Received: 14 March 2023 | Revised: 8 June 2023 | Accepted: 30 June 2023

DOI: 10.1002/cpe.7871

RESEARCH ARTICLE

Sparse matrix-vector and matrix-multivector products for the truncated SVD on graphics processors

José I. Aliaga[1] | Hartwig Anzt[2,3] | Enrique S. Quintana-Ortí[4] | Andrés E. Tomás[1,5]

[1]Depto. de Ingeniería y Ciencia de Computadores, Universitat Jaume I, Castellón de la Plana, Spain

[2]Steinbuch Centre for Computing, Karlsruhe Institute of Technology, Karlsruhe, Germany

[3]Innovative Computing Lab, University of Tennessee, Knoxville, (Tennessee), USA

[4]Depto. de Informática de Sistemas y Computadores, Universitat Politècnica de València, Valencia, Spain

[5]Depto. de Informática, Universitat de València, Valencia, Spain

Correspondence
José I. Aliaga, Depto. de Ingeniería y Ciencia de Computadores, Universitat Jaume I, Castellón de la Plana, Spain.
Email: aliaga@uji.es

Funding information
US Exascale Computing Project, Grant/Award Number: 17-SC-20-SC; U.S. Department of Energy Office of Science; European High-Performance Computing Joint Undertaking (JU), Grant/Award Number: 955558 (eFlows4HPC project); European Union's Horizon 2020 Research and Innovation Programme; Spanish National Plan for Scientific and Technical Research and Innovation (MCIN/AEI/10.13039/50110001 1033), Grant/Award Number: PID2020-113 656RB; Universitat Jaume I, Grant/Award Number: UJI-B2021-58

Summary

Many practical algorithms for numerical rank computations implement an iterative procedure that involves repeated multiplications of a vector, or a collection of vectors, with *both a sparse matrix A and its transpose.* Unfortunately, the realization of these sparse products on current high performance libraries often deliver much lower arithmetic throughput when the matrix involved in the product is transposed. In this work, we propose a hybrid sparse matrix layout, named CSRC, that combines the flexibility of some well-known sparse formats to offer a number of appealing properties: (1) CSRC can be obtained at low cost from the popular CSR (compressed sparse row) format; (2) CSRC has similar storage requirements as CSR; and especially, (3) the implementation of the sparse product kernels delivers high performance for both the direct product and its transposed variant on modern graphics accelerators thanks to a significant reduction of atomic operations compared to a conventional implementation based on CSR. This solution thus renders considerably higher performance when integrated into an iterative algorithm for the truncated singular value decomposition (SVD), such as the randomized SVD or, as demonstrated in the experimental results, the block Golub–Kahan–Lanczos algorithm.

KEYWORDS

graphics processing units, singular value decomposition, sparse matrix-multivector product, sparse matrix-vector product

1 | INTRODUCTION

Many problems in cryptography, image processing, quantum physics, and earth's atmosphere analysis require information about the most significant singular values of a system matrix A; and some efficient algorithms for this task are based on the truncated singular value decomposition (SVD).[1] A central building block in these algorithms are the sparse matrix-vector product (SpMV) and the related sparse matrix-multivector product (SpMM) involving both the matrix A and its transpose. This occurs, for example, in the randomized SVD[2] as well as in the Lanczos procedure based on the block Golub–Kahan–Lanczos method.[3] The SpMV and SpMM kernels play a central role also in the solution of sparse linear systems[4] and sparse eigenvalue problems.[5]

Given the importance of SpMV, a significant effort has been devoted to accelerate the execution of this kernel on a large variety of processor architectures. The significant difficulties faced in this task arise from (1) the low arithmetic intensity of the operation, due to the meager ratio

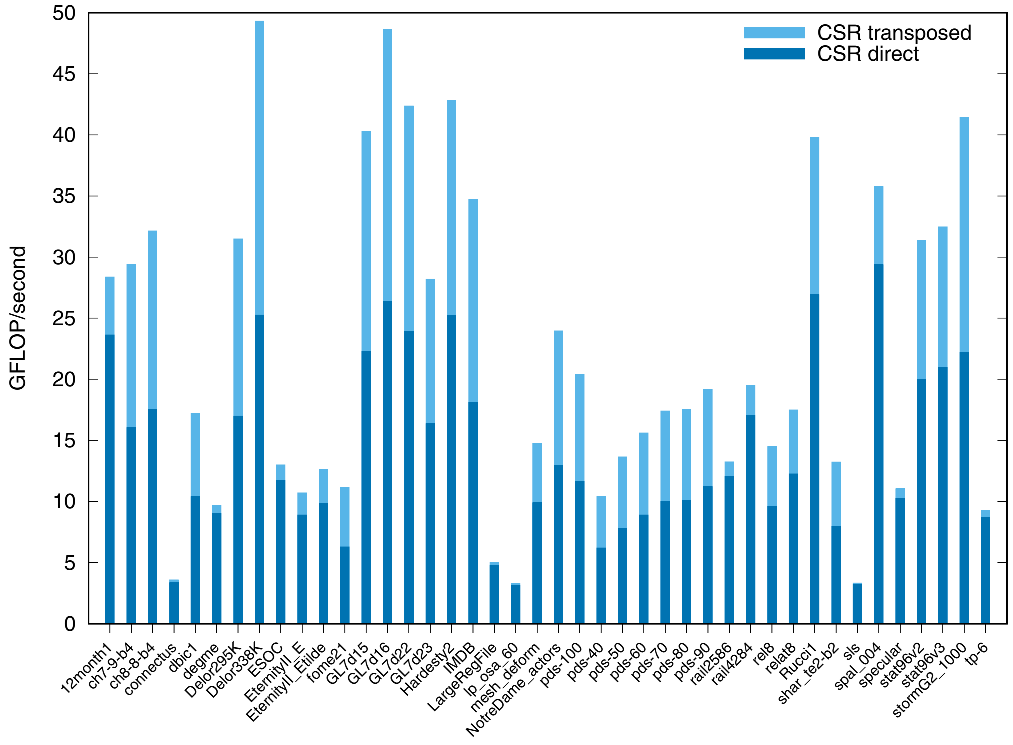

FIGURE 1 Performance of the combined direct and transposed sparse matrix-vector product cuSPARSE kernels (in GFLOPS, or billions of floating point operations per second) using the compressed sparse row format on a NVIDIA A100 GPU for a subset of matrix cases from the SuiteSparse Matrix Collection.

between the number of floating-point operations (flops) and memory accesses, which exacerbates the memory bottleneck of current processor architectures; and (2) the irregular data access pattern, which reduces the cache hit ratio and makes it very challenging to attain a balanced workload distribution in the case of parallel architectures. The combined outcome of these two caveats is that the SpMV kernel is far from delivering the full peak flop performance of current processors, in many cases achieving as little as 10% of that rate.[6-11] For a SpMM kernel involving *k* (column) vectors, the arithmetic intensity of the operation can be increased by a factor of *k*. However, given that in the iterative algorithms for the truncated SVD this problem parameter is selected to be small, the multivector variant mostly remains a memory-bound kernel.

