CACHELAB实验报告

Part A

首先，代码定义了一个 Cache_Line 结构体，用于表示缓存中的每一行。这个结构体包含两个成员：tag 和 time_stamp。其中，tag 用于存储缓存行的标记，time_stamp 用于存储缓存行的时间戳。

然后，定义了一些变量，用于存储命中次数、缺失次数、替换次数等信息，并定义了一个 find 函数，用于在缓存中查找给定的数据。

在 main 函数中，代码读取命令行参数并处理这些参数。然后根据参数中指定的缓存参数（如缓存行数、每行的字节数等）分配缓存内存。最后，代码打开输入文件，读取文件中的每一行，并逐行解析输入数据，并调用 find 函数在缓存中查找数据。
具体实现如下：

```
// 定义一个结构体，用于表示缓存中的每一行
typedef struct
{
    // 用于存储缓存行的标记
    unsigned long long tag;
    // 用于存储缓存行的时间戳
    int time_stamp;
} Cache_Line;

// 定义缓存命中次数、缺失次数和替换次数
int misses, hits, evictions;

// 定义一些变量，用于存储命令行参数
int opt, s, E, b, v = 0;

// 定义当前时间戳
int time;

// 定义一个函数，用于在缓存中查找给定的数据
void find(Cache_Line *cache_line, int s, int b, int E, unsigned address, int ismodify)
{
    // 定义循环变量
    int i;

    // 计算当前地址的索引和标记
    int t = 64 - s - b;
    unsigned long long idx = (unsigned long long)address;
unsigned long long tag = (idx >> (b + s));
idx = ((idx << t) >> (t + b));

// 定义一个指针，用于指向缓存中当前行对应的最近最少使用的行
Cache_Line *least_used = &cache_line[idx * E];

// 遍历当前行，查找给定的数据
for (i = idx * E; i < (idx+1)*E; i++)
{
    // 如果找到了给定的数据，则更新缓存行的时间戳并返回
    if (cache_line[i].time_stamp && cache_line[i].tag == tag)
    {
        // 如果启用了 verbose 模式，则输出 hit 消息
        if (v)
            printf("hit ");

        // 累加命中次数
        hits++;

        // 更新缓存行的时间戳
        cache_line[i].time_stamp=time;

        // 更新 least_used 指针
        least_used = &cache_line[i];

        // 跳转到 modify 标签
        goto modify;
    }

    // 如果当前行最近最少使用，则更新 least_used 指针
    else if (cache_line[i].time_stamp < least_used->time_stamp)
            least_used = &cache_line[i];
    }
    if (v)
        printf("miss ");
```

```
        misses++;
        if (least_used->time_stamp)
        {
            evictions++;
            if (v)
                printf("eviction ");
        }
        least_used->tag = tag;
        least_used->time_stamp = time;
    modify:
        if (ismodify)
        {
            hits++;
            if (v)
                printf("hit ");
        }
    }
```

