

Cache lab 实验报告

该实验分为两个部分，第一个部分是实现一个替换策略为 LRU 的 cache 模拟器，计算相应操作下命中，不命中和替换的个数，第二个部分是改写一个转置矩阵的 C 代码，使其不命中数降低到一定范围。

第一个部分：

实现一个替换策略为 LRU 的 cache 模拟器，并计算命中，不命中和替换的个数，最后调用 printSummary 函数。根据 writeup 的要求，对每一个测试用例，测试程序会用 ./csim [-hv] -s <s> -E <E> -b -t <tracefile> 的格式运行我们的程序，其中 s 是模拟的组索引的位数，E 是每组的行数，b 是块的位数，t 是运行的文件名，因此，我们程序第一个要解决的问题是将这些参数读入，并利用这些参数对我们的 cache (可以用一个结构体表示) 初始化，由于命令行用字符串形式被我们程序读取，我们要给 main 函数设置传入参数来拿到我们的数据，同时我们也要将这些数据转化为临时变量放在 main 函数中，于是可以打出如下代码：

```
void getinfo(int argc, char* argv[], int* s, int* e, int* b, char* filename)
//从argv[]中读数据出来，此处用地址传递，以修改main函数中对应变量的值
{
    for (int i = 1; i < argc; i += 2)
//由于argv[0]是"./test-csim"，没有必要读，所以从1开始
//i+=2是因为每个参数前面都有对应的提示符，因此两个两个一组，对于h和v语句，特殊处理即可
    {
        switch (argv[i][1]) //对应提示符的格式是"-X"，所以是读第二个字符
        {
            case 's':
                *s = atoi(argv[i + 1]);
                break;
            case 'E':
                *e = atoi(argv[i + 1]);
                break;
            case 't':
                strcpy(filename, argv[i + 1]); //拿到文件名
                break;
            case 'b':
                *b = atoi(argv[i + 1]);
                break;
            case 'h': //处理h和v语句
            case 'v':
                i--;
            default: break;
        }
    }
}

int main(int argc, char* argv[]) //argc代表命令行中字符串的个数，argv是一个字符串数组，
//存储了命令行中的所有字符串
```

```

{
    int S, E, B;
    char filename[30]; // 文件名
    getinfo(argc, argv, &S, &E, &B, filename);
    return 0;
}

```

接着初始化我们的 cache，由于我们不需要管 cache 里到底存了啥，块中的数据不需要在 cache 中呈现，同时又由于替换策略为 LRU，我们要一个 time_count 去记录每一行的存在时间，因此我们可以这样设置 cache 这个结构体：

注意：该处全是 int 类型 (32 位) 是由于 trace 文件中的地址索引位为 32 位，64 位则需要 long

```

typedef struct
{
    int valid; // 有效位
    int tag; // 标记位，表示标记位的实际值，注意：不是标记位的位数，而是标记位的实际值，
    这样设计的原因是因为在比较标记位时，可直接对位进行比较，相等时则命中
    int time_count; // 访问时间计算
} line;
typedef struct
{
    line* l; // set 里有 n 行，用指针替代
} set;
typedef struct
{
    int S_count; // 组的个数
    set* Set; // cache 有 n 组，同样用指针，这样好进行动态分配
    int E; // 每组的行数
} cache;

```

接着要初始化 cache，写一个函数对他初始化：

void initialize_cache(cache* cc, int S, int E) // cache 用指针传，以改变 main 函数中的 cache

```

{
    cc->S_count = pow(2, S); // 算出组的总数
    cc->E = E; // 每组的行数直接赋值
    cc->Set = (set*) malloc(cc->S_count * sizeof(set)); // 分配组的空间
    for (int i = 0; i < cc->S_count; i++) {
        cc->Set[i].l = (line*) malloc(cc->E * sizeof(line)); // 对每一组分配行的空间
        for (int j = 0; j < cc->E; j++) // 对每一行的有效位，标记位，时间计算数赋初值
        {
            cc->Set[i].l[j].valid = 0;
            cc->Set[i].l[j].tag = 0;
            cc->Set[i].l[j].time_count = 0;
        }
    }
}

```

初始化好 cache 后，就要根据文件的内容进行命中 & 不命中 & 替换数的计算了，于是接着是读取文件内

容的(根据writeup, 文件内容有四种: 1. 以I开头的行(需要跳过); 2. 以空格+L开头的行(需要计算该行的命中&不命中&替换次数); 3. 以空格+S开头的行(需要计算该行的命中&不命中&替换次数); 4. 以空格+M开头的行, 相当于一次L, 一次S))操作: (下述操作均在main函数中)

```
FILE* fp;//文件指针
fp = fopen(filename, "r");//读文件
if (fp == NULL)return -1;//文件打开异常则返回
char temp_line[max];//每一行的缓冲区, max是一个宏, 大小为100(我自己设的), 在代码的完整版中会有, 以限制非法输入
while (fgets(temp_line, max, fp) != NULL)//fgets函数读文件的每一行, 将该行最多max个字符读到temp_line中
{
    if (temp_line[0] == 'I')continue;//跳过“I”行
    cal_hit_miss_evi(temp_line, &cc, B, S);// cal_hit_miss_evi() 函数计算本行的命中&不命中&替换数, 接下来会详细介绍
}
fclose(fp);//关闭文件指针
```

