HPC Homework3:Report

2023316026 张凯

Tsinghua University

zhang-k23@mails.tsinghua.edu.cn

1. 稀疏矩阵乘法串行实现

首先，为了熟悉稀疏矩阵乘法的数据结构、输入输出以及算法流程，我实现了稀疏矩阵乘法的串行算法。

串行算法实现较为简单，核心包括对 C 矩阵非零元素数目的估计，以及三层 for 循环进行矩阵乘法，同时存储为 CSR 格式等。通过串行实现，为后面并行版本的矩阵非零数目估计提供了基础，也提供了 CSR 格式存储的基本方法。

(1) 预处理代码

```
void preprocess(dist_matrix_t *matA, dist_matrix_t *matB) {
    info_ptr_t p = (info_ptr_t)malloc(sizeof(additional_info_t));
    // Estimate nnz of C
    int est_c_nnz = 0;
    std::vector<int> mask(matA->global_m, -1);
    for (int i = 0; i < matA->global_m; ++i) {
        int row_nnz = 0;
        int row_begin = matA->r_pos[i];
        int row_end = matA->r_pos[i + 1];
        for (int j = row_begin; j < row_end; ++j) {
            int col_idx = matA->c_idx[j];
            for (int k = matB->r_pos[col_idx]; k < matB->r_pos[col_idx + 1]; ++k) {
                int col_idx_b = matB->c_idx[k];
                if (mask[col_idx_b] != i)
                {
                    mask[col_idx_b] = i;
                    row_nnz ++;
                }
            }
        }
        est_c_nnz += row_nnz;
    }
    p->est_c_nnz = est_c_nnz;
    matA->additional_info = p;
    printf("estimate c nnz: %d \n", est_c_nnz);
}
```

(2) 矩阵乘法代码

```cpp
// C = A * B using CSR format
for (int i = 0; i < matA->global_m; ++i) {
    int row_start_A = matA->r_pos[i];
    int row_end_A = matA->r_pos[i + 1];
    // A[i][j]
    for (int row_A = row_start_A; row_A < row_end_A; ++row_A) {
        int col_A = matA->c_idx[row_A];
        float val_A = matA->values[row_A];

        int row_start_B = matB->r_pos[col_A];
        int row_end_B = matB->r_pos[col_A + 1];
        // C[i][:] += A[i][j] * B[j][:]
        for (int row_B = row_start_B; row_B < row_end_B; ++row_B) {
            int col_B = matB->c_idx[row_B];
            c_row[col_B] += val_A * matB->values[row_B];
        }
    }
    for (index_t idx = 0; idx < matC->global_m; idx++) {
        if(c_row[idx] != 0.0f) {
            matC->c_idx[c_nnz] = idx;
            matC->values[c_nnz] = c_row[idx];
            c_nnz ++;
            c_row[idx] = 0;
        }
    }
    matC->r_pos[i + 1] = c_nnz;
}
```

代码保存为 spgemm-optimized-v0.cu，对于数据集文件中的 7 个矩阵，测评结果如下：

表 1.1 spgemm-optimized-v0 与 spgemm-cusparse 性能对比

左侧：preprocess 右侧：compute time

数据集	spgemm-optimized-v0 (s)		spgemm-cusparse (s)		端到端加速比
1138_bus.csr	0.000114	0.001196	0.173365	0.000379	132.6290076
utm5940.csr	0.001000	0.029568	0.174913	0.001656	5.776269301
fe_body.csr	0.007878	1.586779	0.178530	0.003620	0.114225191
ct2010.csr	0.008217	3.555460	0.169990	0.005231	0.049168597
ncvxbqp1.csr	0.002404	1.935281	0.178573	0.002610	0.093504878

lhr17.csr	0.005433	0.250083	0.174664	0.008589	0.717187965
nemeth07.csr	0.017484	0.082584	0.171620	0.007845	1.793430467

可以看到，在某些数据上，串行结果居然能够比 CUSPARSE 更快，这主要是我们的预处理比较简单的缘故。接下来我们做并行优化。

2. 稀疏矩阵乘法并行实现

2.1 稀疏矩阵行并行

A 矩阵按行并行，暂时不考虑 B 的并行，写核函数进行实现，最本质的问题其实是存回 CSR 格式，如果直接按方阵存回再转格式，那么内存会超过 GPU 限制，核函数无法正确运行，因此只能通过 Hash 的方式进行存储。

核心的矩阵乘法没有太大的变化，关键代码是 Hash 存储和格式对齐，我是通过估计每行的非零元素个数，分配 Hash 表实现的，Hash 表用简单的线性探测法进行实现，同时实现了单独的并行对齐核函数，比较巧妙。

