

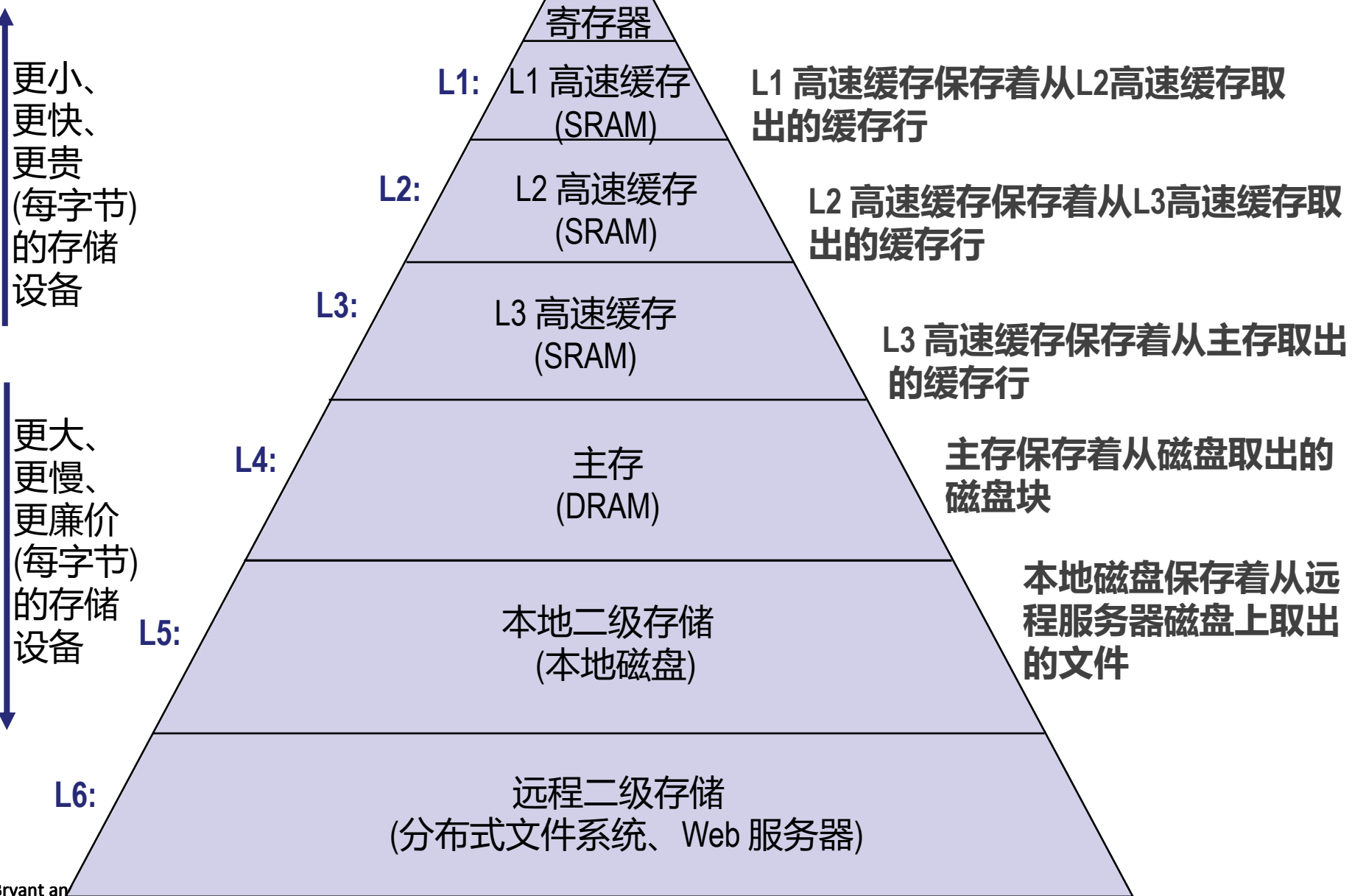
第六章 Part2 :高速缓存存储器

- 教 师： 郑贵滨
- 计算机科学与技术学院
- 哈尔滨工业大学

主要内容

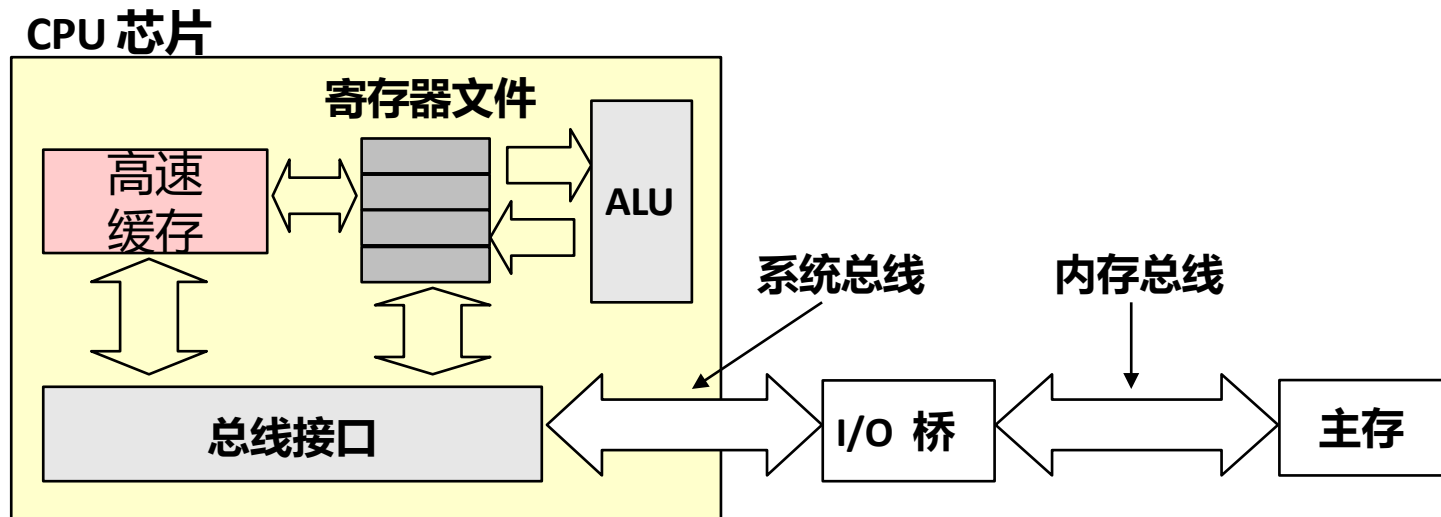
- 高速缓存存储器组织结构和操作
- 高速缓存对程序性能的影响
 - 存储器山
 - 重新排列循环以提高空间局部性
 - 使用分块来提高时间局部性

存储器层次结构

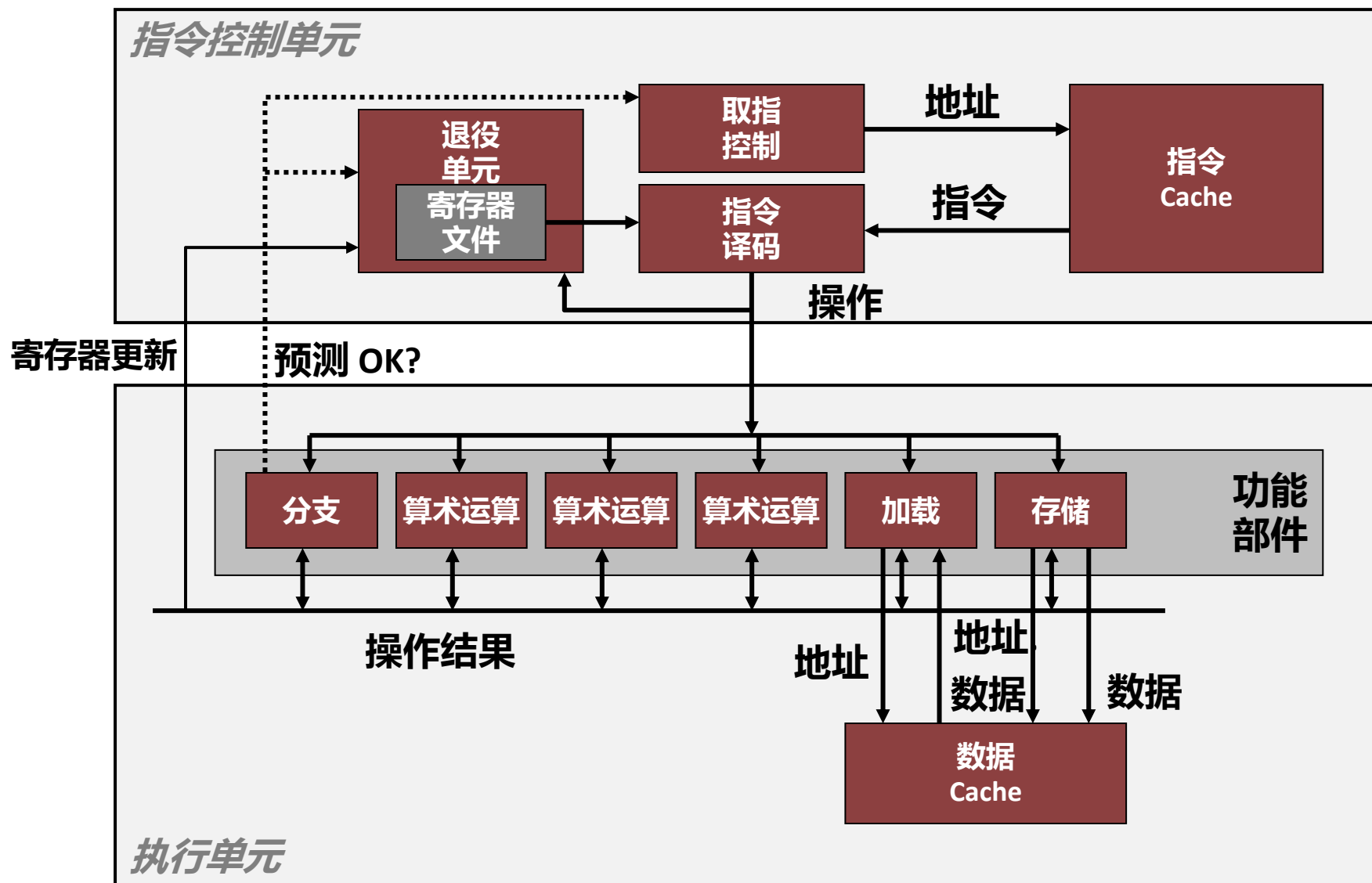


高速缓存存储器

- **高速缓存存储器**是小型的、快速的基于SRAM的存储器是在硬件中自动管理的
 - 保存经常访问主存的块
- CPU 首先查找缓存中的数据
- 典型的系统结构



回顾：现代CPU设计



实例

台式机 PC

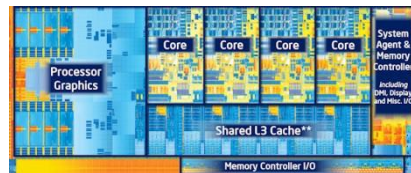


Source: Dell

CPU (Intel Core i7)



