



中山大學
SUN YAT-SEN UNIVERSITY



国家超级计算广州中心
NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

Advanced Computer Architecture

高级计算机体系结构

第9讲：Thread-Level Parallelism (1)

张献伟

xianweiz.github.io

DCS5367, 11/30/2021

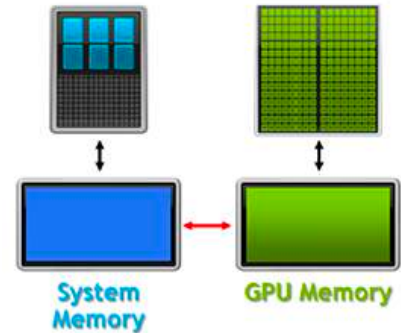
Quiz Questions

- Q1: differences between SIMD and SIMT?
SIMT \approx SIMD + multithreading, much easier programming
- Q2: explain host and device in GPU programming.
host = CPU, device = GPU. Host offloads kernels onto devices.
- Q3: why do we need streams in GPU programming?
To concurrently run kernels, sharing the GPU.
- Q4: explain grid and block in GPU.
Grid: all threads to run the kernel; Grid is further divided into blocks.
- Q5: keyword ‘__shared__’ declares which memory?
Shared memory, private to each SM and shared within a block.
- Q6: when do we use local memory?
Register spilling or array inside a kernel. But ‘local’ is in device mem.

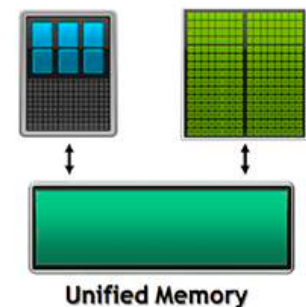
Unified Memory[统一内存]

- Classical model[经典模型]
 - Allocate memory on host
 - Allocate memory on device
 - Copy data from host to device Operate on the GPU data
 - Copy data back to host
- Unified memory model[统一模型]
 - Allocate memory
 - Operate on data on GPU
- Unified Memory is a single memory address space accessible from any processor in a system
 - `cudaMalloc()` → `cudaMallocManaged()`
 - on-demand page migration

Traditional Developer View



Developer View With Unified Memory



Example

```
int N = 1<<20;
float *x, *y;

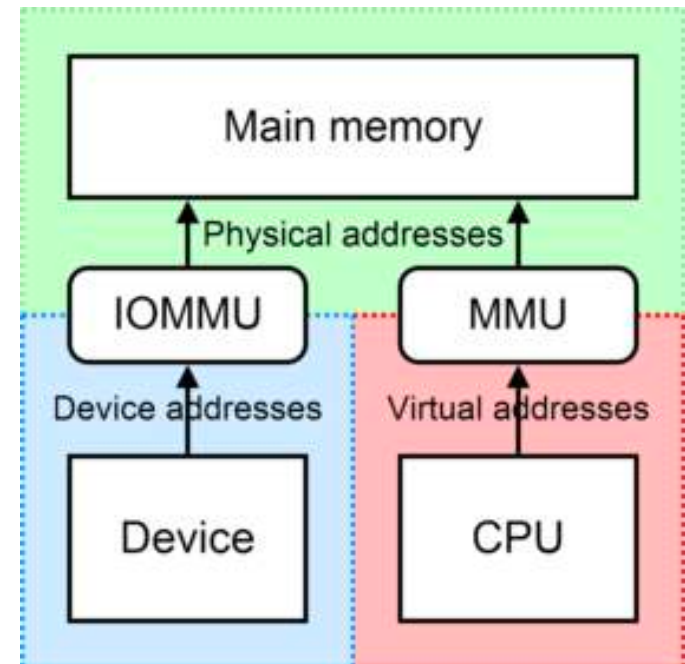
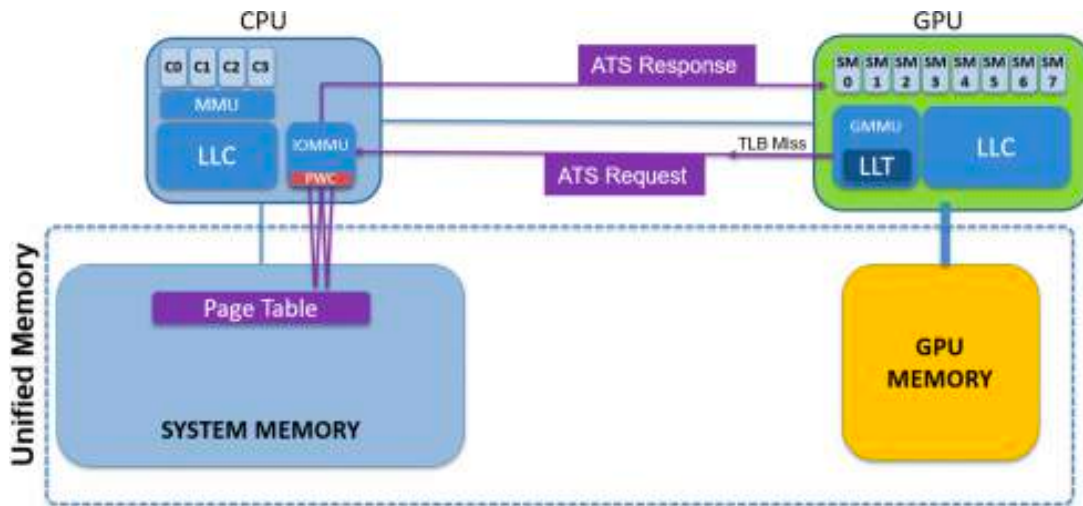
// Allocate Unified Memory -- accessible from CPU or GPU
cudaMallocManaged(&x, N*sizeof(float));
cudaMallocManaged(&y, N*sizeof(float));

// initialize x and y arrays on the host
for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
}

// Launch kernel on 1M elements on the GPU
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;
add<<<numBlocks, blockSize>>>(N, x, y);
```

Address Translation[地址转换]

- GMMU: GPU memory management unit
 - Last level TLB (LLT)
- IOMMU: maps device-visible virtual addresses to physical addresses
 - Page walk caches (PWC)

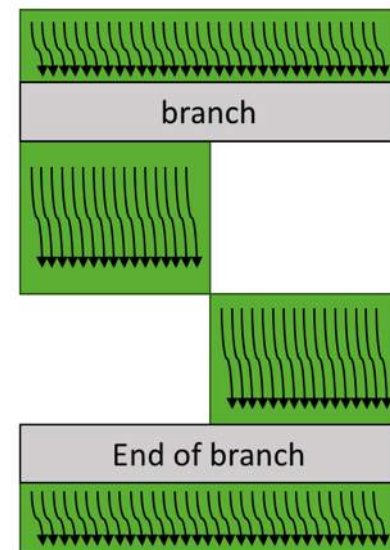


Divergence[分支]