The coordinate sparse format (COO) and the compressed sparse row format (CSR) are two flexible, application-independent sparse matrix layouts[12] upon which many realizations of SpMV and SpMM are built. In particular, this is the case for the implementation of these two kernels in NVIDIA's cuSPARSE library for graphics processing units (GPUs) and Intel's Math Kernel Library (MKL) for general-purpose processors (or CPUs). Unfortunately, as shown in Figure 1 for an NVIDIA A100 GPU, the performance of the SpMV kernel in cuSPARSE largely varies depending on the matrix dimensions, sparsity pattern/degree and, *particularly interesting for this work, whether the operation involves A or its transpose*. Similar comments apply to SpMM and/or the COO format in NVIDIA cuSPARSE, and the corresponding SpMV /SpMM kernels and formats in Intel MKL. At this point, it is worth noticing that many of the low performance datapoints involve A^T, and this could be circumvented by explicitly storing the transpose matrix. However, for the truncated SVD methods considered in this work, that would imply maintaining two copies of the matrix (transpose and non-transpose), duplicating the memory requirements.

The present work revisits the realization of SpMV and SpMM on graphics processing units targeting the special case of sparse matrices with more columns than rows (or vice versa), and making the following specific contributions:

- We propose a new sparse matrix layout, named CSRC (for compressed sparse row/column), that combines some of the characteristics of COO as well as from CSR and its column-wise variant CSC (compressed sparse column).[12] The new format can be obtained from a matrix stored in CSR with a low transformation overhead and is rather competitive in terms of memory requirements with CSR.
- We build implementations of the SpMV /SpMM kernels on top of the CSRC format that provide efficient support for the sparse matrix products involving both *A* and A^T on graphics accelerators with a significant reduction of atomic updates compared to a conventional implementation based on CSR.
- We provide strong experimental evidence of the performance advantages of the CSRC sparse layout and kernel realizations on an NVIDIA A100 GPU. In addition, we evaluate the impact of the new solution when integrated into a block Lanczos algorithm for the truncated SVD.

For reference, all our comparisons in the paper are made against the CSR-based implementations of the sparse matrix products in NVIDIA's cuSPARSE library as this is the golden standard or baseline, in terms of performance, on NVIDIA's GPUs.

The rest of the paper is structured as follows: In Section 2 we describe the new sparse layout and kernel realizations. In Section 3 we discuss the connection between the truncated SVD and the SpMV kernel via the block Lanczos method. In Section 4, we evaluate their performance on an NVIDIA A100 GPU. Finally, we close the paper with a summary and a number of remarks in Section 5.

2 | CSRC SPARSE MATRIX FORMAT

Consider the sparse matrix $A \in \mathbb{R}^{m \times n}$, with n_z nonzero entries, and the goal of computing the SpMV /SpMM operations

$$Y = AX, \quad V = A^T U,$$

where $X, V \in \mathbb{R}^{n \times k}$, $Y, U \in \mathbb{R}^{m \times k}$, $m, n \gg k$ and, without loss of generality, we assume that $m \geq n$. At this point, we remind that, in practice, our work targets sparse matrices with many more rows than columns (i.e., $m \gg n$). Given a block size b, our sparse matrix layout CSRC is a hybrid variant between the CSR and COO formats which stores the sparse matrix using four arrays: p of length $\lceil m/b \rceil + 1$, and r, j, v all three of length n_z, where:

- The entries of p (indices) point to the first element of each row block;
- r and j, respectively, store the row and column indices of the elements; and
- v stores the data values.

The CSRC format combines three common sparse matrix formats: CSR, COO, and CSC. First, the array p is akin to the row index array in the CSR format as, in both cases, the entries of the vectors point to the first element in a row. Second, the array r is equivalent to the row index array in the COO format but differs in that the indices in r are relative to the start of the corresponding row block. As a consequence, the elements of r are in the range $[0, b)$, and therefore they can be stored using fewer bits than the original row indices. Typically $b = 256$ so that r can be encoded as an array of 8-bit numbers instead of 32-bit or 64-bit integers. Finally, the arrays j and a have the same contents as in the CSR and COO formats, but in CSRC they are sorted in increasing column index inside each row block, similarly to the CSC format. In contrast to ELLPACK[6] and its derivatives, the CSRC format does not apply any padding.

The CSRC matrix representation can be cheaply built from the CSR format:

1. p is assembled by sampling the row index array in CSR every b row indices.
2. r is built by traversing the row index array in CSR, computing the distance from the starting row of each row block. After this step, the row index array in CSR is no longer needed and its memory can be released (if necessary).
3. j and v have the same contents as in the CSR format.
4. Finally, r, j, and v are sorted by the value of the column index in CSR for each row block. This procedure is the most expensive step, but it can be performed in parallel for each row block as the order of elements inside a block is independent of the rest.

Figure 2 illustrates how the vectors in the CSR format are transformed to the CSRC format for a simple sparse matrix.

2.1 | Direct product

Figure 3 shows the GPU pseudo-code for the SpMM kernel $Y = AX$ based on the CSRC sparse matrix layout. In this format, the sparse matrix A is stored using the four arrays p, r, j, and v. The dense matrices X, Y are assumed to be stored in column-major order as the target iterative methods for the truncated SVD require each vector to be individually addressed (Furthermore, as the number of columns in X and Y is small, there would be no advantage from maintaining them in row-major format.).

The first loop (line 1) is parallelized via a grid of blocks of threads, with one of these blocks in charge of executing a single row block. Due to the irregular distribution of nonzero entries among the row blocks, depending on the sparsity pattern, each iteration of this loop will exhibit different costs and, therefore, execution times. However, the matrix is split in its large dimension m, yielding a considerable number of rows blocks (specifically, $\lceil m/b \rceil$) so that the scheduler can dynamically balance the workload.

Each block of threads works with only one row block of the sparse matrix. Therefore, it only accesses b elements of each output (column) vector in Y. This allows storing the partial results in shared memory (matrix S), greatly reducing the cost of the procedure. The loops in lines 2–6 initialize this workspace in shared memory.

