CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2021; **00**:1–15
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Compression and Load Balancing for Efficient Sparse Matrix-Vector Product on Multicore Processors and GPUs

José I. Aliaga[1], Hartwig Anzt[2,3], Thomas Grützmacher[2], 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.*
{aliaga,tomasan}@uji.es
[2] *Karlsruhe Institute of Technology, Karlsruhe, Germany.*
{hartwig.anzt,thomas.gruetzmacher}@kit.edu
[3]*Innovative Computing Lab, University of Tennessee, Knoxville (TN), USA*
[4]*Depto. de Informática de Sistemas y Computadores, Universitat Politècnica de València, Valencia, Spain.*
quintana@disca.upv.es
[5]*Depto. de Informática, Universitat de València, Valencia, Spain.*

SUMMARY

We contribute to the optimization of the sparse matrix-vector product by introducing a variant of the coordinate sparse matrix format that balances the workload distribution and compresses both the indexing arrays and the numerical information. Our approach is multi-platform, in the sense that the realizations for (general-purpose) multicore processors as well as graphics accelerators (GPUs) are built upon common principles, but differ in the implementation details, which are adapted to avoid thread divergence in the GPU case or maximize compression element-wise (that is, for each matrix entry) for multicore architectures.
Our evaluation on the two last generations of NVIDIA GPUs as well as Intel and AMD processors demonstrate the benefits of the new kernels when compared with the optimized implementations of the sparse matrix-vector product in NVIDIA's cuSPARSE and Intel's MKL, respectively. Copyright © 2021 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Sparse matrix-vector product; coordinate sparse matrix format; graphics processing units (GPUs); multicore processors (CPUs); workload balancing; compression

1. INTRODUCTION

The sparse matrix-vector product (SpMV) dictates, to a large extent, the performance of a considerable variety of scientific applications. This computational kernel reflects how a discretized linear operator acts on a vector, and therewith plays a central role in the iterative solution of linear systems and eigenvalue problems. Some of the most popular methods that rely on the repetitive application of the SpMV kernel include Krylov subspace solvers, such as the Conjugate Gradient (CG), GMRES, or BiCGSTAB methods [1], and the PageRank algorithm based on the Power Iteration [2]. Therefore, it is not surprising that a significant effort has been devoted to accelerate the execution of this kernel, following the evolution of computer architectures over the past decades.

The SpMV kernel involves a low number of floating-point operations (flops) per memory access. As a result, on current processor architectures, with very fast and wide SIMD (single-instruction, multiple-data) floating-point units but in comparison slow main memory access, even the presence

*Correspondence to: Andrés E. Tomás, Depto. de Ingeniería y Ciencia de Computadores, Universitat Jaume I, Castellón de la Plana, Spain. tomasan@uji.es

Copyright © 2021 John Wiley & Sons, Ltd.

of a deep hierarchy of memory caches cannot avoid that the speed of SpMV is mostly determined by that of the main memory bandwidth. In consequence, the SpMV kernel is far from delivering the full peak flop performance of current processors [3, 4, 5, 6, 7, 8]. Furthermore, the dispersion of the nonzero entries across the sparse matrix results in an irregular memory access pattern, making cache prefetching very difficult and reducing the cache hit ratio [9, 10]. Finally, the irregular sparsity distribution poses a challenge not only to data prefetching but also to achieving a balanced parallel workload distribution. To tackle these difficulties, the optimization of the SpMV routine for a wide range of applications and/or computer architectures tries to intertwine two strategies: 1) given the memory-bound nature of the SpMV kernel, the sparse matrix layout focuses on minimizing the memory footprint of the matrix and therewith the volume of data retrieved from memory; and 2) the operation on the sparse matrix layout aims at balancing the workload across the parallel compute resources[†]. In theory, these two goals are independent, and a strategy that parallelizes the SpMV kernel attaining a fair load balancing can operate on different sparse matrix layouts. However, certain load balancing strategies require information that only some sparse matrix layouts supply. For example, balancing across nonzero elements is easy if each nonzero value is accompanied by its location information (row and column indices), but is much harder if the location information is not available or has to be reconstructed [11]. Furthermore, efficient prefetching and data reuse require the interplay between matrix storage format and parallel processing scheme. In summary, these two goals – memory access minimization via data reuse and efficient parallel execution – cannot be pursued independently, as the selection of a sparse matrix layout induces which parallelization and work balancing schemes are amenable.

Over the years, a large number of application-specific sparse matrix layouts have been proposed [3, 4, 5, 6, 7, 8]. These specialized sparse data structures along with the optimized SpMV kernels often deliver high performance for a target application problem, but generally perform poorly and/or require expensive transformations of the matrix format for other applications. At the other end of the spectrum, for application-independent sparse matrix manipulation and data exchange, the coordinate sparse matrix format (COO) and the compressed sparse row format (CSR) have established themselves as de facto standard [12]. Furthermore, significant efforts have been spent on developing fast SpMV kernels operating on these two formats, and relevant vendor math libraries, such as NVIDIA's cuSPARSE and Intel's MKL, provide architecture-optimized versions of CSR-based and COO-based kernels for the SpMV. Aside from architecture-specific implementations, the development of cross-platform SpMV kernels operating in the COO and CSR formats is also a topic of active research.

In [13] we targeted the optimization of SpMV on GPUs by introducing a variant of the COO format that compresses the integer representation of the matrix indices. In addition, we employed a look-up table (LUT) to avoid the storage of repeated numerical values in the sparse matrix. The realization of the SpMV kernel in that work built upon a workload balancing parallelization scheme for GPUs optimized for matrices with an irregular row distribution of the non-zero entries for CSR and COO [14, 15], inheriting this appealing property. In this paper, we demonstrate that the ideas developed in our previous work are multi-platform, in the sense that they carry over to multicore CPUs as well. In particular, the present work makes the following contributions:

- We abstract the techniques in [14, 15] from their GPU implementation, obtaining an architecture-oblivious generalization of the approach which divides the nonzero entries of the sparse matrix into chunks of equal size. The CPU-specific implementation then distributes the execution of these chunks dynamically across the processor cores, using OpenMP, to improve load balancing while taking care of potential race conditions via atomic updates.
- Furthermore, our multicore variant of the compressed COO format interleaves the indexing information with the numerical values in a very compact format that utilizes between 2 and 13 bytes per matrix entry compared with the 16 bytes per entry for the standard version of COO (assuming 64-bit double-precision values and 32-bit integer indices for the latter format).

[†]An SpMV kernel designed for a sequential execution does not need to account for load balancing, and can therefore exclusively focus on optimizing data compression.

