Towards an Efficient Sparse Storage Format for the SpMM Kernel in GPUs

Renzo Marini, Ernesto Dufrechou(✉), and Pablo Ezzatti

Instituto de Computación (INCO), Facultad de Ingeniería,
Universidad de la República, Montevideo, Uruguay
{rmarini,edufrechou,pezzatti}@fing.edu.uy

Abstract. The sparse matrix-matrix multiply kernel (SpMM) gained significant interest in the last years due to its applications in data science. In 2018, Zhang and Gruenwald [15] proposed the bitmap-based sparse format *bmSparse* and described in detail the implementation of the SpMM for Nvidia GPUs. The novel format is promising in terms of performance and storage space. In this work, we re-implement the algorithm following the authors' guidelines, adding two new stages that can benefit performance. The experiments performed using nine sparse matrices of different sizes show significant accelerations with respect to cuSparse's CSR variant.

Keywords: Sparse matrix matrix multiplication · GPU · Sparse format

1 Introduction

The era of *big data* has brought about a major paradigm shift and the emergence of new problems in which the information is structured in large graphs. The connection between graphs and sparse matrices (those in which the vast majority of their coefficients are equal to zero) has been extensively studied and, because of it, there are important efforts that propose to express graph problems in terms of basic linear algebra operations on sparse matrices. These efforts have attracted the interest of part of the academic community, historically concentrated on the sparse matrix-vector product (SpMV) for its role in solving systems of linear equations, to other operations such as the sparse matrix-sparse matrix product (SpMM), which has important applications in data science. In particular, the most frequent goal is to find algorithms, implementations, and storage formats capable of running efficiently on parallel hardware.

In recent decades, the trend in computer architecture design has been to incorporate multiple cores on the same chip [2]. As a consequence, the use of throughput-oriented processors [7] to accelerate scientific applications has increased. A paradigmatic example is GPUs, which have been used heavily in the context of dense and sparse linear algebra for more than a decade, to the point where efficient implementations are publicly available for most of the standard operations [3,6,13].

© Springer Nature Switzerland AG 2022
R. Chaves et al. (Eds.): Euro-Par 2021, LNCS 13098, pp. 104–115, 2022.
https://doi.org/10.1007/978-3-031-06156-1_9

Sparse Matrix Multiplication (SpMM) operates on two matrices stored in a sparse storage format, that is, a format to avoid explicitly storing null coefficients. As the pattern of nonzero coefficients, and therefore the space necessary to store the result of the operation, will depend on the interaction between the nonzero coefficients of the operands, it must be estimated or calculated from them, which gives the SpMM a higher level of complexity than the SpMV.

In 2018, Zhang and Gruenwald [15] introduced a sparse block format called *bmSparse*, which adapts the bitmap indexing technique used in the context of relational databases (and also for some of the early sparse formats). Although the performance of similar formats has been studied in the context of SpMV [11], the work mentioned is the first to do so with SpMM. The results obtained for the (*WebBase-1M*) sparse matrix were promising, showing better performance than the libraries CUSP [1] and bhSparse [12].

This work focuses on re-implementing the algorithm proposed in [15] incorporating modifications to improve its performance. Among the optimizations considered, we explore the inclusion of two new stages. One avoids making the product of those blocks that will result in a null block due to their pattern of zeros (T_4). The other computes the final storage space and output nonzero pattern before the numerical multiplication stage (T_9). The experimental evaluation was carried out on a set of nine matrices from the SuiteSparse collection with different characteristics, showing that the new stage T_4 of the algorithm allows significant savings in the execution time of the subsequent stages.

The rest of the work is structured as follows. In Sect. 2, the main concepts about the sparse matrix multiplication operation are summarized. Then, in Sect. 3, the details of the SpMM kernel implementation using the *bmSparse* storage format are studied. Later, we present the main proposals in Sect. 4. The experimental evaluation of our proposals on a set of sparse matrices follows. Finally, the main conclusions drawn during the work and the lines of future work to be developed are presented in Sect. 6.

