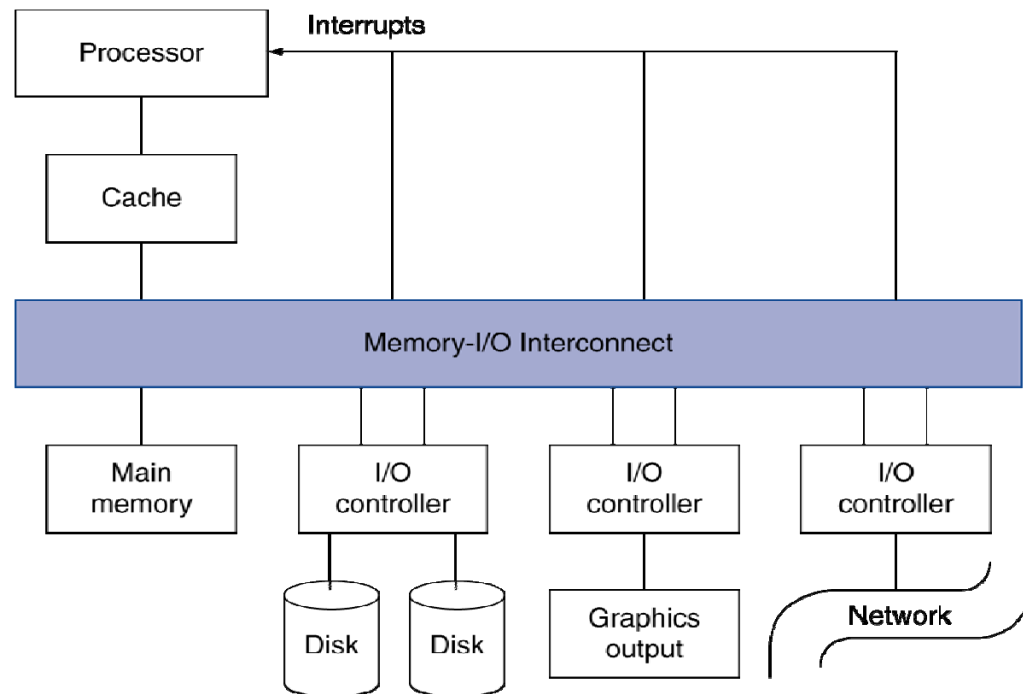
	Core i7-960	GTX 280	GTX 480	Ratio 280/i7	Ratio 480/i7
Number of processing elements (cores or SMs)	4	30	15	7.5	3.8
Clock frequency (GHz)	3.2	1.3	1.4	0.41	0.44
Die size	263	576	520	2.2	2.0
Technology	Intel 45 nm	TCMS 65 nm	TCMS 40 nm	1.6	1.0
Power (chip, not module)	130	130	167	1.0	1.3
Transistors	700 M	1400 M	3100 M	2.0	4.4
Memory bandwidth (GBytes/sec)	32	141	177	4.4	5.5
Single precision SIMD width	4	8	32	2.0	8.0
Double precision SIMD width	2	1	16	0.5	8.0
Peak Single precision scalar FLOPS (GFLOP/sec)	26	117	63	4.6	2.5
Peak Single precision s SIMD FLOPS (GFLOP/Sec)	102	311 to 933	515 to 1344	3.0-9.1	6.6-13.1
(SP 1 add or multiply)	N.A.	(311)	(515)	(3.0)	(6.6)
(SP 1 instruction fused)	N.A.	(622)	(1344)	(6.1)	(13.1)
(face SP dual issue fused)	N.A.	(933)	N.A.	(9.1)	–
Peak double precision SIMD FLOPS (GFLOP/sec)	51	78	515	1.5	10.1



Storage, Bus, and I/O

Introduction

- ❑ I/O devices can be characterized by
 - Behaviour: input, output, storage
 - Partner: human or machine
 - Data rate: bytes/sec, transfers/sec
- ❑ I/O bus connections

