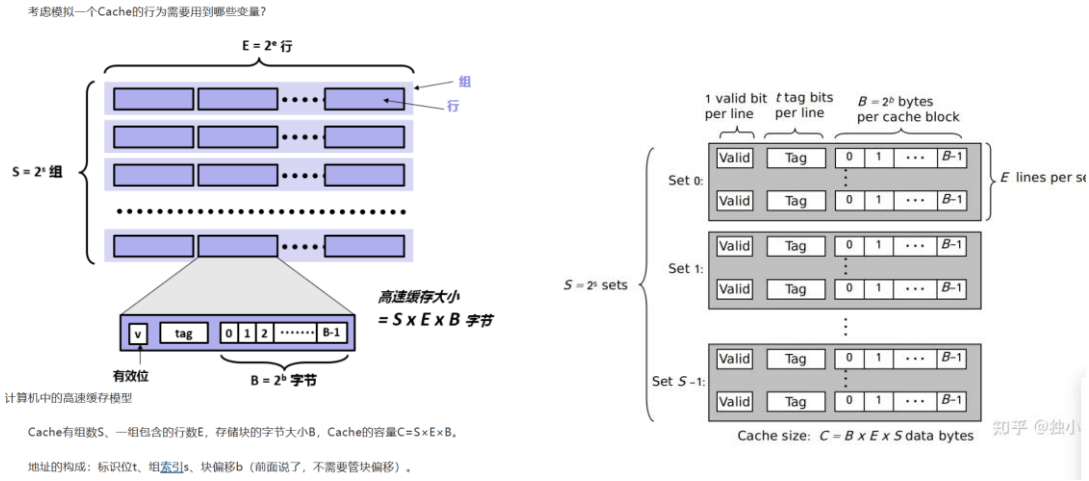
Cache lab 实验报告
姓名：谢志康
学号：22307110187
Part A：
cache 的替换策略为 LRU 算法，因此在 part1 中就模拟一个 LRU 算法实现即可。
LRU 原理与算法实现 - 知乎 (zhihu.com) --- LRU 算法通过这篇文章初步学习
24 张图 7000 字详解计算机中的高速缓存 - 知乎 (zhihu.com) --- 了解高速缓存模型
以下结构定义、函数实现主要根据上述两篇文章按步骤执行。

首先分析一下 csim.c 和 csim.h 中定义的参数
1. ‘h’ 是给出帮助（提示）的意思，这个 usage 函数就是打印帮助信息
2. 在头文件中定义了 verbose=0，若输入‘v‘则 verbose=1；是否打印每次匹配的详细信息
3. 这三个操作是相同的。Optarg（指针）在头文件中定义

- -s 代表cache的set数
- -E 代表每个set中的cache line数
- -b 代表cache line的大小（单位为字节）

4. 由 c 函数这里可知，定义的 numSet 就是 set 的个数，associativity 就是 e（每个 set 中 cache line 的大小），blocksize 是 b 是 cache line 的大小，单位为字节。

在本实验中，模拟的 cache 并不遵从全部组数（S）和 cacheline 为 2 的幂次的一般规定，因此这里全部组数的大小设为 numSet 同等大小就够了（当然开大些肯定也没问题）
普通计算机中高速缓存模型：

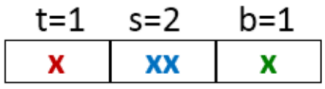


实现——
首先定义一个 cache line 的结构体
初始化函数写在头文件中
接下来实现更新高速缓存的函数：首先要明晰地址如何偏移到准确位置。
（我们可以将高速缓存存储器视为有 S 个高速缓存组的数组 。每个组包含 E 个高速缓存行 。每个行是由一个$B = {2^b}$字节的数据块组成的。）

简单模拟高速缓存模型如下：

4.4 模拟直接映射缓存

下面，我们模拟下直接映射高速缓存的过程，以便加深理解高速缓存是如何工作的。假设，内存地址为4字节，S=4组，E=1行/组，B=2字节/块。 其结构图如下所示。



当 S 和 cacheline 为 2 的幂次时：当我们有一个 address 时，可以通过 address>>b 得到 t、s，再通过取低 s 位，即 0xffffffff>>(32-s) 相与操作得到（eg: s==10，0xffffffff>>22 位，也就低位 10 个 1 与上述 address>>b 相与，取出 s