2 Sparse Matrix Multiplication (SpMM)

Sparse matrix multiplication (SpMM) is a very useful operation in various contexts of linear algebra and graph analysis, with applications such as solvers for algebraic multigrid methods (AMG) [8], triangle counting [4] and breadth-first search (BFS) with multiple sources [9]. Algorithm 1 presents a pseudocode of the row-wise SpMM method presented by [10]. In the case of other typical operations on sparse structures, such as SpMV, a common strategy to improve performance is to exploit prior knowledge about the sparse matrix's sparse pattern to minimize memory operations on global memory [14]. However, SpMM adds additional difficulties since the computational complexity of the problem does not depend solely on the nonzero structure of the separate inputs but on how they interact with each other. On the other hand, in most applications, the SpMM is usually executed only once for each pair of matrices. Therefore,

the optimization techniques that are based on analyzing the nonzero patterns are less effective than in the case of the SpMV, which is usually part of the innermost loop of iterative solvers.

Algorithm 1: SpMM proposed by [10]

1: **for** $a_{i*} \in A$ **do**
2: **for** $a_{ij} \in a_{i*}$ **and** $a_{ij} \neq 0$ **do**
3: **for** $b_{jk} \in b_{j*}$ **and** $b_{jk} \neq 0$ **do**
4: value $= a_{ij} * b_{jk}$
5: **if** $c_{ik} \notin c_{i*}$ **then**
6: $c_{ik} = 0$
7: **end if**
8: $c_{ik} = c_{ik} + value$
9: **end for**
10: **end for**
11: **end for**

The particularities above determine that the major design decisions to consider are how to partition the work to be done between different processing units (in the parallel case).

2.1 The *bmSparse* Storage Format

The *bmSparse* format represents sparse matrices using 8×8 blocks and the following data structures:

- *keys*: An array of integers (uint64_t) represents the position of a block in the block array. The first 32 bits encode the row number, while the last 32 bits encode the column number. The keys appear ordered by row and then by column. The choice of uint64_t to represent the keys makes it possible to represent arrays of up to 2^{32} blocks of columns and rows. For smaller arrays, memory usage could be reduced by modifying the format to allow 32 bits to represent keys.
- *bmps*: An array of integers (uint64_t) that stores in position i, a bitmap associated with the block in position $keys[i]$. Each element of the block is mapped to one bit of the bitmap. The bit is zero if the block element is null and one otherwise.
- *values*: Array with the nonzero values of the array ordered by rows and then by column.
- *offsets*: Array with the start position of each block in *values*.

3 The SpMM with bmSPARSE

Given two input matrices, A and B, stored in *bmSparse* format, the multiplication algorithm for this format performs two principal tasks. First, it has to build

task list that determines which pairs of blocks of A and B must be multiplied and added to a block of the resulting matrix C. The task list can be seen as a list of (i, j, k) tuples, named *tasks*, obtained from the rule:

$$C_{ik} = \sum\nolimits_{j} A_{ij} \times B_{jk}. \tag{1}$$

Once the task list is formed, the algorithm has to process the tasks and construct the output structure. The original algorithm [15] is divided into 7 stages, which are identified as $T_{1...7}$.

The stages T_1, T_2 and T_3 are in charge of creating the task list. Assuming that $A_keys[n]$ stores the key (i, j), the task list is considered as the union of the sets $t'_n, n \in [0, size(A_keys))$, of all the tuples in which the key (i, j) of A_keys participates. If $B_row_j = \{x \mid x \in B_keys \wedge row(x) = j\}$ is defined as the set of *keys* in row j from B, t'_n can be formally expressed as:

$$t'_n = \{\ (i, j, k) \mid A_keys[n] = (i, j) \wedge \exists x \in B_row_j : col(x) = k\}. \tag{2}$$

Note that in the Definition 2, the tasks are represented as tuples of coordinates (i, j, k). However, if $(i, j) = A_keys[n]$ and $(j, k) = B_keys[m]$, the tasks can also be represented by the tuple of positions (n, m). In Eq. (3) the set t_n is defined, which uses the representation of tasks as tuple of positions. There is a one-to-one correspondence between the elements of t'_n and t_n.

$$t_n = \{\ (n, m) \mid A_keys[n] = (i, j) \wedge B_keys[m] \in B_row_j\} \tag{3}$$

We based our implementation on the latter representation, while [15] is based on the former. The main advantage of defining the task list as the union of sets t_n is that these sets can be easily calculated thanks to the characteristics of the format. On the one hand, observe that the second component of the t_n tasks represents positions of the same row of blocks in matrix B.