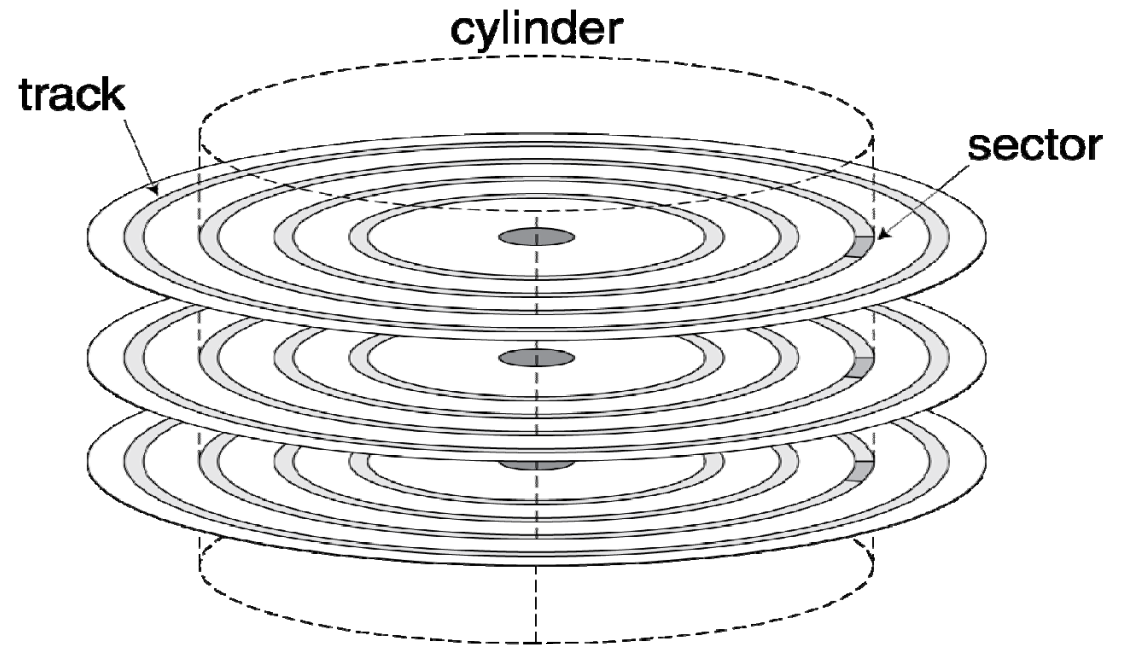


I/O System Characteristics

- ❑ **Dependability is important**
 - **Particularly for storage devices**
- ❑ **Performance measures**
 - **Latency (response time)**
 - **Throughput (bandwidth)**
 - **Desktops & embedded systems**
 - ❑ **Mainly interested in response time & diversity of devices**
 - **Servers**
 - ❑ **Mainly interested in throughput & expandability of devices**

Disk Storage

- ❑ Nonvolatile, rotating magnetic storage



Disk Sectors and Access

- ❑ **Each sector records**
 - Sector ID
 - Data (512 bytes, 4096 bytes proposed)
 - Error correcting code (ECC)
 - ❑ Used to hide defects and recording errors
 - Synchronization fields and gaps
- ❑ **Access to a sector involves**
 - Queuing delay if other accesses are pending
 - Seek: move the heads
 - Rotational latency
 - Data transfer
 - Controller overhead

Flash Storage

- ❑ **Nonvolatile semiconductor storage**
 - 100× – 1000× faster than disk
 - Smaller, lower power, more robust
 - But more \$/GB (between disk and DRAM)