Copyright © 2021 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

- We detail the multicore variant of the compressed COO matrix format and discuss the similarities and differences to the many-core variant of the compressed COO matrix format. We also present in detail the multicore implementation of the SPMV kernel and its parallelization via OpenMP.
- We provide a comprehensive performance analysis of the new realization of SPMV, using a collection of 56 large-scale test matrices from the *Suite Sparse Matrix Collection* [16], and the two last generations of NVIDIA GPUs (Volta and Ampere) as well as two servers respectively equipped with Intel and AMD multicore processors and a large number of cores.

Our idea to compress the indexing information to reduce the pressure on memory bandwidth is shared with other approaches, such as the compressed sparse blocks (CSB) format [4]. This layout 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 multicore processors and GPUs; and we also explore the compression of the numerical data using a LUT.

The rest of the paper is structured as follows: In Section 2, after a short introduction of the CSR and COO sparse matrix formats, we describe the chunk-based multi-threaded parallelization of our approach and the compressed sparse matrix layout we propose for multicore CPUs. In Section 3, we provide a complete experimental evaluation of the new GPU and CPU kernels, both from the points of view of memory requirements and parallel performance. Finally, we close the paper in Section 4 with a summary of the paper and a discussion of future work.

2. SPMV OPTIMIZATION FOR MULTICORE AND MANY-CORE PROCESSORS

COO and CSR are two of the most popular data formats for storing sparse matrices [12]. The former format only stores the numerical values of the matrix nonzero entries along with their explicit column and row indices. The latter reduces the memory footprint of the COO format by replacing the row indices with pointers to the first element in each row of a row-wise sorted matrix.

The balanced and compressed COO SPMV that we presented in [13] builds upon the COO format, reducing the memory footprint by compressing the row–column indexing information. Also, it uses a problem-specific Look Up Table (LUT) for encoding the most frequent numerical values of the matrix. The SPMV kernel operating on this compressed COO format distributes the nonzeros across the parallel resources to favor workload balancing. A kernel using this parallelization strategy was previously identified to provide good performance on GPUs when operating on a standard CSR format [11] or the standard COO format [17]; however, it needs to be enhanced with an on-the-fly decompression of the row–column indices when operating on the compressed COO format. Performance benefits may still be available if the decompression overhead is compensated by the reduction of memory transfers. In the remainder of the paper, we will use the notation "balanced COO" (BCOO) when referring to the balanced SPMV kernel operating on the standard COO format; and "balanced and compressed COO" (BCCOO) when referring to the balanced SPMV kernel operating on the compressed COO format.

In [13], we detailed the implementation, compression, and parallelization of the BCCOO SPMV for many-core GPUs. In this paper, we focus the description on the compression strategy and the parallelization strategy when the target architecture is a multicore CPU, discussing the differences between the multicore implementation and the many-core implementation.

2.1. *Compressed COO format for Multicore Architectures*

Given an $n \times n$ sparse matrix A, with n_z non-zero entries, the COO format stores this matrix using three vectors: say a, i and j, each of dimension n_z, that contain the numerical values of the non-zero elements of the matrix, and their row and column index coordinates, respectively.

Copyright © 2021 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

Besides, we consider a partitioning of the COO sparse matrix into *chunks* of b consecutive nonzero entries, and we assume that these are sorted by increasing order of row index. For simplicity, we assume that n_z is an integer multiple of b in the rest of the section.

The compressed COO data structure maintains the information in compressed form. Specifically, it employs the following four vector arrays to store the nonzero entries of the matrix:

`lut` is a look-up table consisting of an array with the 256 most frequent (nonzero) numerical values appearing in the sparse matrix, each represented using a 64-bit IEEE floating-point value (FP64). A single byte index is thus sufficient to reference a specific entry of `lut`.

`row` is an array of size n_z/b that stores the row index of the first entry of the chunks using a 32-bit integer (INT32) per chunk. As our schemes target the iterative solution of sparse linear systems, we assume that all matrix rows contain at least a nonzero entry (as the matrix would be singular otherwise).

`offset` is also an array of n_z/b INT32 elements, each specifying the position of the first entry of the corresponding chunk in the fourth array.

`data` is an array of $n_z + n$ *tuples*, of variable length each, that contain the information that is described next.

Let us consider a tuple t consisting of the three "'terms" (i_t, j_t, k_t). The first term, i_t, always occupies a single byte and provides the key to decode the information in the other two terms. Specifically:

- The most significant bit of i_t specifies whether the term k_t contains a full FP64 value (and, therefore, occupies 8 bytes) or, alternatively, is a 1-byte index into the `lut` from which the FP64 value can be retrieved.
- The remaining 7 bits of i_t (together with the term j_t) encode the following information, depending on the value of i_t being equal to:

  0xFF or 0x7F ("end-of-row"): The partial values accumulated by the thread processing this row, as part of the partial product between this row and vector x, are accumulated on the corresponding entry of y. The "current row" index is increased by one. The remaining two terms of the tuple are void.

  0xFE or 0x7E: The column index of this entry is specified as an absolute value in j_t taking 4 bytes.

  0xFD or 0x7D: The column index of this entry is encoded in j_t, using 2 bytes, as the difference with respect to the column index of the previous entry (differential encoding).

  Any other: The term i_t itself is the difference with the previous column index, while the term j_t is void.

The key to the compression of the numerical values and indices thus lies in the `data` array, which stores each nonzero value and its corresponding row–column indices in a very compact tuple, of variable length between 1 and 13 bytes. In comparison, the standard COO format requires 4+4 bytes for the row+column indices (an INT32 each) plus 8 bytes for the numerical value (an FP64 number), for a total of 16 bytes per element. The cost of compressing the indexing is negligible if the matrix is already in CSR format; the cost of computing the `lut` is $O(n_z \log n_z)$, and is faster than a vector sorting because it involves less data movement.

2.2. *Parallelization of the COO* SPMV

Consider the SPMV $y := A \cdot x$, where both vectors x, y comprise n components. A trivial parallelization of the COO-based SPMV on a GPU dedicates one thread to multiply a single nonzero entry of the matrix with the corresponding entry of x, using an *atomic operation* to accumulate the

Copyright © 2021 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

partial result on the appropriate component of y. In practice, the performance of this initial GPU-parallel approach can be improved by instructing each thread to multiply 2 or 4 elements of the sparse matrix A with those of the input vector x to update one or more entries of the solution y [13].

The fine-grained mapping of work to threads used in the GPU implementation is far from optimal for a multicore CPU due to the significant differences between the two architectures in terms of the number and functionality of the hardware cores, the distinct overhead due to thread creation/switching, and the cache hierarchy. The general idea to parallelize the COO SPMV is to distribute the computations among the CPU threads with a coarser granularity.

