Lecture 17: Memory Hierarchy and Cache Coherence

Concurrent and Multicore Programming

Department of Computer Science and Engineering
Yonghong Yan
yan@oakland.edu
www.secs.oakland.edu/~yan

Parallelism in Hardware

- Instruction-Level Parallelism
  - Pipeline
  - Out-of-order execution, and
  - Superscalar

- Thread-Level Parallelism
  - Chip multithreading, multicore
  - Coarse-grained and fine-grained multithreading
  - SMT

- Data-Level Parallelism
  - SIMD/Vector
  - GPU/SIMT

Computer Architecture, A Quantitative Approach. 5TH Edition, The Morgan Kaufmann, September 30, 2011 by John L. Hennessy (Author), David A. Patterson

Topics (Part 2)

- Parallel architectures and hardware
  - Parallel computer architectures
  - ☛ **Memory hierarchy and cache coherency**
- Manycore GPU architectures and programming
  - **GPUs architectures**
  - **CUDA programming**
  - Introduction to offloading model in OpenMP and OpenACC
- Programming on large scale systems (Chapter 6)
  - **MPI (point to point and collectives)**
  - Introduction to PGAS languages, UPC and Chapel
- Parallel algorithms (Chapter 8,9 &10)
  - **Dense matrix, and sorting**

Outline

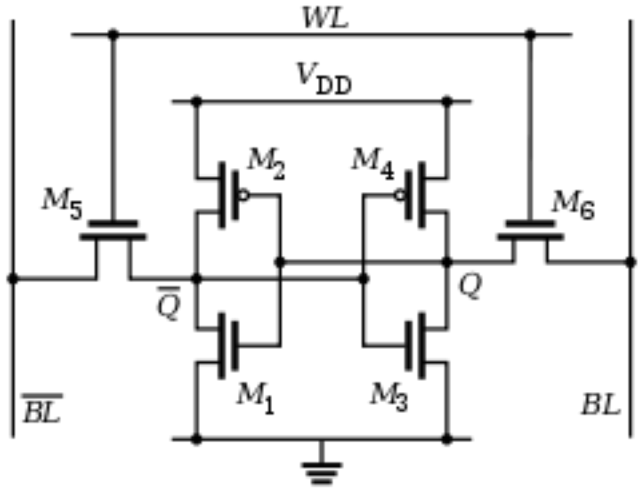
- **Memory, Locality of reference and Caching**
- Cache coherence in shared memory system

Memory until now …

- We've relied on a very simple model of memory for most this class
  - Main Memory is a linear array of bytes that can be accessed given a memory address
  - Also used registers to store values
- Reality is more complex. There is an entire memory system.
  - Different memories exist at different levels of the computer
  - Each vary in their speed, size, and cost

Random-Access Memory (RAM)

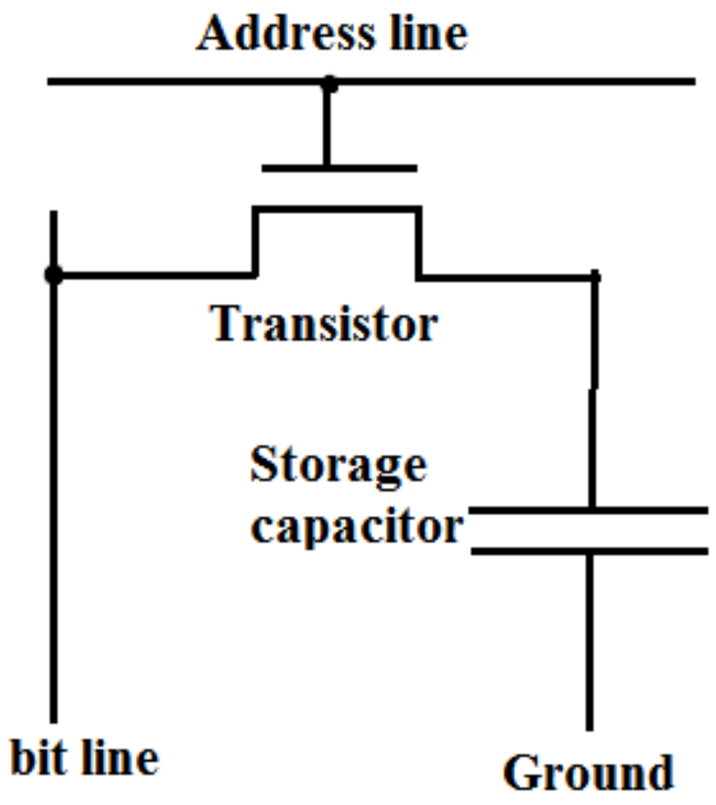
- Key features
  - RAM is packaged as a chip.
  - Basic storage unit is a cell (one bit per cell).
  - Multiple RAM chips form a memory.



- **Static RAM (SRAM)**
  - Each cell stores bit with a six-transistor circuit.
  - Retains value indefinitely, as long as it is kept powered.
  - Relatively insensitive to disturbances such as electrical noise.
  - Faster and more expensive than DRAM.

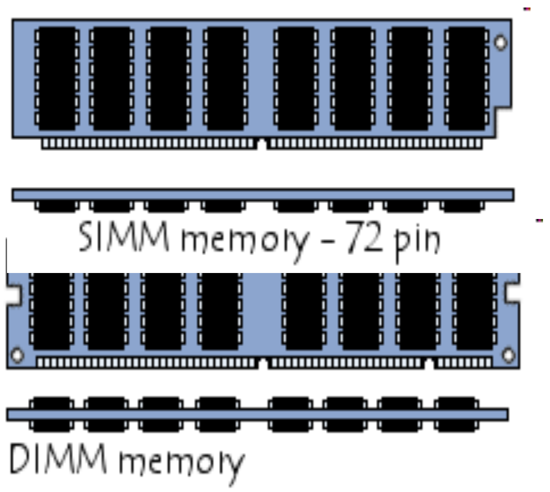
Random-Access Memory (RAM)

- **Dynamic RAM (DRAM)**
  - Each cell stores bit with a capacitor and transistor.
  - Value must be refreshed every 10-100 ms.
  - Sensitive to disturbances.
  - Slower and cheaper than SRAM.



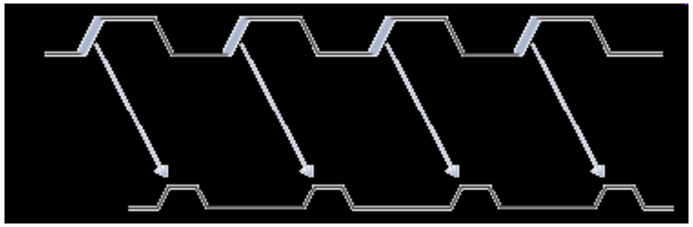
Memory Modules… real life DRAM

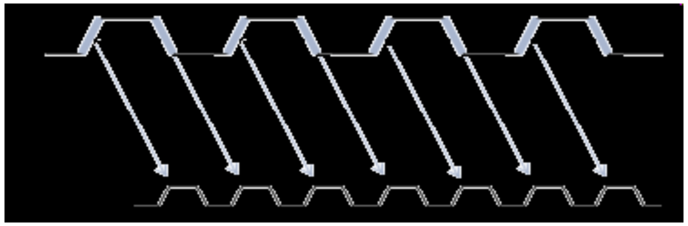
- In reality,
  - Several DRAM chips are bundled into Memory Modules
- SIMMS - Single Inline Memory Module
- DIMMS - Dual Inline Memory Module
- DDR- Dual data Read
  - Reads twice every clock cycle
- Quad Pump: Simultaneous R/ W

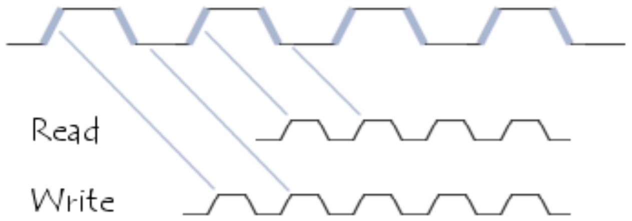


Source for Pictures:
http://en.kioskea.net/contents/pc/ram.php3

SDR, DDR, Quad Pump





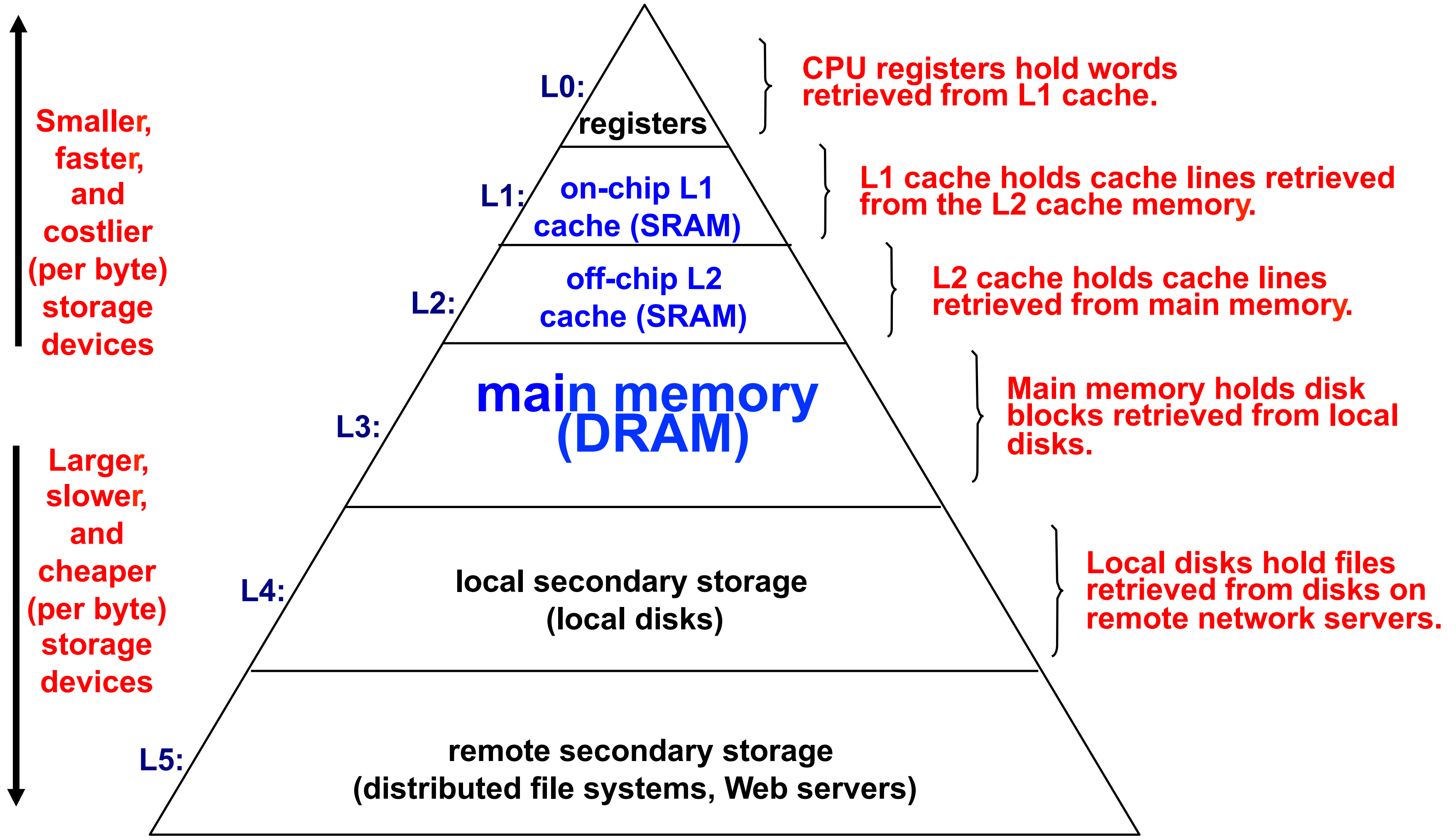


Memory Speeds

- Processor Speeds : 1 GHz processor speed is 1 nsec cycle time.
- Memory Speeds (50 nsec)
- Access Speed gap
  - Instructions that store or load from memory

