Overcoming Load Imbalance for Irregular Sparse Matrices

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

Hartwig Anzt
Karlsruhe Institute of Technology, Germany
University of Tennessee, Knoxville, USA
hanzt@icl.utk.edu

ABSTRACT

In this paper we propose a load-balanced GPU kernel for computing the sparse matrix vector (SpMV) product. Making heavy use of the latest GPU programming features, we also enable satisfying performance for irregular and unbalanced matrices. In a performance comparison using 400 test matrices we reveal the new kernel being superior to the most popular SpMV implementations.

CCS CONCEPTS

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

KEYWORDS

Sparse Matrix Vector Product, GPU, Irregular access pattern, Load Balancing

ACM Reference format:
Goran Flegar and Hartwig Anzt. 2017. Overcoming Load Imbalance for Irregular Sparse Matrices. In *Proceedings of IA^3 '17: Seventh Workshop on Irregular Applications: Architectures and Algorithms, Denver, CO, USA, November 12–17, 2017 (IA^3 '17),* 8 pages.
https://doi.org/10.1145/3149704.3149767

```
// input: A, x, y
// outut: y = y+A*x
void coo_spmv(int nnz, const int *rowidx,
              const int *colidx, const float *val
              const float *x, float *y)
{
  for (int i = 0; i < nnz; ++i) {
    y[rowidx[i]] += val[i] * x[colidx[i]];
  }
}
```

Figure 1: COO SpMV kernel design.

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 are the Krylov subspace methods relying on the SpMV kernel to generate the Krylov subspaces in which the solutions to linear and eigenvalue problems are approximated. At the same time, the SpMV is a frequent bottleneck in complex applications, as it is notorious for sustaining low fractions of peak processor performance. This is partly due to the low arithmetic intensity making the SpMV kernel memory bound on literally all modern architectures, the access overhead induced by storing only the nonzero elements in the matrix, the (in many cases random) access to the input vector. Given the importance of this building block, 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 [3, 6, 11]. For sparse matrices where the nonzeros are distributed in a very structured fashion, it is often possible to derive problem-tailored storage formats, like, e.g., the DIA format for matrices with a tridiagonal structure [6]. A similar situation is given 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 for, e.g., social network representations. For these irregular problems, standard parallelization strategies, like assigning rows to the parallel resources, inevitably result in heavy load imbalance. Furthermore, unstructured sparsity patterns often promote random memory access to the vector values.

In this paper we present a GPU implementation of the sparse matrix-vector product (Section 3) that addresses the challenge of overcoming the load imbalance in unstructured matrices. The kernel is based on the coordinate (COO) format [6], leverages the latest features of the CUDA programming model, and succeeds in achieving high performance for unstructured matrices. In a comprehensive evaluation in Section 4 we identify the developed kernel as competitive or superior to the existing routines. Prior to presenting the new implementation, in Section 2 we review existing efforts for optimizing the sparse matrix vector product on manycore architectures.

2 RELATED WORK

2.1 Sparse Matrix Formats

In the BLAS and LAPACK [1] standard for dense linear algebra, matrices are stored as a sequence of columns, with each column stored as an array of its elements. This allows to easily locate or identify any matrix entry in memory. For matrices where most elements are zero, which is typical for, e.g., finite element discretizations, storing all matrix entries results in significant storage overhead. The computational cost of a matrix vector product increases as well, as a result of explicitly multiplying the zero entries with vector values. Sparse matrix formats aim at reducing the memory footprint (and computational) cost by storing only the nonzero matrix

IA^3 '17, November 12–17, 2017, Denver, CO, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5136-2/17/11...$15.00
https://doi.org/10.1145/3149704.3149767

