Balanced and Compressed Coordinate Layout for the Sparse Matrix-Vector Product on GPUs

José Ignacio Aliaga[1], Hartwig Anzt[2,3], Enrique S. Quintana-Ortí[4(✉)], Andrés E. Tomás[1,5], and Yuhsiang M. Tsai[2]

[1] Dpto. 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, USA
[4] DISCA, Universitat Politècnica de València, Valencia, Spain
quintana@disca.upv.es
[5] Dpto. de Informática, Universitat de València, Valencia, Spain

Abstract. We contribute to the optimization of the sparse matrix-vector product on graphics processing units by introducing a variant of the coordinate sparse matrix layout that compresses the integer representation of the matrix indices. In addition, we employ a look-ahead table to avoid the storage of repeated numerical values in the sparse matrix, yielding a more compact data representation that is easier to maintain in the cache. Our evaluation on the two most recent generations of NVIDIA GPUs, the V100 and the A100 architectures, shows considerable performance improvements over the kernels for the sparse matrix-vector product in cuSPARSE (CUDA 11.0.167).

Keywords: Sparse matrix-vector product · Sparse matrix data layouts · Sparse linear algebra · High performance computing · GPUs

1 Introduction

The sparse matrix-vector product (SpMV) is a fundamental operation for the iterative solution of sparse linear systems since it is usually the computationally most expensive building block in stationary schemes as well as Krylov subspace methods [10].

The SpMV is, in general, a memory-bound operation which means that its performance is strongly determined by the memory access volume and the access pattern dictated by the algorithmic realization of the kernel and the memory bandwidth of the target computer architecture. In this context, the irregularity of the memory accesses turns the parallel optimization of SpMV into a challenging task.

A particular factor which directly influences the implementation and (parallel) performance of SpMV is the data layout of the sparse matrix. The coordinate format (COO) [10] is likely the most intuitive layout: for each non-zero matrix entry, this scheme maintains a 3-tuple with the entry's row and column indices and its numerical value. The compressed sparse row format (CSR) [10] is a flexible alternative that reduces the indexing overhead with respect to COO by storing only starting/ending indices (pointers) for each matrix row, while keeping the same information for the column indices and values as COO. A plethora of application-specific sparse matrix layouts have been proposed over the past decades; see [2–5, 8, 9] among many others. In general, these solutions deliver high performance for some problem domains and/or computer architectures but perform poorly and/or require expensive transformations of the matrix format for others.

In [7] we introduced a balancing parallelization scheme for GPUs optimized for matrices with an irregular row distribution of the non-zero entries. In brief, this scheme: 1) is based on the standard CSR format; 2) requires an inexpensive pre-processing step; and 3) consumes only a minor amount of additional memory compared with significantly more expensive GPU-specific sparse matrix layouts. The new balancing approach departs from the conventional parallelization across matrix rows by instead distributing the workload evenly among the thread teams while avoiding race conditions via atomic transactions with efficient support by hardware in recent GPU architectures. In [6], we extended the idea to the COO format, showing that the resulting kernel is superior to some of the most popular SpMV implementations based on both COO and CSR.

In this paper, we continue our effort towards the optimization of SpMV on GPUs by making the following contributions:

– We propose orthogonal (independent) enhancements of the balancing COO-based scheme in [7] that result in a compressed storage format for the matrix data (indices and values), thus reducing the memory traffic and improving performance.
– We develop a high performance realization of this scheme for the most recent generations of NVIDIA GPUs (Volta and Ampere).
– We provide a complete evaluation of the new kernel in comparison with highly optimized implementations of SpMV, based on COO and CSR, from in NVIDIA cuSPARSE (those in CUDA 11.0.167). Following standard practice, this analysis is performed both from the perspective of memory consumption and GFLOPS (billions of floating-point arithmetic operations, or flops, per second).

The idea of compressing the indexing information to reduce the pressure on memory bandwidth is not original. In this sense, our approach is slightly related to the compressed sparse blocks (CSB) format [3], which partitions the sparse matrix into a regular grid of sparse blocks, each of which is stored in CSR format with the block indices compressed as offsets to a reference. In comparison, we also maintain the indices as offsets, encoded using a shorter number of bits. However, our scheme is based on COO instead of CSR; we divide the nonzero matrix entries (instead of the matrix itself) into regular chunks; we couple this partitioning with a balanced workload distribution for GPUs; and we also explore the compression of the numerical data using a look-ahead table.

The rest of the paper is organized as follows. In Sect. 2, we review the COO format and introduce our new balancing and compressed variant for GPUs based on it. In Sect. 3, we evaluate a standalone implementation of the new scheme for SpMV in comparison with the GPU kernels in NVIDIA cuSPARSE. In addition, in that section, we also assess the impact of the scheme when the SpMV kernel is integrated into the biconjugate gradient stabilized method (BICGSTAB) [10]. Finally, in Sect. 4, we offer some concluding remarks and a brief discussion of open research lines.

2 Balanced and Compressed SpMV

2.1 COO Format

Consider the SpMV $y := A \cdot x$, where A is an $n \times n$ sparse matrix with n_z non-zero entries and x, y are both vectors with n components. The COO format employs three vectors: say a, i and j, each of dimension n_z, to maintain the values of the non-zero elements of the matrix and their row and column index coordinates, respectively. In a direct parallelization of the COO-based SpMV on a GPU, each thread operates with a single nonzero element of the matrix, performing the multiplication with the corresponding entry of x, and using *atomic operations* to accumulate the partial result on the appropriate component of y. The performance of this initial approach can be improved if each thread computes several elements of the result vector, as typically 2 or 4 elements suffice for the compiler to aggregate enough memory access operations to overlap transfer and arithmetic operations. The excerpt of CUDA-like code in Listing 1.1 illustrates this approach for a COO-based SpMV with A stored using vectors `a`, `i`, and `j`. There each thread computes `K` accumulations of the form $y_i := y_i + a_{ij} \cdot x_j$, involving `K` nonzero matrix elements. Note that, for simplicity, we assume that n_z is an exact multiple of $\texttt{B} \cdot \texttt{K}$, where `B` denotes the number of threads per block. Otherwise, the matrix can be padded with explicit zero elements.

In practice, the number of iterations in the loop of the `SpMV_kernel` in Listing 1.1 is small, and the whole loop should be unrolled to attain high performance. For that purpose, it is convenient to pad each matrix row with zeros so that its dimension becomes an exact multiple of `K`.

A second "loop" is implicit in the GPU code as the `B` threads of a block perform the operations for a *chunk* of $\texttt{B} \cdot \texttt{K}$ matrix elements. In current NVIDIA GPUs, the number of threads in a block is limited to 1,024 and must be over 192 for good performance. The compromise value $\texttt{B} = 256$ is rather optimal and provides some advantages from the perspective of the compression technique introduced in the next subsection.

Finally, a third (outermost) "loop" is also implicitly present, for the $\lceil n_z/(\texttt{B} \cdot \texttt{K}) \rceil$ thread blocks. With this approach, the GPU hardware scheduler will dynamically assign blocks to each chunk of the matrix. This is important because the

```
#define W 32    // Warp size
#define B 256   // Number of threads per block
#define K 4     // Number of elements per thread

void SpMV(int n, int nz, int *i, int *j, double *a, double *x, double *y)
    {
    cudaMemset(y, 0, sizeof(double) * n);
    int nc = nz / (B * K);
    dim3 tb(W, B / W);
    SpMV_kernel<<<nc, tb>>>(i, j, a, x, y);
}

__global__ void SpMV_kernel(int *i, int *j, double *a, double *x, double
    *y){
    int    p = blockIdx.x, q = threadIdx.y * W + threadIdx.x;
    double v = 0.0;
    for (int l = 0; l < K; l++) {
        int t  = (p * B + q) * K + l;
            v += a[t] * x[j[t]];
    }
    int row = i[p * B + q];
    atomicAdd(y + row, v);
}
```

Listing 1.1. CUDA code for the SpMV with a simple balancing parallelization scheme and A stored in coordinate format.

execution time of the threads can be quite different given the variations in the access cost to the vectors x and y. The reason is that, although each thread process the same number of elements, the matrix pattern can result in very different cache hits and misses during the accesses to the input vector x. Also, the ordering of the matrix elements can introduce an important number of cache misses in the update of the result vector y. In addition, atomic operations must be used to avoid race conditions in this update. Although atomic primitives have efficient support in modern GPU hardware, they introduce contention among the threads introducing further variations to the execution time.

In principle, the COO format does not enforce any specific ordering of the matrix elements. However, a random ordering will result in poor locality during the accesses to the result y. In contrast, a row-major ordering (such as that used in CSR) renders excellent locality during the same accesses, but with higher contention among threads. To avoid this, a segmented scan is implemented using the intra-block communication primitives available on NVIDIA's GPU. The fragment of CUDA code in Listing 1.2 shows this reduction. There, the variable v stores the values that have to be accumulated and the variable *row* their corresponding row indices.

This solution mimics the highly parallel variant of the classic prefix sum: each thread communicates the accumulated value as well as the row index for that value to the thread in the next level of the hierarchy. The accumulation continues if the received row index matches the index of the row assigned to the receiving thread. Assuming a row-major ordering (i.e., consecutive row indices), the thread with the lowest identifier participating in the accumulation of elements for each row accumulates the partial products for all the products in that row. Only this thread issues a global memory access operation to write the final value to the main memory.

```
#define W 32     // Warp size
for (int l = 1; l < W; l *= 2) {
    int    s = __shfl_down_sync(0xffffffff, row, l);
    double t = __shfl_down_sync(0xffffffff, v, l);
    if (row == s && threadIdx.x + l < W) v += t;
}
int prev = __shfl_up_sync(0xffffffff, row, 1);
if (threadIdx.x == 0 || row != prev) atomicAdd(Y + row, v);
```

Listing 1.2. CUDA code that performs the accumulation on y.

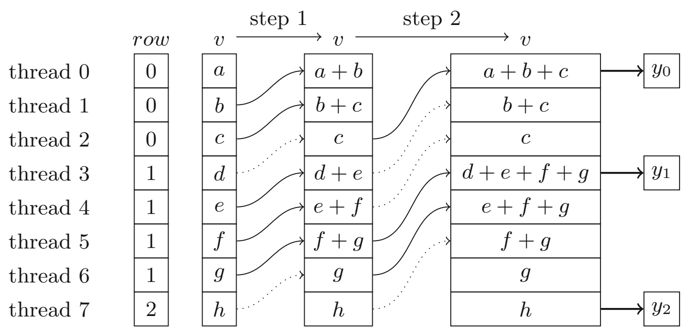

Fig. 1. Diagram of a segmented scan of 8 elements using 8 threads

Figure 1 shows a reduced example using only 8 threads. The first column represents the contents of the *row* variable in each thread and the last column corresponds to the result vector y. The columns in between show the value of v at each step of the loop. Each arrow represents the messages sent among threads, using a dotted line when the received value is not added because it comes across a row boundary. The solid arrows are atomic additions to y in main memory.

This reduction scheme requires 11 communication operations for adding the values for all rows compared with 5 communications in a regular reduction which only computes one sum. Therefore, it is only sub-optimal when the matrix elements processed by a warp pertain to the same row. However, this case is avoided by the compression technique introduced in the next subsection.

2.2 Compression

Following the balanced thread distribution, each block of threads processes exactly a chunk of $\mathtt{B} \cdot \mathtt{K}$ nonzero elements of the sparse matrix. If the matrix elements are ordered row-wise and, by columns inside each row, (as is the case in the CSR format,) each chunk will likely present a significant number of repeated row indices in vector i as well as clustered column indices in vector j. In addition, for some applications, many of the matrix values are repeated. For these reasons, it may be beneficial to use different encodings for each chunk, reducing (compressing) the amount of memory required to store the sparse matrix. This approach avoids thread divergence as the same format is used for all the elements in a chunk. At the same time, the compression level may not be optimal as it needs to account for the values accessed by several threads.

To implement this compression, a handful of auxiliary vectors are required, all of the length $\lceil n_z/(\mathtt{B} \cdot \mathtt{K}) \rceil$ (that is, vectors with one element per chunk). The first vector contains a 1-byte entry per block to specify which particular format is used for that block. Table 1 shows a summary of the possible encodings for a matrix with double precision (DP) floating point data. The row index and element value combinations can be represented by a single bit each, while the column index requires two bits in each 1-byte entry of the vector.

Table 1. Possible encodings of the chunk data for DP data.

	None	8 bits	16 bits	32 bits	64 bits
Row index	×	×			
Column index		×	×	×	
Element value		×			×

Two additional integer vectors then contain the baseline (reference) row and column indices of the elements in the chunk, which correspond to those for the top-leftmost nonzero entry of the sparse matrix in the chunk. Finally, as the space occupied by distinct chunks will be often different, a vector of integers is used to point to the start of each chunk.

Instead of the three original COO vectors (i, j and a), the data of the matrix elements in a chunk are maintained in a *blob* (Binary Large OBject), with the $\mathtt{B}$ row indexes first; followed by the $\mathtt{B} \cdot \mathtt{K}$ column indexes; and finally the $\mathtt{B} \cdot \mathtt{K}$ values. Those blobs are stored contiguously in memory with no alignment issues provided $\mathtt{B}$ and $\mathtt{B} \cdot \mathtt{K}$ are both integer multiples of 8 for DP data (or 4 for single precision values). The values of i and j are stored as offsets relative to the baseline element of the chunk.

For $\mathtt{B} \leq 256$, the row index is encoded using one byte only as most matrices contain at least one element per row. If this is not the case, each empty row is padded with an explicit zero element. If the whole chunk corresponds to a unique row, it is not necessary to store any value for the individual elements, and a regular sum reduction is used instead of the segmented scan. Depending on the nonzero pattern of the matrix, the column index is encoded using 8, 16, or 32 bits. For sparse matrices arising in non-graph applications, the nonzero entries in a row usually appear in clusters, allowing to use fewer bits to encode the column indices. While converting the matrix, a lookup table (LUT) is built containing the 256 most frequent values. If all value entries in the chunk are covered by the LUT, only one byte per element is used to index the right element in the LUT instead of storing the actual floating point values.

Figure 2 shows an example corresponding to a small chunk ($\mathtt{B} = 8$ and $\mathtt{K} = 1$) in compressed COO format. The original COO data is represented left of the arrow and the different elements in the compressed COO format on the right.

In this figure, each column from the blob corresponds to the respective original vector. The first column contains the (row) i indices as an offset to the row baseline. Similarly, the second column contains the (column) j indices as an offset to the column baseline. Finally, the third column contains an index to the LUT where the double precision values are stored. The values of the row and columns offsets are different for each chunk/element but the LUT is common to the whole matrix.

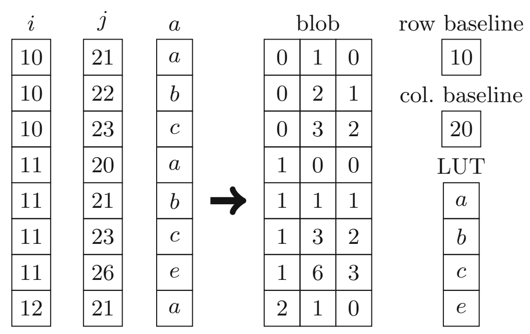

Fig. 2. Example of one chunk in Compressed COO format. In the last column, row/col. baseline specify the offset to be added to the first/second index of each block to obtain the corresponding i/j index; and LUT contains the different values encountered in vector a, which are indirectly referenced via the third entry of the blob.

3 Experimental Results

3.1 Setup and Memory Savings

For the experimental evaluation of the new compressed realization of SpMV, we selected 60 test matrices from the Suite Sparse Matrix Collection [1]. The chosen benchmarks have row/column dimensions larger than 900,000 and arise in a variety of scientific problems excluding graph applications. (Although the adjacency matrices associated with graphs have excellent compression properties, we do not consider them to be interesting use cases for the SpMV kernel as there are more efficient algorithms for graph manipulation.) The test matrices along with some key properties are listed in Table 2.

Figure 3 visualizes the memory overhead of COO and Compressed COO with respect to CSR, assuming a DP floating point representation for the numerical values with all three formats, and a 32-bit integer representation for the indices in CSR and COO. There are some matrices with clustered indices/repeated numerical entries where the compression schemes are especially efficient and, as a result, Compressed COO uses less memory than CSR. For the rest of the matrices, except in two cases, the overhead of Compressed COO over CSR is always smaller than that of regular COO.

Table 2. Test matrices

Matrix	n	n_z	n_z/n	Matrix	n	n_z	n_z/n
1. af_shell10	1,508,065	52,259,885	34.7	31. Geo_1438	1,437,960	60,236,322	41.9
2. atmosmodd	1,270,432	8,814,880	6.9	32. Hamrle3	1,447,360	5,514,242	3.8
3. atmosmodj	1,270,432	8,814,880	6.9	33. Hardesty1	938,905	12,143,314	12.9
4. atmosmodl	1,489,752	10,319,760	6.9	34. Hook_1498	1,498,023	59,374,451	39.6
5. atmosmodm	1,489,752	10,319,760	6.9	35. HV15R	2,017,169	283,073,458	140.3
6. audikw_1	943,695	77,651,847	82.3	36. kkt_power	2,063,494	12,771,361	6.2
7. bone010_M	986,703	23,888,775	24.2	37. ldoor	952,203	42,493,817	44.6
8. bone010	986,703	47,851,783	48.5	38. Long_Coup_dt0	1,470,152	84,422,970	57.4
9. boneS10_M	914,898	18,489,474	20.2	39. Long_Coup_dt6	1,470,152	84,422,970	57.4
10. boneS10	914,898	40,878,708	44.7	40. memchip	2,707,524	13,343,948	4.9
11. Bump_2911	2,911,419	127,729,899	43.9	41. ML_Geer	1,504,002	110,686,677	73.6
12. cage14	1,505,785	27,130,349	18.0	42. nlpkkt120	3,542,400	95,117,792	26.9
13. cage15	5,154,859	99,199,551	19.2	43. nlpkkt160	8,345,600	225,422,112	27.0
14. circuit5M_dc	3,523,317	14,865,409	4.2	44. nlpkkt200	16,240,000	440,225,632	27.1
15. circuit5M	5,558,326	59,524,291	10.7	45. nlpkkt240	27,993,600	760,648,352	27.2
16. Cube_Coup_dt0	2,164,760	124,406,070	57.5	46. nlpkkt80	1,062,400	28,192,672	26.5
17. Cube_Coup_dt6	2,164,760	124,406,070	57.5	47. nv2	1,453,908	37,475,646	25.8
18. CurlCurl_3	1,219,574	13,544,618	11.1	48. Queen_4147	4,147,110	316,548,962	76.3
19. dgreen	1,200,611	26,606,169	22.2	49. rajat31	4,690,002	20,316,253	4.3
20. dielFilterV2real	1,157,456	48,538,952	41.9	50. Serena	1,391,349	64,131,971	46.1
21. dielFilterV3real	1,102,824	89,306,020	81.0	51. ss	1,652,680	34,753,577	21.0
22. ecology1	1,000,000	4,996,000	5.0	52. StocF-1465	1,465,137	21,005,389	14.3
23. ecology2	999,999	4,995,991	5.0	53. stokes	11,449,533	349,321,980	30.5
24. Emilia_923	923,136	40,373,538	43.7	54. t2em	921,632	4,590,832	5.0
25. Flan_1565	1,564,794	114,165,372	73.0	55. thermal2	1,228,045	8,580,313	7.0
26. Freescale1	3,428,755	17,052,626	5.0	56. tmt_unsym	917,825	4,584,801	5.0
27. Freescale2	2,999,349	14,313,235	4.8	57. Transport	1,602,111	23,487,281	14.7
28. FullChip	2,987,012	26,621,983	8.9	58. vas_stokes_1M	1,090,664	34,767,207	31.9
29. CurlCurl_4	2,380,515	26,515,867	11.1	59. vas_stokes_2M	2,146,677	65,129,037	30.3
30. G3_circuit	1,585,478	7,660,826	4.8	60. vas_stokes_4M	4,382,246	131,577,616	30.0

We ran all the following experiments in this section using DP arithmetic on two distinct generations of NVIDIA accelerators:

– A V100 GPU with compute capability 7.0, furnished with 16 GB of main memory, 128 KB L1 cache per streaming processor, and 6 MB of L2 cache. The bandwidth to memory bandwidth is 900 GB/s and the theoretical peak performance is 7.8 DP TFLOPS.
– An A100 GPU with compute capability 8.0, equipped with 40 GB of memory, 1.5 GB/s main memory bandwidth, and a theoretical peak performance of 19.5/9.7 DP TFLOPS with/without DP tensor cores, respectively.

All the codes were compiled using CUDA version 11.0.167.

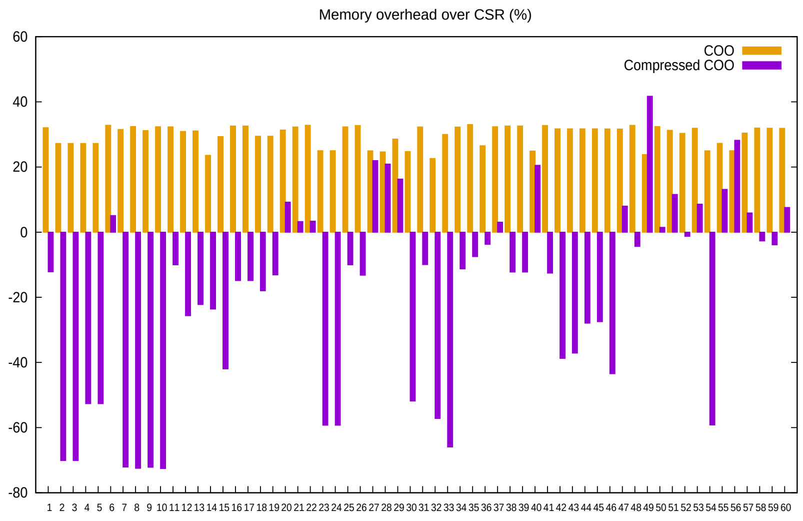

Fig. 3. Memory overhead with respect of the CSR format using DP arithmetic.

3.2 Performance of SpMV

We first compare the computational efficiency of our realization of SpMV against the codes in NVIDIA cuSPARSE. This native library from NVIDIA offers three routines for this computational kernel, two based on CSR and one based on COO. In the following comparisons, we include only the default CSR SpMV algorithm from cuSPARSE as the second CSR-based variant delivers very similar performance for the chosen test matrices. We do not include results for other formats, such as ELL or Hybrid-ELL, which were available in earlier versions of cuSPARSE but are no longer included in the last version of the library.

Figure 4 shows the performance evaluation of NVIDIA's codes against our Compressed COO implementation which applies the memory-reduction techniques described in Sect. 2 to diminish the indexing overhead for the row/column indices as well as data values. The results in the figure, in terms of GFLOPS, show a large performance improvement using Compressed COO for matrices with clustered indices/repeated values. Concretely, we are able to achieve up to 170/250 GFLOPS on the V100/A100 GPUs, respectively. While the compressed COO almost always outperforms the cuSPARSE COO and the cuSPARSE CSR kernels (except for a few outliers where the performance is on par or negligibly lower), the median speed-up over its competitors is 1.4× and 1.25–1.3× on the V100 and the A100 GPUs, respectively. Even though the median speed-up over cuSPARSE CSR and cuSPARSE COO is almost identical, we note that the performance ratios for the distinct problems are more consistent when comparing the COO formats.

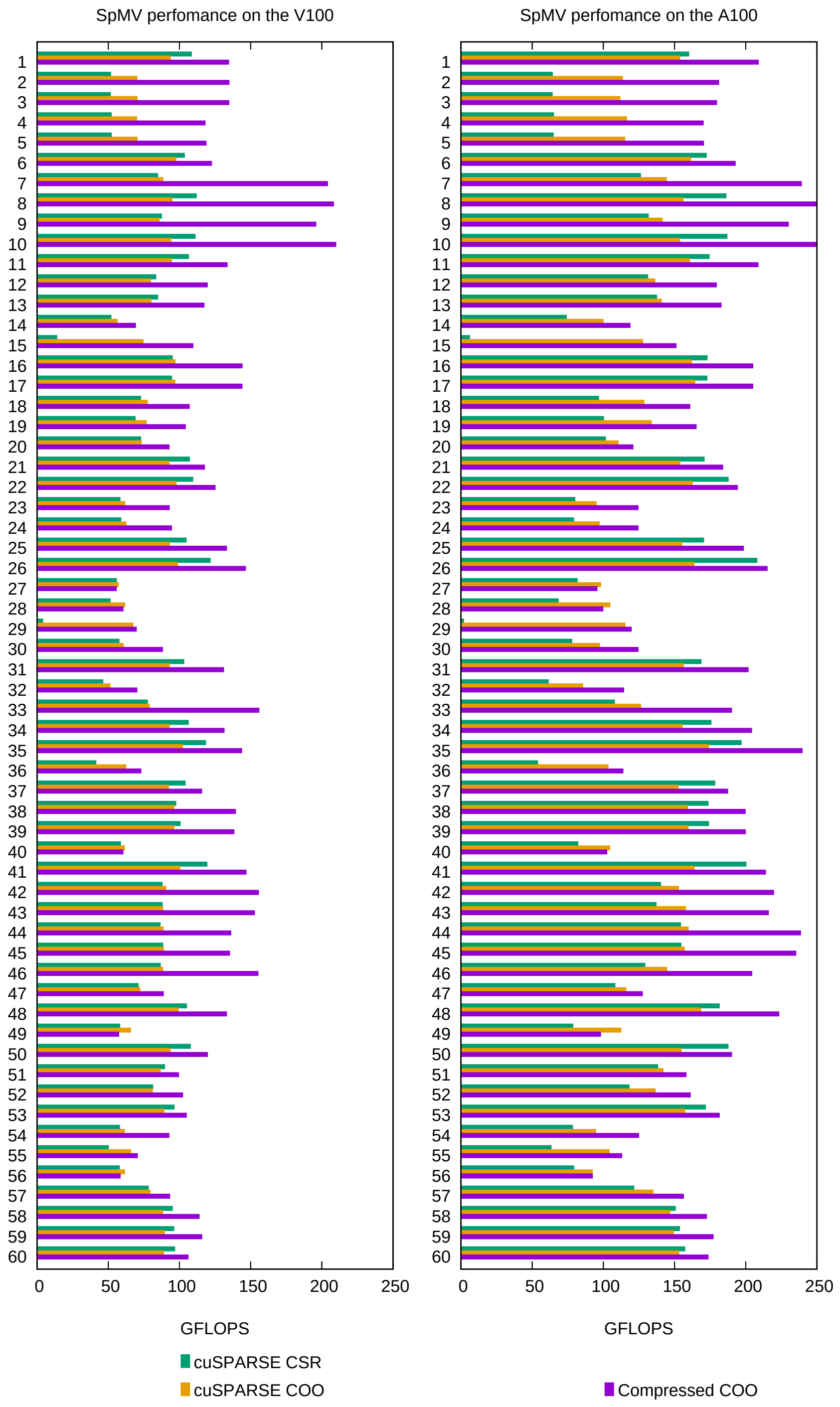

Fig. 4. Performance of the new compressed realization of SpMV against those in cuSPARSE on NVIDIA V100 and A100 GPUs (left and right, resp.)

3.3 Effect on BICGSTAB

We next evaluate the impact of the new compressed kernels for SpMV when integrated into an iterative solver for sparse linear systems based on a Krylov subspace method. For this purpose, we select a BICGSTAB implementation based on CUDA. In the comparison, the BiCGSTAB solver employs the three different SpMV realizations analyzed in the previous subsection: compressed COO, cuSPARSE CSR, and cuSPARSE COO. For a performance comparison, we execute a fixed number of iterations and measure the GFLOPS for the linear systems constructed from the same test cases selected for the standalone evaluation of SpMV.

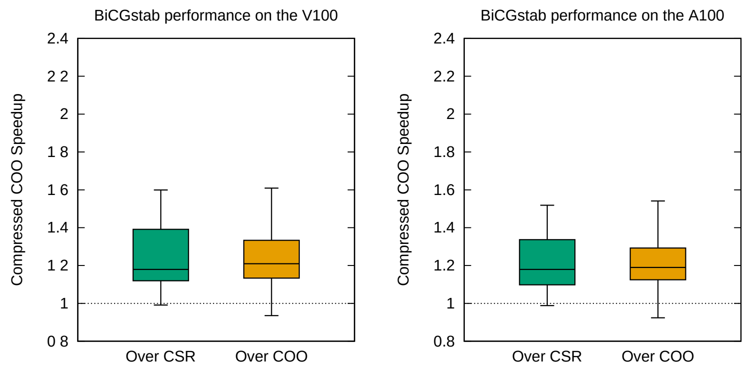

Fig. 5. Performance of the BICGSTAB solver with the new compressed realization of SpMV against those in cuSPARSE on NVIDIA V100 and A100 GPUs (left and right, resp.)

Our experiments with the BiCGSTAB solver using the three SpMV kernels show that the performance benefits of the faster SpMV kernel execution carry over to the BiCGSTAB solver. The acceleration of the BiCGSTAB solver depends on the specific problem and how much the SpMV kernel contributes to the overall runtime cost. In that sense, the speed-ups of BiCGSTAB correlate to a scaled version of the SpMV speed-up values reported in Fig. 4, damped with the problem-specific ratio between SpMV kernel cost vs. BiCGSTAB solver cost. In the end, equipping the BiCGSTAB solver with the compressed COO SpMV kernel improves the overall iterative solver performance for virtually all problems with a median speed-up of about 1.2× on both architectures, see Fig. 5.

4 Concluding Remarks and Future Work

We have adopted our previous balancing approach for SpMV to (virtually) divide the matrix contents into chunks (blocks) of nonzero entries of the same size; map these to the thread blocks; and prevent race conditions via efficient atomic operations. On top of this technique, in this work, we have proposed a new compression scheme that reduces the amount of indexing information that is associated with a COO-based realization of SpMV while maintaining the balanced distribution. For this purpose, the indices of each entry inside the same chunk are maintained as offsets with respect to a baseline row/column index pair, allowing the use of 8-bit encodings for the row indices, and 8/16/32-bit encodings for the column indices depending on the chunk. In addition, the observation that the numerical values in the sparse matrices arising in scientific applications present a considerable number of repetitions, motivates the design of a compression scheme that employs a look-up table.

The experimental results show the benefits of the new format, demonstrating a consistent advantage over the native implementation of the SpMV kernel in NVIDIA's cuSPARSE (CUDA 11.0.167) on the V100 and A100 GPUs.

The matrix format in this paper can be extended to support more efficient encodings. For example, matrix values could be stored in different precisions. Or even not stored at all for graph adjacency matrices that contain a large number of entries equal to one. Furthermore, the presented format is suitable for very large-scale matrices that require 64-bit indices.

Acknowledgements. This work was partially sponsored by the EU H2020 project 732631 OPRECOMP and project TIN2017-82972-R of the Spanish MINECO. Hartwig Anzt and Yuhsiang M. Tsai were supported by the "Impuls und Vernetzungsfond" of the Helmholtz Association under grant VH-NG-1241 and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. The authors would like to thank the Steinbuch Centre for Computing (SCC) of the Karlsruhe Institute of Technology for providing access to an NVIDIA A100 GPU.

References

1. Suitesparse matrix collection (2018). https://sparse.tamu.edu. Accessed Sept 2020
2. Bell, N., Garland, M.: Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical report NVR-2008-004, NVIDIA Corporation, December 2008
3. Buluç, A., Williams, S., Oliker, L., Demmel, J.: Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In: Proceedings of the IEEE International Parallel & Distributed Processing Symposium, pp. 721–733 (2011)
4. Choi, J.W., Singh, A., Vuduc, R.W.: Model-driven autotuning of sparse matrix-vector multiply on GPUs. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2010, pp. 115–126 (2010)
5. Filippone, S., Cardellini, V., Barbieri, D., Fanfarillo, A.: Sparse matrix-vector multiplication on GPGPUs. ACM Trans. Math. Softw. **43**(4), 1–49 (2017)
6. Flegar, G., Anzt, H.: Overcoming load imbalance for irregular sparse matrices. In: Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms, IA3 2017 (2017)
7. Flegar, G., Quintana-Ortí, E.S.: Balanced CSR sparse matrix-vector product on graphics processors. In: Rivera, F.F., Pena, T.F., Cabaleiro, J.C. (eds.) Euro-Par 2017. LNCS, vol. 10417, pp. 697–709. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-64203-1_50

8. Grossman, M., Thiele, C., Araya-Polo, M., Frank, F., Alpak, F.O., Sarkar, V.: A survey of sparse matrix-vector multiplication performance on large matrices. CoRR abs/1608.00636 (2016). http://arxiv.org/abs/1608.00636
9. Liu, W., Vinter, B.: CSR5: an efficient storage format for cross-platform sparse matrix-vector multiplication. In: Proceedings of the 29th ACM on International Conference on Supercomputing, ICS 2015, pp. 339–350. ACM, New York (2015)
10. Saad, Y.: Iterative Methods for Sparse Linear Systems, 2nd edn. SIAM, Philadelphia (2003)