Lecture 13:

Programming Specialized Hardware and Cache Coherence

Parallel Computing
Stanford CS149, Fall 2024

Today's Themes

Nvidia H100

- Asynchronous compute and memory mechanisms ⇒ complex programing
- Simplify with Thunderkittens DSL

SambaNova SN40L

- Dataflow architecture
- Programing model: tiling and streaming with metapipelining

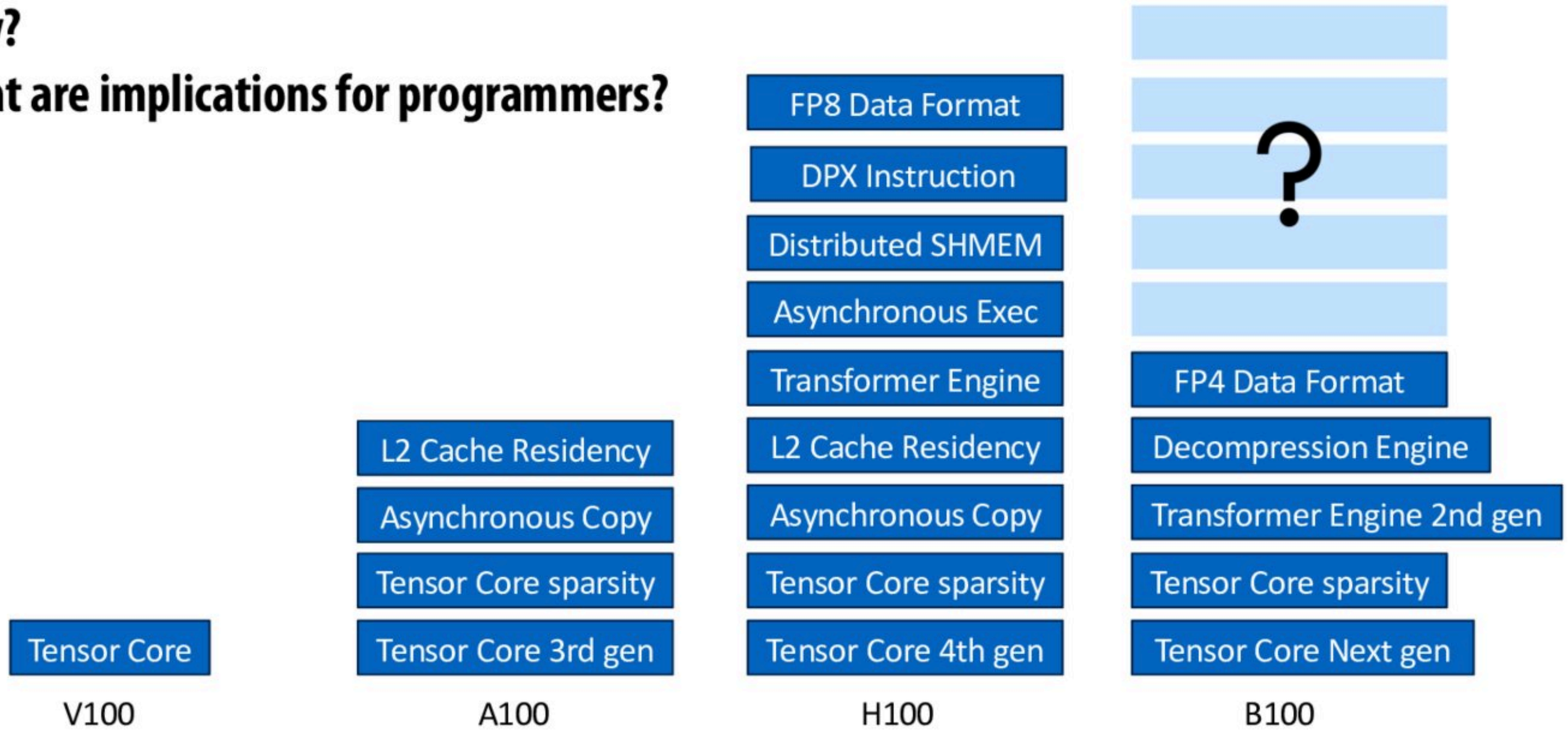
Cache Coherence

- Memory coherence problem
- Cache coherence protocols

Nvidia Chips Becoming More Specialized

Why?

What are implications for programmers?



The Whole H100

144 SMs

Tensor cores (systolic array MMA): 989 TFLOPS (fp16) ⇒ ~90% of TFLOPS

SIMD: 134 TFLOPS (fp16), 67 TFLOPS (fp32) ⇒ ~10% of TFLOPS

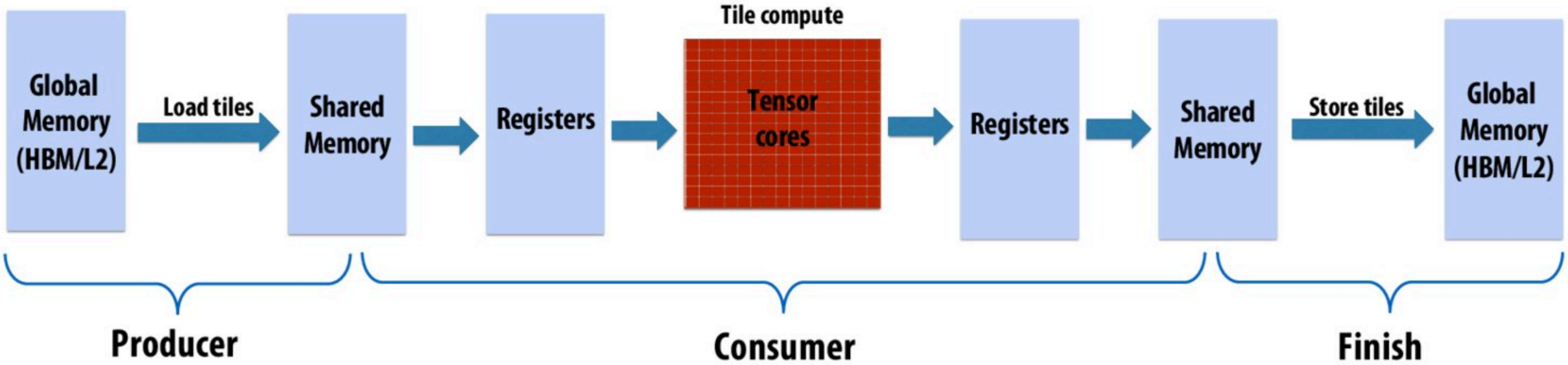
ThunderKittens

A Simple Embedded DSL for AI kernels

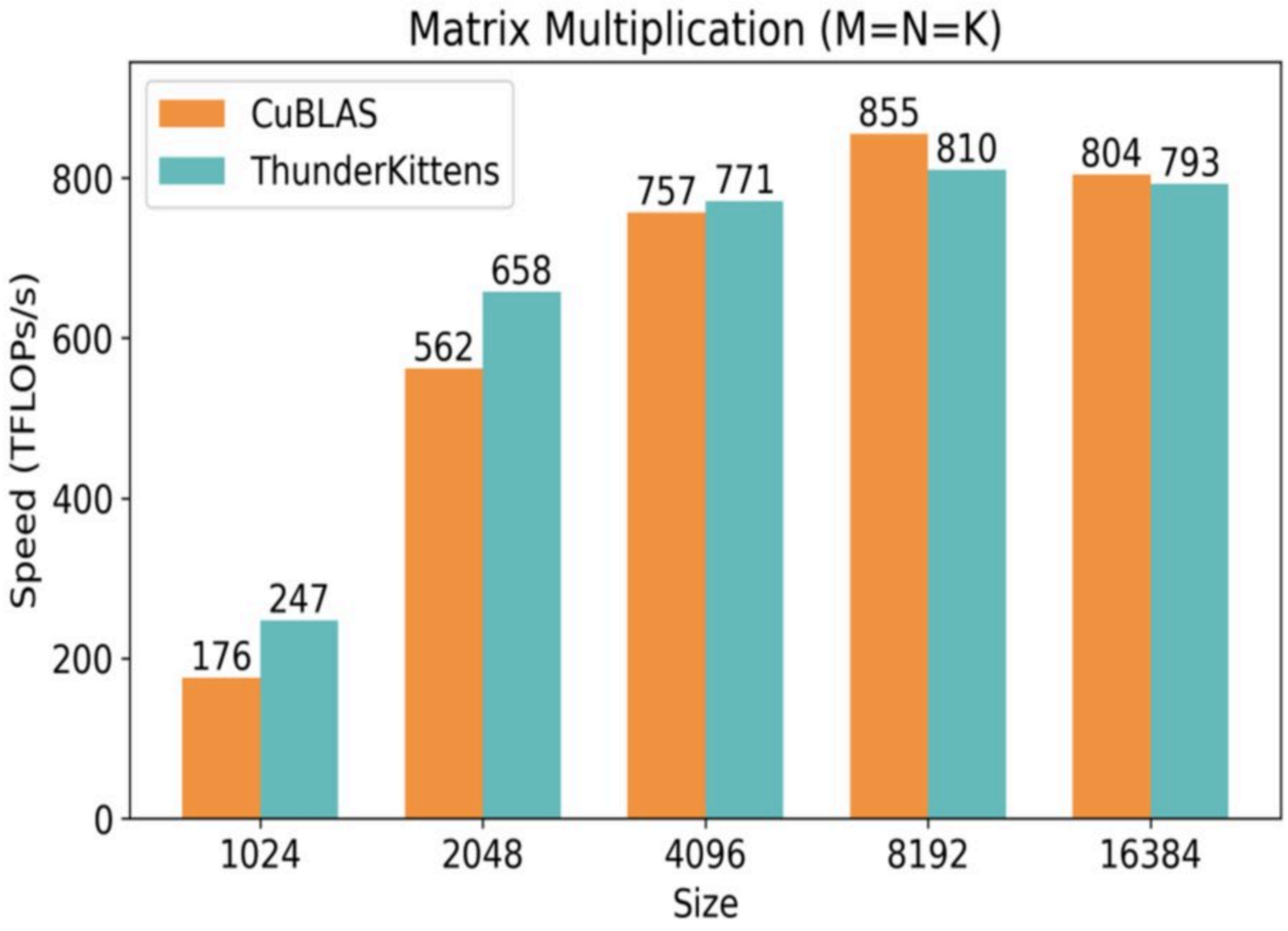
Ben Spector et. al.

- **Design principle #1: tile of 16x16 as primitive data type**
  - **TK manages layouts**
  - **TK provides basic operations**
- **Design principle #2: Asynchrony, everywhere**
  - **Expose primitives for user to manage, if top performance needed**
- **Design principle #3: High-level GPU coordination patterns**
  - **Producer-consumer processing**

Tile Processing Pipeline with ThunderKittens



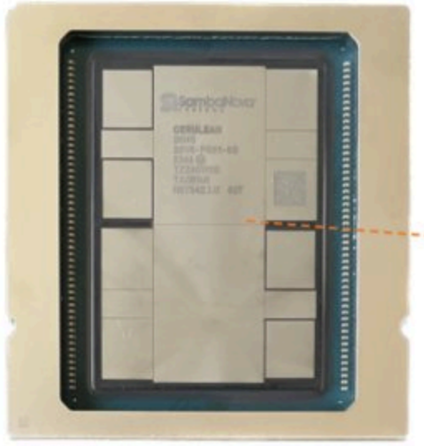
TK Matmul Performance



Can we get asynchrony with a simpler programming model?

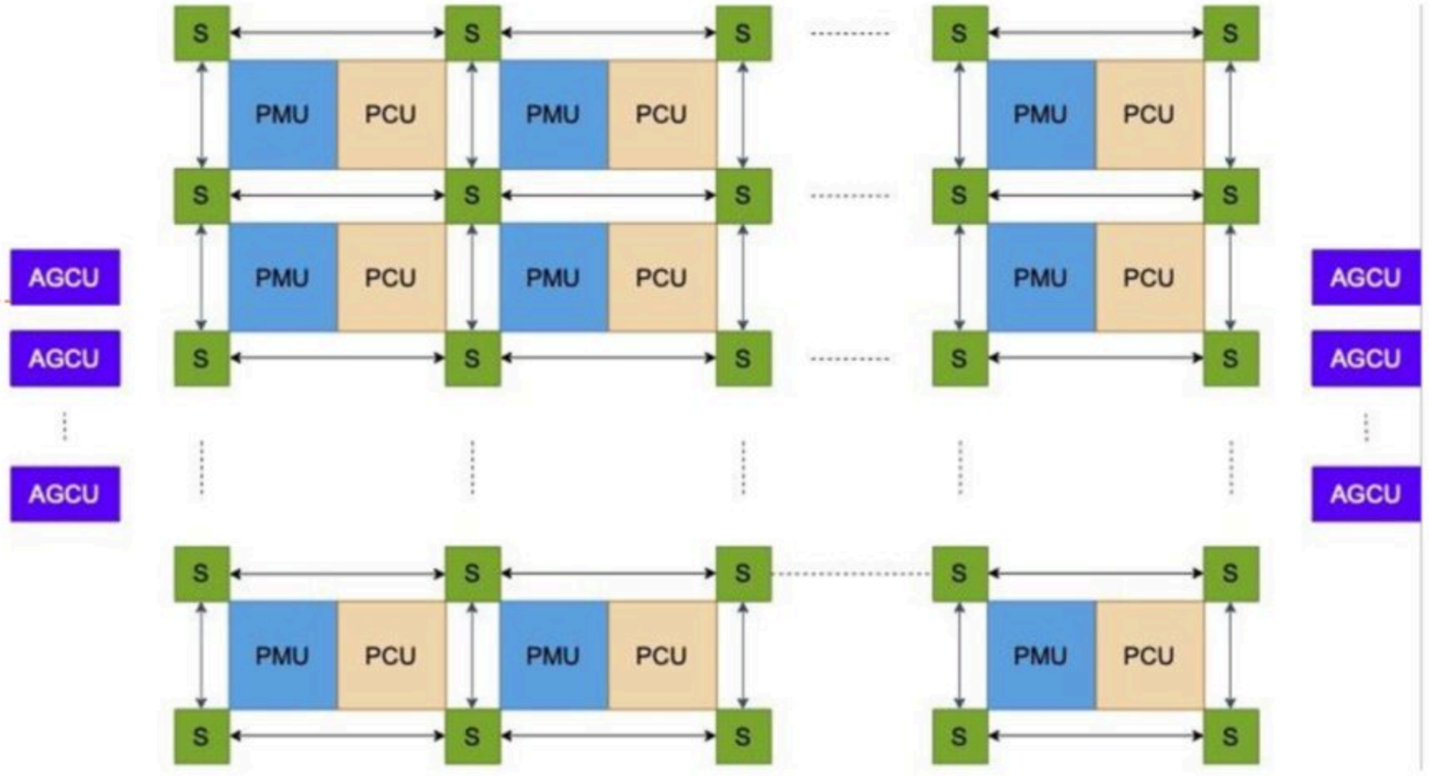
(Hint: Take a data-centric view)

Reconfigurable Dataflow



SambaNova SN40L RDU

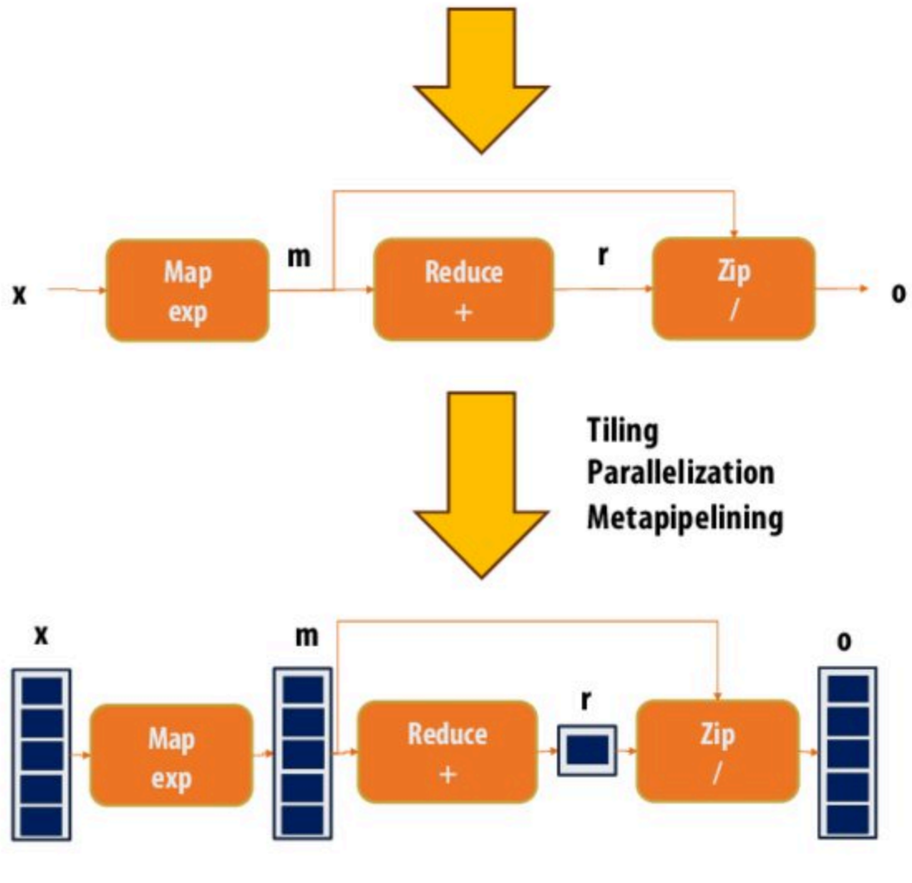
- 1,040 PCUs and PMUs
- 638 TFLOPS (bf16)
- 520 MB on-chip SRAM
- 64 GB HBM
- 1.5 TB DDR

