最终 make 生成的答案在 cachelab-handout 文件夹内

1.Part A

- 阅读实验说明，得知这一部分实际是要编写模拟一个 cache。其中替换的操作选择使用 LRU 策略，为每行设置一个时间戳，时间戳越大表示最后访问时间距离现在越远，所以选择时间戳最大的元素进行替换即可。
- 首先定义 cache 的结构，因为每行 cache 要包括 valid 有效位、tag 标记位和 stamp 时间戳（用于 LRU 算法决定覆盖哪一行 cache），一个组中有多行，一个 cache 有多组。因此将 cache 定义为二维的结构体数组，声明一个对象并写出 cache 的初始化函数 init_cache（malloc 动态分配空间）。读入参数等的操作不再赘述，利用说明中给出的 getopt、fscanf 函数进行读入即可。

```
typedef struct{
    int valid_bits;
    int tag;
    int stamp;
}cache_line, *cache_set, **cache;
```

- 由于仅仅是模拟 cache，不同的操作指令的实际操作很相似，所以共同写为一个函数 trycache，L 和 S 指令相当于调用一次，M 指令相当于调用两次。
- 首先判断是否命中，需要组索引地址处的有效位为 1 且 tag 与请求的标记位一致，hit_count 计数。然后判断是否有空行，需要看是否存在有效位为 0 的行，miss_count 计数。若未命中且没有空行，则需要进行替换，根据 LRU 策略，寻找时间戳最大的进行替换，miss_count 和 evition_count 计数。

```
//用得到的地址尝试访问cache，看是否命中等等
void trycache(unsigned int address){
    int set_address = (address >> b) & ((-1U) >> (64 - s)); //获取组索引
    int tag_address = address >> (b + s);   //获取标记位

    int max_stamp = INT_MIN;
    int max_stamp_index = -1;

    for(int i = 0; i < E; i++){  //查看是否命中
        if(_cache_[set_address][i].tag == tag_address && _cache_[set_address][i].valid_bits == 1){
            _cache_[set_address][i].stamp = 0; //更新时间戳
            hit_count++;  //命中
            if(v){
                printf("hit\n");
            }
            return;
        }
    }
```

```
    for(int i = 0; i < E; i++){   //若未命中，查看是否有空行
        if(_cache_[set_address][i].valid_bits == 0){
            _cache_[set_address][i].valid_bits = 1;
            _cache_[set_address][i].tag = tag_address;
            _cache_[set_address][i].stamp = 0;
            miss_count++;   //未命中，但也未进行替换
            if(v){
                printf("miss\n");
            }
            return;
        }
    }
```

```
//未命中且没有空行，说明要进行替换
for(int i = 0; i < E; i++){   //循环找时间戳最大的行进行替换
    if(_cache_[set_address][i].stamp > max_stamp){
        max_stamp = _cache_[set_address][i].stamp;
        max_stamp_index = i;
    }
}
_cache_[set_address][max_stamp_index].tag = tag_address;
_cache_[set_address][max_stamp_index].stamp = 0;
eviction_count++;
miss_count++;
if(v){
    printf("eviction\n");
}
return;
```

- 更新时间戳的函数 update_stamp，将所有有效的行的时间戳+1

```
//更新时间戳
void update_stamp(){
    for(int i = 0; i < S; i++){
        for(int j = 0; j < E; j++){
            if(_cache_[i][j].valid_bits == 1){
                _cache_[i][j].stamp++;
            }
        }
    }
}
```

- 对不同的指令(L、M、S)分类进行操作，每次操作完都更新时间戳，还要注意在最后将 malloc 申请的空间释放（free）

```
    for(int i = 0; i < S; i++){
        free(_cache_[i]);
    }
    free(_cache_);
}
```

- 进行验证，结果正确

```
[2020202279@work122 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.Part B

（1）32x32

- 根据说明，s = 5, E = 1, b = 5，所以 cache 共有 32 组，每组一行，每行存储 32 字节（8 个 int）。整个 cache 一共可以存放矩阵的 8 行。采取分块策略，用 8x8 分块进行优化。但由于 AB 相同位置的元素是映射在同一个 cache line 上的，所以对于处在对角线上的区域，会产生大量冲突不命中，比如 A 的第四行被 B 的第四行覆盖后，又要再次访问 A 的第四行的情况。所以利用 8 个局部变量 t1-t8，一次性取出一个块中所有元素，

之后不在访问这个块，以此减少冲突不命中的情况。