Consider a partitioning of the sparse matrix by blocks of rows, so that one CPU thread is in charge of performing all the operations that are necessary to multiply the elements of a row block of A with those of x, updating the corresponding entries of y. This scheme yields an embarrassingly-parallel algorithm, where each thread can independently compute a block of the solution vector y. However, the scheme easily leads to workload imbalance, due to differences in the number of nonzero entries per matrix row. The solution proposed in [14] for CSR and in [15] for COO, in both cases with a GPU as the target architecture, is to divide the matrix into blocks of consecutive nonzero entries; and to deal with race conditions via the efficient hardware support for atomic updates in recent GPUs. As we describe next, the same idea carries over with some modifications to a multicore CPU.

In detail, when the target architecture is a multicore processor, we propose to partition the sparse matrix, stored in COO format, into *chunks* of b consecutive nonzero entries, and we instruct the OpenMP runtime to distribute the operations associated with each chunk among the CPU threads, via an OpenMP `#pragma omp parallel for` directive. In comparison with the row-wise distribution, this alternative produces a theoretically optimal partitioning of the workload. However, variations in the actual execution time of each chunk are still possible due to cache misses, leading to a potential workload imbalance. In our approach, this is addressed via the selection of a chuck size b that is small in comparison with n_z, so that the default static OpenMP mapping obtains a good balancing of chunks to threads while avoiding the overhead of a dynamic schedule.

The partitioning and parallelization strategy described in the last paragraph requires careful control of race conditions, via atomic updates, because it can lead to simultaneous updates to the same entry of the solution y by more than one thread. The overhead introduced by atomic updates on the CPU can be reduced by ensuring that the sparse matrix entries are arranged in increasing row index. (Note that this is the same arrangement of the nonzero entries required by the CSR format. Therefore, assembling the COO data structure from the CSR counterpart does not require a re-organization of the non-zero entries.) The reason is that, for this variant of COO a specialized ordering of the rows, race conditions due to update collisions can only occur for the first and last row comprised in each chunk. In contrast, the updates for the solution entries associated with all other rows can proceed in parallel. This scheme thus benefits from chunks that span several rows.

2.3. *Balanced and Compressed COO SPMV for Multicore CPUs*

The BCCOO SPMV kernel for multicore CPUs uses the parallelization strategy presented in Section 2.2 to operate on the compressed COO format presented in Section 2.1. For simplicity, in Listing 1 we only show the code that processes the first row of the chunk, requiring an atomic update of the corresponding entry in vector y. The code for the remaining rows is similar. The four arrays that represent the sparse matrix A (`lut`, `row`, `offset`, and `data`) are accessed via a `struct` named `coo`, which is passed as a pointer to the routine. The `for` loop indexed by variable `h` is parallelized with an OpenMP `#pragma omp parallel for` and processes all the matrix chunks. The `while` loop processes all entries within the current chunk.

While the realizations of SPMV for multicore CPUs and many-core GPUs (see [13] for the latter) share the same ideas in execution strategies in terms of workload partitioning and compression, there exist some relevant differences in the implementation:

- In the GPU realization each thread block processes a single chunk. To prevent thread divergence, all the chunk tuples are encoded in the same way, yielding a less compact format.
- In the GPU case, we employ two auxiliary arrays, `format` and `column`, both of size n_z/b. The entries of the first array indicate whether the nonzero matrix elements in the chunk are

Copyright © 2021 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

```
void COO_Comp_SpMV(struct COO_compressed *coo, double *x, double *y)
{
    for (int i = 0; i < coo->n; i++) y[i] = 0.0;

    // Process all chunks: 0,1,2,...,nb-1=(nz/b)-1
    #pragma omp parallel for
    for (int h = 0; h < coo->nb; h++) {
        long   k = coo->offset[h]; // First entry of this chunk in data
        int    r = coo->row[h];    // Row index of first entry in this chunk
        int    c = 0;   // Default column index of first entry in this chunk
        double s = 0.0; // Partial accumulation

        // Process first row. Special case as it requires atomic update
        while (k < coo->offset[h+1]) {
            // Process tuple t = (i_t,j_t,k_t)
            uint8_t i_t = coo->data[k]; k++; // Term i_t in the tuple

            if (i_t == 0xFF) break; // end of row, exit loop
            switch (i_t & 0x7F) {
                case 0x7E:
                    // Column index is in j_t (4 bytes, absolute index)
                    c = *(uint32_t *)(coo->data + k); k += 4; break;
                case 0x7D:
                    // Column index is in j_t (2 bytes, relative index)
                    c += *(uint16_t *)(coo->data + k); k += 2; break;
                default:
                    // Column index is in i_t (1 byte, relative index)
                    c += i_t & 0x7F;
            }

            if (i_t & 0x80) {
                // Term k_t is the LUT entry with the value
                s += coo->lut[coo->data[k]] * x[c]; k++;
            } else {
                // Value is in term k_t
                s += *(double *)(coo->data + k) * x[c]; k += 8;
            }
         }
         #pragma omp atomic
         y[r] += s;

         // Remaining rows very similar. Omitted for brevity
     }
}
```

Listing 1: Simplified realization of the multi-threaded, OpenMP-based code for SPMV.

stored as an FP64 or a byte (this is, an index to `lut`) and whether the rows of the chunk are encoded using a byte, a 16-bit integer, or an INT32. The second array specifies the smallest column index for all nonzero entries in the chunk.

- The first byte in the array `data` does not inform of the tuple encoding. Instead, it directly encodes the row index (using a single byte in all cases).

3. EXPERIMENTAL PERFORMANCE ANALYSIS

3.1. Hardware and Experimental Setup

For the experimental evaluation of the BCCOO SPMV kernel, we selected 56 test matrices from the Suite Sparse Matrix Collection [16]. 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

Copyright © 2021 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

Table I. Test matrices

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

consider them to be interesting use cases for the SPMV kernel as there are more efficient algorithms for graph manipulation.) The test matrices are listed along with some key properties in Table I. The chunk size was set to 1,024 for both CPU and GPU.

For the experimental evaluation on GPUs, we used two different architectures:

- An NVIDIA A100 ("Ampere") GPU with compute capability 8.0, equipped with 40 GB of memory. This architecture features a bandwidth of 1,555 GB/s to main memory, and a theoretical peak performance of 9.7 DP (double-precision) TFLOPS (10^{12} flops/second) using only the CUDA cores.
- An NVIDIA V100 ("Volta") GPU with compute capability 7.0, furnished with 16 GB of main memory. The bandwidth to main memory is 900 GB/s and the theoretical peak performance is 7.8 DP TFLOPS using only the CUDA cores.

