计算机系统基础

Lab4 Cache Lab

姓名：傅文杰

学号:22300240028

2023 年 12 月 2 日

目录

1 实验 A：编写高速缓存

1.1 实验要求

在 csim.c 下编写一个高速缓存模拟器来对内存读写操作进行正确的反馈。
这个模拟器有 6 个参数：

```
Usage: ./csim-ref [-hv] -s <s> -E <E> -b <b> -t <tracefile>
```

1.1.1 参数说明

- -h: Optional help flag that prints usage info
- -v: Optional verbose flag that displays trace info
- -s <s>: Number of set index bits (S = 2^s is the number of sets)
- -E <E>: Associativity (number of lines per set)
- -b <b>: Number of block bits (B = 2^b is the block size)
- -t <tracefile>: Name of the valgrind trace to replay

对于-v 参数，我们需要初始化一个标记。

```
int verbose = 0;
```

对于文件当中对于缓存的操作，也就是-t 参数读入的文件，我们需要初始化一个数组保存其中的内容。

```
char t[1000];
```

1.1.2 trace 格式说明

输入的 trace 的格式为：空格 + operation address + , + size
operation 有 4 种：

- I 加载指令
- L 加载数据
- S 存储数据
- M 修改数据

1.1.3 模拟器反馈说明

模拟器不需要考虑加载指令，而 M 指令就相当于先进行 L 再进行 S。
模拟器要做出的反馈有 3 种：

- hit：命中，表示要操作的数据在对应组的其中一行
- miss：不命中，表示要操作的数据不在对应组的任何一行
- eviction：驱赶，表示要操作的数据的对应组已满，进行了替换操作

因此我们要初始化这些信息

```
1 int hit_count = 0, miss_count = 0, eviction_count = 0;
```

最后打印出来

```
1 printSummary(hit_count, miss_count, eviction_count);
```

1.2 实验准备

1.2.1 头文件

由于我们不是 shark machine，所以为了调用 getopt 函数我们需要引入头文件 unistd.h 和 getopt.h

1.2.2 替换策略

缓存不命中时，CPU 必须从内存中取出包含这个字的块，并替换一行。书上有两个策略：

- LFU: 替换在过去某个时间窗口内引用次数最少的一行
- LRU: 替换最后一次访问时间最久远的一行

Hints 提示我们使用 LRU counter 计数器，即时间戳。除此以外，还可以使用链表，哈希表加双向链表等复杂数据结构来实现。这里选择较为简单的时间戳方法。

1.2.3 Cache 的结构

对于整个缓存，我们需要定义两个结构体，cache_ 和 cache_line。

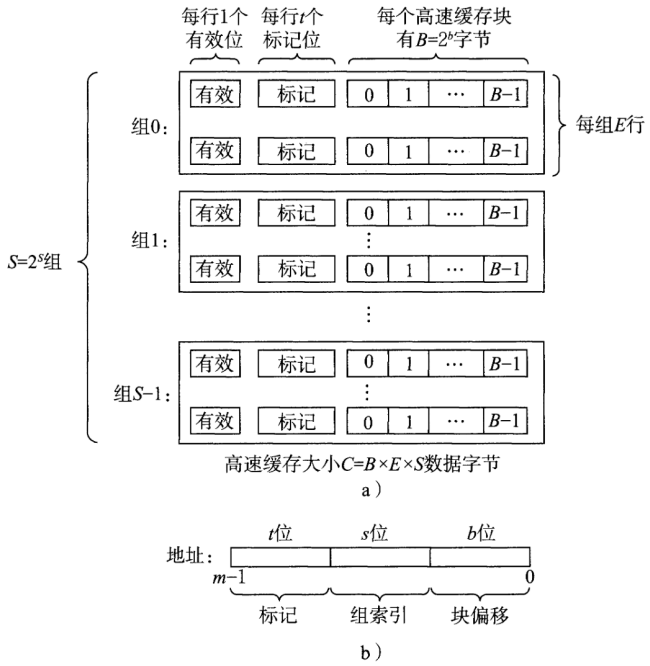

图 6-25 高速缓存(S，E，B，m)的通用组织。a)高速缓存是一个高速缓存组的数组。每个组包含一个或多个行，每个行包含一个有效位，一些标记位，以及一个数据块；b)高速缓存的结构将 m 个地址位划分成了 t 个标记位、s 个组索引位和 b 个块偏移位

