Evaluating the Performance of NVIDIA's A100 Ampere GPU for Sparse and Batched Computations

Hartwig Anzt*†, Yuhsiang M. Tsai*, Ahmad Abdelfattah†, Terry Cojean*, and Jack Dongarra†‡§
*Karlsruhe Institute of Technology, {hartwig.anzt, yu-hsiang.tsai, terry.cojean}@kit.edu
†University of Tennessee, {ahmad, dongarra}@icl.utk.edu
‡Oak Ridge National Laboratory
§University of Manchester

***Abstract*—GPU accelerators have become an important backbone for scientific high performance-computing, and the performance advances obtained from adopting new GPU hardware are significant. In this paper we take a first look at NVIDIA's newest server-line GPU, the A100 architecture, part of the Ampere generation. Specifically, we assess its performance for sparse and batch computations, as these routines are relied upon in many scientific applications, and compare to the performance achieved on NVIDIA's previous server-line GPU.**

***Index Terms*—Sparse Linear Algebra, Sparse Matrix Vector Product, Batched Linear Algebra, NVIDIA A100 GPU**

I. Introduction

Over the last decade, graphics processing units (GPUs) have seen an increasing rate of adoption in high-performance computing (HPC) platforms, and in the June 2020 TOP500 list, more than half of the fastest 10 systems feature GPU accelerators [1]. At the same time, the June 2020 edition of the TOP500 is the first edition listing a system equipped with NVIDIA's new A100 GPU, the HPC line GPU of the Ampere generation. Because the scientific HPC community anticipates this GPU to be the new flagship architecture in NVIDIA's hardware portfolio, we take a look at the performance we achieve on the A100 for sparse and batched computations. The motivation is that many scientific applications are based on the discretization of partial differential equations (PDEs) and the finite element methods used in the discretization process, resulting in sparse systems often carrying an inherent block structure. Therefore, the performance of sparse and batched routines can be seen as indicative of the performance we can expect for complex scientific computing applications.

For the performance assessment, in Section II we first benchmark the bandwidth of the A100 GPU for memory-bound vector operations and compare against NVIDIA's A100 predecessor, the V100 GPU. In Section III, we review the sparse matrix vector product (`SpMV`), a central kernel for sparse linear algebra, and outline the processing strategy used in some popular kernel realizations. In Section IV, we evaluate the performance of `SpMV` kernels on the A100 GPU for more than 2,800 matrices available in the SuiteSparse Matrix Collection [2]. The `SpMV` kernels we consider in this performance evaluation are taken from NVIDIA's latest release of the cuSPARSE library and the Ginkgo linear algebra library [3]. In Section V, we compare the performance of the A100 against its predecessor for complete Krylov solver iterations, which are popular methods for iterative sparse linear system solves. As many scientific applications deal with sparse problems containing a block structure, we recall the idea of batched routines in Section VI, and evaluate their achieved performance in Section VII. We conclude in Section IX with a summary of the performance assessment results and draw some preliminary conclusions on the performance we may expect from the A100 GPU for scientific computing applications.

We emphasize that with this paper, we do not intend to provide another technical specification of NVIDIA's A100 GPU, but instead focus on the reporting of performance we observe on this architecture for sparse linear algebra operations. Still, for convenience, we append a table from NVIDIA's white paper on the NVIDIA A100 Tensor Core GPU Architecture [4], which lists some key characteristics and compares against the predecessor GPU architectures. For further information on the A100 GPU, refer to the white paper [4], and we encourage the reader to digest the performance results we present side-by-side with these technical details.

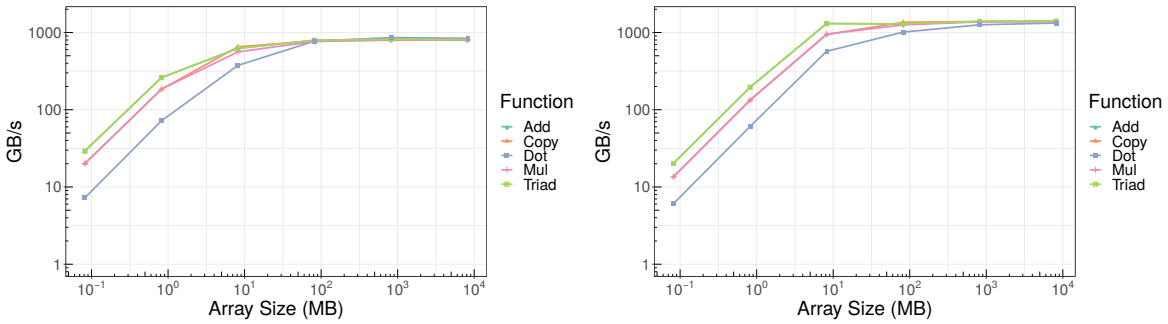
II. Memory Bandwidth Assessment

The performance of sparse linear algebra operations on modern hardware architectures is usually limited by data access rather than compute power. Consequently, for sparse linear algebra, the performance-critical hardware characteristics are the memory bandwidth and the access latency. For main memory access, both metrics are typically somewhat intertwined, in particular on processors operating in streaming mode, like GPUs. In this section, we assess the memory access performance by means of the BabelSTREAM benchmark [5].