DIMM Module Chip Type	Clock Speed (MHz)	Bus Speed (MHz)	Transfer Rate (MB/s)
PC1600 DDR200	100	200	1600
PC2100 DDR266	133	266	2133
PC2400 DDR300	150	300	2400

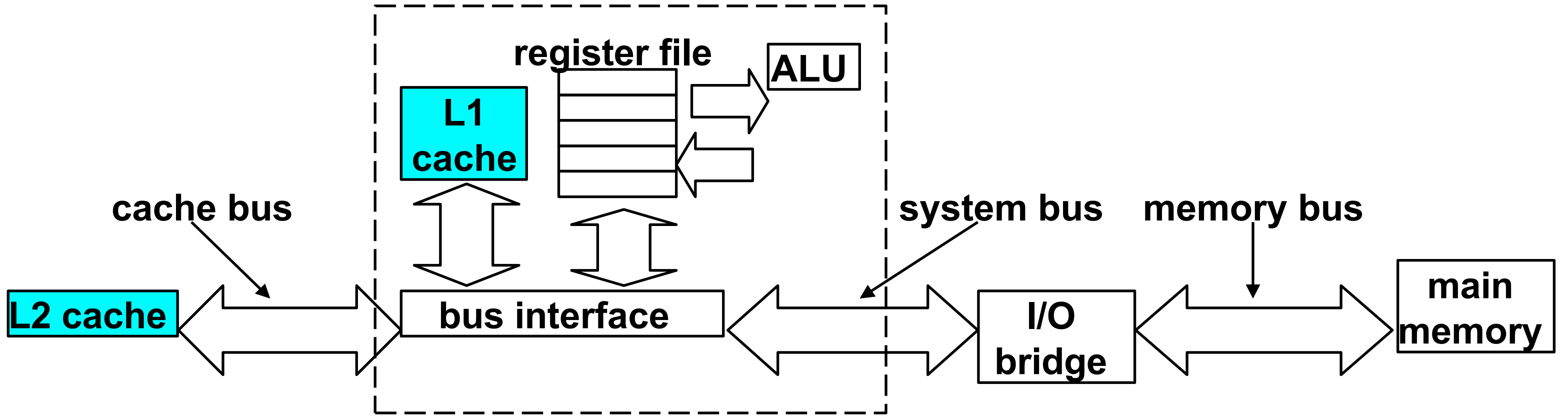
Memory Hierarchy (Review)



Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0:
registers
L1:
on-chip L1 cache (SRAM)
L2:
off-chip L2 cache (SRAM)
L3:
main memory (DRAM)
L4:
local secondary storage (local disks)
L5:
remote secondary storage (distributed file systems, Web servers)
CPU registers hold words retrieved from L1 cache.
L1 cache holds cache lines retrieved from the L2 cache memory.
L2 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 network servers.

Cache Memories (SRAM)

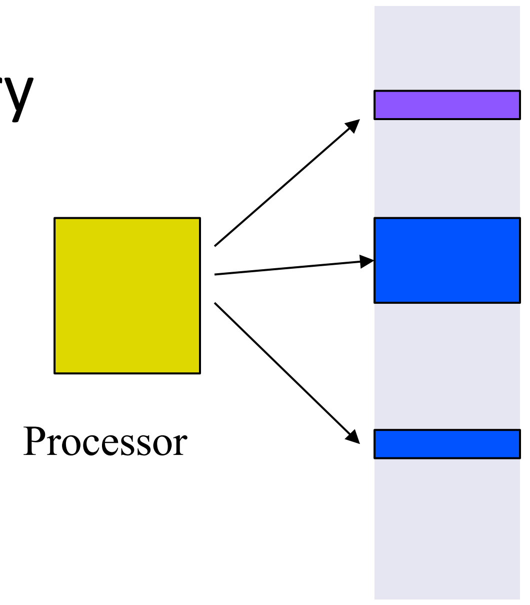
- Cache memories are small, fast SRAM-based memories managed automatically in hardware.
  - Hold frequently accessed blocks of main memory
- CPU looks first for data in L1, then in L2, then in main memory.
- Typical bus structure:



How to Exploit Memory Hierarchy

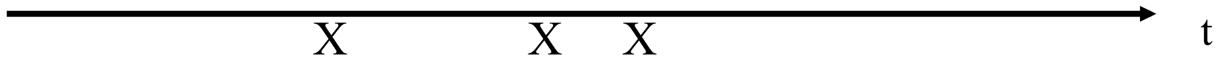
- Availability of memory
  - Cost, size, speed
- Principle of locality
  - Memory references are bunched together
  - A small portion of address space is accessed at any given time

- This space in high speed memory
  - Problem: not all of it may fit

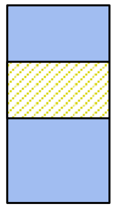


Types of locality

- Temporal locality
  - Tendency to access locations recently referenced



- Spatial locality
  - Tendency to reference locations around recently referenced
  - Location x , then others will be x-k or x+k



Sources of locality

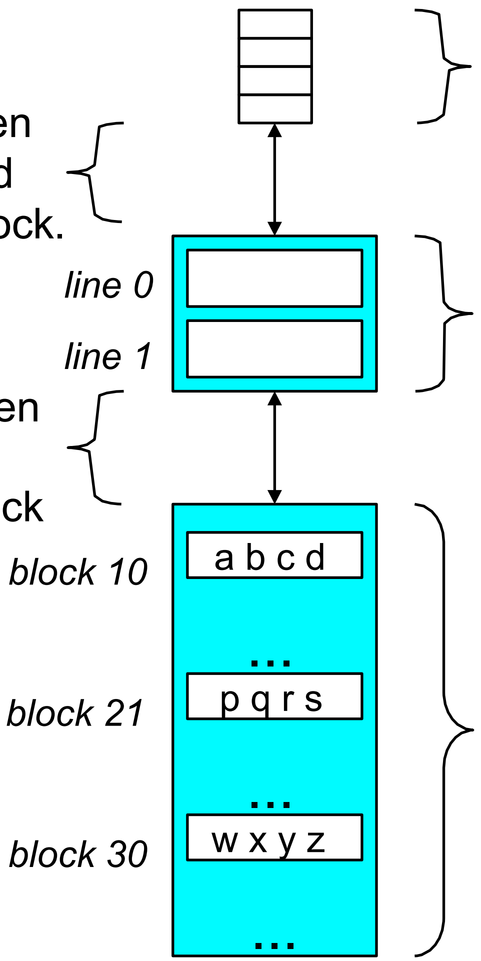
- Temporal locality
  - Code within a loop
  - Same instructions fetched repeatedly
- Spatial locality
  - Data arrays
  - Local variables in stack
  - Data allocated in chunks (contiguous bytes)

```
for (i=0; i<N; i++) {
   A[i] = B[i] + C[i] * a;
}
```

What does locality buy?

- Address the gap between CPU speed and RAM speed
- Spatial and temporal locality implies a subset of instructions can fit in high speed memory from time to time
- CPU can access instructions and data from this high speed memory
- Small high speed memory can make computer faster and cheaper
- Speed of 1-20 nsec at cost of $50 to $100 per Mbyte
- This is Caching!!

Inserting an L1 Cache Between CPU and Main Memory



The tiny, very fast CPU **register file** has room for four 4-byte words.

The transfer unit between the CPU register file and the cache is a 4-byte block.

The small fast **L1 cache** has room for two 4-word blocks.

The transfer unit between the cache and main memory is a 4-word block (16 bytes).

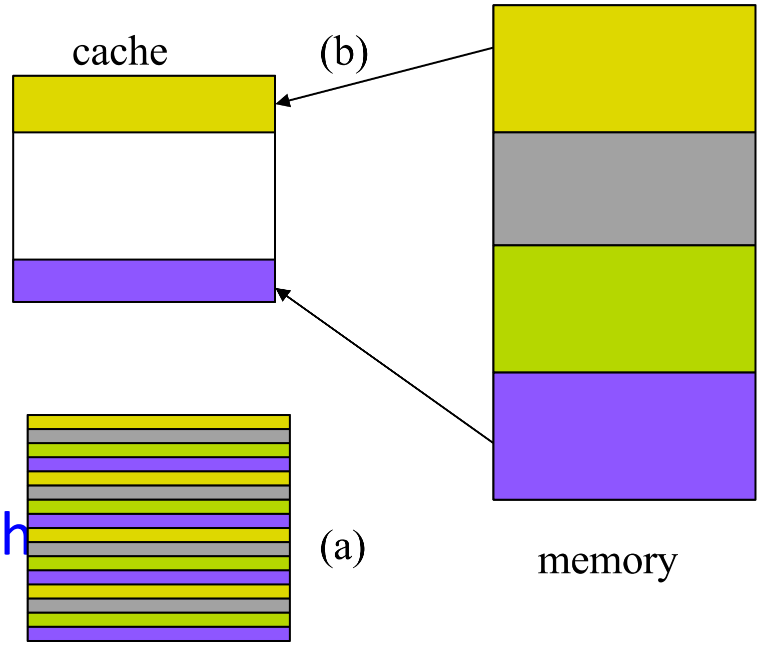
The big slow **main memory** has room for many 4-word blocks.

