哈爾濱工業大學

实验报告

实 验（六）

题　　目　Cachelab

高速缓冲器模拟

专　　业　计算学部

学　　号　1190200708

班　　级　1903008

学　　生　熊峰

指 导 教 师　吴锐

实 验 地 点　G709

实 验 日 期　2021.05.24

计算机科学与技术学院

目 录

第 1 章 实验基本信息 ..... - 3 -

1.1 实验目的 ..... - 3 -
1.2 实验环境与工具 ..... - 3 -
1.2.1 硬件环境 ..... - 3 -
1.2.2 软件环境 ..... - 3 -
1.2.3 开发工具 ..... - 3 -
1.3 实验预习 ..... - 3 -

第 2 章 实验预习 ..... - 5 -

2.1 画出存储器层级结构，标识容量价格速度等指标变化（5 分） ..... - 5 -
2.2 用 CPUZ 等查看你的计算机 CACHE 各参数，写出各级 CACHE 的 C S E B S E B（5 分） ..... - 5 -
2.3 写出各类 CACHE 的读策略与写策略（5 分） ..... - 6 -
2.4 写出用 GPROF 进行性能分析的方法（5 分） ..... - 7 -
2.5 写出用 VALGRIND 进行性能分析的方法（（5 分） ..... - 8 -

第 3 章 CACHE 模拟与测试 ..... - 10 -

3.1 CACHE 模拟器设计 ..... - 10 -
3.2 矩阵转置设计 ..... - 12 -

第 4 章 总结 ..... - 38 -

4.1 请总结本次实验的收获 ..... - 38 -
4.2 请给出对本次实验内容的建议 ..... - 38 -

参考文献 ..... - 39 -

第 1 章 实验基本信息

1.1 实验目的

理解现代计算机系统存储器层级结构
掌握 Cache 的功能结构与访问控制策略
培养 Linux 下的性能测试方法与技巧
深入理解 Cache 组成结构对 C 程序性能的影响

1.2 实验环境与工具

1.2.1 硬件环境

X86-64 CPU；3.60GHz；16G RAM；256G SSD；1T SSD

1.2.2 软件环境

Win 10
Ubuntu 20.04.2 LTS
WSL2

1.2.3 开发工具

Visual Studio 2019；Vim；GCC；GDB；Code::Blocks; CLion 2020.3.1 x64;EDB

1.3 实验预习

上实验课前，必须认真预习实验指导书（PPT 或 PDF）

了解实验的目的、实验环境与软硬件工具、实验操作步骤，复习与实验有关的理论知识。

画出存储器的层级结构，标识其容量价格速度等指标变化

用 CPUZ 等查看你的计算机 Cache 各参数，写出 C S E B e s b 缓存大小 C、分组数量 S、关联度/组内行数 E、块大小 B，及对应的编码位数：组索引位数 s、e、

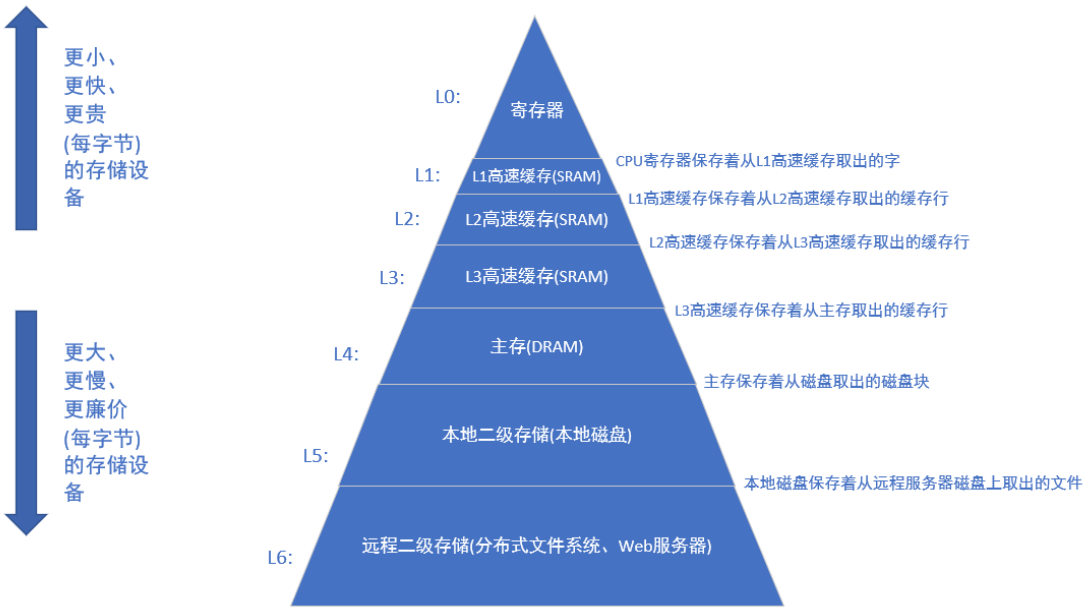
块内偏移位数 b。

写出各类 Cache 的读策略与写策略。

掌握 Valgrind 与 Gprof 的使用方法。

第 2 章 实验预习

2.1 画出存储器层级结构，标识容量价格速度等指标变化（5 分）



2.2 用 CPUZ 等查看你的计算机 Cache 各参数，写出各级 Cache 的 C S E B s e b（5 分）

	C	S	E	B	s	e	b
一级数据缓存：	32KB	64	8	64	6	3	6
一级指令缓存：	32KB	64	8	64	6	3	6
二级缓存：	256KB	1024	4	64	10	2	6
三级缓存：	12MB	16384	12	64	14	log12	6

2.3 写出各类 Cache 的读策略与写策略（5 分）

读策略：

定位组；

检查集合中的任何行是否有匹配的标记；

若存在匹配的标记，且行有效，则命中，将数据返回给 CPU；

若不命中，则从存储器层次结构更低的一层中取出数据，放入高速缓存中，再返回数据。

写策略：

写命中：

直写：立即写入存储器。

写回：推迟写入内存直到行要替换。

写不命中：

写分配：加载到缓存，更新这个缓存行。

非写分配：直接写到主存，不加载到缓存中。

2.4 写出用 gprof 进行性能分析的方法（5 分）

gprof 是 GNU profile 工具，可以运行于 linux、AIX、Sun 等操作系统进行 C、C++、Pascal、Fortran 程序的性能分析，用于程序的性能优化以及程序瓶颈问题的查找和解决。通过分析应用程序运行时产生的“flat profile”，可以得到每个函数的调用次数，每个函数消耗的处理器时间，也可以得到函数的“调用关系图”，包括函数调用的层次关系，每个函数调用花费了多少时间。使用步骤如下：

1. gcc -pg 编译程序
2. 运行程序，程序退出时生成 gmon.out
3. gprof ./prog gmon.out -b 查看输出

以 utf8len 程序为例：

```
baileys@xf1190200708:/mnt/c/Users/Alienware/Desktop/share/Lab2$ gcc -Wall -pg -o utf8len utf8len.c
baileys@xf1190200708:/mnt/c/Users/Alienware/Desktop/share/Lab2$ ./utf8len
字符个数位:2
baileys@xf1190200708:/mnt/c/Users/Alienware/Desktop/share/Lab2$ gprof utf8len gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00        1     0.00     0.00  utf8len

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
           function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
           the function in the gprof listing. If the index is
           in parenthesis it shows where it would appear in
           the gprof listing if it were to be printed.


Copyright (C) 2012-2020 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.
```

```
                     Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) no time propagated

index % time    self  children    called     name
                0.00    0.00       1/1           main [7]
[1]      0.0    0.00    0.00       1          utf8len [1]
-----------------------------------------------
```

2.5 写出用 Valgrind 进行性能分析的方法（（5 分）

Valgrind 包含下列工具：

1、memcheck：检查程序中的内存问题，如泄漏、越界、非法指针等。

2、callgrind：检测程序代码的运行时间和调用过程，以及分析程序性能。

3、cachegrind：分析 CPU 的 cache 命中率、丢失率，用于进行代码优化。

4、helgrind：用于检查多线程程序的竞态条件。

5、massif：堆栈分析器，指示程序中使用了多少堆内存等信息。

6、lackey：

7、nulgrind：

这几个工具的使用是通过命令：valgrand --tool=name 程序名来分别调用的，当不指定 tool 参数时默认是 --tool=memcheck。

以 hellolinux 程序为例：

```
baileys@xf1190200708:/mnt/c/Users/Alienware/Desktop/share/Lab1$ valgrind --tool=memcheck --leak-check=full ./hellolinux
==200== Memcheck, a memory error detector
==200== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==200== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==200== Command: ./hellolinux
==200==
Hello 1190200708熊峰==200==
==200== HEAP SUMMARY:
==200==     in use at exit: 0 bytes in 0 blocks
==200==   total heap usage: 2 allocs, 2 frees, 2,762 bytes allocated
==200==
==200== All heap blocks were freed -- no leaks are possible
==200==
==200== For lists of detected and suppressed errors, rerun with: -s
==200== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

第 3 章 Cache 模拟与测试

3.1 Cache 模拟器设计

提交 csim.c (源代码在附件中)

程序设计思想：主要分为三部分，void initCache(),void freeCache(),void accessData(mem_addr_t addr)。其中 accessData 函数为主要部分，包含 LRU 缓存算法。以下是对三个部分的简单分析：

(1) void initCache()

第一部分主要功能为初始化 cache，首先使用 malloc 函数，分配出 S 个指针类型的空间，作为每一组的指针，若分配失败，则输出提示信息，并结束运行。再对每一组的指针，使用 malloc 函数分配出 E 行的内存，并使每一组的指针指向 E 行。若分配内存失败，则用输出提示信息，并结束运行。分配完空间后，对 cache 中的每组中每一行赋值，将 valid 赋值为 0，tag 赋值为-1，lru 赋值为 1.

(2) void freeCache()

第二部分主要功能为释放分配空间，首先将每一组的每一行的空间释放，再将每组的空间释放。

(3) void accessData(mem_addr_t addr)

第三部分为主要部分，包含 LRU 缓存算法。

首先，通过移位指令，将计算出当前地址的标记位、组索引。

