Part B

一、Part B概述：

（Ⅰ）任务：

①编写一个实现矩阵转置的函数。即对于给定的矩阵A[N][M]，得到矩阵B[M][N]，使得对于任意0<=i<N、0<=j<M，有B[j][i]=A[i][j]，

并且使函数调用过程中对cache的不命中数miss尽可能少。

②在如下函数里面编写最终代码：

char transpose\_submit\_desc[] = "Transpose submission";

void transpose\_submit(int M, int N, int A[N][M], int B[M][N])；

（Ⅱ）测试用例：

用三种不同规模的数组进行测试，规模分别为：

• 32 × 32 (M = 32, N = 32)

• 64 × 64 (M = 64, N = 64)

• 61 × 67 (M = 61, N = 67)

（Ⅲ）细节说明：

①可以编写多个函数来比较不同版本函数的性能，每个函数用如下方式进行注册：

registerTransFunction(transpose\_submit, transpose\_submit\_desc);

将transpose\_submit替换为函数名即可。

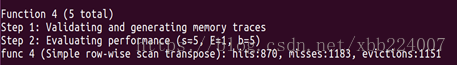
②在test-trans运行后会生成包含函数运行过程中的所有操作（L、S）的文件trace.fi，通过使用Part A中的模拟器以及该文件可以对函数运行的过程进行跟踪分析。

二、示例分析：

（Ⅰ）示例函数运行结果（以M32N32为例）：

①输入命令./test-tran -M 32 -N 32运行：





②同时可以看到生成了包含所有操作的文件trace.fi，我这里注册了四个函数，所以生成了五个文件如下：



③通过查看trace.f4的内容，可以发现其为函数运行过程中的所有操作：



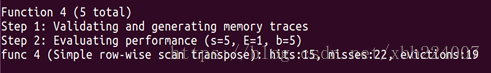
④通过使用Part A中的csim模拟器并用-v选项输出过程：



这里由于32×32的操作数过多，不好直接进行分析，下面以4×4规模的矩形进行具体分析。

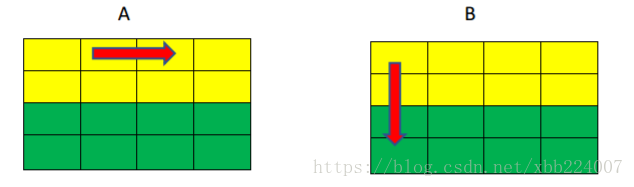
（Ⅱ）具体实例分析（以4×4的矩阵为例）：

①运行得到结果：



可以看到示例函数的misses数较多，下面进行misses过多的原因进行分析。

②为了更好地分析cache的命中与不命中的结果，我们将数组中元素占用高速缓存块的情况标志出来，由于cache的块的大小为32个字节，即8个int型数据，故数组中前8个元素会在用一个块中，后8个在另外一个块中，如下图：