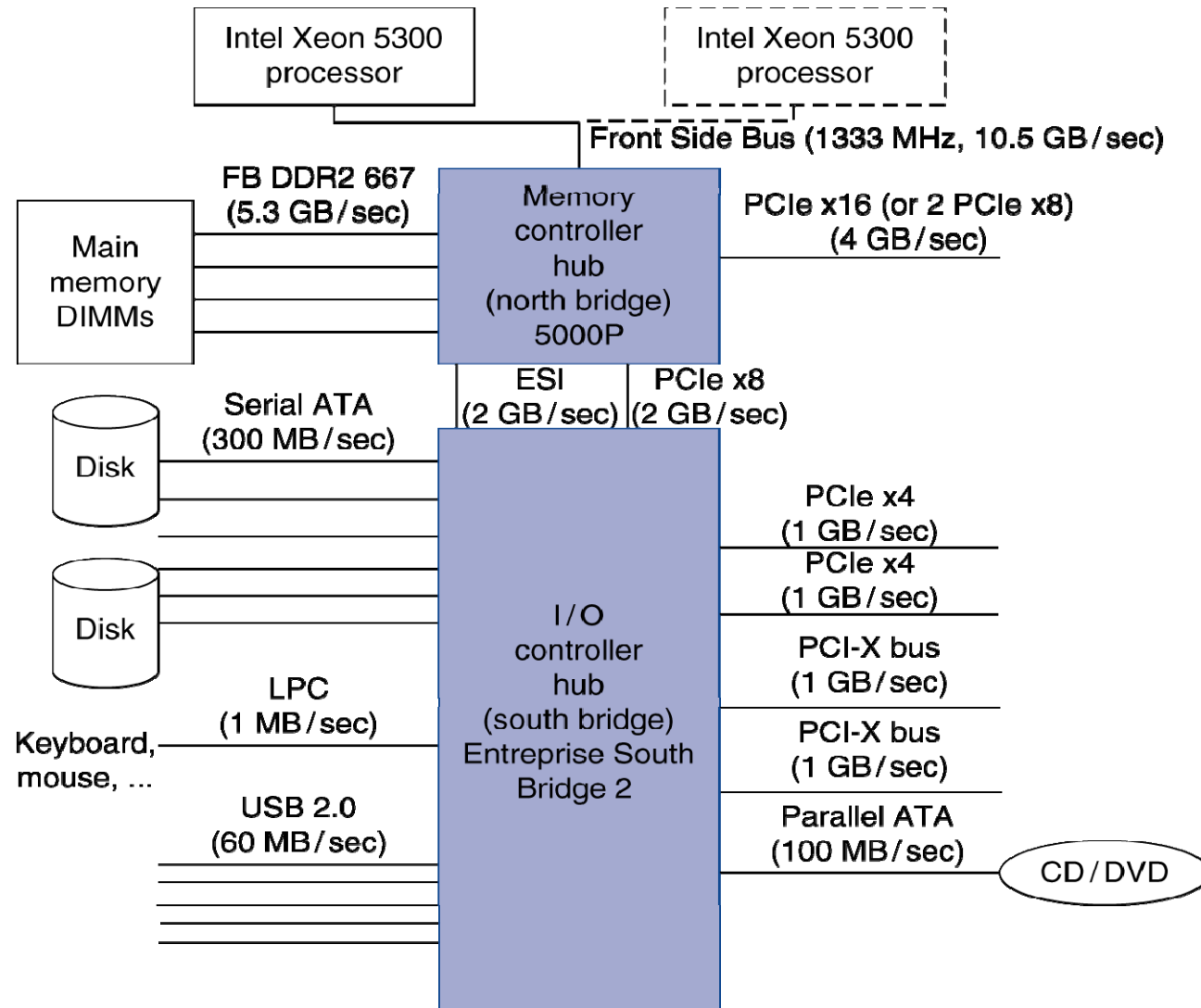
Flash Types

- ❑ **NOR flash: bit cell like a NOR gate**
 - Random read/write access
 - Used for instruction memory in embedded systems
- ❑ **NAND flash: bit cell like a NAND gate**
 - Denser (bits/area), but block-at-a-time access
 - Cheaper per GB
 - Used for USB keys, media storage, ...
- ❑ **Flash bits wears out after 1000's of accesses**
 - Not suitable for direct RAM or disk replacement
 - Wear leveling: remap data to less used blocks

Interconnecting Components – bus etc

- ❑ **Need interconnections between**
 - CPU, memory, I/O controllers
- ❑ **Bus: shared communication channel**
 - Parallel set of wires for data and synchronization of data transfer
 - Can become a bottleneck
- ❑ **Performance limited by physical factors**
 - Wire length, number of connections
- ❑ **More recent alternative: high-speed serial connections with switches**
 - Like networks

Typical x86 PC I/O System



Buses: connecting I/O to CPU and memory

- ❑ Shared communication link (one or more wires)
- ❑ Difficult design:
 - may be bottleneck
 - length of the bus
 - number of devices
 - tradeoffs (buffers for higher bandwidth increases latency)
 - support for many different devices cost
- ❑ Synchronous vs. Asynchronous
 - Synchronous: use a clock and a synchronous protocol, fast and small but every device must operate at same rate and clock skew requires the bus to be short
 - Asynchronous: don't use a clock and instead use a handshaking protocol

I/O Bus Examples

	Firewire	USB 2.0	PCI Express	Serial ATA	Serial Attached SCSI
Intended use	External	External	Internal	Internal	External
Devices per channel	63	127	1	1	4
Data width	4	2	2/lane	4	4
Peak bandwidth	50MB/s or 100MB/s	0.2MB/s, 1.5MB/s, or 60MB/s	250MB/s/lane 1×, 2×, 4×, 8×, 16×, 32×	300MB/s	300MB/s
Hot pluggable	Yes	Yes	Depends	Yes	Yes
Max length	4.5m	5m	0.5m	1m	8m
Standard	IEEE 1394	USB Implementers Forum	PCI-SIG	SATA-IO	INCITS TC T10

Other important bus issues

❑ Bus Arbitration:

- daisy chain arbitration (not very fair)
- centralized arbitration (requires an arbiter), e.g., PCI
- self selection, e.g., NuBus used in Macintosh
- collision detection, e.g., Ethernet