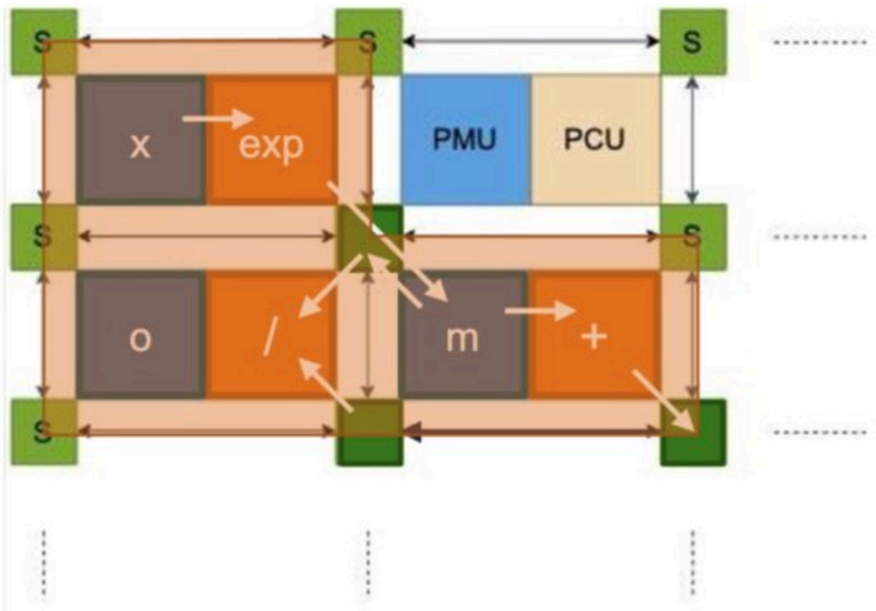


- PCU: Pattern Compute Unit
  - systolic and streaming compute (16 x 8 bf16)
- PMU: Pattern Memory Unit
  - High address generation flexibility and bandwidth (0.5 MB)
- S: Mesh switches
  - High on-chip interconnect flexibility and bandwidth
- AGCU: Address Generator and Coalescing Unit
  - Portal to off-chip memory and IO

Dataflow Programming with Data Parallel Patterns

SIMPLIFIED SOFTMAX $\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$





Place & Route
Codegen

- Composable Compute Primitives: MM, Map, Zip, Reduce, Gather, Scatter …
- Flexible scheduling in space and time ⇒ spatial execution

Metapipelining

Hierarchical coarse-grained pipeline: A "pipeline of pipelines"

- **Exploits nested-loop parallelism**

Convert parallel pattern (loop) into a streaming pipeline

- **Insert pipe stages in the body of the loop**
- **Pipe stages execute in parallel**
- **Overlap execution of multiple loop iterations**

Intermediate data between stages stored in double buffers

- **Handles imbalanced stages with varying execution times**

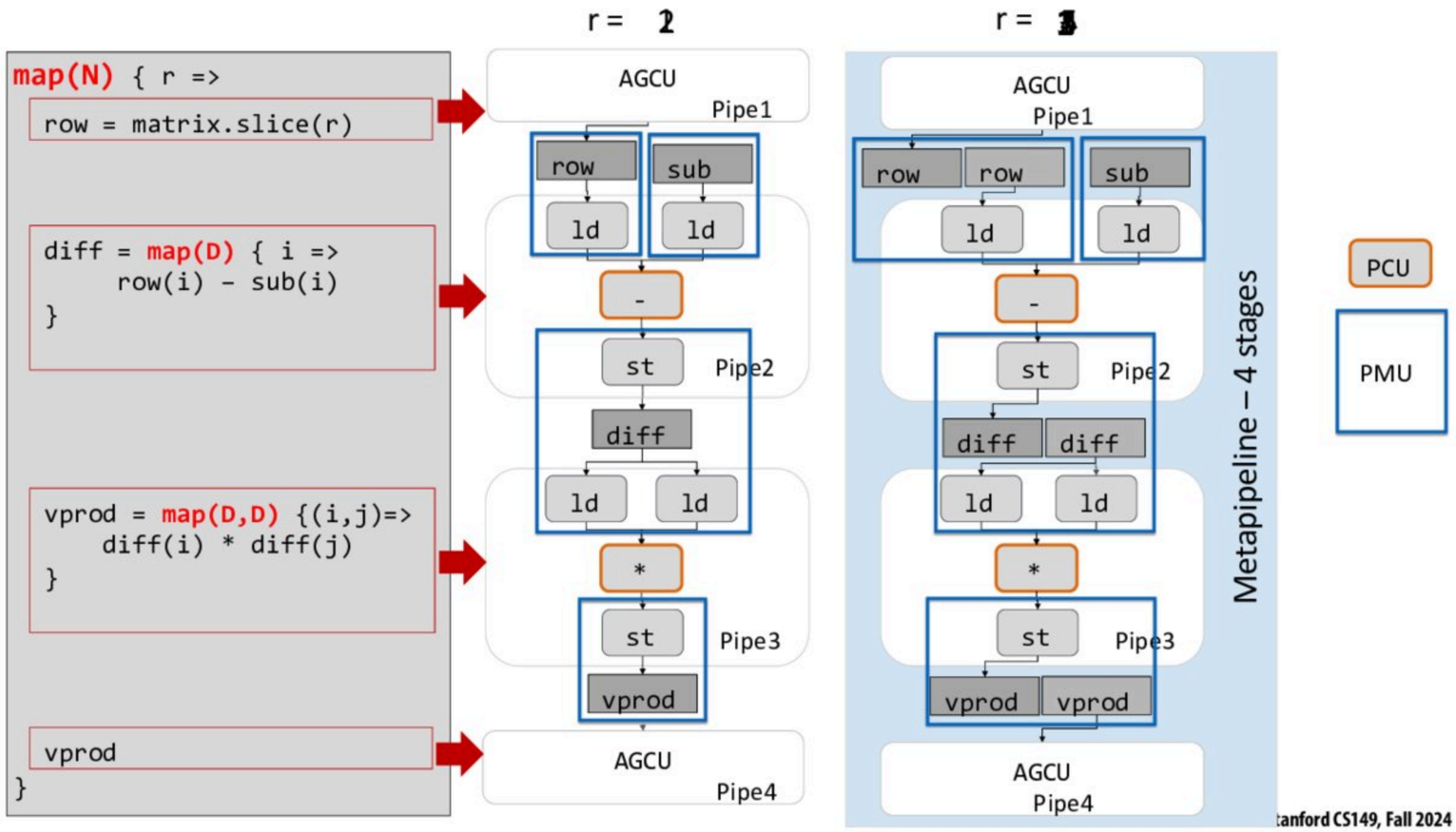
Tiling and fusion

- **Works well with tiling**
- **Buffers can be used to change access pattern (e.g. transpose data)**
- **Metapipelining can work when fusion does not**

Metapipelining Intuition

Gaussian Discriminant Analysis (GDA)

```
map(N) { r =>
  row = matrix.slice(r)

  diff = map(D) { i =>
       row(i) - sub(i)
  }

  vprod = map(D,D) {(i,j)=>
      diff(i) * diff(j)
  }

  vprod
}
```

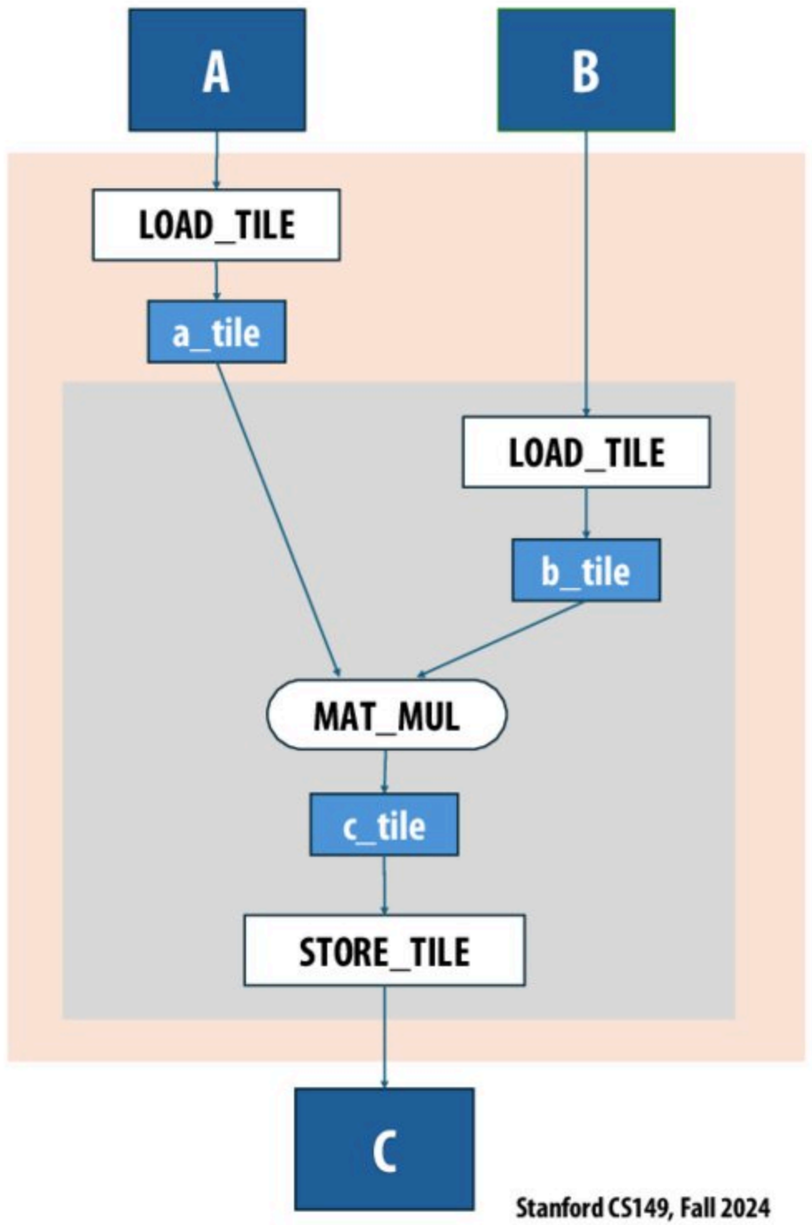


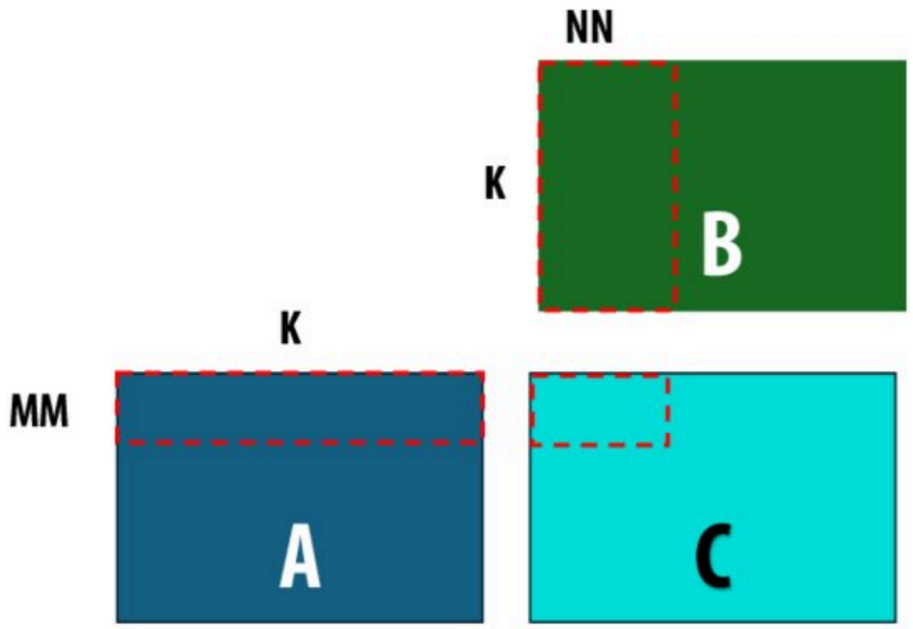
Matmul Metapipeline

```
auto format = DataFormat::kBF16;

int64_t M = args::M.getValue();
int64_t N = args::N.getValue();
int64_t K = args::K.getValue();


auto A = INPUT_REGION("A", (M, K), format);
auto B = INPUT_REGION("B", (K, N), format);
auto C = OUTPUT_REGION("C", (M, N), format);

auto MM = 256; // Tile size along M, assumes to evenly divide M
auto NN =  64; // Tile size along N, assumes to evenly divide N

auto a_tile_shape = std::vector<int64_t>({MM, K});
auto b_tile_shape = std::vector<int64_t>({K, NN});
auto c_tile_shape = std::vector<int64_t>({MM, NN});

METAPIPE(M / MM, [&]() {
     auto a_tile = LOAD_TILE(A, a_tile_shape);
     METAPIPE(N / NN, [&]() {
          auto b_tile = LOAD_TILE(B, b_tile_shape, row_par = 4);
          auto c = MAT_MUL(a_tile, b_tile);
        auto c_tile = BUFFER(c);
          STORE_TILE(C, c_tile);
     });
});
```

