Cache内存

100076202： 计算机系统导论

任课教师：

计卫星　　宿红毅　　张艳

原作者：

Randal E. **Bryant and** David R. O'Hallaron

Carnegie Mellon University

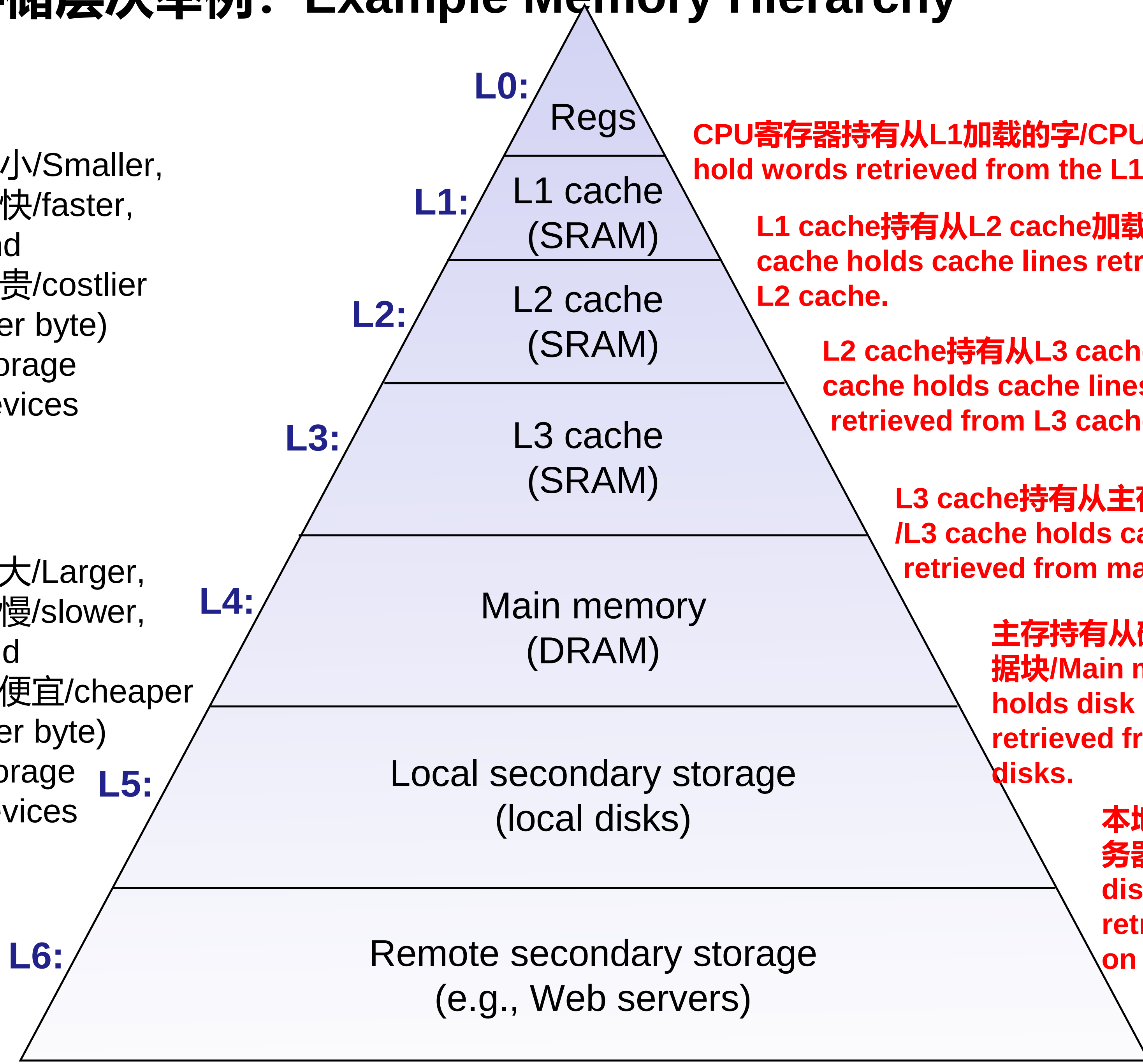
主要内容

- **Cache结构和操作 Cache memory organization and operation**
- Cache对性能的影响 Performance impact of caches
  - 存储性能山丘 The memory mountain
  - 循环变换提升空间局域性 Rearranging loops to improve spatial locality
  - 使用blocking提升时间局域性 Using blocking to improve temporal locality

存储层次举例：Example Memory Hierarchy

更小/Smaller,
更快/faster,
and
更贵/costlier
(per byte)
storage
devices

更大/Larger,
更慢/slower,
and
更便宜/cheaper
(per byte)
storage
devices



CPU寄存器持有从L1加载的字/CPU registers hold words retrieved from the L1 cache.

L1 cache持有从L2 cache加载的cache行/L1 cache holds cache lines retrieved from the L2 cache.

L2 cache持有从L3 cache加载的cache行/L2 cache holds cache lines retrieved from L3 cache

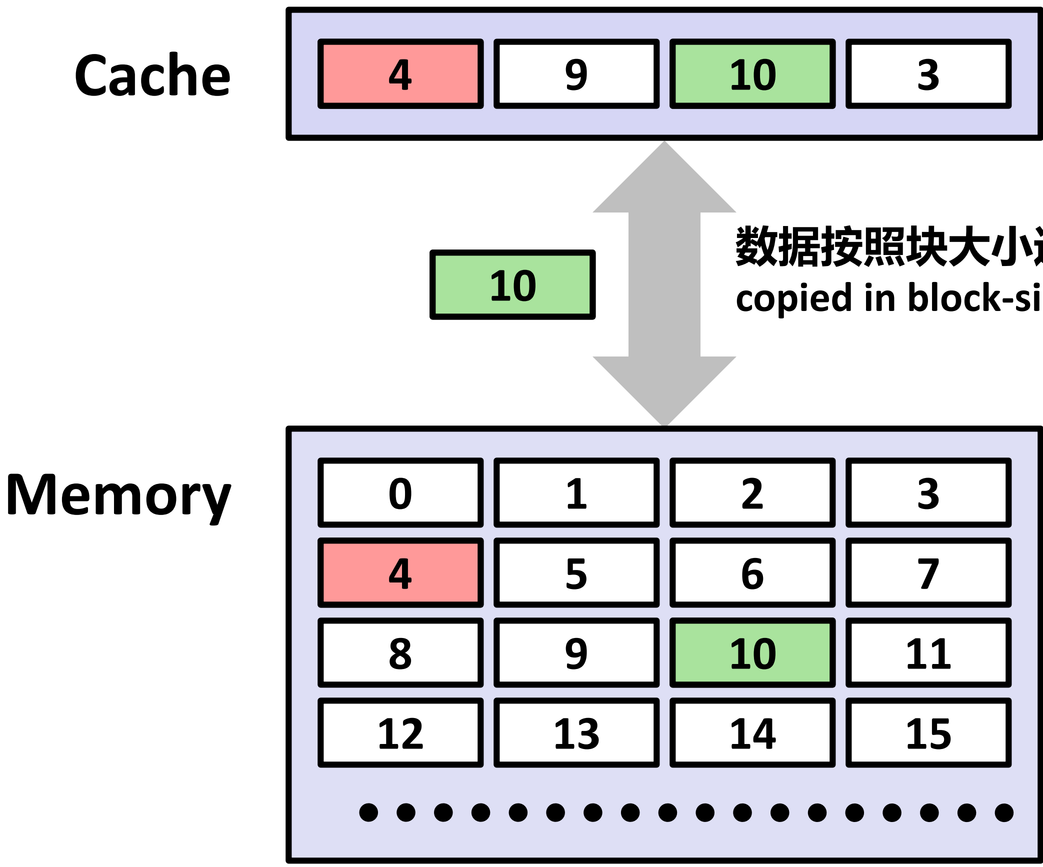
L3 cache持有从主存获取的cache行/L3 cache holds cache lines retrieved from main memory.

主存持有从磁盘加载的数据块/Main memory holds disk blocks retrieved from local disks.

本地磁盘持有从远程服务器加载的数据/Local disks hold files retrieved from disks on remote servers

Cache基本概念 General Cache Concept

更小、更快、成本更高的内存缓存了块集合的子集 / Smaller, faster, more expensive memory caches a subset of the blocks

