Clustering Algorithms on GPU

Shuxin Zheng
Trento, Italy
shuxin.zheng@studenti.unitn.it

ABSTRACT

Sparsity matrix has received a lot of attention due to its potential in deep learning research to improve computational efficiency, reduce the model size, and integrate more closely with brain-like computing. However, the unstructured sparsity patterns do not align well with the vectorized instruction sets has limited the performance of using sparse matrix. In order to further benefit from using sparse matrices, accelerating the matrix multiplication is one of the key points. In this paper, we discussed the work to port a reordering tech into GPU that reorders the sparse matrix so as to build a better structure that has the potential to accelerate the matrix multiplication.

1 INTRODUCTION

Sparsity matrix has received a lot of attention due to its potential in deep learning research to improve computational efficiency, reduce the model size, and integrate more closely with brain-like computing. However, getting the calculation speedup is not as easy as saving the memory. One of the key problems is that unstructured sparsity patterns do not align well with the vectorized instruction sets. Therefore a wide range of research is motivated, such as `structured pruning techniques` like neuron [1] and 2:4 [5], and `Blocking techniques` like reordering [4][6] and partitioning like ParMetis [3].
In order to further benefit from using sparse matrices, accelerating the matrix multiplication is one of the key points. In this project, we port one of the state-of-art techniques from CPU to GPU. To better explain our work, we first briefly discuss the existing techs and our target. Since we are only focusing on matrix multiplication, pruning is not what we need, so only the reordering and partitioning tech will be discussed in this report. However, the partitioning tech like ParMetis is also not suited to our case since it's aiming to minimize the `edge list cut` which means minimizing the communication demand, however, there is no communication needed in our sparse matrix multiplication problem, nor the concept `edge list`. A more detailed discussion will be shown in Section 3.2.4. We finally focus on the reordering tech. Reordering and blocking a sparse matrix in a nice way would allow for faster multiplication on the accelerators, however, are only convenient if the cost of reordering is re-payed by the multiplication speed-up. There for, accelerating the algorithm will no doubt enlarge the application range. Among different research, the algorithm `IterativeClustering`[4] is the baseline we finally decided. We chose this algorithm not only because it's the most state-of-art tech, but also due to it does not have the limitation of block size, and has strong portability (check Section 2.3).
All the code and profiling results can be found in the Github.

2 MAIN DESIGN CHOICES

2.1 Data Structures and Functions

2.1.1 Data Structures. No matter the input file format, all data was converted to the CSR(Compressed Sparse Row) format for clustering, the COO format is also used as an intermediary. Besides, the group message was saved in a one-dimension list that has the length as the number of matrix rows. The list is organized as the index of the list represents the row index in the original matrix, and the content of the list is the new group index of the row, -1 means not grouped. For example, a list as `00223` means there be 5 rows in total, while the first and second rows are in the same group, third and fourth also, the fifth row itself is a single group. There is another list that was used to save the referring row index.

2.2 Distance Function

To check the distance between rows, we introduce the Hamming distance and Jaccard distance. The Hamming distance ($v, w \in 0, 1^n$) is:

$$H(v, w) = |v \backslash w| + |w \backslash v|$$

and the Jaccard distance is

$$J(v, w) = 1 - \frac{|v \bigcap w|}{|v \bigcup w|} = \frac{H(v, w)}{|v \bigcup w|}$$

2.3 Algorithm

2.3.1 Iterative Clustering. To explain the algorithm for the clustering in this project, I would first introduce the `IterativeClustering` algorithm[4] as our baseline. The idea of this algorithm is to iteratively check the distance of un-grouped rows with the current select group, then decide whether to group it or ignore it based on the distance and the given τ. The algorithm is shown in Algorithm 1. In the origin algorithm, when grouping new rows inside the group, the label(p_c) which represents the group characteristic will be updated, but we disregard it since we want to find a simpler idea that the iteration is less dependent so as to ease the difficulty for parallelization. This will lead to a poorer result, but we are currently more focused on the adapting process. To be short, in this way, we make sure all the rows are grouped and the rows inside one group are close enough to the group character(so as each other).