In both cases, the many-core version of the balanced and compressed SPMV kernel was compiled with CUDA version 11.0.167. In the GPU performance evaluation, we compare against the SPMV kernels available in NVIDIA's cuSPARSE library version 11.0.167. As our GPU SPMV kernels run only on the accelerator, the characteristics of the server are not relevant.

For the experimental evaluation on CPUs, we selected the following two platforms:

- A system with one AMD EPYC 7351P processor (16 cores, "Zen" microarchitecture) and 16 GB of RAM. The codes are compiled with Intel's `icc` and MKL 2021.1.
- A system with two Intel Xeon Gold 6230 processors (10 cores per socket, "Skylake" microarchitecture), and 96 GB of RAM. The codes are compiled with Intel `icc` version 19.1 and linked with Intel's MKL 2020.

The SPMV operation is a memory-bound kernel and increasing the number of cores augments the memory bandwidth only to the point of saturation while the capacity of the local memory caches grows linearly with the number of cores, which may increase the performance. The exact threshold for memory saturation depends on the sparse problem: in particular, "denser" problems can be expected to reach that threshold with a smaller number of cores.

Copyright © 2021 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

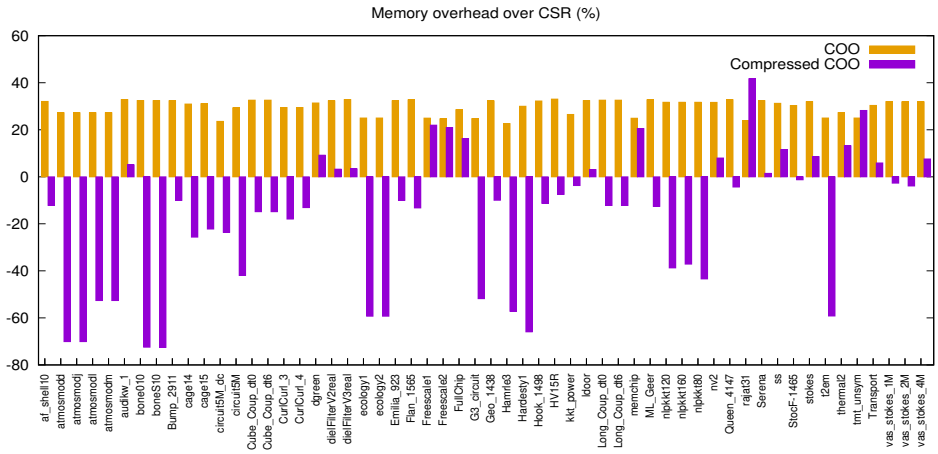

Figure 1. Memory overhead of the many-core GPU version of the compressed COO formats taking the CSR format as baseline. All data is assumed to be stored in DP.

3.2. GPU Performance Analysis

For convenience, we revisit in this subsection the performance that is achieved with our many-core implementation of the BCCOO SpMV on GPUs; see [13] for additional details.

Before doing so, we recall that the performance of SpMV kernels is generally limited by the memory bandwidth. Therefore, the memory footprint of a particular sparse matrix layout can be a good indicator of the performance one can expect from an SpMV kernel based on that sparse matrix format. In Figure 1, we report the memory footprint of the compressed COO realization specifically designed for many-core accelerators relative to the memory footprint of the standard CSR format. For reference, the figure also includes the memory footprint of the standard COO format. In some cases, storing the matrix in compressed COO format can provide attractive memory savings over the standard CSR format while, in other cases, the CSR format provides a more compact storage. Compared with the standard COO format (which always presents a larger memory footprint than the CSR format), the compression renders memory savings for about 2/3 of the problems, while it increases the memory footprint by up to 20% for some matrices.

An aspect not captured by an analysis that is only focused on memory footprint is that COO-based layouts, like the compressed COO format, may allow for better load balancing across the parallel execution units. The reason for this is that balancing the workload in the CSR format generally requires the algorithm to recover the row indices on-the-fly, as those are not stored explicitly [11]. In Figure 2 we offer the results of the performance analysis on the NVIDIA V100 and A100 GPUs. We show the performance of the BCCOO SpMV kernel alongside those of the CSR and COO SpMV kernels available in the NVIDIA cuSPARSE library. The performance results reveal that the BCCOO SpMV kernel outperforms the COO SpMV from NVIDIA's cuSPARSE library for all test matrices except two cases (`rajat31` and `tmt_unsym`) on both GPU architectures. This could be expected as (precisely except for these two cases, see Figure 1,) the compression scheme reduces the volume of data retrieved from main memory, and therewith the execution time, while accommodating an advanced load balancing technique.In comparison with the CSR format, the compressed COO format reduces the memory footprint for less than half of the test cases; see Figure 1. Nevertheless, the performance results reveal that the CSR SpMV kernel in NVIDIA's cuSPARSE is scarcely competitive to the BCCOO SpMV kernel. Specifically, on both architectures, there exist only very few cases where NVIDIA's CSR SpMV achieves higher performance, and even in those cases, the performance advantage is only a few percent. At the same