```
int t_addr = addr>>(s+b);
int s_addr = (addr>>b)&((1<<s)-1);
```

得到标记位、组索引后，可以开始进行匹配。

1> hit

利用已经得到的组索引，对当前地址所对应的组访问，并通过循环，检查该组中的每一行是否存在标记为相同的行，若检测到存在行的标记位与所得标记位相

同，则表明匹配。此时将 hit_count++，再将该组所有行的 lru++，并将该行的 lru 设为 1（由于 1 为 initCache 函数中初始化的大小）。

2> miss

若在第一部分未匹配成功，则将 miss_count++，并检查当前是否存在行尚未读取数据，通过当前地址得到的组索引，遍历该组，查看是否有尚未读取数据的行，若存在，则表明，Cache 尚未“热身”完毕，此时需要从内存中取出数据放入对应的行，因此此时只需要将该组的所有 valid 为 1 的行的 lru++，并将该行的 valid 设为 1，lru 设为 1，tag 设为 t_addr。

3> eviction

若在第一部分未检测到匹配的行，且第二部分为检测到尚未读取数据的行，则表明，此时需要替换当前存在时间最久的数据块。首先将 eviction_count++，再寻找该组中，所对应 lru 最小的行，将该组的所有行的 lru++，再将该行的 tag 设为 t_addr，lru 设为 1.

运行结果如下：

```
baileys@xf1190200708:/mnt/c/Users/Alienware/Desktop/share/Lab6/cachelab-handout$ make
# Generate a handin tar file each time you compile
tar -cvf baileys-handin.tar  csim.c trans.c
csim.c
trans.c
baileys@xf1190200708:/mnt/c/Users/Alienware/Desktop/share/Lab6/cachelab-handout$ ./test-csim
                        Your simulator     Reference simulator
Points (s,E,b)    Hits  Misses  Evicts    Hits  Misses  Evicts
     3 (1,1,1)       9       8       6       9       8       6  traces/yi2.trace
     3 (4,2,4)       4       5       2       4       5       2  traces/yi.trace
     3 (2,1,4)       2       3       1       2       3       1  traces/dave.trace
     3 (2,1,3)     167      71      67     167      71      67  traces/trans.trace
     3 (2,2,3)     201      37      29     201      37      29  traces/trans.trace
     3 (2,4,3)     212      26      10     212      26      10  traces/trans.trace
     3 (5,1,5)     231       7       0     231       7       0  traces/trans.trace
     6 (5,1,5)  265189   21775   21743  265189   21775   21743  traces/long.trace
    27

TEST_CSIM_RESULTS=27
```

测试用例 1 的输出截图（5 分）：

```
     3 (1,1,1)       9       8       6       9       8       6  traces/yi2.trace
```

测试用例 2 的输出截图（5 分）：

```
   3 (4,2,4)       4       5       2       4       5       2  traces/yi.trace
```

测试用例 3 的输出截图（5 分）：

```
3 (2,1,4)     2     3     1     2     3     1  traces/dave.trace
```

测试用例 4 的输出截图（5 分）：

```
3 (2,1,3)     167    71    67    167    71    67  traces/trans.trace
```

测试用例 5 的输出截图（5 分）：

```
3 (2,2,3)     201    37    29    201    37    29  traces/trans.trace
```

测试用例 6 的输出截图（5 分）：

```
3 (2,4,3)     212    26    10    212    26    10  traces/trans.trace
```

测试用例 7 的输出截图（5 分）：

```
3 (5,1,5)     231     7     0    231     7     0  traces/trans.trace
```

测试用例 8 的输出截图（10 分）：

```
6 (5,1,5)  265189  21775  21743  265189  21775  21743  traces/long.trace
```

注：每个用例的每一指标 5 分（最后一个用例 10）——与参考 csim-ref 模拟器输出指标相同则判为正确

3.2 矩阵转置设计

提交 trans.c

程序设计思想：

(由于 wsl2 拖动文件夹较为麻烦，第二部分改用 VMware 进行实验)

32*32：

由于可使用的 cache 的参数为 s=5、E=1、b=5，故 cache 的每个组可以存放一行，且每行中的块的大小为 2^5=32 字节，故每行可以存放八个 int 类型的变量。由于 s=5，故 S = 32，因此 cache 共可以存放 32*8=256 个 int 类型变量。

由于 cache 的参数为 s=5、E=1、b=5，故可得，地址的结构如下。

首先采用普通转置方案，即用两层循环完成矩阵转置操作。

```
void trans(int M, int N, int A[N][M], int B[M][N])
{
    int i, j;

    for (i = 0; i < N; i++) {
        for (j = 0; j < M; j++) {
            B[j][i] = A[i][j];
        }
    }

