Cache Coherence in Multi-cores and Multiprocssors

Mainak Chaudhuri
mainakc@cse.iitk.ac.in

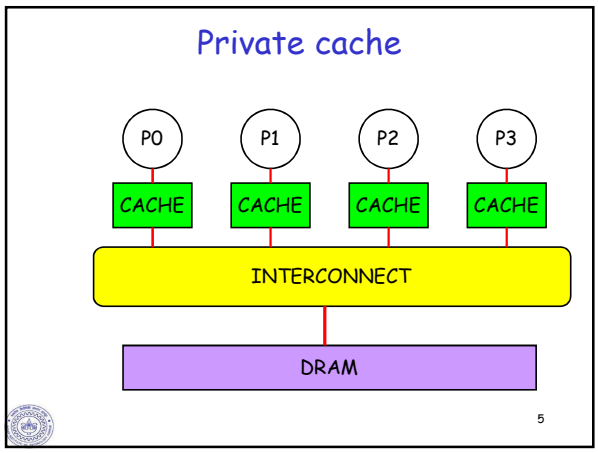
Agenda

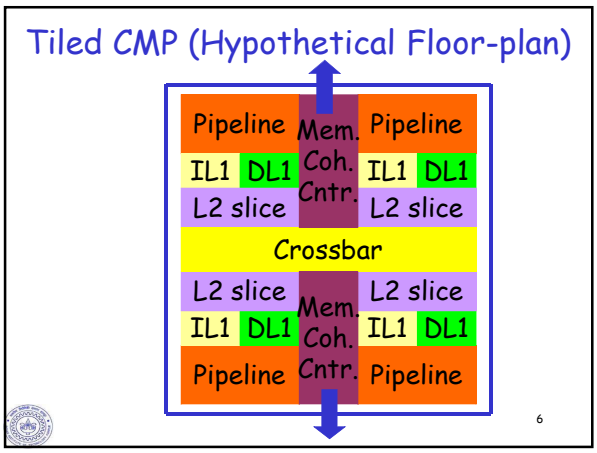
- General architecture
- Cache coherence
  - Directory-based coherence protocols
- Connection with GPUs
- Accelerator coherence

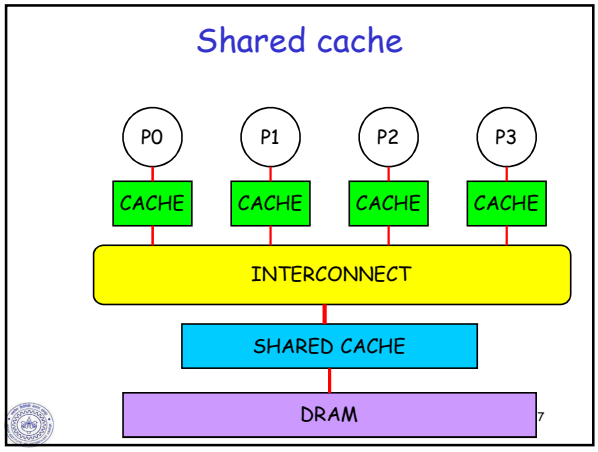
1. General Architecture

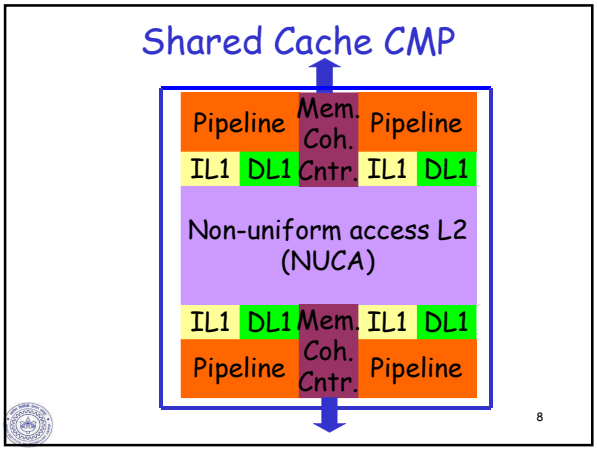
Shared memory multiprocessors

- What do they look like?
  - We will assume that each processor has a hierarchy of caches (possibly shared)
  - We will not discuss shared memory in a time-shared single-thread computer
    - A degenerate case of the following
  - Shared cache (popular in CMPs)
  - Private cache (popular in CMPs and SMPs)
  - Distributed shared memory (popular in medium to large-scale servers)

