Computer Architecture

计 算 机 体 系 结 构

第11讲：DLP & GPU（5）

张献伟

xianweiz.github.io

DCS3013, 11/9/2022

Review Questions

- stream in GPU?

  software abstraction of queue, path to transmit tasks from CPU

- MPS?

  multi-process service to run multi processes on GPU

- cudaMemcpy() is blocking or not?

  Blocking: CPU waits for completion. vs. cudaMemcpyAsync()

- how to synchronize among streams?

  use events: eventRecord(), eventWait()

- shared memory in GPU?

  software-controlled L1 cache in SM, fast data share within block

- when to use local memory?

  register spilling, arrays inside kernels

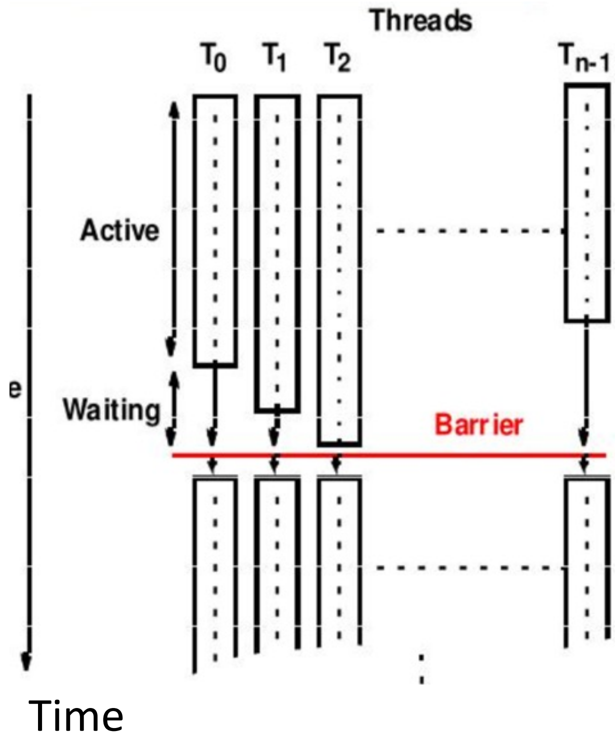
Shared Memory[“共享”存储]

- A per-block, software managed cache or scratchpad
  - Programmer can modify variable declarations with __shared__ to make this variable resident in shared memory
  - Compiler creates a copy of the variable for each block
    - Every thread in that block shares the memory, but threads cannot see or modify the copy of this variable that is seen within other blocks
    - This provides an excellent means by which threads within a block can communicate and collaborate on computations
- CUDA L1 cache and SMEM are unified
  - cudaDeviceSetCacheConfig(enum cudaFuncCache)
- A mechanism is needed to **synchronize** between threads
  - *Thread A* writes a value to shared memory and we want *thread B* to do something with this value
  - We can’t have *thread B* start its work until we know the write from *thread A* is complete

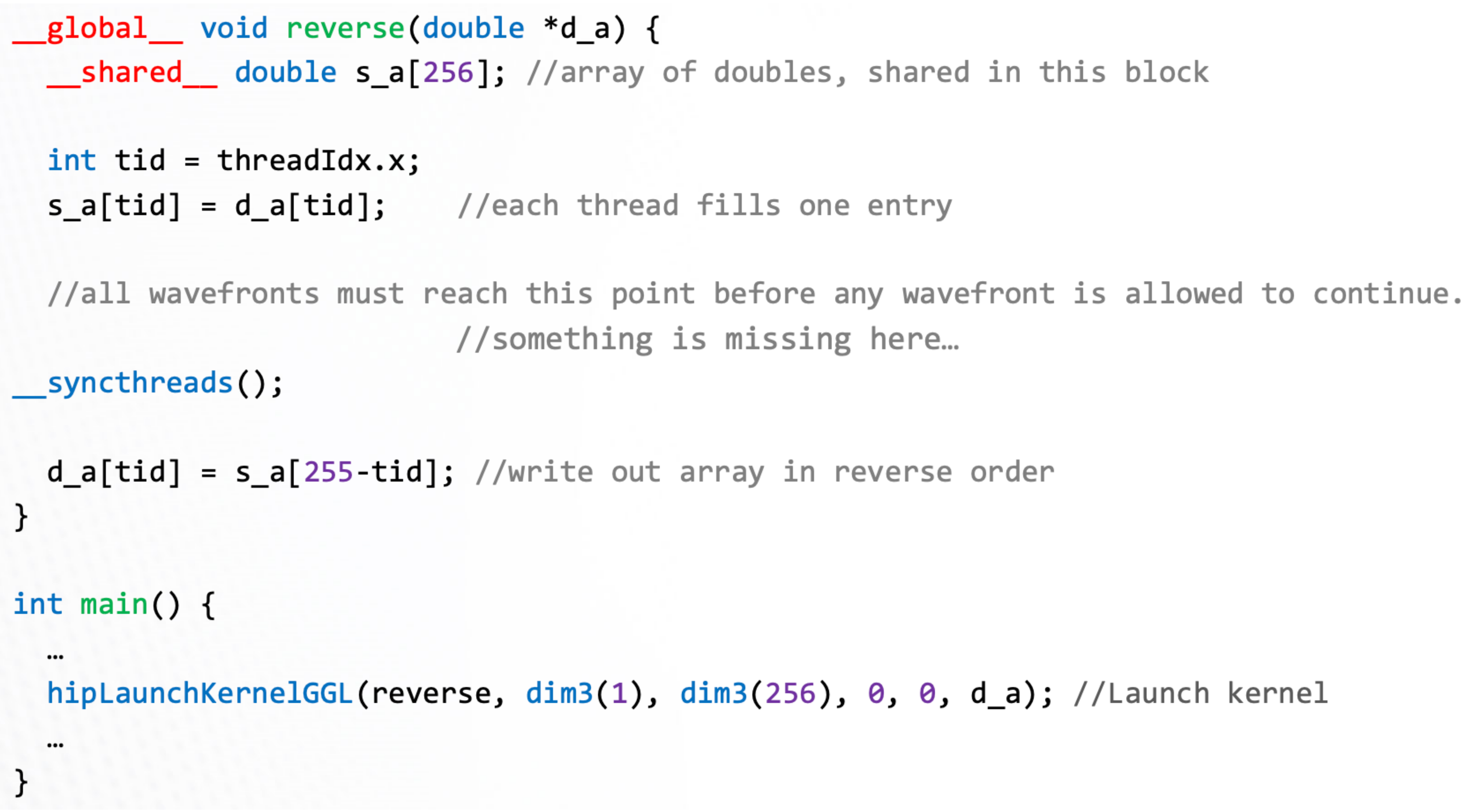
http://www.mat.unimi.it/users/sansotte/cuda/CUDA_by_Example.pdf

Shared Memory (cont.)

- One can specify synchronization points in the kernel by calling __syncthreads()
- __syncthreads() acts as a barrier at which all threads in the block must wait before any is allowed to proceed
  - Guarantees that every thread in the block has completed instructions prior to the __syncthreads() before the hardware will execute the next inst on any thread
  - When the first thread executes the first instruction after __syncthreads(), every other thread in the block has also finished executing up to the __syncthreads()



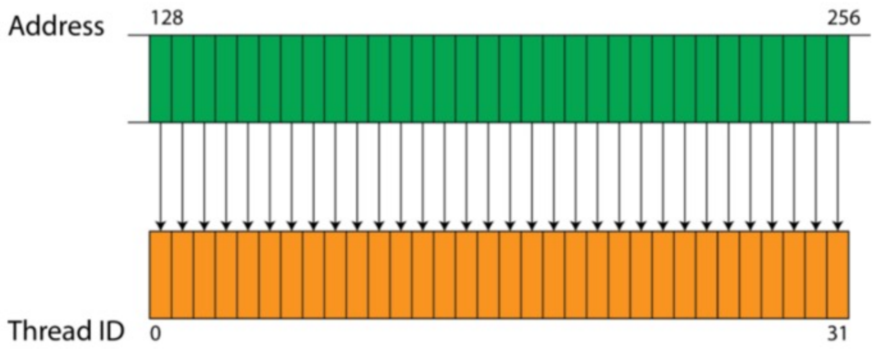
Example

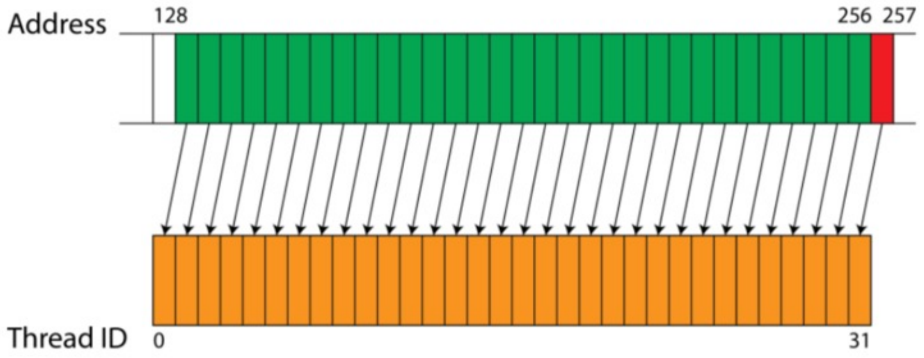


```
__global__ void reverse(double *d_a) {
  __shared__ double s_a[256]; //array of doubles, shared in this block

  int tid = threadIdx.x;
  s_a[tid] = d_a[tid];    //each thread fills one entry

  //all wavefronts must reach this point before any wavefront is allowed to continue.
                          //something is missing here…
__syncthreads();

  d_a[tid] = s_a[255-tid]; //write out array in reverse order
}

int main() {
  …
  hipLaunchKernelGGL(reverse, dim3(1), dim3(256), 0, 0, d_a); //Launch kernel
  …
}
```

Address Coalescing[地址合并]

- Threads in a block are computed a warp at a time (32 threads)
- Global data is read or written in as few transactions as possible by combining memory access requests into a single transaction
  - This is referred to the device coalescing mem stores and reads
- Every successive 128 bytes can be accessed by a warp (or 32 single precision words)
- Not in successive 128 bytes; more data to read

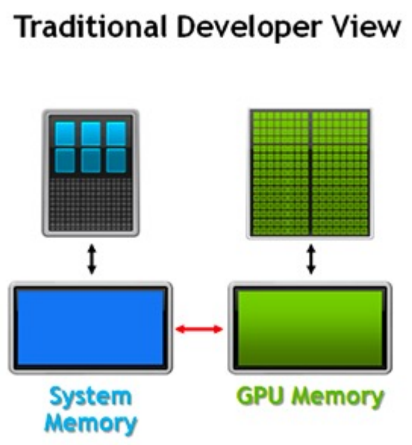


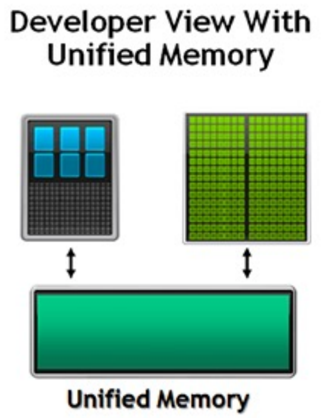


https://www.ce.jhu.edu/dalrymple/classes/602/Class13.pdf

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





Example

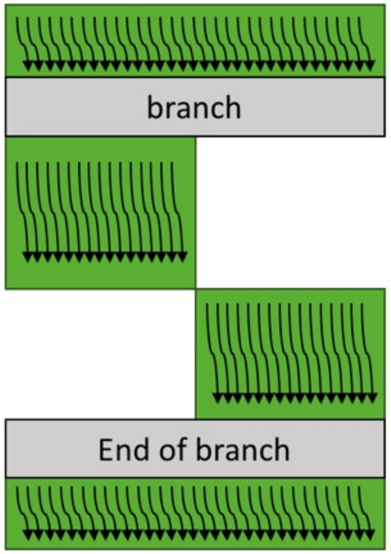
```
int N = 1<<20;
float *x, *y;

// 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);
```

Divergence[分支]

- Within a block of threads, the threads are executes 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 → serialized
- Branch divergence is a major cause for performance degr adation in GPGPUs

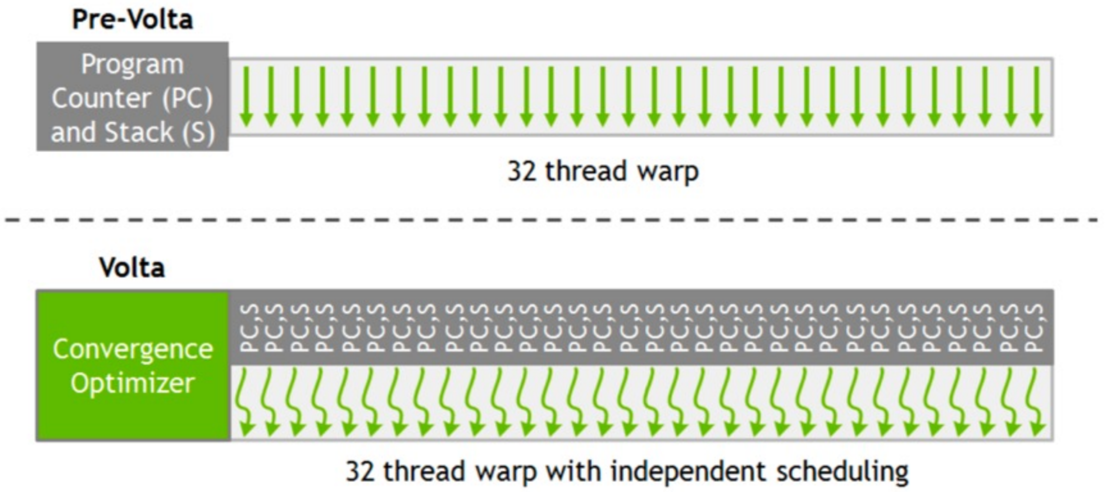
```
...
if ( threadIdx.x < 16 )
{
    ... A ...
}
else
{
    ... B ...
}
...
```

p = (threadIdx.x < 16);
if (p) … A …
if (!p) … B …



Divergence (cont.)

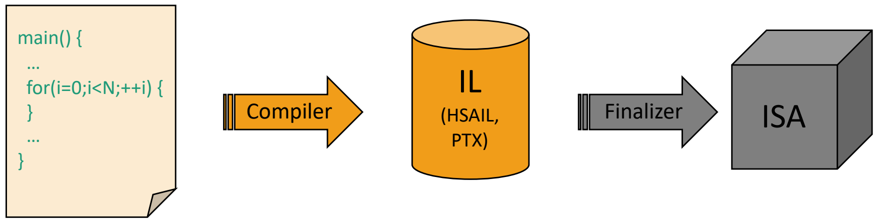
- Pre-Volta GPUs use a single PC shared amongst all 32 threads of a warp, combined with an active mask that specifies which threads of the warp are active at any given time
  - Leaves threads that are not executing a branch inactive
- Since Volta, each thread features its own PC, which allows threads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

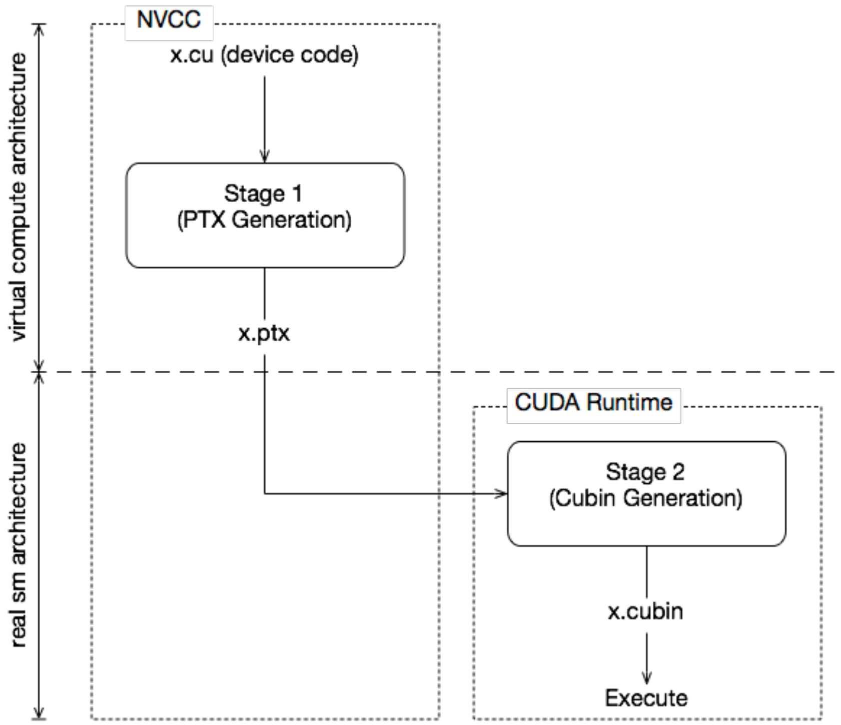

Pre-Volta
Program Counter (PC) and Stack (S)
32 thread warp
Volta
Convergence Optimizer
PC,S
32 thread warp with independent scheduling

https://www.icl.utk.edu/files/publications/2018/icl-utk-1080-2018.pdf

Two-phase Execution[两段式]

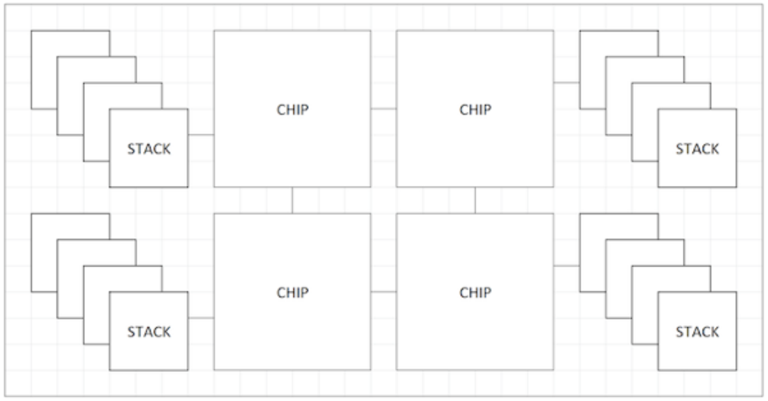
- Compilation workflow
  - Source code → virtual instruction (PTX or HSAIL)
  - Virtual inst → real inst (SASS or GCN)
- **.cu**: CUDA source file, containing host code and device functions
- **.ptx**: PTX intermediate assembly file
- **.cubin**: CUDA device code binary file (CUBIN) for a single GPU architecture

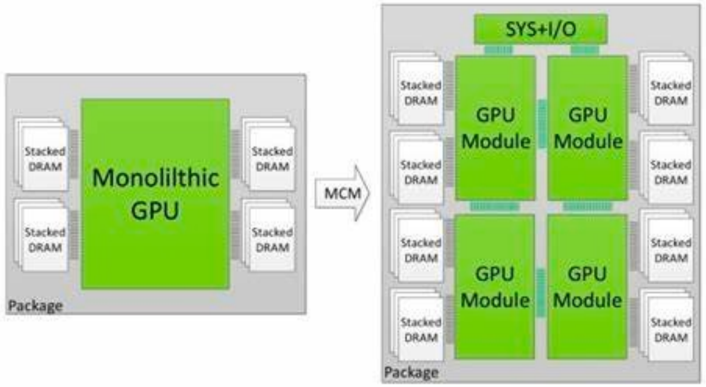




Multi-chip Module

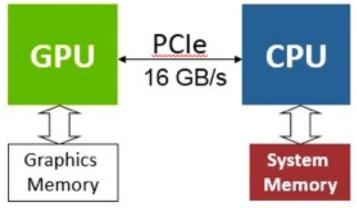
- Aggregating multiple GPU modules within a single package, as opposed to a single monolithic die.
- AMD: Chiplet GPUs
  - MI200: 220 compute units, 14K streaming cores
  - MI100: 120 compute units, 7680 streaming cores
- Nvidia: Multi-Chip-Module (MCM) GPUs
  - Hopper (Ampere -> Lovelace): 300+ SMs,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



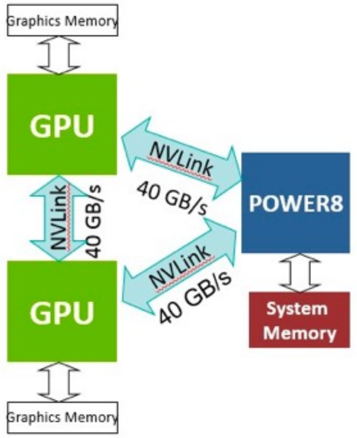


High-speed Links[高速连接]

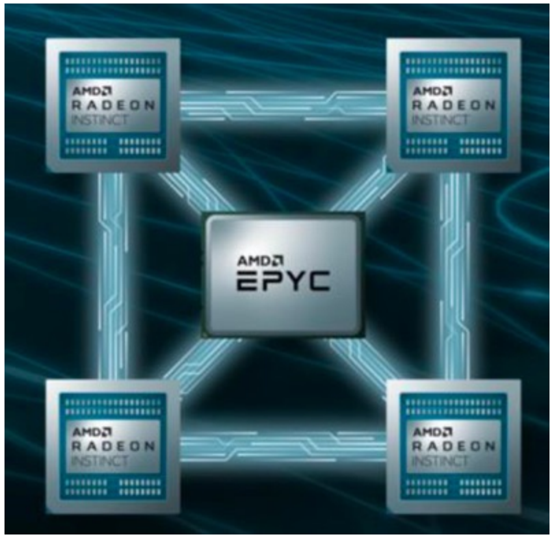
- GPUs are of high compute capability, being bottlenecked on data movement
- High-speed interconnect to achieve significantly higher data movement
  - Nvidia: NVLink
  - AMD: Infinity Fabric
  - Intel: Compute eXpress Link (CXL)



CPU-GPU Systems Connected via PCI-e



NVLink Enables Fast Unified Memory Access between CPU & GPU Memories



Computer Architecture
计 算 机 体 系 结 构

第11讲：Memory（1）

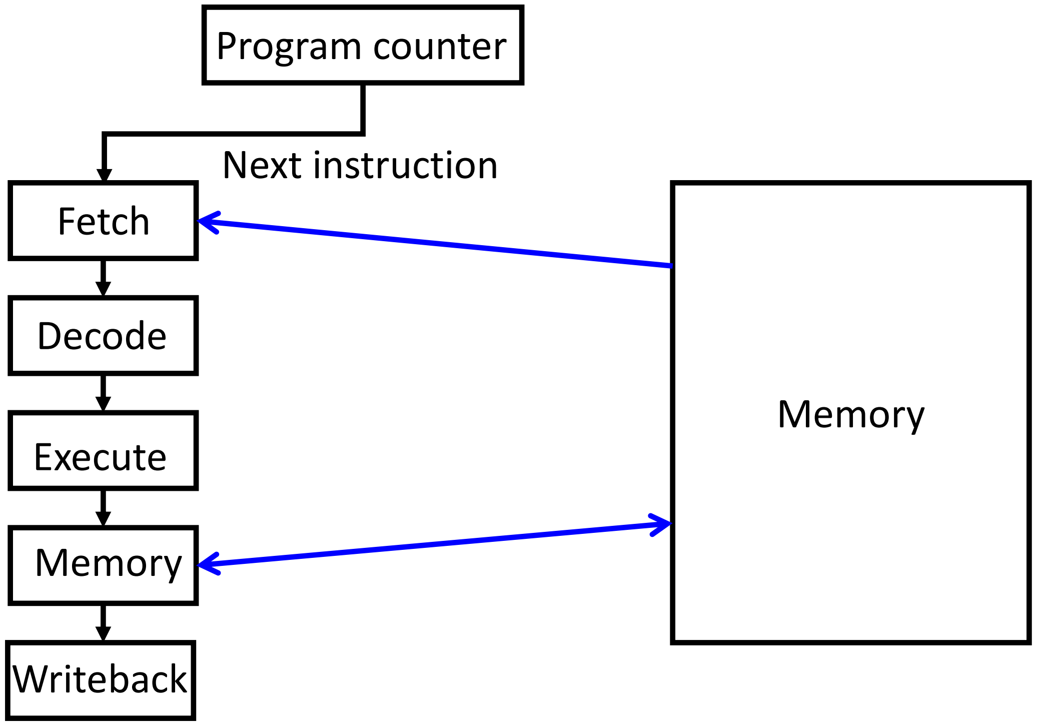
张献伟

xianweiz.github.io

DCS3013, 11/9/2022

Memory Access[存储访问]

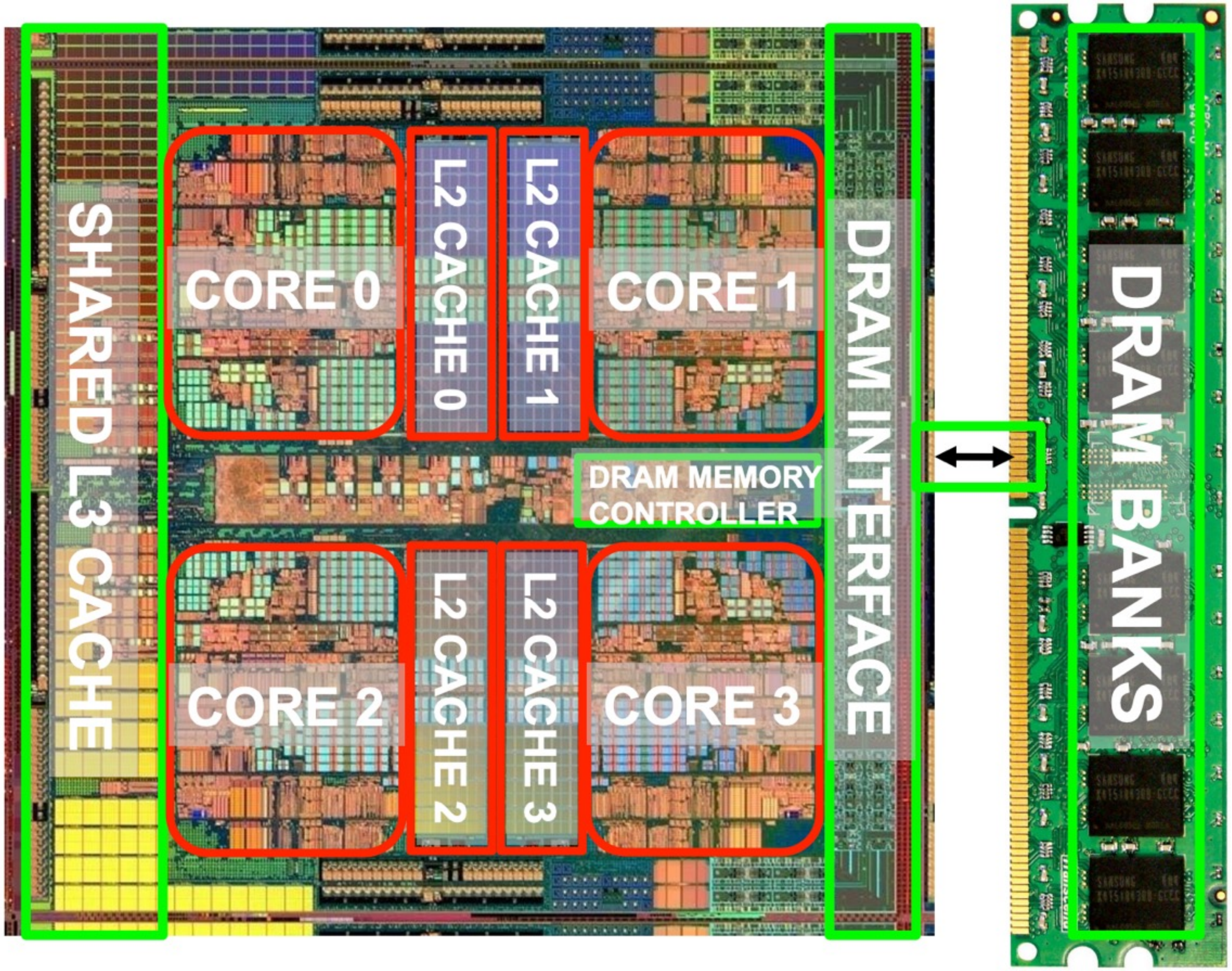
- Programmer’s view: read/write (i.e., load/store)
  - Instruction[指令]
  - Data[数据]



Memory[存储]

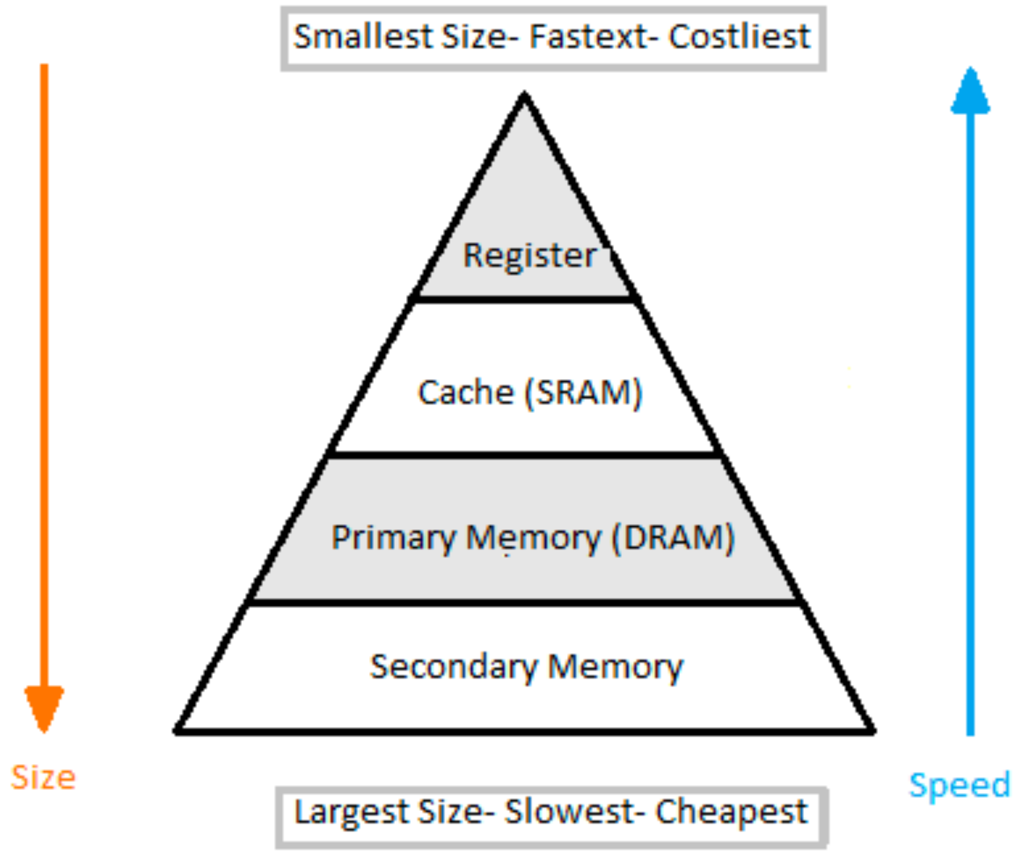
- Ideal memory[理想情况]
  - Zero access time (latency)[零时延]
  - Infinite capacity[无限容量]
  - Zero cost[零成本]
  - Infinite bandwidth (to support parallel accesses)[无限带宽]
- Problem: the requirements are conflicting[问题：需求互斥]
  - Bigger is slower[大容量→长时延]
    - Longer time to determine the location
  - Faster is more expensive[快访问→高成本]
    - More advanced technology
  - Higher bandwidth is more expensive[高带宽→高成本]
    - More access ports, higher frequency, …

Memory in Modern System

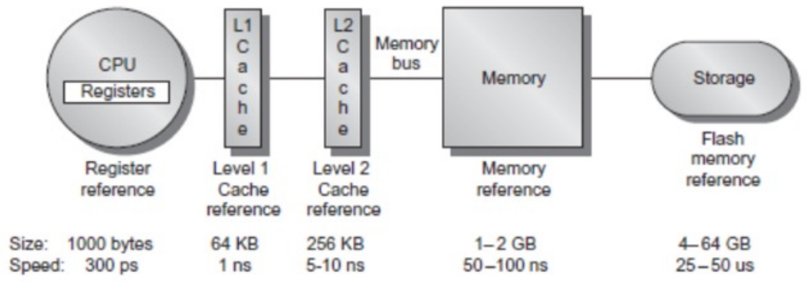


Memory Hierarchy[存储层级]

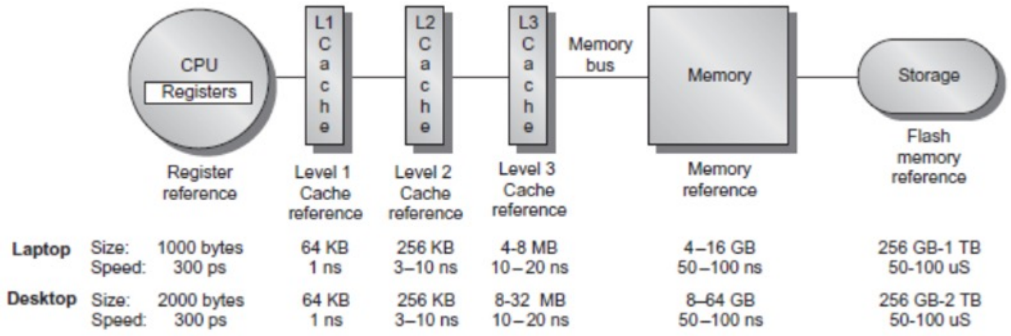
- Goal: provide a memory system with a cost per bit that is almost as low as the cheapest level of memory and a speed almost as fast as the fastest level



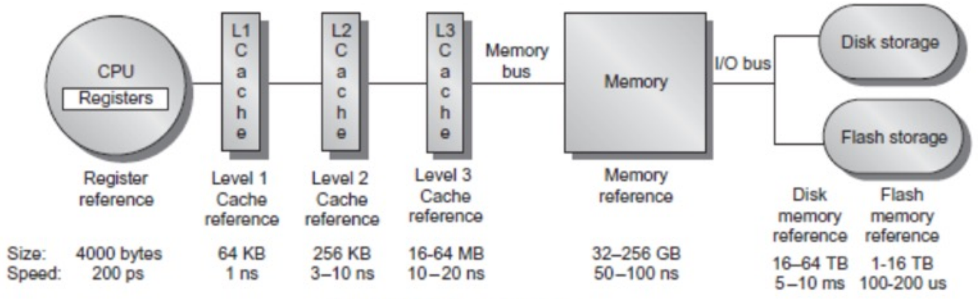
Memory Hierarchy (cont.)



(A) Memory hierarchy for a personal mobile device



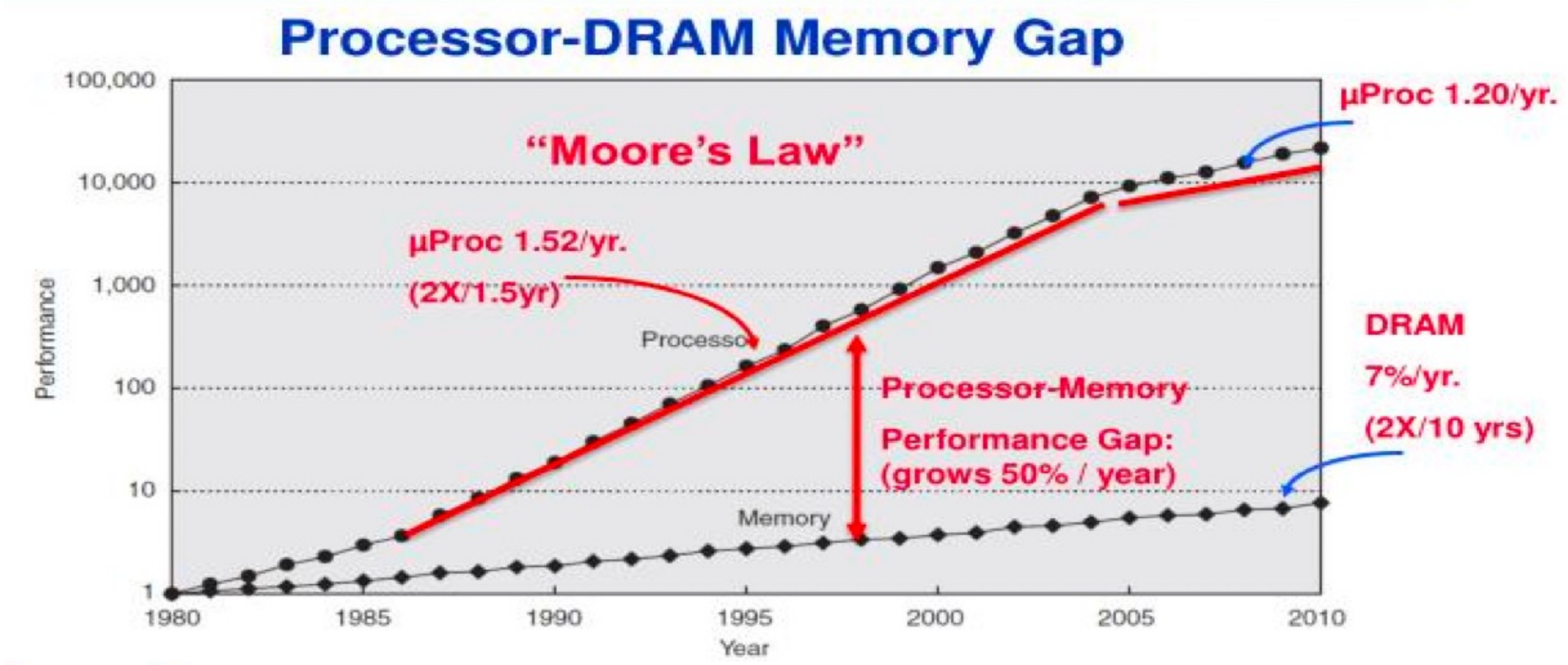
(B) Memory hierarchy for a laptop or a desktop



(C) Memory hierarchy for server

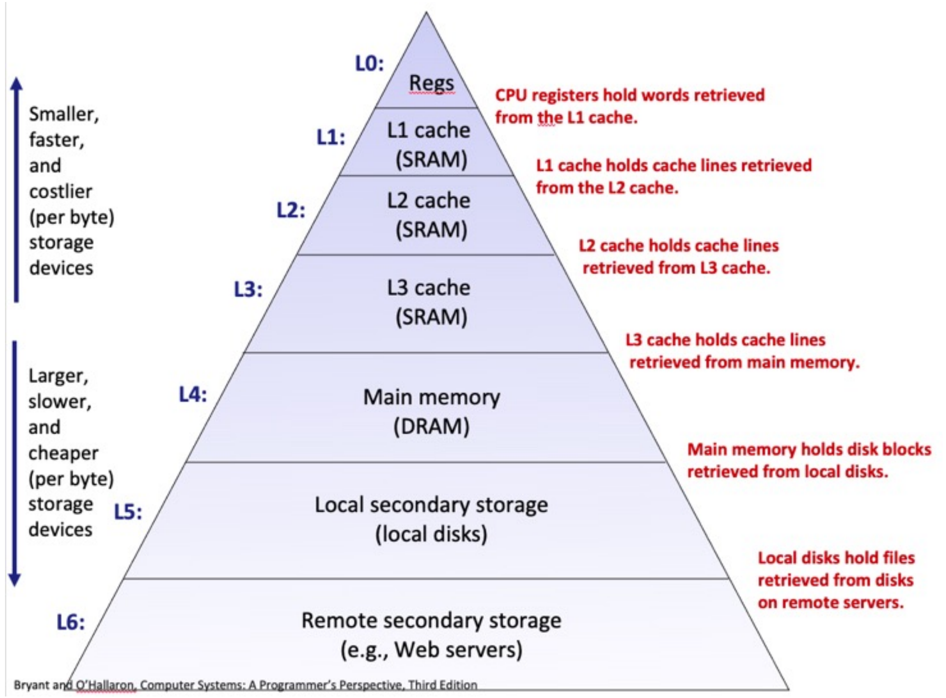
Memory Wall[存储墙]

- On modern machines, most programs that access a lot of data are <u>memory bound</u>
  - Latency of DRAM access is roughly 100-1000 cycles
  - Involves both the limited capacity and the bandwidth of memory transfer



Deeper Hierarchy[更深层级]

- 1980: no cache in micro-processor
- 1989: Intel 486 processor with 8KB on-chip L1 cache
- 1995: Intel Pentium Proc with 256KB on-chip L2 cache
- 2003: Intel Itanium 2 with 6MB on-chip L3 cache
- 2010: 3-level cache on chip, 4th-level cache off chip


Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0:
Regs
L1:
L1 cache (SRAM)
L2:
L2 cache (SRAM)
L3:
L3 cache (SRAM)
L4:
Main memory (DRAM)
L5:
Local secondary storage (local disks)
L6:
Remote secondary storage (e.g., Web servers)
CPU registers hold words retrieved from the L1 cache.
L1 cache holds cache lines retrieved from the L2 cache.
L2 cache holds cache lines retrieved from L3 cache.
L3 cache holds cache lines retrieved from main memory.
Main memory holds disk blocks retrieved from local disks.
Local disks hold files retrieved from disks on remote servers.
Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Memory Locality[局部性]

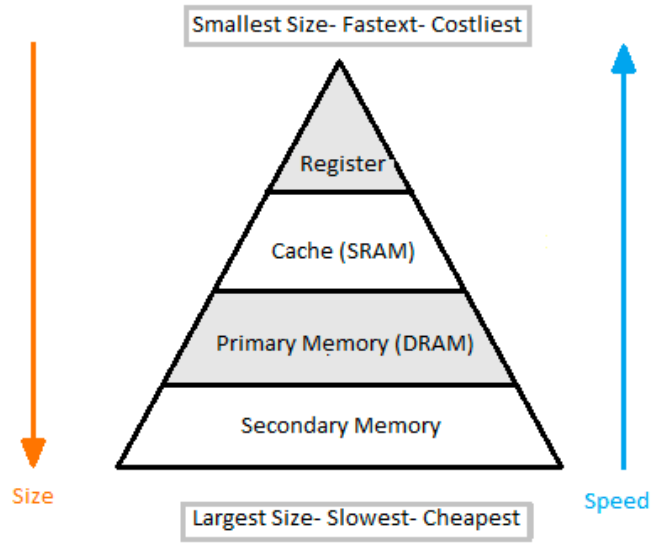
- A "typical" program has a lot of locality in memory references
  - Typical programs are composed of "loops"
- **Temporal**[时间]: a program tends to reference the same memory location many times and all within a small window of time
- **Spatial**[空间]: a program tends to reference a cluster of memory locations at a time
  - Most notable examples:
    - Instruction memory references (sequential execution)
    - Array/data structure references (array traversal)

Caching: Exploit Locality[利用局部性]

- **Temporal**[时间]: recently accessed data will be again accessed in the near future
  - Idea: store recently accessed data in automatically managed fast memory (called cache)
  - Anticipation: the data will be accessed again soon
- **Spatial**[空间]: nearby data in memory will be accessed in the near future (e.g., sequential instruction access, array traversal)
  - Idea: store addresses adjacent to the recently accessed one in automatically managed fast memory
    - logically divide memory into equal size blocks
    - Fetch to cache the accessed block in its entirety
  - Anticipation: nearby data will be accessed soon

Management[管理]

- Q1: Where can a block be placed in the upper level?
  - *(Block placement)*
- Q2: How is a block found if it is in the upper level?
  - *(Block identification)*
- Q3: Which block should be replaced on a miss?
  - *(Block replacement)*
- Q4: What happens on a write?
  - *(Write strategy)*



Management Policies[策略]

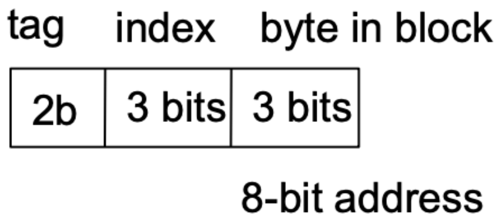
- Manual[手动]: programmer manages data movement across levels
- -- too painful for programmers on substantial programs
  - "core" vs "drum" memory in the 50's
  - still done in some embedded processors (on-chip scratch pad SRAM in lieu of a cache)
- Automatic[自动]: hardware manages data movement across levels, transparently to the programmer
- ++ programmer's life is easier
  - the average programmer doesn't need to know about it
    - You don't need to know how big the cache is and how it works to write a "correct" program! (What if you want a "fast" program?)

Cache Basics[缓存基础]

- **Block** (line): unit of storage in the cache[缓存单位]
  - Memory is logically divided into cache blocks that map to locations in the cache
- When data referenced[使用]
  - HIT: if in cache, use cached data instead of accessing memory
  - MISS: if not in cache, bring block into cache
    - Maybe have to kick something else out to do it
- Some important cache design decisions
  - Placement[放置]: where and how to place/find a block in cache?
  - Replacement[替换]: what data to remove to make room in cache?
  - Granularity of management[粒度]: large, small, uniform blocks?
  - Write policy[写策略]: what do we do about writes?
  - Instructions/data[指令/数据]: do we treat them separately?

Cache Basics (cont.)