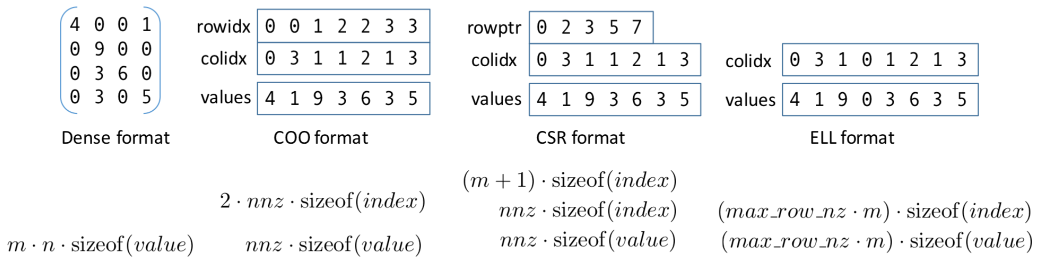
4 0 0 1
0 9 0 0
0 3 6 0
0 3 0 5
Dense format
rowidx 0 0 1 2 2 3 3
colidx 0 3 1 1 2 1 3
values 4 1 9 3 6 3 5
COO format
rowptr 0 2 3 5 7
colidx 0 3 1 1 2 1 3
values 4 1 9 3 6 3 5
CSR format
colidx 0 3 1 0 1 2 1 3
values 4 1 9 0 3 6 3 5
ELL format
$m \cdot n \cdot \text{sizeof}(value)$
$2 \cdot nnz \cdot \text{sizeof}(index)$
$nnz \cdot \text{sizeof}(value)$
$(m+1) \cdot \text{sizeof}(index)$
$nnz \cdot \text{sizeof}(index)$
$nnz \cdot \text{sizeof}(value)$
$(max\_row\_nz \cdot m) \cdot \text{sizeof}(index)$
$(max\_row\_nz \cdot m) \cdot \text{sizeof}(value)$

Figure 2: Different storage formats for a sparse matrix of dimension $m \times n$ containing nnz nonzeros along with the memory consumption.

values. Some formats additionally store a moderate amount of zero elements to enable faster processing when computing matrix vector products. 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 explicitly store only the nonzero elements, along with the row and column index of each element. This storage format, known as coordinate (COO [5]) 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 Compressed Sparse Row (CSR [5]) format can replace the array containing the row indexes with a pointer to the beginning of the distinct rows. While this reduces the data volume, the CSR format requires extra processing to determine the row location of a certain element.

On SIMD architectures, a performance-relevant aspect is to have uniform operations across the SIMD unit. This makes the ELL format [7] attractive for these architectures: In this format, each row is compressed 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 which stores the column position of each entry in the compressed matrix. While typically increasing the storage cost compared to the CSR format, this removes the need for explicitly storing the row pointers. Furthermore, the column indexes (and values) in the distinct rows can be processed in SIMD fashion. Coalescent (SIMD-friendly) memory access is enabled if the value and column index matrices are stored in column-major order.

To reduce the memory overhead, the ELL format can be truncated to a version where 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. This “sliced ELL” (SELL-p [10]) format can be viewed as splitting the matrix into row-blocks and storing each block in ELL format. The formats discussed in this section are visualized in Figure 2.

Aside from these basic formats, there exist variants which arise as combinations of these basic formats: e.g., the hybrid format which stores the matrix partly in ELL and partly in CSR or COO.

2.2 SpMV on manycore architectures

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 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 resulting vector entry back into main memory.

The standard approach of parallelizing the CSR, ELL and SELL-p formats is to distribute the rows among distinct threads (or groups of threads) [7, 12]. For the CSR format, this works fair 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 the rows [9]. The SpMV kernel we present in this paper is based on the COO format, which comes with the advantage of the row index of an element being readily available.

3 DESIGN OF THE COO SPMV GPU KERNEL

The specific SpMV operation we target in this paper is $y := A \cdot x + y$, for $A \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$. This routine updates the vector y by adding the product of the sparse matrix A and the vector x, and allows for flexibility in terms of scaling y prior to the operation (i.e. scaling y with 0 to compute $y = A \cdot x$). It comprises $2 \cdot nnz$ arithmetic operations (with nnz being the number of nonzero elements in A).

In the rest of this section we describe the design of the COO SpMV kernel we propose for manycore architectures. Subsection 3.1 presents the general algorithmic idea, without introducing hardware-specific optimizations. There, we only assume the target device to be a shared memory architecture, with a relatively large number of computational elements (cores) and support for atomic addition of floating point values. The last assumption is required to resolve race conditions which can occur if multiple computational elements are assigned to the same matrix row, i.e. contribute to the computation of the same entry in the output vector. Furthermore, we assume that the memory design favours data locality over random data access, which is true for virtually all modern hardware. Subsection 3.2 describes the hardware-specific optimizations we employ when realizing the COO SpMV algorithm on NVIDIA GPUs using the CUDA programming model.

