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)](https://zhuanlan.zhihu.com/p/161269766) --- 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)](https://zhuanlan.zhihu.com/p/341238720?utm_id=0) --- 了解高速缓存模型

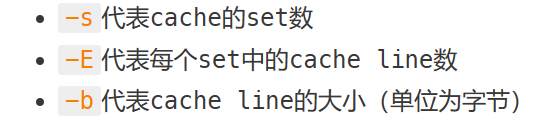
以下结构定义、函数实现主要根据上述两篇文章按步骤执行。

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

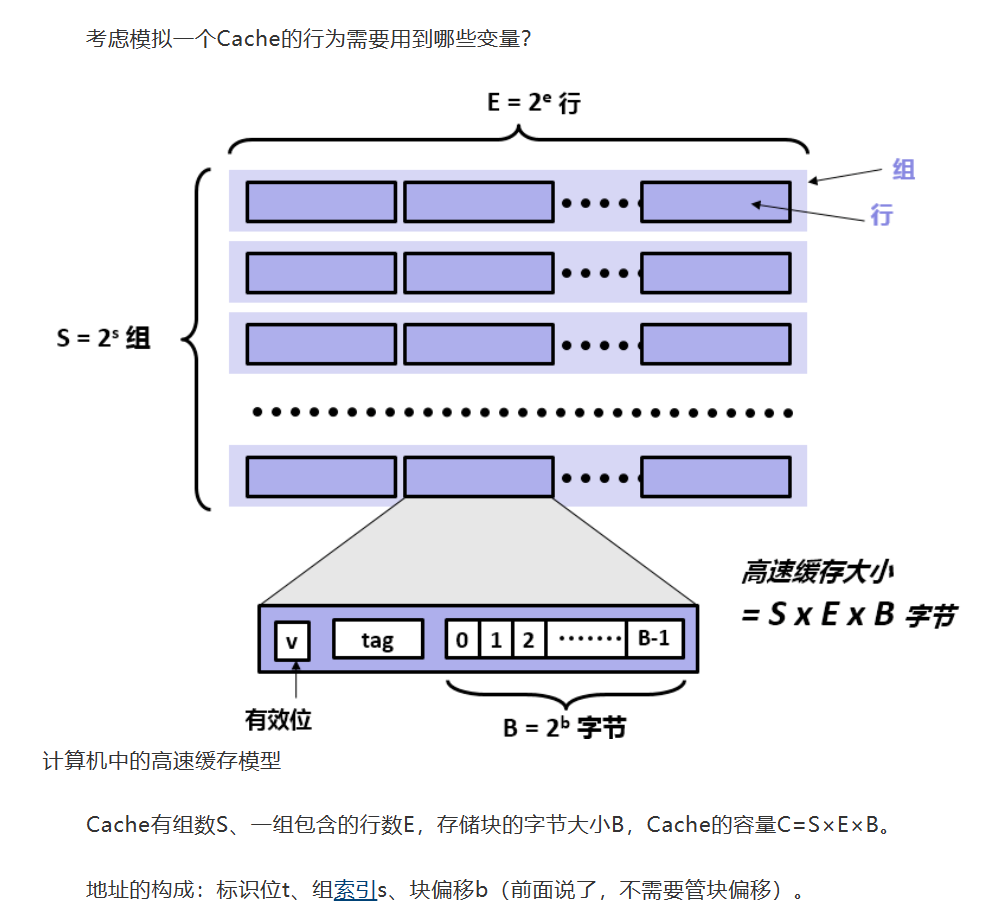
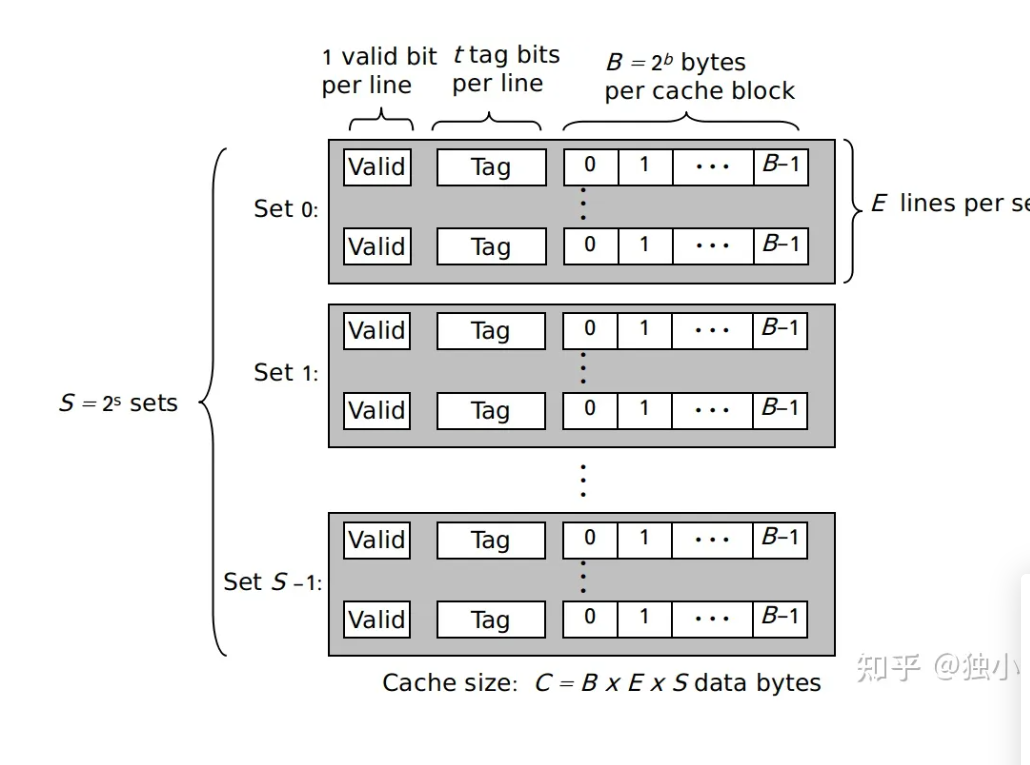
3.这三个操作是相同的。Optarg（指针）在头文件中定义



