Load-balancing Sparse Matrix Vector Product Kernels on GPUs

2

HARTWIG ANZT, Karlsruhe Institute of Technology, Germany and University of Tennessee
TERRY COJEAN, Karlsruhe Institute of Technology, Germany
YEN-CHEN CHEN, National Taiwan University, Taiwan
JACK DONGARRA, University of Tennessee, Oak Ridge National Lab, and University of Manchester, UK
GORAN FLEGAR, University of Jaume I, Spain
PRATIK NAYAK, Karlsruhe Institute of Technology, Germany
STANIMIRE TOMOV, University of Tennessee
YUHSIANG M. TSAI and WEICHUNG WANG, National Taiwan University, Taiwan

Efficient processing of Irregular Matrices on Single Instruction, Multiple Data (SIMD)-type architectures is a persistent challenge. Resolving it requires innovations in the development of data formats, computational techniques, and implementations that strike a balance between thread divergence, which is inherent for Irregular Matrices, and padding, which alleviates the performance-detrimental thread divergence but introduces artificial overheads. To this end, in this article, we address the challenge of designing high performance sparse matrix-vector product (SpMV) kernels designed for Nvidia Graphics Processing Units (GPUs). We present a compressed sparse row (CSR) format suitable for unbalanced matrices. We also provide a load-balancing kernel for the coordinate (COO) matrix format and extend it to a hybrid algorithm that stores part of the matrix in SIMD-friendly Ellpack format (ELL) format. The ratio between the ELL- and the COO-part is determined using a theoretical analysis of the nonzeros-per-row distribution. For the over 2,800 test matrices available in the Suite Sparse matrix collection, we compare the performance against SpMV kernels provided by NVIDIA's cuSPARSE library and a heavily-tuned sliced ELL (SELL-P) kernel that prevents unnecessary padding by considering the irregular matrices as a combination of matrix blocks stored in ELL format.

CCS Concepts: • **Mathematics of computing → Computations on matrices**; • **Computing methodologies → Massively parallel algorithms**;

Additional Key Words and Phrases: Sparse Matrix Vector Product (SpMV), irregular matrices, GPUs

This work was supported by the "Impuls und Vernetzungsfond" of the Helmholtz Association under grant VH-NG-1241, and the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Awards Number DE-SC0016513 and DE-SC-0010042.

Authors' addresses: H. Anzt, Karlsruhe Institute of Technology, Germany and University of Tennessee, USA; email: hanzt@icl.utk.edu; T. Cojean, Karlsruhe Institute of Technology, Germany; email: terry.cojean@kit.edu; Y.-C. Chen, University of Tokyo, Japan; email: yenchen_chen@mist.i.u-tokyo.ac.jp, J. Dongarra, University of Tennessee, USA and Oak Ridge National Lab, USA and University of Manchester, UK; email: dongarra@icl.utk.edu; G. Flegar, University of Jaume I, Spain; email: flegar.goran@gmail.com; P. Nayak, Karlsruhe Institute of Technology, Germany; email: pratik.nayak@kit.edu; S. Tomov, University of Tennessee, USA; email: tomov@cs.utk.edu; Y. M. Tsai, Karlsruhe Institute of Technology, Germany; email: yu-hsiang.tsai@kit.edu; W. Wang, National Taiwan University, Taiwan; email: wwangntu@gmail.com.

2329-4949/2020/03-ART2 $15.00
https://doi.org/10.1145/3380930

ACM Reference format:
Hartwig Anzt, Terry Cojean, Yen-Chen Chen, Jack Dongarra, Goran Flegar, Pratik Nayak, Stanimire Tomov, Yuhsiang M. Tsai, and Weichung Wang. 2020. Load-balancing Sparse Matrix Vector Product Kernels on GPUs. *ACM Trans. Parallel Comput.* 7, 1, Article 2 (March 2020), 26 pages.
https://doi.org/10.1145/3380930

1 INTRODUCTION

Applying a discretized operator in terms of a sparse matrix-vector product (SpMV) is a heavily-used operation in many scientific applications. An example is the power iteration [Corso 1997], an iterative algorithm to identify the largest eigenpair of a sparse matrix that forms the backbone of Google's Page Rank algorithm [Langville and Meyer 2012]. At the same time, the SpMV is the performance bottleneck for many applications and problem settings. In particular, for irregular sparsity patterns, the SpMV executed on parallel hardware is notorious for sustaining low fractions of peak processor performance. This is partly due to the SpMV's low arithmetic intensity, making the SpMV kernel memory bound on virtually all modern architectures, as well as on the access overhead induced by storing only the nonzero elements in the matrix, and the irregular (in many cases random) access to the input vector. Given the importance of the matrix-vector product in a large range of computing applications, significant effort is spent on finding the best way to store sparse matrices and optimizing the SpMV kernel for different nonzero patterns and hardware architectures [Grossman et al. 2016; Liu and Vinter 2015]. For sparse matrices, where the nonzeros are distributed in a very structured fashion, it is often possible to derive problem-tailored storage formats, e.g., the diagonal format (DIA) format for matrices with a tridiagonal structure [Bell and Garland 2009]. A similar situation occurs if the pattern is not very structured, but the nonzero elements are distributed equally across the rows (each row contains a similar number of nonzero elements). Most challenging are the sparsity patterns that are irregular (no recurring sub-pattern can be identified) and unbalanced (the distinct rows have a very different number of nonzero elements). Problems with these characteristics are typical, e.g., in social network representations. For these irregular problems, standard parallelization strategies, like assigning rows to the parallel resources, inevitably result in heavy load imbalance. At the same time, Graphics Processing Unit (GPU)-friendly parallelization strategies like padding the rows to an equal number of nonzero elements and processing the rows in Single Instruction, Multiple Data (SIMD) fashion [Bell and Garland 2009] result in significant memory and computational overhead. Additionally, unstructured sparsity patterns often result in random memory access to the vector values.

Different strategies exist to overcome the load imbalance for irregular matrices and to design SpMV kernels that allow for efficient processing of these problems on SIMD architectures.

One approach is the sliced Ellpack format (ELL) (SELL-P [Anzt et al. 2014]) format that decomposes the irregular matrix into row blocks and handles each row block in the SIMD-friendly ELL (packed) format. Compared to the standard ELL format, SELL-P can efficiently reduce the memory and computational overhead as the padding is no longer determined by the largest nonzeros-per-row value of the complete matrix but of the matrix block [Kreutzer et al. 2014]. While setting the block size to one prevents any overhead (at the price of increased load imbalance), reordering the matrix rows according to the nonzeros-per-row metric and using moderate block sizes can efficiently reduce the overhead while maintaining an SIMD-attractive execution pattern [Kreutzer et al. 2014].

A second approach is to enhance the Compressed Sparse Row (CSR) SpMV kernel with a strategy that splits the work of rows containing many nonzero elements across multiple processing units. This alleviates workload imbalance and improves the performance of the CSR SpMV for imbalanced matrices.

A third approach is to parallelize the SpMV kernel not across rows but across nonzero elements. An example is the load-balancing SpMV kernel based on the coordinate (COO) format [Bell and Garland 2009] that leverages the latest features of the Compute Unified Device Architecture (CUDA) programming model and succeeds in achieving high performance for irregular matrices [Flegar and Anzt 2017].

Somewhat orthogonal is the idea of splitting the SpMV operation into a regular (balanced) contribution of the sparse matrix and an irregular (unbalanced) contribution, and to invoke two separate SpMV kernels: an ELL kernel efficiently handling the regular contribution in a SIMD-fashion, and a load-balancing COO kernel for the irregular part. The resulting "hybrid" SpMV algorithm introduces the kernel launch overhead of the second kernel, but combines the advantages of both SpMV kernel processing schemes. The challenge in this case is how to partition the data into the "regular" and the "irregular" parts.

In this article, we present and compare these four strategies (SELL-P, COO, CSR, and HYB) and their realization on GPUs in detail. All kernels presented are integrated into the open-source Ginkgo library,[1] a modern C++ library originally designed for the iterative solution of sparse linear systems. Given the long list of efforts covering the design and evaluation of SpMV kernels on manycore processors, e.g., see Filippone et al. [2017] for a recent and comprehensive overview of SpMV research, we highlight that this work contains the following novel contributions:

- We complement the recently developed load-balancing COO SpMV [Flegar and Anzt 2017] with the first publicly available production-ready kernel implementation and an extensive performance analysis on NVIDIA's latest Volta architecture, as well as a comparison against the COO SpMV available in NVIDIA's cuSPARSE library;
- Based on a new ELL SpMV kernel, we design a load-balancing SELL-P SpMV kernel and evaluate its performance;
- We provide a CSR SpMV kernel strategy combining both the classical CSR SpMV and a load-balancing SpMV kernel;
- We use a theoretical analysis focusing on the memory footprint to derive an optimal threshold that determines the ELL and COO contributions in the hybrid SpMV algorithm. We also assess the efficiency of the theoretical bound and compare the performance against NVIDIA's hybrid SpMV;
- Up to our knowledge, Ginkgo is the first open source sparse linear algebra library based on C++ that features multiple SpMV kernels suitable for irregular matrices. We present Ginkgo's SpMV capabilities and provide usage examples for the distinct SpMV kernels;
- Ginkgo's sustainable software development cycle features continuous benchmarking (CB) on high-end GPU architectures [Anzt et al. 2019]. The automated SpMV performance analysis covers all matrices available in the Suite Sparse matrix collection [SuiteSparse 2018], providing a comprehensive picture of the efficiency of the distinct load-balancing strategies. The performance can be investigated interactively through the Ginkgo Performance Explorer (GPE [Anzt et al. 2019]), which is employed for the analysis of the distinct SpMV kernels.

Before providing more details about the load-balancing sparse matrix formats and the processing strategy of the related SpMV routines in Section 3, we recall some basic sparse matrix formats and general considerations for parallelizing the SpMV kernel in Section 2. In Section 4, we combine several basic matrix storage formats into so-called "hybrid" formats that combine several basic formats to increase efficiency. In a comprehensive evaluation in Section 5, we compare the

[1] https://ginkgo-project.github.io.

performance of the presented kernels with the SpMV routines available in NVIDIA's cuSPARSE library. As we make matrix formats and SpMV kernels available in the Ginkgo open source library, we devote Section 6 to sustainable software development as a community effort and the Ginkgo Performance Explorer, an interactive performance assessment tool that also allows external developers to benchmark contributions on high-end High Performance Computing (HPC) systems. We conclude in Section 7 with a summary of the observations and an outlook on future work.

2 BASIC SPARSE MATRIX FORMATS

In the Basic Linear Algebra Subprogram (BLAS) and Linear Algebra PACKage (LAPACK) [Anderson et al. 1990] standard for dense linear algebra, matrices are stored as a sequence of columns, with each column stored as an array of its elements [Flegar and Anzt 2017]. This allows to easily locate or identify any matrix entry in memory. For matrices where most elements are zero, which is typical in many cases, e.g., finite element discretizations, this approach is not appropriate as storing all matrix entries results in significant storage overhead, and a computational kernel considering explicit zero elements introduces significant computational overhead.

In response, sparse matrix formats were developed that aim at reducing the memory footprint (and computational cost) by storing only the nonzero matrix values [Barrett et al. 1994]. Some formats additionally store a moderate amount of zero elements to enable faster processing when computing matrix-vector products [Bell and Garland 2009]. Obviously, storing only a subset of the elements requires to accompany these values with information that allows to deduce their location in the original matrix.

A straight-forward idea is to accompany the nonzero elements with the respective row and column indexes. This storage format, known as COO [Barrett et al. 1994] format, allows to determine the original position of any element in the matrix without processing other entries.

Further reduction of the storage cost is possible if the elements are sorted row-wise, and with increasing column-order in every row. (The latter assumption is technically not required, but it usually results in better performance.) Then, the CSR [Barrett et al. 1994] format can replace the array containing the row indexes with a pointer to the beginning of the distinct rows. While this generally reduces the data volume, the CSR format requires extra processing to determine the row location of a certain element.

The performance of a particular SpMV implementation on a specific hardware reflects the complex interactions of different aspects—(1) the volume of matrix sparsity pattern information (in general, indexes); (2) the volume of matrix and vector numeric data (that is, the values); (3) the irregular access to the vector values (determined by the matrix sparsity pattern) and its interaction with the cache configuration and sizes, and the code access order; and (4) the workload imbalance in a parallelized setting. All these aspects have to be considered when optimizing an SpMV kernel for a specific hardware architecture.

SIMD architectures require to have uniform operations across the SIMD unit. In response, the ELL format [Bell and Garland 2009] compresses the original (full) matrix to contain only the nonzero entries and some explicit zeros that are used for padding to enforce an equal length for all rows. The resulting value matrix is accompanied with a column index matrix that stores the column position of each entry in the compressed matrix. While typically increasing the storage cost compared to the CSR format (except for the case of perfectly balanced nonzero distributions), this removes the need for explicitly storing the row pointers. Furthermore, the column indexes (and values) in the distinct rows can be processed in branching-free SIMD fashion, which makes the format attractive for manycore accelerators like GPUs. Coalescent (SIMD-friendly) memory access is enabled if the value and column index matrices are stored in column-major order.

We illustrate the three basic formats discussed in this section in Figure 1.

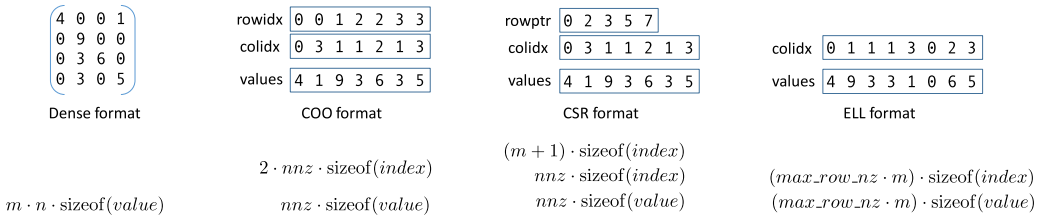

Fig. 1. Different storage formats for a sparse matrix of dimension $m \times n$ containing n_z nonzeros along with the memory consumption [Flegar and Anzt 2017].

```
1  // input: A, x, y
2  // output: y = y+A*x
3  void coo_spmv(int nnz, const int *rowidx,
4                const int *colidx, const float *val
5                const float *x, float *y)
6  {
7    for (int i = 0; i < nnz; ++i) {
8      y[rowidx[i]] += val[i] * x[colidx[i]];
9    }
10 }
```

Fig. 2. COO SpMV kernel design.

Related to the storage format is the question of how to process the multiplication with a vector in parallel. The main challenges in this context are (1) balancing the workload among the distinct cores/threads and (2) allowing for efficient access to the matrix entries and the vector values. The second aspect is relevant in particular on NVIDIA GPUs where each memory access reads 32, 64, or 128 contiguous bytes of memory [NVIDIA Corporation 2018]. In case of fine-grained parallelism, balancing the workload naturally results in multiple threads computing partial sums for one row, which requires careful synchronization when writing the resulting vector entry back into main memory [Flegar and Anzt 2017; Flegar and Quintana-Ortí 2017].

The standard approach of parallelizing the SpMV kernels in the CSR, COO, and ELL formats is to distribute the rows among distinct threads (or groups of threads) [Bell and Garland 2009; Monakov et al. 2010]. For the CSR format, this works fairly well for balanced sparsity patterns, but it can lead to severe load imbalance otherwise. Recently, a strategy for a load balanced CSR SpMV was proposed that parallelizes across the nonzero elements instead of across the rows [Flegar and Quintana-Ortí 2017]. The SpMV kernel we present in this article is based on the COO format, which comes with the advantage of the row index of an element being readily available.

The optimization of the SpMV kernel for GPUs remains a topic of major interest [Dalton et al. 2015; Hong et al. 2019; Merrill and Garland 2016] given the wide-spread use of sparse matrix vector products, e.g., in iterative solvers for linear systems [Anzt et al. 2017] and data analytics [Page et al. 1998]. Many algorithms have been developed to tackle different problem structures, with the prefix-sum computation [Merrill et al. 2015] and the intra-warp communication [Hong et al. 2011] being among the most relevant strategies to balance operations on the CSR sparse matrix data format.

3 SPARSE MATRIX VECTOR KERNEL DESIGNS

3.1 Basic COO SpMV Kernel

In this section, we first present the SpMV kernel's sequential code before moving toward format specific parallelization and optimizations. The SpMV kernel computes $y := A \cdot x + y$ for $A \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$. A more general operation $y := \alpha A \cdot x + \beta y$ can easily be derived from this kernel by scaling y prior to the operation (i.e., scaling y with 0 to compute $y = A \cdot x$) [Flegar and Anzt 2017]. Figure 2 shows the algorithm for the COO sparse matrix representation format. For COO,

the kernel has three arrays as input, each of size n_z, the number of nonzero elements of the matrix. Two arrays use integer values for row and column indexes, *rowidx* and *colidx*, respectively, and one array stores the values, namely *val*. The kernel loops over all nonzero elements of the matrix in line 7. Depending on the format, this loop could access rows or columns one after the other, rather than access element after element. The core operation of COO SpMV is in line 8 of Figure 2. It is composed of a (fused) multiply-add between an element of A, and elements of x and y, indexed by the values in arrays *rowidx* and *colidx*. The element of y is then updated with the result of this operation.

3.2 Balancing COO SpMV Kernel

A central idea of the load-balancing COO SpMV proposed in Flegar et al. [2017] is to parallelize across the nonzero elements, not the rows. This strategy, however, requires atomic operations to synchronize the partial sums computed by different threads. The scope of the COO SpMV is not limited to a specific hardware, but can be adapted for any architecture that satisfies the following constraints: (1) The architecture has a large number of computational elements (e.g., cores); (2) All computational elements share the same main memory; (3) It is possible to atomically add floating point values in main memory from all computational elements; (4) The memory design favors data locality over random data access. These properties are fulfilled by virtually all modern shared memory manycore systems, particularly by newer NVIDIA GPU architectures [NVIDIA Corporation 2018]. In particular, double precision atomic operations are natively supported since NVIDIA's Pascal micro-architecture.

ALGORITHM 1: Load-balancing COOkernel algorithm.

```
1:  Get ind index of the first element to be processed by this thread
2:  Get current_row = rowidx[ind].
3:  Compute the first value c = A[ind] × x[colidx[ind]]
4:  for i = 1 .. nz_per_chunk; i+ = warpsize do
5:      Compute next_row, row index of the next element to be processed
6:      if any thread in the warp's next_row != current_row or it is the final iteration then
7:          Compute the segmented scan according to current_row.
8:          if first thread in segment then
9:              atomic_add c on output vector by the first entry of each segment
10:         end if
11:         Reinitialize c = 0
12:     end if
13:     Get the next index ind
14:     Compute c+ = A[ind] × x[colidx[ind]]
15:     Update current_row to next_row
16: end for
```

3.2.1 Algorithm. The algorithm for the load-balancing COO SpMV kernel is shown in Algorithm 1 and Figure 3. As previously stated, the kernel parallelizes across nonzero elements of the problem matrix A. Similar-sized and contiguous chunks of nonzero elements are assigned to different compute units. For CUDA parallelization, a group of 32 threads (one "warp") processes one chunk in SIMD fashion. While it is possible to use non-contiguous chunks (e.g., distribute the data in round-robin fashion), this potentially results in lower data locality causing a performance loss. To ensure load balancing, all chunks are of similar size. Furthermore, to reduce the number of transactions from main memory, it is important to enable coalesced memory access.

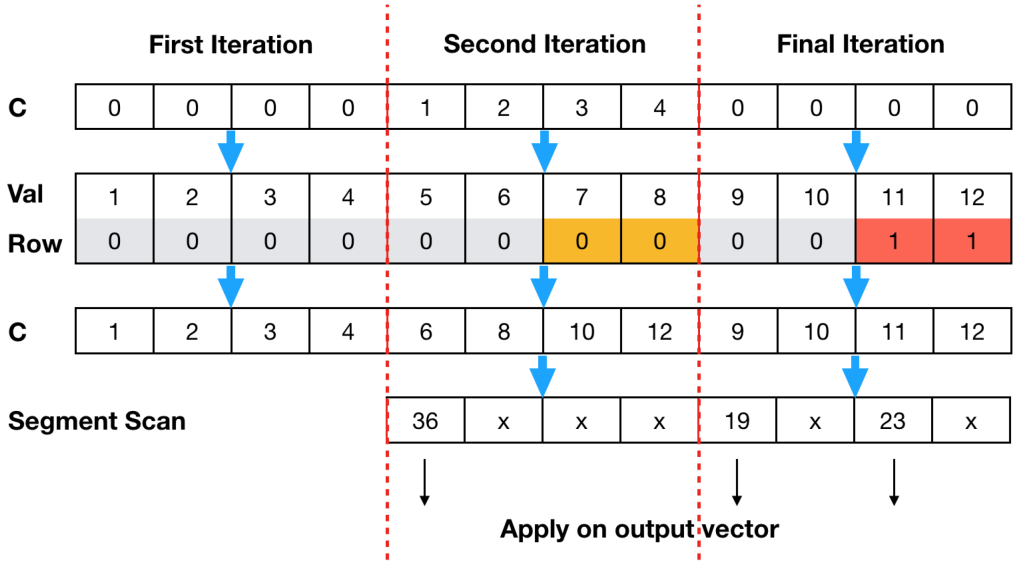

Fig. 3. Visualization of the load-balancing COO kernel. For brevity, we assume that the input vector x of $A \times x$ is all-ones, so we do not need to show the column index of A. The collision of the current row (orange) and the next row (red) invokes the first reduction. The final computation in this warp invokes the second reduction.

The SpMV kernel initially sets the local values in lines 1–3 of Algorithm 1, in particular, the variable c that is used to locally accumulate the partial sums of products of $A \times x$. In line 4, the threads of the warp process the nonzeros they are assigned to, each thread using a stride of $warpsize = 32$, which ensures full coalescence in all memory accesses. Aside from coalesced memory access, assigning warps to chunks allows to leverage the fast intra-warp communication via warp shuffles accessing the values in registers of threads part of the same warp. Up to NVIDIA's Volta architecture (where bitmasks were added [NVIDIA Corp. 2017]), the threads of a warp execute in synchronous fashion. In line 8, the threads of the warp invoke a warp vote function to assess whether multiple threads attempt to issue the atomic operation in main memory (in line 9). Without the warp vote, if multiple threads in the warp are progressing to the next row and attempt to issue a main memory atomic, the synchronized nature of the warp execution would result in a large number of atomic collisions, which are detrimental to the performance of the kernel. Therefore, all threads in a warp decide via a warp vote whether there is at least one of them issuing an atomic write. If such a thread is identified, all threads execute the warp-level segmented scan operation in line 7, with segments defined by the distinct values of the *rowidx* array currently processed by the warp. Afterward, the thread with the lowest index operating on this row performs an atomic addition to update the output vector y in main memory; see lines 8–10.

Figure 3 visualizes the processing of the SpMV kernel using a configuration of four threads working on a chunk of 12 elements. In the first two iterations, all threads work on row 0; in the last iteration, two threads operate on row 1. In consequence, the last two iterations invoke the segmented scan due to the different *current_row* and *next_row* values for two of the threads. The segmented scan reduces the partial sums before the atomic operation adds the final result to the output vector in main memory.

3.2.2 Further Optimizations. To decrease the number of atomic operations (and to increase the amount of computation handled in registers), the three arrays comprising the matrix data

are sorted for increasing *rowidx*. This ensures that each compute unit performs only one atomic operation per *rowidx* value, while also ensuring data locality for the vector y (which can be beneficial on hardware that implements atomic operations in a shared cache file, like the one used in NVIDIA and AMD GPUs). This way, the use of segmented scan avoids atomic collisions between threads of the same warp. Atomic collisions between threads of distinct warps are still possible, but these are unlikely as (1) threads of distinct warps operate on distinct data chunks (so the number of overlapping rows is limited) and (2) distinct warps are not perfectly synchronized. The segmented scan approach radically reduces the number of atomic collisions, but increases the number of arithmetic operations in the algorithm. As the SpMV kernel is heavily memory bound, it can be expected that additional arithmetic operations rarely impact the overall performance as long as they can be overlapped with memory operations. For completeness, we mention that the approach of avoiding intra-warp atomic collisions was initially proposed for the balanced CSRI SpMV kernel in Flegar and Quintana-Ortí [2017].

A good heuristic to improve the access pattern to the input vector x is to additionally sort each set of matrix elements with the same *rowidx* value with respect to increasing *colidx* value. The effectiveness of this approach is highly sensitive to the sparsity pattern of the matrix, but this is a common problem of virtually all SpMV formats and algorithms. (The only exception to this problem known to the authors is the Compressed Sparse Column (CSC) format that focuses on structured access to the vector x at the price of complicated access to the vector y.)

The new instruction set available in the Volta architecture, released after the initial design of the balanced COO kernel [Flegar and Anzt 2017], offers an alternative solution to the atomic collision problem. NVIDIA's "warp match" instructions can be used to obtain a bitmask of all the threads in a warp that supplied the same value to the instruction as the current thread. This allows to quickly form groups of threads working on the same row. The groups can be used as input to the extended warp shuffle instructions available in the Volta architecture, which now have an additional parameter allowing to specify the bitmask of the thread group that participates in the data exchange. In combination, these instructions enable the implementation of a reduction algorithm for an arbitrary subset of the warp. While the algorithm based on multiple reductions reduces the total amount of synchronizations and value exchanges, there is a tradeoff in terms of a more complicated addressing scheme required to select the correct communicators in each stage of the reduction, i.e., threads that take part in the data exchange. Additionally, while the original algorithm has a fixed number of five stages (scan of 32 values), the number of stages in the multiple reductions algorithm depends on the sizes of the communicator cardinality. As a consequence, the stage loop of the multiple reductions algorithm cannot be unrolled, resulting in an increased number of logical operations and jump instructions. Our experiments on a large number of matrices revealed that replacing the segmented scan algorithm with multiple reductions results in lower performance, and we thus use the original algorithm also in kernels for the Volta architecture. However, we mention these optimizations as we expect that the features introduced in the Volta architecture will be improved in future architecture generations.

A different aspect independent of the synchronization of the partial sums is the optimization of the number of chunks we assign to each multiprocessor. In contrast to the classical latency-minimizing CPU hardware leveraging deep cache hierarchies, NVIDIA GPUs use a latency-hiding approach where the streaming multiprocessors are oversubscribed with warps. The intention of having a larger number of active warps is to quickly switch in-between them to cover memory latency [Anzt et al. 2016]. If threads in a warp issue a memory operation, those threads will stall while waiting for the memory transaction to complete. To combat this, rather than allowing the hardware to stall, the warp scheduler may find a warp that is not waiting for a memory operation to complete, and issue the execution of this warp instead. This constant juggling of active warps

allows the GPU to tolerate the high memory latency and keep the compute cores occupied. To enable latency hiding, the number of generated chunks on this hardware should be higher than the amount of parallel processing resources (e.g., simultaneous warps per SM). On the other hand, a high number of chunks increases the chance of atomic collisions as more chunks may contain data located in the same matrix row. We introduce an "oversubscribing" parameter ω that determines the number of threads allocated per each physical core (e.g., simultaneous warps per SM; e.g., $\omega = 2$ means that the number of threads is two times larger than the number of physical cores, while $\omega = 4$ means that there are four threads assigned to each physical core). In previous research, we identified the optimal choice for ω being heavily dependent on the size of the problem [Flegar and Anzt 2017]. The setting:

$$\omega = \begin{cases} 8 & \text{for} \quad n_z < 2 \cdot 10^5, \\ 32 & \text{for} \quad 2 \cdot 10^5 \leq n_z < 2 \cdot 10^6, \\ 128 & \text{for} \quad 2 \cdot 10^6 \leq n_z, \end{cases}$$

however, provides good performance for a wide range of problems [Flegar and Anzt 2017].

3.3 CSR SpMV Kernel

Ginkgo supports multiple CSR implementations through the concept of strategies. The strategies supported are load-balancing CSR (CSRI), merge-path CSR, the classical CSR algorithm, cuSPARSE's own CSR implementation and an automatic strategy. Algorithm 2 provides the scheme for the automatic CSR strategy implemented in Ginkgo, which interfaces different CSR implementations. The two implementations used underneath this strategy are the "classical CSR" in which one CUDA thread processes one row, and the "load-balancing CSR," which uses a scheme similar to the COO SpMV described in Section 3.2 to balance the workload across the compute resources. Precisely, if the target matrix has a large number of rows, the strategy selects the load-balancing CSRI SpMV (see lines 1–2). The load-balancing CSRI SpMV is also chosen if there exists a row with 65 or more elements (see lines 5–6). Otherwise, i.e., for small matrices and few elements in every row, the classical CSR strategy is used.

ALGORITHM 2: Ginkgo's CSR strategy.

```
1:  if #rows > 10^6 then
2:      Use load-balance CSR Kernel
3:  else
4:      Compute max_row_nz = the maximal number of stored element per rows.
5:      if max_row_nz > 64 then
6:          Use load-balance CSR Kernel
7:      else
8:          Use classical CSR Kernel
9:      end if
10: end if
```

3.4 ELL SpMV Kernel

In Algorithm 3, we present the ELL SpMV kernel implemented in Ginkgo to target SIMD architectures like GPUs. This kernel splits the matrix A into chunks of rows, and assigns a group of $subwarp\_size = 2^k$ ($k \leq 5$) threads to each chunk. The number of threads assigned to a chunk is computed via Algorithm 4. Generally, the size of the subwarp is increased with the number of nonzero elements accumulated in a single row. However, if $subwarp\_size = 32$, multiple groups of threads are assigned to the same row; see line 8 in Algorithm 4. This strategy aims at increasing the

ALGORITHM 3: ELL Early Stopping SpMV Kernel.

```
1:  Initial Value c = 0
2:  Compute y_start = start of row chunk of A assigned to this subwarp
3:  Compute y_end = end of row chunk of A assigned to this subwarp
4:  for idx ∈ [y_start, y_end], idx + = subwarp_size do
5:      Compute ind = index of this element in the ELLformat
6:      if A(row, colidx[ind]) is fill-in then
7:          break
8:      end if
9:      Perform local operation c += A(row, colidx[ind]) * x[colidx[ind]]
10: end for
11: Perform reduction of c on the warp
12: if thread 0 in subwarp then
13:     atomic_add c on the output vector
14: end if
```

ALGORITHM 4: Decision of the number of warps per row and the subwarp size in ELL

```
1:  subwarp_size = 1
2:  nwarps_per_row = 1
3:  ell_ncols = maximum number of non zero elements per row
4:  nwarps = total number of warps available on the GPU
5:  if ell_ncols / nrows > 1e − 2 then
6:      subwarp_size = min(32, 2^ceil(log2(ell_ncols)))
7:      if subwarp_size == 32 then
8:          nwarps_per_row = max(min(ell_ncols/32, nwarps/nrows), 1)
9:      end if
10: end if
```

occupancy of the GPU multiprocessors when targeting short-and-wide matrices that accumulate many elements in few rows. After the subwarp size is determined, the chunk range for a specific subwarp is computed in lines 2–3 in Algorithm 3, and the threads process the data in the chunk with the loop in lines 4–10. An important aspect of this algorithm is highlighted in lines 6–8, which stop the processing of a row if the threads reach the fill-in area. Without this early stopping, depending on the nonzero imbalance, significant resources are wasted by working on the padding elements. After completion of the matrix vector multiplication step, the partial sums accumulated in thread-local variables are reduced (line 11) and added to the output vector in global memory; see line 13. Even though this operation must be an atomic operation as multiple subwarps (part of multiple thread blocks) may operate on the same row, the chance of atomic collisions is low due to the previous warp-local reduction in line 11.

3.5 SELL-P SpMV Kernel

For matrices where the nonzero elements are not distributed equally across the rows, the ELL format can introduce significant overhead. An efficient strategy to reduce the padding overhead the ELL format introduces (in particular for unbalanced matrices) is to decompose the original matrix into blocks, each containing multiple rows, and to store the distinct blocks in ELL format. SIMD architectures suggest to adapt the size of the blocks to the SIMD-length, such that only row-blocks with the height of the SIMD-length are padded to the same number of nonzero elements, but the rows in distinct blocks can differ in the number of nonzero elements. The resulting storage format basically consists of a set of ELL matrices with an additional row pointer indicating the

ALGORITHM 5: SELL-P SpMV Kernel

```
1: Initial value c = 0;
2: Get length = the length of this slice;
3: Compute row = the row index of this thread;
4: for ind ∈ [0, length), ind + = 1 do
5:     Compute ind = index of this element in the SELL-P format;
6:     Perform local operation c += A(row, colidx[ind]) *x[colidx[ind]];
7: end for
8: Set c in the output vector;
```

start of the distinct row-blocks. However, the size of the blocks in the "sliced ELL (SELL)" format is not tied to the SIMD length, but can be optimized with respect to the matrix properties. In particular, the memory footprint can motivate to use a block size smaller than the SIMD length, as the padding of the rows in a block is determined by the row with the largest nonzero count in this block, which can be significantly smaller than the largest nonzero count considering all rows of the matrix or all rows part of a block with the size of the SIMD length [Anzt et al. 2014]. Generally, the padding overhead decreases for smaller block sizes, and the observation that a block size of 1 eliminates all padding reveals that the SELL format is a compromise between the ELL format and the CSR format with the block size being the control—A block size of 1 results in the CSR format, a block size of n (with n being the matrix size) results in the ELL format [Kreutzer et al. 2014].

The SELL format can heavily benefit from an initial matrix reordering step that sorts the matrix rows for increasing (or decreasing) nonzero count [Kreutzer et al. 2014]. This way, the matrix row with the closest nonzero count is adjacent in the reordered matrix and (except for block boundaries) located in the same ELL block. For a fixed block size, ordering the matrix rows for increasing (or decreasing) nonzero count minimizes the padding. However, sorting the rows and reordering a matrix can become an expensive preprocessing step that is only justified for a high number of SpMV kernel invocations. As quantifying the cost of reordering and quantifying the benefits in repetitive SpMV calls is difficult in a comparative performance study, we refrain from including this option in the experimental part of this article.

For the CUDA architectures we target in the performance tests, we previously proposed a strategy where multiple threads are handling the partial sums in a row; see Figure 4 [Anzt et al. 2014]. This strategy can be much faster for matrices with a moderate or large number of nonzeros in each row, however, potentially requires some additional padding to maintain SIMD execution mode for cases where the *n_z_per_row* count is not divisible by the number of threads t assigned to this row. We therefore refrain from assigning multiple threads to the same row, and instead use the straightforward SELL-P SpMV as given in Figure 4 and pseudocode in Algorithm 5. In the experimental section, we consider the SELL-P format with a block size of $b = 64$ as this setting gives good results for the many matrices with a moderate *n_z_per_row* ratio.

4 HYBRID MATRIX FORMATS AND OPTIMAL MATRIX SPLITTING

The general idea of the hybrid ("HYB") SpMV is to split the matrix into two parts: a regular part that is processed using the SIMD-friendly ELL SpMV kernel, and an irregular part that is processed using a SpMV kernel more suitable for unbalanced and irregular sparsity patterns. The goal is to optimize the overall performance by using kernels that are runtime-optimal for the respective contributions. The standard approach is to combine the ELL kernel with a COO kernel, which is, for example, the hybrid realization in NVIDIA's cuSPARSE library [NVIDIA Corporation 2018]. Motivated by the facts that the matrix contribution not covered by the ELL kernel is expected

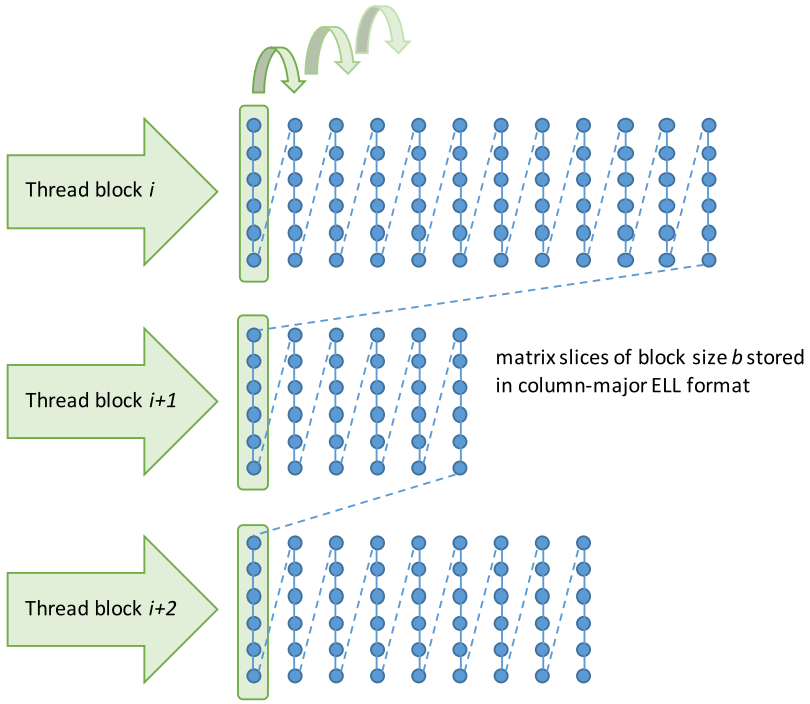

Fig. 4. Visualization of the SELL-P memory layout and the SELL-P SpMV kernel procedure including the reduction step using the blocksize $b = 6$ (corresponding to SELL-6).

to be irregular, and the load-balancing COO kernel providing very good performance for these problems [Flegar and Anzt 2017], we decide to complement the ELL kernel with the load-balancing COO kernel. We note that, in this setting, invoking the ELL kernel for processing the regular contribution first removes the need for scaling the y-vector prior to the COO kernel invocation when computing $y = A \cdot x$.

A central question in the HYB SpMV is how to split the matrix into the two formats. The choice of the threshold can have significant performance impact: handling too many elements in each row with the ELL format can introduce significant overhead in terms of zero-padding, while choosing a too small threshold will handle too many elements with the more expensive irregular SpMV kernel. Previous approaches [Bell and Garland 2009; NVIDIA Corporation 2018] either rely on user-provided hard-coded thresholds determining how many elements in each row are stored in ELL format, or provide an automated threshold detection based on the distribution of nonzeros among matrix rows and (outdated) observation that the ELL format is at most three times faster than COO for balanced matrices. The first strategy provides high tunability, while completely ignoring the distribution of nonzeros (unless the user incorporates that analysis into its threshold detection), while the second takes into account the nonzero distribution at the expense of tunability. In this work, we propose a generalized strategy, which incorporates both the automatic analysis of the nonzero distribution in the matrix and user feedback how to translate this nonzero distribution into the threshold parameter. The advantage of such an approach is that the user-supplied parameter can incorporate properties of the system and the COO and ELL SpMV implementations, and indirectly steer the selection of the threshold, while the threshold itself is still determined using the properties of the nonzero pattern. This also means that the same parameter can be used for a

wide range of matrices, which implies that an autotuning procedure can be performed to find a near-optimal value for the entire matrix set.

Since SpMV is memory bound on virtually all modern architectures, the proposed threshold selection is based on minimizing the memory footprint of the HYB matrix format. The single tunable parameter represents other effects impacting performance—such as communication overhead, performance of atomic operations, or even the variation in implementation quality of COO and ELL kernels—and enables to modulate between the COO kernel and the ELL kernel.

First, we analyze the effect of the threshold selection on the total volume of data required to store the HYB matrix. Let A be a sparse matrix with n rows and let n_i be the number of nonzeros in row i, $i = 1 \ldots n$. For storing the first t elements in each row in ELL and the rest in COO format, the total memory requirement $S(t)$ for the hybrid format derives as:

$$S(t) = n \cdot t \cdot (v + p) + (v + 2p) \sum_{i=1}^{n} f(n_i - t), \tag{1}$$

where v is the size of the values, p is the size of indexes, and f is a function defined as:

$$f(x) = \begin{cases} x & : x \geq 0 \\ 0 & : \text{otherwise.} \end{cases} \tag{2}$$

In this model, S is a continuous, piece-wise linear function, differentiable everywhere except in $t = n_i,\ i = 1, \ldots n$. Thus, the minimum of S (i.e., the minimal storage requirements) is located at a boundary of two linear segments. Consequentially, the optimal value for k belongs to the set $R = \{n_i \mid i = 1, \ldots, n\}$. A more detailed analysis reveals that the slope S' of the function in a point $t \notin R$ derives as:

$$S'(t) = n(v + p) - (v + 2p) \sum_{i=1}^{n} f'(n_i - t) \tag{3}$$

$$= n(v + p) - (v + 2p) \left|\{x \in R \mid x \geq t\}\right|. \tag{4}$$

Thus, the initial slope of $-np$ for $t = 0$ increases monotonically in each following segment by $v + 2p$ until it reaches $n(v + p)$. We deduce that the global minimum of the function is reached on the boundary of the segments where the negative slope turns positive. In other words, this is on the boundary of segments $i - 1$ and i, where $(i - 1)(v + 2p) - np < 0$ and $i(v + 2p) - np \geq 0$, i.e., at the beginning of segment:

$$i = \left\lfloor \frac{np}{v + 2p} + 1 \right\rfloor. \tag{5}$$

As the boundaries of the segments are defined by the nonzero-per-row counts of the matrix A, finding the threshold minimizing the memory requirement is equivalent to identifying the $\left\lfloor \frac{np}{v+2p} + 1 \right\rfloor$-th smallest nonzero-per-row count. This is the $\frac{np}{v+2p}$-th n-quantile of the nonzero-per-row distribution, or the value at point $x = \frac{p}{v+2p}$ of the quantile function Q_R:

$$Q_R(x) := \min \{t \in \mathbb{N} \mid x < F_R(t)\} \tag{6}$$

$$F_R(t) := \frac{|\{r \in R \mid r \leq t\}|}{|R|}. \tag{7}$$

The function Q_R for a (multi)set of a nonzero-per-row distribution R can be efficiently computed at point x by using a selection algorithm to select the $\lfloor |R| + 1 \rfloor$-th order statistics in the set. State-of-the-art selection algorithms offer average sequential time complexity of $O(n)$. Effective parallelization methods are also available [Chandrashekar and Sahin 2014; Ribizel and Anzt 2019].

Table 1. Some Key Values of the Threshold Parameter x

x	Description of threshold $t = Q_R(x)$.
<0	The entire matrix is stored in COO format.
0	No padding is allowed in the ELL part.
$\frac{1}{4}$	Optimal storage consumption for 64-bit values.
$\frac{1}{3}$	Optimal storage consumption for 32-bit values.
	Threshold used in Bell and Garland [2009].
$1 - \frac{1}{\lvert R \rvert}$	The entire matrix is stored in ELL format.

In summary, the threshold that offers optimal storage consumption is $t = Q_R(\frac{p}{v+2p})$. For 32-bit indexes and double precision matrix entries (64-bit values), this is equal to $Q_R(0.25)$. For single precision (32-bit values), the optimal is achieved for $Q_R(\frac{1}{3})$. The earlier strategy used in Bell and Garland [2009] can also be expressed in terms of the quantile function and is equivalent to selecting $Q_R(\frac{1}{3})$ as the threshold. Other values $Q_R(x)$ can also be used as thresholds: small values of x prefer the COO format and avoid padding in ELL, while larger values of x promote extra padding to reduce the amount of data that must be stored in COO. Thus, the solution we propose is to expose the selection of parameter x to the user (or the autotuning tool), while the implementation incorporates the matrix-specific data Q_R to generate the final threshold $t := Q_R(x)$. In Table 1, we list some key values of parameter x along with their effects on the storage scheme.

5 EXPERIMENTAL PERFORMANCE ASSESSMENT

5.1 Experiment Setup

The experimental analysis was conducted on an NVIDIA V100 "Volta" GPU, with support for CUDA compute capability 7.0 [NVIDIA Corp. 2017]. All GPU kernels were encoded and compiled in the CUDA framework, using CUDA version 9.2.

The performance evaluation covers the entire SuiteSparse matrix collection [SuiteSparse 2018]. Some of the over 2,800 test matrices contain dense rows, which make it virtually impossible to convert them to ELL format. Those problems are ignored in the ELL SpMV performance assessment.

All experiments are performed in double precision arithmetic and the GFLOP/s rates are computed assuming that the number of flops is always $2n_z$, where n_z is the number of nonzeros of the test matrix (even when padding is used).

5.2 SpMV Performance Analysis

We first evaluate the performance of the load-balancing COO SpMV kernel. In Figure 5, we compare the performance [GFLOP/s] achieved by Ginkgo's load-balancing COO SpMV with the COO kernel available in NVIDIA's cuSPARSE library. In the scatter plot, each dot represents one test matrix. The matrices are ordered according to increasing nonzero count, not accounting for features in the nonzero distribution. Despite that the irregularity of a matrix heavily impacts the SpMV kernels' efficiency, we can identify for both COO realizations a clear trend of the upper performance bound where most of the test cases are accumulated. The illustration reveals that Ginkgo's COO SpMV achieves much higher performance than NVIDIA's COO kernel. Table 2 provides some performance counters extracted for both the cuSPARSE and Ginkgo COO SpMV kernels for five matrices. The cuSPARSE SpMV invokes several kernels. Given that according to `nvprof` the kernel `coo_fastpass_ref<double, bool=1>(...)` accounts for over 96% of the execution time, we consider this the dominating kernel and report the counters for this kernel, only. The results in

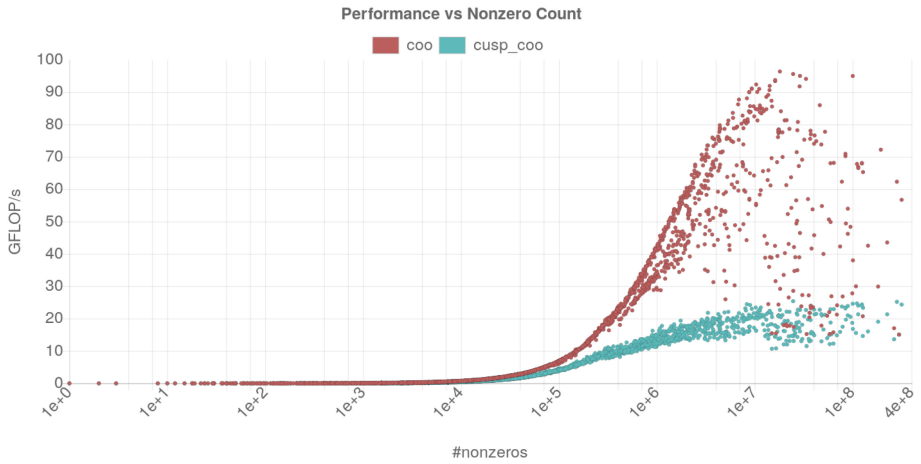

Fig. 5. Performance of Ginkgo's COO SpMV and NVIDIA's COO SpMV (cusparseDhybmv with CUSPARSE_HYB_PARTITION_USER and threshold of 0) part of the cuSPARSE library.

Table 2. Performance Metrics Extracted from nvprof for cuSPARSE and Ginkgo's COO SpMV Kernels

Name	Library	Occupancy	Device Memory Read (GB/s)	Device Memory Write (GB/s)	Global Hit Rate	Multiprocessor Activity
adaptive	cuSPARSE	0.119352	179.40	14.381	23.90	74.97
	Ginkgo	0.972729	515.10	43.722	7.56	99.17
delaunay_n22	cuSPARSE	0.124688	179.63	11.380	19.12	77.60
	Ginkgo	0.968915	577.14	39.206	17.13	98.94
hugebubbles-00020	cuSPARSE	0.123527	293.04	15.887	23.32	75.32
	Ginkgo	0.986120	413.95	27.276	0.71	97.24
web-BerkStan	cuSPARSE	0.120214	150.28	8.7993	15.68	73.02
	Ginkgo	0.934222	648.92	39.590	41.50	97.09
web-Google	cuSPARSE	0.124072	255.49	14.193	18.70	78.55
	Ginkgo	0.928319	634.57	46.526	3.97	97.43

Table 2 reveal that Ginkgo's COO implementation exhibits better device memory throughput, GPU occupancy, and overall streaming multiprocessor activity. On the other hand, for several cases, the cuSPARSE implementation exhibits better global memory hit rate in the L1 cache.

The second SpMV kernel that we consider is the Ginkgo CSR SpMV that automatically interfaces to either the load-balancing CSRI kernel or the classical CSR; see Section 3.3. For the CSR format, a comparison against NVIDIA's counterpart is challenging as cuSPARSE contains several SpMV kernels for the CSR format. In Figure 6, we compare the performance of the different cuSPARSE CSR kernels for the complete set of test matrices. Ignoring a few corner cases, we can identify the cusp_csr kernel (cusparseDcsrmv) as the overall performance winner. In Figures 7 and 8, we therefore compare the Ginkgo CSR SpMV with the cusparseDcsrmv CSR kernel available in NVIDIA's cuSPARSE library.

Figure 7 shows that both SpMV kernels reach a maximum performance of 120 GFlop/s. For matrices with a nonzero count between 1e+5 and 1e+6, there is a significant performance difference between Ginkgo and cuSPARSE CSR SpMV implementations. To further analysis, we relate in Figure 8 the Ginkgo vs. cuSPARSE speedup to the variance in the nonzero-per-row distribution.

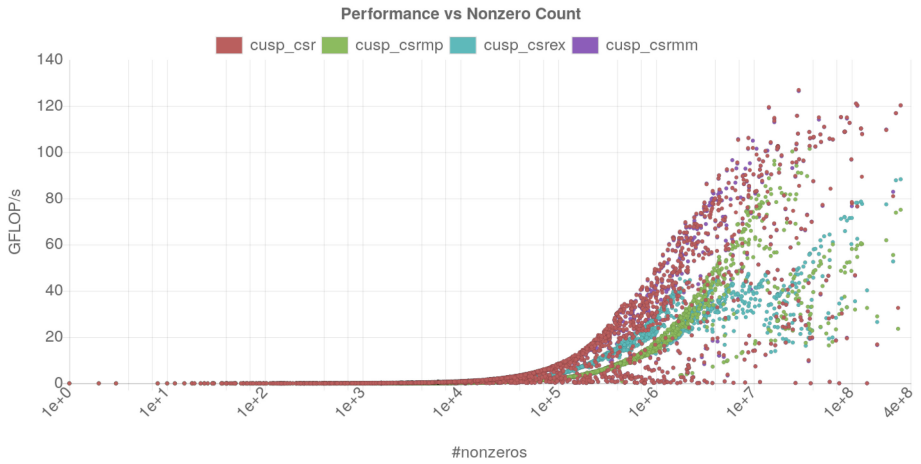

Fig. 6. Performance of the distinct CSR SpMV kernels available in NVIDIA's cuSPARSE library (CUDA version 9.2).

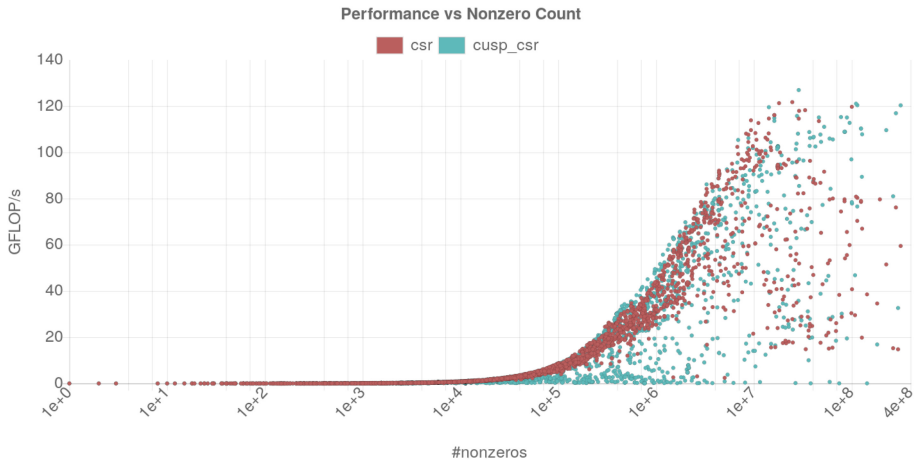

Fig. 7. Performance of Ginkgo's CSR SpMV (automatic strategy) and NVIDIA's CSR SpMV (cusparseDcsrmv) part of the cuSPARSE library.

This ratio represents the matrix imbalance, with very balanced matrices appearing on the left side of the figure, and very imbalanced matrices on the right side of the figure. Both implementations are comparable for moderate imbalance ratios, Ginkgo being faster for 1,878 of the 2,743 test matrices considered (68%). For very imbalanced nonzero distributions, Ginkgo's load-balancing CSR kernel outperforms the cuSPARSE CSR SpMV by almost three orders of magnitude.

Acknowledging that cusp_csrmp is specifically designed to achieve high performance also for imbalanced matrices, we compare in Figure 9 Ginkgo's CSR SpMV with cusp_csrmp. The results reveal that cusp_csrmp efficiently reduces the speedup of Ginkgo's CSR SpMV from three orders of magnitude to at most 40×. At the same time, Ginkgo's CSR SpMV is the clear winner in this face-to-face comparison. Precisely, Ginkgo is faster than the cuSPARSE counterpart for 2,645 of the 2,743 test matrices considered (96%).

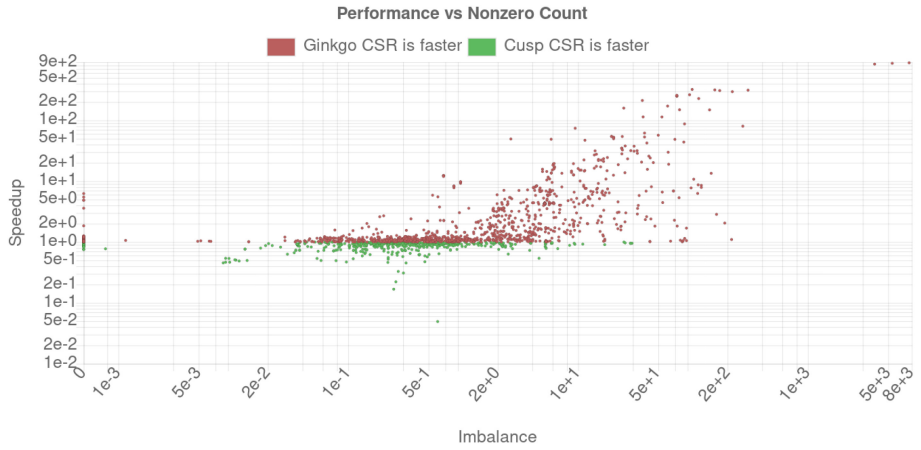

Fig. 8. Performance of Ginkgo's CSR SpMV (automatic strategy) relative to cuSPARSE CSR SpMV related to the variance of the nonzero-per-row distribution. The results are shown for matrices with more than 10,000 nonzero elements.

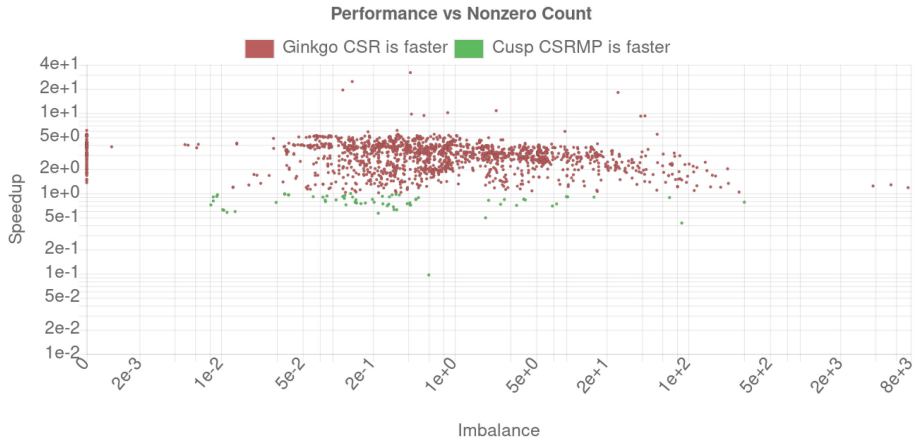

Fig. 9. Performance of Ginkgo's CSR SpMV (automatic strategy) relative to cuSPARSE cusp_csrmp related to the variance of the nonzero-per-row distribution. The results are shown for matrices with more than 10,000 nonzero elements.

A different SpMV kernel suitable for irregular sparse matrices that we consider is the hybrid SpMV algorithm complementing the COO with the ELL format for the regular matrix contribution. As an intermediate step, we compare in Figures 10 and 11 the performance of Ginkgo's ELL SpMV with NVIDIA's counterpart. In this analysis, we only ignore the test problems where the ELL format exceeds the memory capacity of the Volta GPU. Comparing Figure 10 with Figure 5 reveals that for large and balanced matrices, the ELL kernel can exceed the performance of the COO kernel (>120 GFLOPs for individual problems in ELL vs. 100 GFLOPs for the COO kernel). For matrices containing more than 1e+5 nonzero elements, there are significant performance advantages on the Ginkgo side. To further investigate this observation, we again visualize Ginkgo's speedup in relation to the $\frac{max\_row\_nnz}{numrows}$ ratio in Figure 11. We notice that for sparsity ratios smaller than $1e-2$, both kernels provide similar performance. For sparsity ratios larger than $1e-2$, Gingko's ELL SpMV is significantly faster than the ELL SpMV from cuSPARSE. This can likely be linked to

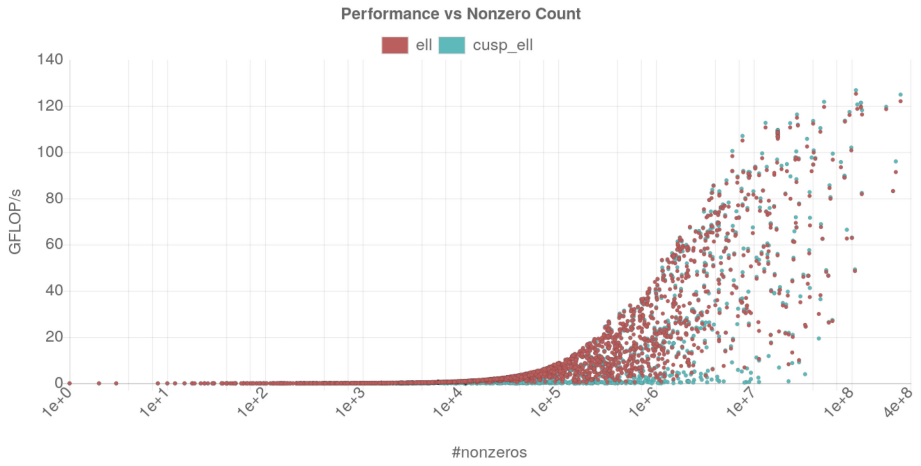

Fig. 10. Performance of Ginkgo's ELL SpMV and NVIDIA's ELL SpMV (cusparseDhybmv with CUSPARSE_HYB_PARTITION_MAX) part of the cuSPARSE library.

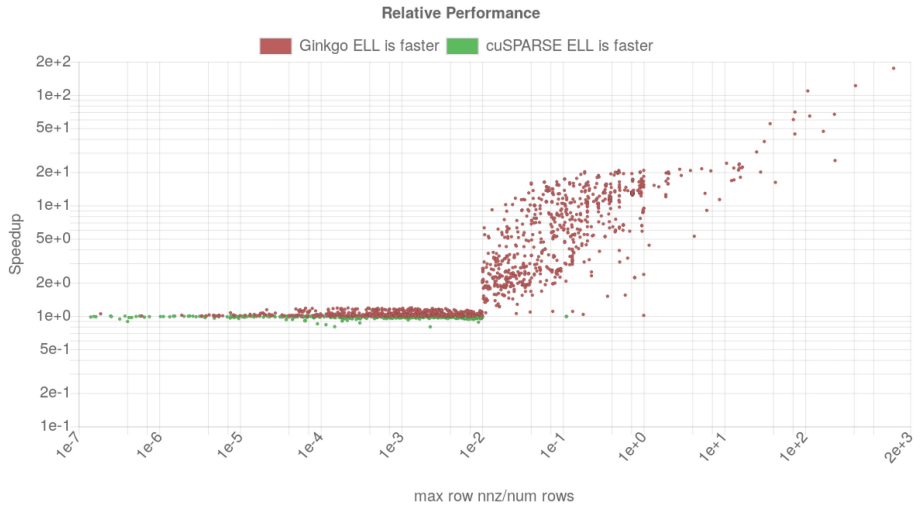

Fig. 11. Performance of Ginkgo's ELL SpMV relative to the cuSPARSE ELL SpMV. The results are shown for matrices with more than 10,000 nonzero elements.

the specific parallelization pattern used for Ginkgo's ELL SpMV which, starting at this arbitrary threshold, splits the rows into chunks and assigns multiple warps to each row. This kernel design turns out to succeed in adapting to matrices with a few rows with many nonzero elements.

Before comparing Ginkgo's hybrid SpMV to NVIDIA's counterpart, we assess in Figure 12 which strategy renders the best performance using a "performance profile" [Gould and Scott 2016]. The performance profile is a visualization that is ideal for comparing the performance of algorithms on problem sets that are otherwise difficult to illustrate (e.g., they are too large, or there is no reasonable metric to determine how challenging a particular problems is). Each algorithm A from the set $\mathbb{A}$ of all algorithms being compared on a problem set Φ is represented via its *performance function* $f_A = f_{A,\mathbb{A},\Phi}$. The value $f_A(t)$ in point t of the performance function is defined as the percentage of problems $\phi \in \Phi$ where the performance of A is not more than t times worse than the

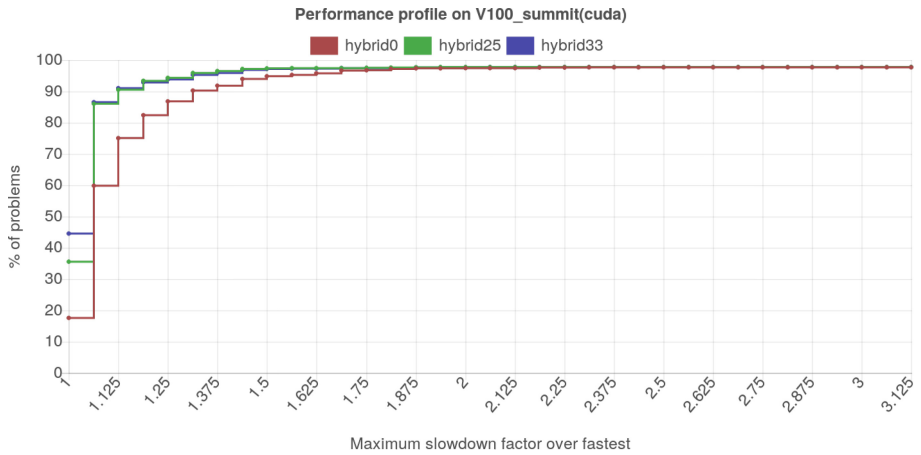

Fig. 12. Performance profile comparing the different splitting strategies in Ginkgo's HYB kernel.

performance of the best algorithm in $\mathbb{A}$. Formally, if $p(A, \phi)$ is the performance of algorithm $A \in \mathbb{A}$ on problem $\phi \in \Phi$, the performance function is defined as:

$$f_A(t) := \frac{|\{\phi \in \Phi \mid t \cdot p(A, \phi) \geq \max_{B \in \mathbb{A}} p(B, \phi)\}|}{|\Phi|}. \tag{8}$$

The performance profile is a set $P_{\mathbb{A},\Phi} := \{f_A \mid A \in \mathbb{A}\}$ of all performance functions of $\mathbb{A}$, and is usually visualized by sampling the performance functions at fixed intervals and plotting them as lines in a line plot. See the book of D. Higham and N. Higham [2005] for more details about performance profiles.

To increase the significance of the performance profile, we consider in Figure 12 (and all other profiles in this section) only test matrices with at least 100,000 nonzero elements. The performance profile reveals that there exists no overall best splitting strategy, but each threshold has its niche. The $Q_R(\frac{1}{3})$ (labeled "hybrid33") is the best splitting strategy for 45% of the problems, the $Q_R(.25)$ (labeled "hybrid25") and the $Q_R(0)$ (labeled "hybrid0" and not allowing for any padding) win 37% and 18% of the test cases, respectively. In terms of generality, the hybrid25 and hybrid33 are the overall winners. Their performance profiles grow faster than that of hybrid0, quickly reaching a similar level. For less than 12% of the problems, there is a threshold configuration that results in a more than 25% slower kernel. This is expected as all hybrid strategies handle the irregular matrix contribution with the load-balancing COO kernel, and already the COO kernel itself generalizes well across all matrices.

With the mission of the hybrid (HYB) SpMV to generalize well, we favor making the hybrid25 strategy the default setting in the comparison against NVIDIA's HYB SpMV algorithm in Figure 13. In Section 3, we previously derived the hybrid25 strategy as optimal choice in the theoretical analysis optimizing the memory footprint. Cache effects, however, remain outside the scope of this analysis, and with Figure 12 not identifying an overall superior splitting strategy, it is not surprising that NVIDIA's hybrid SpMV algorithm (using a different splitting strategy) outperforms Ginkgo's HYB SpMV for individual cases. However, Ginkgo's HYB kernel achieves higher performance for the majority of the test cases. Comparing between Ginkgo's formats, we notice that the HYB algorithm fails to preserve the performance peaks of the ELL SpMV. The reason behind this is that the hybrid kernel always stores part of the matrix in the more general (thus more complex) COO format. At the same time, the HYB SpMV generalizes well and is capable of handling matrices

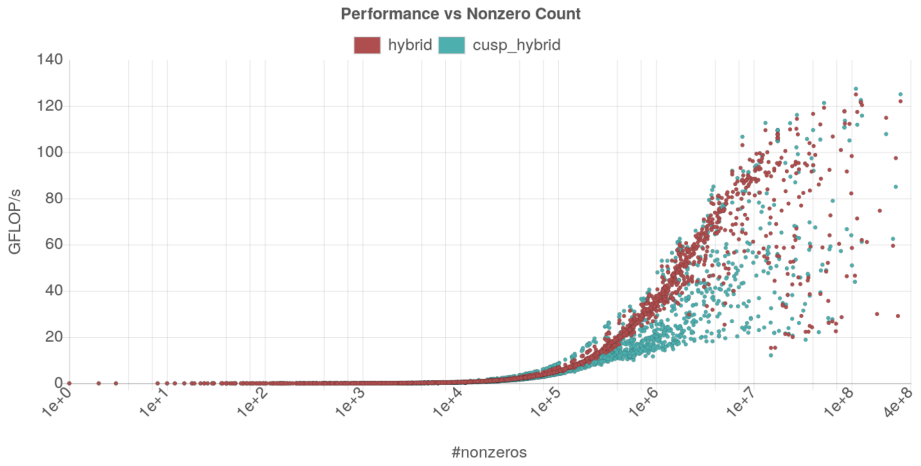

Fig. 13. Performance of Ginkgo's HYB SpMV and NVIDIA's HYB SpMV (cusparseDhybmv) part of the cuSPARSE library.

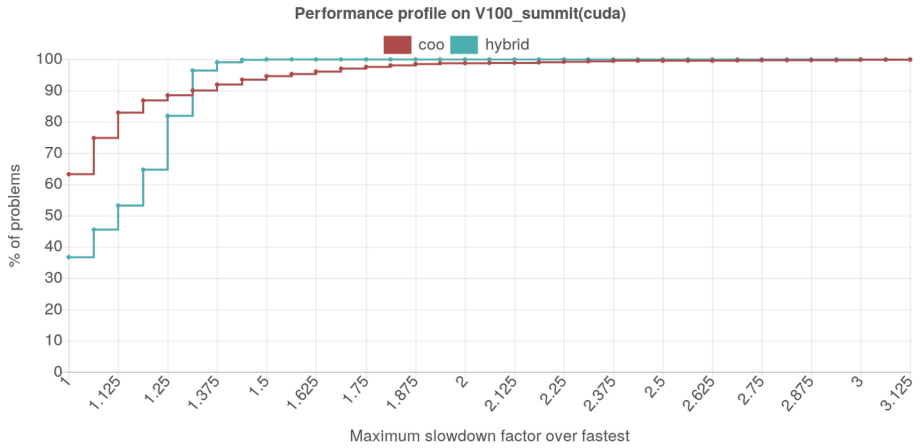

Fig. 14. Performance profile comparing the COO and HYB SpMV part of Ginkgo.

that remain outside the scope of the ELL format due to the memory requirements exceeding the architecture specifications.

To assess whether the strategy of combining the SIMD-friendly ELL kernel with the load-balancing COO kernel renders benefits over the COO kernel, we visualize in Figure 14 a comparison between these two formats. We notice that both formats have different niches with HYB SpMV being the fastest format for about 38% of the test cases and COO SpMV for 62% of the test cases. Furthermore, for 99% of the test cases, the HYB SpMV is at most 37.5% slower than the COO SpMV. For none of the test matrices is the HYB SpMV more than 50% slower than the COO SpMV. Conversely, the COO SpMV is at most twice slower than the HYB SpMV. We conclude that the HYB SpMV generalizes better than the COO SpMV.

We conclude the performance assessment with Figure 15, where we include all presented formats in a single performance profile. We acknowledge that this comparison is by no means comprehensive, as there exist many SpMV kernels often tuned for specific matrix properties (such

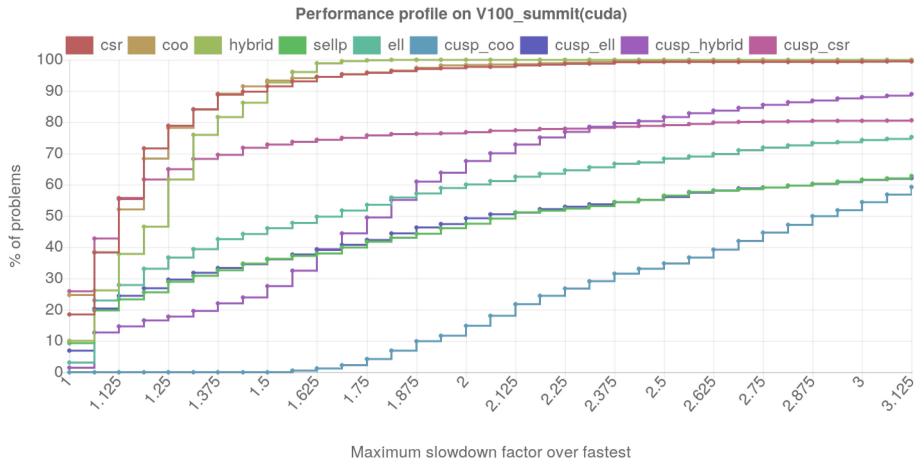

Fig. 15. Performance profile comparing a list of SpMV kernels.

as diagonal matrices). However, we refrain from including problem-specific formats in the comparison as we focus on formats that generalize well.

Figure 15 identifies Ginkgo's HYB, COO, and CSR SpMV routines being the overall winners in generality. The cuSPARSE CSR kernel provides the best performance for around 26% of the matrices, but the slope is flat and only reaches 80% of all test cases. The performance profiles of the other NVIDIA counterparts start much lower and exhibit a smaller slope. Ginkgo's ELL and SELL-P formats have similar performance profiles, with advantages on the ELL side, and both being superior or on par with NVIDIA's ELL kernel. In conclusion, Ginkgo's open-source kernels are at least competitive, in many cases superior to NVIDIA's cuSPARSE library.

5.3 Conversion Timings

What we did not account for in the previous performance experiments on the SpMV kernels is the cost of converting in-between the matrix formats. The conversion timings are relevant as—though a Ginkgo user can instantiate a matrix in any supported format—in most cases, a top-level application will likely handle the system matrix in some standard format, i.e., the resource-optimal CSR format. In response, Ginkgo also supports high-performance matrix format conversion kernels for multicore architectures and GPUs. In this section, we report performance results for the GPU conversion kernels for a set of sample matrices. As reporting plain numbers gives little insight into the actual cost, and the user in most cases is interested in whether a format conversion pays off for multiple SpMV kernel invocations, we report the conversion cost from and to CSR as multiple of CSR SpMV invocations. In Table 3, we list the sample matrices we use for the performance evaluation of this analysis along with some key characteristics. All test problems have a very unbalanced nonzero distribution, for which the ELL format incurs significant overhead due to the use of padding with explicit zeros. Despite that, all problems fit into GPU memory for all the formats considered. In Figure 16, we visualize the conversion cost for the distinct formats supported by Ginkgo in terms of Ginkgo CSR SpMV invocations. We observe that for none of the problems, the conversion overhead exceeds 280 CSR SpMV invocations. For challenging problems like hugebubbles-00020, the conversion cost can be as low as 2 CSR SpMV. Generally, converting from CSR to another format is more expensive than the reverse direction. Except for the highly-optimized HYB format (and some conversions involving the ELL format), the conversions are even cheaper than 50 CSR SpMV invocations. Hence, although only indicative, the results reveal that in

Table 3. Details of Matrices from the SuiteSparse Matrix Collection Used for the Conversion Routines Evaluation

Name	n	n_z	Empty rows
adaptive	6,815,744	13,624,320	425,984
delaunay_n22	4,194,304	12,582,869	555,807
hugebubbles-00020	21,198,119	31,790,179	3,530,509
web-BerkStan	685,230	7,600,595	4,744
web-Google	916,428	5,105,039	176,974

Fig. 16. Performance of Ginkgo's format conversion routines in amount of CSR SpMV on NVIDIA V100 hardware.

the context of a sparse iterative solver featuring a high number of SpMV invocations, converting the matrices to the most efficient SpMV format remains a viable strategy despite the conversion overhead.

6 SUSTAINABLE SOFTWARE DEVELOPMENT AND OPEN SOURCE CODE

As newer hardware architectures incorporate elaborate features and specialized instruction sets, the algorithms that can leverage them become increasingly complex. As a result, it can no longer be expected that a domain scientist interested in using the new method can re-implement it solely by following the descriptions provided in a scientific paper.

Thus, the method must be made available to the broader community while ensuring interoperability with other software. To make this possible, a software development process that guarantees the quality of the software also has to be employed. The simplest strategy to ensure all of these requirements is to develop the implementation of the new method as part of an existing open source library. For this reason, the matrix formats and SpMV implementations presented in this work have been implemented inside the Ginkgo library. While it is a relatively new addition to the HPC software landscape, its focus on providing extensible high-level abstractions allows us to implement the new formats and algorithms as first-class citizens in the library. For readers not familiar with the library, we provide a brief description in Section 6.1.

Using Ginkgo as the backbone for the implementations enables us to leverage Ginkgo's ecosystem of tools and utilities to ensure high quality, correctness, usability and performance reproducibility. Its existing build system allows compiling new methods with ease, the continuous integration (CI) system ensures that the new code is compilable and correct by building it and running the unit tests after each change, and ensures there are no regressions. The new methods have been added to Ginkgo's benchmark runners, allowing easy reproducibility of benchmark results obtained in this article. In addition, the results are uploaded to Ginkgo's web-accessible benchmark results repository, where they can be freely accessed using the performance analysis tool "Ginkgo Performance Explorer" (see Section 6.2 for more details). Finally, a simple usage example of Ginkgo's SpMV interface is showcased in Section 6.3.

6.1 Ginkgo Open Source Linear Algebra Library

Ginkgo is an open source library written in C++, which targets high-performance sparse linear algebra on single-node manycore and heterogeneous architectures. The library focuses on solving sparse linear systems and accommodates a large variety of matrix formats, state-of-the-art iterative (Krylov) solvers, and preconditioners.

Unlike classical BLAS- or LAPACK-based libraries, Ginkgo is a "linear operator algebra library" that focuses on the concept of the linear operator to provide better composability and extensibility, and provides a clean and uniform interface for all the linear algebra objects used in Ginkgo. Using these design goals and guidelines, Ginkgo provides high performance implementation of multiple solvers, matrix formats, and preconditioners. Ginkgo is designed to support transparent execution on distinct devices and favor device specific optimizations by providing multiple extendable backends (OpenMP, CUDA, etc.).

The Ginkgo library has a strong focus on software sustainability, and this ensures the ease of use and contributing. Aside from being used as framework for algorithm research, Ginkgo aims to provide a numerical software ecosystem designed for the easy adoption by the scientific computing community. This requires sophisticated guidelines and high quality code. As such, Ginkgo follows the guidelines and policies of the Extreme-scale Scientific Software Development Kit (xSDK [2018]) and the Better Scientific Software (BSSw [2018]) initiatives. To facilitate contributions, Ginkgo is open source with a modified BSD license and publicly accessible on GitHub[2]. Ginkgo is open to external contributions through a peer review system and provides issues and development efforts tracking [Anzt et al. 2019].