In stage T_5 the task list is ordered so that tasks associated with the same output block are contiguous. Using the representation of tasks as tuples of coordinates (i, j, k), the above is equivalent to ordering according to (i, k). Because the *task_list* array uses another representation of tasks, to determine the relative order between two items a conversion is done before comparing them.

The T_6 stage is in charge of determining which blocks of the resulting matrix will be non-null, which corresponds to the keys array of the format *bmSparse*. To achieve this, the *task_list* array is interpreted as an array of tasks represented as (i, j, k).

Stage T_7 processes the tasks to generate the resulting blocks. Processing a task involves building a dense version of the input blocks, performing the multiplication, and adding the partial block of results to the corresponding output block.

In stage T_8, the array of values generated in stage T_7 is taken as input, and a new array is created that contains only the non-null values of the original array. Relative orders remain.

Most of the implementation was performed using primitives from the *Thrust* API, with the exception of T_7 that required developing specific CUDA kernels.

4 Main Extensions

In addition to the variants made to the steps defined in [15], In this work, two new stages are proposed, T_4 and T_9 with the aim of improving performance, saving work in other stages.

Different versions of the algorithm are generated by including these stages, each one associated with a different sequence of stages. Valid sequences are represented by paths in the directed graph of Fig. 1. The layout followed by [15] is comparable to the one specified in the previous section, that is, $T_{1,2,3,5,6,7,8}$.

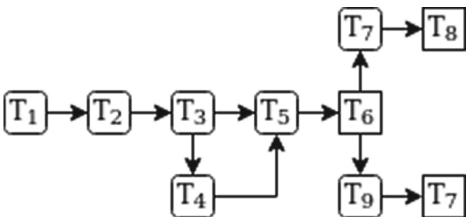

Fig. 1. Possible execution paths.

4.1 T_4: Null-Task Filtering

The result of executing a task is the product of two blocks, which will then be added to one of the blocks of the resulting matrix. If the bitmap of that product is null, that task will not affect the result of the SpMM and could be ignored. In [15], the bitmap of a task is calculated in stage T_7, once the product of the blocks has already been made. However, the bitmap of the product of two blocks can be obtained using only the bitmaps of those blocks, without the need for floating-point operations. Stage T_4 consists of calculating the bitmaps of each task, and eliminating from the task list those that do not contribute to the values of the resulting matrix. It is an optional stage in the sense that the correctness of the algorithm does not depend on it. The motivation for this step is to achieve higher performance in the later stages.

The filtering of tasks is performed using the primitive *thrust::remove_if*, which takes as input the array of tasks and a functor that determines if the bitmap that would be obtained when executing the task is null.

The functor implementation iterates over each dimension of the resulting bitmap, checking for possible intersections between the bits in each row of A and the corresponding column of B. The function returns false if a non-null bit is found.

4.2 T_9: Calculate C Bitmaps Based on A and B's Bitmaps

The stage T_9 computes the array of bitmaps of the resulting matrix using the input matrices' bitmaps.

For this purpose, the first step is to define an iterator of type *thrust::make_transform_iterator* to generate the bitmap resulting from executing a task with the incoming bitmaps. To calculate the output bit (i, k), the associated functor iterates over the dimension j of both input bitmaps. When determining which bit should be in 1 in each bitmap, it starts with a bitmap with 1 in the first position ($0x8000000000000000$) and shifts it to the right according to the row and column number. In case both corresponding bits are in 1, the results are accumulated in the *result* bitmap by a bitwise *or*, as shown in Listing 1.1.

