cachelab第二次实验日志

任务 1.运行 test-trans，以 M32N32 矩阵为例，通过 csim-ref 详细选项(-v)在缓存跟踪 trace.f 文件中观察结果

```
lh@lh-VirtualBox:~/cachelab-handout$ ./test-trans -M 32 -N 32

Function 0 (2 total)
Step 1: Validating and generating memory traces
Validation error at function 0! Run ./tracegen -M 32 -N 32 -F 0 for details.
Skipping performance evaluation for this function.

Function 1 (2 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 1 (Simple row-wise scan transpose): hits:870, misses:1183, evictions:1151

Summary for official submission (func 0): correctness=0 misses=2147483647

TEST_TRANS_RESULTS=0:2147483647
```

如上图所示运行指令./test-trans -M 32 -N 32 生成 trace.f1 文件。通过 csim-ref 在缓存文件trace.f1 中观察结果如下：

```
lh@lh-VirtualBox:~/cachelab-handout$ ./csim-ref -v -s 5 -E 1 -b 5 -t trace.f1
S 18c08c,1 miss
L 18c0c0,8 miss
L 18c084,4 hit
L 18c080,4 hit
L 10c080,4 miss eviction
S 14c080,4 miss eviction
L 10c084,4 miss eviction
L 10d074,4 hit
S 14cf7c,4 miss eviction
L 10d078,4 hit
S 14cffc,4 miss eviction
L 10d07c,4 hit
S 14d07c,4 miss eviction
S 18c08d,1 miss eviction
hits:870 misses:1183 evictions:1151
```

任务 2. 编写 csim.c 处理本条命令行同时输出显示标记位和组号

```c
while (fscanf(tracefile, "%s %x,%d", opt, &addr, &size) != EOF)
{
    if (strcmp(opt, "I") == 0)
        continue;
    int setBits = getSet(addr, s, b);
    int tagBits = getTag(addr, s, b);
    printf("set:%d ,tag:%d", setBits, tagBits);
    if (isVerbose == 1)
        printf("%s %x,%d", opt, addr, size);
    if (strcmp(opt, "S") == 0)
    {
        storeData(&cache, addr, size, setBits, tagBits, isVerbose);
    }
    if (strcmp(opt, "M") == 0)
    {
        modifyData(&cache, addr, size, setBits, tagBits, isVerbose);
    }
    if (strcmp(opt, "L") == 0)
    {
        loadData(&cache, addr, size, setBits, tagBits, isVerbose);
    }
    if (isVerbose == 1)
        printf("\n");
}
```

```
lh@lh-VirtualBox:~/cachelab-handout$ ./csi
set:4 ,tag:1584S 18c08c,1 miss
set:6 ,tag:1584L 18c0c0,8 miss
set:4 ,tag:1584L 18c084,4 hit
set:4 ,tag:1584L 18c080,4 hit
set:4 ,tag:1072L 10c080,4 miss eviction
set:4 ,tag:1328S 14c080,4 miss eviction
set:4 ,tag:1072L 10c084,4 miss eviction
set:8 ,tag:1328S 14c100,4 miss
```

任务 3.运行 test-trans 以 M4N4矩阵为例，并分析示例函数 miss 过多的原因

1、运行结果截图

```
lh@lh-VirtualBox:~/cachelab-handout$ ./test-trans -M 4 -N 4
set:4 ,tag:1072L 10c080,4 miss eviction
set:4 ,tag:1328S 14c080,4 miss eviction
set:4 ,tag:1072L 10c084,4 miss eviction
set:4 ,tag:1328S 14c090,4 miss eviction
set:4 ,tag:1072L 10c088,4 miss eviction
set:5 ,tag:1328S 14c0a0,4 miss
set:4 ,tag:1072L 10c08c,4 hit
hits:15 misses:22 evictions:19
```

2、分析 miss 过多原因

说明：输入的 cache 的参数有 32 个组，每个组有 1 行，每行包含一个32字节的块。但是，我们可以看到只使用了第4、5和6组，所以我们只需要分析第4和5组。显示的部分非常长，经过观察，它们实际上是一个循环，所以只需要分析其中一个循环。

【操作 L 对应数组 A 的操作,每次横向++;操作 S 对应数组B 操作,每次纵向++】

1)访问 A[0][0]，A[0][0]映射的块在cache 中，但tag 不同，所以miss 并 eviction，将块加载入cache

2)访问 B[0][0]，B[0][0]映射的块在 cache 中，但 tag 不同，所以miss 并eviction，将数组 B 对应的块存入 cache

3)访问 A[0][1]，A[0][1]映射的块在 cache 中，但 tag 不同，所以miss 并eviction，重新将数组 A 对应的块加载入 cache
4)访问 B[1][0]，B[1][0]映射的块在cache 中，但tag 不同，所以miss 并 eviction，重新将数组 B 对应的块存入 cache。
5)访问 A[0][2]，A[0][2]映射的块在cache 中，但tag 不同，所以miss 并 eviction，重新将数组 A 对应的块加载入 cache。
6)访问 B[2][0]，B[2][0]映射的块不在 cache 中，由于第一次使用，对应块不命中，所以 miss，将数组 B 对应的块存入 cache。
7)访问 A[0][3]，A[0][3]映射的块在cache 中，且标记位相同，故命中。