We show the BabelSTREAM benchmark results for both an NVIDIA V100 GPU Figure 1a and an NVIDIA A100 GPU Figure 1b. The figures reflect a significant bandwidth improvement for all operations on the A100 compared to the V100. For an array of size 8.2 GB, the V100 reaches, for all operations, a performance between 800 and 840 GB/s whereas the A100 reaches a bandwidth between 1.33 and 1.4 TB/s.

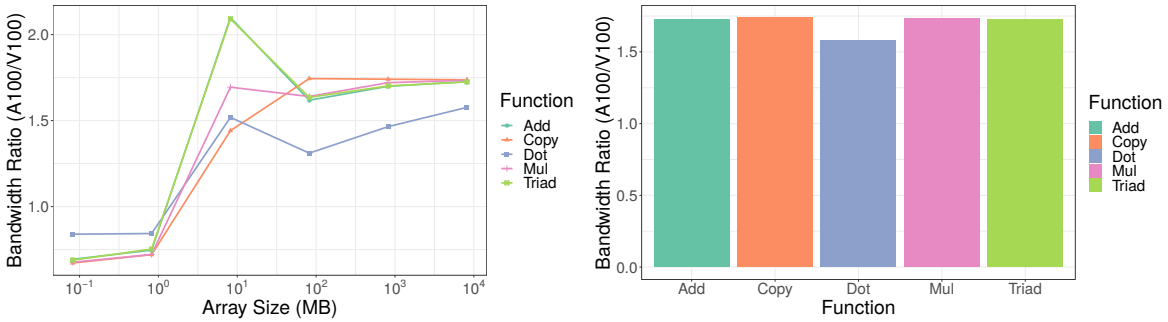
Figure 2 shows the data as a ratio between the A100 and V100 bandwidth performance for all operations. In Figure 2a, we show the performance ratio for increasing array sizes and observe the A100 providing a lower bandwidth than the V100

978-1-6654-2265-9/20/$31.00 ©2020 IEEE
DOI 10.1109/PMBS51919.2020.00009



(a) V100 (b) A100

Fig. 1: Performance of the BabelSTREAM benchmark on both V100 and A100 NVIDIA GPUs.



(a) Evolution of the A100/V100 ratio for different array sizes.

(b) A100/V100 ratio for an array of size 8.2 GB.

Fig. 2: Ratio of the performance improvement of A100 compared to V100 NVIDIA GPUs for the BabelSTREAM benchmark.

GPU for arrays of small size. For large array sizes, the A100 has a significantly higher bandwidth, converging towards a speedup factor of 1.7 for most memory access benchmarks (see Figure 2b).

III. Sparse Matrix Vector Product

The sparse matrix-vector product (`SpMV`) is a heavily used operation in many scientific applications. The spectrum ranges from the power iteration [6], an iterative algorithm for finding eigenpairs in Google's Page Rank algorithm [7], to iterative linear system solvers like Krylov solvers that form the backbone of many finite element simulations. Given this importance, we focus particularly on the performance of the `SpMV` operation on NVIDIA's A100 GPU. However, the performance not only depends on the hardware and the characteristics of the sparse matrix, but also on the specific `SpMV` format and processing strategy. Generally, all `SpMV` kernels aim to reduce the memory access cost (and computational cost) by storing only the nonzero matrix values [8]. Some formats additionally store a moderate amount of zero elements to enable faster processing when computing matrix-vector products [9]. However, independent of the strategy, since `SpMV` kernels store only a subset of the elements, they share the need to accompany these values with information that enables deduction of their location in the original matrix [10]. We here recall in Figure 3 some widespread sparse matrix storage formats and kernel parallelization techniques.

