CprE 381: Computer Organization and Assembly Level Programming

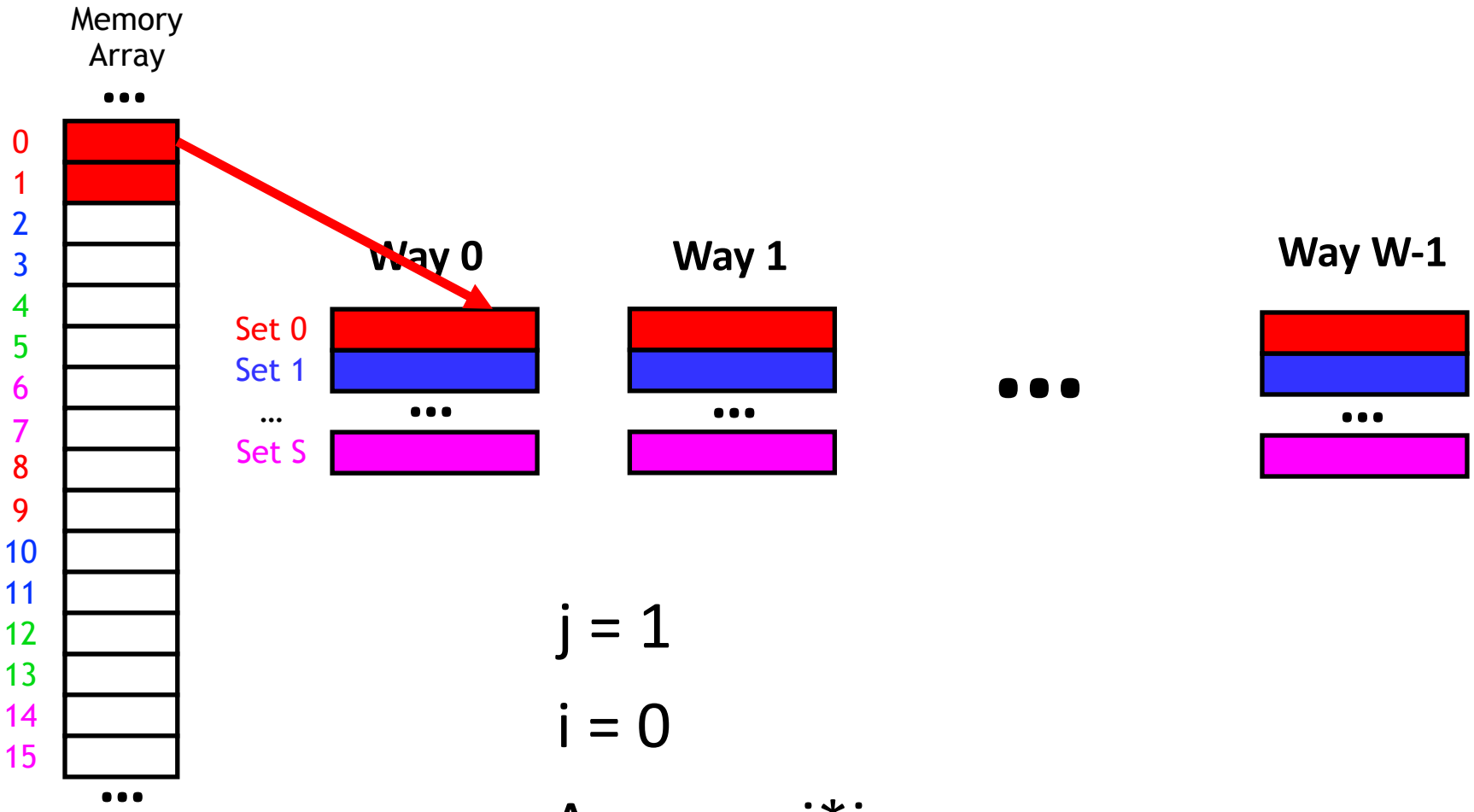
GPU Architectures

Henry Duwe
Electrical and Computer Engineering
Iowa State University

Administrative

- Part 4 due in lab this week
  - WARNING: No extensions!!!
- Final Exam
  - When: Mon May 6 at 7:30am
  - Where: **Marston 2155 (everyone)**
  - What: Data Forwarding and Control Hazards through HW security
- Please take online course assessment