}
```

```
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:1766, misses:287, evictions:255
```

由此采用普通转置的方法的 misses = 1183，远远大于实验要求的 300，此时需要采用分块的方法，降低不命中次数。

对普通转置方案的不命中次数简要分析：

(1)对角线元素的冲突不命中：

以第一块为例，在开始前，miss=0。

如图所示，在完成转置操作的时候，首先在 chache 中匹配 A[0][0]，由于此时 cache 为空，故发生冷不命中，此时 miss++。矩阵 A 中的第一行作为 cache 第一行中的块，放入 cache 中，再将需要保存的值保存到寄存器中。

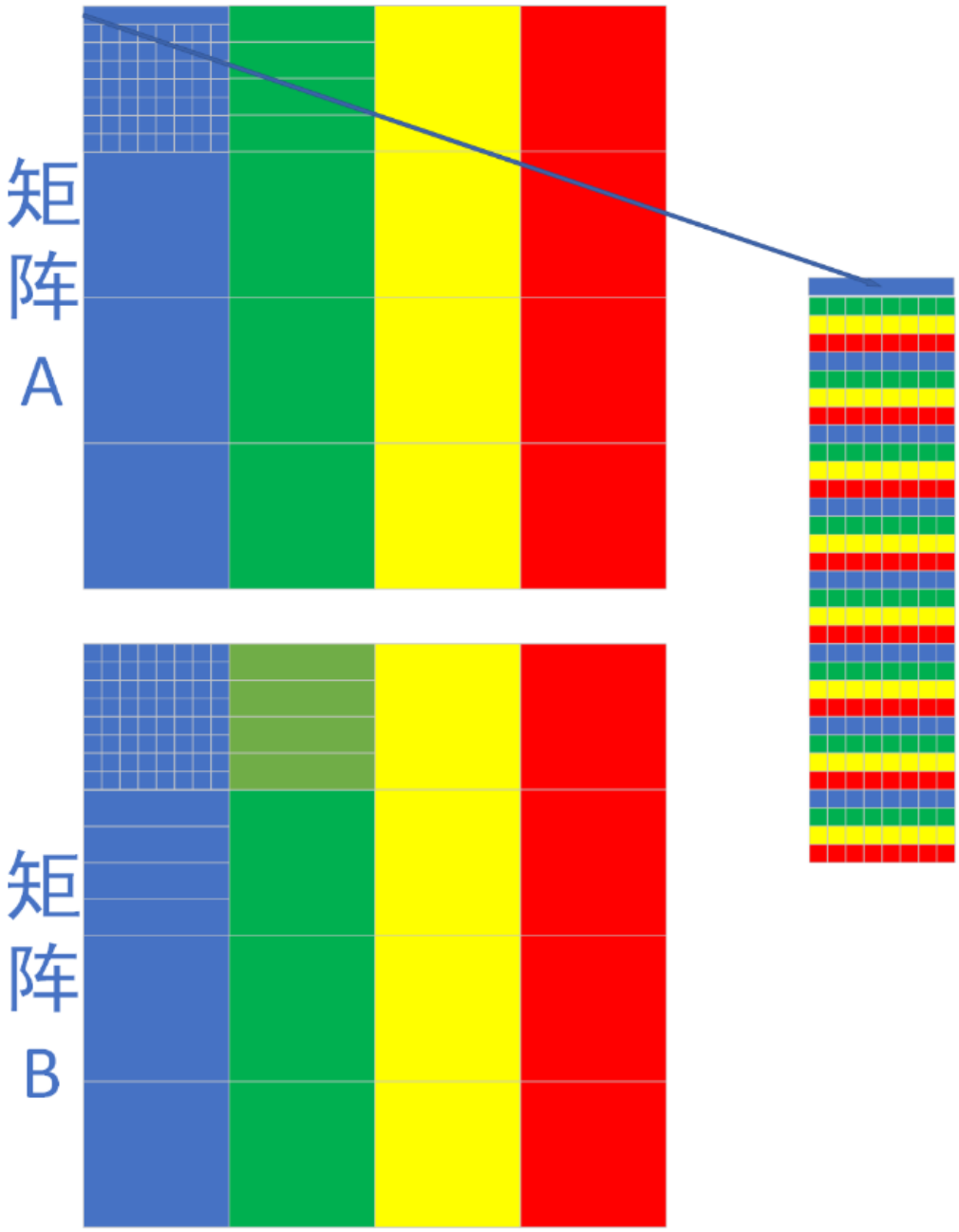
此时在 cache 中匹配 B，匹配失败，miss++，再将 B[0][0]所对应的块，放入 cache 中，根据地址，可以计算到，B 放入的位置与 A 放入的位置相同此时再将 B 所需要的值，从寄存器中放到 cache 中，即完成了对 B[0][0]的赋值操作。

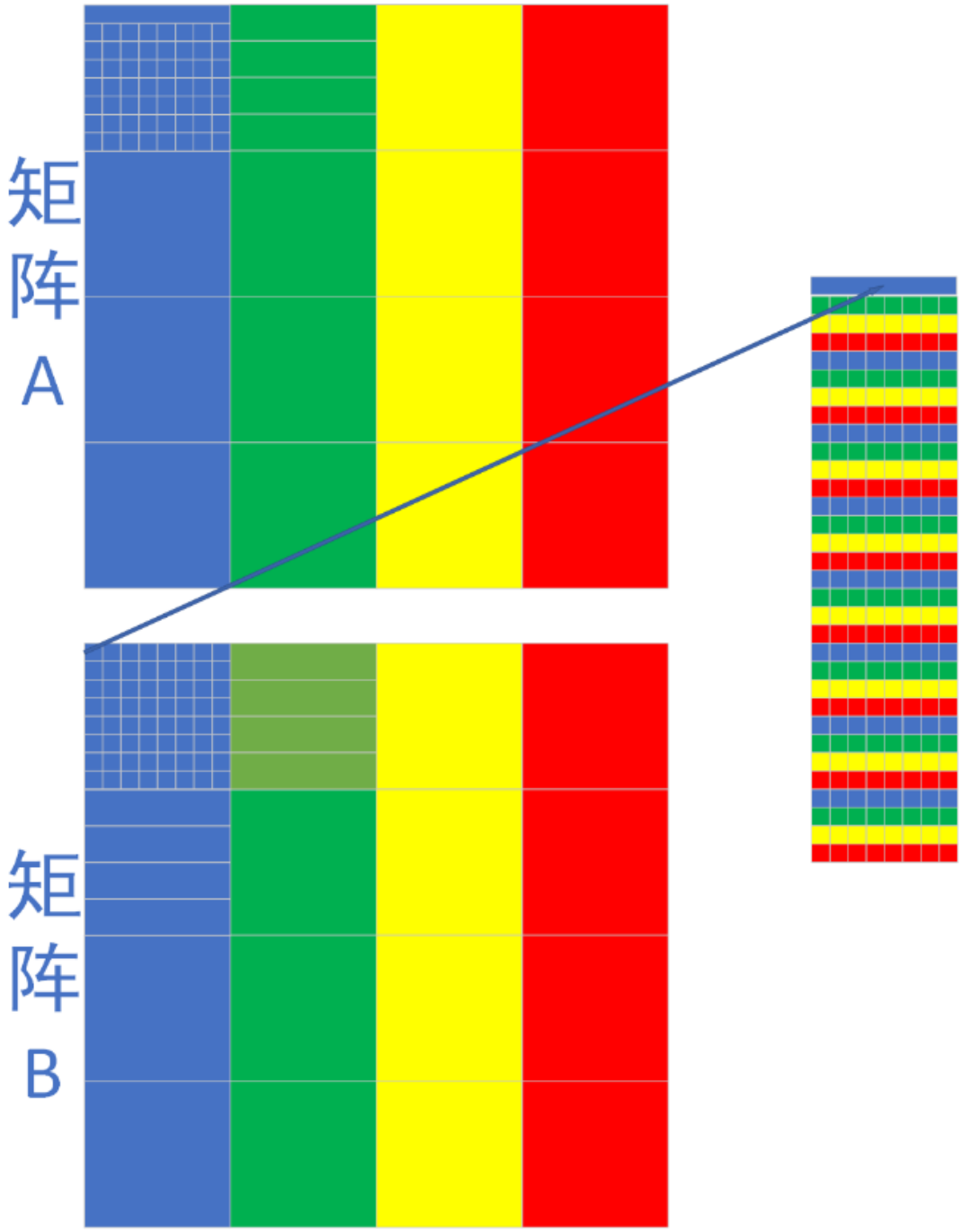
此时 j++，由于第一次将 B 的块放入了与 A 中相同的块，因此读取 A 的数

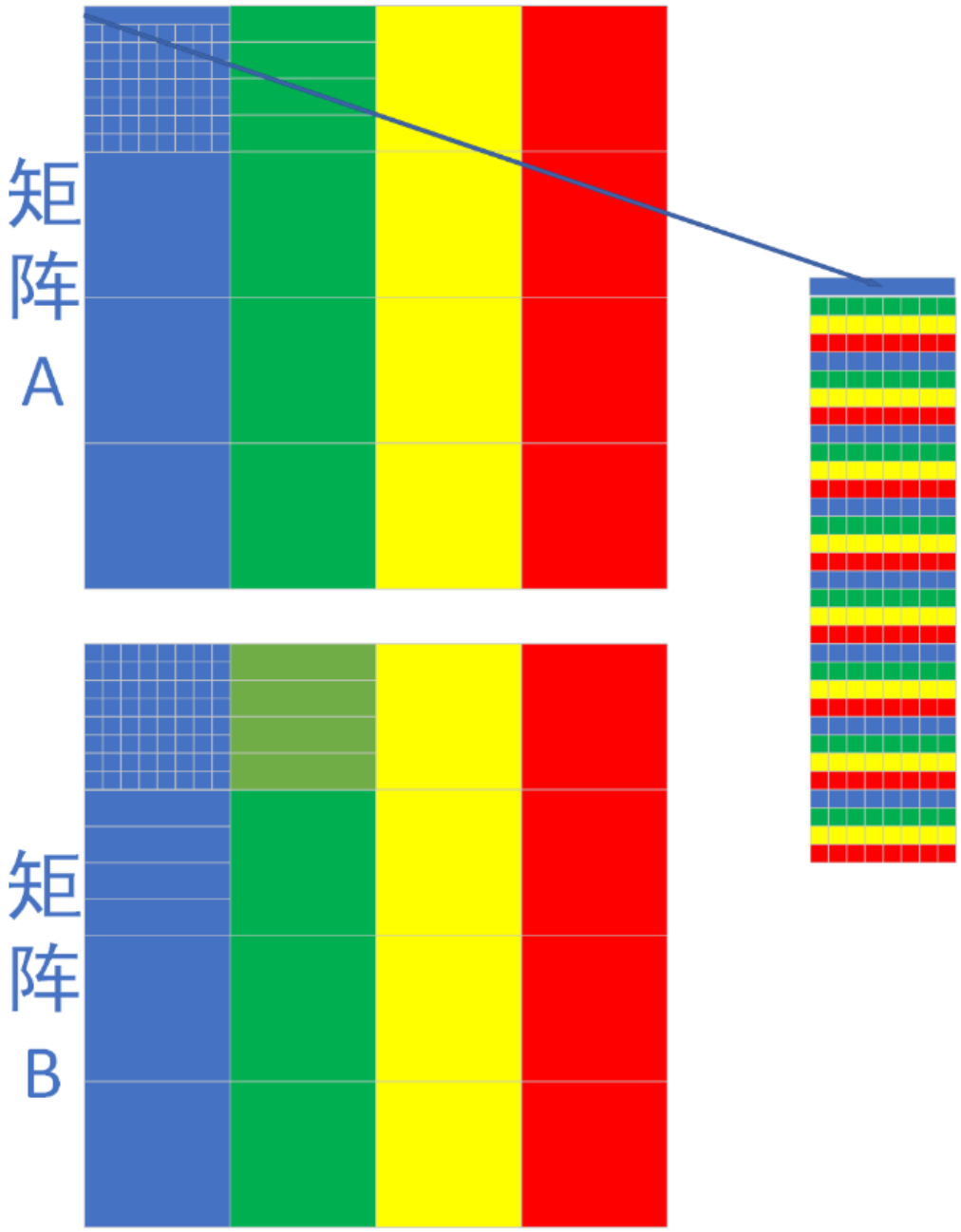
据时，会发生冲突不命中，因此此时需要，将 A 的值重新读入 cache 中，因此此时会发生冲突不命中，miss++。

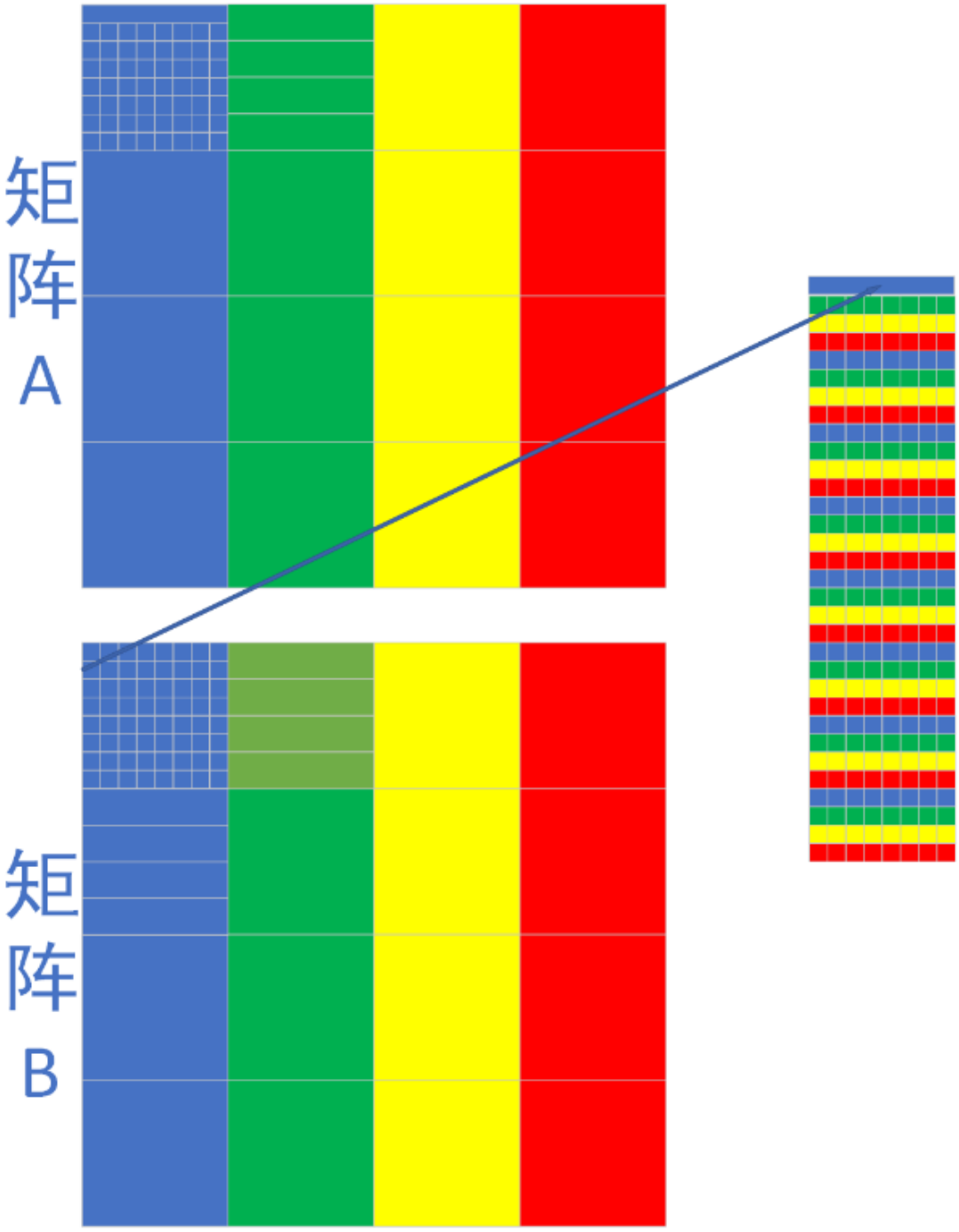
在读取 B[1][0]时，同样由于冷不命中发生 miss++。而此时，B[1][0]所对应的块应该放入 cache 的第五行(通过 b 计算可得)。因此在 j<8 时，都会发生冷不命中，而此时不与第一块中的 A 的块冲突，因此此 8*8 的方块内冲突次数为 10 * 8 = 80.而 B 矩阵共 4 块对角线与此相似的矩阵，因此由此发生的 miss 次数应该为 miss = 80 * 4 = 320。