Source: PC Magazine



Source: techreport.com

主板

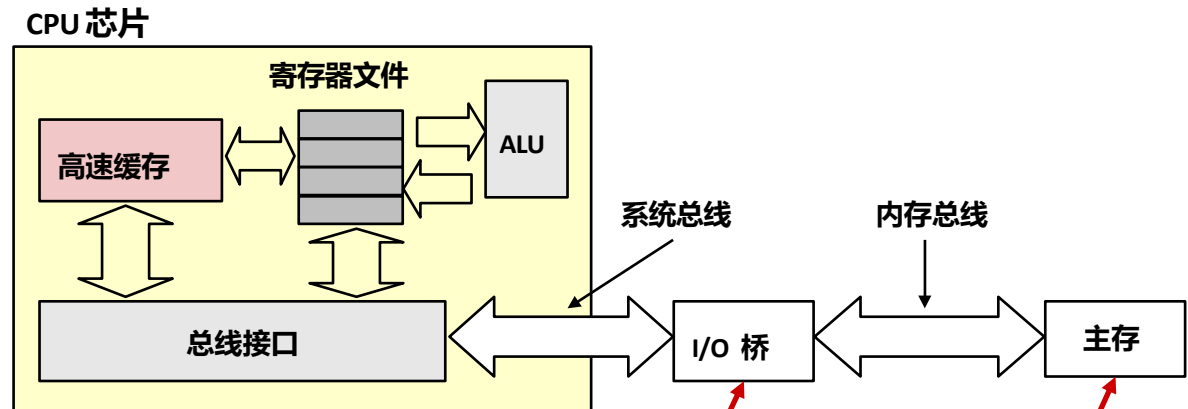


Source: Dell

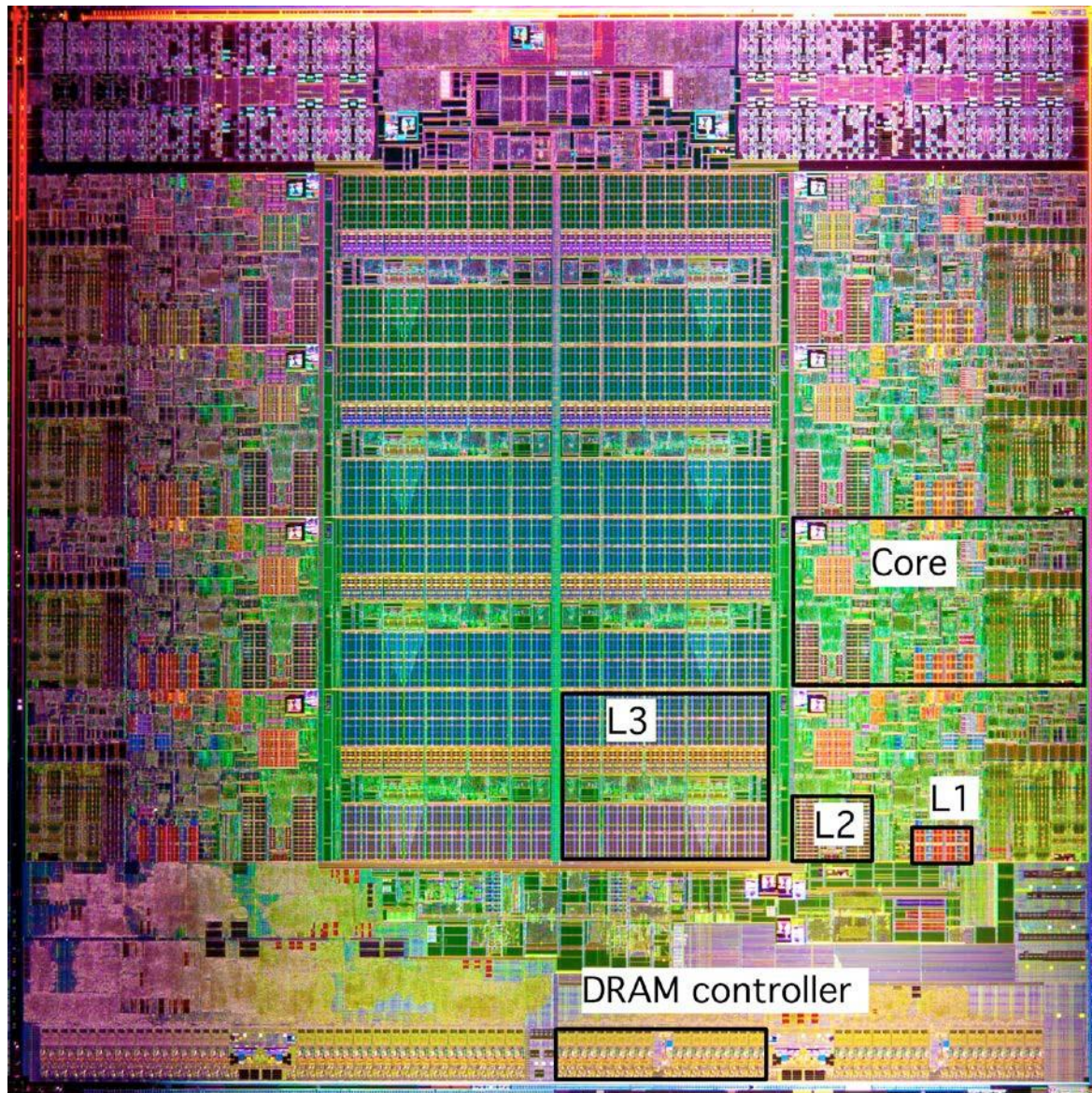
主存 (DRAM)



Source: Dell



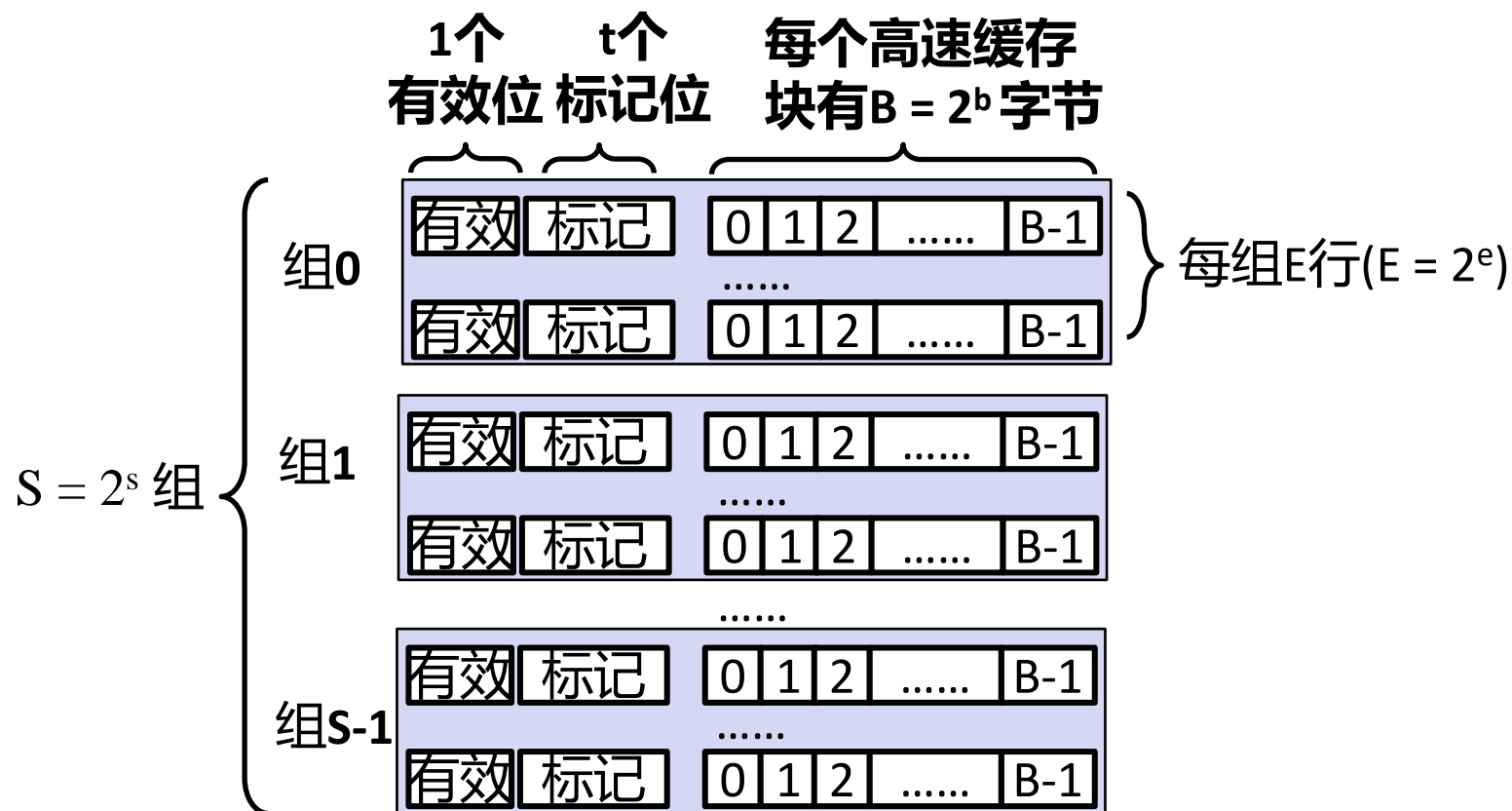
实例(Cont.)



Intel Sandy Bridge
Processor Die

L1: 32KB 指令 + 32KB 数据
L2: 256KB
L3: 3–20MB

通用的高速缓存存储器组织结构(S, E, B)