2.3.2 Parallel Iterative Clustering. We are now going to adapt the clustering algorithm to GPU. Considering the `IterativeClustering`, we can see that the calculation of the distance in the `if` condition inside the `for` loop is, in fact, independent of each other. So with this property, our basic idea is to parallelize the distance calculation. We keep the process of finding the new group the same as the serial one, but let each thread take responsibility for one or more rows. We ask the threads to check the distance of the rows it is responsible for with the chosen group. If the distance is smaller than τ, the thread assigns this row as a part of the current group; otherwise, do

```
Input: a set of rows $V$, a threshold $\tau$
Output: a set of sets of rows $C$
 1: $C \leftarrow \emptyset$
 2: while $V \neq \emptyset$ do
 3:     $v \leftarrow$ an element extracted from $V$
 4:     $c \leftarrow v$
 5:     $p_c \leftarrow v$
 6:     for $w \in V$ do
 7:         if $dist(w, pc) \leq \tau$ then
 8:             $V \leftarrow V \setminus \{w\}$
 9:             $c \leftarrow c \cup \{w\}$
10:         end if
11:         $C \leftarrow C \cup \{c\}$
12:     end for
13: end while
14: return $C$
```

Algorithm 1: IterativeClustering

nothing and go to the next iteration. The new algorithm is shown in Algorithm 2.

```
Input: a set of rows $V$, a threshold $\tau$
Output: a set of sets of rows $C$
 1: $C \leftarrow \emptyset$
    assigns the rows R to each thread
 2: while $V \neq \emptyset$ do
 3:     if threadid == 0 then
 4:         $v \leftarrow$ an element extracted from $V$
 5:         $c \leftarrow v$
 6:         $p_c \leftarrow v$
 7:     end if
 8:     sync all thread
        do in parallel
 9:     for $r \in R$ do
10:         if $r \in V \& dist(r, pc) \leq \tau$ then
11:             $V \leftarrow V \setminus \{r\}$
12:             $c \leftarrow c \cup \{r\}$
13:         end if
14:         $C \leftarrow C \cup \{c\}$
15:     end for
        end parallel
16:     sync all thread
17: end while
18: return $C$
```

Algorithm 2: ParallelIterativeClustering

3 RESULT & ANALYSIS

The correctness is checked by comparing the reordered density. The raw profiling files and more detailed messages can be found in the github. The performance is shown in Table 1.

3.1 Synchronization issue

The most important problem here in parallel development is the synchronization problem. The thread synchronization should be guaranteed for all the threads inside the grid. The critical section in this algorithm is the calculation for the referring list(e.g lines 3-7). We put another sync function before the end of the while loop to reduce the sync issue once - at the very first, the set V is all the same and already updated, so no need to wait for an update from all the threads. To achieve these requirements, it's not enough to just use the function `__syncthreads()` since it only synchronized the threads inside one block. To solve this problem, I included the tech `cooperative_groups` to synch up the thread inside the whole grid. The `cooperative_groups` ensured that all the threads have finished the calculation before, and therefore it's safe to start the next step.

3.2 Complexity Analysis

3.2.1 Computational Complexity Analysis. The complexity for each distance computation is $2 \times min(rank1, rank2)$ where the rank means the number of non-zero values in the row. The complexity is brought from two parts(n is the reference list size):

(1) Finding the referring row(s) - O(1) for searching and O(1) for comparison among referring list(at most $C_n^2 \times 2 \times min(rank1, rank2)$, but n is usually small, so we think it's a constant).
(2) Comparing between each row, at worst case the complexity of $nrows \times (nrows - 1) \times nnz$

So the computation time should grow as the number of rows and the number of non-zero values grow in theory. But there be some other influencing factors: tau and n(reference list size) will affect the execution time - the smaller tau/ larger n, the longer the time spent. The influence from tau is due to: the higher tau, the easier the row to find a group, so less comparison should be handled. While for the case of n, the larger n, the more time a row need to compare with others. (However, decreasing tau and increasing n does not really means the new density will increase since we do not consider the block boundary message. In the case considered, it will for sure give benefits)

3.2.2 Memory Complexity Analysis. Memory boundary is for the size of CSR matrix. The boundary (global memory size of CPU/GPU (A30)) is combined of several part:

(1) CSR message: $((nrows + 1) + nnz) \times sizeof(int)$
(2) Referring list and result list: $(nrows + n) \times sizeof(int)$
(3) tau: $sizeof(float)$

3.2.3 Real Execution Time Analysis. To evaluate the performance of our development, we collect several datasets with different sizes and formats. The execution time is listed in Table 1 and Table 2 below(only clustering time counted). It's not hard to find that when the matrix size is small, the overhead of parallelization(mostly due to synchronization) is the main barrier to acceleration. In case the matrix is larger, things are somehow very different. By checking the result, we find that when we obtained the low speedup, the final group number is much larger than the high speedup cases. This means the clustering needs more iterations and therefore, more synchronization overhead. So in a word, the final execution time does not only depend on data size but also the data distribution.

3.2.4 Comparation with ParMetis. ParMetis is also a state-of-art tool for partitioning graphs. However, we did not choose it as

Dataset Size ($nrows, ncols, nnz$)	**CPU time** ms	**GPU time** ms	**groupSize**	**speedup** $Sp = T_s/T_p$	**origin block density**	**new block density**
(28, 29, 250)	0.04	0.54784	5	0.073	0.390625	0.325521
(60, 61, 353)	0.259	2.50163	22	0.104	0.16714	0.141426
(289, 289, 1089)	4.379	26.3342	137	0.166	0.122415	0.0802624
(494, 494, 1080)	6.43	50.6829	174	0.127	0.0428299	0.0349379
(1138, 1138, 2596)	40.063	221.846	439	0.181	0.0561807	0.043804
(2358, 2358, 9815)	211.375	533.595	394	0.396	0.126848	0.102582
(3296, 3296, 6432)	1161.43	1678.89	1158	0.692	0.0492165	0.0462494
(3918, 3918, 16697)	1094.61	1574.49	652	0.695	0.122888	0.104733
(6144, 6143, 613871)	12745.5	5195.88	1028	2.453	0.205136	0.135628
(8141, 8141, 1012521)	41882	2207.37	990	18.974	0.599766	0.625693
(8275, 8298, 103689)	19985.7	9087.07	2725	2.199	0.022288	0.0248805
(9507, 9507, 591626)	15785.2	1316.35	378	11.992	0.706416	0.688475
(9507, 9507, 684169)	14990.2	1156.89	324	12.957	0.715251	0.703067
(11949, 11949, 80519)	168349	18287.8	4253	9.206	0.113702	0.0911145
(12009, 12009, 118522)	51114.3	17404.9	3805	2.937	0.0182708	0.0349614
(21608, 94757, 238714)	TLE	164399	5958	-	0.0336473	0.0336865
(58226, 58228, 214078)	TLE	160861	14896	-	0.0259844	0.026782
(65536, 65536, 2456398)	TLE	TLE	-	-	-	-

Table 1: The clustering time for each dataset(n=1, tau=0.6, block size=8, time-limited=5 minutes), TLE means time limit exceed, and - means data not valid

our baseline mainly because this tool is not aiming to accelerate the matrix multiplication. To be more detailed, ParMetis, which is based on Metis, is the parallel Metis version using MPI run on CPU. The algorithm inside Metis is aiming to minimize the edge cut for a graph to minimize the communication cost while in the meantime balancing the calculation load, which means the result of partitioning will be several partitions that have a similar number of vertex. However, our idea does not focus on load balance and just wants to gather all the rows that are close enough together and will later combine them all together to form a new matrix. In short, ParMetis determined the distance of the `vertex` using the edge (also the weighted edge if exists), while our algorithm determined the distance of the `row` by using the idea of Jaccard distance. To consider a graph, Metis is gathering the parts that are close in physics while our idea is gathering the parts that are similar in the collection of connection vertex. Having discussed the different design targets, we can now say that these two techs are definitely not comparable while they are both state-of-art techniques in their own field.

4 OPTIMIZATION

Since ncu is not valid in our cluster, we can not analyze the detailed time consumption for each function. According to what we have learned from the course, I tried to increase the amount of calculation in each thread. There be two ways to achieve this target:

(1) increases the number of rows every thread is responsible for
(2) increases the size of referring row list

Besides, making use of the block message is another idea worth a try. We have this idea because we have not yet considered the block boundary, which, should be one of the important concepts in our design. To make it easier to understand, we give an example below. In the three rows' matrix, when considering the block size = (4,4), according to our current design, the second row will be grouped with the first one. But obviously, it's not a nice choice since we should better group it with the third row.

```
1 0 0 0 1 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 1 1 1
```

4.1 Increases the number of rows

In our original development, each thread will take the response for more than one row when the number of rows is very large. To be detailed, the number of rows is decided by: # of rows / # of total threads. So what we do is ask the threads to take responses for more threads when the number of rows is small. Part of the test result is shown in table2. Only a matrix with a small number of rows may benefit from this. From the results, we can see the execution time does not vary when we decreased the used thread number. So we can conclude that the execution speed is limited by the slowest thread, or say, the largest iteration time that is highly related to the real non-zero value location.

4.2 Increases the size of referring row

We also test the case for increasing the size of referring row list(by setting the `ref_size` in cuda_impl.cuh), it can give more benefits for the execution, though it will give different results(the benefits did not only come from increasing the amount of calculation but also due to less iteration, therefore, less sync issue). The execution time is listed in table3. This idea will have better performance when the dataset is larger but still depends on the non-zero value location(check the case(6144,6143,613871) and (8141, 8141, 1012521)).

Dataset Size $(nrows, ncols, nnz)$	Origin time ms	Multi-row=4 ms	Multi-row=8 ms
(289, 289, 1089)	26.3342	27.1729	42.6342
(494, 494, 1080)	50.6829	52.8937	64.1741
(1138, 1138, 2596)	221.846	208.728	234.402
(2358, 2358, 9815)	533.595	539.789	573.161
(3296, 3296, 6432)	1678.89	1653.73	1691.7
(3918, 3918, 16697)	1574.49	1439.57	1451.83
(6144, 6143, 613871)	5195.88	5925.97	7570.19

Table 2: The multi-row per thread optimized clustering time for small dataset(n=1, tau=0.6, block size=8, time-limited=5 minutes)

For small datasets, enlarging the referring list size will slow down the execution, because it will lead to more useless comparison. In our test cases, size = 4 seems to be the one have the best performance, but to obtain the best configuration, more analysis should be taken out.

4.3 Make use of the block message (coarse-grained reordering)

In order to achieve this idea, we start processing the column index when reading the matrix. Specifically, we store only one column index as the mask index if there is more than one non-zero value located in the same block for each row. By adopting this operation, we guarantee that the rows that have similar block distribution will definitely be grouped together. With this property, the result matrix should have better performance when facing different accelerating architectures such as easy adapting to tensor core[2]. Also, this idea can somehow decrease the memory needed by storing a sparse matrix. The performance is collected in Table 4. To make the result more visible, we marked the result block density in red to show that the result is not as good as unmask case.

It's not hard to find that making use of the block message will give better results on reordering than directly checking the distance. However, there be some cases that the unmasked case performs better than the mask case, this is most probably due to the coarse-grained checking, which means some detailed messages will be ignored. By further checking the results of reordering, we can say that the best result is somehow limited in coarse-grained when the block density is somehow already large. Some possible optimization is also discussed in Section 5. Though the best result of the masked comparison is limited, the worst case is always better than the unmasked case since it will never split the rows which should originally be in the same block into two different groups. A combination of these two ideas can be checked to further improve the result.

5 CONCLUSION

5.1 Conclusion

This project implements a GPU version for clustering similar rows in a sparse matrix. We port the specially decided algorithm `IterativeClustering` into GPU and tried some acceleration strategies. The speedup by using GPU compared to the serial one is really case-depending - it's generally influenced by the number of rows and non-zeros values of the matrix, but also strongly affected by the data distributions. By adapting the strategy to increase the number of rows in response for each thread, we didn't get significant improvements in time but found a way to reduce resource consumption. By increasing the size of referring rows, we see some amazing results, though the results are still strongly influenced by the data distribution. The bad data distributions(in the worst case, each row itself as a group) will lead to more synchronization overhead, and therefore hardly any improvement or even deceleration from parallelization can be obtained.

5.2 Further consideration

5.2.1 Further acceleration. Besides the strategy we adopted during this report, there be other ways we can do to improve the performance. One of them is to gradually reduce the running block, which means making better use of the resources in the same block. An idea to achieve this is to build an array to collect the rows that are not grouped and distribute them to the thread according to the threadIdx. In this idea, we are not going to waste the power on the rows which are not needed for further calculation, but what we need is to build a list structure in GPU. The lib `thrust` may help for dealing.

5.2.2 Improve the result. In order to further improve the result of reordering, several strategies can be taken:

(1) Make use of the block row size message. If a group does not have a number of rows that can be divided by block row size, it means there will be some blocks that are combined by two or more different groups after reordering. It's not good for our results. The idea is to fill the group with empty rows so no block will be combined by different groups.
(2) Combine the coarse-grained comparison and fine-grained one to guarantee the worst case while striving for optimal results.

5.2.3 Possibility to Multi-GPU. With the current memory complexity O(n), we can already support a huge sparse matrix in CSR format. However, we could still discuss the possibility to adapt this algorithm to multiple GPU not only for the larger matrix but also for the potential acceleration. The idea is:

(1) split the matrix into n parts (n=number of GPUs)

Dataset Size $(nrows, ncols, nnz)$	**ref size = 1(origin)** ms	**ref size = 2** ms	**ref size = 4** ms	**ref size = 8** ms
(28, 29, 250)	0.54784	1.20627	2.32653	4.21171
(60, 61, 353)	2.50163	2.50163	4.20659	7.8592
(289, 289, 1089)	26.3342	19.3331	19.2072	26.6537
(494, 494, 1080)	50.6829	33.2298	27.2118	31.6242
(1138, 1138, 2596)	221.846	150.781	102.66	98.6962
(2358, 2358, 9815)	533.595	367.27	334.006	337.228
(3296, 3296, 6432)	1678.89	888.922	495.07	340.592
(3918, 3918, 16697)	1574.49	962.307	776.645	767.145
(6144, 6143, 613871)	5195.88	4923.45	5746.06	8249.24
(8141, 8141, 1012521)	2207.37	2514.66	5609.26	14145.3
(8275, 8298, 103689)	9087.07	6258.49	5184.57	5486.29
(9507, 9507, 591626)	1316.35	1259.77	2303.62	5510.96
(9507, 9507, 684169)	1156.89	1232.26	2606.34	6370.76
(11949, 11949, 80519)	18287.8	9575.29	5429.32	3934.81
(12009, 12009, 118522)	17404.9	10301.8	6817.66	5406.74
(21608, 94757, 238714)	164399	148748	140988	138661
(58226, 58228, 214078)	160861	87993	51643.8	35244.4
(65536, 65536, 2456398)	TLE	TLE	TLE	TLE
(444815, 465017, 833540)	TLE	297174	206403	166784

Table 3: The multi-referring rows optimized clustering time for each dataset(tau=0.6, block size=8, time-limited=5 minutes)

Dataset Size $(nrows, ncols, nnz)$	**unmask time** ms	**mask time** ms	**origin block density**	**mask block density**
(28, 29, 250)	0.54784	0.607232	0.390625	0.390625
(60, 61, 353)	2.50163	3.88813	0.16714	0.157589
(289, 289, 1089)	26.3342	24.4081	0.122415	0.122415
(494, 494, 1080)	50.6829	86.2269	0.0428299	0.0428299
(1138, 1138, 2596)	221.846	298.811	0.0561807	0.0561807
(2358, 2358, 9815)	533.595	691.833	0.126848	0.108076
(3296, 3296, 6432)	1678.89	1700.73	0.0492165	0.0601077
(3918, 3918, 16697)	1574.49	1458.29	0.122888	0.112066
(6144, 6143, 613871)	5195.88	1252.59	0.205136	0.219717
(8141, 8141, 1012521)	2207.37	7750.36	0.599766	0.604141
(8275, 8298, 103689)	9087.07	12987.4	0.022288	0.0233604
(9507, 9507, 591626)	1316.35	4698.04	0.706416	0.676583
(9507, 9507, 684169)	1156.89	4096.37	0.715251	0.682248
(11949, 11949, 80519)	18287.8	20958.4	0.113702	0.113908
(12009, 12009, 118522)	17404.9	34151.3	0.0182708	0.0248926
(21608, 94757, 238714)	164399	120160	0.0336473	0.0341238

Table 4: The clustering time for each dataset(n=1, tau=0.05, block size=8, time-limited=5 minutes)

(2) each GPU does the same thing as in the single one
(3) calculates a mask for each group to represent all of the rows inside it, and assigns the mask as the column index for the first row of the group but ignores all the other column indexes for the other rows.
(4) gathering the group information(group index, group mask, and a list which records the group message), treat the group as a row and go to step 2 till no group can be gathered together anymore.

This idea will involve communication costs, so only give benefits when the cost can be made up by decreasing the calculation time for grouping. But in any case, this algorithm has the potential to support even larger matrix. Also, this idea can also be fit into the single GPU case since we can use the coarse-grained reordering in the first iteration to operate as a partitioner and then use the fine-grained reordering.

REFERENCES

[1] Abdolghani Ebrahimi and Diego Klabjan. 2021. Neuron-based Pruning of Deep Neural Networks with Better Generalization using Kronecker Factored Curvature Approximation. *CoRR* abs/2111.08577 (2021). arXiv:2111.08577 https://arxiv.org/abs/2111.08577

[2] Nick Evanson. 2020. Explainer: What Are Tensor Cores? (2020). https://www.techspot.com/article/2049-what-are-tensor-cores/.

[3] George Karypis. 2011. *METIS and ParMETIS*. Springer US, Boston, MA, 1117–1124. https://doi.org/10.1007/978-0-387-09766-4_500

[4] Paolo Sylos Labini, Massimo Bernaschi, Werner Nutt, Francesco Silvestri, and Flavio Vella. 2022. Blocking Sparse Matrices to Leverage Dense-Specific Multiplication. In *2022 IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. 19–24. https://doi.org/10.1109/IA356718.2022.00009

[5] Asit K. Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stosic, Dusan Stosic, Ganesh Venkatesh, Chong Yu, and Paulius Micikevicius. 2021. Accelerating Sparse Deep Neural Networks. *CoRR* abs/2104.08378 (2021). arXiv:2104.08378 https://arxiv.org/abs/2104.08378

[6] Grégoire Pichon, Mathieu Faverge, Pierre Ramet, and Jean Roman. 2017. Reordering strategy for blocking optimization in sparse linear solvers. *SIAM J. Matrix Anal. Appl.* 38, 1 (2017), 226–248.