过程图如下：

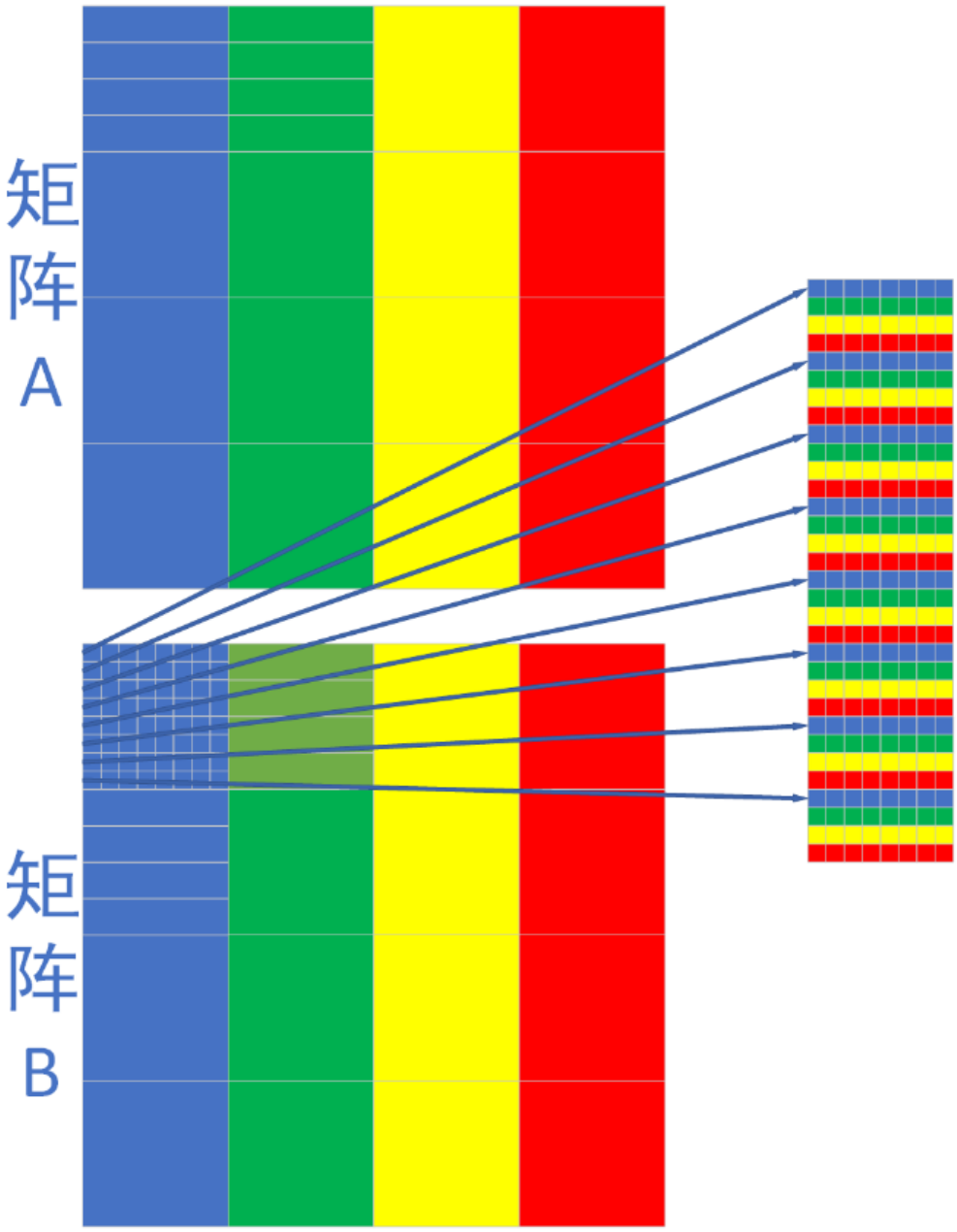

矩
阵
A
矩
阵
B


矩
阵
A
矩
阵
B


矩
阵
A
矩
阵
B



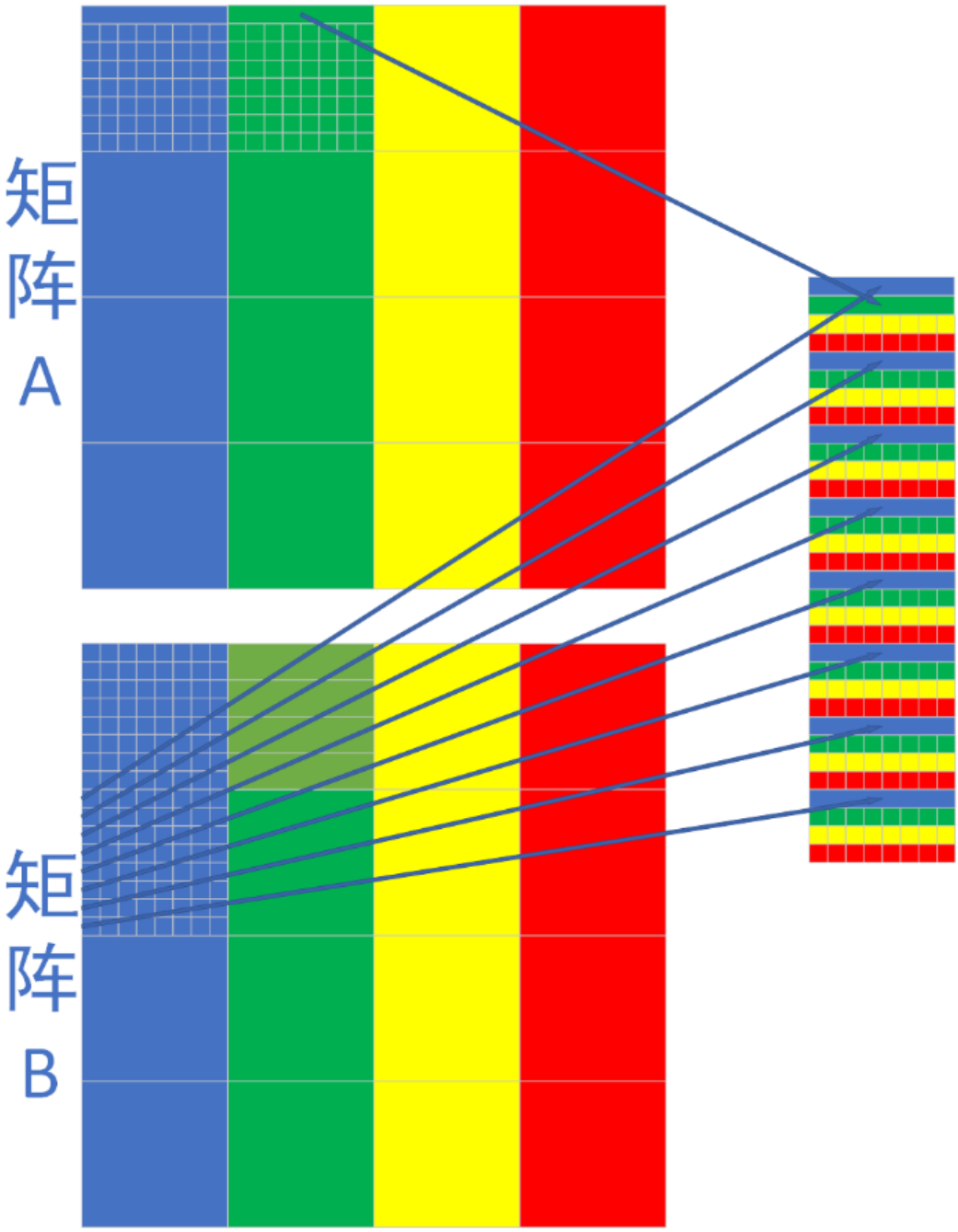
省略掉当 j>2 时的流程图，具体可以按图 3 依次类推。总体映射关系如下所示。



(2)非对角线元素

当 j>8 时，由于 cache 的规格，此时 B 中值应该重新放入第一行中，而此时 A 应该放入第二行，B 中的元素依次每搁四行放入 cache 中，并从 cache 中取出 A 的数据，保存到 B 中，因此只会发生 9 次冷不命中，故此时对于这样的小方块来说，总的 miss = 9 * 8 = 72。而一共有 12 个类似小方块，因此对于非对角线元素，miss = 72 * 12 = 864.

具体流程如下：



以上分析，基于 8 <= j < 16，若此时 j 继续增大时，此时不会继续访问 cache 中已经存在的元素，故此时需要重新访问内存，并将相应的值保存到 cache 中

因此，基于以上，可以得出 miss = 864 + 320 = 1184

1184 与实际 1183 非常接近。

从中，可以分析出一些减少 cache 不命中次数的策略。

分析修改策略：

造成 cache 不命中的很大一部分原因是因为每次 j+=8 后，由于继续向下取新的行，此时会发生冲突不命中，而若将每次取出的 B 的行都写入完毕，会减少很多不命中的次数。

因此需要将矩阵分为 8*8 的小块，每次将每个 8*8 的矩阵执行转置操作。尤其在非对角线元素处，可以极大减少，缓存不命中的次数。

而在对角线元素，采用用几个变量保存 A 中的值，来减少不命中的次数。

因此，将代码根据以上分析，修改为如下：