- Memory is logically divided into fixed-size blocks
- Each block maps to a location in the cache, determined by the index bits in the address
  - Used to index into the **tag and data stores**

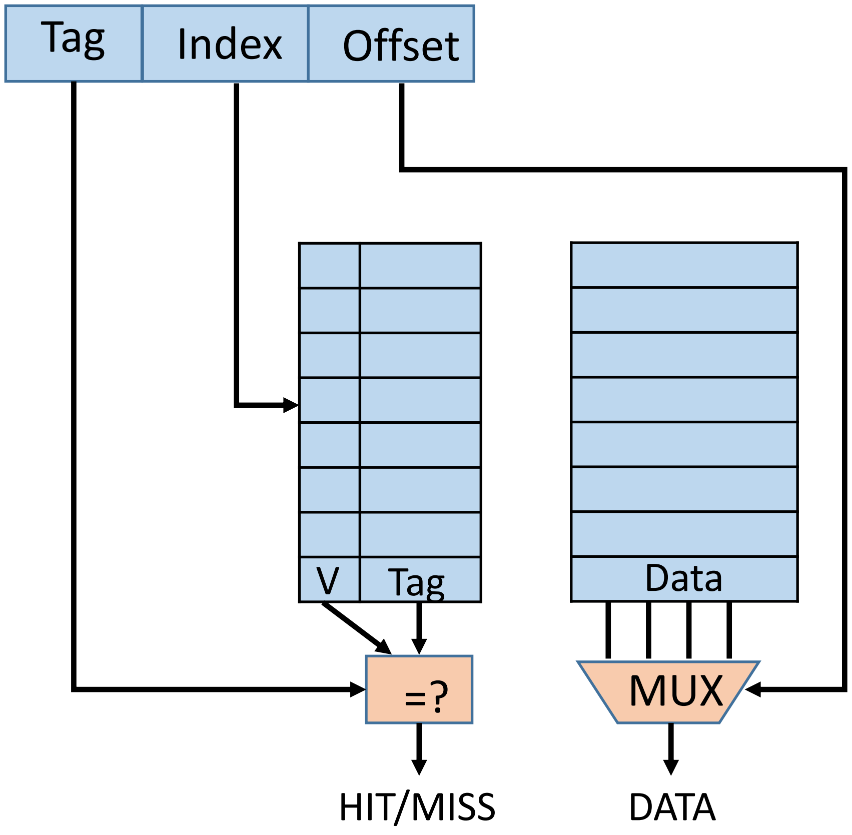


- Cache access steps
  - 1) index into the tag and data stores with index bits in address
  - 2) check valid bit in tag store
  - 3) compare tag bits in address with the stored tag in tag store
- If a block is in the cache (cache hit), the stored tag should be valid and match the tag of the block

Cache Basics (cont.)

- Assume byte-addressable memory
  - Capacity: 256 bytes → 8-bit address
  - Block: 8 bytes → 3-bit offset
  - #blocks: 32 (256/8)
- Assume cache
  - Capacity: 64 bytes → 3-bit index
    - Holding 8 blocks (64/8)
- What is a tag store?
  - Tag
  - Metadata
    - Valid bit
    - Replacement policy bits
    - Dirty bit
    - ECC

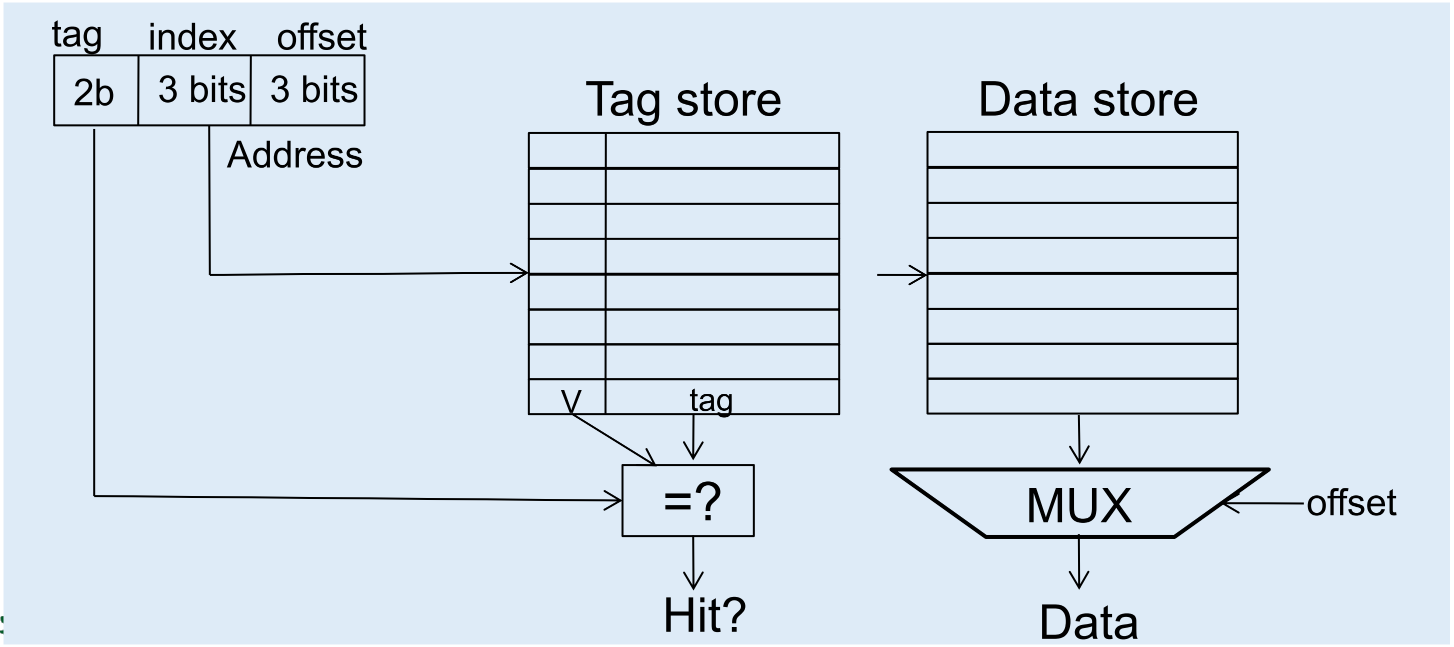
tag	index	byte in block
2b	3 bits	3 bits



MUX: multiplexer (数据选择器)

Direct Mapped[直接映射]

- For each item (block) of data in memory, there is **exactly one** location in the cache where it might be
- Two blocks in memory that map to the same index in the cache cannot be present in the cache at the same time
  - Addresses A/B have the same index bits but different tag bits
  - A, B, A, B, A, B, A, B, … → all misses


tag
index
offset
2b
3 bits
3 bits
Address
Tag store
Data store
V
tag
=?
MUX
offset
Hit?
Data