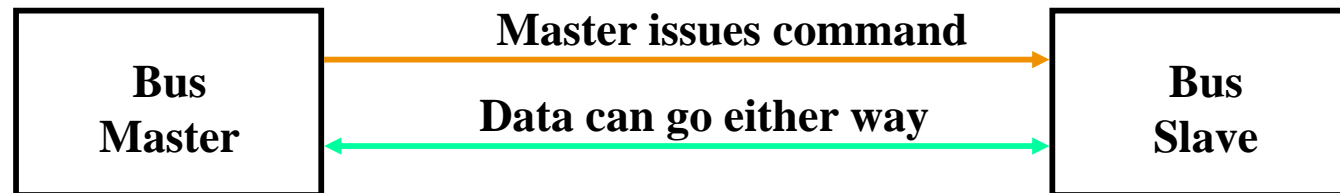
❑ Operating system:

- Polling
- Interrupts
- DMA

❑ Performance Analysis techniques:

- queuing theory
- Simulation
- analysis, i.e., find the weakest link (see “I/O System Design”)

Master versus Slave



- ❑ A bus transaction includes two parts:
 - Issuing the command (and address) – request
 - Transferring the data – action
- ❑ Master is the one who starts the bus transaction by:
 - issuing the command (and address)
- ❑ Slave is the one who responds to the address by:
 - Sending data to the master if the master ask for data
 - Receiving data from the master if the master wants to send data

Giving Commands to I/O Devices

- ❑ **Two methods are used to address the device:**
 - Special I/O instructions
 - Memory-mapped I/O
- ❑ **Special I/O instructions specify:**
 - Both the device number and the command word
 - ❑ Device number: the processor communicates this via a set of wires normally included as part of the I/O bus
 - ❑ Command word: this is usually send on the bus's data lines
- ❑ **Memory-mapped I/O:**
 - Portions of the address space are assigned to I/O device
 - Read and writes to those addresses are interpreted as commands to the I/O devices
 - User programs are prevented from issuing I/O operations directly:
 - ❑ The I/O address space is protected by the address translation

I/O Register Mapping

❑ Memory mapped I/O

- Registers are addressed in same space as memory
- Address decoder distinguishes between them
- OS uses address translation mechanism to make them only accessible to kernel