在计算命中&不命中&替换数之前, 我们需要对应的变量来存储命中&不命中&替换数, 我将这三个变量设置到全局区:

```
int hit = 0, miss = 0, e = 0;//hit: 命中数; miss: 不命中数; e:替换数
```

接着是cal_hit_miss_evi函数:

```
void cal_hit_miss_evi(char* this_line, cache* c, int B, int S)//传入B(块的位数), 以计算tag的位数(我们的cache只保留标记位的实际值, 而不保留标记位的位数)
{
    int hang = c->E;//每组的行数, 取出来后面方便使用
    int address;//每行的地址索引
    char mode;//确定是M, 还是L, 还是S
    int matched_set;//匹配到的组
    int longest_time = 0;//初始化被替换行的下标, 需要替换时才会被用到
    int temp_tag;//该行地址索引的标记位的实际值
    sscanf(this_line, "%c %x", &mode, &address);//boomlab中出现的函数, 可以将字符串转化为对应格式的数据, 此处将mode直接变为字符即可, 而trace文件中的地址索引是16进制表示, 故转化为%x(16进制)形式存入address
    address >>= B;//把块的偏移位去掉
    matched_set = address & (0x7fffffff >> (31 - S));//计算对应的组位置, 此处的 右移 和 31 - S 操作是为了确定address的低多少位是组索引位
    temp_tag = address >> S;//得到标记位
    for (int i = 0; i < hang; i++) { //遍历对应组的每一行, 看是否命中
        if (c->Set[matched_set].l[i].valid == 1 && temp_tag == c->Set[matched_set].l[i].tag)//有效位为1且标记相同, 则命中
        {
            if (mode == 'M')hit++; //由于M是L+S, 第一次就命中的话, 第二次肯定还命中, 所以额外加一次
        }
    }
}
```

```

        hit++; //命中就加一次
        for (int j = 0; j < hang; j++) //每次读完文件的一行，必须更新时间
        {
            if (c->Set[matched_set].l[j].valid == 1) {
                if (i == j) c->Set[matched_set].l[j].time_count = 0; //命中位的时间
                else c->Set[matched_set].l[j].time_count++; //其他位的时间加1
            }
        }
        return; //命中后返回
    }
}

miss++; //行走完了还没返回，说明没命中
//注意：没命中分为冷不命中和冲突不命中，此时要对情况进行判断，因此，要再遍历一遍对
//应组的所有行，看有没有空行，有空行就插入，没空行才替换
for (int i = 0; i < hang; i++)
{
    if (c->Set[matched_set].l[i].valid == 0) //遇到空行
    {
        c->Set[matched_set].l[i].valid = 1; //有效位改为1
        c->Set[matched_set].l[i].tag = temp_tag; //更新标记位
        for (int j = 0; j < hang; j++)
            if (i != j && c->Set[matched_set].l[j].valid == 1) //对本行以外的行加1来
            更新时间
                c->Set[matched_set].l[j].time_count++;
        if (mode == 'M') hit++; //如果是M模式，第一次不命中，第二次一定命中(第一次会
        将tag写入cache，所以第二次一定命中)
        return;
    }
}

e++; //需要替换，替换次数加1
for (int i = 0; i < hang; i++) //遍历每一行，找到时间最久的那一行
    if (c->Set[matched_set].l[i].time_count >
        c->Set[matched_set].l[longest_time].time_count)
        longest_time = i; //更新被替换行
c->Set[matched_set].l[longest_time].tag = temp_tag; //找到后进行替换
for (int j = 0; j < hang; j++) //更新时间
    if (c->Set[matched_set].l[j].valid == 1)
        c->Set[matched_set].l[j].time_count++;
c->Set[matched_set].l[longest_time].time_count = 0; //被替换的一行时间置0
if (mode == 'M') hit++; //如果是M模式，第一次不命中，第二次一定命中(第一次会将tag
写入cache，所以第二次一定命中)
}

```

计算完成后，就是一些释放内存的操作：

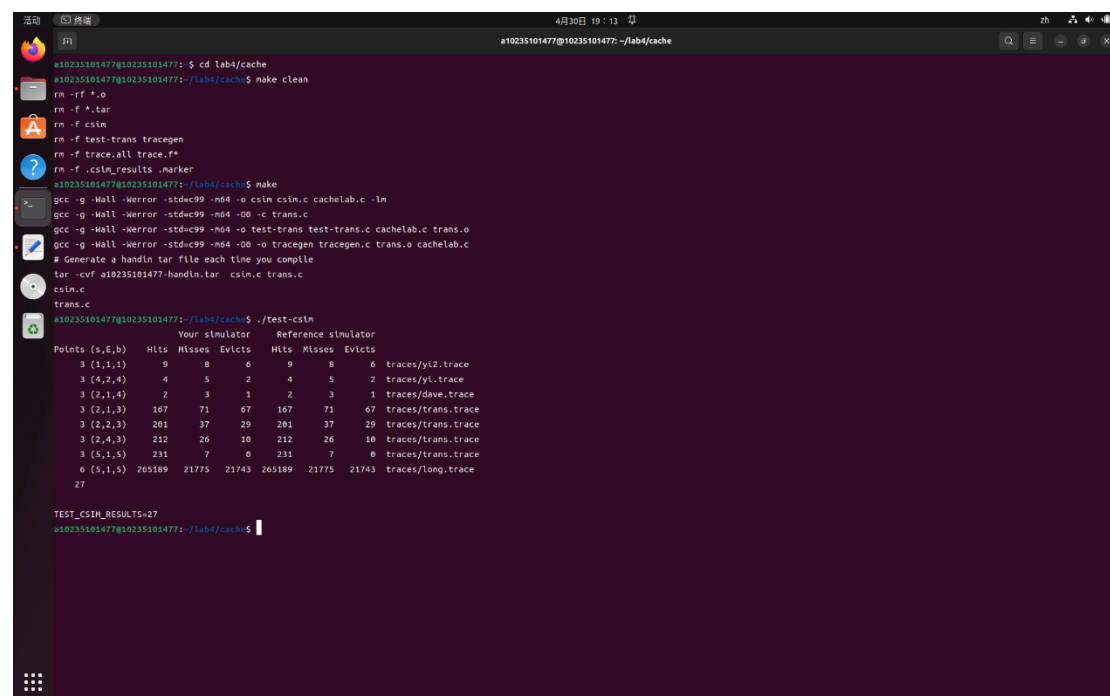
```
for (int i = 0; i < cc.S_count; i++)//释放内存
{
    free(cc.Set[i].l);
}
free(cc.Set);
```

