实验环境

服务器：腾讯云 蜂驰型BF1 | BF1.MEDIUM4 2核心 4GB（7天试用版）

操作系统：Ubuntu Server 18.04 LTS 64位

GCC版本：gcc (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0

找到的三个问题

1.程序运行时占用内存过大，导致被kill

```
root@VM-0-16-ubuntu:~/workshop# dmesg | grep -i workshop
[ 1468.832665] workshop invoked oom-killer: gfp_mask=0x14000c0(GFP_KERNEL),
nodemask=(null), order=0, oom_score_adj=0
[ 1468.834944] workshop cpuset=/ mems_allowed=0
[ 1468.834949] CPU: 1 PID: 18736 Comm: workshop Tainted: G           OE
 4.15.0-213-generic #224-Ubuntu
[ 1468.869028] [16018]     0 16018   112828    65818   679936        0
  0 workshop
[ 1468.869029] Memory cgroup out of memory: Kill process 16018 (workshop) score
976 or sacrifice child
[ 1468.871060] Killed process 16018 (workshop) total-vm:451312kB, anon-
rss:261364kB, file-rss:1908kB, shmem-rss:0kB
[ 1468.880892] oom_reaper: reaped process 16018 (workshop), now anon-rss:0kB,
file-rss:0kB, shmem-rss:0kB
```

2.vector_append在多线程下使用但并不是线程安全的

```
void vector_append(Vector *vec, int value)
{
    if (vec->size == vec->capacity) {
        vec->capacity *= 2;
        vec->data = realloc(vec->data, vec->capacity * sizeof(int));
    }
    vec->data[vec->size++] = value;
}
```

上面函数中的vec->size++、vec->capacity、vec->data都可能因为多线程的竞争出现同步问题。

3.mul.c中mul函数行优先导致CPU缓存命中率过低

```
void mul(int msize, int tidx, int numt, Vector *vec, TYPE a[][NUM], TYPE b[]
[NUM], TYPE c[][NUM], TYPE t[][NUM])
{
    int i,j,k;

    for (i = tidx; i < msize; i = i + numt) {
        for (j = 0; j < msize; j++) {
            for (k = 0; k < msize; k++) {
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
            }
            vector_append(vec, c[i][j]);
        }
    }
}
```

优化手段

1.程序运行时占用内存过大，导致被kill

由于vec在本程序中并没用到，所以直接注释，避免其运行时导致程序过大被kill

2.vector_append在多线程下使用但并不是线程安全的

可使用锁等方式解决该问题，但这不是本测试优化的重点，因此尚未实现

3.mul.c中mul函数行优先导致CPU缓存命中率过低

分块计算

常规分块运算提升了CPU缓存的空间与时间局部性，有效避免了CPU大量从内存中置换内容到缓存中来

```
void mul(int msize, int tidx, int numt, Vector *vec, TYPE a[][NUM], TYPE b[]
[NUM], TYPE c[][NUM], TYPE t[][NUM])
{
    for (int bi = tidx * BLOCK_SIZE; bi < msize; bi += numt * BLOCK_SIZE) {
        for (int bk = 0; bk < msize; bk += BLOCK_SIZE) {
            for (int bj = 0; bj < msize; bj += BLOCK_SIZE) {
                int i_end = min(bi + BLOCK_SIZE, msize);
                int k_end = min(bk + BLOCK_SIZE, msize);
                int j_end = min(bj + BLOCK_SIZE, msize);
                for (int i = bi; i < i_end; i++) {
                    for (int k = bk; k < k_end; k++) {
                        TYPE a_ik = a[i][k];
                        #pragma omp simd
                        for (int j = bj; j < j_end; j++) {
                            c[i][j] += a_ik * b[k][j];
                        }
                    }
                }
            }
        }
    }
}
```