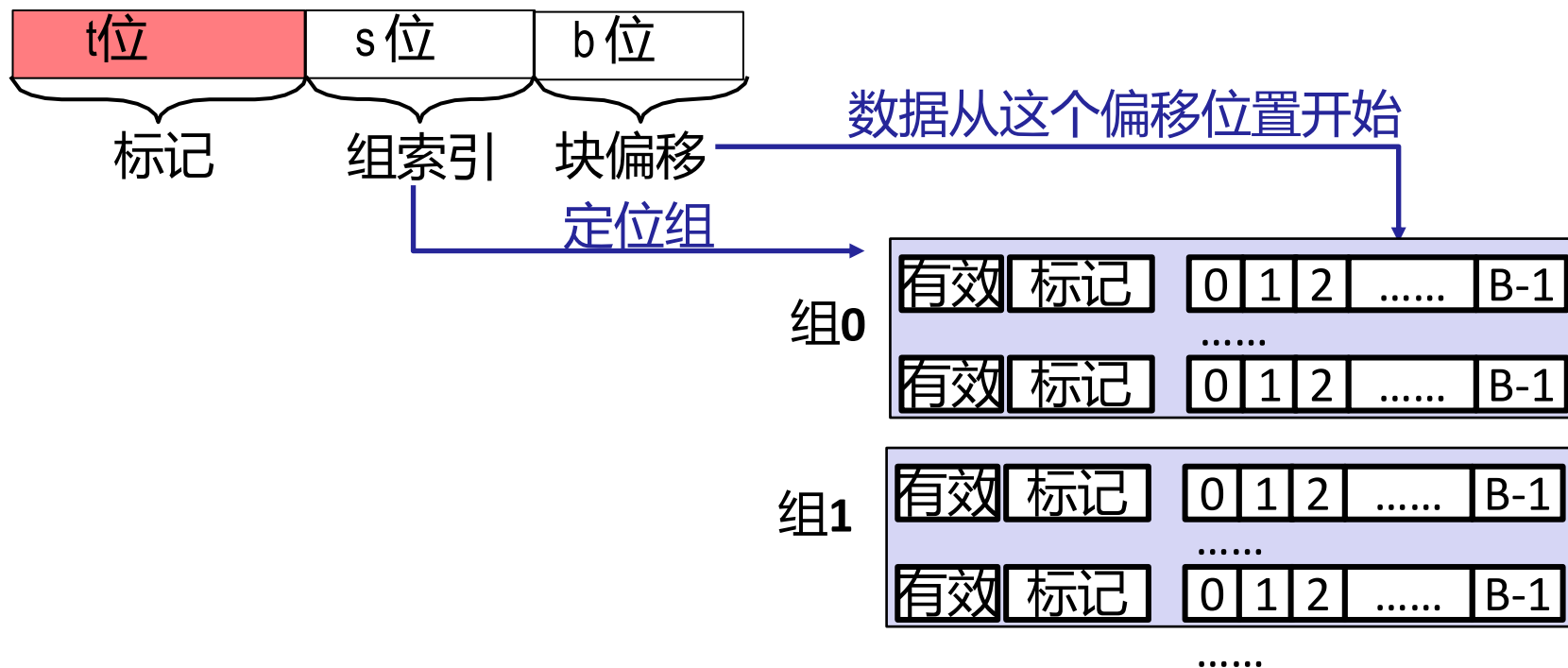


高速缓存大小 $C = S \times E \times B$ **数据字节**

高速缓存读

- 定位组
- 定位行：检查集合中的任何行是否有匹配的标记
是 + 行有效 —— 命中
- 定位数据：从块偏移开始的数据

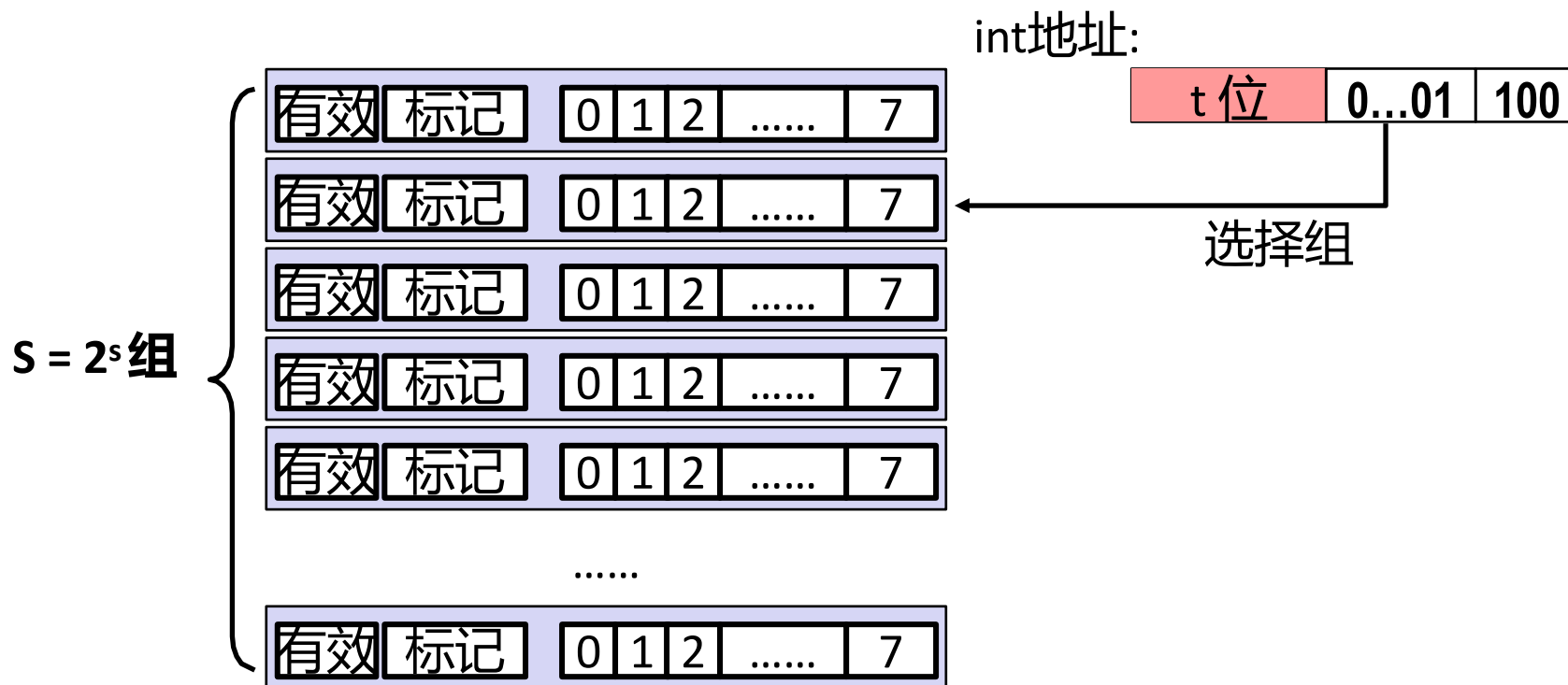
地址:



1、直接映射高速缓存 ($E = 1$)

- 直接映射: 每一组只有一行

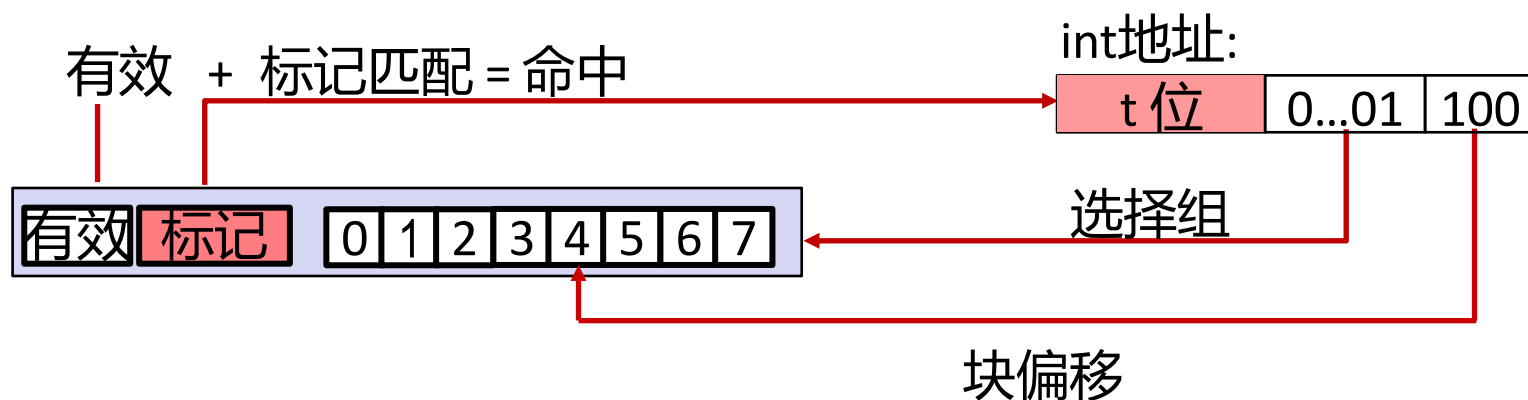
假设: 缓存块大小为8字节



方式1: 直接映射高速缓存 ($E = 1$)

- 直接映射: 每一组只有一行

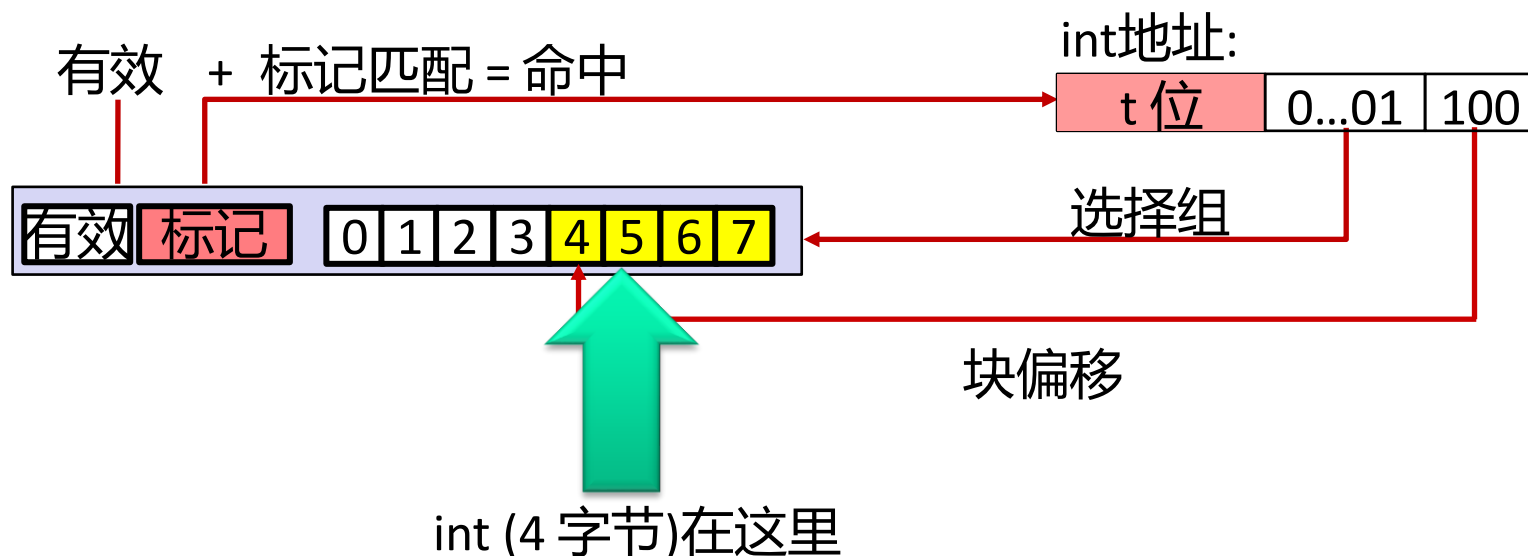
假设: 缓存块大小为8字节



方式1: 直接映射高速缓存 ($E = 1$)

- 直接映射: 每一组只有一行

假设: 缓存块大小为8字节



如果标记不匹配: 旧的行被驱逐、替换

直接映射高速缓存模拟

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

M=16 字节 (4-位 地址), B=2 字节/块, S=4 组,
E=1 块/组

地址跟踪(读, 每次读一个字节):