最后调用函数printSummary

```
printSummary(hit, miss, e);//按要求调用函数
return 0;//结束main函数
```

以上代码的注释大部分是在写实验报告中写的，完整版中(见csim.c文件)只有一些注释，接下来是跑分的截图。

跑分结果截图：



```
4月30日 19:13
a10235101477@10235101477:~/lab4/cache
a10235101477@10235101477:~/lab4/cache$ cd lab4/cache
a10235101477@10235101477:~/lab4/cache$ make clean
rm -rf *.o
rm -f *.tar
rm -f csim
rm -f test-trans tracegen
rm -f trace.all trace.f*
rm -f .csim_results .marker
a10235101477@10235101477:~/lab4/cache$ make
gcc -g -Wall -Werror -std=c99 -m64 -o csim csim.c cachelab.c -lm
gcc -g -Wall -Werror -std=c99 -m64 -O0 -c trans.c
gcc -g -Wall -Werror -std=c99 -m64 -o test-trans test-trans.c cachelab.c trans.o
gcc -g -Wall -Werror -std=c99 -m64 -O0 -o tracegen tracegen.c trans.o cachelab.c
# Generate a handin tar file each time you compile
tar -cvf a10235101477-handin.tar csim.c trans.c
csim.c
trans.c
a10235101477@10235101477:~/lab4/cache$ ./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/yi1.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
a10235101477@10235101477:~/lab4/cache$
```

第二部分：

该部分有三个小实验，根据writeup的要求，我们可以分别改写32*32，64*64，61*67三个矩阵的转置函数，提供的cache参数是组索引位数为5，偏移量的位数为5，高速映射，故有32个set，每个set可以存32个字节。

先看第一个32*32的(不命中数需小于300)：

由于转置后的数组和原数组在cache中映射位置可能不同，故我们先要查看两个数组的映射位置是否相同，故我们先提交一遍tran.c里自带的转置函数，利用writeup里面告诉我们的-v选项，生成对应的trace，来看数组的地址：

```

a10235101477@10235101477: /lab4/cache$ cd lab4/cache
a10235101477@10235101477: /lab4/cache$ make clean
rm -rf *.o
rm -rf *.tar
rm -f csim
rm -f test-trans tracegen
rm -f trace.all trace.*
rm -f csim_results.marker
a10235101477@10235101477: /lab4/cache$ make
gcc -g -Wall -Werror -std=c99 -m64 -o csim csim.c cachelab.c -lm
gcc -g -Wall -Werror -std=c99 -m64 -O0 -c trans.c
gcc -g -Wall -Werror -std=c99 -m64 -o test-trans test-trans.c cachelab.c trans.o
gcc -g -Wall -Werror -std=c99 -m64 -O0 -o tracegen tracegen.c trans.o cachelab.c
# Generate a handin tar file each time you compile
tar -xvf a10235101477-handin.tar csim.c trans.c
csim.c
trans.c
a10235101477@10235101477: /lab4/cache$ ./test-trans -N 32 -M 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:869, misses:1184, evictions:1152

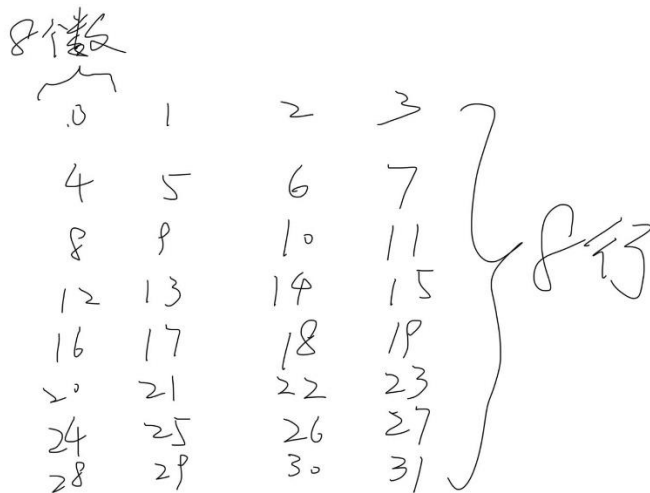
Function 1 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
Func 1 (Simple row-wise scan transpose): hits:869, misses:1184, evictions:1152

Summary for official submission (func 0): correctness=1 misses=1184

TEST_TRANS_RESULTS=1:1184
a10235101477@10235101477: /lab4/cache$ ./csim-ref -v -s 5 -E 1 -b 5 -t trace.F0
S 18C8B0,1 miss
L 18C8B0,8 miss
L 18C8B0,4 miss
L 18C8B0,4 hit
L 18C8B0,8 miss eviction
L 18C8B0,8 miss eviction
L 18C8B0,4 miss eviction
S 18C120,4 miss
L 18C8B0,4 hit
S 18C1A0,4 miss