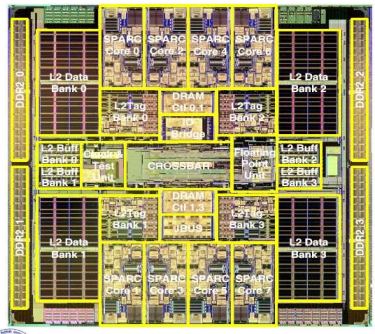

Private cache
P0
P1
P2
P3
CACHE
CACHE
CACHE
CACHE
INTERCONNECT
DRAM
5


Tiled CMP (Hypothetical Floor-plan)
Pipeline
Mem. Coh. Cntr.
Pipeline
IL1
DL1
IL1
DL1
L2 slice
L2 slice
Crossbar
L2 slice
L2 slice
Mem. Coh. Cntr.
IL1
DL1
IL1
DL1
Pipeline
Pipeline
6


Shared cache
P0
P1
P2
P3
CACHE
CACHE
CACHE
CACHE
INTERCONNECT
SHARED CACHE
DRAM
7


Shared Cache CMP
Pipeline
Mem. Coh. Cntr.
Pipeline
IL1
DL1
IL1
DL1
Non-uniform access L2 (NUCA)
IL1
DL1
Mem. Coh. Cntr.
IL1
DL1
Pipeline
Pipeline
8

Niagara Floor-plan



Features:

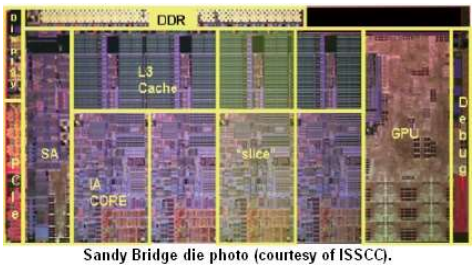
- Eight 64b Multithreaded SPARC Cores
- Shared 3MB L2 Cache
- 16KB ICache per Core
- 8KB DCache per Core
- Four 144b DDR-2 DRAM Interfaces (400 MTs)
- 3.2GB/s JBUS I/O
- Crypto: Public Key (RSA)
- Extensive RAS

Technology:

- 90nm CMOS Process
- 9LM Copper Interconnect
- Power: 63 Watts @ 1.2GHz
- Die Size: 378mm^2
- 279M Transistors
- Package: Flip-chip ceramic LGA (1933 pins)

9

Quad-core Sandy Bridge

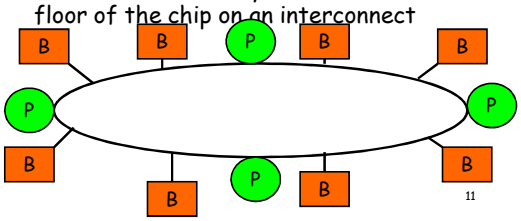


Sandy Bridge die photo (courtesy of ISSCC).

10

Shared vs. private in CMPs

- Shared caches are often very large in the CMPs
  - They are banked to avoid worst-case wire delay
  - The banks are usually distributed across the floor of the chip on an interconnect



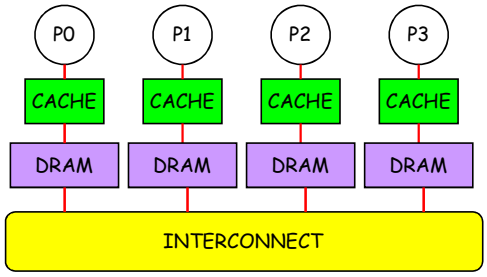
11

Shared vs. private in CMPs

- In shared caches, getting a block from a remote bank takes time proportional to the physical distance between the requester and the bank
  - Non-uniform cache architecture (NUCA)
- This is same for private caches, if the data resides in a remote cache
- Shared cache may have higher average hit latency than the private cache
  - Hopefully most hits in the latter will be local
- Shared caches are most likely to have less misses than private caches
  - Latter wastes space due to replication