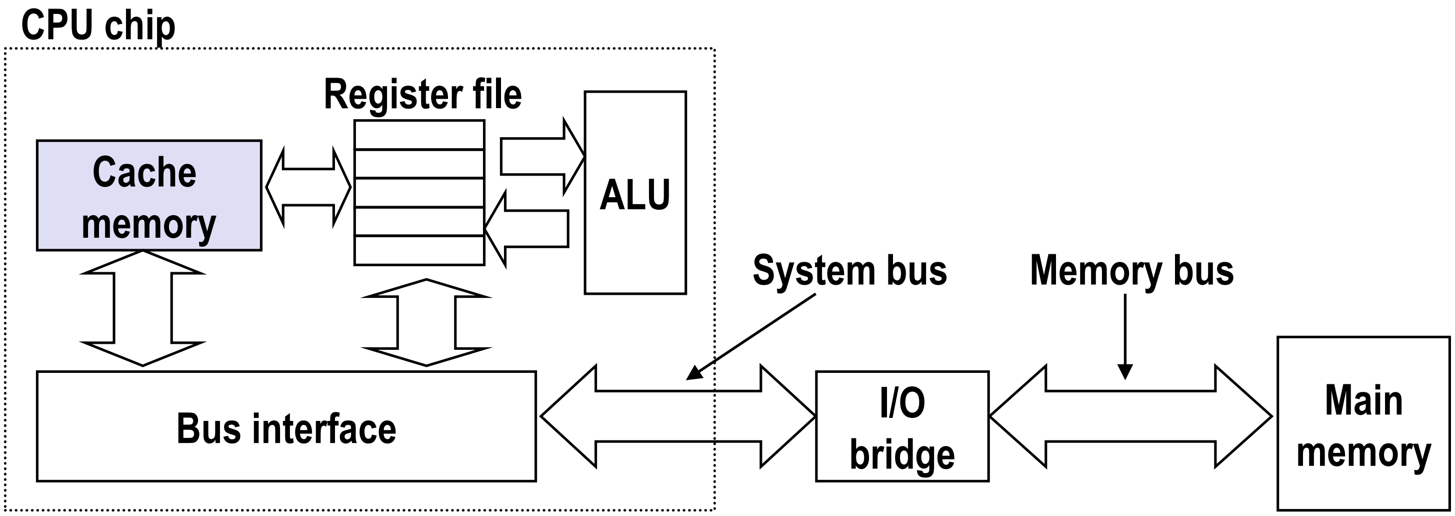


数据按照块大小进行拷贝传输/Data is copied in block-sized transfer units

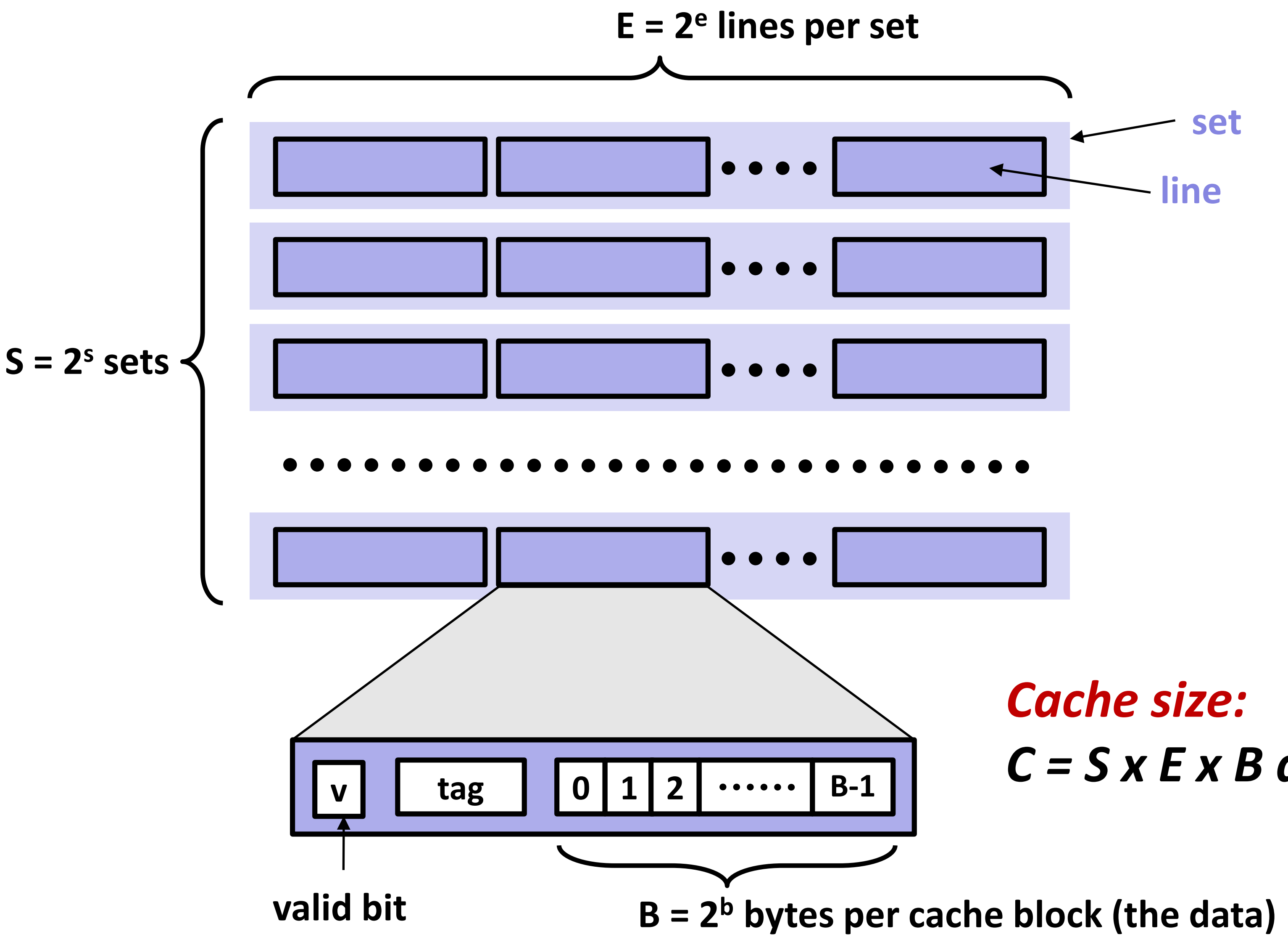
更大、更慢、更便宜的内存被划分为多个大小相同的块/Larger, slower, cheaper memory viewed as partitioned into “blocks”

Cache内存 Cache Memories

- **Cache是有硬件自动管理的容量较小的SRAM Cache memories are small, fast SRAM-based memories managed automatically in hardware**
  - 持有从主存装入的频繁访问的内容 Hold frequently accessed blocks of main memory
- **CPU首先在Cache中查找数据 CPU looks first for data in cache**
- **典型系统结构 Typical system structure:**



Cache组织结构 General Cache Organization (S, E, B)

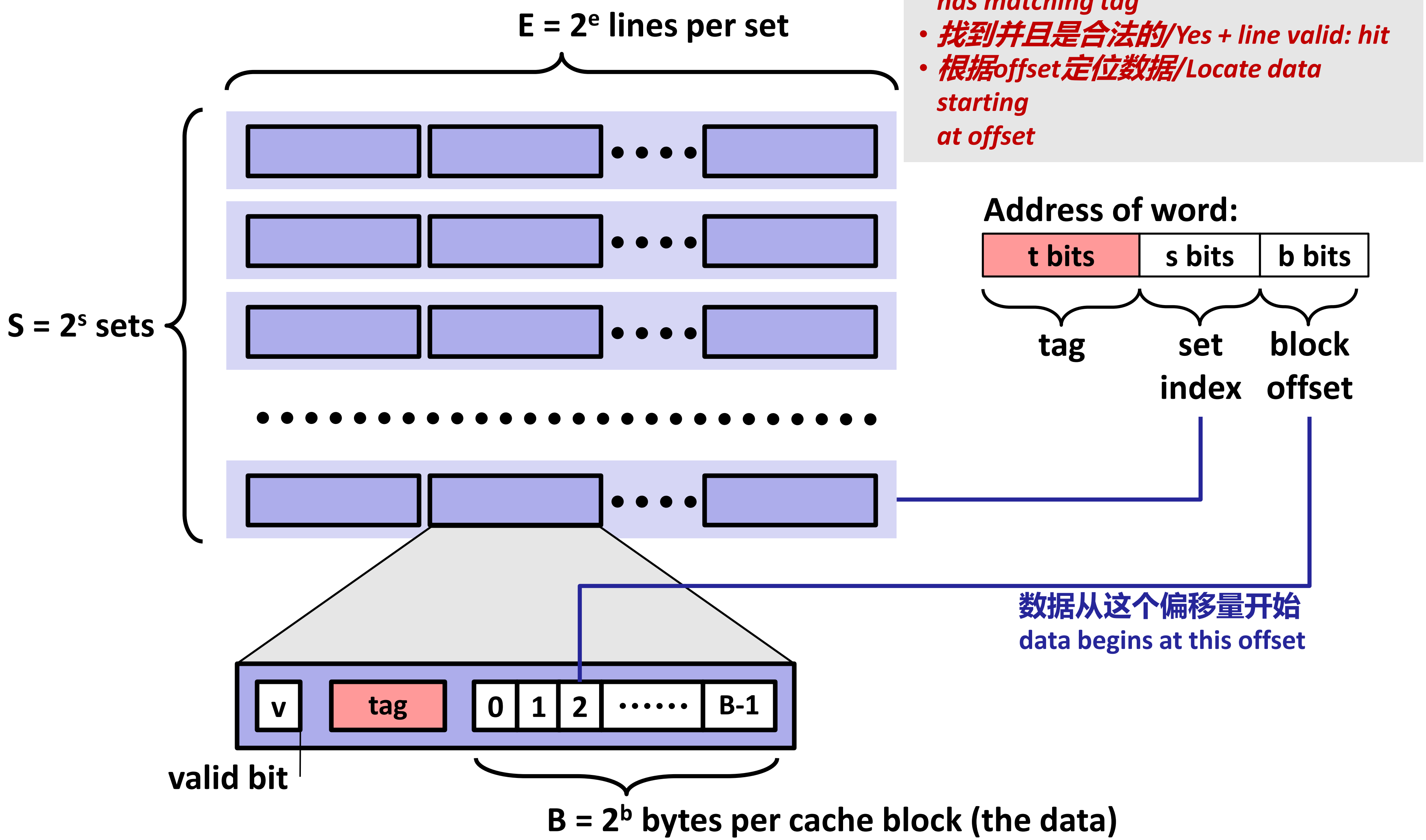


Cache size:

C = S x E x B data bytes

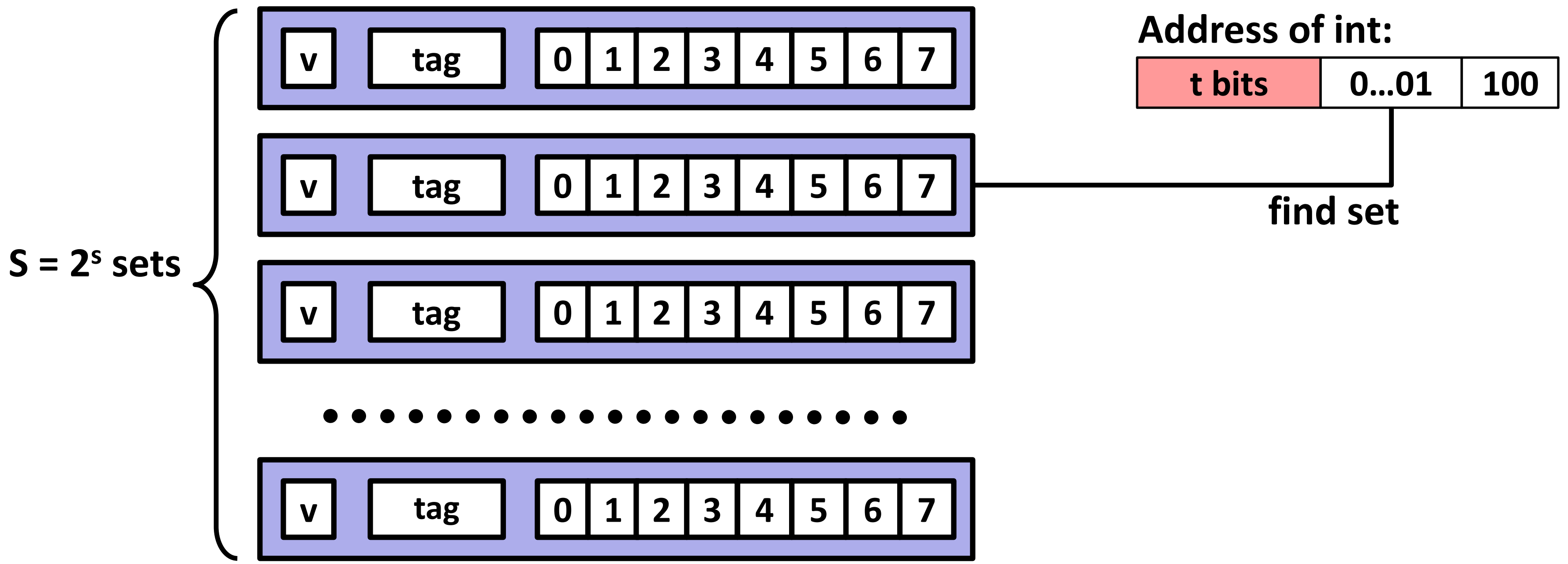
Cache读操作 Cache Read

- *查找组 / Locate set*
- *检查组里的块是否有匹配的tag/Check if any line in set has matching tag*
- *找到并且是合法的/Yes + line valid: hit*
- *根据offset定位数据/Locate data starting at offset*



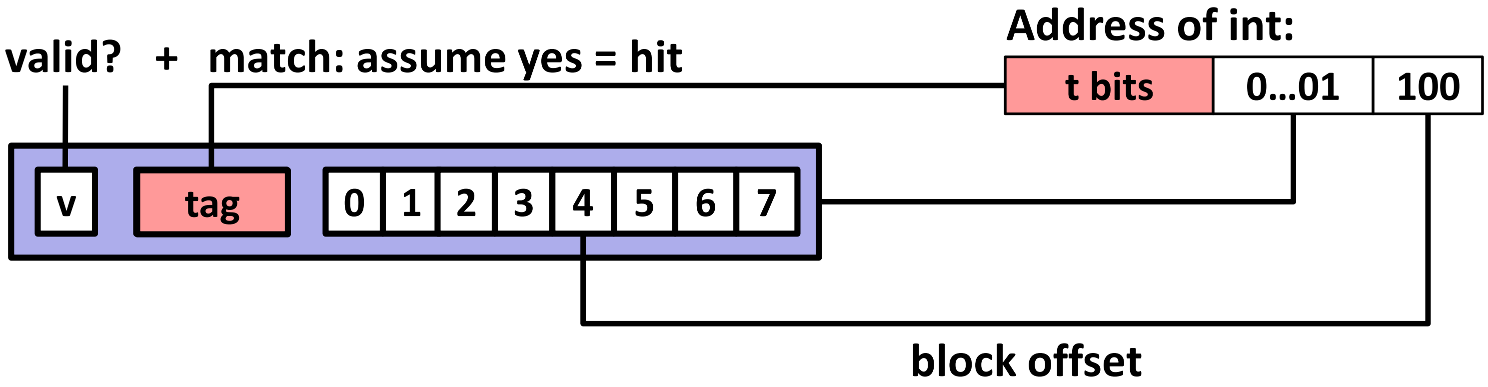
直相联映射 Example: Direct Mapped Cache (E = 1)

直相联映射：每行一组 Direct mapped: One line per set
假设每个Cache行8个字节 Assume: cache block size 8 bytes



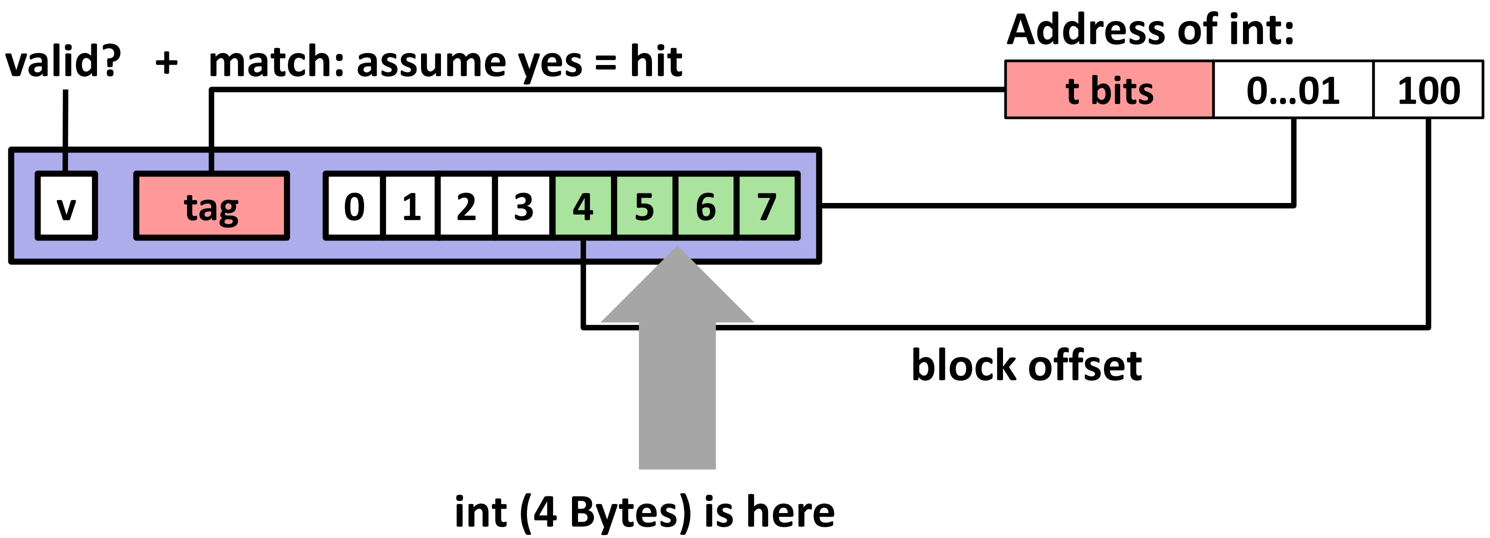
直相联映射 Example: Direct Mapped Cache (E = 1)

直相联映射：每行一组 Direct mapped: One line per set
假设每个Cache行8个字节 Assume: cache block size 8 bytes



直相联映射 Example: Direct Mapped Cache (E = 1)

直相联映射：每行一组 Direct mapped: One line per set
假设每个Cache行8个字节 Assume: cache block size 8 bytes



如果标签不匹配，则进行换出还如操作
If tag doesn't match: old line is evicted and replaced

直相联映射Cache示意 Direct-Mapped Cache Simulation

t=1	s=2	b=1
x	xx	x

M=16 bytes (4-bit addresses), B=2 bytes/block,
S=4 sets, E=1 Blocks/set

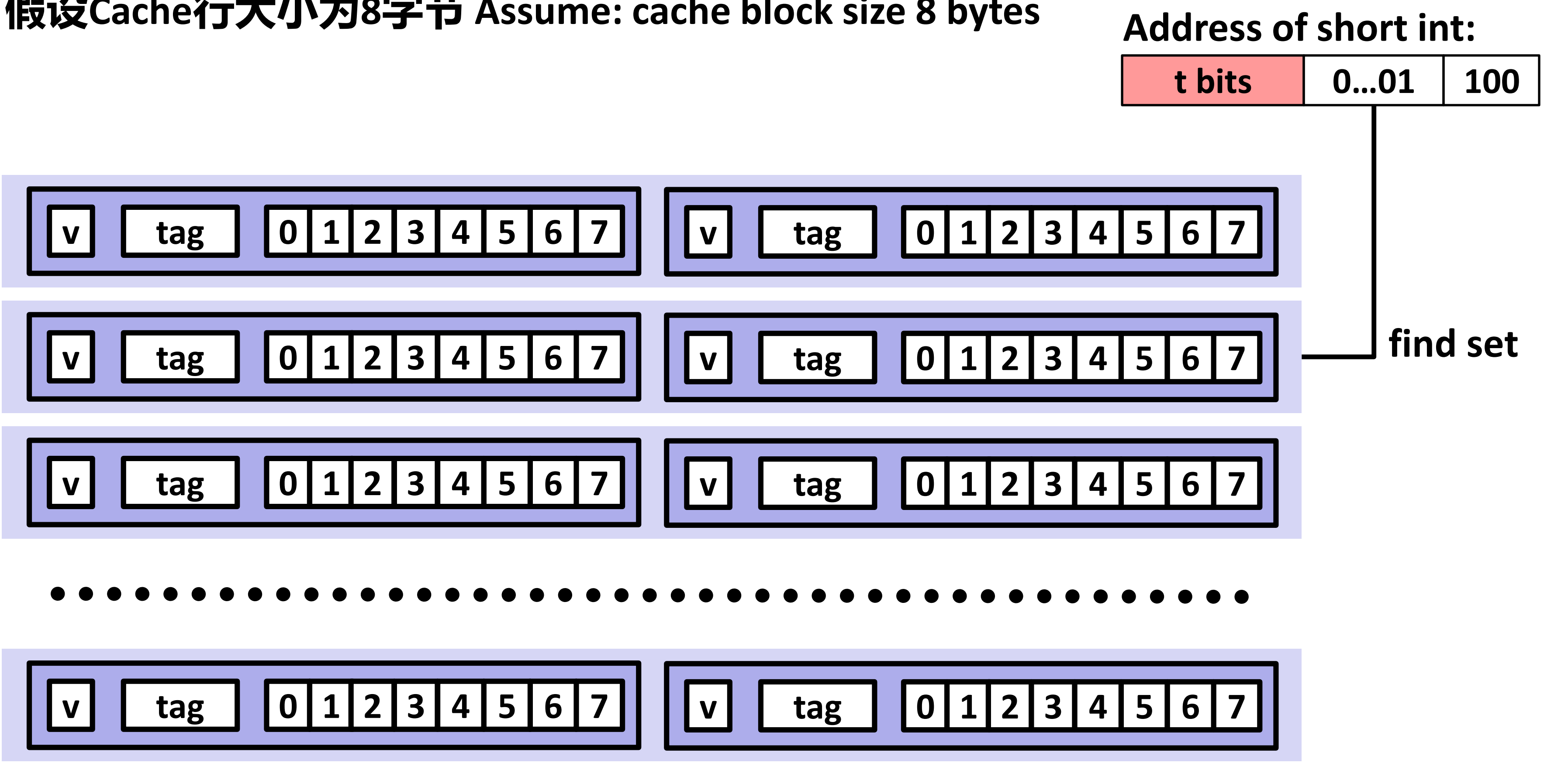
地址序列/
Address trace (reads, one byte per read):

