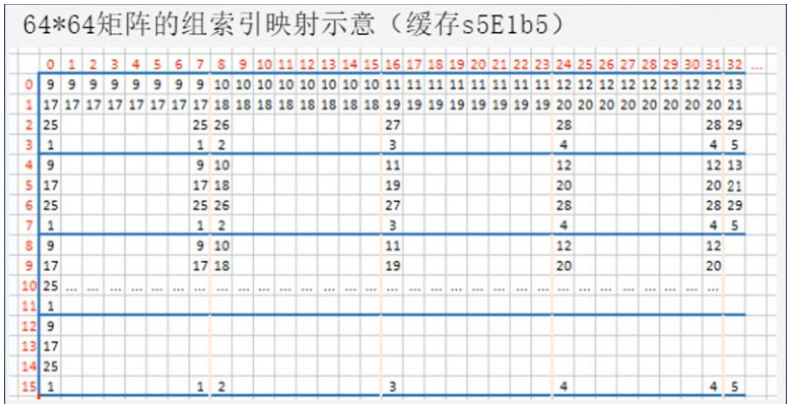
一、基于 8 分块对 M64N64 的处理

```
ubuntu@ubuntu:~/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:3586, misses:4611, evictions:4579

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=4611

TEST_TRANS_RESULTS=1:4611
```

使用的代码依旧是上次 cachelab 实验时，使用的 8*8 分块加对角线优化的方案，可以看到，miss 数达到了 4723。与不进行优化相比相差无几。

二、追踪分析其组索引，分析优化处理过程



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	…
0	9	9	9	9	9	9	9	9	10	10	10	10	10	10	10	10	11	11	11	11	11	11	11	11	12	12	12	12	12	12	12	12	13	
1	17	17	17	17	17	17	17	17	18	18	18	18	18	18	18	18	19	19	19	19	19	19	19	19	20	20	20	20	20	20	20	20	21	
2	25							25	26								27								28							28	29	
3	1							1	2								3								4							4	5	
4	9							9	10								11								12							12	13	
5	17							17	18								19								20							20	21	
6	25							25	26								27								28							28	29	
7	1							1	2								3								4							4	5	
8	9							9	10								11								12							12		
9	17							17	18								19								20							20		
10	25	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…		
11	1																																	
12	9																																	
13	17																																	
14	25																																	
15	1							1	2								3								4							4	5	

可以看到，按行来看，与之前一样每 8 个连续的元素对应一个组，而按列来看，之前 M32N32 是 8 个元素一个组，而现在则变为了 4 个连续的元素一个组。对 A 数组来说没有什么太大的影响，同样是第一个 miss 之后 7 个 hit，而对于 B 数组来说，在进行第 5 到 8 个元素时会对之前 4 个进行驱逐，这就导致了整个 B 数组都会发生 miss，使得 miss 数很大。

既然按列来看为 4 个元素，那么或许可以将 8*8 的分块与 4*4 的分块相结合来进行一个优化，主要是优化对 B 数组的 miss 数。

所以先进行了 4*4 分块的尝试，发现是可行的，miss 数降到了 1795，接下来就是对 8*8 分块的一个进一步的优化。

```
for (i=0;i<N;i+=8) {
    for (j=0;j<M;j+=8){
        for (i1=i;i1<i+4;i1++) {
            temp1=A[i1][j];temp2=A[i1][j+1];temp3=A[i1][j+2];temp4=A[i1][j+3];
            temp5=A[i1][j+4];temp6=A[i1][j+5];temp7=A[i1][j+6];temp8=A[i1][j+7];

            B[j][i1]=temp1;B[j+1][i1]=temp2;B[j+2][i1]=temp3;B[j+3][i1]=temp4;
            B[j][i1+4]=temp5;B[j+1][i1+4]=temp6;B[j+2][i1+4]=temp7;B[j+3][i1+4]=temp8;
        }
        for (j1=j;j1<j+4;j1++) {
            temp1=A[i+4][j1];temp2=A[i+5][j1];temp3=A[i+6][j1];temp4=A[i+7][j1];
            temp5=B[j1][i+4];temp6=B[j1][i+5];temp7=B[j1][i+6];temp8=B[j1][i+7];

            B[j1][i+4]=temp1;B[j1][i+5]=temp2;B[j1][i+6]=temp3;B[j1][i+7]=temp4;
            B[j1+4][i]=temp5;B[j1+4][i+1]=temp6;B[j1+4][i+2]=temp7;B[j1+4][i+3]=temp8;
        }
        for (i1=i+4;i1<i+8;i1++) {
            temp1=A[i1][j+4];temp2=A[i1][j+5];temp3=A[i1][j+6];temp4=A[i1][j+7];

            B[j+4][i1]=temp1;B[j+5][i1]=temp2;B[j+6][i1]=temp3;B[j+7][i1]=temp4;
        }
    }
}
```

```
ubuntu@ubuntu:~/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:9066, misses:1179, evictions:1147

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=1179

TEST_TRANS_RESULTS=1:1179
```

