Flexible Batched Sparse Matrix-Vector Product on GPUs

Hartwig Anzt
Karlsruhe Institute of Technology, Germany
University of Tennessee, Knoxville, USA
hartwig.anzt@kit.edu

Gary Collins
University of Tennessee, Knoxville, USA
gcollin7@vols.utk.edu

Jack Dongarra
University of Tennessee, Knoxville, USA
Oak Ridge National Laboratory, USA
University of Manchester, Manchester, UK
dongarra@icl.utk.edu

Goran Flegar
Universidad Jaume I, Castellon, Spain
flegar@icc.uji.es

Enrique S. Quintana-Ortí
Universidad Jaume I, Castellon, Spain
quintana@icc.uji.es

ABSTRACT

We propose a variety of *batched* routines for concurrently processing a large collection of small-size, independent sparse matrix-vector products (SpMV) on graphics processing units (GPUs). These batched SpMV kernels are designed to be flexible in order to handle a batch of matrices which differ in size, nonzero count, and nonzero distribution. Furthermore, they support three most commonly used sparse storage formats: CSR, COO and ELL. Our experimental results on a state-of-the-art GPU reveal performance improvements of up to 25× compared to non-batched SpMV routines.

CCS CONCEPTS

• **Mathematics of computing → Mathematical software performance**;

KEYWORDS

Sparse matrix-vector product, batched routines, GPUs.

ACM Reference Format:
Hartwig Anzt, Gary Collins, Jack Dongarra, Goran Flegar, and Enrique S. Quintana-Ortí. 2017. Flexible Batched Sparse Matrix-Vector Product on GPUs. In *Proceedings of ScalA17: 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA17).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3148226.3148230

1 INTRODUCTION

Applying an operator discretized as a sparse matrix in terms of a sparse matrix-vector product (SpMV) is a heavily utilized kernel in many scientific applications. A practical example are Krylov subspace methods, which rely on SpMV to generate the Krylov subspaces used to approximate the solution of linear systems and eigenvalue problems. At the same time, SpMV frequently poses a performance bottleneck of sparse linear algebra algorithms, as this memory-bounded operation is notorious for delivering low fractions of peak performance on current computer architectures. Given the importance of SpMV, significant effort has been spent on finding the best strategy to store sparse matrices and optimizing this kernel for distinct nonzero distributions and hardware architectures, including multicore processors and graphics processing units (GPUs).

In general, scientific applications require the multiplication of a single, large and sparse matrix with an input vector. In this paper, we address a different scenario composed of the multiplication of a large set of "small" sparse matrices with their corresponding vectors. Although this use case is less prominent, it occurs for example in the context of astrophysics simulations. Our goal is to make the community aware that, under these circumstances, replacing a standard routine with a "batched" SpMV kernel often results in significant performance improvements. Following a brief discussion of related work in Section 2, Section 3 presents different strategies for processing a batch of SpMV calls/sparse matrices on GPUs via a number of flexible routines that are designed to handle the most commonly-used sparse matrix storage formats. In Section 4 we then assess the performance of the new kernels, by comparing them against the standard implementations of SpMV in cuSPARSE [13] and MAGMA-sparse [4]. While all kernels are tested on a production line GPU, it can be expected that the kernel design as well as the benefits carry over to other architectures. We conclude in Section 5 with some remarks and an outlook on future research directions.

2 RELATED WORK

2.1 SpMV on manycore architectures

Improving the performance of SpMV on modern architectures is an active field of research. A critical factor is the selection of an appropriate sparse matrix format, which reduces the storage cost by maintaining only the nonzero values but has to keep some additional information in order to derive the location of the elements.

The simplest idea is to explicitly store only the nonzero elements along with the row and column indices (i.e. coordianates) of each element. This coordinate (COO) format [5] allows to determine the original position of any element in the matrix without processing any other entries.

If the elements are sorted row-wise and, for performance reasons, in increasing column-order within each row, the storage cost can be

ScalA17, November 12–17, 2017, Denver, CO, USA

ACM ISBN 978-1-4503-5125-6/17/11…$15.00
https://doi.org/10.1145/3148226.3148230

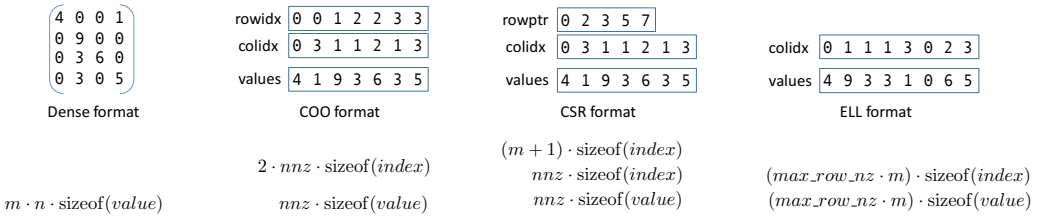

Figure 1: Basic storage formats for an $m \times n$ sparse matrix with nnz nonzeros along with memory consumption.

reduced. The "Compressed Sparse Row" (CSR [5]) format replaces the array containing the row indices with a pointer to the beginning of the distinct rows, reducing the amount of data required to store the row information, at the cost of additional processing necessary to determine the row location of the elements.