```
int i, j, k, h;
int t1, t2, t3, t4, t5, t6, t7, t8;
if(M == 32 && N == 32){
    for(i = 0; i < N; i += 8){
        for(j = 0; j < M; j += 8){
            for(k = i; k < i + 8; k++){
                t1 = A[k][j]; t2 = A[k][j + 1]; t3 = A[k][j + 2]; t4 = A[k][j + 3];
                t5 = A[k][j + 4]; t6 = A[k][j + 5]; t7 = A[k][j + 6]; t8 = A[k][j + 7];

                B[j][k] = t1; B[j + 1][k] = t2; B[j + 2][k] = t3; B[j + 3][k] = t4;
                B[j + 4][k] = t5; B[j + 5][k] = t6; B[j + 6][k] = t7; B[j + 7][k] = t8;
            }
        }
    }
```

- 经验证，miss 数为 287，满足要求

（2）64x64

- 对于 64x64 的矩阵，cache 只能存放矩阵的 4 行。所以如果还使用 8x8 的分块，会因为映射到相同的块，在按列访问 B 时产生大量的冲突不命中。因此，希望在同样的 miss 次数下，把更多的数据先暂存到 B 中的其他位置，在之后的步骤中再进行移动。考虑如下步骤：1.先把 A 的前四行（4x8）转置存入 B 的前四行，这样前四行前四列就已转置完成，而后四列存放在 B 的右上角，不必再访问 A 的前四行；2.按列依次读取 A 的左下角，按行读取 B 的右上角，都暂存在变量 t1-8 中，然后分别按行填入 B 的右上角和 B 的左下角；3.把 A 的右下角填入 B 的右下角。按照这个顺序访问，B 数组每块的元素中不命中数可以明显减少，达到要求。

```
    if(M == 64 && N == 64){
        for(i = 0; i < N; i += 8){
            for(j = 0; j < M; j += 8){
                for(k = i; k < i + 4; k++){
                    t1 = A[k][j]; t2 = A[k][j+1]; t3 = A[k][j+2]; t4 = A[k][j+3];
                    t5 = A[k][j+4]; t6 = A[k][j+5]; t7 = A[k][j+6]; t8 = A[k][j+7];

                    B[j][k] = t1; B[j+1][k] = t2; B[j+2][k] = t3; B[j+3][k] = t4;
                    B[j][k+4] = t5; B[j+1][k+4] = t6; B[j+2][k+4] = t7;B[j+3][k+4] = t8;
                }
                for(k = j; k < j + 4; k++){
                    t1 = B[k][i+4]; t2 = B[k][i+5]; t3 = B[k][i+6]; t4 = B[k][i+7];
                    t5 = A[i+4][k]; t6 = A[i+5][k]; t7 = A[i+6][k]; t8 = A[i+7][k];

                    B[k][i+4] = t5; B[k][i+5] = t6; B[k][i+6] = t7; B[k][i+7] = t8;
                    B[k+4][i] = t1; B[k+4][i+1] = t2; B[k+4][i+2] = t3; B[k+4][i+3] = t4;
                }
                for(k = j + 4; k < j + 8; k++){
                    t1 = A[i+4][k]; t2 = A[i+5][k]; t3 = A[i+6][k]; t4 = A[i+7][k];
                    B[k][i+4] = t1; B[k][i+5] = t2; B[k][i+6] = t3; B[k][i+7] = t4;

                }
            }
```

- 经验证，miss 数为 1219，满足要求

（3）61x67

- 由于矩阵规模不再是 8 的倍数，所以需要多次尝试不同的分块策略。经过尝试各种大小规模的分块，发现 17x17 的分块可以使 miss 数达到 2000 以下，满足要求，所以使用该分块策略。

```
    if(M == 61 && N == 67){
        for(i = 0; i < N; i += 17){
            for(j = 0; j < M; j += 17){
                for(k = i; k < i + 17 && k < N; k++){
                    for(h = j; h < j + 17 && h < M; h++){
                        B[h][k] = A[k][h];
                    }
                }
            }
        }
```

- 经验证，miss 数为 1950，满足要求

```
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        1950
          Total points    53.0        53
[2020202279@work122 cachelab-handout]$
```