∴ set_index = (address>>b) & (0xffffffff>>(32-s));

同理：target_index = address>>(s+b);

而当它们并不是 2 的幂次时：我们不能简单使用移位来索引位置，直接利用除法就好，地址整体除以 b，将 s 放置在低位，之后对其大小取余得到有效的位数（非 2 的幂次，也就是有效的值）。Target 同理，地址除以后两项相乘，将 target 放在低位取出有效值。

（target_index 那里有很神奇的事是：后面写成 blockSize+numSet 也是能过的（））

```
set_index = (address / blockSize) % numSet;
target_index = address / (blockSize * numSet);
```

接下来就可以定义 update_cache 函数，更新高速缓存。

采用 LUR 策略，时间最久没用的就被更新，每个单元设置一个 time，初始值为-1，激活后为 0。当 evict 时重置时间为 0，整体时间最大的也就是最久没有使用的。

1. hit：目标位索引到相等，且该处已经激活

```
if (cache[set_index][i].target == target_index
    && cache[set_index][i].valid_bits == 1)
{
    //命中，其实若有target，肯定已激活
```

2. miss：若没有命中就是 miss。然后在当前 cacheline 行找有没有没激活的，有的话激活它存储起来。
3. 若没有（即当前行全部都激活了，没地方存新的了），执行 evict，找出当前 cacheline 时间最大的（最久没用的），更新掉，evict++

```
//高速缓存已满，且没有hit，需要淘汰一个最久没用的。
//找出最大时间，也就是存在最久没有用，LRU
int memotime = -1; int memoi = 0;
for (int i = 0; i < associativity; i++)
{
    if (cache[set_index][i].time > memotime)
    {
        memotime = cache[set_index][i].time;
        memoi = i;
    }
}
evictions++;
```

最后定义一个全局函数：每当执行一个操作后，整体 cacheline 所有时间都应该加一（整体 cache 的时间流控制）

```
void exist_time_increase()
{
    for (int j = 0; j < associativity; j++)
    {
        if (cache[set_index][j].valid_bits) cache[set_index][j].time++;
    }
}
```

最后 main 函数：’ M’修改数据操作，相当于先读再写，执行并更新，两次，其余两种操作都是执行并更新一次。

实验结果如下——

```
kurumi@kurumi:/mnt/d/another_C/cachelab-handout/cachelab-handout$ make
gcc -g -Wall -Werror -std=c99 -m64 -o csim csim.c cachelab.c -lm
kurumi@kurumi:/mnt/d/another_C/cachelab-handout/cachelab-handout$ ./test-csim
                        Your simulator     Reference simulator
Points (s,E,b)    Hits  Misses  Evicts    Hits  Misses  Evicts
     3 (1,1,1)       1      16      15       1      16      15  traces/yi2.trace
     3 (4,2,4)       3       6       3       3       6       3  traces/yi.trace
     3 (2,1,4)       0       5       4       0       5       4  traces/dave.trace
     3 (2,1,3)      79     159     157      79     159     157  traces/trans.trace
     3 (2,2,3)     128     110     106     128     110     106  traces/trans.trace
     3 (2,4,3)     180      58      50     180      58      50  traces/trans.trace
     3 (5,1,5)     149      89      84     149      89      84  traces/trans.trace
     6 (5,1,5)  165997  120967  120962  165997  120967  120962  traces/long.trace
    27

TEST_CSIM_RESULTS=27
```

Part B:

目的——cache miss 的次数尽可能少。

第一部分 48*48

首先来分析一下我们用到的 cache，s = 48，E = 1，b = 48，即每个 cache line 大小为 48 字节，共有 48 个 cache line，每个 set 中只有 1 个 cache line——也就是，共 48 组，每组能存下 12 个 int 变量（还就那个要求至多能开 12 个 int 局部变量（不包含循环变量）当然是选择用满））。

Cache 总共能存下 12*48 个 int，1 中数组有 48*48 个 int，也就是，把 cache 存满后，能存下数组中的前 12 行（1/4）。选择采用经典的分块技术，由于我们刚好有 12 个自由变量可以用，正好存一个 cache line，以空间换时间，把一行一次性读完，就可以减少 cache 中的冲突 miss。

```
for (int t = 0; t < 12; t++) v[t] = A[k][j + t];
for (int t = 0; t < 12; t++) B[j + t][k] = v[t];
```