3.1 COO SpMV

The most natural approach to exploit hardware concurrency in an SpMV routine based on the COO format is to parallelize the loop traversing the nonzero elements in the matrix (line 7 in Figure 1) by splitting the workload into similar-sized subsets, and distributing those among the parallel resources. This corresponds to assigning a contiguous "chunk" of the array containing the matrix values (along with the corresponding chunks of column and row index arrays), to each computational element. While it is possible to use non-contiguous chunks (e.g., distribute the data in round-robin fashion) this could break data locality, resulting in performance loss. To ensure load balancing among different computational elements, all chunks should be of similar size. In addition, to reduce the number of memory transactions from main memory, it is important to aim for aligned memory access and to use every data element brought into cache. This is crucial when implementing the memory-bound SpMV kernel as its performance is largely dependent on the data access efficiency. Therefore, assuming that all three arrays describing the matrix in COO format are aligned in memory, each chunk should start at a cache line boundary, and ideally comprise an integer number of cache lines.

In summary, efficient data access and optimal load balancing imposes two restrictions on chunk sizes: 1) each chunk should be an integer multiple of the cache line size, and 2) the sizes of any two chunks should differ by at most one cache line.

While the above strategy yields a perfect distribution of the input matrix A, which typically comprises the majority of the data, it has several implications we discuss next.

The core operation (line 8 in Figure 1) of COO SpMV 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. Since multiple computational elements can operate on matrix elements which have the same `rowidx` values (which corresponds to matrix elements located in the same row), this update is prone to race conditions. To resolve write conflicts, the (fused) multiply-add can be replaced by multiplication, followed by an atomic addition of the result into the correct position of vector y. This, however, requires more arithmetic instructions than the original approach, and uses a large number of expensive atomic operations, which may cause significant overhead in case of atomic collisions (i.e. multiple atomic operations requesting the same data entity at the same time). To alleviate the problem, we accumulate the results of several iterations of the for loop with the same update destination in registers private to the computational element. If the `rowidx` value of the next data element is different to what was processed previously, the accumulated results are written back to main memory using an atomic addition.

In this strategy, to decrease the number of atomic operations, (and increase the amount of computation handled in registers,) the three arrays comprising the matrix data should be sorted with respect to the increasing `rowidx` values. This will ensure that each computational element performs only one atomic operation per `rowidx` value, while also ensuring data locality for vector y (which can be beneficial on hardware that implements atomic operations in a shared cache file, like used in NVIDIA and AMD GPUs). Finally, 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 CSC format, where structured access to the vector x is ensured at the price of complicated access to the vector y.)

3.2 CUDA realization of COO SpMV

We realize the specific implementation of the general approach described in the previous subsection using the CUDA programming model. This model has all of the features required by the introductory paragraph of this section, with atomic instructions being the only critical component. While older generations of NVIDIA GPUs emulate double precision atomics in software (by using 64-bit atomic CAS), the new Pascal architecture offers native support for double precision atomics.

A naive implementation assigns one chunk of memory to each CUDA thread. This, however, inevitably results in non-coalescent reads to the matrix A, which is detrimental to performance. To ensure coalescent memory access, each chunk should be assigned to one warp (a group of 32 threads), and a thread i of the warp should read elements at positions $32j + i$, $j = 0, 1, \ldots$ of the chunk. (This is the motivation to use a more platform-agnostic term "computational element" in the previous subsection, as the terms "thread" or "core" may have different meanings in distinct programming models and/or hardware.)

The main problem with this "warp-level approach" is the use of atomic operations. If each thread in the warp attempts to issue an atomic addition whenever it progresses to the next row, this will cause a large number of atomic collisions, since all the threads in a warp execute in lockstep (i.e. perfectly synchronized). A workaround is to conclude each iteration of the loop (line 7–9 in Figure 1) with a "warp vote function" in which all threads in a warp decide whether there is at least one of them that needs to write its results into global memory. If such a thread is identified, all threads collectively execute a warp-level segmented scan operation on their private registers, with segments defined by the distinct values of the `rowidx` array currently processed by the warp. The segmented scan (as opposed to a simple reduction) is required to ensure correct operation if the warp handles multiple rows of the matrix. For each of the handled `rowidx` values, the segmented scan determines the thread with the lowest thread index among all the threads that operate on this row. Then, these threads accumulate the partial sums present in their registers, and each of the threads with the lowest index will issue a global atomic addition before it progresses to a different `rowidx` value. This strategy 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

