CSAPP:CacheLab实验

原创

大白不白 于 2018-06-21 13:33:05 发布

趁期末考试复习了《深入理解计算机系统》第六章，进一步了解了cache的原理。想着写篇博客帮助巩固一下。有些地方写得可能不是很好，希望多多包涵，同时也欢迎指出。

cachelab一共分为两部分，PartA是让你模拟cache运行的过程，就是模拟cache的行为。PartB是一个矩阵转置，给出了三种规模，你的任务就是尽可能的提高高速缓存的命中率，它会根据你的miss，hits，eviction这三个值的大小进行打分，评分的范围已给出。

好啦实验介绍完毕。下面开始进入正式环节。

PartA

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 need 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 ?

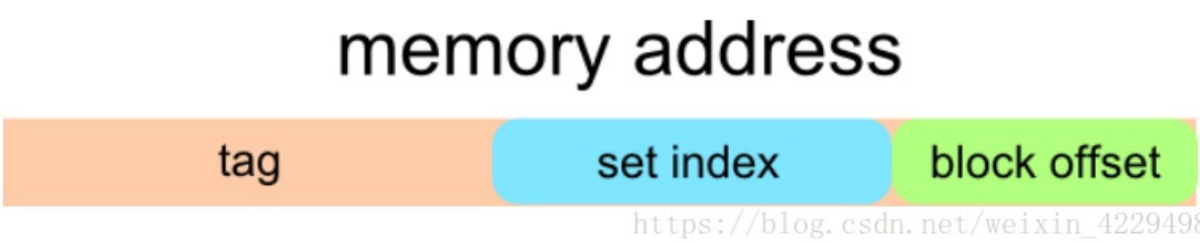
这是原版文档的要求。大概就是告诉你它只是一个模拟缓存的行为并不是真实缓存，内存的数据不用存储，不使用块偏移，因此地址种b并不重要，简单的计算hits、miss、evictions的值。然后就是你的模拟器需要适用于不同的（s，b，E），同时给出运行时间。最后就是要求你使用这个LRU最近最少使用策略，意思就是使用这个策略进行行的牺牲，因此cache发生miss的时候，就会从它的k+1层存储器读取新的拷贝，假设在空间已满的情况下（未满的时候不需要替换行）,新读取的数据需要存储在cache里面，但空间已满，只能选择驱逐（原文是驱逐的意思，也就是覆盖某一行），然后覆盖的方法就是这个LRU策略。LRU策略就是：选择那个最后被访问的时间距离现在最远的块。代码体现的话就是给cache的每一行绑一个Lrunumber，这个值越小（你也可以设置为最大，随自己喜欢）表示它最后一次被访问的时间距离现在最远，可以作为牺牲行。

一、那么模拟一个cache的行为需要用到哪些变量？

cache有组数S、一组包含的行数E，存储块的字节大小B，cache的容量C（C=S*E*B)

地址的构成：标识位t、组索引s、块偏移b（前面说了实验不使用块偏移，但计算标记位的时候需要用到）

示意图：



我简单解释下上面说的这些变量：

首先是cache的构成:

cache有2^s组，每组有E行（E分为三种情况：E=1，称为直接映射高速缓存；1<E<C/B:组相联高速缓存；E=C/B，全相连高速缓存，也就是一组包含了所有的行），然后是每行的构成：（1）有效位vild(取0和1，1表示存储了有效信息，0表示没有，判断命中的时候需要用到）（2）标记位t=m-(s+b),当cpu要读取某个地址的内容时，就会将某个地址的第一个部分：标记位与它进行对比，匹配相等则命中，也就是锁定在某一行（3）数据块B，B负责存储这个地址的内容，把B想象成字节数组就好，共有B-1个小块（别和B这个数据块搞混咯,下标从0开始，因此是B-1）。在这里简单提下b的作用吧，虽然实验没用到。b理解为数据块B这个数组的下标就好，也就是你要从B[b]这个位置开始读取字。注意这里的数据都是2进制的。