- Within a block of threads, the threads are executed in groups of 32 called a warp
 - All threads in a warp do the same thing at the same time
- What happens if different threads in a warp need to do different things?
 - A logical predicate and two predicated instructions \rightarrow serialized
- Branch divergence is a major cause for performance degradation in GPGPUs

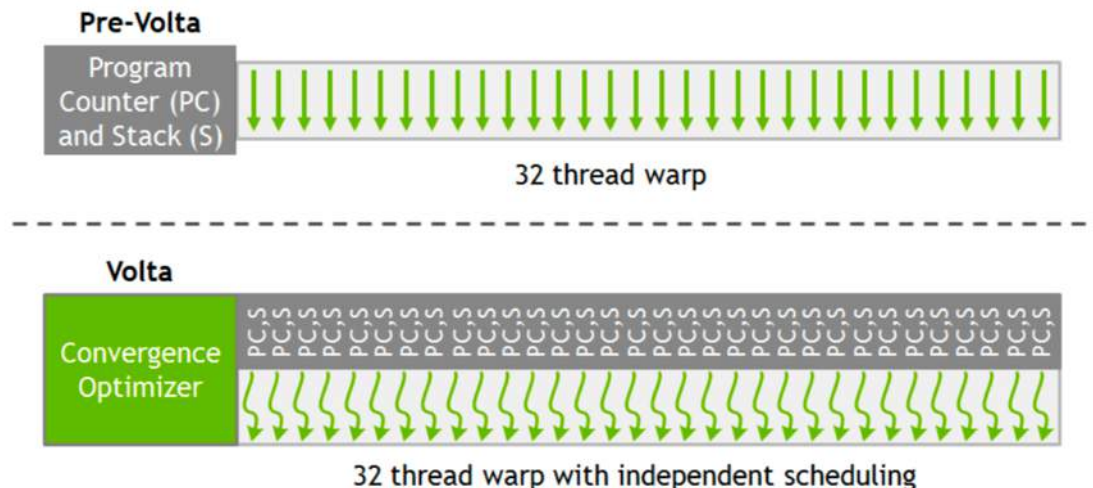
```
...  
if ( threadIdx.x < 16 )  
{  
    ... A ...  
}  
else  
{  
    ... B ...  
}  
...
```

$p = (\text{threadIdx.x} < 16);$
if (p) ... A ...
if (!p) ... B ...



Divergence (cont.)

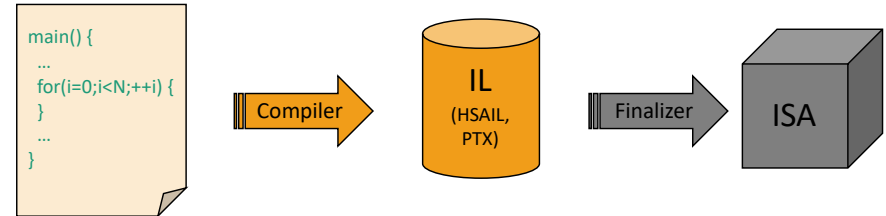
- Pre-Volta GPUs use a **single PC** shared amongst all 32 threads of a warp, combined with an **active mask** that specifies which threads of the warp are active at any given time
 - Leaves threads that are not executing a branch inactive
- Since Volta, each thread features its **own PC**, which allows threads of the same warp to execute different branches of a divergent section simultaneously



Two-phase Execution[两段式]

- Compilation workflow

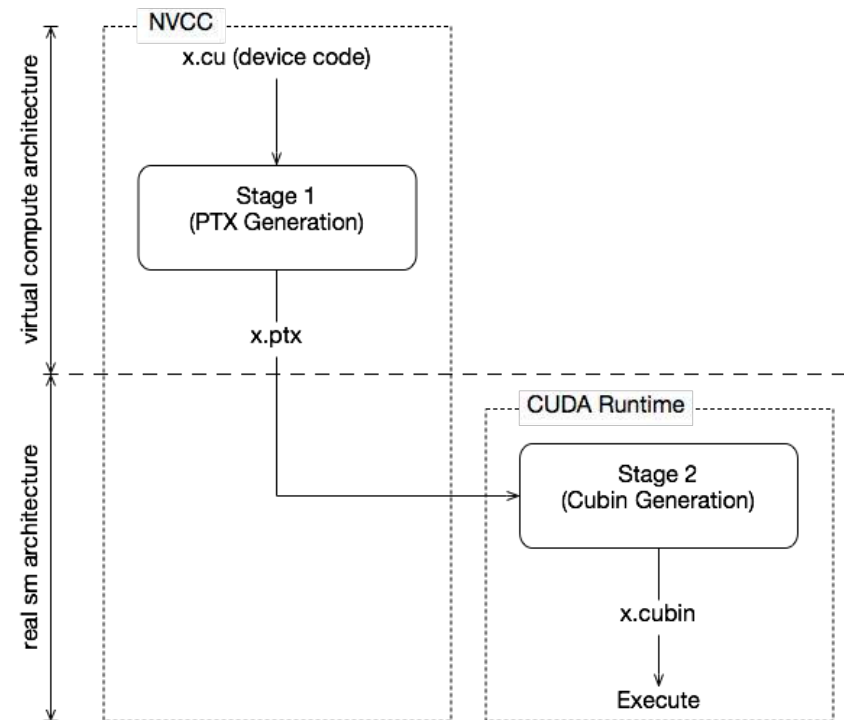
- Source code → virtual instruction (PTX or HSAIL)
- Virtual inst → real inst (SASS or GCN)



- **.cu**: CUDA source file, containing host code and device functions

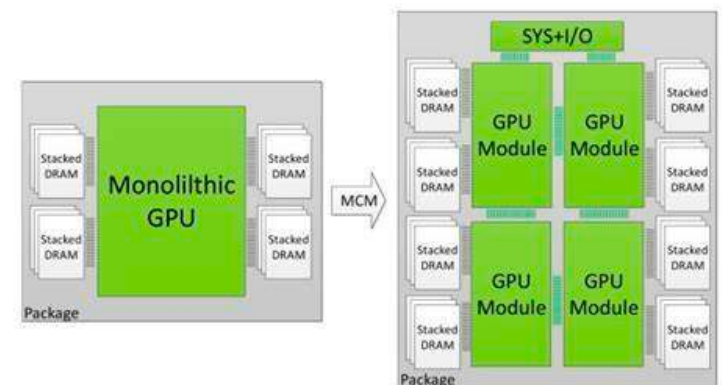
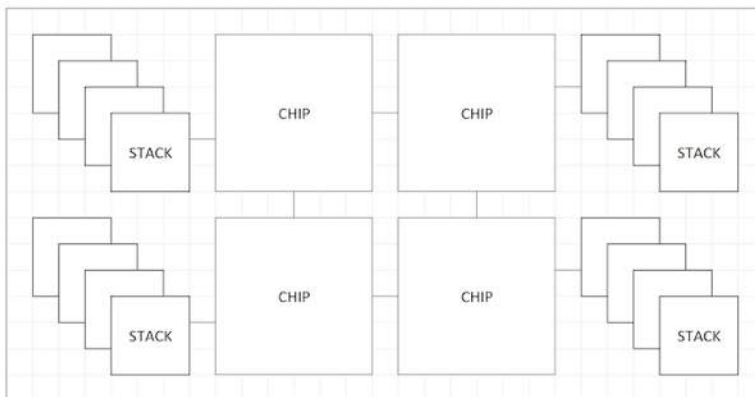
- **.ptx**: PTX intermediate assembly file