What info. Does a cache need

- Cache: A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.
- You essentially allow a smaller region of memory to hold data from a larger region. Not a 1-1 mapping.
- What kind of information do we need to keep:
  - The actual data
  - Where the data actually comes from
  - If data is even considered valid

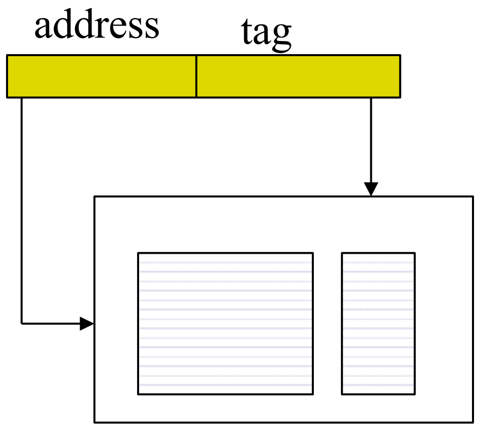
Cache Organization

- Map each region of memory to a smaller region of cache
- Discard address bits
  - Discard lower order bits (a)
  - Discard higher order bits (b)
- Cache address size is 4 bits
- Memory address size is 8 bits
- In case of a)
  - 0000xxxx is mapped to 0000 in cache
- In case of b)
  - xxxx0001 is mapped to 0001 in cache



Finding data in cache

- Part of memory address applied to cache
- Remaining is stored as tag in cache
- Lower order bits discarded
- Need to check if 00010011
  - Cache index is 0001
  - Tag is 0011
- If tag matches, hit, use data
- No match, miss, fetch data from memory

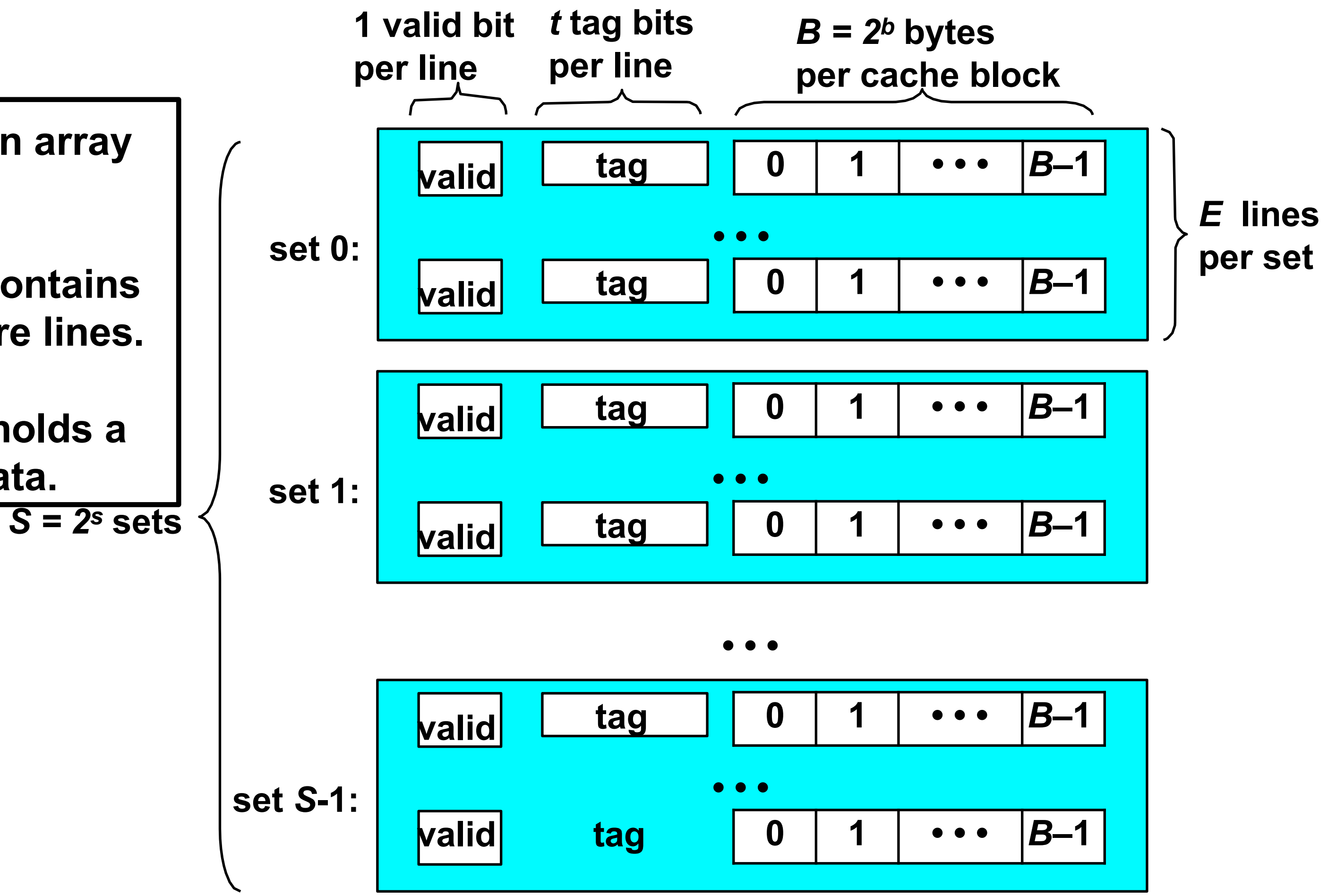


General Org of a Cache Memory

Cache is an array of sets.

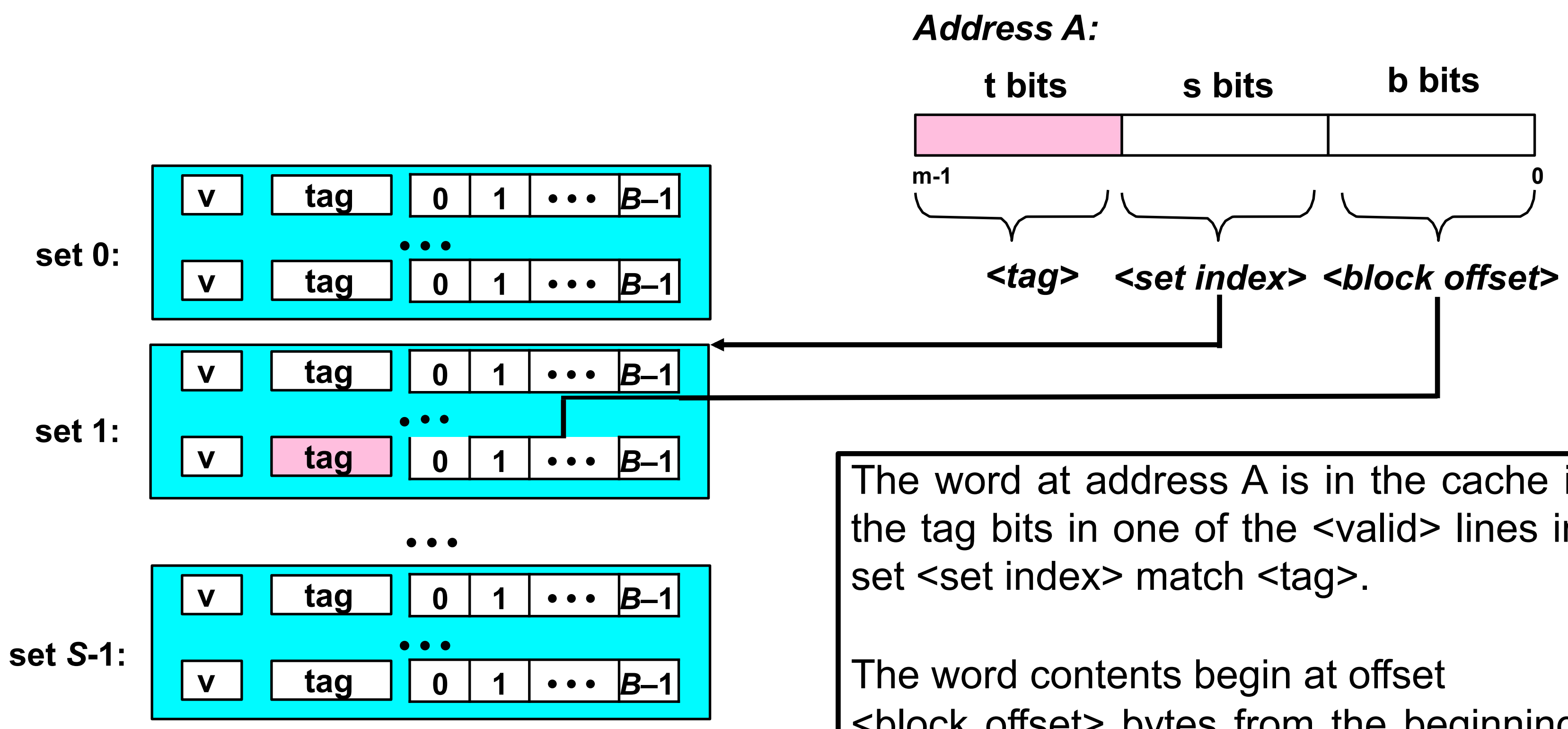
Each set contains one or more lines.

Each line holds a block of data.



Cache size: $C = B \times E \times S$ data bytes

Addressing Caches

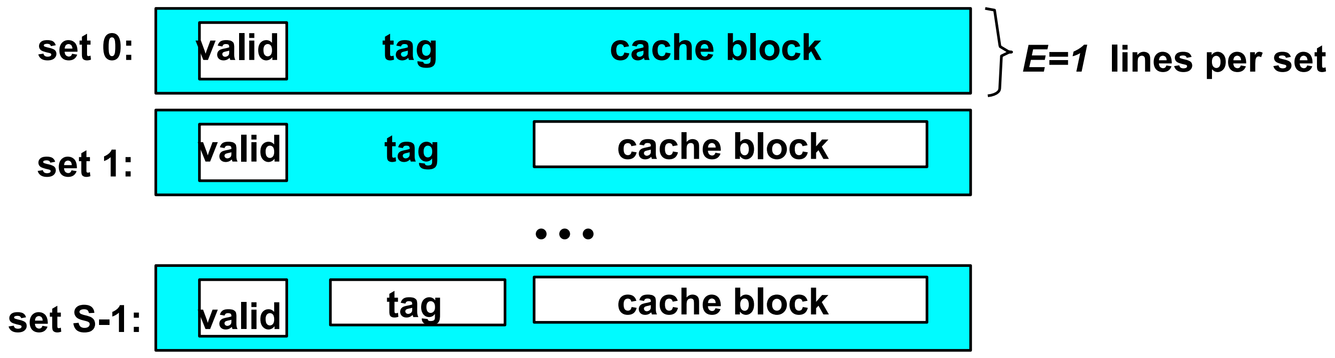


The word at address A is in the cache if the tag bits in one of the <valid> lines in set <set index> match <tag>.

The word contents begin at offset <block offset> bytes from the beginning of the block.