Copyright © 2021 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

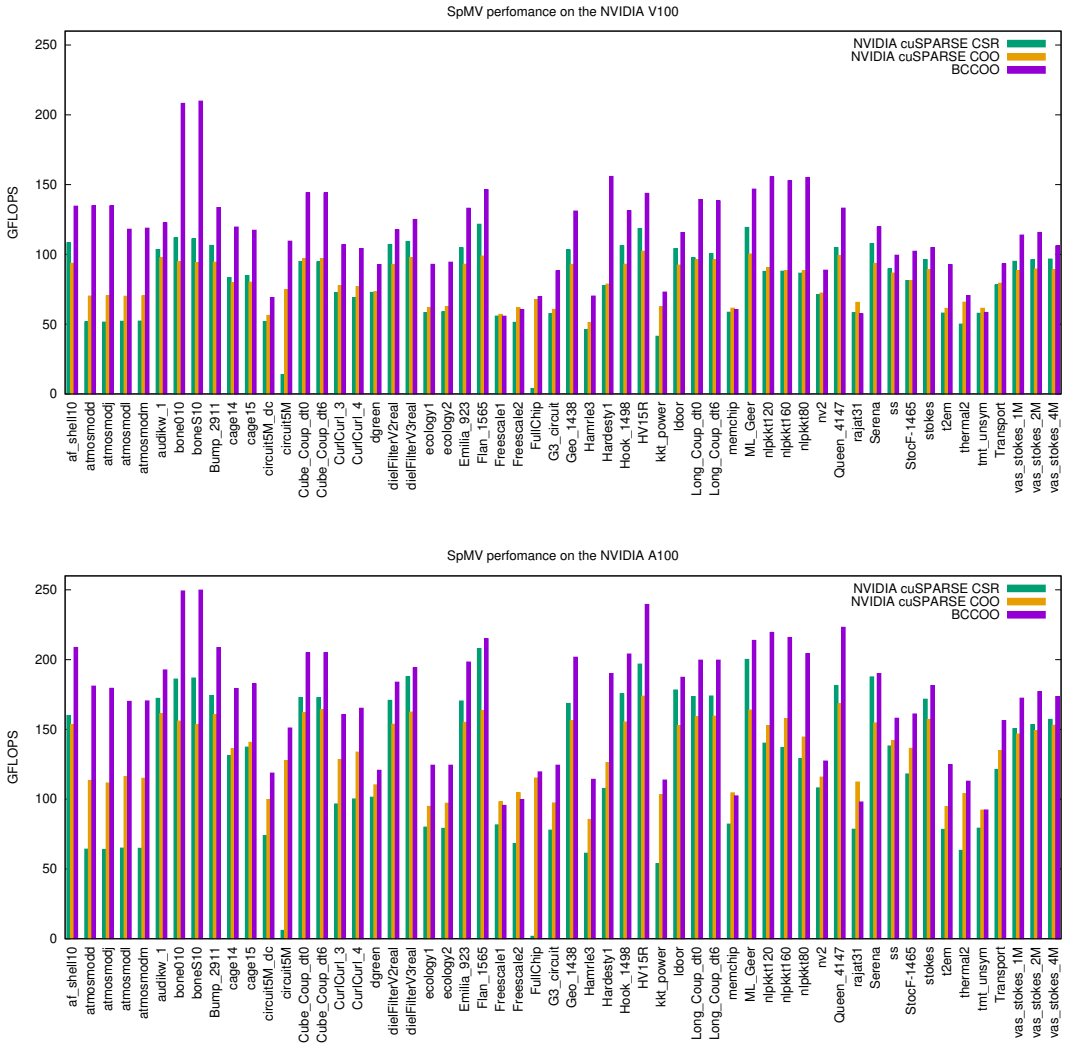

Figure 2. Performance of the BCCOO SpMV in comparison to the cuSPARSE COO SpMV and the cuSPARSE CSR SpMV on NVIDIA V100 (top) and A100 (bottom) GPUs.

time, for most cases, the balanced and compressed SpMV significantly outperforms NVIDIA's CSR SpMV, offering an average speedup of 20% across all test cases.

3.3. CPU Performance Analysis

In Section 2, we discussed the common design principles and realization differences between the compression of the COO format for multicore CPUs and many-core GPUs. Broadly speaking, the compression for CPUs can be more aggressive as it does not require a uniform compression for all values of the same data chunk to avoid thread divergence – the multicore implementation compresses the values element-wise. In Figure 3, we report the memory footprint of the specific compressed COO realization for multicore architectures relative to that of the standard CSR format. Like in the GPU case, storing the matrix in (the conventional) COO increases the memory footprint over the CSR format for all test cases. At the same time, and unlike the many-core version of the compressed COO shown in Figure 1, the multicore version of the compressed COO exhibits a smaller memory footprint than the CSR format for all test cases. A direct comparison between the many-core variant of compressed COO in Figure 1 and the multicore variant of compressed COO

Copyright © 2021 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

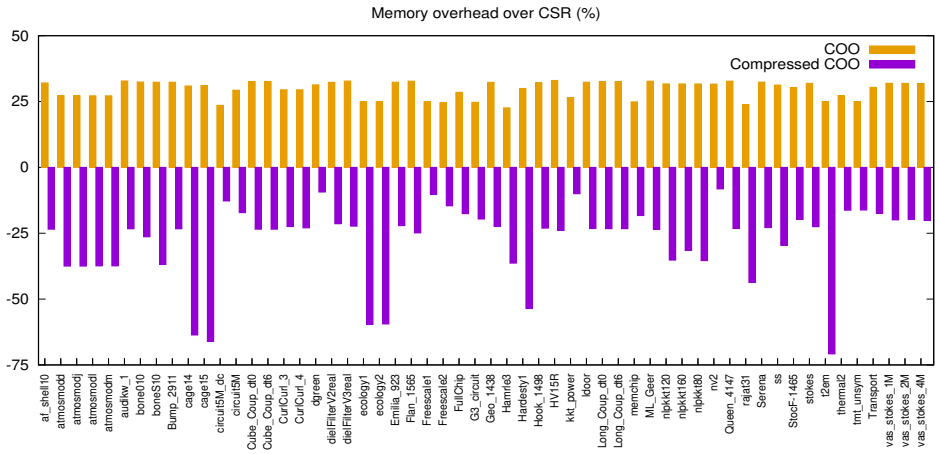

Figure 3. Memory overhead of the multicore CPU version of the compressed COO format taking the CSR format as baseline. All data is assumed to be stored in DP.

in Figure 3 indicates that the multicore version of the compressed COO succeeds in reducing the memory footprint more aggressively.

Next, we analyze the performance of the multicore version of the BCCOO SpMV kernel. For this purpose, we compare this realization against the CSR SpMV kernel available in Intel's MKL library as well as a BCOO SpMV kernel that is furnished with the same parallelization strategy as the BCCOO SpMV kernel, but features no compression. Intel's MKL kernel for the COO SpMV is not considered in the analysis because it is not parallelized, and achieves the same performance for all experimental configurations. In Figure 4 and Figure 5, we show the performance characteristics of the distinct SpMV kernels, for the EPYC- and Xeon-based platforms using a single core, all cores of a single socket (16 or 10 cores), and the full server in the case of the Intel platform (two sockets, 20 cores), with a perfect one-thread-per-core binding in both servers. The MKL CSR SpMV achieves on average an attractive 1.2× / 3.8× speedup when running on a single socket of the EPYC / Xeon server, and a 4.9× speedup when running on the full server for the latter platform. The BCOO SpMV kernel achieves a 1.3× / 3.3× speedup when running 16/10 cores of a socket of the EPYC / Xeon server. However, the parallel efficiency decreases to a speedup of 3.8× when running on 20 cores of the Xeon server as the kernel then hits the "memory wall": the addition of arithmetic power fails to increase the performance further as the memory access limits the kernel speed. In contrast, the BCCOO SpMV kernel suffers from the decompression overhead when running on a single core (resulting in lower performance than the BCOO SpMV), but exhibits better scalability as it reduces the pressure from the memory bandwidth, and can thus fully exploit the additional arithmetic resources. The BCCOO SpMV kernel achieves a 6.3× / 11.9× speedup when running on a single socket of the EPYC / Xeon server, and an 21.3× speedup when running on the full server for the latter platform. This super-linear speedup for BCCOO on Xeon server is due to the enlarged cache available when running on multiple cores and the use of two threads per core (hyperthreading). Comparing the absolute performances, the BCOO SpMV kernel outperforms the BCCOO kernel on average by 50% / 30% on a single core, but the BCCOO SpMV kernel is 2.0× / 3.3× faster when running on 16 / 20 cores of the EPYC / Xeon servers. These speedups are exclusively due to the reduced memory access enabled by data compression. Comparing the performance of all SpMV kernels, we recognize that MKL's CSR SpMV is the overall winner for small core counts. This may be expected as 1) the sequential SpMV execution does not have to account for load balancing but exclusively focuses on minimizing the memory access and arithmetic operations; and 2) when running on a few cores only, the SpMV kernel does not saturate the memory bandwidth, and the additional arithmetic necessary to decompress the indexing information cannot