- **.cubin**: CUDA device code binary file (CUBIN) for a single GPU architecture



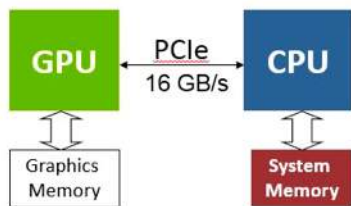
Multi-chip Module

- Aggregating multiple GPU modules within a single package, as opposed to a single monolithic die.
- AMD: Chiplet GPUs
 - MI200: 220 compute units, 14K streaming cores
 - MI100: 120 compute units, 7680 streaming cores
- Nvidia: Multi-Chip-Module (MCM) GPUs
 - Hopper (Ampere -> Lovelace): 300+ SMs, 40K+ CUDA cores
 - A100: 128 SMs, 8192 CUDA cores

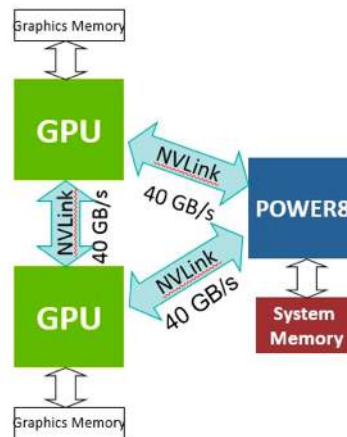


High-speed Links[高速连接]

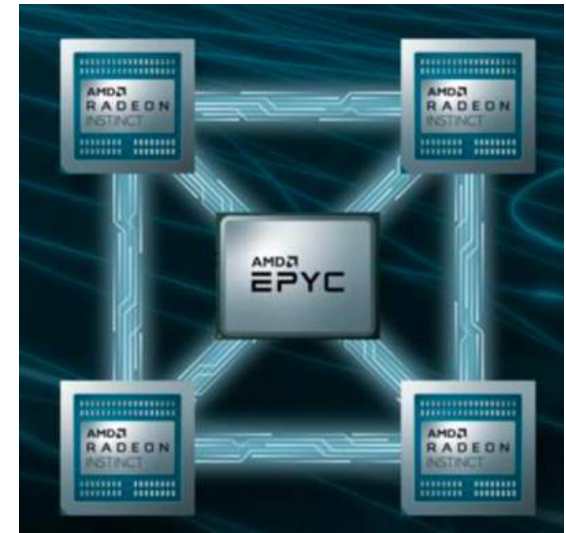
- GPUs are of high compute capability, being bottlenecked on data movement
- High-speed interconnect to achieve significantly higher data movement
 - Nvidia: NVLink
 - AMD: Infinity Fabric
 - Intel: Compute eXpress Link (CXL)



CPU-GPU Systems Connected via PCI-e

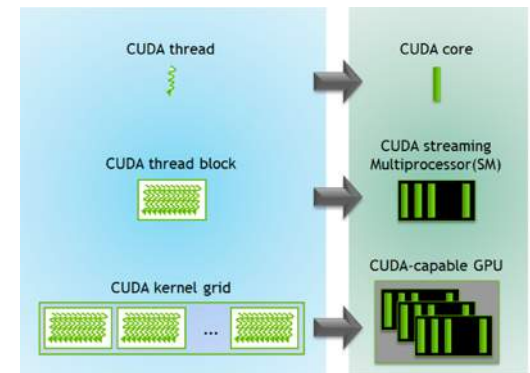


NVLink Enables Fast Unified Memory Access between CPU & GPU Memories



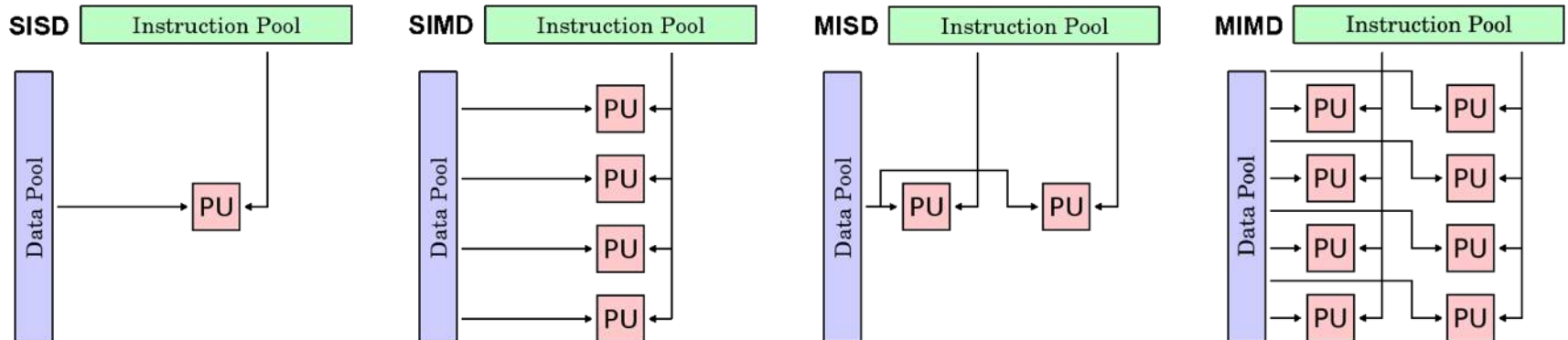
Summary of DLP/GPU[总结]

- Data level parallelism
 - SIMD: operates on multiple data with on single instruction
 - AVX-512 on Intel CPU is the typical example
 - SIMT: consists of multiple scalar threads executing in a SIMD manner
 - GPU is the example with threads executing the same instruction
- GPU hardware and thread organization
 - Device → SM → SIMD/Partition → Core
 - Grid → Block → Warp → Thread
- GPU programming
 - Streams to support concurrency
 - Memory hierarchy and usage (thread, cache/smem, global)
 - Advanced topics: virtualization, profiling/tuning, divergence, etc



