Cachelab Report

胡译文 2021201719

Part A

这一部分相当于一道模拟题。模拟一个 Cache。根据相联组 Cache 的结构，采用结构体模拟一个 Cacheline，一维数组模拟组内的行，二位数组模拟组。

```
typedef struct {
    int valid, tag, idle;  // There's no need to store the read content of cache
} CacheLine;

CacheLine **cache;
```

在访问过程中，主要有三种情况，第一是 Hit：

```
// hit
for (int i = 0; i < E; i++) {
    if (miss_idx == -1 && current_group[i].valid == 0) {
        miss_idx = i;
    }
    if (current_group[i].tag == tag) {
        current_group[i].idle = 0;
        HIT++;
        return ;
    }
}
```

而无论是 Miss 还是 Eviction 都要计算 `miss_count`。

```
// miss
if (miss_idx != -1) {
    current_group[miss_idx].valid = 1;
    current_group[miss_idx].tag = tag;
    current_group[miss_idx].idle = 0;
    return ;
}
```

如上图，在判断 Hit 的过程中就可以判断，是否需要替换以及如果仅需读取，读取的组内行号为多少。如果需要替换，则需要再次遍历组内所有行，选取最远空闲行进行替换。

```
// miss & eviction
EVICTION++;
int max_idle = MIN, max_idle_idx;
for (int i = 0; i < E; i++) {
    if (current_group[i].idle > max_idle) {
        max_idle = current_group[i].idle;
        max_idle_idx = i;
    }
}
current_group[max_idle_idx].tag = tag;
current_group[max_idle_idx].idle = 0;
```