Listing 1.1. Computation of the task bitmap.

```
#define F 0x8000000000000000;
...
uint64_t result = 0;
for (int i = 0; i < 8; i++)
  for (int k = 0; k < 8; k++)
    for (int j = 0; j < 8; j++) {
      const bool A_bit_set = A_bmp & (F >> (i * 8 + j));
      const bool B_bit_set = B_bmp & (F >> (j * 8 + k));
      if (A_bit_set && B_bit_set) result |= F >> (i * 8 + k);

    }
```

Once we have all the bitmaps associated with the block multiplications, *thrust::reduce_by_key* is used to perform a bitwise *or* between the bitmaps of the same output block. Computing the output bitmaps beforehand eliminates the need of the T_8 (compression) stage. However, since only the positions of the nonzero elements, and not their values, are taken into account, it is possible that some of the elements turn out to be null and some explicit zeros end up being stored.

5 Experimental Evaluation

This section makes an experimental evaluation of different variants of the SpMM algorithm based on *bmSparse* described previously.

5.1 Platform Setup and Test Cases

All testing is done on a system comprised of an Intel i7-9750H@2.60 GHz CPU and an NVIDIA GeForce GTX 1660 Ti GPU, Turing architecture. GPU programming is done using CUDA 10.2, and the associated Thrust [3] parallel algorithms library.

Square sparse matrices obtained from the *SuiteSparse Matrix Collection* [5] are used, identified with a number from 1 to 9. The matrices used store single-precision floating-point numbers (*floats*). The characteristics of each matrix are presented in Table 1.

Performance measured by runtime is compared to that of the cuSPARSE [3] library, also part of the CUDA Toolkit.

Table 1. Main characteristics of the matrices used. Arrays 1–3 have dimension close to 10^4, matrices 4–6 have dimension close to 10^5 and the remainder to 10^6.

Name	Id.	Blocks	NNZ	Dimension
cryg10000	1	8613	49699	10000
Goodwin_030	2	20728	312814	10142
ted_A_unscaled	3	13761	424587	10605
Goodwin_095	4	203725	3226066	100037
matrix_9	5	148928	2121550	103430
hcircuit	6	90082	513072	105676
webbase-1M	7	550761	3105536	1000005
t2em	8	572656	4590832	921632
atmosmodd	9	1410884	8814880	1270432

5.2 SpMM Algorithm Performance

We start the analysis by discussing the execution time of each stage. Table 2 details the base implementation (V_{base}) runtimes by multiplying each matrix by itself, broken down according to the stages of the method.

Table 2. Execution time (in µs) of multiplying sparse matrices using the base variant of the SpMM algorithm based on *bmSparse*. Arrays are assumed to be in device memory, that is, transfer time is not included.

Stage	Runtime by matrix								
	1	2	3	4	5	6	7	8	9
T_1	53	58	56	731	185	391	693	1084	2000
T_2	39	43	41	269	172	110	251	436	1858
T_3	202	536	339	1808	1668	1255	3788	2122	6238
T_5	318	1201	642	10355	8994	6906	25392	8505	42552
T_6	303	419	308	2284	2493	2130	5985	2515	7020
T_7	397	2036	965	19477	20618	17313	55158	19693	77953
T_8	58	1264	2659	14121	857	2238	46206	3155	5662
Total	1370	5557	5010	49045	34987	30343	137473	37510	143283

In this implementation, the T_1 stage calls the primitive *thrust::reduce_by_key* on the array of keys, so it is expected that the duration of T_1 depends mostly on the size of that array, which corresponds to the third column of Table 1. This hypothesis can be corroborated from Tables 1 and 2, where the duration of T_1 can be observed to increase with the number of blocks. Something similar happens in stage T_2, where the vector *B_count* is accessed for each element of *A_keys*.

The main goal of stage T_3 is to create the array *task_list*, Therefore, the times of T_3 should depend mainly on the number of tasks that are part of the task list. This hypothesis can be corroborated in Fig. 2, which shows a linear dependence between the number of tasks and execution times.