(1) Hash 表实现

```
// Calculate hash value
int hash = col_B % tableSize;
// Traverse the nodes in a hash table, using linear probing to handle collisions
int index = hash;
bool found = false;
int count = 0;
while (!found && count < tableSize) {
    // Use atomic operations to read information from hash table nodes.
    int stored_col_idx = Row_HashTable[index].col_idx;
    if (stored_col_idx == -1 || stored_col_idx == col_B) {
        found = true;
    } else {
        index = (index + 1) % tableSize;
    }
    count ++;
}
// If a matching col_idx is found, update the value.
if (Row_HashTable[index].col_idx == col_B) {
    Row_HashTable[index].val += val;
}
// If no matching col_idx is found, insert a new node.
else {
    Row_HashTable[index].col_idx = col_B;
    Row_HashTable[index].val = val;
}
```

(2) 并行对齐核函数

```cpp
// Kernel to count non-zero elements in each row
__global__ void ToRealCSR (const index_t* d_r_pos_C, const index_t* d_c_idx_C, const data_t* d_val_C,
                           index_t* m_r_pos_C, index_t* m_c_idx_C, data_t* m_val_C,
                           const index_t* d_est_c_nnz_rows_begin, const int global_m) {
    int row_id = blockIdx.x * blockDim.x + threadIdx.x;
    if (row_id >= global_m) return;
    if (row_id == 0) {
        int c_nnz = 0;
        m_r_pos_C[0] = 0;
        for (int i=0; i<global_m; ++i) {
            c_nnz += d_r_pos_C[i];
            m_r_pos_C[i + 1] = c_nnz;
        }
    }
    __syncthreads();
    int idx_begin_origin = d_est_c_nnz_rows_begin[row_id];
    int idx_begin_new = m_r_pos_C[row_id];
    for (int j=0; j<d_r_pos_C[row_id]; ++j) {
        m_c_idx_C[idx_begin_new + j] = d_c_idx_C[idx_begin_origin + j];
        m_val_C[idx_begin_new + j] = d_val_C[idx_begin_origin + j];
    }
}
```

代码保存为 spgemm-optimized-v1.cu，对于数据集文件中的 7 个矩阵，测评结果如下：

表 2.1 spgemm-optimized-v1 与 spgemm-cusparse 性能对比

左侧：preprocess 右侧：compute time

数据集	spgemm-optimized-v1 (s)		spgemm-cusparse (s)		端到端加速比
1138_bus.csr	0.00012	0.000329	0.173365	0.000379	386.9576837
utm5940.csr	0.000986	0.004282	0.174913	0.001656	33.51727411
fe_body.csr	0.010222	0.009659	0.17853	0.00362	9.162013983
ct2010.csr	0.0127	0.011675	0.16999	0.005231	7.188553846
ncvxbqp1.csr	0.002995	0.009786	0.178573	0.00261	14.17596432
lhr17.csr	0.006127	0.055936	0.174664	0.008589	2.952693231
nemeth07.csr	0.012094	0.009621	0.17162	0.007845	8.264563666

矩阵行并行之后，算法实现了对于 CUSPARSE 极高的加速比，这主要是我在预处理阶段没有识别特殊的模式，也没有做复杂的格式转换，因此大大降低了预处理的时间。

但是，算法只对 A 的行进行了并行，B 的列并没有并行，似乎会降低性能，因此进一步尝试并行。

2.2 稀疏矩阵行列并行

在 B 的列维度上通过 y 方向线程进行并行，预处理阶段即分配线程。核心代码为 y 线程分配和 Hash 的原子操作，保证各列同时读写时不会冲突。

(1) 预处理线程分配

```
dim3 block,grid;
block.x = 1;
block.y = WARP_SIZE;
grid.x = matA->global_m;
grid.y = ceiling(matA->global_m,WARP_SIZE);
p->block = block;
p->grid = grid;
printf("Threads block cutting succeeded ! \n");
```

(2) 乘法核函数 y 线程分配及共享内存

```
index_t row_id = blockIdx.x;
index_t col_id_idx = blockIdx.y * WARP_SIZE + threadIdx.y;
if (row_id >= global_m) return;
// C = A * B using CSR format
int ptr_a_begin = d_r_pos_A[row_id];
int ptr_a_end = d_r_pos_A[row_id + 1];
// Shared data of row A
__shared__ int sm_idx_A[WARP_SIZE];
__shared__ float sm_val_A[WARP_SIZE];

int tableSize = d_est_c_nnz_rows_end[row_id] - d_est_c_nnz_rows_begin[row_id];
int hash_begin = d_est_c_nnz_rows_begin[row_id];
HashNode* Row_HashTable = d_HashTable + hash_begin;
// A[i][j]
for (int ptr_a = ptr_a_begin; ptr_a < ptr_a_end; ptr_a += WARP_SIZE) {
    int thr_ptr_a = ptr_a + threadIdx.y;
    if (thr_ptr_a < ptr_a_end) {
        sm_idx_A[threadIdx.y] = d_c_idx_A[thr_ptr_a];
        sm_val_A[threadIdx.y] = d_val_A[thr_ptr_a];
    }
    __syncthreads();
    int sm_end = min(WARP_SIZE, ptr_a_end - ptr_a);
    // C[i][:] += A[i][j] * B[j][:]
    for (int i = 0; i < sm_end; ++i) {
        index_t col_A = sm_idx_A[i];
        index_t ptr_b_begin = d_r_pos_B[col_A];
        index_t ptr_b_end = d_r_pos_B[col_A + 1];
        index_t nnz_b_row = ptr_b_end - ptr_b_begin;
        if (col_id_idx < nnz_b_row) {
            index_t col_B = d_c_idx_B[ptr_b_begin + col_id_idx];
            data_t val = sm_val_A[i] * d_val_B[ptr_b_begin + col_id_idx];
```