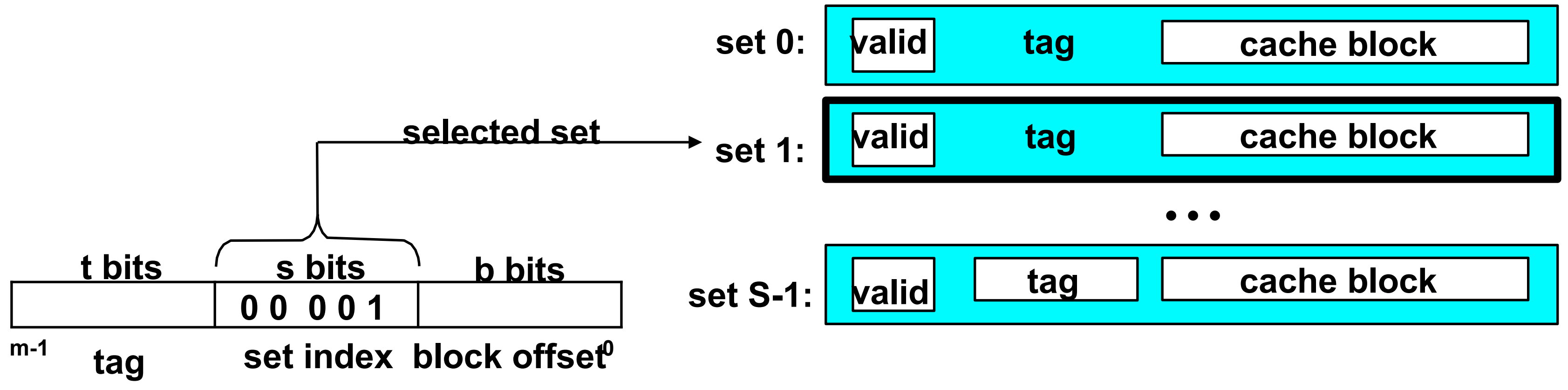
Direct-Mapped Cache

- Simplest kind of cache
- Characterized by exactly one line per set.



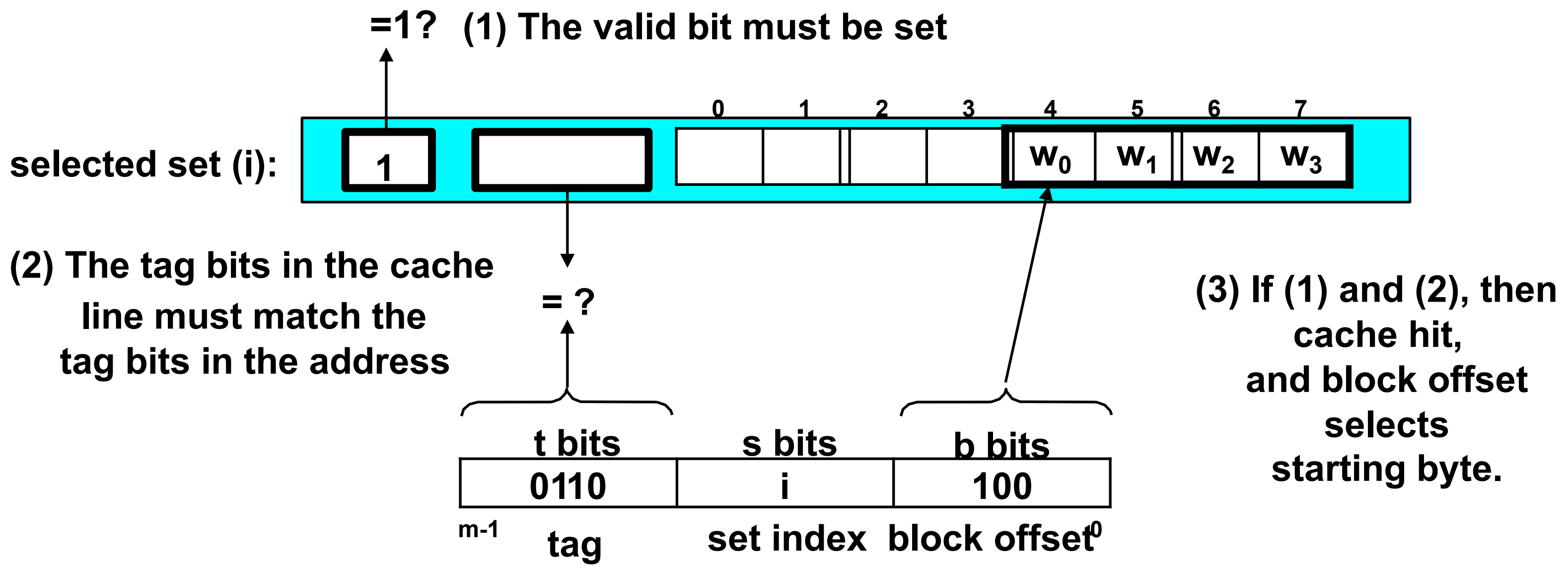
Accessing Direct-Mapped Caches

- Set selection
  - Use the set index bits to determine the set of interest.



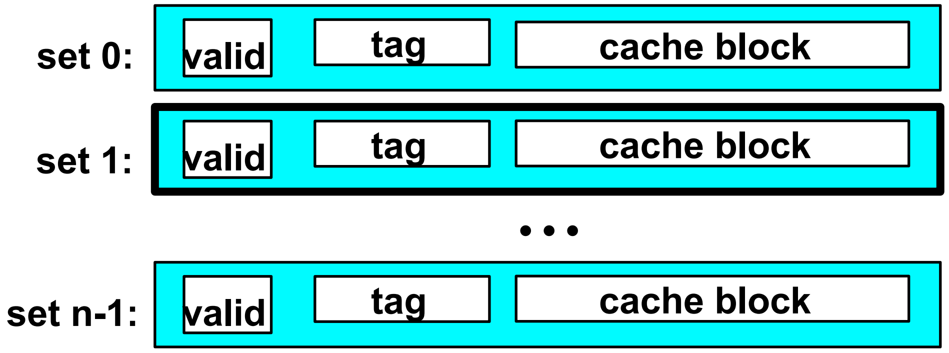
Accessing Direct-Mapped Caches

- Line matching and word selection
  - Line matching: Find a valid line in the selected set with a matching tag
  - Word selection: Then extract the word


=1? (1) The valid bit must be set
selected set (i):
1
0
1
2
3
4
5
6
7
w_0
w_1
w_2
w_3
(2) The tag bits in the cache line must match the tag bits in the address
= ?
(3) If (1) and (2), then cache hit, and block offset selects starting byte.
t bits
s bits
b bits
0110
i
100
m-1
tag
set index
block offset
0

Example: Direct mapped cache

- 32 bit address, 64KB cache, 32 byte block
- How many sets, how many bits for the tag, how many bits for the offset?



Write-through vs write-back

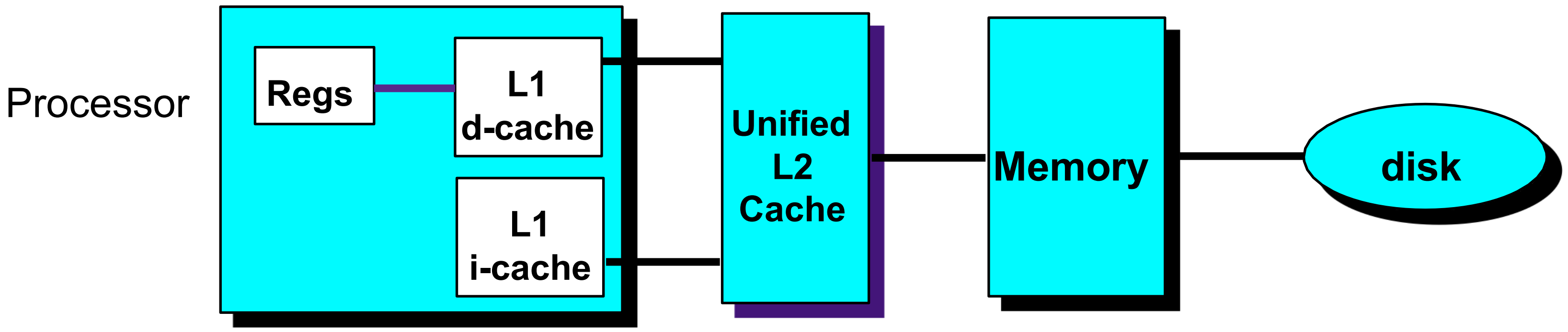
- What to do when an update occurs?
- Write-through: immediately
  - Simple to implement, synchronous write
  - Uniform latency on misses
- Write-back: write when block is replaced
  - Requires additional dirty bit or modified bit
  - Asynchronous writes
  - Non-uniform miss latency
  - Clean miss: read from lower level
  - Dirty miss: write to lower level and read (fill)

Writes and Cache

- Reading information from a cache is straight forward.
- What about writing?
  - What if you're writing data that is already cached (write-hit)?
  - What if the data is not in the cache (write-miss)?
- Dealing with a write-hit.
  - Write-through - immediately write data back to memory
  - Write-back - defer the write to memory for as long as possible
- Dealing with a write-miss.
  - write-allocate - load the block into memory and update
  - no-write-allocate - writes directly to memory
- Benefits? Disadvantages?
- Write-through are typically no-write-allocate.
- Write-back are typically write-allocate.

Multi-Level Caches

- Options: separate data and instruction caches, or a unified cache



size:	200 B	8-64 KB	1-4MB SRAM	128 MB DRAM	30 GB
speed:	3 ns	3 ns	6 ns	60 ns	8 ms
$/Mbyte:			$100/MB	$1.50/MB	$0.05/MB
line size:	8 B	32 B	32 B	8 KB	

larger, slower, cheaper →

Cache Performance Metrics

- Miss Rate
  - Fraction of memory references not found in cache (misses/ references)
  - Typical numbers:
    - 3-10% for L1
    - can be quite small (e.g., < 1%) for L2, depending on size, etc.
- Hit Time
  - Time to deliver a line in the cache to the processor (includes time to determine whether the line is in the cache)
  - Typical numbers:
    - 1 clock cycle for L1
    - 3-8 clock cycles for L2
- Miss Penalty
  - Additional time required because of a miss
    - Typically 25-100 cycles for main memory

Writing Cache Friendly Code

- Repeated references to variables are good (temporal locality)
- Stride-1 reference patterns are good (spatial locality)
- Examples:
  - cold cache, 4-byte words, 4-word cache blocks

```
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Miss rate = 1/4 = 25%

```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Miss rate = 100%

Matrix Multiplication Example

- Major Cache Effects to Consider
  - Total cache size
    - Exploit temporal locality and blocking)
  - Block size
    - Exploit spatial locality

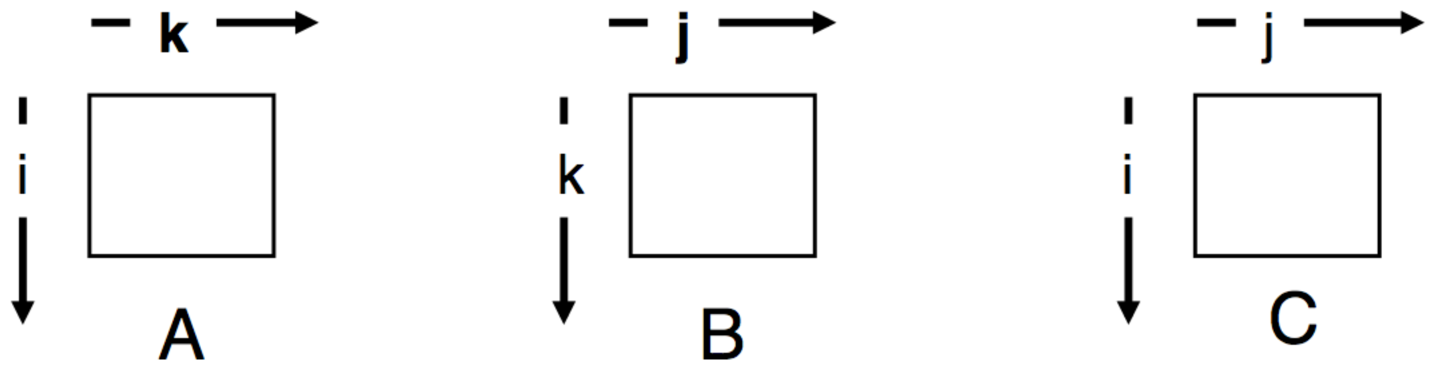
- Description:
  - Multiply N x N matrices
  - O(N^3) total operations
  - Accesses
    - N reads per source element
    - N values summed per destination
      - but may be able to hold in register