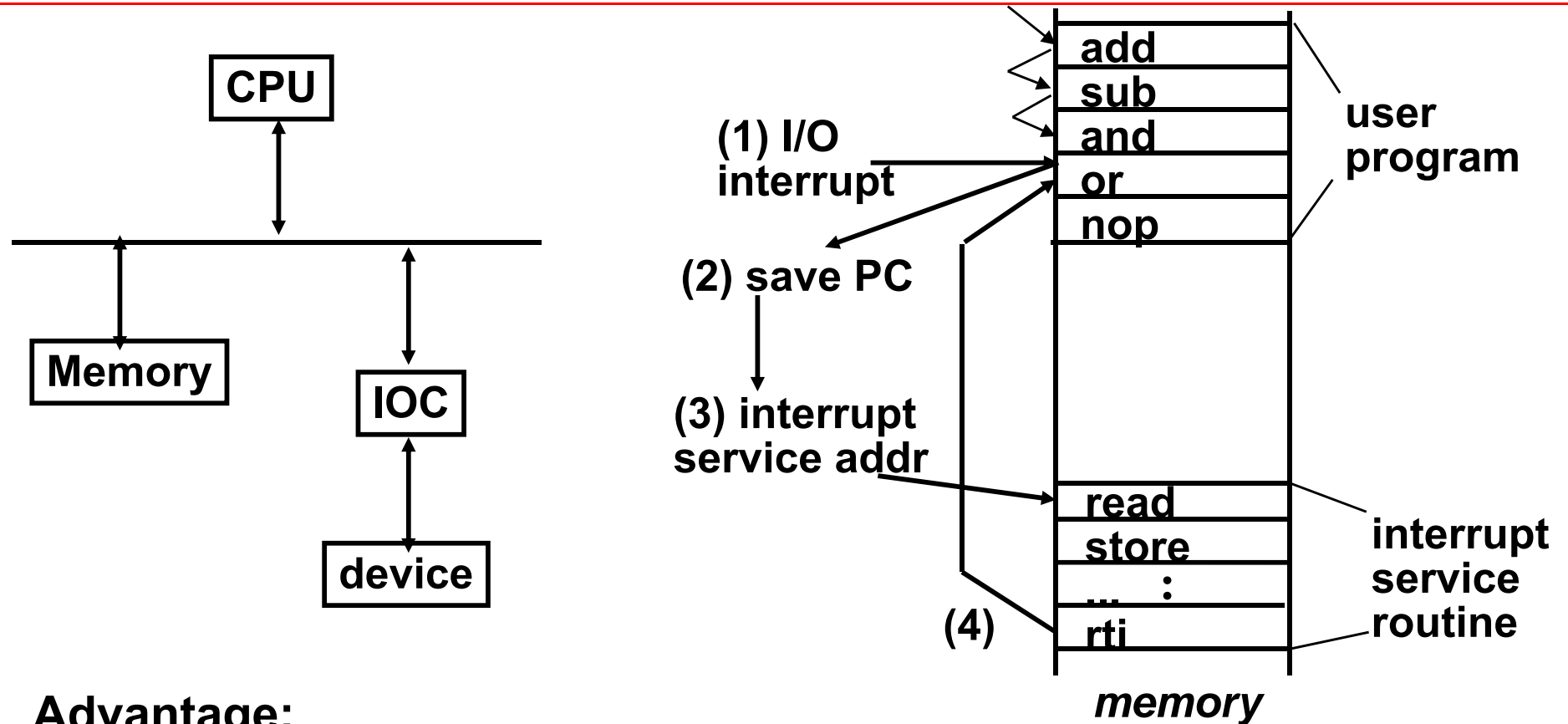
❑ I/O instructions

- Separate instructions to access I/O registers
- Can only be executed in kernel mode
- Example: x86

I/O Device Notifying the CPU

- ❑ **The CPU needs to know when:**
 - The I/O device has completed an operation
 - The I/O operation has encountered an error
- ❑ **This can be accomplished in two different ways:**
 - **Polling:**
 - ❑ The I/O device put information in a status register
 - ❑ The CPU periodically check the status register
 - **I/O Interrupt:**
 - ❑ Whenever an I/O device needs attention from the processor, it interrupts the processor from what it is currently doing.

Interrupt Driven Data Transfer



❑ Advantage:

- User program progress is only halted during actual transfer

❑ Disadvantage, special hardware is needed to:

- Cause an interrupt (I/O device)
- Detect an interrupt (processor)
- Save the proper states to resume after the interrupt (processor)

I/O Interrupt

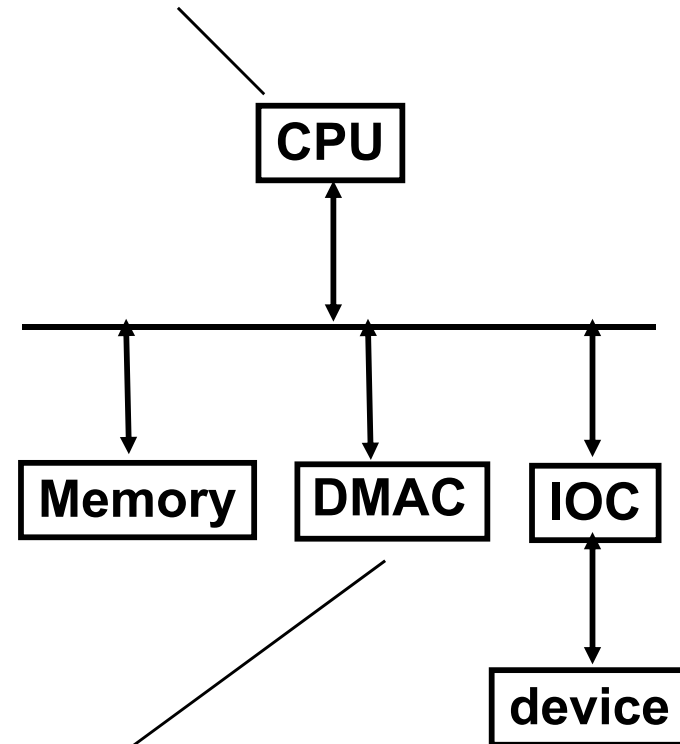
- ❑ **An I/O interrupt is just like the exceptions except:**
 - An I/O interrupt is asynchronous
 - Further information needs to be conveyed
- ❑ **An I/O interrupt is asynchronous with respect to instruction execution:**
 - I/O interrupt is not associated with any instruction
 - I/O interrupt does not prevent any instruction from completion
 - ❑ You can pick your own convenient point to take an interrupt
- ❑ **I/O interrupt is more complicated than exception:**
 - Needs to convey the identity of the device generating the interrupt
 - Interrupt requests can have different urgencies:
 - ❑ Interrupt request needs to be prioritized

DMA :

Delegating I/O Responsibility from the CPU

- ❑ **Direct Memory Access (DMA):**
 - External to the CPU
 - Act as a maser on the bus
 - Transfer blocks of data to or from memory without CPU intervention

CPU sends a starting address, direction, and length count to DMAC. Then issues "start".