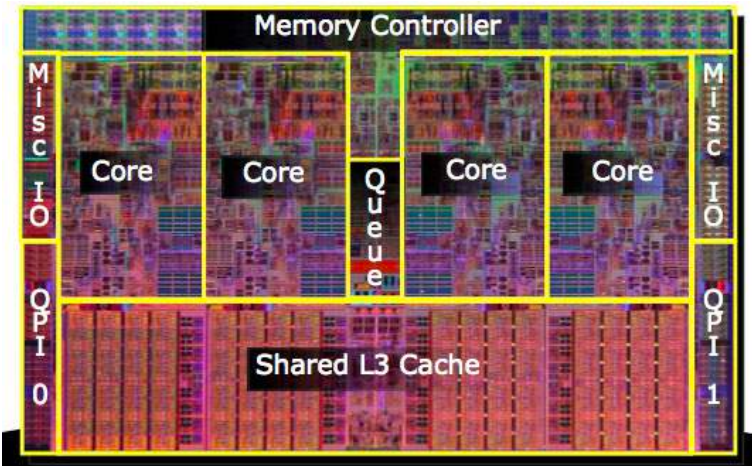
Flynn's Taxonomy[分类]

- SISD: single instruction, single data
 - A serial (non-parallel) computer
- **SIMD**: single instruction, multiple data
 - Best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing
- MISD: multiple instruction, single data
 - Few (if any) actual examples of this class have ever existed
- **MIMD**: multiple instruction, multiple data
 - Examples: supercomputers, multi-core PCs, VLIW

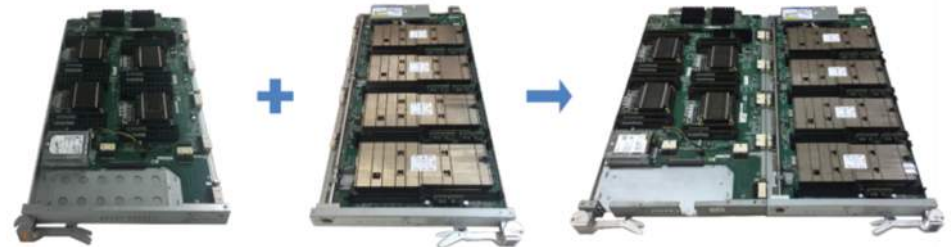


MIMD[多指令多数据]

- Machines using MIMD have **a number of processors** that function asynchronously and independently
- Each processor fetches its own instructions and operates on its own data
- At any time, different processors may be executing different instructions on different pieces of data



4 Intel Xeon CPUs + 4 FT Matrix-2000 = 2 Compute Nodes



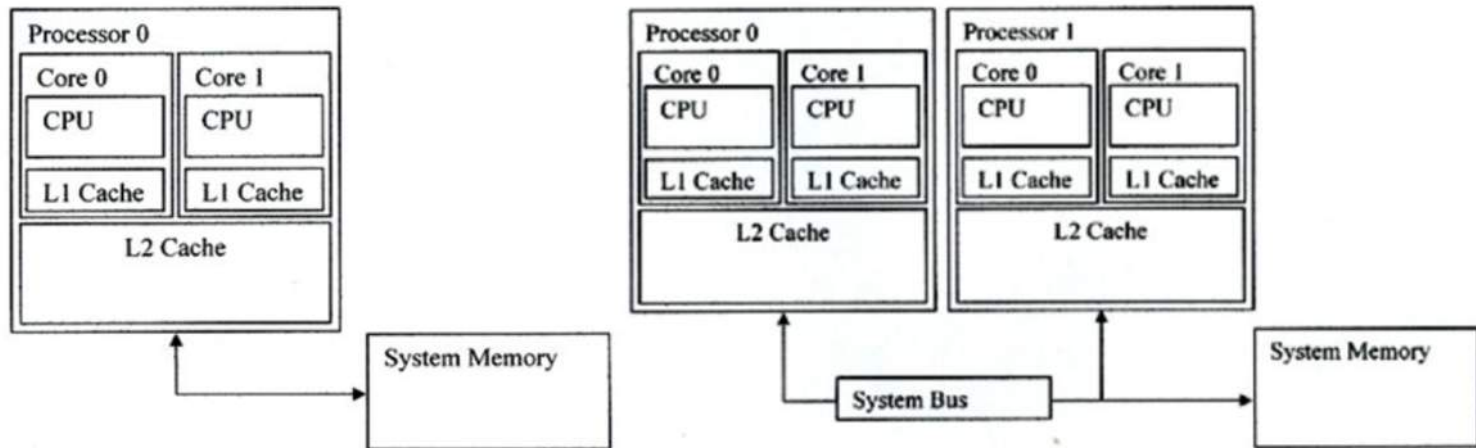
Multiprocessor[多处理器]

- **Multi-processor**

- **Multiple CPUs** tightly coupled to cooperate on a problem
- Each CPU may be a multicore design

- **Multicore processor**

- Multiprocessor where the CPU cores coexist on a single processor chip (i.e., **single CPU** w/ multi cores)



Multi-Core Processor with Shared L2 Cache

Multi-Processor System with Cores that share L2 Cache

Why Multiprocessor?[使用的几个原因]

- Not that long ago, multiprocessors were expensive, exotic machines
- Reason **#1**: running out of ILP that we can exploit[ILP有限]
 - Can't get much better performance out of a single core that's running a single program at a time
- Reason **#2**: power/thermal constraints[能耗/散热限制]
 - Even if we wanted to just build fancier single cores at higher clock speeds, we'd run into power and thermal obstacles
- Reason **#3**: Moore's Law[摩尔定律]
 - Lots of transistors → what else are we going to do with them?
 - Historically: use transistors to make more complicated cores with bigger and bigger caches
 - But we just saw that this strategy has run into problems

How to Keep Multiprocessor Busy?

