Sparse Linear Algebra on AMD and NVIDIA GPUs – The Race Is On

Yuhsiang M. Tsai[1], Terry Cojean[1], and Hartwig Anzt[1,2](✉)

[1] Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany
hartwig.anzt@kit.edu
[2] University of Tennessee, Knoxville, TN 37996, USA

Abstract. Efficiently processing sparse matrices is a central and performance-critical part of many scientific simulation codes. Recognizing the adoption of manycore accelerators in HPC, we evaluate in this paper the performance of the currently best sparse matrix-vector product (SpMV) implementations on high-end GPUs from AMD and NVIDIA. Specifically, we optimize SpMV kernels for the CSR, COO, ELL, and HYB format taking the hardware characteristics of the latest GPU technologies into account. We compare for 2,800 test matrices the performance of our kernels against AMD's hipSPARSE library and NVIDIA's cuSPARSE library, and ultimately assess how the GPU technologies from AMD and NVIDIA compare in terms of SpMV performance.

Keywords: Sparse matrix vector product (SpMV) · GPUs · AMD · NVIDIA

1 Introduction

The sparse matrix vector product (SpMV) is a heavily-used and performance-critical operation in many scientific and industrial applications such as fluid flow simulations, electrochemical analysis, or Google's PageRank algorithm [11]. Operations including sparse matrices are typically memory bound on virtually all modern processor technology. With an increasing number of high performance computing (HPC) systems featuring GPU accelerators, there are significant resources spent on finding the best way to store a sparse matrix and optimize the SpMV kernel for different problems.

In this paper, we present and compare four SpMV strategies (COO, CSR, ELL, and HYB) and their realization on AMD and NVIDIA GPUs. We furthermore assess the performance of each format for 2,800 test matrices on high-end GPUs from AMD and NVIDIA. We also derive performance profiles to investigate how well the distinct kernels generalize. All considered SpMV kernels are integrated into the Ginkgo open-source library[1], a modern C++ library designed for the iterative solution of sparse linear systems, and we demonstrate

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

P. Sadayappan et al. (Eds.): ISC High Performance 2020, LNCS 12151, pp. 309–327, 2020.
https://doi.org/10.1007/978-3-030-50743-5_16

that these kernels often outperform their counterparts available in the AMD hipSPARSE and the NVIDIA cuSPARSE vendor libraries.

Given the long list of efforts covering the design and evaluation of SpMV kernels on manycore processors, see [2,7] for a recent and comprehensive overview of SpMV research, we highlight that this work contains the following novel contributions:

- We develop new SpMV kernels for COO, CSR, ELL and HYB that are optimized for AMD and NVIDIA GPUs and outperform existing implementations. In particular, we propose algorithmic improvements and tuning parameters to enable performance portability.
- We evaluate the performance of the new kernels against SpMV kernels available in AMD's hipSPARSE library and NVIDIA's cuSPARSE library.
- Using the 2,800 test matrices from the Suite Sparse Matrix Collection, we derive performance profiles to assess how well the distinct kernels generalize.
- We compare the SpMV performance limits of high-end GPUs from AMD and NVIDIA.
- 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 with back ends for both, AMD's and NVIDIA's GPUs.
- We ensure full result reproducibility by making all kernels publicly available as part of the Ginkgo library, and archiving the performance results in a public repository[2].

Before providing more details about the sparse matrix formats and the processing strategy of the related SpMV routines in Sect. 3, we recall some basics about sparse matrix formats in Sect. 2. In Sect. 3.4, we combine several basic matrix storage formats into the so-called "hybrid" format (HYB) that splits the matrix into parts to exploit the performance niches of various basic formats. In a comprehensive evaluation in Sect. 4, we first compare the performance of Ginkgo's SpMV functionality with the SpMV kernels available in NVIDIA's cuSPARSE library and AMD's hipSPARSE library, then derive performance profiles to characterize all kernels with respect to specialization and generalization, and finally compare the SpMV performance of AMD's RadeonVII GPU with NVIDIA's V100 GPU. We conclude in Sect. 5 with a summary of the observations.

2 Review of Sparse Matrix Formats

For matrices where most elements are zero, which is typical for, e.g., finite element discretizations or network representations, storing all values explicitly is expensive in terms of memory and computational effort. In response, sparse matrix formats reduce the memory footprint and the computational effort by focusing on the nonzero matrix values [3]. In some cases, additionally storing some zero elements can improve memory access and data-parallel processing [4].

[2] https://github.com/ginkgo-project/ginkgo-data/tree/2020_isc.

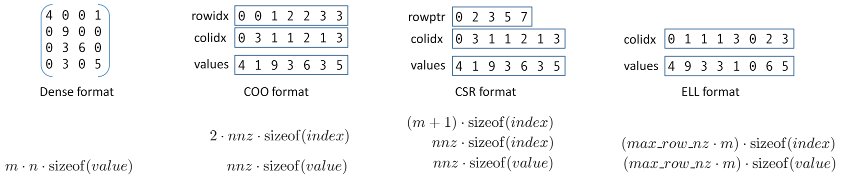

4 0 0 1
0 9 0 0
0 3 6 0
0 3 0 5
Dense format
$m \cdot n \cdot \text{sizeof}(value)$
rowidx 0 0 1 2 2 3 3
colidx 0 3 1 1 2 1 3
values 4 1 9 3 6 3 5
COO format
$2 \cdot nnz \cdot \text{sizeof}(index)$
$nnz \cdot \text{sizeof}(value)$
rowptr 0 2 3 5 7
colidx 0 3 1 1 2 1 3
values 4 1 9 3 6 3 5
CSR format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

Fig. 1. Different storage formats for a sparse matrix of dimension $m \times n$ containing n_z nonzeros along with the memory consumption [6].

While there exists a long and still expanding list of sparse matrix formats (some of them tailored towards specific problems), we illustrate some of the most common basic formats (DENSE, COO, CSR, ELL) in Fig. 1.

The optimization of the SpMV kernel for manycore GPUs remains a topic of major interest [5,9,12]. Many of the most recent algorithm developments increase the efficiency by using prefix-sum computations [13] and intra-warp communication [10] on modern manycore hardware.