In order to ensure the correctness of the parallel implementations inside Ginkgo, extensive tests are implemented and all the parallel implementations are consistently tested against a reference sequential implementation through the use of a CI system. This CI system tests the software's successful compilation and execution on multiple machine architectures and libraries in order to ensure the portability of Ginkgo. Ginkgo also makes use of Continuous Benchmarking to prevent any performance regression and bolster performance improvements [Anzt et al. 2019].

6.2 Availability of Performance Results

To ease comparison with other implementations, the results presented in this work have been uploaded to Ginkgo's performance data repository[3] in the same format as used for the rest of Ginkgo's benchmarks. This data can be obtained freely by any interested party and analyzed using any of the data analysis tools that can process JavaScript Object Notation (JSON) files (such as

[2] https://github.com/ginkgo-project/ginkgo.
[3] https://github.com/ginkgo-project/ginkgo-data branch 2019spmv-paper.

```
using Exec = gko::!Executor Type!Executor;
using Mtx = gko:matrix::!Matrix Type!<>;
using Vec = gko::matrix::Dense<>;

auto exec = Exec::create();

auto matrix = gko::read<Mtx>(std::ifstream("matrix.mtx"), exec,
                             !Matrix Setttings!);
auto vector = gko::read<Vec>(std::ifstream("vector.mtx"), exec);
auto result = Vec::create_with_config_of(lend(vector));
auto alpha = gko::initialize<Vec>({1.0}, exec);
auto beta  = gko::initialize<Vec>({2.0}, exec);

// result = matrix * vector
matrix->apply(lend(vector), lend(result));
// result = alpha * matrix * vector + beta * result
matrix->apply(lend(alpha), lend(vector), lend(beta), lend(result));
```

Fig. 17. Using an SpMV in Ginkgo.

Table 4. The Executor and Matrix Type Flavors Available in Ginkgo

Executor Type	Reference	Omp	Cuda		
Matrix Type	Coo	Csr	Ell	Hybrid	Sellp

Table 5. Matrix Types and the Corresponding Settings Available in Ginkgo

↓ Matrix Type	Matrix settings →
Coo	–
Csr	Strategy = {Automatic, Classical, Cusparse, Load Balance, Merge Path}
Ell	Number of stored elements per row **and** Stride
Hybrid	Number of stored elements per row **and** Stride **and** a Strategy with parameters on {Column Limit, Imbalance Limit, Imbalance Bounded Limit, Automatic}
Sellp	Slice size **and** Stride factor

MATLAB or Python). It can also be analyzed via Ginkgo's online data analysis tool, GPE[4], which was used to generate the plots presented in this work. Compared to other solutions, GPE has the advantage of already providing useful example scripts as a starting point, and does not require the user to install any additional software.

6.3 Ginkgo SpMV Usage Example

The SpMV operation in Ginkgo is quite simple. Consistent with the interface of Ginkgo, an SpMV is seen as a linear operator applicable to a vector (or more generally, a dense matrix). Each of the matrix formats implements an `apply` function that can be *applied* to a vector. An advanced apply is also provided for an advanced scaled multiplication of the vector in question. The sample code in Figure 17 demonstrates how to load data and execute an SpMV with Ginkgo.

Table 4 shows the different options available to the user for the placeholders `!Executor Type!` and `!Matrix Type!`, and Table 5 shows the options available for the `!Matrix Settings!` placeholder.

[4]https://ginkgo-project.github.io/gpe/.

7 SUMMARY AND OUTLOOK

We have presented load-balancing sparse matrix vector kernels suitable for the processing of irregular matrices on GPU architectures. For a hybrid SpMV format complementing a load-balancing kernel for the irregular part with a SIMD-friendly kernel for the balanced matrix contribution, we presented a theoretical analysis aiming at optimizing the matrix splitting. Considering all matrices available in the Suite Sparse matrix database, we analyzed the performance of the distinct kernels and compared with counterparts available in NVIDIA's cuSPARSE library. We also investigated the cost of converting between the standard CSR format and more sophisticated formats. We made all SpMV kernels publicly available in the Ginkgo linear algebra library, and accompany the open-source code with a performance database and a web tool that allows to interactively investigate the performance characteristics.

REFERENCES

Edward Anderson, Zhaojun Bai, Jack Dongarra, Anne Greenbaum, Alan McKenney, Jeremy Du Croz, Sven Hammarling, James Demmel, Christian Bischof, and Danny Sorensen. 1990. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing (Supercomputing'90)*. IEEE Computer Society Press, Los Alamitos, CA, 2–11. http://dl.acm.org/citation.cfm?id=110382.110385.

Hartwig Anzt, Yen-Chen Chen, Terry Cojean, Jack Dongarra, Goran Flegar, Pratik Nayak, Enrique S. Quintana-Ortí, Yuhsiang M. Tsai, and Weichung Wang. 2019. Towards continuous benchmarking: An automated performance evaluation framework for high performance software. In *Proceedings of the Platform for Advanced Scientific Computing Conference (PASC'19)*. ACM, New York, NY, Article 9, 11 pages. DOI: https://doi.org/10.1145/3324989.3325719

Hartwig Anzt, Edmond Chow, and Jack Dongarra. 2016. *On Block-asynchronous Execution on GPUs*. Technical Report 291. LAPACK Working Note.

Hartwig Anzt, Mark Gates, Jack Dongarra, Moritz Kreutzer, Gerhard Wellein, and Martin Köhler. 2017. Preconditioned Krylov solvers on GPUs. *Parallel Comput.* 68 (Oct. 2017), 32–44. DOI: https://doi.org/10.1016/J.PARCO.2017.05.006

Hartwig Anzt, Stanimire Tomov, and Jack Dongarra. 2014. *Implementing a Sparse Matrix Vector Product for the SELL-C/SELL-C-σ Formats on NVIDIA GPUs*. Technical Report ut-eecs-14-727. University of Tennessee.

Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June Donato, Jack Dongarra, Viktor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. 1994. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA.

Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC'09)*. ACM, New York, NY, Article 18, 11 pages. DOI: https://doi.org/10.1145/1654059.1654078

Better Scientific Software (BSSw). Retrieved August 2018 from https://bssw.io/.

Girish Chandrashekar and Ferat Sahin. 2014. A survey on feature selection methods. *Comput. Electr. Eng.* 40, 1 (Jan. 2014), 16–28. DOI: https://doi.org/10.1016/j.compeleceng.2013.11.024

Gianna M. Del Corso. 1997. Estimating an eigenvector by the power method with a random start. *SIAM J. Matrix Anal. Appl.* 18, 4 (Oct. 1997), 913–937. DOI: https://doi.org/10.1137/S0895479895296689

Steven Dalton, Sean Baxter, Duane Merrill, Luke Olson, and Michael Garland. 2015. Optimizing sparse matrix operations on GPUs using merge path. In *2015 IEEE International Parallel and Distributed Processing Symposium*. 407–416. DOI: https://doi.org/10.1109/IPDPS.2015.98

Salvatore Filippone, Valeria Cardellini, Davide Barbieri, and Alessandro Fanfarillo. 2017. Sparse matrix-vector multiplication on GPGPUs. *ACM Trans. Math. Softw.* 43, 4, Article 30 (Jan. 2017), 49 pages. DOI: https://doi.org/10.1145/3017994

Goran Flegar and Hartwig Anzt. 2017. Overcoming load imbalance for irregular sparse matrices. In *Proceedings of the 7th Workshop on Irregular Applications: Architectures and Algorithms (IA3'17)*. ACM, New York, NY, Article 2, 8 pages. DOI: https://doi.org/10.1145/3149704.3149767

Goran Flegar and Enrique S. Quintana-Ortí. 2017. Balanced CSR sparse matrix-vector product on graphics processors. In *Euro-Par 2017: Parallel Processing*, Francisco F. Rivera, Tomás F. Pena, and José C. Cabaleiro (Eds.). Springer International Publishing, Cham, 697–709.

Nicholas Gould and Jennifer Scott. 2016. A note on performance profiles for benchmarking software. *ACM Trans. Math. Softw.* 43, 2, Article 15 (Aug. 2016), 5 pages. DOI: https://doi.org/10.1145/2950048

Max Grossman, Christopher Thiele, Mauricio Araya-Polo, Florian Frank, Faruk O. Alpak, and Vivek Sarkar. 2016. A survey of sparse matrix-vector multiplication performance on large matrices. *CoRR* abs/1608.00636 (2016). arxiv:1608.00636 http://arxiv.org/abs/1608.00636

Desmond Higham and Nick Higham. 2005. *Matlab Guide*. Society for Industrial and Applied Mathematics. DOI: https://doi.org/10.1137/1.9780898717891 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9780898717891

Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. 2019. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, February 16-20, 2019*. 300–314. DOI: https://doi.org/10.1145/3293883.3295712

Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. 2011. Accelerating CUDA graph algorithms at maximum warp. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, February 12–16, 2011*. 267–276. DOI: https://doi.org/10.1145/1941553.1941590

Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop. 2014. A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD Units. *SIAM J. Scientific Computing* 36, 5 (2014), C401–C423. DOI: https://doi.org/10.1137/130930352 arXiv:http://dx.doi.org/10.1137/130930352

Amy N. Langville and Carl D. Meyer. 2012. *Google's PageRank and Beyond: The Science of Search Engine Rankings*. Princeton University Press, Princeton, NJ.

Weifeng Liu and Brian Vinter. 2015. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS'15)*. ACM, New York, NY, 339–350. DOI: https://doi.org/10.1145/2751205.2751209

Duane Merrill and Michael Garland. 2016. Merge-based parallel sparse matrix-vector multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'16)*. IEEE Press, Piscataway, NJ, Article 58, 12 pages. http://dl.acm.org/citation.cfm?id=3014904.3014982.

Duane Merrill, Michael Garland, and Andrew S. Grimshaw. 2015. High-performance and scalable GPU graph traversal. *TOPC* 1, 2 (2015), 14:1–14:30. DOI: https://doi.org/10.1145/2717511

Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. 2010. Automatically tuning sparse matrix-vector multiplication for GPU architectures. In *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers (HiPEAC'10)*. Springer-Verlag, Berlin, 111–125. DOI: https://doi.org/10.1007/978-3-642-11515-8_10

NVIDIA Corp.2017. Whitepaper: NVIDIA TESLA V100 GPU ARCHITECTURE.

[NVIDIA Corporation 2018] NVIDIA Corporation 2018. *NVIDIA CUDA Toolkit* (9.2 ed.). NVIDIA Corporation.

Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1998. The PageRank citation ranking: Bringing order to the Web. In *Proceedings of the 7th International World Wide Web Conference*. Brisbane, Australia, 161–172. citeseer.nj.nec.com/page98pagerank.html.

Tobias Ribizel and Hartwig Anzt. 2019. Approximate and exact selection on GPUs. In *The 9th International Workshop on Accelerators and Hybrid Exascale Systems (AsHES)*, Vol. Available online: http://bit.ly/SampleSelectGPU.

SuiteSparse. 2018. Matrix Collection. Retrieved April 2018 from https://sparse.tamu.edu.

xSDK. Extreme-scale Scientific Software Development Kit. Retrieved August 2018 from https://xsdk.info/.

Received December 2018; revised September 2019; accepted October 2019