	0	1	2	3
0	a	c	e	f
1	b	d		
2	g		i	
3	h			j

	CSR vectors									
i	0	4	6	8	10					
j	0	1	2	3	0	1	0	2	0	3
v	a	c	e	f	b	d	g	i	h	j

	Assembling of p									
p	0	6	10							
i	0	4	6	8	10					
j	0	1	2	3	0	1	0	2	0	3
v	a	c	e	f	b	d	g	i	h	j

	Building of r									
p	0	6	10							
r	0	0	0	0	1	1	0	0	1	1
j	0	1	2	3	0	1	0	2	0	3
v	a	c	e	f	b	d	g	i	h	j

	Sorting vectors									
p	0	6	10							
r	0	1	0	1	0	0	0	1	0	1
j	0	0	1	1	2	3	0	0	2	3
v	a	b	c	d	e	f	g	h	1	j

FIGURE 2 Simple 4 × 4 sparse matrix (left) and how the compressed sparse row format is transformed to compressed sparse row/column format with *b* = 2 (right).

```
Input: $p \in \mathbb{N}^{\lceil m/b \rceil + 1}$, $r, j \in \mathbb{N}^{n_z}$, $v \in \mathbb{R}^{n_z}$, $X \in \mathbb{R}^{n \times k}$
Output: $Y \in \mathbb{R}^{m \times k}$

1   parallel for $g := 1, 2, \ldots, \lceil m/b \rceil$
        /* Initialization of S */
2       parallel for $h := 1, 2, \ldots, b$
3           parallel for $i := 1, 2, \ldots, k$
4               $S[h, i] := 0$
5           end
6       end
        /* Accumulation on S */
7       parallel for $h := p[g], p[g] + 1, \ldots, p[g + 1] - 1$
8           for $i := 1, 2, \ldots, k$
9               atomic $S[r[h], i] := S[r[h], i] + X[j[h], i] * v[h]$
10          end
11      end
        /* Update Y */
12      parallel for $h := 1, 2, \ldots, b$
13          parallel for $i := 1, 2, \ldots, k$
14              $Y[h, i] := S[h, i]$
15          end
16      end
17  end
```

FIGURE 3 Pseudo-code for the direct product *Y* = *AX* in compressed sparse row/column.

The main computation corresponds to the loops in lines 7–11. Each iteration of the loop in line 7 is associated with one element in the sparse matrix row block and is distributed across all threads in the block. To attain high performance, it is important to have sufficient work to keep all threads in a block busy. In the CSRC format, this is ensured since there are at least *b* elements (one per row) in the worst scenario. The loop in line 8 computes one row of *Y* from one row of *X*. This loop is performed sequentially by each thread as *k* is typically small. Note that the values of the arrays *r*, *j*, and *v* are the same through the loop and their access is coalesced. As any thread could access any position inside *S*, to avoid race conditions the accumulation is performed using atomic operations. This type of low-level operations on shared memory are very efficient in NVIDIA GPUs and much faster than an alternative implementation with padding. The number of conflicts is greatly reduced by the specific ordering of the sparse matrix elements by columns. This order guarantees that consecutive threads will work on as many elements of one column as possible, that is, with consecutive addresses in *S*.

Finally, the loop in lines 12–17 updates the result vector *Y* with the values from *S*. As each group of *b* elements inside the same column of *Y* is updated by just one block of threads, atomic operations are not required and memory accesses are fully coalesced.

```
Input: $p \in \mathbb{N}^{\lceil m/b \rceil + 1}$, $r, j \in \mathbb{N}^{n_z}$, $v \in \mathbb{R}^{n_z}$, $U \in \mathbb{R}^{m \times k}$
Output: $V \in \mathbb{R}^{n \times k}$

      /* Setup V */
1     parallel for $h := 1, 2, \ldots, b$
2         parallel for $i := 1, 2, \ldots, k$
3             $V[h, i] := 0$
4         end
5     end
      /* Accumulation on V */
6     parallel for $g := 1, 2, \ldots, \lceil m/b \rceil$
7         parallel for $h := p[g], p[g] + 1, \ldots, p[g+1] - 1$
9             for $i := 1, \ldots, k$
                  /* $V[j[h], i] := V[j[h], i] + U[r[h], i] * v[h]$ */
10                reduction($U[r[h], i] * v[h]$, $V[:, i]$, $j[h]$)
11            end
12        end
13    end
```

FIGURE 4 Pseudo-code for the transposed product $V = A^T U$ in compressed sparse row/column.

2.2 | Transposed product

Figure 4 displays the GPU pseudo-code for the transposed SpMM $V = A^T U$ with the sparse matrix stored in the CSRC format. The code is composed of two CUDA kernels: one for zeroing the result V (lines 1–5); and a second one for computing the actual product (lines 6–13). The first kernel can be trivially implemented with coalesced memory accesses. The second kernel is structured similarly to the direct product in Figure 3, with each block of threads computing the partial product of a row block. Identically to the direct product, each row of V is computed sequentially from one row of U (line 9).

In the main kernel, any thread of any block can access any element of V. Therefore, all memory updates must be performed atomically on global memory. Even with the efficient hardware support for atomic operations in NVIDIA GPUs, this type of memory access is slow. Thanks to the ordering by columns inside each row block, consecutive threads will most likely compute values inside a particular column only. Therefore, instead of performing an atomic update in each thread corresponding to the same position in V, we have developed a segmented scan operation that combines values among threads that are working with on the same column. This segmented was used previously in[13] and its CUDA code is reproduced in Listing 1.

```
__device__ inline void reduction(double s, double *y, int jh)
{
  for (int th = 1; th < 32; th *= 2) {
    int jd = __shfl_down_sync(0xffffffff, j[h], th);
    double t = __shfl_down_sync(0xffffffff, s, th);
    if (jh == jd && threadIdx.x + th < 32) s += t;
  }
  int prev = __shfl_up_sync(0xffffffff, jh, 1);
  if (threadIdx.x == 0 || jh != prev) atomicAdd(y + jh, s);
}
```

Listing 1: CUDA code that performs the accumulation on y.

Figure 5 shows a reduced example of the segmented scan using eight threads. The first column represents the contents of the array j (column index) for each thread and the last column corresponds to the result vector $V[:, i]$. The columns in between show the value of the partial sum s at each step of the loop. The arrows represent the messages exchanged between threads, with a dotted line denoting those cases when the received value is not added because it crosses a column boundary. The double arrows are the only atomic additions to $V[:, i]$ in the main memory.

