实验三：稀疏矩阵-矩阵乘（SpMM）

计04 何秉翔 2020010944

1. 实现方法

1.1 CPU

我们尝试了以下几种优化方法：

1. 改变内外循环顺序，具体而言，对内层循环 `j` 和 `k` 进行交换，以使得对于输出矩阵 `vout` 的写入可以在最内层循环之外进行写入，而不需要在最内层循环内部每一次都访问 `vout`，因此可以先对一个寄存器进行累加，最后在最内层循环之外对 `vout` 进行写入。
2. 使用 `openmp` 对最外层循环进行并行，经过测试，调整线程数为 28 时效果最优。
3. 使用 `openmp` 对最外层循环并行时，测试不同调度策略对性能的影响，经过测试，发现对于 `arxiv` 数据集使用 `guided` 最优，其余两个数据集使用 `dynamic` 最优，这两种策略对于稀疏矩阵负载不均衡的问题都有不少的缓解。
4. 对第二层循环尝试不同的循环展开维度，经过测试，对于 `k = 256` 时的 `arxiv`，较大的循环展开维度性能较优，但对于另外两个数据集并不是。
5. 使用 `#pragma omp simd` 指令进行 `simd` 操作，有一定的优化效果。

1.2 GPU

我们尝试了以下几种优化方法：

1. 消除 `warp divergence`：注意到一个 `warp` 的执行时间由最慢的线程决定，因此我们考虑让一个 `warp` 内部的线程工作量基本一致。为此我们让 **`warp` 内的每个线程都负责计算相同个数的 `C` 元素**，并且**这些 `C` 中的元素处于同一行**。

   此时可以发现，对于一个 `C` 元素的计算，涉及到稀疏矩阵 `A` 的一行与稠密矩阵 `B` 的一列相乘相加，假设这一行的非零元有 nnz_r 个，则一个 `C` 元素的计算量为 $O(nnz_r)$，而由于这个 `warp` 内的每个线程处理相同个数的 `C` 元素，而且 `C` 中的元素处于同一行，此时涉及的非零元个数一致，计算量基本一致。

   具体实现上，我们分别针对 `k = 32` 和 `k = 256` 考虑，对于 `k = 32`，我们可以让一个线程处理一个 `C` 元素，因此一行 32 个元素交由一个 `warp` 处理。而对于 `k = 256` 而言，我们有多种选择，可以让一个线程处理两个甚至更多的 `C` 元素，经过测试，我们让**一个线程处理两个 `C` 元素**，因此一行 256 个元素交由 128 个线程处理，也就是 4 个 `warp`。

2. 优化 `GPU` 访存：根据前面消除 `warp divergence` 的方法，我们可以设置一个线程块大小为 32 个线程，正好为一个 `warp` 大小。我们注意到，对于这样一个线程块（`warp`），其中的所有线程都需要遍历稀疏矩阵 `A` 的同一行，因此我们考虑将 `A` 存到共享内存内。

   然而也注意到，一个线程块仅有 $48KB$ 的共享内存，而一行稀疏矩阵的非零元可能远远不止这个大小。因此我们考虑设置一个 `tile`，调整 `TILE_SIZE` 来测试。在一个线程块（`warp`）内的线程访问稀疏矩阵的行时，我们按段来访问，对于每一段，先将稀疏矩阵的 `idx` 和 `val` **从全局内存载入共享内存**，然后调用 `sync` 进行同步，接着再进行矩阵乘计算，最后将结果累加到各自线程负责的 `C` 元素上。

   另一方面，对于循环，我们使用**寄存器**来作为累加变量，在最后循环结束后，赋值给全局内存的 `C` 元素上，这样对于全局内存 `C` 的访问只需一次。

3. 消除 `load imbalance` 和局部性：即便 `warp divergence` 的现象基本消除，但是由于稀疏矩阵中非零元分布的严重不均，不同 `warp` 之间的差异仍非常大，因此我们尝试了以下三种思路进行优化，但**均告失败**：