```

数组首地址相同(上面那几个18几几的是本地变量，放到寄存器之后就不再出现)，故对于数组同一个位置的数，转置后数组与原数组都将映射到同一个set，故可以画出下面数组的部分set对应表：



(上面的数字代表映射到的组位置，由于一个块32字节，能存8个int，此处一个数表示数组中该位置的8个数)，可以看出映射规律是每8行一个循环，不过当时并没有看出什么头绪，就随便试了一些转置方法，写了一个先转后24列，再转前8列的代码跑了试一下：

代码：

```
int i, j, k, tmp;
```

```

for (i = 0; i < N; i++) {
    for (j = 8; j < M; j++) {
        tmp = A[i][j];
        B[j][i] = tmp;
    }
}

for (i = 0; i < N; i++) {
    for (j = 0; j < 8; j++) {

```

```

        tmp = A[i][j];
        B[j][i] = tmp;
    }
}

```

结果:

```

#10235101477@10235101477: /code/0005 $ make
gcc -g -Wall -Werror -std=c99 -m64 -O0 -c trans.c
gcc -g -Wall -Werror -std=c99 -m64 -O0 -o test-trans test-trans.c cachelab.c trans.o
gcc -g -Wall -Werror -std=c99 -m64 -O0 -o tracegen tracegen.c trans.o cachelab.c
# Generate a handin tar file each time you compile
tar -cvf ai0235101477-handin.tar csln.c trans.c
csln.c
trans.c
#10235101477@10235101477: /code/0005 $ ./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:1079, misses:974, evictions:942

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

Summary for official submission (func 0): correctness=1 misses=974

***
TEST_TRANS_RESULTS=1:974
#10235101477@10235101477: /code/0005 $

```

结果确实少了一点,但是离要求(300以下)还差得远,根据writeup里面的提示,我们要用分块的技术进行优化,这里由于数组每8行映射的set相同,此时就考虑按8*8进行分块(每8行set相同,且每个块正好存8个int,如果多了(如16*16),set和块都装不下(原数组的列就是目标数组的行),少了, set和块都有剩余),可以打出如下代码:(这里由于两个数组的映射规律相同,对于每个set都会有冲突不命中的现象(对角线上的元素映射的位置相同),不过暂时不管,先看一下能不能进300)

```

int i, j, k, y, tmp;
for (i = 0; i < N; i += 8)
    for (j = 0; j < M; j += 8)
        for (k = i; k < i + 8; k++)
            for (y = j; y < j + 8; y++) {
                tmp = A[k][y];
                B[y][k] = tmp;
            }

```

跑分结果:

```

gcc -g -Wall -Werror -std=c99 -m64 -O0 -c trans.c
gcc -g -Wall -Werror -std=c99 -m64 -O0 -o test-trans test-trans.c cachelab.c trans.o
gcc -g -Wall -Werror -std=c99 -m64 -O0 -o tracegen tracegen.c trans.o cachelab.c
# Generate a handin tar file each time you compile
tar -cvf ai0235101477-handin.tar csln.c trans.c
csln.c
trans.c
#10235101477@10235101477: /code/0005 $ ./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:1709, misses:344, evictions:312

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

Summary for official submission (func 0): correctness=1 misses=344

TEST_TRANS_RESULTS=1:344
#10235101477@10235101477: /code/0005 $

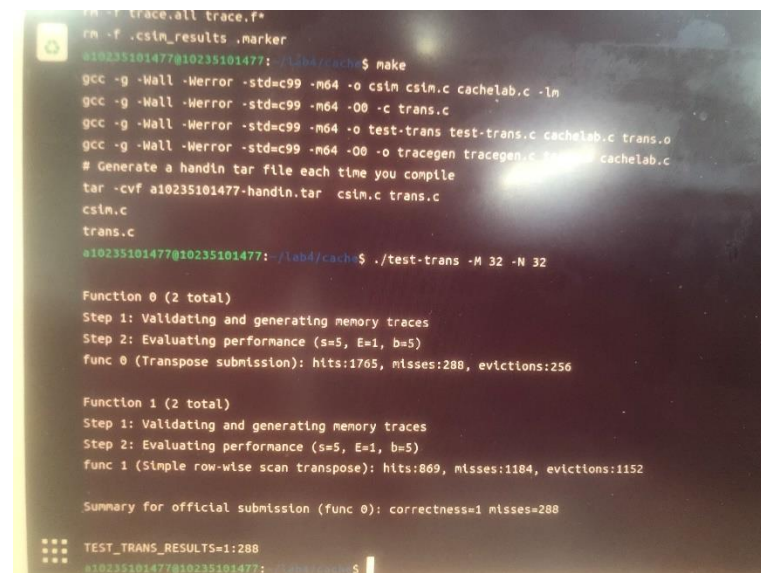
```