其中 cache_ 对应的是整个缓存 (S, E, B, m) 的通用组织，由于 $M = 2^m$ 代表总的地址数目，可以由其他变量计算出，因此可以省略。

```
1 typedef struct cache_
```

```
{
    int S;
    int E;
    int B;
    Cache_line **line;
} Cache;
Cache *cache = NULL;
```

cache_line 对应的是缓存中的行的信息，除了有效位和标记外，还需要 LRU 方法的时间戳。

```
typedef struct cache_line
{
    int valid;
    int tag;
    int time_tamp;
} Cache_line;
```

1.3 主函数

由 1.1.1 可知我们需要接受 6 个参数，可以参考如下的 example。

我们的局部变量为：opt(设置成 char 类型), $s(S = 2^s$ 是组的个数 $), E($ 是行数 $), b(B = 2^b$ 是

Part (a) : getopt Example

```
int main(int argc, char** argv){
    int opt,x,y;
    /* looping over arguments */
    while(-1 != (opt = getopt(argc, argv, "x:y:"))){
        /* determine which argument it's processing */
        switch(opt) {
            case 'x':
                x = atoi(optarg);
                break;
            case 'y':
                y = atoi(optarg);
                break;
            default:
                printf("wrong argument\n");
                break;
        }
    }
}
```

- **Suppose the program executable was called "foo". Then we would call "./foo -x 1 –y 3" to pass the value 1 to variable x and 3 to y.**

每个缓冲块的字节数)。
我们需要定义的函数为：

1. 打印帮助：print_help 函数
2. Cache 操作：Init_Cache 初始化缓存，get_trace 读取 trace 中的操作，free_Cache 释放缓存

代码如下：

```
int main(int argc, char *argv[])
{
    char opt;
    int s, E, b;
    while (-1 != (opt = getopt(argc, argv, "hvs:E:b:t:")))
    {
        switch (opt)
        {
        case 'h':
            print_help();
            exit(0);
        case 'v':
            verbose = 1;
            break;
        case 's':
            s = atoi(optarg);
            break;
        case 'E':
            E = atoi(optarg);
            break;
        case 'b':
            b = atoi(optarg);
            break;
        case 't':
```

```
25              strcpy(t, optarg);
26              break;
27          default:
28              print_help();
29              exit(-1);
30          }
31      }
32      Init_Cache(s, E, b);
33      get_trace(s, E, b);
34      free_Cache();
35      printSummary(hit_count, miss_count, eviction_count);
36      return 0;
37  }
```

1.4 Cache 相关函数

1.4.1 初始化

初始化 $S = 2^s$ 组数，E 每组 E 行，$B = 2^b$ 每个告诉缓存块的字节数;
为 Cache 分配空间，并产生一个指向它的指针 (Cache* cache);
为 cache 的二维数组分配空间 (S 组一维 cache_line)，每个一维 cache_line 数组有 E 个 cache_line;
每个 cache_ling 的有效位、标记、时间戳分别初始化为 0, -1, 0。代码如下：

```
1 void Init_Cache(int s, int E, int b)
2 {
3     int S = 1 << s;
4     int B = 1 << b;
5     cache = (Cache *)malloc(sizeof(Cache));
6     cache->S = S;
7     cache->E = E;
8     cache->B = B;
```

```
    cache->line = (Cache_line **)malloc(sizeof(Cache_line *) * S);
    for (int i = 0; i < S; i++)
    {
        cache->line[i] = (Cache_line *)malloc(sizeof(Cache_line) * E);
        for (int j = 0; j < E; j++)
        {
            cache->line[i][j].valid = 0;
            cache->line[i][j].tag = -1;
            cache->line[i][j].time_tamp = 0;
        }
    }
}
```

1.4.2 释放空间

每次 malloc 分配空间之后一定要释放空间。我们在初始化的时候为缓存、缓存的每个组、每个组的每行分配了空间，需要对应地释放掉。

代码如下：

```
void free_Cache()
{
    int S = cache->S;
    for (int i = 0; i < S; i++)
    {
        free(cache->line[i]);
    }
    free(cache->line);
    free(cache);
}
```

