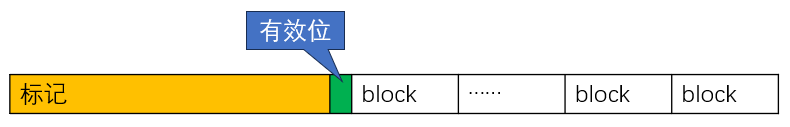
缓存是现代计算机的重要部件，了解缓存的工作机制可以编写出缓存友好的程序，提高程序的速度。

partA

实现一个缓存模拟器，需要了解缓存的工作细节，完成后可以对缓存有更深入的理解。这个实现没有唯一的答案，不同的人实现方式都有差别。相对难点是实现LRU替换，如果换错了行会导致后边缓存行为的一系列错误。具体原理书里已经讲的非常详细了，相信只要理解了实现起来比较容易。我的代码见附1，需要的可以参考。

对于我代码的几点说明：

1. 我把行中的有效位和标记位放到了一个64位的变量中，有效位在最低位，标记位是高63位；



1. 组中的每一行用一个数字表示其访问情况，数字最大（E-1）的表示刚刚访问过，数字最小（0）表示距上次访问最久远，每次命中、miss、替换都要调整。
2. 模拟实际不需要为block分配空间，只需要有效位和标记即可，我刚开始不知道就分配了，后续懒得改了。

partB

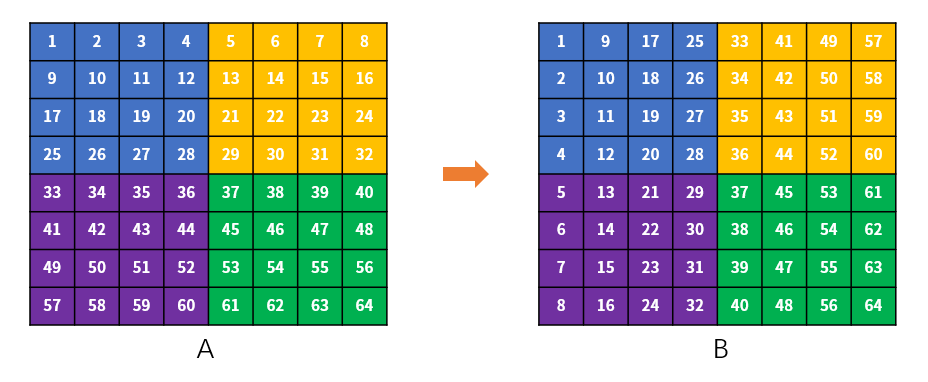
这个稍微有点难度，同样考察对缓存的理解，不过是从应用的角度。即便在partA中实现了缓存模拟器，到应用阶段还是会遇到很多问题，可见一个概念想要真正熟悉需要从不同的角度不断练习。

这部分的核心思想是数据加载到行中，应用尽用，尽可能让这行数据只加载这一次，以后就不需要再换回来。

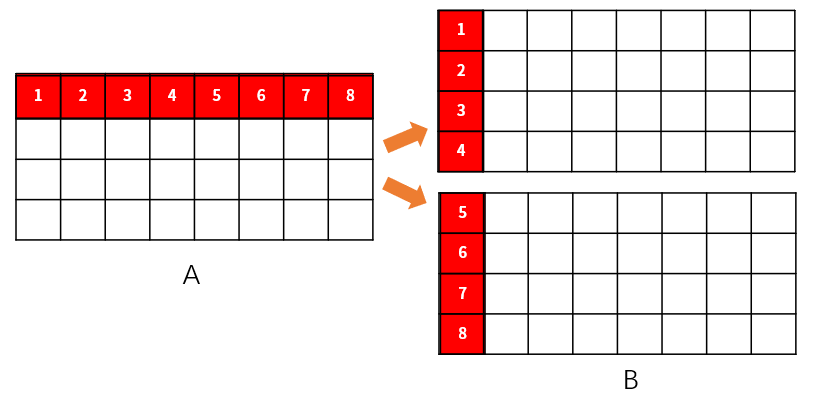
缓存大小是1k，块大小是32字节，矩阵元素是int型，4字节，所以一个块可以放8个数，根据前面的原则，在处理时要尽可能一次处理8个数。另外还有个技术是blocking（分块），利用分块可以减少miss数，具体原理可以看这篇官方手册推荐的文章<http://csapp.cs.cmu.edu/public/waside/waside-blocking.pdf>。缓存只有1k，所以对于32x32的矩阵可以缓存8行而没有冲突，64x64的矩阵可以缓存4行而没有冲突。由于有这个区别，很多人的处理方法是分开处理，32x32专门写一个程序，64x64专门写一个，61x67专门写一个。当然这样可以拿满分，不过我觉得不够优雅，所以我想用一个程序实现对三种矩阵的处理，最后虽然没有拿到满分（差0.1），但是却收获满满。

具体的方法真的不太容易想到，我也是网上看到再优化的，下面具体来说说。

我们的目的是从矩阵A变到矩阵B，先来分析哪些情况可能会出现问题。



如果一次一行处理8个数，32x32没什么问题，但是64x64不行，存前4个数B加载前4行，但是存后4个数的时候，由于64x64一次只能缓存4行，所以矩阵B的后四行会与前4行发生替换，下一次还会重复这个过程，这样一直换入换出很明显命中率不高。