12

Distributed shared memory



2. Cache Coherence

Cache coherence

- Nothing unique to multiprocessors
  - Even uniprocessor computers need to worry about cache coherence
  - For sequential programs we expect a memory location to return the latest value written
  - For concurrent programs running on multiple threads or processes on a single processor we expect the same model to hold because all threads see the same cache hierarchy (same as shared L1 cache)
  - For multiprocessors there remains a danger of using a stale value: hardware must ensure that cached values are coherent across the system and they satisfy programmers' intuitive memory model

Cache coherence: Example

- Assume a write-through cache
  - P0: reads x from memory, puts it in its cache, and gets the value 5 [(x, P0)=5]
  - P1: reads x from memory, puts it in its cache, and gets the value 5 [(x, P0)=5, (x, P1)=5]
  - P1: writes x=7, updates its cached value and memory value [(x, P0)=5, (x, P1)=7, (x, M)=7]
  - P0: reads x from its cache and gets the value 5
  - P2: reads x from memory, puts it in its cache, and gets the value 7 (now the system is completely incoherent) [P0=5, P1=7, P2=7]
  - P2: writes x=10, updates its cached value and memory value [P0=5, P1=7, P2=10, M=10]

Cache coherence: Example

- Consider the same example with a writeback cache
  - P0 has a cached value 5, P1 has 7, P2 has 10, memory has 5 (since caches are not write through)
  - The state of the line in P1 and P2 is M while the line in P0 is clean
  - Eviction of the line from P1 and P2 will issue writebacks while eviction of the line from P0 will not issue a writeback (clean lines do not need writeback)
  - Suppose P2 evicts the line first, and then P1
  - Final memory value is 7: we lost the store x=10 from P2

What went wrong?

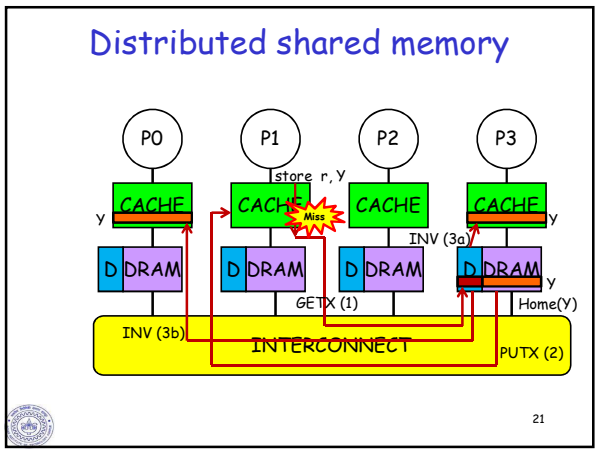
- For write through cache
  - The memory value may be correct if the writes are correctly ordered
  - But the system allowed a store to proceed when there is already a cached copy
  - Lesson learned: must invalidate all cached copies before allowing a store to proceed
- Writeback cache
  - Problem is even more complicated: stores are no longer visible to memory immediately
  - Writeback order is important
  - Lesson learned: do not allow more than one copy of a cache line in M state