Review: HW11 Cache Test 1

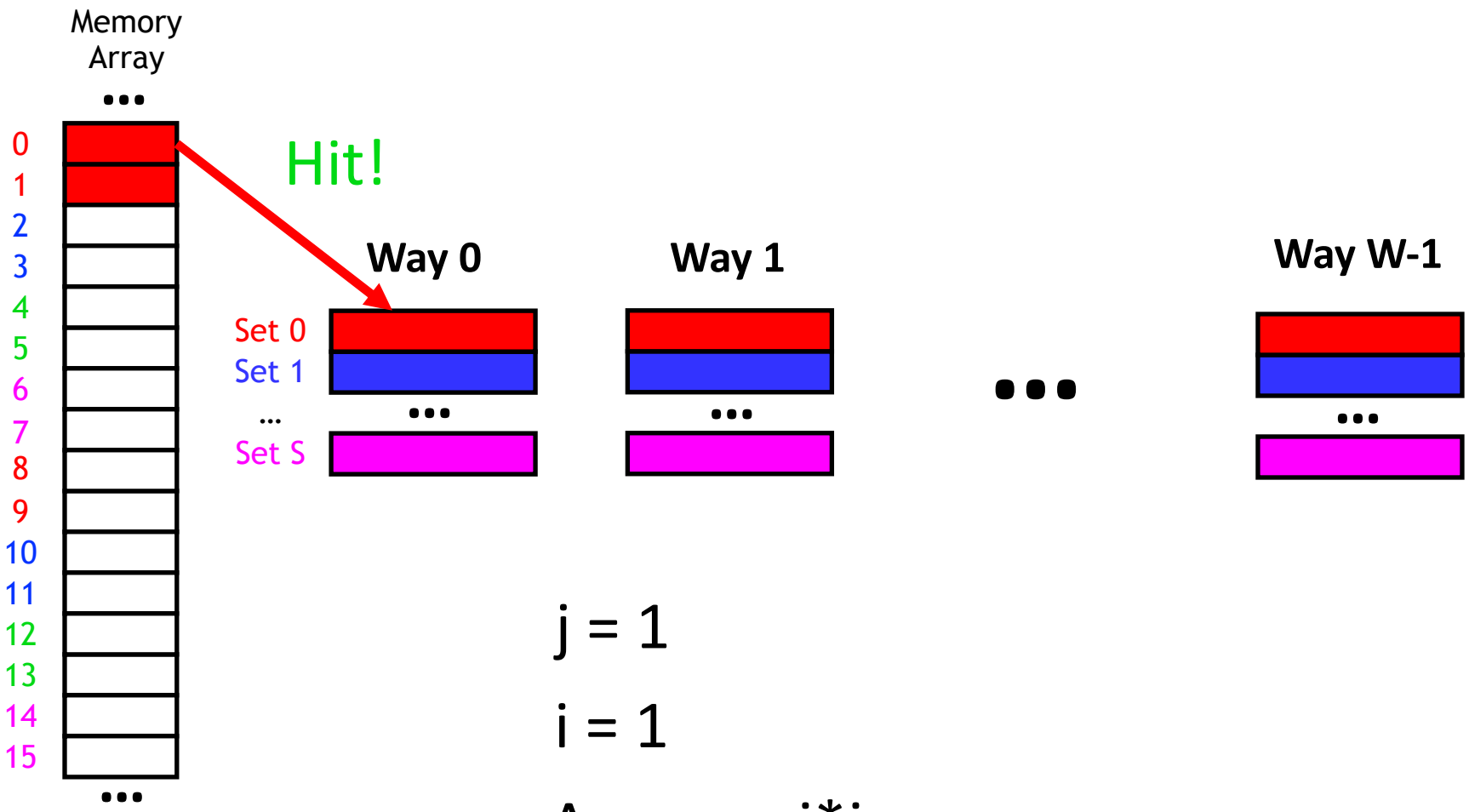


j = 1

i = 0

Access = i*j

Review: HW11 Cache Test 1

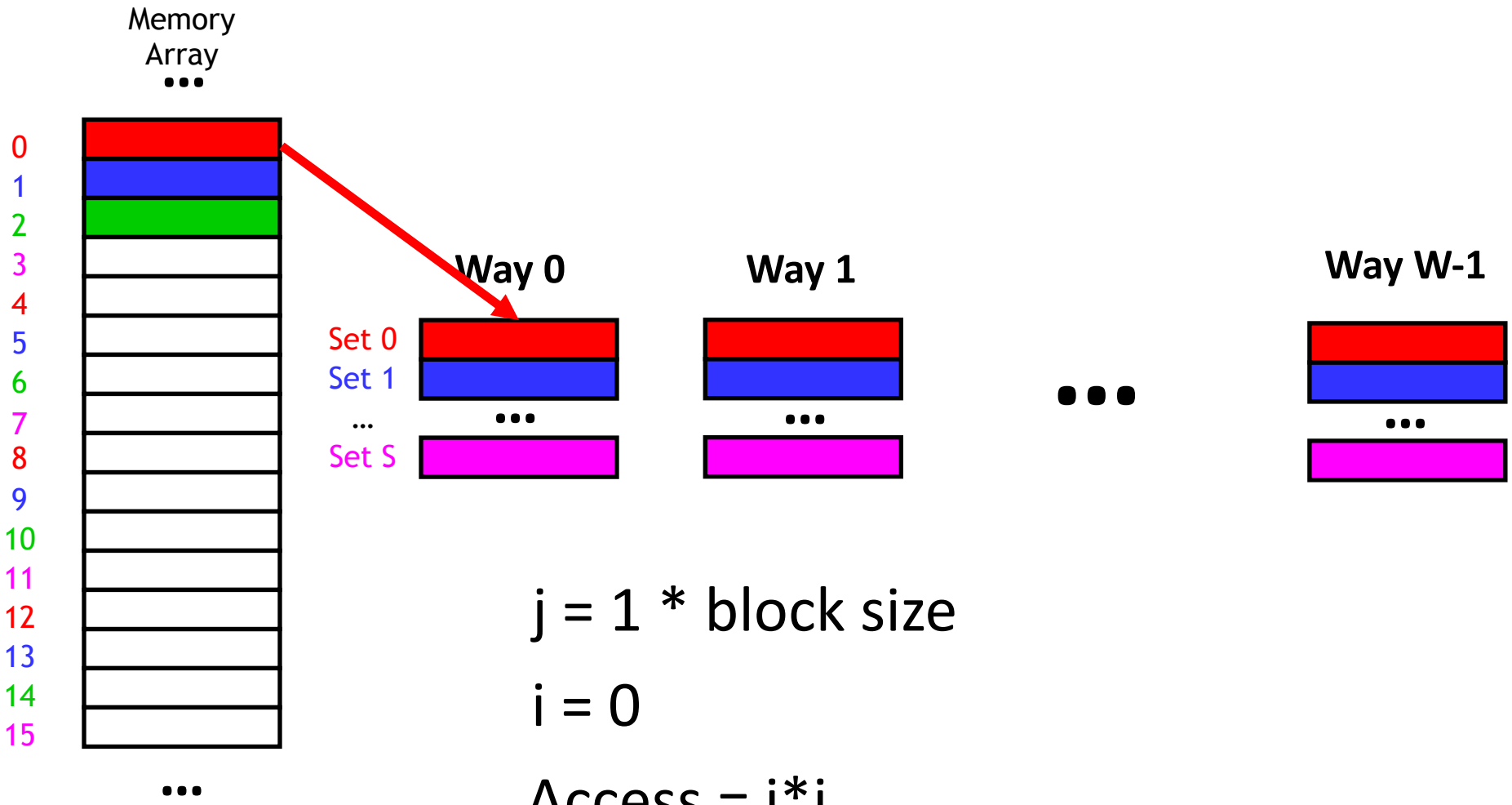


j = 1

i = 1

Access = i*j

Review: HW11 Cache Test 1

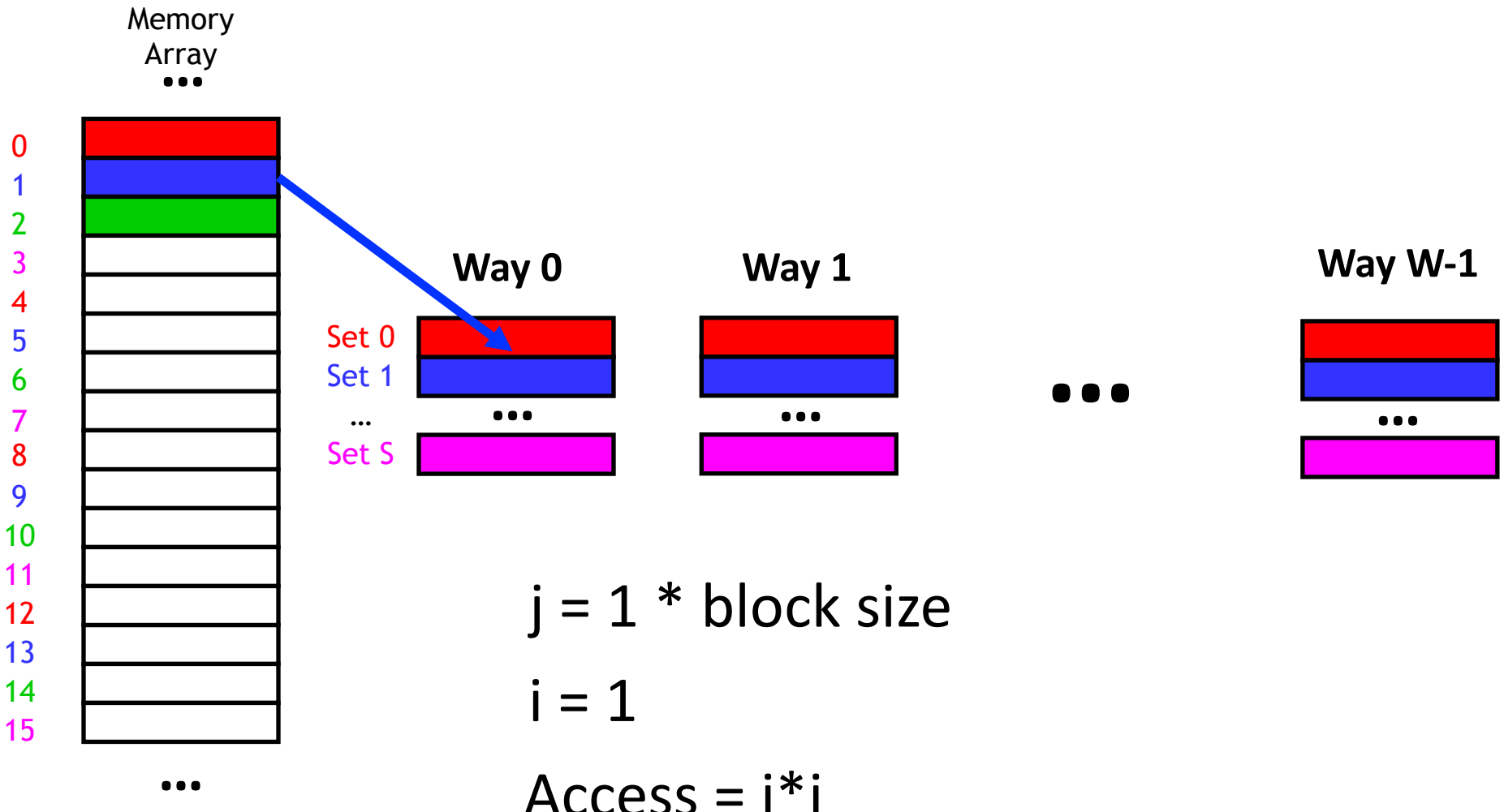


j = 1 * block size

i = 0

Access = i*j

Review: HW11 Cache Test 1