   - 使用 `METIS` 工具，通过修改 `CMakeLists.txt`，配置必要的库等操作，我们使用 `METIS_NodeND` `api` 进行图结构的预处理，根据处理结果对 `A` 进行行重排，最后计算完成后对 `C` 进行一次逆重排，但是发现在 `METIS_NodeND` 的使用中出现了一些问题，而且 `metis` 工具貌似并没有比较好的文档供查阅，因此该方法尝试失败，所尝试的部分位于 `preprocessSparseMatrix` 方法。
   - 有了基本行重排的思想后，我们尝试自己写一个行重排的算法，具体而言，我们根据稀疏矩阵中每行的非零元数目进行排序，以此为基础来进行行重排，在核函数中赋值 `vout` 对应位置元素时，我们将全排列的行 `id` 对应关系传入核函数，以此直接赋值到 `vout` 对应位置，不需要再进行逆重排。最后结果发现效果并不明显，也许是自己对行重排的策略的理解有问题，有待进一步探究，所尝试的部分位于 `rowPermutation` 方法。
   - 我们不进行行重排，而是直接考虑不同的 `warp` 处理不同数目的行，若某行非零元多，则只处理一行，若非零元少，则可以处理多行，具体而言找到该数据集中最大的行非零元数目，以此为基础来提前记录每个 `warp` 处理的

`begin_row_idx` 和 `end_row_idx`，最终效果发现反而更差，可能是因为增加了对于全局内存的访问等等，所尝试的部分位于 `get_begin_and_end` 方法。

2. 实验结果

2.1 CPU

对于 `CPU` 版本的程序，我们根据如下命令：

```
srun -p gpu ~/PA3/script/run_all_CPU.sh
```

分别设置 `k = 32` 和 `k = 256` 进行测试，在三个数据集 `arxiv`、`ddi` 以及 `yelp` 的测试时间和相对于朴素实现 `ref` 的加速比如下：

1. `k = 32`

数据集	spmm_cpu_opt (s)	spmm_cpu_ref (s)	加速比
arxiv	0.00732724	0.244758	33.40
ddi	0.00832527	0.411055	49.37
yelp	0.0905359	3.10078	34.25

2. `k = 256`

数据集	spmm_cpu_opt (s)	spmm_cpu_ref (s)	加速比
arxiv	0.0495807	1.76942	35.69
ddi	0.0793547	3.08933	38.93
yelp	0.527031	21.2295	40.28

2.2 GPU

对于 `GPU` 版本的程序，我们根据如下命令：

```
srun -p gpu ~/PA3/script/run_all.sh
```

分别设置 `k = 32` 和 `k = 256` 进行测试，在 13 个数据集的测试时间和相对于 `cusparse` 的加速比如下：

1. `k = 32`

数据集（nnz）	spmm_opt (s)	cusparse (s)	加速比	吞吐量（nnz/s）
arxiv (1166243)	0.00109927	0.000770396	0.7008	1,060,924,977
collab (2358104)	0.000608618	0.0013337	2.1914	3,874,522,278
citation (30387995)	0.00897942	0.0164968	1.8372	3,384,182,386
ddi (2135822)	0.000275622	0.000641161	2.3262	7,749,098,403
protein (79122504)	0.00818187	0.0245838	3.0047	9,670,467,020
ppa (42463862)	0.0102076	0.0183981	1.8024	4,160,024,100
reddit.dgl (114615891)	0.0214932	0.0483177	2.2480	5,332,658,283
products (123718280)	0.0319088	0.0558511	1.7503	3,877,246,402
youtube (5980886)	0.00306238	0.0036426	1.1895	1,953,018,894
amazon_cogdl (264339468)	0.0535813	0.123914	2.3126	4,933,427,670
yelp (13954819)	0.00350738	0.00658181	1.8766	3,978,701,766
wikikg2 (16109182)	0.00440413	0.00714768	1.6229	3,657,744,435
am (5668682)	0.011891	0.00373897	0.3144	476,720,377