While being very popular for manycore architectures in general, the ELL format [6] is particularly suited for GPUs. In this layout, the distinct rows are padded with zeros to ensure they are all of the same length. While this typically increases the storage cost, it removes the need to maintain the row pointers, and enables processing the column-indices (and values) in distinct rows in SIMD fashion. Furthermore, coalescent memory access is attained if the matrix containing the nonzero elements is stored in column-major order.

The three basic formats targeted in our batched kernels are illustrated in Figure 1. In addition to these basic formats, there exist many other variants, which often arise as a combination of the basic formats. For example, the hybrid format stores the matrix partly in ELL and partly in CSR/COO; and the sliced ELL format (SELL-p) [11] chops the matrix into row blocks and stores each block in ELL format.

Related to the storage format is the question of how to parallelize SpMV. The main challenges in this context are: 1) balancing the workload among the distinct cores/threads; and 2) ensuring an efficient access to the matrix entries and the vector values. The second aspect is in particular relevant on NVIDIA GPUs where each memory access reads 128 contiguous bytes of memory [13]. 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 result entry back into main memory. In this paper, we exclusively focus on (batched) SpMV implementations for GPU architectures. For a comprehensive overview about the CUDA programming model and its implications, see [2, 13].

2.2 Batched routines

The development of specialized routines for an operation involving many problems of small size that are pairwise independent, and can thus be handled simultaneously, has recently gained a lot of attention due to their heavy application in machine learning [1]. The motivation for designing these kernels is that the amount of hardware concurrency in manycore processors such as GPUs often exceeds the level of parallelism exploited by conventional routines. Consequently, handling the distinct problems in sequence utilizes only a fraction of the hardware resources and incurs a significant kernel launch overhead. In response to this, there are several efforts among the high performance computing community to standardize the interface for a batched version of the basic linear algebra subprograms (BLAS) and more complex functionality built on top of it [9].

3 DESIGN OF FLEXIBLE BATCHED SPMV KERNELS FOR GPUS

3.1 Flexible batched SpMV

A batched sparse SpMV for scientific applications often comes with some boundary conditions, which allows optimizing the kernel for this specific setting. Examples are situations where all SpMV operations of the batch have:

- the same system size (which, as long as the sparsity pattern does not change too drastically, allows to use explicit zero padding to fix the sparsity pattern);
- the same nonzero-per-row distribution (allows reuse of row pointers/row indices);
- the same nonzero locations (allows reuse of row pointers/row indices and column indices);
- the same values but distinct sparsity patterns (allows reuse of the values);
- the same matrix scaled by a scalar (which allows to rewrite the batched SpMV as a sparse matrix-matrix product and scaling the column of the distinct vectors).

In our case, we design our *flexible* kernels to tackle the most general case: the systems can differ in size, nonzero count, nonzero distribution, and values. This solution offers greater flexibility, and even allows to process a batch of problems coming from several concurrently-running applications.

3.2 GPU kernel design

A central aspect in the design of the batched kernels is the optimization of the access to the vectors. For this purpose we initially read the vectors into shared memory, which significantly reduces the cost of accessing them in the multiplication phase. As shared memory access is limited to the thread block, it is a natural choice to assign one thread blocks to each problem of the matrix batch. We design the batched SpMV kernels with focus on matrices of size up to 1,024 rows. Technically it is possible to process also larger systems, but targeting batches containing small matrices makes this a reasonable design choice.

In this paper we implement and compare six kernels for processing a batch of multiple SpMVs, with matrices stored in either CSR, COO or ELL format. While the properties of the COO and ELL formats inherently result in balanced workload distributions

for each problem, we consider four implementations for the CSR format that use different strategies to balance the work.

Currently, all implementations use one CUDA thread block per problem. Thus, the same amount of resources, in terms of shared memory and number of threads, is allocated to each problem in the batch. We recognize this can result in workload imbalance in-between the distinct problems. However, optimizing the resource allocation to the characteristics of the distinct matrices in the batch remains outside the scope of this work.

3.3 COO

The first listing in Figure 2 (routine `SpMV_COO`) offers a sequential implementation of the `SpMV` kernel based on COO. The natural approach to exploit hardware concurrency in this case is to parallelize the loop traversing the nonzero elements in the matrix (line 2 in the code). This strategy comes with two advantages: 1) the data access to the matrix is coalescent; and 2) the workload is perfectly balanced. The disadvantage is that multiple threads may be assigned to elements located in the same row, and careful synchronization is necessary to ensure the partial sums are handled correctly. In order to ensure correct synchronization, in our batched implementation we combine atomic operations with thread-local partial sums. Moreover, each thread, typically handling multiple elements located in the same row, does not write its partial sum to global memory until all elements of the row are processed. In addition, intra-warp atomic collisions are avoided using warp-local segmented scans before each write. In the remainder of the paper, we use the abbreviation `COO` when referring to the flexible batched `SpMV` routine based on COO.

3.4 CSR

A simple parallelization of `SpMV` based on CSR is obtained by mapping the distinct rows to different threads. This corresponds to parallelizing the outer for-loop in the second listing in Figure 2 (`SpMV_CSR`). This variant was first described for GPUs in [6], under the name CSR-scalar. Even though CSR-scalar does not require any synchronization, it typically suffers from noncoalescent memory accesses for matrices containing more than one nonzero per row. This flaw becomes more apparent with increasing matrix density. Additionally, for unbalanced nonzero distributions, CSR-scalar exhibits severe workload imbalance as, after processing their rows, all threads of a warp remain idle until the thread processing the densest row has completed its work.

To alleviate the issues with CSR-scalar, the authors of [6] proposed an alternative implementation: CSR-vector, which maps each row to one warp (group of 32 threads). This strategy removes two drawbacks at a time: Assigning a warp to a row allows for coalescent memory access; and it improves the workload balancing for irregular sparsity patterns. However, for "very sparse" matrices containing only few nonzeros per row, CSR-vector wastes a significant amount of computational resources.

CSR-smart aims to alleviate the drawbacks of CSR-scalar and CSR-vector while combining their strengths. This kernel, recently implemented in the CUSP library [7], allocates a "vector" of threads (a subset of a warp) to process each row. The number of threads in a vector is determined at runtime. The strategy extracts the average number of nonzeros per row from the input matrix, and sets the vector size to the smallest power of two equal to or larger than this number (up to 32). Although this approach may render some workload imbalance for irregular sparsity patterns, it resolves both the CSR-scalar's noncoalescent memory reads as well as the problem of idle resources in CSR-vector's. In our batched CSR-smart kernel we calculate the vector length for each problem individually. This allows that thread blocks launched by the same kernel process different matrices of the batch with different vector lengths.

For convenience, we will use the abbreviations `CSR_scal`, `CSR_vec` and `CSR_smart` to refer to the flexible batched implementations of the CSR-scalar, CSR-vector and CSR-smart kernels, respectively.

3.5 CSR-I

A reorganization of the `SpMV` loops as shown in the third listing of Figure 2 (`SpMV_CSRI`) can yield a perfectly balanced implementation for the CSR format [10]. Concretely, by parallelizing the outer loop of this variant (line 4), each warp is assigned the same percentage of nonzero elements. This mimics `COO`, and makes CSR-I especially appealing for irregular sparsity patterns (hence the "I" in the name). However, other CSR variants can be expected to outperform CSR-I [10] for regular sparsity patterns as the latter: 1) requires atomic operations for synchronizing the distinct warps writing to the same output vector location; 2) exhibits higher arithmetic intensity to minimize the amount of atomic collisions; 3) potentially reads some elements of the row pointer multiple times if the majority of rows have few nonzeros; and 4) requires a preprocessing step to determine the starting value of the `row` variable for each warp. In the batched CSR-I implementation (`CSRI`), the preprocessing step occurs once for each matrix of the batch, while every invocation of the `CSRI` kernel on the same matrix batch will reuse this information. As many applications require a high number of `SpMV` calls, (e.g., iterative solvers,) we do not account for the runtime of this preprocessing step in the performance measurements in Section 4.

In the original non-batched CSR-I implementation, the optimal level of thread concurrency is selected depending on the (single) problem characteristics and the hardware resources. A straightforward approach in the batched CSR-I implementation distributes the resources equally across the problems in the batch. However, in the limit of increasing batch size, this strategy assigns one warp to each problem. At the same time, the amount of shared memory required per problem remains constant, i.e. equal to the size of the problem. The shared memory thus becomes the factor that constrains the number of thread blocks which can run concurrently on each multiprocessor of the GPU. Therefore, to prevent low occupancy, the number of threads assigned to each problem is not allowed to drop below the point where the shared memory becomes the occupancy-limiting factor.

3.6 ELL

The implementation of the flexible batched `SpMV` kernel based on the ELL format (we denote the batched kernel "`ELL`") is an immediate derivation of the standard ELL `SpMV` from [6]: Each thread of the block processes a different row, forming the partial sums of its row in thread-local memory, while the thread block traverses the column indices and values from left to right. After completion, the

```
void SpMV_COO(int nnz, int *rowidx, int *colidx, float *val, float *x, float *y) {
  for (int i = 0; i < nnz; ++i) {
    y[ rowidx[i] ] += val[i] * x[ colidx[i] ];
  }
}
```

```
void SpMV_CSR(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {
  for (int i = 0; i < m; ++i) {
    for (int j = rowptr[i]; j < rowptr[i+1]; ++j)
      y[i] +=  val[j] * x [ colidx[j] ];
  }
}
```

```
const int T = thread_count;
void SpMV_CSRI(int m, int *rowptr, int *colidx, float *val, float *x, float *y) {
  int row = -1, next_row = 0, nnz = rowptr[m];
  for (int k = 0; k < T; ++k) {
    for (int i = k*nnz / T; i < (k+1)*nnz / T; ++i) {
      while (i >= next_row) next_row = rowptr[++row+1];
      y[row] += val[i] * x[ colidx[i] ];
}}}
```

```
void SpMV_ELL(int m, int max_nnz, int *colidx, float *val, float *x, float *y) {
  for (int i = 0; i < m; ++i) {
    for (int j = 0; j < max_nnz; ++j) {
      y[i] += val[i+j*m] * x[ colidx[i+j*m] ];
    }
  }
}
```

Figure 2: Sequential C implementations of basic SpMV algorithms.

intermediate results are written into the output vector locations in global memory. Conversely to the standard ELL kernel, the values of the input vector are read from shared memory, and the thread block size is adjusted to the matrix size such that each thread block handles one problem of the matrix batch.

4 PERFORMANCE EVALUATION

4.1 Experiment setup

For the performance analysis, we use the following test benchmark consisting of 32 matrices from the SuiteSparse matrix collection [8]: pivtol, tomography, west0989, G45, GD02_a, Si2, ck656 ex25, jgl011 lock_700, fs_541_1, mbeacxc, dwt_918, mcfe, dwt_607, rbsa480, gr_30_30, bcsstk34, cage8, ex27, can_838, bcsstm34, USAir97, bp_1600, rotor2, msc00726, nos3, G2, dwt_992, ex2, tols90, bcsstk02. We focus on square matrices of order up to 1024, with real entries, including a variety of problems to cover a large spectrum. Concretely, this subset contains matrices with size $n \in [11, 1015]$, number of nonzeros $nnz \in [76, 38352]$, and $nnz/n \in [3.0, 66.0]$. The specific operation we target is $y := A \cdot x + y$, which requires $2 \cdot nnz$ floating-point arithmetic operations (flops), and allows for much flexibility in terms of scaling y before the operation (e.g., scaling y with 0 to compute $y = A \cdot x$).

The performance evaluation is split into three parts. In the first experiment, we quantify the performance advantages that the flexible batched routines provide over the standard SpMV kernels when processing a homogeneous collection (batch) consisting of an increasing number of identical matrices.

In the second part, we compare the batched kernels against each other, using homogeneous batches consisting either of the problems from the test benchmark, or custom-engineered matrices where we control the density and nonzero distribution. Although it is possible to write a much more efficient kernel for this problem setting (which, in particular, reuses the information about the problem size and the nonzero locations), we argue that this experiment is useful to extract information about which format and kernel to choose for batches containing similar problems.

In the third part of the experimental analysis, we consider batches comprising problems with different characteristics. First, we look into settings where all matrices in the batch are of similar size but differ in the sparsity pattern. For this purpose, we select 12 matrices from the test benchmark of order 800–1,024, and create the batch by appending these problems in random order. Second, we consider batches containing any of the matrices in the test benchmark.

All experiments were conducted on the compute nodes of the PizDaint supercomputer at the Swiss National Computing Centre (CSCS). Although irrelevant for the performance analysis, the host contains an Intel E5-2690 v3 (Haswell) processor with 12 cores. All computations were executed by the NVIDIA Tesla P100 GPU (compute capability 6.0), using double precision arithmetic, for which NVIDIA lists a peak performance of 5.3 TFLOPs (10^{12} flops/second). The P100 is equipped with 16 GB of main memory that are accessed at a theoretical peak bandwidth of 732 GB/s. Using NVIDIA's CUDA toolkit version 8.0, we designed the flexible batched routines to be integrated into the MAGMA-sparse software library [3]. MAGMA-sparse was also used as experiment ecosystem, and provided the standard SpMV reference implementations.

4.2 Experimental results

We first quantify the benefits of leveraging custom-designed batched kernels over the standard SpMV routines when processing batches of small matrices. For this purpose, we select 12 problems with between 800 and 1,024 unknowns from the test benchmark, and

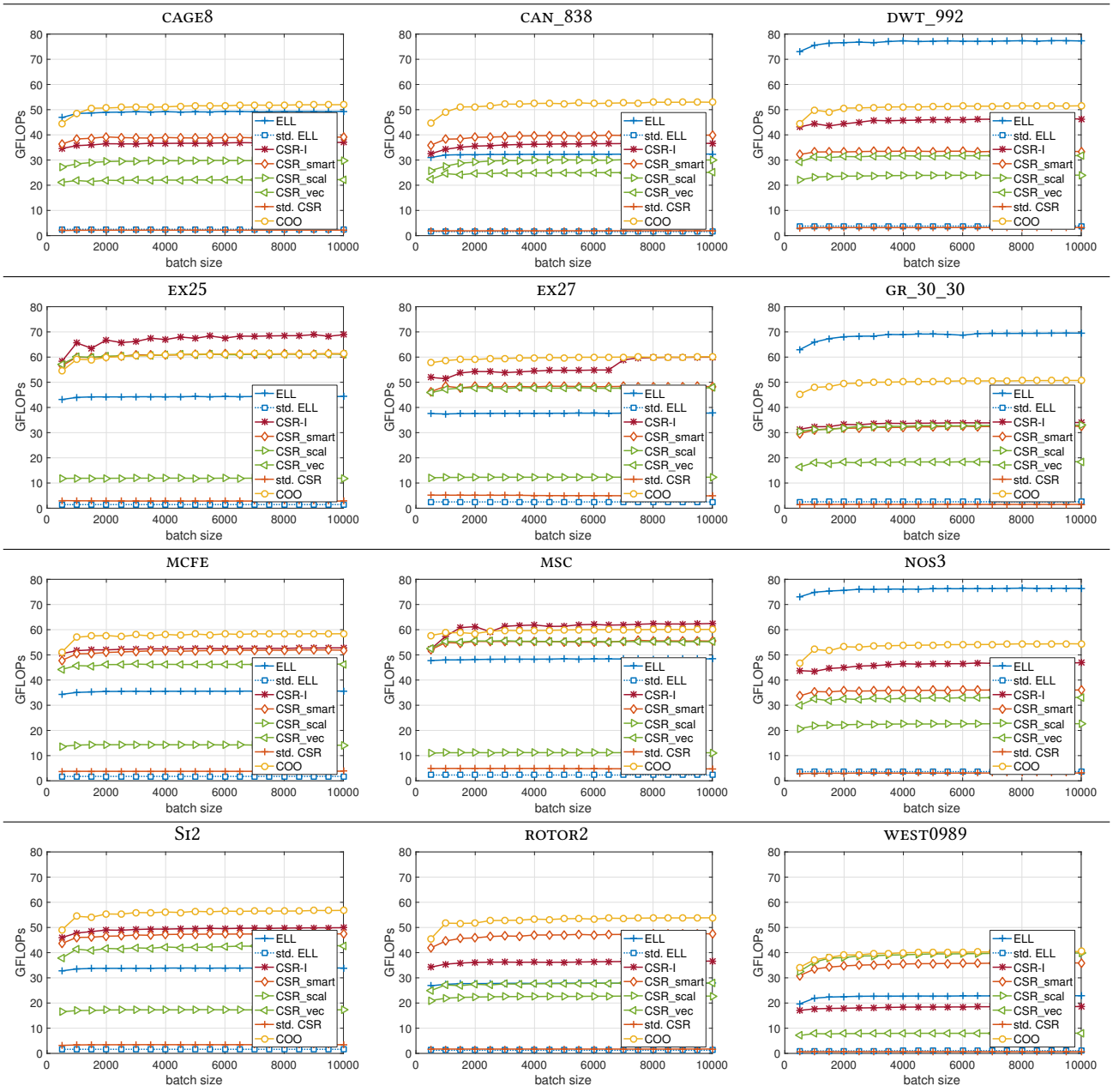

Figure 3: Performance of the standard and the flexible batched SpMV routines for homogeneous batches.

create homogeneous batches. In Figure 3 we visualize the performance achieved by the standard SpMV routines versus the flexible batched SpMV kernels when processing batches of increasing size. In order to process a batch with multiple matrices via the standard SpMV kernels, we loop over the kernel invocations. For the standard CSR kernel, MAGMA-sparse simply interfaces to NVIDIA's cuSPARSE library [13]; the standard ELL kernel is the implementation available in MAGMA-sparse [4].

The results of this experiment reveal that the performance of the standard SpMV kernels never exceeds 5 GFLOPs. Furthermore, although there is no clear winner among the batched SpMV kernels, they all complete the operation at least 10× faster than their standard counterparts. For balanced problems, such as dwt992, gr_30_30, and nos3, the performance of the ELL kernel surpasses 70 GFLOPs, a rate which is unmatched by any other kernel. At the other end of the spectrum, the CSRI kernel achieves very good

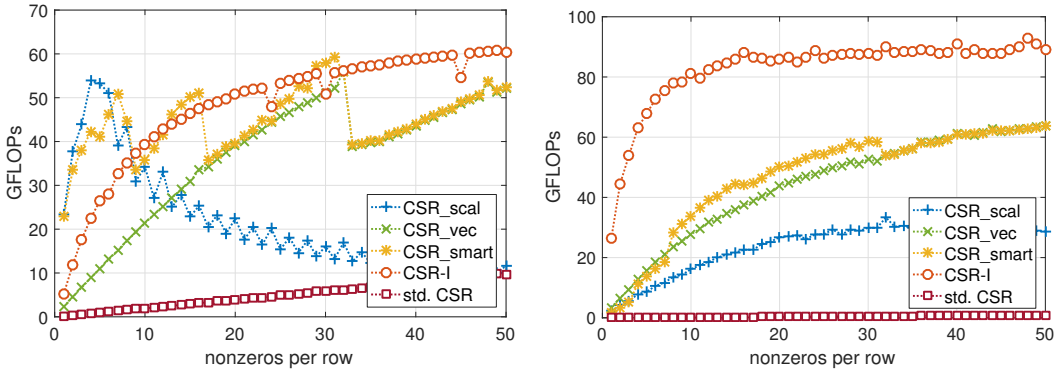

Figure 4: Performance of the standard and flexible batched CSR-based SpMV routines for a homogeneous batch consisting of 1000 square matrices of order 1024 with controlled density and nonzero distribution. The nonzeros are either distributed equally among the rows (left) or accumulated in few rows (right).

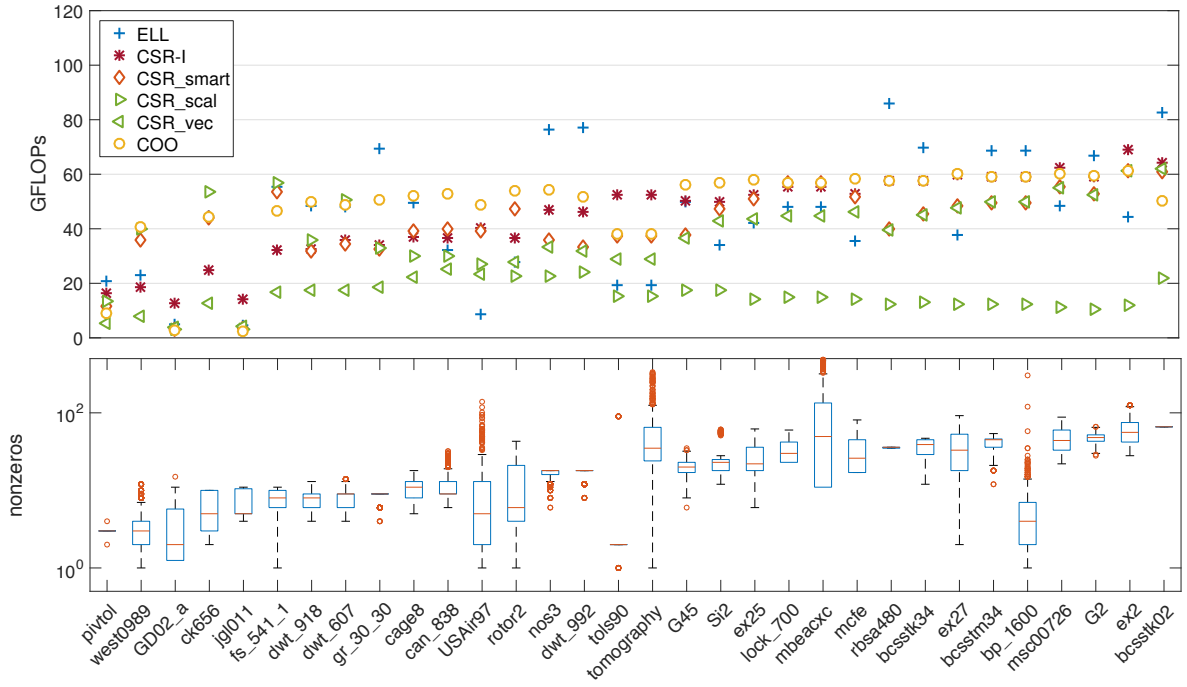

Figure 5: Performance of the flexible batched SpMV routines for all matrices in the test benchmark ordered in increasing nonzero-per-row ratio.

performance for unbalanced problems containing many nonzero elements, such as EX25 and MSC. In these cases, the ELL kernel suffers from a significant zero-padding overhead. For the very sparse problem WEST0989 the fastest options are CSR_scal and COO. Overall, we acknowledge that the COO kernel achieves very good performance across the complete test suite. In addition to being the fastest option in most of the cases, the COO kernel is the second-best choice in all remaining cases where a different format is superior.

Next, we focus on the CSR format, for which we developed four kernels that differ in how they balance the workload. For reference, we also include the standard CSR SpMV from NVIDIA's cuSPARSE in this analysis. For the next experiment we generate a homogeneous batch containing 1,000 square matrices of size 1,024 and vary the density and nonzero distribution. We analyze the performance in relation to the average number of nonzero elements per row. On the left-hand side of Figure 4, we test a balanced nonzero distribution. The results show that the batched CSR_scal offers very good performance for low nonzero-per-row ratios. This comes from the fact that the data reads are mostly coalescent. The performance of CSR_scal drops in case there are more than 4 nonzeros

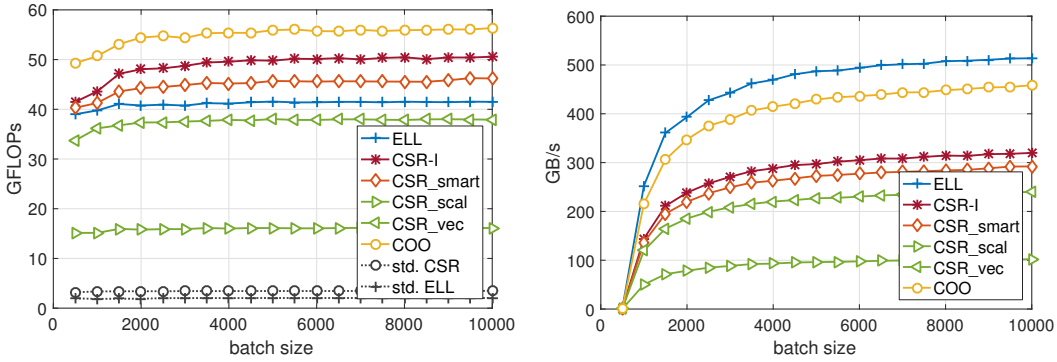

Figure 6: Performance (left) and sustained memory bandwidth (right) of the flexible batched SpMV routines applied to a heterogeneous batch consisting of a random compilation of the matrices analyzed in Figure 3.

per row, while the performance of CSR_vec continues to improve. CSR_smart is a good trade-off between CSR_scal and CSR_vec, as it provides between 35 and 60 GFLOPs for most of the tested scenarios. The sequence of standard CSR SpMV calls never delivers more than 5 GFLOPs. Especially for increasing nonzero count, the performance of CSRI is competitive or even superior to CSR_smart. However, for low nonzero-per-row ratios, CSR_scal and CSR_smart are faster. Conversely, on the right-hand side plot of Figure 4, CSRI gives the best performance in all cases. This is expected as this experiment configures a batch of extremely unbalanced matrices with the nonzeros accumulated in a few rows. We recall that the good performance that CSRI achieves for this problem comes at the price of a preprocessing step to calculate the balancing information.

In Figure 5 we analyze the performance of the flexible batched SpMV kernels achieved for a homogeneous batch of 10000 matrices of the test benchmark. The matrices are ordered in the x-axis according to increasing nonzero-per-row ratio. A larger value of this parameter makes the CSR_scal kernel less attractive while the performance of CSR_vec increases with the density. CSRI and CSR_smart outperform CSR_vec and CSR_scal in most cases. None of the CSR-based kernels is competitive to the COO kernel, which can be identified as overall winner in this experiment. Only for balanced matrices, the ELL kernel outperforms all other competitors. However, the performance of ELL is very problem-dependent, and for unbalanced nonzero distributions, it yields low performance.

We now turn to heterogeneous batches. First, we compose a batch with the matrices we analyzed in Figure 3. These matrices are very different in their nonzero pattern, but they all share similar size (800–1024 rows/columns). In Figure 6 we show performance (left) and bandwidth (right) for the distinct batched SpMV kernels. In the latter, we also account for explicit zeros read into the multiprocessors. We notice that the ELL kernel achieves memory access rates beyond 500 GB/s, which is about 70% of the theoretical peak [12]. COO attains around 450 GB/s; and CSR_smart and CSRI deliver around 300 GB/s. However, the memory bandwidth is not the relevant factor in terms of runtime performance and, although the ELL kernel was the performance winner for selected problems in Figure 3, it only achieves about 40 GFLOPs in this experiment. Higher throughput is achieved by CSR_smart (45 GFLOPs), CSRI (50 GFLOPs), and COO (55 GFLOPs). We mention that it is possible to improve the ELL kernel by using the sliced ELL format instead [4]. There, the overhead introduced from zero-padding is reduced by enforcing the same nonzero count only for those rows located in the same block. However, we refrain from this optimization step as we expect the benefit to be moderate: the height of the distinct row-blocks should at least match the warp size (32), which is relatively large compared to the small matrices we focus on.

Finally, we consider batches containing all of the matrices in the test benchmark, arranged in random order. Figure 7 (right) shows that the ELL kernel sustains a memory access rate around 500 GB/s. At the same time, the performance drops from 40 to 30 GFLOPs, which is likely due to the large number of small and unbalanced test matrices in the batch. Conversely, the performance of the other formats is not affected, and CSR_smart, CSRI and COO exceed 45, 50 and 55 GFLOPs, respectively.

We conclude that across all sparsity formats, the COO kernel achieves the best performance for heterogeneous batches. If the batch consists of balanced matrices only, the ELL kernel becomes the preferred choice, achieving up to 80 GFLOPs. In a one-touch-only scenario where all matrices are stored in CSR format, the CSR_smart kernel is the best option. If a preprocessing step is justified by a high number of kernel invocations, the CSRI kernel is much faster for batches consisting of unbalanced problems.

5 SUMMARY AND OUTLOOK

We have developed and implemented a set of flexible batched SpMV kernels that accommodate the CSR, COO and ELL sparse matrix storage formats. The routines can efficiently process matrix batches where each problem is different in terms of size, nonzero count and nonzero pattern. Although the performance of the distinct kernels is very problem-dependent, our experimental results on an NVIDIA P100 GPU, using batches comprising very different matrices, reveled that the developed kernels based on COO and CSR are able to sustain a performance of about 50 GFLOPs. This corresponds to a 25× speed-up compared to the use of a sequence of invocations to standard implementations of SpMV.

In the future we plan to further optimize the formats by determining the resources allocated to the distinct problems based on

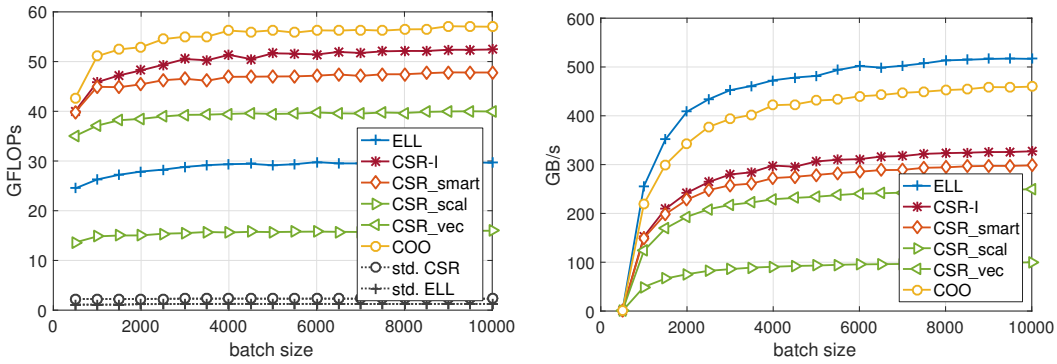

Figure 7: Performance (left) and sustained memory bandwidth (right) of the flexible batched SpMV routines applied to a heterogeneous batch consisting of a random compilation of all the matrices in the test benchmark.

the matrix characteristics. Furthermore, we want to extend the performance assessment to also account for the energy usage, and compare resource efficiency with other manycore architectures that feature a more sophisticated cache hierarchy.

ACKNOWLEDGMENTS

This work was partly funded by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Award Number DE-SC-0010042. H. Anzt was supported by the "Impuls und Vernetzungsfond" of the Helmholtz Association under grant VH-NG-1241. G. Flegar and E. S. Quintana-Ortí were supported by projects TIN2014-53495-R of the Spanish *Ministerio de Economía y Competitividad* and the EU H2020 project 732631 OPRECOMP.

The authors want to acknowledge the access to the PizDaint supercomputer at the Swiss National Supercomputing Centre granted under the project #d65.

REFERENCES

[1] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. 2016. *Performance, design, and autotuning of batched GEMM for GPUs.* Springer International Publishing, Cham, 21–38.

[2] H. Anzt, E. Chow, and J. Dongarra. 2016. *On block-asynchronous execution on GPUs.* Technical Report 291. LAPACK Working Note.

[3] Hartwig Anzt, Mark Gates, Jack Dongarra, Moritz Kreutzer, Gerhard Wellein, and Martin Köhler. 2017. Preconditioned Krylov solvers on GPUs. *Parallel Comput.* (2017), –. https://doi.org/10.1016/j.parco.2017.05.006

[4] H. Anzt, S. Tomov, and J. 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.

[5] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. 1994. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.* SIAM, Philadelphia, PA.

[6] Nathan Bell and Michael Garland. 2008. *Efficient Sparse Matrix-Vector Multiplication on CUDA.* NVIDIA Technical Report NVR-2008-004. NVIDIA Corp.

[7] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. 2014. CUSP: Generic Parallel Algorithms for Sparse Matrix and Graph Computations. (2014). http://cusplibrary.github.io/ Version 0.5.0.

[8] T. A. Davis and Y. Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. on Mathematical Software* 38, 1 (2011), 1–25. https://doi.org/10.1145/2049662.2049663

[9] J. Dongarra, I. S. Duff, M. Gates, A. Haidar, S. Hammerling, J. Higham, J. Hogg, P. Valero-Lara, D. Relton, S. Tomov, and M. Zounon. 2016. *A Proposed API for Batched Basic Linear Algebra Subprograms.* Technical Report 2016.25. The University of Manchester, ISSN 1749-9097.

[10] Goran Flegar and Enrique S. Quintana-Ortí. accepted. Balanced CSR Sparse Matrix-Vector Product on Graphics Processors. In *EuroPar 2017.*

[11] 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. https://doi.org/10.1137/130930352 arXiv:http://dx.doi.org/10.1137/130930352

[12] NVIDIA. 2016. Whitepaper: NVIDIA Tesla P100. WP-08019-001_v01.1. (2016).

[13] NVIDIA. 2017. *CUDA toolkit V8.0.*