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Variable `sum` *held in register*

Miss Rate Analysis for Matrix Multiply

- Assume:
  - Line size = 32BYTES (big enough for 4 64-bit words)
  - Matrix dimension (N) is very large
    - Approximate 1/N as 0.0
  - Cache is not even big enough to hold multiple rows
- Analysis Method:
  - Look at access pattern of inner loop



Layout of C Arrays in Memory (review)

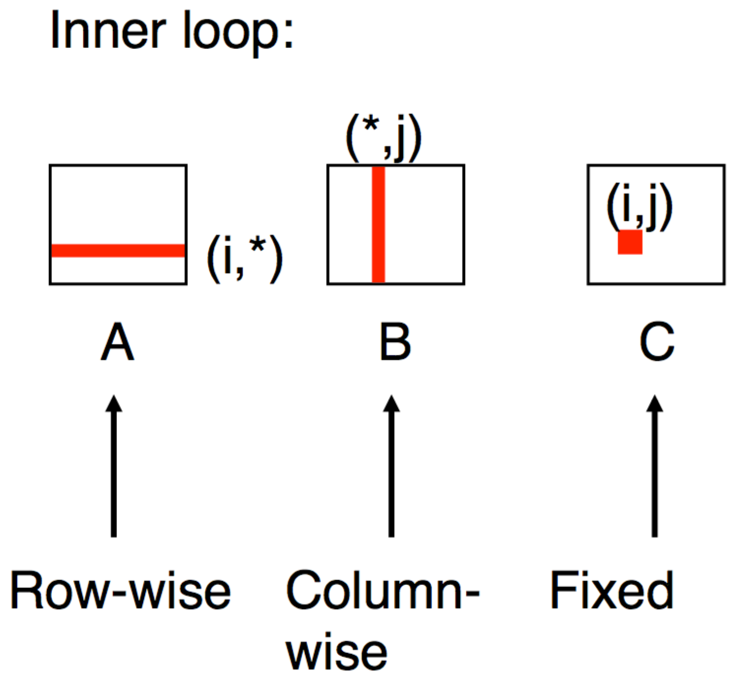
- C arrays allocated in row-major order
  - each row in contiguous memory locations
- Stepping through columns in one row:

  -
    ```
    for(i = 0; i < N; i++)
        sum += a[0][i];
    ```
  - accesses successive elements
  - if block size (B) > 4 bytes, exploit spatial locality
    - compulsory miss rate = 4 bytes / B
- Stepping through rows in one column:

  -
    ```
    for(i = 0; i < n; i++)
       sum += a[i][0];
    ```
- accesses distant elements
- no spatial locality!
  - compulsory miss rate = 1 (i.e. 100%)

Matrix Multiplication (ijk)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

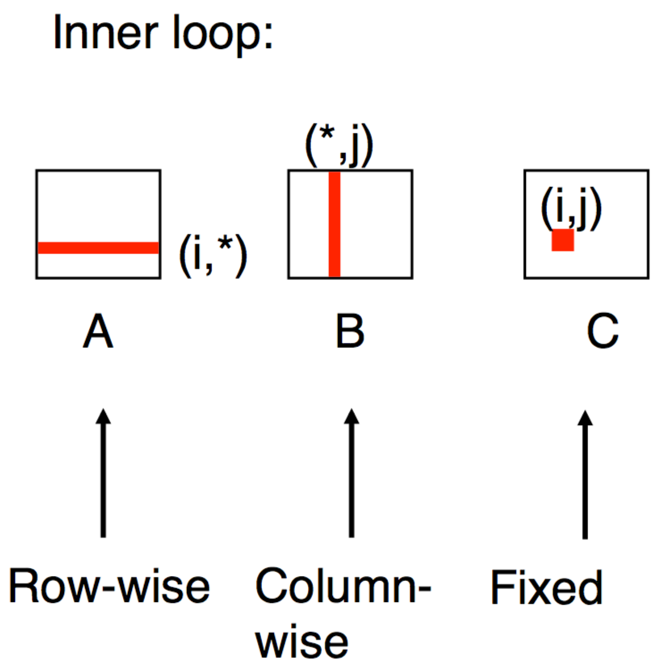


- Misses per Inner Loop Iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (jik)

```
/* jik */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum
  }
}
```

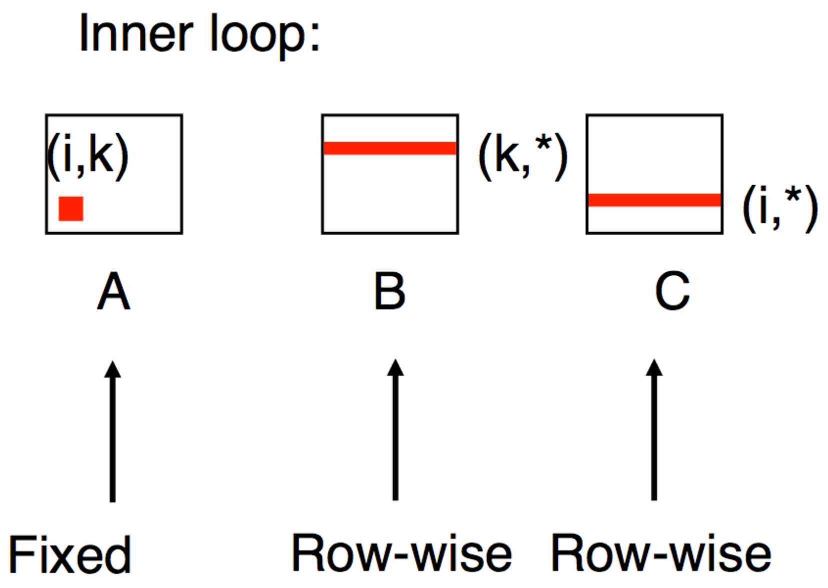


- Misses per Inner Loop Iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (kij)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

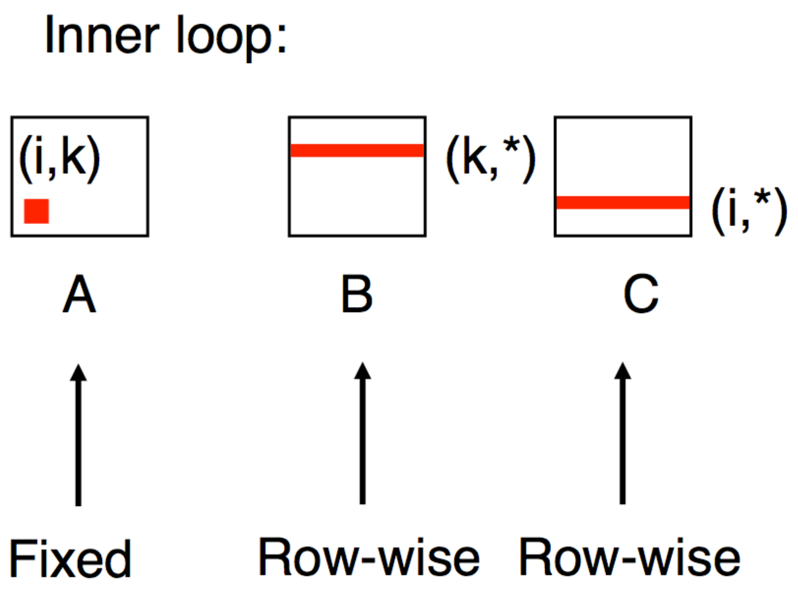


- Misses per Inner Loop Iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (ikj)

```
/* ikj */
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

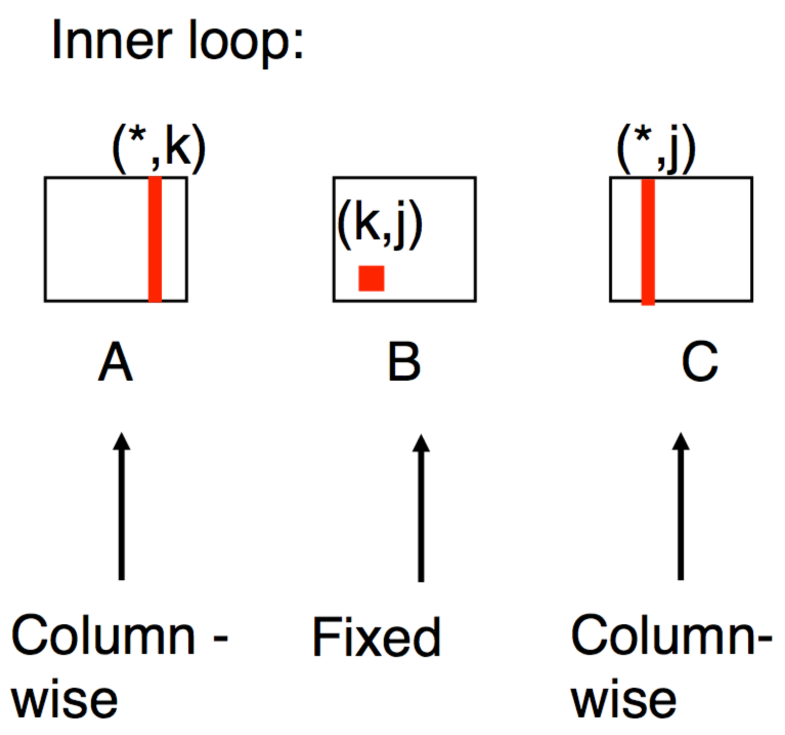


- Misses per Inner Loop Iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (jki)

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

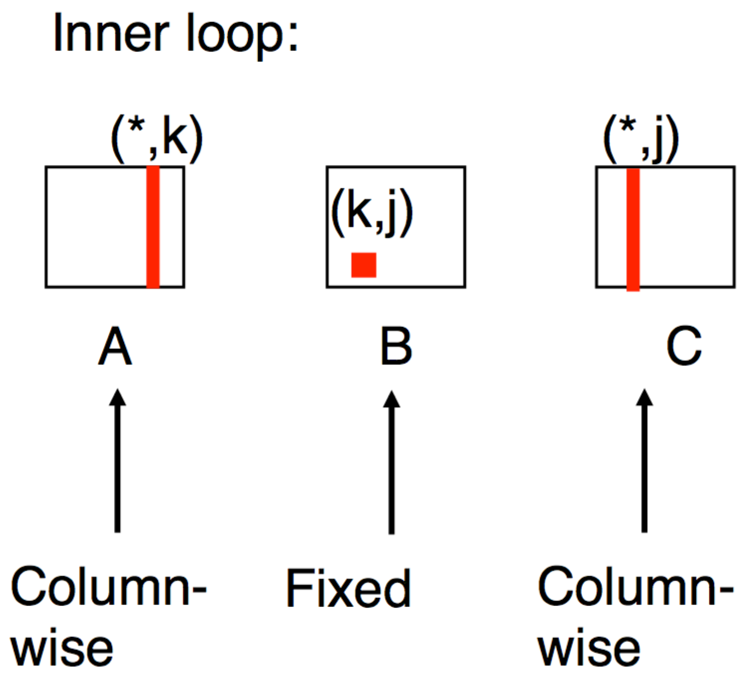


- Misses per Inner Loop Iteration:

A	B	C
1.0	0.0	1.0

Matrix Multiplication (kji)