可以看出不命中数(344次)下降了很多,但是仍没到300以内,故不得不想办法处理对角线上的元素,由于writeup中允许我们使用至多12个本地变量,因此在转置时我们可以将8个变量先从原数组中取出(即一行中8个变量一起转置),再赋值到目标数组(这样还可以少写一个for循环),这样,即

使是对角线的情况，也不用担心刚拿了一个出来就被驱逐了的现象，降低不命中数：

```
int i, j, k;
int a, b, c, d, e, f, g, h;
for (i = 0; i < N; i += 8)
    for (j = 0; j < M; j += 8)
        for (k = i; k < i + 8; k++)
        {
            a = A[k][j];
            b = A[k][j + 1];
            c = A[k][j + 2];
            d = A[k][j + 3];
            e = A[k][j + 4];
            f = A[k][j + 5];
            g = A[k][j + 6];
            h = A[k][j + 7];
            B[j][k] = a;
            B[j + 1][k] = b;
            B[j + 2][k] = c;
            B[j + 3][k] = d;
            B[j + 4][k] = e;
            B[j + 5][k] = f;
            B[j + 6][k] = g;
            B[j + 7][k] = h;
        }
```

跑分结果：



```
rm -f trace.all trace.*
rm -f .csim_results .marker
a10235101477@10235101477: /lab4/cache$ make
gcc -g -Wall -Werror -std=c99 -m64 -o csim csim.c cachelab.c -lm
gcc -g -Wall -Werror -std=c99 -m64 -O0 -c trans.c
gcc -g -Wall -Werror -std=c99 -m64 -o test-trans test-trans.c cachelab.c trans.o
gcc -g -Wall -Werror -std=c99 -m64 -O0 -o tracegen tracegen.c cachelab.c
# Generate a handin tar file each time you compile
tar -cvf a10235101477-handin.tar csim.c trans.c
csim.c
trans.c
a10235101477@10235101477: /lab4/cache$ ./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:1765, misses:288, evictions:256

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

Summary for official submission (func 0): correctness=1 misses=288

*** TEST_TRANS_RESULTS=1:288
a10235101477@10235101477: /lab4/cache$
```

288次，成功！

接下来我们看64*64的（不命中数需小于1300）：

同样像上面一样画出set的映射图：

8个数

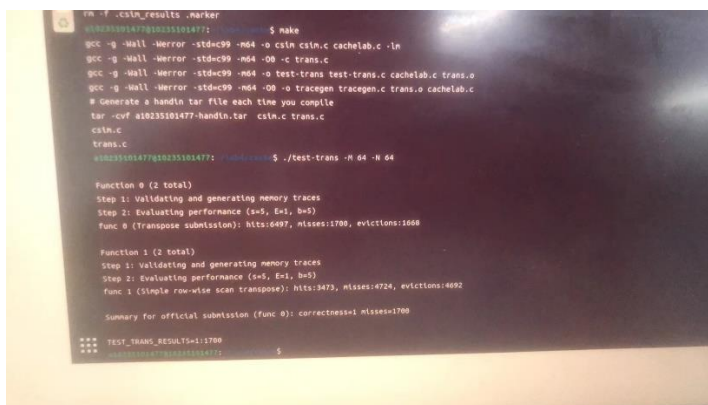
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31

行

由于这次是每4行set重复一遍，我们试着4*4分块：

```
int i, j, k;
int a, b, c, d;
for (i = 0; i < 64; i += 4)
    for (j = 0; j < 64; j += 4)
    {
        for (k = i; k < i + 4; k++)
        {
            a = A[k][j];
            b = A[k][j + 1];
            c = A[k][j + 2];
            d = A[k][j + 3];
            B[j][k] = a;
            B[j + 1][k] = b;
            B[j + 2][k] = c;
            B[j + 3][k] = d;
        }
    }
```

跑分结果：



1700次，还不错，不过不够，由于是4*4分块，原数组行的缓存并没有全部用上，故我又试图写一个4*8的版本：

```
int i, j, k;
int a, b, c, d, e, f, g, h;
for (i = 0; i < 64; i += 4)
```

```

for (j = 0; j < 64; j += 8)
{
    for (k = i; k < i + 4; k++)
    {
        a = A[k][j];
        b = A[k][j + 1];
        c = A[k][j + 2];
        d = A[k][j + 3];
        e = A[k][j + 4];
        f = A[k][j + 5];
        g = A[k][j + 6];
        h = A[k][j + 7];
        B[j][k] = a;
        B[j + 1][k] = b;
        B[j + 2][k] = c;
        B[j + 3][k] = d;
        B[j + 4][k] = e;
        B[j + 5][k] = f;
        B[j + 6][k] = g;
        B[j + 7][k] = h;
    }
}

```

跑分结果：

```

$ make
gcc -g -Wall -Werror -std=c99 -m64 -o csln csln.c cachelab.c -lm
gcc -g -Wall -Werror -std=c99 -m64 -O0 -c trans.c
gcc -g -Wall -Werror -std=c99 -m64 -O0 -c test-trans test-trans.c cachelab.c trans.o
gcc -g -Wall -Werror -std=c99 -m64 -O0 -o tracegen tracegen.c trans.o cachelab.c
# Generate a handin tar file each time you compile
tar -cvf ai0235101477-handin.tar csln.c trans.c
csln.c
trans.c
ai0235101477@10235101477:~/cacheLab$ ./test-trans -H 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:3585, misses:4612, evictions:4589

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:3473, misses:4724, evictions:4692

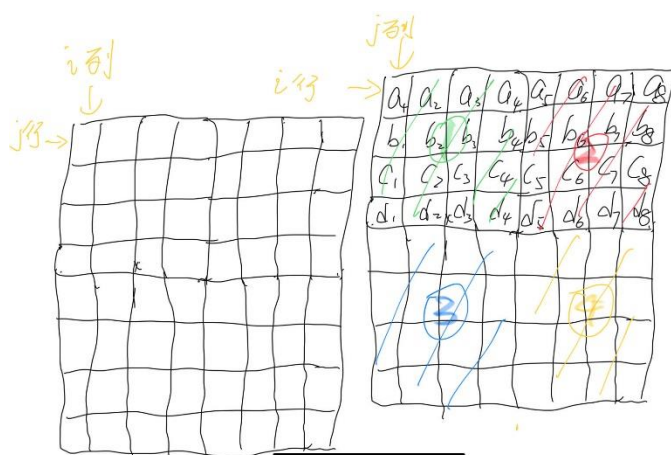
Summary for official submission (func 0): correctness=1 misses=4612

*** TEST_TRANS_RESULTS=1:4612
*** ai0235101477@10235101477:~/cacheLab$

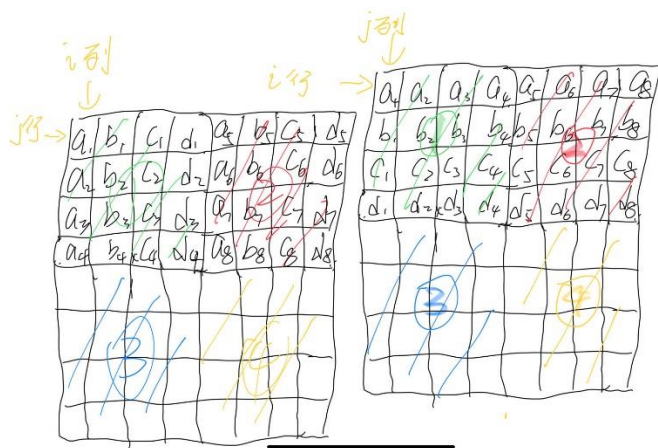
```

反而更多了。。。分析原因，发现这样其实与 8×8 一致，当转置到目标数组时，由于每4行的set是相同的，如果一次转8个，目标数组的set一定会冲突(目标数组的行是原数组的列，转8个下来，会导致目标数组的set交替冲突)，故要另寻他法(笔者此时已山穷水尽，下面的解法参考了网络上的做法)：

还是 8×8 的转化方法，只不过我们用特殊一点的转置方式：



如图所示，右边代表转置前的矩阵，左边代表转置后的矩阵，我们将这个8*8的矩阵分为4*4的4个部分，在进行转置时，我们将绿色的1块正常转置，同时利用B块缓存中尚未利用的部分(即把2的部分转置到新的2的部分(如下图)，这样cache的缓存可以用到极致)：



但是由于右边矩阵2的部分应转置到左边矩阵3的部分，接下来我们将右边矩阵2的部分挪到3的部分，同时将右边矩阵3的部分转置到左边矩阵的2部分，对于这一步，由于2的这部分其内部已转置成功，我们再将左边矩阵2的部分移到3的部分时，可以以行为单位进行转置，这样我们可以利用到B的前四行缓存尚未被驱逐的性质，使得左边矩阵的2的部分全部命中(利用4个本地变量)，同时我们再将右边矩阵3的部分进行转置(1, 2的部分已经转置完了，因此可以处理3, 4了)，最后再对4的部分进行转置即可，代码如下：

```
int i, j, k;
int a, b, c, d, e, f, g, h;
for (i = 0; i < 64; i += 8)
    for (j = 0; j < 64; j += 8)
    {
        for (k = i; k < i + 4; k++)
        {
            a = A[k][j];
            b = A[k][j + 1];
            c = A[k][j + 2];
            d = A[k][j + 3];
            e = A[k][j + 4];
            f = A[k][j + 5];
```

```

        g = A[k][j + 6];
        h = A[k][j + 7];
        B[j][k] = a;
        B[j + 1][k] = b;
        B[j + 2][k] = c;
        B[j + 3][k] = d;
        B[j][k + 4] = e;
        B[j + 1][k + 4] = f;
        B[j + 2][k + 4] = g;
        B[j + 3][k + 4] = h;
    }
    for (k = j; k < j + 4; k++)
    {
        a = B[k][i + 4];
        b = B[k][i + 5];
        c = B[k][i + 6];
        d = B[k][i + 7];
        e = A[i + 4][k];
        f = A[i + 5][k];
        g = A[i + 6][k];
        h = A[i + 7][k];
        B[k][i + 4] = e;
        B[k][i + 5] = f;
        B[k][i + 6] = g;
        B[k][i + 7] = h;
        B[k + 4][i] = a;
        B[k + 4][i + 1] = b;
        B[k + 4][i + 2] = c;
        B[k + 4][i + 3] = d;
    }
    for (k = i + 4; k < i + 8; k++)
    {
        a = A[k][j + 4];
        b = A[k][j + 5];
        c = A[k][j + 6];
        d = A[k][j + 7];
        B[j + 4][k] = a;
        B[j + 5][k] = b;
        B[j + 6][k] = c;
        B[j + 7][k] = d;
    }
}

```

跑分结果:

```
gcc -g -Wall -Werror -std=c99 -m64 -o csm csm.c cachelab.c -lm
gcc -g -Wall -Werror -std=c99 -m64 -O0 -c trans.c
gcc -g -Wall -Werror -std=c99 -m64 -o test-trans test-trans.c cachelab.c trans.o
gcc -g -Wall -Werror -std=c99 -m64 -O0 -o tracegen tracegen.c trans.o cachelab.c
# Generate a handin tar file each time you compile
tar -cvf a10235101477-handin.tar csm.c trans.c
csm.c
trans.c
a10235101477@a10235101477:~/lab4/cache$ ./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:9017, misses:1228, evictions:1190

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:3473, misses:4724, evictions:4692

Summary for official submission (func 0): correctness=1 misses=1228

TEST_TRANS_RESULTS=1:1228
a10235101477@a10235101477:~/lab4/cache$
```

1228次，通过！

接下来我们看61*67的(不命中数需小于2000)：

该矩阵的set映射图及其不规则(每一行都会有“落单的”元素，有些行开头的元素和上一行末尾的元素映射到了同一个组)，因此，我们先尝试一些规律的分块方法，看一下不命中数如何(显然不命中数会比64*64的要大)，先是8*8：

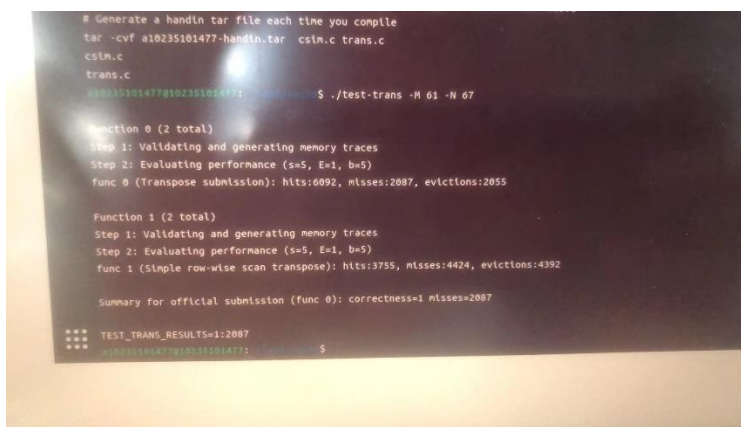
```
int i, j, k;
int a, b, c, d, e, f, g, h;
for (i = 0; i + 8 < 67; i += 8) {
    for (j = 0; j + 8 < 61; j += 8)
    {
        for (k = i; k < i + 8; k++) {
            a = A[k][j];
            b = A[k][j + 1];
            c = A[k][j + 2];
            d = A[k][j + 3];
            e = A[k][j + 4];
            f = A[k][j + 5];
            g = A[k][j + 6];
            h = A[k][j + 7];
            B[j][k] = a;
            B[j + 1][k] = b;
            B[j + 2][k] = c;
            B[j + 3][k] = d;
            B[j + 4][k] = e;
            B[j + 5][k] = f;
            B[j + 6][k] = g;
            B[j + 7][k] = h;
        }
    }
}
```

```

}
for (i = 64; i < 67; i++) {
    for (j = 0; j < 61; j++)
        B[j][i] = A[i][j];
}
for (int y = 0; y < 64; y++) {
    for (j = 56; j < 61; j++)
    {
        B[j][y] = A[y][j];
    }
}
}

```

跑分结果:



2087次，差一点，试一下12*12的:

```

int i, j, k;
int a, b, c, d, e, f, g, h;
for (i = 0; i + 12 < 67; i += 12) {
    for (j = 0; j + 12 < 61; j += 12)
    {
        for (k = i; k < i + 12; k++) {
            a = A[k][j];
            b = A[k][j + 1];
            c = A[k][j + 2];
            d = A[k][j + 3];
            e = A[k][j + 4];
            f = A[k][j + 5];
            g = A[k][j + 6];
            h = A[k][j + 7];
            B[j][k] = a;
            B[j + 1][k] = b;
            B[j + 2][k] = c;
            B[j + 3][k] = d;
            B[j + 4][k] = e;
            B[j + 5][k] = f;

```

```

        B[j + 6][k] = g;
        B[j + 7][k] = h;
        a = A[k][j + 8];
        b = A[k][j + 9];
        c = A[k][j + 10];
        d = A[k][j + 11];
        B[j + 8][k] = a;
        B[j + 9][k] = b;
        B[j + 10][k] = c;
        B[j + 11][k] = d;
    }
}
}
for (i = 60; i < 67; i++) {
    for (j = 0; j < 61; j++)
        B[j][i] = A[i][j];
}
for (int y = 0; y < 64; y++) {
    for (j = 60; j < 61; j++)
    {
        B[j][y] = A[y][j];
    }
}
}