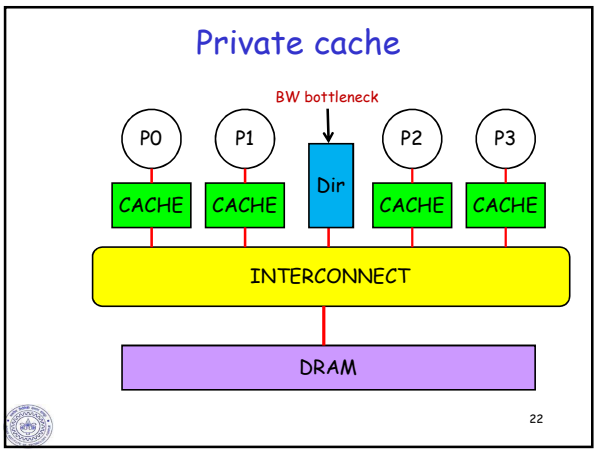
Implementations

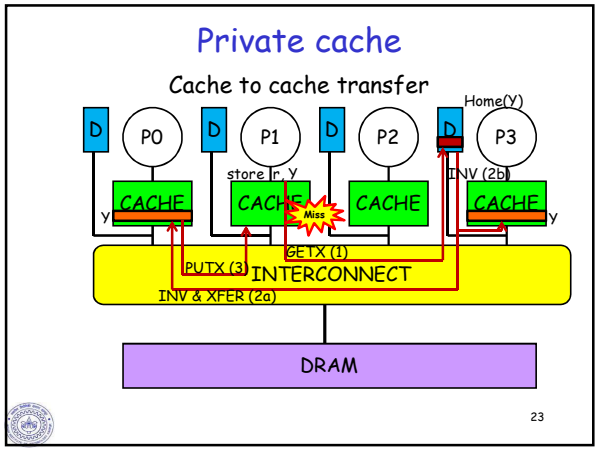
- Must invalidate all cached copies before allowing a store to proceed
  - Need to know where the cached copies are
  - Solution1: Never mind! Just tell everyone that you are going to do a store
    - Leads to broadcast snoopy protocols
    - Popular with small-scale bus-based CMPs and SMPs
    - AMD Opteron implements it on a distributed network (the Hammer protocol)
  - Solution2: Keep track of the sharers and invalidate them when needed
    - Where and how is this information stored?
    - Leads to directory-based scalable protocols

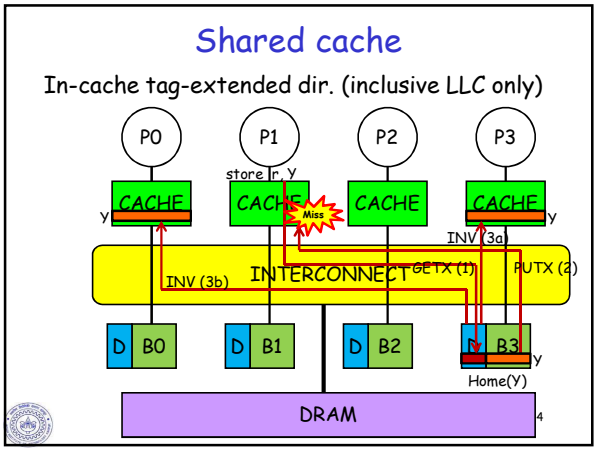
Implementations

- Directory-based protocols
  - Maintain one directory entry per memory block
  - Each directory entry contains a sharer bitvector and state bits
  - Concept of home node in distributed shared memory multiprocessors
  - Concept of sparse directory for on-chip coherence in CMPs


Distributed shared memory
P0
P1
P2
P3
store r, y
CACHE
Miss
y
INV (3a)
D DRAM
Home(Y)
GETX (1)
INV (3b)
INTERCONNECT
PUTX (2)
21

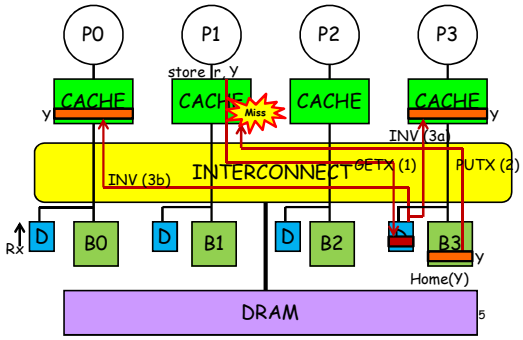

Private cache
BW bottleneck
P0
P1
P2
P3
Dir
CACHE
INTERCONNECT
DRAM
22


Private cache
Cache to cache transfer
D
P0
P1
P2
P3
Home(Y)
store r, y
INV (2b)
CACHE
Miss
y
GETX (1)
PUTX (3)
INTERCONNECT
INV & XFER (2a)
DRAM
23