稀疏矩阵

但事实上上述方法还可通过判断是否为稀疏矩阵，然后再转换为Compressed Sparse Row格式进行存储。格式如下：

```
typedef struct {
    int *row_ptr;       //记录每一行非零元素的起始位置
    int *col_idx;       //记录每个非零元素所在的列索引
    TYPE *values;       //记录每个非零元素的值
    int num_nonzeros;   //记录非零元素的个数
} SparseMatrixCSR;
```

假设两个矩阵A、B：

$$A = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 3 & 0 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 6 & 0 \\ 7 & 0 & 8 \\ 0 & 9 & 0 \end{bmatrix} \quad C = AB$$

则存储该稀疏矩阵的数据结构为：

```
row_ptr_A = [0, 2, 3]
values_A  = [1, 2, 3]
col_idx_A = [0, 2, 1]

row_ptr_B = [0, 1, 3, 4]
values_B  = [6, 7, 8, 9]
col_idx_B = [1, 0, 2, 1]
```

则矩阵C[0,0]通过查找A一行的元素，B的第一列元素来计算值。具体过程如下：

- 通过A的row_ptr[0:1]得知其第一行元素的值在values[0:2-1]内，即1、2
- 通过row_ptr_A[0]获取A元素的行过查找col_idx_B该行相同的值，并获取对应values
- 使对应的values相乘

```
void csrMultiply(int msize, int tidx, int numt, Vector *vec, SparseMatrixCSR *A,
SparseMatrixCSR *B, TYPE C[][NUM], TYPE temp[][NUM]){
        for (int block_i = tidx * BLOCK_SIZE; block_i < msize; block_i += numt *
BLOCK_SIZE) {
        int block_end = (block_i + BLOCK_SIZE < msize) ? block_i + BLOCK_SIZE :
msize;
        for (int i = block_i; i < block_end; i++) {
            TYPE local_temp[NUM] = {0};
            for (int j = A->row_ptr[i]; j < A->row_ptr[i + 1]; j++) {
                int a_col = A->col_idx[j];
                TYPE a_val = A->values[j];
                for (int k = B->row_ptr[a_col]; k < B->row_ptr[a_col + 1]; k++)
{
                    int b_col = B->col_idx[k];
                    TYPE b_val = B->values[k];
                    local_temp[b_col] += a_val * b_val;
                }
            }
```

```
                for (int col = 0; col < NUM; col++) {
                    c[i][col] += local_temp[col];
                }
            }
        }
    }
```

测试结果

原始版本测试结果

```
Matrix multiply iteration 1: cost 55.876 seconds
Matrix multiply iteration 2: cost 57.314 seconds
Matrix multiply iteration 3: cost 50.579 seconds
Matrix multiply iteration 4: cost 57.729 seconds
Matrix multiply iteration 5: cost 58.558 seconds
Matrix multiply iteration 6: cost 57.841 seconds
Matrix multiply iteration 7: cost 58.263 seconds
Matrix multiply iteration 8: cost 58.097 seconds
Matrix multiply iteration 9: cost 58.543 seconds
Matrix multiply iteration 10: cost 58.377 seconds
```