主函数

```
int opt, s, E, b, v = 0;
int main(int argc, char *argv[])
{
    // 定义一些变量，并初始化部分变量
    char *t;
    // 使用getopt()函数来解析命令行参数
    while ((opt = getopt(argc, argv, "hvs:E:b:t:")) != -1)
    {
        switch (opt)
        {
        case 'h':
            // 如果遇到了'-h'参数，则打印帮助信息并退出程序
            printf("Usage: ./csim [-hv] -s <num> -E <num> -b <num> -t <file>\n\
Options:\n\
  -h         Print this help message.\n\
  -v         Optional verbose flag.\n\
  -s <num>   Number of set index bits.\n\
  -E <num>   Number of lines per set.\n\
  -b <num>   Number of block offset bits.\n\
  -t <file>  Trace file.\n\
\n\
Examples:\n\
  linux>  ./csim -s 4 -E 1 -b 4 -t traces/yi.trace\n\
  linux>  ./csim -v -s 8 -E 2 -b 4 -t traces/yi.trace");
            return 0;
        case 'v':
            // 如果遇到了'-v'参数，则将变量v置为1
            v = 1;
            break;
        case 's':
            // 如果遇到了'-s'参数，则将参数值设为相应的值
            s = atoi(optarg);
            break;
        case 'E':
            E = atoi(optarg);
            break;
        case 'b':
            b = atoi(optarg);
            break;
        case 't':
            t = optarg;
            break;
        default:
            printf("wrong argument\n");
            return 0;
        }
    }
    int  S = (1 << s);
    Cache_Line *cache_line = (Cache_Line *)malloc(S * E * sizeof(Cache_Line));
    memset(cache_line, 0, S * E);

    FILE *pFile;
    pFile = fopen(t, "r");
    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)
    {
        if(identifier=='I')
            continue;
        time++;
```

```
        if (v)
            printf("%c %x,%d ", identifier, address, size);
        find(cache_line, s, b, E, address, identifier == 'M');
        if (v)
            printf("\n");

    }
    fclose(pFile); // remember to close file when don
    free(cache_line);
    printSummary(hits, misses, evictions);
    return 0;
}
```

Part B

```
// 定义 BLOCK 的值为 8
#define BLOCK 8
void transpose_submit(int M, int N, int A[N][M], int B[M][N])
{
    if(M==32&&N==32)
        trans_32_32(M,N,A,B);
    else if(M==64&&N==64)
        trans_64_64(M,N,A,B);
    else
        trans_61_67(M,N,A,B);
}
```

首先，由于Cache的参数(S,E,B)=(32,1,32),我们可以算出一个Cacheline 能装下32/4=8个int。对于不同大小的矩阵，我们肯定分类进行优化，下面依次介绍具体的实现。

32 * 32

```
// 定义函数 trans_32_32，用于实现矩阵转置
void trans_32_32(int M, int N, int A[32][32], int B[32][32])
{
    // 定义循环变量
    int i, j, i1, j1;
    // 外层循环：按 BLOCK 的步长遍历矩阵 A 的每一列
    for (j = 0; j < N; j += BLOCK)
    {
        // 内层循环：按 BLOCK 的步长遍历矩阵 A 的每一行
        for (i = 0; i < M; i += BLOCK)
        {
            // 判断当前行和列是否相同
            if (i != j)
            {
                // 如果不同，转置矩阵 A 的当前行和列，并存储在矩阵 B 的对应位置
                for (i1 = i; i1 < i + BLOCK; i1++)
                    for (j1 = j; j1 < j + BLOCK; j1++)
                        B[j1][i1] = A[i1][j1];
            }
            else
            {
                // 如果相同，判断当前行加上 BLOCK 是否超出矩阵 A 的行数
                if (i + BLOCK < M)
                {
                    // 如果不超出，转置矩阵 A 的当前行和列，并存储在临时空间的对应位置
                    for (i1 = i; i1 < i + BLOCK; i1++)
                        for (j1 = j; j1 < j + BLOCK; j1++)
                            B[j1][i1 + BLOCK] = A[i1][j1];
                    // 将临时空间的当前行和列赋值给矩阵 B 的对应位置
                    for (i1 = i; i1 < i + BLOCK; i1++)
                        for (j1 = j; j1 < j + BLOCK; j1++)
                            B[i1][j1] = B[i1][j1 + BLOCK];
                }
                else
                {
                    // 如果超出，转置矩阵 A 的当前行和列，并存储在矩阵 B 的对应位置
                    for (i1 = i; i1 < i + BLOCK; i1++)
                        for (j1 = j; j1 < j + BLOCK; j1++)
                            B[j1][i1] = A[i1][j1];
                }
            }
        }
    }
}
```