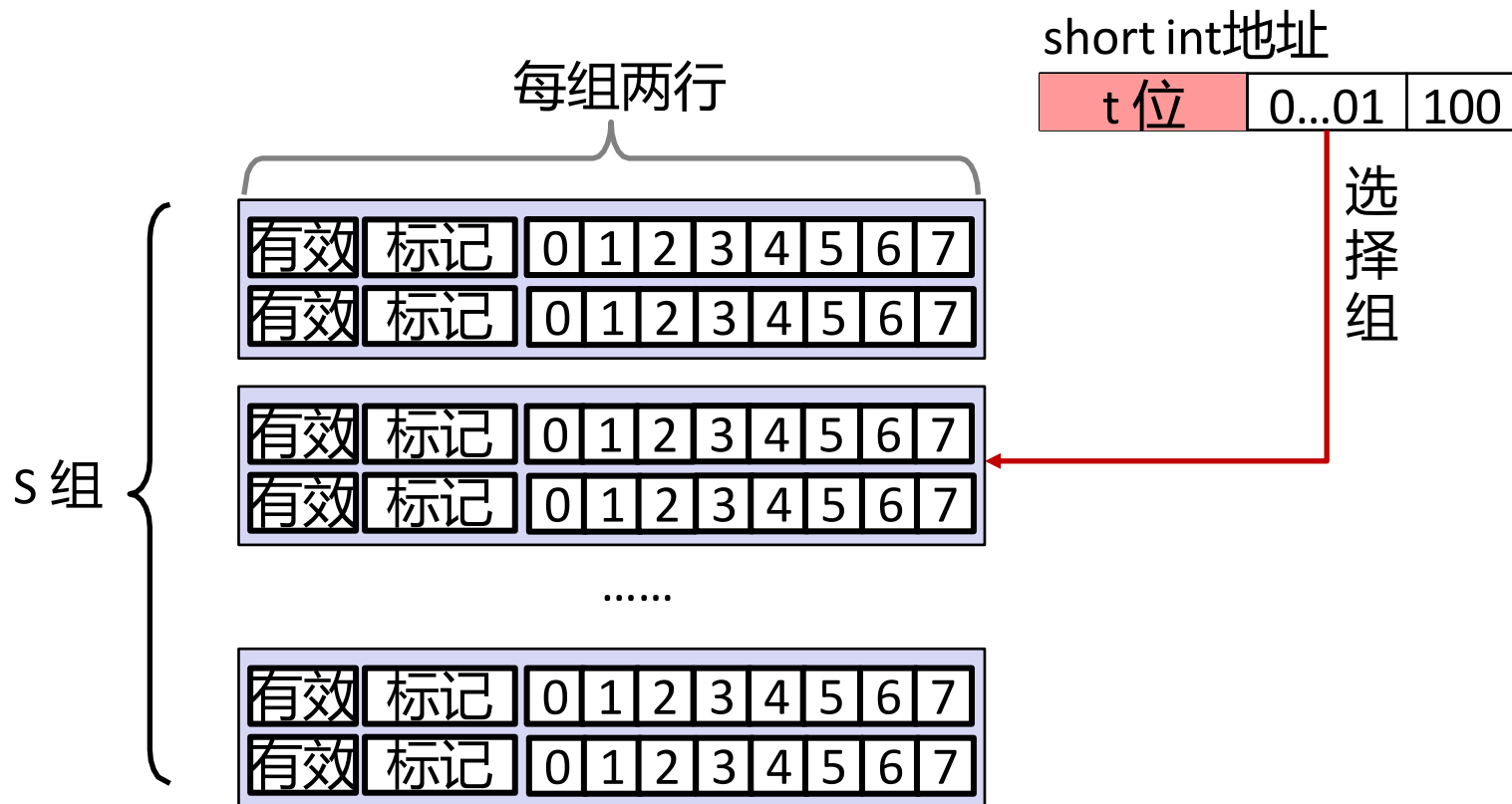
0 [0000₂], 不命中
 1 [0001₂], 命中
 7 [0111₂], 不命中
 8 [1000₂], 不命中
 0 [0000₂] 不命中

	v	标记	块
组0	1	0	M[0-1]
组1	0	?	?
组2	0	?	?
组3	1	0	M[6-7]

方式2: E-路组相联高速缓存 (E = 2)

E = 2: 每组两行

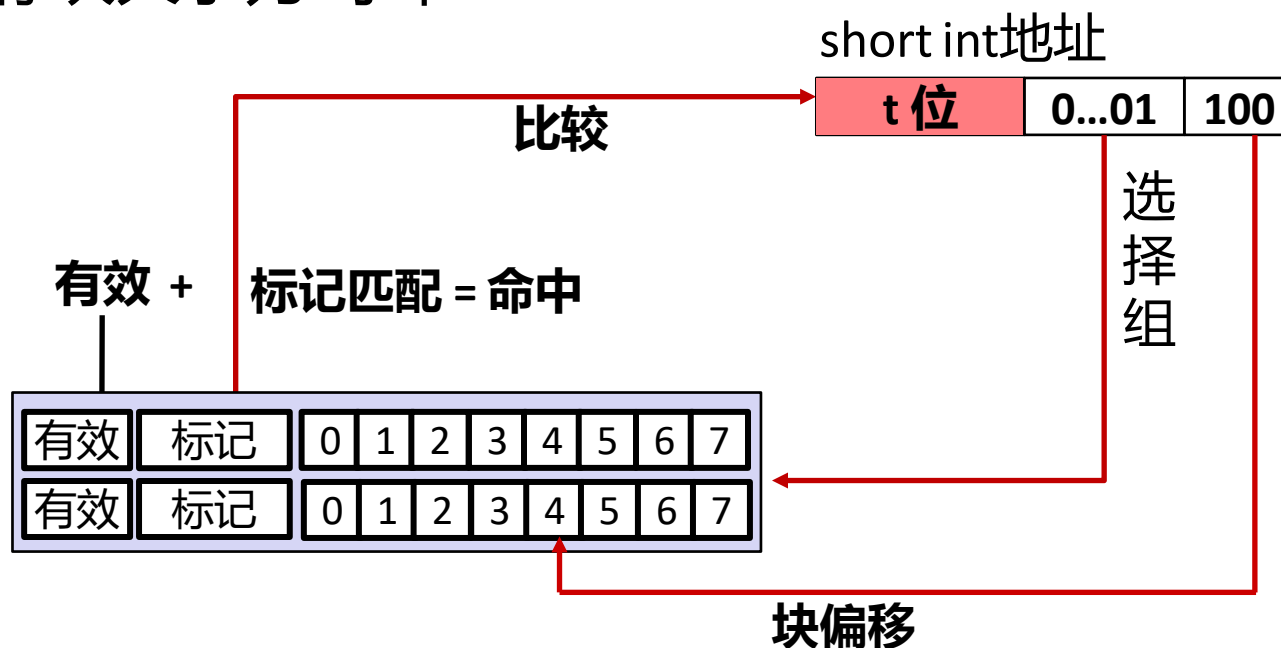
假设:缓存块大小为8字节



方式2: E-路组相联高速缓存(E = 2)

E = 2: 每组两行

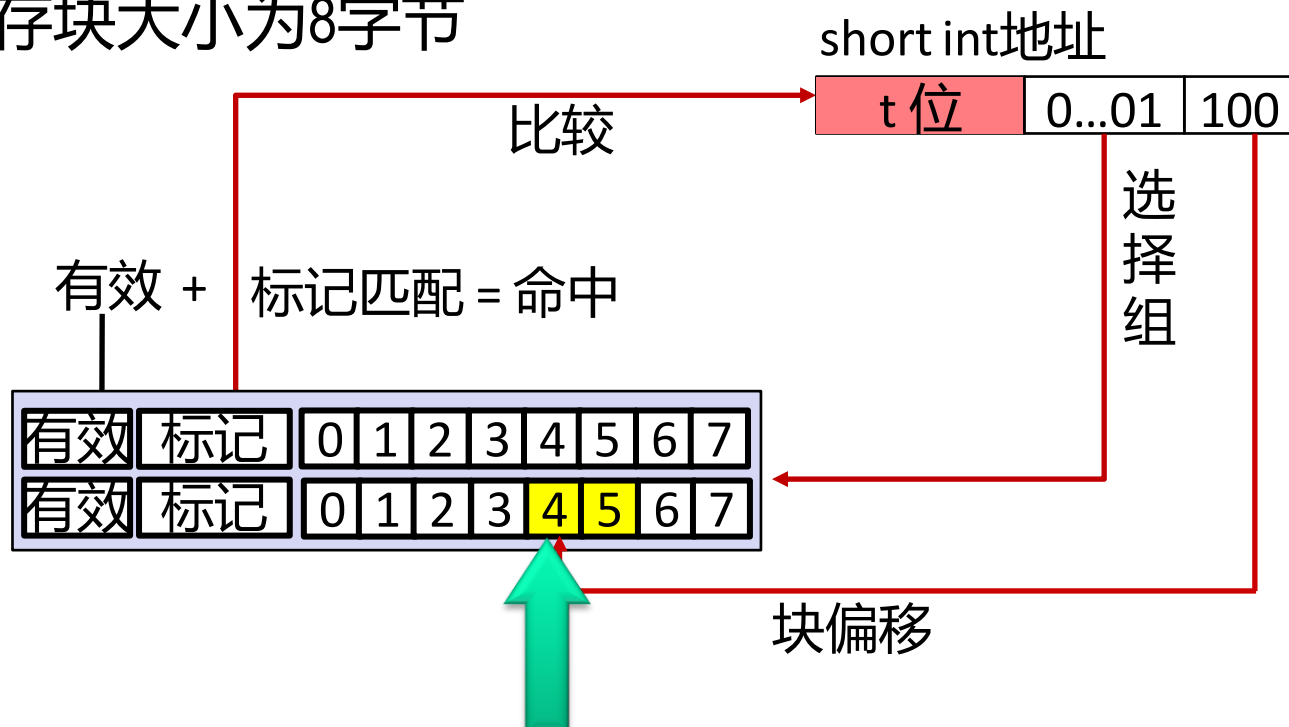
假设:缓存块大小为8字节



方式2: E-路组相联高速缓存(E = 2)

E = 2: 每组两行

假设:缓存块大小为8字节



short int (2 字节) 在这里

不匹配:

- 在组中选择1行用于驱逐和替换
- 替换策略: 随机、最近最少使用(LRU), ...

方式2: 2-路组相联缓存模拟

t=2 s=1 b=1

xx	x	x
----	---	---

M=16 字节地址, B=2 字节/块, S=2 组,
E=2 块/组

地址跟踪(读, 每次读一个字节):

0 [0000₂], miss

1 [0001₂], hit

7 [0111₂], miss

8 [1000₂], miss

0 [0000₂] hit

	v	标记	块
组0	1	10	M[8-9]
	1	00	M[0-1]
组1	1	01	M[6-7]
	0	?	?

Cache的种类

■ 1、直接映射： $E=1$

主存中的一个块只能映射到Cache的某一特定块。

■ 2、全相联映射： $S=1$

主存中任何一块都可以映射到Cache中的任何一块。

■ 3、组相联映射： $E>1 \ \&\& \ S>1$

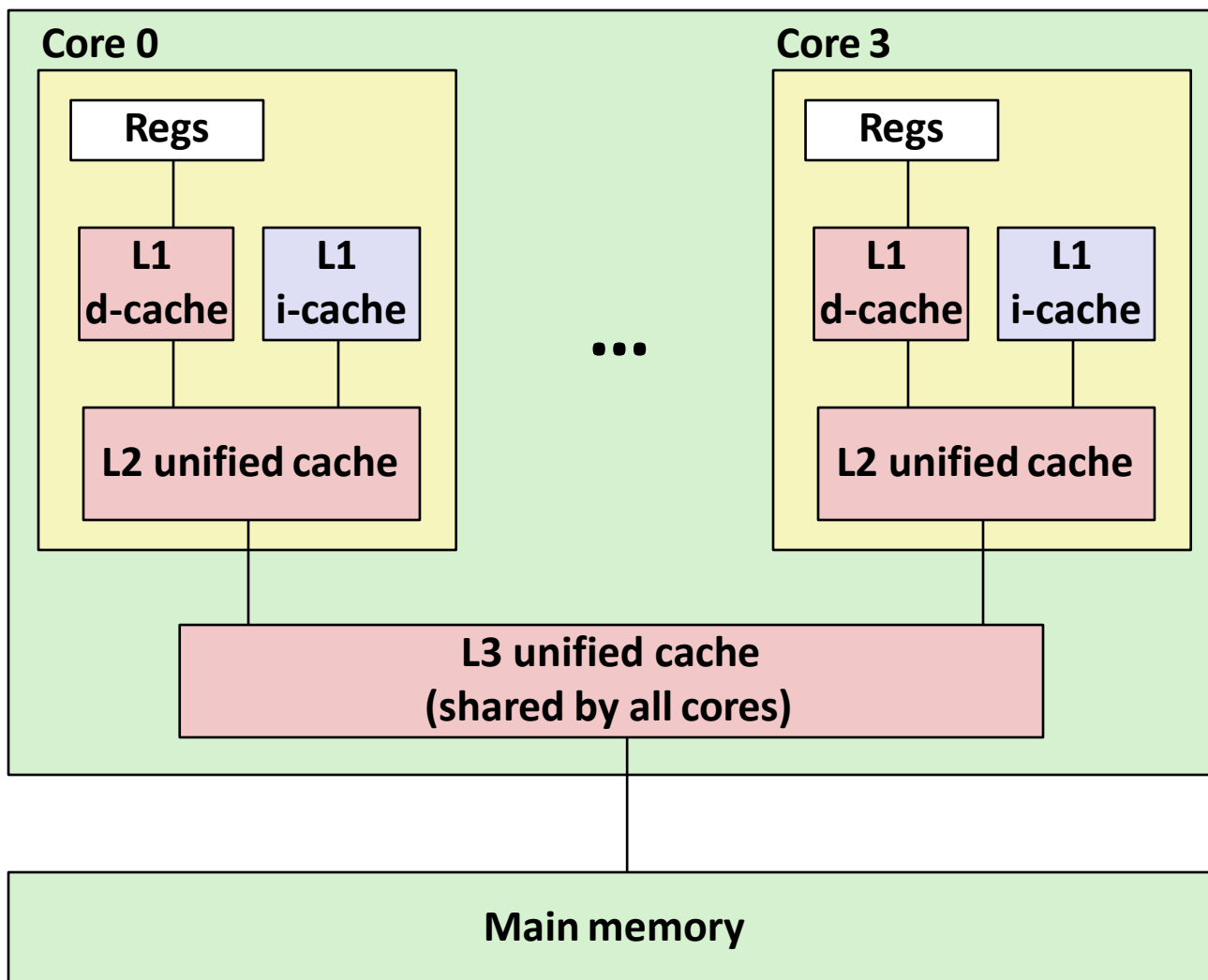
是前两种方法的折中方案，适度兼顾二者的优点，尽量避免二者的缺点，普遍采用。

关于怎么写?