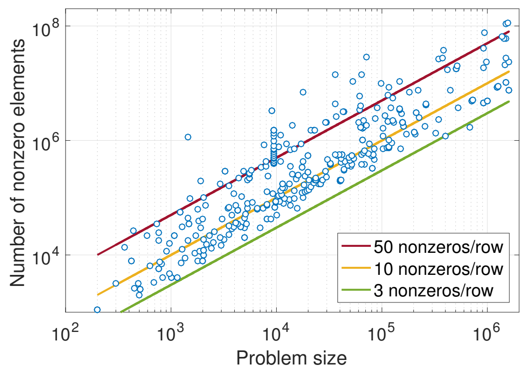

Figure 3: Nonzero count vs. size for the considered test matrices. For convenience we added density baselines for 3 nnz/row, 10 nnz/row, and 50 nnz/row.

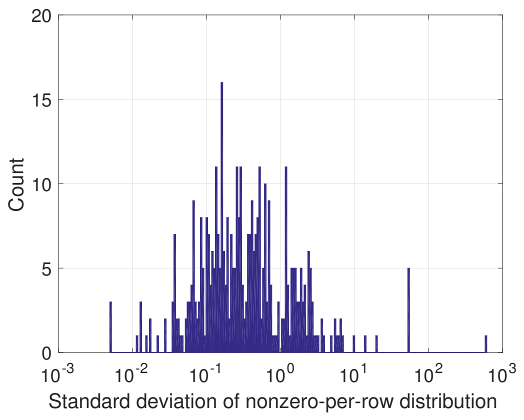

Figure 4: Histogram for the (standard deviation / avg) of the nonzero-per-row metric. Few problems have a higher standard deviation than 10^2.

2) distinct warps are not perfectly synchronized. The segmented scan approach radically reduces the number of atomic collisions, however 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 this approach of avoiding intra-warp atomic collisions was also used to construct a balanced CSR SpMV kernel in Flegar et al. [9].

An additional optimization on NVIDIA GPUs is related to the choice of the number of chunks. In contrast to the classical latency-minimizing CPU hardware, enabled by sophisticated cache hierarchy, NVIDIA GPUs use a latency-hiding approach where the computational units are oversubscribed with warps. The intention of having a larger number of warps active is to quickly switch in-between them to cover memory latency [2]. 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. 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., $\omega = 2$ 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). The oversubscribing parameter is subject to hardware- and problem-specific optimization, and we experimentally identify reasonable choices in Section 4.

4 PERFORMANCE ASSESSMENT

4.1 Test matrices

For the experimental performance analysis, we use a set of 400 matrices from the SuiteSparse matrix collection [8]. This collection comprises a large number of matrices that differ in the algebraic field (real, complex, pattern), the shape (square, rectangular), and matrix-specific characteristics such as size and nonzero pattern. For the performance assessment we focus on real, square matrices that have a pairwise different nonzero pattern. In Figure 3 we visualize the size and nonzero count of the chosen test matrices. In order to quantify the imbalance of the nonzero distribution of a matrix, we use the standard deviation of the nonzero-per-row metric, see Figure 4.

4.2 Experiment setup

All experiments were conducted on the GPU-accelerated compute nodes of the PizDaint supercomputer at the Swiss National Computing Centre (CSCS). Although irrelevant for the performance analysis, we mention that the host composes of an Intel E5-2690 v3 processor (codename Haswell) with 12 cores running at 2.6 GHz. All computations are executed by the NVIDIA Tesla P100 GPU (compute capability 6.0) for which NVIDIA lists a double precision peak performance of 5.3 TFLOPs (10^{12} floating point operations per second). We use double precision in all experiments. The P100 is equipped with 16 GB main memory, which is accessed at a theoretical bandwidth of 732 GB/s. Using the bandwidth test that ships with CUDA 8.0 (and that puts equal pressure on memory reads and memory writes) we were able to achieve 497 GB/s. Using NVIDIA's CUDA toolkit version 8.0, we design the COO kernel to integrate into the MAGMA-sparse software library [4]. MAGMA-sparse is also used as experiment ecosystem, and provided the SpMV reference implementations. Specifically, the reference implementations are:

CSR The CSR-based SpMV kernel we consider is part of NVIDIA's cuSPARSE library.