优化版本测试结果

稀疏因子：测试中用于控制生成矩阵稀疏程度的参数，0.2则说明2048 * 2048个矩阵元素中仅存在2048 * 2048 * 0.2 个非零元素。

我在main函数中生成了稀疏因子为0.2的矩阵，然后分别传入分块计算方法以及使用CSR结构分块计算的方法中，比较它们的运行时间，发现其提高了将近10倍，而这又相比最初的版本提高了近150倍。

```
Sparse Blocked Matrix Multiply iteration 1: cost 0.343 seconds
Blockd Matrix Multiply iteration 1: cost 5.582 seconds
Sparse Blocked Matrix Multiply iteration 2: cost 0.341 seconds
Blockd Matrix Multiply iteration 2: cost 5.569 seconds
Sparse Blocked Matrix Multiply iteration 3: cost 0.339 seconds
Blockd Matrix Multiply iteration 3: cost 5.568 seconds
Sparse Blocked Matrix Multiply iteration 4: cost 0.342 seconds
Blockd Matrix Multiply iteration 4: cost 5.591 seconds
Sparse Blocked Matrix Multiply iteration 5: cost 0.347 seconds
Blockd Matrix Multiply iteration 5: cost 5.596 seconds
Sparse Blocked Matrix Multiply iteration 6: cost 0.348 seconds
Blockd Matrix Multiply iteration 6: cost 5.571 seconds
Sparse Blocked Matrix Multiply iteration 7: cost 0.344 seconds
Blockd Matrix Multiply iteration 7: cost 5.597 seconds
Sparse Blocked Matrix Multiply iteration 8: cost 0.349 seconds
Blockd Matrix Multiply iteration 8: cost 5.591 seconds
Sparse Blocked Matrix Multiply iteration 9: cost 0.341 seconds
Blockd Matrix Multiply iteration 9: cost 5.608 seconds
Sparse Blocked Matrix Multiply iteration 10: cost 0.340 seconds
Blockd Matrix Multiply iteration 10: cost 5.631 seconds
Sparse Blocked Matrix Multiply iteration 11: cost 0.339 seconds
```

```
Blockd Matrix Multiply iteration 11: cost 5.610 seconds
Sparse Blocked Matrix Multiply iteration 12: cost 0.343 seconds
Blockd Matrix Multiply iteration 12: cost 5.601 seconds
Sparse Blocked Matrix Multiply iteration 13: cost 0.339 seconds
Blockd Matrix Multiply iteration 13: cost 5.604 seconds
Sparse Blocked Matrix Multiply iteration 14: cost 0.343 seconds
Blockd Matrix Multiply iteration 14: cost 5.605 seconds
Sparse Blocked Matrix Multiply iteration 15: cost 0.343 seconds
Blockd Matrix Multiply iteration 15: cost 5.618 seconds
```

稀疏因子越小，则csrMultiply的运行效率相比传统的分块计算越高；反之，稀疏因子经过测试，发现即使稀疏因子为1.0，使用CSR方式进行分块计算的速度也仅仅比传统分块计算慢0.1~0.2秒左右

```
Sparse Blocked Matrix Multiply iteration 1: cost 5.762 seconds
Blockd Matrix Multiply iteration 1: cost 5.516 seconds
Sparse Blocked Matrix Multiply iteration 2: cost 5.760 seconds
Blockd Matrix Multiply iteration 2: cost 5.569 seconds
Sparse Blocked Matrix Multiply iteration 3: cost 5.733 seconds
Blockd Matrix Multiply iteration 3: cost 5.599 seconds
Sparse Blocked Matrix Multiply iteration 4: cost 5.692 seconds
Blockd Matrix Multiply iteration 4: cost 5.592 seconds
Sparse Blocked Matrix Multiply iteration 5: cost 5.674 seconds
Blockd Matrix Multiply iteration 5: cost 5.546 seconds
Sparse Blocked Matrix Multiply iteration 6: cost 5.692 seconds
Blockd Matrix Multiply iteration 6: cost 5.590 seconds
Sparse Blocked Matrix Multiply iteration 7: cost 5.709 seconds
Blockd Matrix Multiply iteration 7: cost 5.587 seconds
Sparse Blocked Matrix Multiply iteration 8: cost 5.729 seconds
Blockd Matrix Multiply iteration 8: cost 5.601 seconds
Sparse Blocked Matrix Multiply iteration 9: cost 5.731 seconds
Blockd Matrix Multiply iteration 9: cost 5.519 seconds
Sparse Blocked Matrix Multiply iteration 10: cost 5.721 seconds
Blockd Matrix Multiply iteration 10: cost 5.594 seconds
Sparse Blocked Matrix Multiply iteration 11: cost 5.686 seconds
Blockd Matrix Multiply iteration 11: cost 5.617 seconds
Sparse Blocked Matrix Multiply iteration 12: cost 5.706 seconds
Blockd Matrix Multiply iteration 12: cost 5.568 seconds
Sparse Blocked Matrix Multiply iteration 13: cost 5.704 seconds
Blockd Matrix Multiply iteration 13: cost 5.601 seconds
Sparse Blocked Matrix Multiply iteration 14: cost 5.725 seconds
Blockd Matrix Multiply iteration 14: cost 5.579 seconds
Sparse Blocked Matrix Multiply iteration 15: cost 5.682 seconds
Blockd Matrix Multiply iteration 15: cost 5.583 seconds
```

