

实验要求

- 第六章：存储器层次结构
 - 基本的存储技术，并描述它们是如何被组织成层次结构的
 - 详细介绍高速缓存存储器，它对应用程序性能的影响最大
 - 分析和改进C程序的局部性
- 实验要求
 - Part(A)要求实现一个缓存模拟器，根据输入参数 s, E, b 来创建指定规格的cache，同时提供了不同的文件，每个文件都指定了对cache的一系列操作，要求计算特定规格的cache对特定文件的Hit数、Miss数和Eviction数
 - Part(B)要求对矩阵转置进行优化，给定了cache的规格，对于不同规格的矩阵，制定不同的访问策略，使得矩阵转置时尽可能少的对cache产生Miss数

实验要求

Part(A)

Part(B)

32*32

64*64

思考8*8分块

思考4*4分块

提高数组B的命中

61*67

总结

Part(A)

利用 `getopt` 函数解析命令行中的参数，将 s 换算成 S 组，创建二维数组 `cache[S][E]`，其中的元素是结构体 `cache_line`，每个结构体包含 `valid_bit`，`tag` 和 `stamp`，并进行初始化

逐行读取文件中的数据， L 和 S 都只访问一次缓存，而 M 访问两次缓存，记得结束时要利用 `free` 收回空间，并关闭打开的文件

访问缓存会遇到三种情况：命中、未命中、未命中并置换。每次操作完都要将操作的行的时间戳置为0，接着对有效位为1的所有行更新时间，因此所有有效的行的时间戳最小为1，而无效的行的时间戳全部为0

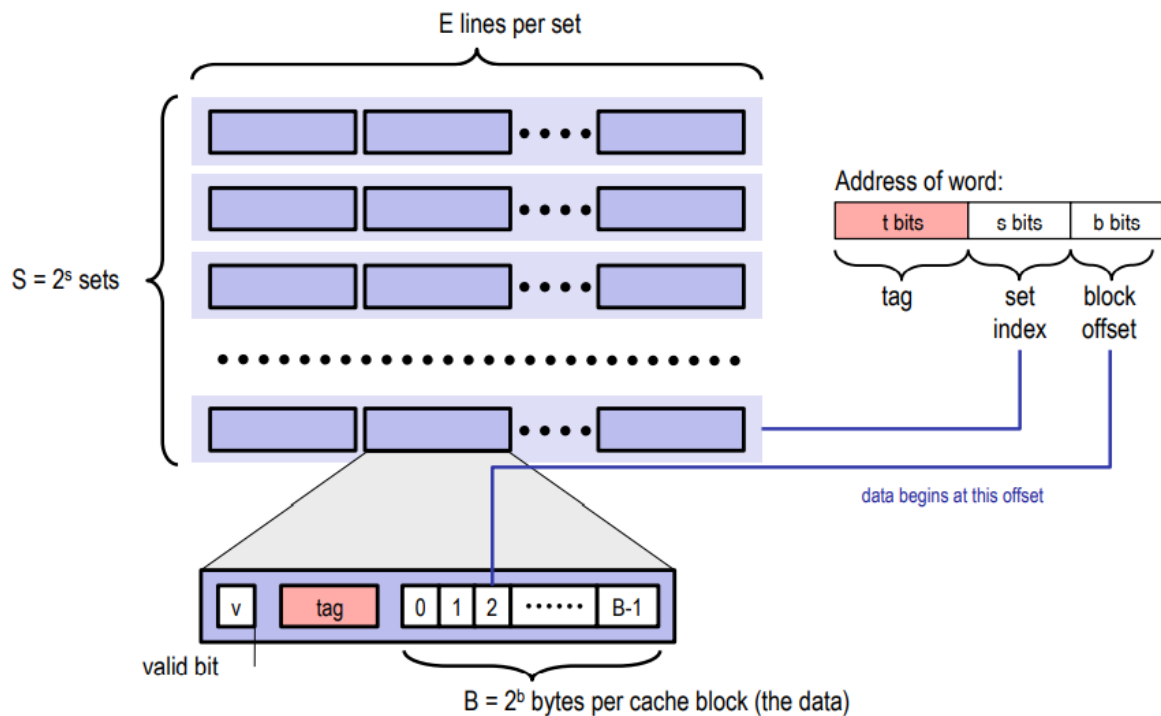
命令行包含 `-v` 参数时，需要对操作结果进行统计，操作结果可按以下分类：

- L (加载)： 命中，未命中，未命中置换
- S (存储)： 命中，未命中，未命中置换
- M (先加载再存储)： 命中命中，未命中命中，未命中置换命中 （第二次一定命中）

- 地址结构

- 地址是十六进制表示
- 地址占32位

Visual Cache Terminology



- 策略
 - 块(Block)偏移没有被使用，因此不考虑**b**
 - 因为每次在内存和缓存之间进行传输的都是一**整块**：如果块存在的话，一定命中；如果块不存在的话，会将一整块都调入缓存，以后即使是不同的地址，只要都属于同一块，则一定会命中
 - 采用最近未被使用策略 LRU

Part (a) : Cache simulator

- **A cache simulator is NOT a cache!**
 - Memory contents NOT stored
 - Block offsets are NOT used – the b bits in your address don't matter.
 - Simply **count** hits, misses, and evictions
- **Your cache simulator needs to work for different s, b, E, given at run time.**
- **Use LRU – Least Recently Used replacement policy**
 - Evict the least recently used block from the cache to make room for the next block.
 - Queues ? Time Stamps ?