CSR5 The CSR5 SpMV kernel is based on modifying the CSR format for achieving higher performance. The implementation is

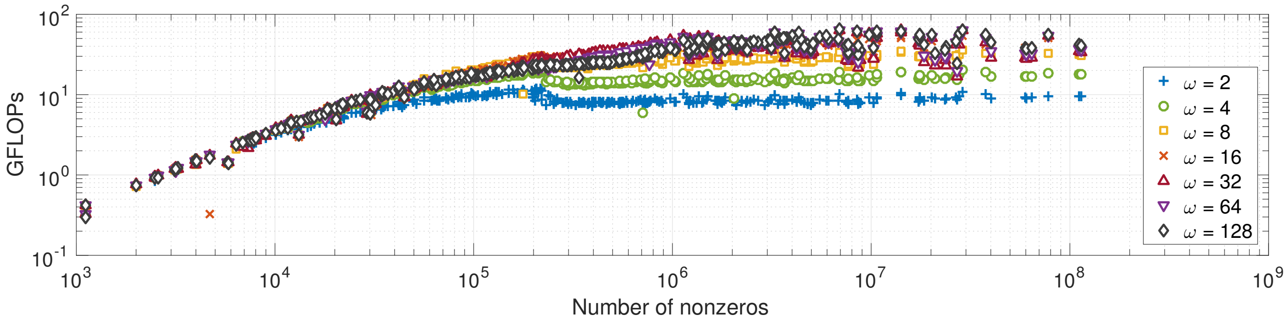

Figure 5: Evaluating the effect of the parameter ω on the performance of the COO kernel. The matrices are ordered according to increasing nonzero count.

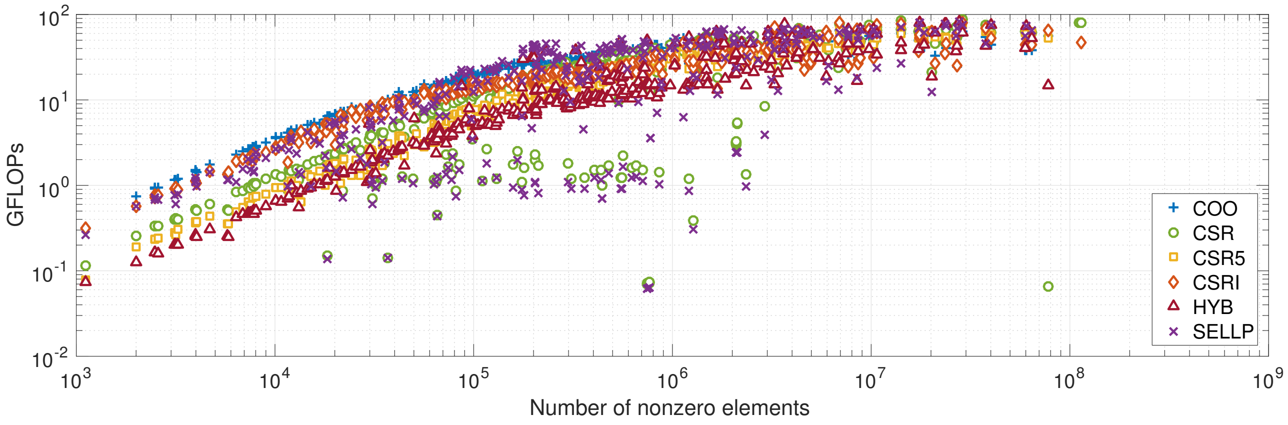

Figure 6: Performance of the distinct SpMV kernels for all problems included in the test suite. The matrices are ordered according to increasing nonzero count.

part of the MAGMA-sparse software stack, details about the kernel are presented in Liu et al. [11].

CSRI The design of the CSRI SpMV kernel is very similar to the COO kernel we propose. It tries to enable load balancing for unbalanced matrices stored in CSR by using atomic addition operations [9].

HYB The hybrid SpMV kernel we consider combines the ELL format for the regular (balanced) part of the matrix with the COO format for the irregular part of the matrix. We use the implementation available in NVIDIA's cuSPARSE library.

SELLP The SELL-p kernel is also part of the MAGMA-sparse software ecosystem, and has proven to be very efficient for balanced problems [12].