根据上方的组索引映射图来看，如果对 4*8 的矩阵进行处理，不会增加大量的 miss 数，于是在最内部的第一个循环中，先将一个 4*8 的矩阵从 A 放到 B 对应的位置（仍为 4*8），之后在第二个循环中，将这个 4*8 矩阵的右半部分 4*4 从 B 放到 B 中的正确的位置，并将 A 中左侧 4*4 下方的 4*4 矩阵放到 B 中正确的位置，最后一个循环则将剩下的一个 4*4 的矩阵从 A 放到 B 中正确的位置。这样使得 miss 数降为了 1179.

三、对 M61N67 处理

依次进行分块处理，查看其 miss 数，从 4 分块开始依次加 1 直到 8，得到的数据如下：

4 分块：2425	5 分块：2296	6 分块：2244	7 分块：2152	8 分块：2118
9 分块：2092	10 分块：2076	11 分块：2089	12 分块：2057	13 分块：2048
14 分块：1996	15 分块：2021	16 分块：1992	17 分块：1950	18 分块：1961
19 分块：1979	20 分块：2002	21 分块：1957	22 分块：1959	23 分块：1928
24 分块：2015	25 分块：2107	26 分块：2202	27 分块：2298	28 分块：2400
29 分块：2495	30 分块：2595	31 分块：2591	32 分块：2590	

原本我是打算找下规律的，结果发现真的没有什么规律，就从 4 分块开始依次找到 32 分块，统计得出，当分块为 23 时，可以使 miss 数最低，小于 2000，符合要求。

然而，在将代码整合到 transpose_submit 函数中时，由于一次不经意间的写代码时的错误，不小心在进行分块的两层循环处理时，弄混了 i 和 j，导致其交换了位置。然而正是这次不经意间的错误，使得 miss 数更加的低了。然后我再次进行了测试，发现从 4 分块到 14 分块依次递减，到 15 会有一个增加，而 16 分块 17 分块则再次减少，此后的值均高于 17 分块，17 时最低为 1813。

```
int i, j, i1, j1;
int block = 17;
int temp;
for (i = 0; i < N; i += block)
    for (j = 0; j < N; j += block)
        for (i1 = j; i1 < j + block && i1 < N; ++i1)
            for (j1 = i; j1 < i + block && j1 < M; ++j1) {
                temp = A[i1][j1];
                B[j1][i1] = temp;
            }
```

四、整理实验信息

a) 实验目标：PartA 编写一个 cache 模拟器，理解 cache 与地址的映射关系，了解 cache 的组行块，了解 LRU 更新策略的机制，深入理解 miss hit eviction 的产生及原因。PartB 通过完成矩阵的转置，来学会如何编写对 cache 友好的代码，更加具体的理解 cache 的工作机制。

b) 资源：valgrind-3.12.0，cachelab-handout，Ubuntu-18.04，cachelab 介绍的 pdf 和 ppt 文件，moodle 系统上的讲解视频以及实验引导。