- Single core processors exploit ILP
 - Multiprocessors exploit TLP: **thread-level parallelism**
- What's a thread?
 - A program can have one or more threads of control
 - Each thread has its own PC and own arch registers
 - All threads in a given program share resources (e.g., memory)
- OK, so where do we find more than one thread?
 - Option #1: Multiprogrammed workloads
 - Run multiple single-threaded programs at same time
 - Option #2: Explicitly multithreaded programs
 - Create a single program that has multiple threads that work together to solve a problem

Thread-Level Parallelism[线程级并行]

- Thread-Level parallelism[并行]
 - Have multiple program counters
 - Uses **MIMD** model
 - Targeted for tightly-coupled shared-memory multiprocessors
- Why TLP?[原因]
 - Hard to further increase core performance (e.g., clock speed)
 - Hard to find and exploit more ILP
- Implementation[实现]
 - **Multiprocessor**[多处理器]
 - Multicore processor[多核处理器]
 - Multi-processor[多个处理器]
 - **Multithreaded processor**[多线程处理器]

Multithreading[多线程]

- **Basic idea:** processor resources are expensive and should not be left idle
- On **uniprocessor**, multithreading occurs by *time-division multiplexing*[时分复用]
 - Processor switches between different threads
 - *Context switching* happens frequently enough user perceives threads as running at the same time
- **Multithreaded processor:** single CPU core that can execute multiple threads simultaneously
 - Switching
 - Simultaneous multithreading (SMT) → “hyperthreading” (Intel)

Classifying Multiprocessors[分类]

- Interconnection networks[互连网络]

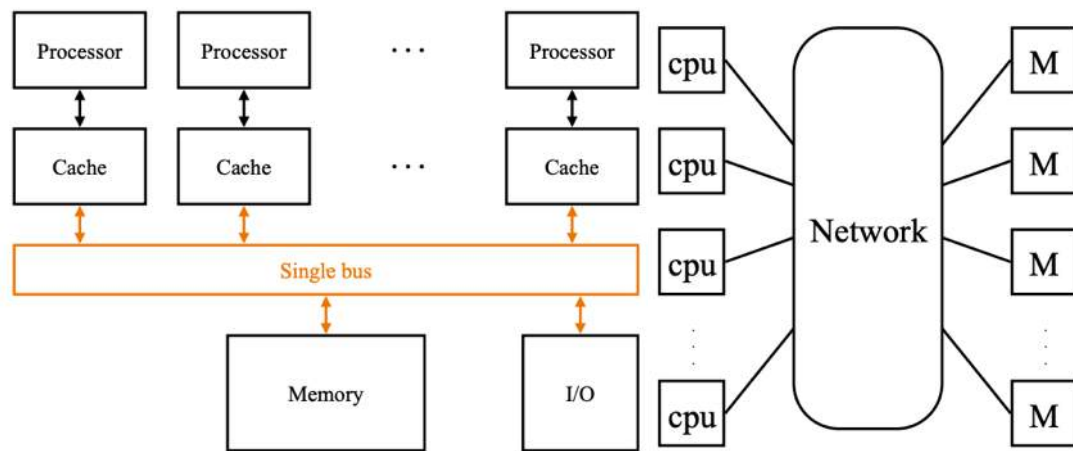
- Bus
- Network

- Memory topology[内存]

- UMA
- NUMA

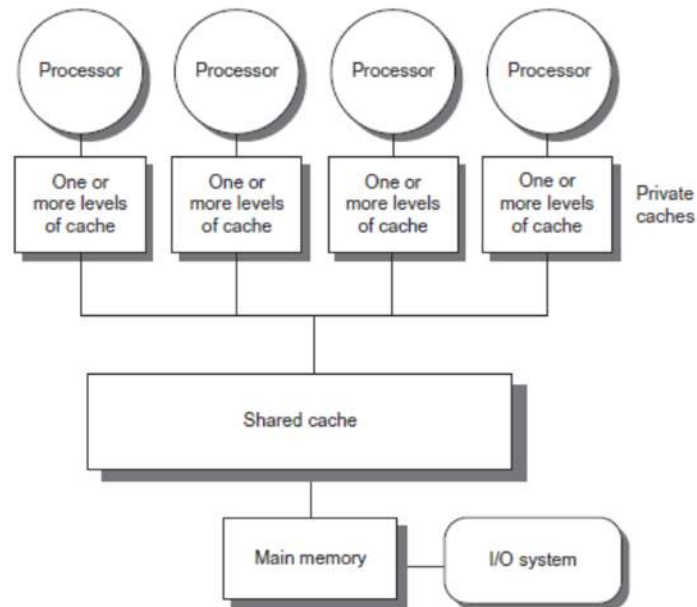
- Programming model[编程模型]

- Shared memory[共享内存]: every processor can name every address location
- Message passing[消息传递]: each processor can name only its local memory. Communication is through explicit messages



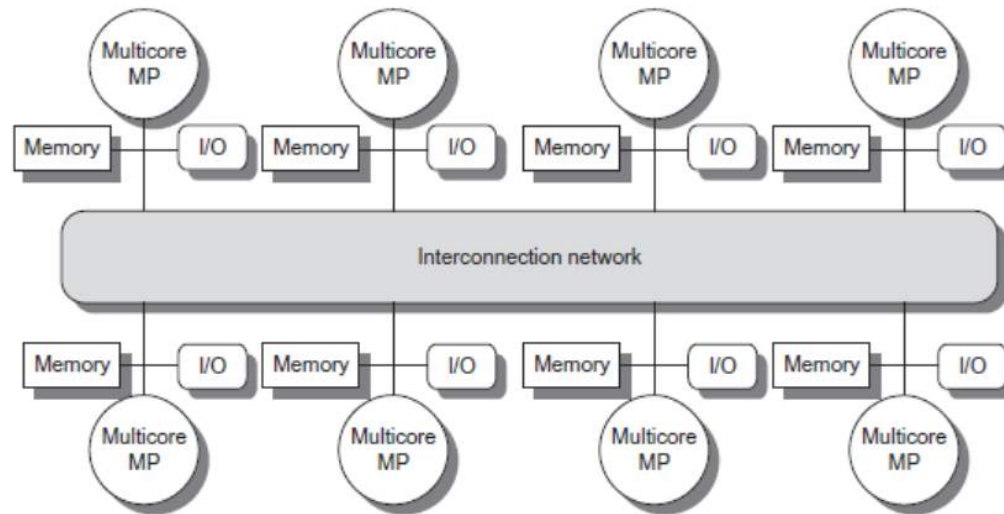
SMP[对称型]

- Symmetric (shared-memory) multiprocessors (SMPs)
 - A.k.a., centralized shared-memory multiprocessors
 - A.k.a., uniform memory access (UMA) multiprocessors
 - Small number of cores (typically ≤ 8)
 - Share a single centralized memory that all processors have equal access to, hence “**symmetric**”
 - Uniform access latency