1.4.3 缓存操作

由 1.1.3，对于缓存的操作，我们只需要实现读操作和写操作就能完成所有的操作。但其实读操作就相当于“不写”的写操作，我们可以把它们统一为更新操作。除此之外，我们还需要想办法得到输入的标记位和组序号。因此代码大致架构如下：

```
void get_trace(int s, int E, int b)
{
    FILE *pFile;
    pFile = fopen(t, "r");
    if (pFile == NULL)
    {
        exit(-1);
    }
    char identifier;
    unsigned address;
    int size;
    // Reading lines like " M 20,1" or "L 19,3"
    while (fscanf(pFile, " %c %x,%d", &identifier, &address, &
       size) > 0)
    {
        int op_tag = ...;
        int op_s = ...;
        switch (identifier)
        {
        case 'M':
            update_info(op_tag, op_s);
            update_info(op_tag, op_s);
            break;
        case 'L':
            update_info(op_tag, op_s);
            break;
        case 'S':
```

```
27                update_info(op_tag, op_s);
28                break;
29            }
30        }
31        fclose(pFile);
32    }
```

首先，由第 4 页上地址的图示，我们可以通过位运算得到标记和组索引。
将地址右移 (s+b) 位就得到标记 (C 语言算术右移)。
将地址右移 b 位，再和 $0x\underbrace{0\cdots0}_{(64-s)bits}\underbrace{1\cdots1}_{(s)bits}$ 相与即可。
代码如下：

```
1 int op_tag = address >> (s + b);
2 int op_s = (address >> b) & ((unsigned)(-1) >> (64 - s));
```

更新操作：如果有效位设置了并且 tag 符合，那么命中，否则不命中。
不命中时：如果有空行，那么缓存需要从内存中取出这个块并替换空行，否则，即 cache_line

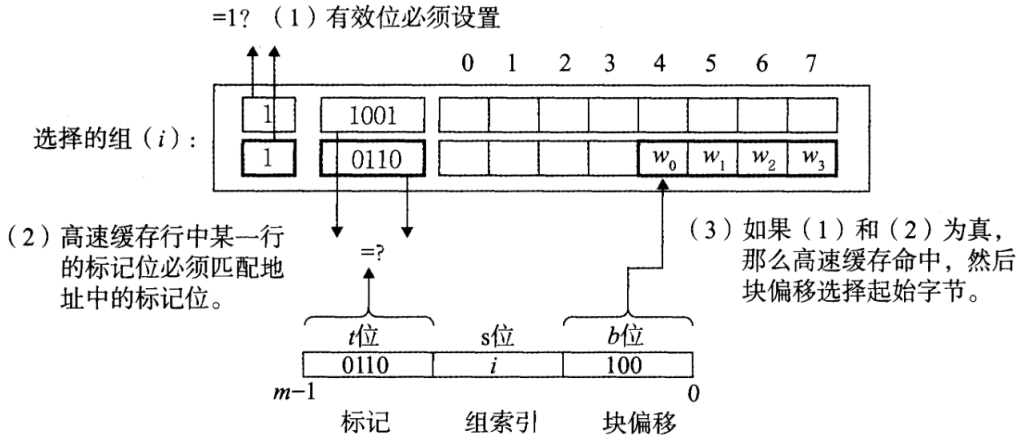

图 6-34 组相联高速缓存中的行匹配和字选择

满了，我们需要找到时间戳最大的行，并替换。
最后要记得更新：如果命中，需要更新时间戳，如果没有命中，需要把有效位设置为 1，把标记位设置为从内存中读的 tag，并更新时间戳。
时间戳的更新方法为：对于操作的缓存行，时间戳更新为 0，对于没有操作的缓存行，时间戳

