日志 2.7

一、 八分块优化 M64N64

首先采用 M32N32 的 8 分块优化方法，发现 miss4000+，感到惊讶，因此对其进行一个理论分析：64*64 的一行占 8 个 block，因此 4 行就占满了。这意味着按照 32*32 的 8 分块会导致写入数组 B 的 8 个行时，由于前 4 行和后 4 行的冲突导致总是不命中。接着改为 4 分块优化，miss 降低了一半是 2200 左右，没有达到要求。4 分块虽然避开了 8 分块的冲突。但是一个块有 8 个元素。因此需要一个块需要载入两次。也有缺陷。

经过思考，采用了一些策略，改进了 8 分块的优化方法，该方法充分地利用了块，而且减少了写入数组 B 时的冲突不命中。方法是将每个 8 分块分为 3 个部分转置

1. 第一部分是 A 数组上方的一半，将其按行进行转置，但是写入数组 B 的时候，右上角 4*4 的一块先放在 B 数组的右上角一块，这样没有写入数组 B 的后四行，没有造成冲突。
2. 第二部分是 A 数组左下角的 4*4 的一块，将其按列进行转置。因为我们有 8 个临时变量，可以同时存储 A 数组将要转置的一列 4 个元素，已经 B 数组占了位置需要转移的 4 个元素。由此减小冲突。
3. 第三部分是 A 数组右下角的 4*4 的一块。将其按列进行转置。这个就正常转置即可。不会冲突。

经过上述思想产生的代码如下：（每一个 for 代表一个转置部分）

```
for(int x = i; x < i+4; x++)
{
    tmp0 = A[x][j];     tmp1 = A[x][j+1];
    tmp2 = A[x][j+2];   tmp3 = A[x][j+3];
    tmp4 = A[x][j+4];   tmp5 = A[x][j+5];
    tmp6 = A[x][j+6];   tmp7 = A[x][j+7];
    B[j][x] = tmp0;     B[j+1][x] = tmp1;
    B[j+2][x] = tmp2;   B[j+3][x] = tmp3;
    B[j][x+4] = tmp4;   B[j+1][x+4] = tmp5;
    B[j+2][x+4] = tmp6; B[j+3][x+4] = tmp7;
}
for(int y = j; y < j+4; y++)
{
    tmp0 = A[i+4][y];   tmp1 = A[i+5][y];
    tmp2 = A[i+6][y];   tmp3 = A[i+7][y];
    tmp4 = B[y][i+4];   tmp5 = B[y][i+5];
    tmp6 = B[y][i+6];   tmp7 = B[y][i+7];
    B[y][i+4] = tmp0;   B[y][i+5] = tmp1;
    B[y][i+6] = tmp2;   B[y][i+7] = tmp3;
    B[y+4][i] = tmp4;   B[y+4][i+1] = tmp5;
    B[y+4][i+2] = tmp6; B[y+4][i+3] = tmp7;
}
```

```
for(y = j+4; y < j+8; y=y+2)
{
    tmp0 = A[i+4][y];   tmp1 = A[i+5][y];
    tmp2 = A[i+6][y];   tmp3 = A[i+7][y];
    tmp4 = A[i+4][y+1]; tmp5 = A[i+5][y+1];
    tmp6 = A[i+6][y+1]; tmp7 = A[i+7][y+1];
    B[y][i+4] = tmp0;   B[y][i+5] = tmp1;
    B[y][i+6] = tmp2;   B[y][i+7] = tmp3;
    B[y+1][i+4] = tmp4; B[y+1][i+5] = tmp5;
    B[y+1][i+6] = tmp6; B[y+1][i+7] = tmp7;
}
```

结果为 miss1163 个，达到了要求

二、 追踪 M64N64 组索引，分析 8 分块优化过程

修改 csim 代码使得其可以输出组索引和标记位,输入指令./csim -v -s 5 -E 1 -b 5 -t trace.f0 > f0.txt。得到文件后进行分析：

1. 第一部分转置时，A 的每一行读取第一个会冷不命中，其他的都命中，B 写入时，只有第一次冷不命中，其余都命中
2. 第二部分转置时，A 的列读第一次冷不命中，其他都命中，B 写入时，由于先

写入的地址还在块中，因此命中，开新地址时，第一次冷不命中，其余命中

3. 第三部分转置时，全部命中。

三、 编写代码优化 M61N67

M61N67 情况下是不均匀分块因此也不能很细致地对 cache 进行微操优化。但是条件放地比较宽泛。因此用分块方法对其依次测试，得到的表格如下：

4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
24	22	22	21	21	20	20	20	20	20	19	20	19	19	19	19	20
25	96	24	52	18	92	76	89	57	48	96	21	92	50	61	79	02