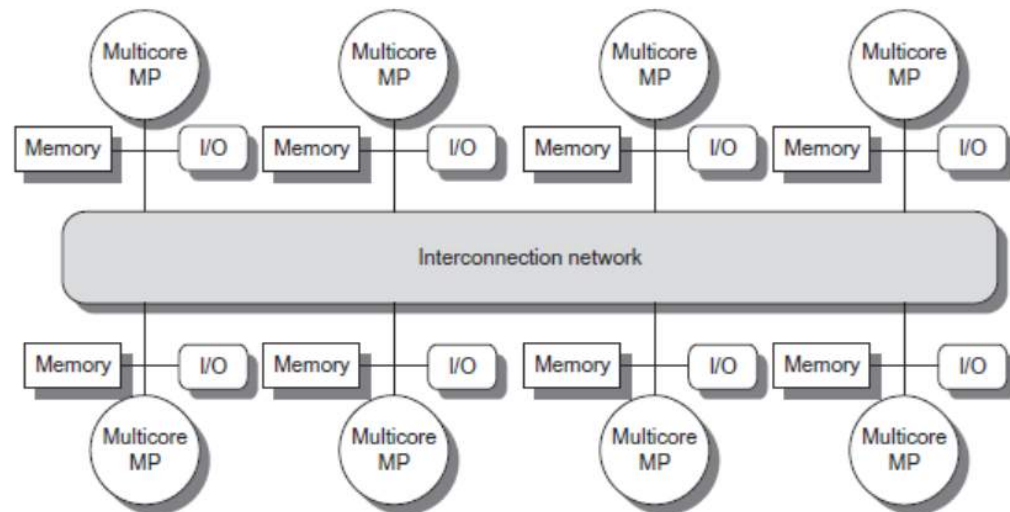
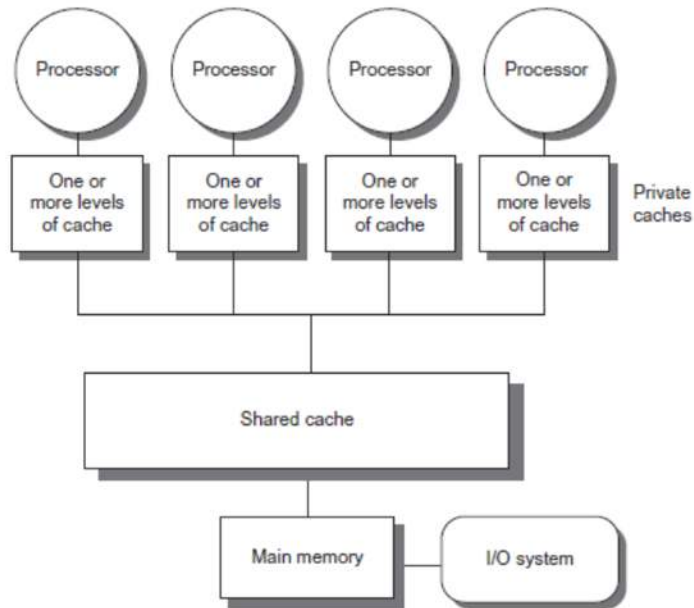
DSM[分布式共享内存]

- Distributed shared memory (DSM)
 - Memory distributed among processors
 - Non-uniform memory access/latency (NUMA)
 - The access time depends on the location of a data word in memory
 - Processors connected via direct (switched) and non-direct (multi-hop) interconnection networks



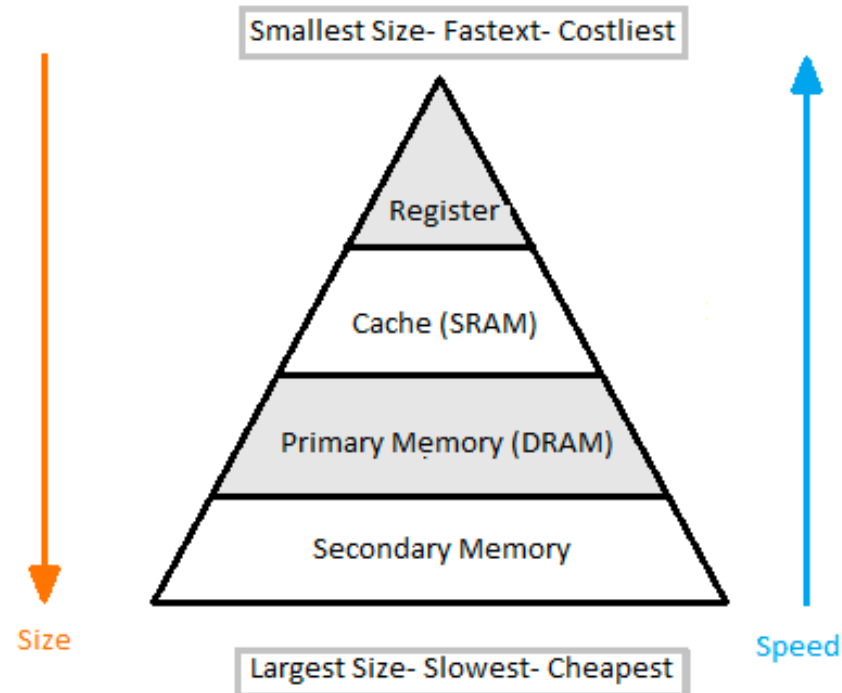
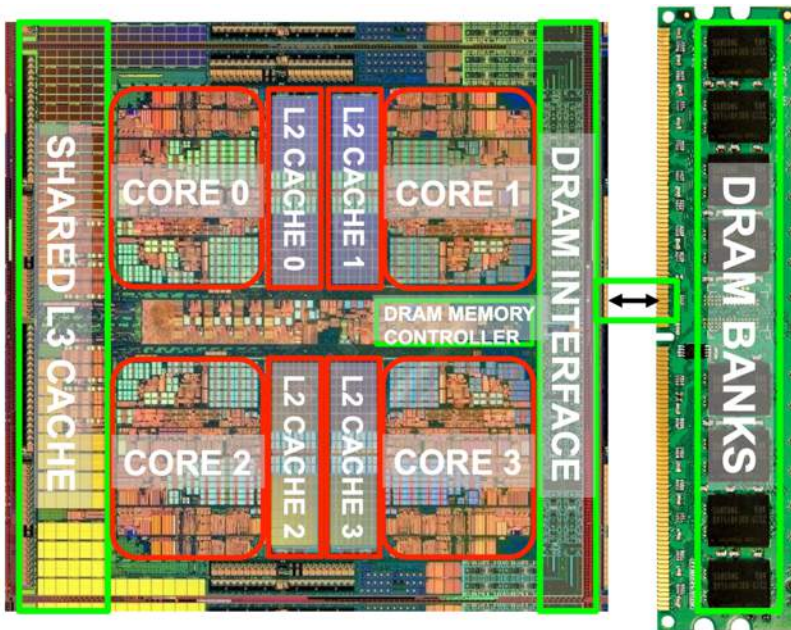
Shared Memory[共享内存]

- The term “**shared memory**” associated with both SMP and DSM refers to the fact that the address space is shared
 - Communication among threads occurs through the shared address space
 - Thus, a memory reference can be made by any processor to any memory location