`getopt()` 为命令行提供了辅助，其中必选参数以 `:` 标记。

Part B

文档中有提示使用“分块”技术，但是难点在于弄清楚什么是“分块”，以及如何“分块”。

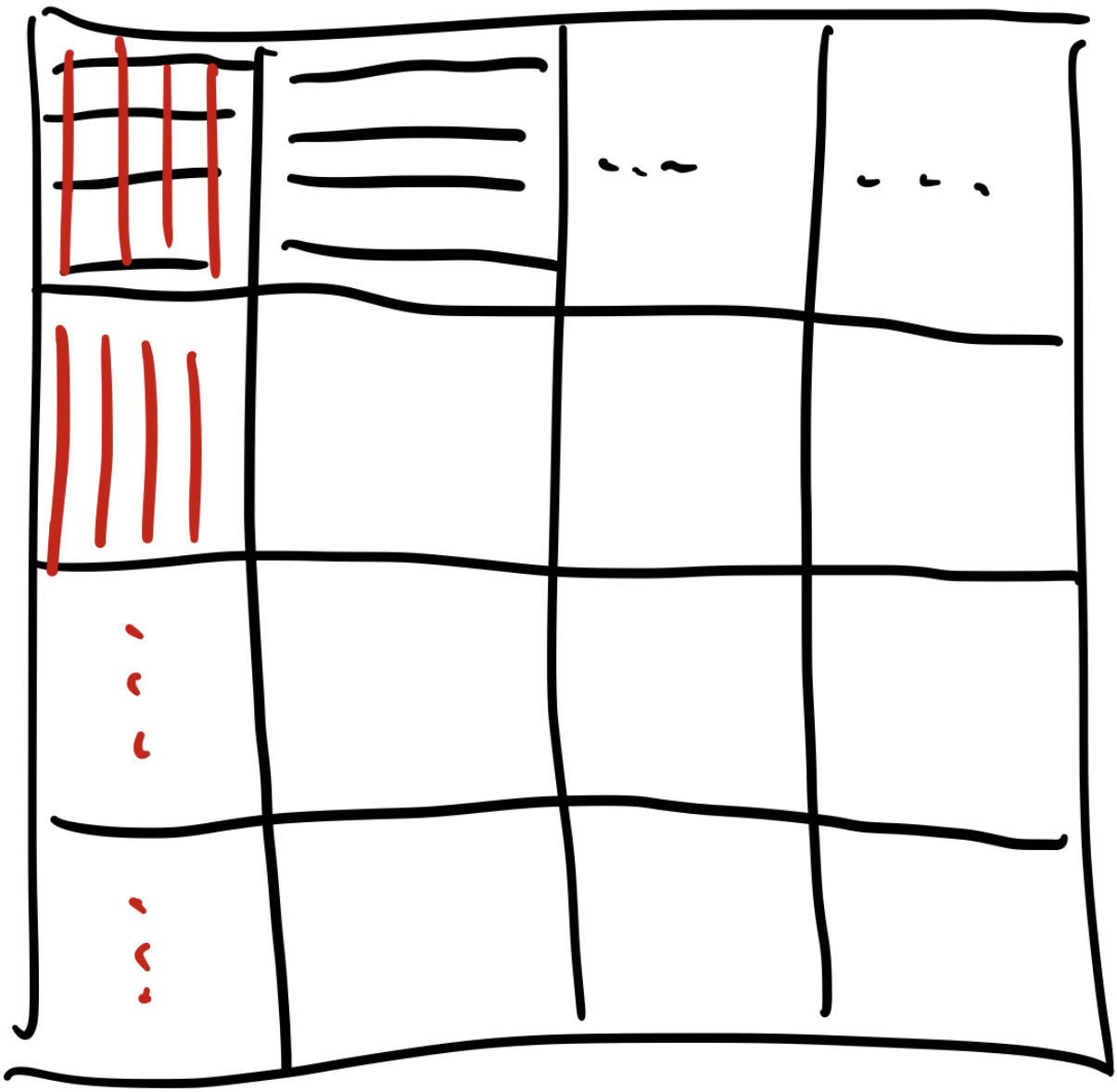
最开始的思维惯性，导致思考本题出现了偏差 —— 将转置操作误以为是 inplace 的。因此最初将“分块”理解为将矩阵切分为子矩阵，再交换子矩阵位置的同时，交换子矩阵内部元素。但事实上，复制到 B 数组的操作，很

大的简化了题目。同时还需要注意的是，每个数组元素仅访问一次，因此所有 Cacheline 都是用完即扔，仅需将块大小匹配一个 Cacheline 即可，无需考虑多个 Cacheline 之间交互。

而如何分块将是接下来讨论的重点，即兼顾“读”和“写”的同时，最大化 Cache 的利用。

首先对 Cache 进行分析。总共有 $2^5 = 32$ 组，每组有 1 行 Cacheline，每个 Cacheline 有 2^5 字节的高速缓存块。因此每个 Cacheline 可以存放 $2^5/4 = 8$ 个 int，因此将矩阵按 8 的大小分块预计有较优效果。结果显示 miss 次数降低，但仍然为达到理论最优。

进一步分析，在 32×32 的矩阵中，如下图所示，黑色代表读取，红色代表存储，总共使用 32 行 Cacheline，恰好存下。



由于 Cache 高速缓存块的字节数 2^b 与矩阵边边数相同，因此读、写同一位置的 A 和 B 矩阵拥有相同的内存组索引，即映射到同一相联组（因为仅有一个 Cacheline 所以也是相同 Cacheline 中）。在对 $A_{i,i}$ 转置的时候，会产生多余替换。将交会行提前可以解决该问题，但判断当前处在哪一行较为烦琐。因此考虑到可以使用变量，且恰好在运行个数内，所以先将当前块逐一读出，再逐一写出。

```
inline void _t_32(int M, int N, int A[N][M], int B[N][M]) {
    int i, gi, gj;
    int t0, t1, t2, t3, t4, t5, t6, t7;
    for (gi = 0; gi < N; gi += 8) {
        for (gj = 0; gj < M; gj += 8) {
            for (i = gi; i < gi + 8; i++) {
                t0 = A[i][gj];
                t1 = A[i][gj+1];
                t2 = A[i][gj+2];
                t3 = A[i][gj+3];
                t4 = A[i][gj+4];
                t5 = A[i][gj+5];
                t6 = A[i][gj+6];
                t7 = A[i][gj+7];
                B[gj][i] = t0;
                B[gj+1][i] = t1;
                B[gj+2][i] = t2;
                B[gj+3][i] = t3;
                B[gj+4][i] = t4;
                B[gj+5][i] = t5;
                B[gj+6][i] = t6;
                B[gj+7][i] = t7;
            }
        }
    }
}
```

对 64×64 的矩阵故技重施，但效果不佳——不同的矩阵位置被映射到了相同块。每四行就可以填满一个 Cache，因此如果将 blocksize 维持在 8 会导致每次写操作会产生替换。因此将 blocksize 减小为 4 。

```
inline void _t_64(int M, int N, int A[N][M], int B[N][M]) {
    int i, gi, gj;
    int t0, t1, t2, t3;
    for (gi = 0; gi < N; gi += 4) {
        for (gj = 0; gj < M; gj += 4) {
            for (i = gi; i < gi + 4; i++) {
                t0 = A[i][gj];
                t1 = A[i][gj+1];
                t2 = A[i][gj+2];
                t3 = A[i][gj+3];
                B[gj][i] = t0;
                B[gj+1][i] = t1;
                B[gj+2][i] = t2;
                B[gj+3][i] = t3;
            }
        }
    }
}
```

依然 miss 较高。因此进一步考察 miss 原因。在 8×8 的块中，因为超过了 4 的高度导致 miss，因此若改用 8×4 的块大小，并“借用”Cache 内其他位置，使得保证访问位置不超过 4 的高度，就能进一步减小 miss。这样将 8×8 的块可以划分成三部分处理：

```
for (k = gi; k < gi + 4; k++) {
    t0 = A[k][gj  ];
    t1 = A[k][gj+1];
    t2 = A[k][gj+2];
    t3 = A[k][gj+3];
    t4 = A[k][gj+4];
    t5 = A[k][gj+5];
    t6 = A[k][gj+6];
    t7 = A[k][gj+7];
    B[gj  ][k] = t0;
    B[gj+1][k] = t1;
    B[gj+2][k] = t2;
    B[gj+3][k] = t3;
    B[gj  ][k + 4] = t4;  // borrow
    B[gj+1][k + 4] = t5;
    B[gj+2][k + 4] = t6;
    B[gj+3][k + 4] = t7;
}
```

在第二部分处理时，`t4` ~ `t7` 和 `t0` ~ `t4` 的处理顺序不能颠倒，因为这需要利用内存中保存的 Cache，如果颠倒则被替换。

```
for (k = gj; k < gj + 4; k++) {
    t0 = B[k][gi+4];  // retreive
    t1 = B[k][gi+5];
    t2 = B[k][gi+6];
    t3 = B[k][gi+7];
    t4 = A[gi+4][k];  // inner transpose
    t5 = A[gi+5][k];
    t6 = A[gi+6][k];
    t7 = A[gi+7][k];
    B[k][gi+4] = t4;
    B[k][gi+5] = t5;
    B[k][gi+6] = t6;
    B[k][gi+7] = t7;
    B[k+4][gi  ] = t0;  // renew
    B[k+4][gi+1] = t1;
    B[k+4][gi+2] = t2;
    B[k+4][gi+3] = t3;
}
```

最后再对不会造成替换的部分直接处理：

```
for (t0 = gi+4; t0 < gi+8; t0++) {
    for (t1 = gj+4; t1 < gj+8; t1++) {
        B[t1][t0] = A[t0][t1];
    }
}
```

这样“借用”的方式增加了读写的时间，但减少了 Cache miss。这样的设计在高速内存中是合理的，但是在磁盘读写中可能会因为磁盘读写时间而更慢。