2. k = 256

数据集（nnz）	spmm_opt (s)	cusparse (s)	加速比	吞吐量（nnz/s）
arxiv (1166243)	0.00352652	0.00299706	0.8499	330,706,476
collab (2358104)	0.00434216	0.0052008	1.1977	543,071,651
citation (30387995)	0.0691967	0.0788479	1.1395	439,153,818
ddi (2135822)	0.00155084	0.00155227	1.0009	1,377,203,322
protein (79122504)	0.0681947	0.0804364	1.1795	1,160,244,183
ppa (42463862)	0.0809044	0.0849719	1.0503	524,864,680
reddit.dgl (114615891)	0.183067	0.202119	1.1041	626,087,121
products (123718280)	0.247488	0.258374	1.0440	499,896,076
youtube (5980886)	0.0143425	0.0144209	1.0055	417,004,427
amazon_cogdl (264339468)	0.434197	0.515597	1.1875	608,800,770
yelp (13954819)	0.0266581	0.0299772	1.1245	523,473,878
wikikg2 (16109182)	0.0153077	0.0166462	1.0874	1,052,358,094
am (5668682)	0.0242484	0.0134049	0.5528	233,775,507

最后我们的程序可以在 22 个数据集（除去 arxiv 和 am）左右上超过 cusparse 的性能，在 10 个点左右能达到性能线的要求。

3. 不同优化对性能产生的影响

我们以 `k = 32` 下的 `am` 为例来展示不同优化对性能的影响，最初 `baseline` 为 $0.6s$ 左右：

3.1 消除 warp divergence

在消除 `warp divergence` 之后，性能得到大幅度提升，运行时间从 $0.6s$ 降为 $0.03s$，性能提升近 20 倍。

3.2 优化 GPU 访存

接着我们考虑将稀疏矩阵 `A` 分段存入共享内存，在内部循环使用寄存器进行暂存，合理利用各级缓存，运行时间从 $0.03s$ 降为 $0.014s$。

3.3 预处理稀疏矩阵

我们尝试使用 `METIS` 工具或者自己进行行重排，由于种种原因此处优化并不明显，甚至带来更坏的结果，具体的尝试方法在**实现方法**中已详细阐述。

3.4 参数微调

我们从以下几个参数进行微调：

1. 对于 `k = 256`，考虑每个线程处理的元素数目 `NUM_ELE_PER_THREAD`，调整范围为 `1`、`2`、`4`。
2. 对于 `k = 256`，考虑处理不同元素数目时每个线程的映射方式，比如 `NUM_ELE_PER_THREAD = 2` 时，一行 128 个线程，共四个 `warp`，第一个 `warp` 处理第 `0 ~ 63` 这 64 个非零元，对于第 $i\ (0 \leq i < 32)$ 个线程，考虑：
   - 处理第 $2i$、$2i+1$ 个非零元
   - 处理第 i、$i+32$ 个非零元
3. 调整 `TILE_SIZE`，即分段将稀疏矩阵 `A` 的一行存入共享内存的跨度。

经过调整，运行时间从 $0.014s$ 降为 $0.011s$，并不明显，因此最有效且最应该尝试的方法应该是预处理稀疏矩阵，但我们所尝试的方法都失败了。

4. 最后

我们除了修改 `spmm_opt.h`，`spmm_opt.cu`，`spmm_cpu_opt.h`，`spmm_cpu_opt.cpp` 之外，还修改了 `PA3/CMakeLists.txt` 用于加载 `METIS` 以及必要的库，虽然最终并没有使用到相关的第三方库。

谢谢助教和老师的精心准备，在本学期的几次 `PA` 中，我感觉收获非常大，这是本科期间其他课程均带来不了的收获，也只有这门课让我在真正意义上入门了高性能，体会到了一步步优化程序的感觉，理解集群参数对程序的影响，掌握几种常用的编程方式，收获颇丰！