ICS Lab4-Cachelab WriteUp

part A

实现方式很多，这里提供一种实现思路

cacheLine

```
struct cacheLine
{
    int valid;
    int tag;
    int counter;
};
```

模拟pipeline

- 初始化cache：`malloc`
- 读取trace文件
- 读取每一行的地址和操作，模拟cache操作（因为不涉及对主存的操作，所以读写cache是一样的，`modify`操作对应一次读一次写）
    - 计算地址对应的setIndex和tag
    - 检查hit还是miss
        - miss：LRU替换，采用简单的计数器算法，使用counter记录cacheLine未被使用的次数
            - 如果有空闲的cacheLine，直接使用
            - 如果没有空闲的cacheLine，找到counter最大的cacheLine，替换
        - hit：pass
    - 更新counter，读写的cacheLine的counter置为0，set中的其他cacheLine的counter加1
- 释放cache申请的内存：`free`

part B

采用分块的思路，每次处理一个块

cache参数：

- blockSize：48 byte = 12 * `sizeof(int)`
- asscociativity：1（意味着映射到同一个set必发生替换）
- numSet：48

48 * 48

- 一个块的大小为12 * `sizeof(int)`，即最多存放12个int
- 相邻行的地址相差48个int，即4个cacheLine，所以最多可以处理`numSet/4 = 12`行

- 所以采用`12 * 12`的分块策略，将大矩阵划分为许多个`12 * 12`的小矩阵，每次对一个`12 * 12`小矩阵进行处理
- 实现`12 * 12`分块即可通关荣誉部分
- 进一步优化可以考虑对角线冲突，先把A的一整个`12 * 12`块读进B，然后在B内部进行转置，这样就不会发生对角线冲突

96 * 96

- 一个块的大小为12 * `sizeof(int)`，即最多存放12个int
- 相邻行的地址相差96个int，即8个cacheLine，所以最多可以处理`numSet/8 = 6`行
- 所以采用`6 * 12`的分块策略，将大矩阵划分为许多个`12 * 12`的小矩阵，在小矩阵的处理中，最多对`6 * 12`的矩阵进行处理
- 要注意对于`12 * 12`的矩阵，上下两个`6 * 12`矩阵间是会发生冲突的，因为它们相差6行，即相差48个cacheLine，所以会映射到同一个set
- 不考虑对角线冲突（有`1/12`的几率不发生对角线冲突），即假设A和B之间是不会发生冲突的，只考虑A内部和B内部的冲突
- 把握局部性原理，已经在cache中的部分要尽可能去用，无论是现在用还是以后用

step 0

A

A1	**A2**
A3	**A4**

B

( )	**( )**
( )	**( )**

step 1

A

A1	**A2**	**cached**
A3	**A4**	

B

A1	**A2**	**cached**
( )	**( )**	

miss: 6 + 6 = 12

step 2

A

A1	A2	
A3	A4	cached

B：先将A3移动到B的右上角，再将替换下来的A2移动到B的左下角

A1	A3	
A2	( )	cached

miss = 6 + 6 = 12

step 3

A

A1	A2	
A3	A4	cached

B

A1	A3	
A2	A4	cached

miss = 0

实验

miss期望：(12 + 12) * (96 / 12) * (96 / 12) = 1536
实际miss：1788
相差不大，差距部分应该由对角线冲突引起

93 * 99

- 没什么规律，可以遍历查找合适的分块策略，12 * 12分块是能过的

part C

用基本的分块策略实现就能过，此处等同于`part B`中的`48 * 48`部分，应该采用`8 * 8`分块，每次计算一个`8 * 8`的矩阵乘以一个`8 * 8`的矩阵，然后将结果写入即可
采用质朴的方法计算`8 * 8`乘以`8 * 8`就可以过（每次计算一行点积一列）；矩阵乘法也可以看作“行”或“列”的线性组合，一次对一行进行操作，可以更优

参考代码

我在`trans.c`中用不同算法做了许多实验，大家可以自行体会每个算法有什么不同，得到的miss次数的理论值是多少，和实际值相差多少