A straightforward idea is to accompany the nonzero elements with the respective row and column indices. This storage format, known as coordinate (COO [8]) format, allows determining the original position of any element in the matrix without processing other entries (see the first row in Figure 3). A standard parallelization of this approach assigns the matrix rows to the distinct processing elements (cores). However, if few rows contain a significant portion of the overall nonzeros, a significant imbalance of the kernel and poor performance can result. A workaround is to distribute the nonzeros across the parallel resources (see the right-hand side in Figure 3). However, as this can result in several processing elements contributing partial sums to the same vector output entry, sophisticated techniques for lightweight synchronization are needed to avoid write conflicts [11].

Starting from the COO format, 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 condition is technically not required, but it usually results in better

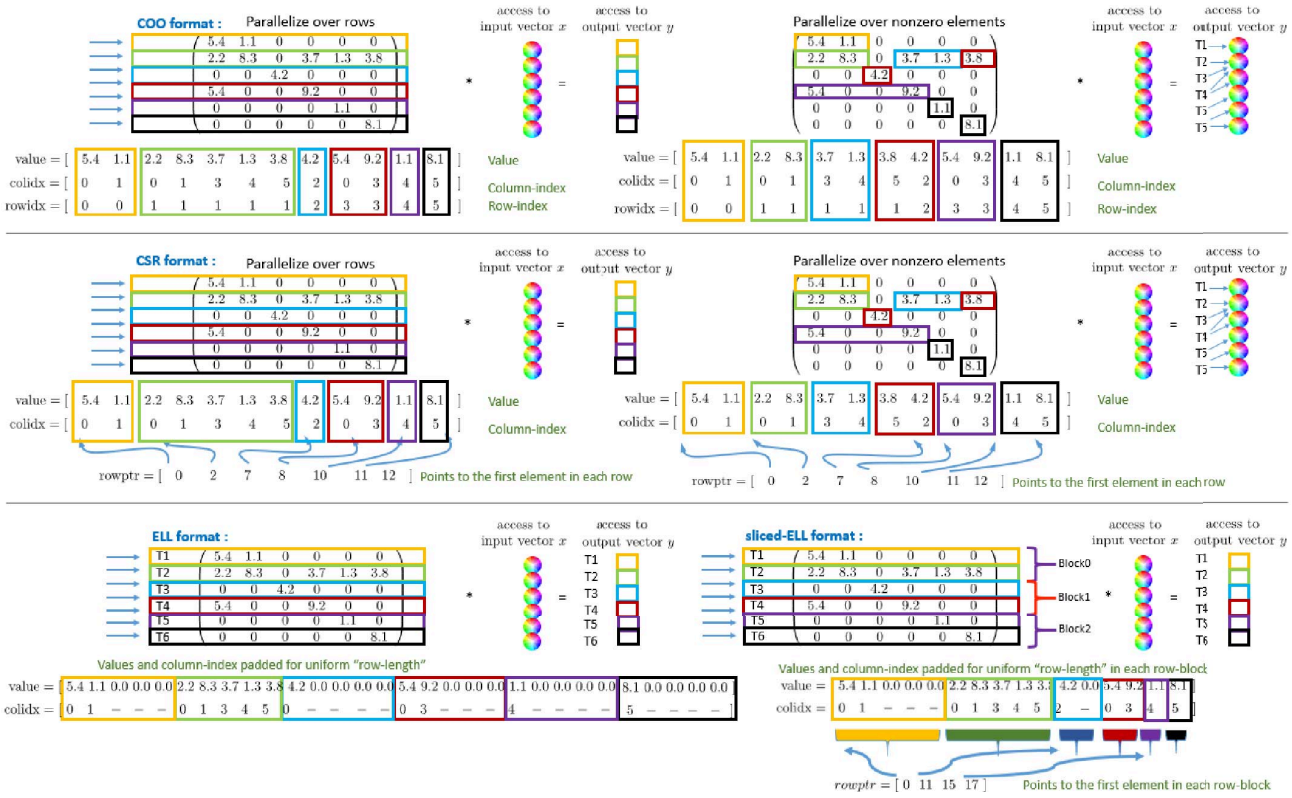
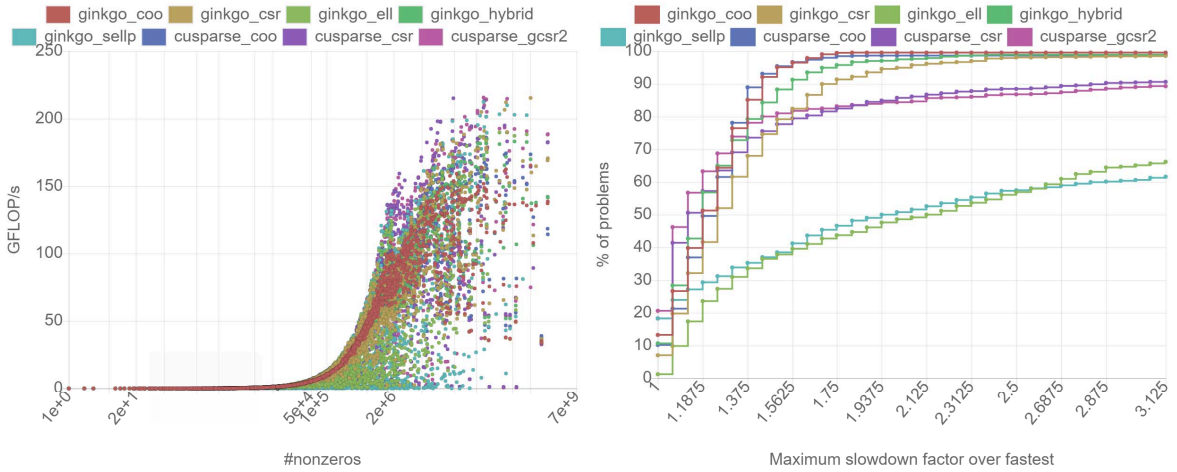