```
for (int i = 0; i < N; i+=8) {
        for (int j = 0; j < M; j+=8) {
            for(int p = i;p<i+8;p++)
            {
                int val_0 = A[p][j];
                int val_1 = A[p][j+1];
                int val_2 = A[p][j+2];
                int val_3 = A[p][j+3];
                int val_4 = A[p][j+4];
                int val_5 = A[p][j+5];
                int val_6 = A[p][j+6];
                int val_7 = A[p][j+7];
                B[j][p] = val_0;
                B[j+1][p] = val_1;
                B[j+2][p] = val_2;
                B[j+3][p] = val_3;
                B[j+4][p] = val_4;
```

```
                    B[j+5][p] = val_5;
                    B[j+6][p] = val_6;
                    B[j+7][p] = val_7;
                }
            }
        }
```

对此代码分析：

位于对角线的元素：

此时由于每次将 A 中的一行元素读完，故此时不会重复的取相同的 A，减少缓冲区冲突次数，由于将 A 中的一行元素一次性赋值 B，此时 B 中的 8*8 的小方块，应完全位于 cache 区，故每次在读取 A 的时候导致 cache 发生的冲突，再读取 B 所对应的一行后，可以写入 8 个 B 的值。

这样的话只会造成 23 次冲突，23 次冲突分别如下：

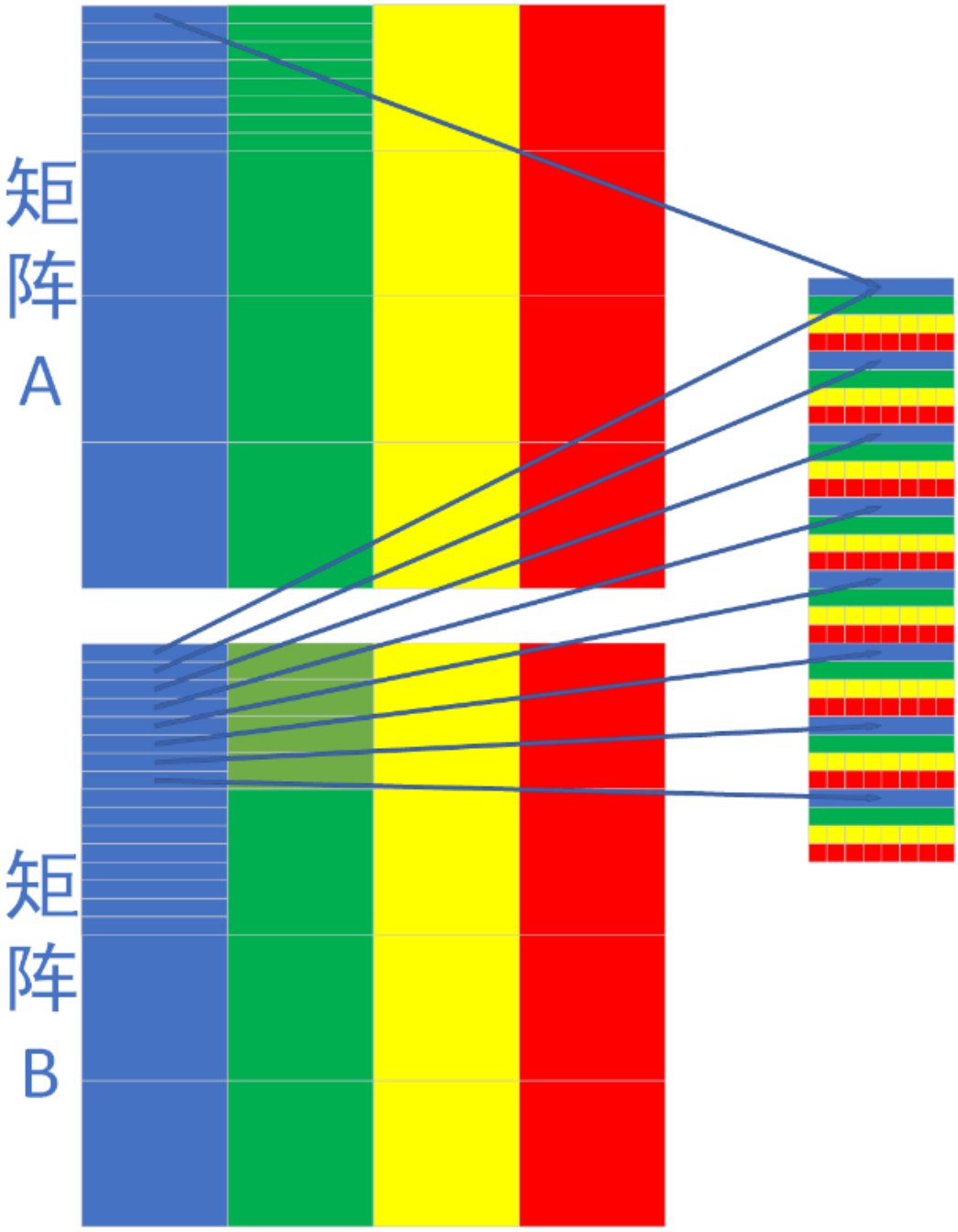
开始时候 miss = 0.

第一次读取 A 的值，miss++，将 A 的值写入 B 中，由于此时 B 中元素全不在 cache 中，因此此时需要将 8 行都存入 cache 中。因此此时 miss+=8；

第二次读取 A 得知，miss++，再将 A 的值写入 B 中，由于此时仍有 7 行的 B 的元素，保存在 cache 中，因此 miss++即可。

第三次到第八次如第二次所示。故 miss = 9 + 2 * 7 = 23.

因此此时对于对角线上的元素 miss = 23 * 4 = 92.



对于非对角线元素：

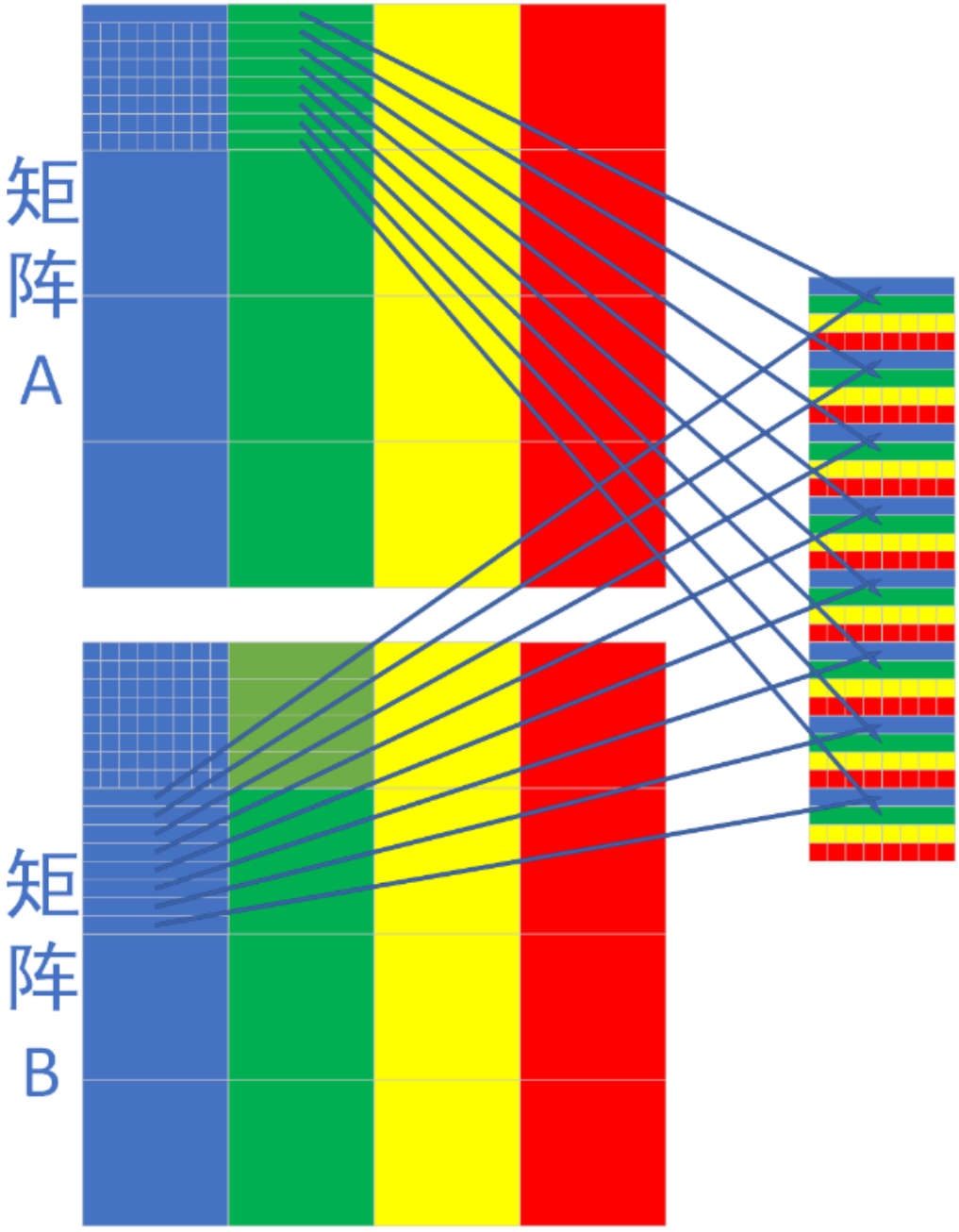
如下图所示，此时一共会造成 16 次的冲突不命中，故 miss = 16 * 12 = 192。

16 次的冷不命中如下：

第一次读取 A 的值保存在 cache 中，再读取 B 中 8*8 的小矩阵所有块，因

此此时 miss+=9，而再之后每次赋值时，只需要将 A 中元素读进 cache 即可。

因此此时 miss = 9 + 7 = 16.故非对角线元素 miss = 16 * 12 = 192.



故此时通过计算 misses = 192 + 92 = 284.

284 与实际结果 287 非常接近。

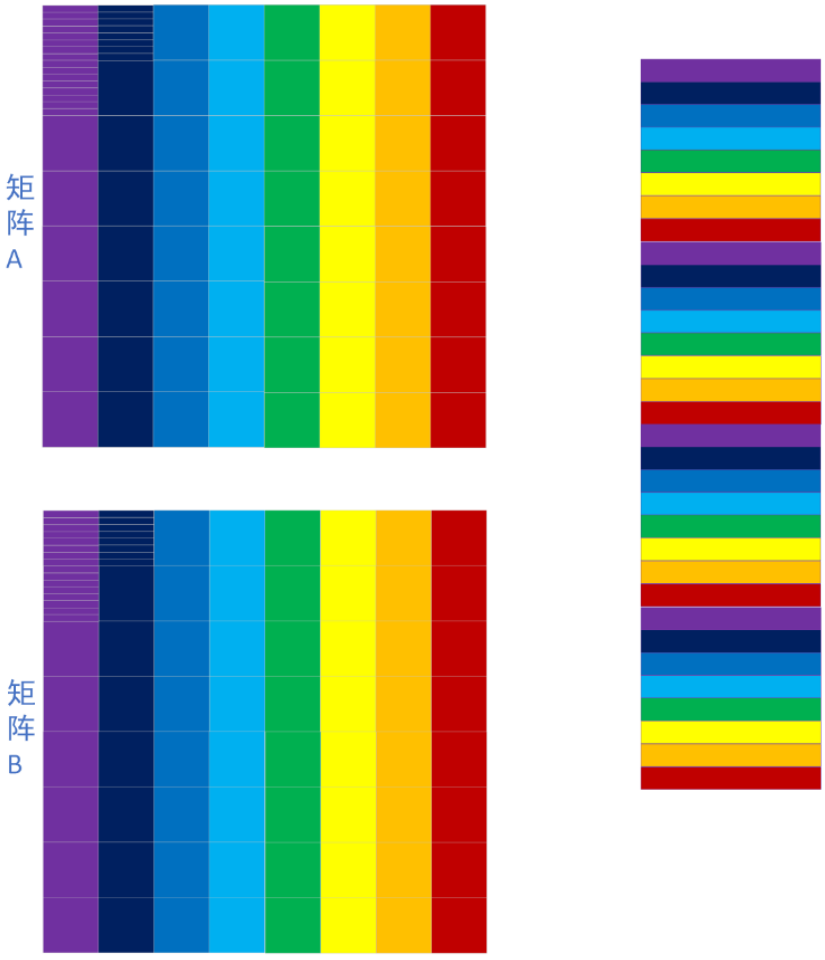
```