利用局部变量暂存并实现转置，避免对 cache 的重复加载。

仅仅在模板代码中加上这一步后，就能拿到<450 次 miss 的成绩：

```
kurumi@kurumi:/opt/cachelab-handout$ sudo ./test-trans -M 48 -N 48

Function 0 (1 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=48, E=1, b=48)
func 0 (Transpose submission): hits:4181, misses:434, evictions:386

Summary for official submission (func 0): correctness=1 misses=434

TEST_TRANS_RESULTS=1:434
```

第二部分 96*96

12*48/96==6

这次 cache 存满只能存下数组的六行了，所以如果像 1 中使用 12 x 12 的分块，一定会在写入 B 的时候造成大量冲突 miss，因为映射到了相同的块。

6*12 → 12*6： 势必造成 6*6 的 int 并没有用到，这样的配置会导致每一个 A 的 cache 块只有 6 个 int 数据会被利用到，而其余 6 个数据需要下次载入才可利用，产生大量 miss。（这个有点难想，借用了网上 cache lab 的思路）

在这 12*12 的框架再细分成 4 个 6*6 的块，为了能够将浪费的 6 个 int 数据有效利用起来，所以可以考虑将多的数据暂时放入数组 B 的 cache 中，以待后续的操作，这样就可以避免二次载入相同的 cache 块，极大降低 miss 次数。

总体代码结构与 1 相似。细分部分——（按象限分为第 1，2，3，4 块）

1） 将细分的四块中的 1，2 两块转置：（2 先存好，待处理）

```
for (int x = i; x < i + 6; ++x)
{
    for (int t = 0; t < 12; t++) v[t] = A[x][j + t];
    for (int t = 0; t < 6; t++) B[j + t][x] = v[t];
    for (int t = 0; t < 6; t++) B[j + t][x + 6] = v[t + 6];
}
```

2） 将 2 转到 3，同时，将原本的 3 转到 2

```
for (int y = j; y < j + 6; ++y)
{
    for (int t = 0; t < 6; t++) v[t] = A[i + t + 6][y];
    for (int t = 0; t < 6; t++) v[t + 6] = B[y][i + t + 6];
    for (int t = 0; t < 6; t++) B[y][i + t + 6] = v[t];
    for (int t = 0; t < 6; t++) B[y + 6][i + t] = v[t + 6];
}
```

3） 将最后的 4 转到 4

```
for (int x = i + 6; x < i + 12; ++x)
{
    for (int t = 0; t < 6; t++) v[t] = A[x][j + t + 6];
    for (int t = 0; t < 6; t++) B[j + t + 6][x] = v[t];
}
```

综上，每个大块依旧如 1 中代码所示，暂存转置降低重复载入 cache 即可，只是为了每个大块利用率再高些，我们再将其分成四个小块，之后正常逻辑实现转置即可。

也拿到了荣誉分——成绩如下：

```
kurumi@kurumi:/opt/cachelab-handout$ sudo ./test-trans -M 96 -N 96

Function 0 (1 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=48, E=1, b=48)
func 0 (Transpose submission): hits:21257, misses:1790, evictions:1742

Summary for official submission (func 0): correctness=1 misses=1790

TEST_TRANS_RESULTS=1:1790
```

第三部分：93*99

这一部分其实没啥难度，无非就是不对称了，只需将其对称的部分按如上方式处理完，剩下的一些边角再单独处理即可（这部分数据较少，整体上不会带来太多的 miss）

```
for (int j = 0; j < M / 12 * 12; j += 12)
    for (int i = 0; i < N / 12 * 12; ++i)
```

让其为 12 的倍数，与 1，2 部分同理处理即可

剩下三个边角部分单独处理一下就行

成绩如下：

```
kurumi@kurumi:/opt/cachelab-handout$ sudo ./test-trans -M 93 -N 99

Function 0 (1 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=48, E=1, b=48)
func 0 (Transpose submission): hits:15757, misses:2772, evictions:2724

Summary for official submission (func 0): correctness=1 misses=2772

TEST_TRANS_RESULTS=1:2772
```

part C --honor part：

矩阵乘法算法，cache 参数：s = 32, E = 1, b = 32。32 行，每行能存下 4 个 int 数据。

https://www.cs.cmu.edu/afs/cs/academic/class/15213-s16/www/recitations/recitation07-cachelab.pdf