对32×32的矩阵，每行需要32/8=4个组的缓存，最多能同时存32/4=8行的矩阵，所以，我们可以直接按8*8的块的转置，由于对角线直接转置会出现冲突，我们可以借用下一次即将读取的内存，即 `B[j1][i1 + BLOCK]` 作为缓存来复制一个副本，然后再转置回去，这样的话也不会增加额外的冲突。

64*64

对64×64的矩阵，每行需要64/8=8个组的缓存，最多能同时存32/8=4行的矩阵，如果直接取BLOCK为8会出现大量冲突，取BLOCK为4miss会降到1800左右，但是这样平均的miss会大概有总数的1/4，不能充分发挥B=8的效果，所以我们可以再次借鉴32×32的思路，不过由于这种情况需要的临时空间更多，分类讨论的情况会增加，具体分为

- 对角线的前62个块 (需要4块4*4的临时空间)

- 下边界上的块转置 （需要1块4*4的临时空间）
- 正常块 （需要1块4*4的临时空间）

```
void trans_64_64(int M, int N, int A[N][M], int B[M][N])
{
    int i, j, i1, j1;
    for (j = 0; j < N; j += BLOCK)
        for (i = 0; i < M; i += BLOCK)// 对每个分块进行循环
        {
            if (i + BLOCK < M)
            {
                if (i == j && i + 2 * BLOCK < M) // 如果块在矩阵的对角线上且i + 2 * BLOCK < M
                {
                    for (i1 = i; i1 < i + BLOCK / 2; i1++)
                        for (j1 = j; j1 < j + BLOCK; j1++)
                            B[i1][j1 + BLOCK] = A[i1][j1];
                    for (i1 = i + BLOCK / 2; i1 < i + BLOCK; i1++)
                        for (j1 = j; j1 < j + BLOCK; j1++)
                            B[i1 - BLOCK / 2][j1 + 2 * BLOCK] = A[i1][j1];
                    for (i1 = i; i1 < i + BLOCK / 2; i1++)
                        for (j1 = j; j1 < j + BLOCK / 2; j1++)
                            B[j1][i1] = B[i1][j1 + BLOCK];
                    for (i1 = i + BLOCK / 2; i1 < i + BLOCK; i1++)
                        for (j1 = j; j1 < j + BLOCK / 2; j1++)
                            B[j1][i1] = B[i1 - BLOCK / 2][j1 + 2 * BLOCK];
                    for (i1 = i; i1 < i + BLOCK / 2; i1++)
                        for (j1 = j + BLOCK / 2; j1 < j + BLOCK; j1++)
                            B[j1][i1] = B[i1][j1 + BLOCK];
                    for (i1 = i + BLOCK / 2; i1 < i + BLOCK; i1++)
                        for (j1 = j + BLOCK / 2; j1 < j + BLOCK; j1++)
                            B[j1][i1] = B[i1 - BLOCK / 2][j1 + 2 * BLOCK];
                }
                else// 标准块矩阵转置
                {
                    for (i1 = i; i1 < i + BLOCK / 2; i1++)
                        for (j1 = j; j1 < j + BLOCK / 2; j1++)
                            B[j1][i1] = A[i1][j1];
                    for (i1 = i; i1 < i + BLOCK / 2; i1++)
                        for (j1 = j + BLOCK / 2; j1 < j + BLOCK; j1++)
                            B[j1 - BLOCK / 2][i1 + BLOCK] = A[i1][j1];
                    for (i1 = i + BLOCK / 2; i1 < i + BLOCK; i1++)
                        for (j1 = j; j1 < j + BLOCK / 2; j1++)
                            B[j1][i1] = A[i1][j1];
                    for (i1 = i + BLOCK / 2; i1 < i + BLOCK; i1++)
                        for (j1 = j + BLOCK / 2; j1 < j + BLOCK; j1++)
                            B[j1][i1] = A[i1][j1];
                    for (i1 = i; i1 < i + BLOCK / 2; i1++)
                        for (j1 = j + BLOCK / 2; j1 < j + BLOCK; j1++)
                            B[j1][i1] = B[j1 - BLOCK / 2][i1 + BLOCK];