4.3 Experimental results

In a first experiment, we analyze the effect of the oversubscribing-parameter ω on the performance of the COO kernel. In Figure 5 we order the matrices for increasing nonzero count, and report the performance for the ω-values 2, 4, 8, 16, 32, 64, and 128. We notice that the performance differences increase with the nonzero count. For small nonzero counts, the differences are negligible. The optimal choice for the ω parameter then takes turns: Moderate oversubscribing with $\omega = 8/\omega = 16$ is the performance winner for systems with about 10^5 nonzeros, $\omega = 32/\omega = 64$ is superior for systems with about 10^6 nonzeros, and $\omega = 128$ seems to be the best parameter choice beyond that. The nonzero count of a matrix is one of the characteristics known prior to the SpMV invocation. Hence, a straight-forward optimization step for the COO kernel is given by choosing ω on a heuristic derived from Figure 5. In the rest of the paper we define the COO as the kernel that chooses

$$\omega = \begin{cases} 8 & \text{for} \quad nnz < 10^5, \\ 32 & \text{for} \quad 10^5 \le nnz < 10^6, \\ 128 & \text{for} \quad 10^6 \le nnz. \end{cases}$$

Next, we compare the COO kernel with the reference kernels previously listed. In Figure 6 we order the test matrices according to increasing nonzero count, and visualize the performance of all SpMV kernels we consider in this analysis.

Independent of the SpMV kernel, the performance linearly increases with the nonzero count of the problems until it stagnates around 90 GFLOPs.

Furthermore, the visualization suggests that the COO kernel is the fastest kernel for almost all test matrices with less than $2 \cdot 10^5$ nonzero elements. For problems containing more nonzero elements

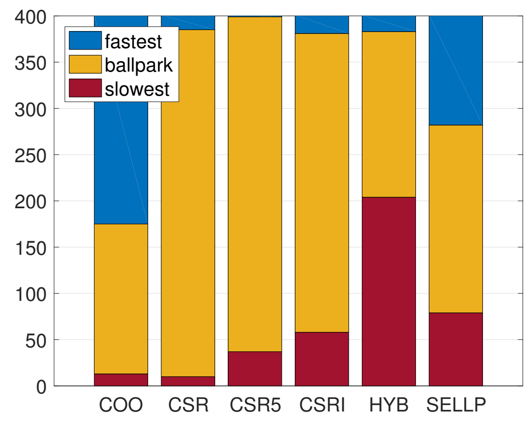

Figure 7: Fastest kernel comparison: blue bars represent the number of problems for which the kernel was the fastest, red bars count the problems for which the kernel was the slowest in the comparison.

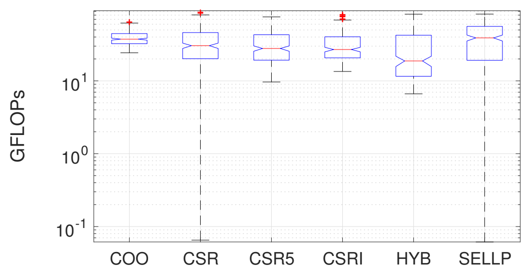

Figure 8: Performance statistics for the distinct kernels over the test cases containing more than $2 \cdot 10^5$ nonzero elements.

it is difficult to identify an overall winner. This is partly because multiple performance indicators are covering each other, and distinct problems, although different in sparsity pattern, may have the same nonzero count, and are therefore arranged in the same place on the x-axis. Overall, it is difficult to extract from this figure for how many problems a specific kernel is the fastest. We answer this question in Figure 7 where we report for how many problems a certain kernel was the performance winner (blue bar) and for how many problems a certain kernel gave the worst performance (red bar). If the kernel was neither the fastest nor the slowest for a certain problem, it is counted as "ballpark." In the end, for each kernel we get a bar of the same length split into three colors; the sum of all blue parts and the sum of all red parts equals the number of test cases, respectively.

Overall, the COO kernel wins most cases, and it is followed by the SELLP SpMV. CSR, CSRI, and HYB win only few cases, CSR5 not a single one. On the other hand, CSR5 and CSR are rarely the slowest kernels, while COO, CSRI, SELLP, and HYB lose significantly more cases.

Looking at the complete test suite containing all 400 matrices, we include problems that are "small" with respect to the computational workload of the SpMV. For those, even the winning kernel in Figure 7 achieves only low execution performance. In scientific applications, these "easy" problems are typically rather handled via a direct solver than an iteration method based on the SpMV kernel. As we are in particular interested in the performance for problems that are the characteristic target we limit the further analysis to the 248 problems containing more than $2 \cdot 10^5$ nonzero elements.

Kernel	min	max	average	median	standard-dev.
COO	24.29	64.32	38.86	37.24	9.16
CSR	0.07	87.43	32.77	30.43	20.07
CSR5	9.66	75.56	31.79	27.15	15.58
CSRI	13.47	81.21	31.85	26.84	14.44
HYB	6.64	82.43	27.98	18.74	20.22
SELLP	0.06	82.62	36.42	38.64	22.46

Table 1: Statistical information on the GFLOPs metric of the SpMV kernel for the 248 matrices containing more than $2 \cdot 10^5$ nonzero elements.

In Figure 8 we compare the distinct SpMV kernels in the GFLOPs metric for the problems containing more than $2 \cdot 10^5$ nonzero elements. We accompany this graph with some numeric information in Table 1 where we additionally list the average performance. For this metric, the COO format turns out to be the overall winner with an average 38.86 GFLOPs. Looking at the median, the SELLP and COO kernels achieve the highest execution rate (38.64 GFLOPs and 37.24 GFLOPs, respectively). They outperform the closest competitor CSR by about 25%. The lowest median performance of 18.74 GFLOPs is achieved by the HYB kernel. At the same time, the HYB kernel achieves significantly higher performance (up to 82.43 GFLOPs) for specific test cases, see Table 1. Only the SELLP and the CSR kernel achieve higher performance for balanced and regular problems (82.62 GFLOPs and 87.43 GFLOPs, respectively).

Most noticeable in Figure 8 is the variation of the COO performance being radically smaller than for any of the other formats: 50% of the performance numbers are within a 6 GFLOP/s range from the median, the standard deviation is 9.16. The lowest performance number for the COO kernel is 24 GFLOPs. Only the CSRI kernel is competitive in handling unbalanced problems, with 50% of the performance numbers between 20 GFLOPs and 60 GFLOPs, a standard deviation of 14.44, and the lowest performance being 13.47 GFLOPs (see Table 1). For SELLP and CSR, the performance values spread across a large range. In particular, the lowest performance numbers (0.06 GFLOPs for SELLP and 0.07 GFLOPs for CSR) are far from the median. The central box (upper/lower quantiles) are for these kernels a multiple of those for COO. Also the upper and lower whiskers are significantly further apart. For SELLP, this is expected as, for unbalanced matrices, the nonzero padding to a block-uniform nonzero count introduces significant performance-detrimental overhead, as well as load imbalance in-between the matrix blocks. Load imbalance is also the culprit for CSR's poor performance. Hence, the formats delivering the best performance for balanced test matrices are not suitable for irregular unbalances problems. The COO format, although achieving only 64.32 GFLOPs for the best case, proves to handle irregular problems well, with the

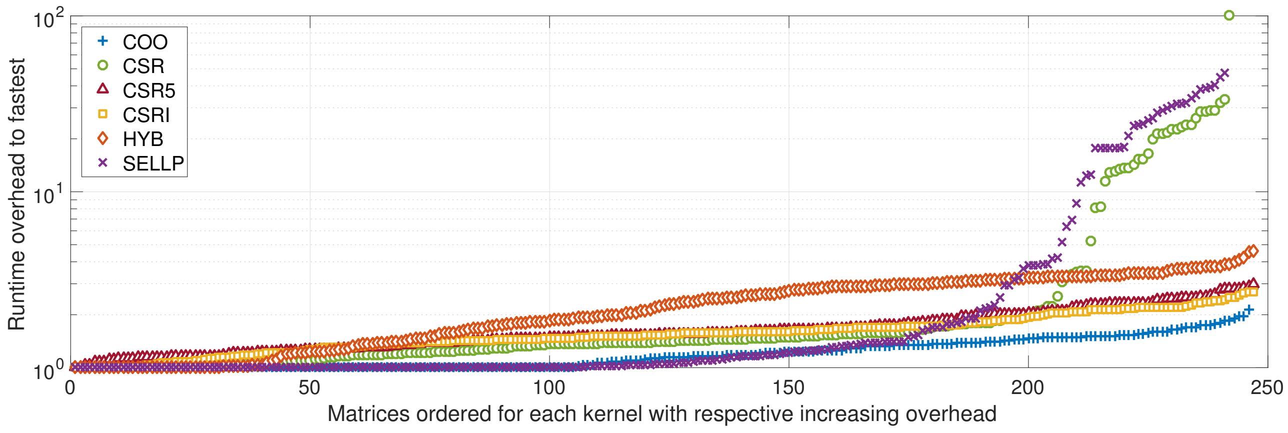

Figure 9: Runtime overhead of the distinct SpMV kernels. For each kernel, the matrices are ordered with respect to increasing overhead. Only test matrices with more than $2 \cdot 10^5$ nonzero elements are considered.

highest average performance, a competitive median, the smallest variation, and the highest minimal performance.

