CacheLab 报告

csim 分数	case1 speedup	case2 speedup	case3 speedup	weighted speedup
100.00	5.96	5.81	4.74	5.43

Part A: cache 模拟器

实现简述

把每个缓存行构建一个结构体，这个结构体存储着缓存行的有效位valid，tag值tag和替换数据LRU三个数据。

每次有新的数据需要读或者写的时候，先通过位运算和掩码操作得到b位，s位和tag，然后根据s位对应到对应的s组，和组内所有的缓存行的tag进行比较，然后根据valid判断是否应该进行以下三者操作其一：

1.缓存命中，hit加一

2.缓存未命中且有空行，替换空行，并且miss加一

3.缓存未命中且无空行，替换LRU值最高的行，miss和eviction都加一

三种情况在命中后，按照缓存定义维护该组内各个行的LRU，tag和valid数据即可

亮点

可以不存储缓存行的数据，因此本质上读操作和写操作对缓存的操作完全相同，可以共用同一套代码

Part B: 矩阵乘法优化

亮点

1.矩阵分块：通过将B矩阵分块操作，可以最大限度的利用B矩阵的空间局部性，保证每次能够尽可能得到B矩阵的下一个需要找到的元素

2.寄存器动态管理：及时释放不需要的寄存器，避免超过寄存器上限，能够分块分的更大

3.参数调整：尝试使用不同的分块参数，寻找miss较低解

我认为的最优秀的实现排序

1. case2
2. case1
3. case3

case1

	cache miss	register miss	latency
case1	604	4608	13668

代码大致如下：

```
#define BP 16

void gemm_case1(ptr_reg A, ptr_reg B, ptr_reg C, ptr_reg buffer) {
    for (reg i = 0; i < m; ++i) {                // 遍历A的每一行
        for (reg j = 0; j < p; j += BP) {        // 按列块遍历C和B
            // 初始化累加寄存器
            reg c[BP];
            for (reg k = 0; k < BP; ++k) {
                c[k + 0] = 0;
            }
            for (reg k = 0; k < n; ++k) {        // 遍历A的列和B的行
                reg a = A[i * n + k];            // 加载A[i][k]到寄存器

                for (reg h = 0; h < BP; ++h) {
                    reg* b = new reg;
                    *b = B[k * p + j + h];
                    c[h + 0] += a * *b;
                    delete b;
                }
            }

            // 将累加结果写回C矩阵
            for (reg k = 0; k < BP; ++k) {
                C[i * p + j + k] = c[k + 0];
            }
        }
    }
}
```

miss分析：case1采用的是单列16列一块的分块方法，同时B矩阵的刚好是16列。同时因为A B矩阵在地址上是连续的，因此B矩阵每一行加载到缓存行中的时候刚好对齐在同一个缓存行，每次取出B矩阵的一行的数据的时候都能够充分利用到空间局部性，也就是说可以做到只进行一次cache的miss。最外层的for循环对A的行进行循环，一共进行16次；循环内部对第i行不同列的A进行寄存器加载的时候，理论上A的空间也是连续的也就是说在循环for (reg k = 0; k < n; ++k)内部，理论上只需要加载一次A矩阵即可，然而B加载情况和A存在冲突，每次都需要重新把A放入缓存进行加载，大大增加了cachemiss数量。每次最后的写回C矩阵操作也存在一次cachemiss，计算理论上cache需要有16×33次miss。实际上的miss次数更多，应当是由于cache为直接映射方式，可能存在C矩阵和A，B矩阵冲突的情况，导致miss次数更多。

regmiss的次数是大循环内把c寄存器变成0需要16次miss，加载A矩阵需要16次miss，加载B矩阵需要16×16次miss，因此大循环内一共进行了288次miss，大循环一共16次，即进行4608次miss。

代码中BP是分块的列数参数，本题中设置为16，最大程度上利用空间局部性的方法就是把B的一行数据全部分块读入，能够最大化优化速率。我尝试将BP设为8时，优化速率大约是5.6倍。

case2

	cache miss	register miss	latency
case2	4704	35840	106400

改变的代码大致如下：

```
// 处理尾部列块
            reg c[BP2];
            for (reg k = 0; k < BP2; ++k) {
                c[k + 0] = 0;
            }
            for (reg k = 0; k < n; ++k) {          // 遍历A的列和B的行
                reg a = A[i * n + k];              // 加载A[i][k]到寄存器

                for (reg h = 0; h < BP2; ++h) {
                    reg* b = new reg;
                    *b = B[k * p + BP1 + h];
                    c[h + 0] += a * *b;
                    delete b;
                }
            }

            // 将累加结果写回C矩阵
            for (reg k = 0; k < BP2; ++k) {
                C[i * p + BP1 + k] = c[k + 0];
            }
        }
```

本题的代码思路和第一问几乎完全一致，实现结果也和第一问完全一致。这道题目由于reg数量有限，经过测试，列块的最大值是22列，无法一块包含所有的列，因此我将B矩阵分成两块，列长度分别为BP1和BP2，由于这道题和case1同样是B矩阵完全内存对齐的，因此显然在BP1=BP2=16的时候表现最好。

miss方面，大循环是32次，列块循环是2次，每个列块循环内按照case1的计算方法，大约是32×2+1=65次循环，计算下来cachemiss次数大约是65×2×32=4160次miss，考虑到直接映射结构的影响，和4704次miss几乎一致。

寄存器miss方面，大循环是32次，列块循环是2次，初始化C进行16次miss，遍历A的列进行32次miss，遍历B列块进行16*32次矩阵，计算方式为32×2×(16+32+16×32)=35840次，与预期完全符合。

改变部分的代码是为case3准备的，为了防止BP1和BP2不一样的情况，对B矩阵的剩余列进行处理，由于case2的BP1=BP2=16，因此即使复用case1的代码也没有影响。

case3

	cache miss	register miss	latency
case3	7254	38812	147622

本题代码除参数BP1和BP2与第二问不同以外，其余部分全部相同。BP1和BP2经过测试，在BP1=17，BP2=14的时候表现最好。这道题的主要问题是数据与第一问，第二问不同，无法进行对齐，即使访问一个列块中的B矩阵的一行也无法保证这一行数据全部都在同一个cacheline里面，因此只能利用部分的空间局部性来改善代码效果。

miss方面，这道题的cachemiss因为内存不对齐，与理论上的miss次数相差甚远，按理来说假设B矩阵每次遍历列块都能做到处在同一个缓存行内，按照前两题的估计方法，cachemiss应当在5千左右，与实际相差甚远。

寄存器方面，大循环进行了31次，循环内对C寄存器初始化进行一共31次，对A寄存器赋值进行2×37次，对B矩阵遍历进行31×37次，一共是31×（31+2×37+31×37）=38812次，与实际结果完全一致。

进行的尝试

1.尝试消除循环。我原以为每次循环的寄存器k进行++操作的时候，也会产生多余的regmiss，循环次数相当多就会产生非常大量的寄存器miss，然而实际上++操作没有产生任何miss，将循环拆解没有任何的优化效果。

2.尝试case3的动态分块：尝试每次遍历B矩阵的时候，按照B矩阵这一行地址的情况，将不在同一个缓存行的部分分开成为两部分，能够保证B矩阵的每一行能按列分成大约2块，每行的列块大小可能不固定。理论上来讲对case3一定存在提升，鉴于时间原因，且优化过于复杂没有深入完成实现。

反馈/收获/感悟/总结

参考的重要资料

参考了题目中给出的csapp官方分块教程