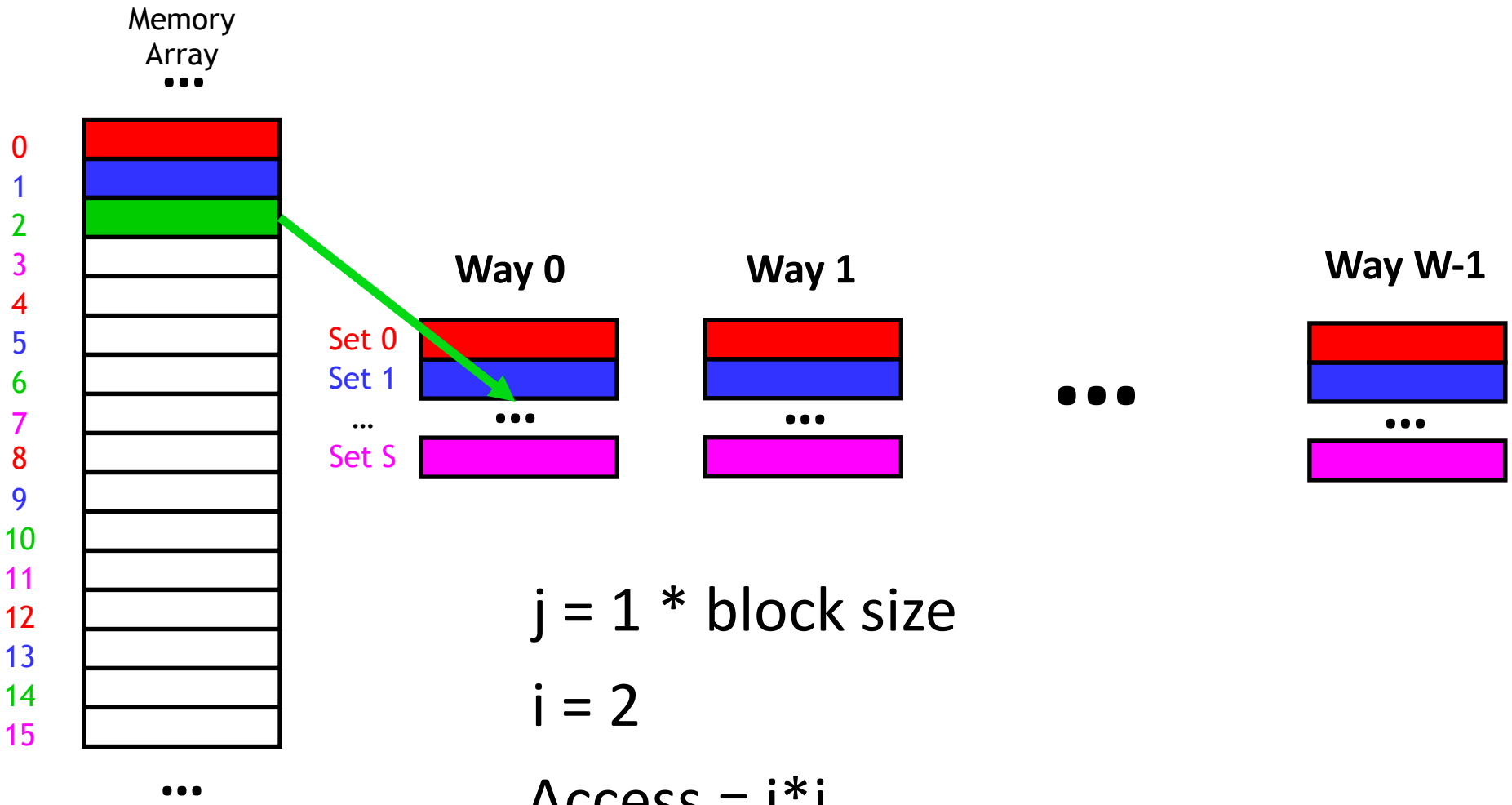


j = 1 * block size

i = 1

Access = i*j

Review: HW11 Cache Test 1

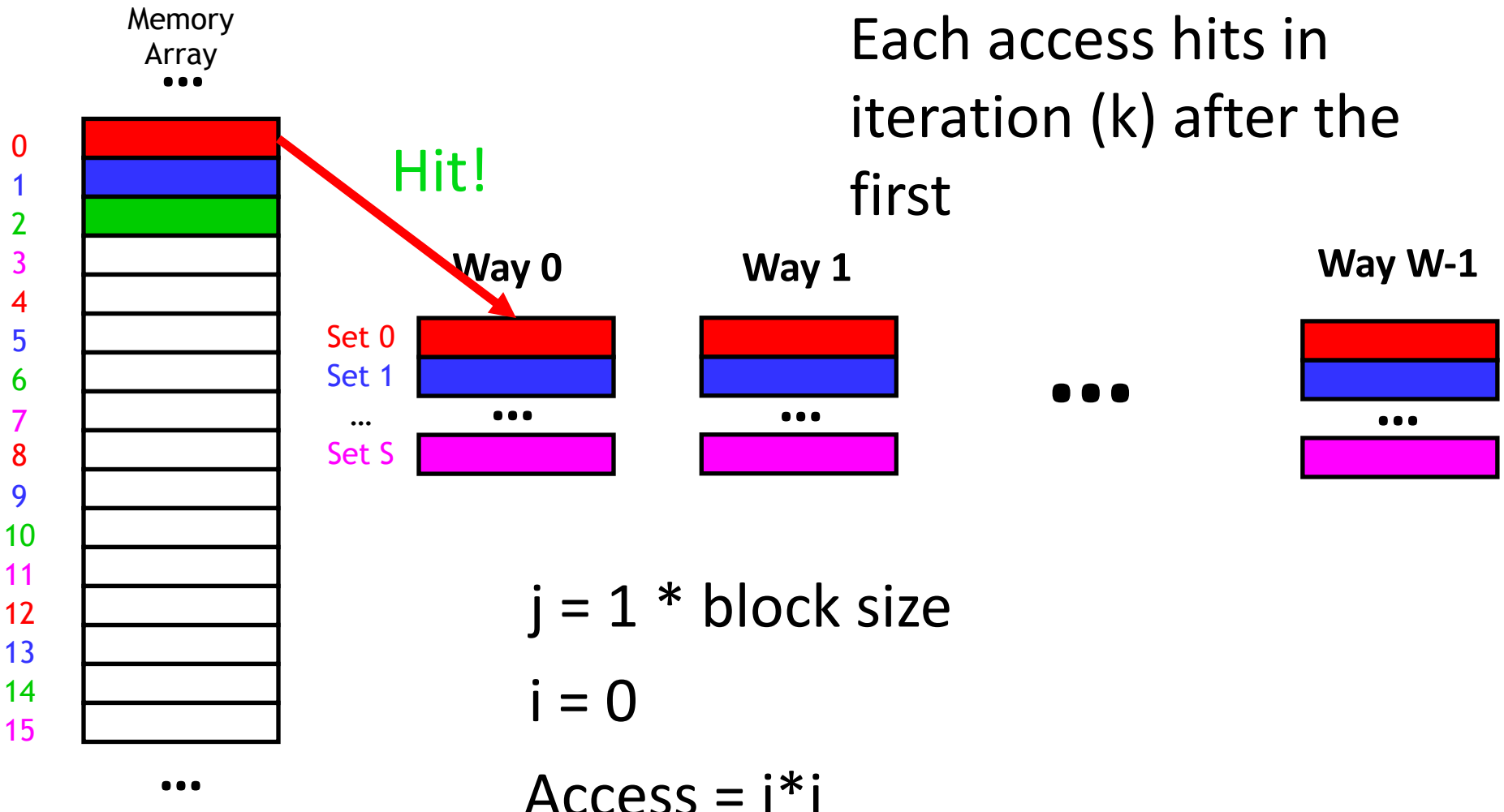


j = 1 * block size

i = 2

Access = i*j

Review: HW11 Cache Test 1



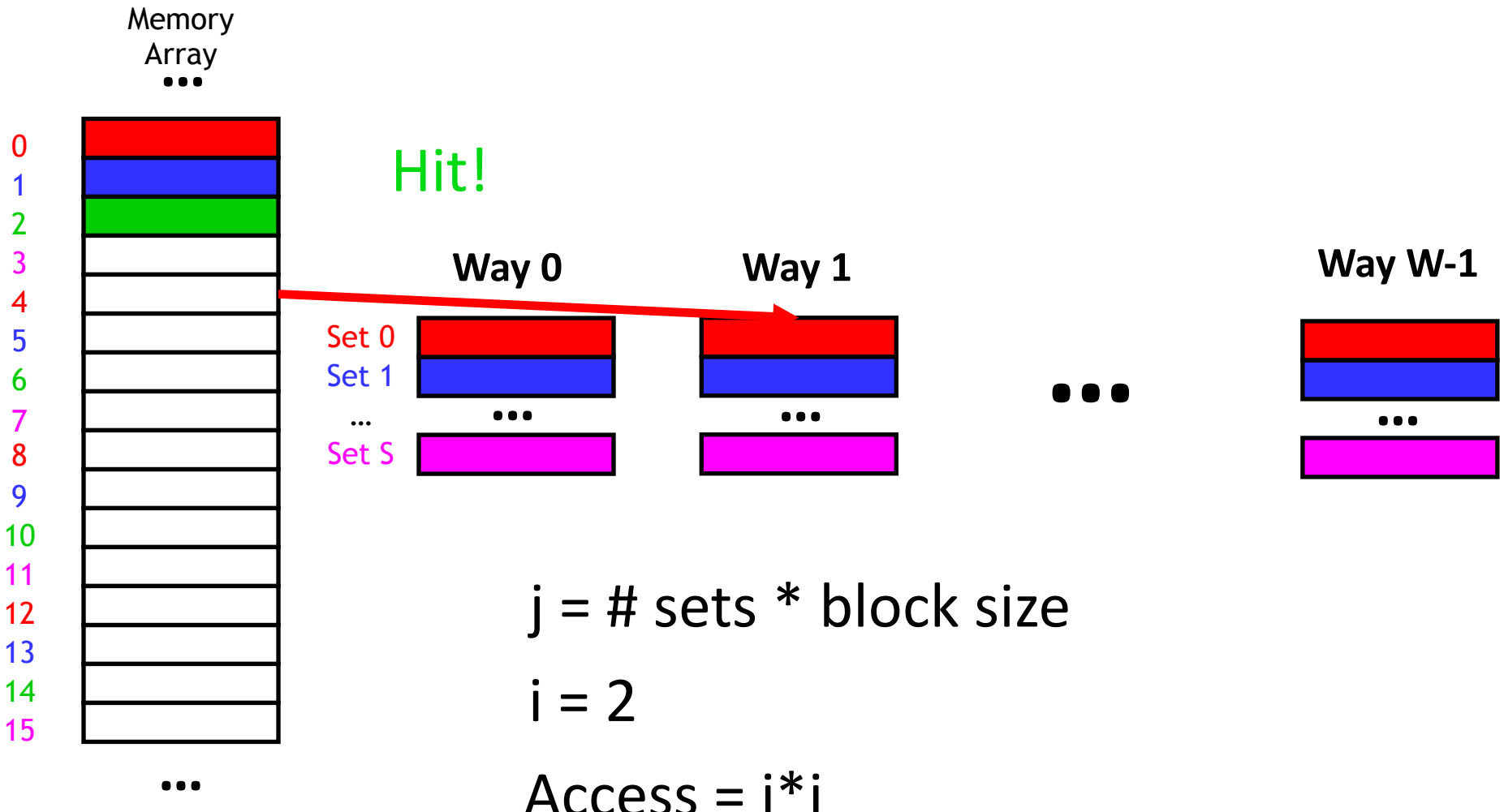
Each access hits in iteration (k) after the first