```

跑分结果:

```

gcc -g -Wall -Werror -std=c99 -m64 -O0 -c trans.c
gcc -g -Wall -Werror -std=c99 -m64 -O test-trans test-trans.c cachelab.c trans.o
gcc -g -Wall -Werror -std=c99 -m64 -O0 -o tracegen tracegen.c trans.o cachelab.c
# Generate a handin tar file each time you compile
tar -cvf a102351014777102351014771 tar csln.c trans.c
csln.c
trans.c
a102351014777102351014771:~/CS212$ ./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:11, misses:2276, evictions:2244

Function 1 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
Func 1 (Single row-wise scan transpose): hits:3755, misses:4424, evictions:4392

Summary for official submission (Func 0): correctness=1 misses=2276

*** TEST_TRANS_RESULTS=1:2276
*** a102351014777102351014771:~/CS212$

```

2276次。。。更多了，再试一下16*16:

```

for (i = 0; i + 16 < 67; i += 16) {
    for (j = 0; j + 16 < 61; j += 16)
    {
        for (k = i; k < i + 16; k++) {
            a = A[k][j];
            b = A[k][j + 1];
            c = A[k][j + 2];
            d = A[k][j + 3];
            e = A[k][j + 4];

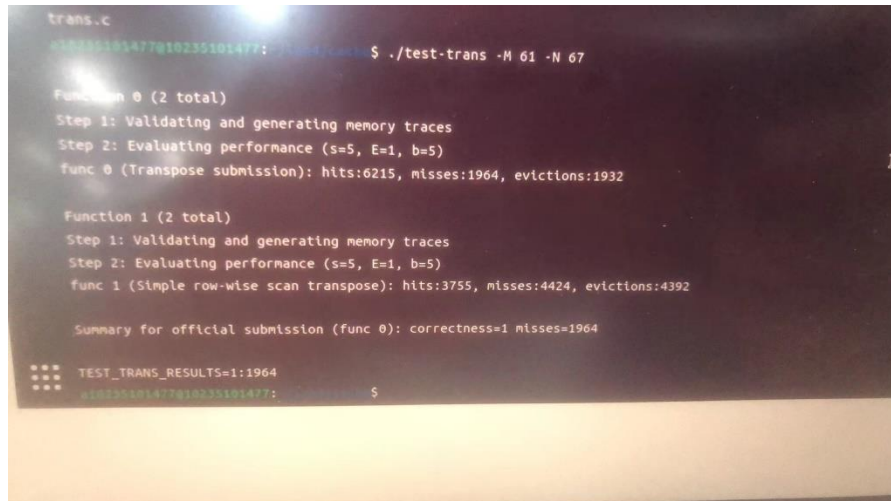
```

```

        f = A[k][j + 5];
        g = A[k][j + 6];
        h = A[k][j + 7];
        B[j][k] = a;
        B[j + 1][k] = b;
        B[j + 2][k] = c;
        B[j + 3][k] = d;
        B[j + 4][k] = e;
        B[j + 5][k] = f;
        B[j + 6][k] = g;
        B[j + 7][k] = h;
        a = A[k][j + 8];
        b = A[k][j + 9];
        c = A[k][j + 10];
        d = A[k][j + 11];
        e = A[k][j + 12];
        f = A[k][j + 13];
        g = A[k][j + 14];
        h = A[k][j + 15];
        B[j + 8][k] = a;
        B[j + 9][k] = b;
        B[j + 10][k] = c;
        B[j + 11][k] = d;
        B[j + 12][k] = e;
        B[j + 13][k] = f;
        B[j + 14][k] = g;
        B[j + 15][k] = h;
    }
}
}
for (i = 64; i < 67; i++) {
    for (j = 0; j < 61; j++)
        B[j][i] = A[i][j];
}
for (int y = 0; y < 64; y++) {
    for (j = 48; j < 61; j++)
    {
        B[j][y] = A[y][j];
    }
}
}

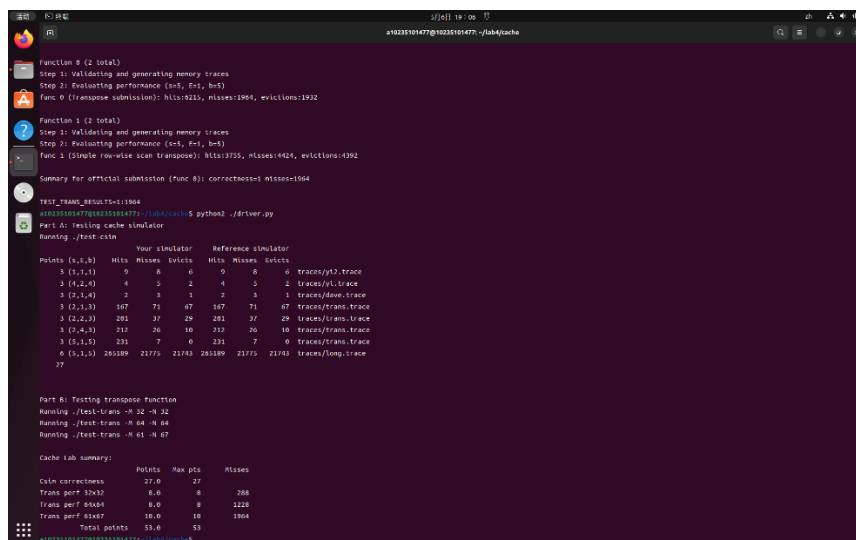
```

跑分结果:



这次降到2000以内了，16*16的代码即可作为答案。

下面是两个部分完整跑分截图：



后记：

每次做lab都让我感到人脑的强大。。。这次lab的64*64的矩阵转置让我去想的话这辈子都想不出这样的解法。。。这解法将cache的空间局部性用到了极致。。。。