(3) Hash 原子操作

```
// Calculate hash value
int hash = col_B % tableSize;
// Traverse the nodes in a hash table, using linear probing to handle collisions
int index = hash;
bool found = false;
while (!found) {
    // Use atomic operations to read information from hash table nodes.
    int stored_col_idx = atomicAdd(&(Row_HashTable[index].col_idx), 0);
    if (stored_col_idx == -1) {
        // If a matching col_idx is found, update the value.
        int old_val = atomicCAS(&(Row_HashTable[index].col_idx), -1, col_B);
        if (old_val == -1) {
            // Successfully claimed the slot, now write the value
            atomicExch(&(Row_HashTable[index].val), val);
            found = true;
        }
    } else if (stored_col_idx == col_B) {
        // Found matching col_idx, update the value
        atomicAdd(&(Row_HashTable[index].val), val);
        found = true;
    }
    // Move to the next slot using linear probing
    index = (index + 1) % tableSize;
}
```

这里的原子操作非常容易出错，也是我 debug 最久的环节。

代码保存为 spgemm-optimized-v2.cu，对于数据集文件中的 7 个矩阵，测评结果如下：

表 2.2 spgemm-optimized-v2 与 spgemm-cusparse 性能对比

左侧：preprocess 右侧：compute time

数据集	spgemm-optimized-v2 (s)		spgemm-cusparse (s)		端到端加速比
1138_bus.csr	0.000127	0.000395	0.173365	0.000379	332.8429119
utm5940.csr	0.000957	0.011259	0.174913	0.001656	14.4539129
fe_body.csr	0.009891	0.393389	0.178530	0.003620	0.451671295
ct2010.csr	0.012170	0.768058	0.169990	0.005231	0.224576662
ncvxbqp1.csr	0.002782	0.457854	0.178573	0.002610	0.393332262
lhr17.csr	0.005945	0.214153	0.174664	0.008589	0.832597298
nemeth07.csr	0.011730	0.058561	0.171620	0.007845	2.553171814

可见，列并行之后，结果反而变差了，这主要是由于避免 Hash 冲突时的原

子操作浪费了很多时间导致的。因此，在矩阵稀疏行非零元素非常少的情况下，只使用行并行是有效的。但是，如果 B 矩阵行非零元素很密集，那么使用 spgemm-optimized-v2 版本的行列同时并行是有效的。

最终，我们此次作业版本采取 spgemm-optimized-v1，保存为 spgemm-optimized.cu，从而在给定的 7 个数据集情况下取得最好的效果。

3. 总结

总之，在提供的矩阵数据集情况及 C = A * A 情况下，采用算法 2.1 是有效的，能够超越 CUSPARSE 的性能；对于 C = A * B 且矩阵 B 的行非零元素较为密集的情况下，采用算法 2.2 是有效的，应该经过并行度调优之后也能够超越 CUSPARSE 的性能。但是由于提供的数据集有限，且 2.1 的效果普遍优于 2.2 的效果，因此这里就不再集成 2.1 和 2.2 算法写总的算法了，如果需要测试 benchmark，需要按照需求根据实际情况进行测试取用。

此外，我实现的效果最好的算法没有考虑负载均衡，这是因为在提供的数据集上稀疏矩阵行非零元素非常少，如果考虑负责均衡，最简单的方式是根据估计的非零元素数目划分 Task，确定每个线程块负责的 ptr_begin 和 ptr_end，然后行列并行，但是从 2.2 的实验结果来看，在已有的数据集上，加入了列并行必须要考虑原子操作，就会大大降低程序的性能，因此可以预计在大部分的稀疏矩阵情况下，即使是实现了负载均衡，效果也未必有 2.1 中直接的行并行效果更好。限于时间，这里没有进行实现，在后续的多 GPU 版本中，如果效果理想，我将进行相应的实现。

注：spgemm-optimized-v0.cu，spgemm-optimized-v1.cu，spgemm-optimized-v2.cu 版本均能够通过新版的 utils.cu 的正确性测试。

参考文献

[1] Yuyao Niu, Zhengyang Lu, Haonan Ji, Shuhui Song, Zhou Jin, and Weifeng Liu. Tilespgemm: a tiled algorithm for parallel sparse general matrix-matrix multiplication on gpus. In Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Program- ming, pages 90–106, 2022.

[2] Weifeng Liu and Brian Vinter. An efficient gpu general sparse matrix- matrix multiplication for irregular data. In 2014 IEEE 28th international parallel and distributed processing symposium, pages 370–381. IEEE, 2014.