Matmul Metapipe

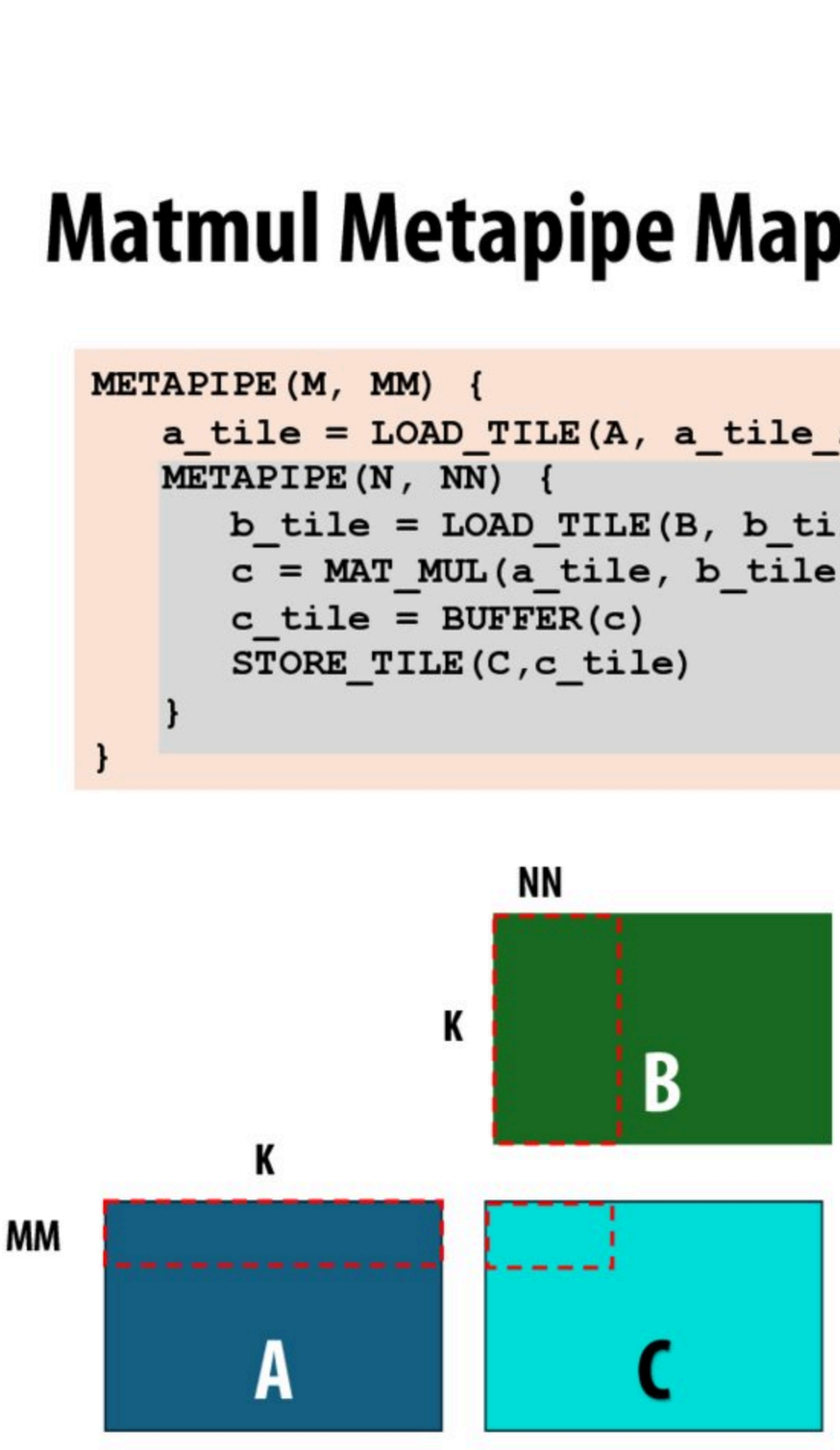
```
METAPIPE(M, MM) {
    a_tile = LOAD_TILE(A, a_tile_shape)
    METAPIPE(N, NN) {
        b_tile = LOAD_TILE(B, b_tile_shape)
        c = MAT_MUL(a_tile, b_tile, row_par = 4)
        c_tile = BUFFER(c)
        STORE_TILE(C,c_tile)
    }
}
```

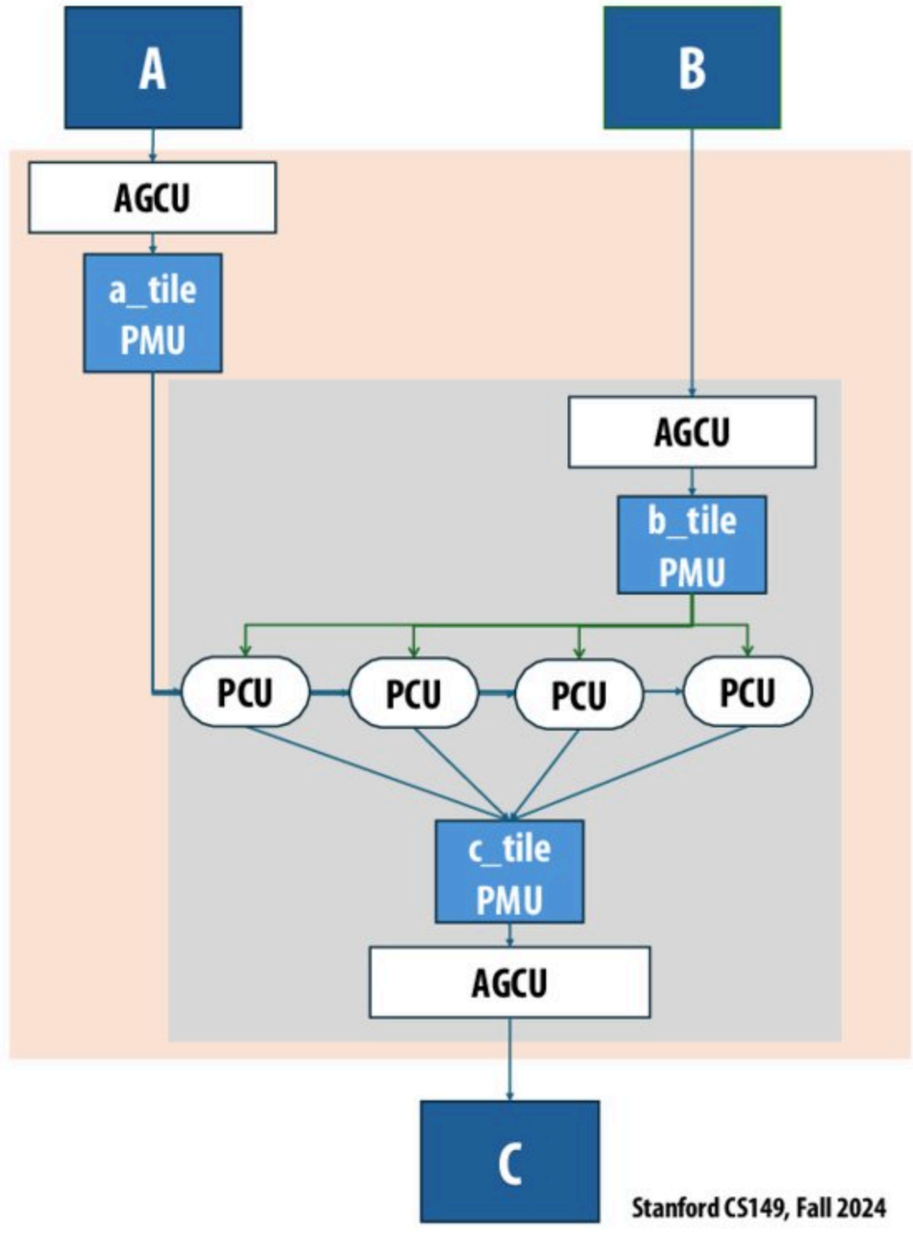




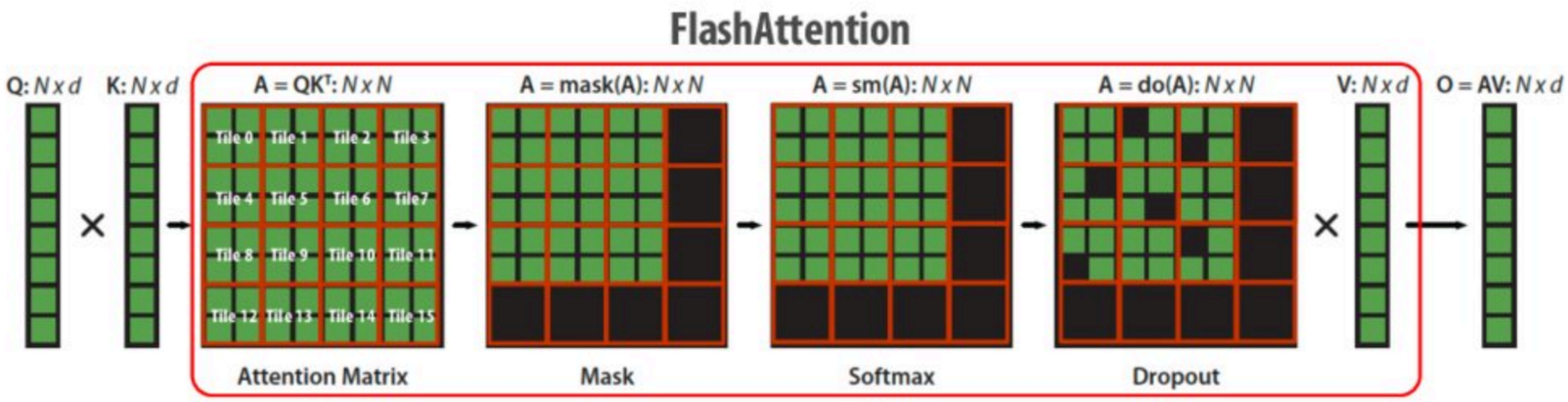
Matmul Metapipe Mapping

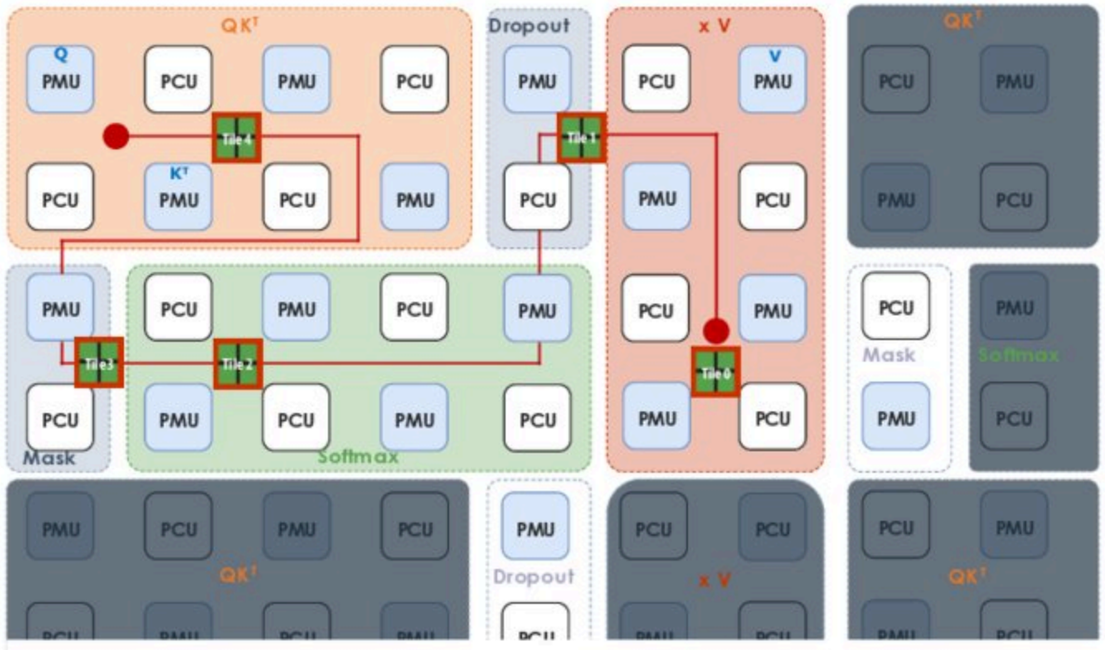
```
METAPIPE(M, MM) {
    a_tile = LOAD_TILE(A, a_tile_shape)
    METAPIPE(N, NN) {
        b_tile = LOAD_TILE(B, b_tile_shape)
        c = MAT_MUL(a_tile, b_tile, row_par = 4)
        c_tile = BUFFER(c)
        STORE_TILE(C,c_tile)
    }
}
```