如果一次处理4个数，而缓存一行可以加载8个数，很明显缓存利用率只有1/2，剩下的4个数在处理前很可能会被换出去，下次处理再加载回来，命中率还是不高。

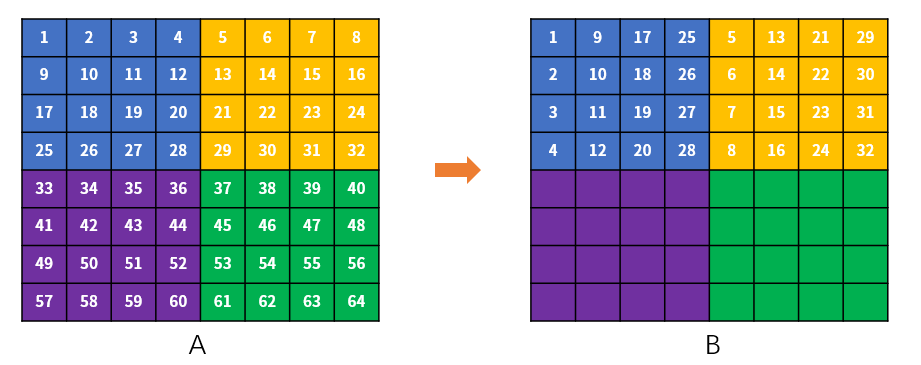
还有一种情况是A和B的某些地址位于相同的行，这样存B的时候会把A换出去，下次读A的数据又要加载进来，这样的命中率也会下降。

针对以上问题，我们提出以下方案，

1. 首先分块是8x8，一次大循环处理8行8列，8x8的块进一步分成4x4的块；
2. 用8个中间变量存需要处理的数据，这样即便有冲突被换出去也没关系；
3. 用B的空间暂存A后四个数据，这样A即便被换出去也没关系；

具体过程如下：

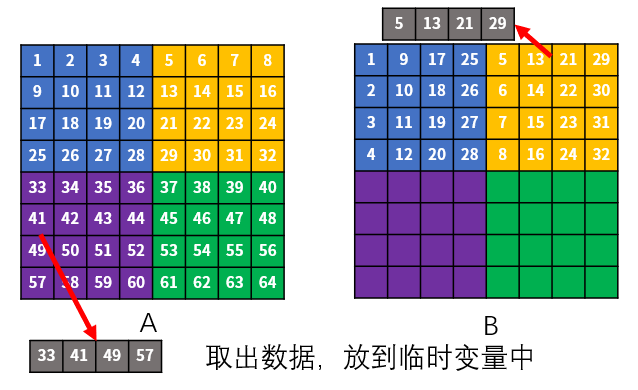
1. 先处理前4行8个数，前4列正常处理，A的后4列转置后暂存在B的后4列中。



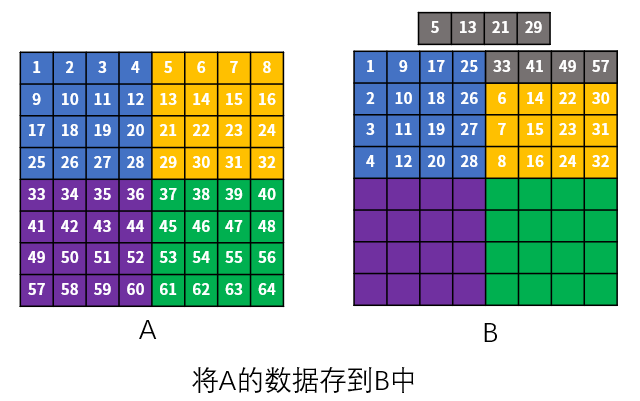
这里我们可以发现，B右上角黄色块的数据就是最终B紫色块的数据，接下来我们想办法复制过去就行。

1. 下面的一系列处理很巧妙，充分利用了缓存的特点。

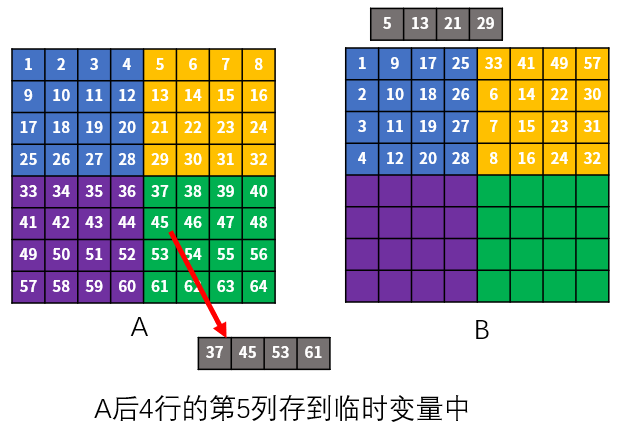
先把B的第一行后4列暂存到临时变量中，把A的后4行的第一列存到临时变量中，如果是64x64，此时A的前4行被换了出去，缓存中是A的后4行；