3 Sparse Matrix Vector Kernel Designs

We realize all SpMV kernels in the vendors' native languages: CUDA for NVIDIA GPUs and HIP for AMD GPUs. Given the different hardware characteristics, see Table 1, we optimize kernel parameters like group size for the distinct architectures. More relevant, for the CSR, ELL, and HYB kernels, we modify the SpMV execution strategy for the AMD architecture from the strategy that was previously realized for NVIDIA architectures [2].

3.1 Balancing COO SpMV Kernel

Flegar et al. [6] introduced a load-balancing COO SpMV based on the idea of parallelizing across the nonzeros of a sparse matrix. This way, all threads have the same workload, and coalesced access to the column indexes and the values of the sparse matrix is enabled. At the same time, parallelizing across nonzeros requires the use of atomicAdd operations to avoid race conditions, see Algorithm 1.

Flegar et al. [6] also introduced an "oversubscribing" parameter ω that controls the number of threads allocated to each physical core. When increasing the oversubscribing, we have more active threads to hide the latency of data access and atomicAdds [1]. At the same time, it increases the number of atomicAdds invocations and the overhead of context switching. Using an experimental assessment on all of the 2,800 matrices from the Suite Sparse Matrix Collection, Flegar et al. [6] identifies oversubscribing parameters ω_{NVIDIA} that draw a good balance between these aspects. Similarly to Flegar et al. [6], we use experiments to identify good choices ω_{AMD} for AMD architectures by considering oversubscribing

Algorithm 1. Load-balancing COO kernel algorithm.

```
Get $ind$ = index of the first element to be processed by this thread
Get $current\_row = rowidx[ind]$.
Compute the first value $c = A[ind] \times x[colidx[ind]]$
for i = 0 .. $nz\_per\_warp$; $i+ = warpsize$ do
    Compute $next\_row$, row index of the next element to be processed
    if any thread in the warp's $next\_row$ != $current\_row$ or it is the final iteration then
        Compute the segmented scan according to $current\_row$.
        if first thread in segment then
            atomicAdd $c$ on output vector by the first entry of each segment
        end if
        Reinitialize $c = 0$
    end if
    Get the next index $ind$
    Compute $c+ = A[ind] \times x[colidx[ind]]$
    Update $current\_row$ to $next\_row$
end for
```

parameters $\omega = 2^k (0 \leq k \leq 7)$. In the Ginkgo library and our experiments, we use the setting

$$\omega_{\text{NVIDIA}} = \begin{cases} 8 & (n_z < 2 \cdot 10^5), \\ 32 & (2 \cdot 10^5 \leq n_z < 2 \cdot 10^6), \\ 128 & (2 \cdot 10^6 \leq n_z) \end{cases} \quad \omega_{\text{AMD}} = \begin{cases} 2 & (n_z < 10^5), \\ 8 & (10^5 \leq n_z < 10^7). \\ 32 & (10^7 \leq n_z) \end{cases}$$

3.2 CSR SpMV Kernel

The most basic CSR SpMV kernel (*basic* CSR) assigns only one thread to each row, which results in notoriously low occupancy of GPU. In Algorithm 2, we assign a "subwarp" (multiple threads) to each row, and use warp reduction mechanisms to accumulate the partial results before writing to the output vector. This *classical* CSR assigning multiple threads to each row is inspired by the performance improvement of the ELL SpMV in [2]. We adjust the number of threads assigned to each row to the maximum number of nonzeros in a row. We select

$$\text{subwarp size} = 2^k (0 \leq k \leq 5 \text{ (NVIDIA) or } 6 \text{ (AMD)})$$

as the closest number smaller or equal to the maximum number of nonzeros in a row, i.e.

$$\text{subwarp size} = \max\left\{2^t \leq max\_row\_nnz | t \in \mathbb{Z}, 0 \leq t \leq \log_2(\text{device warpsize})\right\}$$

In Fig. 5 in Sect. 4, we visualize the performance improvements obtained from assigning multiple threads to each row and observe that the *basic* CSR SpMV is not always slower. In particular for very unbalanced matrices, assigning the same parallel resources to each row turns out to be inefficient. In response, we

Algorithm 2. Ginkgo's classical CSR kernel.

```
Get $row$ = the row index
Compute $subrow$ = the step size to next row
Get $step\_size$ = the step size to next element of value.
Initialize value $c = 0$
for $row = row .. \#rows, row+ = subrow$ do
    for $idx = row\_ptr[row] .. row\_ptr[row + 1], idx+ = step\_size$ do
        Compute $c = val[idx] * b[col[idx]]$
    end for
    Perform warp reduction of $c$ on the warp
    if thread 0 in subwarp then
        Write $c$ to the output vector
    end if
end for
```

design a *load-balancing* CSRI which follows the strategy of the COO SpMV described in Sect. 3.1 to balance the workload across the compute resources. For an automatic strategy selection in Algorithm 3, we define two variables *nnz_limit* and *row_len_limit* to control the kernel selection on NVIDIA and AMD GPUs. *nnz_limit* reflects the limit of total nonzero count, and *row_len_limit* reflects the limit of the maximum number of stored elements in a row. For AMD GPUs, *nnz_limit* is 10^8 and *row_len_limit* is 768. For NVIDIA GPUs, *nnz_limit* is 10^6 and *row_len_limit* is 1024.

Algorithm 3. Ginkgo's CSR strategy.

```
Compute $max\_row\_nnz$ = the maximal number of stored element per rows.
if $\#nnz > nnz\_limit$ or $max\_row\_nnz > row\_len\_limit$ then
    Use load-balance CSR Kernel
else
    Use classical CSR Kernel
end if
```

3.3 ELL SpMV Kernel

In [2], the authors demonstrated that the ELL SpMV kernel can be accelerated by assigning multiple threads to each row, and using an "early stopping" strategy to terminate thread blocks early if they reach the padding part of the ELL format. Porting this strategy to AMD architectures, we discovered that the non-coalesced global memory access possible when assigning multiple threads to the rows of the ELL matrix stored in column-major format can result in low performance. The reason behind this is that the strategy in [2] uses threads of the same group to handle one row, which results in adjacent threads always reading matrix elements that are m (matrix size or stride) memory locations apart. To overcome this problem, we rearrange the memory access by assigning the threads of the same group to handle one column like the classical ELL kernel, but assigning several groups to each row to increase GPU usage. Because the threads handling the elements of a row may be of the same thread block but are no longer part of the same warp, we can not use warp reduction for the partial sums but need to invoke atomicAdds on shared memory. Figure 2 visualizes the different memory access strategies.

In our experiments, we set the "group_size" to multiple of 32 for both AMD and NVIDIA architectures. The group_size is the number of contiguous element read by thread block, and the num_group is the number of thread in the thread block accessing the same row. We use block size = 512 in ELL kernel. To make the "group_size" is the multiple of 32, we set the max of $num\_group = block\_size/min\_group\_size = 512/32 = 16$. We visualize in Fig. 8 the improvement of the new ELL SpMV kernel over the kernel previously employed [2].

Algorithm 4. Ginkgo's ELL SpMV kernel.

```
 1: Initialize Value $c = 0$
 2: Compute $row$ = the row idx
 3: Compute $y$ = the start index of row
 4: Compute $step\_size$ = the step size to next element
 5: Initialize shared memory $data$
 6: for $idx = y$ .. $max\_row\_nnz$, $idx += step\_size$ do
 7:     Compute $ind$ = index of this element in the ELL format
 8:     if $A(row, colidx[ind])$ is padding then
 9:         break
10:     end if
11:     Perform local operation $c+ = A(row, colidx[ind]) * x[colidx[ind]]$
12: end for
13: Perform atomicAdd $c$ to $data[threadIdx.x]$
14: if thread 0 in group then
15:     atomicAdd $data[threadIdx.x]$ on the output vector
16: end if
```

In Algorithm 4, we present the ELL SpMV kernel implemented in Ginkgo for SIMD architectures like GPUs. The number of groups assigned to a row is computed via Algorithm 5. Generally, the number of the group is increased with the number of nonzero elements accumulated in a single row. However, if $num\_group = 16$, multiple thread block may be assigned to the same row, see line 8 in Algorithm 5. This strategy aims at increasing the occupancy of the GPU multiprocessors when targeting short-and-wide matrices that accumulate many elements in few rows. After the group is determined, the start index for a specific thread is computed in lines 2 in Algorithm 4 with the step size which is same as the total number of threads accessing the same row. The threads process the data with the loop in lines 6–12. This kernel still uses the early stopping in lines 8–10 introduced in [2]. After completion of the matrix vector

Algorithm 5. Ginkgo's automatic ELL kernel configuration.

```
Initialize num_group = 1
Initialize nblock_per_row = 1
Compute ell_ncols = maximum number of non zero elements per row
Get nwarps = total number of warps available on the GPU
if ell_ncols / nrows > $1e-2$ then
    Compute num_group = $min(16, 2^{ceil(log_2(ell\_ncols))})$
    if num_group == 16 then
        Compute nblock_per_row = $max(min(ell\_ncols/16, nwarps/nrows), 1)$
    end if
end if
```

multiplication step, the partial sums accumulated in thread-local variables are reduced (line 13) and added to the output vector in global memory, see line 15. Even though this operation requires an atomic operation as multiple groups (part of distinct thread blocks) may operate on the same row, the chance of atomic collisions is small due to the previous reduction in line 13.

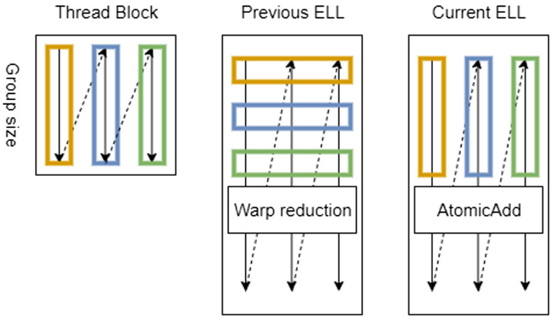


Fig. 2. Comparison of the memory access for different ELL SpMV kernels.

3.4 Hybrid Matrix Formats and Optimal Matrix Splitting

Ginkgo's hybrid ("HYB") format splits the matrix into two parts and stores the regular part in the ELL format and the irregular part in the COO format. Flegar et al. [6] demonstrated that Ginkgo's COO SpMV achieves good performance for irregular matrices on NVIDIA architectures, and the results in Sect. 4 confirm that Ginkgo's COO SpMV performs well also on AMD architectures. How the HYB format partitions a matrix into the ELL and the COO part impacts the memory requirements and performance. Anzt et al. [2] derived strategies basing the partitioning on the nonzeros-per-row distribution of the matrix. We modify this strategy by adding a condition based on the ratio between the maximum nonzeros-per-row and the number of rows. For R being the set of the nonzeros-per-row values, we define the function Q_R and F_R:

$$Q_R(x) := \min\{t \in \mathbb{N} \mid x < F_R(t)\}, \qquad F_R(t) := \frac{|\{r \in R \mid r \leq t\}|}{|R|}.$$

We recall that Anzt et al. [2] introduced hybrid$\{n\}$ which takes the nonzeros of the row at the $n\%$-quantile in the ascending ordering of the nonzero-per-row values, $Q_R(n\%)$. A variant denoted with "hybridminstorage" selects

$$n\% = \left\lfloor \frac{\#rows \times sizeof(index)}{sizeof(value) + 2 \times sizeof(index)} + 1 \right\rfloor$$

according to the (bit-)size of the value and index arrays, i.e. hybridminstorage is hybrid25 when storing the values in 64-bit doubles and the indexes in 32-bit integers [2]. In this paper, we enhance the hybrid$\{n\}$ partitioning from Anzt et al. [2] by enforcing the limitation that the maximum nonzero-per-row of the ELL part can at most be $\#rows * 0.0001$. We consider the resulting strategy "hybridlimit$\{n\}$" and select hybridlimit33 (label "HYB") as our default strategy according to the performance evaluation in Fig. 11 in Sect. 4.

4 Experimental Performance Assessment

4.1 Experiment Setup

In this paper, we consider NVIDIA's V100 (SXM2 16 GB) GPU with support for compute capability 7.0 [14] and AMD's RadeonVII with compute capability gfx906. See Table 1 for some hardware specifications [16]. We note that the AMD RadeonVII is not a server-line GPU, but provides the same memory bandwidth as the AMD HPC GPU MI50, and thus should be comparable for memory bound operations such as the SpMV kernels. We use the major programming ecosystems for the distinct architectures - CUDA for NVIDIA GPUs and HIP for AMD GPUs. CUDA GPU kernels were compiled using CUDA version 9.2, and HIP GPU kernels were compiled using HIP version 2.8.19361.

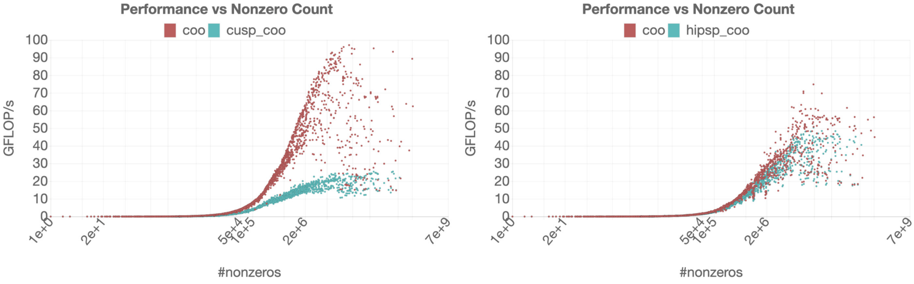
Table 1. Specifications of the V100 SXM2 16 GB and the RadeonVII [16].

	Warpsize	Bandwidth	FP64 performance	L1 cache	L2 cache
V100	32	897 GB/s	7.834 TFLOPS	128 KB	6 MB
RadeonVII	64	1024 GB/s	3.360 TFLOPS	16 KB	4 MB

The performance evaluation covers more than 2,800 test matrices of the Suite Sparse Matrix Collection [15]. Some matrices contain dense rows, which makes the conversion to the ELL format virtually impossible. We ignore those matrices in the performance evaluation of the ELL SpMV kernel.

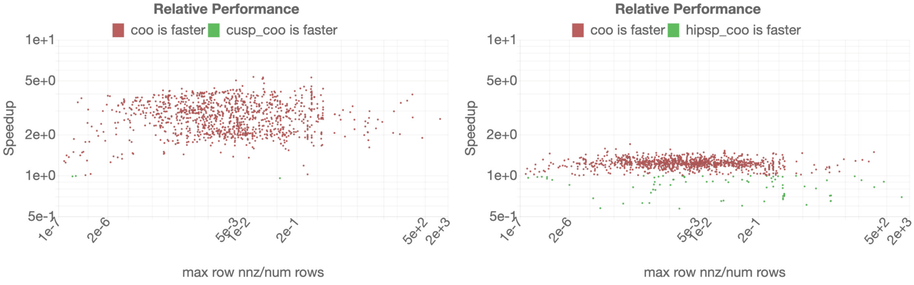
All experiments are performed in IEEE double precision arithmetic, and the GFLOP/s rates are computed under the assumption that the number of flops is always $2n_z$, where n_z is the number of nonzeros of the test matrix (ignoring padding).

4.2 COO SpMV Performance Analysis



(a) V100 (b) RadeonVII

Fig. 3. Performance of GINKGO'S and vendors' COO SPMV

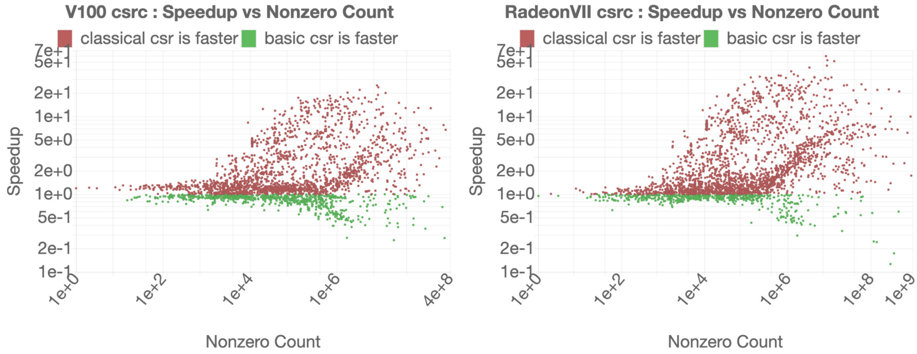


(a) V100 (b) RadeonVII

Fig. 4. Releative performance of GINKGO'S and vendors' COO SPMV (Color figure online)

We first evaluate the performance of the load-balancing COO SPMV kernel. In Fig. 3a, we compare against cuSPARSE's COO kernel (cusparseDhybmv with `CUSPARSE_HYB_PARTITION_USER` and threshold of 0), in Fig. 3b, we compare against hipSPARSE's COO kernel (hipsparseDhybmv with `HIPSPARSE_HYB_PARTITION_USER` and threshold of 0). Each dot reflects one test matrix from the Suite Sparse collection. The x-axis is the nonzero count of the matrix, and the y-axis is the performance in GFLOP/s. In Fig. 4, we present the speedup of GINKGO'S SPMV over cuSPARSE's COO implementation and hipSPARSE's COO implementation, respectively. Red dots reflect test matrices where Ginkgo outperforms the vendor library, green dots reflect cases where the vendor library is faster. Despite the fact that the irregularity of a matrix heavily impacts the SpMV kernels' efficiency, we can observe that Ginkgo's COO SpMV achieves much higher performance than both NVIDIA's and AMD's COO kernels in most cases. Overall, Ginkgo achieves an average speedup of about 2.5x over cuSPARSE's COO SpMV and an average speedup of about 1.5x over hipSPARSE COO SpMV.

4.3 CSR SpMV Performance Analysis



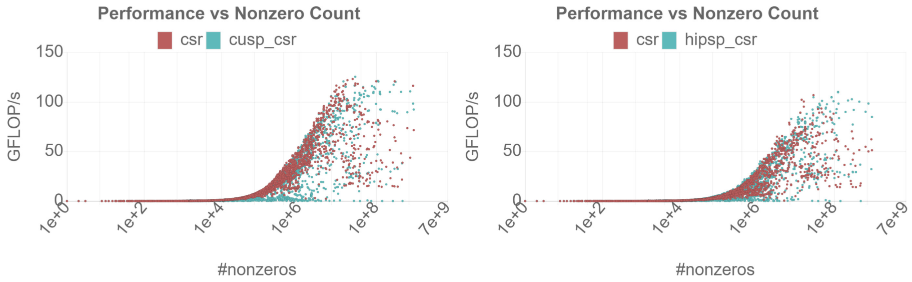
(a) V100

(b) RadeonVII

Fig. 5. Performance improvement of (current) classical CSR SpMV and (previous) basic CSR SpMV.

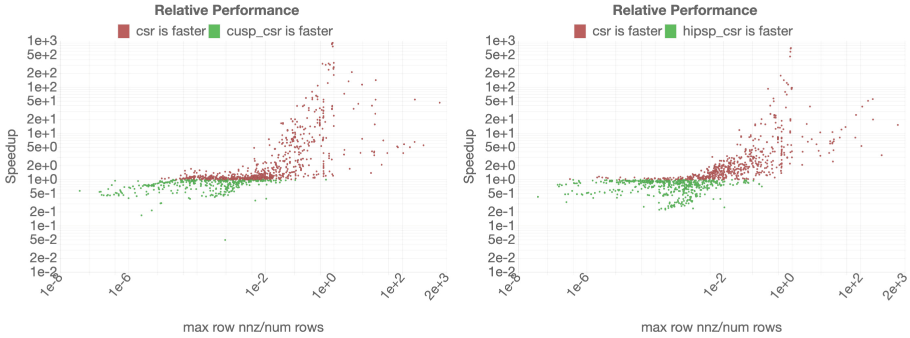
In the CSR SpMV performance analysis, we first demonstrate the improvement of assigning multiple threads to each row (*classical* CSR) over the implementation assigning only one thread to each row (*basic* CSR) see Fig. 5 for the CUDA and AMD backend, respectively. For a few matrices with many nonzeros, the *basic* CSR is 5x–10x faster than the *classical* CSR. To overcome this problem, we use Algorithm 3 in Ginkgo that chooses the load-balancing CSRI algorithm for problems with large nonzero counts.

Next, we compare the performance of the Ginkgo CSR SpMV (that automatically interfaces to either the load-balancing CSRI kernel or the classical CSR, see Sect. 3.2) with the vendors' CSR SpMV. Anzt et al. [2] identified the cusp_csr kernel (cusparseDcsrmv) as the overall performance winner among the different NVIDIA CSR implementations. For the AMD CSR SpMV kernel, we use the CSR kernel (hipsparseDcsrmv) provided in hipSPARSE. For completeness, we mention that the rocSPARSE library (outside the HIP ecosystem) contains a CSR kernel that renders better SpMV performance for irregular matrices on AMD GPUs. We refrain from considering it as we want to stay within the



(a) V100 (b) RadeonVII

Fig. 6. Performance of Ginkgo's and vendors' CSR SpMV



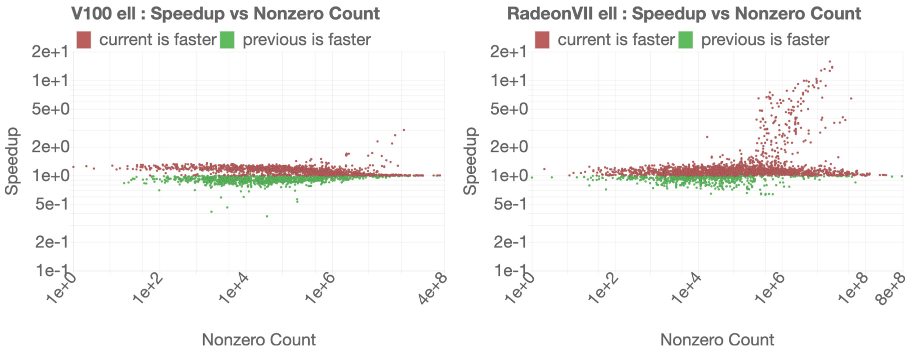
(a) V100 (b) RadeonVII

Fig. 7. Relative performance of Ginkgo's and vendors' CSR SpMV

HIP ecosystem, which is anticipated to serve as primary dissemination tool for AMD's sparse linear algebra technology.

In Fig. 6, we compare the Ginkgo CSR SpMV with the `cusparseDcsrmv` CSR kernel available in NVIDIA's cuSPARSE library and the `hipsparseDcsrmv` CSR kernel available in AMD's hipSPARSE library, respectively. In the relative performance analysis, Fig. 7, we use the ratio $\frac{max(row\_n_z)}{num\_rows}$ for the x-axis as this is the parameter used in Ginkgo's CSR SpMV to decide which CSR algorithm is selected. Ginkgo CSR achieves significant speedups for large x-values (up to 900x speedup on V100 and 700x speedup on RadeonVII). At the same time, there are a few cases where the Ginkgo CSR SpMV is slower than the library implementations (up to 20x slowdown on V100 and 5x slowdown on RadeonVII).

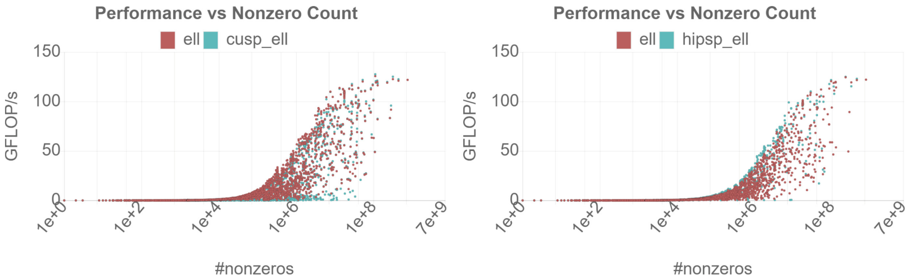
4.4 ELL SpMV Performance Analysis



(a) V100 (b) RadeonVII

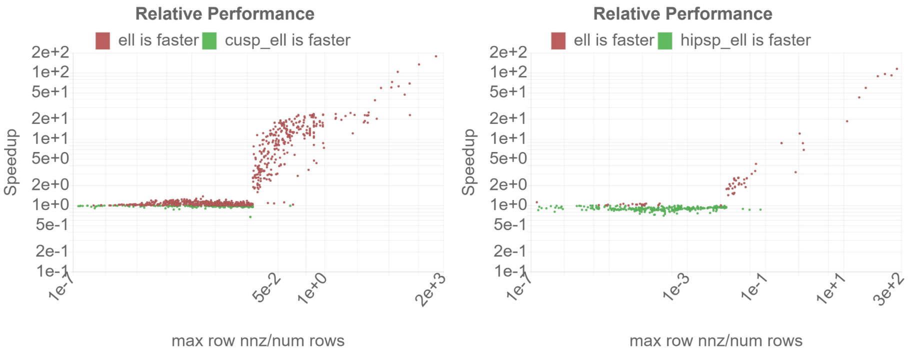
Fig. 8. Relative performance of GINKGO'S current ELL SPMV against the previous one