其中的箭头为两个数组的访问数组的顺序：A数组按行访问，B数组按列访问。

④结合模拟器跟踪结果以及上图进行具体的分析：

1、A数组访问A[0][0]，冷不命中，将块11装入cache。

2、B数组访问B[0][0]，虽然B[0][0]所映射的块11在cache中，但是标记位不同，造成冲突不命中，重新将数组B对应的块11装入cache。

3、A数组访问A[0][1]，虽然A[0][1] 所映射的块11在cache中，但是标记位不同，造成冲突不命中，重新将数组A对应的块11装入cache。

4、B数组访问B[1][0]，虽然B[1][0]所映射的块11在cache中，但是标记位不同，造成冲突不命中，重新将数组B对应的块11装入cache。

5、A数组访问A[0][2]，虽然A[0][2]所映射的块11在cache中，但是标记位不同，造成冲突不命中，重新将数组A对应的块11装入cache。

6、B数组访问B[2][0]，B[2][0] 所映射的块12不在cache中，冷不命中，将数组B对应的块12装入cache。

7、A数组访问A[0][3]，A[0][3]所映射的块11在cache中，且标记位相同，故命中。

8、B数组访问B[3][0]，B[3][0]所映射的块12在cache中，且标记位相同，故命中。

9、A数组访问A[1][0]，A[1][0]所映射的块11在cache中，且标记位相同，故命中。

10、B数组访问B[0][1]，虽然B[0][1] 所映射的块11在cache中，但是标记位不同，造成冲突不命中，重新将数组B对应的块11装入cache。

11、A数组访问A[1][1]，虽然A[1][1]所映射的块11在cache中，但是标记位不同，造成冲突不命中，重新将数组A对应的块11装入cache。

12、B数组访问B[1][1]，虽然B[1][1]所映射的块11在cache中，但是标记位不同，造成冲突不命中，重新将数组B对应的块11装入cache。

13、A数组访问A[1][2]，虽然A[1][2]所映射的块11在cache中，但是标记位不同，造成冲突不命中，重新将数组A对应的块11装入cache。

14、B数组访问B[2][1]，B[2][1] 所映射的块12在cache中，且标记位相同，故命中。

15、A数组访问A[1][3]，A[1][3]所映射的块11在cache中，且标记位相同，故命中。

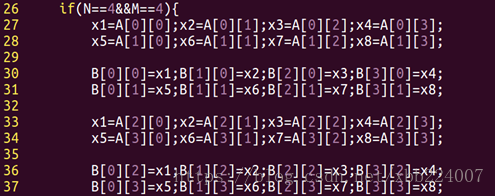
16、B数组访问B[3][1]，B[3][1]所映射的块12在cache中，且标记位相同，故命中。

17、剩余的操作以同样的方法可以分析得出类似的过程。

⑤由以上分析可以看出miss过多的原因是在访问两个数组的过程中存在太多的冲突不命中，而造成传统不命中的原因是B数组与A数组中下标相同的元素会映射到同一个cache块，如上述的2~6步骤，就是不断地发生了冲突不命中。

⑥改善方法：

由以上地分析可以想到，要减少miss数，自然就是要解决冲突不命中的问题。冲突不命中实际上就是在访问同一个块中的两个元素的时候，由于中间访问了其它的块，导致已经加载的块被驱逐，进而第二次访问时不命中。基于这个原因，我们可以一次性访问同一个块中的多个元素，访问完以后便不再需要访问这个块了，从而可以大大地减少冲突不命中的数目。这里前8个元素在同一个块中，我们可以直接将这8个元素取出来，然后这8个元素所在的块便不再需要访问了，具体代码如下：



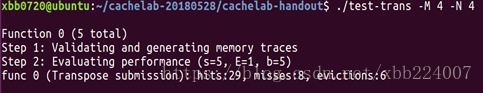
第27、28行就是取A数组中的前两行元素；

第30、31行就是对于A数组中前两行进行转置；

第33、34行就是取A数组中的后两行元素；

第36、37行就是对于A数组中后两行进行转置；

最终运行结果如下：



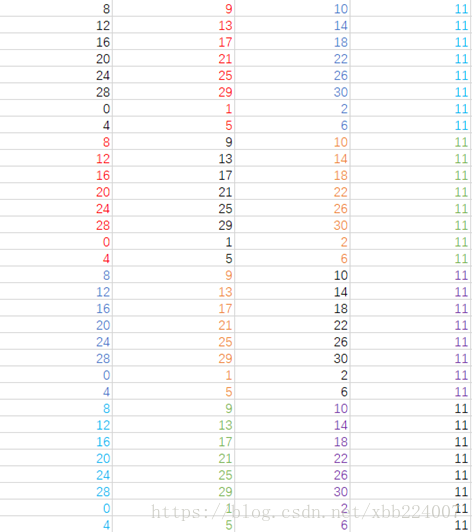
miss数由22减少为了8，显然有了一个非常大的提升，当然还有优化的空间，但这里不再展开。

