SpMM 实验报告

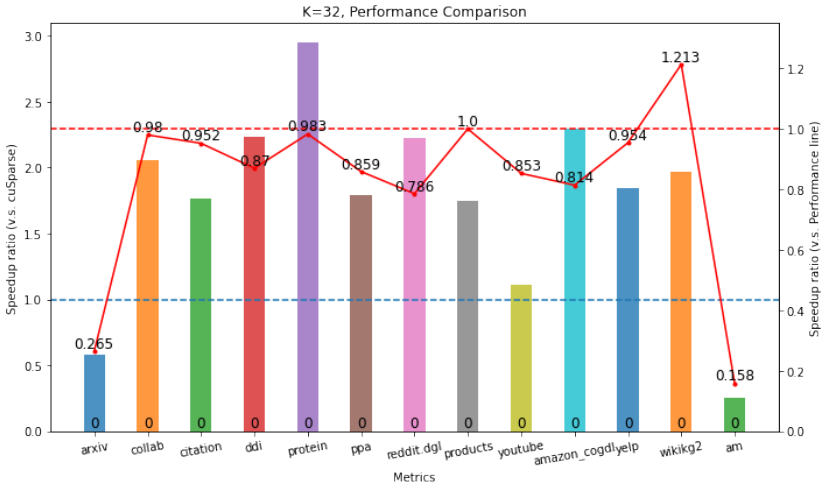
容逸朗　2020010869

实现方法

- Ref 的实现方式是每个线程完成目标矩阵一整行的计算。由于数据的稀疏情况有所不同，对于非零元较多的行，和一个线程完成一个值（1×1）的实现方式相比，运算时间会显著提升，注意到每个线程块需要等待其中的所有线程完成工作后才能释放资源，因此不应为单一线程分配过多的任务。

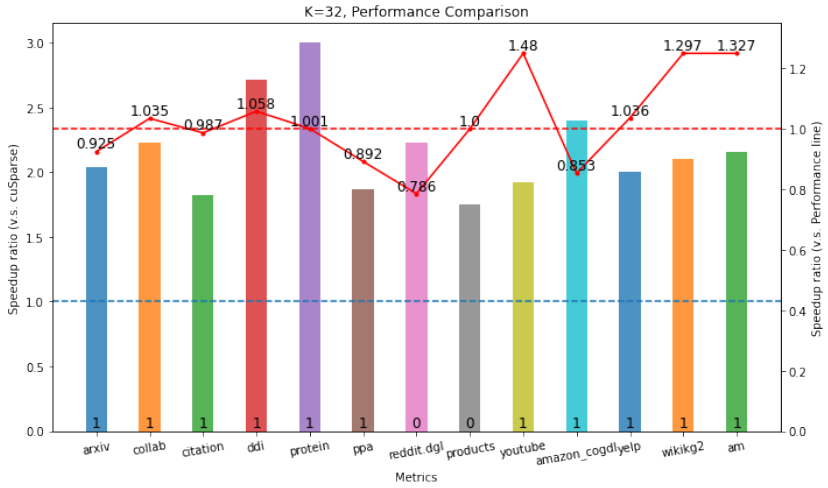
1. 矩阵分片：行合并访存（CRC）方法

- 为了解决 Warp divergence 的问题，一种解决方法是把一个线程块映射到稀疏矩阵的几行，每个 Warp 负责计算一行数据，而 Warp 中的 32 个线程分别对应稠密矩阵的 32 列（Yang et al., 2018），为实现上的简便，当 K=256 时，我们分配 8 个线程块来分别计算矩阵的 $[32 \times bid, 32 \times (bid + 1))$ 列。
- 具体实现时，我测试了两种数据共享的方式，第一种是利用 Warp 级原语广播数据，另一种是使用共享内存实现的方式（Huang et al., 2020），由于两者性能相近，后文提到的 Kernel 都使用 SM 的方式实现。
- 实现此 Kernel 后，在大部分的测试集中基本可以达到和 **cuSparse** 同等的性能：