j = 1 * block size

i = 0

Access = i*j

Review: HW11 Cache Test 1

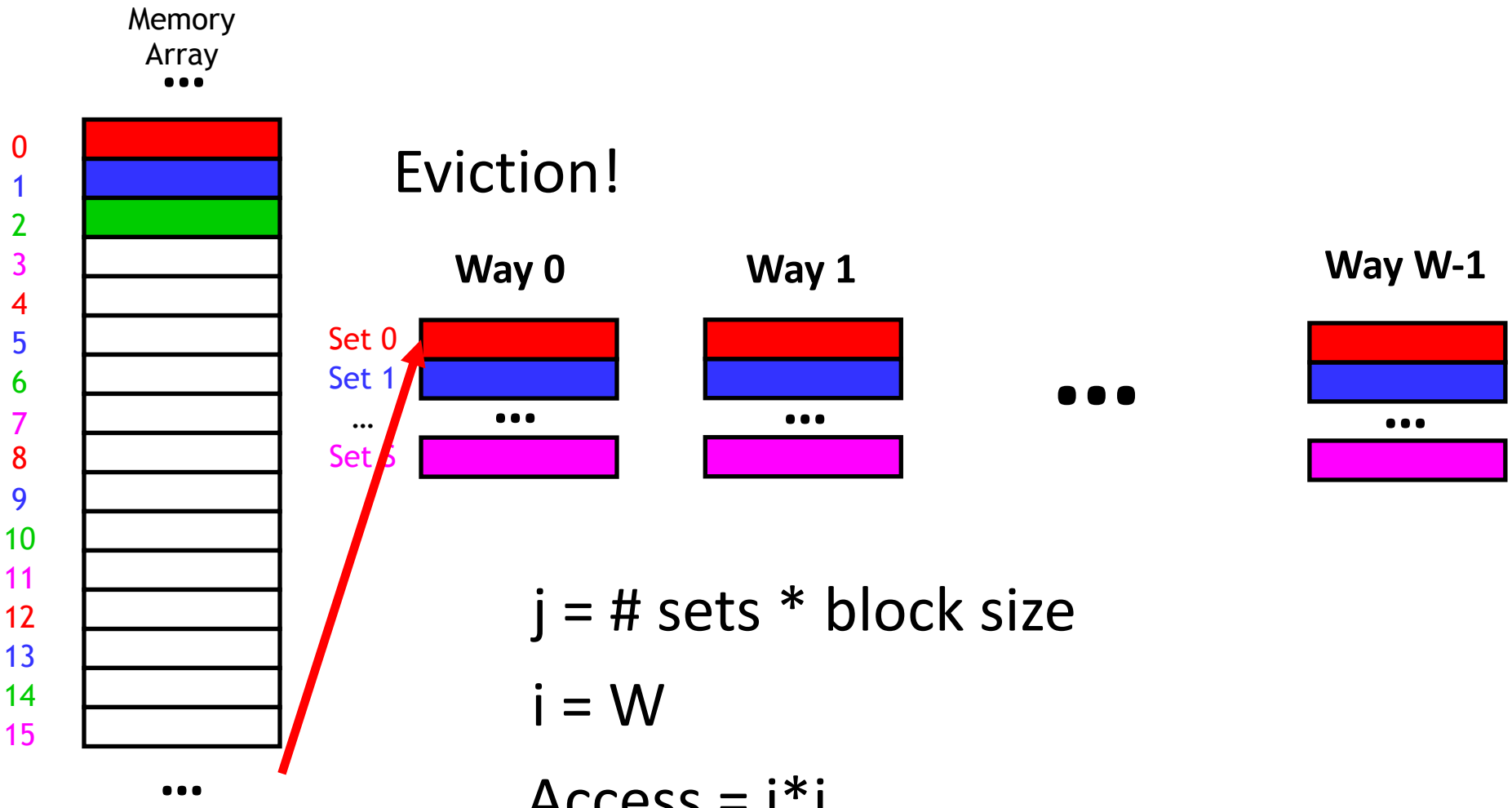


j = # sets * block size

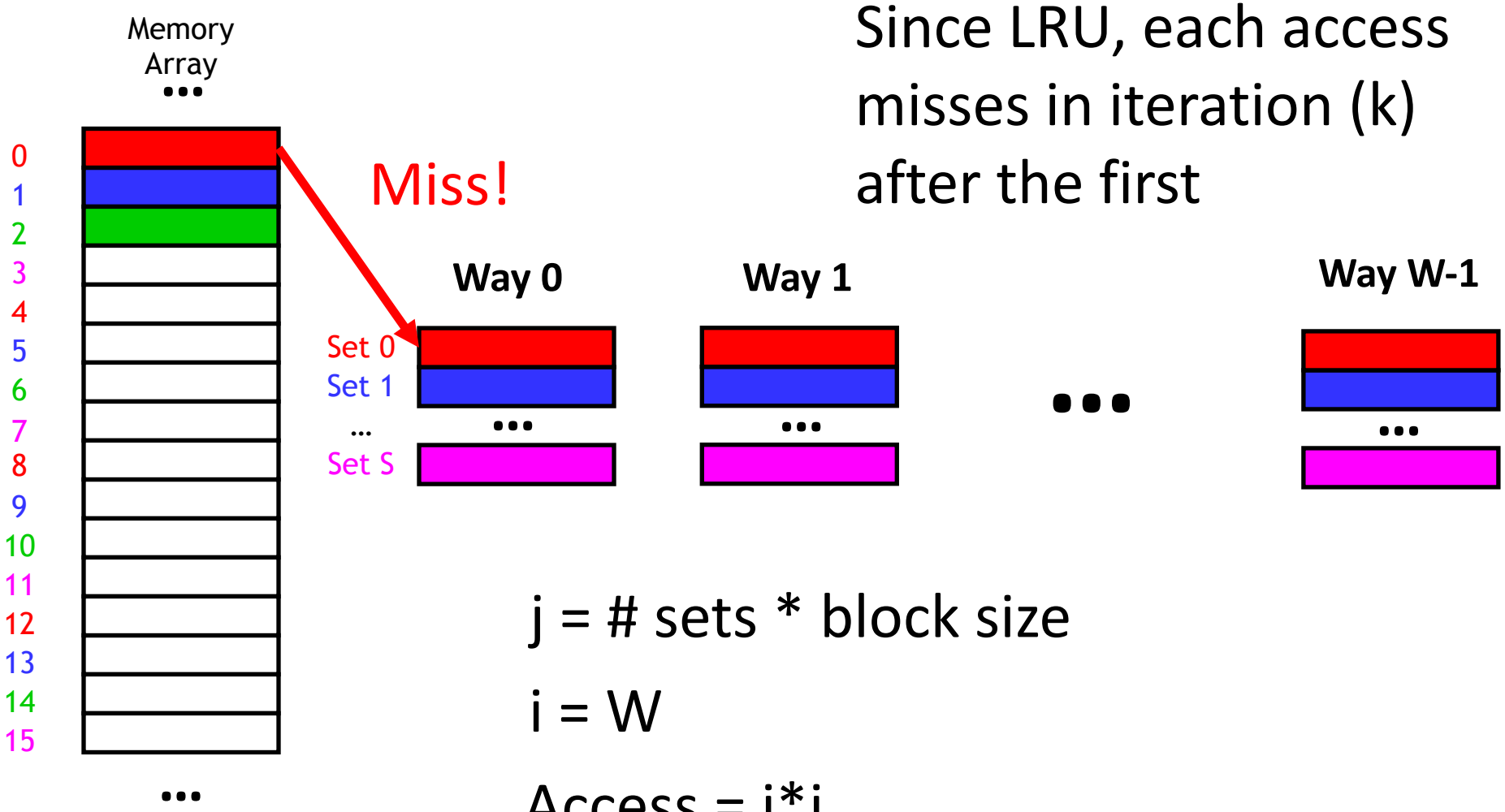
i = 2

Access = i*j

Review: HW11 Cache Test 1



Review: HW11 Cache Test 1



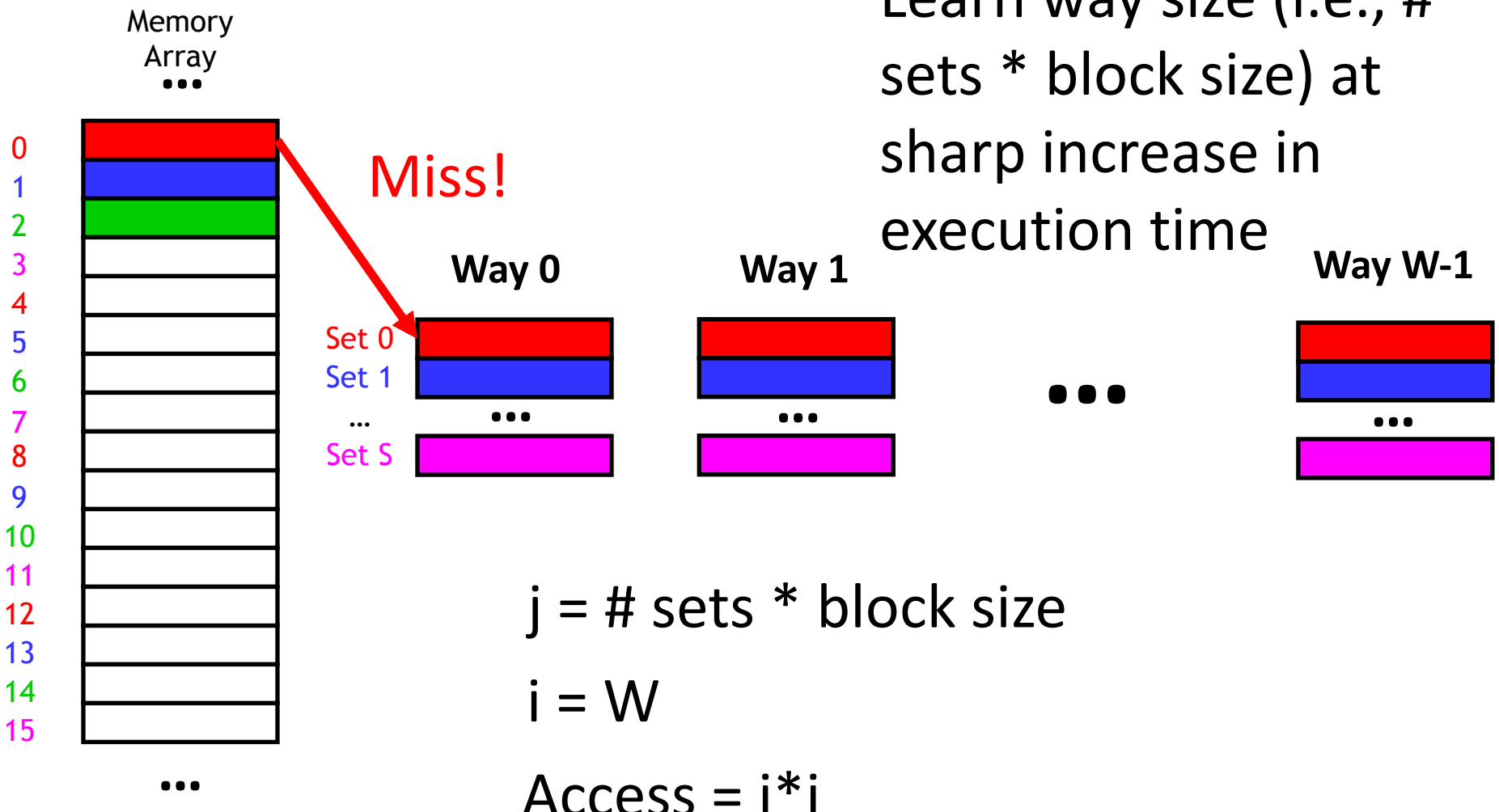
Since LRU, each access misses in iteration (k) after the first

j = # sets * block size

i = W

Access = i*j

Review: HW11 Cache Test 1



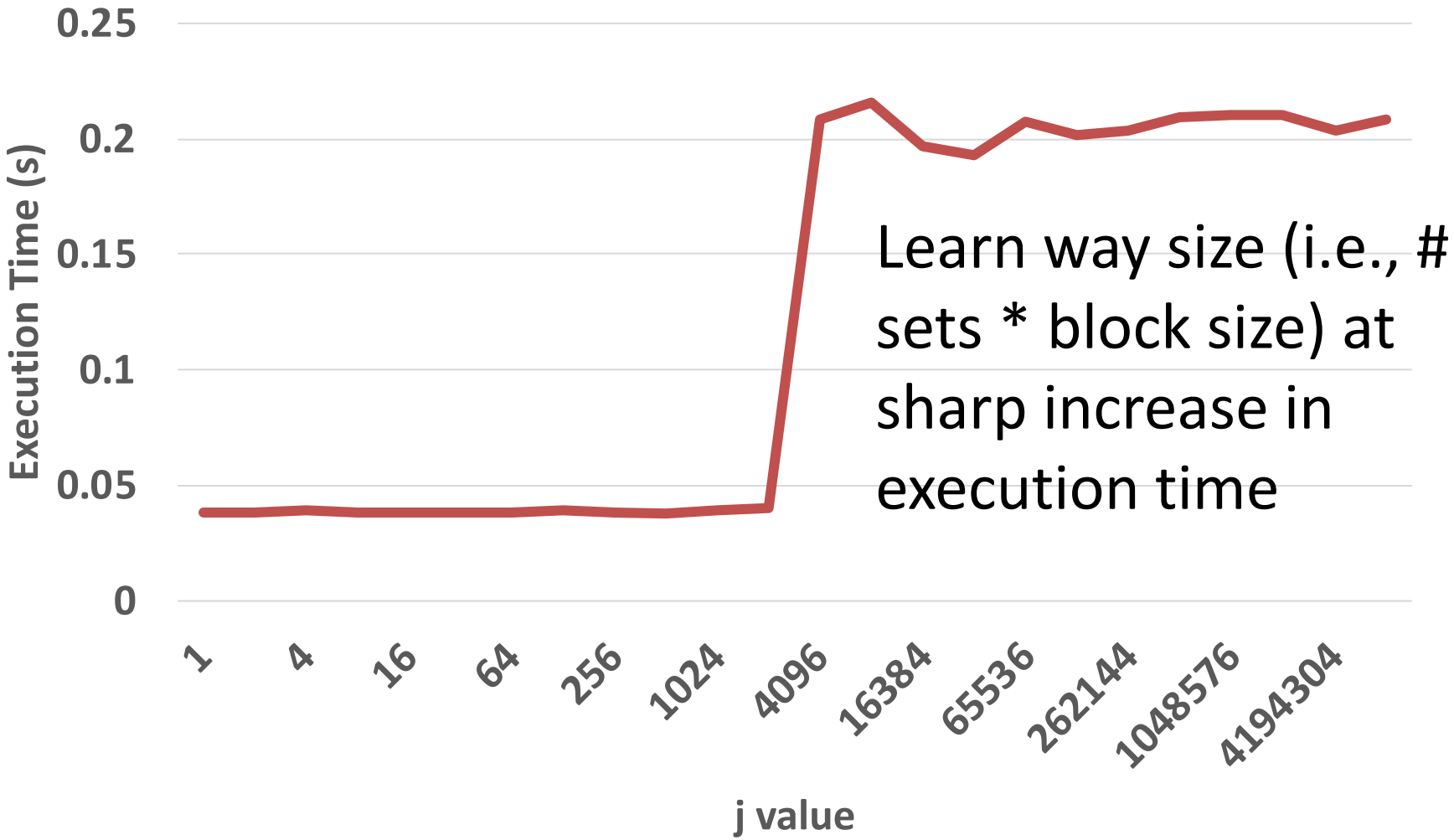
Learn way size (i.e., # sets * block size) at sharp increase in execution time