Copyright © 2021 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

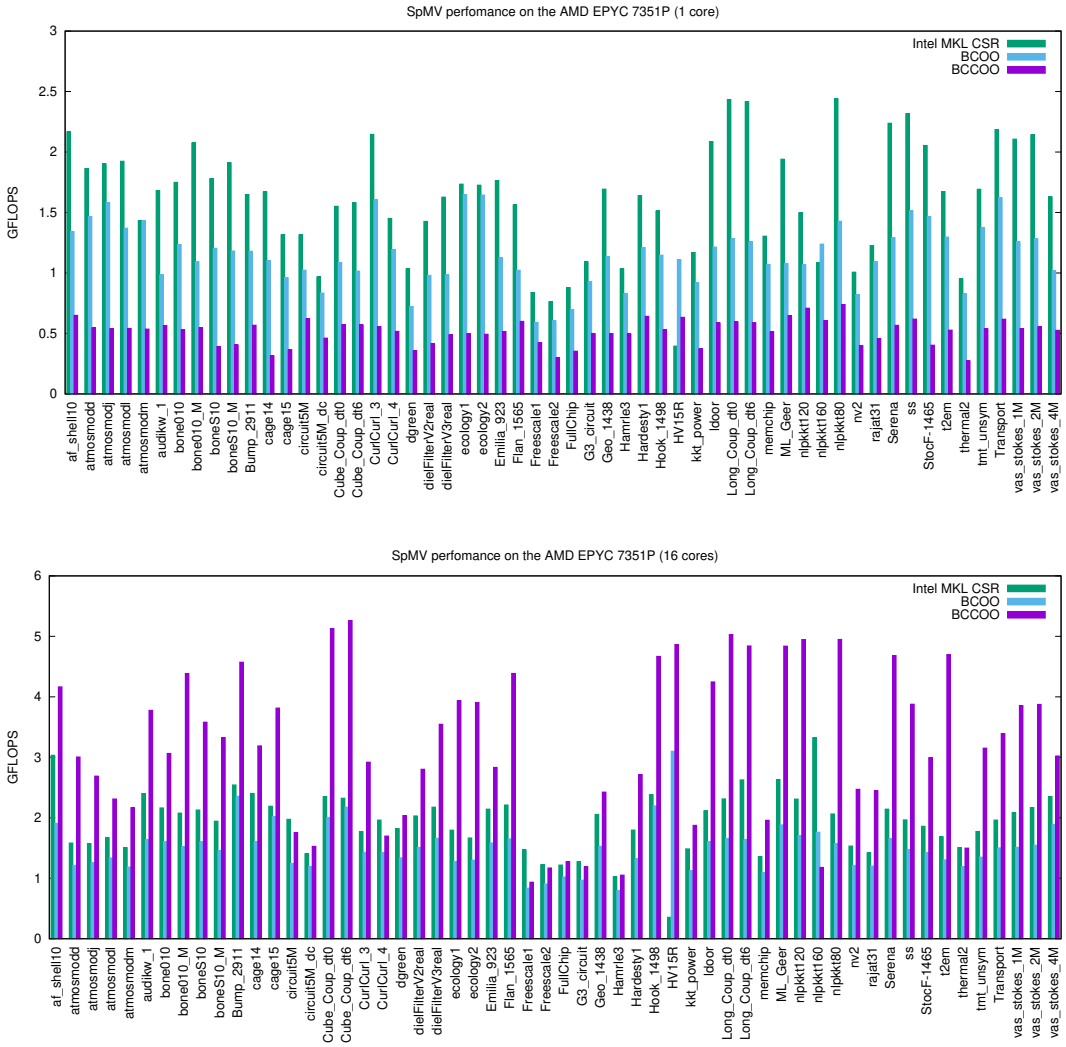

Figure 4. Performance of the BCCOO SPMV in comparison with Intel's MKL COO SPMV and Intel's MKL CSR SPMV using 1 core (top) and the full server (1 socket and 16 cores, bottom) of the AMD EPYC 7351P platform.

be completely hidden behind memory accesses, thus reducing the performance of the floating-point units. On a single socket, the BCCOO SPMV is competitive with the MKL CSR SPMV, and when operating with the full server, the BCCOO SPMV regularly outperforms the MKL CSR SPMV.

3.4. Speedup Analysis

Finally, we quantify the speedup that the BCCOO SPMV kernel can render over the SPMV kernels in the vendor libraries. For this, in Figure 6 we display the speedup that BCCOO achieves over NVIDIA's cuSPARSE library on the V100 and A100 GPUs. We observe that, on both architectures, the BCCOO SPMV executes on average of 1.4× faster than the CSR and COO SPMV kernels available in NVIDIA's cuSPARSE library. In Figure 7 and Figure 8, we present a similar analysis comparing the BCCOO SPMV against the MKL CSR SPMV and the BCOO SPMV using