DMAC provides handshake signals for Peripheral Controller, and Memory Addresses and handshake signals for Memory.

DMA/Cache Interaction

- ❑ If DMA writes to a memory block that is cached
 - Cached copy becomes stale
- ❑ If write-back cache has dirty block, and DMA reads memory block
 - Reads stale data
- ❑ Need to ensure cache coherence
 - Flush blocks from cache if they will be used for DMA
 - Or use non-cacheable memory locations for I/O

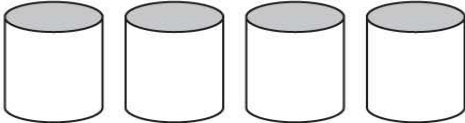
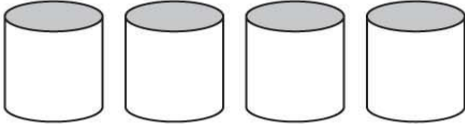
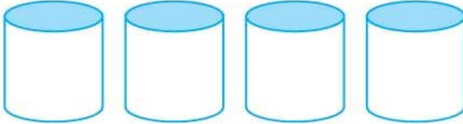
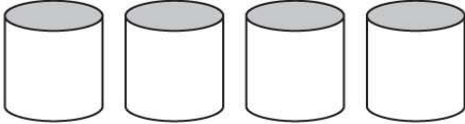
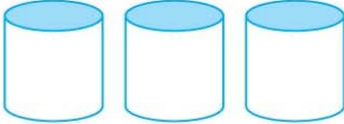
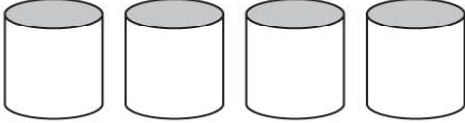

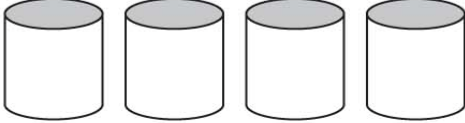

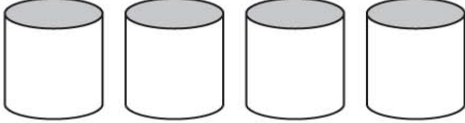

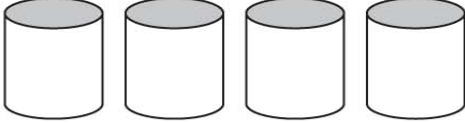
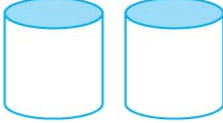
DMA/VM Interaction

- ❑ **OS uses virtual addresses for memory**
 - DMA blocks may not be contiguous in physical memory
- ❑ **Should DMA use virtual addresses?**
 - Would require controller to do translation
- ❑ **If DMA uses physical addresses**
 - May need to break transfers into page-sized chunks
 - Or chain multiple transfers
 - Or allocate contiguous physical pages for DMA

RAID

- ❑ **Redundant Array of Inexpensive (Independent) Disks**
 - Use multiple smaller disks (c.f. one large disk)
 - Parallelism improves performance
 - Plus extra disk(s) for redundant data storage
- ❑ **Provides fault tolerant storage system**
 - Especially if failed disks can be “hot swapped”
- ❑ **RAID 0**
 - No redundancy (“AID”?)
 - ❑ Just stripe data over multiple disks
 - But it does improve performance

RAID Summary

	Data disks	Redundant check disks
RAID 0 (No redundancy) Widely used		
RAID 1 (Mirroring) EMC, HP(Tandem), IBM		
RAID 2 (Error detection and correction code) Unused		
RAID 3 (Bit-interleaved parity) Storage concepts		
RAID 4 (Block-interleaving parity) Network appliance		
RAID 5 (Distributed block-interleaved parity) Widely used		
RAID 6 (P + Q redundancy) Recently popular		

RAID 1 & 2

❑ RAID 1: Mirroring

- **N + N disks, replicate data**
 - ❑ Write data to both data disk and mirror disk
 - ❑ On disk failure, read from mirror

❑ RAID 2: Error correcting code (ECC)

- **N + E disks (e.g., 10 + 4)**
- **Split data at bit level across N disks**
- **Generate E-bit ECC**
- **Too complex, not used in practice**