++。注意更新时间戳的前提是有效位为 1。

代码如下：

```
void update(int i, int op_s, int op_tag)
{
    cache->line[op_s][i].valid=1;
    cache->line[op_s][i].tag = op_tag;
    for(int k = 0; k < cache->E; k++)
        if(cache->line[op_s][k].valid==1)
            cache->line[op_s][k].time_tamp++;
    cache->line[op_s][i].time_tamp = 0;
}
int find_LRU(int op_s)
{
    int max_index = 0;
    int max_stamp = 0;
    for(int i = 0; i < cache->E; i++){
        if(cache->line[op_s][i].time_tamp > max_stamp){
            max_stamp = cache->line[op_s][i].time_tamp;
            max_index = i;
        }
    }
    return max_index;
}
int is_full(int op_s)
{
    for (int i = 0; i < cache->E; i++)
    {
        if (cache->line[op_s][i].valid == 0)
            return i;
    }
    return -1;
```

```
30 }
31 int get_index(int op_s, int op_tag)
32 {
33     for (int i = 0; i < cache->E; i++)
34     {
35         if (cache->line[op_s][i].valid && cache->line[op_s][i].
               tag == op_tag)
36             return i;
37     }
38     return -1;
39 }
40 void update_info(int op_tag, int op_s)
41 {
42     int index = get_index(op_s, op_tag);
43     if (index == -1)
44     {
45         miss_count++;
46         if (verbose)
47             printf("miss ");
48         int i = is_full(op_s);
49         if(i==-1){
50             eviction_count++;
51             if(verbose) printf("eviction");
52             i = find_LRU(op_s);
53         }
54         update(i,op_s,op_tag);
55     }
56     else{
57         hit_count++;
58         if(verbose)
59             printf("hit");
```

```
60          update(index,op_s,op_tag);
61      }
62  }
```

2 实验 B：优化矩阵转置

2.1 实验要求

在 trans.c 中写转置函数使得 cache miss 尽可能少。

- 32×32 的矩阵需要使 miss 次数小于 300
- 64×64 的矩阵需要使 miss 次数小于 1300
- 61×67 的矩阵需要使 miss 次数小于 2000

这里的缓存参数为 $(s = 5, E = 1, b = 5)$，即 $S = 2^s = 2^5 = 32$ 个 CacheLine、每个 CacheLine 有 $E = 1$ 组、每组可以存储 $B = 2^b = 2^5 = 32$ 个 Byte$(8 \times sizeof(int))$。
如果我们用直接的暴力矩阵转置：

```
void trans_submit(int M, int N, int A[N][M], int B[M][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            int tmp = A[i][j];
            B[j][i] = tmp;
        }
    }
}
```

在终端中运行测试：

```
./test-trans -M 32 -N 32
```

得到结果：

```
Function 1 (2 total)
Step 1: Validating and generating memory traces
```

```
Step 2: Evaluating performance (s=5, E=1, b=5)
func 1 (Simple row-wise scan transpose): hits:869, misses:1184,
    evictions:1152
```

显然不符合要求，所以我们需要对循环中具体的计算过程进行优化。
我们不妨先对 miss 次数进行大致的分析：

1. 读入 $A[0][0]$，miss，此时 $A[0][0], \cdots, A[0][7]$ 被载入缓存
   写入 $B[0][0]$，miss，此时 $B[0][0], \cdots, B[0][7]$ 被载入缓存
2. 读入 $A[0][1]$，hit
   写入 $B[1][0]$，miss，此时 $B[1][0], \cdots, B[1][7]$ 被载入缓存
3. $\cdots$
4. 读入 $A[0][7]$，hit
   写入 $B[7][0]$，miss，此时 $B[7][0], \cdots, B[7][7]$ 被载入缓存
   至此，A 的缓存只有第一行有效，而 B 的缓存全部有效。
5. 读入 $A[0][8]$,miss,eviction,replacement,此时 $A[0][0], \cdots, A[0][7]$ 被替换为 $A[0][8], \cdots, A[0][15]$
   写入 $B[8][0]$,miss,eviction,replacement,此时 $B[0][0], \cdots, B[0][7]$ 被替换为 $B[8][0], \cdots, B[8][7]$
6. 读入 $A[0][9]$，hit
   写入 $B[9][0]$,miss,eviction,replacement,此时 $B[1][0], \cdots, B[1][7]$ 被替换为 $B[9][0], \cdots, B[9][7]$
7. $\cdots$
8. 读入 $A[0][15]$，hit
   写入 $B[15][0]$,miss,eviction,replacement,此时 $B[7][0], \cdots, B[7][7]$ 被替换为 $B[15][0], \cdots, B[15][7]$
   至此，A 的缓存仍然只有第一行有效，而 B 的缓存全部有效，并且被全部更替过了一遍。
   如此往复，直到 A 的第一行读完后，A 有 $32/8 = 4$ 次 miss，而 B 有 32 次 miss。
   可以粗略地将上述过程 $\times 32$ 来进行估计，miss $\approx 36 \times 32 = 1152$。但是程序跑出来 miss $= 1184$，可见实际中的 miss 会比理论上多。