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——

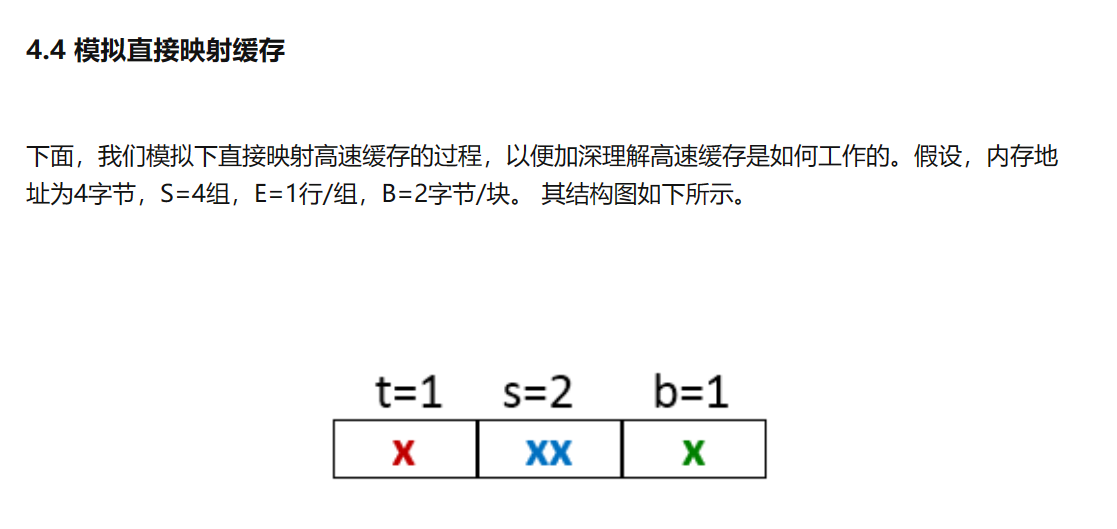
首先定义一个cache line的结构体

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：

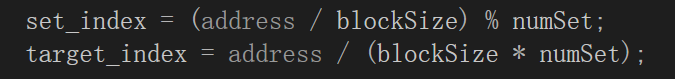


当S和cacheline为2的幂次时：当我们有一个address时，可以通过 address>>b 得到t、s，再通过取低s位，即0xffffffff>>(32-s) 相与操作得到（eg: s==10，0xffffffff>>22位，也就低位10个1与上述address>>b相与，取出s

∴ set\_index = (address>>b) & (0xffffffff>>(32-s));

同理：target\_index = address>>(s+b);