Function 1 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 1 (Simple row-wise scan transpose): hits:1766, misses:287, evictions:255
```

64*64：

对于 64*64 的矩阵，由于 32*32 的经验，初步想法是，将矩阵分为 8*8 的矩阵，再将 8*8 的矩阵划分为 4 个 4*4 的矩阵。

具体实现细节：

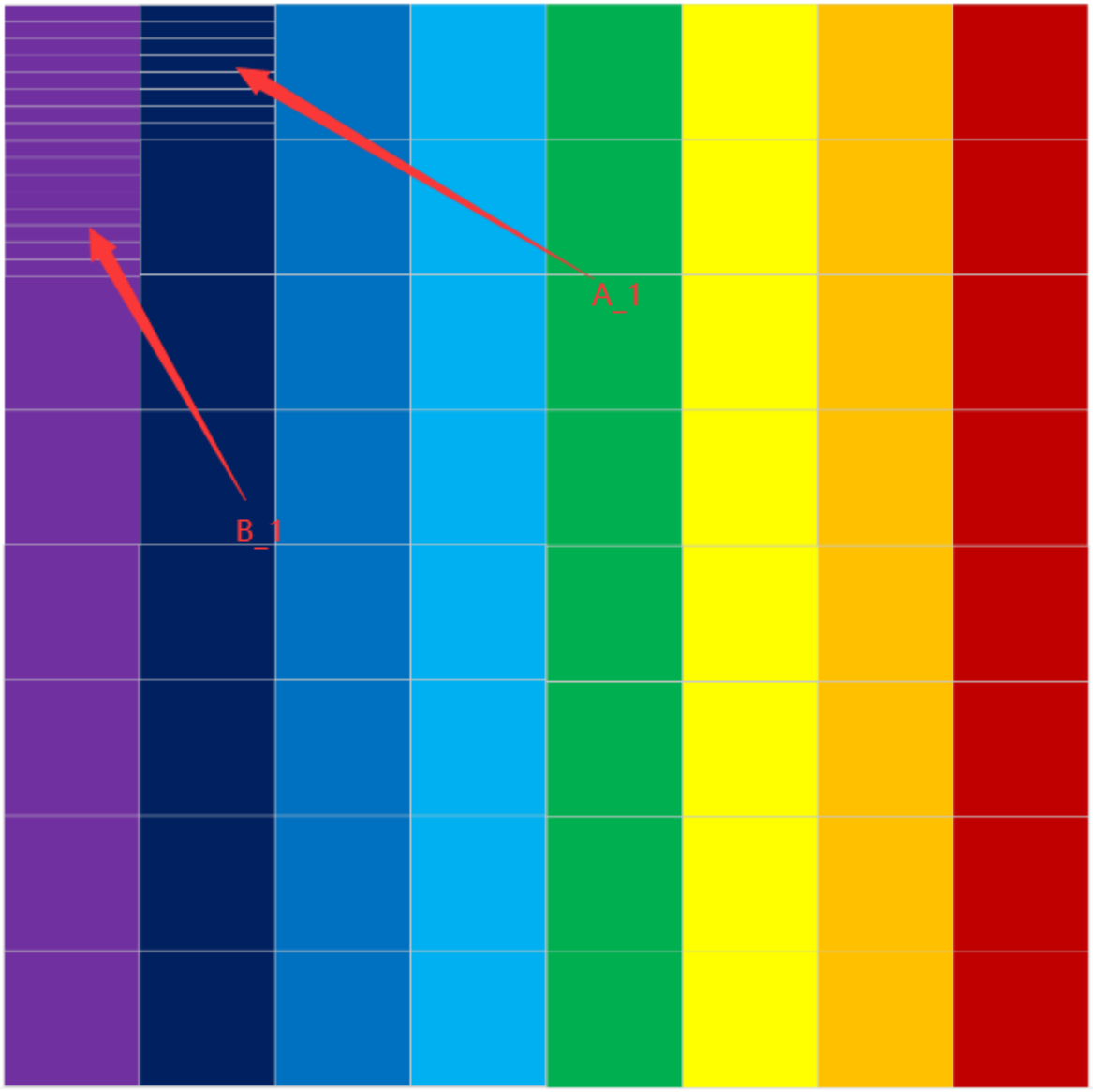
此时矩阵元素对应 cache 的块的关系如图所示：



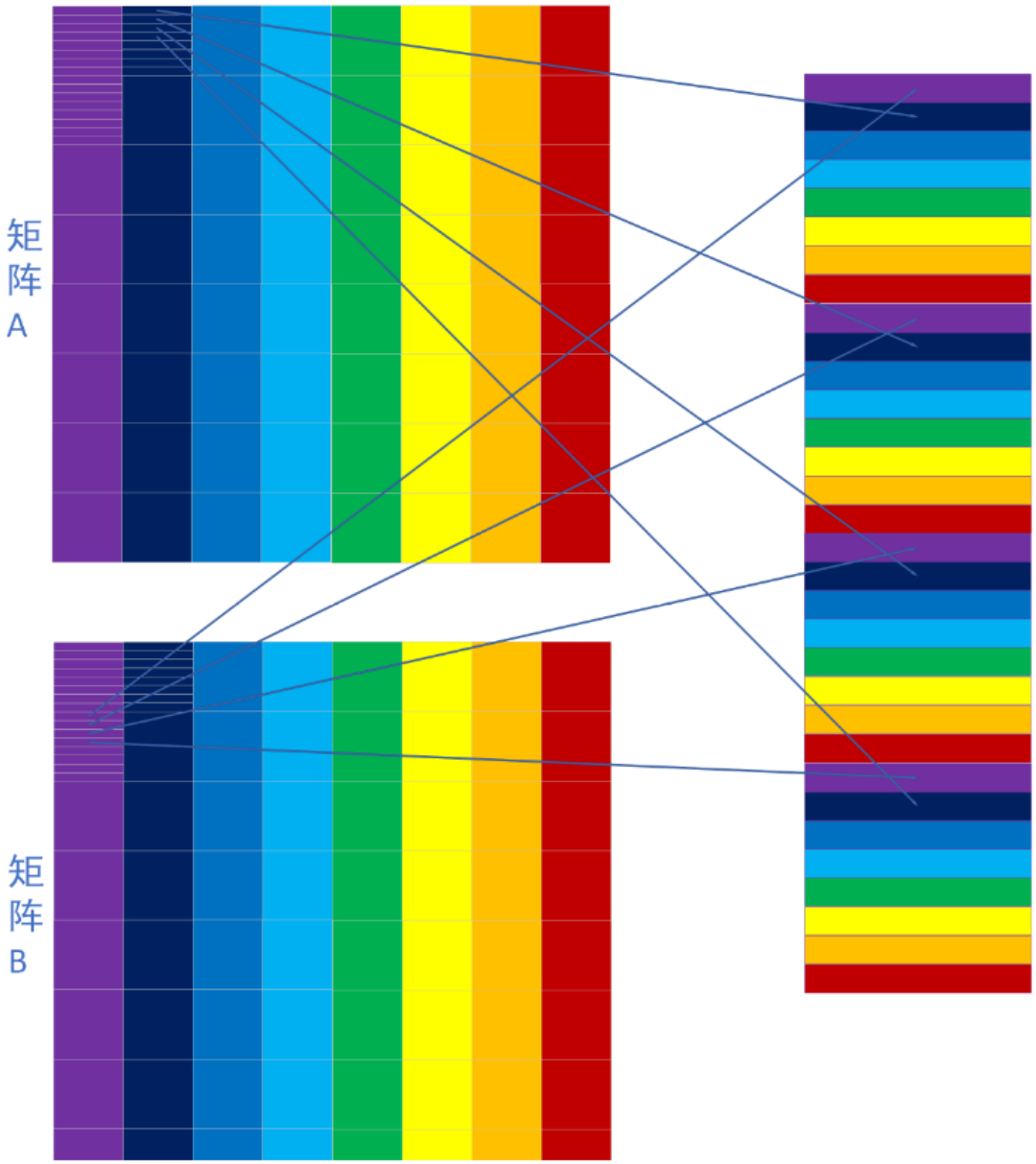
对于非对角线元素：

举例说明：

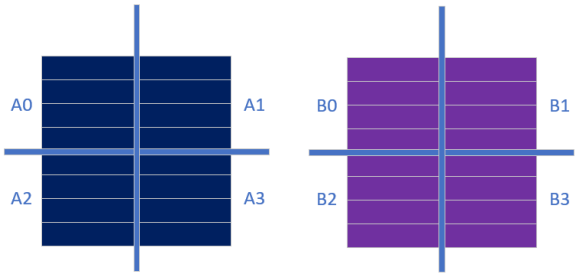
下面将说明，取如图矩阵中相似位置，在 A 中 8*8 矩阵 A_1 是如何转置并放入在 B 中相似位置处 B_1 位置中的矩阵。假设初始 miss = 0.



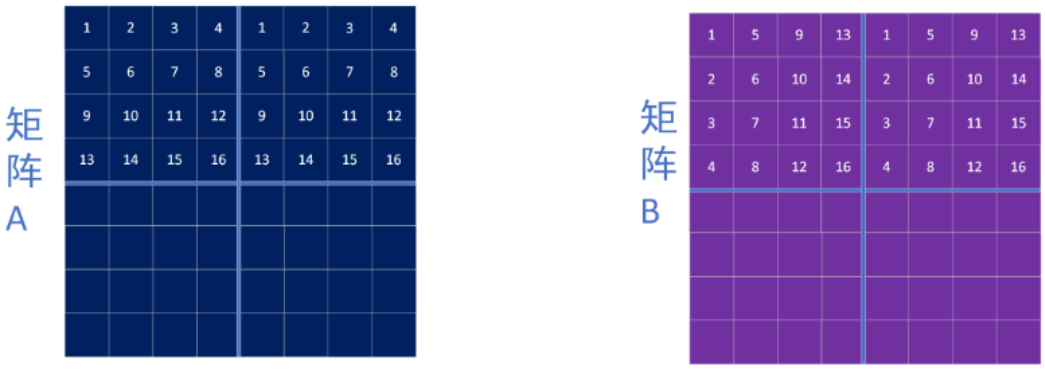
首先将 A_1 的前四行，放入 cache 中，此时发生不命中 miss += 4.再将 B_1 的前四行放入 cache 中，同样发生不命中 miss += 4.此时 A_1，B_1 的前四行，都保存在 cache 中。如下图所示：



为了方便说明具体过程，将 A_1、B_1 矩阵划分为四块。分别为 A0，A1，A2，A3，B0，B1，B2，B3.如下图所示：



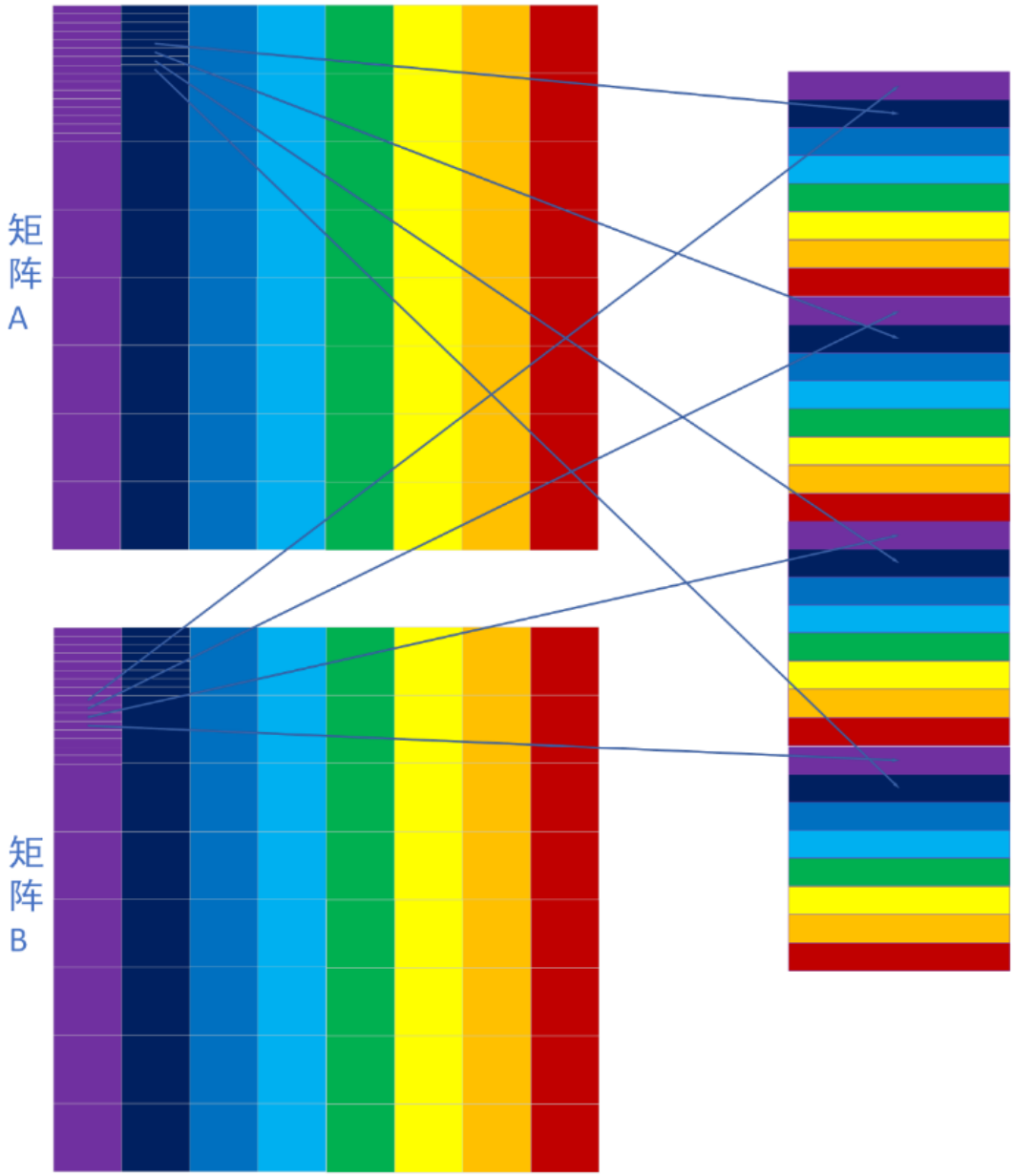
因此此时 A0，A1，B0，B1 均被加载到 cache 中，此时可以将 A0，A1 中的元素保存到 B0，B1 中，并将 A0、A1 中的元素转置。此时效果如下：



此时 misses = 8.

但是此时得到的 B1 需要将所有元素放入 B2 中，并且从 A_1 中读入 A2 并保存到 B1 中。

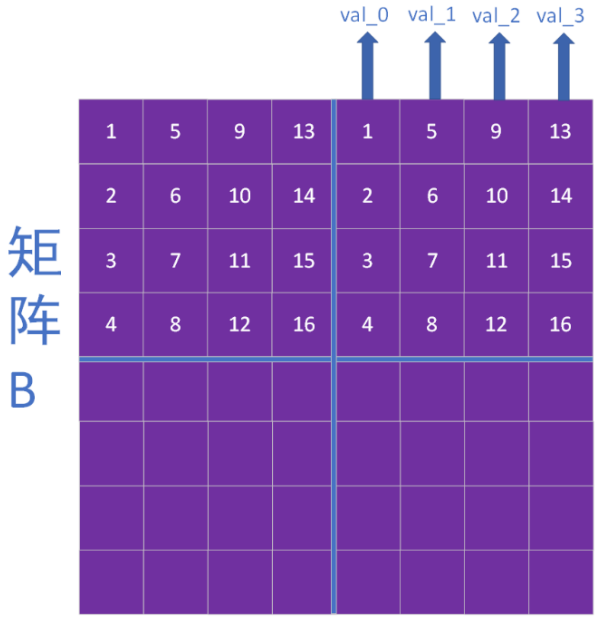
由于此时 A1、A2 中的元素已经获取，所以此时可以将 A2、A3 中的元素替换掉原来 A0、A1 中的元素。即如下图所示：



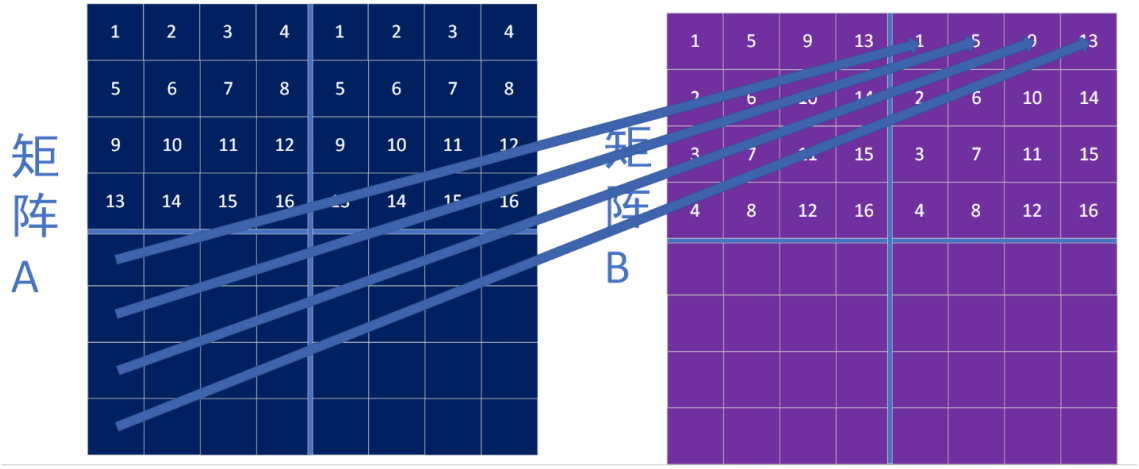
由于发生冲突不命中，此时 miss += 4 。此时需要尝试将 A2，A3 的内容写入 B2，B3 中，并将 B1，B2 的位置互换。

为了尽量减少缓冲区不命中次数，采取如下方法：

(1) 将 B0、B1 中的一行(由于 cache 每一行可以存储 8 个 int 型变量)中 B1 部分的值读取变量 val_0，val_1，val_2，val_3 中保存。



(2) 从 A2 中读取对应一列中的四个元素并保存到 B1 中的行中。



(3) 在从内存中读取 B2 中的一行，会发生冲突，此时冲突不命中，miss++，此时将 B2 中的行保存到 B0、B1 对应的行中，此时再用保存的四个变量 val_0，val_1，val_2，val_3 对 B2 中的行写入。(此处由于 A0、A1 在保存到 B0、B1 时候就已经完成了转置，因此此处用 B1 的四个变量来写入 B2 的行最终导致效果是转置)

如上述所示，依次完成所有的转置操作后，将 A3 部分转置写入 B3 中，由于此时 A3、B3 都在 cache 中，故不会发生不命中。

计算上述过程的 miss 次数：miss = 8 + 4 +1 * 4 = 16.

故对于所有非对角线块的 miss 次数为 16 * 16 = 896.

对于对角线元素：

此处通过对非对称元素分析后写出的代码，根据对称处的元素优化后，进行分析：