Fig. 3: Overview of sparse matrix formats and SpMV kernel design.



(a) SpMV performance profile on A100.SpMV

(b) Performance profile on A100.

Fig. 4: Left: SpMV kernel performance on the A100 GPU considering 2,800 test matrices from the SuiteSparse Matrix Collection. Right: Corresponding Performance profile for all SpMV kernels considered.

performance.) Then, this compressed sparse row (CSR [8]) format can replace the array containing the row indexes with a pointer to the beginning of the distinct rows, shown in the second row in Figure 3. While this generally reduces the data volume, the CSR format requires extra processing to determine the row location of a certain element. For a standard parallelization over the matrix rows, the row location is implicitly given, and no additional information is needed. However, similar to the COO format, better load balancing is available when parallelizing across nonzero elements. This

requires the matrix row information and sophisticated atomic writes to the output vector [12].

A strategy that reduces the row indexing information even further is to pad all rows to the same number of nonzero elements and accompany the values only with the column indices. In this ELL format [9] (the left part of the third row in Figure 3), the row index of an element can be deduced from its location in the array storing the values in consecutive order and the information of how many elements are stored in each row. While this format is attractive for vector processors as it enables execution in single instruction, multiple data (SIMD) fashion with coalesced memory access, its efficiency heavily depends on the matrix characteristics: for well-balanced matrices, it optimizes memory access cost and operation count; but if one or a few rows contain many more nonzero elements than the others, ELL introduces a significant padding overhead and quickly becomes inefficient [10].

An attractive strategy to reduce the padding overhead the ELL format introduces for unbalanced matrices is to decompose the original matrix into blocks containing multiple rows, and to store the distinct blocks in ELL format. Rows of the same block contain the same number of nonzero elements, rows in distinct blocks can differ in the number of nonzero elements. In this Sliced ELL format ([SELL] [13] see the right part of the third row in Figure 3), the row pointer can not completely be omitted like in the ELL case, but a row pointer to the beginning of every block is needed. In this sense, the SELL format is a trade-off between the ELL format on the one side and the CSR format on the other side. Indeed, choosing a block size of 1 results in the CSR format; choosing a block size of the matrix size results in the ELL format. In practice, the block size is adjusted to the matrix properties and the characteristics of the parallel hardware (i.e., SIMD-with [14]).

Another strategy that balances between the effectiveness of the ELL `SpMV` kernel and the more general CSR/COO `SpMV` kernels is to combine the formats in a "hybrid" `SpMV` kernel [10]. The concept behind this is to store the balanced part of the matrix in ELL format and the unbalanced part in CSR or COO format. The `SpMV` operation then invokes two kernels, one for the balanced part and one for the unbalanced part.

There have been significant research efforts with all these strategies to optimize the `SpMV` kernels' performance on GPU architectures (see, e.g., [10], [15]–[19] and references therein). We here focus on the `SpMV` kernels of the cuSPARSE and Ginkgo libraries that are well known to belong to the most efficient implementations available.