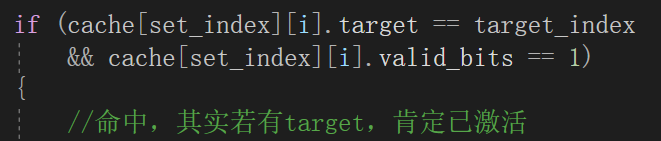
而当它们并不是2的幂次时：我们不能简单使用移位来索引位置，直接利用除法就好，地址整体除以b，将s放置在低位，之后对其大小取余得到有效的位数（非2的幂次，也就是有效的值）。Target同理，地址除以后两项相乘，将target放在低位取出有效值。（target\_index那里有很神奇的事是：后面写成blockSize+numSet也是能过的（））



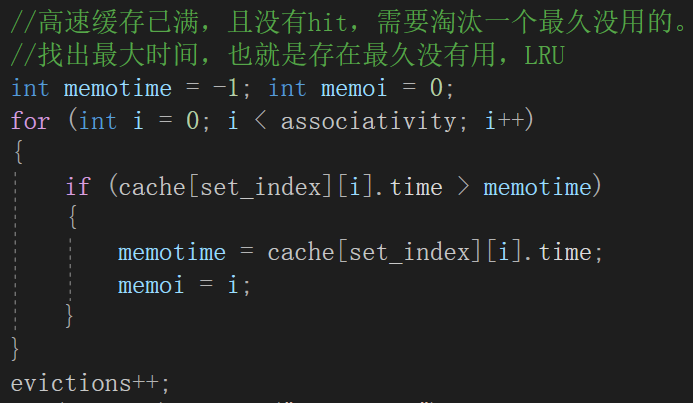
接下来就可以定义update\_cache函数，更新高速缓存。

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。

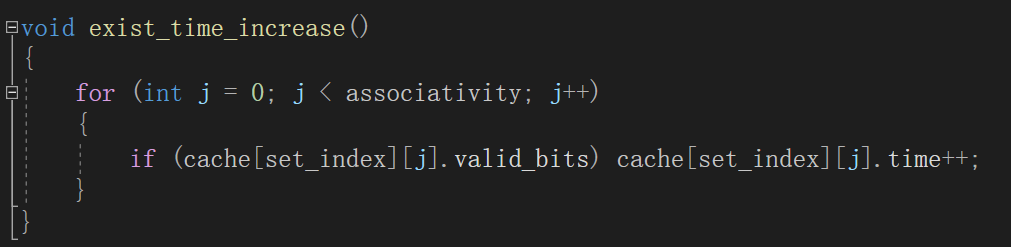
1. hit：目标位索引到相等，且该处已经激活



1. 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。
2. 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++

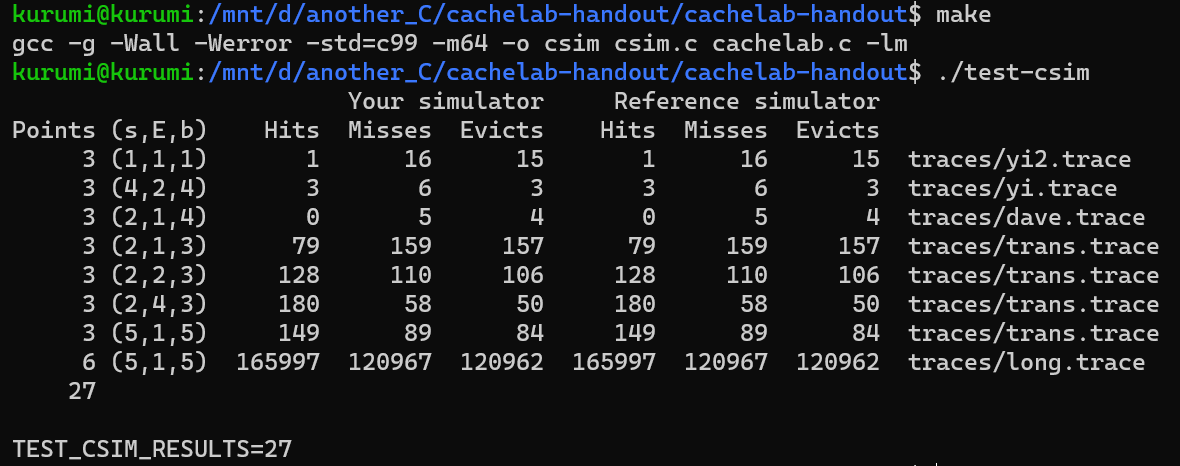


最后定义一个全局函数：每当执行一个操作后，整体cacheline所有时间都应该加一（整体cache的时间流控制）



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——



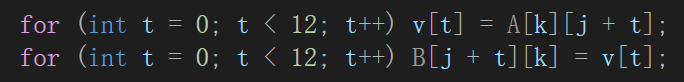
Part B:

目的——cache miss的次数尽可能少。

第一部分 48\*48

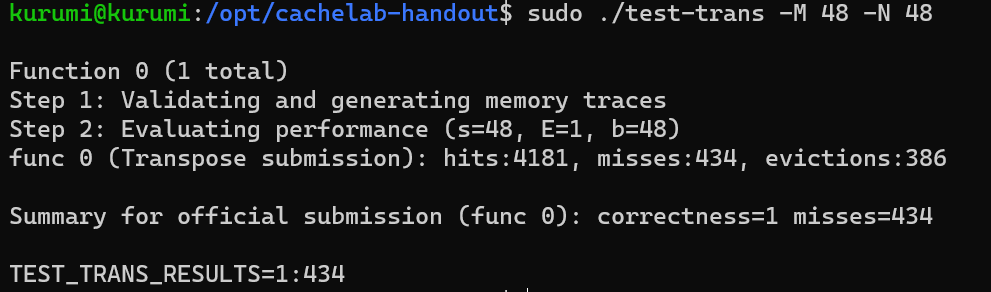
首先来分析一下我们用到的cache，s = 48, E = 1, b = 48，即每个cache line大小为48字节，共有48个cache line，每个set中只有1个cache line——也就是，共48组，每组能存下12个int变量（还就那个要求至多能开12个int局部变量（不包含循环变量）当然是选择用满））。

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。



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：



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。

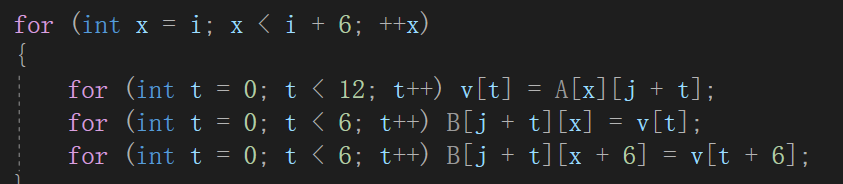
6\*12 🡪 12\*6： 势必造成6\*6的int并没有用到，这样的配置会导致每一个A的cache块只有6个int数据会被利用到，而其余6个数据需要下次载入才可利用，产生大量miss。

（这个有点难想，借用了网上cache lab的思路）

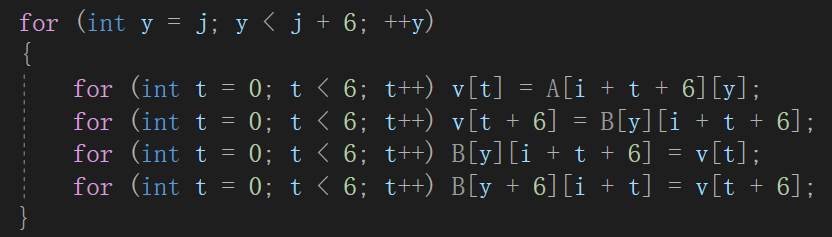
在这12\*12的框架再细分成4个6\*6的块，为了能够将浪费的6个int数据有效利用起来，所以可以考虑将多的数据暂时放入数组B的cache中，以待后续的操作，这样就可以避免二次载入相同的cache块，极大降低miss次数。