```
/* kji */
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```



- Misses per Inner Loop Iteration:

A	B	C
1.0	0.0	1.0

Summary of Matrix Multiplication

ijk (& jik):
- 2 loads, 0 stores
- misses/iter = **1.25**

```
for (i=0; i<n; i++)
  { for (j=0; j<n; j++)
  {
     sum = 0.0;
     for (k=0; k<n; k++)
        sum += a[i][k] * b[k][j];
     c[i][j] = sum;
  }
}
```

kij (& ikj):
- 2 loads, 1 store
- misses/iter = **0.5**

```
for (k=0; k<n; k++)
  { for (i=0; i<n; i++)
  {
     r = a[i][k];
     for (j=0; j<n; j++) c[i]
        [j] += r * b[k][j];
  }
}
```

jki (& kji):
- 2 loads, 1 store
- misses/iter = **2.0**

```
for (j=0; j<n; j++) {
   for (k=0; k<n; k++)
      { r = b[k][j];
      for (i=0; i<n; i++) c[i]
         [j] += a[i][k] * r;
   }
}
```

Outline

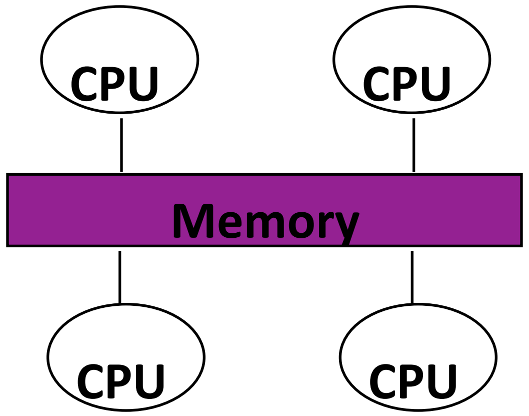
- Memory, Locality of reference and Caching
- **Cache coherence in shared memory system**

Shared memory systems

- All processes have access to the same address space
  - E.g. PC with more than one processor
- Data exchange between processes by writing/reading shared variables
  - Shared memory systems are easy to program
  - Current standard in scientific programming: OpenMP
- Two versions of shared memory systems available today
  - Centralized Shared Memory Architectures
  - Distributed Shared Memory architectures

Centralized Shared Memory Architecture

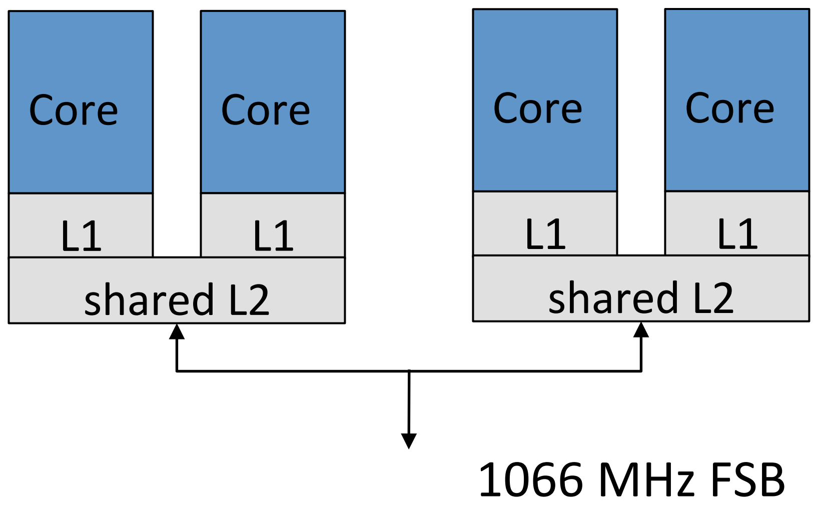
- Also referred to as Symmetric Multi-Processors (SMP)
- All processors share the same physical main memory



- Memory bandwidth per processor is limiting factor for this type of architecture
- Typical size: 2-32 processors

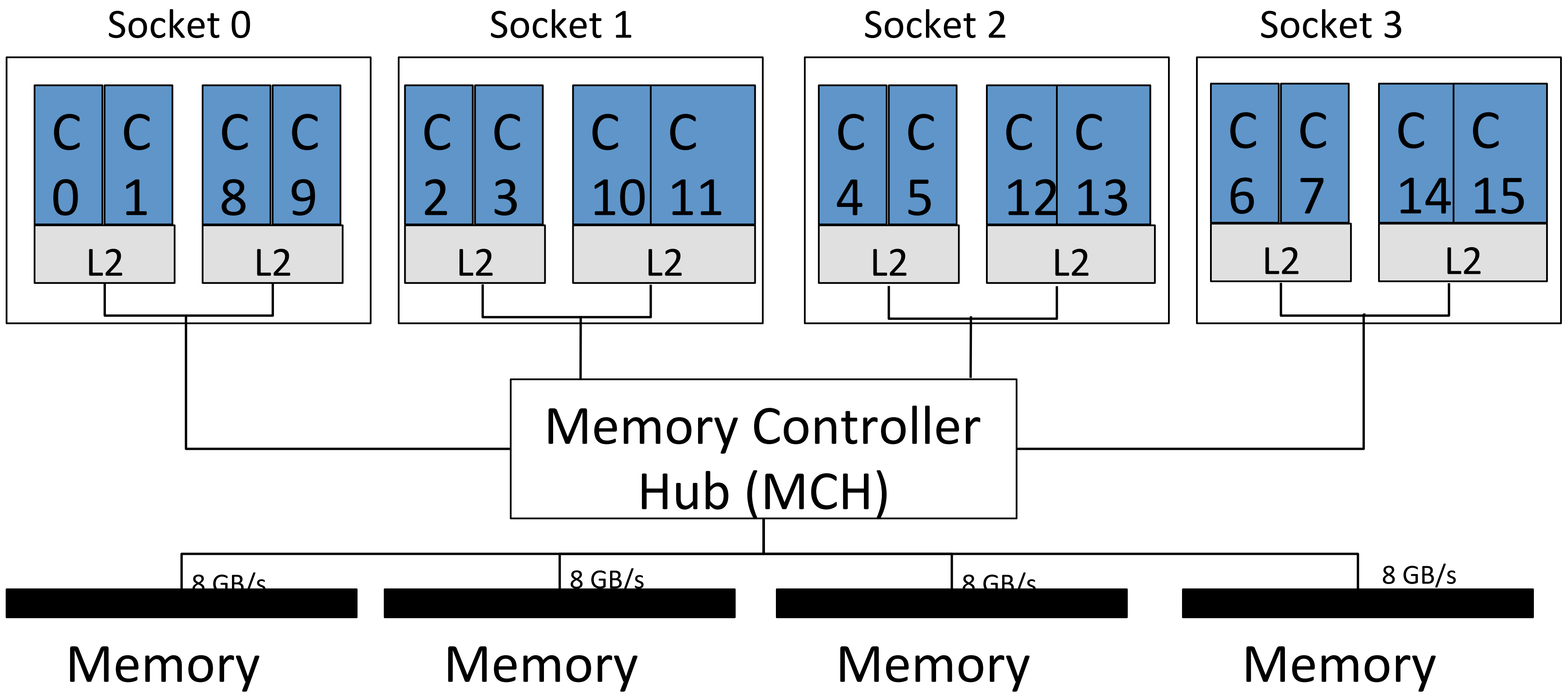
Centralized shared memory system (I)

- Intel X7350 quad-core (Tigerton)
  - Private L1 cache: 32 KB instruction, 32 KB data
  - Shared L2 cache: 4 MB unified cache



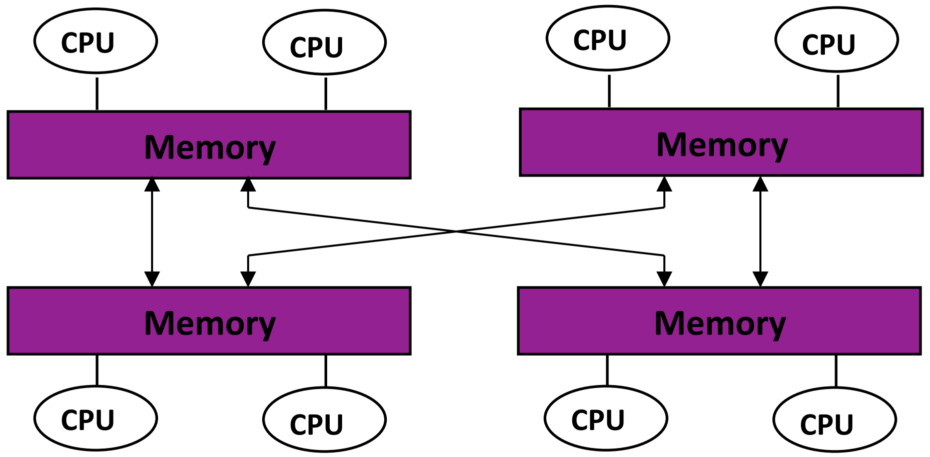
Centralized shared memory systems (II)

- Intel X7350 quad-core (Tigerton) multi-processor configuration



Distributed Shared Memory Architectures

- Also referred to as Non-Uniform Memory Architectures (NUMA)
- Some memory is closer to a certain processor than other memory
  - The whole memory is still addressable from all processors
  - Depending on what data item a processor retrieves, the access time might vary strongly



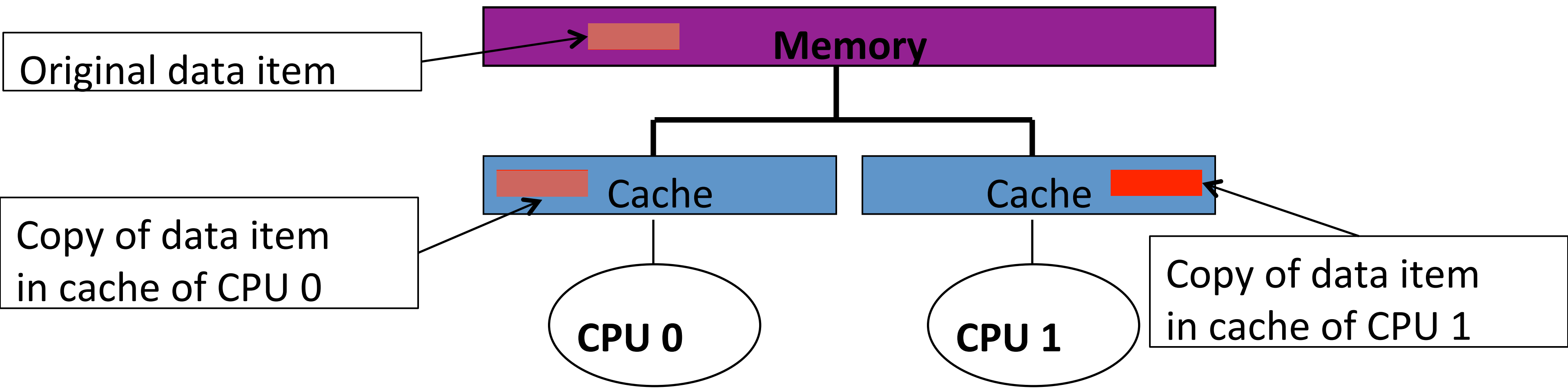
NUMA architectures (II)