17 分块的代码如下：

```
for(i = 0; i < N; i=i+17)
{
    for(j = 0; j < M; j=j+17)
    {
        for(x = i;((x<N)&&(x<i+17)); x++)
        {
            for(y = j;((y<M)&&(y<j+17)); y++)
            {
                B[y][x] = A[x][y];
            }
        }
    }
}
```

17 分块的结果如下：

```
yihuahuahua@ubuntu:~/桌面/cash/cachelab-handout$ ./test-trans -M 61 -N 67

Function 0 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:6229, misses:1950, evictions:1918
```

实验报告 2.2

一、回顾日志 2.4~2.7，描述实验目标、实验资源、实验步骤（含关键命令行及说明）

- 实验目标

  了解 cache 对 C 语言程序产生的影响

  具体目标为两个模块：第一个模块是完成一个缓存模拟器，了解计算机中缓存的是如何命中，执行如何策略进行驱逐等操作；第二个模块是编写一个矩阵转置函数，针对 cache 性能进行优化，了解了如何编写出 cache 友好的程序

- 实验资源

  实验配套的实验指导书与 ppt 指引了整个实验

  模块 1 中提供了 csim-ref 示例文件，作为标准指引了缓存模拟器的制作。同时通过了老师的课程引导完成了函数的构建。

  模块 2 中提供了 test-trans 文件，评估了自己编写的程序是否正确，以及打印了 miss 数目。同时将 C 语言代码转换为缓存模拟器可以识别的文件，方便逐步分析。

- 实验步骤

  在实验开始之前，安装了一系列配套环境。除了 vim，gcc，gdb，python 外，按照要求安装了缓存跟踪程序 valgrind。该件已经被提供，解压后验证即可

  $$vakgrind --log-fd=1 --tool=lackey -v --trace-mem=yes ls -l$$

  环境配置完成后，开始完成 PartA，首先测试 csim-ref 模拟器：

  $$./csim-ref\ -h$$

  $$./csim-ref\ -s\ 4\ -E\ 1\ -b\ 4\ -t\ traces/yi.trace$$

  $$./csim-ref\ -v\ -s\ 4\ -E\ 1\ -b\ 4\ -t\ traces/yi.trace$$

  按照其格式以及课程引导上的函数声明构建函数。与 csim-ref 做比较。然后输入如下指令进行验证：

  $$./csim\ -h$$

./csim －s 4 －E 1 －b 4 －t traces/yi.trace

./csim－v －s 4 －E 1 －b 4 －t traces/yi.trace

自己验证后，输入如下指令进行系统测试：

./test－csim

PartA 完成后，来攻克 PartB

首先测试 test-trans,输入如下指令：

./test－trans －M 32－N 32

接着为了弄清原理，先分析 M4N4 情况。用 csim，并将结果输出到文件：

./test－trans －M 4－N 4

./csim－v －s 5 －E 1 －b 5 －t traces.f0 ＞ f0.txt

由此不断地分析改进，分别通过如下指令完三个规模的测试：

./test－trans －M 32－N 32

./test－trans －M 64－N 64

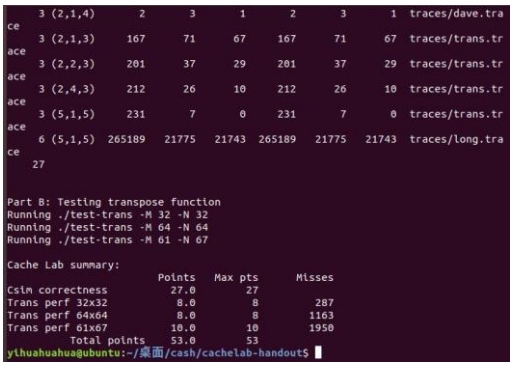
./test－trans －M 61－N 67

最终执行指令：

./driver.py

二、展示最终结果，并解释

最终结果如图所示：



ParA 部分：分别对应 6 个测试样例，经过我的程序运行出的命中，不命中，驱逐数分别与 6 个测试样例相同，我就得到了分数

PartB 部分：对于 32*32 优化到了 miss300 一下，得到了满分，对于 64*64 优化到了 1300 以下，得到了满分，对于 61*67 优化到了 2000 以下，最终满分通过。

三、实验总结

完成了此次试验之后，感慨良多。本次实验的挑战性在我看了远远大于了其他的实验，因此需要耗费更多的精力，但完成了本次实验也更是获益良多。

1. 知识收获

   经过本次实验的历练，较好地掌握了 cache 对于程序的影响，更是清楚了 preflab 中循环展开的优点所在。清楚明白了 cache 的知识。同时在 PartA 部分中编写函数，为了对于参数，一改之前习惯与使用全局变量的方法而选用了要求的传递指针的方法，丰富了编程技能。

2. 方法收获

   本次实验的挑战就是在给定时间完成所必须获取的信息，由此需要和同学进行讨论以及网上获取资料。感到了对于信息查找能力的提升。

3. 技能得到

   经过本次实验，掌握了一些面对 OJ 题目缩短时间的方法。