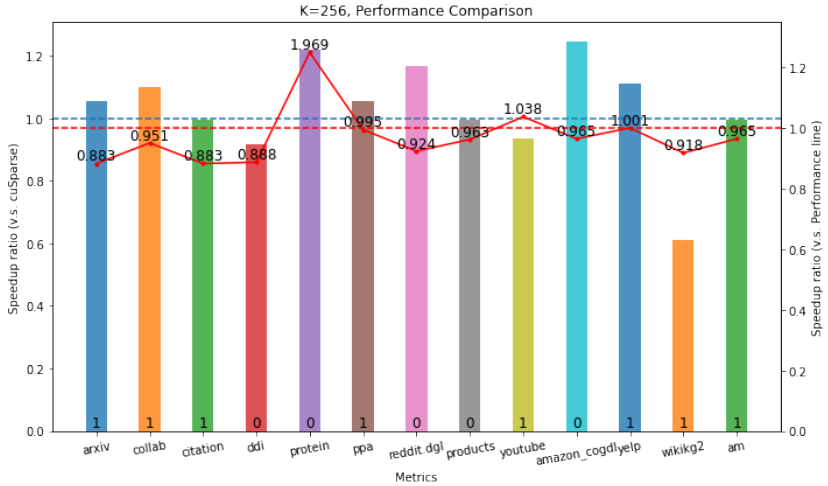


2. 负载不均衡：分层 tiling 方法

- 接下来，针对较差的 arxiv, youtube 和 am 数据集做分析。以 arxiv 为例，有 62006 行（36.6%）为全零行，有 31101 行（18.3%）仅有一个非零元，与之相对的是，有超过 100 行的非零元个数超过 300 个，更有 30 行的非零元数量在 5000 个以上。在这种情况下，如果对于所有行我们都一视同仁分配 32 个线程来计算的话，会导致较明显的负载不均衡现象，这也是导致 arxiv, youtube 为 am 性能不佳的主要问题。
- 因此，我们应当分配更多的线程来计算这些较 “稠密” 的行，一种方式是把所有的行切分为相同大小的 tile，然后每个线程计算这个 tile 的数据和稠密矩阵一列的结果。（Gale et al., 2020）
- 不过为了实现上的简便，我采用了一种替代方法：稠密行（非零元个数大于 256 的行）采用新的 Kernel 计算，此时每个线程块中包含 32×32 个线程（用于把分配到的非零元平均划分为 32 段计算），每行按照数据量划分为 1 到 5 个线程块，由于一行被切分到多个 Warp 中计算，所以需要利用原子加操作统整结果。对于稀疏行，则仍然使用原来的 Kernel 计算。
- 按非零元数量分流后，可以看见在 **K=32** 的测试集下至少可以达到 1.7 倍 **cuSparse** 的性能：