0	**$[0\underline{00}0_2]$,**	miss
1	**$[0\underline{00}1_2]$,**	hit
7	**$[0\underline{11}1_2]$,**	miss
8	**$[1\underline{00}0_2]$,**	miss
0	**$[0\underline{00}0_2]$**	miss

	v	Tag	Block
Set 0	1	0	M[0-1]
Set 1			
Set 2			
Set 3	1	0	M[6-7]

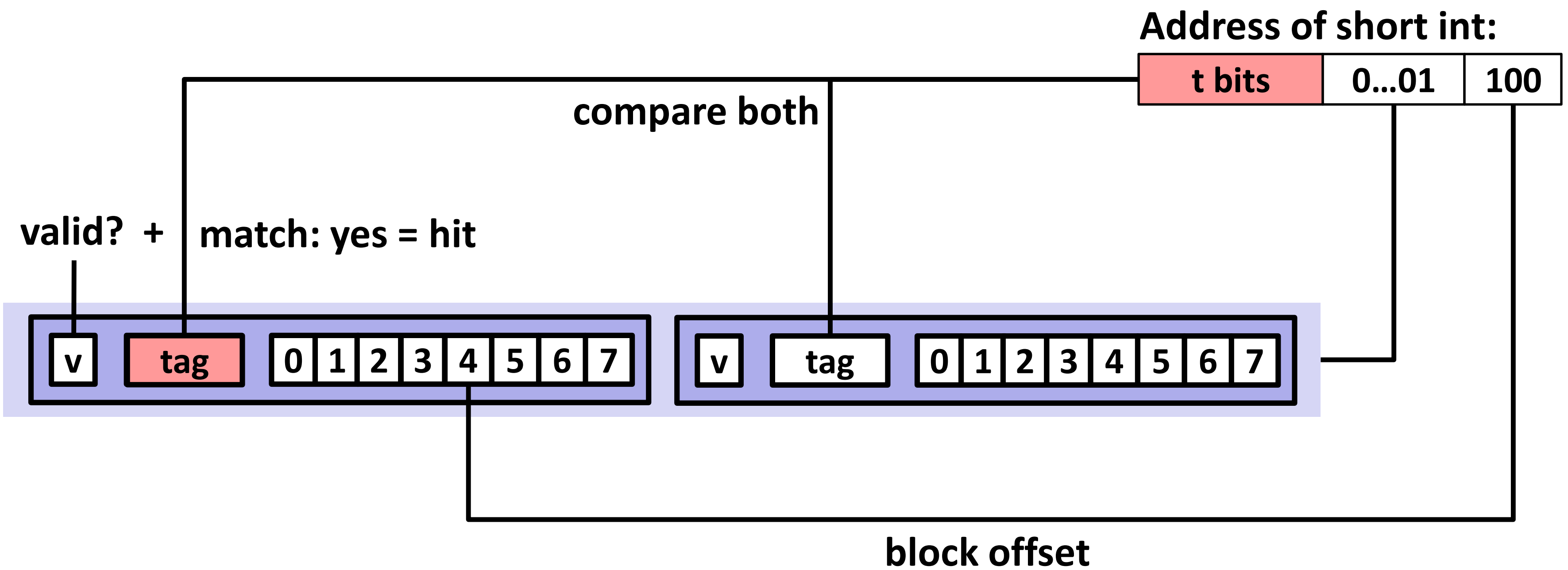
E路组相联Cache E-way Set Associative Cache (Here: E = 2)

每行2组： E = 2: Two lines per set
假设Cache行大小为8字节 Assume: cache block size 8 bytes



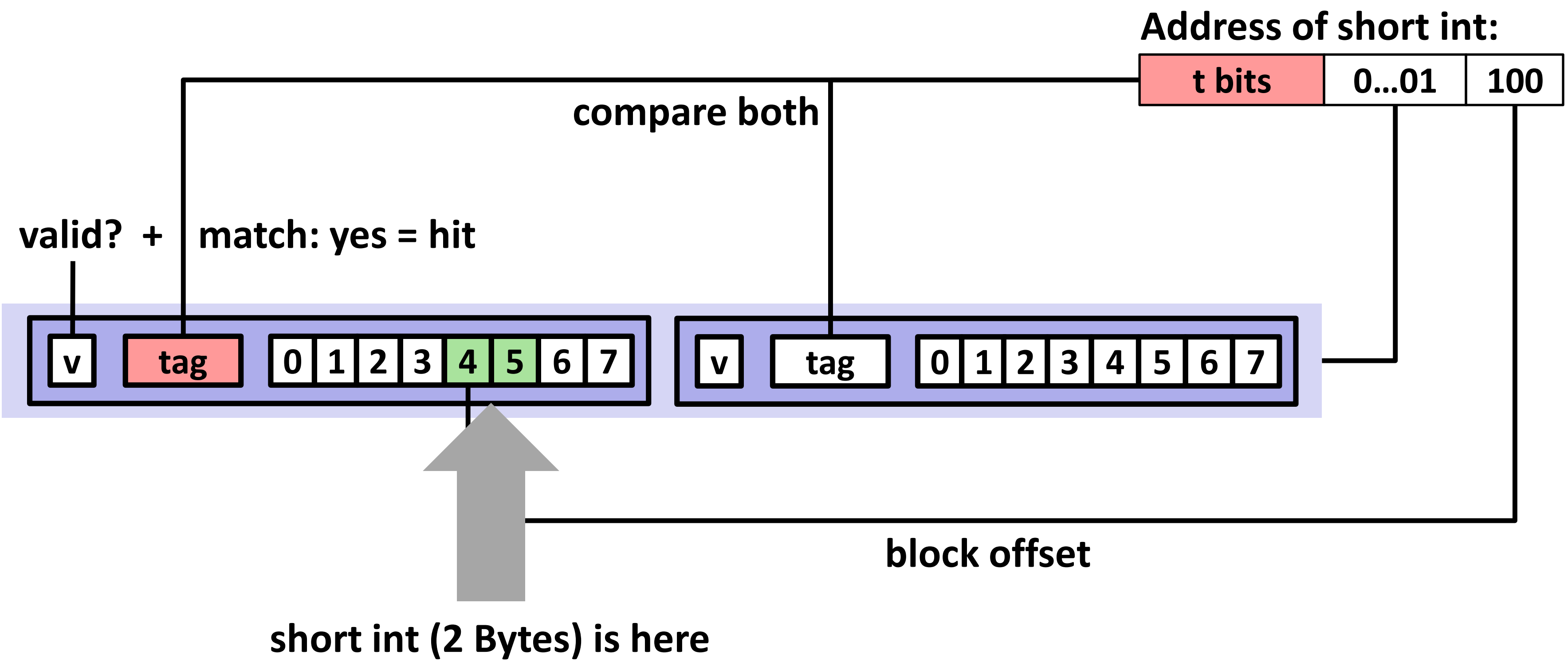
E路组相联Cache E-way Set Associative Cache (Here: E = 2)

每行2组：E = 2: Two lines per set
假设Cache行大小为8字节 Assume: cache block size 8 bytes



E路组相联Cache E-way Set Associative Cache (Here: E = 2)

每行2组： E = 2: Two lines per set
假设Cache行大小为8字节 Assume: cache block size 8 bytes



不匹配 No match:

- 选中相应组中的一行进行换出还如 One line in set is selected for eviction and replaceme
- 替换策略：随机，最近最少使用 Replacement policies: random, least recently used (LR

2路组相联Cache示意 2-Way Set Associative Cache Simulation

t=2	s=1	b=1
xx	x	x

M=16 byte addresses, B=2 bytes/block,
S=2 sets, E=2 blocks/set

地址序列：Address trace (reads, one byte per read):