Blocked Matrix Multiplication: Code

```
c = (double*) calloc(n*n, sizeof(double));

void matrix_mult(double *a, double *b, double *c, int n) {
   int i, j, k;
   /* process each row of blocks */
   for (i=0; i<n; i+=B)
      /* process each column of blocks */
      for (j=0; j<n; j+=B)
         /* run the dot-product of the current row/column of blocks */
         for (k=0; k<n; k+=B)
            /* perform BxB mini matrix multiplications */
            for (int i1=i; i1<i+B; i1++)
               for (int j1=j; j1<j+B; j1++)
                  for (int k1=k; k1<k+B; k1++)
                     c[i1*n+j1] += a[i1*n+k1] * b[k1*n+j1];
}
```

boldface shows changes from non-blocked

30

```
kurumi@kurumi:/opt/cachelab-handout/honor-part$ sudo ./test-mul -M 32 -N 32

Function 0 (1 total)
Step 1: Validating and generating memory traces

Step 2: Evaluating performance (s=32, E=1, b=32)
func 0 (multiply submission): hits:121105, misses:10998, evictions:10966

Summary for official submission (func 0): correctness=1 misses=10998

TEST_MUL_RESULTS=1:10998
```

失败……

https://inst.eecs.berkeley.edu/~cs61c/fa22/labs/lab07/#exercise-2-loop-ordering-and-matrix-multiplication

整体分块逻辑是没有问题的，已经比爆做要好三倍不止了，但是还需优化。主要在循环的次序改变上，总访问次数一样时，改变访问顺序得当能极大降低访问 miss 次数。

（这里改了整整一天……最后在助教的提示下改成功了）

主要思路，分块后在内层——将((A+B)_8+C)_8 改为 (A_8+B_8)_8+C_8

AB 都访问处理好后，再对 C 进行操作，能极大降低 miss 次数。

核心代码如下——试着调整循环顺序（前期外层将 k 和 j 的顺序改变，大约减少了 100miss 左右。当时就算一个调整循环次序的 hint 了……）

```
    for (int jj = j; jj < j + 8; jj++) {
        for (int kk = k; kk < k + 8; kk++) {
            v[kk - k] = A[ii][kk];
            v[kk - k] *= B[kk][jj];
            tem[jj - j] += v[kk - k];
        }
    }
    for (int jj = j; jj < j + 8; jj++) {
        C[ii][jj] += tem[jj - j];
    }
```

最终 driver.py 截图如下——

```
kurumi@kurumi:/opt/cachelab-handout$ sudo python2 driver.py
part A: Testing cache simulator
the following are original points, which will be transformed into the final points
Running ./test-csim
                        Your simulator     Reference simulator
Points (s,E,b)    Hits  Misses  Evicts    Hits  Misses  Evicts
     3 (1,1,1)       1      16      15       1      16      15  traces/yi2.trace
     3 (4,2,4)       3       6       3       3       6       3  traces/yi.trace
     3 (2,1,4)       0       5       4       0       5       4  traces/dave.trace
     3 (2,1,3)      79     159     157      79     159     157  traces/trans.trace
     3 (2,2,3)     128     110     106     128     110     106  traces/trans.trace
     3 (2,4,3)     180      58      50     180      58      50  traces/trans.trace
     3 (5,1,5)     149      89      84     149      89      84  traces/trans.trace
     6 (5,1,5)  165997  120967  120962  165997  120967  120962  traces/long.trace
    27


part A final points: 27 * (40 / 27) =     40.0

Part B: Testing transpose function
Running ./test-trans -M 48 -N 48
Running ./test-trans -M 96 -N 96
Running ./test-trans -M 93 -N 99
Running honor-part

Cache Lab summary:
                        Points   Max pts      Misses
Csim correctness          40.0        40
Trans perf 48x48          14.0        14         434
Trans perf 96x96          14.0        14        1790
Trans perf 93x99          12.0        12        2772
Trans perf honor-part      5.0         5        3930
          Total points    85.0        85
```