j = # sets * block size

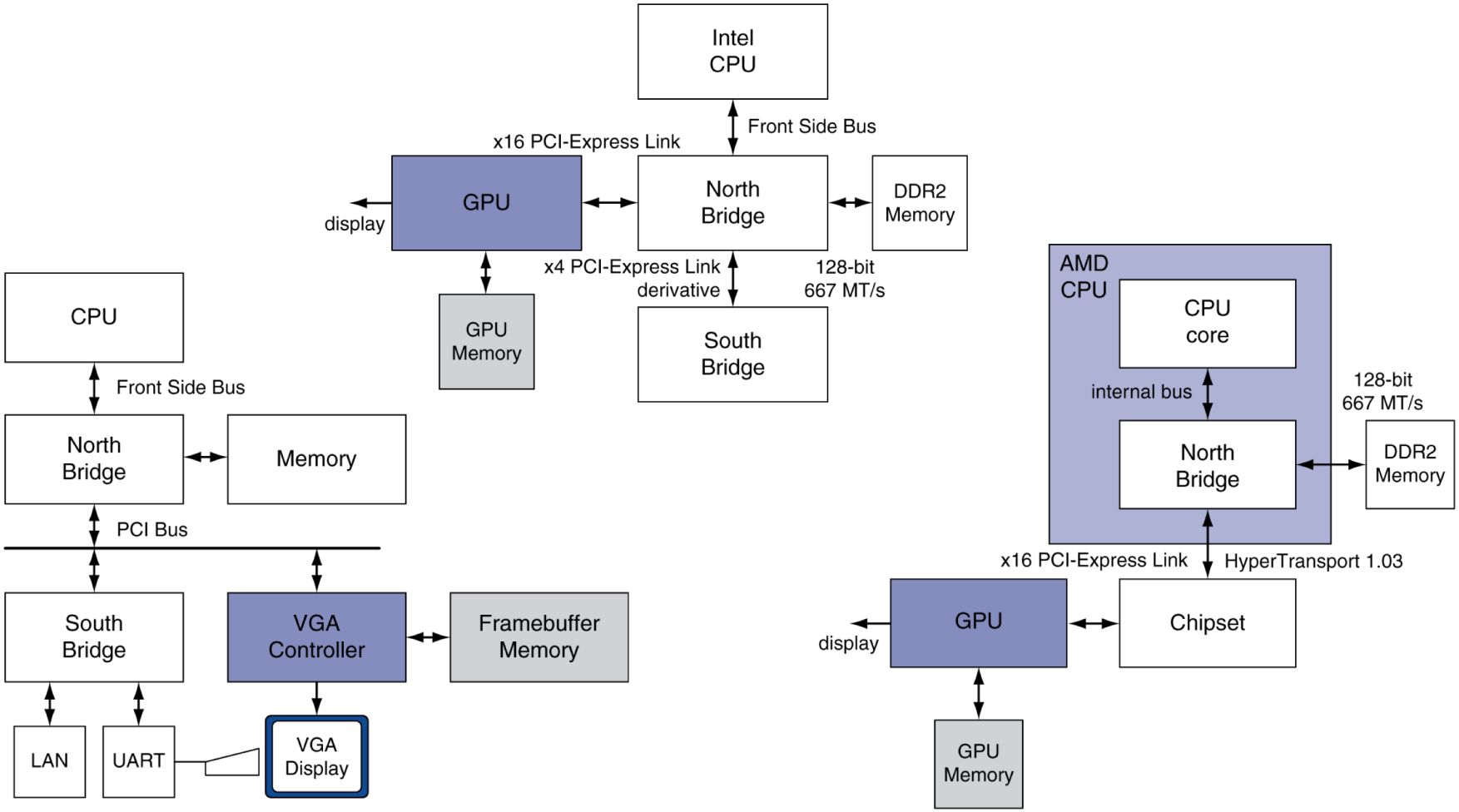
i = W

Access = i*j

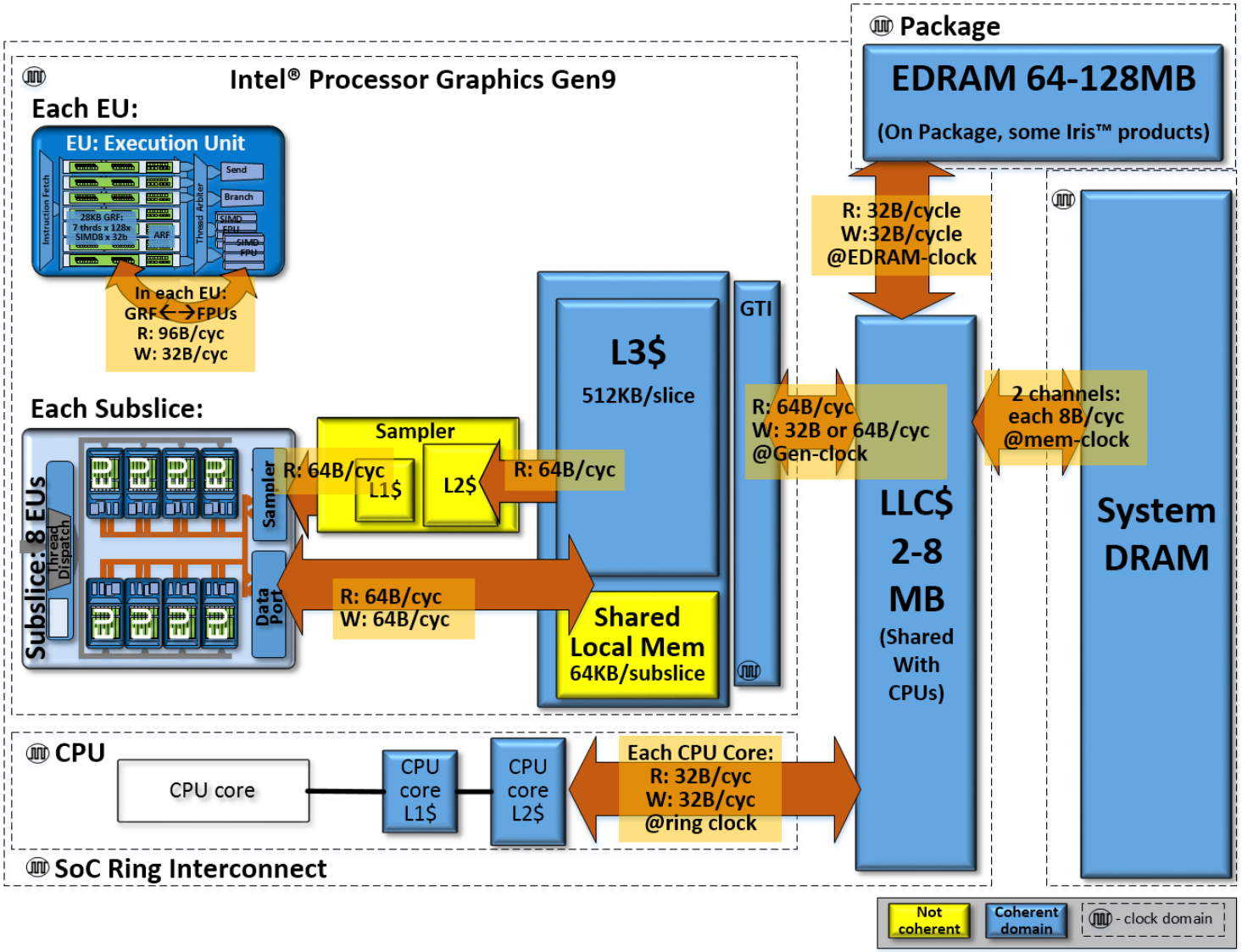
Review: HW11 Cache Test 1



Graphics in the System



Graphics in the System



GPU Architectures

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

Programming Interfaces

- Parallel Thread Execution (PTX) – Assembly
  - A low-level parallel thread execution virtual machine and instruction set architecture (ISA)
- High-level programming language – CUDA extension
  - Extensions to standard programming languages such as C/C++, Matlab, Python, and so on.

- Key concept:
  - All threads execute a single kernel in parallel (defined using PTX or CUDA extensions)
  - 1000s of threads are required for full utilization
  - “Free” context switching between threads

Kernels and Threads

- Parallel programs are executed as *kernels* on the GPU (also referred to as the *device*)
  - Only one kernel can execute at a time
  - All CUDA *threads* execute the same kernel code
    - A kernel is written from a single thread's perspective
- CUDA thread management
  - No thread swapping penalties
  - Supports 1000s of threads
    - Uses them to increase efficiency
  - Every thread has an ID automatically assigned

Kernels and Threads

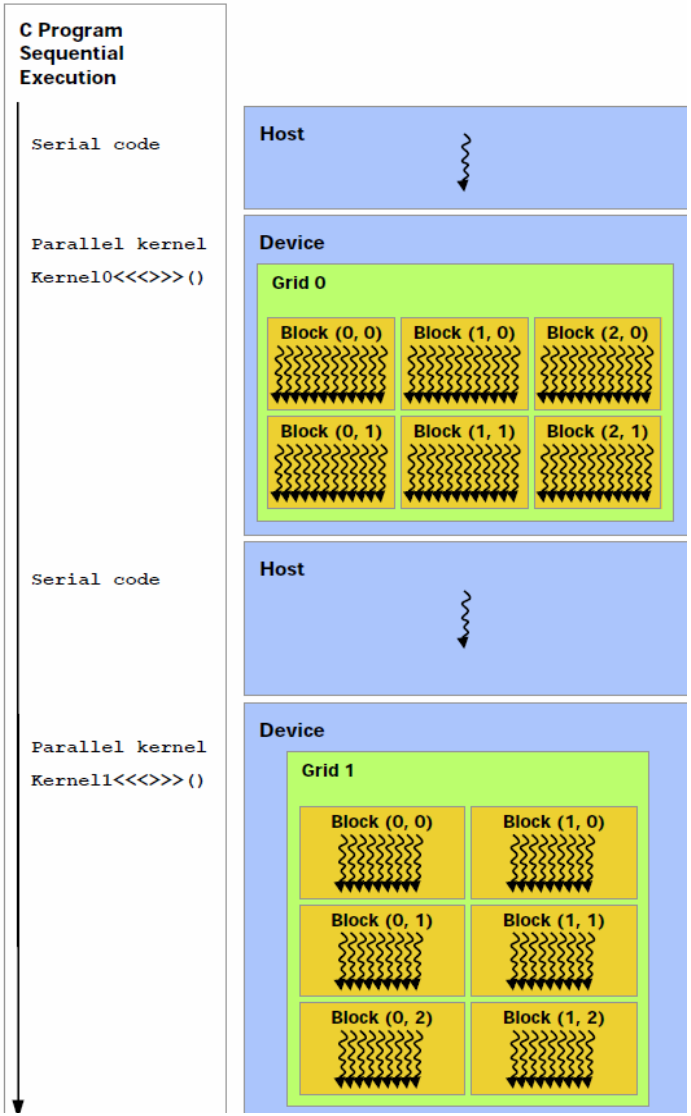
- Parallel programs are executed as *kernels* on the GPU (also referred to as the *device*)
  - Only one kernel can execute at a time
- **In-class Assessment!**
- **Access Code: Needle**
- **Note: sharing access code to those outside of classroom or using access code while outside of classroom is considered cheating**
  - Supports 1000s of threads
    - Uses them to increase efficiency
  - Every thread has an ID automatically assigned

Kernels and Threads

- Parallel programs are executed as *kernels* on the GPU (also referred to as the *device*)
  - Only one kernel can execute at a time
  - All CUDA *threads* execute the same kernel code
    - A kernel is written from a single thread's perspective
- CUDA thread management
  - No thread swapping penalties
  - Supports 1000s of threads
    - Uses them to increase efficiency
  - Every thread has an ID automatically assigned

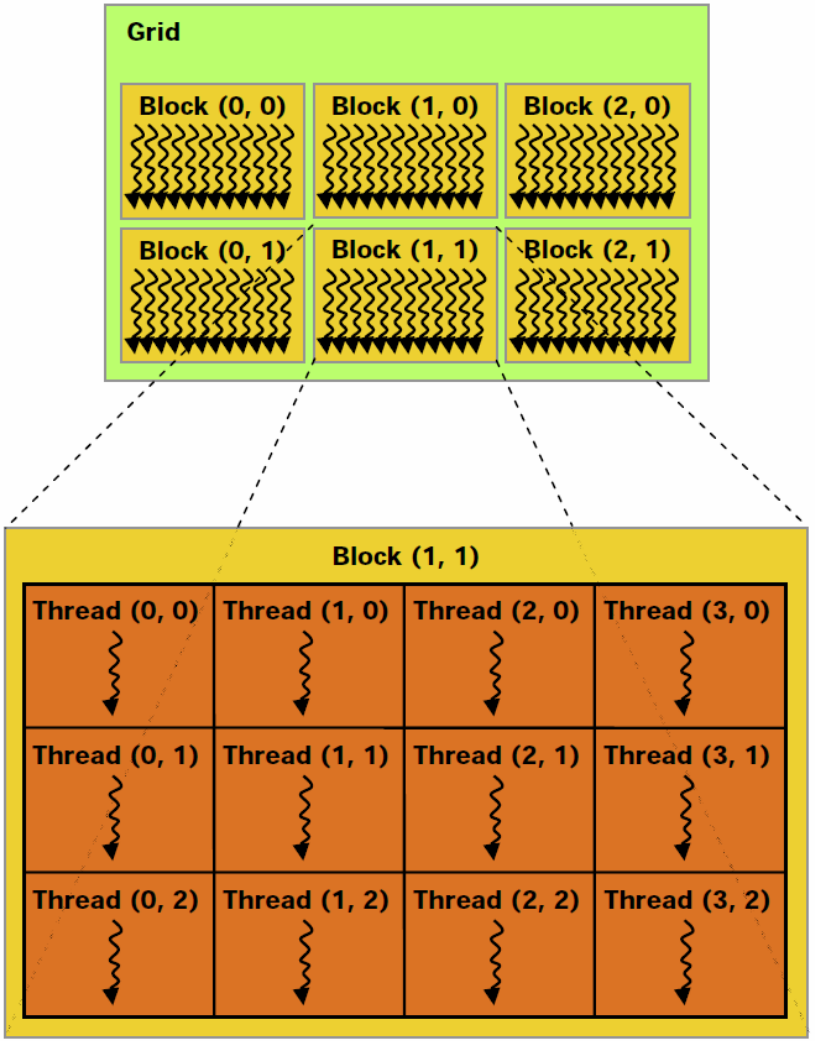
Scalability

- For cooperation between all threads
  - Multiple kernel launches
  - Only global memory is preserved between kernel launches