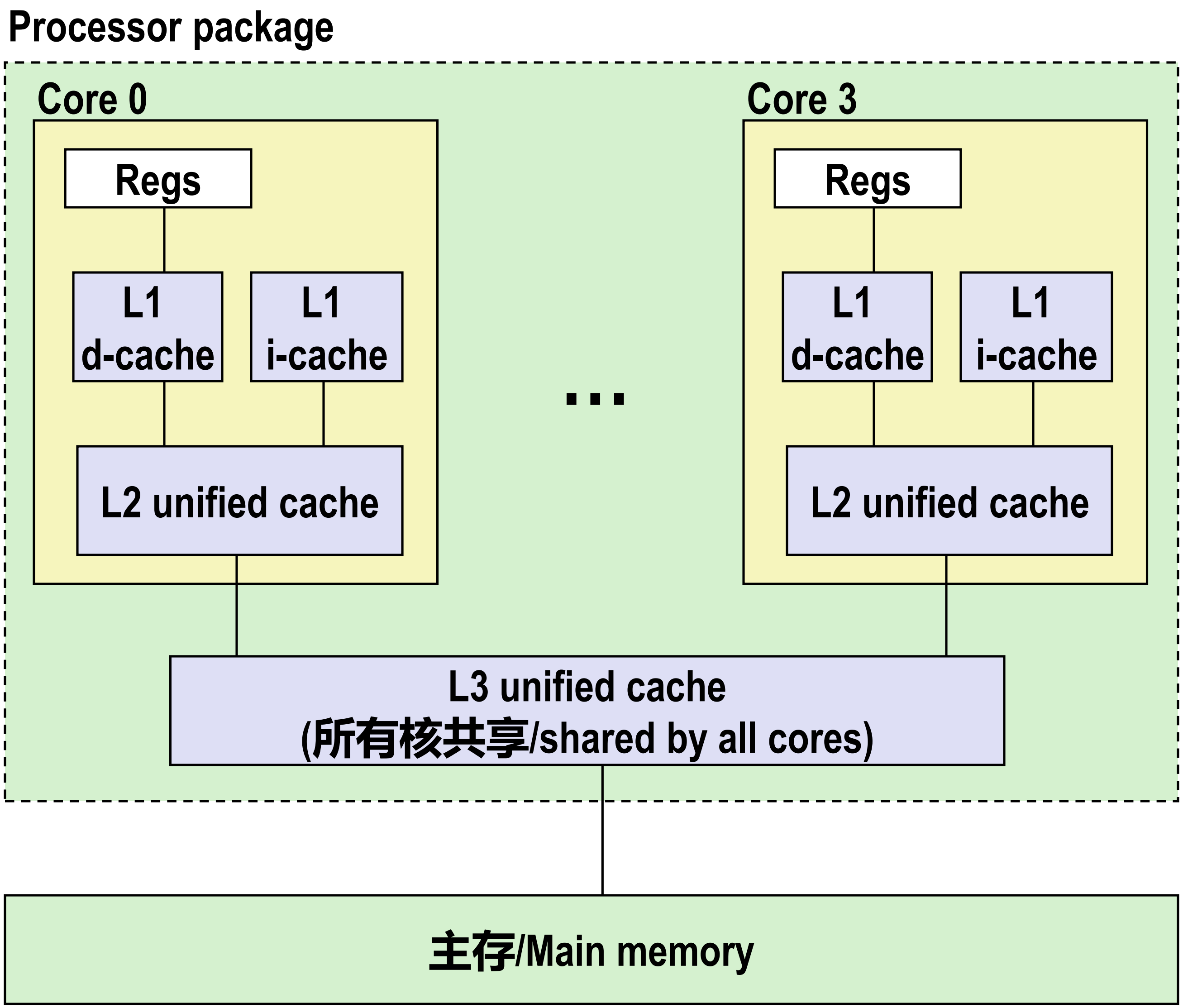
0	**$[00\underline{0}0_2]$,**	miss
1	**$[00\underline{0}1_2]$,**	hit
7	**$[01\underline{1}1_2]$,**	miss
8	**$[10\underline{0}0_2]$,**	miss
0	**$[00\underline{0}0_2]$**	hit

	v	**Tag**	**Block**
Set 0	1	00	M[0-1]
	1	10	M[8-9]
Set 1	1	01	M[6-7]
	0		

Cache写操作 What about writes?

- **多数据副本 Multiple copies of data exist:**
  - L1, L2, L3, Main Memory, Disk
- **写命中时如何处理？ What to do on a write-hit?**
  - 写透（直接写入内存）Write-through (write immediately to memory)
  - 写回（替换式写回）Write-back (defer write to memory until replacement of line)
    - 需要脏比特位标识 Need a dirty bit (line different from memory or not)
- **写丢失时如何处理？ What to do on a write-miss?**
  - 写分配（装载进Cache后进行更新）Write-allocate (load into cache, update line in cache)
    - 如果后续还有写操作时比较好 Good if more writes to the location follow
  - 非写分配（直接写入内存，不装载）No-write-allocate (writes straight to memory, does not load into cache)
- **通常策略 Typical**
  - 写透 + 非写分配 Write-through + No-write-allocate
  - **写回 + 写分配 Write-back + Write-allocate**

i7 Cache层次结构 Intel Core i7 Cache Hierarchy



L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 10 cycles

L3 unified cache:
8 MB, 16-way,
Access: 40-75 cycles

Block size: 64 bytes for all caches.

Cache性能评价 Cache Performance Metrics

- **丢失率 Miss Rate**
  - 内存引用没有在Cache中找到的比率 Fraction of memory references not found in cache (misses / accesses)
    = 1 – hit rate
  - 通常的Cache丢失率 Typical numbers (in percentages):
    - 3-10% for L1
    - L2也可能很小，依赖Cache大小 can be quite small (e.g., < 1%) for L2, depending on size, etc.
- **命中时间 Hit Time**
  - 从Cache行到处理器的时间 Time to deliver a line in the cache to the processor
    - 包括判断Cache是否命中的时间 includes time to determine whether the line is in the cache
  - 通常的时间 Typical numbers:
    - L1 Cache 4个时钟周期 4 clock cycle for L1
    - L2 Cache 10个时钟周期 10 clock cycles for L2
- **丢失开销 Miss Penalty**
  - 丢失需要额外的时间 Additional time required because of a miss
    - 主存的访问周期50~200 typically 50-200 cycles for main memory (Trend: increasing!)

Cache性能评价 Cache Performance Metrics

- **命中和丢失之间的差距较大 Huge difference between a hit and a miss**
  - 如果只有L1和主存，则会是100x Could be 100x, if just L1 and main memory
- **99%的命中率的性能是97%的两倍 Would you believe 99% hits is twice as good as 97%?**
  - 假设 Consider:
    Cache命中需要1个周期 cache hit time of 1 cycle
    Cache丢失需要2个周期 miss penalty of 100 cycles
  - 平均访问时间 Average access time:
    97% hits: 1 cycle + 0.03 * 100 cycles = **4 cycles**
    99% hits: 1 cycle + 0.01 * 100 cycles = **2 cycles**
- **This is why "miss rate" is used instead of "hit rate"**

编写Cache友好的代码 Writing Cache Friendly Code

- **让最常见的最快 Make the common case go fast**
  - 关注主要函数的内层循环 Focus on the inner loops of the core functions

- **减少内层循环的丢失率 Minimize the misses in the inner loops**
  - 重复访问（时间局域性）Repeated references to variables are good (temporal locality)
  - 连续访问（空间局域性）Stride-1 reference patterns are good (spatial locality)

关键点：我们对局部性的定性概念是通过我们对缓存存储器的理解来量化的

Key idea: Our qualitative notion of locality is quantified through our understanding of cache memories

主要内容

- Cache结构和操作/Cache memory organization and operation
- **Cache对性能的影响 Performance impact of caches**
  - 存储性能山丘 The memory mountain
  - 循环变换提升空间局域性 Rearranging loops to improve spatial locality
  - 使用blocking提升时间局域性 Using blocking to improve temporal locality

内存性能山丘 The Memory Mountain

- **读吞吐率Read throughput (读带宽 read bandwidth)**
  - 每秒从主存读取的字节数 Number of bytes read from memory per second (MB/s)

- **存储山丘： Memory mountain: 根据空间和时间局域性测量的读吞吐率 Measured read throughput as a function of spatial and temporal locality.**
  - 刻画内存系统性能的简单方法 Compact way to characterize memory system performance.

测试函数 Memory Mountain Test Function

```c
long data[MAXELEMS];  /* Global array to traverse */

/* test - Iterate over first "elems" elements of
 *        array "data" with stride of "stride", using
 *        using 4x4 loop unrolling.
 */
int test(int elems, int stride) {
    long i, sx2=stride*2, sx3=stride*3, sx4=stride*4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems, limit = length - sx4;

    /* Combine 4 elements at a time */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i+stride];
        acc2 = acc2 + data[i+sx2];
        acc3 = acc3 + data[i+sx3];
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 + data[i];
    }
    return ((acc0 + acc1) + (acc2 + acc3));
}
```

mountain/mountain.c

使用elems和stride的组合多次调用test()/Call test() with many combinations of elems and stride.

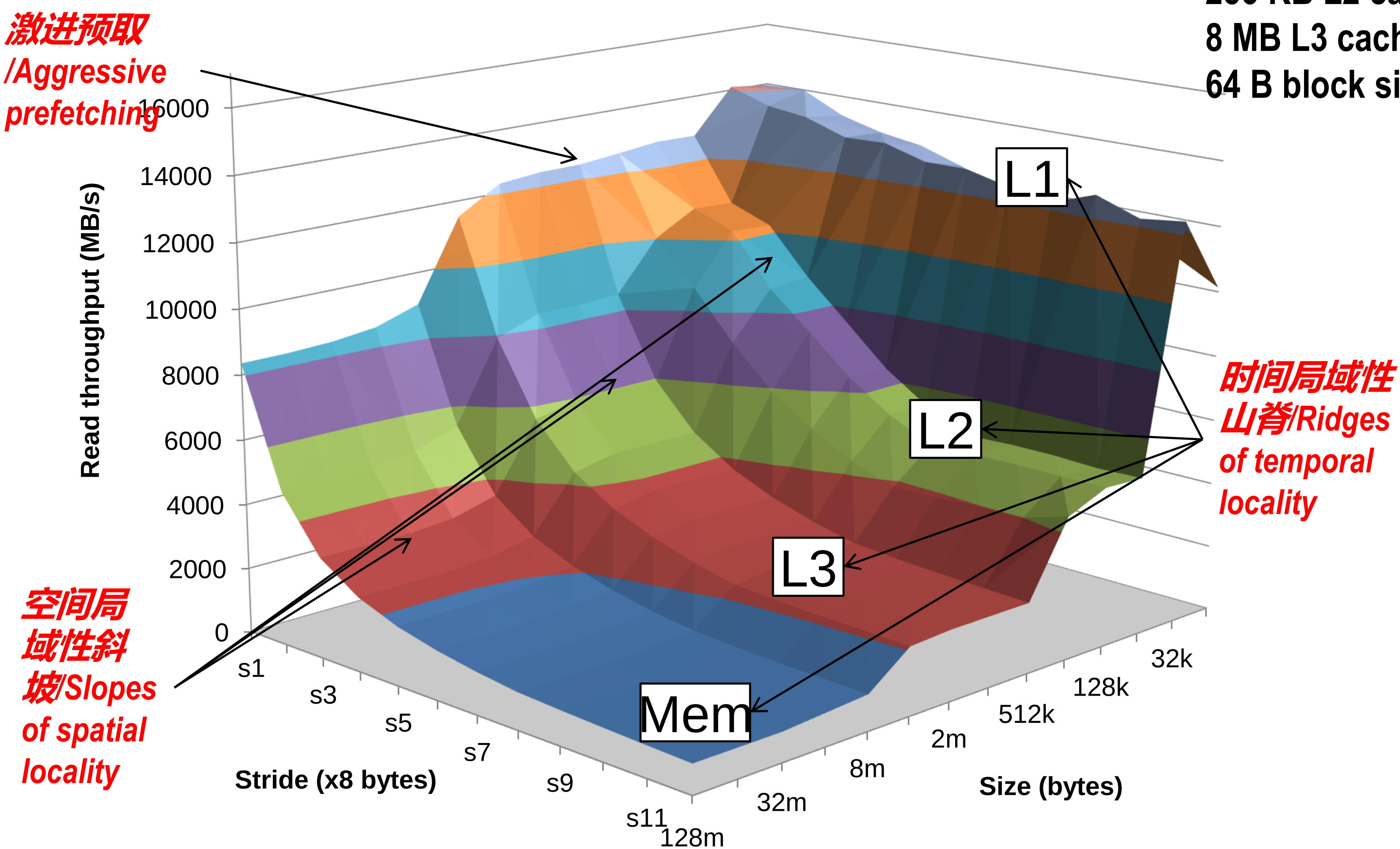
对于每个elems和stride组合/For each elems and stride:

1. 调用test()一次预热cache/Call test() once to warm up the caches.

2. 再次调用test()测量吞吐率/Call test() again and measure the read throughput(MB/s)

内存性能山丘

Core i7 Haswell
2.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size



主要内容

- Cache结构和操作 Cache memory organization and operation
- Cache对性能的影响 Performance impact of caches
  - 存储墙性能山丘 The memory mountain
  - 循环变换提升空间局域性 Rearranging loops to improve spatial locality
  - 使用blocking提升时间局域性 Using blocking to improve temporal locality

矩阵乘法示例 Matrix Multiplication Example

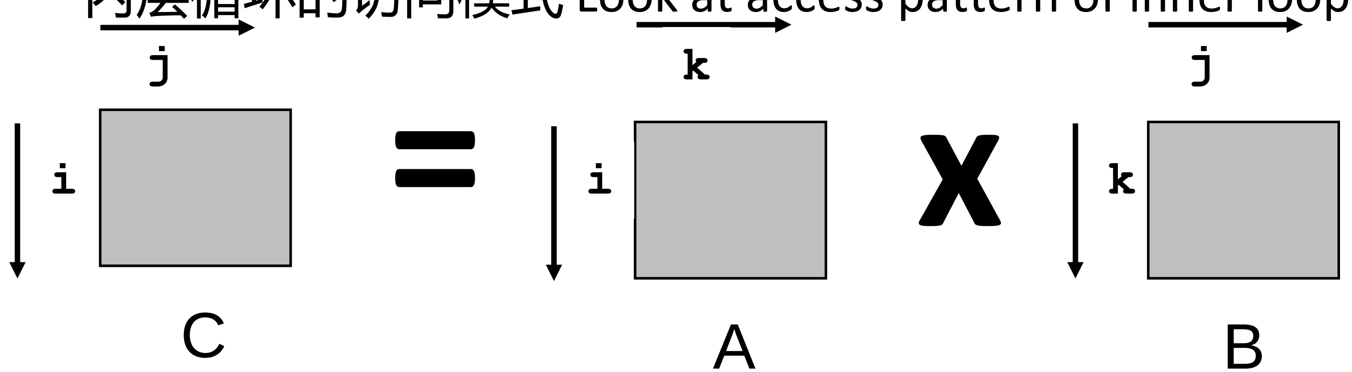
- **描述 Description:**
  - N x N 矩阵相乘/Multiply N x N matrices
  - 矩阵元素是双精度浮点（8字节）Matrix elements are doubles (8 bytes)
  - 总计O(N^3) 个操作/O(N^3) total operations
  - 每个输入元素读取N次/N reads per source element
  - 每个目标元素N次求和/N values summed per destination
    - 但是可能缓存在寄存器里面/but may be able to hold in register

Variable sum held 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;
  }
}                              matmult/mm.c
```

矩阵乘法的Cache失率分析 Miss Rate Analysis for Matrix Multiply

- **假设 Assume:**
  - 块大小/Block size = 32B (可以容纳四个double/big enough for four doubles)
  - 矩阵维度N非常大/Matrix dimension (N) is very large
    - 1/N 近似为0/Approximate 1/N as 0.0
  - Cache不足以容纳多行数据 Cache is not even big enough to hold multiple rows
- **分析方法 Analysis Method:**
  - 内层循环的访问模式 Look at access pattern of inner loop



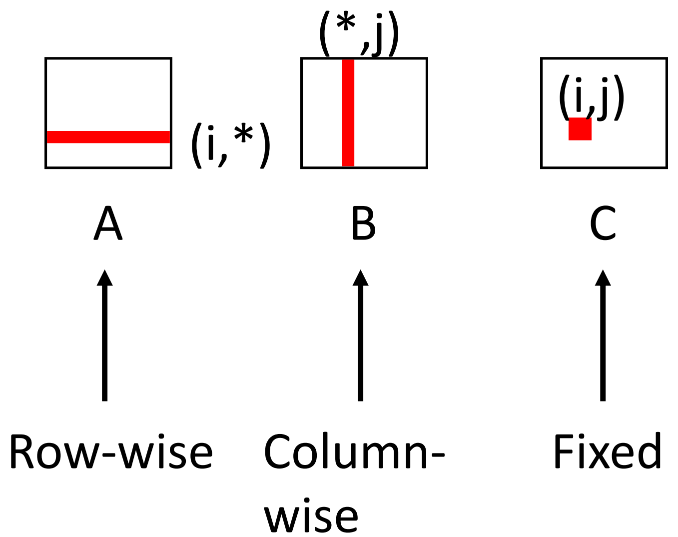
C数组的内存布局 Layout of C Arrays in Memory (review)

- **C数组按照行存储 C arrays allocated in row-major order**
  - 每行连续存储 each row in contiguous memory locations
- **访问每行的每个元素 Stepping through columns in one row:**
  - ```
    for (i = 0; i < N; i++)
      sum += a[0][i];
    ```
  - 访问连续元素 accesses successive elements
  - 如果块大小/if block size (B) > sizeof(a_{ij}) bytes, exploit spatial locality
    - miss rate = sizeof(a_{ij}) / B
- **访问每列的每个元素 Stepping through rows in one column:**
  - ```
    for (i = 0; i < n; i++)
      sum += a[i][0];
    ```
  - 访问不连续元素 accesses distant elements
  - 无空间局域性 no spatial locality!
    - 丢失率/miss rate = 1 (i.e. 100%)

矩阵乘法/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;
  }
}                         matmult/mm.c
```

Inner loop:



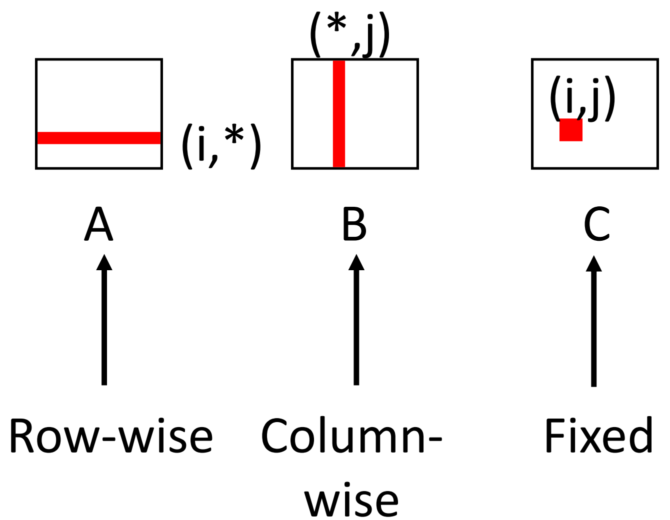
<u>每次循环迭代的丢失率Misses per inner loop iteration:</u>

<u>A</u>	<u>B</u>	<u>C</u>
0.25	1.0	0.0

矩阵乘法/ 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
  }
}                                matmult/mm.c
```

Inner loop:



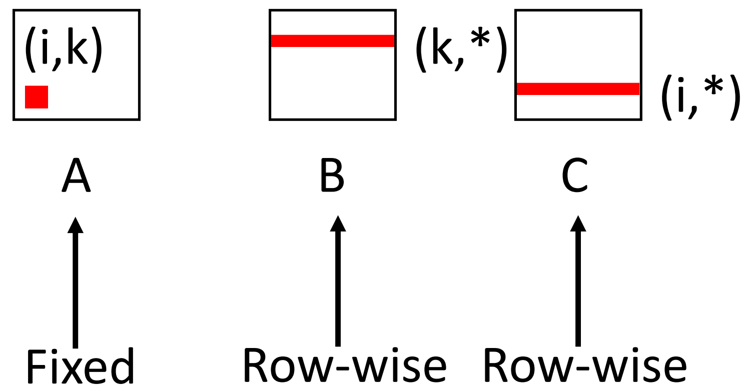
每次内层循环的丢失率/Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

矩阵乘法/ 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];
  }
}                                matmult/mm.c
```

Inner loop:

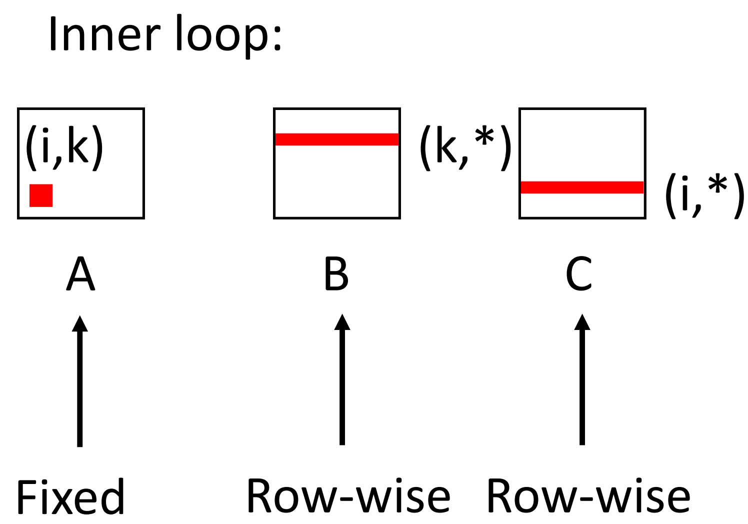


<u>每个内层循环的丢失率/Misses per inner loop iteration:</u>

<u>A</u>	<u>B</u>	<u>C</u>
0.0	0.25	0.25

矩阵乘法/ 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];
  }
}
```
matmult/mm.c



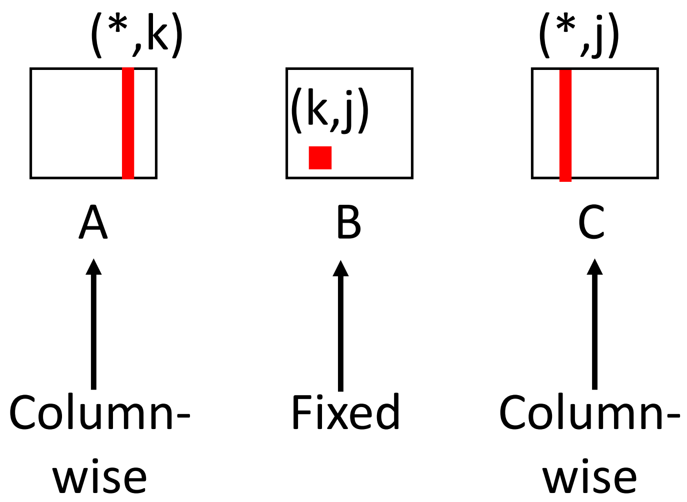
<u>每个内层循环的丢失率/Misses per inner loop iteration:</u>

A	B	C
0.0	0.25	0.25

矩阵乘法/ 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;
  }
}
```
matmult/mm.c

Inner loop:



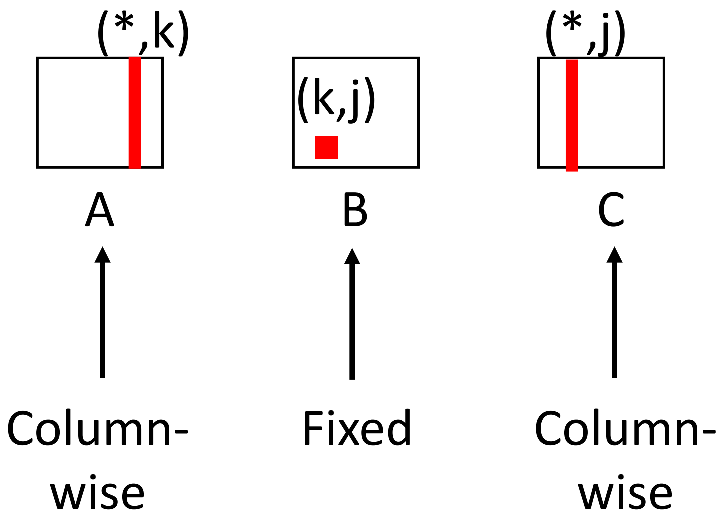
每个内层循环的丢失率/Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

矩阵乘法 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;
  }
}
                              matmult/mm.c
```

Inner loop:



<u>每个内层循环的丢失率/ Misses per inner loop iteration:</u>

<u>A</u>	<u>B</u>	<u>C</u>
1.0	0.0	1.0

矩阵乘法总结 Summary of Matrix Multiplication

```
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;
  }
}
```

ijk (& jik):

- 2 loads, 0 stores
- misses/iter = **1.25**

```
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];
 }
}
```

kij (& ikj):

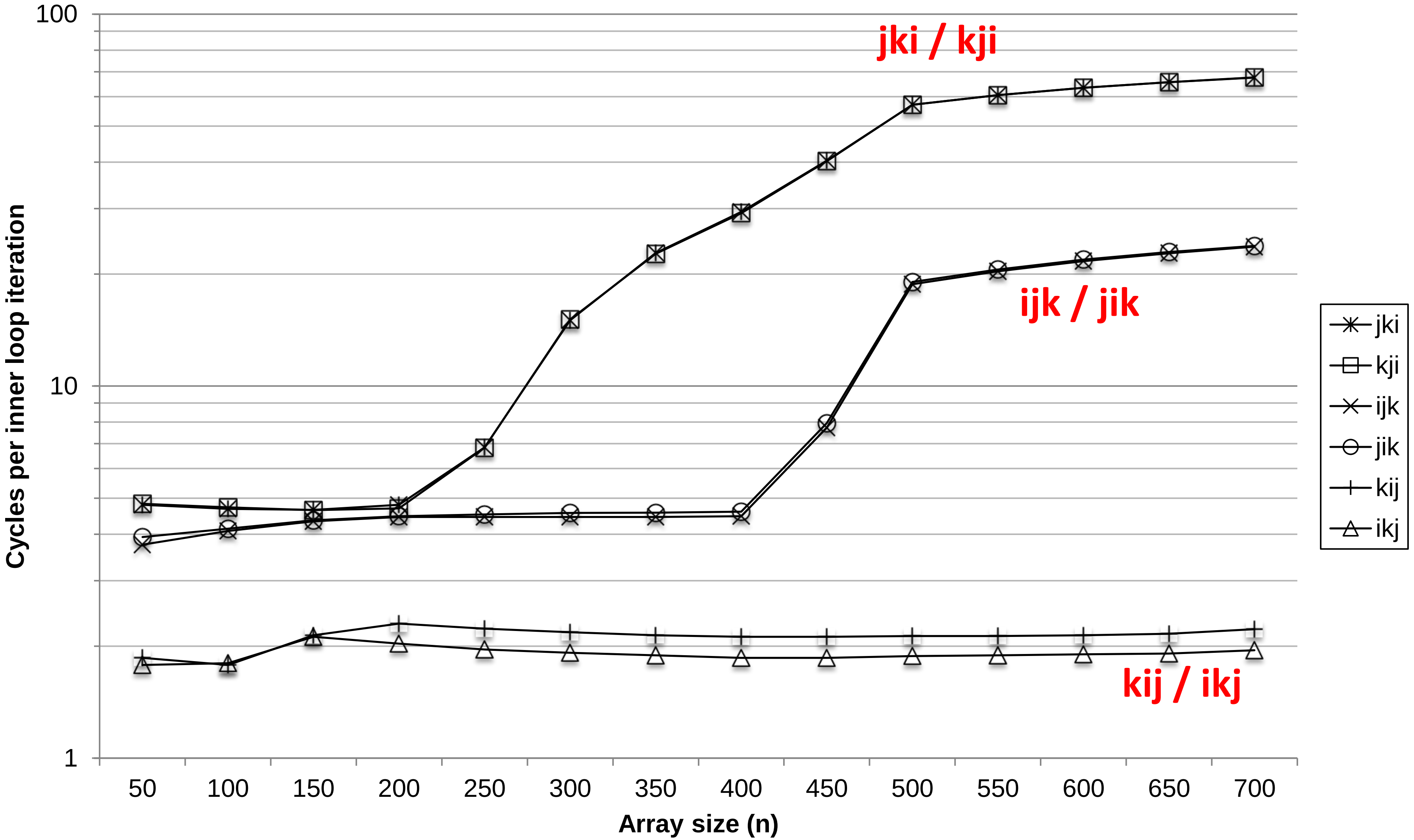
- 2 loads, 1 store
- misses/iter = **0.5**