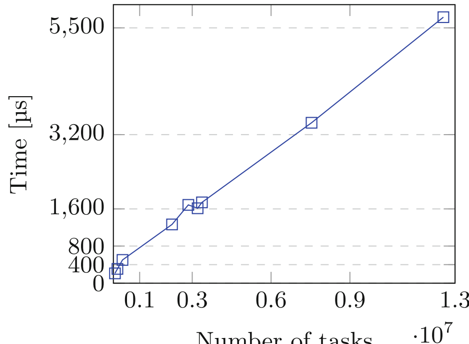

Fig. 2. Execution time (in µs) of T_3 stage as a function of the number of tasks in the task list.

Although the number of tasks is the variable that best predicts the duration of T_3, this information is typically not available before doing the multiplication. However, this duration could be estimated using the number of blocks and the dimension of an array. On the one hand, the more blocks, the more likely tasks will be formed between pairs of them. On the other hand, as the dimension increases, there are more possible positions for the blocks, which decreases the probability that they will form a task. This explains why, with a few exceptions, the T_3 times increase with the number of blocks.

In stages T_5, T_6 and T_7 a similar pattern is repeated between the execution times of each matrix, since the task list is also processed in these stages.

5.3 Impact of Executing T_4

Table 3 shows the runtimes corresponding to each stage of the variant that includes stage T_4 (V_{T4}). Comparing the results with Table 2, it can be observed that, despite the extra cost of including the T_4 stage, the overall performance does not deteriorate in any of the evaluated cases, obtaining execution time reductions of up to 40%.

Table 4 details, for each matrix, the number of tasks that are part of the *task_list* vector, built in stage T_3, and the percentage of tasks that are later discarded at stage T_4. It can be observed that for the selected set of matrices, the number of tasks eliminated varies significantly and that these can represent a high percentage of the tasks generated in T_3.

From Fig. 3, it can be seen that in stages T_5, T_6 and T_7, a linear reduction in execution time is obtained with respect to the amount of discarded tasks.

Table 3. Execution time (in µs) of multiplying sparse matrices using the V_{T4} variant. Arrays are assumed to be in device memory, that is, transfer time is not included.

Stage	Runtime by matrix								
	1	2	3	4	5	6	7	8	9
T_1	53	58	56	731	185	391	693	1084	2000
T_2	39	43	41	269	172	110	251	436	1858
T_3	202	536	339	1808	1668	1255	3788	2122	6238
T_4	42	91	58	592	492	385	1310	603	2249
T_5	167	1080	630	8947	7141	4199	14626	7944	33276
T_6	174	397	303	2106	1997	1391	3519	2387	6002
T_7	257	1731	949	16711	15938	10648	31330	17783	61583
T_8	57	1619	2635	17340	1017	3254	50965	4998	7945
Total	991	5555	5011	48504	28610	21633	106482	37357	121151

Table 4. Amount of tasks generated in T_3 and percentage of tasks eliminated in stage T_4.

Matrix	# tasks	% removed tasks
1	59512	36,3
2	346798	14,4
3	153486	1,6
4	3369528	13,4
5	3205366	20,0
6	2231426	38,5
7	7540274	42,6
8	2851764	7,8
9	12556668	22,1

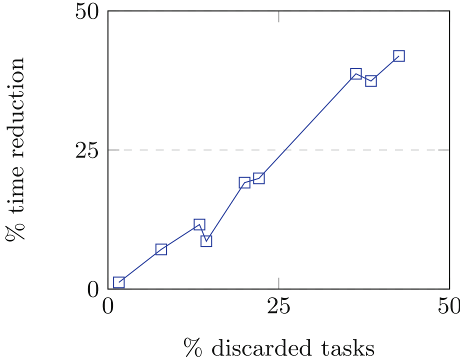

Fig. 3. Percentage of time reduction of stages T_5, T_6 and T_7 based on the percentage of tasks eliminated in stage T_4

On the other hand, and regarding the performance of the stage T_4 itself, it can be corroborated from the data in the Tables 3 and 4, that the execution times depend primarily on the size of the task list, as it happens in the T_3 stage.