- 变量名称

Part (a) : Hints

- **A cache is just 2D array of *cache lines*:**
 - `struct cache_line cache[S][E];`
 - $S = 2^s$, is the number of sets
 - E is associativity
- **Each cache_line has:**
 - Valid bit
 - Tag
 - LRU counter (only if you are not using a queue)

```
#include "cachelab.h"

#include<getopt.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

typedef struct{
```

```

    int valid_bit, tag, stamp;
}cache_line;           //每一行的结构体，有效位、tag地址、时间戳

cache_line** cache = NULL; //二维数组
char buffer[20];          //读取每一行的字符串时，使用的缓冲区
int hits=0;               //命中次数
int misses=0;             //未命中次数
int evictions=0;          //置换次数
int v=0;
int s,E,b;               //分别是缓存组号所占的位数、每组的行数、块偏移（暂时未用到）
int S;                   //缓存有多少组
char filename[30];        //保存要打开的文件名
void updateTime();        //先声明，后定义

//返回H, M, E, 分别表示命中、未命中、未命中置换
char update(unsigned address)
{
    //定义为无符号数的-1，将其左移会用0来补位
    int s_address = (address >> b) & ((-1U) >> (32-s));
    int t_address = address >> (b+s);

    for(int i=0;i<E;i++)    //命中
    {
        if(cache[s_address][i].tag==t_address && cache[s_address][i].valid_bit==1)
        {
            hits++;
            cache[s_address][i].stamp=0;
            updateTime();
            return 'H';
        }
    }

    for(int i=0;i<E;i++)    //未命中
    {
        if(cache[s_address][i].valid_bit==0)
        {
            misses++;
            cache[s_address][i].valid_bit=1;
            cache[s_address][i].tag=t_address;
            cache[s_address][i].stamp=0;
            updateTime();
            return 'M';
        }
    }

    int maxTime=0; int evictE=0;    //根据时间戳，利用LRU进行置换
    for(int i=0;i<E;i++)           //未命中且发生置换
    {
        //查找最大的时间戳，并记录其所在的行号
        if(cache[s_address][i].stamp > maxTime)
        {
            maxTime = cache[s_address][i].stamp;
            evictE=i;
        }
    }
    misses++;
    evictions++;
}

```

```

        cache[s_address][evictE].valid_bit=1;
        cache[s_address][evictE].tag=t_address;
        cache[s_address][evictE].stamp=0;
        updateTime();
        return 'E';
    }

void updateTime()    //更新时间戳，将有效位为1的行，stamp++
{
    for(int i=0;i<S;i++)
    {
        for(int j=0;j<E;j++)
        {
            if(cache[i][j].valid_bit==1)
                cache[i][j].stamp++;
        }
    }
    return;
}

//测试-v参数的统计量是否正确
int testHits=0;           //测试命中次数
int testMisses=0;        //测试未命中次数
int testEvictions=0;     //测试置换次数
//命令行中包含参数-v时使用printw
//identifier: L加载, S存储, M先加载再存储; result: H命中, M未命中, E未命中置换
void printv(char identifier, unsigned address, int size, char result)
{
    if(identifier=='L' || identifier=='S')    //命中, 未命中, 未命中置换
    {
        switch(result)
        {
            case 'H':
                printf("%c %x,%d : Hits\n", identifier, address, size);
                testHits++;
                break;
            case 'M':
                printf("%c %x,%d : Misses\n", identifier, address, size);
                testMisses++;
                break;
            case 'E':
                printf("%c %x,%d : Misses,Evictions\n", identifier, address,
size);
                testMisses++;
                testEvictions++;
                break;
        }
    }
    else    //命中命中, 未命中命中, 未命中置换命中(第二次一定命中)
    {
        switch(result)
        {
            case 'H':
                printf("%c %x,%d : Hits,Hits\n", identifier, address, size);
                testHits+=2;
                break;
            case 'M':
                printf("%c %x,%d : Misses,Hits\n", identifier, address, size);