- 存在多个数据副本:
 - L1, L2, L3, 主存, 磁盘
- 在写命中时要做什么?
 - 直写 (Write-through, 立即写入存储器)
 - 写回 (Write-back, 推迟到缓存行要替换时才写入内存)
 - 需要一个修改位 (标识缓存行与内存是否相同/有修改)
- 写不命中时要做什么?
 - 写分配 (Write-allocate 加载到缓存, 更新这个缓存行)
 - 如后续有较多向该位置的写, 优势明显
 - 非写分配 (No-write-allocate 直接写到主存中, 不加载到缓存中)
- 典型方案
 - 直写 + 非写分配
 - 写回 + 写分配

Intel Core i7高速缓存层次结构

处理器封装



L1 指令高速缓存和 数据高速缓存:

32 KB, 8-way,
访问时间: 4周期

L2 统一的高速缓存:

256 KB, 8-way,
访问时间: 10 周期

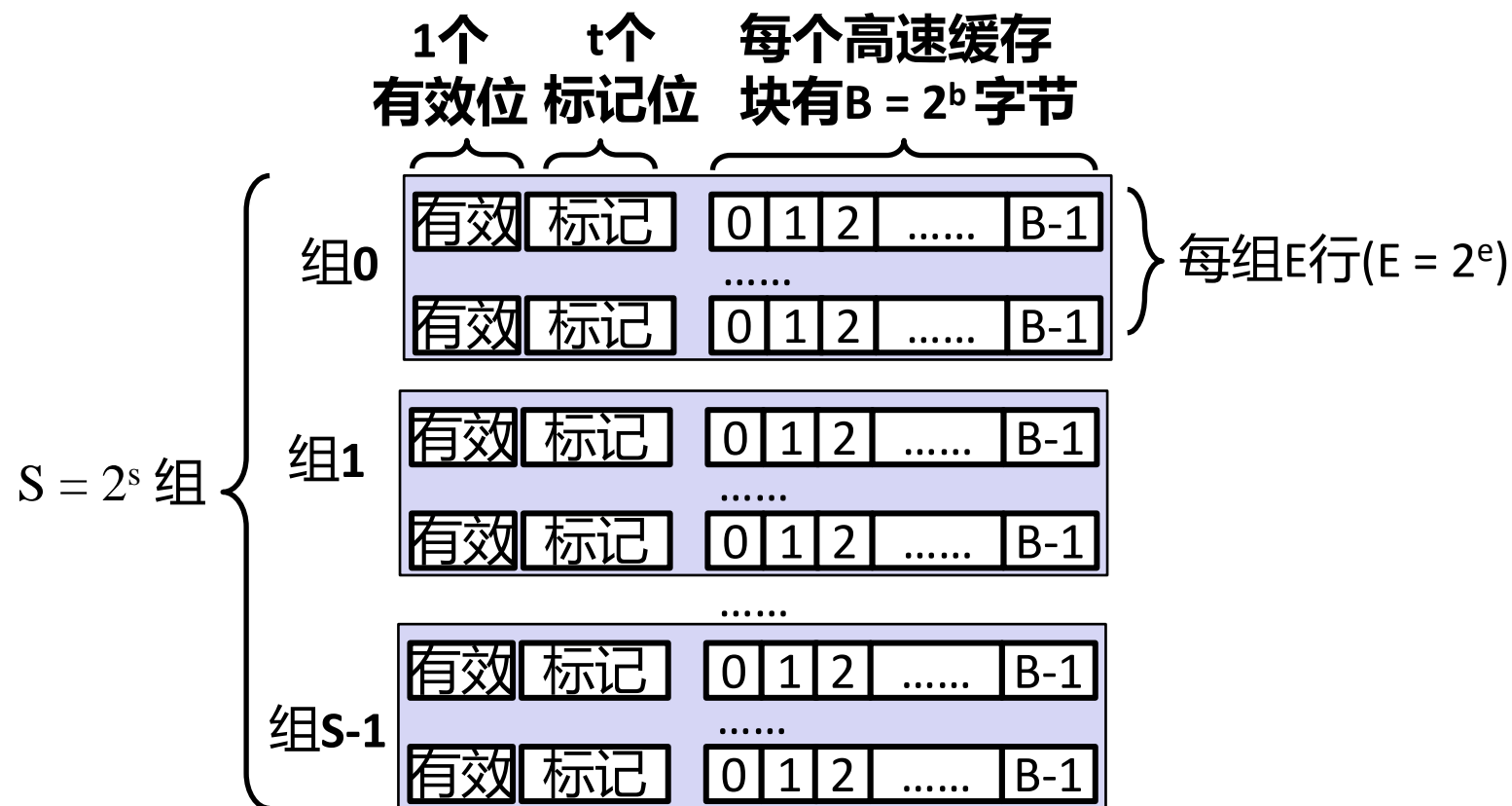
L3 统一的高速缓存:

8 MB, 16-way,
访问时间: 40-75 周期

块大小:

所有缓存都是64 字节

通用的高速缓存存储器组织结构(S, E, B)



高速缓存大小 $C = S \times E \times B$ **数据字节**

例子: Core i7 L1 数据缓存

32 KB、8路组相联
64 字节/块
48/52 位地址范围

B =

S = s =

E = e =

C =

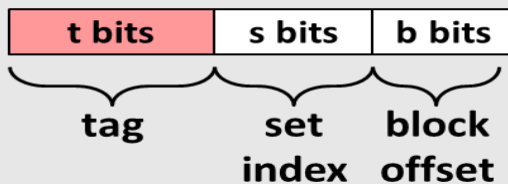
$S = 2^s$ 组



高速缓存大小 $C = S \times E \times B$ 数据字节

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Address of word:



块偏移: ?位

组索引: ?位

标 记: ?位

栈地址:

0x00007f7262a1e010

块偏移: 0x??

组索引: 0x??

标 记: 0x??

例子: Core i7 L1 数据缓存

32 KB、8路组相联
64 字节/块
48/52 位地址范围

$$B = 64$$

$$S = 64, s = 6$$

$$E = 8, e = 3$$

$$C = 64 \times 64 \times 8 = 32,768$$

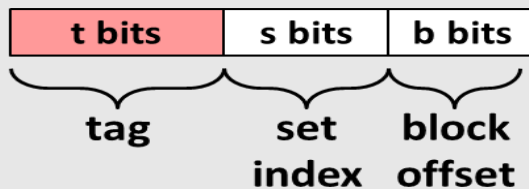
$$S = 2^s \text{ 组}$$



高速缓存大小 $C = S \times E \times B$ 数据字节

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Address of word:



块偏移: 6 bits

组索引: 6 bits

标 记: 剩余全部bit
(36/40 bits)

栈地址:

0x00007f7262a1e010

0000 0001 0000

块偏移: 0x10

组索引: 0x0

标 记: 0x7f7262a1e

高速缓存性能指标

■ 不命中率

- 一部分内存引用在缓存中没有找到 (不命中 / 访问)
= $1 - \text{命中率}$
- 典型数值(百分比):
 - L1: 3-10%
 - L2: 可以相当小(e.g., $< 1\%$) , 依赖于L2大小等因素。

■ 命中时间

- 从高速缓存向处理器发送一行的时间
 - 时间包括判断行是否在缓存中
- 典型数值:
 - L1 4个时钟周期
 - L2 10个时钟周期

■ 不命中处罚

- 由于不命中需要额外的时间
 - 通常主存需50-200周期(趋势: 增加!)

不命中率 VS 命中率

■ 命中 or 不命中：差距巨大

- 如果只有L1 和 主存，那么可以差100倍
- 你会相信99%命中率要比97%好两倍？

■ 假 设

- 缓存命中时间为1个周期
- 不命中处罚要100个周期

■ 平均访问时间

- 97% 命中率: $1 \text{ 周期} + 0.03 \times 100 \text{ 周期} = 4 \text{ 周期}$
- 99% 命中率: $1 \text{ 周期} + 0.01 \times 100 \text{ 周期} = 2 \text{ 周期}$

■ 这就是为什么用 “不命中率” 而不是 “命中率”

编写高速缓存友好的代码

- 让常见的情况运行得快
 - 专注在核心函数和内循环上
- 使用内层循环的缓存不命中数量降到最低
 - 反复引用变量好(时间局部性)
 - 步长为1的引用模式好(空间局部性)
- 关键思想

局部性的定性概念是通过缓冲存储器的理解而量化

主要内容

- 高速缓存的组织结构和运算
- 高速缓存对程序性能的影响
 - 存储器山
 - 重新排列以提升空间局部性
 - 使用块来提高时间局部性

存储器山

■ 读吞吐量 (读带宽)

- 每秒从存储系统中读取的字节数(MB/s)

■ 存储器山：测量读取吞吐量作为空间和时间局部性的函数

- 描述存储系统性能的一种直观、紧凑的方式

存储器测试函数

```

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));
}

```

用多种elems、stride组合调用test()