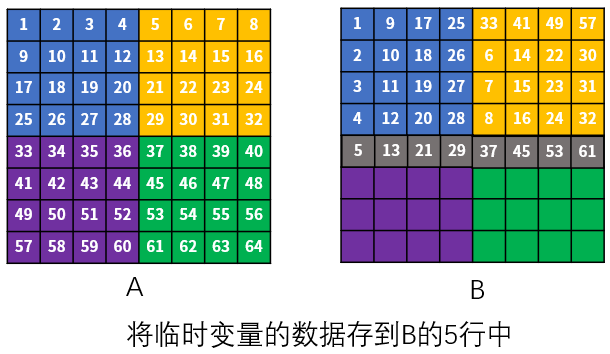
将暂存的A数据存到B的第一行后4列，此时B的第一行已经是最终的数据；



取出A的后四行的第5列存入临时变量中，此时A的后四行位于缓存中，可以命中。

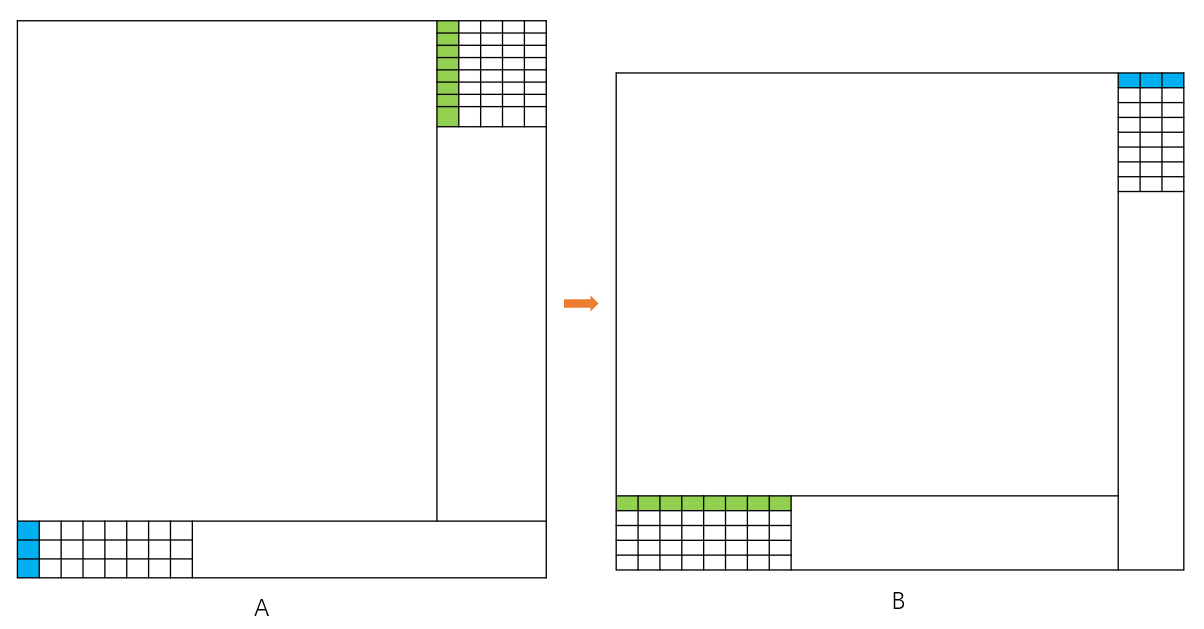


将临时变量的值存到B的第5行。如果是64x64，此时B的第5行会把第一行从缓存中换掉，而第一行已经是最终的数据，换出去也没关系。

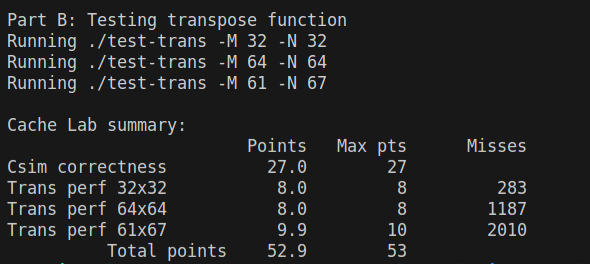


以此类推，将上述过程再重复三次，处理剩下的数据。此过程充分利用了缓存中的数据，换出去的行以后不会再访问，大大提高了缓存命中率。

对于61x67，由于行列不是8的整数倍，所以经过上面的处理后会有遗漏，需要单独处理。对于列，一次取出一列8个数据，实测这样的命中率对于没对齐的矩阵命中率很高，对于剩余的行，同样按列读取，按行存命中率要高。



使用上面的方法，测试结果如下：



32x32和64x64的表现都很好，61x67多了10个miss，这10个miss实在不知道怎么优化了。

具体实现的代码如下

这个lab个人觉得还是有点难度的，尤其是之前完全不了解缓存机制的情况下，更是让人迷惑。不过，正是有这些疑惑，才逼自己一行行的分析trace，看看程序的实际行为如何，进而发现问题，找到解决的办法，这个过程中也不断的加深了对缓存的理解和认识。

附1：