三、测试用例分析（M32×N32）

（Ⅰ）分析矩阵中各元素所在块的情况：

同4×4的分析方法，我们分析各个元素所在块的情况，并将其标记：

由于每个块可以存4个int型的数据，cache一共有32个块，故对于32×32的矩阵而言，每一行的32个元素占4个组，每8行会占满整个cache，如下：

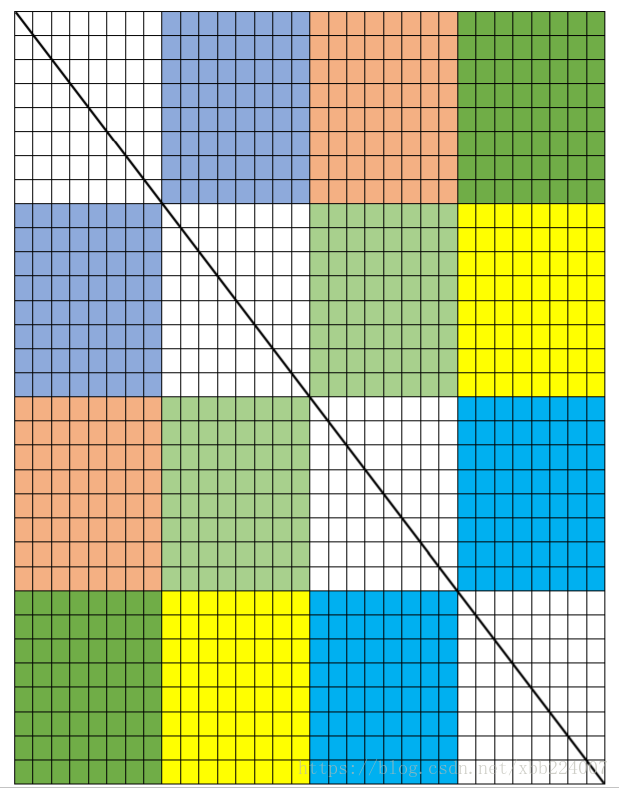


其中每一个格子代表一个块，即数组中的8个元素，每个格子中的数字表示组号，从0到31。

（Ⅱ）矩阵转置后元素的分布情况：

对于转置后的矩阵，每个元素的所在的位置与之前相比会关于矩阵的对角线对称，故转置后的情况可以由下图表示：

其中颜色相同的两个区域表示这两个区域的元素存在对应关系。



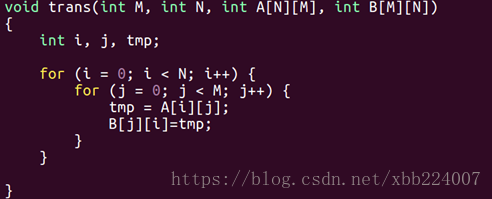
这里可能会有疑问，为什么标记对应区域的时候要标记8×8的区域呢？我们看到（Ⅰ）中的图，可以发现，以8×8来标记区域，那么两个区域所在的组是没有交集的，这也就意味着在这两个区域进行元素的转置时并不会发生冲突不命中的情况。而如果对于A数组直接按行访问，B数组直接按列访问就会得到大量的冲突不命中。下面展示这两种情况的具体结果。

（Ⅲ）转置过程中元素的访问情况：

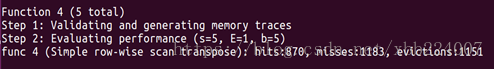
（Ⅳ）初步运行结果：

①A数组直接按行访问，B数组直接按列访问：

代码：



运行结果：



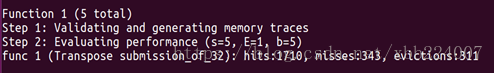
可以看到有大量的miss数。

②将数组进行8分块：

代码：



运行结果：



相比第一种情况而言miss已经有了十分明显的减少，但是依然达不到最优的要求（miss<300）。

（Ⅴ）继续优化（下面的分析比较繁琐，可选择性跳过）：

显然上面我们并没有考虑处于对角线上的区域（未涂色部分），而其余区域的不命中数已经达到了下限，即每一个块恰好不命中一次，所以我们接下来主要分析对角线上的区域。

由于对角线的元素转置以后的位置任然在同一个区域，所以很可能会有大量的冲突不命中，这十分类似于最前面分析的那个4×4的例子。所以我们同样根据4×4规模中的优化方法，即每次取一个块中的所有元素，取完以后便不再访问这个块来减少冲突不命中。

代码如下：



运行结果如下：



这样优化以后，miss数达到了287，已经满足最优的要求了。