高等计算机体系结构实验

学号	S211000804	姓名	骆一鑫	专业年级班级	计科 21 级 2 班
学号	S211000820	姓名	金洁茜	专业年级班级	计科 21 级 1 班
实验日期	2021.12.19	实验项目	程序优化实验		

1. 实验内容

1.1. 优化矩阵转置。

如何在软件编译过程中发挥 cache 的性能：

总的来说，就是减少 miss 的数量，增加有效的数据访问的数量。

可以从以下几种来解决

（1）cache 对齐

数据跨越两个 cache line，就意味着两次 load 或者两次 store。如果数据结构是 cache line 对齐的， 就有可能减少一次读写。数据结构的首地址 cache line 对齐，意味着可能有内存浪费（特别是 数组这样连续分配的数据结构），所以需要在空间和时间两方面权衡。

（2）把相关代码放在一起

把相关代码放在一起有两个涵义，一是相关的源文件要放在一起；二是相关的函数在 object 文件 里面，也应该是相邻的。这样，在可执行文件被加载到内存里面的时候，函数的位置也是相邻的。 相邻的函数，冲突的几率比较小。而且相关的函数放在一起，也符合模块化编程的要求：那就是 高内聚，低耦合。

（3）分支预测

代码在内存里面是顺序排列的。对于分支程序来说，如果分支语句之后的代码有更大的执行几率， 那么就可以减少跳转，一般 CPU 都有指令预取功能，这样可以提高指令预取命中的几率。分支预测 用的就是 likely/unlikely 这样的宏，一般需要编译器的支持，这样做是静态的分支预测。现在也有 很多 CPU 支持在 CPU 内部保存执行过的分支指令的结果（分支指令的

cache），所以静态的分支预测 就没有太多的意义。如果分支是有意义的，那么说明任何分支都会执行到，所以在特定情况下，静态 分支预测的结果并没有多好，而且 likely/unlikely 对代码有很大的侵害（影响可读性），所以一般不 推荐使用这个方法。

（4）数据预取

指令预取是 CPU 自动完成的，但是数据预取就是一个有技术含量的工作。数据预取的依据是预取的数据 马上会用到，这个应该符合空间局部性，但是如何知道预取的数据会被用到，这个 要看上下文的关系。一般来说，数据预取在循环里面用的比较多，因为循环是最符合空间局部性的代码。

但是数据预取的代码本身对程序是有侵害的（影响美观和可读性），而且优化效果不一定很明显（命中的概率）。数据预取可以填充流水线，避免访问内存的等待，还是有一定的好处的。

（5）提前计算

有些变量，需要计算一次，多次使用的时候。最好是提前计算一下，保存结果，以后再引用，避免每次都 重新计算一次。函数多了，有时就会忽略这个函数都做了些什么，写程序的人可以不了解，但是优化的时候 不能不了解。能使用常数的地方，尽量使用常数，加减乘除都会消耗 CPU 的指令，不可不查。

想要优化 cache，还是需要熟悉 cache 和机器运行你的代码时是怎么做的，同时还需要多多练习。多学多想多做。

2. 实验步骤

2.1. Part A

实现一个根据指令模拟缓存读取的程序（从逻辑熟悉）

2.1.1. 实现-h 选项的实现

```
int checkArvgH()              //检查 -h 参数
{
    if(cacheInfo.h==1)        //有的话 输出帮助信息，并且不再执行
    {
        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");
        printf("\n");
        printf("Examples:\n");
        printf("  linux>  ./csim-ref -s 4 -E 1 -b 4 -t traces/yi.trace\n");
        printf("  linux>  ./csim-ref -v -s 8 -E 2 -b 4 -t traces/yi.trace\n");
        return 1;
    }
    return 0;
}
```

```
lyx@ubuntu:~/jsjxtsy/partA/cachelab-handout$ ./csim -h -v -s 4 -E 1 -b 4 -t trac
es/yi.trace
Usage: ./csim-ref [-hv] -s <num> -E <num> -b <num> -t <file>
Options:
  -h         Print this help message.
  -v         Optional verbose flag.
  -s <num>   Number of set index bits.
  -E <num>   Number of lines per set.
  -b <num>   Number of block offset bits.
  -t <file>  Trace file.

Examples:
  linux>  ./csim-ref -s 4 -E 1 -b 4 -t traces/yi.trace
  linux>  ./csim-ref -v -s 8 -E 2 -b 4 -t traces/yi.trace
lyx@ubuntu:~/jsjxtsy/partA/cachelab-handout$ ./csim-ref -h -v -s 4 -E 1 -b 4 -t
traces/yi.trace
Usage: ./csim-ref [-hv] -s <num> -E <num> -b <num> -t <file>
Options:
  -h         Print this help message.
  -v         Optional verbose flag.
  -s <num>   Number of set index bits.
  -E <num>   Number of lines per set.
  -b <num>   Number of block offset bits.
  -t <file>  Trace file.

Examples:
  linux>  ./csim-ref -s 4 -E 1 -b 4 -t traces/yi.trace
  linux>  ./csim-ref -v -s 8 -E 2 -b 4 -t traces/yi.trace
lyx@ubuntu:~/jsjxtsy/partA/cachelab-handout$
```

2.1.2. 创建逻辑 cache（初始化）

定义（因为我们不真实储存数据，同时约定不会跨越块，所以 b 信息其实无用），设计一个标记位（组相同，看标记位，组和标记位的组合可以使储存离散，故实现小地址映射大地址），因为使用 LRU，还需要记录使用时间，建立一个时间戳属性，越小代表越近使用（但是理论上无 0，因为最近一次就是上一次，没有上 0 次的说法，故 0 可当做有效无效位的标志）。

```
typedef struct
{
    //unsigned long b;          //不真实存储数据，没必要存b信息
    unsigned long tag;          //标记位
    unsigned long timestamp;    // 时间戳   因为要用LRU替换啊   越大，最近使用越久远，越要替换   简单说替换最大的
} cacheLine;
cacheLine ** cache;
```

实现，就是动态申请二维数组。

```
92 void initCache()              //初始化cache[S][E]
93 {
94     cache=(cacheLine **)malloc(sizeof(cacheLine*)*(cacheInfo.S));
95     for(int i=0; i<cacheInfo.S; i++)
96     {
97         *(cache+i) = (cacheLine*)malloc(sizeof(cacheLine)*cacheInfo.E);
98     }
99     for(int i=0; i<cacheInfo.S; i++)
00     {
01         for(int j=0; j<cacheInfo.E; j++)
02         {
03             cache[i][j].tag = 0xffffffff;          //标志位 全1
04             cache[i][j].timestamp = 0;             //默认为0   0<=>无效
05         }
06     }
07 }
```

2.1.3. 读取数据

定义文件指针，然后试图访问路径文件。

```
FILE * file=fopen(cacheInfo.t,"r");     //创建指向文件的指针   对文件进行操作
if(file==NULL)                          //打开失败
{
    printf("打开文件失败\n请检查路径\n");
    return 1;                           //最好不要常规的返回0   表示这次是错误结束程序
}
else {
    printf("打开成功\n");
}
```

测试：

```
lyx@ubuntu:~/jsjxtsy/partA/cachelab-handout$ ./csim  -s 4 -E 1 -b 4 -t traces/yi.trace
打开成功
hits:0 misses:0 evictions:0
lyx@ubuntu:~/jsjxtsy/partA/cachelab-handout$
```

使用 fscanf 函数读取规律文件内容，尝试输出。

```
//int fscanf(FILE * stream, const char * format, [argument...]);  //遇到空格，换行结束
char opt;                           //操作类型    L  S  M
unsigned long address;              //地址
int block;                          //访问字节数
while (fscanf(file," %c %lx,%d", &opt, &address, &block) > 0)
{
    printf("我读取的： %c %lx,%d\n",opt,address,block);

}
fclose(file);
```

测试：

```
lyx@ubuntu:~/jsjxtsy/partA/cachelab-handout$ ./csim  -s 4 -E 1 -b 4 -t traces/yi.trace
打开成功
我读取的： L 10,1
我读取的： M 20,1
我读取的： L 22,1
我读取的： S 18,1
我读取的： L 110,1
我读取的： L 210,1
我读取的： M 12,1
hits:0 misses:0 evictions:0
```

2.1.4. 根据读取的数据，模拟 cache 的操作

使用读取的参数，解析（L,S 访问一次 cache，M 访问两次）

```
while (fscanf(file," %c %lx,%d", &opt, &address, &block) > 0)
{
    //printf("我读取的： %c %lx,%d\n",opt,address,block);

    switch(opt)
    {
        case 'I':               //不做操作
            break;
        case 'L':
            find(address);
            break;
        case 'M':
            find(address);
        case 'S':
            find(address);
            break;
    }
    time();


}
fclose(file);
```