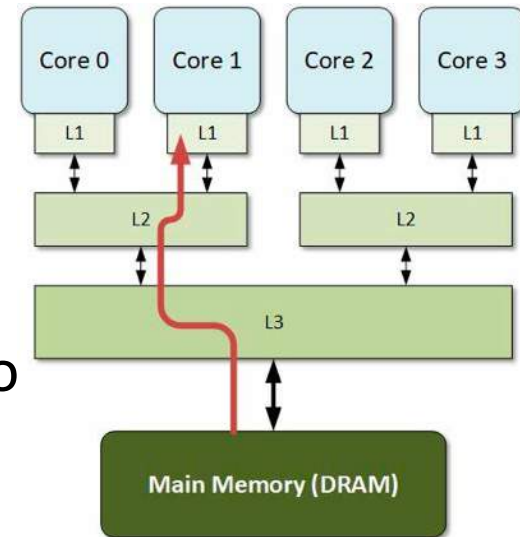
There Exist Caches

- Recall memory hierarchy, with **cache** being provided to shorten access latency
 - Each core of multiprocessors has a cache (or multiple caches)
- Caching complicates the **data sharing**



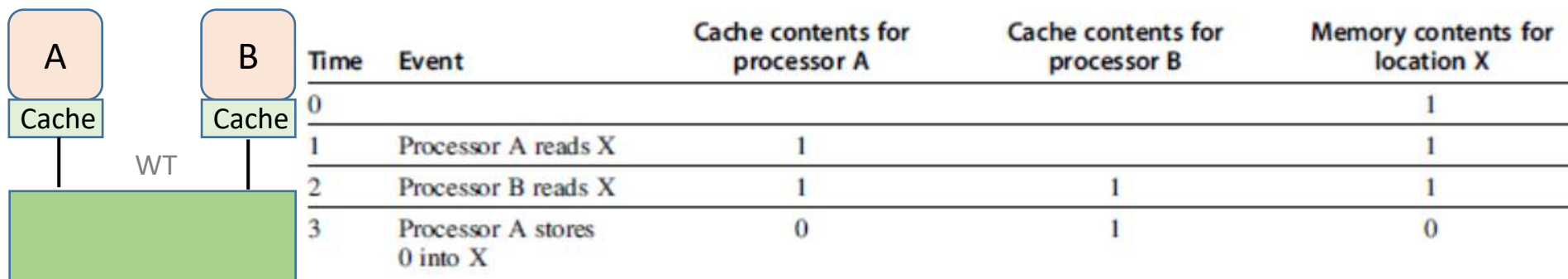
Data Caching[数据缓存]

- **Private** data: used by a single processor
- **Shared** data: used by multiple processors
 - Essentially providing communication among the processors through reads and writes of the shared data
- Caching private data
 - Migrated to cache, reducing access time
 - No other processor uses the data (identical to uniprocessor)
- Caching shared data
 - Replicated in multiple caches
 - Reduced access latency, reduced contention
 - Introduces a new problem: **cache coherence**



Cache Coherence[缓存一致性]

- Processors may **see different values** of the same data
 - The view of memory held by two different processors is through their individual caches, which, without any additional precautions, could end up seeing two different values
- Cache **coherence problem**[缓存一致性问题]
 - Conflicts between global state (main memory) and local state (private cache)
 - At time 4, what if processor B reads X?



A memory system is coherent, if

- A read by processor P to location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always return the value written by P
 - Preserves program order
- A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are **sufficiently separated in time** and no other writes to X occur between the two accesses
 - Defines the notion of what it means to have a coherent view of memory
- Writes to the same location are serialized; that is, two writes to the same location by any two processors are seen in the same order by all processors
 - Write serialization

Consistency also Matters[内存一致性]

- The three properties are sufficient to ensure coherence
- However, **when** a written value will be seen is also important
 - A write of X on one processor precedes a read of X on another processor by a very small time, it may be impossible to ensure that the read returns the value of the data written, since the written data may not even have left the processor at that point
- **Memory consistency**: when a written value must be seen by a reader

Thread 1

```
(1) A = 1  
(2) print(B)
```

Thread 2

```
(3) B = 1  
(4) print(A)
```

A and B are initially both 0

What this program can output?

- 01: (1)(2)(3)(4) or (3)(4)(1)(2)
- 11: (1)(3)(2)(4) or (1)(3)(4)(2)
- 00?

Coherence vs. Consistency[对比]

Coherence

- Coherence in writing means that all the ideas in a paragraph flow smoothly from one sentence to the next sentence.
- With coherence, the reader has an easy time understanding the ideas that you wish to express.

- Coherence[缓存一致性]

- Defines **what** values can be returned by a read
- All reads by any processor must return the most recently written value
- Writes to the **same location** by any two processors are seen in the same order by all processors

- Consistency[内存一致性]

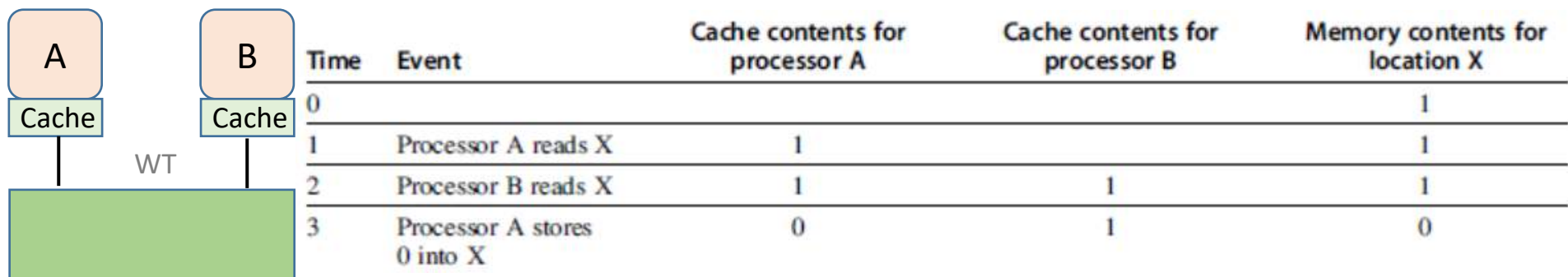
- Determines **when** a written value will be returned by a read
- Consistency insures that writes to **different locations** will be seen in an order that makes sense, given the source code
- If a processor writes location A followed by location B, any processor that sees the new value of B must also see the new value of A

Enforcing Coherence[保证一致性]

- Coherent caches provide
 - **Migration:** movement of data[搬运]
 - A data item can be moved to a local cache and used there in a transparent fashion
 - **Replication:** multiple copies of data[备份]
 - Make a copy of the data item in the local cache, so that shared data can be simultaneously read
- Whose responsibility? Software?
 - Can programmer ensure coherence if caches invisible to sw?
 - What if the ISA provided a cache flush instruction?
 - FLUSH-LOCAL A: flushes/invalidates the cache block containing address A from a processor's local cache
 - FLUSH-GLOBAL A: flushes/invalidates the cache block containing address A from all other processors' caches
 - FLUSH-CACHE X: flushes/invalidates all blocks in cache X

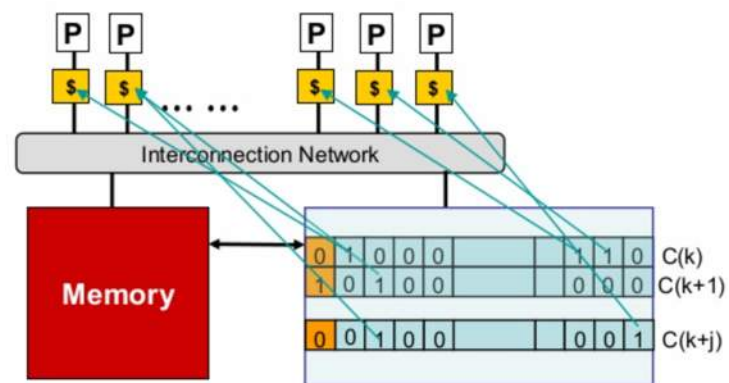
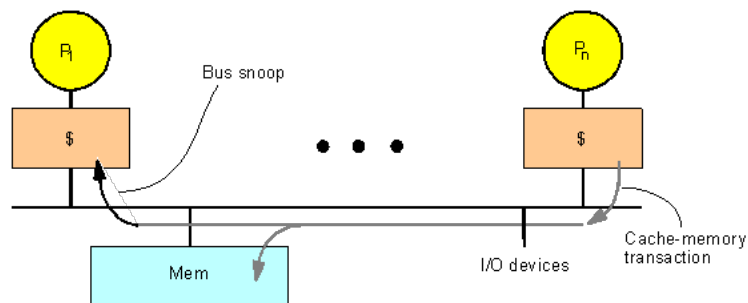
Enforcing Coherence (cont.)

- Software solutions are of high overheads
 - And, programming burden
- Multiprocessors adopt a hardware solution to maintain coherent caches[硬件方案]
 - Supporting the migration and replication is critical to performance in accessing shared data
- For the example,
 - Invalidate all other copies of X when B writes to it



Coherence Protocols[缓存一致性协议]

- Cache **coherence protocols**: the rules to maintain coherence for multiple processors
 - Key is to track the state of any sharing of a data block
- Two classes of protocols
 - Snooping[窥探]
 - Each core tracks sharing status of each block
 - Directory based[基于目录]
 - Sharing status of each block kept in one location



■ 1 modified bit for each cache block in memory

■ 1 presence bit for each processor, each cache block in memory

Snooping Coherence Protocols[窥探]

- Write invalidation protocol[写无效]
 - Ensure that a processor has exclusive access to a data item before it writes that item
 - Exclusive access ensure that no other readable or writable copies of an item exist when the write occurs
 - All other cached copies of the item are **invalidated** (👉 that's the name)
- Write update/broadcast protocol[写更新]
 - Update all the cached copies of data item when that item is written
 - Must broadcast all writes to shared cache lines, and thus consumes considerably more bandwidth
- Write invalidation protocol is by far the most common
 - We'll focus on it

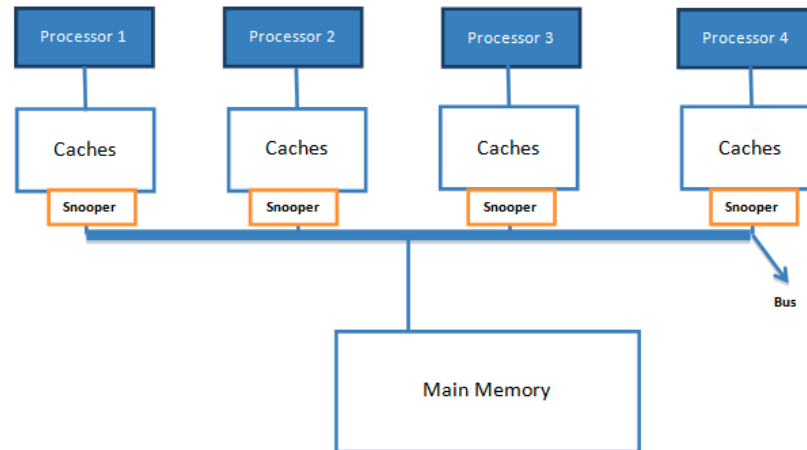
Write Invalidation Protocol[写无效]

- Write invalidate
 - On write, invalidate all other copies
 - Use bus itself to serialize
- Example
 - Invalidation protocol working on a snooping bus for a single block (X) with write-back caches

Processor activity	Bus activity	Contents of processor A's cache	Contents of processor B's cache	Contents of memory location X
Neither cache initially holds X and the value of X in memory is 0				
Processor A reads X	Cache miss for X	0		0
Processor A reads X, migrating from memory to the local cache				
Processor B reads X	Cache miss for X	0	0	0
Processor B reads X, migrating from memory to the local cache				
Processor A writes a 1 to X	Invalidation for X	1		0
Processor A writes X, invalidating the copy on B				
Processor B reads X	Cache miss for X	1	1	1
Processor B reads X, A responds with the value canceling the mem response and updates both B's cache and memory				

Snoopy Implementation[窥探实现]

- Key is to use bus, or another broadcast medium, to perform invalidates
- To perform an **invalidate**
 - The processor simply acquires bus access and broadcasts the address to be invalidated on the bus[获得总线，广播地址]
 - All processors continuously snoop on the bus, watching the addresses[窥探总线，收听地址]
 - The processors check whether the address on the bus is in their cache. If so, the corresponding data in the cache is invalidated[核对地址，作废数据]



Snoopy Implementation (cont.)

- When a write to a block that is shared occurs,[写到共享块]
 - The writing processor must acquire bus access to broadcast its invalidation
- If two processors attempt to write shared blocks at the same time,[两个处理器同时写到共享块]
 - Their attempts to broadcast an invalidate operation will be serialized when they arbitrate for the bus[串行‘无效’操作]
 - The first processor to obtain bus access will cause any other copies of the block it is writing to be invalidated[作废数据]
 - If the processors were attempting to write the same block, the serialization enforced by the bus also serializes their writes[串行写操作]

Snoopy Implementation (cont.)

- Locate a data item when a cache miss occurs,[找到数据]
 - For write-through cache, easy to find the recent value[写通]
 - All written data are always sent to the memory
 - For write-back cache, harder to find the most recent value[写回]
 - The newest value can be in a private cache rather than in the shared cache or memory
- Happily, write-back caches can use the same snooping scheme both for cache misses and for writes[同样窥探]
 - Each processor snoops every address placed on the shared bus[每个处理器窥探每个地址]
 - If a processor finds that it has a dirty copy of the requested cache block, it provides that block in response to the read request and causes the memory (or L3) access to be aborted[某个处理器拥有脏数据→ 响应]