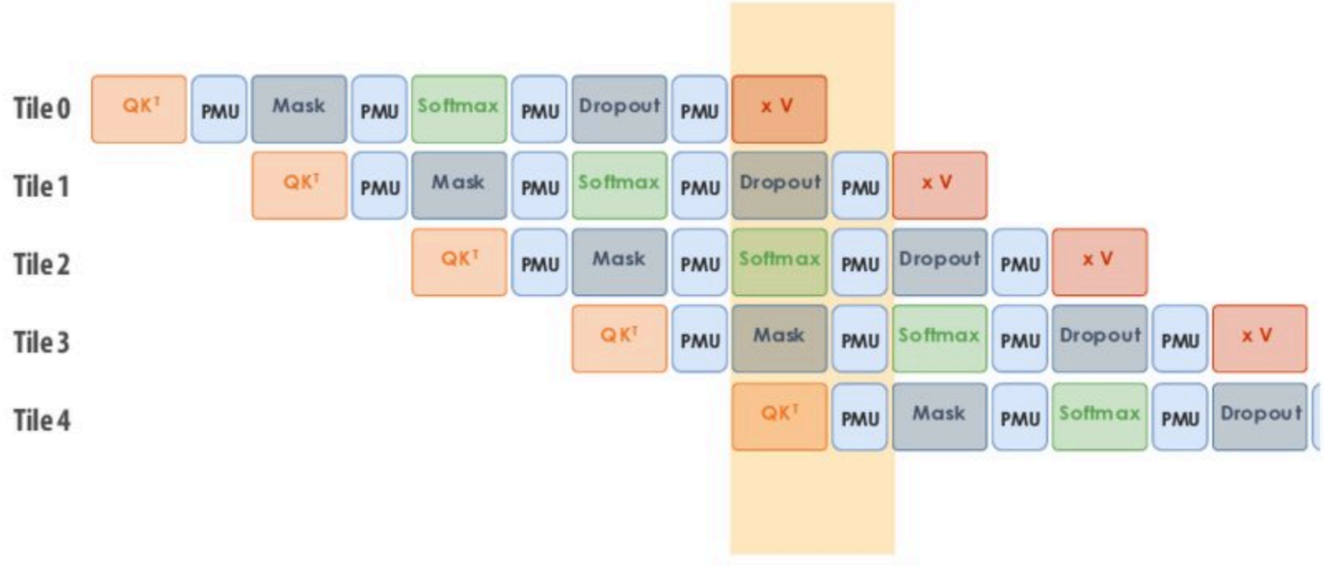


FlashAttention Metapipeline



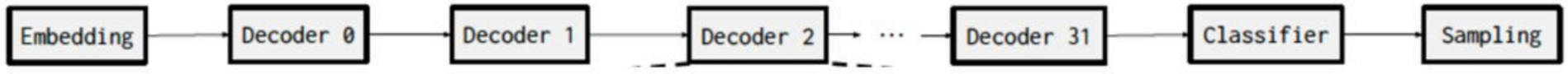


Dataflow execution with token control ⇒ no lock-based synchronization



MetaPipeline = Streaming Dataflow

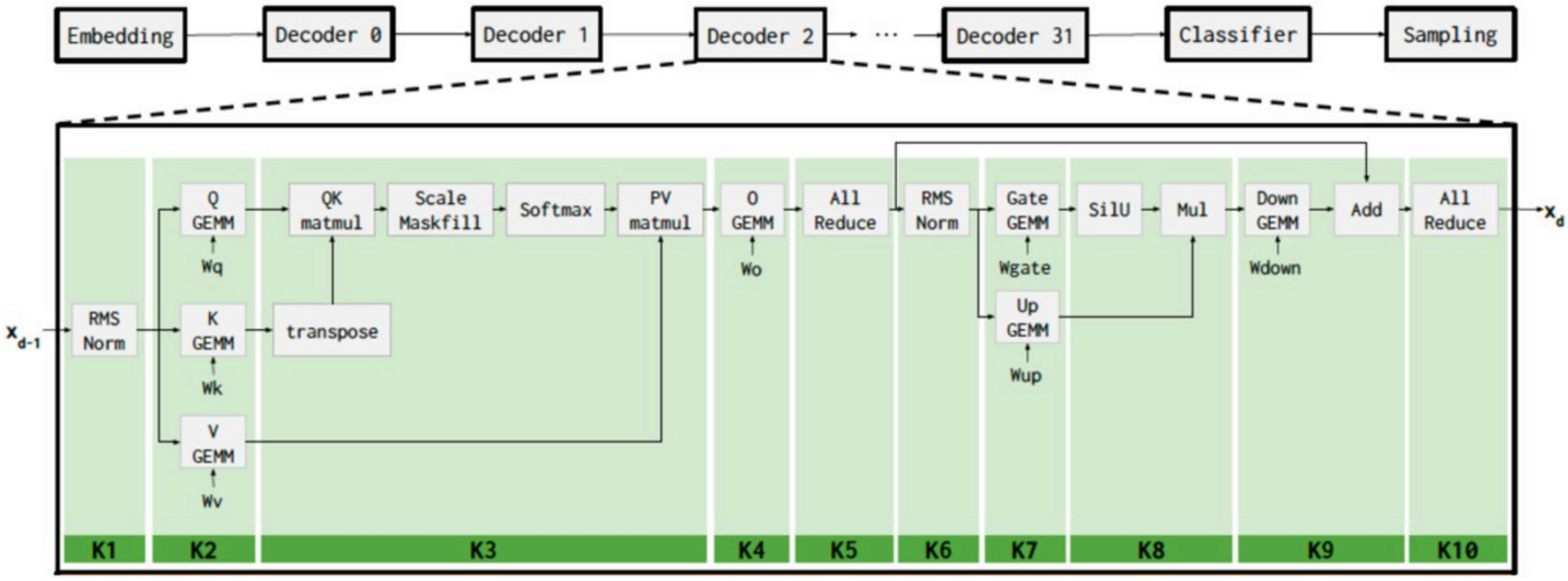
Llama 3.1 8B



Llama3.1-8B with 32 decoder layers

Tensor Parallel Llama 3.1 8B

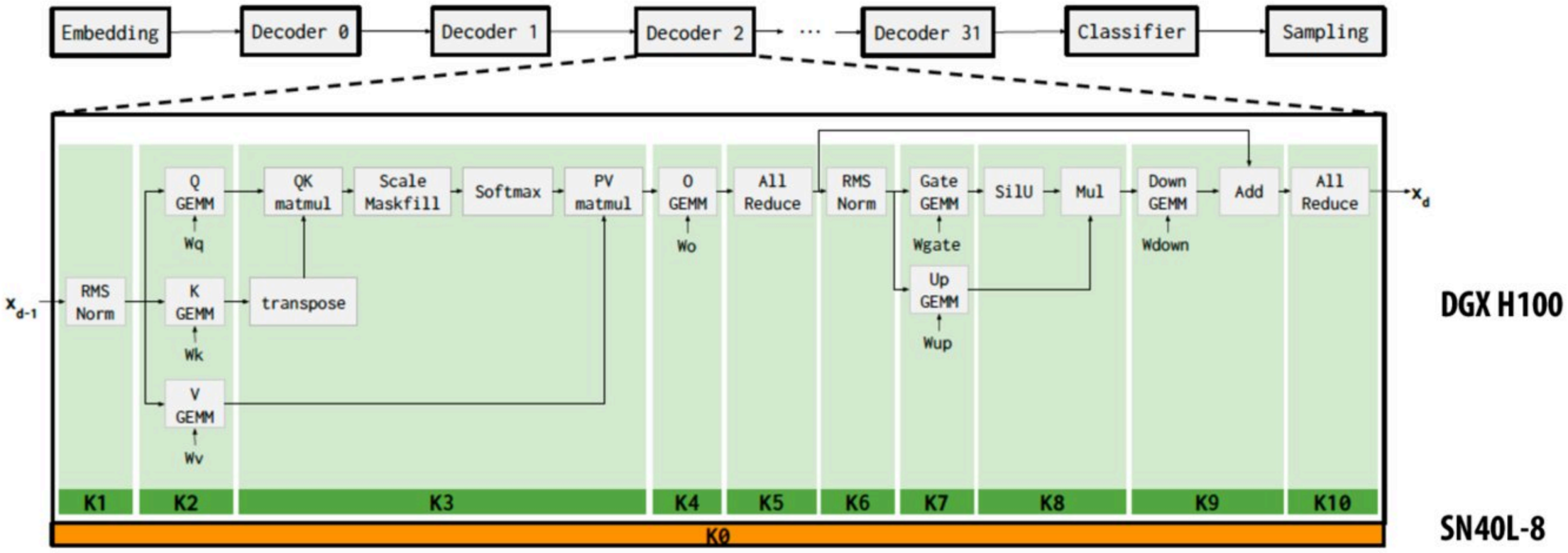
Parallelize across 8 chips



DGX H100

Tensor Parallel Llama 3.1 8B

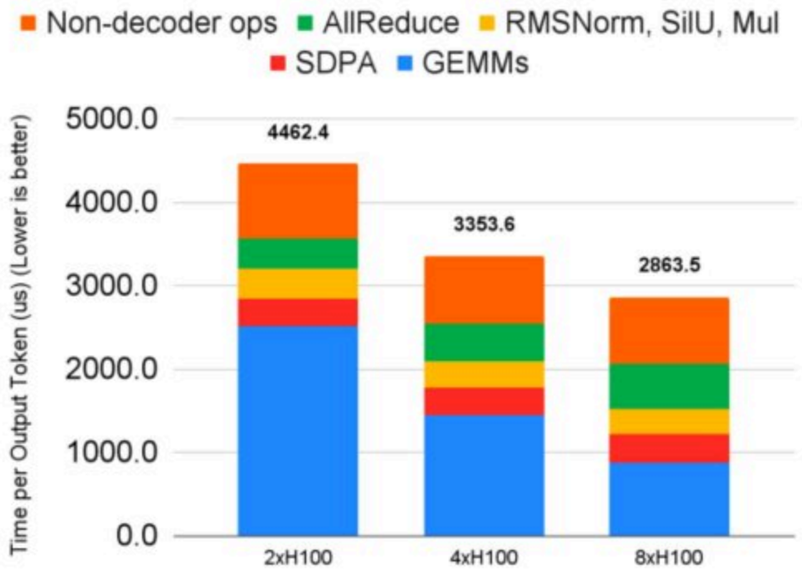
Parallelize across 8 chips



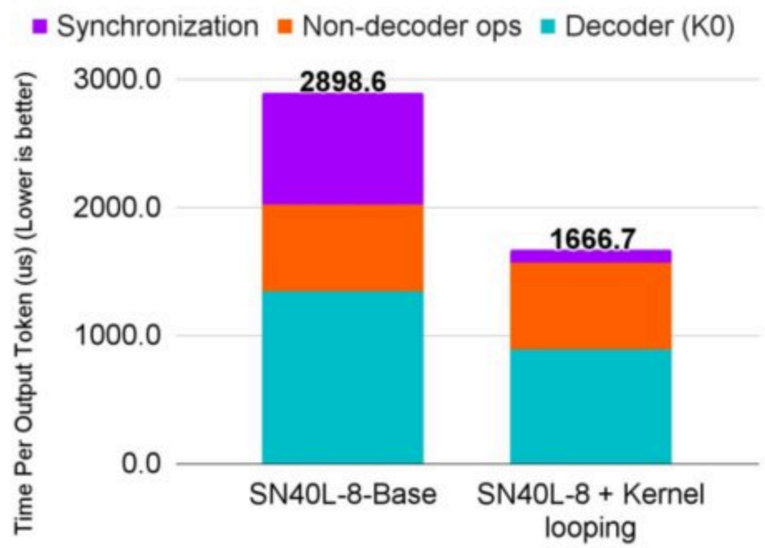
SN40L: single kernel

- Allreduce is asynchronous and pipelined with other operators
- Kernel looping further reduces overheads

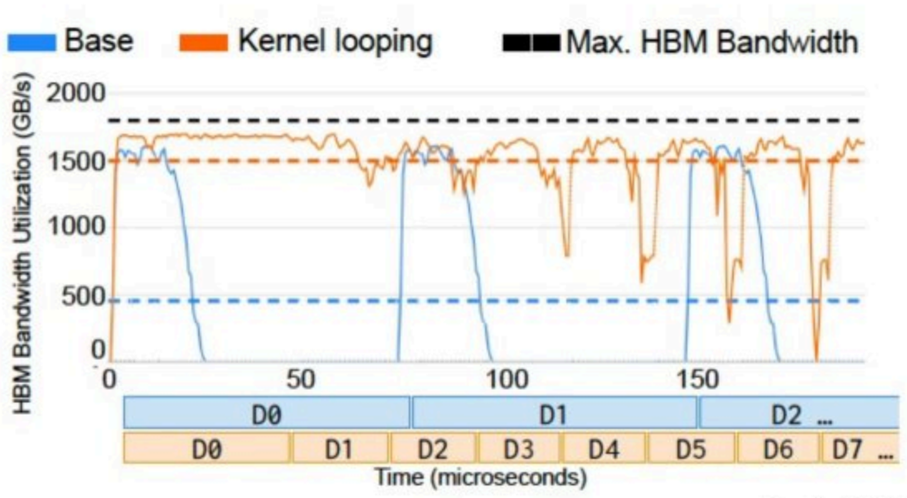
DGX H100 vs. SN40L-8



H100: 1000 TFLOPS, 3 TB/s HBM



SN40L: 638 TFLOPS, 1.6 TB/s HBM



Summary: Specialized Hardware and Programming for DNN Processing

Specialized hardware for executing key DNN computations efficiently

Feature large/many matrix multiply units

Large amounts of on-chip storage for fast access to intermediates

Customized/configurable datapaths to directly move intermediate data values between processing units (schedule computation by laying it out spatially on the chip)

H100: Asynchronous compute and memory mechanisms ⇒ complex programming

- Need ThunderKittens to manage complexity

SN40L: Dataflow model with metapipelining ⇒ simpler programming model

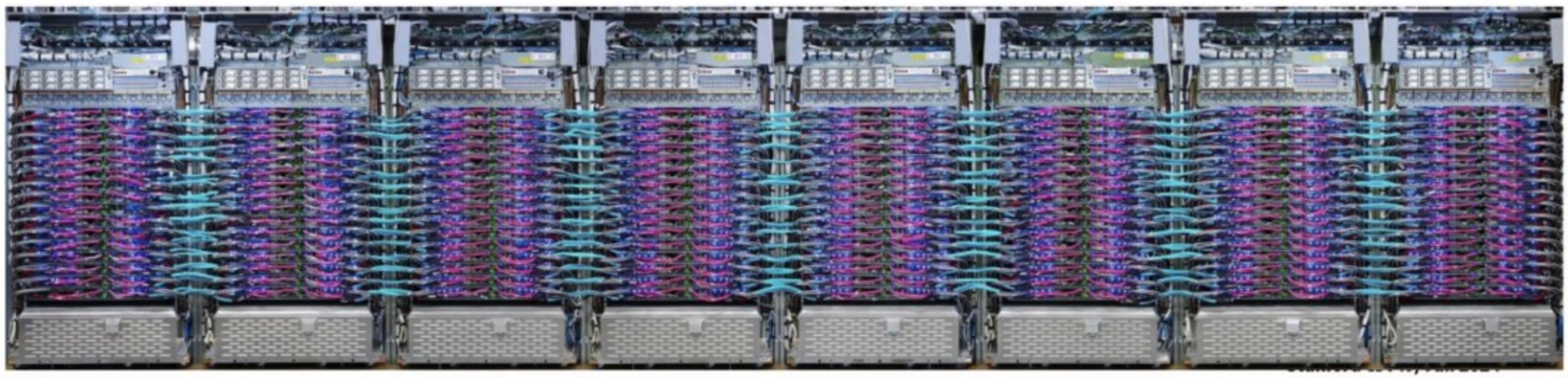
- Sophisticated compiler to optimize and map to dataflow hardware

Minimizing synchronization overheads required for peak performance

H100

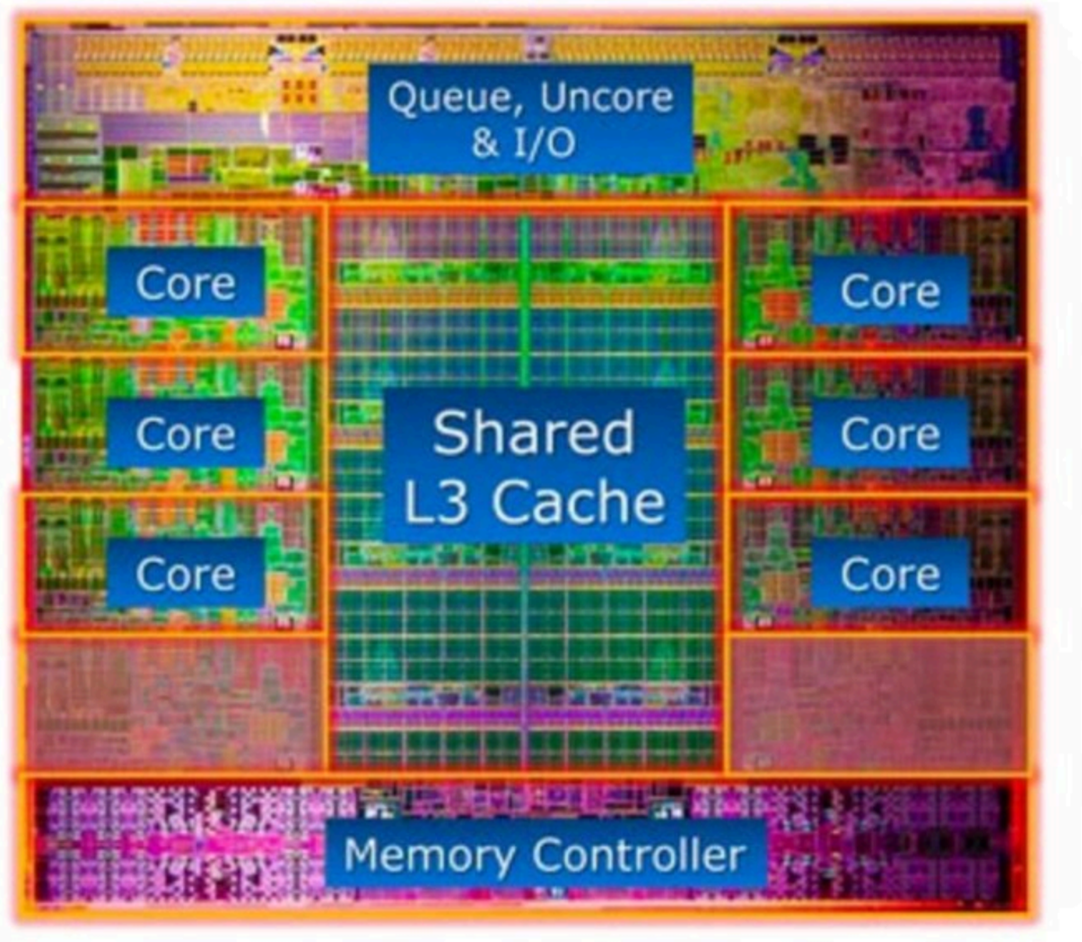
SN40L

TPU supercomputer (1024 TPU v3 chips)



Intel Core i7

Intel® Core™ i7-3960X Processor Die Detail



30% of the die area is cache

Rewview:Cache example 1

Array of 16 bytes in memory

	Address	Value
Line 0x0	0x0	16
	0x1	255
	0x2	14
	0x3	0
Line 0x4	0x4	0
	0x5	0
	0x6	6
	0x7	0
Line 0x8	0x8	32
	0x9	48
	0xA	255
	0xB	255
Line 0xC	0xC	255
	0xD	0
	0xE	0
	0xF	0

Assume:

Total cache capacity of 8 bytes

Cache with 4-byte cache lines
(So 2 lines fit in cache)

Least recently used (LRU)
replacement policy

time

Address accessed	Cache action	Cache state (after load is complete)
0x0	"cold miss", load 0x0	
0x1	hit	
0x2	hit	
0x3	hit	
0x2	hit	
0x1	hit	
0x4	"cold miss", load 0x4	
0x1	hit	

There are two forms of "data locality" in this sequence:

Spatial locality: loading data in a cache line "preloads" the data needed for subsequent accesses to <u>different addresses</u> in the same line, leading to cache hits

Temporal locality: repeated accesses to the <u>same address</u> result in hits.

Review: Cache example 2

Array of 16 bytes in memory

	Address	Value
Line 0x0	0x0	16
	0x1	255
	0x2	14
	0x3	0
Line 0x4	0x4	0
	0x5	0
	0x6	6
	0x7	0
Line 0x8	0x8	32
	0x9	48
	0xA	255
	0xB	255
Line 0xC	0xC	255
	0xD	0
	0xE	0
	0xF	0

Assume:

Total cache capacity of 8 bytes

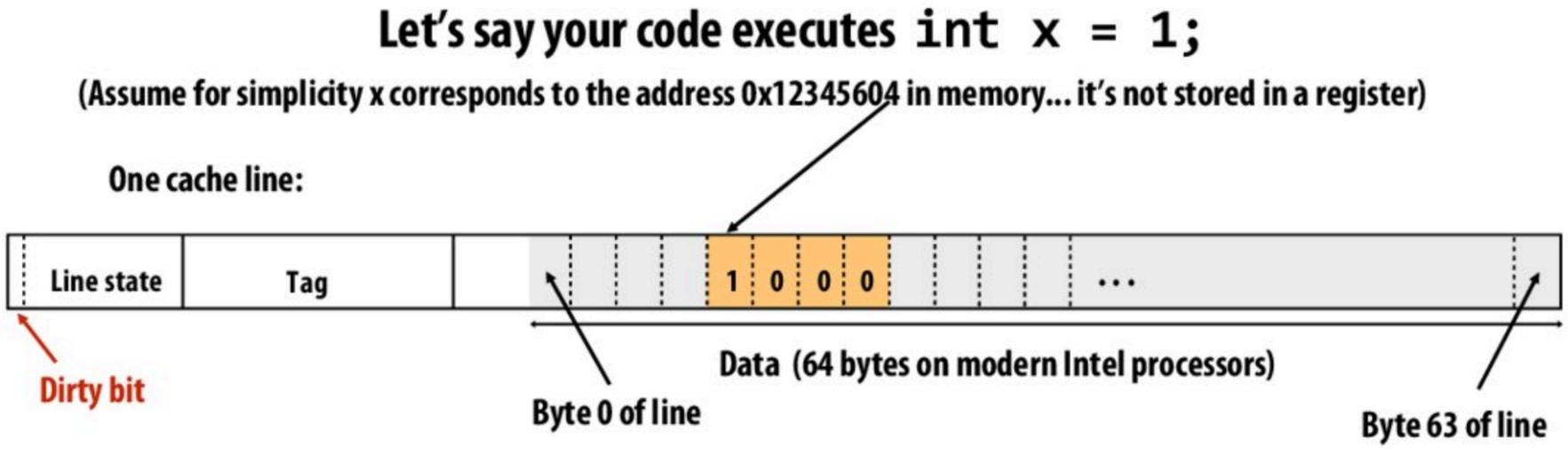
Cache with 4-byte cache lines
(So 2 lines fit in cache)

Least recently used (LRU)
replacement policy

time

Address accessed	Cache action	Cache state (after load is complete)
0x0	"cold miss", load 0x0	
0x1	hit	
0x2	hit	
0x3	hit	
0x4	"cold miss", load 0x4	
0x5	hit	
0x6	hit	
0x7	hit	
0x8	"cold miss", load 0x8 (evict 0x0)	
0x9	hit	
0xA	hit	
0xB	hit	
0xC	"cold miss", load 0xC (evict 0x4)	
0xD	hit	
0xE	hit	
0xF	hit	
0x0	"capacity miss", load 0x0 (evict 0x8)	

Cache Design



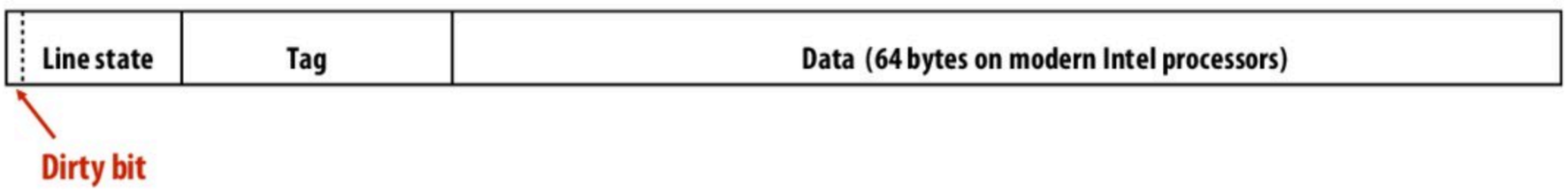
Do you know the difference between a write back and a write-through cache?

What about a write-allocate vs. write-no-allocate cache?

Behavior of write-allocate, write-back cache on a write miss (uniprocessor case)

Example: processor executes `int x = 1;`

1. Processor performs write to address that "misses" in cache
2. Cache selects location to place line in cache, if there is a dirty line currently in this location, the dirty line is written out to memory
3. Cache loads line from memory ("allocates line in cache")
4. Whole cache line is fetched and 32 bits are updated
5. Cache line is marked as dirty

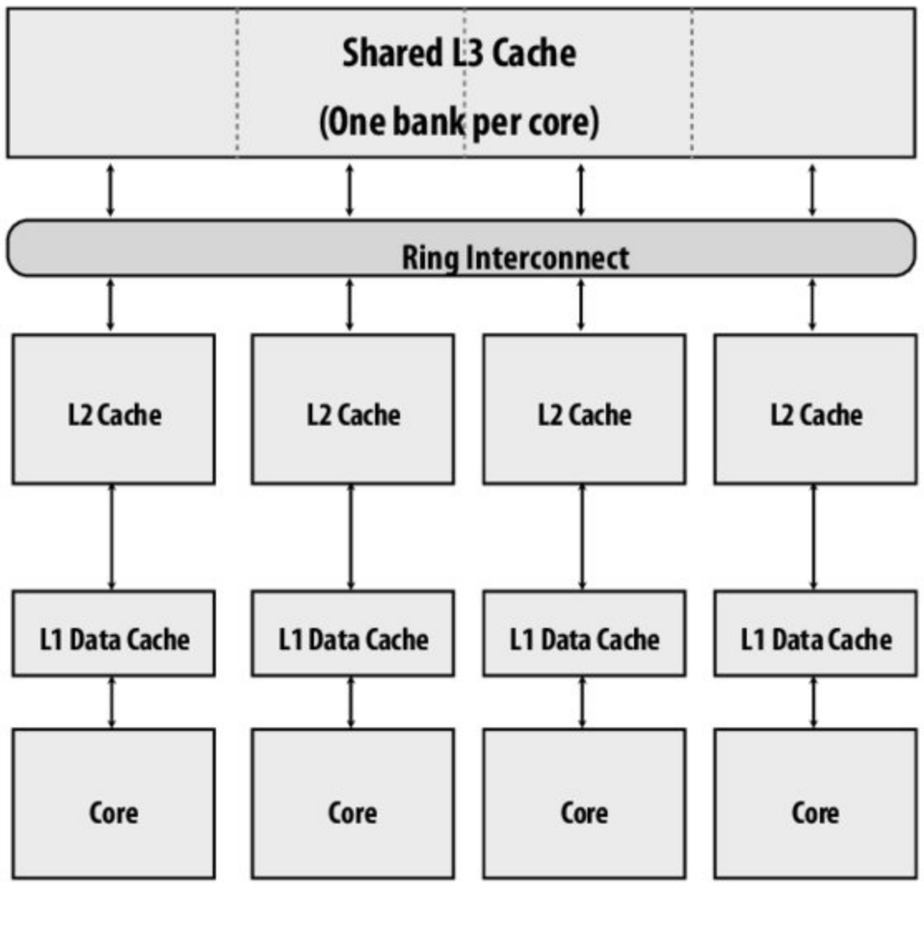


Cache hierarchy of Intel Skylake CPU (2015)

Caches exploit locality

3 Cs cache miss model

- Cold
- Capacity
- Conflict



L3: (per chip)
8 MB, inclusive
16-way set associative
32B / clock per bank
42 cycle latency

64 byte cache line size

L2: (private per core)
256 KB
4-way set associative, write back
64B / clock, 12 cycle latency

L1: (private per core)
32 KB
8-way set associative, write back
2 x 32B load + 1 x 32B store per clock
4 cycle latency

Support for:
72 outstanding loads
56 outstanding stores

Source: Intel 64 and IA-32 Architectures Optimization Reference Manual (June 2016)

Review: Shared address space model (abstraction)

Threads Reading/writing to shared variables

- Inter-thread communication is implicit in memory operations
- Thread 1 stores to X
- Later, thread 2 reads X (and observes update of value by thread 1)

- Manipulating synchronization primitives
- e.g., ensuring mutual exclusion via use of locks

This is a natural extension of sequential programming

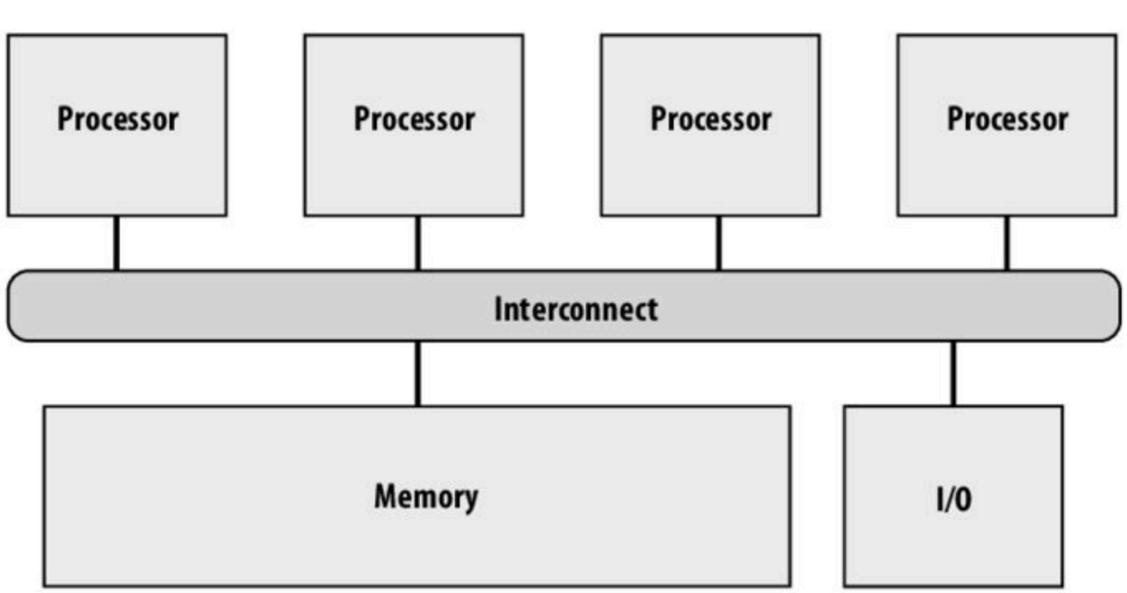
A shared memory multi-processor

Processors read and write to shared variables

- **More precisely: processors issue load and store instructions**

A reasonable expectation of memory is:

- **Reading a value at address X should return the last value written to address X by any processor**

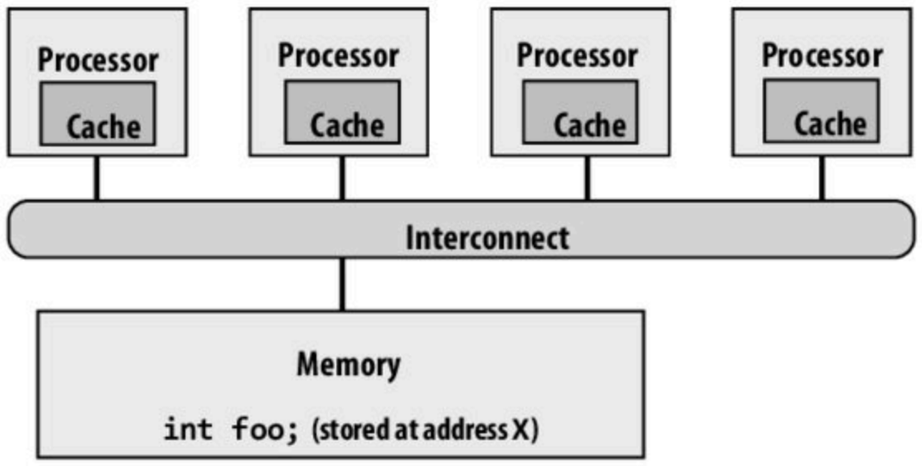


(A simple view of four processors and their shared address space)

The cache coherence problem

Modern processors replicate contents of memory in local caches

Problem: processors can observe different values for the same memory location



The chart at right shows the value of variable `foo` (stored at address X) in main memory and in each processor's cache

Assume the initial value stored at address X is 0

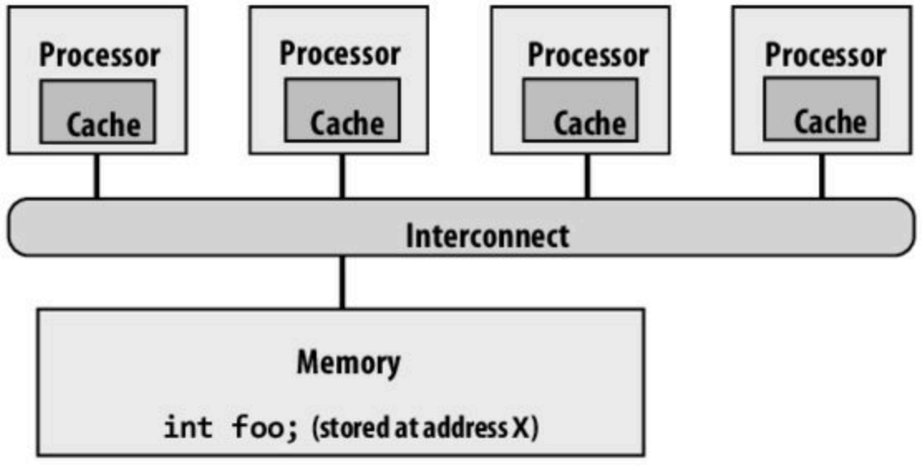
Assume write-back cache behavior

Action	P1 $	P2 $	P3 $	P4 $	mem[X]
					0
P1 load X	0 miss				0
P2 load X	0	0 miss			0
P1 store X	1	0			0
P3 load X	1	0	0 miss		0
P3 store X	1	0	2		0
P2 load X	1	0 hit	2		0
P1 load Y (assume this load causes eviction of X)		0	2		1

The cache coherence problem

Modern processors replicate contents of memory in local caches

Problem: processors can observe different values for the same memory location



Is this a mutual exclusion problem?

Can you fix the problem by adding locks to your program?

NO!

This is a problem created by replicating the data stored at address X in local caches

The chart at right shows the value of variable `foo` (stored at address X) in main memory and in each processor's cache

Assume the initial value stored at address X is 0

Assume write-back cache behavior

How could we fix this problem?

Action	P1 $	P2 $	P3 $	P4 $	mem[X]
					0
P1 load X	0 miss				0
P2 load X	0	0 miss			0
P1 store X	1	0			0
P3 load X	1	0	0 miss		0
P3 store X	1	0	2		0
P2 load X	1	0 hit	2		0
P1 load Y (assume this load causes eviction of X)		0	2		1

The memory coherence problem

Intuitive behavior for memory system: reading value at address X should return the last value written to address X by any processor.

Memory coherence problem exists because there is both global storage (main memory) and per-processor local storage (processor caches) implementing the abstraction of a single shared address space.

Intuitive expectation of shared memory

Intuitive behavior for memory system: reading value at address X should return the last value written to address X by any processor.

On a uniprocessor, providing this behavior is fairly simple, since writes typically come from one source: the processor

- **Exception: device I/O via direct memory access (DMA)**

Problems with the intuition

Intuitive behavior: reading value at address X should return the last value written to address X by any processor

What does "last" mean?

- **What if two processors write at the same time?**
- **What if a write by P1 is followed by a read from P2 so close in time that it is impossible to communicate the occurrence of the write to P2 in time?**

In a sequential program, "last" is determined by program order (not time)

- **Holds true within one thread of a parallel program**
- **But we need to come up with a meaningful way to describe order across threads in a parallel program**

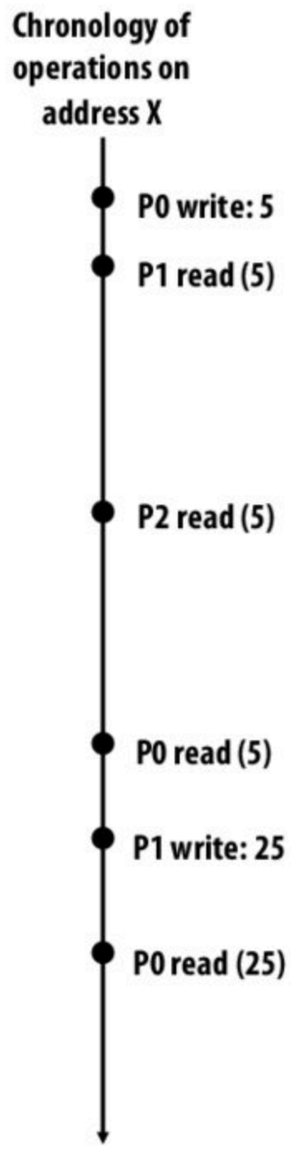
Definition: Coherence

A memory system is coherent if:

The results of a parallel program's execution are such that for each memory location, there is a hypothetical serial order of all program operations (executed by all processors) to the location that is consistent with the results of execution, and:

1. **Memory operations issued by any one processor occur in the order issued by the processor**

2. **The value returned by a read is the value written by the last write to the location… as given by the serial order**



Implementation: Cache Coherence Invariants

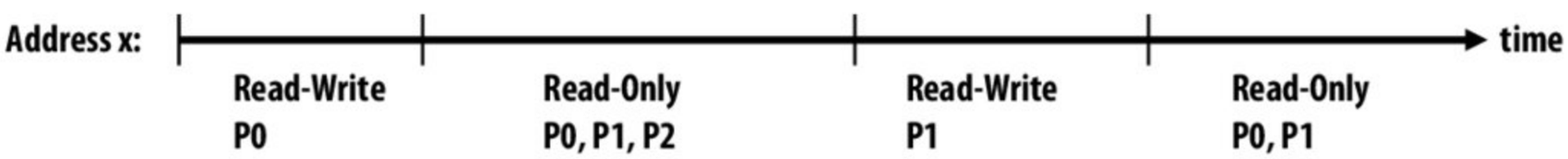
For any memory address x, at any given time period (epoch):

Single-Writer, Multiple-Read (SWMR) Invariant

- **Read-write epoch: there exists only a single processor that may write to x (and can also read it)**
- **Read-Only- epoch: some number of processors that may only read x**

Data-Value Invariant (write serialization)

- **The value of the memory address at the start of an epoch is the same as the value of the memory location at the end of its last read-write epoch**



Implementing coherence

Software-based solutions (coarse grain: VM page)

- OS uses page-fault mechanism to propagate writes
- Can be used to implement memory coherence over clusters of workstations
- We won't discuss these solutions
- Big performance problem: false sharing (discussed later)

Hardware-based solutions (fine grain: cache line)

- "Snooping"-based coherence implementations (today)
- Directory-based coherence implementations (briefly)

Shared caches: coherence made easy

One single cache shared by all processors

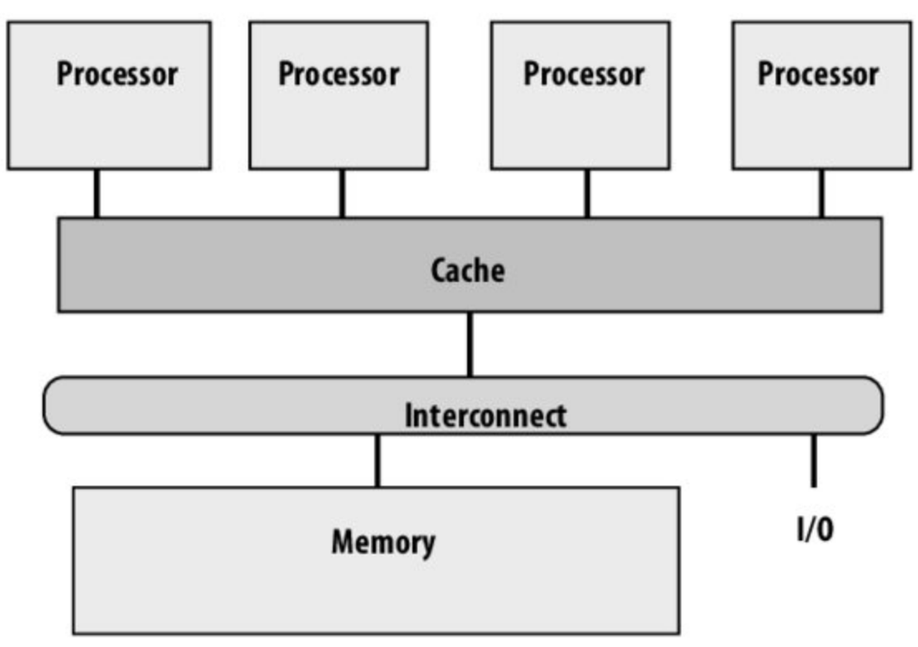
- Eliminates problem of replicating state in multiple caches

Obvious scalability problems (since the point of a cache is to be local and fast)

- Interference (conflict misses) / contention due to many clients (destructive)

But shared caches can have benefits:

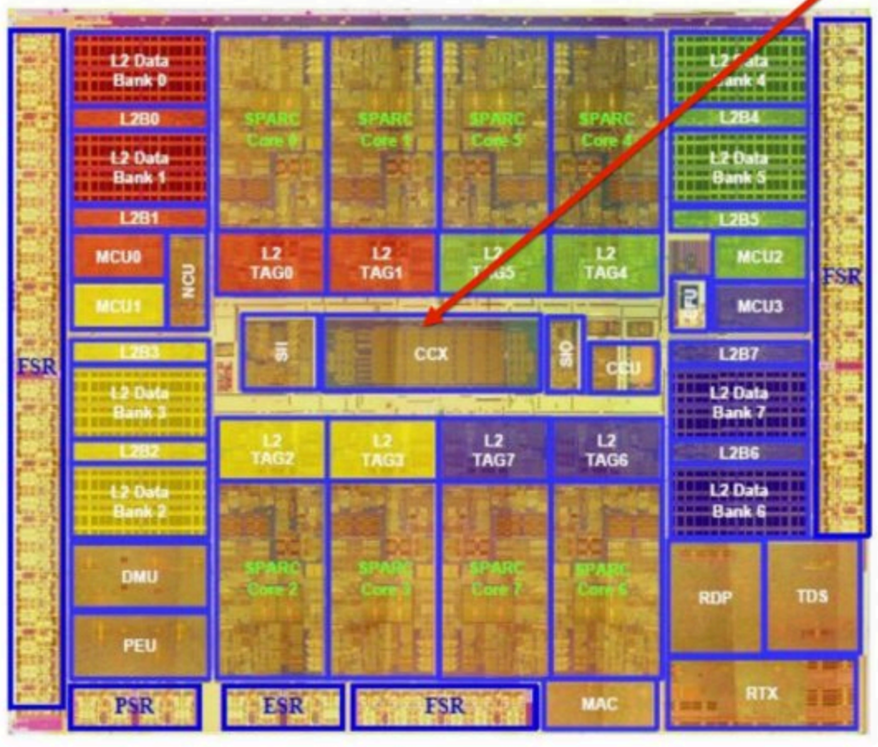
- Facilitates fine-grained sharing (overlapping working sets)
- Loads/stores by one processor might pre-fetch lines for another processor (constructive)



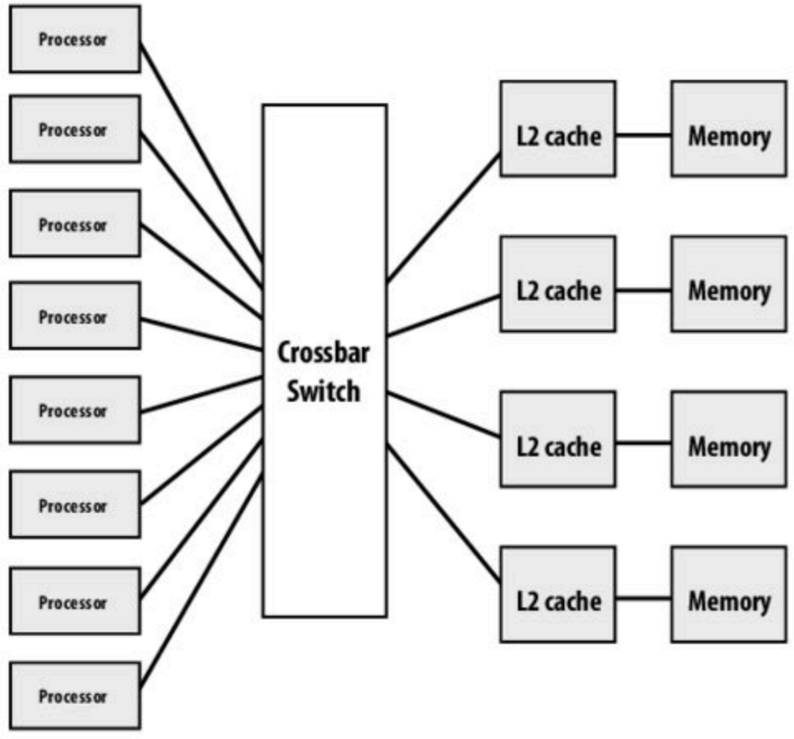

```
forall (i= 0; i++; i< N)
  x[i] = y[i] + y[i+1] + y[i+2];
```

SUN Niagara 2 (UltraSPARC T2)

Note area of crossbar (CCX):
about same area as one core on chip



Eight cores



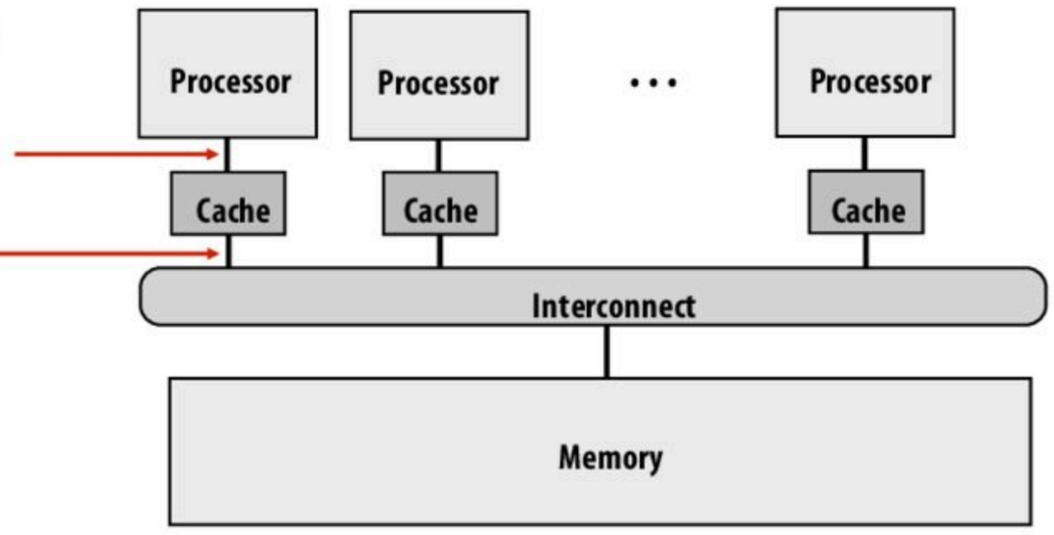
Snooping cache-coherence schemes

Main idea: all coherence-related activity is broadcast to all processors in the system (more specifically: to the processor's cache controllers)

Cache controllers monitor ("they snoop") memory operations, and follow cache coherence protocol to maintain memory coherence

Notice: now cache controller must respond to actions from "both ends":

1. LD/ST requests from its local processor
2. Coherence-related activity broadcast over the chip's interconnect



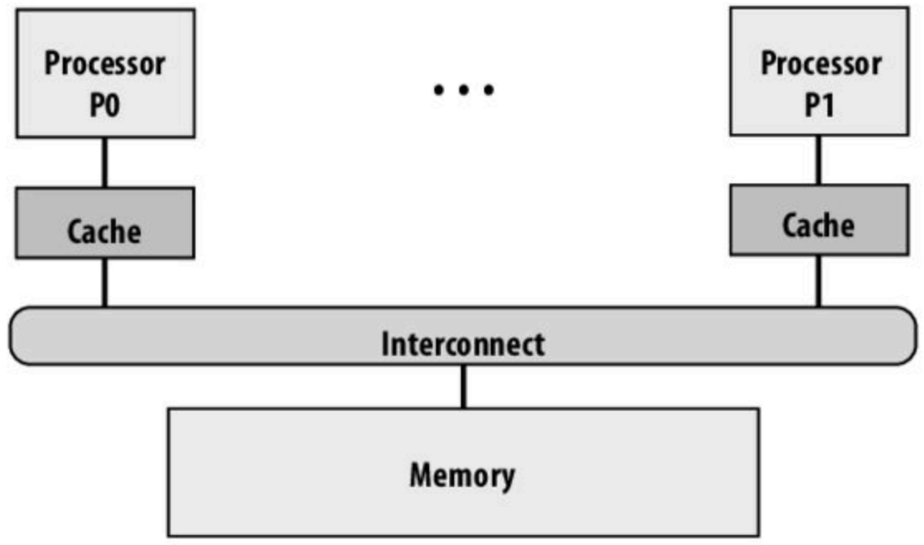
Very simple coherence implementation

Let's assume:

1. Write-through caches
2. Granularity of coherence is cache line

Coherence Protocol:

- Upon write, cache controller broadcasts invalidation message
- As a result, the next read from other processors will trigger cache miss



(processor retrieves updated value from memory due to write-through policy)

Action	Interconnect activity	P0 $	P1 $	mem location X
				0
P0 load X	cache miss for X	0		0
P1 load X	cache miss for X	0	0	0
P0 write 100 to X	invalidation for X	100		100
P1 load X	cache miss for X	100	100	100

Write-through policy is inefficient

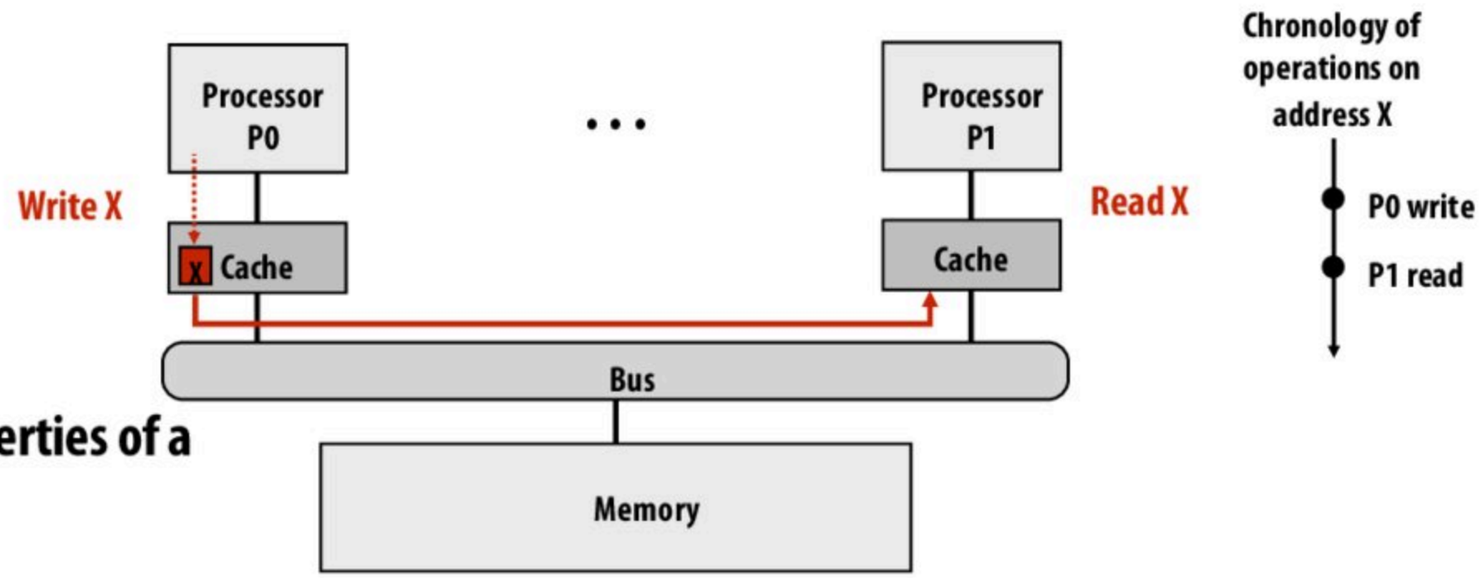
Every write operation goes out to memory

- Very high bandwidth requirements

Write-back caches absorb most write traffic as cache hits

- Significantly reduces bandwidth requirements
- But now how do we maintain cache coherence invariants?
- This requires more sophisticated coherence protocols

Cache coherence with write-back caches



What are two important properties of a bus?

Dirty state of cache line now indicates exclusive ownership (Read-Write Epoch)

- Modified: cache is only cache with a valid copy of line (it can safely be written to)
- Owner: cache is responsible for propagating information to other processors when they attempt to load it from memory (otherwise a load from another processor will get stale data from memory)

Cache Coherence Protocol

Algorithm that maintains cache coherent invariants

The logic we are about to describe is performed by each processor's cache controller in response to:

- **Loads and stores by the local processor**
- **Messages from other caches on the bus**

If all cache controllers operate according to this described protocol, then coherence will be maintained

- **The caches "cooperate" to ensure coherence is maintained**

Invalidation-based write-back protocol

Key ideas:

A line in the "modified" state can be modified without notifying the other caches

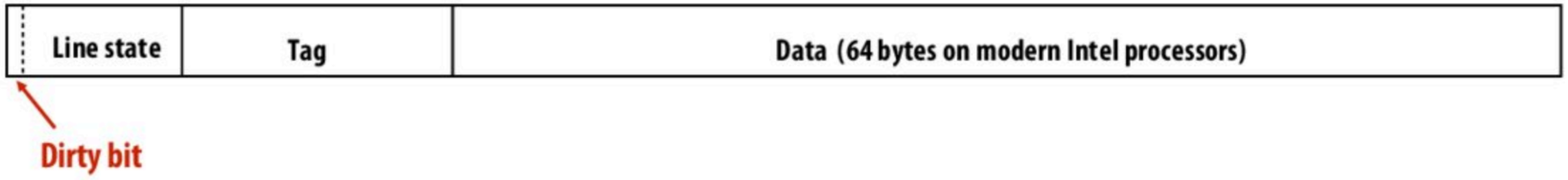
Processor can only write to lines in the modified state

- Need a way to tell other caches that processor wants exclusive access to the line
- We accomplish this by sending message to all the other caches

When cache controller sees a request for modified access to a line it contains

- It must invalidate the line in its cache

Recall cache line state bits



MSI write-back invalidation protocol

Key tasks of protocol

- Ensuring processor obtains exclusive access for a write
- Locating most recent copy of cache line's data on cache miss

Three cache line states

- Invalid (I): same as meaning of invalid in uniprocessor cache
- Shared (S): line valid in one or more caches, memory is up to date
- Modified (M): line valid in exactly one cache (a.k.a. "dirty" or "exclusive" state)

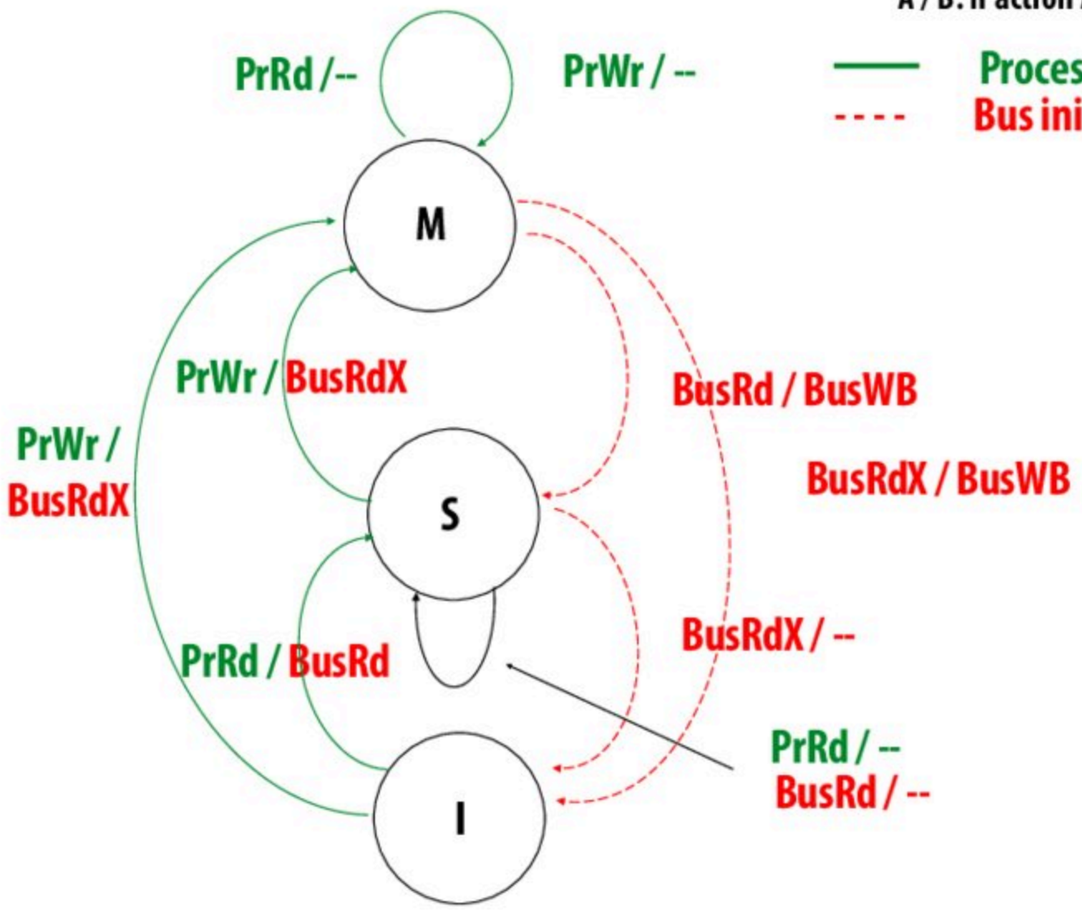
Two processor operations (triggered by local CPU)

- PrRd (read)
- PrWr (write)

Three coherence-related bus transactions (from remote caches)

- BusRd: obtain copy of line with no intent to modify
- BusRdX: obtain copy of line with intent to modify
- BusWB: write dirty line out to memory

Cache Coherence Protocol: MSI State Transition Diagram



Abbreviation	Action
PrRd	Processor Read
PrWr	Processor Write
BusRd	Bus Read
BusRdX	Bus Read Exclusive
BusWB	Bus Writeback

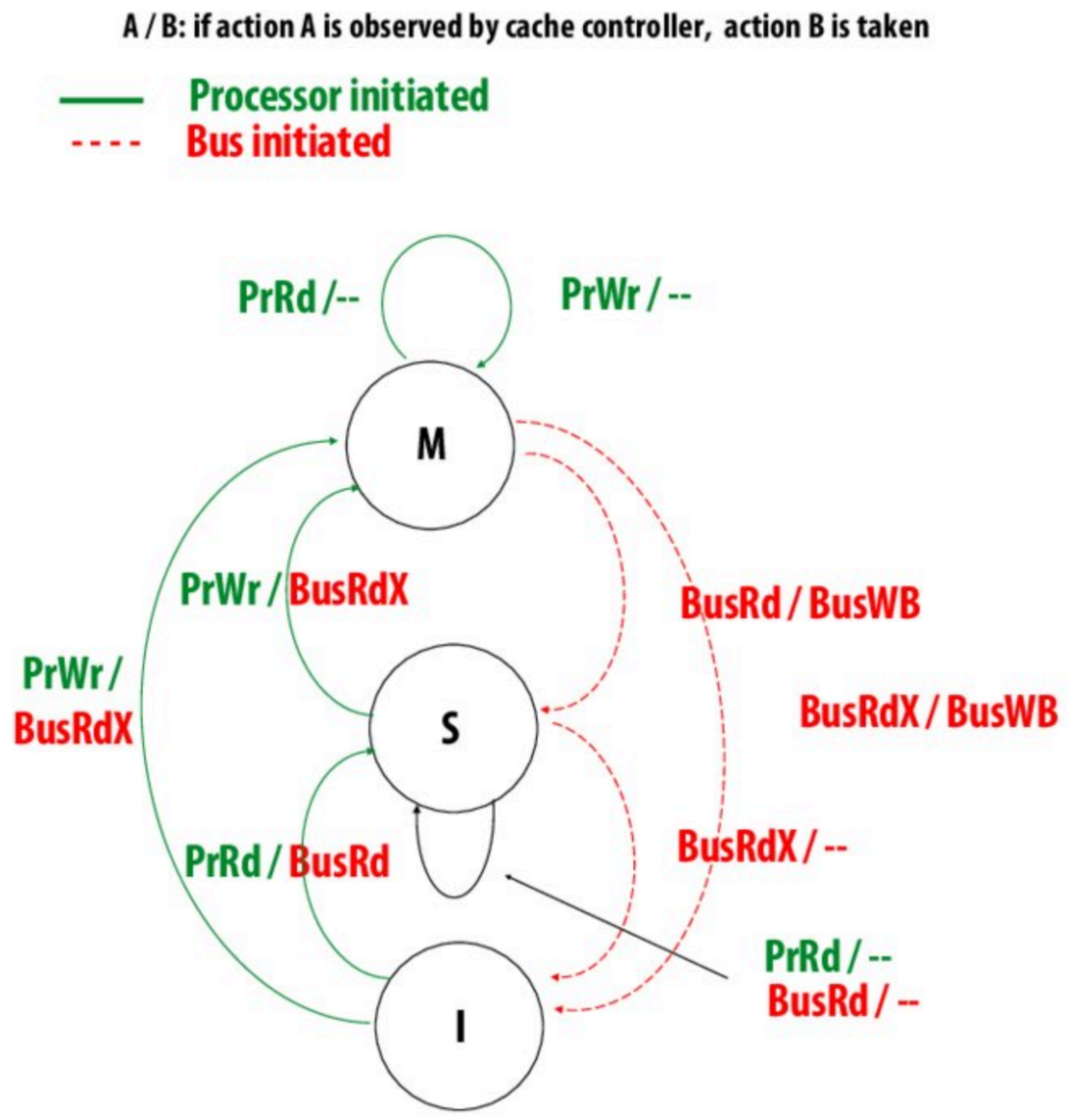
MSI Invalidate Protocol

Read obtains block in "shared"

- even if only cached copy

Obtain exclusive ownership before writing

- BusRdX causes others to invalidate
- If M in another cache, will cause writeback
- BusRdX even if hit in S
  - promote to M (upgrade)



* Remember, all caches are carrying out this logic independently to maintain coherence

A Cache Coherence Example

Proc Action	P1 $-state	P2 $-state	P3 $-state	Bus Trans	Data from
P1 read x	S	--	--	BusRd	Memory
P3 read x	S	--	S	BusRd	Memory
P3 write x	I	--	M	BusRdX	Memory
P1 read x	S	--	S	BusRd	P3 $
P1 read x	S	--	S		P1 $
P2 write x	I	M	I	BusRdX	Memory

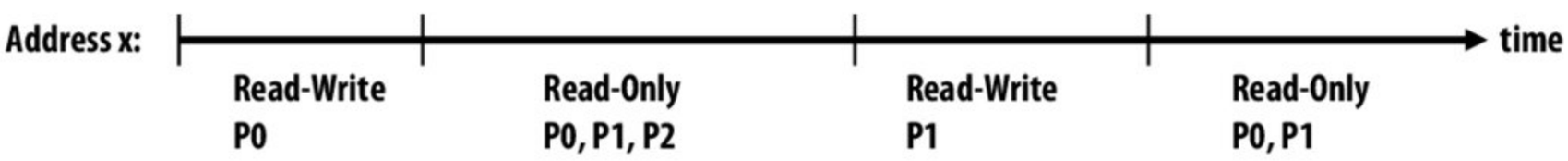
Single writer, multiple reader protocol

Why do you need Modified to Shared?

Communication increases memory latency

How Does MSI Satisfy Cache Coherence?

1. **Single-Writer, Multiple-Read (SWMR) Invariant**
2. **Data-Value Invariant (write serialization)**



Summary: MSI

A line in the M state can be modified without notifying other caches

- No other caches have the line resident, so other processors cannot read these values
- (without generating a memory read transaction)

Processor can only write to lines in the M state

- If processor performs a write to a line that is not exclusive in cache, cache controller must first broadcast a read-exclusive transaction to move the line into that state
- Read-exclusive tells other caches about impending write

("you can't read any more, because I'm going to write")

- Read-exclusive transaction is required even if line is valid (but not exclusive… it's in the S state) in processor's local cache (why?)
- Dirty state implies exclusive

When cache controller snoops a "read exclusive" for a line it contains

- Must invalidate the line in its cache
- Because if it didn't, then multiple caches will have the line

  (and so it wouldn't be exclusive in the other cache!)

MESI invalidation protocol

MSI requires two interconnect transactions for the common case of reading an address, then writing to it

- **Transaction 1: BusRd to move from I to S state**
- **Transaction 2: BusRdX to move from S to M state**

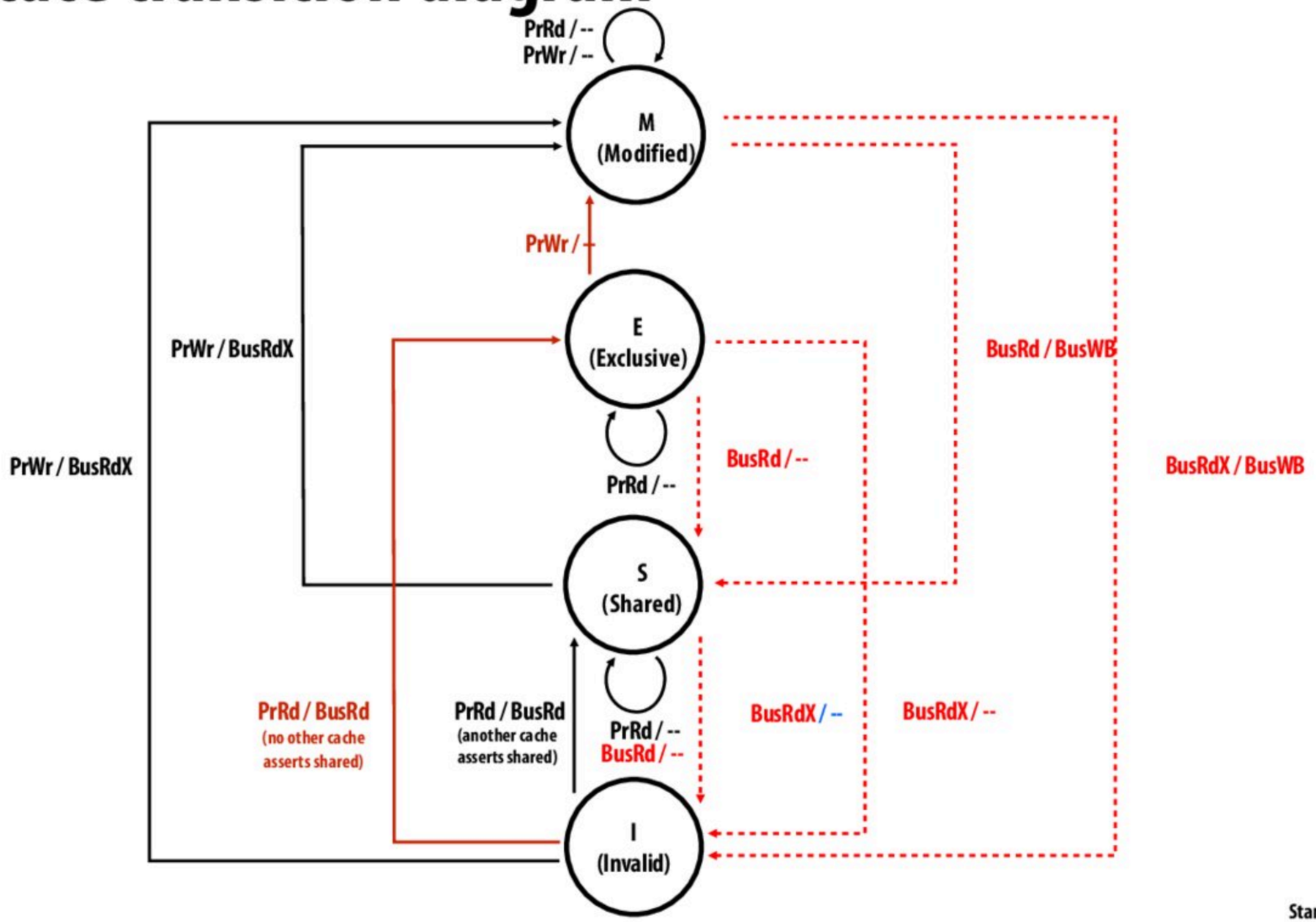
MESI, not Messi!

This inefficiency exists even if application has no sharing at all

Solution: add additional state E ("exclusive clean")

- **Line has not been modified, but only this cache has a copy of the line**
- **Decouples exclusivity from line ownership (line not dirty, so copy in memory is valid copy of data)**
- **Upgrade from E to M does not require an bus transaction**

MESI state transition diagram



Scalable cache coherence using directories

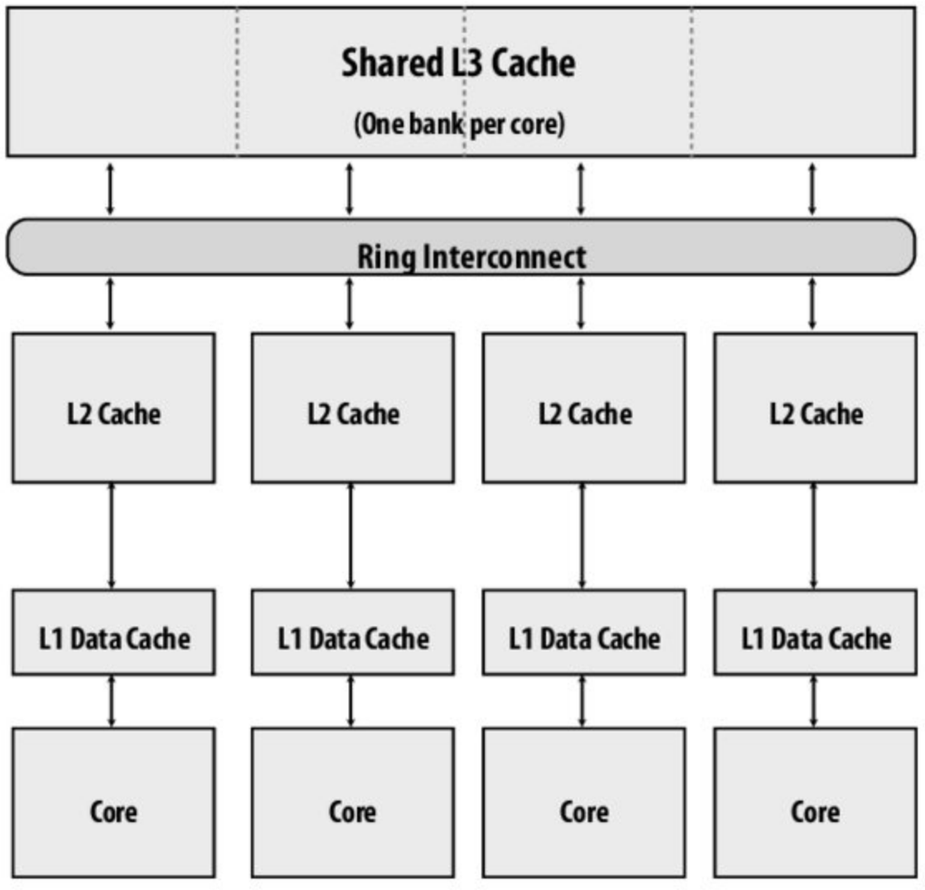
Snooping schemes broadcast coherence messages to determine the state of a line in the other caches: not scalable

Alternative idea: avoid broadcast by storing information about the status of the line in one place: a "directory"

- **The directory entry for a cache line contains information about the state of the cache line in all caches.**
- **Caches look up information from the directory as necessary**
- **Cache coherence is maintained by point-to-point messages between the caches on a "need to know" basis (not by broadcast mechanisms)**

- **Still need to maintain invariants**
  - **SWMR**
  - **Write serialization**

Directory coherence in Intel Core i7 CPU



L3 serves as centralized directory for all lines in the L3 cache

- **Serialization piont**

(Since L3 is an inclusive cache, any line in L2 is guaranteed to also be resident in L3)

Directory maintains list of L2 caches containing line
Instead of broadcasting coherence traffic to all L2's, only send coherence messages to L2's that contain the line

(Core i7 interconnect is a ring, it is not a bus)

Directory dimensions:

- **P=4**
- **M = number of L3 cache lines**

Implications of cache coherence to the programmer

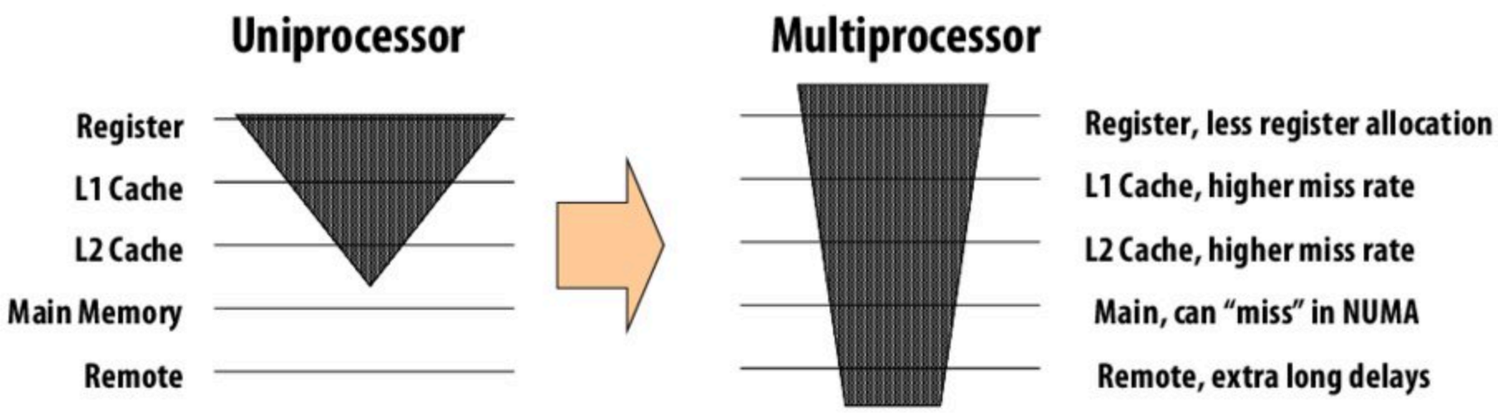
Communication Overhead

Communication time is a key parallel overhead

- **Appears as increased memory access time in multiprocessor**
  - **Extra main memory accesses in UMA systems**
    - **Must determine increase in cache miss rate vs. uniprocessor**
  - **Some accesses have higher latency in NUMA systems**
    - **Only a fraction of a % of these can be significant!**

$$AMAT_{Multiprocessor} > AMAT_{Uniprocessor}$$

Average Memory Access Time (AMAT) = $\sum_0^n$ frequency of access × latency of access



Core i7 Xeon 5500 Series Data Source Latency (approx.)

L1 hit, ~4 cycles
L2 hit, ~10 cycles
L3 hit, line unshared ~40 cycles
L3 hit, shared line in another core ~65 cycles
L3 hit, modified in another core ~75 cycles remote
Local DRAM ~30 ns (~120 cycles)
Remote DRAM ~100 ns (~400 cycles)

Use VTune to learn about memory system performance

Memory Access Analysis for Cache Misses and High Bandwidth Issues

Use the Intel® VTune™ Profiler's Memory Access analysis to identify memory-related issues, like NUMA problems and bandwidth-limited accesses, and attribute performance events to memory objects (data structures), which is provided due to instrumentation of memory allocations/de-allocations and getting static/global variables from symbol information.

NOTE:
Intel® VTune™ Profiler is a new renamed version of the Intel® VTune™ Amplifier.

How It Works

Grouping: Bandwidth Domain / Bandwidth Utilization Type / Function / Call Stack

Bandwidth Domain / Bandwidth Utilization Type / Function / Call Stack	CPU Time	Memory Bound	Loads	Stores	LLC Miss Count	Average Latency (cycles)
DRAM, GB/sec	9.703s	64.3%	6.517.0...	4.141.26...	191.811.508	92
High	4.253s	56.8%	2.345.0...	2.111.23...	119.007.140	115
[Total]	4.177s	54.6%	2.345.0...	2.111.23...	119.007.140	115
__intel_sse2_rep_memcpy	0.177s	100.0%	175.000...	63.000.945	0	223
__do_softirq	0.012s	0.0%	0	0	0	0
run_timer_softirq	0.001s	0.0%	0	0	0	0
__do_page_fault	0.001s	0.0%	0	0	0	0
numa_migrate_prep	0.001s	0.0%	0	0	0	0
task_cputime	0s	0.0%	0	1.409.821	0	0
Medium	2.880s	70.3%	2.765.0...	981.414...	52.853.171	83

Memory Access analysis type uses hardware event-based sampling to collect data for the following metrics:

- **Loads** and **Stores** metrics that show the total number of loads and stores
- **LLC Miss Count** metric that shows the total number of last-level cache misses
  - **Local DRAM Access Count** metric that shows the total number of LLC misses serviced by the local memory
  - **Remote DRAM Access Count** metric that shows the number of accesses to the remote socket memory
  - **Remote Cache Access Count** metric that shows the number of accesses to the remote socket cache
- **Memory Bound** metric that shows a fraction of cycles spent waiting due to demand load or store instructions
  - **L1 Bound** metric that shows how often the machine was stalled without missing the L1 data cache
  - **L2 Bound** metric that shows how often the machine was stalled on L2 cache
  - **L3 Bound** metric that shows how often the CPU was stalled on L3 cache, or contended with a sibling core
  - **L3 Latency** metric that shows a fraction of cycles with demand load accesses that hit the L3 cache under unloaded scenarios (possibly L3 latency limited)
  - **NUMA: % of Remote Accesses** metric shows percentage of memory requests to remote DRAM. The lower its value is, the better.
  - **DRAM Bound** metric that shows how often the CPU was stalled on the main memory (DRAM). This metric enables you to identify **DRAM Bandwidth Bound**, **UPI Utilization Bound** issues, as well as **Memory Latency** issues with the following metrics:
    - **Remote / Local DRAM Ratio** metric that is defined by the ratio of remote DRAM loads to local DRAM loads
    - **Local DRAM** metric that shows how often the CPU was stalled on loads from the local memory
    - **Remote DRAM** metric that shows how often the CPU was stalled on loads from the remote memory
    - **Remote Cache** metric that shows how often the CPU was stalled on loads from the remote cache in other sockets
- **Average Latency** metric that shows an average load latency in cycles

Unintended communication via false sharing

What is the potential performance problem with this code?

```
// allocate per-thread variable for local per-thread accumulation
int myPerThreadCounter[NUM_THREADS];
```

Why might this code be more performant?

```
// allocate per thread variable for local accumulation
struct PerThreadState {
  int myPerThreadCounter;
  char padding[CACHE_LINE_SIZE - sizeof(int)];
};
PerThreadState myPerThreadCounter[NUM_THREADS];
```

Demo: false sharing

```
void* worker(void* arg) {

    volatile int* counter = (int*)arg;

    for (int i=0; i<MANY_ITERATIONS; i++)
        (*counter)++;

    return NULL;
}
```

threads update a per-thread counter many times

```
void test1(int num_threads) {

    pthread_t threads[MAX_THREADS];
    int       counter[MAX_THREADS];

    for (int i=0; i<num_threads; i++)
         pthread_create(&threads[i], NULL,
                        &worker, &counter[i]);

     for (int i=0; i<num_threads; i++)
         pthread_join(threads[i], NULL);
}
```

Execution time with num_threads=8 on 4-core system: 14.2 sec

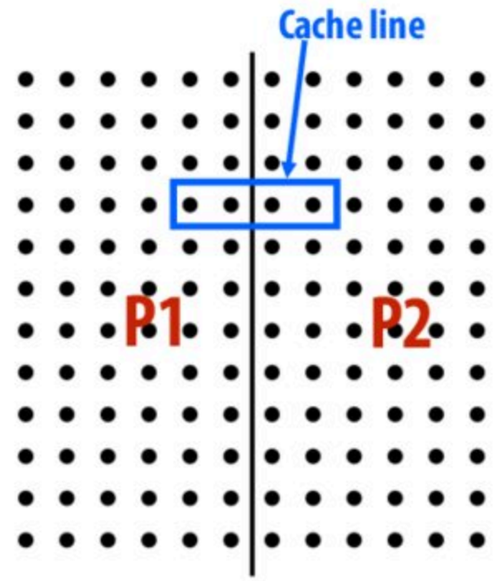
```
struct padded_t {
    int  counter;
    char padding[CACHE_LINE_SIZE - sizeof(int)];
};

void test2(int num_threads) {

    pthread_t threads[MAX_THREADS];
    padded_t  counter[MAX_THREADS];


    for (int i=0; i<num_threads; i++)
        pthread_create(&threads[i], NULL,
                       &worker, &(counter[i].counter));

    for (int i=0; i<num_threads; i++)
        pthread_join(threads[i], NULL);
}
```

Execution time with num_threads=8 on 4-core system: 4.7 sec

False sharing

Condition where two processors write to different addresses, but addresses map to the same cache line

Cache line "ping-pongs" between caches of writing processors, generating significant amounts of communication due to the coherence protocol

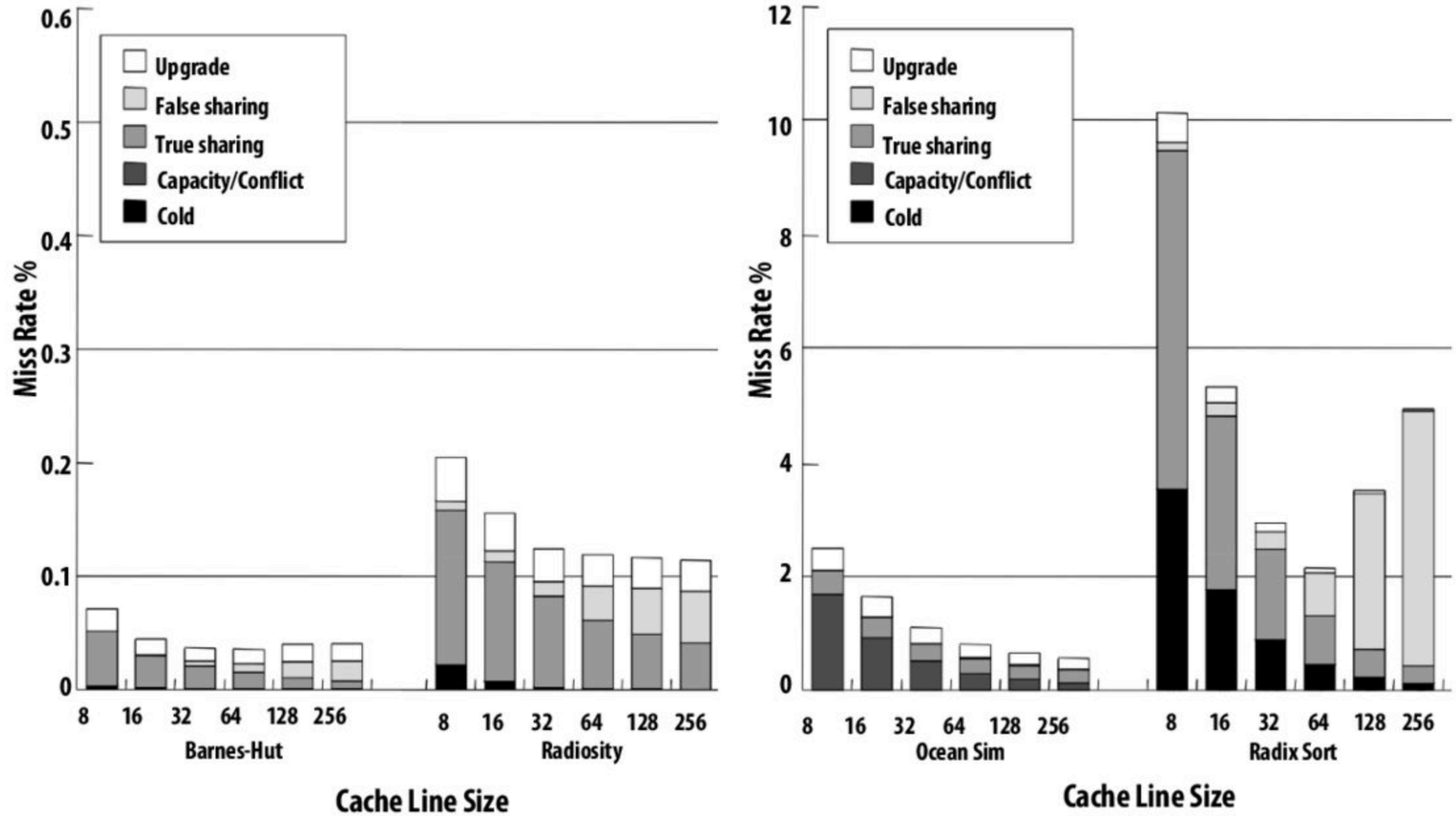


No inherent communication, this is entirely artifactual communication (cachelines > 4B)

False sharing can be a factor in when programming for cache-coherent architectures

Impact of cache line size on miss rate

Results from simulation of a 1 MB cache (four example applications)



* Note: I separated the results into two graphs because of different Y-axis scales

Figure credit: Culler, Singh, and Gupta

Summary: Cache coherence

The cache coherence problem exists because the abstraction of a single shared address space is not implemented by a single storage unit

- Storage is distributed among main memory and local processor caches
- Data is replicated in local caches for performance

Main idea of snooping-based cache coherence: whenever a cache operation occurs that could affect coherence, the cache controller broadcasts a notification to all other cache controllers in the system

- Challenge for HW architects: minimizing overhead of coherence implementation
- Challenge for SW developers: be wary of artifactual communication due to coherence protocol (e.g., false sharing)

Scalability of snooping implementations is limited by ability to broadcast coherence messages to all caches!

- Scaling cache coherence via directory-based approaches