```
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;
 }
}
```

jki (& kji):

- 2 loads, 1 store
- misses/iter = **2.0**

i7矩阵乘法性能 Core i7 Matrix Multiply Performance



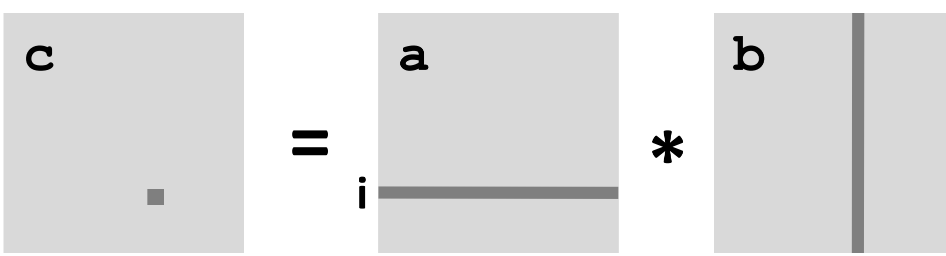
主要内容

- Cache结构和操作/Cache memory organization and operation
- Cache对性能的影响/Performance impact of caches
  - 存储墙/The memory mountain
  - 循环变换提升空间局域性Rearranging loops to improve spatial locality
  - 使用blocking提升时间局域性/Using blocking to improve temporal locality

矩阵乘法 Example: Matrix Multiplication

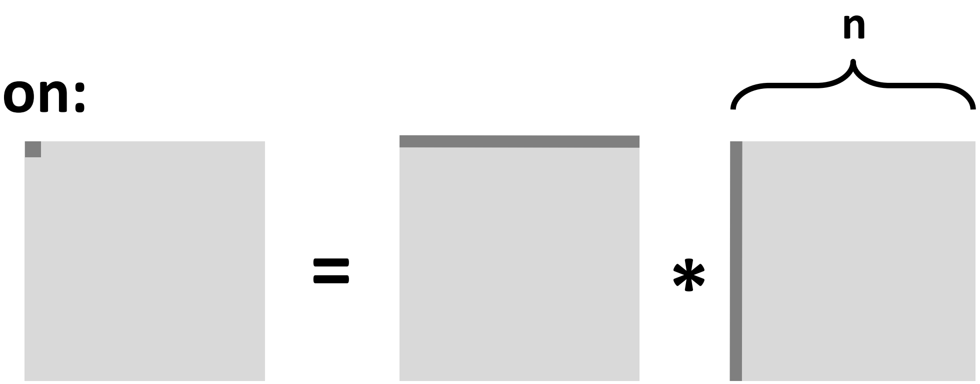
```c
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n + j] += a[i*n + k] * b[k*n + j];
}
```



Cache丢失分析 Cache Miss Analysis

- **假设 Assume:**
  - 矩阵元素类型是双精度浮点 Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
- **第一次迭代 First iteration:**
  - n/8 + n = 9n/8 misses

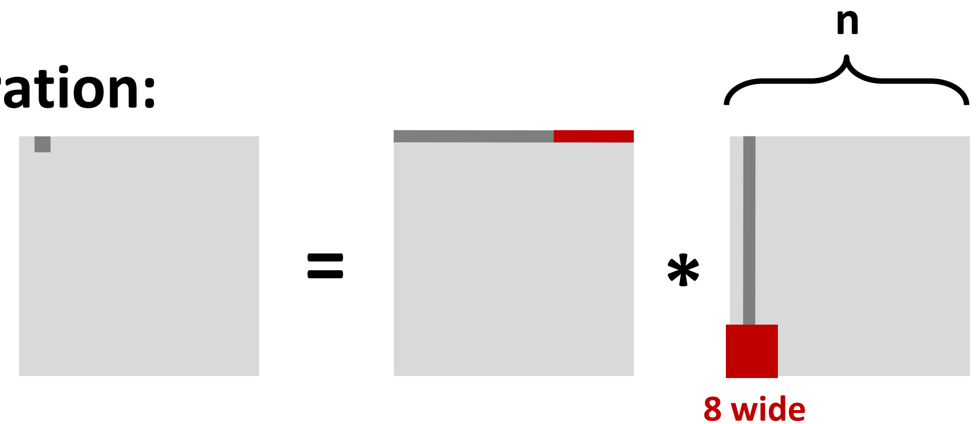


  - Afterwards in cache: (schematic)

Cache丢失分析 Cache Miss Analysis

- **假设 Assume:**
  - 矩阵元素类型是双精度浮点 Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)

- **第二次迭代 Second iteration:**
  - Again:
    n/8 + n = 9n/8 misses



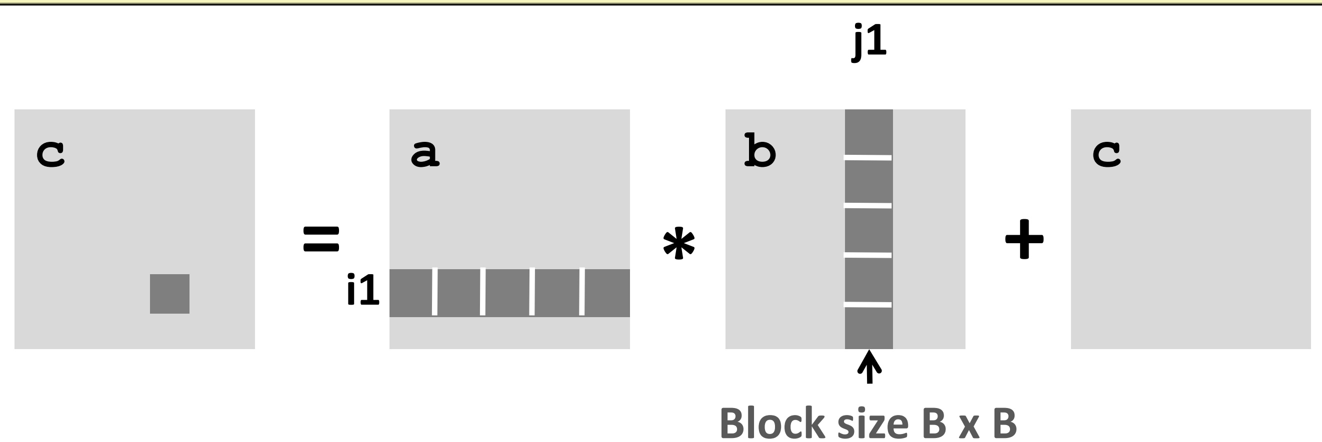
- **总计丢失 Total misses:**
  - $9n/8 * n^2 = (9/8) * n^3$

分块矩阵乘法 Blocked Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i++)
                    for (j1 = j; j1 < j+B; j++)
                        for (k1 = k; k1 < k+B; k++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```

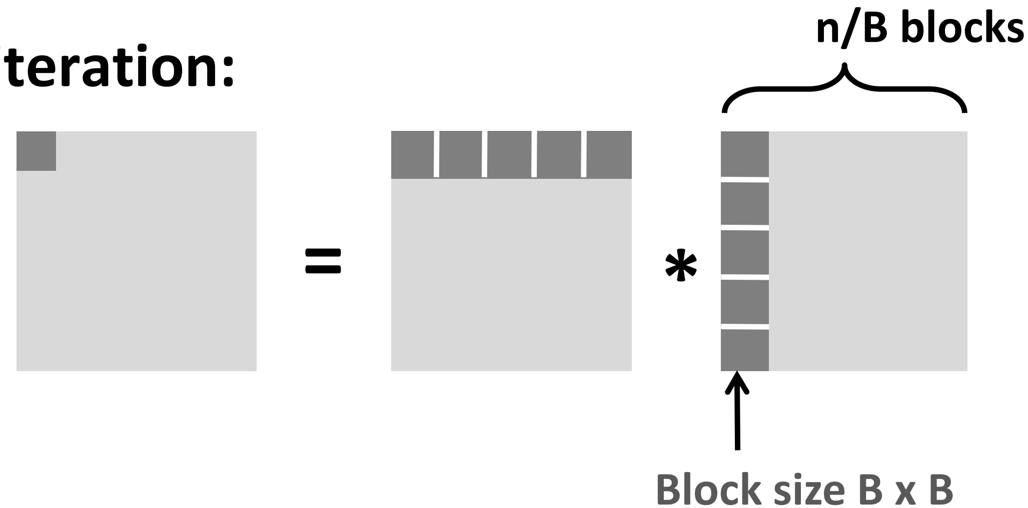
matmult/bmm.c



Cache丢失分析 Cache Miss Analysis

- **假设 Assume:**
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks fit into cache: $3B^2 < C$

- **第一次迭代 First (block) iteration:**
  - $B^2/8$ misses for each block
  - $2n/B * B^2/8 = nB/4$ (omitting matrix c)

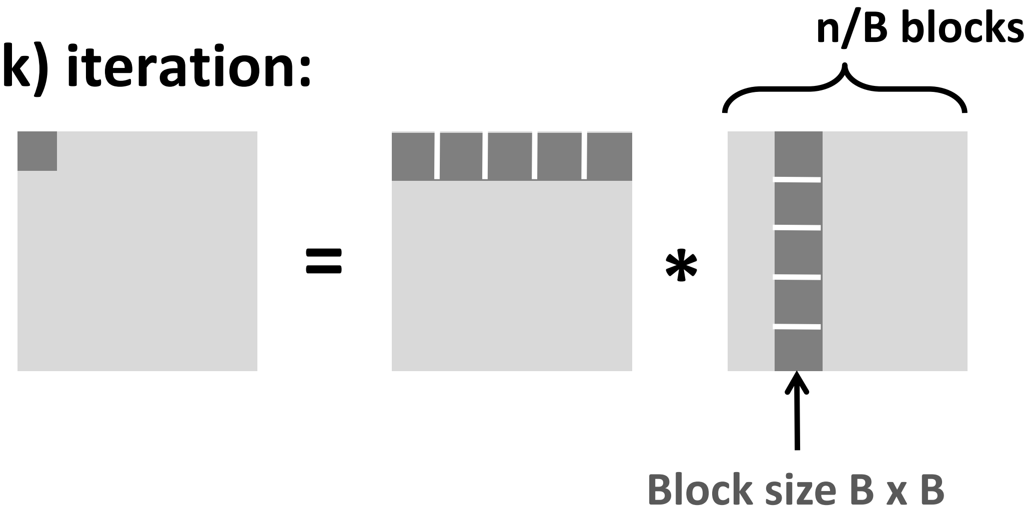


  - Afterwards in cache (schematic)

Cache丢失分析 Cache Miss Analysis

- **假设 Assume:**
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks fit into cache: $3B^2 < C$

- **第二次迭代 Second (block) iteration:**
  - Same as first iteration
  - $2n/B * B^2/8 = nB/4$



- **总计丢失 Total misses:**
  - $nB/4 * (n/B)^2 = n^3/(4B)$

分块总结 Blocking Summary

- **无分块 No blocking: (9/8) * n^3**
- **分块 Blocking: 1/(4B) * n^3**

- **Suggest largest possible block size B, but limit $3B^2 < C$!**

- **性能差异原因分析 Reason for dramatic difference:**
  - 矩阵乘法有天然的时间局域性 Matrix multiplication has inherent temporal locality:
    - 输入数据规模 Input data: $3n^2$, 计算规模 computation $2n^3$
    - 每个元素使用次数 Every array elements used O(n) times!
  - But program has to be written properly

Cache总结 Cache Summary

- **Cache对程序性能影响巨大 Cache memories can have significant performance impact**

- **编写程序时需要特别设计 You can write your programs to exploit this!**
  - 关注内层循环，大量计算和访存 Focus on the inner loops, where bulk of computations and memory accesses occur.
  - 充分利用空间局域性， 按顺序连续读取 Try to maximize spatial locality by reading data objects with sequentially with stride 1.
  - 充分利用时间局域性，一次读入多次使用 Try to maximize temporal locality by using a data object as often as possible once it’s read from memory.