```
//第一步将 A0、A1 转置放入 B 中
for (p = j; p < j + 4; p++)
{
    val_0 = A[i][p];
    val_1 = A[i+1][p];
    val_2 = A[i+2][p];
    val_3 = A[i+3][p];
    B[p][i] = val_0;
    B[p][i+1] = val_1;
    B[p][i+2] = val_2;
    B[p][i+3] = val_3;
}
for (p = j+4; p < j + 8; p++)
{
    val_0 = A [i][p];
    val_1 = A[i + 1][p];
    val_2 = A[i + 2][p];
    val_3 = A[i + 3][p];
    B[p-4][i+4] = val_0;
    B[p-4][i + 5] = val_1;
    B[p-4][i + 6] = val_2;
```

```
B[p-4][i + 7] = val_3;
}
```

首先将 A 中的四行放入 cache 中，此时发生冷不命中，miss+=4，并用四个变量表示，再将 B 中的四行读入缓存，发生冲突不命中，此时 miss+=1，再将变量中保存到的 A 中的值，放入到 B 的 cache 中，此时 miss = 4 + 1 = 5.之后将 A 中元素取回 cache 时，由于 cache 中已经保存了三行 A 中的元素，故此时，实际上 miss+=1，即可完成取 A 的操作。再将 B 取回时，冲突不命中，miss++。

循环以上四此后，即将 A0 转置并放入 B0 中，此时 miss = 4 + 4 + 2 * 3 = 11.

此时 cache 中对应报纸保存的 A 与 B 的情况如下：

接下来需要将 A1 部分放入 B1 中，此时 i 由于 cache 中存在三行 A，一行 B，故此时再执行取回 A 的操作时，实际上只发生了一次冲突不命中，因此此时 miss++。之后写入 B 中时，发生不命中，miss++。因此实际上在四次循环中，每次不命中的次数为 2，因此 miss = 2 * 4 = 8.

因此完成第一步将 A0、A1 元素转置并放入 B0、B1 的操作，miss = 11 + 8 =19.

完成第一步后，cache 中保存的 A 与 B 的情况如图所示：