2.2 32×32

分块策略是一种常用的增加 cache hit 的策略。
具体考虑为：

- 当我们读完 $A[0][0], \cdots, A[0][7]$ 时，B 的缓存已经满了：
  $B[0][0], B[0][1], \cdots, B[0][7]$
  $B[1][0], B[1][1], \cdots, B[1][7]$
  $\cdots$
  $B[7][0], B[7][1], \cdots, B[7][7]$
  但这其中被有效利用的只有第一列。
  那么第二列到第七列对应 A 中的什么呢？
  $A[1][0], \cdots, A[1][7]$
  $A[2][0], \cdots, A[2][7]$
  $\cdots$
  $A[7][0], \cdots, A[7][7]$
  这些正好是 32×32 方阵的第一个 8×8 的小方阵，也称之为块 Block。
  如果我们一个块一个块地转置，那么每个块的 miss$= 8 + 8 = 16$，总共的 miss 大致可估为 $16 \times 32 = 256$

- 代码如下：

```
void transpose_32x32(int M, int N, int A[N][M], int B[M][N])
{
    for (int i = 0; i < 32; i += 8)
        for (int j = 0; j < 32; j += 8)
            for (int k = i; k < i + 8; k++)
                for(int s = j; s < j + 8; s ++)
                    B[j+s][i+k] = A[i+k][j+s];
}
```

- 在终端中运行测试：

```
Function 0 (2 total)
Step 1: Validating and generating memory traces
```

```
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:1709, misses:344,
    evictions:312
```

miss 为 344，与估计有较大误差，这是为什么呢？

- **The first row of Matrix A evicts the first row of Matrix B**
- Matrix A and B are stored in memory at addresses such that both the first elements align to the same place in cache!
- 这对于对角线的元素尤其致命。例如，当我们读 $A[4][4]$ 时，已经把第 4 行存进去了，但是，当我们写 $B[4][4]$ 时，会先驱逐 $A[4][x]$ 这行，并将 $B[4][x]$ 这行存进去，接下来读 $A[4][5]$ 时，又会驱逐 $B[4][x]$ 这行。因此会多造成两次 miss。
- 因此我们理解了原代码中给出的暴力转置使用临时变量 tmp 的原因，以及 writeup 中这条限制"You are allowed to define at most 12 local variables of type int per transpose function." 的由来。
- 所以我们的修改方法为：将最后一层循坏拆开，先把 A 的一行中的元素一经读入就存到寄存器中，再将这些值分别赋给对应的 B 中的位置，避免了 A 的一行中从左往右读值时由于和 B 地址冲突多造成的 miss。这样能够很好地处理非对角线的元素，但是对于对角线的元素还是会出现一些 miss，存在进一步优化的空间。
- 代码如下：

```
void transpose_32x32(int M, int N, int A[N][M], int B[M][N])
{
    for (int i = 0; i < 32; i += 8)
        for (int j = 0; j < 32; j += 8)
            for (int k = i; k < i + 8; k++)
            {
                int a_0 = A[k][j];
                int a_1 = A[k][j + 1];
                int a_2 = A[k][j + 2];
```

```
                int a_3 = A[k][j + 3];
                int a_4 = A[k][j + 4];
                int a_5 = A[k][j + 5];
                int a_6 = A[k][j + 6];
                int a_7 = A[k][j + 7];
                B[j][k] = a_0;
                B[j + 1][k] = a_1;
                B[j + 2][k] = a_2;
                B[j + 3][k] = a_3;
                B[j + 4][k] = a_4;
                B[j + 5][k] = a_5;
                B[j + 6][k] = a_6;
                B[j + 7][k] = a_7;
            }
}
```

- 在终端中运行测试：

```
Function 0 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:1765, misses:288,
   evictions:256
```

2.3 64×64

由于这次矩阵是 64×64 的，所以 cache 最多只能存储 4 行矩阵，如果用 8×8 分块，那么在转置后 4 行时会与前四行冲突。所以这时会想到 4×4 分块，但是从理论上分析，就算达到了每块的最低 miss：$4 + 4 = 8$，总共的 miss 也要有大概 $16 \times 16 \times 8 = 2048$ 次，显然不符合要求。这样的话还是考虑 8×8 的情况，如果能结合 4×4 的思想，那么还可以进一步优化。具体步骤为（假设每个 8×8 的矩阵被划分成左上（1）、右上（2）、左下（3）、右下（4）四块）：