                }
            }
            else if (j + BLOCK < N)// 下边界上的块转置

            {
                for (i1 = i; i1 < i + BLOCK / 2; i1++)
                    for (j1 = j; j1 < j + BLOCK / 2; j1++)
                        B[j1][i1] = A[i1][j1];
                // output(arr2);
                for (i1 = i; i1 < i + BLOCK / 2; i1++)
                    for (j1 = j + BLOCK / 2; j1 < j + BLOCK; j1++)
                        B[j1 - BLOCK / 2 + BLOCK + BLOCK - 1][i1 - (M - 1) * BLOCK] = A[i1][j1];
                // output(arr2);
                for (i1 = i + BLOCK / 2; i1 < i + BLOCK; i1++)
                    for (j1 = j; j1 < j + BLOCK / 2; j1++)
                        B[j1][i1] = A[i1][j1];
```

```
                for (i1 = i + BLOCK / 2; i1 < i + BLOCK; i1++)
                    for (j1 = j + BLOCK / 2; j1 < j + BLOCK; j1++)
                        B[j1][i1] = A[i1][j1];
                for (i1 = i; i1 < i + BLOCK / 2; i1++)
                    for (j1 = j + BLOCK / 2; j1 < j + BLOCK; j1++)
                        B[j1][i1] = B[j1 - BLOCK / 2 + BLOCK + BLOCK - 1][i1 - (M - 1) * BLOCK];
            }
            else//右下角的块转置
            {
                for (i1 = i; i1 < i + BLOCK / 2; i1++)
                    for (j1 = j + BLOCK / 2; j1 < j + BLOCK; j1++)
                        B[j1][i1] = A[i1][j1];
                for (i1 = i; i1 < i + BLOCK / 2; i1++)
                    for (j1 = j; j1 < j + BLOCK / 2; j1++)
                        B[j1 + BLOCK / 2][i1 + BLOCK / 2] = A[i1][j1];
                for (j1 = j + BLOCK / 2; j1 < j + BLOCK; j1++)
                    for (i1 = i + BLOCK / 2; i1 < i + BLOCK; i1++)
                        B[j1 - BLOCK / 2][i1 - BLOCK / 2] = B[j1][i1];
                for (i1 = i + BLOCK / 2; i1 < i + BLOCK; i1++)
                    for (j1 = j; j1 < j + BLOCK / 2; j1++)
                        B[j1][i1] = A[i1][j1];
                for (i1 = i + BLOCK / 2; i1 < i + BLOCK; i1++)
                    for (j1 = j + BLOCK / 2; j1 < j + BLOCK; j1++)
                        B[j1][i1] = A[i1][j1];
            }
            //output(arr2);
        }
}
```

正常块和对角线的前62个块可以做到只有cold miss，最后两个块没有优化，可以估算出miss大概为64*62*/8+228*8=1024+256=1280，真实值为1293，刚好过线，因此我们不必继续优化了

61*67

```
#define BLOCK0 16
void trans_61_67(int M, int N, int A[N][M], int B[M][N])
{
    int i, j, i1, j1;
    for(i=0;i<N;i+=BLOCK0)
        for(j=0;j<M;j+=BLOCK0)
            for(i1=i;i1<i+BLOCK0&&i1<N;i1++)
                for(j1=j;j1<j+BLOCK0&&j1<M;j1++)
                    B[j1][i1] = A[i1][j1];
}
```

这个由于它的行列数不等，我们可以直接尝试不同的BLOCK大小，然后就发现取BLOCK0 16时的冲突就已经低于2000了
最终的测试结果：

```
[2021201746@work122 cachelab-handout]$ ./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         280
Trans perf 64x64           8.0         8        1293
Trans perf 61x67          10.0        10        1992
          Total points    53.0        53
```