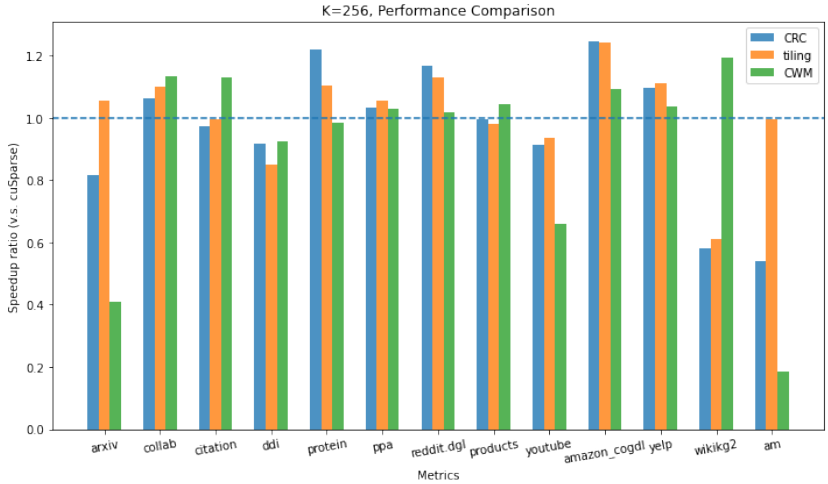


- 在 **K=256** 的测试集下，大部分数据能达到和 **cuSparse** 相近的性能：



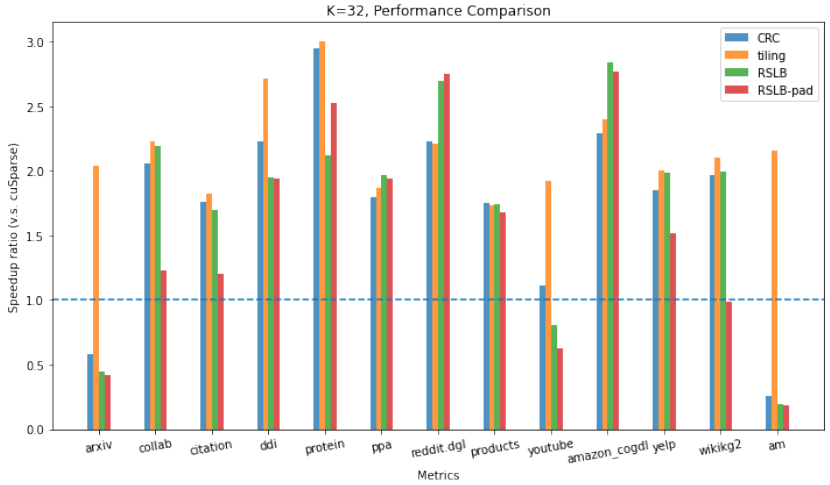
3. 矩阵分片：粗粒度 Warp 合并（CWM）方法

- 从上面的性能图可见，数据集 wikikg2 在 K=256 下的性能和 **cuSparse** 相比仍有一大段距离，通过分析 wikikg2 的数据内容，可以得知，数据集中仅有 0.1% 的行有多于 30 个非零元，而最多非零元个数的一行也只有小于 1000 个的非零元。
- 对于如此稀疏的矩阵，在计算每一行的 256 列时分为 8 个线程块的话会造成大量资源浪费，因此在一个线程中计算的值由原本的一个（1×1）改为了四个（1×4），减少了线程块的数量。（Huang et al., 2020）
- 利用这一优化后，在 wikikg2 测例上可以取得超过 **cuSparse** 的性能：



4. 负载不均衡：Row Swizzle 负载均衡方法

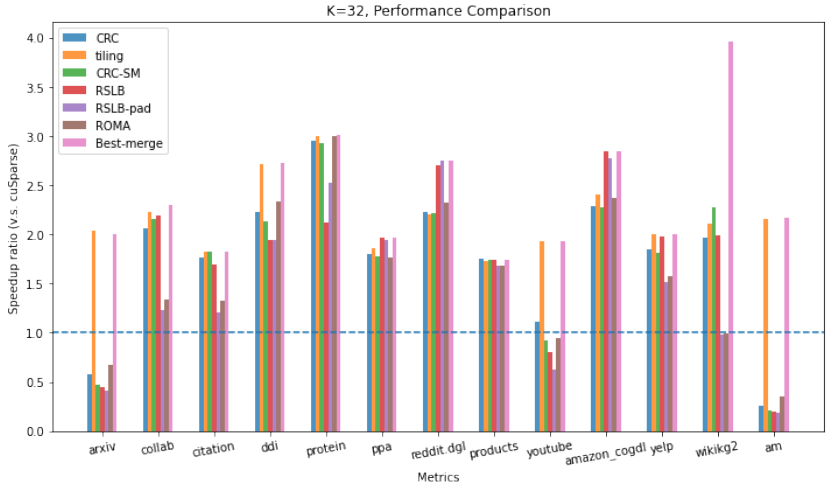
- 接下来目光回到 **K=32** 中性能较差的 reddit.dgl 和 amazon_cogdl 数据集，通过分析发现他们的共同点是有“稠密行”（行中非零元个数大于 3%）。为此，我们可以通过重新映射行的方式来把计算量相近的行放在相邻的 Warp 中，使得 SM 和 Warp 都满足负载均衡的条件。
- 方法十分简单，按照行中非零元的个数由大至小排列并依次映射到 Warp 中即可。（Gale et al., 2020）



- 从中可见，使用本方法后，reddit.dgl 和 amazon_cogdl 数据集均取得了约 20% 的性能提升。

5. 预处理：合并储存格

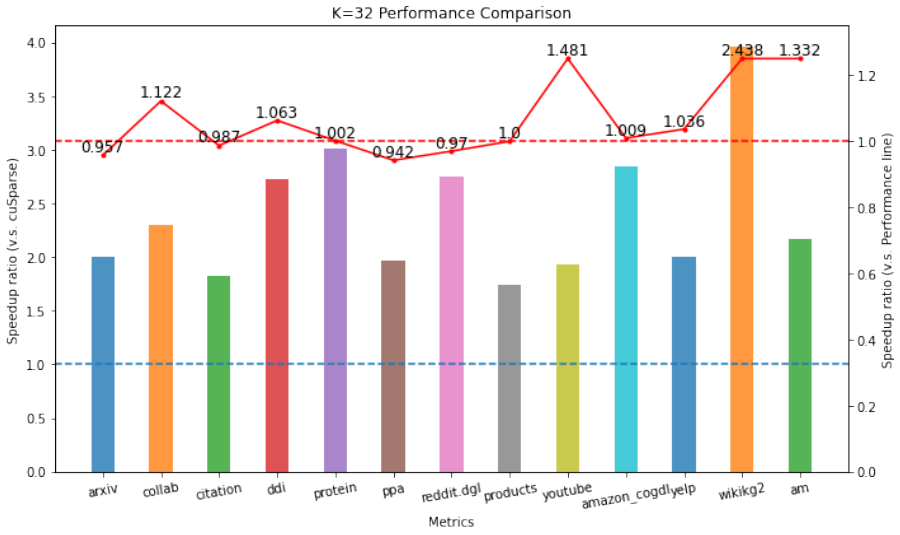
- 注意到稀疏矩阵数据中存在不少个指向同一个位置的值，因此可以在预处理阶段把他们合并为一个值，避免无意义的访存。
- 利用此种优化并结合前述最优的方法后，wikikg2 测例额外提升了一倍的性能：