c) 实验步骤：

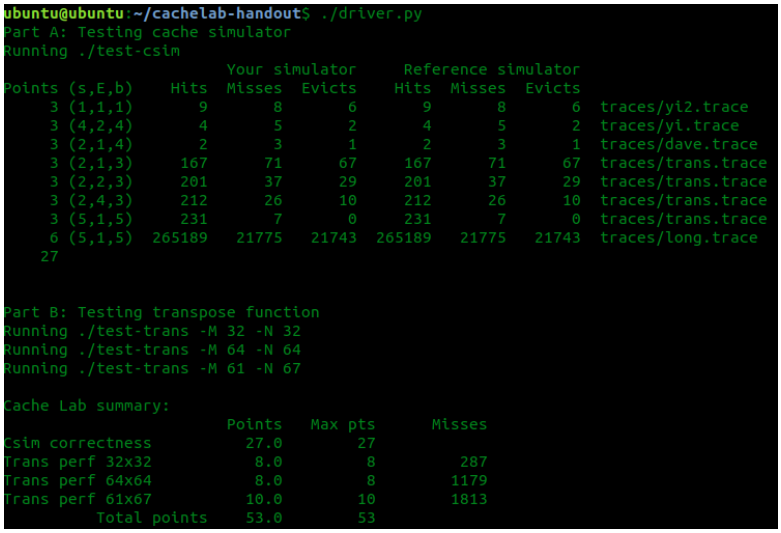
i. 安装环境：通过 sudo apt-get install 安装 valgrind 和 build-essential 软件包来安装 gcc，make 指令等内容。

ii. PartA 部分：通过./csim-ref -h 来了解各个命令行的作用，./csim-ref -v -s x -E x -b x -t xxx.trace 来使用附带的 cache 模拟器了解其行为。之后依据 csim-ref 来编写自己的 csim，包括依据输入的 sEb 对 cache 进行动态分配空间，对 L、S、M 指令的处理，对 LRU 策略的使用以及如何判断 miss 或 hit。最后可以用./test-csim 来检测编写的 csim 的正确性

iii. PartB 部分：用./test-trans -M x -N x 来生成 trace.f1 文件，可以通过 partA 部分编写的 cache 模拟器来对其进行分析。之后就是编写 trans.c 文件来对其进行优化，处理的矩阵有三个：32*32 64*64 61*67，依次完成三个部分。最后通过./test-trans -M x -N x 来查看优化后的 miss 数。

五、实验结果

在 cachelab.pdf 中可以看到，PartA 部分有 27 points，在 PartB 部分有 27 points（包含 1 个正确性 point），代码风格方面有 6 points。



在上半部分是对 PartA 的检测，程序通过比较 csim 得到的 miss hit eviction 数和 csim-ref 得到的，如果相同则得到相应的分，最左侧一栏标明了每一个 trace 文件对应的分值，最下方为总得分数，可以看到得到了 27 分，csim 编写正确。

下半部分是对 PartB 的检测，通过测试三个数组的 miss 数，小于一个给定值则得到满分，如果大于该值但不超过上限会根据 miss 数得到对应的分值。中间 Points 一栏为得到的分数，Max pts 一栏为该项的最大分数，最右侧为 miss 数。可以看到，末尾的 total points 为 53 分，得到了满分。

通过查看得到的分数，可以表明该实验是正确的。

六、实验总结

这次的实验给我学习 CSAPP 第六章的内容留下了深刻的印象，在对 cache 进行模拟的过程中，让我完全的掌握了 cache 与地址之间的映射关系，以及 cache 各个部分的大小的分配，所以在之后学习理论课程时，就很轻松。

同时，知道了还有 getOpt 这一个函数的存在，以及第一次使用了 main 函数中的两个参数，曾经以为没有什么用，这次用到了发现作用还挺大。在做 PartB 时，也是讲课本上学到的知识通过实践来加深自己的理解，第一次编写了对 cache 友好的代码，也学会了根据组索引来编写代码，充分的利用 cache。。

在学习方法方面，则还是体会到了那一句话：纸上得来终觉浅，绝知此事要躬行。

而且本次实验也再次提高了我的 debug 能力，由于在编写 csim.c 的时候，需要对 cache 进行分配以及有大量对指针的操作，所以某些时候如果处理不好，就会产生段错误。在帮同学 debug 时，也是在这方面耗费了较多的精力。