5.4 Impact of Executing T_9

To answer whether it is convenient to perform (our variant of) the calculation of the array of bitmaps and offsets before the multiplication, Table 5 compares a variant that incorporates the previous calculation of the output bitmaps ($V_{T4\&T9}$) with the version V_{T4}. In this table, it can be seen that the time of the T_7 stage for the $V_{T4\&T9}$ version is less than that of the V_{T4} version in all cases. The main reason is that, in the first case, the calculation of the output offsets and bitmaps array is considered part of T_9, while in the second, it is considered part of T_7. However, when comparing the total time of both versions, the inverse relationship occurs. This is because stage T_8 in V_{T4} runs in less time than stage T_9 in $V_{T4\&T9}$. In the implementation used by $V_{T4\&T9}$, almost 90% the runtime of stage T_9 is spent executing the functor that generates bitmaps from tasks. This was easily verified replacing the functor by a trivial one in an informal experiment.

Table 5. Average duration (in µs) of the stages after T_6 in the V_{T4} and $V_{T4\&T9}$ variants. The last two rows show the sum of the times of the mentioned stages.

Stage	Versión	Runtime by matrix								
		1	2	3	4	5	6	7	8	9
T_7	V_{T4}	260	1730	952	16793	15985	10635	31320	17782	60671
	$V_{T4\&T9}$	226	1650	887	16099	14094	9381	28287	16105	53778
T_8	V_{T4}	76	225	222	1304	2399	1995	5464	2452	6727
T_9	$V_{T4\&T9}$	233	1002	621	8301	5723	3521	10750	5957	21081
Total	V_{T4}	336	1955	1174	18097	18384	12630	36784	20234	67398
	$V_{T4\&T9}$	459	2652	1508	24400	19818	12902	39038	22062	74860

Since the T_4 stage performs a job similar to that of T_9, a possible improvement is to calculate the bitmaps of those tasks that are non-null in the T_4 stage. In this way, the T_9 stage would only be in charge of reducing the bitmaps of the tasks that correspond to the same C block. As future work, it is interesting to study this change and possible optimizations to the functor that calculates the bitmaps of each task.

Comparison with CuSPARSE. Table 6 compares the execution times of the V_{T4} variant based on the *bmSparse* format and the implementation for the CSR format included in cuSPARSE. It is observed that V_{T4} has a better performance in all matrices except the first one, where there is a small difference that favors cuSPARSE. The matrix with the most significant difference is matrix 5, where the execution time of cuSPARSE is approximately 5× longer than that of *bmSparse*.

Table 6. Execution times (in µs) of the V_{T4} variant based on *bmSparse* and the implementation of cuSPARSE based on CSR. In both cases, values of type `float` and input arrays previously loaded into device memory are assumed.

	Matrix								
	1	2	3	4	5	6	7	8	9
Time cuSPARSE	882	3998	5841	46555	140496	22113	455073	48310	202372
Time bmSPARSE	991	5555	5011	48504	28610	21633	106482	37357	121151
Relative perf.	0.89×	0.72×	1.17×	0.96×	4.91×	1.02×	4.27×	1.29×	1.67×

6 Concluding Remarks

The SpMM is a sparse matrix operation that has interesting applications in data science. The *bmSparse* format, presented by Zhang and Gruenwald [15], is a novel bitmap-based sparse format, specially conceived to achieve high performance for the SpMM in throughput-oriented processors such as GPUs. The use of bitmaps effectively addresses one of the main challenges of the SpMM, which is to determine the nonzero pattern of the output matrix from the two input matrices, and the preliminary results presented by the authors are promising.