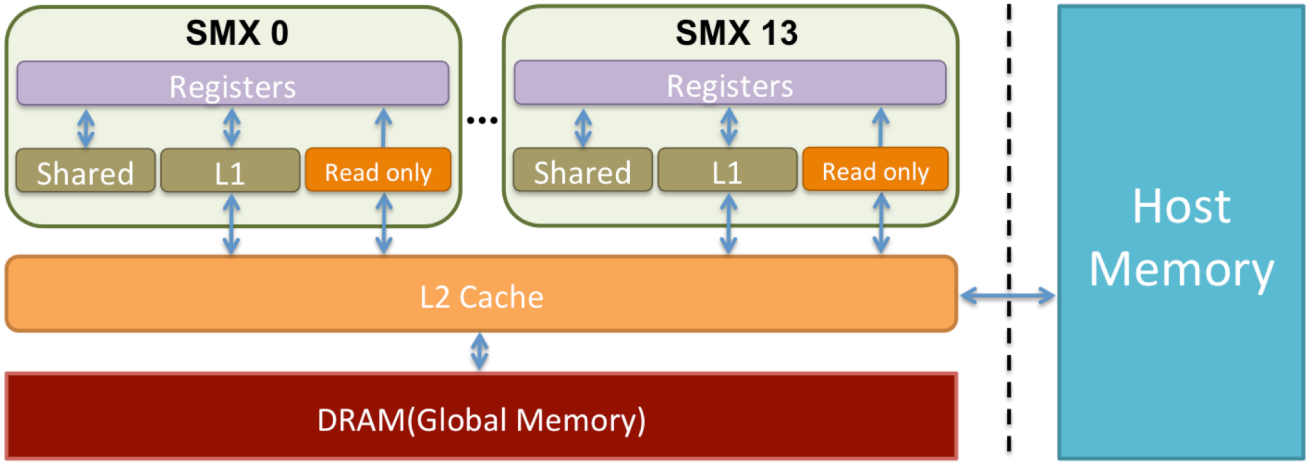


Thread Organization

- Thread hierarchy
  - An array of threads are formed in a *block*
    - Thread arrays can be 3-Dimensional
  - Blocks are arranged in a *grid*
    - Grids can be 3-Dimensional
    - All blocks in a grid have the same number of threads
  - Only one grid can be created per kernel execution
- How would you organize 1024 threads?

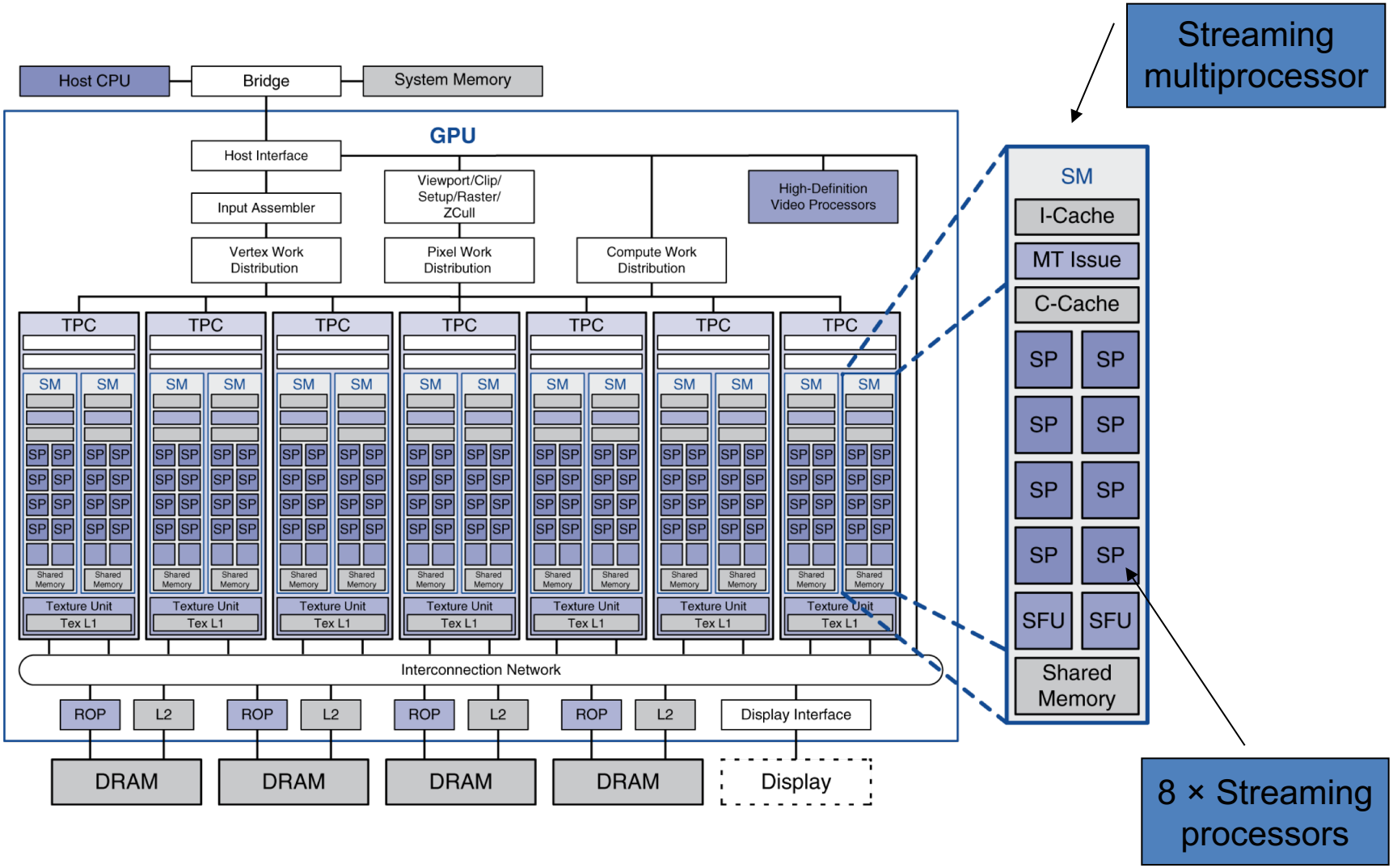


GK110 Architecture



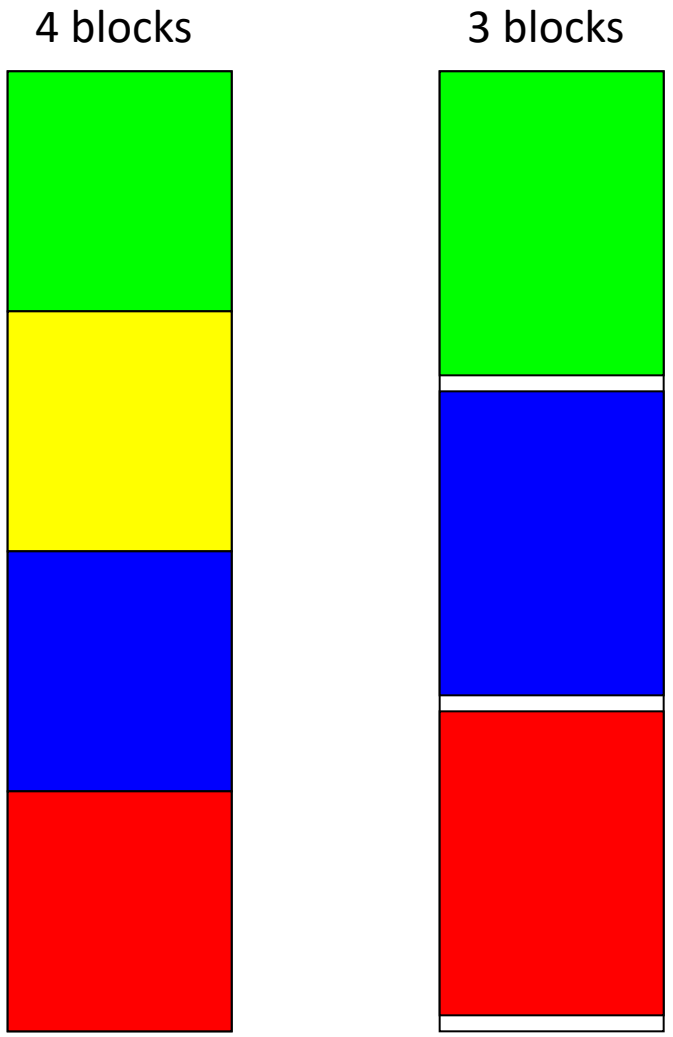
- Single Instruction Multi-Thread (SIMT)
  - Different then SIMD
  - Different then MIMD
- Threads are assigned to a specific streaming processor
- Large register count
  - 65,536 registers per SM
- 16, 32 or 48 KB shared memory