Copyright © 2021 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

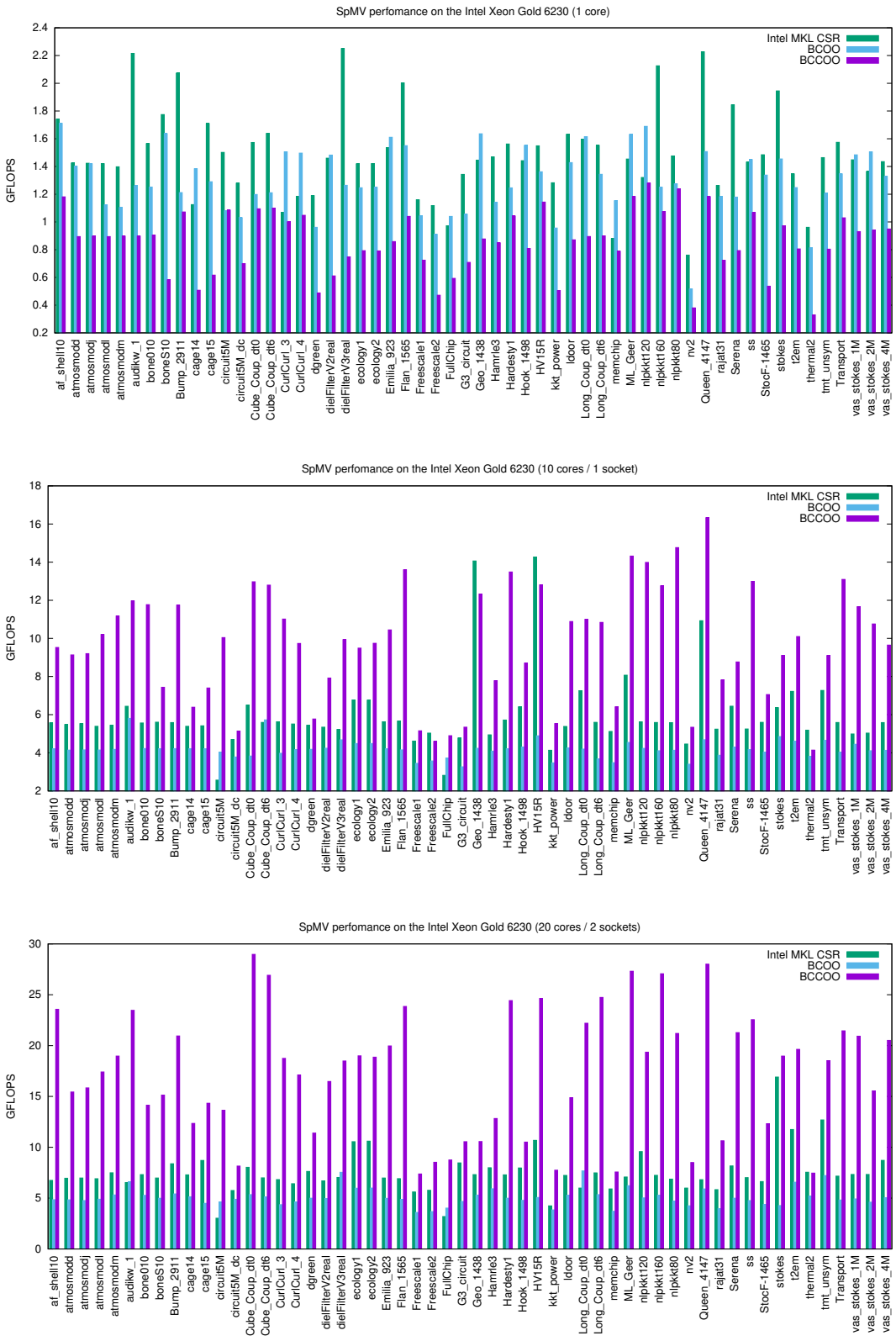

Figure 5. Performance of the BCCOO SPMV in comparison with Intel's MKL COO SPMV and Intel's MKL CSR SPMV using 1 core (top), a single socket (10 cores, center), and the full server (2 sockets and 20 cores, bottom) of the Intel Xeon Gold 6230 platform.

Copyright © 2021 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

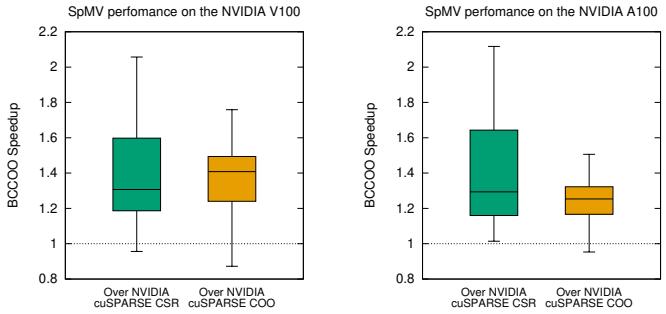

Figure 6. Speedup analysis comparing the performance of the BCCOO SpMV against the COO SpMV and the CSR SpMV from NVIDIA's cuSPARSE library on the NVIDIA V100 (left) and NVIDIA A100 (right).

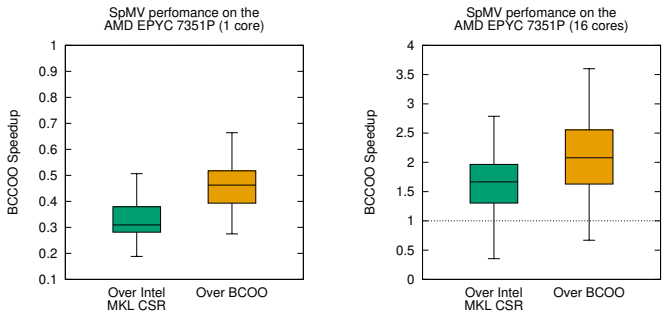

Figure 7. Speedup analysis comparing the performance of the BCCOO SpMV against Intel's CSR SpMV and the BCOO SpMV on the AMD EPYC 7351P.

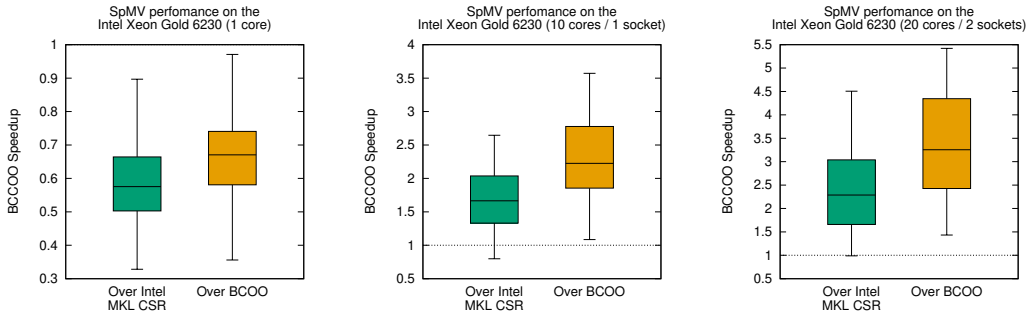

Figure 8. Speedup analysis comparing the performance of the BCCOO SpMV against Intel's CSR SpMV and the BCOO SpMV on the Intel Xeon Gold 6230.

different core counts of the Intel and AMD systems.[‡] On average, when running on a single core of the EPYC platform, BCCOO is almost three times slower than Intel's CSR SpMV kernel, but 70% faster when the kernels are executed using the full socket. Similarly, on the Xeon architecture, when using only a single core, Intel's CSR SpMV is twice faster than BCCOO, but on the same architecture, BCCOO outperforms Intel's CSR SpMV when both kernels are executed using all 20 cores of the server.