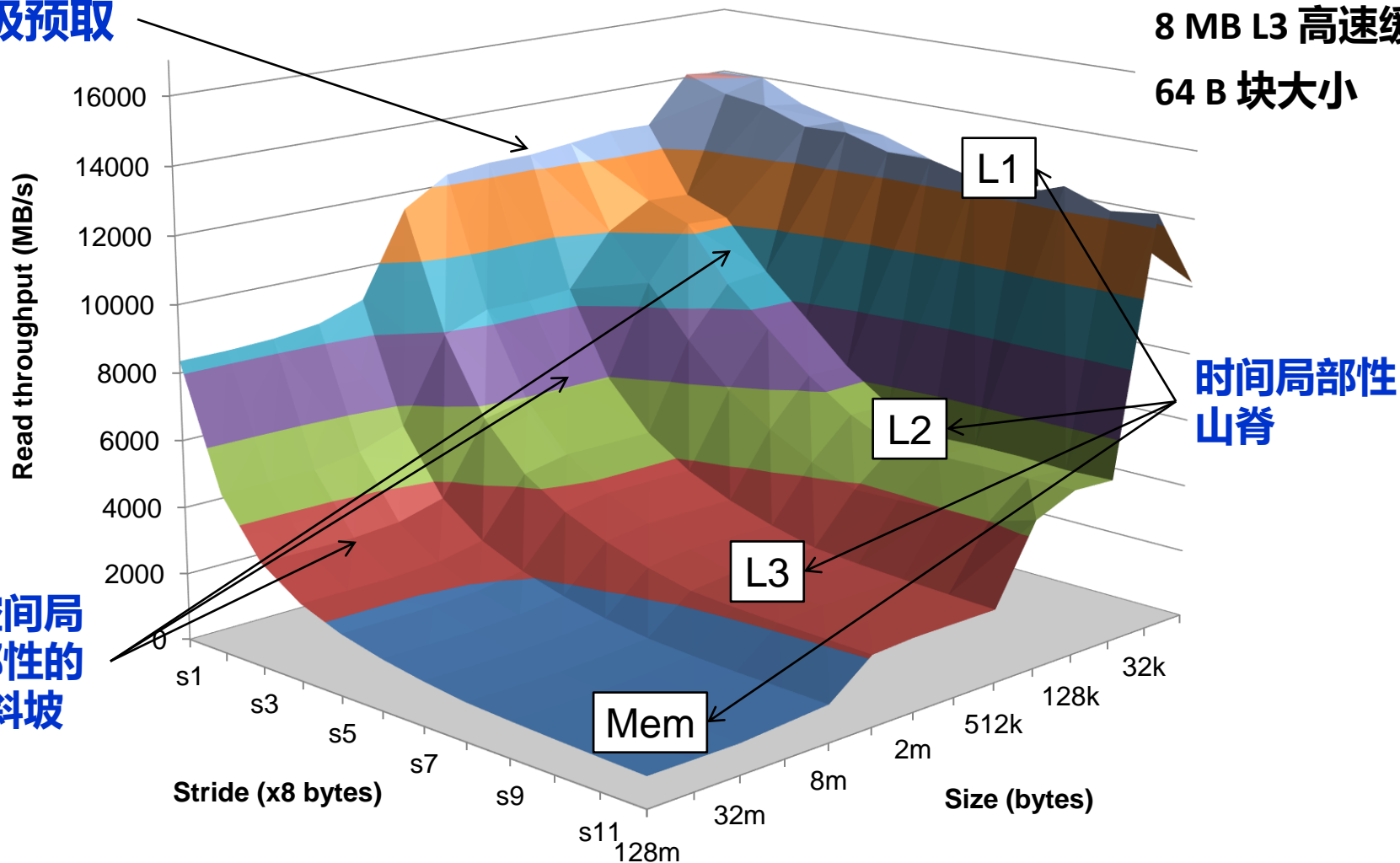
对于每个 elems和stride:

1. 调用一次test()函数，预热缓存
2. 再次调用test() 函数，测量读操作的吞吐量(MB/s)

存储器山

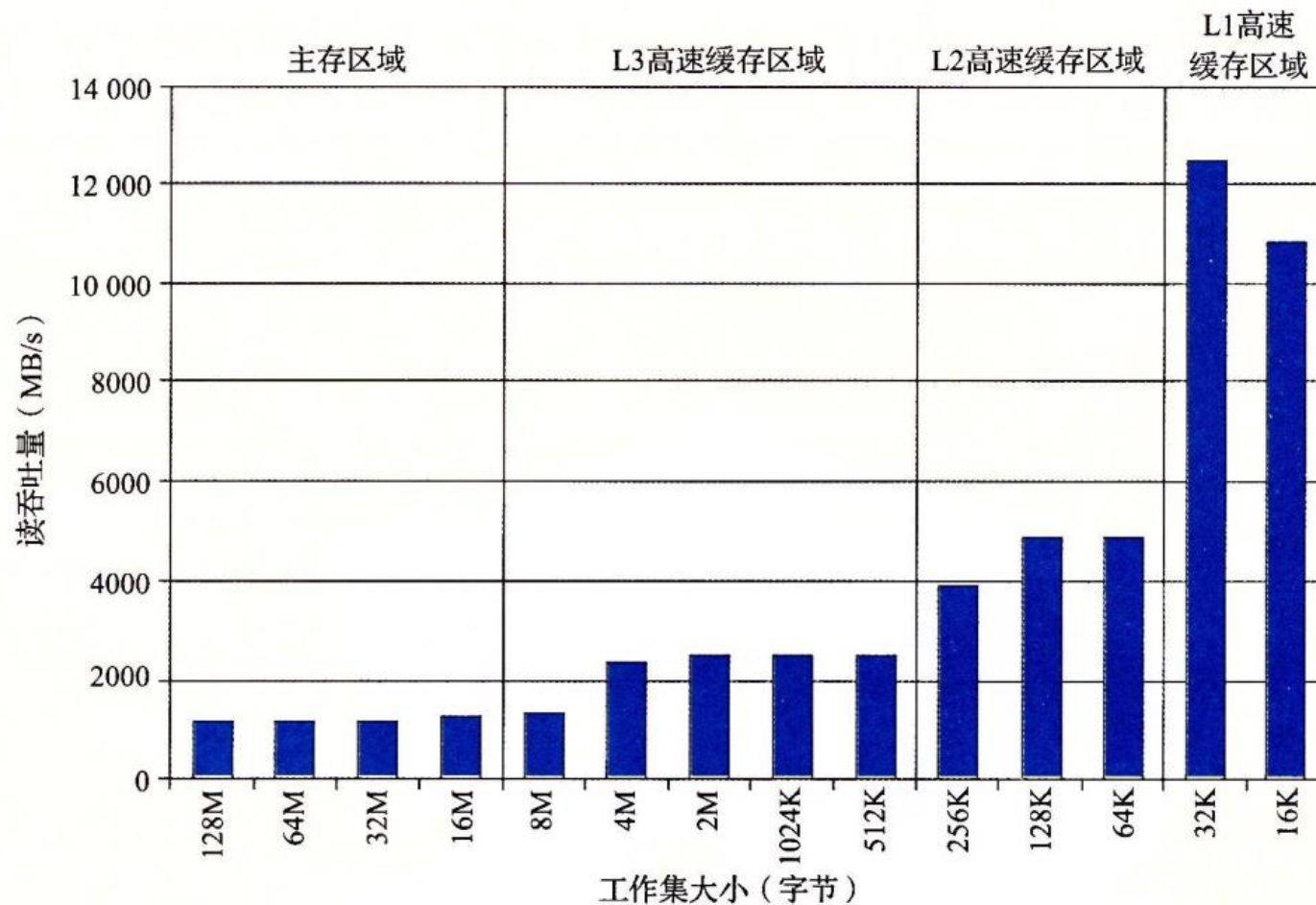
Core i7 Haswell
2.1 GHz
32 KB L1 高速缓存
256 KB L2 高速缓存
8 MB L3 高速缓存
64 B 块大小

积极预取



工作集大小vs缓存尺寸关系

■ stride=8时，吞吐量和workset的关系



主要内容

- 高速缓存的组织结构和运算
- 高速缓存对程序性能的影响
 - 存储器山
 - **重新排列以提升空间局部性**
 - 使用块来提高时间局部性

矩阵乘法的例子

■ 描述:

- $N \times N$ 矩阵相乘
- 矩阵元素类型是 doubles (8 字节)
- 总共 $O(N^3)$ 个操作
- 每个元素都要读 N 次
- 每个目标中都要对 N 个值求和
 - 但也可以保存在寄存器中

```
/* 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;  
    }  
}
```

变量sum保存在寄存器中

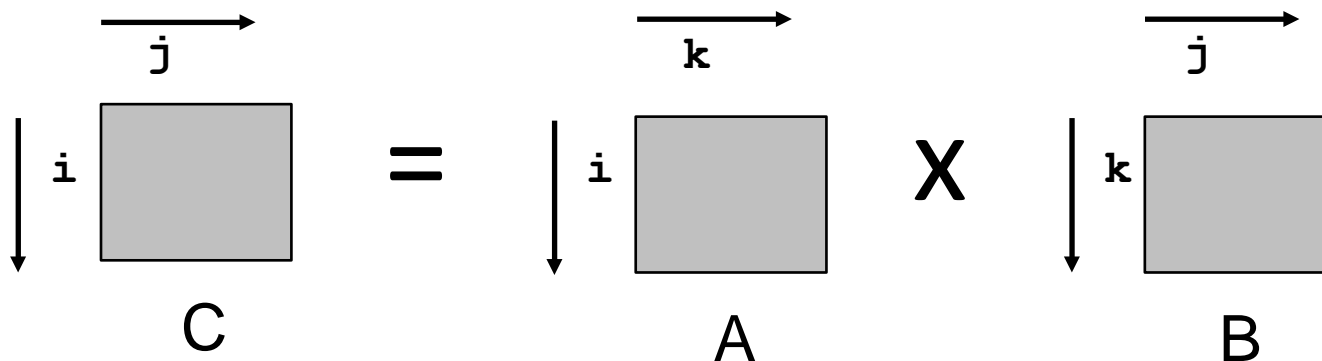
矩阵相乘不命中率分析

■ 假设:

- 块大小 = 32B (足够容纳4个double数)
- 矩阵的维数N非常大: $1/N$ 大约为 0.0
- 缓存并未大到足够容纳多行

■ 分析方法:

- 看内循环的访问模式



内存中C数组的布局(回顾)

- C 数组分配按行顺序
 - 每行在连续的内存位置
- 按行扫描:
 - `for (i = 0; i < N; i++)`
 `sum += a[0][i];`
 - 访问连续的元素
 - 如果块大小 $(B) > \text{sizeof}(a_{ij})$ 字节, 利用空间局部性
 不命中率 = $\text{sizeof}(a_{ij}) / B$
- 按列扫描:
 - `for (i = 0; i < n; i++)`
 `sum += a[i][0];`
 - 访问远隔的元素
 - 没有空间局部性!
 - 不命中率 = 1 (i.e. 100%)

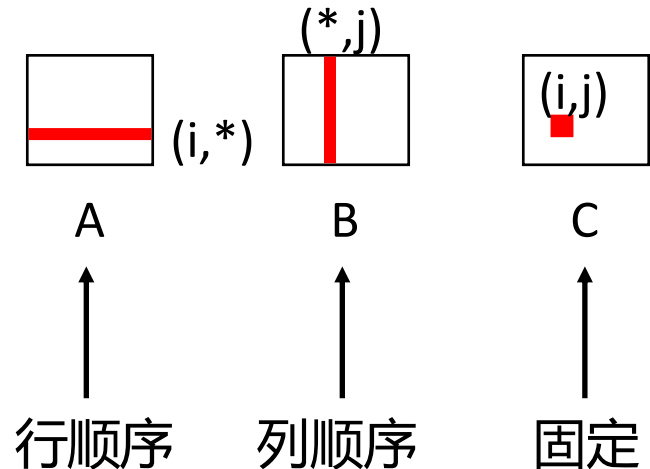
矩阵乘法(ijk)

```

/* 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

```

内层循环:



每次内层循环迭代的不命中数:

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

块大小 = 32B (4 doubles)

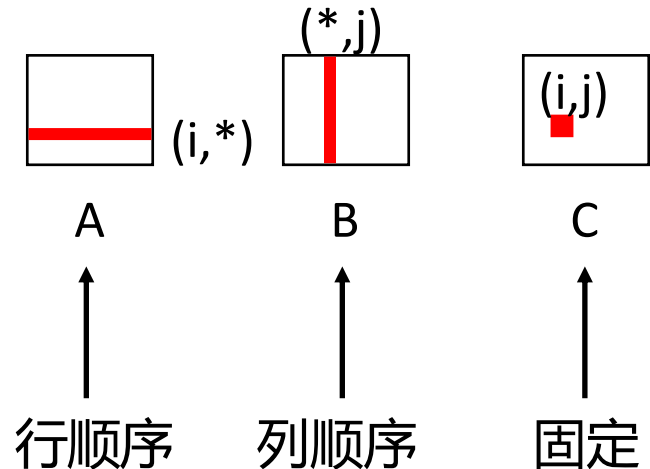
矩阵乘法(jik)

```

/* ijk */
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

```

内层循环:



每次内层循环迭代的不命中数:

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

块大小 = 32B (4 doubles)

矩阵乘法(kij)

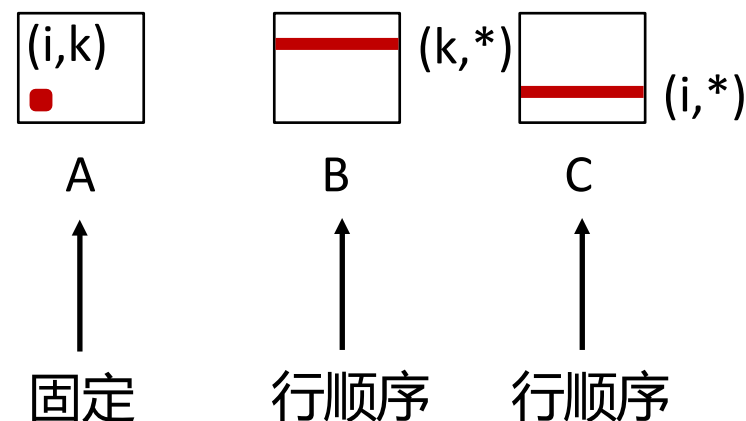
```

/* 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

内层循环:



每次内层循环迭代的不命中数:

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

块大小 = 32B (4 doubles)

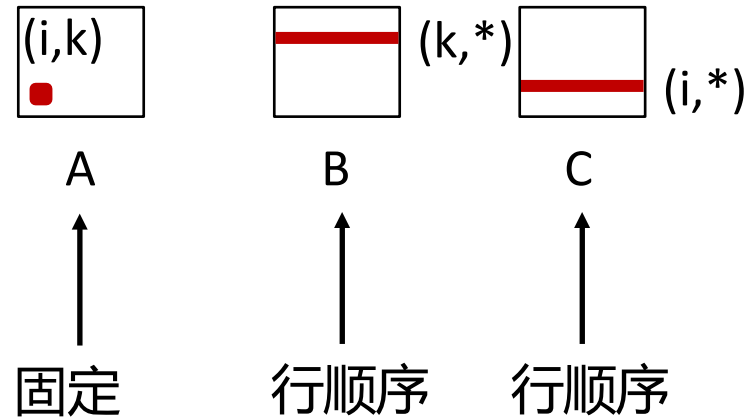
矩阵乘法(ikj)

```

/* 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

```

内层循环:



每次内层循环迭代的不命中数:

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

块大小 = 32B (4 doubles)

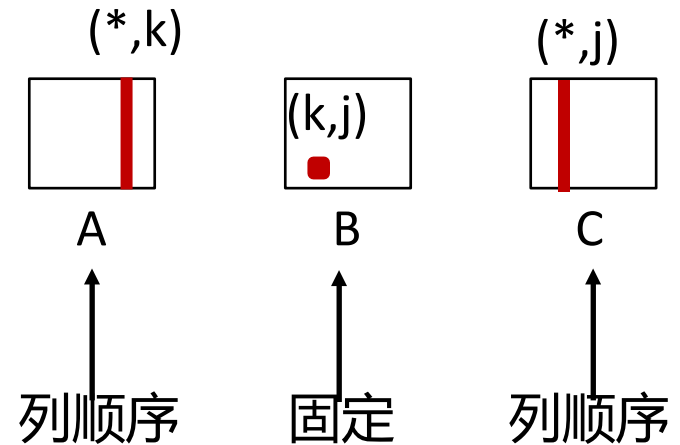
矩阵乘法 (jki)

```

/* 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

```

内层循环:



每次内层循环迭代的不命中数:

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

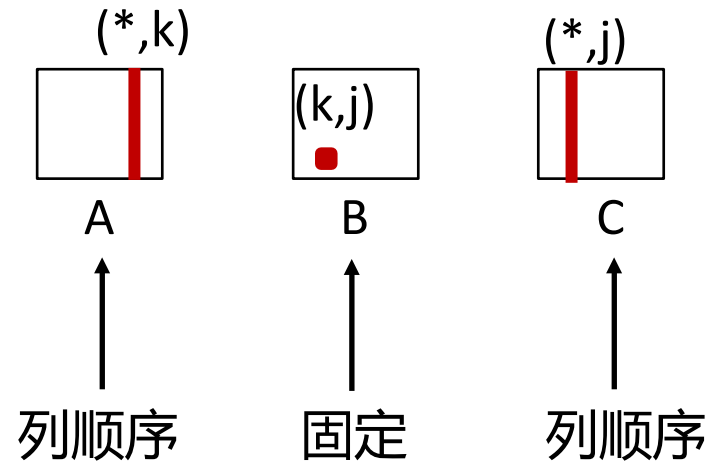
块大小 = 32B (4 doubles)

矩阵乘法 (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
  
```

内层循环:



每次内层循环迭代的不命中数:

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

块大小 = 32B (4 doubles)

矩阵乘法总结

```
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;
  }
}
```

```
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];
  }
}
```

```
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;
  }
}
```

ijk(& jik):

- 2 加载, 0 存储
- 每次迭代的不命中率=
1.25

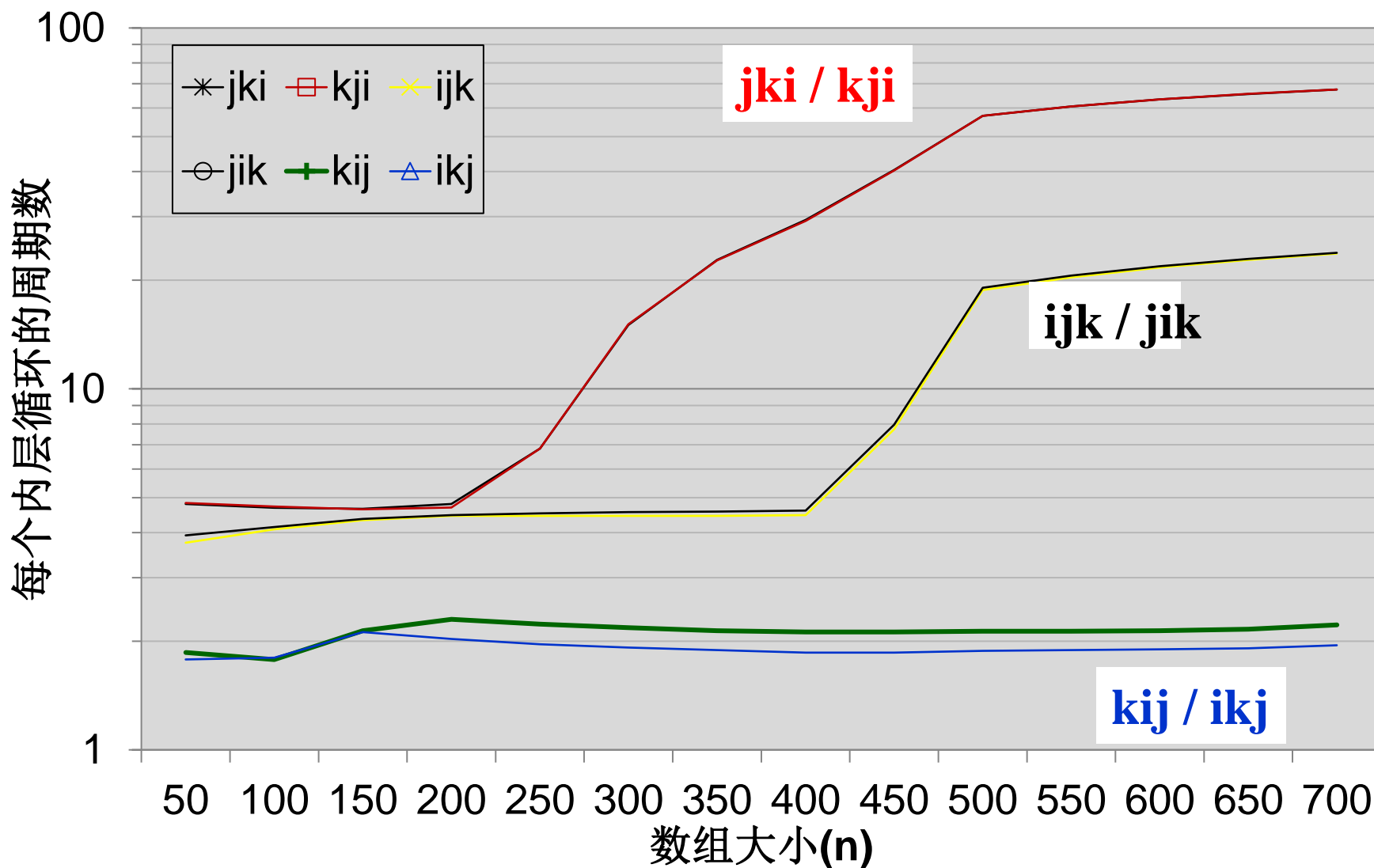
kij(& ikj):

- 2加载, 1存储
- 每次迭代的不命中率=
0.5

jki(& kji):

- 2加载, 1存储
- 每次迭代的不命中率=
2.0

Core i7矩阵乘法性能



主要内容

- 高速缓存的组织结构和运算
- 高速缓存对程序性能的影响
 - 存储器山
 - 重新排列以提升空间局部性
 - 使用块来提高时间局部性

例子:矩阵乘法

```
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;
```

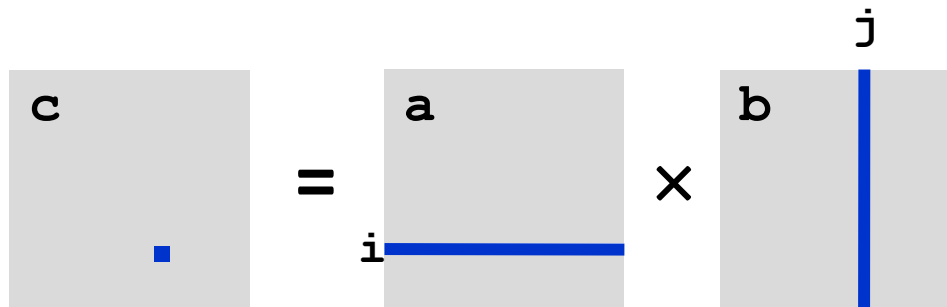
```
    for (i = 0; i < n; i++)
```

```
        for (j = 0; j < n; j++)
```

```
            for (k = 0; k < n; k++)
```

```
                c[i*n + j] += a[i*n + k] * b[k*n + j];
```

```
}
```