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）

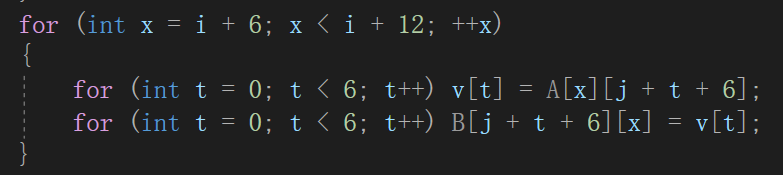
1. 将细分的四块中的1，2两块转置：（2先存好，待处理）



1. 将2转到3，同时，将原本的3转到2

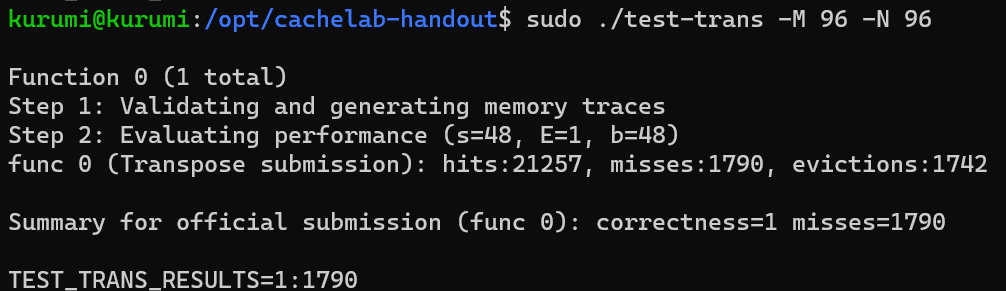


1. 将最后的4转到4



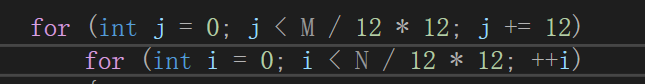
综上，每个大块依旧如1中代码所示，暂存转置降低重复载入cache即可，只是为了每个大块利用率再高些，我们再将其分成四个小块，之后正常逻辑实现转置即可。

也拿到了荣誉分——成绩如下：



第三部分：93\*99

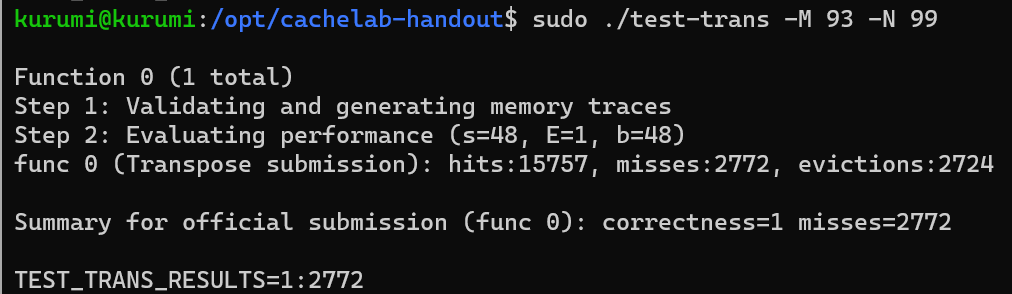
这一部分其实没啥难度，无非就是不对称了，只需将其对称的部分按如上方式处理完，剩下的一些边角再单独处理即可（这部分数据较少，整体上不会带来太多的miss）