[‡]We do not compare against the COO SpMV available in Intel's MKL library as this kernel is not parallelized.

Copyright © 2021 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

4. CONCLUDING REMARKS AND FUTURE WORK

We have proposed an SpMV kernel that combines load-balancing with compression to reduce the pressure on memory while using the available compute power of modern CPUs and GPUs efficiently. We have developed kernel realizations for the most recent many-core GPUs from NVIDIA as well as server-line multi-core processors from Intel and AMD with a large number of cores. While the implementations for these two types of systems present some architecture-dependent differences, tuned to optimize performance, the scheme is based on a few key common ideas:

- The workload is distributed by partitioning the (nonzero) matrix entries into chunks of small size (in practice, about 1,024 for both GPU and CPU), which are then mapped to the threads by the runtime (CUDA for NVIDIA and OpenMP for Intel and AMD). Race conditions are avoided via atomic updates and an arrangement of the matrix rows to minimize collisions. We thus prioritize obtaining a fair workload distribution over other embarrassingly parallel row-wise oriented alternatives.
- The indexing overhead is reduced by aggressively compacting the COO indices into 1, 2, or 4 bytes for the columns, and as little as a few bits for the rows.
- The numerical information is compacted as well by moving the most frequent numerical values to a Look Up Table and then using a 1-byte reference as a pointer into the table.

The global combination of these ideas yields high performance realizations of the SpMV kernel that consistently outperform the vendor-optimized native kernels in the case of NVIDIA's GPUs and provide higher scalability in the case of Intel and AMD multicore architectures. This provides a strong demonstration that 1) optimizing for workload distribution is crucial for irregular matrix computations; and 2) the cost of decompressing data structures can be largely amortized for memory-bound algorithms.

As part of future work, we plan to investigate the extension of some of the ideas presented in this paper to the CSR sparse matrix format as well as the NUMA-aware optimization of the SpMV kernel. We also plan to migrate the GPUs realizations to AMD GPUs.

ACKNOWLEDGEMENT

J. I. Aliaga, E. S. Quintana-Ortí, and A. E. Tomás were supported TIN2017-82972-R of the Spanish MINECO. H. Anzt and T. Grützmacher 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. Saad Y, Schultz MH. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.* July 1986; **7**:856–869, doi:10.1137/0907058. URL http://portal.acm.org/citation.cfm?id=14063.14074.
2. Langville A, Meyer C. *Google's PageRank and Beyond: The Science of Search Engine Rankings*. Princeton University Press, 2009. URL http://books.google.es/books?id=hxvB14-I0twC.
3. Bell N, Garland M. Efficient sparse matrix-vector multiplication on CUDA. *NVIDIA Technical Report NVR-2008-004*, NVIDIA Corporation Dec 2008.
4. Buluç A, Williams S, Oliker L, Demmel J. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. *Proc. IEEE Int. Parallel & Distributed Processing Symposium*, 2011; 721–733, doi:10.1109/IPDPS.2011.73.
5. Filippone S, Cardellini V, Barbieri D, Fanfarillo A. Sparse matrix-vector multiplication on GPGPUs. *ACM Trans. Math. Softw.* 2017; **43**(4), doi:10.1145/3017994.
6. Choi JW, Singh A, Vuduc RW. Model-driven autotuning of sparse matrix-vector multiply on GPUs. *Proc. 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, PPoPP '10, 2010; 115–126, doi:10.1145/1693453.1693471.
7. 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* 2016; **abs/1608.00636**. URL http://arxiv.org/abs/1608.00636.

Copyright © 2021 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

8. Liu W, Vinter B. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. *Proc. 29th ACM on International Conference on Supercomputing*, ICS '15, ACM: New York, NY, USA, 2015; 339–350, doi:10.1145/2751205.2751209.
9. Mittal S. A survey of recent prefetching techniques for processor caches. *ACM Comput. Surv.* Aug 2016; **49**(2), doi:10.1145/2907071. URL https://doi.org/10.1145/2907071.
10. Chen TF, Baer JL. Effective hardware-based data prefetching for high-performance processors. *IEEE Trans. on Computers* 1995; **44**(5):609–623, doi:10.1109/12.381947.
11. Flegar G, Quintana-Ortí ES. Balanced CSR sparse matrix-vector product on graphics processors. *Euro-Par 2017: Parallel Processing - 23rd International Conference on Parallel and Distributed Computing, Santiago de Compostela, Spain, August 28 - September 1, 2017, Proceedings*, *Lecture Notes in Computer Science*, vol. 10417, Rivera FF, Pena TF, Cabaleiro JC (eds.), Springer, 2017; 697–709, doi:10.1007/978-3-319-64203-1_50. URL https://doi.org/10.1007/978-3-319-64203-1_50.
12. Barrett R, Berry M, Chan TF, Demmel J, Donato J, Dongarra J, Eijkhout V, Pozo R, Romine C, der Vorst HV. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM: Philadelphia, PA, 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. *Lecture Notes in Computer Science, 18th Int. Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms – HeteroPar'20*, Springer, 2020. To appear.
14. Flegar G, Quintana-Ortí ES. Balanced CSR sparse matrix-vector product on graphics processors. *Euro-Par 2017: Parallel Processing*, Springer Int. Pub., 2017; 697–709, doi:10.1007/978-3-319-64203-1_50.
15. Flegar G, Anzt H. Overcoming load imbalance for irregular sparse matrices. *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms*, IA3'17, 2017, doi:10.1145/3149704.3149767.
16. Suitesparse matrix collection. https://sparse.tamu.edu 2018. Accessed in September 2020.
17. Grützmacher T, Anzt H. A modular precision format for decoupling arithmetic format and storage format. *Euro-Par 2018: Parallel Processing Workshops*, Mencagli G, B Heras D, Cardellini V, Casalicchio E, Jeannot E, Wolf F, Salis A, Schifanella C, Manumachu RR, Ricci L, *et al.* (eds.), Springer International Publishing: Cham, 2019; 434–443.

Copyright © 2021 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls