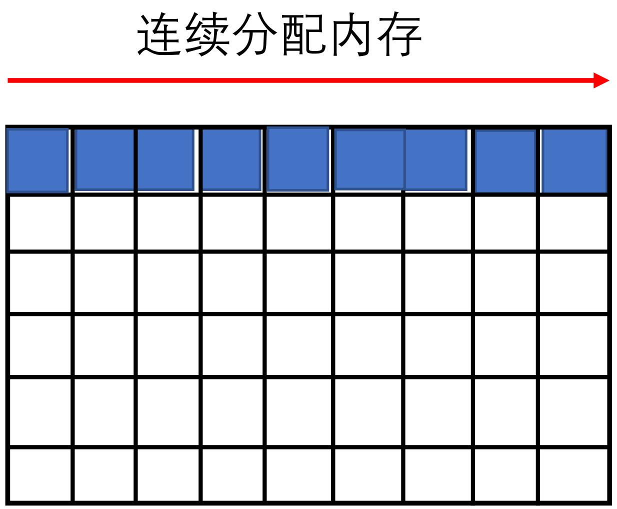
高速缓存友好的代码

什么是cache 友好的代码?

举例:C语言的数组元素是以行为主顺序分配的,同一行的元素被分配在连续的内存空间中



- 对同一行的元素依次访问:
  - for (i = 0; i < N; i++)

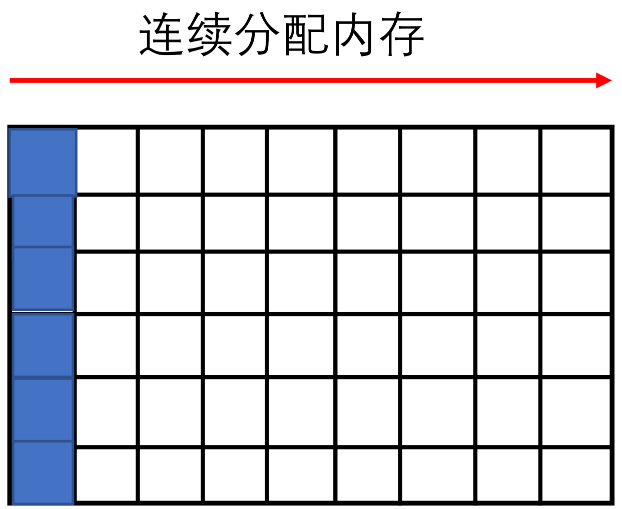
    sum += a[0][i];

- 访问连续的元素;利用到了空间局部性(cache 友好的代码)
- cache 失效率=

  数组元素的大小/缓存块的大小

什么是cache 友好的代码？

举例：C语言的数组元素是以行为主顺序分配的，同一行的元素被分配在连续的内存空间中

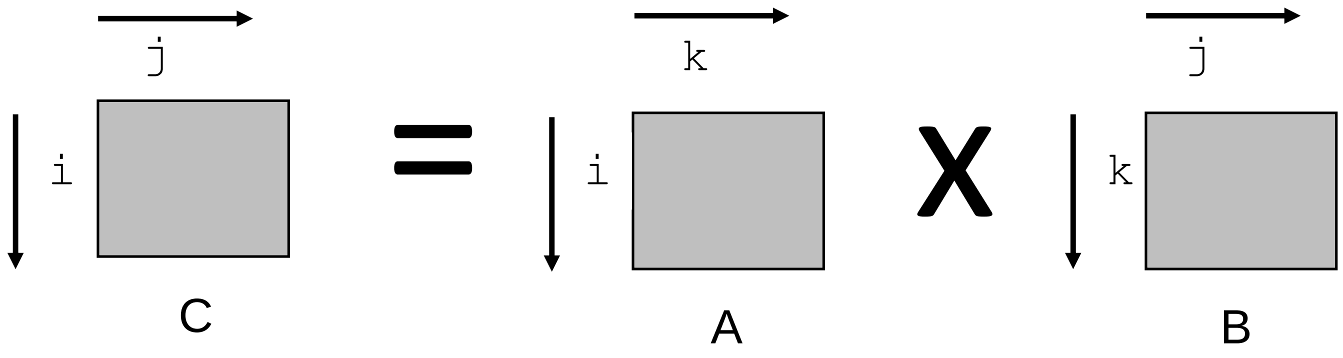


- 对同一列的元素依次访问：
  - ```
    for (i = 0; i < n; i++)
      sum += a[i][0];
    ```
- 访问有一定间隔的元素，没有利用到空间局部性
- cache 失效率 = 1 (即： 100%)
    ```

高速缓存友好的代码的关键思路

- 让常用部分（common case）执行得更快
  - 关注核心函数的内层循环
- 使缓存失效率次数最小
  - 重复引用同一变量(时间局部性)
  - **优先访问邻近的变量** (空间局部性)

举例： 矩阵相乘



- n x n 的矩阵相乘
- 总共的计算次数为 $O(n^3)$

```
/* 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
被保存在寄存器中

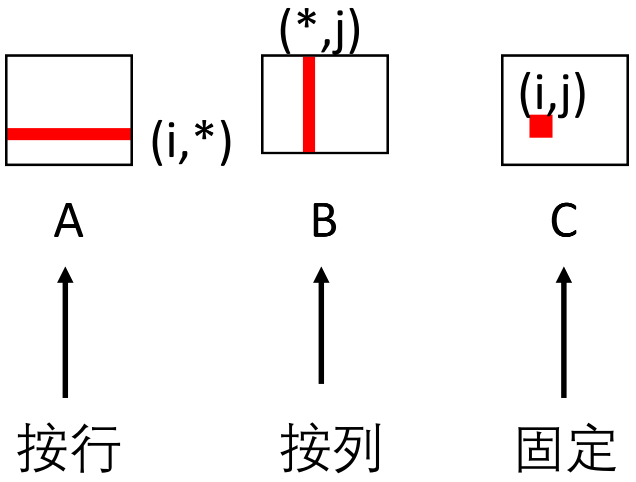
矩阵相乘时cache的失效率 （miss rate）

- 假设:
  - 矩阵的元素类型为double (元素大小为8 bytes)
  - 缓存块的大小 = 64 B (可以存放下8个double类型的元素)
  - 矩阵的维度 (n) 非常大
  - 缓存的大小不足以存下多行元素
- 分析：最内层循环的cache失效率

矩阵相乘(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;
  }
}
```

内层循环:



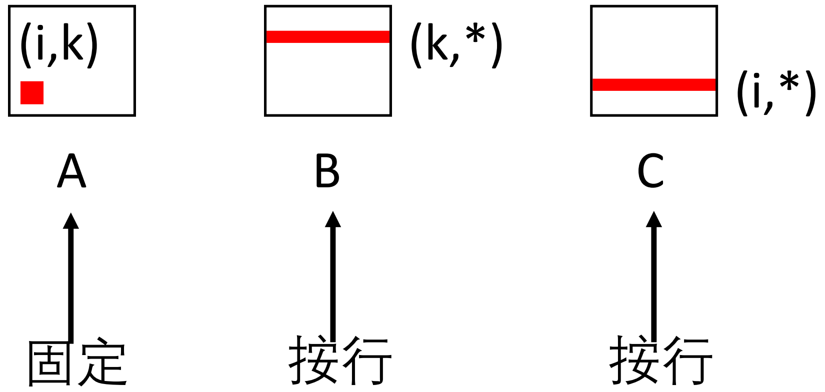
<u>每一个最内层循环的cache失效次数:</u>

<u>A</u>	<u>B</u>	<u>C</u>
0.125	1.0	0.0

缓存块的大小= 64B (可以存放下8个double类型的元素)

矩阵相乘 (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];
  }
}
```

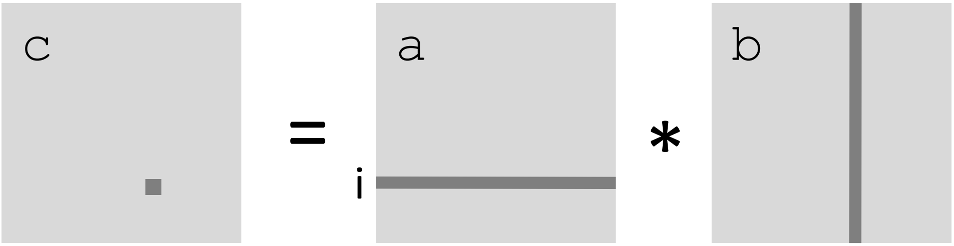


<u>每一个最内层循环的cache失效次数:</u>

A	B	C
0.0	0.125	0.125

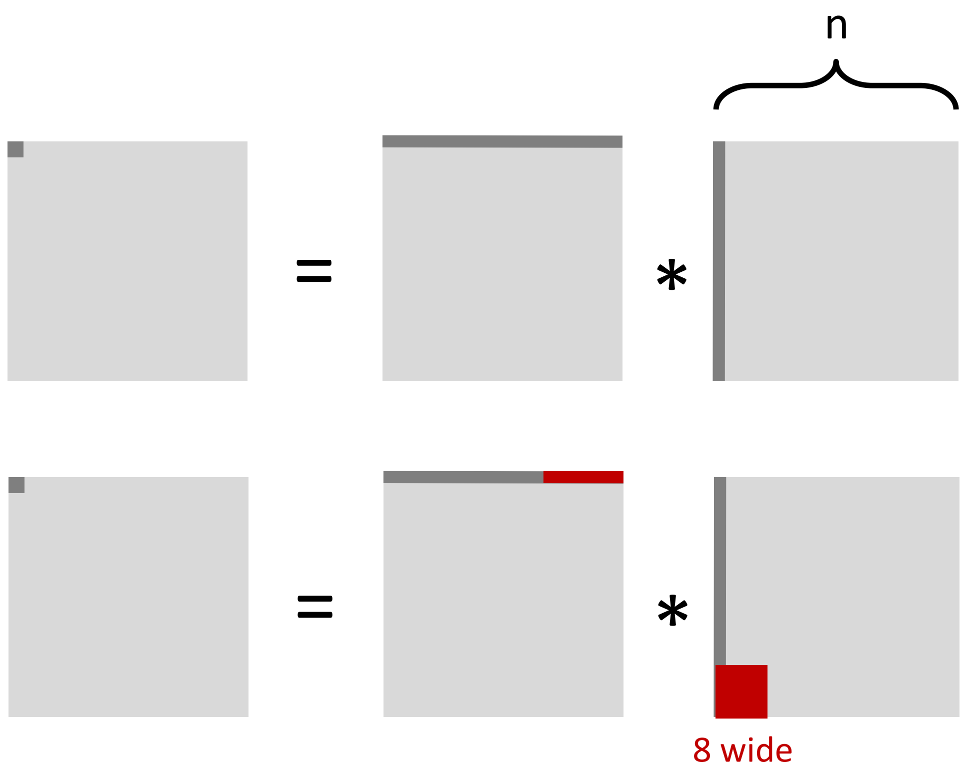
举例：矩阵相乘

```
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];
}
```



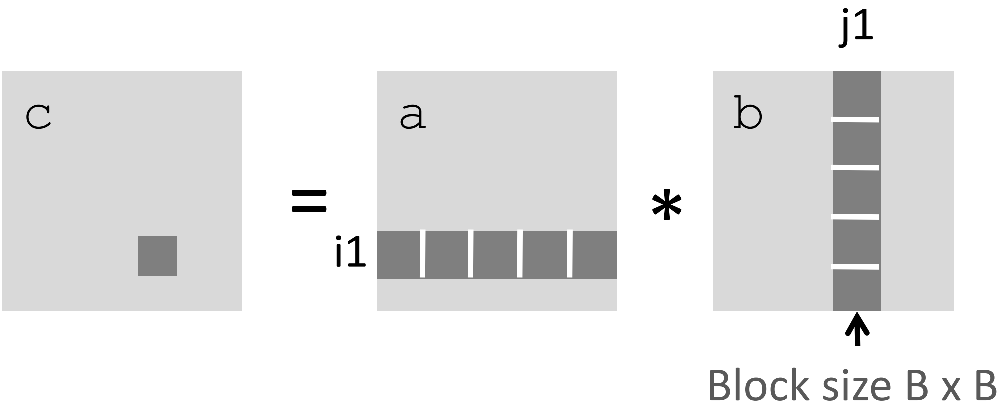
缓存失效分析

- 假设:
  - 矩阵元素的数据类型是double
  - 缓存的块大小 = 8 doubles
  - 缓存的大小 C << n (远远小于 n)
- 最内层循环:
  - n/8 + n = 9n/8 次失效
- 总共的miss 数:
  - $9n/8 * n^2 = (9/8) * n^3$



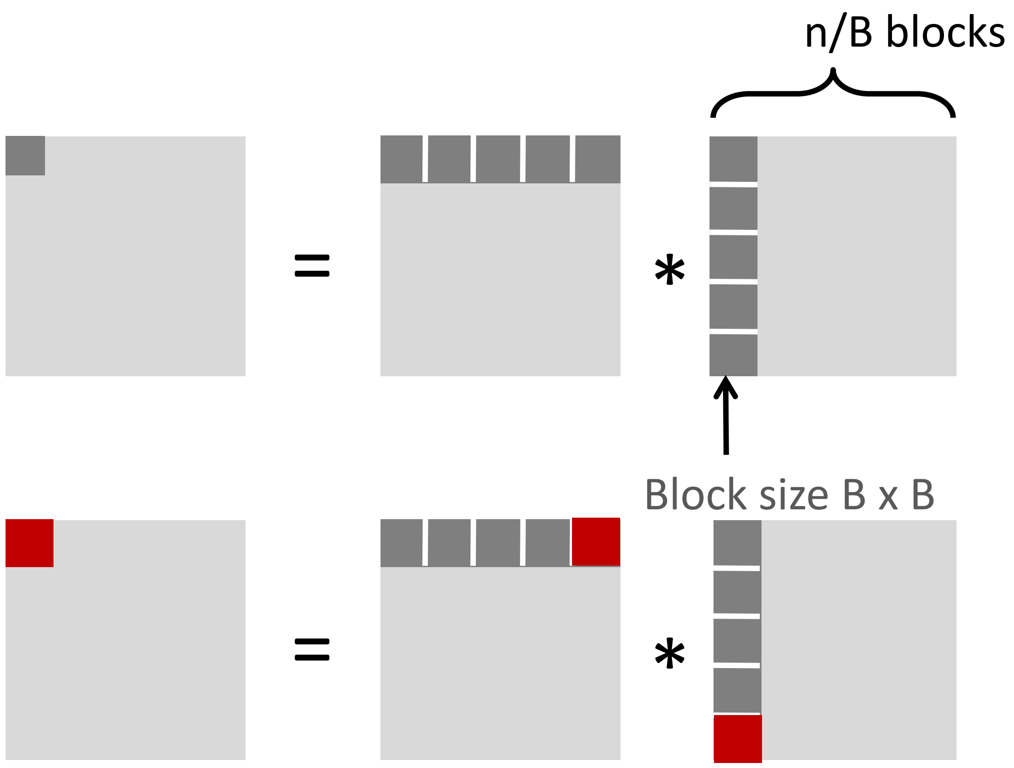
```c
c = (double *) calloc(sizeof(double), n*n);

/* 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+=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];
}
```

矩阵分块相乘



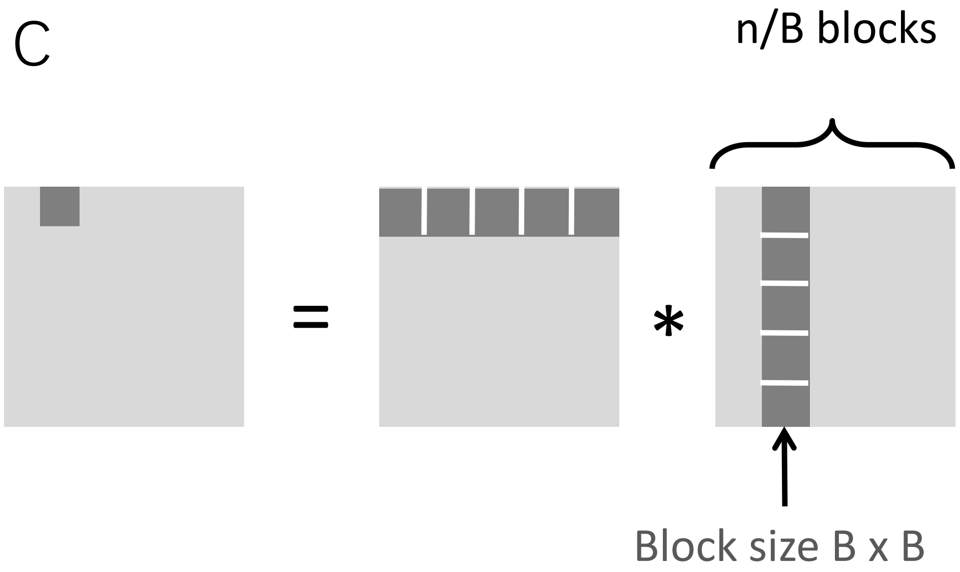
缓存失效分析

- 假设:
  - Cache数据块的大小= 8 doubles
  - 缓存的大小 C << n (远小于n)
  - 三个分块■可以同时放入缓存:$3B^2 < C$
- 一次（分块）迭代:
  - 每个块产生 $B^2/8$ 次错失
  - 失效次数：$2n/B * B^2/8 = nB/4$



缓存错失分析

- 假设:
  - 缓存块的大小= 8 doubles
  - 缓存的大小 C << n (远小于n)
  - 三个分块 ■ 可以同时放入缓存:$3B^2 < C$
- 每次（分块）迭代:
  - 失效数：$2n/B * B^2/8 = nB/4$
- 总共的错失数:
  - $nB/4 * (n/B)^2 = n^3/(4B)$



比较两种矩阵相乘算法

- 不分块的cache miss次数: $(9/8) * n^3$
- 分块的cache miss次数: $1/(4B) * n^3$
- 我们希望尽可能大的分块大小 B, 但限制 $3B^2 < C$
- 差异巨大的原因:
  - 矩阵乘法具有固定的时间局部性:
    - 输入数据: $3n^2$, 计算量: $2n^3$
    - 每个数组元素使用O(n)次!

小结

- 矩阵相乘的两种实现方式
- 程序员应理解高速缓存的原理，书写缓存友好的代码

谢谢！