This reduction scheme employs communication operations among threads in order to reduce the number of atomic operations in global memory required for adding the values of V. As all communications are among threads inside a warp, they are very efficient thanks to the CUDA shuffle primitives. The ordering by columns ensures that consecutive threads will work with all elements in a sparse matrix column. The best scenario is when all elements of a column fit inside a warp as, in such case, a single atomic operation to global memory is issued for that column. In contrast, if a column spans more than one warp (32 threads), then an average of $\lceil l/32 \rceil$ atomic operations per column will be issued. Even in this case, the number of atomic operations is significantly reduced with respect to a trivial implementation. Reducing further the number of atomic operations requires a

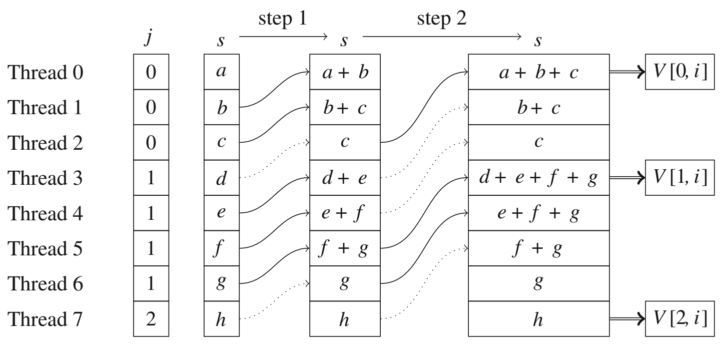

FIGURE 5 Diagram of a segmented scan of 8 elements using eight threads.

complex codification or the addition of zero matrix elements as padding. Both options incur significant overhead, blurring most of the gains obtained from the decrease in the number of global memory transactions.

3 | TRUNCATED SVD VIA THE BLOCK LANCZOS METHOD

The SVD of the matrix $A \in \mathbb{R}^{m \times n}$ is given by

$$A = U\Sigma V^T, \tag{1}$$

where $\Sigma = diag(\sigma_1, \sigma_2, \dots, \sigma_n) \in \mathbb{R}^{m \times n}$ is a diagonal matrix with the singular values of A, while $U \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{n \times n}$ are orthogonal matrices whose columns respectively contain the left and right singular vectors of the matrix.[1] In some applications, we are interested in obtaining a *truncated (or compact) SVD*, of a certain order r, so that

$$U_T \Sigma_T V_T^T \approx A, \tag{2}$$

where $\Sigma_T = diag(\sigma_1, \sigma_2, \dots, \sigma_r) \in \mathbb{R}^{r \times r}$, and U_T, V_T comprise the first r columns of U, V, respectively. Compared with (1), the computational cost of computing this truncated SVD can be considerably lower, especially when the desired rank r is much smaller than the number of matrix columns n.

Figure 6 presents the LancSVD procedure for computing r iterations of the block Golub–Kahan–Lanczos bidiagonalization method,[3] with the block size parameterized by b. The truncated SVD can be computed from this bidiagonalization as $U_T \Sigma_T V_T^T = QU\Sigma V^T P^T$ where $B = U\Sigma V^T$. In practice, a significant part of the computational cost of this algorithm is due to the matrix-vector products (or matrix-matrix products, in case the parameter $b > 1$) involving A and its transpose; see lines 3 and 5.

```
Input: $A \in \mathbb{R}^{m\times n}$, $Q_1 \in \mathbb{R}^{m\times b}$ ; parameters $r \in [1, n]$; $b \geq 1$
Output: $Q \in \mathbb{R}^{m\times r}$, $P \in \mathbb{R}^{n\times r}$, $B \in \mathbb{R}^{r\times r}$

1    $k = r/b$
2    for $i = 1, 2, \dots, k$
3        $P_i = A^T Q_i$
4        Orthogonalize $P_i$ against $[P_1 P_2 \dots P_{i-1}]$ obtaining $L_i^T$
5        $Q_{i+1} = AP_i$
6        Orthogonalize $Q_{i+1}$ against $[Q_1 Q_2 \dots Q_i]$ obtaining $R_i$
7    end
8    $Q = [Q_1 Q_2 \dots Q_i]$
9    $P = [P_1 P_2 \dots P_i]$
10   $B = \begin{bmatrix} L_1 & & & & \\ R_1 & L_2 & & & \\ & R_2 & \ddots & & \\ & & \ddots & L_{k-1} & \\ & & & R_{k-1} & L_k \end{bmatrix}$
```

FIGURE 6 Truncated singular value decomposition via the block Lanczos method with one-side full orthogonalization and basic restart.

4 | EXPERIMENTAL EVALUATION

4.1 | Setup

All the tests in this section were carried out in an NVIDIA A100 GPU with 40 GB of DDR5 RAM. The graphics accelerator is connected to a server equipped with an AMD EPYC 7282 processor. However, the role of the CPU is negligible as almost all the calculations are performed on the GPU. We used a recent NVIDIA SDK with the associated cuSPARSE library (version 11.7.5.86).

The sparse matrices in the evaluation were obtained from the SuiteSparse Matrix Collection,[14] selecting tall-and-skinny matrices ($m \geq 2n$), or vice-versa, with at least 200,000 rows or columns; see Table 1. To help the interpretation of the results, the matrices that originally have more columns than rows are transposed.

4.2 | Memory usage

Figure 7 reports the memory overhead of the CRSC format, with a block size $b = 256$, using the CSR format as the baseline. Here, the memory consumption is measured as the total number of bytes required for all arrays utilized by each format. The compression level achieved by CRSC format is similar to CSR within ±10% except in one case, where CSRC consumes 15% less space than CSR. The memory utilization rate of CSRC is rather good considering that it requires an additional byte for the row index of each matrix element. This extra storage is compensated by the savings attained in the row block indices, which is reduced by a factor 256× due to vector p specifying the initial (column) index of each row block instead of the initial (column) index of each row.

4.3 | Performance of sparse products

Figure 8 compares the performance of the SPMV kernel in cuSPARSE based on CSR and the new implementation that relies on CSRC. The plot includes two bars per matrix, the left one for CSR and the right one for CSRC. Each bar displays the combined performance of the direct and

TABLE 1 Matrices used in the experiments.

Matrix	Rows	Columns	n_z	Matrix	Rows	Columns	n_z
12month1	872, 622	12471	22, 624, 727	ch7-9-b4	317, 520	105, 840	1, 587, 600
ch8-8-b4	376, 320	117, 600	1, 881, 600	connectus	394, 792	512	1, 127, 525
dbic1	226, 317	43200	1, 081, 843	degme	659, 415	185, 501	8, 127, 528
Delor295K	1, 823, 928	295, 734	2, 401, 323	Delor338K	887, 058	343, 236	4, 211, 599
ESOC	327, 062	37830	6, 019, 939	EternityII_E	262, 144	11077	1, 503, 732
EternityII_Etilde	204, 304	10054	1, 170, 516	fome21	216, 350	67748	465, 294
GL7d15	460, 261	171, 375	6, 080, 381	GL7d16	955, 128	460, 261	14, 488, 881
GL7d22	822, 922	349, 443	8, 251, 000	GL7d23	349, 443	105, 054	2, 695, 430
Hardesty2	929, 901	303, 645	4, 020, 731	IMDB	896, 308	428, 440	3, 782, 463
LargeRegFile	2, 111, 154	801, 374	4, 944, 201	lp_osa_60	243, 246	10, 280	1, 408, 073
mesh_deform	234, 023	9393	853, 829	NotreDame_actors	392, 400	127, 823	1, 470, 404
pds-100	514, 577	156, 243	1, 096, 002	pds-40	217, 531	66, 844	466, 800
pds-50	275, 814	83060	590, 833	pds-60	336, 421	99, 431	719, 557
pds-70	390, 005	114, 944	833, 465	pds-80	434, 580	129, 181	927, 826
pds-90	475, 448	142, 823	1, 014, 136	rail2586	923, 269	2586	8, 011, 362
rail4284	1, 096, 894	4284	11, 284, 032	rel8	345, 688	12, 347	821, 839
relat8	345, 688	12, 347	1, 334, 038	Rucci1	1, 977, 885	109, 900	7, 791, 168
shar_te2-b2	200, 200	17, 160	600, 600	sls	1, 748, 122	62, 729	6, 804, 304
spal_004	321, 696	10, 203	46, 168, 124	specular	477, 976	1600	7, 647, 040
stat96v2	957, 432	29, 089	2, 852, 184	stat96v3	1, 113, 780	33841	3, 317, 736
stormG2_1000	1, 377, 306	528, 185	3, 459, 881	tp-6	1, 014, 301	142, 752	11, 537, 419

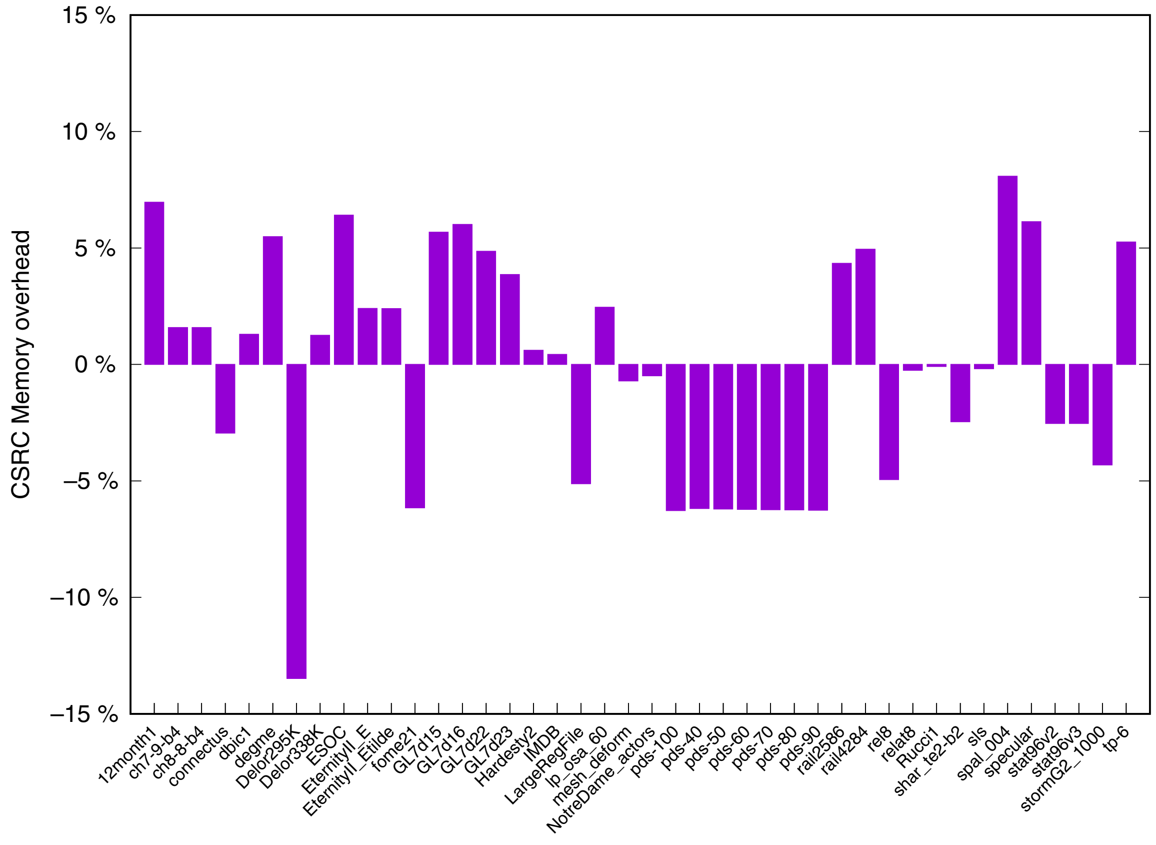

FIGURE 7 Memory overhead of the compressed sparse row/column format compared to compressed sparse row.

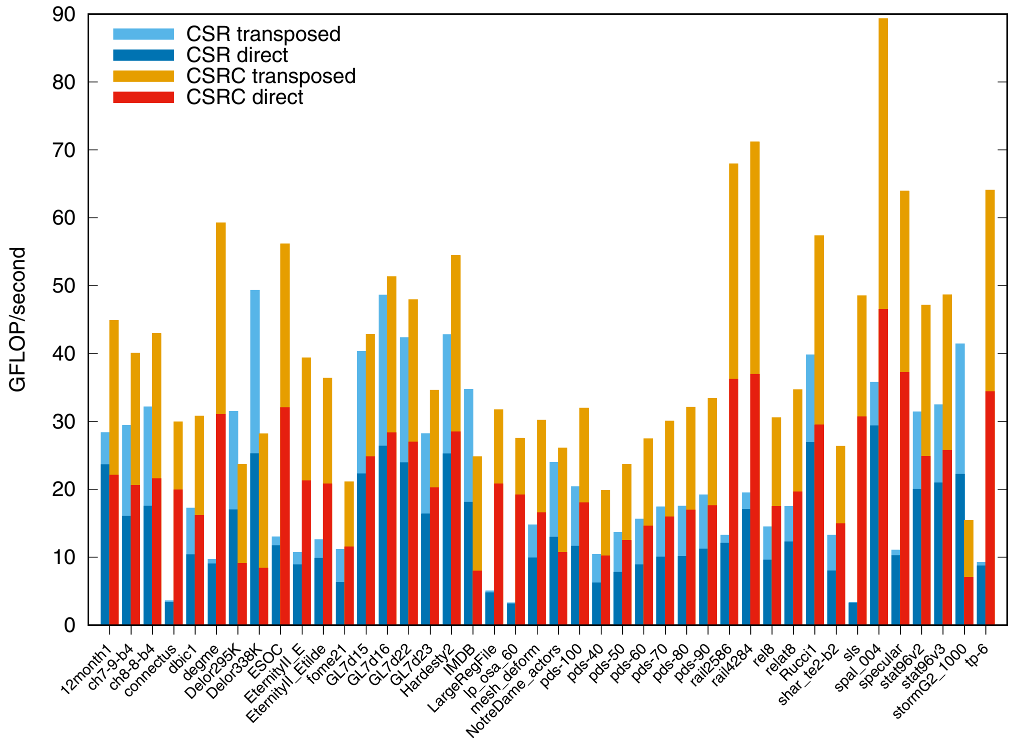

FIGURE 8 Performance of the compressed sparse row/column implementation of sparse matrix-vector product and the corresponding routine in NVIDIA cuSPARSE on the A100 GPU.

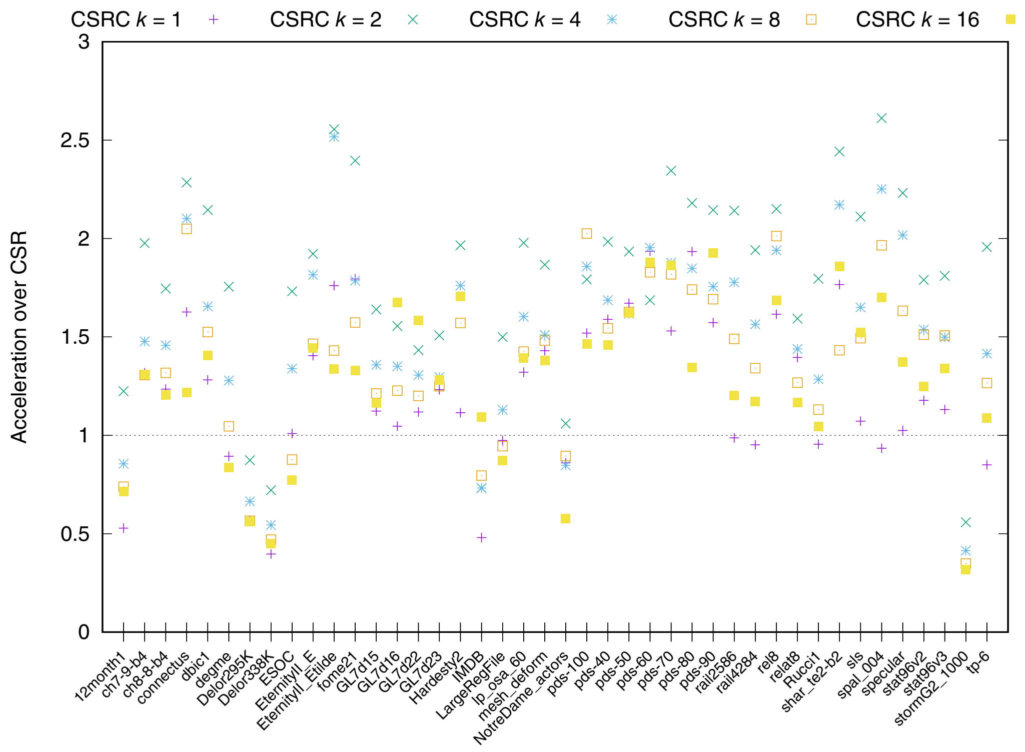

FIGURE 9 Acceleration of the compressed sparse row/column implementation of sparse matrix-multivector product over the corresponding routine in NVIDIA cuSPARSE on the A100 GPU for the product $Y = AX$ and different number of columns in X (values of k).

transposed products, in two different tones: dark for the direct product and light for the transposed variant. Performance is measured in terms of (billions of double precision) floating-point operations (flops) per second, where the number of operations is estimated as two times the number of nonzero elements of the sparse matrix.

In general, the performance rates of CSR (dark blue) and CSRC (dark red) are similar for the direct product, although in a few cases, CSR is faster. In contrast, CSRC clearly outperforms CSR for the transposed product, and the advantage largely compensates the superior performance attained by CSR in the direct product.

The algorithms for the truncated SVD compute the same number of direct and transposed products. From the results in the previous experiment, the CSRC format can be thus expected to deliver superior performance in all but four cases. Among these special matrices, the differences for `12month1` and `Delor295K` are small, but matrices `Delor338K` and `stormG1_100` show a significant advantage for the CSR format. However, the three `Delor` matrices in the experimental subset seem “problematic.” (Actually a fifth matrix in this class, `Delor64K`, was not included in the comparison because the cuSPARSE routine fails with a run-time exception while the CSRC implementation works fine.) The `stormG1_100` matrix is almost empty except for just two diagonals. Furthermore, 62% and 34% of its nonzero elements are respectively 1.0 and −1.0. In this case, a specialized matrix product routine can easily outperform any implementation based on a more general format such as CSR or CSRC.

Figure 9 reports the performance of the SpMM kernel $Y = AX$ in CSRC format compared to the corresponding routine in cuSPARSE, for different values of k (number of columns in the input matrix X). Here the comparison shows the ratio between the execution time of the cuSPARSE routine and that of our CSRC routine, exposing the acceleration of CSRC over CSR. In general, there is a clear pattern, with CSR being the better option (acceleration factors smaller than one) for large k but CSRC offering a preferred alternative for small values of k.

Figure 10 shows the performance of the analogous experiment with the transposed product $V = A^T U$. In this case, CSRC is clearly superior, with a remarkable advantage over CSR. In order to visualize more clearly those cases with differences close to one, the plot in Figure 11 zooms into the range [0,5], exposing that CSRC is faster than CSR in almost all cases. In those cases where CSR is superior to CSRC, it leads by a very small margin.

4.4 | Impact on the truncated SVD

To close the experimental analysis, Figure 12 presents the impact experienced when integrating the routines for the sparse products into a block-Lanczos iterative method, applied to compute 16 largest singular values of each matrix. This last plot shows the execution time of the whole iterative method with a fixed number of iterations and different values of the block size k (equivalent to the number of columns in X, U). As in

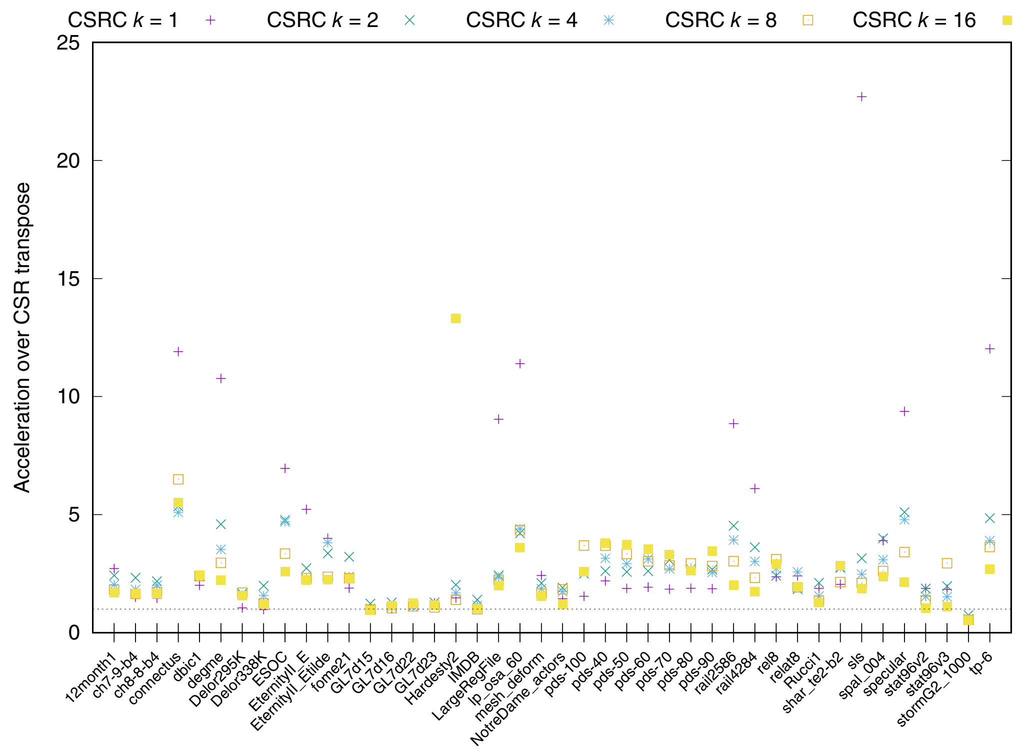

FIGURE 10 Acceleration of the compressed sparse row/column implementation of sparse matrix-multivector product over the corresponding routine in NVIDIA cuSPARSE on the A100 GPU for the product $V = A^T U$ and different number of columns in U (values of k).

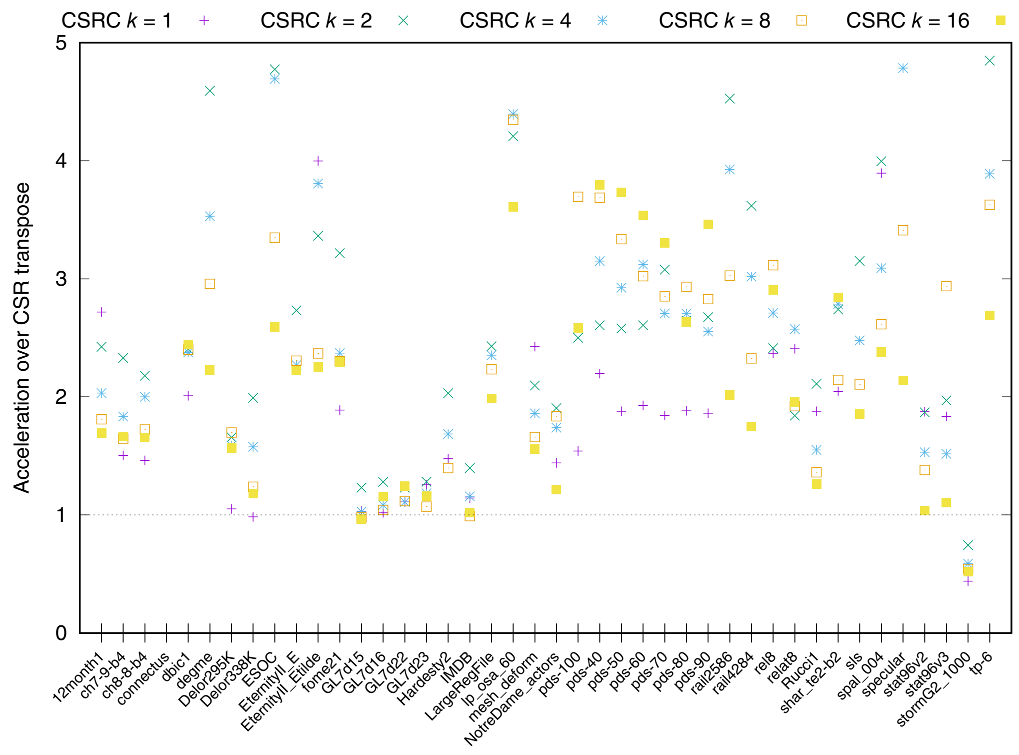

FIGURE 11 Performance of the compressed sparse row/column transposed sparse matrix product compared to compressed sparse row with k vectors on the A100 GPU (zoom up to 5x).

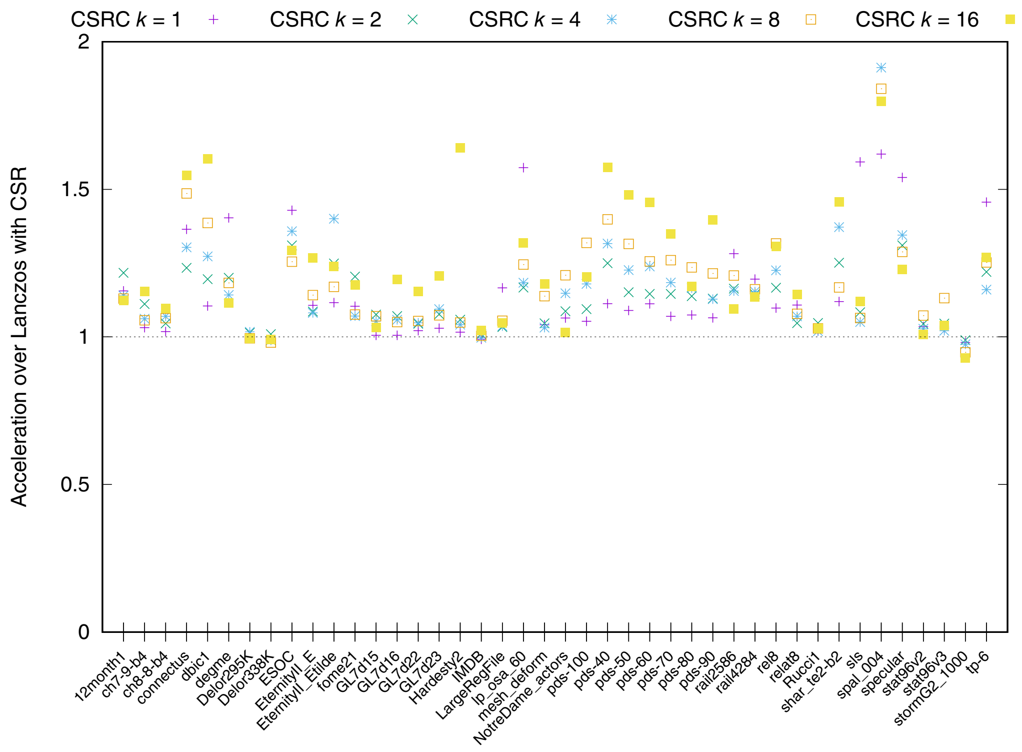

FIGURE 12 Performance of the singular value decomposition Lanczos iteration with k block size using the compressed sparse row/column format compared to compressed sparse row on the A100 graphics processing unit.

the previous comparison, the baseline time refers to the cuSPARSE routine based on CSR, and we report that cost divided by execution time of the same algorithm except for the use of CSRC in the SPMV /SPMM kernels. The Lanczos method computes a direct product plus a transposed one per iteration, interleaved with re-orthogonalizations (via classical Gram-Schmidt) to stabilize the Lanczos method. In our implementation, the re-orthogonalization steps are computed in the GPU using the cuBLAS library. In practice, these dense operations consume a significant part of the time of the whole procedure. Even though the sparse products are not the most expensive part of the iterative method, the CSRC format improves significantly the global execution time over CSR. Again CSR is only faster for all block sizes in the pathological `stormG1_1000` matrix discussed earlier.

5 | CONCLUDING REMARKS

In this paper, we have presented the sparse matrix format CSRC together with an efficient implementation of the SPMV /SPMM kernels for GPUs built on top of it. The CSRC GPU implementation delivers significantly higher performance than the cuSPARSE kernels for the sparse products based on CSR format for tall-and-skinny matrices when the sparse matrix is transposed. In addition, the CSRC kernel is competitive with the cuSPARSE kernel for the direct sparse matrix vector product.

Overall, any iterative method that requires a balanced number of both types of products should experience significantly higher performance without requiring two copies of the matrix (transposed and nontransposed). We illustrate this scenario for a blocked variant of the Lanczos SVD only, but this approach could also benefit transpose-free linear solvers such as the family of methods derived from BiCG or QMR.

ACKNOWLEDGMENTS

This work received funding from the US Exascale Computing Project (17-SC-20-SC), with the collaboration of US Department of Energy Office of Science; Spanish National Plan for Scientific and Technical Research and Innovation: PID2020-113656RB of MCIN/AEI/10.13039/501100011033; Universitat Jaume I: UJI-B2021-58; and the European High-Performance Computing Joint Undertaking (JU) under grant agreement number 955558 (eFlows4HPC project). The JU receives support from the European Union's Horizon 2020 Research and Innovation Programme, and Spain, Germany, France, Italy, Poland, Switzerland, Norway.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available in SuiteSparse Matrix Collection at http://sparse.tamu.edu.

ORCID

José I. Aliaga https://orcid.org/0000-0001-8469-764X
Enrique S. Quintana-Ortí https://orcid.org/0000-0002-5454-165X

REFERENCES

1. Golub G, Loan CV. *Matrix Computations*. 3rd ed. The Johns Hopkins University Press; 1996.
2. Halko N, Martinsson PG, Tropp JA. Finding structure with randomness: probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Rev*. 2011;53(2):217-288. doi:10.1137/090771806
3. Golub GH, Luk FT, Overton ML. A block Lanczos method for computing the singular values and corresponding singular vectors of a matrix. *ACM Trans Math Softw*. 1981;7(2):149-169. doi:10.1145/355945.355946
4. Saad Y, Schultz MH. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J Sci Stat Comput*. 1986;7:856-869. doi:10.1137/0907058
5. Knyazev AV. Toward the optimal preconditioned eigensolver: locally optimal block preconditioned conjugate gradient method. *SIAM J Sci Comput*. 2001;23(2):517-541. doi:10.1137/S1064827500366124
6. Bell N, Garland M. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report No. NVR-2008-004. NVIDIA Corporation. 2008.
7. Buluç A, Williams S, Oliker L, Demmel J. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. Paper presented at: Proceedings of IEEE International Parallel & Distributed Processing Symposium. 2011; Anchorage, AK:721-733. doi:10.1109/IPDPS.2011.73
8. Filippone S, Cardellini V, Barbieri D, Fanfarillo A. Sparse matrix-vector multiplication on GPGPUs. *ACM Trans Math Softw*. 2017;43(4):1-49. doi:10.1145/3017994
9. Choi JW, Singh A, Vuduc RW. Model-driven autotuning of sparse matrix-vector multiply on GPUs. Paper presented at: Proceedings of 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'10; 2010:115-126. doi:10.1145/1693453.1693471
10. Grossman M, Thiele C, Araya-Polo M, Frank F, Alpak FO, Sarkar V. A survey of sparse matrix-vector multiplication performance on large matrices. *CoRR abs/1608.00636*; 2016. http://arxiv.org/abs/1608.00636
11. Liu W, Vinter B. CSR5: an efficient storage format for cross-platform sparse matrix-vector multiplication. *Proceedings of 29th ACM on International Conference on Supercomputing, ICS'15*. ACM; 2015:339-350. doi:10.1145/2751205.2751209
12. Barrett R, Berry M, Chan TF, et al. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. 2nd ed. SIAM; 1994.
13. Aliaga JI, Anzt H, Quintana-Ortí ES, Tomás AE, Tsai YM. Balanced and compressed coordinate layout for the sparse matrix-vector product on gpus. In: Balis B, Heras D, Antonelli L, et al., eds. *Euro-Par 2020: Parallel Processing Workshops*. Bangalore: Springer International Publishing; 2021:83-95.
14. Davis TA, Hu Y. The University of Florida sparse matrix collection. *ACM Trans Math Softw*. 2011;38(1):1-25. doi:10.1145/2049662.2049663

How to cite this article: Aliaga JI, Anzt H, Quintana-Ortí ES, Tomás AE. Sparse matrix-vector and matrix-multivector products for the truncated SVD on graphics processors. *Concurrency Computat Pract Exper*. 2023;e7871. doi: 10.1002/cpe.7871