RAID 3: Bit-Interleaved Parity

❑ N + 1 disks

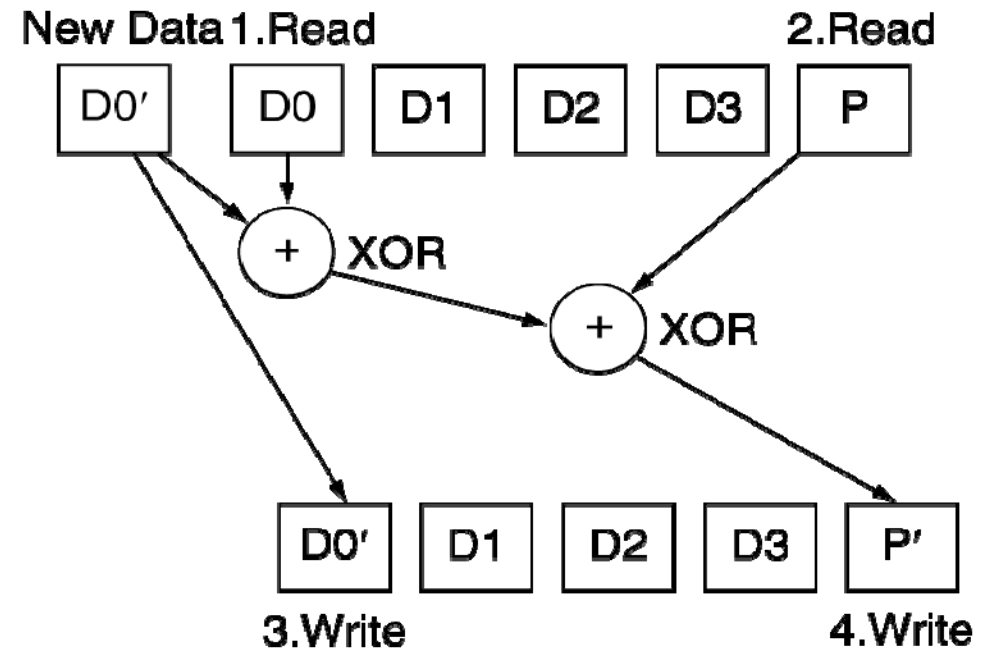
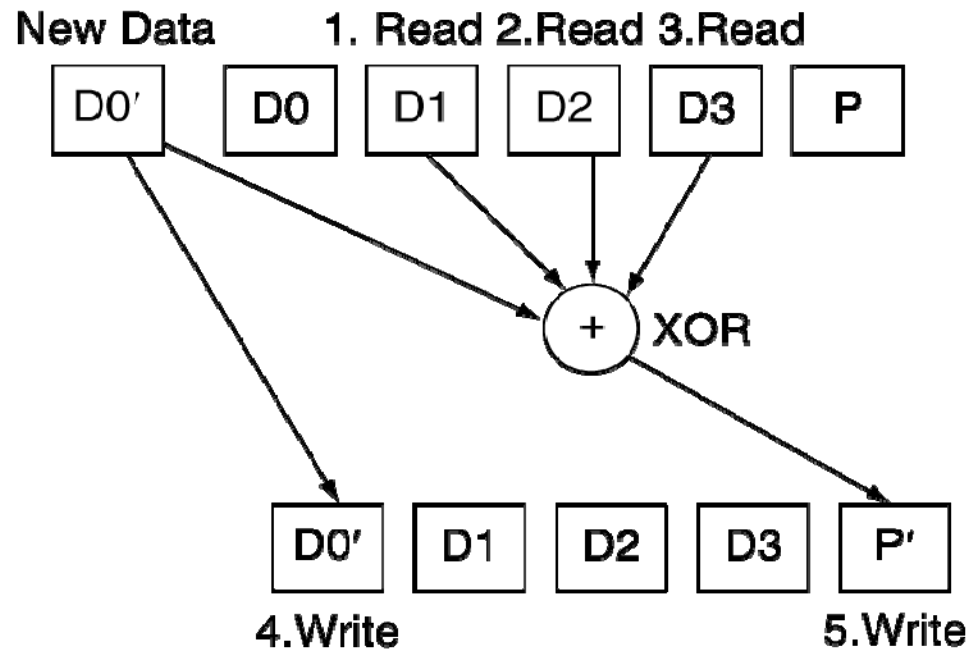
- Data striped across N disks at byte level
- Redundant disk stores parity
- Read access
 - ❑ Read all disks
- Write access
 - ❑ Generate new parity and update all disks
- On failure
 - ❑ Use parity to reconstruct missing data