Shared cache
In-cache tag-extended dir. (inclusive LLC only)
P0
P1
P2
P3
store r, y
CACHE
Miss
y
INV (3a)
INV (3b)
INTERCONNECT
GETX (1)
PUTX (2)
D B0
D B1
D B2
D B3
Home(Y)
DRAM
4

Shared cache

Sparse directory; R = #dir. entries/#private$ blocks



Implementations

- Sparse directory
  - Organized as a set-associative tagged cache of directory entries
  - Responsible for keeping track of only those blocks currently in the private portion of the cache hierarchy
  - Ideally, sparse directory should have number of sets equal to the number of sets in the private cache and number of ways equal to aggregate associativity of all private caches
    - This is not a scalable organization
    - A practically implementable sparse directory is over-provisioned by factor R to eliminate some of the conflicts

Implementations

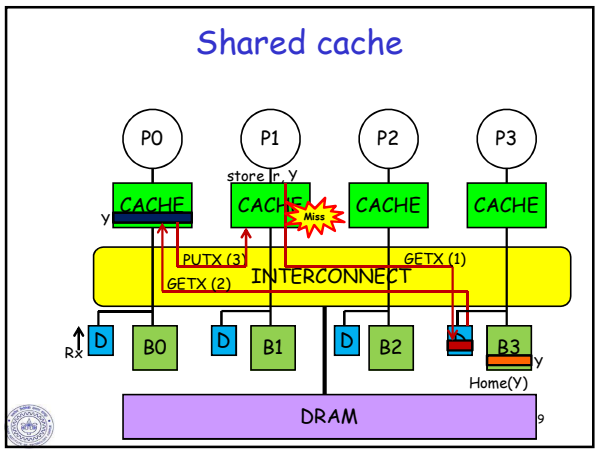
- Do not allow more than one copy of a cache line in M state
  - Need some form of access control mechanism
  - Before a processor does a store it must take "permission" from the current "owner" (if any)
  - Need to know who the current owner is
    - Either a processor or main memory
  - Solution1 and Solution2 apply here also

27

Implementations

- Latest value must be propagated to the requester
  - Notion of "latest" is very fuzzy
  - Once we know the owner, this is easy
  - Solution1 and Solution2 apply here also

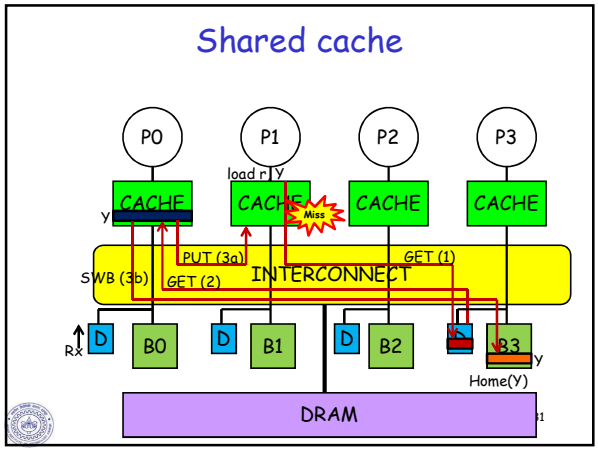
28



Implementations

- Invariant: if a cache block is not in M state in any processor, memory must provide the block to the requester
  - Memory must be updated when a block transitions from M state to S state
  - Note that a transition from M to I always updates memory in systems with writeback caches (these are normal writeback operations)
- Most of the implementations of a coherence protocol deals with uncommon cases and races

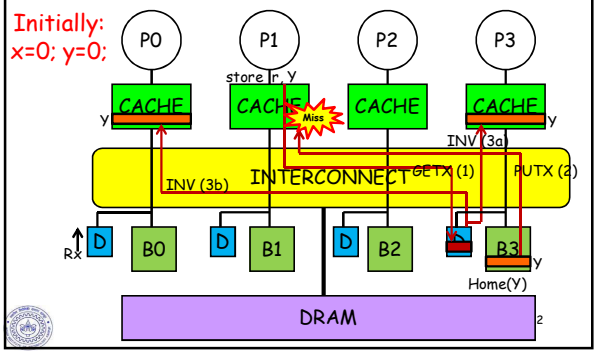
30



Invalidation acknowledgments

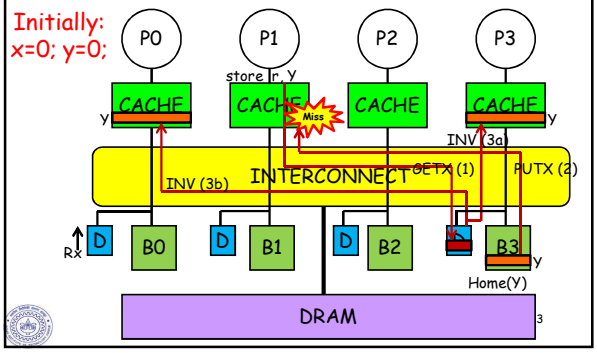
P1: y=1; P0: while (!y); x=1; P3: while(!x); print y;

Initially:
x=0; y=0;



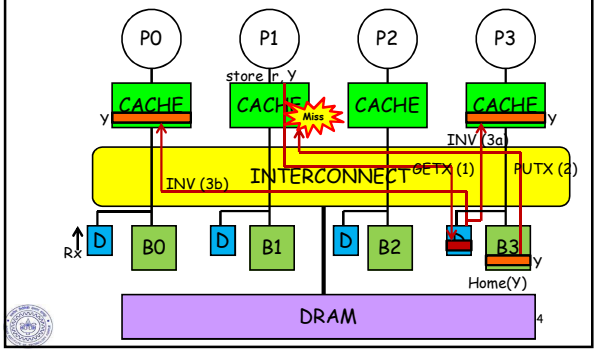
Invalidation acknowledgments

P1: y=1; x=1; P3: while(!x); print y;



Invalidation acknowledgments

Inval. acks are needed to establish store completion



2. Connection with GPUs

35

Cache coherence in GPUs

- GPUs have L1 caches private to each streaming multiprocessor and a shared L2 cache
  - L1 caches are not coherent
    - Why?
    - Overheads of a directory-based cache coherence protocol: directory storage, network traffic, additional latency on critical path
    - Which one of these overheads could be prohibitive for GPUs?
  - Software must take charge of coherence
    - No sharing across streaming multiprocessors: only embarrassingly parallel programs can be written
    - Flavors of load and store instructions to convey to hardware whether to bypass the L1 cache

36

Cache coherence in GPUs

- Flavors of loads/stores
  - Default flavor (ld.ca and st.ca) enables L1 caching
  - Load/Store to global data can use the global caching flavors (ld.cg and st.cg) to disable L1 caching
    - Data is cached in L2 cache only
    - __ldcg(addr)
- Coherence between CPU and GPU
  - Usually coarse-grain (page level)
  - Fine-grain coherence (block level) is limited to specialized integrated architectures e.g., Grace-Hopper

Ordering of global accesses in GPUs

- Flavors are not enough to enforce ordering of global loads/stores
  - Example:
    - T1 on SM1 executes A=1; flag=1;
    - T2 on SM2 executes while (!flag); x=A+1;
    - All loads/stores use global flavors
    - There is no guarantee that T2 will use the new value of A because the stores A=1 and flag=1 may get reordered in the interconnect between SM1 and the respective L2 cache banks
  - Need fence instructions to enforce the desired memory operation order within SM/across SM/across GPUs/across CPU-GPU
    - T1 should have A=1; fence; flag=1;

4. Accelerator Coherence

Accelerators

- Hardware coprocessors to speed up execution of domain-specific functions
  - Graphics: GPUs
  - Crypto: crypto accelerators
  - Floating-point arithmetic: FP coprocessors
  - AI/ML accelerators
- Accelerators usually operate as slaves of the main processor (aka the host processor)
  - Programs annotate portions of the computation that should be off-loaded to the accelerator
  - Host initiates accelerator execution

Accelerators

- Discrete accelerator
  - Separate accelerator chip connected to the host
  - Accelerator chip comes with its own cache hierarchy and memory
  - Data may have to be copied explicitly or implicitly from host memory to accelerator memory before accelerator can operate on the data
  - Example: GPU cards connected via PCIe bus
    - DMA engines are employed to copy data from host memory to GPU memory
    - Copy may be explicitly done in the program or implicitly done by the GPU driver

41

Accelerators

- Integrated accelerator
  - Accelerator integrated into the host processor chip
  - Accelerator can share parts of the host's cache and memory resources
  - Data copy between accelerator and host unnecessary
    - Efficient fine-grain data sharing between host and accelerator possible through physically shared cache and memory
  - Computationally less provisioned than discrete accelerators due to power (thermal) and area constraints

42

Accelerators

- Integrated accelerator
  - Example: Integrated GPU in Intel CPUs starting from Sandy Bridge processor (2011)
    - GPU shares the L3 cache, on-chip memory controllers, and main memory with the on-chip cores
    - Intel's Crystalwell processor (2013) introduced an L4 eDRAM cache of capacity up to 128 MB to support the high BW demand of the integrated GPU
      - Sunset after Skylake processor due to improvements in the main three-level cache hierarchy
  - Example: Integrated GPU in AMD CPUs starting from Llano family (2011)
    - GPU does not share any cache with CPU cores, but shares the memory controllers and main memory

43

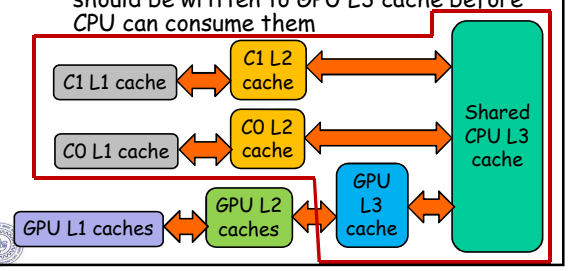
Accelerator coherence

- Accelerator-level parallelism (ALP): program partitioned between host CPU cores and accelerator(s)
  - May have data shared between CPU cores and accelerators in read/write mode
  - Requires coherence among CPU cores and accelerators
- Integrated GPU with last-level cache shared with CPU
  - GPU's last-level cache is kept coherent with the CPU cores' private caches (Intel's implementation)
  - Fully coherent hierarchy is expensive

44

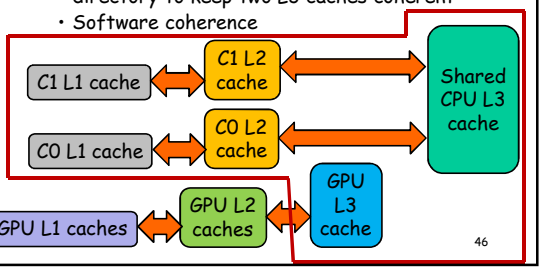
Accelerator coherence

- Integrated GPU with last-level cache shared with CPU
  - GPU's last-level cache is kept coherent with the CPU cores' private caches: GPU results should be written to GPU L3 cache before CPU can consume them



Accelerator coherence

- Integrated GPU without shared cache
  - Coherence options
    - CPU caches and last-level GPU cache kept coherent by hardware: two-bit wide second-level directory to keep two L3 caches coherent
    - Software coherence



46

Accelerator coherence

- Integrated GPU without shared cache
  - Hardware coherence: bandwidth of second-level directory could be a bottleneck
    - This bottleneck is present in Intel's integrated GPUs also, but masked under shared L3 cache bandwidth (dir. BW = L3 cache tag BW)
  - Software coherence (AMD Llano)
    - Before GPU starts working, relevant dirty data must be written back from CPU cache hierarchy to main memory
    - GPU results must be flushed out of GPU cache hierarchy before CPU can consume them
    - No guarantee of coherence if CPU and GPU simultaneously read/write from/to some shared data (same for Intel's integrated GPUs)