We have re-implemented the algorithm based on the directions of [15] and proposed two new stages that can improve its performance. T_4 stage removes unnecessary tasks from the task list based on the bitmaps of the blocks of A and B that form each task. The experimental results on a set of nine sparse matrices from the SuiteSparse Matrix Collection show that the savings on the time of subsequent stages greatly compensate for the addition of T_4, achieving interesting runtime reductions. The addition of T_9, which computes the resulting bitmaps from the bitmaps of A and B blocks, did not result in a performance gain, although several possible optimizations to this stage were identified. The variant that includes T_4 (V_{T4}) is superior to cuSPARSE in 6 out of 9 test cases and achieves up to 5× runtime reduction.

We intend to fine-tune each stage of the algorithm for future work, concentrating on optimizing the most time-consuming stages. We are also interested in adapting the implementation to harness GPUs equipped with Tensor Cores.

Acknowledgment. This work is partially funded by the ANII-MPI project *Efficient computational methods for numerical linear algebra on heterogeneus architectures*. Additionally, the authors thank PEDECIBA Informática and the University of the Republic, Uruguay.

References

1. Bell, N., Garland, M.: Cusp: generic parallel algorithms for sparse matrix and graph computations (2012). http://cusp-library.googlecode.com. Version 0.3.0
2. Blake, G., Dreslinski, R.G., Mudge, T.: A survey of multicore processors. IEEE Signal Process. Mag. **26**(6), 26–37 (2009). https://doi.org/10.1109/MSP.2009.934110

3. cuSPARSE: CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/cusparse/index.html
4. Davis, T.A.: Graph algorithms via suitesparse: graphblas: triangle counting and k-truss. In: 2018 IEEE High Performance extreme Computing Conference (HPEC), pp. 1–6 (2018). https://doi.org/10.1109/HPEC.2018.8547538
5. Davis, T.A., Hu, Y.: The university of Florida sparse matrix collection. ACM Trans. Math. Softw. **38**(1), 1–25 (2011). https://doi.org/10.1145/2049662.2049663
6. Dufrechou, E., Ezzatti, P., Quintana-Ortí, E.S.: Selecting optimal SpMV realizations for GPUs via machine learning. Int. J. High Perform. Comput. Appl. **35**(3), 254–267 (2021). https://doi.org/10.1177/1094342021990738
7. Garland, M., Kirk, D.B.: Understanding throughput-oriented architectures. Commun. ACM **53**(11), 58–66 (2010). https://doi.org/10.1145/1839676.1839694
8. Georgii, J., Westermann, R.: A streaming approach for sparse matrix products and its application in Galerkin multigrid methods. Electron. Trans. Numer. Anal. **37**, 3–5 (2010)
9. Gilbert, J.R., Reinhardt, S., Shah, V.B.: High-performance graph algorithms from parallel sparse matrices. In: Kågström, B., Elmroth, E., Dongarra, J., Waśniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 260–269. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75755-9_32
10. Gustavson, F.G.: Two fast algorithms for sparse matrices: multiplication and permuted transposition. ACM Trans. Math. Softw. **4**(3), 250–269 (1978). https://doi.org/10.1145/355791.355796
11. Kannan, R.: Efficient sparse matrix multiple-vector multiplication using a bitmapped format. In: 20th Annual International Conference on High Performance Computing, pp. 286–294 (2013). https://doi.org/10.1109/HiPC.2013.6799135
12. Liu, W., Vinter, B.: A framework for general sparse matrix-matrix multiplication on GPUs and heterogeneous processors. J. Parallel Distrib. Comput. **85**(C), 47–61 (2015). https://doi.org/10.1016/j.jpdc.2015.06.010
13. Volkov, V., Demmel, J.W.: Benchmarking GPUs to tune dense linear algebra. In: SC 2008: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, pp. 1–11 (2008). https://doi.org/10.1109/SC.2008.5214359
14. Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., Demmel, J.: Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In: SC 2007: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, pp. 1–12 (2007). https://doi.org/10.1145/1362622.1362674
15. Zhang, J., Gruenwald, L.: Regularizing irregularity: bitmap-based and portable sparse matrix multiplication for graph data on GPUs. In: Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA). GRADES-NDA 2018. Association for Computing Machinery, New York (2018). https://doi.org/10.1145/3210259.3210263