以上这些都可以从那个原版的PPT读取，只是我讲得更详细些，这也是我自己的理解。

二、自带函数的介绍：

cache.h里面自带了一些函数，PPT给出了要用到的几个的介绍：

1.getopt（）

如果函数声明丢失，则在Unix命令行上自动解析元素，通常在循环中调用以检索参数，它的返回值存储在局部变量中，

当getopt（）返回-1时，没有更多的选项。（百度翻译的）一句话就是它是解析你的那个命令行的。

2.fscanf（）

读入测试文件的，自带有调用就好了。具体使用方式如下：（写的时候粘贴复制一下就好了。）

```
FILE * pFile; //pointer to FILE object
pFile = fopen ("tracefile.txt",“r"); //open file for reading
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){
// Do stuff
}

fclose(pFile); //remember to close file when done
```

3.Malloc/free

分配和释放内存空间的函数。

具体用法如下：

```
Some_pointer_you_malloced = malloc(sizeof(int));

Free(some_pointer_you_malloced);
```

分配了内存用完的时候记得释放，还有不要释放没有分配的内存。

三、根据（一）可以写出下面结构体：（基于c语言）

定义行的属性：

```
typedef struct{

    int valid;       //有效位
    int tag;         //标识位
    int LruNumber;   //牺牲行的时候要用的，具体上面说了。
} Line;
```

定义组的属性：

```
typedef struct{
    Line* lines;    //用于存储一组中包含的行
} Set;
```

定义cache的属性:

```
typedef struct {
    int set_num;    //组数
    int line_num;   //行数
    Set* sets;      //cache的空间，模拟cache
} Sim_Cache;
```

四、函数实现：

1.getopt(),命令行解析函数，主要用来解析你输入的那个命令。命令中包含的关键字就那么几个，s，E，b，v(是否打印标记位和组索引）atoi函数是将字符转换成整数的函数。

```
int get_Opt(int argc,char **argv,int *s,int *E,int *b,char *tracefileName,int *isVerbose){
    int c;
    while((c = getopt(argc,argv,"hvs:E:b:t:"))!=-1)
    {
        switch(c)
        {
        case 'v':
            *isVerbose = 1;
            break;
        case 's':
            checkOptarg(optarg);
            *s = atoi(optarg);
            break;
        case 'E':
            checkOptarg(optarg);
            *E = atoi(optarg);
            break;
        case 'b':
            checkOptarg(optarg);
            *b = atoi(optarg);
            break;
        case 't':
            checkOptarg(optarg);
            strcpy(tracefileName,optarg);
            break;
        case 'h':
        default:
            printHelpMenu();
            exit(0);
        }
    }
    return 1;
}
```


2.文本处理函数，也就是读入测试文件的，直接用它自带的fscanf就好了。在cache.h那个头文件里面。

3.检查参数合法性：主要目的就是检查输入的命令是否合法。

```
void checkOptarg(char *curOptarg){
    if(curOptarg[0]=='-'){
        printf("./csim :Missing required command line argument\n");
        printHelpMenu();
        exit(0);
    }
}
```


4.打印帮助文档的函数，当输入的命令出错的时候，运行这个函数，也就是说你解释一下这些关键字代表的意思就好啦。

```
void printHelpMenu(){
    printf("Usage: ./csim-ref [-hv] -s <num> -E <num> -b <num> -t <file>\n");
    printf("Options:\n");
    printf("-h         Print this help message.\n");
    printf("-v         Optional verbose flag.\n");
    printf("-s <num>   Number of set index bits.\n");
    printf("-E <num>   Number of lines per set.\n");
    printf("-b <num>   Number of block offset bits.\n");
    printf("-t <file>  Trace file.\n\n\n");
    printf("Examples:\n");
    printf("linux>  ./csim -s 4 -E 1 -b 4 -t traces/yi.trace\n");
    printf("linux>  ./csim -v -s 8 -E 2 -b 4 -t traces/yi.trace\n");
}
```