NVIDIA Tesla Architecture



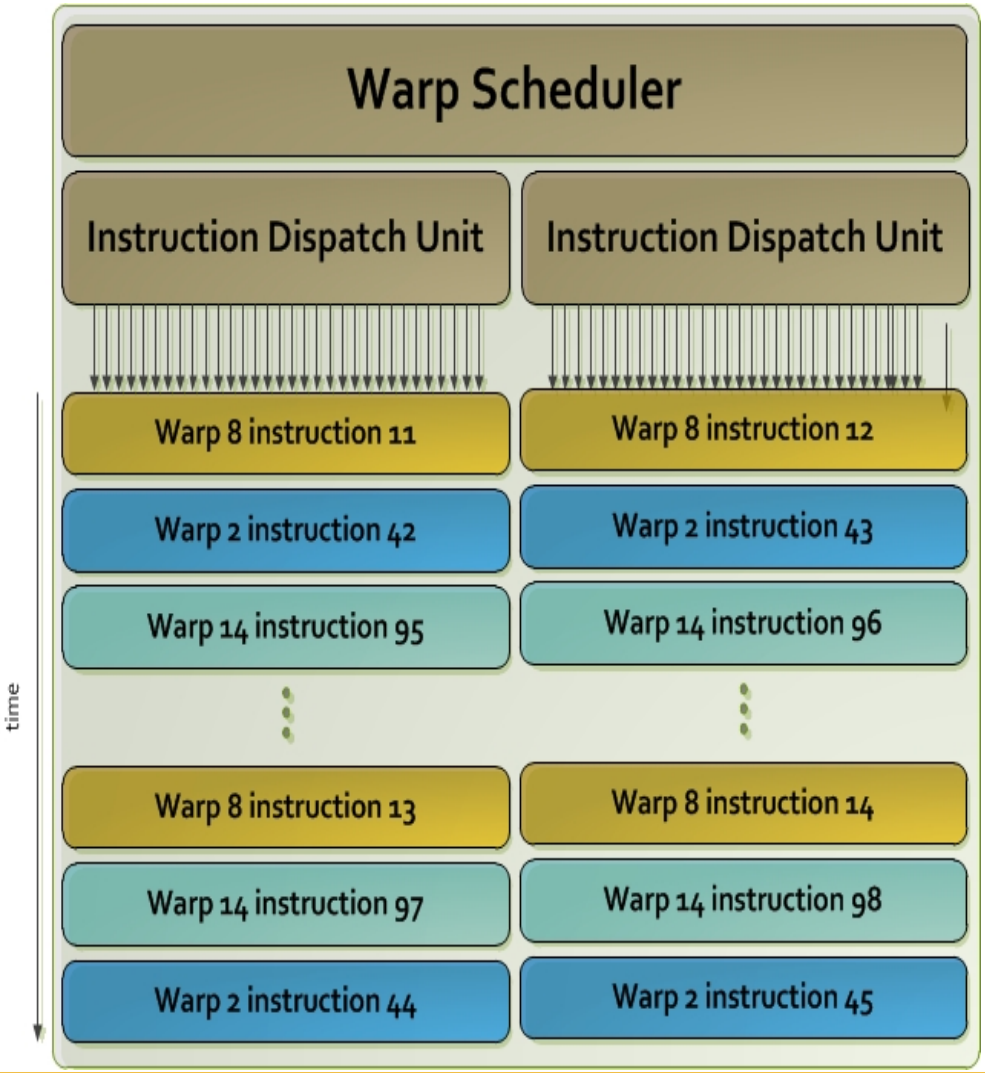
Programmer's View of Reg. File

- There are 65,536 registers in each SM in the GK110
  - This is an implementation decision, not part of CUDA
  - Registers are dynamically partitioned across all blocks assigned to the SM
  - Once assigned to a block, the register is NOT accessible by threads in other blocks
  - Each thread in the same block only access registers assigned to itself
  - One of the limiting resources



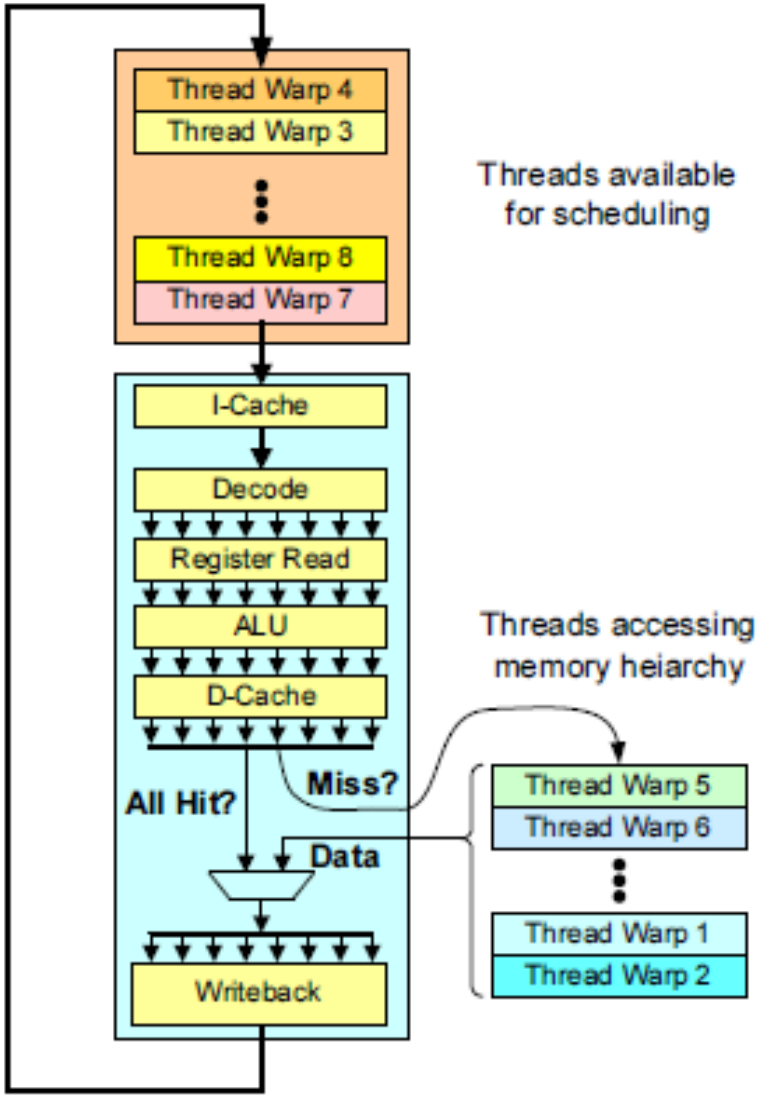
Scheduling

- Blocks are statically scheduled to a SM at runtime
- Threads are grouped into *warps* (32 threads)
  - Warps run in lock-step (one instruction for all threads)
- Every cycle the warp scheduler selects 4 warps and two independent instructions per warp can be dispatched to SIMD pipeline



Hiding Memory Latency

- Global memory reads can take 400 to 600 cycles
- Remove warp from available warps until memory request has finished
  - Hides entire memory latency if there is enough available warps

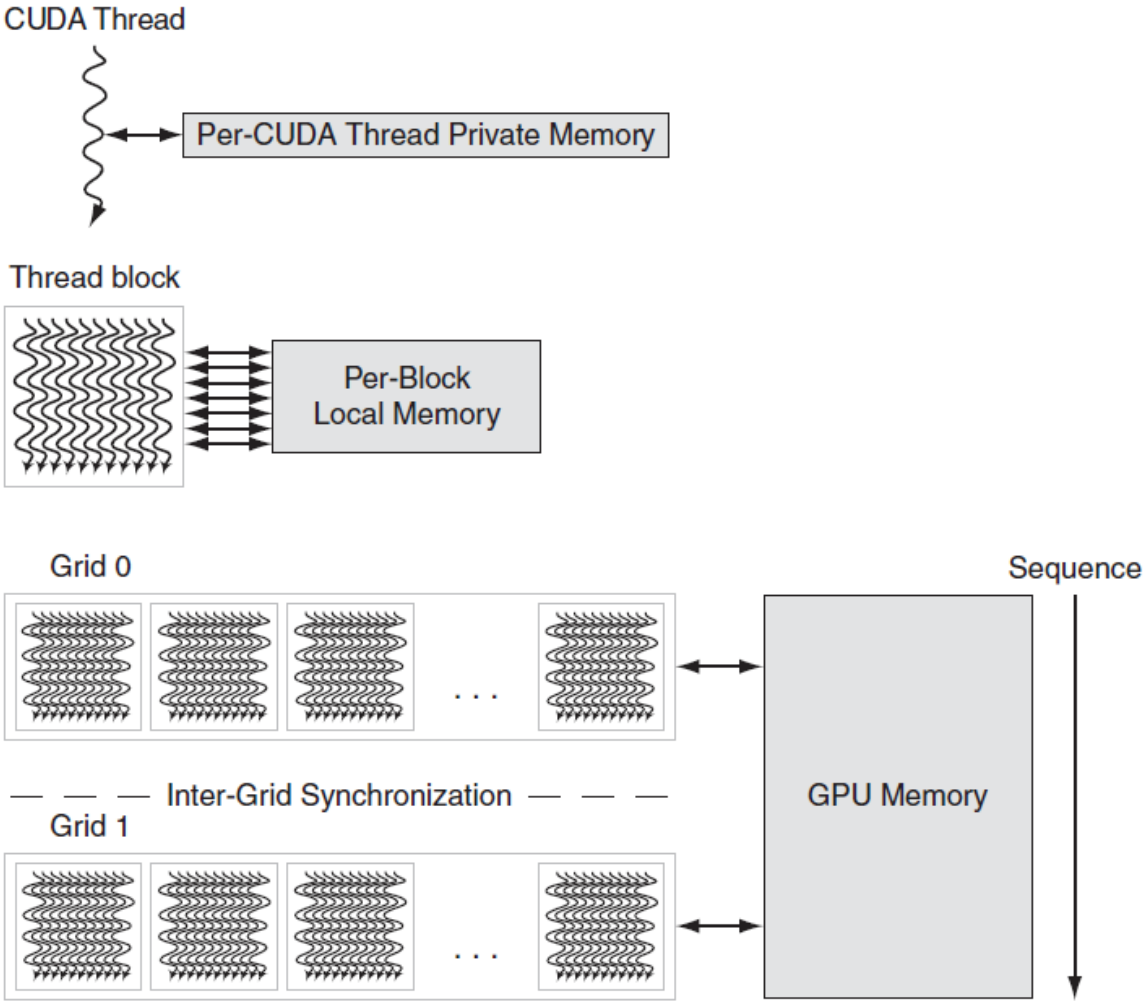


Classifying GPUs

- Don't fit nicely into SIMD/MIMD model
  - Conditional execution in a thread allows an illusion of MIMD
    - But with performance degredation
    - Need to write general purpose code with care

	Static: Discovered at Compile Time	Dynamic: Discovered at Runtime
Instruction-Level Parallelism	VLIW	Superscalar
Data-Level Parallelism	SIMD or Vector	**Tesla Multiprocessor**

