CacheLab 实验报告

10235501419 李佳亮

本实验分为两部分，第一部分要求模拟一个cache，第二部分要求根据给定的缓存结构优化矩阵转置函数。

Part A: Writing a Cache Simulator

1. 任务说明与初步思路

根据实验文档，在本部分，我们要在`csim.c`中编写代码，以`valgrind memory trace`文件作为输入，在控制台输出总的`hit`，`miss`和`eviction`数，实现handout中`csim-ref`的效果：

```
linux> ./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

linux> ./csim-ref -h
• -h: Optional help flag that prints usage info
• -v: Optional verbose flag that displays trace info
• -s <s>: Number of set index bits (S = 2s is the number of sets)
• -E <E>: Associativity (number of lines per set)
• -b <b>: Number of block bits (B = 2b is the block size)
• -t <tracefile>: Name of the valgrind trace to replay
```

而`valgrind memory trace`文件内容格式如下：

```
I 0400d7d4,8
 M 0421c7f0,4
 L 04f6b868,8
 S 7ff0005c8,8
/* Each line denotes one or two memory accesses. The format of each line is
[space(only for 'I')]operation address,size */
```

拆分本任务，可从本任务中提取几个关键部分：

- 读取命令行参数；
- 模拟高速缓存的数据结构；
- 读取`trace`文件并模拟缓存过程。

通读实验报告后，我们可以获得下面几个提示：

- 只需要模拟`S(data store)`，`L(data load)`和`M(data modify, i.e. data load following by data store)`这三个指令；

- 缓存采用`LRU(least-recently used)`的替换策略来驱逐缓存块；
- 由于缓存结构由命令行参数决定，故会用到`malloc()`函数；
- 输出采用`printSummary`函数输出，`-v`和`-h`命令行参数可选择不实现；
- 可以采用`unistd.h`与`getopt.h`中的`getopt()`函数读取命令行参数，可参考`man 3 getopt`。

在写代码之前，会有如下问题：

- 如何管理可变个命令行参数?
  - 使用`getopt()`函数。
- `S`，`L`和`M`的指令有何区别?
  - 本例中`S`和`L`指令没有区别，都是检查缓存中是否有地址中的元素，有则`hit`无则`miss`后载入缓存；
  - `M = L + S`，故事实上第二个`S`一定会`hit`；
  - **容易迷惑的点：**`miss`后会把相应数据载入缓存，但不会再访问缓存命中并打印`hit`；`eviciton`后会用`LRU`策略驱逐块，之后会再次访问缓存不命中、再次打印`miss`，而同样地，`miss`后会把相应数据载入缓存，但不会再访问缓存命中打印`hit`。
- 选用什么数据结构来实现`cache`，需要实现`cache`的哪部分功能?
  - 初步思路是用结构体实现`cacheline`，再以二维数组的形式来组织成`cache`；
  - `cache`的功能不必完全实现，由于不需要存入具体的数据，因此在`cacheline`结构体中不必添加`B`和用于存放数据的成员数组`cacheblock`；
  - 可以注意到，参数`b`的唯一作用就是计算标记位`t`的值来解析地址，与`b`相关的参数也不必集成在`cache`结构体中。

下面分部分完成本任务。

2. 实验过程

部分 1 - 读取命令行参数

根据实验文档，我们要接收的命令行参数为：

```
Usage: ./csim [-hv] -s <s> -E <E> -b <b> -t <tracefile>
```

我们采用`getopt`函数解析命令行参数，使用方法可参考官方文档或Linux下getopt()函数的简单使用。其中，为了方便调试，在这里我们实现`-v`功能，用一个全局变量`int showDetails`来判断未来模拟过程中是否输出细节信息。

```
int showDetails = 0;

void readCommandline(int argc, char* argv[], int *s, int *E, int *b, char
*tracefile){
    const char *optstr = "hvs:E:b:t:";
    int opt;
    while((opt = getopt(argc, argv, optstr)) != -1){
        switch(opt){
            case 'h':
                printHelpMenu();    // 打印帮助菜单
                exit(0);
            case 'v':
```

```
                showDetails = 1;    // 设置全局变量showDetails为真
                break;
            case 's':
                *s = atoi(optarg);   // 外部变量 optarg - 指向当前选项参数字符串的指针
                break;
            case 'E':
                *E = atoi(optarg);
                break;
            case 'b':
                *b = atoi(optarg);
                break;
            case 't':
                strncpy(tracefile, optarg, strlen(optarg));
                break;
        }
    }
}
```

部分 2 - 模拟高速缓存的数据结构

高速缓存的最小结构单元是行，即`cacheline`，它可以用结构体来实现，含有以下成员变量：

```
typedef struct Cache{
    int valid;          // 有效位的值
    unsigned tag;       // 标记位的值
    int recentTimeUsed; // 用于LRU策略，记录最后一次被访问的'时间'
}Cacheline;
```

`E`个`cacheline`组合成一个`set`，`S`个`set`组合成一个`cache`。因此，`cache`可视为是一个`S*E`的`cacheline`二维数组。为了还要集成其他的有关`cache`的信息如`S`，`E`和`B`，我们把`cache`也做成一个结构体，成员变量如下：

```
typedef struct Cache{
    int S, E;
    Cacheline **cachelines; // cacheline的二维数组
}Cache;
```

下面为`Cacheline`和`Cache`依次编写初始化方法。

```
void initCacheline(Cacheline *cacheline){
    cacheline->valid = 0;
    cacheline->tag = 0;
    cacheline->recentTimeUsed = 0;
}
void initCache(int S, int E, Cache *cache){
    cache->S = S;
    cache->E = E;
    cache->cachelines = (Cacheline **)malloc(sizeof(Cacheline *)*S);
    for(int i=0;i<S;i++){
        cache->cachelines[i] = (Cacheline *)malloc(sizeof(Cacheline)*E);
        // ★易错！注意 &cache->cachelines[i][j] 和 cache->cachelines[i]的关系
        for(int j=0;j<E;j++){
            initCacheline(&cache->cachelines[i][j]);
        }
```

```
    }
}
```

由于初始化`Cache`对象的过程中用了动态内存分配，需要为`Cache`编写释放内存的函数：

```
void freeCache(Cache *cache){
    for(int i=0;i<cache->S;i++){
        free(cache->cachelines[i]);
        cache->cachelines[i] = NULL;
    }
    free(cache->cachelines);
    cache->cachelines = NULL;
}
```

其他'成员函数'将在下一部分实现。

部分 3 - 读取`trace`文件并模拟高速缓存过程

首先，按行从`trace`文件中得到指令及其对应的地址，并通过第二章所学的位运算从地址中解析出`tag`，`set index`的信息，并进行模拟。前面已经知道`L`和`S`所做的操作实际上相同，而`M=L+S`，故只需要编写一个函数`simulate()`来模拟收到指令后缓存的操作。

```
// result[3]：记录总的hit, miss, eviction数的数组
void doCacheSimulation(char *tracefile, int s, int b, Cache *cache, int
result[3]){
    // 读取数据文件
    FILE *fp = fopen(tracefile, "r");
    char buf[256];
    char type;
    char *addr; // 存储address的字符串临时变量
    unsigned address, tag, setIndex;

    while(fgets(buf, 255, fp)){
        // 不进行"I"指令
        if(buf[0] != ' '){
            continue;
        }
        if(buf[strlen(buf)-1] == '\n'){
            buf[strlen(buf)-1]  = '\0';
        }
        if(showDetails)
            printf("\n%s", buf);
        // 分解行指令
        type = buf[1];
        /* 易错！ address is hex! */
        addr = strtok(buf+3, ",");
        char *endptr;
        address = strtol(addr, &endptr, 16);
        // 拆解地址：t位tag位，s位set index，b位block offset
        tag = address >> (s+b);
        setIndex = (address >> b) & ((1<<s)-1);  // (1<<s)-1=2^s-1=0b11..1（s个1）

        switch(type){
```

```
            // L for data load, S for data store
            case 'L':
            case 'S':
                simulate(setIndex, tag, result, cache);
                break;
            // M for data modify
            case 'M':
                simulate(setIndex, tag, result, cache);
                simulate(setIndex, tag, result, cache);
                break;
        }
    }
}
```

下面这四个函数用于模拟访问缓存的操作。`checkCache()` 访问缓存并返回缓存状态 `MISS`，`HIT` 或 `EVICTION`（用宏定义映射到 `0`，`1`，`2`）；`loadCache()` 在 `miss` 后向缓存中加载相应数据；`LRU_evict()` 驱逐最近最少使用的块，维护一个全局变量 `int timeflow` 来模拟时间流动，记录各个块的最后访问时间。

```
#define HIT 0
#define MISS 1
#define EVICTION 2

int timeflow = 0;   // 模拟时间流动

// ...

// 返回缓存的状态
int checkCache(unsigned setIndex, unsigned tag, Cache *cache){
    int validCount = 0;
    for(int i=0;i<cache->E;i++){
        Cacheline *cur = &cache->cachelines[setIndex][i];
        if(cur->valid){
            if(cur->tag == tag){
                if(showDetails)
                    printf(" hit");
                /** 易遗漏！hit之后也要修改timeflow！ */
                cur->recentTimeUsed = ++timeflow;
                return HIT;
            }
            validCount++;
        }
    }
    if(validCount==cache->E){
        if(showDetails)
            printf(" eviction");
        return EVICTION;
    }
    if(showDetails)
        printf(" miss");
    return MISS;
}
// miss后向cache中加载内容
void loadCache(unsigned tag, unsigned setIndex, Cache *cache){
    for(int i=0;i<cache->E;i++){
```

```
        Cacheline *cur = &cache->cachelines[setIndex][i];
        if(cur->valid == 0){
            cur->valid = 1;
            cur->tag = tag;
            cur->recentTimeUsed = ++timeflow;
            return;
        }
    }
}
// 在cache的setIndex组中查找最近最少使用的cacheline
void LRU_evict(unsigned setIndex, Cache *cache){
    int minTimeUsed = 0x7fffffff; // ★易错：应该设置的足够大！0x7fff都不够
    Cacheline *lru;
    for(int i=0;i<cache->E;i++){
        Cacheline *cur = &cache->cachelines[setIndex][i];
        if(cur->recentTimeUsed < minTimeUsed){
            minTimeUsed = cur->recentTimeUsed;
            lru = cur;
        }
    }
    initCacheline(lru);
}

void simulate(unsigned setIndex, unsigned tag, int *result, Cache *cache){
    int res = checkCache(setIndex, tag, cache);
    result[res]++;
    // 如果HIT，无需再进行操作
    // 如果MISS，向cache加载相应数据
    if(res == MISS){
        loadCache(tag, setIndex, cache);
    }
    // 如果EVICTION，驱逐后再次访问缓存不命中，之后向cache加载相应数据
    else if(res == EVICTION){
        LRU_evict(setIndex, cache);
        res = checkCache(setIndex, tag, cache);
        result[res]++;
        /* 事实上eviction后的miss是必然的，故这两行可直接改为result[MISS]++ */
        loadCache(tag, setIndex, cache);
    }
}
```

部分 4 - Putting task A all together

将各个部分整合在一起，`main()`函数如下：

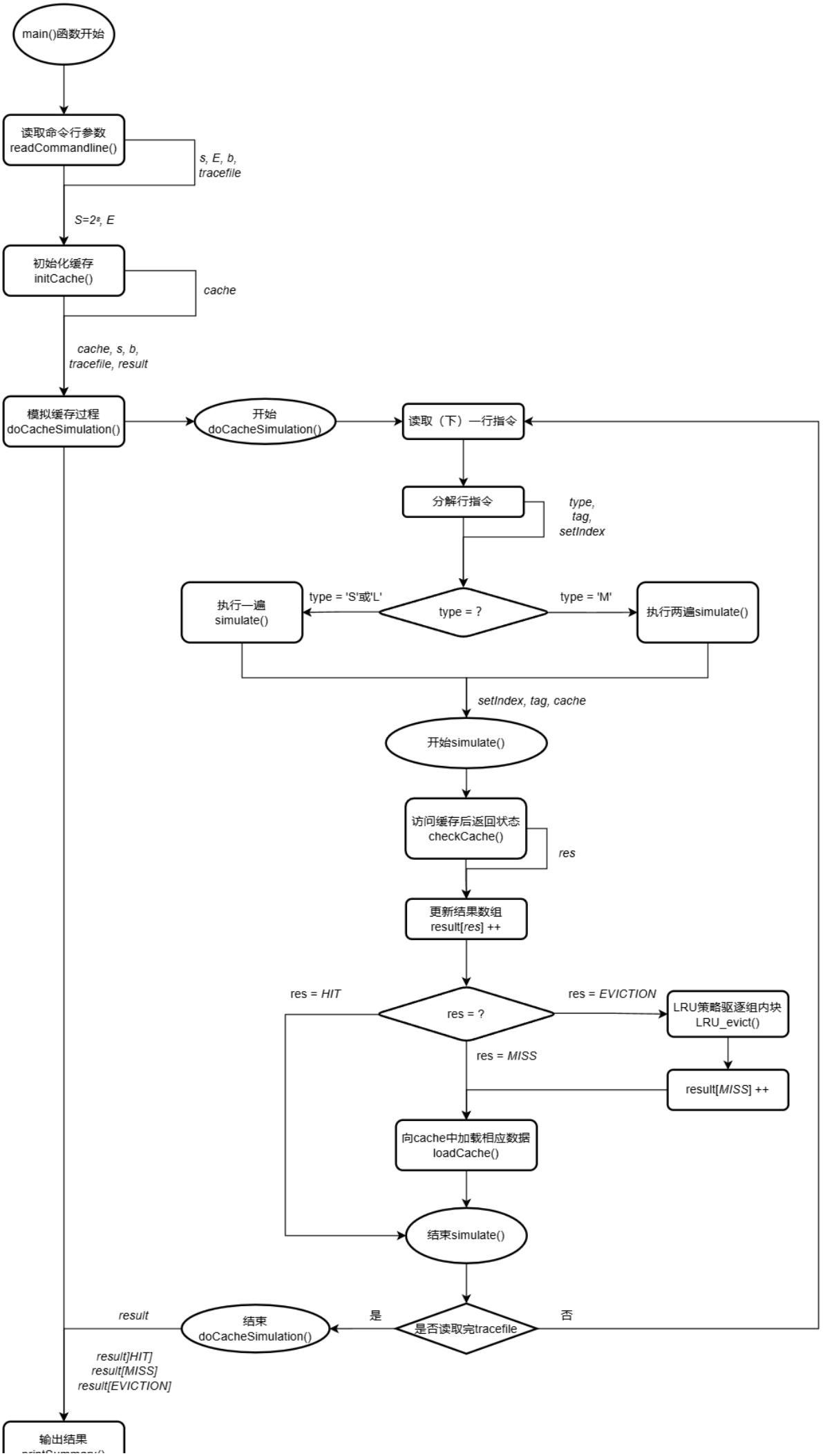
```
int main(int argc,char* argv[]){
    // 处理命令行参数
    int s, S, E, b;
    char tracefile[128];
    readCommandline(argc, argv, &s, &E, &b, tracefile);
    S = 1<<s;   // 用左移表示幂运算

    // 初始化cache
    Cache cache;
    initCache(S, E, &cache);
```

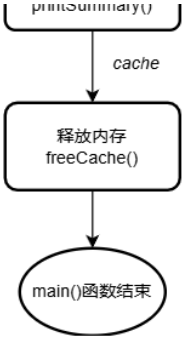
```
    // 模拟缓存过程
    int result[3] = {0};
    doCacheSimulation(tracefile, s, b , &cache, result);


    // 输出结果
    if(showDetails)
        printf("\n");
    printSummary(result[HIT], result[MISS], result[EVICTION]);

    // 回收内存
    freeCache(&cache);

    return 0;
}
```

完整`csim.c`见CSAPP/4.cachelab/cachelab-handout/csim.c at main · zzsyppt/CSAPP。

3. 实验结果

下面给出`csim.c`的流程图：


main()函数开始
读取命令行参数
readCommandline()
s, E, b,
tracefile
S=2ˢ, E
初始化缓存
initCache()
cache
cache, s, b,
tracefile, result
模拟缓存过程
doCacheSimulation()
开始
doCacheSimulation()
读取（下）一行指令
分解行指令
type,
tag,
setIndex
执行一遍
simulate()
type = 'S'或'L'
type = ?
type = 'M'
执行两遍simulate()
setIndex, tag, cache
开始simulate()
访问缓存后返回状态
checkCache()
res
更新结果数组
result[res] ++
res = HIT
res = ?
res = EVICTION
LRU策略驱逐组内块
LRU_evict()
res = MISS
result[MISS] ++
向cache中加载相应数据
loadCache()
结束simulate()
是
是否读取完tracefile
否
result
结束
doCacheSimulation()
result]HIT]
result[MISS]
result[EVICTION]
输出结果



执行`make`和`./test-csim`，结果如下：

```
zzsyp@Mark-PC2:~/csapp/cachelab/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
```

Part B: Optimizing Matrix Transpose

1. 任务说明

根据实验文档，在本部分，我们要在`trans.c`中编写代码，在给定的高速缓存结构：`s = 5, E = 1, b = 5`下，分别针对三种矩阵（`32*32`，`64*64`，`67*61`）编写矩阵转置的代码，使他们的`miss`数分别少于`300`，`1300`，`2000`。我们的代码需要在

```
char transpose_submit_desc[] = "Transpose submission";
void transpose_submit(int M, int N, int A[N][M], int B[M][N]){
    /* TODO */
}
```

中编写，但我们也可以根据实验文档的提示在`registerFunctions()`中注册其他转置函数，所有注册过的转置函数`function i`都会在`test-trans`调试时输出缓存访问情况，并把缓存访问信息（`valgrind memory trace`文件）写入`trace.fi`文件中，我们可以通过上一部分中的`csim-ref`来查看具体的缓存访问信息。

根据缓存的参数，我们可以算出，这是一个直接映射缓存，共有`32`组，每组的大小为`32`字节，可以存放`8`个`int`型变量。

2. 实验过程

部分 1 - 32×32矩阵转置

`trans.c`中给出了一个平凡的矩阵转置函数，它按内存存放顺序依次取矩阵`A`中的元素，立即将其放在矩阵`B`中的对应位置。使用`trans.c`进行测试，我们发现其`miss`数为`1184`。

此刻，我们要确认几个可能会“想当然”的问题：

- 一定是`a[0]~a[7]，a[8]~a[15]，...`分别存放在同一个缓存块中吗（是否可能会是`a[0]，a[1]~a[8]，...`存放在一个块中）？
- `A`和`B`下标相同的元素所被映射进的缓存块一定相同吗?
- 如果上面两个问题的答案为否，分别是否会对结果产生影响?

经过验证检索`csim`处理后的`trace.f1`，在本次实验中，`A`的首地址为`0x0010d0a0`，`B`的首地址为`0014d0a0`，故前两个问题的回答都是肯定的，验证过程此处不再赘述；第三个问题暂不考虑。

下面我们先尝试计算`1184`是如何得到的：

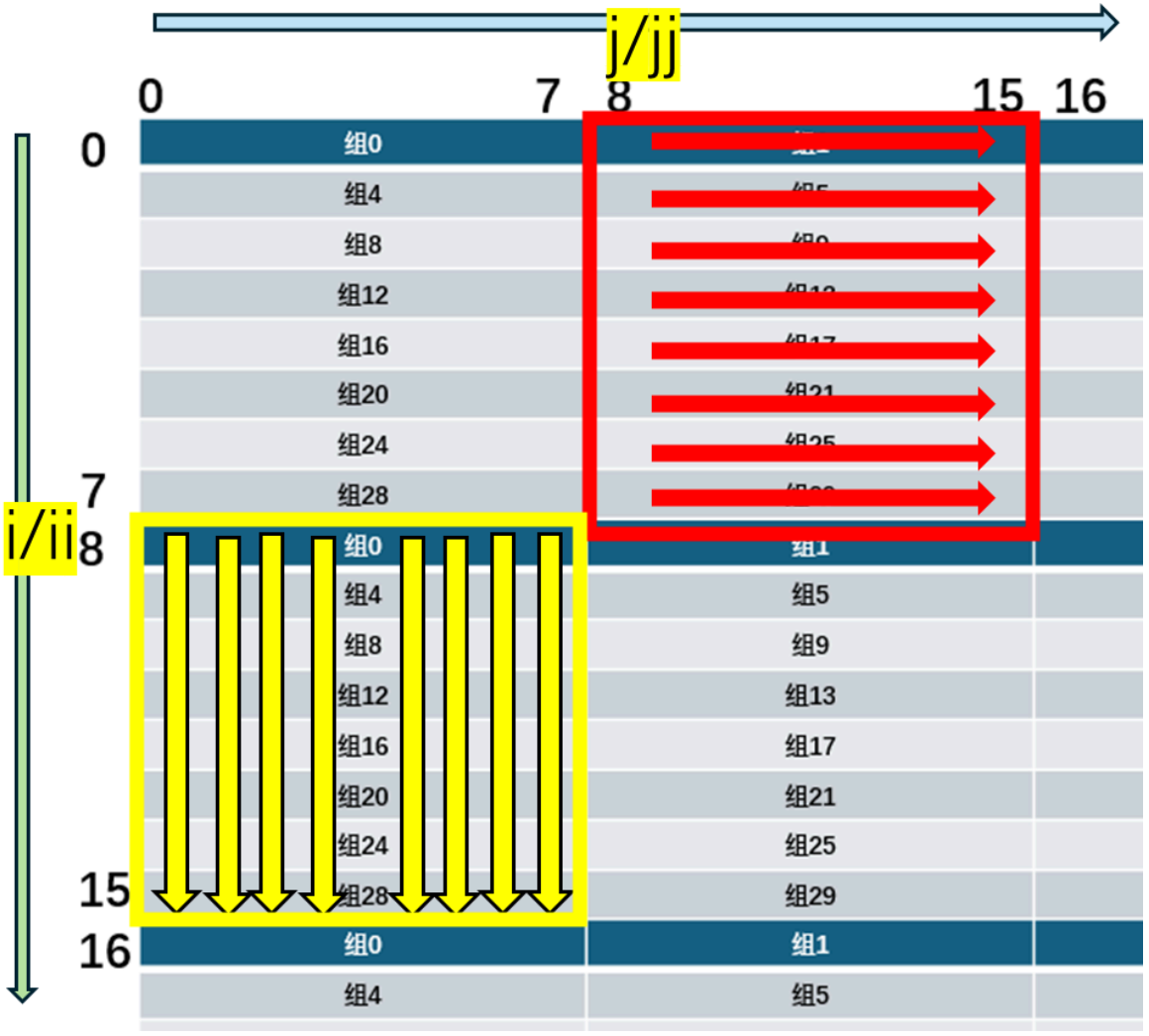
由于缓存有`32`组，每组存放`8`个`int`变量，且经过分析得到矩阵`A`和`B`相同元素所存放进的缓存块是相同的，所以，`A`，`B`矩阵的各个元素所存放的缓存位置为如下情况：

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

- (1) `A`：横着遍历，每次遍历到新的一组的元素就会发生`miss`，共计`128`个；
- (2) `B`：竖着遍历，每遍历一个元素都是新的一组，共计`32*32=1024`个；
- (3) 实验文档特别强调了我们要关注对角线元素的`conflict miss`：以对角线元素`[0][0]`的转置为例。如访问`A[0][0]`时，`组0`是被`A`占用的状态（这一占用导致的`miss`已在(1)中计入）；此时加载`B[0][0]`时，`B`也占用`组0`（这一占用导致的`miss`已在(2)中计入）；接着加载`A[0][1]`时又会发生`miss`，`组0`再次被`A`占用（这一`miss`未被(1)(2)中计入）。每次转置一个对角线元素，不在(1)(2)考虑内的`miss`次数是1个。而对于`[31][31]`的元素，由于访问`B[31][31]`后转置即结束，不会发生新的未计入的`miss`，故(1)(2)中未考虑到的`miss`共计`31`个。

综上，理论上共计有`1183`次`miss`，与实际结果相近。

下面对 32×32 的矩阵转置进行优化，根据上述分析，主要是减少遍历 B 的 miss 数。根据实验文档的提示，我们采用 分块(blocking) 技术。显然，采用大小为 8×8 子矩阵是符合缓存结构的。 B 的子矩阵的第一列一旦被加载进缓存后，子矩阵后续元素的访问都会命中。我们把矩阵分块，对 A 的每个子矩阵分别求转置放入 B 的相应位置中，示意图与代码如下：




```
void trans_32_32(int A[32][32], int B[32][32]){
    int ii, jj, i, j;
    // ii, jj在外层遍历矩阵，指向每个分块矩阵的第一行/列
    for(ii = 0; ii < 32; ii += 8){
        for(jj = 0; jj < 32; jj += 8){
            // i, j遍历子矩阵，指向子矩阵的某一行/列在原矩阵中的行/列
            for(i = ii; i < 8 + ii; i++){
                for(j = jj; j < 8 + jj; j++){
                    B[j][i] = A[i][j];
                }
            }
        }
    }
}
```

这里我们未考虑对角线元素的 conflict miss 的优化。经过测试，共有 344 次 miss，未达到满分标准，于是我们继续考虑对角线元素的优化。

我们发现，A的子矩阵的一行的8个元素是逐个进行读的；而若对应的B的子矩阵位于原矩阵的对角线上，B访问对角线元素会导致保存A的子矩阵的那一行的缓存块被驱逐，从而访问A的子矩阵的下一个元素时，会发生miss。为解决掉这个miss，解决方案就是，在读取A的子矩阵的某行的第一个元素时，就把这一行的8个元素保存到局部变量中，然后直接用局部变量把数据赋给B的组矩阵。

（具体来说，仍以[0][0]举例。访问A[0][0]时，A[0][0]~A[0][7]一并被加载进组0。而访问B[0][0]时，会导致组0中的块被驱逐，所以下次加载A[0][1]时，又会发生miss。所以解决方案就是，在访问A[0][0]时把A[0][0]~A[0][7]一起保存到临时变量中，再把这些临时变量赋给B[0][0]~B[7][0]。）

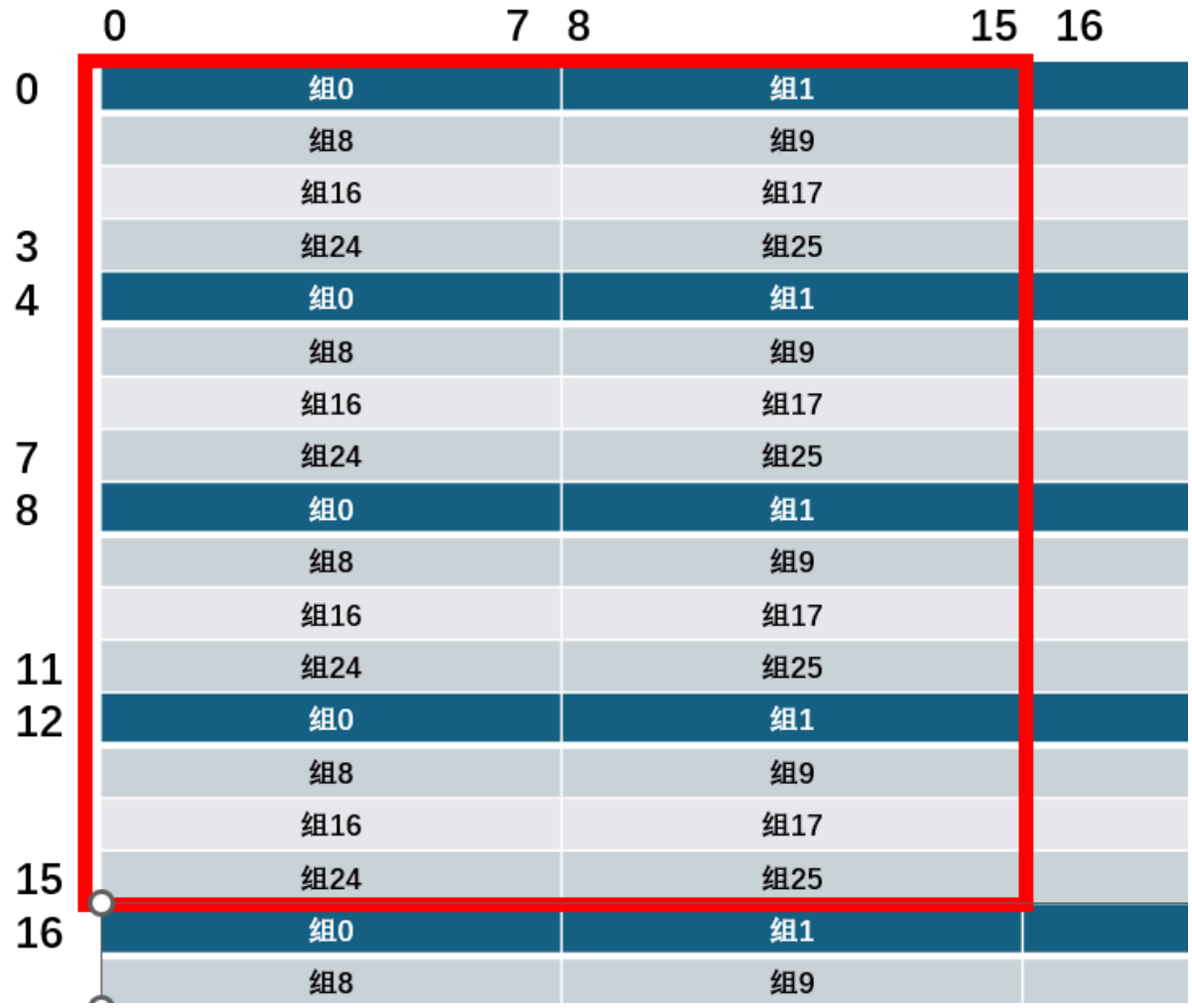
```
void trans_32_32(int A[32][32], int B[32][32]){
    int ii, jj, i, t1, t2, t3, t4, t5, t6, t7, t8;
    // ii, jj在外层遍历矩阵，指向每个分块矩阵的第一行/列
    for(ii = 0; ii < 32; ii += 8){
        for(jj = 0; jj < 32; jj += 8){
            // i遍历子矩阵，指向子矩阵的某一行/列在原矩阵中的行/列
            for(ii = ii; i < 8 + ii; i++){
                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];
                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;
            }
        }
    }
}
```

经过测试，优化对角线后的miss数为288，符合满分标准。

```
zzsyp@Mark-PC2:~/csapp/cachelab/cachelab-handout$ ./test-trans -M32 -N32

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
```

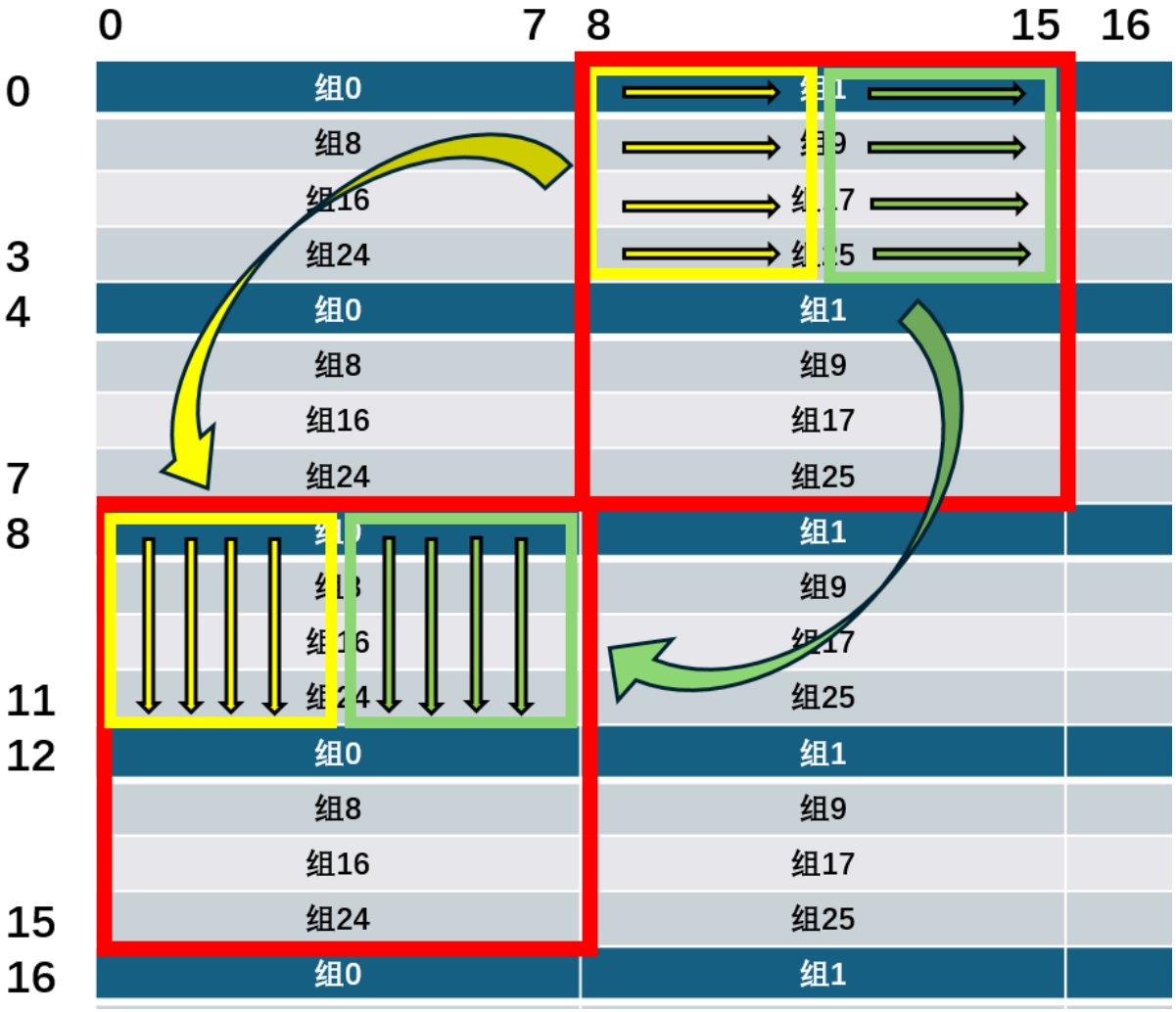
部分 2 - 64×64矩阵转置

A，B矩阵各个元素的存放位置如下图（截取了左上角16×16的部分）。

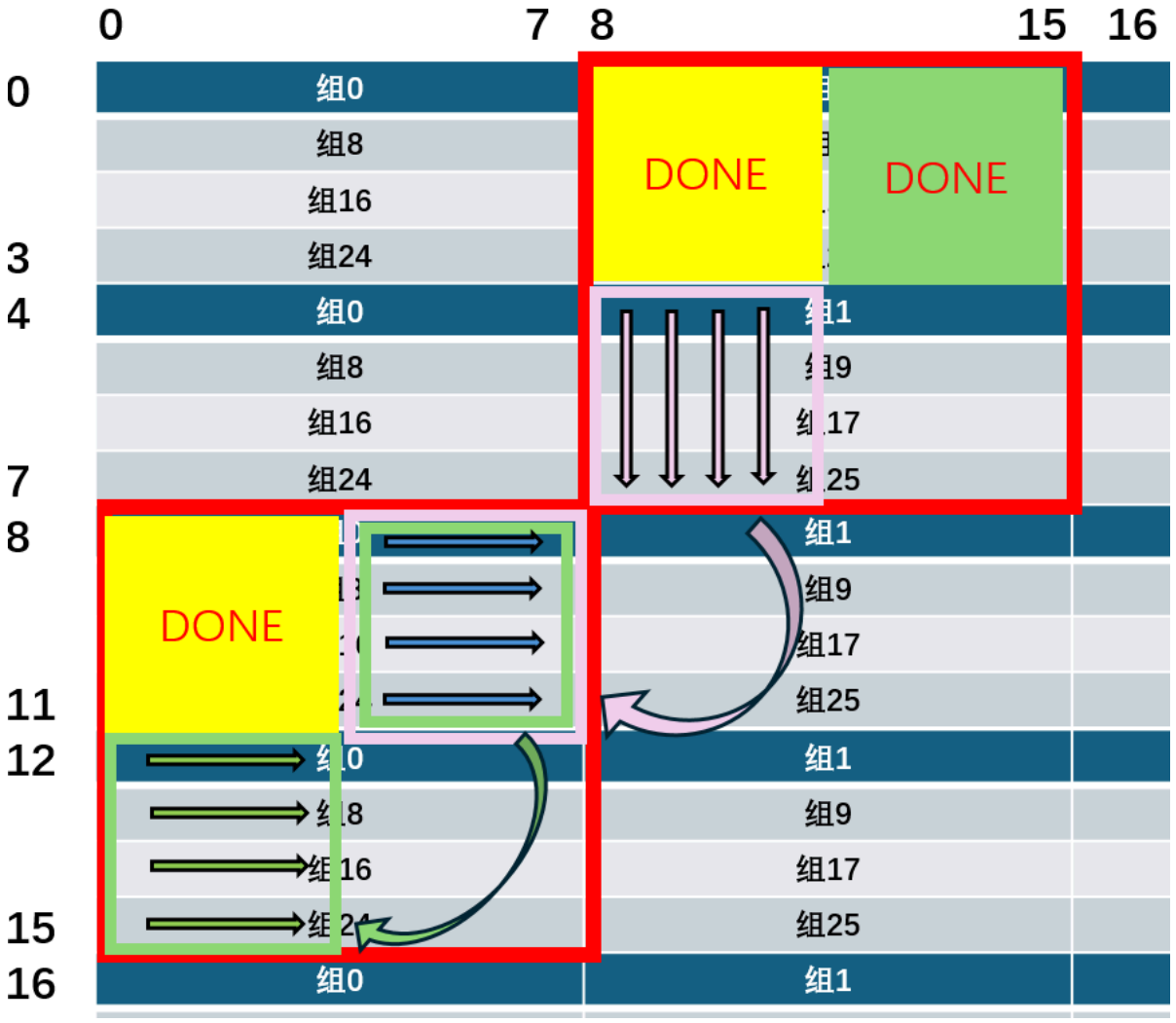


仍采用分块的思路。如果继续采用子矩阵为 8×8 的分块方式，可以看到对 B 矩阵的访问仍会造成抖动；而如果我们把矩阵分成 4×4 的子矩阵，虽然 B 矩阵的访问不再有抖动，但 A 矩阵的缓存没有得到充分利用，测试结果为 1700 次 miss。我们要找到这么一种方法，让 A 和 B 的缓存都得到充分利用，翻译过来即：**访问 A 时要尽量一连串读取 8 个值，访问 B 时要尽量一口气把竖着的四个组存满。**

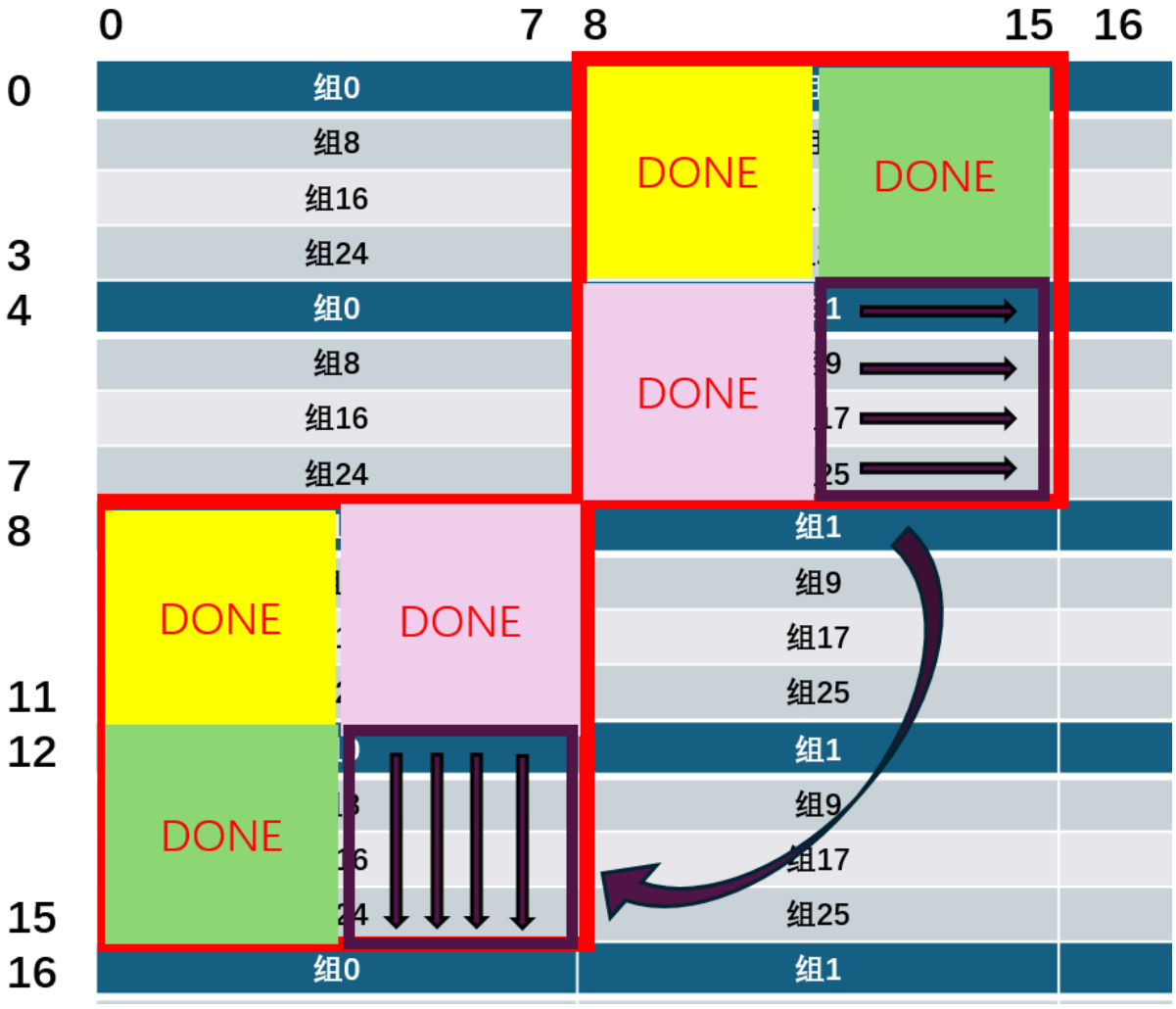
因此，我们找到这么一种策略，如下图所示：



（1）把 A 的 8×8 子矩阵的前四行的转置放进 B 的前四行，如图所示。黄色部分的转置已正确放到目标位置，而绿色部分的转置本应放到左下角，现在为提高缓存命中率先暂时放到图中所示位置。这一步的 miss 包括 4 个 A 的 miss 和 4 个 B 的 miss。



（2）这一步同时把绿色部分放到正确的位置，并把粉色部分放到正确的位置。以第一轮循环为例来解释：这里先用 `4` 个中间变量把 `B` 中错误放置的第一行值（缓存在 `旧的组0` 中）暂存，再把 `A` 的粉红色部分的第一列值放到正确的位置（还缓存在 `旧的组0` 中）。现在 `旧的组0` 已经不再会被用到，因此即可把中间变量存进其正确的位置（缓存会驱逐 `旧的组0`，加载 `新的组0`）。这一步的 `miss` 包括①加载 `A` 粉色部分第一列时的 `4` 个 `miss`；②正确保存绿色部分的 `4` 个 `miss`。



（3）最后的紫色部分直接放进正确位置即可，这些元素都在缓存中，不会发生`miss`。

注. 在（1）（3）中，也需要启用在**部分1**中的对角线优化策略，采用临时变量保存而非直接循环，否则不能达到满分标准。

代码如下：

```
void trans_64_64(int A[64][64], int B[64][64]){
    int ii, jj, i, j, t1, t2, t3, t4, t5, t6, t7, t8;
    // ii, jj在外层遍历矩阵，指向每个分块矩阵的第一行/列
    for(ii = 0; ii < 64; ii += 8){
        for(jj = 0; jj < 64; jj += 8){
            // i, j遍历子矩阵，指向子矩阵的某一行/列在原矩阵中的行/列
            // (1)
            // 这里采用临时变量而不直接循环，是因为优化对角线元素
            for(i = ii; i < 4 + ii; i++){
                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];
                B[jj][i] = t1;
                B[jj+1][i] = t2;
                B[jj+2][i] = t3;
                B[jj+3][i] = t4;
```

```
                B[jj][i+4] = t5;
                B[jj+1][i+4] = t6;
                B[jj+2][i+4] = t7;
                B[jj+3][i+4] = t8;
            }
            // (2)
            for(j = jj; j < 4 + jj; j++){
                // i.暂存错误的元素
                t1 = B[j][ii+4];
                t2 = B[j][ii+5];
                t3 = B[j][ii+6];
                t4 = B[j][ii+7];
                // ii.把粉色部分放进正确的位置
                B[j][ii+4] = A[ii+4][j];
                B[j][ii+5] = A[ii+5][j];
                B[j][ii+6] = A[ii+6][j];
                B[j][ii+7] = A[ii+7][j];
                // iii.把错误的元素放进正确的位置
                B[j+4][ii] = t1;
                B[j+4][ii+1] = t2;
                B[j+4][ii+2] = t3;
                B[j+4][ii+3] = t4;
            }
            // (3)
            // 这里采用临时变量而不直接循环，是因为优化对角线元素
            for(int i = ii + 4; i < ii + 8; i++){
                t1 = A[i][jj+4];
                t2 = A[i][jj+5];
                t3 = A[i][jj+6];
                t4 = A[i][jj+7];
                B[jj+4][i] = t1;
                B[jj+5][i] = t2;
                B[jj+6][i] = t3;
                B[jj+7][i] = t4;
            }
        }
    }
}
```

每个`8×8`子矩阵（对角线子矩阵除外）的转置有`16`次`miss`，共有`64`个`8×8`子矩阵，所以理论总计有不少于`16*64=1024`次`miss`。经过测试，共有`1228`次`miss`。

```
zzsyp@Mark-PC2:~/csapp/cachelab/cachelab-handout$ ./test-trans -M64 -N64

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:1196
```

部分 3 - 67×61矩阵转置

经过测试，采用`16×16`的子矩阵分块策略可以满足满分要求。注意循环时需要添加边界条件，避免数组越界。代码如下：

```
void trans_61_67(int A[67][61], int B[61][67]){
    // ii, jj在外层遍历矩阵，指向每个分块矩阵的第一行/列
    for(int ii = 0; ii < 67; ii += 16){
        for(int jj = 0; jj < 61; jj += 16){
            // i,j 遍历子矩阵，指向子矩阵的某一行/列在原矩阵中的行/列
            for(int i = ii; i < 16 + ii && i < 67; i++){
                for(int j = jj; j < 16 + jj && j < 61; j++){
                    B[j][i] = A[i][j];
                }
            }
        }
    }
}
```

```
zzsyp@Mark-PC2:~/csapp/cachelab/cachelab-handout$ ./test-trans -M61 -N67

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:6186, misses:1993, evictions:1961
```

实验结果

由于handout中的`driver.py`由`python2`语言编写，删除并重新下载`python3`重构后的`driver.py`。

```
wget https://gitee.com/lin-xi-269/csapplab/raw/master/lab5cachelab/cachelab-handout/driver.py
```

运行`python3 driver.py`，得分结果如下：

```
zzsyp@Mark-PC2:~/csapp/cachelab/cachelab-handout$ python3 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         288
Trans perf 64x64           8.0         8        1228
Trans perf 61x67          10.0        10        1993
          Total points    53.0        53
```

总结

通过本实验学习到的知识点总结：

- 学会了利用`getopt()`读取命令行参数；
- 学会了用二维数组和结构体模拟缓存；
- 在为缓存动态分配内存时，加深了对二阶指针、二维数组和指针数组的概念间的理解，以及积累了多维数组使用`malloc`和`free`函数的经验；
- 学会了用工具`draw.io`绘制简单的流程图；
- 初步探索尝试了通过分块技术优化矩阵有关运算的缓存命中率，再加上一些访问顺序上的技巧，使得原有缓存块在被覆盖掉之前尽可能多的被访问。

最优解参考

本实验报告`Part B`的三个解法均非最优解。最优解可参考：

`32×32`：深入理解计算机系统-cachelab_yi.traces-CSDN博客

`64×64`：CSAPP - Cache Lab的更(最)优秀的解法 - 知乎