模拟访问 cache

解析组索引和标记位信息。

```
int tagLen = (addressSize - cacheInfo.b - cacheInfo.s);

int s = ((address<<tagLen)>>(cacheInfo.b+tagLen));          //组索引
unsigned long tag = address>>(cacheInfo.b + cacheInfo.s);    // 标记
```

通过组索引找到的组，某行（块）标记位与访问标记位相同（Hit）。

```
for(int i=0;i<cacheInfo.E;i++)          //直接在缓存中找到   hit
{
    if(cache[s][i].tag ==tag)
    {
        cache[s][i].timestamp = 1;      //表示最近一次使用过
        hit++;

        return;
    }
}
```

未直接 hit，那么已经 miss，考虑是否有 eviction。如果该组有无效的行（块）（时间戳为 0），则直接使用即可，不然就需要替换。

```
for(int i=0;i<cacheInfo.E;i++)
{
    if(cache[s][i].timestamp == 0)          //有空的（无效的)缓存
    {
        cache[s][i].tag = tag;              //当前地址映射到这个空的里  miss
        cache[s][i].timestamp = 1;          //表示最近一次使用过
        miss++;

        return;
    }
}
```

没有无效的行（块），miss+eviction。

```
int max_stamp=0;        //当前最大的时间戳
int max_i;              //最大时间戳位置
for(int i=0;i<cacheInfo.E;i++)
{
    if(cache[s][i].timestamp >max_stamp)
    {
        max_stamp = cache[s][i].timestamp;
        max_i = i;
    }
}
miss++;
eviction++;

cache[s][max_i].tag = tag;
cache[s][max_i].timestamp = 1;          //被替换  更新时间戳  最近一次使用
```

每次操作，有效行（块）时间戳都加 1。

```
void time()
{
    for(int i=0; i<cacheInfo.S; i++)
    {
        for(int j=0; j<cacheInfo.E; j++)
        {
            if(cache[i][j].timestamp>0)cache[i][j].timestamp++;
        }
    }
}
```

测试：

```
lyx@ubuntu:~/jsjxtsy/partA/cachelab-handout$ ./csim -v -s 4 -E 1 -b 4 -t traces/yi.trace
打开成功
 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
```

2.1.5. 加入-v 参数

操作和地址：

```
char opt;                           //操作类型    L  S  M
unsigned long address;              //地址
int block;                          //访问字节数
while (fscanf(file," %c %lx,%d", &opt, &address, &block) > 0)
{
    //printf("我读取的： %c %lx,%d\n",opt,address,block);
    if(cacheInfo.v==1)printf(" %c %lx,%d", opt,address,block);  //-v 操作
    switch(opt)
    {
        case 'I':               //不做操作
            break;
        case 'L':
            find(address);
            break;
        case 'M':
            find(address);
        case 'S':
            find(address);
            break;
    }
    time();                     //每次使用，时间戳整体加1
    if(cacheInfo.v==1)printf("\n");                    //-v 操作

}
fclose(file);
```

hit:

```
//printf("\ns=%d  tag=%lx\n",s,tag);
for(int i=0;i<cacheInfo.E;i++)            //直接在缓存中找到   hit
{
    if(cache[s][i].tag ==tag)
    {
        cache[s][i].timestamp = 1;        //表示最近一次使用过
        hit++;
        if(cacheInfo.v==1)printf(" hit");
        return;
    }
}
```

miss:

```
for(int i=0;i<cacheInfo.E;i++)
{
    if(cache[s][i].timestamp == 0)          //有空的（无效的)缓存
    {
        cache[s][i].tag = tag;             //当前地址映射到这个空的里  miss
        cache[s][i].timestamp = 1;         //表示最近一次使用过
        miss++;
        if(cacheInfo.v==1)printf(" miss");
        return;
    }
}
```

miss+eviction:

```
int max_stamp=0;        //当前最大的时间戳
int max_i;              //最大时间戳位置
for(int i=0;i<cacheInfo.E;i++)
{
    if(cache[s][i].timestamp >max_stamp)
    {
        max_stamp = cache[s][i].timestamp;
        max_i = i;
    }
}
miss++;
eviction++;
if(cacheInfo.v==1)printf(" miss eviction");
cache[s][max_i].tag = tag;
cache[s][max_i].timestamp = 1;        //被替换  更新时间戳  最近一次使用
```

测试：

```
lyx@ubuntu:~/jsjxtsy/partA/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
```

2.2. Part B

分析几个固定大小的矩阵并简单优化（从操作熟悉）

2.2.1. 参数设置

设置 s=5、E=1、b=5

故可以得出，缓存有 32（2^5）组，每组一行（块），每块 32（2^5）字节。有观察到，需要转置的矩阵存储的是 int 型数据，故每行（块）可储存 8 个数据。

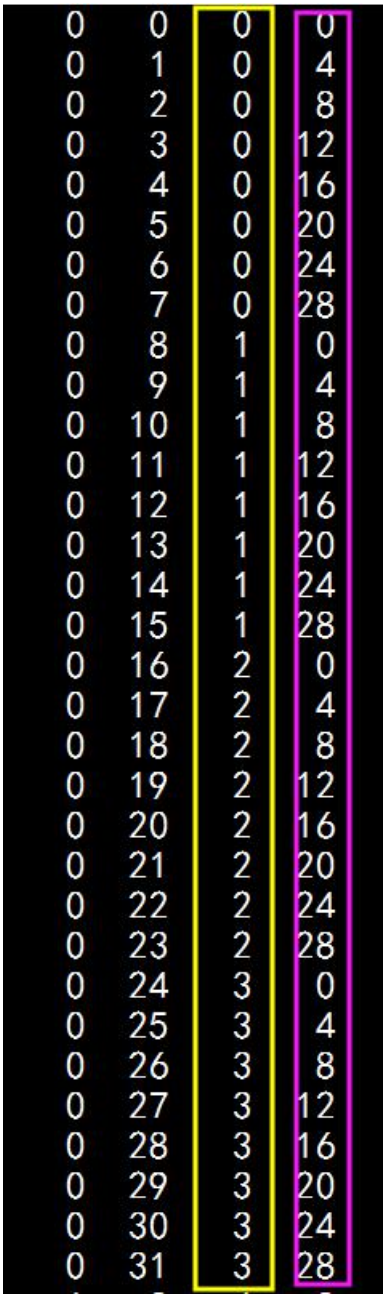
2.2.2. 32×32 矩阵

那么我们先得到组信息，就可以模拟出缓存的过程。

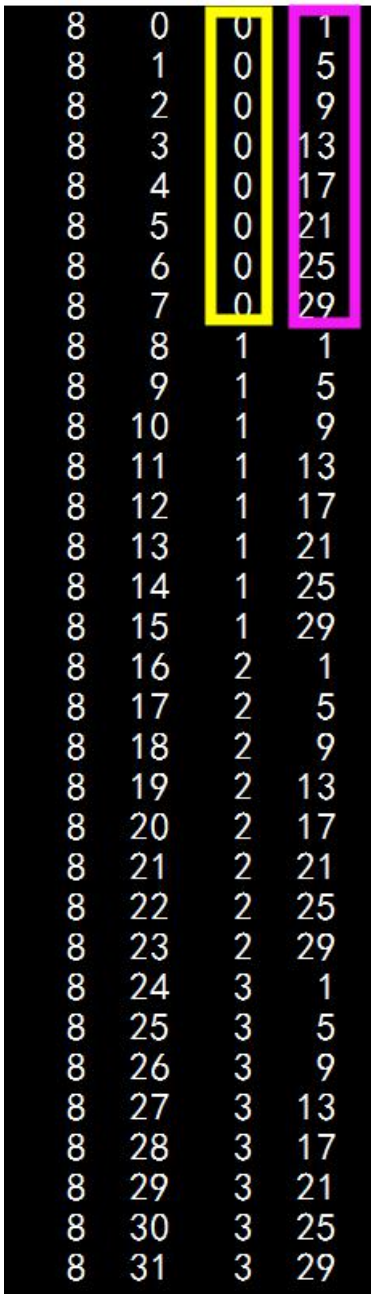
然后写了一个矩阵每个位置对应第几组的小代码（每组只有一行，后文不再强调）寻找规律。

```
#define N 32
#define M 32
int a[N][M];
int main(){
    int i=0,j=0,k=0,l=0;
    while(!(i==N-1&&j==M-1))
    {
        //cout<<setw(4)<<i<<setw(4)<<j<<setw(4)<<k<<endl;
        a[i][j]=l;
        j++;
        if(j==M)
        {
            j=0;
            i++;
        }
        k++;
        if(k==8){
            k=0;
            l++;
            if(l==32)l=0;
        }
    }
    //cout<<setw(4)<<i<<setw(4)<<j<<setw(4)<<k<<endl;
    a[i][j]=l;
    for(int i=0;i<N;i++){
        for(int j=0;j<M;j++){
            cout<<setw(4)<<i<<setw(4)<<j<<setw(4)<<a[i][j]<<setw(4)<<a[j][i]<<endl;
        }
    }
```

下图参数信息是：i ,j,A（i,j）位置对应的组，B（j,i）位置对应的值（为了数字敏感，未把数字对应含义的注释信息放在输出里）。



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



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

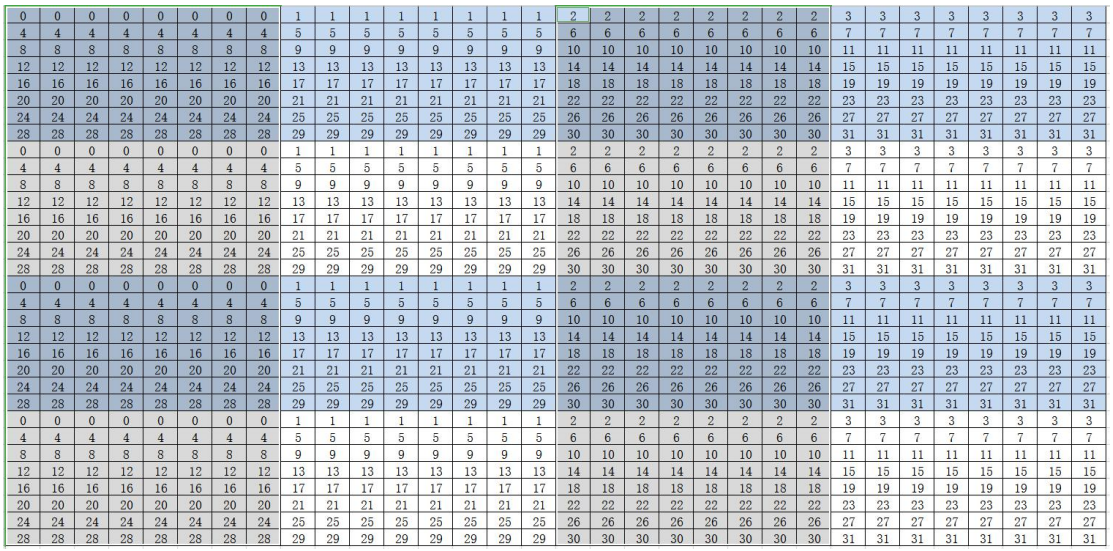
可以看到，对于逐行置换的置换策略，对于 A 数组是 cache 友好的，读到缓存的数都利用了一遍才丢出缓存只有刚进入缓存时 miss（1/8），但是对 B 数组就不是这样了，每次读一组，用其中一个数，然后就被替换，一直 miss（100%）。故这样大概是 32*32/8+32*32=1152 次 miss。

```
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
```

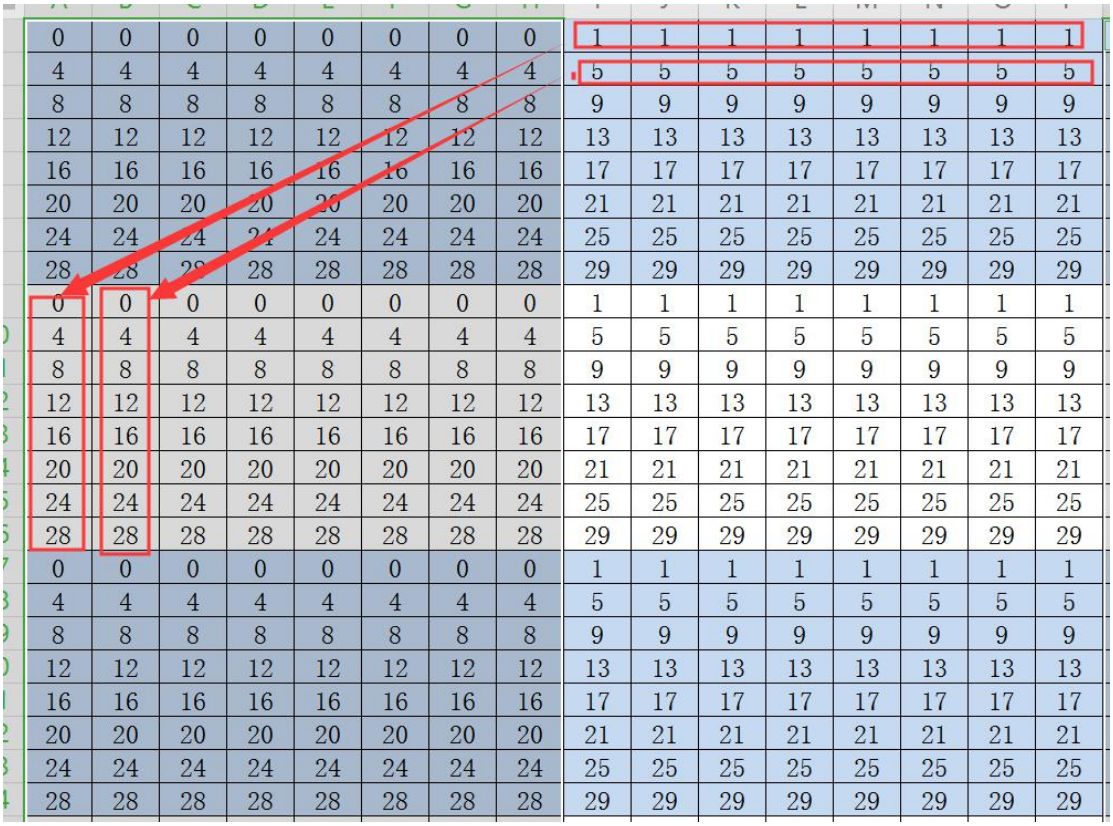
和预估的差不多，然后开始优化吧。

通过刚刚的分析，我们可以知道，只要每次操作数是 8 的整倍数（8,16,32）（一个组一起操作），那么对 A 缓存就是最友好状态（当然，可能为了整体最友好，会舍弃 A 的部分甚至全部友好型，需要做权衡，当然这里不需要）。那么 B 呢？

画出下图（A，B 都是 32×32，都是下图形状），然后分析。



（横的是 A，竖的是 B，但他们图一样，为了方便就在一张图上分析）



稍微细心点能够发现，对于 B 而言，他想要缓存友好，那么要求 A 分块读取，而不能按行读取，且此时也使 A 有良好的缓存友好性~~（虽然空间友好性被破坏，代码可读性变差）~~。

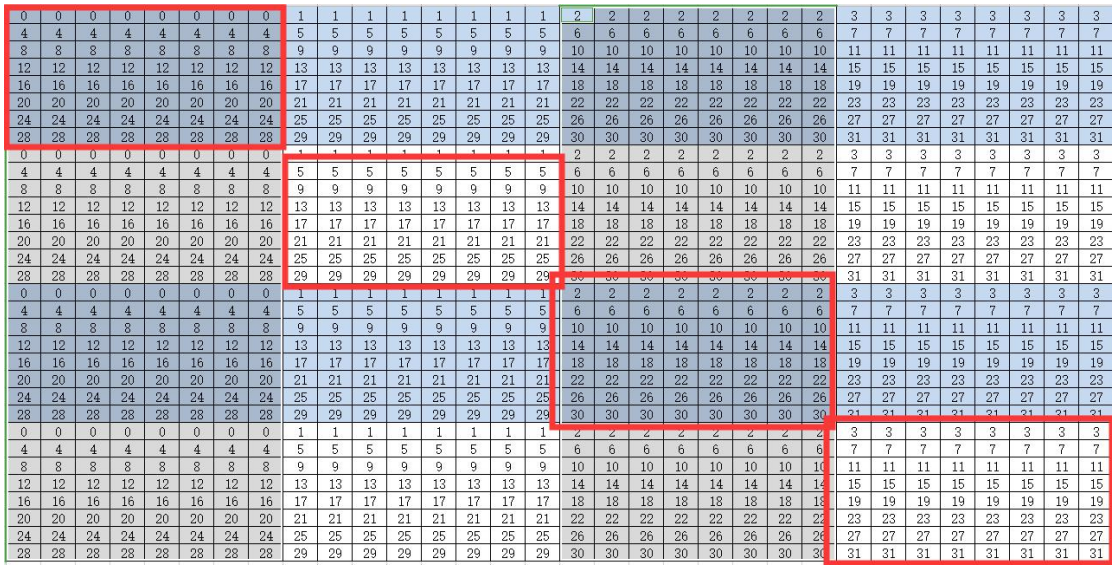
32*32/8+32*32/8=256。

```
char transpose_submit_desc[] = "Transpose submission";
void transpose_submit(int M, int N, int A[N][M], int B[M][N])
{
    //32×32  换成8×8
    int ii,jj,i,j;
    for(ii=0;ii<32;ii+=8){                  //行8为界
        for(jj=0;jj<32;jj+=8){              //列8为界
            for(i=ii;i<ii+8;i++){           //在小块里操作
                for(j=jj;j<jj+8;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:1710, misses:343, evictions:311
```

和预估的不一样（此时我满脑袋问号）。

仔细分析，终于找到了原因：对角线上的数据，A 和 B 都是同样的组，读 A 时，A 进入，然后写 B 驱赶 A，读 A 又驱赶 B……相互驱赶造成冲突不命中于是 miss 增加。



非对角线只有刚读（写）是 miss，故 1/8，占 16 份中的 12 份故非对角线：

(32*32*（12/16）/8)*2=192

对角线是 A 每行第一次读一次 miss，然后（转置位于对角线那个数据时）写 B 会驱逐一次 A，故再次读取会 miss（每块的最后一个例外，不需要再次读取）。

A：（7*2+1）*4=60

对角线的块中，B 分四种情况（以 0,4,8..28 为例）（以块为单位）：

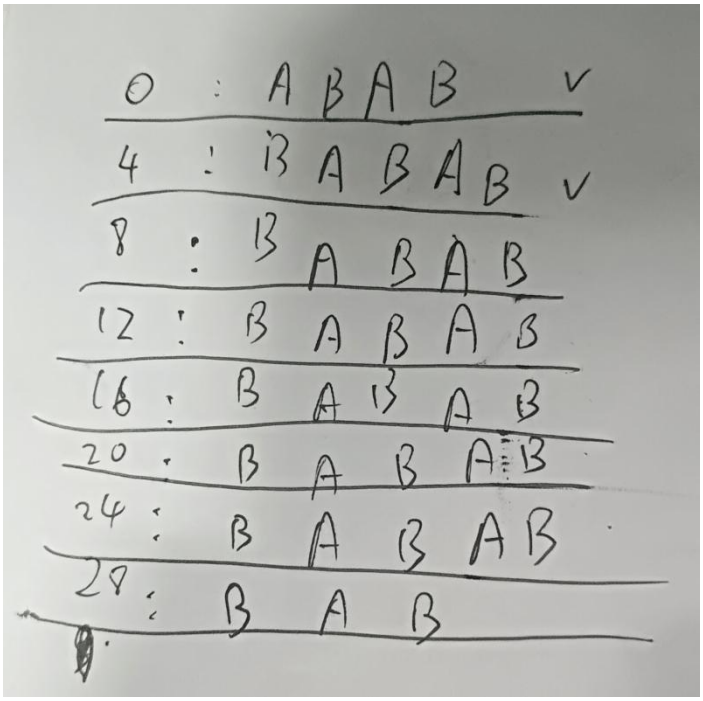
①块的首行（B 中）（对应 0 组）：应该是读入 A 后写 B 此时第一次 miss（针对 B），然后，A 会驱逐 0 组中的 B。再后面的 A 的第二行第一个去转置 B 的首行时，B 再次 miss。首行 2 次 miss。

②块的第 2~7 行，一开始是 B 占领缓存（B miss）（A 第一行转置到 B 时），然后的 2~7 行转置 B 的 2~7 列时，驱逐 B，然后 A 转置到对角线时，需要写 B，B miss，然后继续用 A 转置 B，直到 A 的这一行结束。但是 A 的下一行对角线之前的转置是对应 B 的上面行，所以最后还是 B 驱逐 A，B miss。2~6 行 3 次 miss。

③B 的尾行，与②相同，唯一的区别是最后写了 B，不会被 A 拿走（A 读完了），同时也不会因为下面行的转置导致（下面还有行＋又是对角线＝＞不是块末尾，后面还有元素＝＞这里还会被 A 驱逐）B 的再一次 miss。尾行 2 次 miss。

B:（2+3*6+2）*4=88

下图是第一块的缓存模拟。



192+60+88=340

接近 343。

```
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
```

处理好对角线位置

```
if(N==32){
    //32×32  换成8×8
    int ii,jj,i,j;
    int t1,t2,t3,t4,t5,t6,t7,t8;
    for(ii=0;ii<32;ii+=8){              //行8为界
        for(jj=0;jj<32;jj+=8){          //列8为界
            if(ii!=jj){                                 //非对角线
                for(i=ii;i<ii+8;i++){       //在小块里操作
                    for(j=jj;j<jj+8;j++){
                        B[j][i]=A[i][j];
                    }
                }
            }
            else{                                   //对角线  特殊处理
                for(int i=ii;i<ii+8;i++){           //用变量保存，避免了同时访问A，B数组，可以减少因相互驱逐造成的miss
                    //用变量报错A数组
                    t1=A[i][jj]  ;t2=A[i][jj+1];t3=A[i][jj+2];t4=A[i][jj+3];
                    t5=A[i][jj+4];t6=A[i][jj+5];t7=A[i][jj+6];t8=A[i][jj+7];
                    //将变量赋值给B数组
                    B[jj][i]=t1  ;B[jj+1][i]=t2;B[jj+2][i]=t3;B[jj+3][i]=t4;
                    B[jj+4][i]=t5;B[jj+5][i]=t6;B[jj+6][i]=t7;B[jj+7][i]=t8;
                }
            }
        }
    }
}
```

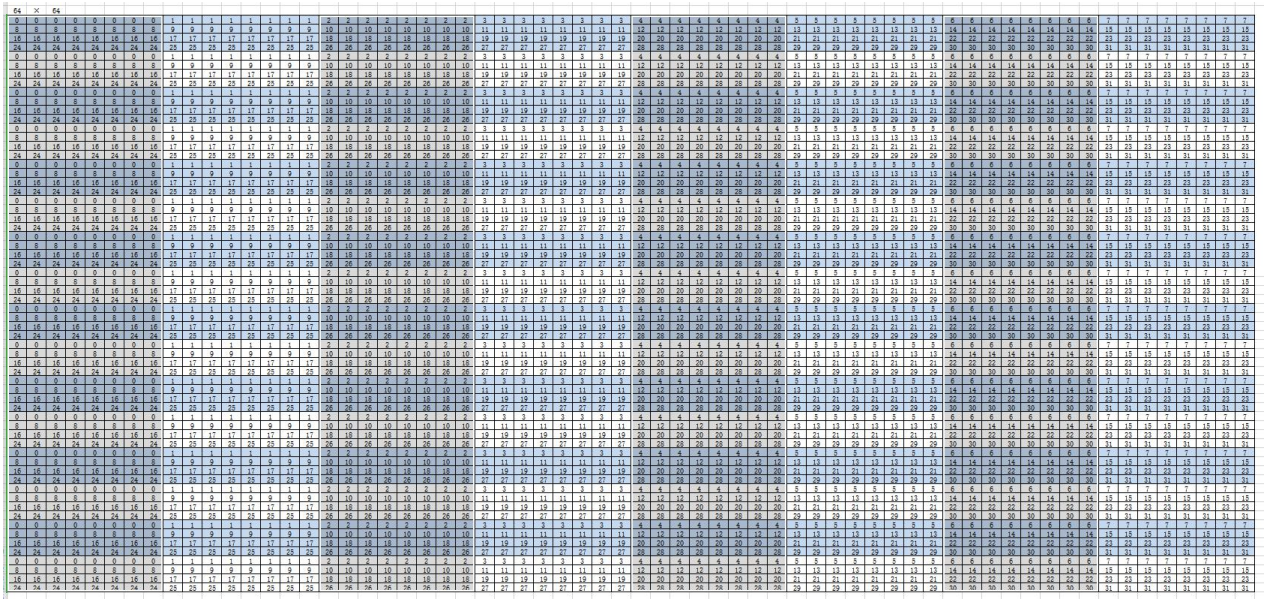
现在，非对角线还是 192 个，对角线上，A 每行 miss 一次，B 除了第一行，其余都 miss 两次，一次是 A 的第一行赋值给变量后，写进 B 的第一列，B miss，然后就是 A 每次都要驱逐 B（因为要赋值给变量，然后变量写进 B），B 驱逐 A 一次，B miss。所以除首行 B miss2 次，首行只有一次 B 驱逐 A（A 第一行造成）。故 8+2*7+1=23

192+23*4=284。

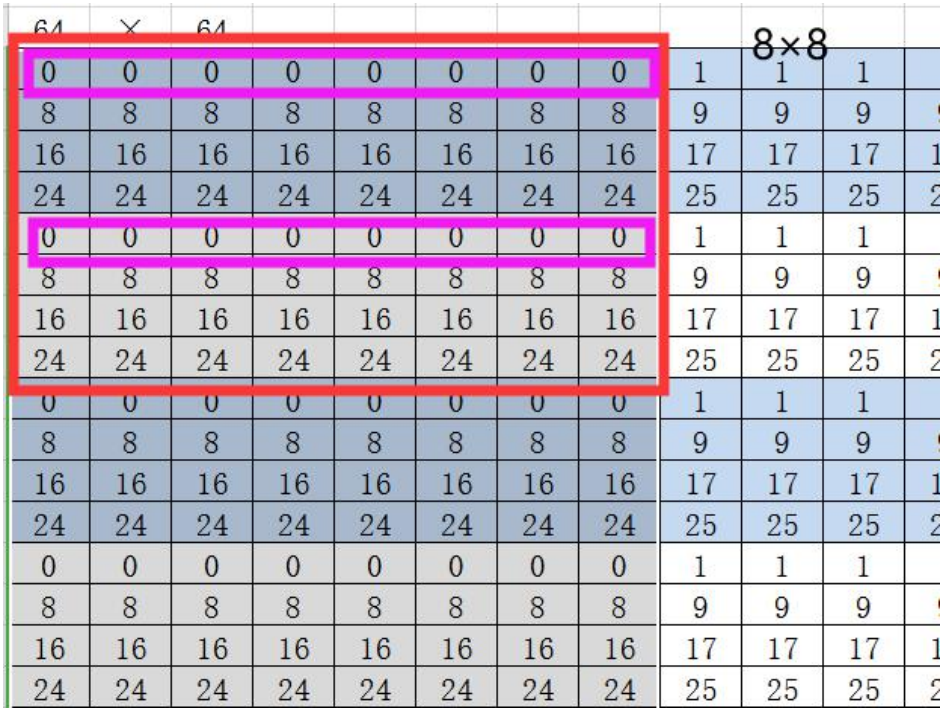
```
lyx@ubuntu:~/jsjxtsy/partA/cachelab-handout$ ./test-trans -N 32 -M 32

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: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
```

2.2.3. 64×64：

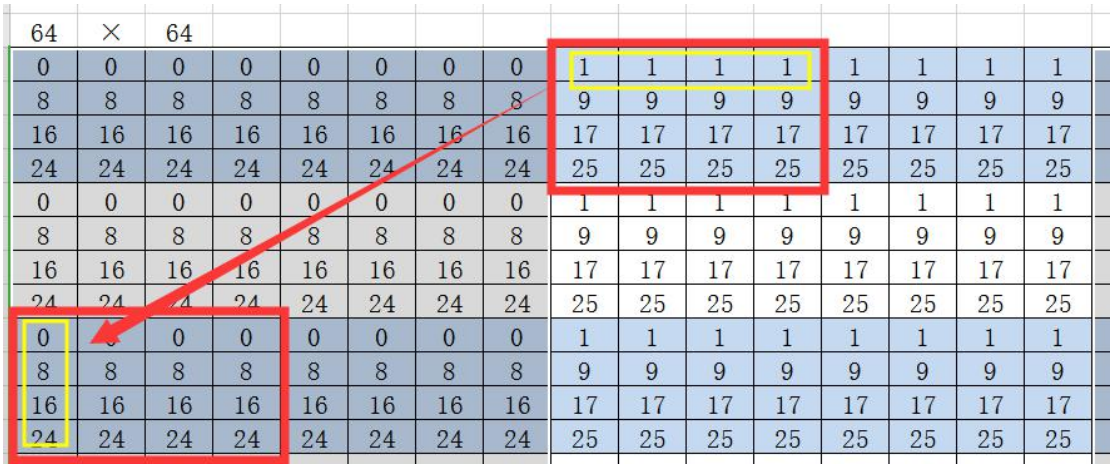
先拉出一个组信息（没想到粘上来这么小）：



可以看到，现在 8×8 分块已经不再像 32×32 里那么适用了，因为第 5 行就应经开始冲突了。

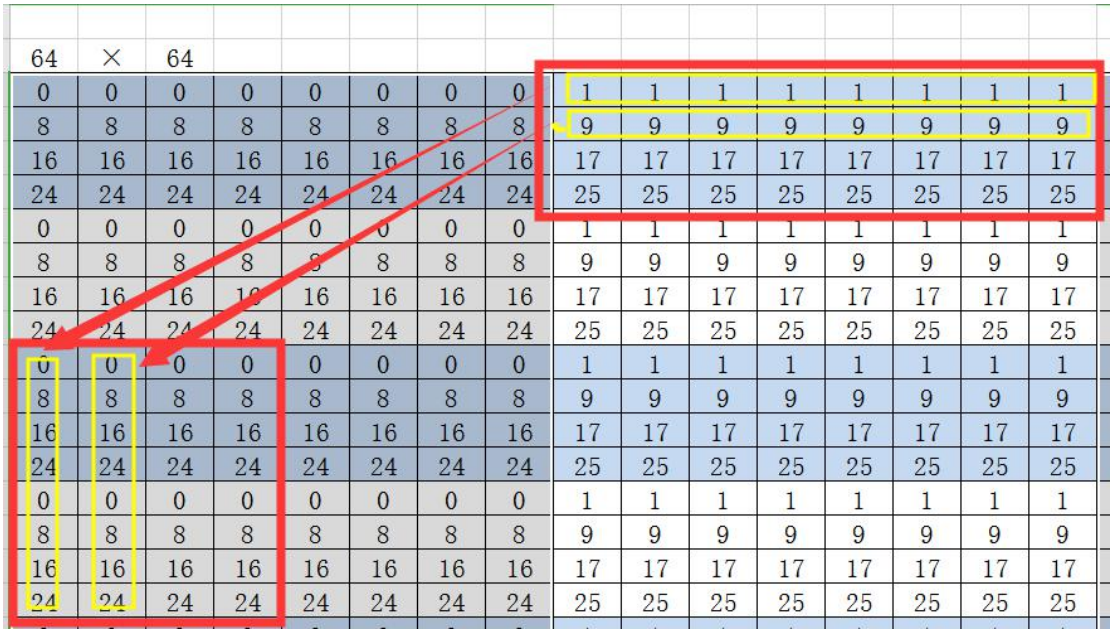


那么思考 4×8 和 4×4 分块。



可以看到，冲突没那么多，但是，缓存有 8 个数据量，每次就使用了 4 个，每次刚使用还会有 miss，所以 cache 友好还是不太好。

（4×4 和 4×8 类似，且 4×4　A，B 的利用看起来都是 4 个数据量，而事实上，读完 A 的前 4 个，又会马上读 A 的后 4 个，只要不被 B 驱逐，A 的利用量其实就是 8；4×8　A 表 8 个，B 表 1 个，B 一直疯狂 miss，故只研究 4×4 即可）



4×8

64	×	64													
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
8	8	8	8	8	8	8	8	9	9	9	9	9	9	9	9
16	16	16	16	16	16	16	16	17	17	17	17	17	17	17	17
24	24	24	24	24	24	24	24	25	25	25	25	25	25	25	25
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
8	8	8	8	8	8	8	8	9	9	9	9	9	9	9	9
16	16	16	16	16	16	16	16	17	17	17	17	17	17	17	17
24	24	24	24	24	24	24	24	25	25	25	25	25	25	25	25
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
8	8	8	8	8	8	8	8	9	9	9	9	9	9	9	9
16	16	16	16	16	16	16	16	17	17	17	17	17	17	17	17
24	24	24	24	24	24	24	24	25	25	25	25	25	25	25	25

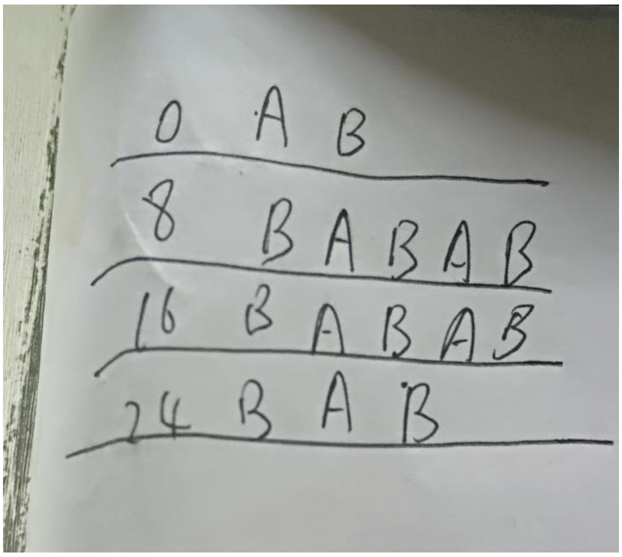
```
}
else if(N==64){
    int ii,jj,i,j;
    //int t1,t2,t3,t4,t5,t6,t7,t8;
    for(ii=0;ii<64;ii+=4){              //行4为界
        for(jj=0;jj<64;jj+=4){          //列4为界
                for(i=ii;i<ii+4;i++){        //在小块里操作
                    for(j=jj;j<jj+4;j++){
                        B[j][i]=A[i][j];
                    }
                }
        }
    }

}
```

非对角线：

B：只有刚读时 miss，1/4 的 miss。

A：看起来是 1 / 4 ，其实是 1 / 8 。

对角线：同上文分析 8×8，就不文字分析了，画出模拟图。



故总的 miss:(64*64/4+64*64/8)*(56/64) + 15*4*8=1824

```
trans.c
lyx@ubuntu:~/jsjxtsy/partA/cachelab-handout$ ./test-trans -N 64 -M 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:6306, misses:1891, evictions:1859
```

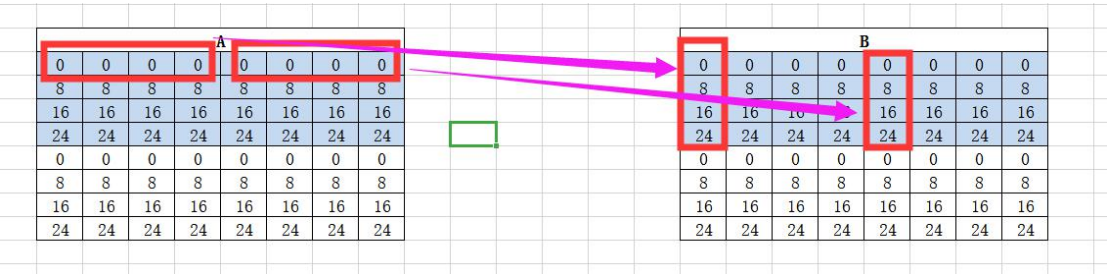
与计算结果相差不大（8×8 分成 4×4 肯定不能计算出 4×4 的，然后扩大 4 倍，肯定互相有影响，不过不影响大概计算）

另，对角线部分也可像 32×32 时那样优化，但是因为非对角线都超过 1300 了（1344），所以暂时没必要在这个基础上优化。

仔细分析，64×64 的两个矩阵，单单第一次的 miss 都有 64*64*2/8=1024 次了，非常接近 1300 了。而 4×4 对缓存的利用还是不够高，所以结果离 1300 还差的远，然后又尝试了一些其他方法，都达不到要求（而且差得远，觉得这个真的难），然后就上网看了看别的大神怎么做的。然后发现了一种，让我惊叹的做法——把 B 当一个“缓存”！

```
int i,j,ii,jj;
int t1,t2,t3,t4,t5,t6,t7,t8;          //为了减少A，B冲突不命中
for(ii=0;ii<64;ii+=8){                //因为cache的行（块）是8，所以为了充分利用cache，8×8
    for(jj=0;jj<64;jj+=8){            //下面的操作  针对的是每个8×8
        //按A的行 ，B的列
        for(i=ii,j=jj;i<ii+4;i++){   //A 的上半部分 与B的上半部分（4×8）
            //先储存A的值，减少冲突不命中
            t1=A[i][j];
            t2=A[i][j+1];
            t3=A[i][j+2];
            t4=A[i][j+3];
            t5=A[i][j+4];
            t6=A[i][j+5];
            t7=A[i][j+6];
            t8=A[i][j+7];
            //B左上角，此时的值直接转置
            B[j][i]=t1;
            B[j+1][i]=t2;
            B[j+2][i]=t3;
            B[j+3][i]=t4;
            //B的右上角，此值不是最终转置的值，只是暂时存放（B成了缓存容器）
            B[j][i+4]=t5;
            B[j+1][i+4]=t6;
            B[j+2][i+4]=t7;
            B[j+3][i+4]=t8;
        }
```

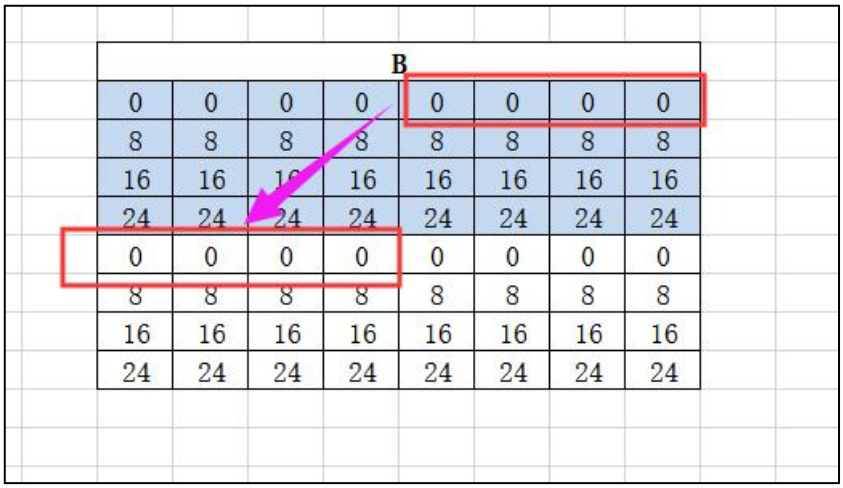
这个步骤的过程

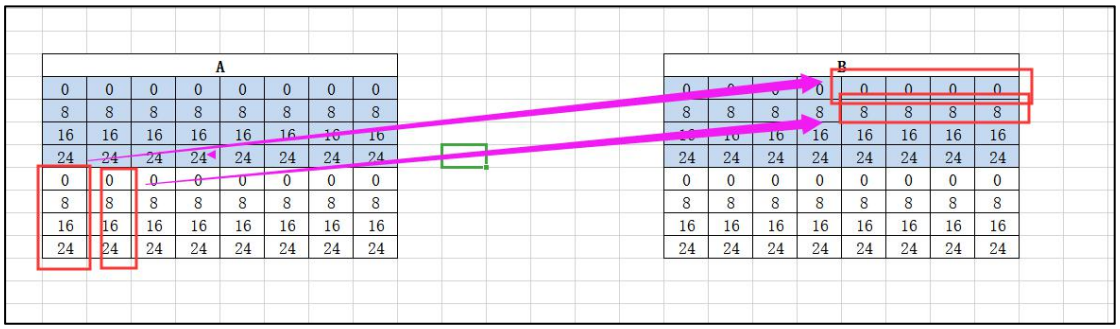


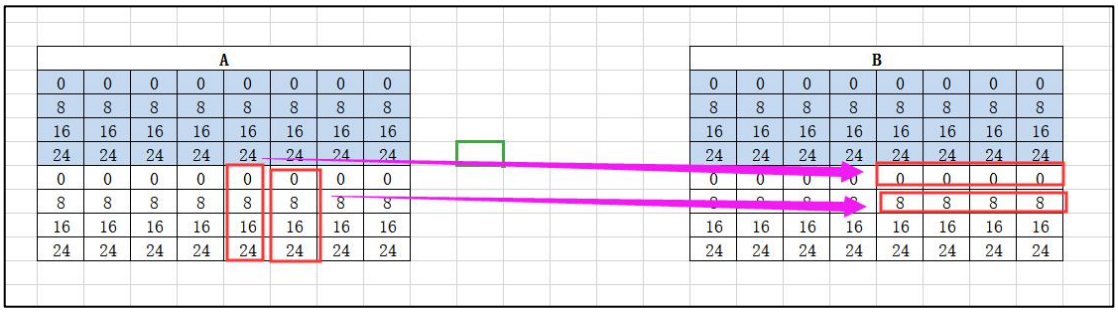
按这个过程把上一部分 A 读完，并完成转置（此时 B 的左上已经转置完成，右上只是充当储存容器）。

```
//按B的行，A的列
for(j=jj+4,i=ii;j<jj+8;j++){ //读A的下半部分（2个4×4），处理B的3个（转置未完成的）4×4
    //变量储存A的左下 ，减少冲突不命中
    t1=A[i+4][j-4];
    t2=A[i+5][j-4];
    t3=A[i+6][j-4];
    t4=A[i+7][j-4];
    //读取一开始储存在B右上的信息
    t5=B[j-4][i+4];
    t6=B[j-4][i+5];
    t7=B[j-4][i+6];
    t8=B[j-4][i+7];
    //B的右上重新赋值，这才是正确的转置信息
    B[j-4][i+4]=t1;
    B[j-4][i+5]=t2;
    B[j-4][i+6]=t3;
    B[j-4][i+7]=t4;
    //B的左下   对应A的右上（一开始放在B右上）
    B[j][i]=t5;
    B[j][i+1]=t6;
    B[j][i+2]=t7;
    B[j][i+3]=t8;
    //B的右下
    B[j][i+4]=A[i+4][j];
    B[j][i+5]=A[i+5][j];
    B[j][i+6]=A[i+6][j];
    B[j][i+7]=A[i+7][j];
}
```

这个步骤主要是以下部分：



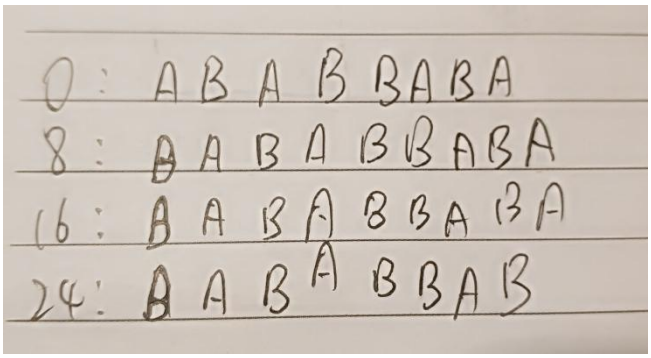




分析：

64×64，问题主要是冲突不命中和 cache 利用率的一个矛盾，这个算法，较为完美的解决了这个矛盾，读 A 时就读 A，能不冲突操作的，直接操作，不能的，先把 A 的信息储存在已在缓存的 B（换成 B 的其他两块都不行，会增加加载进缓存的 miss）充分利用缓存，同时保证不需要再读 A 的这行，减少冲突不命中。

这样的话，非对角线只有冷不命中（真的厉害），对角线有 34 个 miss（分析和上文类似，只给出我的模拟图）



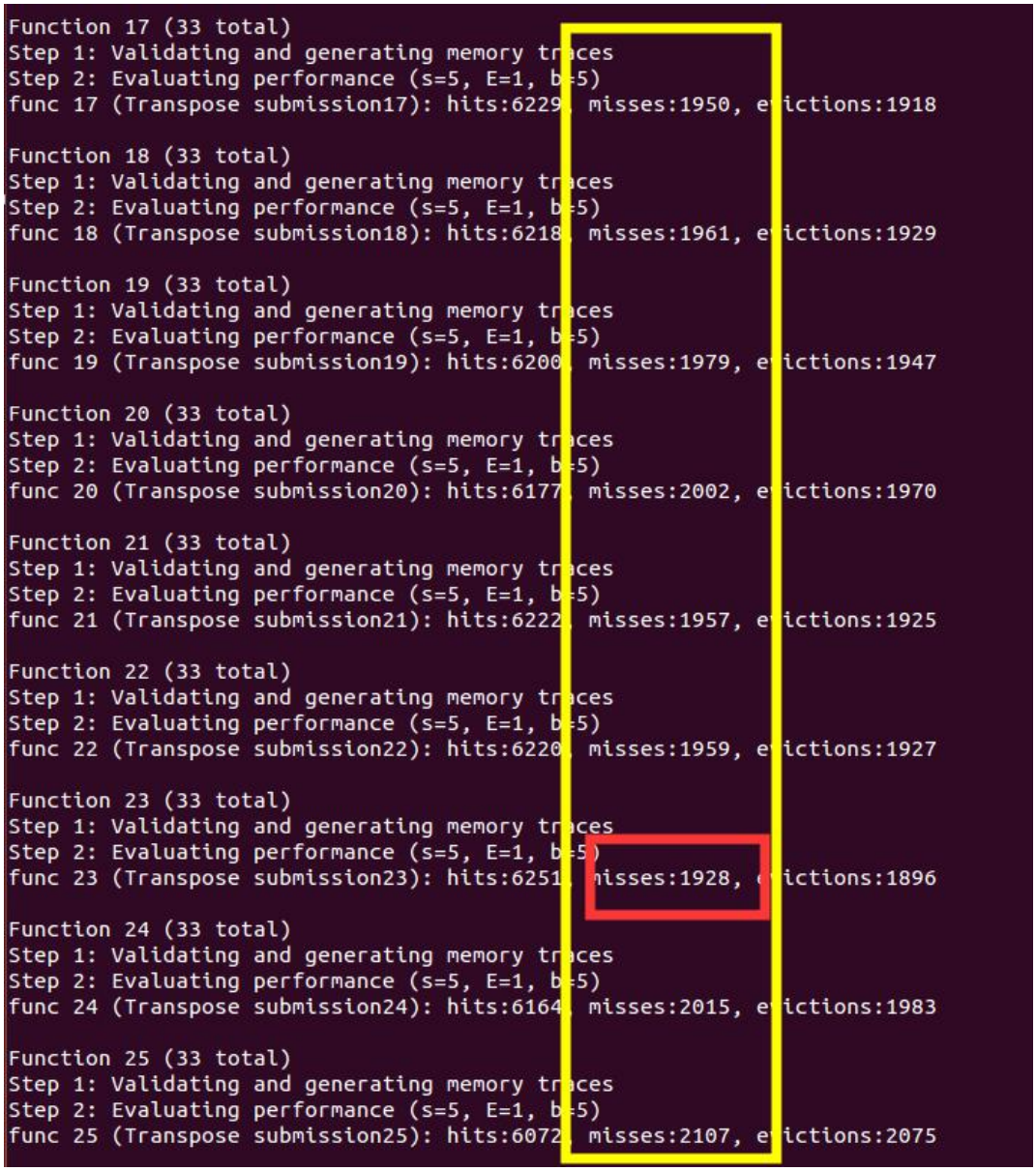
那么，最终结果为：8*（64-8）*2+34*8=1168。

```
lyx@ubuntu:~/jsjxtsy/partA/cachelab-handout$ ./test-trans -N 64 -M 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:9074, misses:1171, evictions:1139
```

与计算较为统一。

2.2.4. 61×67：

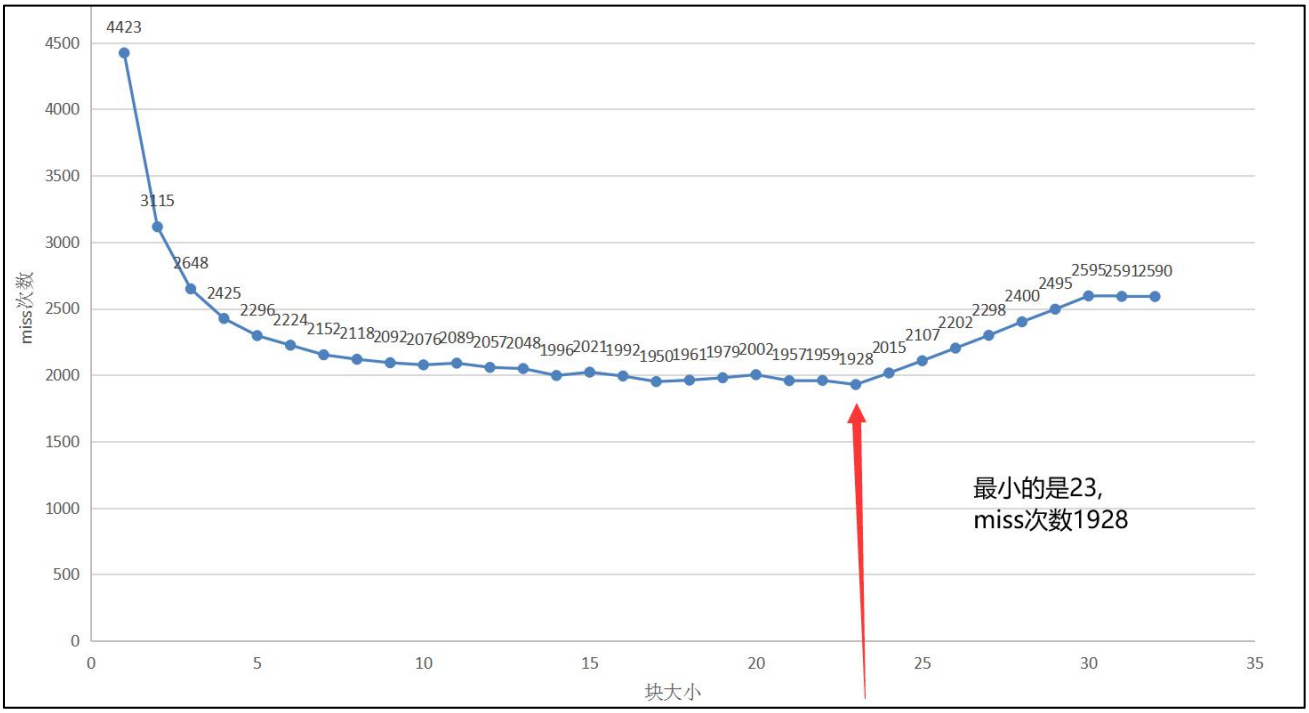
这个的分组较乱，无规律，难以找到规律看出用多大的块合适，但是由于要求比较松（2000，几乎是单单冷不命中的 2 倍），因此无需考虑多余处理，直接尝试转置操作，尝试换用不同的边长分块即可。



```
else if(N==67)
{
    int i, j, k, l;
    int size = 23;
    for (i = 0; i < N; i += size) {
        for (j = 0;  j < M; j += size) {
            for (k = i; k < i + size && k < N; k++) {
                for (l = j; l < j + size && l < M; l++) {
                    B[l][k] = A[k][l];
                }
            }
        }
    }
}
```

下图是块大小和 miss 次数：

	1	2	3	4	5	6	7	8	
	4423	3115	2648	2425	2296	2224	2152	2118	
	9	10	11	12	13	14	15	16	
	2092	2076	2089	2057	2048	1996	2021	1992	
	17	18	19	20	21	22	23	24	
	1950	1961	1979	2002	1957	1959	1928	2015	
	25	26	27	28	29	30	31	32	
	2107	2202	2298	2400	2495	2595	2591	2950	



所以选取块大小 23 即可。

然后再尝试优化一下对角线（冲突不命中），但是因为对不齐，对角线不一定有用。

61	×	67							
0	0	0	0	0	0	0	0	1	1
8	8	8	8	8	9	9	9	9	9
16	16	17	17	17	17	17	17	17	17
25	25	25	25	25	25	25	26	26	26