```

```

        testMisses++;
        testHits++;
        break;
    case 'E':
        printf("%c %x,%d : Misses,Evictions,Hits\n", identifier,
address, size);
        testMisses++;
        testEvictions++;
        testHits++;
        break;
    }
}
return;
}

int main(int argc, char** argv)
{
    int opt;
    while(-1!=(opt=getopt(argc, argv, "hvs:E:b:t:")))
    {
        switch(opt)
        {
            case 'h':
                break;
            case 'v':
                v=1;
                break;
            case 's':
                s=atoi(optarg);
                break;
            case 'E':
                E=atoi(optarg);
                break;
            case 'b':
                b=atoi(optarg);
                break;
            case 't':
                strcpy(filename,optarg);    //字符串拷贝函数
                break;
            default:
                printf("wrong argument\n");
                break;
        }
    }

    S = (1<s);    //s位可以表示缓存共有S组
    cache = (cache_line**)malloc(sizeof(cache_line*)*S);    //创建二维数组
    for(int i=0;i<S;i++)
    {
        cache[i]=(cache_line*)malloc(sizeof(cache_line)*E);
    }
    for(int i=0;i<S;i++)    //二维数组初始化
    {
        for(int j=0;j<E;j++)
        {
            cache[i][j].valid_bit=0;
            cache[i][j].tag=0;
            cache[i][j].stamp=0;

```

```

    }
}

FILE* fp = fopen(filename, "r");    //fp必须被使用，否则会产生warnings，所有
warnings都被看做errors
if(fp == NULL)
{
    printf("The File is wrong!\n");
    exit(-1);
}

char identifier;    //分别记录操作符号，地址，所需要的字节数（暂时无用）
unsigned address;
int size;
char result;    //result保存对地址的操作结果：命中H，未命中M，未命中置换E
while(fgets(buffer, 20, fp))    //处理打开文件的每一行的操作，将每一行读入到buffer字符数
组中
{
    //scanf标准输入，fscanf文件输入，sscanf指定字符串输入
    //文件的每一行的第一个字符为空
    sscanf(buffer+1, "%c %x,%d", &identifier, &address, &size);

    switch(identifier)
    {
        case 'L':    //L和S都只访问一次缓存
        case 'S':
            result=update(address);
            if(v)
                printV(identifier, address, size, result);
            break;
        case 'M':    //M访问两次缓存，相当于先L后S
            result=update(address);
            update(address);
            if(v)
                printV(identifier, address, size, result);
            break;
    }
}

fclose(fp);    //关闭文件
for(int i=0; i<S; i++)    //回收空间
    free(cache[i]);
free(cache);

//输出-v参数的测试统计量
if(v)
    printf("testHits:%d testMisses:%d
testEvictions:%d\n", testHits, testMisses, testEvictions);

printSummary(hits, misses, evictions);
return 0;
}

```

```

root@ubuntu:/home/csapp/Desktop/Lab4/cachelab-handout# ./test-csim
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
root@ubuntu:/home/csapp/Desktop/Lab4/cachelab-handout#

```

Part(B)

注意点

- $s = 5, E = 1, b = 5$
- 最多使用12个局部变量
- 不准修改A, 但可以修改B
- 重点考虑**对角线**

在 `transpose_submit()` 函数中定义以下12个局部变量

```

int x1,x2,x3,x4,x5,x6,x7,x8;
int i,j,x,y;

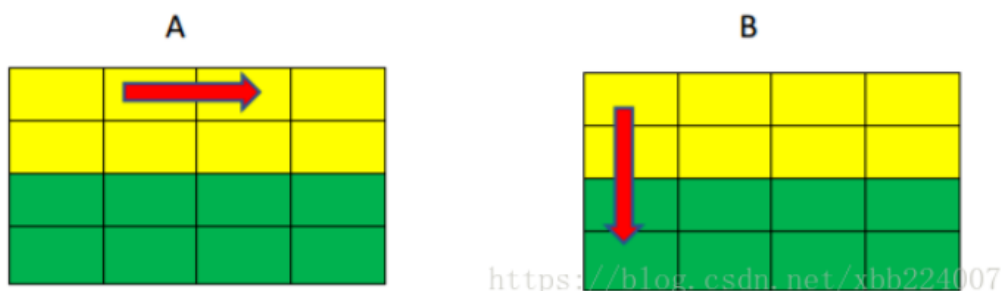
```

用4*4的小例子测试一下

```

./test-trans -M 4 -N 4
vim trace.f1
./csim -v -s 5 -E 1 -b 5 -t trace.f1

```



A中的黄绿两部分, 分别占缓存中不同的两个块

B中的黄绿两部分, 分别占缓存中不同的两个块

A和B中的黄色部分会映射到同一组, 而每一组只有一行, 因此一定会**交替置换**, 绿色部分同理


```

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

```

查看结果

```

L 602100,4 : Misses,Evictions A[0][0] //01000 00000 A黄
S 642100,4 : Misses,Evictions B[0][0] //01000 00000 B黄
L 602104,4 : Misses,Evictions A[0][1]
S 642110,4 : Misses,Evictions B[1][0]
L 602108,4 : Misses,Evictions A[0][2] //前5行 A黄和B黄交替置换，最后缓存中是A黄
S 642120,4 : Misses B[2][0] //01001 00000 B绿
L 60210c,4 : Hits A[0][3]
S 642130,4 : Hits B[3][0]
L 602110,4 : Hits A[1][0]
S 642104,4 : Misses,Evictions B[0][1] //B黄
L 602114,4 : Misses,Evictions A[1][1]
S 642114,4 : Misses,Evictions B[1][1]
L 602118,4 : Misses,Evictions A[1][2] //A黄，之间四行A黄和B黄交替置换
S 642124,4 : Hits B[2][1]
L 60211c,4 : Hits A[1][3]
S 642134,4 : Hits B[3][1]
L 602120,4 : Misses,Evictions A[2][0] //01001 00000 A绿
S 642108,4 : Misses,Evictions B[0][2] //后面同理，略
L 602124,4 : Hits A[2][1]
S 642118,4 : Hits B[1][2]
L 602128,4 : Hits A[2][2]
S 642128,4 : Misses,Evictions B[2][2]
L 60212c,4 : Misses,Evictions A[2][3]
S 642138,4 : Misses,Evictions B[3][2]
L 602130,4 : Misses,Evictions A[3][0]
S 64210c,4 : Hits B[0][3]
L 602134,4 : Hits A[3][1]
S 64211c,4 : Hits B[1][3]
L 602138,4 : Hits A[3][2]
S 64212c,4 : Misses,Evictions B[2][3]
L 60213c,4 : Misses,Evictions A[3][3]
S 64213c,4 : Misses,Evictions B[3][3]

```

```

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:15, misses:22, evictions:19

```

miss过多的原因是在访问两个数组的过程中存在太多的冲突不命中，而造成传统不命中的原因是B数组与A数组中下标相同的元素会映射到同一个cache块，如上述的1~5步骤，就是不断地发生了冲突不命中

- 优化：将其分成2*4的小块，每次读取8个元素放在局部变量中，之后**即使发生替换，被替换的元素也不会再通过读取缓存访问了**
- 猜测：从汇编角度，局部变量保存在寄存器中

```

if(N==4 && M==4)
{
    for(i=0;i<4;i+=2)        //分成2*4的小块
        for(j=0;j<4;j+=4)
        {
            x1=A[i][j];x2=A[i][j+1];x3=A[i][j+2];x4=A[i][j+3];
            x5=A[i+1][j];x6=A[i+1][j+1];x7=A[i+1][j+2];x8=A[i+1][j+3];

            B[j][i]=x1;B[j+1][i]=x2;B[j+2][i]=x3;B[j+3][i]=x4;
            B[j][i+1]=x5;B[j+1][i+1]=x6;B[j+2][i+1]=x7;B[j+3][i+1]=x8;
        }
}

```

```

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:29, misses:8, evictions:6

```

正常只有5个misses，因为有额外操作

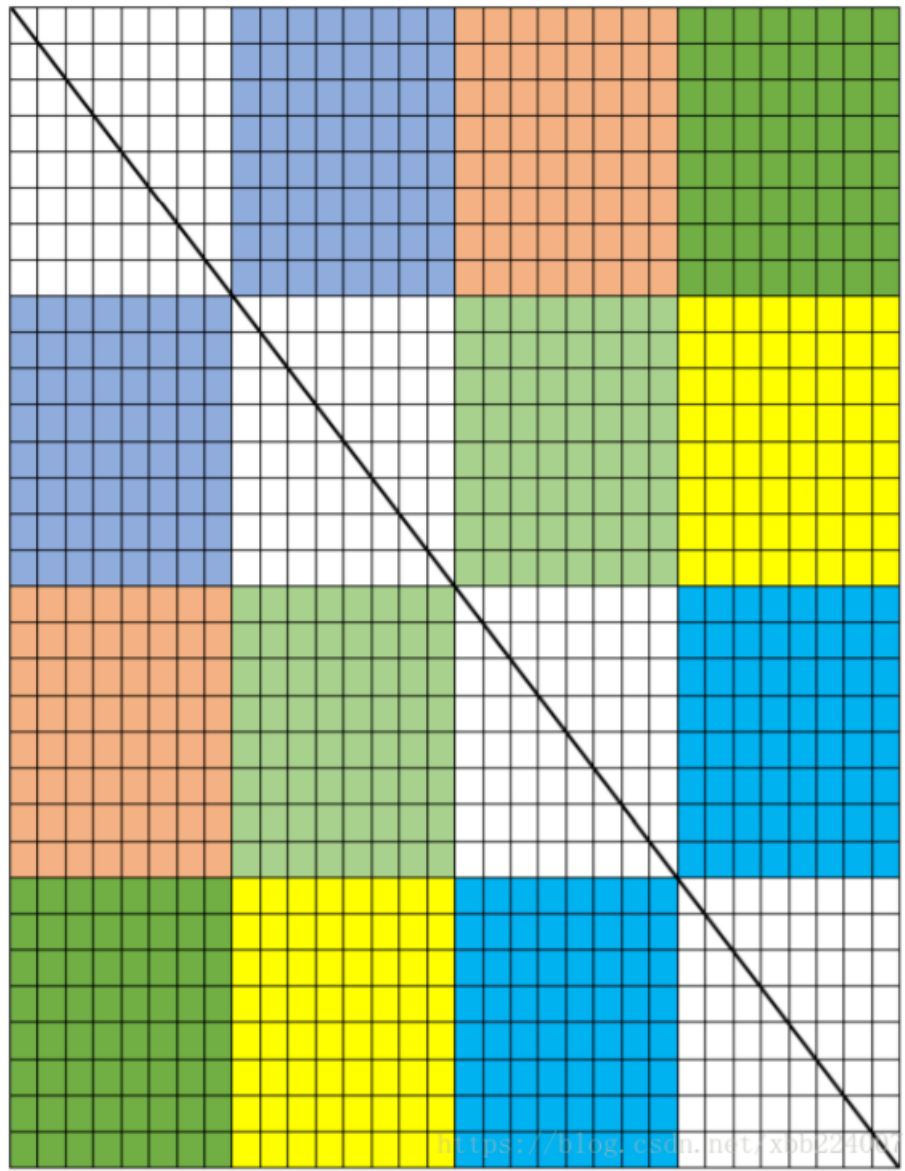
32*32

每个块可以放8个int，每组一块，共有32组，因此整个32*32矩阵的前8行就可以完全填满cache，同时每一块都不会发生冲突

将矩阵进行8*8进行分块

同一列中，相距8行的元素会映射到cache中的同一组发生冲突，因此不在对角线上、转置时两两对应的8*8块是不会发生冲突的

而在对角线上的元素，则会像上文中4*4分块演示那样发生交替置换的现象



- 完全未优化 (未分块)

```
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
```

- 8*8分块 (未处理对角线分块)

```
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:1710, misses:343, evictions:311
```

- 8*8分块, 同时优化了对角线分块

```
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
```

```

if(N==32 && M==32)
{
    for(i=0;i<N;i+=8)          //分成8*8的小块
        for(j=0;j<M;j+=8)
            for(x=i;x<i+8;x++) //逐行访问小块
                {
                    if(i==j)    //对角线分块，将每行的8个元素同时处理，共有8行
                    {
                        x1=A[x][j];x2=A[x][j+1];x3=A[x][j+2];x4=A[x][j+3];
                        x5=A[x][j+4];x6=A[x][j+5];x7=A[x][j+6];x8=A[x][j+7];

                        B[j][x]=x1;B[j+1][x]=x2;B[j+2][x]=x3;B[j+3][x]=x4;
                        B[j+4][x]=x5;B[j+5][x]=x6;B[j+6][x]=x7;B[j+7][x]=x8;
                    }
                    else        //非对角线分块（不会因为缓存冲突而发生置换）
                    {
                        for(y=j;y<j+8;y++)
                            B[y][x] = A[x][y];
                    }
                }
}

```

计算Misses数

- 非对角线上的块
 - A组：12块，每块64个元素，每行8个元素，每访问一行产生一次Miss，因此每个元素的Miss率为八分之一

$$12 * 64 * (1/8) = 96$$
 - B组：同理为 96
- 对角线上的块
 - A组：4块，每访问一行产生一次Miss，每个元素的Miss率为八分之一

$$4 * 64 * (1/8) = 32$$
 - B组：4块，每访问一行产生两次Miss，每个元素的Miss率为四分之一

$$4 * 64 * (1/4) = 64$$

B组按列访问，每次保存对角线上的元素时（**左上角元素特殊考虑**），正常B组会因为前一列的按列保存导致B组的8行都在缓存中，但因为上一步A组按行读取该对角线元素所在的行，置换了B组保存该对角线元素所在行占据的缓存块，所以B组每行8个元素会有两个元素发生Miss（**左上角元素特殊考虑**）

m	h	h	h	h	h	h	h
m	m	h	h	h	h	h	h
m	h	m	h	h	h	h	h
m	h	h	m	h	h	h	h
m	h	h	h	m	h	h	h
m	h	h	h	h	m	h	h
m	h	h	h	h	h	m	h
m	h	h	h	h	h	h	m

- 特殊情况

对于B组，对角线上的块的左上角元素，其所在的行只有这一个元素发生了Miss，因此要减去4

$$96 + 96 + 32 + 64 - 4 = 284$$

程序运算结果为287，因为有3个Miss是函数调用过程中的多余开销

64*64

思考8*8分块

```
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
```

对于64*64的矩阵而言，每一行的64个元素占8个组，故每4行会占满整个cache

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

以两个对称的红色8*8分块为例

B组按列访问元素，前4行和后4行所映射的块是相同的

1. 在访问完前4行的第一列以后，访问后4的第一列行时，由于冲突不命中，会导致原来的块被驱逐
2. 接着再访问前4行的第二列时，由于原来的块已经被驱逐，这里又会导致冲突不命中，并将后4行的块驱逐
3. 这样在访问后4行的第二列时又会产生冲突不命中

如此反复下去，最终B组访问的区域中所有的元素均会不命中。

思考4*4分块

此时不考虑优化对角线

```

if(N==64 && M==64)
{
    for(i=0;i<N;i+=4)
        for(j=0;j<M;j+=4)
            for(x=i;x<i+4;x++)
            {
                x1=A[x][j];x2=A[x][j+1];x3=A[x][j+2];x4=A[x][j+3];

                B[j][x]=x1;B[j+1][x]=x2;B[j+2][x]=x3;B[j+3][x]=x4;
            }
}

```

Function 2 (3 total)

Step 1: Validating and generating memory traces

Step 2: Evaluating performance (s=5, E=1, b=5)

func 2 (A simple transpose): hits:6498, misses:1699, evictions:1667

可以看到与8*8分块相比优化了许多，但仍然未达到1300以内

问题所在

由于一个块的大小为8个int型数据，所以我们按照4*4来分块还是**没有充分利用每次加载以后的每个块**。具体还是**表现在对B数组的访问**

由于是进行4分块，所以对于每一个高速缓存块都会进行两次访问（将4*4分块看作是8*8分块划分成4份）

对于A数组而言两次访问的间隔不会出现将原来的块覆盖的情况（同一行的前四列和后四列保存在同一块中先后访问，之后不会再访问）

前4行的前4列->前4行后4列->后4行的前4列->后4行的后4列

而对于B数组而言，由于访问顺序为：**前4行的前4列->后4行的前4列->前4行后4列->后4行的后4列**

由于后4行的前4列所在的块会覆盖前4行的前4列的块，所以在后面的两次访问均会又有一次不命中，**所以对于B数组每个块而言，都会有两次不命中**，而对于8*8分块而言，**每个块都会不命中8次**，这是4*4分块的优化之处，而两次的命中也是不足之处。

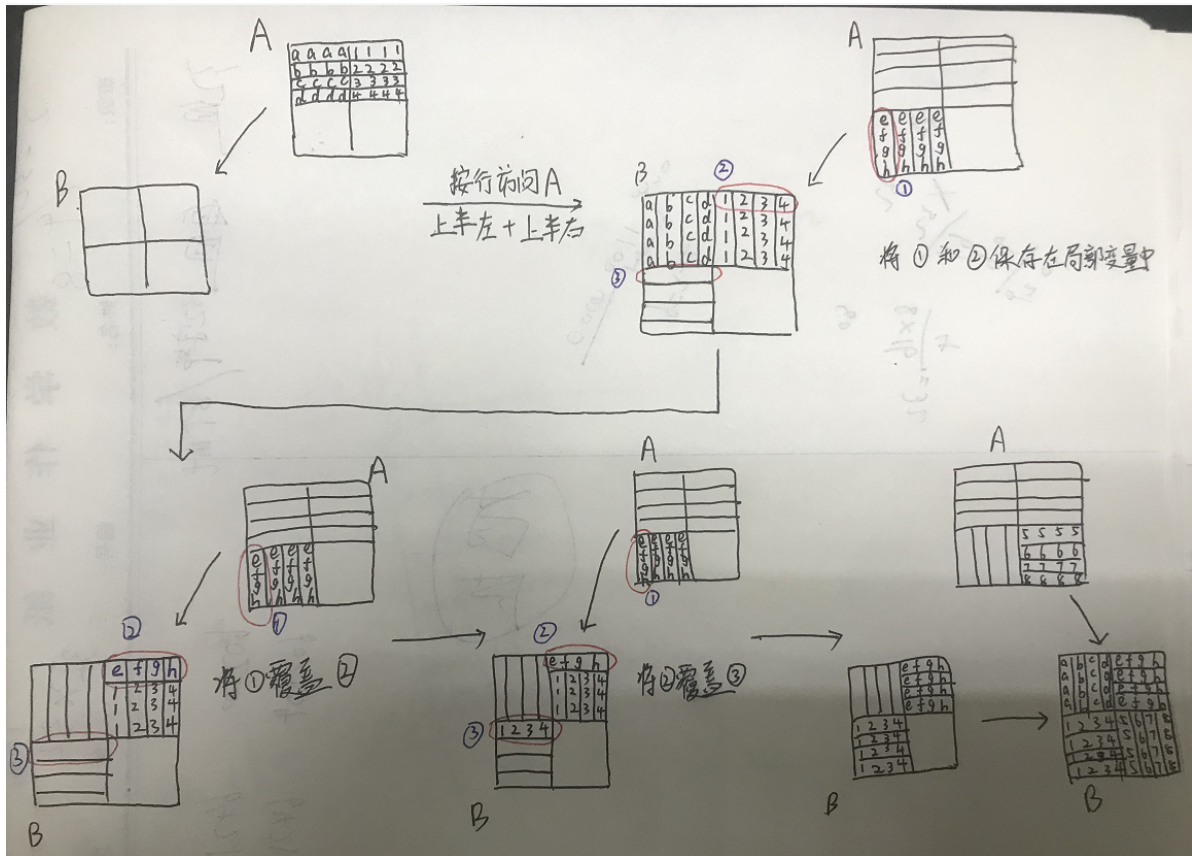
提高数组B的命中

8*8分块，但每次处理一个大分块时，将其看作四个小的4*4小分块

- 我们将A组的前4行全部存入B组的前4行（**逐行读，每行读8个元素**），这个过程中**A的左上**已经转置完成，但是对于**A的右上**还没有放入应该放的位置（**B的左下**），但是为了不再访问同一个块，我们同时将数据取出，存入还没有用到的区域中（**B的右上**）
- 对**A的左下**进行转置（**逐列读，每列读4个元素**）
 - 将**A的左下**的一列4个元素保存到局部变量1中
 - 将**B的右上**的一行4个元素保存到局部变量2中
 - 先将局部变量1覆盖到保存到局部变量2中的B的右上那一行
 - 再将局部变量2覆盖到**B的左下的指定行**（即最终位置）

此时B的左下的指定行会置换B的右上那一行占据的缓存，但因为以后不会再访问B的右上那一行，所以没有关系

- 对A的右下进行转置（正常处理）



对B数组进行分析

- B数组访问前4行
- B数组逐行访问前4行后4列与后4行前4列

此时发生了置换，但被置换的块之后不会再访问，同时后面访问的就是已经置换完的块

- B数组逐行访问后4行后4列

按照这个顺序访问以后，显然对于B数组中的每一个块的元素，只会有一个不命中

```
Function 0 (3 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
```

(对角线上的元素没有特殊处理)

```
if(N==64 && M==64)
{
    for(i=0; i<N; i+=8)
        for(j=0; j<M; j+=8)
        {
```



```

    for(x=i;x<i+4;x++) //对A的前四行进行处理，每行读取8个元素，只有A的左上块放入
最终位置
    {
        x1=A[x][j]; x2=A[x][j+1]; x3=A[x][j+2]; x4=A[x][j+3];
        x5=A[x][j+4]; x6=A[x][j+5]; x7=A[x][j+6]; x8=A[x][j+7];

        B[j][x]=x1; B[j+1][x]=x2; B[j+2][x]=x3; B[j+3][x]=x4;
        B[j][x+4]=x5; B[j+1][x+4]=x6; B[j+2][x+4]=x7; B[j+3][x+4]=x8;
    }
    for(y=j;y<j+4;y++) //逐列处理A的左下块，同时也将A的右上块(B的右上位置)放到最
最终位置(B的左下位置)
    {
        x1=A[i+4][y]; x2=A[i+5][y]; x3=A[i+6][y]; x4=A[i+7][y];
        x5=B[y][i+4]; x6=B[y][i+5]; x7=B[y][i+6]; x8=B[y][i+7];

        B[y][i+4]=x1; B[y][i+5]=x2; B[y][i+6]=x3; B[y][i+7]=x4;
        B[y+4][i]=x5; B[y+4][i+1]=x6; B[y+4][i+2]=x7; B[y+4][i+3]=x8;
    }
    for(x=i+4;x<i+8;x++) //逐行读取A的右下块，正常处理
    {
        x1=A[x][j+4]; x2=A[x][j+5]; x3=A[x][j+6]; x4=A[x][j+7];

        B[j+4][x]=x1; B[j+5][x]=x2; B[j+6][x]=x3; B[j+7][x]=x4;
    }
}

```

计算Misses数

- 非对角线上的块：56块
 - A组：56 * 64 * (1/8) = 448
 - B组：同理为448
- 对角线上的块：8块

总结规律：按行读的Miss率为1/4，按列读的Miss率为1/2（**右上角 4*4 特殊考虑**）

◦ A组：

- 左上角 4*4：Miss率 1/4
- 右上角 4*4：Miss率 0
- 左下角 4*4：Miss率 1/2
- 右下角 4*4：Miss率 1/4

$$8 * 16 * (1/4 + 0 + 1/2 + 1/4) = 128$$

◦ B组：

- 左上角 4*4：Miss率 1/2（**8个对角线上的块的左上角元素特殊考虑**）
- 右上角 4*4：Miss率 0
- 左下角 4*4：Miss率 1/4
- 右下角 4*4：Miss率 1/2

$$8 * 16 * (1/2 + 0 + 1/2 + 1/4) = 160$$

$$448 + 448 + 128 + 160 - 8 = 1176$$

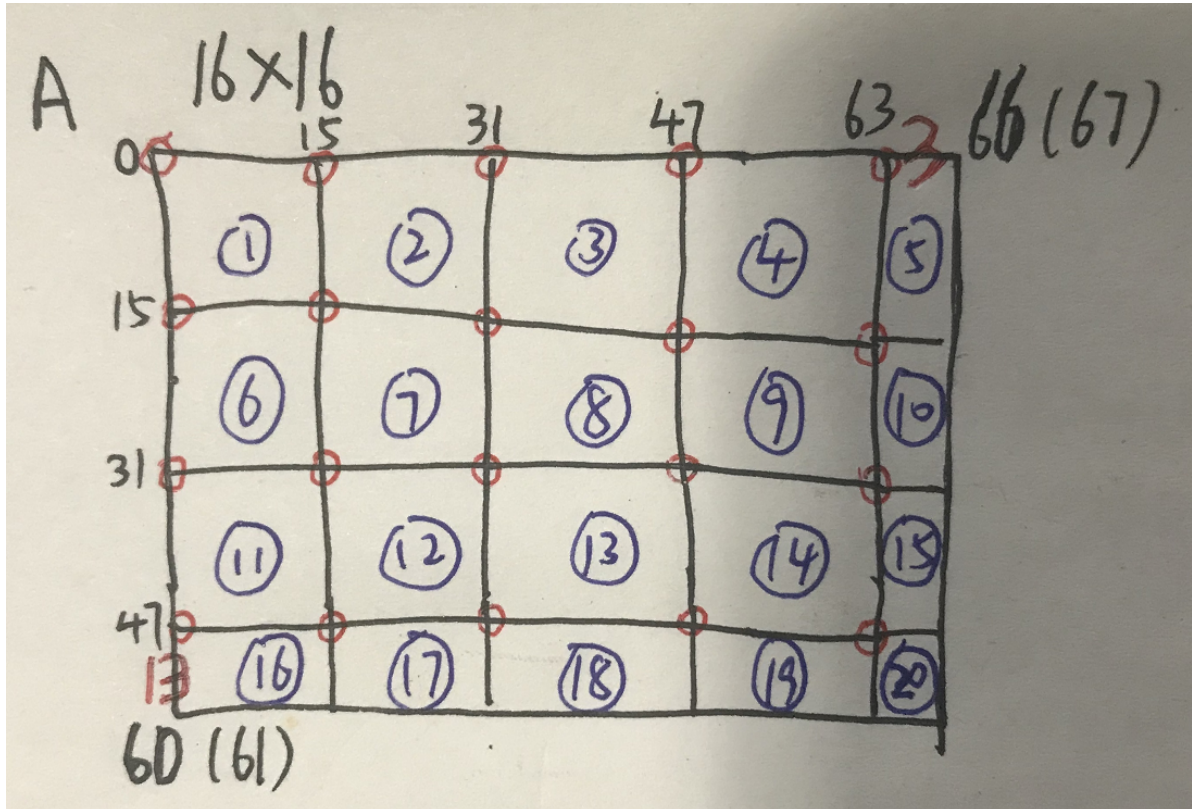
程序运算结果为1179，因为有3个Miss是函数调用过程中的多余开销

61*67

- 完全未优化的情况

```
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
```

不规则的matrix，本质也是用分块来优化Cache的读写，但是不能找到比较显然的规律看出来间隔多少可以填满一个Cache。由于要求比较松，可以尝试一些分块的大小，直接进行转置操作



- 16*16分块

i 和 j 每次都是定位到分块的左上角元素

因此即使定位到不规则分块时， i 和 j 也不会超过 N 和 M

只需要在处理不规则分块时，保证 x 和 y 不超过 N 和 M 就可以了

```

if(N==67 && M==61)
{
    for(i=0;i<N;i+=16)        //16*16分块
    {
        for(j=0;j<M;j+=16)
        {
            for(x=i; x<N && x<i+16; x++)    // x<N是限制不规则分块, x<i+16是限制16*16
规则分块
                for(y=j; y<M && y<j+16; y++)    // y<M 和 y<j+16 同理
                    B[y][x]=A[x][y];            //直接转置
        }
    }
}

```

```

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:6187, misses:1992, evictions:1960

```

- 17*17分块

```

if(N==67 && M==61)
{
    for(i=0;i<N;i+=17)        //17*17分块
    {
        for(j=0;j<M;j+=17)
        {
            for(x=i; x<N && x<i+17; x++)
                for(y=j; y<M && y<j+17; y++)
                    B[y][x]=A[x][y];
        }
    }
}

```

```

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:6229, misses:1950, evictions:1918

```

总结

```

root@ubuntu:/home/csapp/Desktop/Lab4/cachelab-handout# ./driver.py
Part A: Testing cache simulator
Running ./test-csim

```

Points	(s,E,b)	Your simulator			Reference simulator			
		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	1179
Trans perf 61x67	10.0	10	1950
Total points	53.0	53	

在运行完程序之后，结果符合要求的同时，我还手动计算了Miss率，来对运行结果进行验证。这使我对缓存，在理论和实践上，都有了更加深刻的理解。