- Reduces the memory bottleneck compared to SMPs
- More difficult to program efficiently
  - E.g. first touch policy: data item will be located in the memory of the processor which uses a data item first
- To reduce effects of non-uniform memory access, caches are often used
  - ccNUMA: cache-coherent non-uniform memory access architectures
- Largest example as of today: SGI Origin with 512 processors

Distributed Shared Memory Systems

Cache Coherence

- Real-world shared memory systems have caches between memory and CPU
- Copies of a single data item can exist in multiple caches
- Modification of a shared data item by one CPU leads to outdated copies in the cache of another CPU


Original data item
Memory
Cache
Cache
Copy of data item in cache of CPU 0
Copy of data item in cache of CPU 1
CPU 0
CPU 1

Cache coherence (II)

- Typical solution:
  - Caches keep track on whether a data item is shared between multiple processes
  - Upon modification of a shared data item, ‘notification’ of other caches has to occur
  - Other caches will have to reload the shared data item on the next access into their cache
- Cache coherence is <u>only</u> an issue in case multiple tasks access the same item
  - Multiple threads
  - Multiple processes have a joint shared memory segment
  - Process is being migrated from one CPU to another

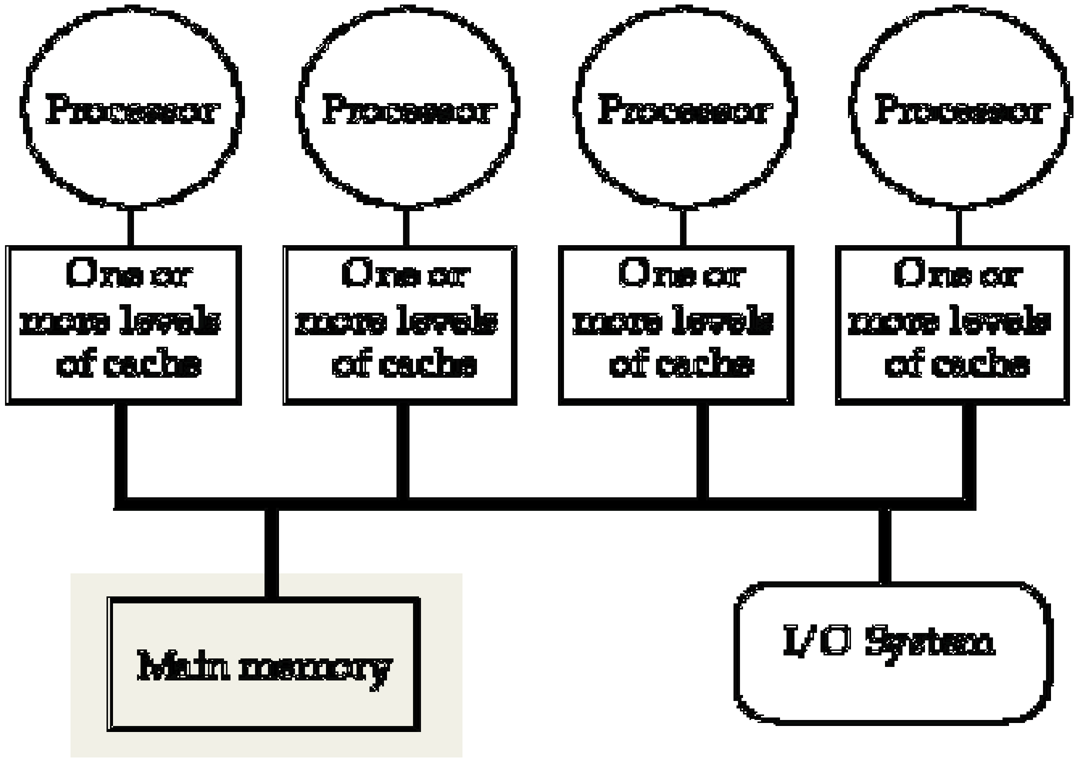
Cache Coherence Protocols

- Snooping Protocols
  - Send all requests for data to all processors
  - Processors snoop a bus to see if they have a copy and respond accordingly
  - Requires broadcast, since caching information is at processors
  - Works well with bus (natural broadcast medium)
  - Dominates for centralized shared memory machines
- Directory-Based Protocols
  - Keep track of what is being shared in centralized location
  - Distributed memory => distributed directory for scalability (avoids bottlenecks)
  - Send point-to-point requests to processors via network
  - Scales better than Snooping
  - Commonly used for distributed shared memory machines

Categories of cache misses

- Up to now:
  - Compulsory Misses: first access to a block cannot be in the cache (cold start misses)
  - Capacity Misses: cache cannot contain all blocks required for the execution
  - Conflict Misses: cache block has to be discarded because of block replacement strategy
- In multi-processor systems:
  - Coherence Misses: cache block has to be discarded because another processor modified the content
    - true sharing miss: another processor modified the content of the request element
    - false sharing miss: another processor invalidated the block, although the actual item of interest is unchanged.

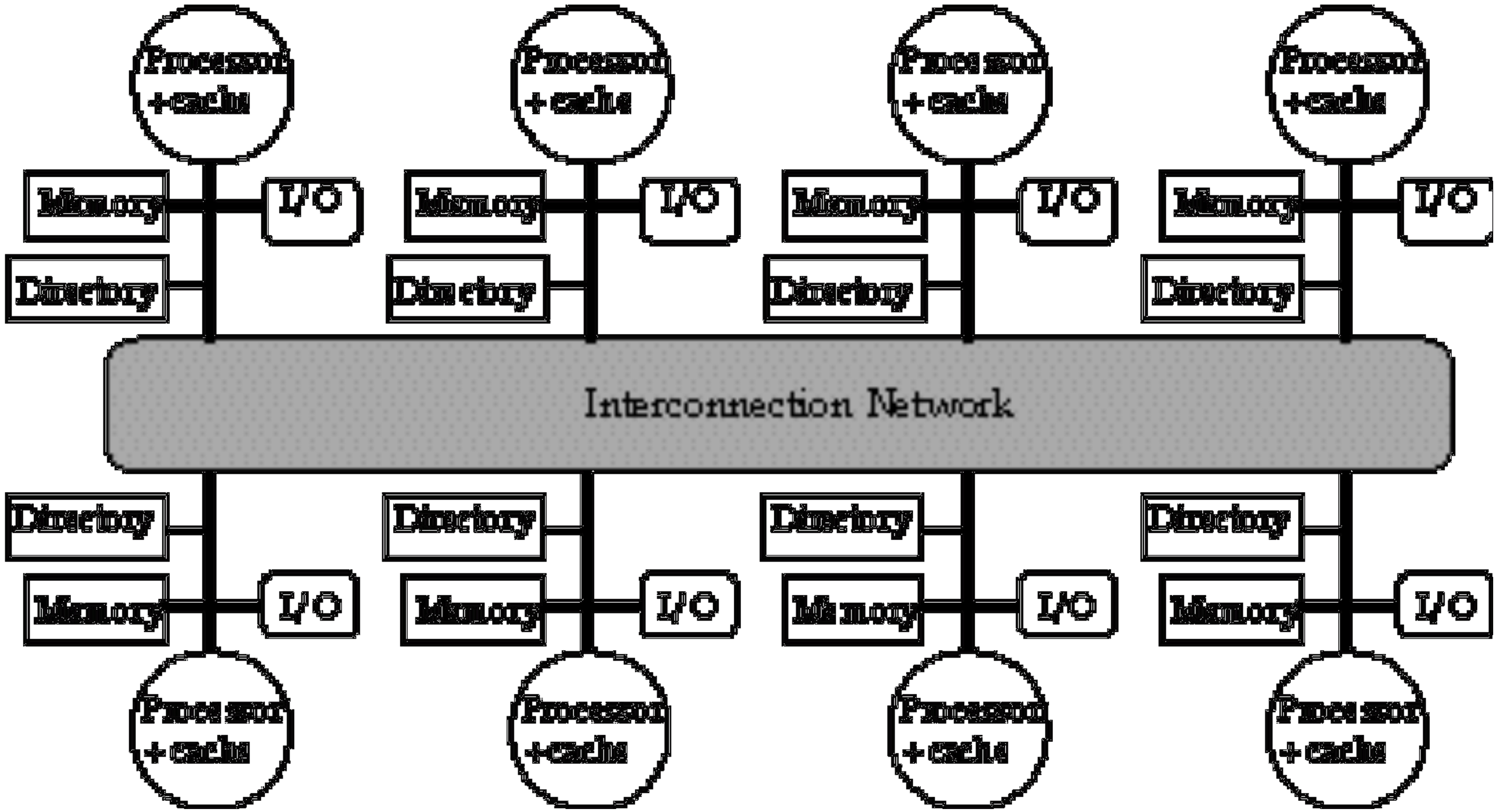
Bus Snooping Topology


Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Main memory
I/O System

Larger Shared Memory Systems

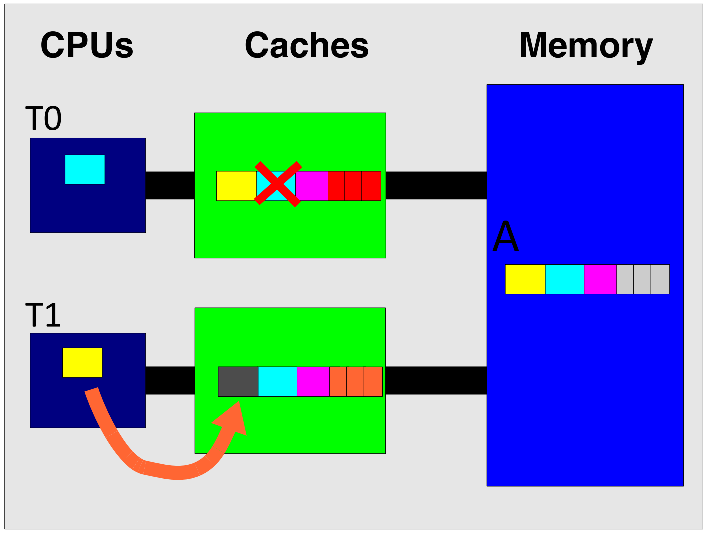
- Typically Distributed Shared Memory Systems
- Local or remote memory access via memory controller
- Directory per cache that tracks state of every block in every cache
  - Which caches have a copy of block, dirty vs. clean, ...
- Info per memory block vs. per cache block?
  - PLUS: In memory => simpler protocol (centralized/one location)
  - MINUS: In memory => directory is ƒ(memory size) vs. ƒ(cache size)
- Prevent directory as bottleneck?
  distribute directory entries with memory, each keeping track of which processors have copies of their blocks