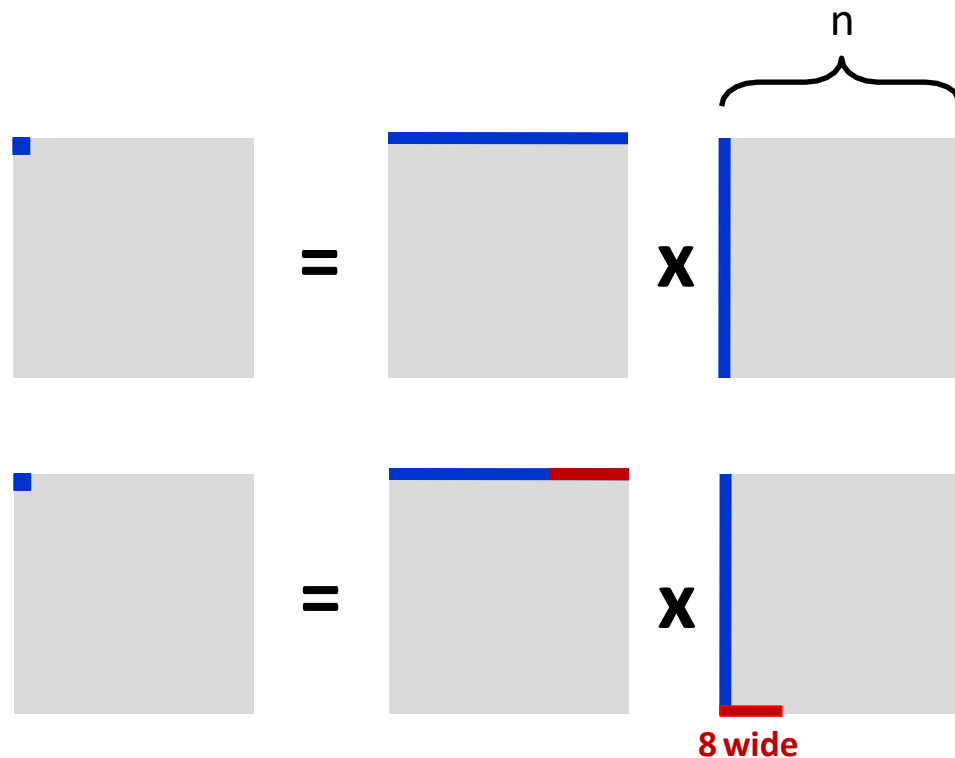
缓存不命中分析

■ 假设:

- 矩阵元素类型是doubles
- 缓存块 = 8 doubles
- 缓存大小 $C \ll n$

■ 第一次迭代:

- $n/8 + n = 9n/8$ 不命中
- 第一次迭代结束时, 缓存示意图



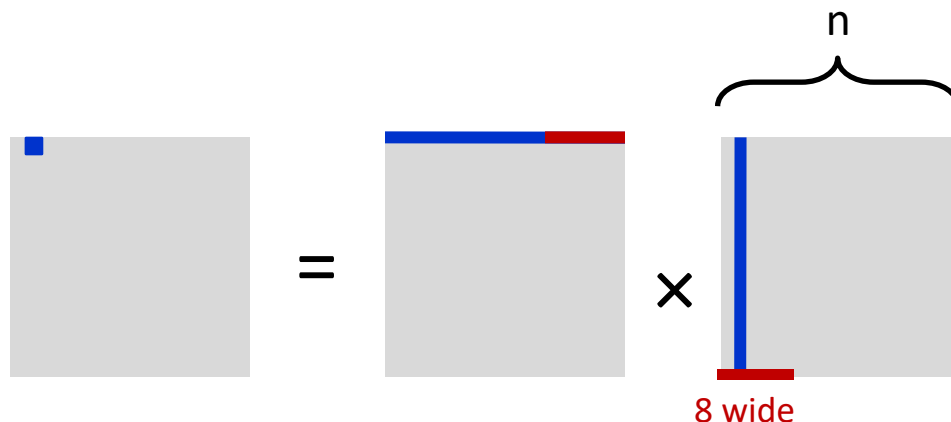
缓存不命中分析

■ 假设:

- 矩阵元素类型是doubles
- 缓存块 = 8 doubles
- 缓存大小 $C \ll n$

■ 第二次迭代:

$n/8 + n = 9n/8$ 不命中



■ 不命中总数:

$$n^2 \times 9n/8 = (9/8) n^3$$

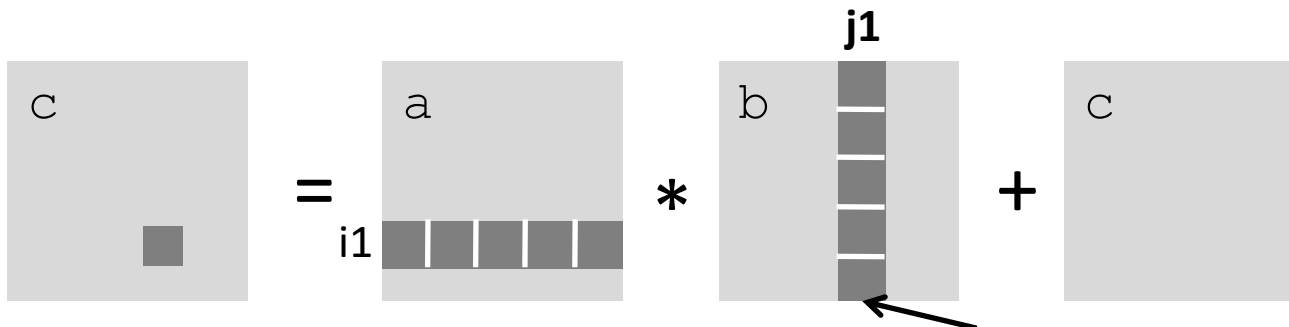
分块矩阵乘法

```

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 += T)
        for (j = 0; j < n; j += T)
            for (k = 0; k < n; k += T)
                /* T×T mini matrix multiplications */
                for (i1 = i; i1 < i+T; i1++)
                    for (j1 = j; j1 < j+T; j1++)
                        for (k1 = k; k1 < k+T; k1++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}

```

matmult/bmm.c



矩阵分块尺寸: $T \times T$ 个数

缓存不命中分析

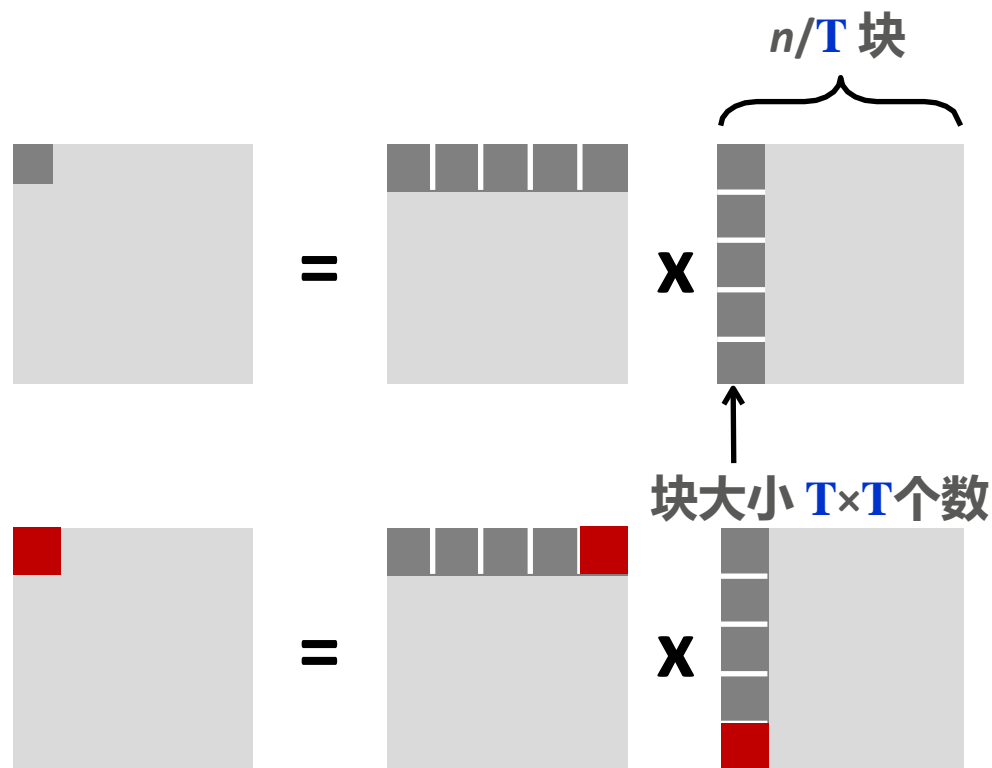
■ 假设:

- 缓存块: 8 doubles
- 缓存大小 $C \ll n$
- 缓存可放入三个矩阵块 \blacksquare : $3T^2 < C$

■ 第一次 (块) 迭代:

- 每块 $T^2/8$ 个不命中,
不命中的总数:
 $(n/T + n/T) \times T^2/8 = nT/4$
(忽略矩阵 c)

- 第一次迭代结束时, 缓存中的块 (红色) 示意图



缓存不命中分析

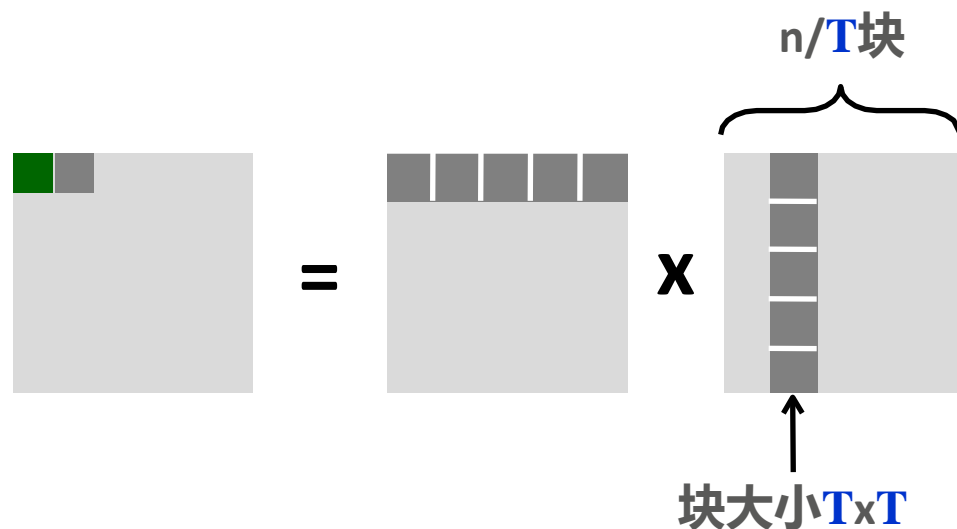
■ 假设:

- 缓存块: 8 doubles
- 缓存大小 $C \ll n$
- 缓存可放入三个矩阵块 \blacksquare : $3T^2 < C$

■ 第二次 (块) 迭代:

- 同第一次迭代相同,
不命中的总数:

$$(n/T + n/T) \times T^2/8 = nT/4$$



■ 总不命中率:

- $(n/T)^2 \times nT/4 = n^3/(4T)$

分块总结

- 不分块: $(9/8) n^3$
- 分 块: $1/(4\mathbf{T}) n^3$
- 块大小 \mathbf{T} 使用最大的可能值 \mathbf{T}_{\max} , 限制:
$$3\mathbf{T}_{\max}^2 < C$$
- 巨大差距的原因:
 - 矩阵乘法有天生的时间局部性:
 - 输入数据: $3n^2$, 计算 $2n^3$
 - 每个数据组元素计算的时间复杂度（时间开销）: $O(n)$
 - 必须恰当地编写程序

高速缓存总结

- 高速缓存对程序性能有显著影响
- 可以在程序中利用这一点！
 - 聚焦内层循环：大部分计算和内存访问都发生在这里。
 - 最大化空间局部性：以1为步长，按数据在内存中的存储顺序读取。
 - 最大化程序时间局部性：从内存中读入一个数据对象后，尽可能频繁/多地使用它。