同时为了验证算法的正确性，本文打印了原始方法、分块乘法、稀疏分块乘法在c[0] [0]、c[1024] [1024]、c[2023] [2023]处的结果：

```
Matrix multiply iteration 1: cost 55.126 seconds
11428096000.000000, 720037888.000000 , 2690289664.000000
Matrix multiply iteration 2: cost 52.001 seconds
11428096000.000000, 720037888.000000 , 2690289664.000000
Matrix multiply iteration 3: cost 56.654 seconds
11428096000.000000, 720037888.000000 , 2690289664.000000
Matrix multiply iteration 4: cost 56.590 seconds
11428096000.000000, 720037888.000000 , 2690289664.000000
Matrix multiply iteration 5: cost 51.664 seconds
```

```
11428096000.000000, 720037888.000000 , 2690289664.000000

稀疏因子：1.0（与原始文件数据相同）
Sparse Blocked Matrix Multiply iteration 1: cost 5.690 seconds
11428096000.000000, 720037888.000000, 2690289664.000000
Blockd Matrix Multiply iteration 1: cost 5.568 seconds
11428096000.000000, 720037888.000000, 2690289664.000000
Sparse Blocked Matrix Multiply iteration 2: cost 5.690 seconds
11428096000.000000, 720037888.000000, 2690289664.000000
Blockd Matrix Multiply iteration 2: cost 5.596 seconds
11428096000.000000, 720037888.000000, 2690289664.000000
Sparse Blocked Matrix Multiply iteration 3: cost 5.705 seconds
11428096000.000000, 720037888.000000, 2690289664.000000
Blockd Matrix Multiply iteration 3: cost 5.615 seconds
11428096000.000000, 720037888.000000, 2690289664.000000
Sparse Blocked Matrix Multiply iteration 4: cost 5.783 seconds
11428096000.000000, 720037888.000000, 2690289664.000000
Blockd Matrix Multiply iteration 4: cost 5.625 seconds
11428096000.000000, 720037888.000000, 2690289664.000000
Sparse Blocked Matrix Multiply iteration 5: cost 5.763 seconds
11428096000.000000, 720037888.000000, 2690289664.000000
Blockd Matrix Multiply iteration 5: cost 5.611 seconds
11428096000.000000, 720037888.000000, 2690289664.000000


稀疏因子：0.2
Sparse Blocked Matrix Multiply iteration 1: cost 0.343 seconds
438899049.000000, 48968717.000000, 128641168.000000
Blockd Matrix Multiply iteration 1: cost 5.556 seconds
438899049.000000, 48968717.000000, 128641168.000000
Sparse Blocked Matrix Multiply iteration 2: cost 0.342 seconds
401976079.000000, 38033782.000000, 79979092.000000
Blockd Matrix Multiply iteration 2: cost 5.606 seconds
401976079.000000, 38033782.000000, 79979092.000000
Sparse Blocked Matrix Multiply iteration 3: cost 0.346 seconds
514838853.000000, 15311690.000000, 103171024.000000
Blockd Matrix Multiply iteration 3: cost 5.605 seconds
514838853.000000, 15311690.000000, 103171024.000000
Sparse Blocked Matrix Multiply iteration 4: cost 0.346 seconds
421532284.000000, 12859121.000000, 111613240.000000
Blockd Matrix Multiply iteration 4: cost 5.623 seconds
421532284.000000, 12859121.000000, 111613240.000000
Sparse Blocked Matrix Multiply iteration 5: cost 0.343 seconds
543661075.000000, 17761571.000000, 66624320.000000
Blockd Matrix Multiply iteration 5: cost 5.630 seconds
543661075.000000, 17761571.000000, 66624320.000000
```