Distributed Directory MPs



False Sharing in OpenMP

- False sharing
  - When at least one thread write to a cache line while others access it
    - Thread 0: = A[1] (read)
    - Thread 1: A[0] = … (write)
- Solution: use array padding

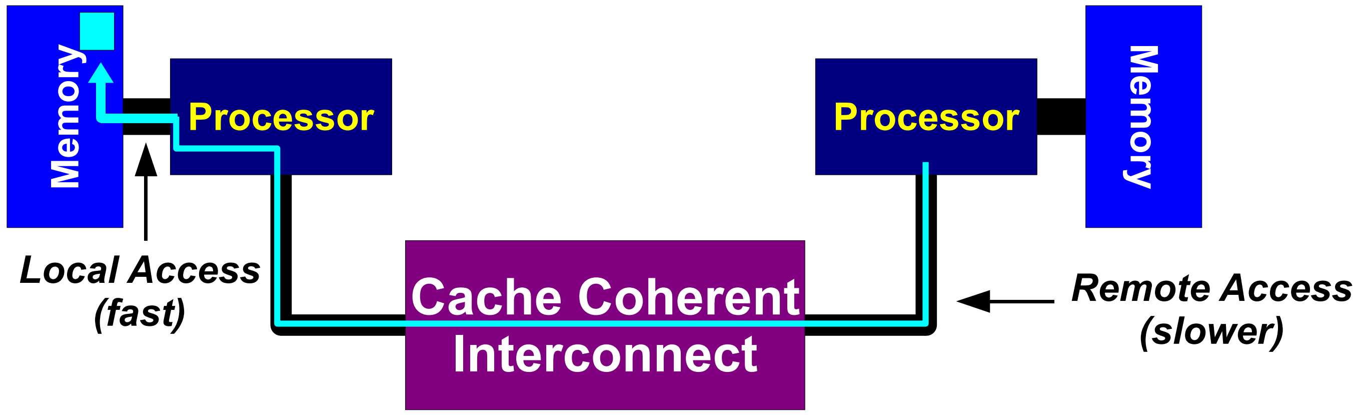


```
int a[max_threads];
#pragma omp parallel for schedule(static,1)
for(int i=0; i<max_threads; i++)
     a[i] +=i;
```

```
int a[max_threads][cache_line_size];
#pragma omp parallel for schedule(static,1)
for(int i=0; i<max_threads; i++)
     a[i][0] +=i;
```

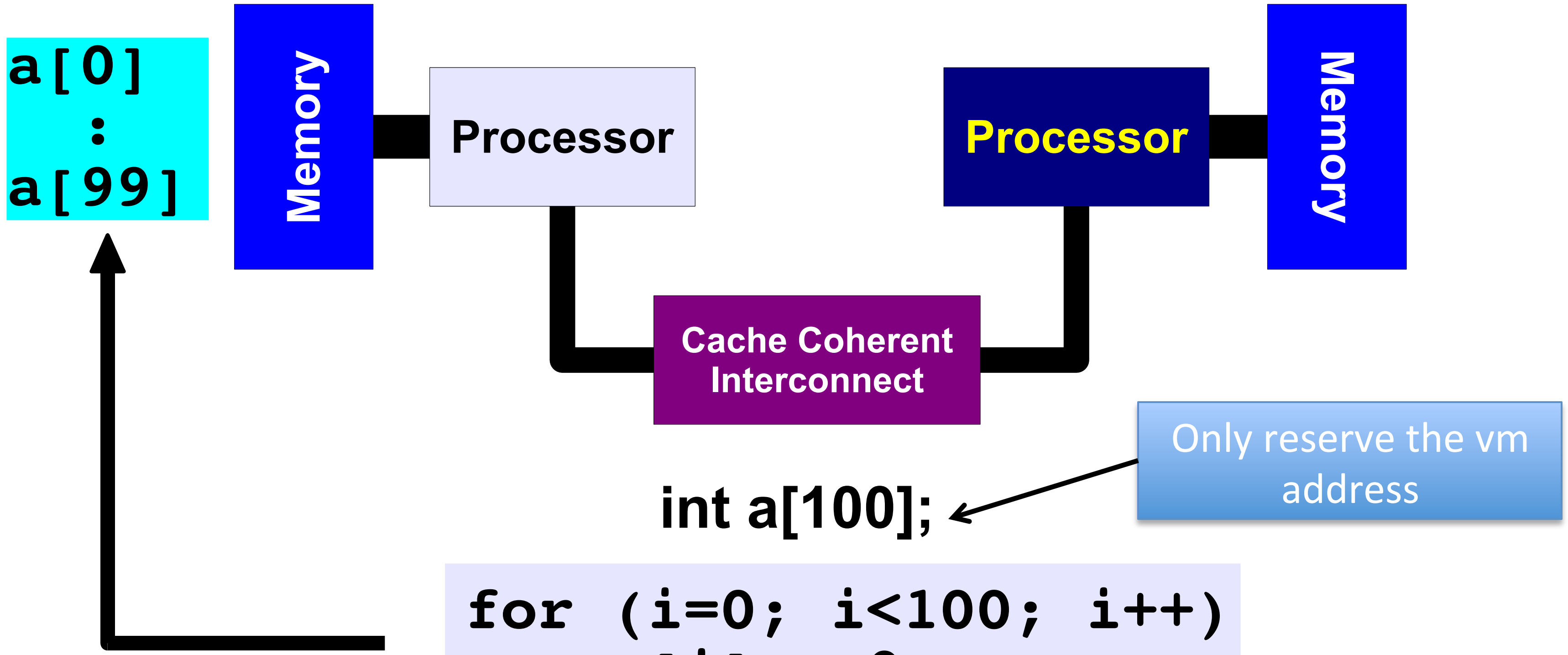
NUMA and First Touch Policy

- Data placement policy on NUMA architectures



- First Touch Policy
  - The process that first touches a page of memory causes that page to be allocated in the node on which the process is running

NUMA First-touch placement/1

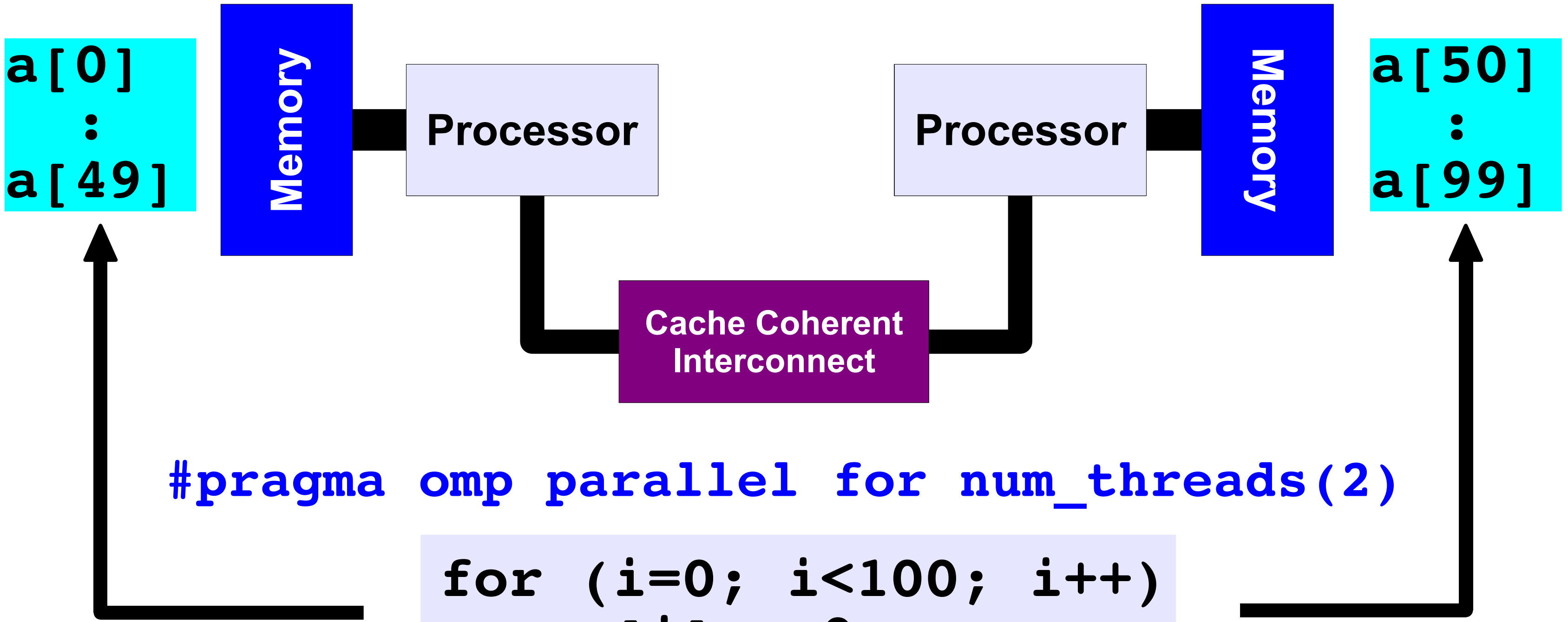


int a[100];

```
for (i=0; i<100; i++)
      a[i] = 0;
```

First Touch
All array elements are in the memory of the processor executing this thread

NUMA First-touch placement/2



```
#pragma omp parallel for num_threads(2)
for (i=0; i<100; i++)
        a[i] = 0;
```

First Touch
Both memories each have "their half" of the array

Work with First-Touch in OpenMP

- First-touch in practice
  - Initialize data consistently with the computations

```
#pragma omp parallel for
for(i=0; i<N; i++) {
   a[i] = 0.0; b[i] = 0.0 ; c[i] = 0.0;
}
readfile(a,b,c);

#pragma omp parallel for
for(i=0; i<N; i++) {
    a[i] = b[i] + c[i];
}
```

Concluding Observations

- Programmer can optimize for cache performance
  - How data structures are organized
  - How data are accessed
    - Nested loop structure
    - Blocking is a general technique
- All systems favor "cache friendly code"
  - Getting absolute optimum performance is very platform specific
    - Cache sizes, line sizes, associativities, etc.
  - Can get most of the advantage with generic code
    - Keep working set reasonably small (temporal locality)
    - Use small strides (spatial locality)
  - Work with cache coherence protocol and NUMA first touch policy

References

- Computer Architecture, A Quantitative Approach. 5TH Edition, The Morgan Kaufmann, September 30, 2011 by John L. Hennessy (Author), David A. Patterson
- A Primer on Memory Consistency and Cache Coherence Daniel J. Sorin Mark D. Hill David A. Wood, SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE Mark D. Hill, Series Editor, 2011