5.更新Lru计数值的函数：**访问第x组E行，根据标记为将该行的LRU设置为最大值，然后其它行用一个for循环减一即可。**

```
void updateLruNumber(Sim_Cache *sim_cache,int setBits,int hitIndex){
        sim_cache->sets[setBits].lines[hitIndex].LruNumber = MAGIC_LRU_NUM;
        int j;
        for(j=0;j<sim_cache->line_num;j++){//更新其他行的LruNumber
            if(j!=hitIndex) sim_cache->sets[setBits].lines[j].LruNumber--;
        }
}
```


6.查找牺牲行的函数：

遍历找出第x组，找到最小的LRU所在的行，那么该行就是牺牲行。关于怎么找，无非就算设置标记位，不断比较和更新即可。

```
int findMinLruNumber(Sim_Cache *sim_cache,int setBits){
    int i;
    int minIndex=0;
    int minLru = MAGIC_LRU_NUM;
    for(i=0;i<sim_cache->line_num;i++){
        if(sim_cache->sets[setBits].lines[i].LruNumber < minLru){
            minIndex = i;
            minLru = sim_cache->sets[setBits].lines[i].LruNumber;
        }
    }
    return minIndex;
}
```


7.判断是否命中：命中在前面说了，要满足两个条件：（1）设置了有效位；（2）请求的地址的标记位与cache地址的标记位一致。最后记得更新一下Lru计数值。

```
int isMiss(Sim_Cache *sim_cache,int setBits,int tagBits){
    int i;
    int isMiss = 1;
    for(i=0;i<sim_cache->line_num;i++){
        if(sim_cache->sets[setBits].lines[i].valid == 1 && sim_cache->sets[setBits].lines[i].tag == tagBits){
            isMiss = 0;
            updateLruNumber(sim_cache,setBits,i);
        }
    }
    return isMiss;
}
```


8.更新高速缓存cache的函数：cache的容量有限，当满的时候需要牺牲行（或者说驱逐某行），先遍历当前组，判断它满了没有，如何判断是否满，可以遍历所有的行，只要有一个有效位为0，（有效位的作用是说明该行是否存储了数据，通俗的理解就是是否为空）则该组未满。如果没有满的话：将有效位不为1的那一行更新有效位为1，同时更新标记位和LRU计数值。如果满了，找出最小LRU所在的行作为牺牲行。

```
int updateCache(Sim_Cache *sim_cache,int setBits,int tagBits){
    int i;
    int isfull = 1;
    for(i=0;i<sim_cache->line_num;i++){
        if(sim_cache->sets[setBits].lines[i].valid == 0){
            isfull = 0; //该组未满
            break;
        }
    }
    if(isfull == 0){
        sim_cache->sets[setBits].lines[i].valid = 1;
        sim_cache->sets[setBits].lines[i].tag = tagBits;
        updateLruNumber(sim_cache,setBits,i);
    }else{
        //组已经满，需要牺牲行
        int evictionIndex = findMinLruNumber(sim_cache,setBits);
        sim_cache->sets[setBits].lines[evictionIndex].valid = 1;
        sim_cache->sets[setBits].lines[evictionIndex].tag = tagBits;
        updateLruNumber(sim_cache,setBits,evictionIndex);
    }
    return isfull;
}
```


9.读取数据的函数，亦即操作L：指导PPT上说命中则hit++，不命中则miss++且如果牺牲行eviction++，如果有v指令就打印访问结果。那么可以用一个标记来判断是否有-v指令。

```
void loadData(Sim_Cache *sim_cache,int addr,int size,int setBits,int tagBits ,int isVerbose){

    if(isMiss(sim_cache,setBits,tagBits)==1){ //没有命中
        misses++;
        if(isVerbose == 1) printf("miss ");
        if(updateCache(sim_cache,setBits,tagBits) == 1){//该组已满，需要牺牲行
            evictions++;
            if(isVerbose==1) printf("eviction ");
        }
    }else{ //命中
       hits++;
       if(isVerbose == 1) printf("hit ");
    }
}
```

10.存储数据的函数，亦即操作S：原理跟L操作一样也是修改标记位和组索引指令的，然后调用一下L操作那个函数即可。

```
void storeData(Sim_Cache *sim_cache,int addr,int size,int setBits,int tagBits ,int isVerbose){
    loadData(sim_cache,addr,size,setBits,tagBits,isVerbose);
}
```


11.修改数据的函数，亦即操作M:这个操作实际就是执行了一次L和一次S。

```
/*M 操作*/
void modifyData(Sim_Cache *sim_cache,int addr,int size,int setBits,int tagBits ,int isVerbose){
    loadData(sim_cache,addr,size,setBits,tagBits,isVerbose);
    storeData(sim_cache,addr,size,setBits,tagBits,isVerbose);
}
```


12.获取标记位和组索引：前面说了一个当要访问一个地址（暂且称为请求地址吧）的内容时，这个请求地址是这样的结构：标记位(t) 组索引(s) 块偏移(b)，然后cache中地址的结构：有效位(v) 标记位(v) 数据块(B)。我们现在的任务是从这个请求地址中获取t和s，才能从cache中寻找需要访问的内容。地址位数未知，但问题不大，比如我求标记位的时候直接将它右移到最右边即可，组索引由于右移的时候带着标记位，因此要和一个数与运算一下，把它“剪”下来。

```
int getSet(int addr,int s,int b){
    addr = addr >> b;
    int mask =  (1<<s)-1;
    return addr &mask;
}
```



```
int getTag(int addr,int s,int b){
    int mask = s+b;
    return addr >> mask;
}
```


13.最后写一个初始化cache的函数就好了。

```c
/*初始化模拟内存*/
void init_SimCache(int s,int E,int b,Sim_Cache *cache){
    if(s < 0){
        printf("invaild cache sets number\n!");
        exit(0);
    }
    cache->set_num = 2 << s; //2^s 组
    cache->line_num = E;
    cache->sets = (Set *)malloc(cache->set_num * sizeof(Set));
    if(!cache->sets){
        printf("Set Memory error\n");
        exit(0);
    }
    int i ,j;
    for(i=0; i< cache->set_num; i++)
    {
        cache->sets[i].lines = (Line *)malloc(E*sizeof(Line));
        if(!cache->sets){
            printf("Line Memory error\n");
            exit(0);
        }
        for(j=0; j < E; j++){
            cache->sets[i].lines[j].valid = 0;
            cache->sets[i].lines[j].LruNumber = 0;
        }
    }
    return ;
}
```


14.main函数代码：

```c
int misses;
int hits;
int evictions;
int main(int argc,char **argv){
    int s,E,b,isVerbose=0;
    char tracefileName[100],opt[10];

    int addr,size;
    misses = hits = evictions =0;

    Sim_Cache cache;

    get_Opt(argc,argv,&s,&E,&b,tracefileName,&isVerbose);
    init_SimCache(s,E,b,&cache);
    FILE *tracefile = fopen(tracefileName,"r");

    while(fscanf(tracefile,"%s %x,%d",opt,&addr,&size) != EOF){
        if(strcmp(opt,"I")==0)continue;
        int setBits = getSet(addr,s,b);
        int tagBits = getTag(addr,s,b);
        printf("---------------------------------\n");
        printf("setBits:%x  tagBits:%x\n",setBits,tagBits);
        if(isVerbose == 1) printf("%s %x,%d ",opt,addr,size);
        if(strcmp(opt,"S")==0) {
            storeData(&cache,addr,size,setBits,tagBits,isVerbose);
        }
        if(strcmp(opt,"M")==0) {
            modifyData(&cache,addr,size,setBits,tagBits,isVerbose);
        }
        if(strcmp(opt,"L")==0) {
            loadData(&cache,addr,size,setBits,tagBits,isVerbose);
        }
        if(isVerbose == 1) printf("\n");
    }
    printSummary(hits,misses,evictions);
    return 0;
}
```


hit、miss、eviction三个变量定义成全局好些。

然后贴一下运行的结果图：（对比标准的缓存与自己写的缓存，引用标准的缓存程序加-ref）

```
lzh6666@ubuntu:~/Downloads/cachelab-handout$ ./csim-ref -v -s 4 -E 1 -b 4 -t traces/yi.trace
L 10,1 miss
M 20,1 miss hit
L 22,1 hit
S 18,1 hit
L 110,1 miss eviction
L 210,1 miss eviction
M 12,1 miss eviction hit
hits:4 misses:5 evictions:3
lzh6666@ubuntu:~/Downloads/cachelab-handout$ ./csim -s 4 -E 1 -b 4 -t traces/yi.trace
---------------------------------
setBits:1  tagBits:0
---------------------------------
setBits:2  tagBits:0
---------------------------------
setBits:2  tagBits:0
---------------------------------
setBits:1  tagBits:0
---------------------------------
setBits:1  tagBits:1
---------------------------------
setBits:1  tagBits:2
---------------------------------
setBits:1  tagBits:0
hits:4 misses:5 evictions:3
lzh6666@ubuntu:~/Downloads/cachelab-handout$
```


yi.trace的内容如下：

yi.trace

```
1  L 10,1
2  M 20,1
3  L 22,1
4  S 18,1
5  L 110,1
6  L 210,1
7  M 12,1
```


它就是测试了这7条指令。下面我们来简要分析一下：

1.对于地址0x10进行访问：(标蓝的表示组索引）
x10=0000...0001 0000，块偏移值为最低四位，故组索引s=1;
一开始的时候cache是空的，因此第一次访问的时候为miss。
2.执行指令M 20，连续对地址0x20进行连续两次访问：
0x20=000...0010 0000，组索引s=2;
所以第一次访问的时候没有要的内容，访问结果为miss；然后cache从低一级存储器读取第一次访问需要的内容，第二次访问的时候有需要的内容且标记位相等，所以第二次访问的时候结果为hit；
3.对地址0x22进行访问：
0x22=000...0010 0100，组索引s=2;
由于操作2以将该块存入高速缓存且标记位都相等，故结果为hit;

4.对地址0x18进行访问：

0x18=000...0001 0100,组索引s=1；

由于之前的操作，该块已存入高速缓存且标记位都为0，故访问结果为hit

5.对地址0x110进行访问：

0x110=000...0001 0001 0000 故组索引s=1；

操作4将该块存入高速缓存了但标记位不相等，故访问结果为miss,发生一次eviction；

6.对地址0x210进行访问：

0x210=000...0010 0001 0000，故组索引s=1，这里标记位为2跟操作5读取新的行的标记位不匹配，故访问结果为miss,cache读取新的行，发生一次eviction。

7.对地址0x12进行连续两次访问：

0x12=000...0001 0010,组索引s=1;

第一次访问结果为miss，因为在操作6的时候发生了一次行替换把该块驱逐了即使标记位相等,cache重新读取该块，发生一次eviction，那么第二次肯定为hit。（跟操作1类似的）

故总共：hits=4 ； miss=5；eviction=3；

下面是打分的运行图：

运行指令./test-csim:

```
lzh6666@ubuntu:~/Downloads/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
```


可以看到跟标准的缓存一样，但还有地方可以优化，期待大佬指出！

PartB：

PartB我就不做过多的赘述了。

以下是PartB的一些要求：

1.最多只能定义 12 个局部变量

2.不使用任何的数组，不调用任何类似于 malloc 的开辟内存的函数

3.测试矩阵的规模为 32 × 32，64 × 64，61 × 67

4.测试 cache 的组成：32 组， 每组一行，每个块 32 字节。对于编写的函数，miss 个数越少越好。

PartB打分规则如下：

- 32×32: 8 points if $m < 300$, 0 points if $m > 600$
- 64×64: 8 points if $m < 1,300$, 0 points if $m > 2,000$
- 61×67: 10 points if $m < 2,000$, 0 points if $m > 3,000$

矩阵转置让我联想到上次perlaba实验对图像旋转时用的分块处理方法，这里可分成32x32，16x16，8x8，4x4,cache每块的字节数是32，也就是8个int，因此为了有效地利用cache的块容量，选取8x8的分块效果是最好的。

首先编写处理32x32的函数：

分块后，对角线上的块由于位置没有改变，所以需要另外处理。（否决了）尝试了很久之后我换一种方法，就是我直接将每一块的八个int直接取出来就好了。就算是对角线的元素也满足。代码如下：

```
int i, j, k, temp0, temp1, temp2, temp3, temp4, temp5, temp6, ter
   if ( M == 32 )
   {
      for (j = 0; j < 32; j = j+8)
      {
          for (i = 0; i < 32; i++)
          {
               temp0 = A[i][j];
               temp1 = A[i][j+1];
               temp2 = A[i][j+2];
               temp3 = A[i][j+3];
               temp4 = A[i][j+4];
               temp5 = A[i][j+5];
               temp6 = A[i][j+6];
               temp7 = A[i][j+7];
               B[j][i] = temp0;
               B[j+1][i] = temp1;
               B[j+2][i] = temp2;
               B[j+3][i] = temp3;
               B[j+4][i] = temp4;
               B[j+5][i] = temp5;
               B[j+6][i] = temp6;
               B[j+7][i] = temp7;
           }
      }
   }
```


运行结果如图：

```
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        0.0         8     invalid
Trans perf 61x67        0.0        10     invalid
      Total points     35.0        53
```


满分，效果不错，是否64x64也能继续沿用这种办法呢？试一试咯。

当矩阵规模增到到64x64的时候，由于矩阵规模比较大，无法在cache的一行全部加载矩阵的一行，还是先分成8x8的块。我处理是思路是这样的：每次处理一个块，也就是像32x32那样取出每行的8个元素，然后将 每行的0-3号元素正常的放到B[0-3][0]这些位置去（也就是正常的转置),剩下的四个元先放置B矩阵4x4块的最右边保存，注意这个时候我已经取出来了。然后再用一个循环去把这些值放到B正确的位置去，然后A[4]到A[7]也是上述的处理。对整个矩阵重复这个操作即可。简单的示意图如下：

A[0][0]			A[0][4]
A[0][1]			A[0][5]
A[0][2]			A[0][6]
A[0][3]			A[0][7]

但是这个方法有个缺点，就是越到后面用来存储最后四个元素的空间就越少（这个肯定是一个能优化的地方，我暂时没想到如何解决，期待大佬指出！）

参考代码如下：

```
for(i=0;i<64;i+=8)
        for(j=0;j<64;j+=8)
        {
                for(k=j;k<j+4;++k)
                {
                        tmp0=A[k][i];
                        tmp1=A[k][i+1];
                        tmp2=A[k][i+2];
                        tmp3=A[k][i+3];
                        tmp4=A[k][i+4];
                        tmp5=A[k][i+5];
                        tmp6=A[k][i+6];
                        tmp7=A[k][i+7];
                        B[i][k]=tmp0;
                        B[i][k+4]=tmp4;
                        B[i+1][k]=tmp1;
                        B[i+1][k+4]=tmp5;
                        B[i+2][k]=tmp2;
                        B[i+2][k+4]=tmp6;
                        B[i+3][k]=tmp3;
                        B[i+3][k+4]=tmp7;
                }
                for(k=i;k<i+4;++k)
                {
                        tmp0=B[k][j+4];
                        tmp1=B[k][j+5];
                        tmp2=B[k][j+6];
                        tmp3=B[k][j+7];
                        tmp4=A[j+4][k];
                        tmp5=A[j+5][k];
                        tmp6=A[j+6][k];
                        tmp7=A[j+7][k];
                        B[k][j+4]=tmp4;
                        B[k][j+5]=tmp5;
                        B[k][j+6]=tmp6;
                        B[k][j+7]=tmp7;
                        B[k+4][j]=tmp0;
                        B[k+4][j+1]=tmp1;
                        B[k+4][j+2]=tmp2;
                        B[k+4][j+3]=tmp3;
                }
                for(k=i+4;k<i+8;++k)
                {
                        tmp0=A[j+4][k];
                        tmp1=A[j+5][k];
                        tmp2=A[j+6][k];
                        tmp3=A[j+7][k];
                        B[k][j+4]=tmp0;
                        B[k][j+5]=tmp1;
                        B[k][j+6]=tmp2;
                        B[k][j+7]=tmp3;
                }
        }
```


运行结果如图：

```
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        1219
Trans perf 61x67           0.0        10     invalid
        Total points      43.0        53
lzh6666@ubuntu:~/Downloads/cachelab-handout$
```


有大佬优化那个miss到1123，我相信我这个优化了上面说的那个问题也能达到那个值。又是一个满分，因为不满分不做下一个2333。

最后是61x67,乍一看这玩意的size有点稀奇，还想引用刚才的分块大小，但是给了我一个invild！不得不放弃这种办法了，后来想过增大分块的大小，之前说8x8的分块效果是最好的，那只是针对32x32或者64x64这种跟cache的容量大小成倍数关系的矩阵，而且也不一定非要用8x8的分块啊！后来我调整了分块的大小，测试了四组，最终确定17是最好的。

贴图贴图:

1.8x8:

```
 else if(M==61)
{
        for(i=0;i<61;i+=8)          //B[i][j]=A[j
            for(j=0;j<67;j+=8)
                for(k=j;k<j+8 && k<67;++k)
                    for(m=i;m<i+8 && m<61;++m)
                    {
                            tmp0=A[k][m];
                            B[m][k]=tmp0;
                    }
}
```

```
    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        1219
Trans perf 61x67          10.0        10        1913
        Total points      53.0        53
lzh6666@ubuntu:~/Downloads/cachelab-handout$
```


1913还能接受，起码得到满分了。但还可以继续优化!

2.16x16:

```
 else if(M==61)
{
        for(i=0;i<61;i+=16)                 //B[i
            for(j=0;j<67;j+=16)
                for(k=j;k<j+16 && k<67;++k)
                    for(m=i;m<i+16 && m<61;++m)
                    {
                            tmp0=A[k][m];
                            B[m][k]=tmp0;
                    }
}
```

```
    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        1219
Trans perf 61x67          10.0        10        1816
        Total points      53.0        53
lzh6666@ubuntu:~/Downloads/cachelab-handout$
```


emmm,下降明显!继续增大看看。

3.17x17:

```
*/
    else if(M==61)
    {
            for(i=0;i<61;i+=17)                 //B[i
                for(j=0;j<67;j+=17)
                    for(k=j;k<j+17 && k<67;++k)
                        for(m=i;m<i+17 && m<61;++m)
                        {
                                tmp0=A[k][m];
                                B[m][k]=tmp0;
                        }

    }

}
```

```
e
    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        1219
Trans perf 61x67          10.0        10        1813
        Total points      53.0        53
lzh6666@ubuntu:~/Downloads/cachelab-handout$
```


下降的幅度很小了，所以到这里差不多了，但还是增大看看。

4.18x18：

```
    else if(M==61)
    {
            for(i=0;i<61;i+=18)                 //B[i
                for(j=0;j<67;j+=18)
                    for(k=j;k<j+18 && k<67;++k)
                        for(m=i;m<i+18 && m<61;++m)
                        {
                                tmp0=A[k][m];
                                B[m][k]=tmp0;
                        }

    }
```

```
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        1219
Trans perf 61x67          10.0        10        1825
        Total points      53.0        53
lzh6666@ubuntu:~/Downloads/cachelab-handout$
```


大于18的都不用考虑，这里开始miss上升了。

然后贴个最终的运行截图吧。

运行命令：./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        1219
Trans perf 61x67          10.0        10        1813
        Total points      53.0        53
```


有写得不好的地方欢迎指出。