First, we investigate the performance improvement we obtain by changing the memory access strategy for the ELL SPMV kernel, see Sect. 3. Interestingly, moving to the new ELL SPMV algorithm does not render noteworthy performance improvements on NVIDIA's V100 GPU, as can be seen in Fig. 8a. At the same time, the performance improvements are significant for AMD's RadeonVII, as shown in Fig. 8b. In the new ELL SPMV algorithm, we improve the global memory access at the cost of atomicAdd operations on shared memory (which are more expensive than warp reductions). In consequence, the current ELL SPMV is not always faster than the previous ELL SPMV.



(a) V100 (b) RadeonVII

Fig. 9. Performance of GINKGO'S and vendors' ELL SPMV (Color figure online)

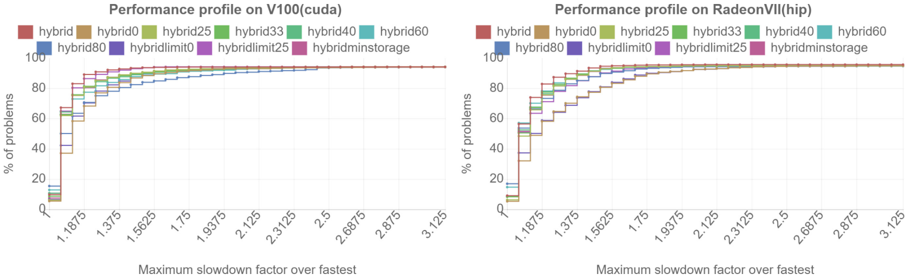


(a) V100 (b) RadeonVII

Fig. 10. Relative performance of Ginkgo's and vendors' ELL SpMV

In Fig. 9, we compare Ginkgo's ELL SpMV kernel against cuSPARSE cusparseDhybmv with `CUSPARSE_HYB_PARTITION_MAX` ELL kernel and hipSPARSE hipsparseDhybmv with `HIPSPARSE_HYB_PARTITION_MAX` ELL kernel, respectively. hipSPARSE ELL employs a limitation not to process matrices that have more than $\frac{\#nnz-1}{\#rows}+1$ elements in a row. Thus, we have much fewer data points for the hipSPARSE ELL SpMV (the blue points in Fig. 9b). In Fig. 10, Ginkgo's ELL is faster than their counterparts available in the vendors libraries if the ratio $\frac{max(row\_n_z)}{num\_rows} > 10^{-2}$. For the other cases, Ginkgo and the vendor libraries are comparable in their ELL SpMV performance.

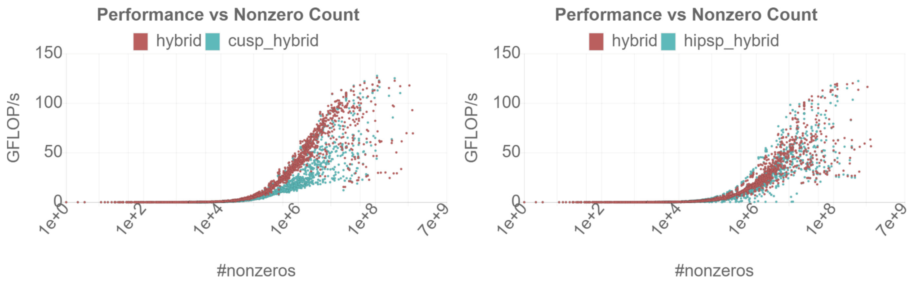
4.5 HYB SpMV Performance Analysis



(a) V100 (b) RadeonVII

Fig. 11. Performance profile comparing the different Ginkgo HYB splitting strategies

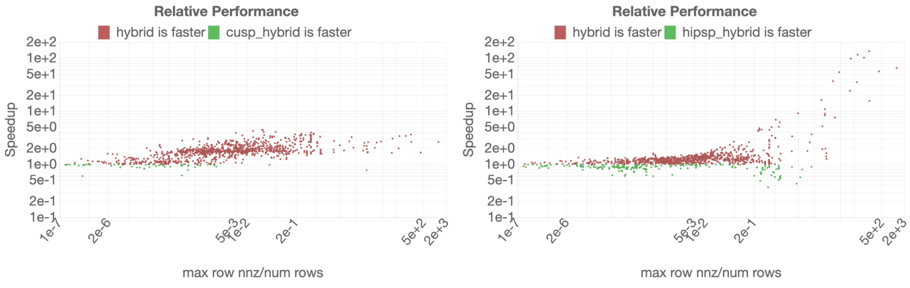
Before comparing against the vendor implementations, we investigate the performance of our HYB SpMV kernel for different partitioning strategies denoted by hybrid$\{n\}$, hybridlimit$\{n\}$, and hybridminstorage (which is same as hybrid25) as introduced in Sect. 3.4. We use a performance profile [8] on all Suite Sparse matrices to compare the strategies with respect to specialization and generalization. Using a performance profile allows to identify the test problem share (y-axis) for a maximum acceptable slowdown compared to the fastest algorithm (x-axis). In Fig. 11, we visualize the performance profiles for the V100 and RadeonVII architectures. Although the hybrid strategy (which corresponds to hybridlimit33) does not win in terms of specialization (maximum slowdown of 1), we favor this strategy since it provides the best generality: when considering a maximum acceptable slowdown factor of less than 1.75, this format wins in terms of problem share.



(a) V100

(b) RadeonVII

Fig. 12. Performance of Ginkgo's and vendors' HYB SpMV



(a) V100

(b) RadeonVII

Fig. 13. Relative performance of Ginkgo's and vendors' HYB SpMV

In Fig. 12, we see that Ginkgo's HYB SpMV achieves similar peak performances like cuSPARSE's cusparseDhybmv HYB SpMV and hipSPARSE's hipsparseDhybmv HYB SpMV, but Ginkgo has much higher performance averages than cuSPARSE or hipSPARSE. Figure 13a and Fig. 13b visualize the HYB SpMV performance relative to the vendor libraries, and we identify significant speedups for most problems and moderate slowdowns for a few cases.

4.6 All SpMV Performance Profile Analysis

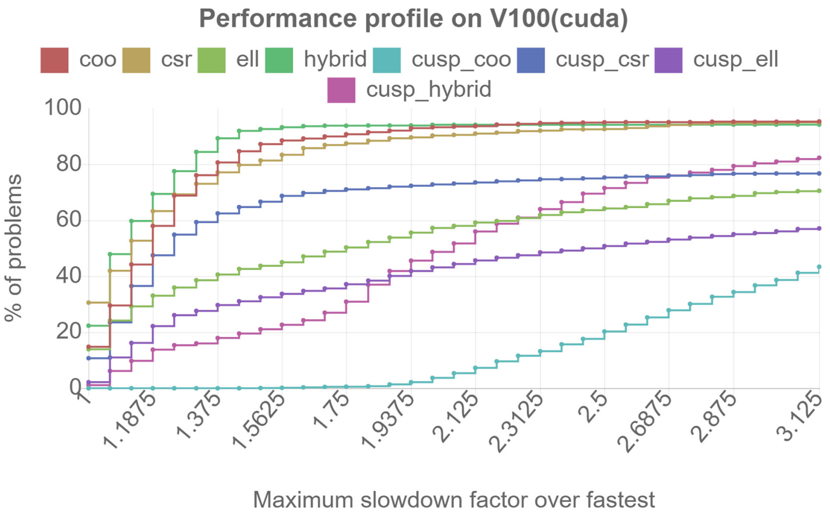


Fig. 14. Performance profile comparing multiple SpMV kernels on V100.

In Fig. 14, we use the performance profile to assess the specialization and generalization of all matrix formats we consider. In Fig. 14, Ginkgo's CSR is the fastest for about 30% of the test cases, and Ginkgo's HYB is the winner in terms of generality (if the acceptable slowdown factor is larger than 1.0625). Very similarly, in Fig. 15, Ginkgo's CSR is the fastest kernel for roughly 30% of the test cases, and Ginkgo's HYB is the generalization-winner if the acceptable slowdown factor is larger than 1.375. We note that the hipSPARSE ELL stays at a low problem ratio as it employs a limitation to not process matrices that have more than $\frac{\#nnz-1}{\#rows}+1$ elements in a row.

We already noticed in the analysis comparing Ginkgo's different SpMV kernels to the vendor libraries that AMD's hipSPARSE library generally features much better-engineered kernels than NVIDIA's cuSPARSE library. In consequence, also the performance profiles of AMD's SpMV kernels are much closer to Ginkgo's SpMV kernel profiles than NVIDIA's SpMV kernel profiles.

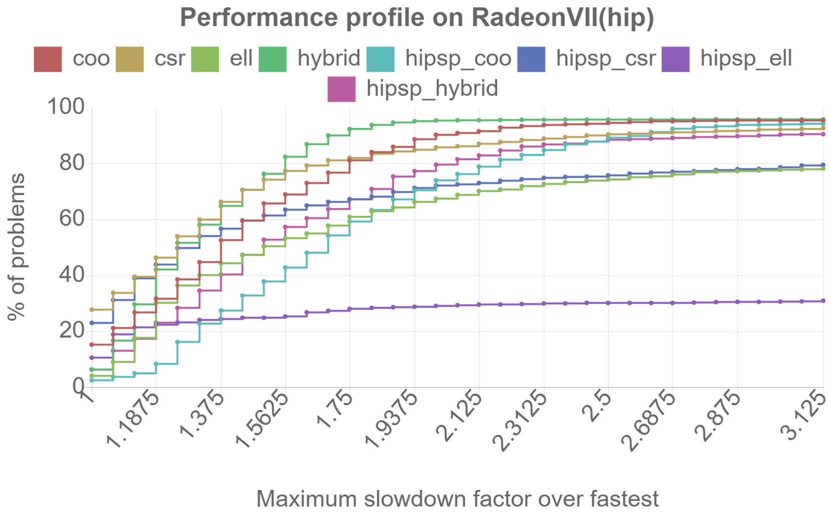


Fig. 15. Performance profile comparing multiple SpMV kernels on Radeon VII.

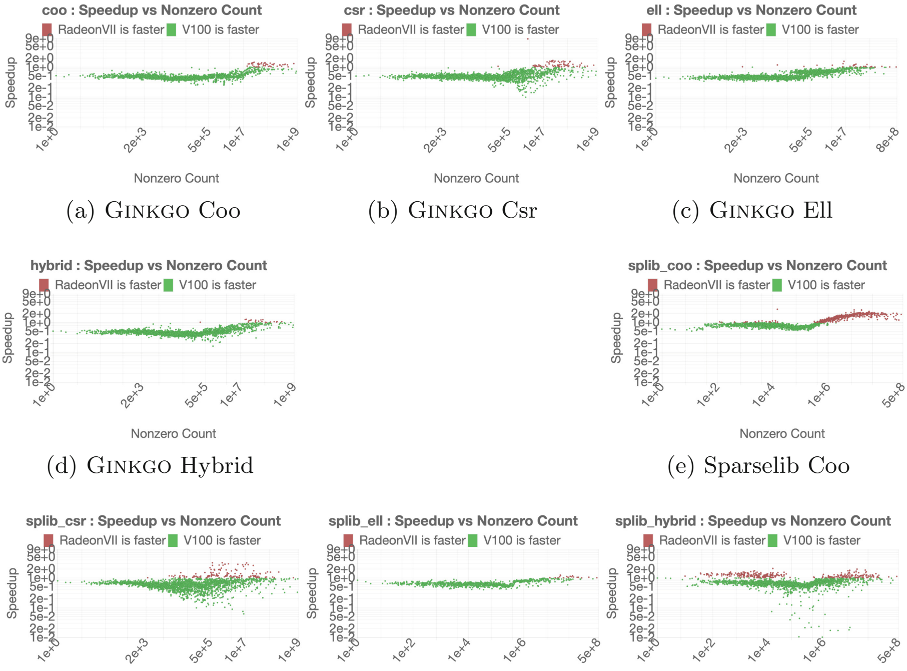
4.7 RadeonVII vs V100 SpMV Performance Analysis

We finally compare the SpMV performance limits of RadeonVII and V100 in Fig. 16. We consider both Ginkgo's back ends for the two architectures, and the SpMV kernels available in the vendor libraries (labeled "Sparselib").

In most cases, the V100 is faster than RadeonVII, but the speedup factors are moderate, with an average around 2x. RadeonVII shows better performance for matrices that contain many nonzeros. The higher memory bandwidth of the RadeonVII might be a reason for these performance advantages, but as there are typically many factors (such as context switch, warp size, the number of multiprocessors, etc.) affecting the performance of SpMV kernels, identifying the origin of the performance results is difficult.

While NVIDIA's V100 outperforms AMD's RadeonVII in most tests, we acknowledge that the price for a V100 (16 GB SXM2) is currently more than an order of magnitude higher than for a RadeonVII[3]

[3] In December 2019, the list price for NVIDIA's V100 (16 GB SXM2) is US$ 10,664.-, the list price for AMD's RadeonVII is US$ 699.-.



(a) Ginkgo Coo (b) Ginkgo Csr (c) Ginkgo Ell

(d) Ginkgo Hybrid (e) Sparselib Coo

(f) Sparselib Csr (g) Sparselib Ell (h) Sparselib Hybrid

Fig. 16. Comparison of the SpMV kernel implementations of hipSPARSE on RadeonVII and cuSPARSE on V100

5 Summary and Outlook

In this paper, we have presented a comprehensive evaluation of SpMV kernels for AMD and NVIDIA GPUs, including routines for the CSR, COO, ELL, and HYB format. We have optimized all kernels for the latest GPU architectures from both vendors, including new algorithmic developments and parameter tuning. All kernels are part of the Ginkgo open source library, and typically outperform their counterparts available in the vendor libraries NVIDIA cuSPARSE and AMD hipSPARSE. We accompany te kernel release with a performance database and a web tool that allows investigating the performance characteristics interactively. We also conducted an extensive SpMV performance comparison on both AMD RadeonVII and NVIDIA V100 hardware. We show that despite NVIDIA's V100 providing better performance for many cases, AMD's RadeonVII with the hipSPARSE library is able to compete against NVIDIA's V100 in particular for matrices with a high number of non zero elements. In addition, we note that due to the price discrepancy between the two hardware (AMD's RadeonVII is roughly 6.6% of the price of an NVIDIA's V100), the AMD hardware provides a much better performance-per-dollar ratio. This may indicate that after a long period of NVIDIA dominating the HPC GPU market, AMD steps up to recover a serious competitor position.

Acknowledgment. This work was supported by the "Impuls und Vernetzungsfond" of the Helmholtz Association under grant VH-NG-1241, and the US Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

References

1. Anzt, H., Chow, E., Dongarra, J.: On block-asynchronous execution on GPUs. Technical report 291, LAPACK Working Note (2016)
2. Anzt, H., et al.: Load-balancing sparse matrix vector product kernels on GPUs. ACM Trans. Parallel Comput. **7**(1), 1–26 (2020). https://doi.org/10.1145/3380930
3. Barrett, R., et al.: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd edn. SIAM, Philadelphia (1994)
4. Bell, N., Garland, M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC 2009, pp. 1–11. ACM, New York (2009). https://doi.org/10.1145/1654059.1654078
5. Dalton, S., Baxter, S., Merrill, D., Olson, L., Garland, M.: Optimizing sparse matrix operations on GPUs using merge path. In: 2015 IEEE International Parallel and Distributed Processing Symposium, pp. 407–416, May 2015
6. Flegar, G., Anzt, H.: Overcoming load imbalance for irregular sparse matrices. In: Proceedings of the 7th Workshop on Irregular Applications: Architectures and Algorithms, IA3 2017, pp. 1–8. ACM, New York (2017). https://doi.org/10.1145/3149704.3149767
7. Grossman, M., Thiele, C., Araya-Polo, M., Frank, F., Alpak, F.O., Sarkar, V.: A survey of sparse matrix-vector multiplication performance on large matrices. CoRR (2016). http://arxiv.org/abs/1608.00636
8. Higham, D., Higham, N.: Matlab Guide. Society for Industrial and Applied Mathematics (2005). https://doi.org/10.1137/1.9780898717891
9. Hong, C., Sukumaran-Rajam, A., Nisa, I., Singh, K., Sadayappan, P.: 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, USA, 16–20 February 2019, pp. 300–314 (2019). https://doi.org/10.1145/3293883.3295712
10. Hong, S., Kim, S.K., Oguntebi, T., Olukotun, K.: 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, USA, 12–16 February 2011, pp. 267–276 (2011). https://doi.org/10.1145/1941553.1941590
11. Langville, A.N., Meyer, C.D.: Google's PageRank and Beyond: The Science of Search Engine Rankings. Princeton University Press, Princeton (2012)
12. Merrill, D., Garland, M.: Merge-based parallel sparse matrix-vector multiplication. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, pp. 1–12. IEEE Press, Piscataway (2016). http://dl.acm.org/citation.cfm?id=3014904.3014982

13. Merrill, D., Garland, M., Grimshaw, A.S.: High-performance and scalable GPU graph traversal. ACM Trans. Parallel Comput. (TOPC) **1**(2), 1–30 (2015). https://doi.org/10.1145/2717511
14. NVIDIA Corp.: Whitepaper: NVIDIA Tesla V100 GPU Architecture (2017)
15. SuiteSparse: Matrix Collection (2018). https://sparse.tamu.edu. Accessed Apr 2018
16. TechPowerUp: GPU database (2019). https://www.techpowerup.com/gpu-specs/