47

Accelerator coherence

- Discrete GPU+CPU
  - Coherence options
    - Hardware coherence: cache-block grain
      - Possible only if CPU and GPU are connected over a coherence-aware interconnect
      - Possible only if CPU, GPU, and interconnect are manufactured by the same industry or in collaboration
      - Example: Grace CPU + Hopper GPU of Nvidia (referred to as a superchip, but actually a multi-chip module)
    - Hardware coherence: page-grain
    - Software coherence
      - Explicit copy operations done by program

48

Accelerator coherence

- Grace-Hopper coherence
  - Grace CPU has a three-level fully coherent cache hierarchy
    - Private L1+L2 caches per core and an L3 cache shared by CPU cores
  - Hopper GPU has a two level cache hierarchy
    - Private L1 data cache per SM and a shared L2 cache
    - L1 data cache is locally invalidate and write-through for global stores
  - Grace-Hopper coherent interconnect keeps the Grace cache hierarchy and Hopper L2 cache coherent using a high-bandwidth directory-based coherence protocol
    - CPU stores do not propagate to Hopper L1 caches

Accelerator coherence

- Grace-Hopper coherence
  - Supports acquire-release based lazy software coherence
  - Acquire is a synchronization operation with the intent to read (and possibly modify) the synchronization variable
    - Examples: lock acquire, flag read
  - Release is a synchronization operation with the intent to write to the synchronization variable
    - Example: lock release, flag write
  - Acquire-release must be used to propagate CPU stores to GPUs

Accelerator coherence

- Grace-Hopper coherence
  - Initially, x=0 and flag=0
  - CPU thread: x=1; flag=1;
  - GPU thread: while (!flag); use x;
    - Mark "flag=1" as a release, which makes the compiler insert a fence instruction right before "flag=1" (store to x may not commit its value immediately without such a fence instruction in Grace; worse, "x=1" may commit *after* "flag=1")
    - Mark the read of flag in GPU thread as an acquire, which flushes the L1 cache of the SM housing the GPU thread (known as self-invalidation)
      - Ensures that the reads of flag and x both will go to the Hopper L2 cache thereby entering the coherent domain

Accelerator coherence

- Grace-Hopper coherence (more example)
  - Suppose a CPU thread and a GPU thread execute a critical section
  - CPU thread: lock(L); x++; unlock(L)
  - GPU thread: lock(L); x++; unlock(L)
    - Mark "unlock(L)" as a release, which makes the compiler insert a fence instruction right before "unlock(L)" (ensures that store to x commits before unlock commits)
    - Mark "lock(L)" in GPU thread as an acquire, which self-invalidates the L1 cache of the SM housing the GPU thread
      - Ensures that the reads of L and x both will go to the Hopper L2 cache thereby entering the coherent domain

Accelerator coherence

- Page-grain hardware coherence in discrete GPU+CPU
  - Copies of a page with RO permission can exist in CPU memory as well as GPU memory
  - Store to a page from a CPU or a GPU invalidates the mapping of the page in all devices except the writer and migrates the latest copy of the page to the writer's memory
    - Fine-grain RW sharing is too costly
    - Cost can be reduced by migrating only diffs and by advertising stores only at release operations
    - Rich literature on page-grain coherence in shared memory multiprocessors (late 1980s to 1990s)
  - May take help from OS/driver

53

Accelerator coherence

- Software coherence
  - Options:
    - Explicitly copy data from CPU to GPU and back (managed by application developer)
    - Let the driver/OS do the copy operation triggered by faults due to CPU/GPU access to a page
    - Copies are done always at a grain that is multiple of page size
    - Optimization: OS/driver can infer access pattern and prefetch pages
  - Fine-grain RW sharing is costly

54