Finally, we want to assess the performance penalty of choosing one specific kernel vs. choosing the problem-specific best kernel. Obviously, the problem-specific best format is unknown a priori, and one would have to test all kernels prior to the the performance-relevant run, or use machine learning techniques for making a good guess. Again, we focus on the problems containing more than $2 \cdot 10^5$ nonzero elements. In the analysis we identify the optimal format for each test matrix, and scale the performance of all kernels to this baseline. We then sort the matrices with increasing overhead for every kernel individually, and visualize this characteristic curve in Figure 9. Hence, the order of the matrices is different for each kernel, but the overheads are increasing in all datasets. The objective of minimizing the slowdown corresponds to minimizing the area below the curve. The longer a curve stays at 1, the more test cases a certain kernel wins. We notice that the overhead stays low particularly for COO, CSRI, and CSR5. For SELLP and CSR, the initially moderate overhead for balanced problems quickly grows for irregular problems. HYB deals better with these cases, however it already starts of with a larger overhead than COO and the CSR variants. Overall, COO has a radically lower overhead than any of the competitors. Another key observation is that (ignoring two outliers) the COO kernel never exhibits a slowdown factor larger than two. This implies that choosing the COO format results in an SpMV kernel which is in the worst case two times slower than the (unknown) optimal choice. This clearly makes COO the overall winner in this metric as well.

5 SUMMARY AND OUTLOOK

We addressed the challenge of overcoming the load-imbalance in the sparse matrix vector product for irregular matrices. We developed and implemented an SpMV kernel for GPUs that is based on the COO format. Using a large collection of test matrices we compared the performance of the new kernel to the (up to our knowledge) best SpMV kernels available: the CSR and hybrid kernels which are part of NVIDIA's cuSPARSE library, the SELL-P and CSR5 kernels part of the MAGMA-sparse software library, and the CSRI kernel, which balances the workload via atomic operations. We used different metrics to quantify the performance: median absolute performance in GFLOPs and its variation, kernel winning most test cases, and smallest overhead compared to the best kernel included in the test suite. For the chosen test suite containing 400 matrices, the proposed COO-based SpMV performs radically better on irregular matrix structures, and ultimately wins all considered performance metrics.

In the future we want to focus on multi-GPU architectures and optimize the developed kernel for hybrid (Multicore+Manycore) execution. Furthermore, we are convinced that the strategies reducing the impact of global write conflicts via warp-vote functions and introducing additional operations are also applicable to other computational problems of irregular nature.

ACKNOWLEDGMENTS

H. Anzt was supported by the "Impuls und Vernetzungsfond" of the Helmholtz Association under grant VH-NG-1241. G. Flegar was 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.

Furthermore, the authors would like to thank Enrique S. Quintana-Ortí for comments on an earlier version of the paper.

REFERENCES

[1] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. Lapack: A portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Supercomputing '90, pages 2–11, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.
[2] H. Anzt, E. Chow, and J. Dongarra. On block-asynchronous execution on GPUs. Technical Report 291, LAPACK Working Note, 2016.
[3] H. Anzt, S. Tomov, and J. Dongarra. 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, March 2014.

[4] Hartwig Anzt, Mark Gates, Jack Dongarra, Moritz Kreutzer, Gerhard Wellein, and Martin Köhler. Preconditioned krylov solvers on {GPUs}. *Parallel Computing*, pages –, 2017.

[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. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.* SIAM, Philadelphia, PA, 1994.

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

[7] Nathan Bell and Michael Garland. Efficient Sparse Matrix-Vector Multiplication on CUDA, December 2008.

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

[9] Goran Flegar and Enrique S. Quintana-Ortí. "balanced csr sparse matrix-vector product on graphics processors". In *EuroPar 2017*, accepted.

[10] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop. 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):C401–C423, 2014.

[11] Weifeng Liu and Brian Vinter. 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, pages 339–350, New York, NY, USA, 2015. ACM.

[12] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. 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, pages 111–125, Berlin, Heidelberg, 2010. Springer-Verlag.

[13] NVIDIA. *NVIDIA CUDA TOOLKIT V8.0*, June 2017.