但就能看到的，对角线还是有，可尝试一下。

```
lyx@ubuntu:~/jsjxtsy/partA/cachelab-handout$ ./test-trans -N 67 -M 61

Function 0 (4 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:6258, misses:1921, evictions:1889
```

可以看到，还是减少了几个。

2.3. Part C

优化矩阵转置

1. 首先这个程序局部性不太好，要么读的 src 缓存不友好，要么写的 dst 缓存不友好，优先考虑写（即让 dst 缓存友好）。

```
char rotate1_descr[] = "rotate1: Current working version";
void rotate1(int dim, pixel *src, pixel *dst)
{
    register int i, j;

    for (i = 0; i < dim; i++)
        for (j = 0; j < dim; j++)
            dst[RIDX(i, j, dim)] = src[RIDX(j, dim-i-1, dim)];
}
```

```
Speedup          6.5     13.0     7.7     5.7     8.4     7.9

Rotate: Version = rotate1: Current working version:
Dim              64      128      256     512     1024    Mean
Your CPEs        2.1     2.3      3.0     4.4     7.0
Baseline CPEs    14.7    40.1     46.4    65.9    94.5
Speedup          7.0     17.7     15.5    14.9    13.6    13.1
```

效果还行，程序 CPE 平均提高 13.1 倍。

2. 尝试分成 4*4 的小块，提高空间局部性。

```
char rotate4_descr[] = "rotate4: Current working version";
void rotate4(int dim, pixel *src, pixel *dst)
{
    register int i, j, ii, jj;
    for(i=0; i<dim; i+=4)
    {
        for(j=0; j<dim; j+=4)
        {
            for(ii=i; ii<i+4; ii++) {
                for(jj=j; jj<j+4; jj++)
                    dst[RIDX(ii, jj, dim)] = src[RIDX(jj, dim-ii-1, dim)];
            }
        }
    }

}
```

```
Member 1: 骆一鑫
Student ID 1: 201726010123

Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             64      128     256     512     1024    Mean
Your CPEs       2.1     3.0     5.8     11.2    10.8
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         7.0     13.4    8.0     5.9     8.8     8.3

Rotate: Version = rotate4: Current working version:
Dim             64      128     256     512     1024    Mean
Your CPEs       2.1     2.2     2.7     2.9     6.1
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         7.0     18.2    17.0    22.4    15.4    15.0
```

效果一般，程序 CPE 平均提高 15 倍。

3. 采用 32*32 分块，4*4 路循环展开。

```
void rotate5(int dim, pixel *src, pixel *dst)
{
    register int i, j, ii, jj;
    for(ii=0; ii<dim; ii+=32)
    {
        for(jj=0; jj<dim; jj+=32)
        {
            for(i=ii; i<ii+32; i+=4) {
                for(j=jj; j<jj+32; j+=4){
                    dst[RIDX(dim-1-j, i, dim)] = src[RIDX(i,j, dim)];
                    dst[RIDX(dim-2-j, i, dim)] = src[RIDX(i,j+1, dim)];
                    dst[RIDX(dim-3-j, i, dim)] = src[RIDX(i,j+2, dim)];
                    dst[RIDX(dim-4-j, i, dim)] = src[RIDX(i,j+3, dim)];

                    dst[RIDX(dim-1-j, i+1, dim)] = src[RIDX(i+1,j, dim)];
                    dst[RIDX(dim-2-j, i+1, dim)] = src[RIDX(i+1,j+1, dim)];
                    dst[RIDX(dim-3-j, i+1, dim)] = src[RIDX(i+1,j+2, dim)];
                    dst[RIDX(dim-4-j, i+1, dim)] = src[RIDX(i+1,j+3, dim)];

                    dst[RIDX(dim-1-j, i+2, dim)] = src[RIDX(i+2,j, dim)];
                    dst[RIDX(dim-2-j, i+2, dim)] = src[RIDX(i+2,j+1, dim)];
                    dst[RIDX(dim-3-j, i+2, dim)] = src[RIDX(i+2,j+2, dim)];
                    dst[RIDX(dim-4-j, i+2, dim)] = src[RIDX(i+2,j+3, dim)];


                    dst[RIDX(dim-1-j, i+3, dim)] = src[RIDX(i+3,j, dim)];
                    dst[RIDX(dim-2-j, i+3, dim)] = src[RIDX(i+3,j+1, dim)];
                    dst[RIDX(dim-3-j, i+3, dim)] = src[RIDX(i+3,j+2, dim)];
                    dst[RIDX(dim-4-j, i+3, dim)] = src[RIDX(i+3,j+3, dim)];
                }
            }
        }
    }

}
```

测试效果如下：

```
Student ID 1: 201726010123

Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             64      128     256     512     1024    Mean
Your CPEs       2.3     3.1     5.9     11.7    11.7
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         6.4     12.9    7.8     5.7     8.1     7.8

Rotate: Version = rotate5: Current working version:
Dim             64      128     256     512     1024    Mean
Your CPEs       2.7     2.6     2.9     3.3     6.4
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         5.5     15.6    15.8    20.2    14.7    13.2
```

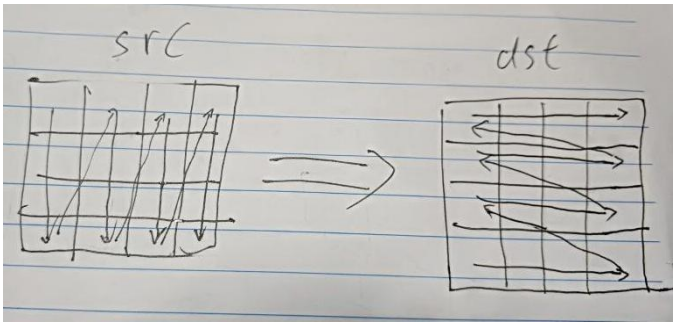
效果还是不错的，平均提高 13 倍。但是没想到还没有上一个 4*4 的效果好

4. 利用循环分块、循环展开以及消除不必要的存储器引用，这里采用 32*1 分块，32 路循环展开。

```
char rotate6_descr[] = "rotate6: Current working version"
void rotate6(int dim, pixel *src, pixel *dst)
{
    int i, j;
    dst += (dim-1)*dim;
    for (i = 0; i < dim; i+=32){
            for (j = 0; j < dim; j++){
                *dst=*src;
                src+=dim;
                dst+=1;

                *dst=*src;
                src+=dim;
                dst+=1;

                *dst=*src;
                src+=dim;
                dst+=1;

                *dst=*src;
                src+=dim;
                dst+=1;

                *dst=*src;
                src+=dim;
                dst+=1;

                *dst=*src;
                src+=dim;
                dst+=1;

                *dst=*src;
                src+=dim;
                dst+=1;

                *dst=*src;
                src+=dim;
                dst+=1;
```

```
                *dst=*src;
                src+=dim;
                dst+=1;

                *dst=*src;
                src+=dim;
                dst+=1;

                *dst=*src;
                src+=dim;
                dst+=1;

                *dst=*src;
                src+=dim;
                dst+=1;

                *dst=*src;
                src+=dim;
                dst+=1;

                *dst=*src;
                src+=dim;
                dst+=1;

                *dst=*src;
                src+=dim;
                dst+=1;

                *dst=*src;
                src+=dim;
                dst+=1;

                *dst=*src;
                src+=dim;
```

```
                *dst=*src;
                src+=dim;
                dst+=1;

                *dst=*src;
                src+=dim;
                dst+=1;

                *dst=*src;
                src++;
                src -= (dim<<5)-dim;
                dst-=31+dim;
                }
            dst+=dim*dim;
            dst+=32;
            src += (dim<<5)-dim;
        }
}
```



先将 src 第一块中（32 行 dim 列）的第一列（32 行）转为 dst 第一块（dim 行 32 列）中的最后一行（32 列），

再将 src 第一块中（32 行 dim 列）的第二列（32 行）转为 dst 第一块（dim 行 32 列）中的倒数第二行（32 列），

……

再将 src 第二块中（32 行 dim 列）的第一列（32 行）转为 dst 第二块（dim 行 32 列）中的最后一行（32 列），

……

```
Member 1: 骆一鑫
Student ID 1: 201726010123

Rotate: Version = naive_rotate: Naive baseline implementation:
Dim             64      128     256     512     1024    Mean
Your CPEs       2.3     3.2     6.1     11.4    11.7
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         6.4     12.7    7.7     5.8     8.1     7.8

Rotate: Version = rotate6: Current working version:
Dim             64      128     256     512     1024    Mean
Your CPEs       2.0     2.0     2.1     2.0     4.5
Baseline CPEs   14.7    40.1    46.4    65.9    94.5
Speedup         7.4     20.4    22.6    32.8    21.2    18.8
```

效果很理想，大约提高 18 倍。