❑ Not widely used

RAID 4: Block-Interleaved Parity

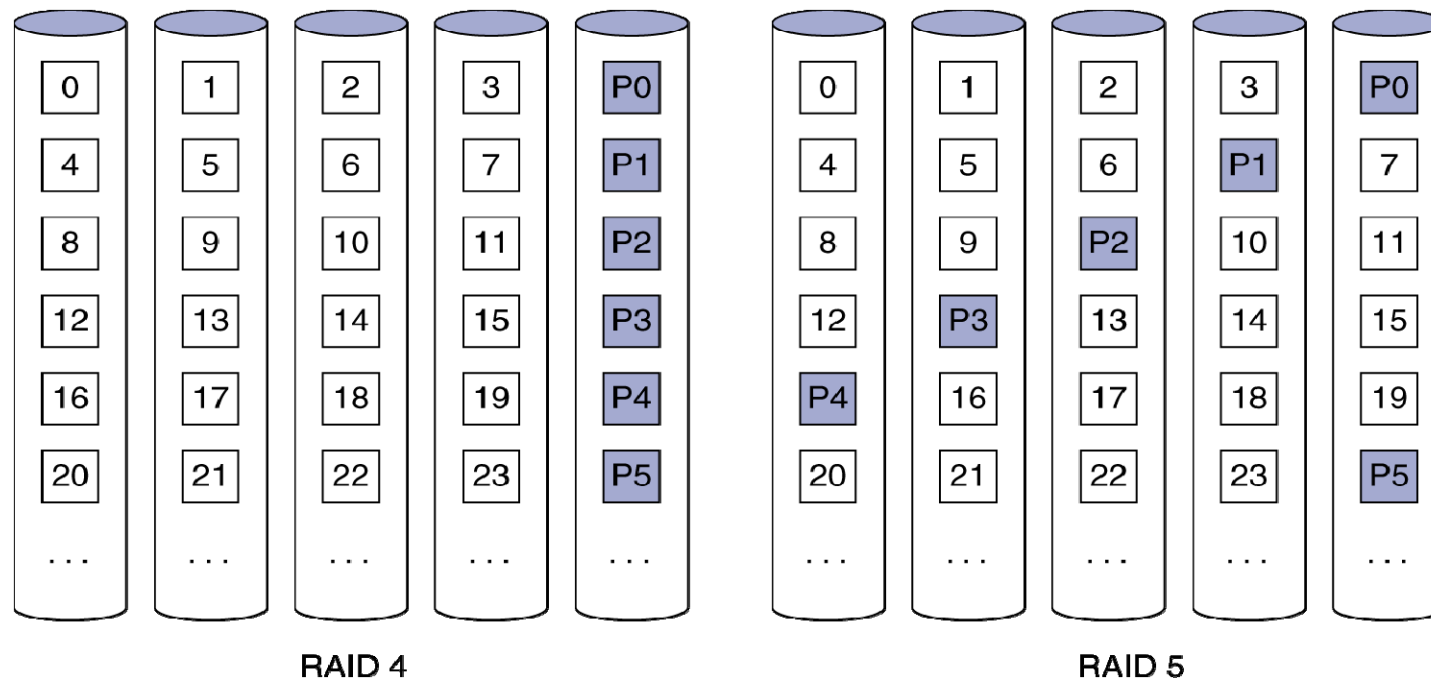
- ❑ **N + 1 disks**
 - Data striped across N disks at block level
 - Redundant disk stores parity for a group of blocks
 - Read access
 - ❑ Read only the disk holding the required block
 - Write access
 - ❑ Just read disk containing modified block, and parity disk
 - ❑ Calculate new parity, update data disk and parity disk
 - On failure
 - ❑ Use parity to reconstruct missing data
- ❑ **Not widely used**

RAID 3 vs RAID 4



RAID 5: Distributed Parity

- ❑ **N + 1 disks**
 - Like RAID 4, but parity blocks distributed across disks
 - ❑ Avoids parity disk being a bottleneck
- ❑ **Widely used**



RAID 6: P + Q Redundancy

❑ N + 2 disks

- Like RAID 5, but two lots of parity
- Greater fault tolerance through more redundancy

❑ Multiple RAID

- More advanced systems give similar fault tolerance with better performance

RAID Summary

- ❑ **RAID can improve performance and availability**
 - High availability requires hot swapping
- ❑ **Assumes independent disk failures**
 - Too bad if the building burns down!
- ❑ **See “Hard Disk Performance, Quality and Reliability”**
 - <http://www.pcguide.com/ref/hdd/perf/index.htm>