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

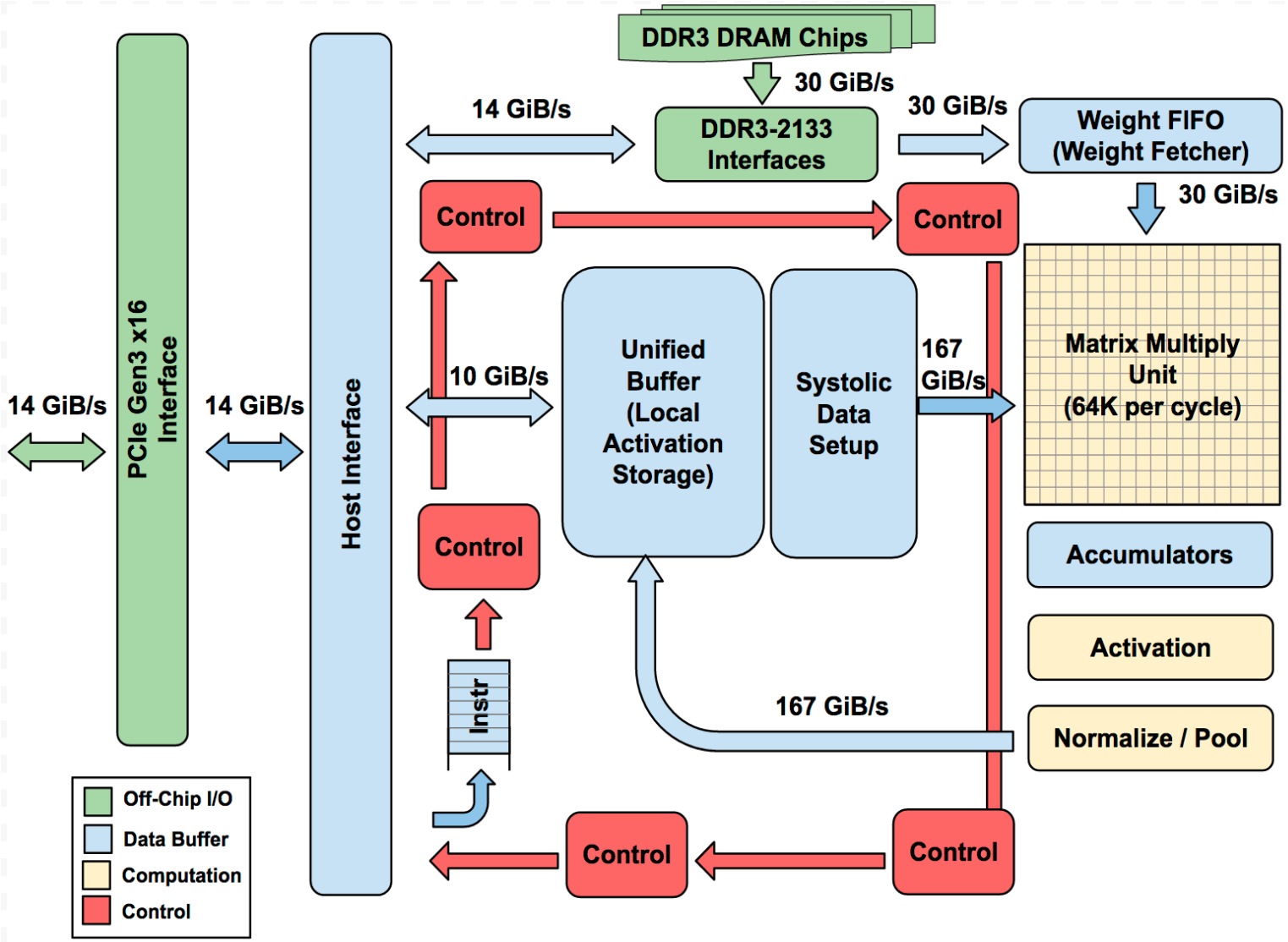
Guide to GPU Terms

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).

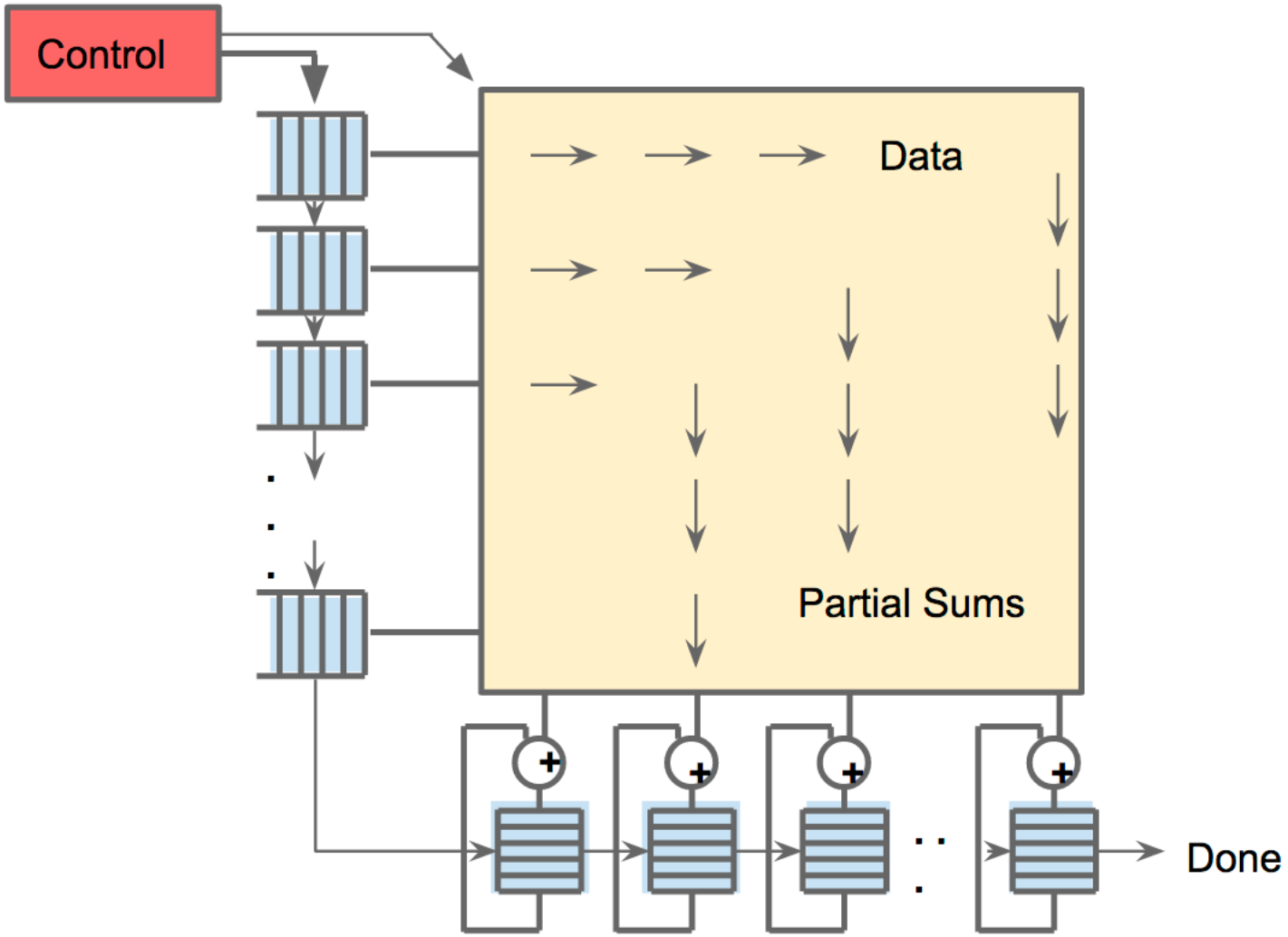
Tensor Processing Unit (TPU)

- "We concluded that the few applications that could run on special hardware could be done virtually for free using the excess capacity of our large datacenters, and it's hard to improve on free. That changed in 2013 when a projection showed people searching by voice for three minutes a day using speech recognition DNNs would double our datacenters' computation demands, which would be very expensive using conventional CPUs."

Tensor Processing Unit (TPU)



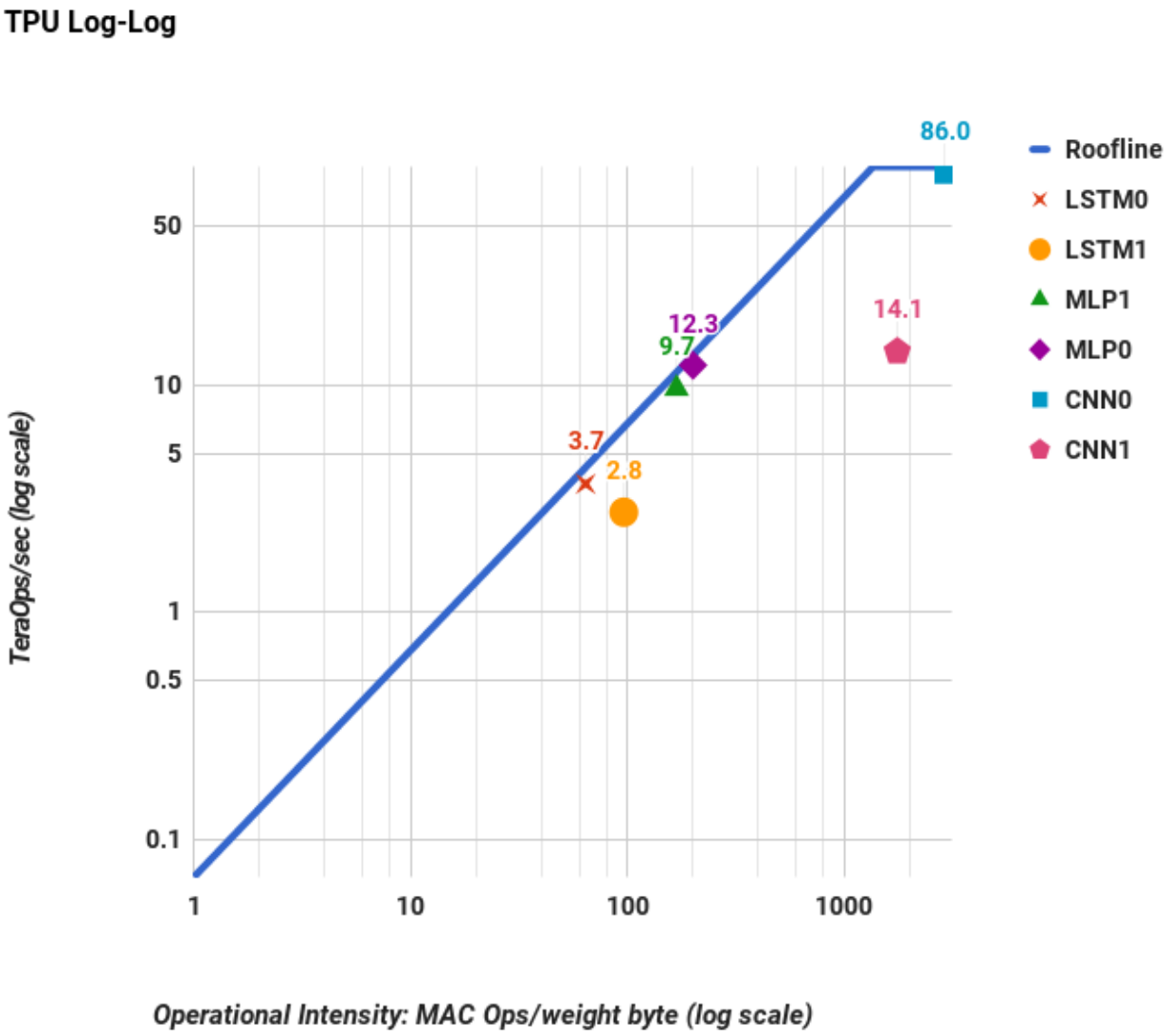
TPU – Systolic Array



Tensor Processing Unit (TPU)

- Inference apps usually emphasize response-time over throughput
  - K80 GPU is just a little faster for inference than the Haswell CPU, despite it having much higher peak performance and memory bandwidth
- CNNs are just 5% of our datacenter workload
- TPU 15X – 30X faster at inference than K80 GPU and Haswell CPU
- Four of the six NN apps are memory bound on TPU
  - TPU with same memory as the K80 GPU, it would be 30X – 50X faster than the GPU and CPU
- Despite having a much smaller and lower power chip, the TPU has 25 times as many MACs and 3.5 times as much on-chip memory as the K80 GPU
- Performance/Watt of TPU is 30X – 80X over contemporary CPUs and GPUs

TPU – Roofline



Acknowledgments

- These slides contain material developed and copyright by:
  - Joe Zambreno (Iowa State)
  - Akhilesh Tyagi (Iowa State)
  - David Patterson (UC Berkeley)
  - Mary Jane Irwin (Penn State)
  - Christos Kozyrakis (Stanford)
  - Onur Mutlu (Carnegie Mellon)
  - Krste Asanović (UC Berkeley)
  - Karu Sankaralingam (UW Madison)
  - Morgan Kaufmann