原因：

当A完成一组读取时，B有很高的概率写入同一组，但此时标记位不同，因此miss;B写入后，下一个A读取时，标记位不同，将导致再次miss。这样，循环将导致此逻辑中miss较多。为了减少miss,有必要解决冲突不命中问题,也就是说,当两个元素在同一块访问，因为中间访问了其它的块,加载块将被驱逐,导致第二次访问时不命中。这样就可以同时访问同一个块中的多个元素，之后就不再需要访问这个块，从而减少了冲突不命中的数量。

任务 4. 运行 test-trans 以 M32N32矩阵为例，并分析示例函 miss 过多的原因

```
lh@lh-VirtualBox:~/cachelab-handout$ ./csim -v -s 5 -E 1 -b 5 -t trace.f1>test.txt
set:4 ,tag:1072L 10c080,4 miss eviction
set:4 ,tag:1328S 14c080,4 miss eviction
set:4 ,tag:1072L 10c084,4 miss eviction
set:8 ,tag:1328S 14c100,4 miss
set:4 ,tag:1072L 10c088,4 hit
set:12 ,tag:1328S 14c180,4 miss
set:4 ,tag:1072L 10c08c,4 hit
set:16 ,tag:1328S 14c200,4 miss
set:4 ,tag:1072L 10c090,4 hit
set:20 ,tag:1328S 14c280,4 miss
set:4 ,tag:1072L 10c094,4 hit
set:24 ,tag:1328S 14c300,4 miss
set:4 ,tag:1072L 10c098,4 hit
set:28 ,tag:1328S 14c380,4 miss
set:4 ,tag:1072L 10c09c,4 hit
set:0 ,tag:1329S 14c400,4 miss
set:5 ,tag:1072L 10c0a0,4 miss
set:4 ,tag:1329S 14c480,4 miss eviction
set:5 ,tag:1072L 10c0a4,4 hit
set:8 ,tag:1329S 14c500,4 miss eviction
```

分析：

当数组 A 从第一个块从左往右读取数据时，数组 B 从第一个块从上至下写入数据，当 A 、 B 的 tag 不同时却访问了同一个块这就会导致 miss 和 eviction。A 继续右移访问块 1 就会 hit，而 B 每次向下访问新的块就会导致 miss（对应的组都不同）。直到 A 跳出第一个块到达 A[0][8]时到 set 5 会miss， 而B 到达B[8][0]与 B[9][0]的时候回到set 4 与 set 8，这时因为一开始存的是A，与 A tag 不同的B 只能 miss 并且eviction。

任务 5.分别按 4和8 分块编写 transpose_submit()代码，记录test-trans以M32N32矩阵下miss数目结果，进一步编写代码重新处理相同下标的对角线上元素来再次优化

1、4分块：

```c
char transpose_submit_desc[] = "Transpose submission";
void transpose_submit1(int M, int N, int A[N][M], int B[M][N])
{
    int i, j, k;
    for (i = 0; i < N; i += 4)
    {
        for (j = 0; j < M; j += 4)
        {
            for (k = 0; k < 4; k++)
            {
                B[j][i + k] = A[i + k][j];
                B[j + 1][i + k] = A[i + k][j + 1];
                B[j + 2][i + k] = A[i + k][j + 2];
                B[j + 3][i + k] = A[i + k][j + 3];
            }
        }
    }
}
```

```
Function 0 (5 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 0 (Transpose submission): hits:1566, misses:487,
```

2、8 分块：

```c
char transpose_submit_desc[] = "Transpose submission";
void transpose_submit2(int M, int N, int A[N][M], int B[M][N])
{
    int i, j, k;
    for (i = 0; i < N; i += 8)
    {
        for (j = 0; j < M; j += 8)
        {
            for (k = 0; k < 8; k++)
            {
                B[j][i + k] = A[i + k][j];
                B[j + 1][i + k] = A[i + k][j + 1];
                B[j + 2][i + k] = A[i + k][j + 2];
                B[j + 3][i + k] = A[i + k][j + 3];
                B[j + 4][i + k] = A[i + k][j + 4];
                B[j + 5][i + k] = A[i + k][j + 5];
                B[j + 6][i + k] = A[i + k][j + 6];
                B[j + 7][i + k] = A[i + k][j + 7];
            }
        }
    }
}
```

```
Function 1 (5 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 1 (Transpose submission): hits:1710, misses:343,
```