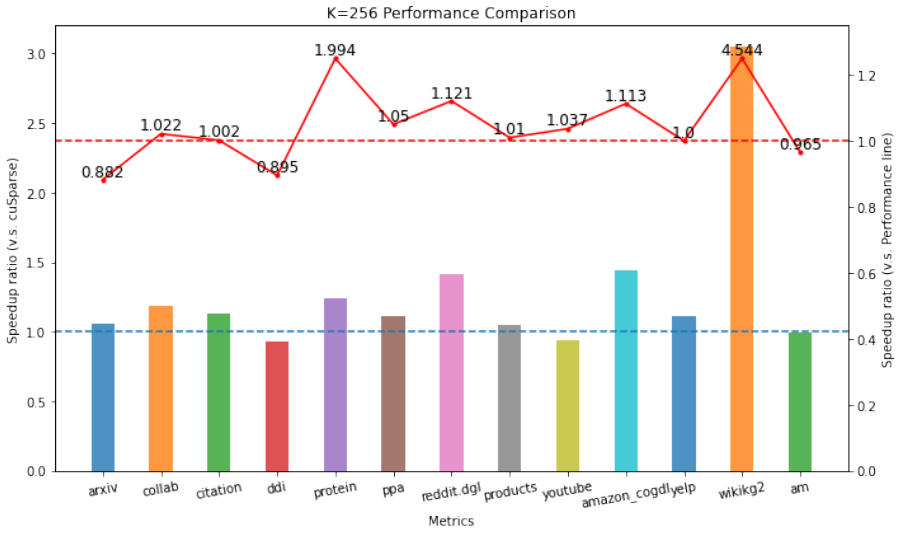
性能

1. GPU 测例

- 代码运行时间相对于 **cuSparse** 实现和性能线的加速比如下图所示：
- $\mathbf{K = 32}$，吞吐量为 $5.401 \times 10^9 nnz/s$：



- $\mathbf{K = 256}$，吞吐量为 $8.239 \times 10^8 nnz/s$ ：



- 具体的运行数据如下，时间单位为 μs：

数据集	K=32			K=256		
	cuSparse	运行时间	加速比	cuSparse	运行时间	加速比
arxiv	730.38	365.68	**1.997304**	2992.67	2833.24	**1.056271**
collab	1271.57	**552.81**	**2.300206**	5203.84	**4405.04**	**1.180950**
citation	16450.7	9029.95	**1.821793**	78939.9	**69845.1**	**1.130214**
ddi	640.51	**235.79**	**2.716408**	1547.41	1675.17	0.928604
protein	24608.9	**8192.28**	**3.003913**	80420.0	**65224.7**	**1.232968**
ppa	18396.3	9346.13	**1.968333**	84900.4	**76192.0**	**1.114295**
reddit.dgl	48197.0	17527.7	**2.749762**	202134	**142990**	**1.413623**
products	55891.1	32005.0	**1.746324**	258312	**247483**	**1.043757**
youtube	3641.26	**1891.09**	**1.925482**	14426.3	**15422.0**	0.935436
amazon_cogdl	123938	**43589.6**	**2.843293**	516151	**362440**	**1.424101**
yelp	6579.24	**3280.24**	**2.005719**	29979.7	**26984.1**	**1.111014**
wikikg2	7146.41	**1804.95**	**3.959301**	16638.7	**5502.20**	**3.048690**
am	3738.07	**1726.13**	**2.165578**	13397.1	13467.7	0.994758

- 注：粗体表示加速比达到性能线或运行时间满足吞吐量要求。

2. CPU 测例

- 具体的运行数据如下，时间单位为 μs：

数据集	K=32 运行时间	K=256 运行时间
arxiv	3814.27	40804.2
ddi	5476.16	55700.3
yelp	63907.3	457285

参考资料

1. C. Yang, A. Buluç and J. D. Owens, "Design principles for sparse matrix multiplication on the GPU," https://arxiv.org/abs/1803.08601, 2018.
2. G. Huang, G. Dai, Y. Wang and H. Yang, "GE-SpMM: General-purpose Sparse Matrix-Matrix Multiplication on GPUs for Graph Neural Networks", https://arxiv.org/abs/2007.03179, 2020.
3. T. Gale, M. Zaharia, C. Young and E. Elsen, "Sparse GPU Kernels for Deep Learning," https://ieeexplore.ieee.org/document/9355309, 2020.