```
//第二步用四个变量取出 B1 中四个元素
                for (p = j; p < j + 4; p++)
                {
                    val_0 = B[p][i + 4];
                    val_1 = B[p][i + 5];
                    val_2 = B[p][i + 6];
                    val_3 = B[p][i + 7];
                    B[p][i + 4] = A[i + 4][p];
                    B[p][i + 5] = A[i + 5][p];
                    B[p][i + 6] = A[i + 6][p];
                    B[p][i + 7] = A[i + 7][p];
                    B[p+4][i] = val_0;
                    B[p+4][i + 1] = val_1;
                    B[p+4][i + 2] = val_2;
                    B[p+4][i + 3] = val_3;
                }
```

第二步，将 B1 中的元素放入 B2,并将 A2 中的元素放入 B1。

第二步中，首先要读取 B 中的元素，由于此行与 cache 中保存的行不相同，因此，此时需要重新读入 B，此时 miss++。再将 A 读入 cache 中，发生冲突不命中 miss+=4，再对 B 写入时，发生冲突不命中 miss++.再将 B[p+4]读入后，冲突不命中，miss++，在之后的循环中，每次需要先将 B 读入，发生不命中，miss++，再将 A 读入，发生两次不命中，miss+=2.再将 B 读入，发生不命中，miss++，再将 B[p+4]读入，此时 miss++，因此此步骤 miss = 7 + （1 + 2 + 1+1）* 3 = 22.

完成第二步后，cache 中 A、B 的分布情况如下：

接下来第三部分：

```
//补齐剩余 B3 部分
                for (p = i; p < i + 4; p++)
                {
                    val_0 = A[p + 4][j + 4];
                    val_1 = A[p + 4][j + 5];
                    val_2 = A[p + 4][j + 6];
                    val_3 = A[p + 4][j + 7];
```

```
                B[j + 4][p + 4] = val_0;
                B[j + 5][p + 4] = val_1;
                B[j + 6][p + 4] = val_2;
                B[j + 7][p + 4] = val_3;
        }
```

由于 cache 中保存 A[p+4]，因此再读取 B 的时候才发生冲突不命中，此时 miss+=3，在之后的循环中，再将 A 读入时，再发生冲突不命中，miss++，再将 A 的值写入 B，miss++，因此第三部分 miss = 3+2 * 3 = 9.

因此 misses = 896 + (19 + 22 + 9) * 8 = 1296 .

此时按照预估应该小于 1300.

实际运行结果如下：

```
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:8946, misses:1299, evictions:1267
```

运行结果为 1299 小于 1300，且与预估值基本相同。

61*67：

此处沿用之前的思路，发现分割为 8*8，miss 次数大于 2000，因此调整为 16*8 后，测试 miss = 1953，满足小于 2000 的要求，通过测试。

代码如图所示：

```
for( i = 0 ; i < N ; i += 16)
{
    for( j = 0 ; j < M ; j += 8)
    {
        for( p = i ; p < i + 16 && p < N; p++)
        {
            for( q = j ; q < j + 8 && q < M; q++)
            {
                B[q][p] = A[p][q];
            }
        }
    }
}
```

32×32（10 分）：运行结果截图

```
xf1190200708@1190200708:~/桌面/cachelab-handout$ ./test-trans -M 32 -N 32

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:1766, misses:287, evictions:255

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:870, misses:1183, evictions:1151

Summary for official submission (func 0): correctness=1 misses=287

TEST_TRANS_RESULTS=1:287
```

64×64（10 分）：运行结果截图

```
xf1190200708@1190200708:~/桌面/cachelab-handout$ ./test-trans -M 64 -N 64

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:8946, misses:1299, evictions:1267

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:3474, misses:4723, evictions:4691

Summary for official submission (func 0): correctness=1 misses=1299

TEST_TRANS_RESULTS=1:1299
```

61×67（20 分）：运行结果截图

```
xf1190200708@1190200708:~/桌面/cachelab-handout$ ./test-trans -M 61 -N 67

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:6226, misses:1953, evictions:1921

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:3756, misses:4423, evictions:4391

Summary for official submission (func 0): correctness=1 misses=1953

TEST_TRANS_RESULTS=1:1953
```

运行 driver.py 截图：

```
xf1190200708@1190200708:~/桌面/cachelab-handout$ python2 driver.py
Part A: Testing cache simulator
Running ./test-csim
                        Your simulator     Reference simulator
Points (s,E,b)    Hits  Misses  Evicts    Hits  Misses  Evicts
     3 (1,1,1)       9       8       6       9       8       6  traces/yi2.trace
     3 (4,2,4)       4       5       2       4       5       2  traces/yi.trace
     3 (2,1,4)       2       3       1       2       3       1  traces/dave.trace
     3 (2,1,3)     167      71      67     167      71      67  traces/trans.trace
     3 (2,2,3)     201      37      29     201      37      29  traces/trans.trace
     3 (2,4,3)     212      26      10     212      26      10  traces/trans.trace
     3 (5,1,5)     231       7       0     231       7       0  traces/trans.trace
     6 (5,1,5)  265189   21775   21743  265189   21775   21743  traces/long.trace
    27


Part B: Testing transpose function
Running ./test-trans -M 32 -N 32
Running ./test-trans -M 64 -N 64
Running ./test-trans -M 61 -N 67

Cache Lab summary:
                        Points   Max pts      Misses
Csim correctness          27.0        27
Trans perf 32x32           8.0         8         287
Trans perf 64x64           8.0         8        1299
Trans perf 61x67          10.0        10        1953
          Total points    53.0        53
```

第 4 章 总结

4.1 请总结本次实验的收获

加深了 cache 的理解。
加深了对 cache 的写回读策略与写策略。
加深了缓存不命中的理解。
掌握 gprof 进行性能分析的方法。
掌握 Valgrind 进行性能分析的方法。

4.2 请给出对本次实验内容的建议

本次实验内容过于复杂，可以拆分为两个实验，一次实验任务量过大，part B 部分说明不够详细，需要花费大量时间理解。

注：本章为酌情加分项。

参考文献

[1] RANDALE.BRYANT, DAVIDR.O‘HALLARON. 深入理解计算机系统[M]. 机械工业出版社, 2011.