3、对角线优化：循环展开,通过一次获取块中的所有元素，然后不再访问该块来减少冲突不命中。设置变量每次存储一行A，避免加载B[i][i]后再加载A[i]行导致驱逐。

```c
char transpose_submit_desc[] = "Transpose submission";
void transpose_submit3(int M, int N, int A[N][M], int B[M][N])
{
    int i, j, k;
    int temp0, temp1, temp2, temp3;
    for (i = 0; i < N; i += 4)
    {
        for (j = 0; j < M; j += 4)
        {
            for (k = 0; k < 4; k++)
            {
                temp0 = A[i + k][j];
                temp1 = A[i + k][j + 1];
                temp2 = A[i + k][j + 2];
                temp3 = A[i + k][j + 3];
                B[j][i + k] = temp0;
                B[j + 1][i + k] = temp1;
                B[j + 2][i + k] = temp2;
                B[j + 3][i + k] = temp3;
            }
        }
    }
}
```

```
Function 2 (5 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 2 (Transpose submission): hits:1614, misses:439, evictions:407
```

```c
char transpose_submit_desc[] = "Transpose submission";
void transpose_submit4(int M, int N, int A[N][M], int B[M][N])
{
    int i, j, k;
    int temp0, temp1, temp2, temp3, temp4, temp5, temp6, temp7;
    for (i = 0; i < N; i += 8)
    {
        for (j = 0; j < M; j += 8)
        {
            for (k = 0; k < 8; k++)
            {
                temp0 = A[i + k][j];
                temp1 = A[i + k][j + 1];
                temp2 = A[i + k][j + 2];
                temp3 = A[i + k][j + 3];
                temp4 = A[i + k][j + 4];
                temp5 = A[i + k][j + 5];
                temp6 = A[i + k][j + 6];
                temp7 = A[i + k][j + 7];
                B[j][i + k] = temp0;
                B[j + 1][i + k] = temp1;
                B[j + 2][i + k] = temp2;
                B[j + 3][i + k] = temp3;
                B[j + 4][i + k] = temp4;
                B[j + 5][i + k] = temp5;
                B[j + 6][i + k] = temp6;
                B[j + 7][i + k] = temp7;
            }
        }
    }
}
```

```
Function 3 (5 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 3 (Transpose submission): hits:1766, misses:287, evictions:255
```

列表记录数组 B 第 1 个 8*8 块写操作的命中情况：

miss evictions	hit	hit	hit	hit	hit	hit	hit
miss	miss evictions	hit	hit	hit	hit	hit	hit
miss	hit	miss evictions	hit	hit	hit	hit	hit
miss	hit	hit	miss evictions	hit	hit	hit	hit
miss	hit	hit	hit	miss evictions	hit	hit	hit
miss	hit	hit	hit	hit	miss evictions	hit	hit
miss	hit	hit	hit	hit	hit	miss evictions	hit
miss	hit	hit	hit	hit	hit	hit	miss evictions

任务 6.分析采用分块技术后 miss 改善的原因，按4分块编写transpose_submit()代码，记录test_trans以M32N32矩阵下miss数目结果；以两个 4*4 为例分析 hit 增多的原因

（4 分块代码以及 miss 数截图入任务 5 的对角线优化）

1、miss 改善原因分析：

32x32 分块，8x8 一块，每一组块的组号都不同，如果一组一组转置的话不会 eviction。分块的目的主要是在对称的 8x8 块读取时不会有块相同的部分，使得读取 A 与存储 B 时不会有冲突 eviction miss， 避免了读一次写一次同一组都驱逐的 miss。对角线优化过程主要是：先设置 8 个变量存储 A 数组连续一个组的 8 个数字，因为在同一个组里，不会有miss 发生，然后读到 B 里，减少对角线块的同一组读写驱逐情况。如果当 i=j 的时候，按照第五部分修改，在 A 访问 i 之前 B 不访问还可以有少许优化，避免每一行的一次驱逐。

```
Function 2 (5 total)
Step 1: Validating and generating memory traces
Step 2: Evaluating performance (s=5, E=1, b=5)
func 2 (Transpose submission): hits:1614, misses:439, evictions:407
```

2、4x4 分析

访问 A[0][0]miss，将A[0][]一行加载到 cache 第 8 行，而写 B[0][0]访问 cache 第 8 行，此时会 miss，继续转置 A[0][1]A[0][3]由于已经将这些数据加载到临时变量对应寄存器中，所以直接hit。而对于B[1][0]B[3][0]由于第一次访问，所以 miss。对于A[0][4]由于 set8 已经被 B[0][]这一行占据所以发生驱逐，对于以后的 A[0][5]A[0][7]则会连续命中。B[5][0]B[7][0] 由于第一次访问，所以 miss。以此类推，可以得出：A 矩阵的第一列将会不命中，其余部分全命中 ；B 矩阵的第一列和每个 44 规模的对角线将会不命中，其余部分全部命中。

hit发现比未优化时增多了，但在对角线上仍然有miss。这是因为在对角线优化之前，B每次读取与A相同的下标时，都会驱逐 miss。hit增加的原因是在B访问了第一列之后，所有剩余的列都已经在cache中，并且都将命中。而且，分块后，A和B的组号是不同的，所以不存在冲突eviction miss。