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

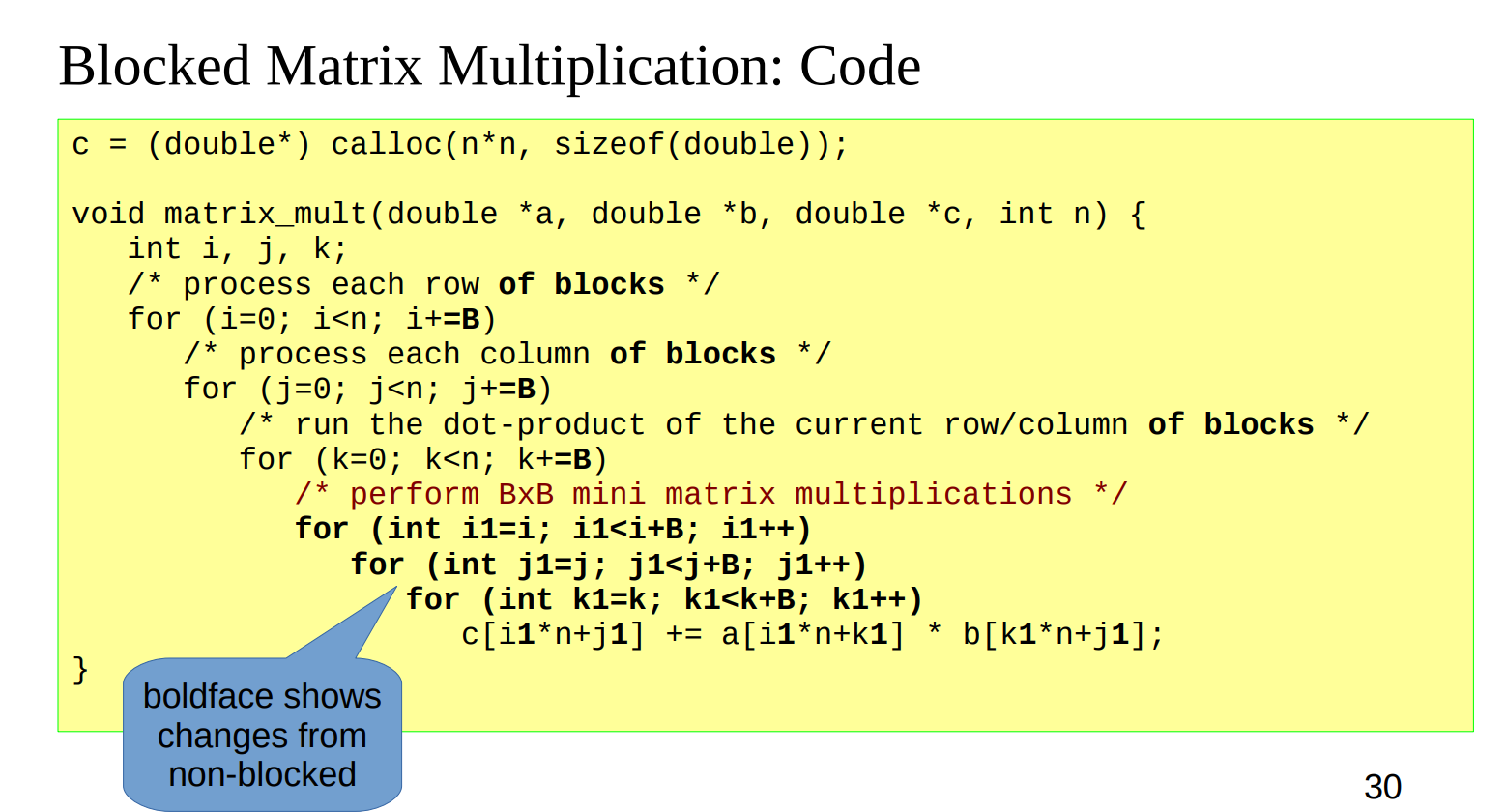
成绩如下：

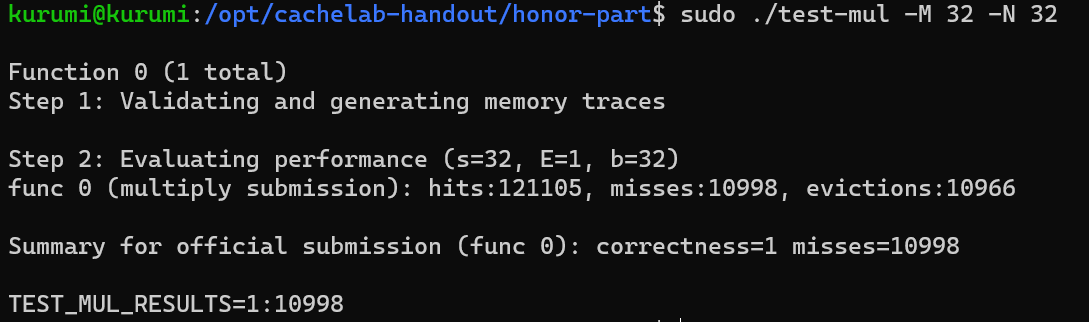


part C --honor part：

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>





失败……

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

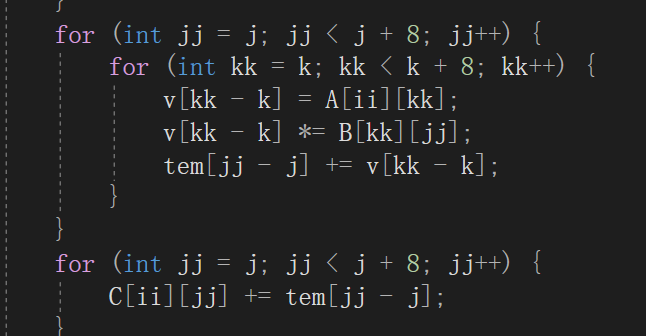
整体分块逻辑是没有问题的，已经比爆做要好三倍不止了，但是还需优化。主要在循环的次序改变上，总访问次数一样时，改变访问顺序得当能极大降低访问miss次数。

（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……）



最终driver.py截图如下——