1. 将 A 的 1,2 转置成 1',2' 复制给 B 的 1,2(A 按行操作，B 按列操作)

```
for (int k = i; k < i + 4; k++)
{
    a_0 = A[k][j + 0];
    a_1 = A[k][j + 1];
    a_2 = A[k][j + 2];
    a_3 = A[k][j + 3];
    a_4 = A[k][j + 4];
    a_5 = A[k][j + 5];
    a_6 = A[k][j + 6];
    a_7 = A[k][j + 7];

    B[j + 0][k] = a_0;
    B[j + 1][k] = a_1;
    B[j + 2][k] = a_2;
    B[j + 3][k] = a_3;
    B[j + 0][k + 4] = a_4;
    B[j + 1][k + 4] = a_5;
    B[j + 2][k + 4] = a_6;
    B[j + 3][k + 4] = a_7;
}
```

2. 将 B 的 2 存下 (按行操作)

```
for (int k = j; k < j + 4; k++)
{
    a_0 = B[k][i + 4];
    a_1 = B[k][i + 5];
    a_2 = B[k][i + 6];
    a_3 = B[k][i + 7];
}
```

3. 将 A 的 3 转置成 3' 复制给 B 的 2(A 按列操作，B 按行操作)

```
for (int k = j; k < j + 4; k++)
{
    a_4 = A[i + 4][k];
    a_5 = A[i + 5][k];
    a_6 = A[i + 6][k];
    a_7 = A[i + 7][k];

    B[k][i + 4] = a_4;
    B[k][i + 5] = a_5;
    B[k][i + 6] = a_6;
    B[k][i + 7] = a_7;
}
```

4. 将存下的 B 的原来的 2 复制给 B 的 3(按行操作)

```
for (int k = j; k < j + 4; k++)
{
    B[k + 4][i + 0] = a_0;
    B[k + 4][i + 1] = a_1;
    B[k + 4][i + 2] = a_2;
    B[k + 4][i + 3] = a_3;
}
```

5. 将 A 的 4 转置成 4' 复制给 B 的 4(A 按行操作，B 按列操作)

```
for (int k = i + 4; k < i + 8; k++)
{
    a_4 = A[k][j + 4];
    a_5 = A[k][j + 5];
    a_6 = A[k][j + 6];
    a_7 = A[k][j + 7];

```

```
        B[j + 4][k] = a_4;
        B[j + 5][k] = a_5;
        B[j + 6][k] = a_6;
        B[j + 7][k] = a_7;
}
```

在终端中运行测试：

```
Function 0 (1 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:9017, misses:1228, evictions
    :1196
```

2.4 61×67

组索引没有那么紧密，可以不用考虑对角线的情况，试了几次可以用 16×16 分块解决。代码如下：

```
void transpose_61x67(int M, int N, int A[N][M], int B[M][N]){
    for (int i = 0; i < N; i += 16)
        for (int j = 0; j < M; j += 16)
            for (int k = i; k < i + 16 && k < N; k++)
                for (int s = j; s < j + 16 && s < M; s++)
                    B[s][k] = A[k][s];
}
```

在终端中运行测试：

```
Function 0 (1 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:6186, misses:1993, evictions
    :1961
```

3 参考资料

1. 关于矩阵和缓存的关系，尤其是 miss 和 eviction 是怎么发生的，
   https://zhuanlan.zhihu.com/p/410662053写的十分清晰：
   “A、B 两个数组在内存中是按行存放的——也就是说，矩阵中**同一行**的元素在内存中具有连续的地址，而相邻两行，第一行行尾元素和第二行行首元素地址相连。”————地址和缓存关系紧密，不同的地址存储方式会导致映射到的 set 不同。

2. 这次作业虽然达到了要求，但其实还有更多优化的空间，
   https://zhuanlan.zhihu.com/p/79058089中对于 32×32 和 64×64 的矩阵转置优化的特别好。

3. 在 64×64 矩阵转置中 8×8 分块中的 4 个 4×4 小块的转置策略；以及高速缓存的实现。参考了 https://zhuanlan.zhihu.com/p/484657229。