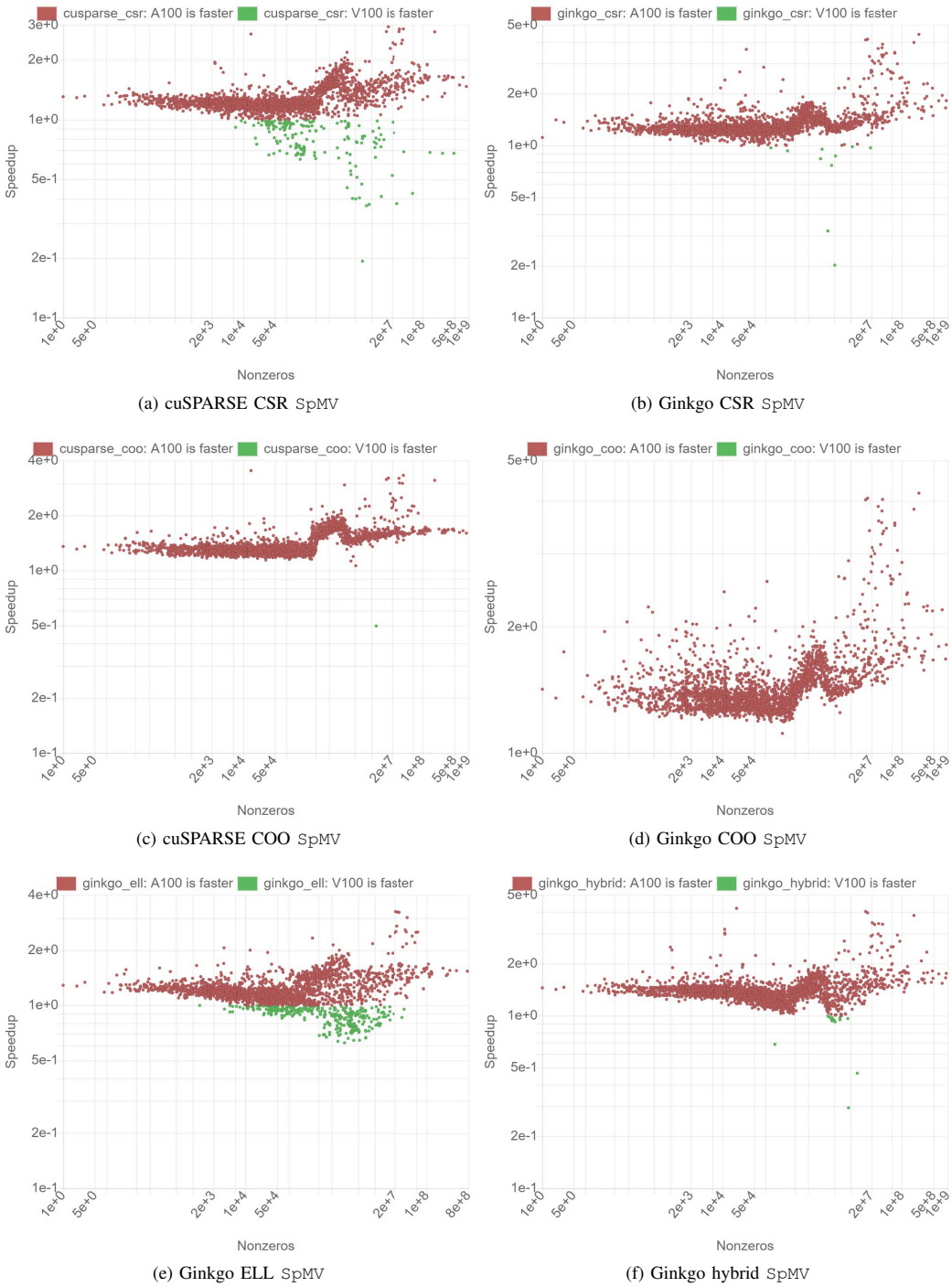
IV. SpMV Performance Assessment

For the `SpMV` kernel performance assessment, we consider more than 2,800 matrices from the SuiteSparse Matrix collection [2]. Specifically, we consider all real matrices, except for Figure 4b where we require the matrix to contain more than 1e5 nonzero elements. On both GPUs, we run the `SpMV` kernels from NVIDIA's cuSPARSE library in version 11.0 (11.0.167) and the Ginkgo open-source library version 1.3 [3] with additional IDR-solver branch.

On the left-hand side of Figure 4, we show the initial performance of all considered `SpMV` kernels on the A100 GPU for all test matrices: each dot represents the performance one of the `SpMV` kernels achieves for one test matrix. While it is impossible to draw strong conclusions, we can observe that some CSR-based `SpMV` kernels exceed 200 GFLOP/s. This is consistent with the roofline model [20]: if we assume every entry of a CSR matrix needs 12 bytes (8 bytes for the fp64 values, 4 bytes for the column index, and ignoring the row pointer), assume all vector entries are cached (ignoring access to the input and output vectors), and assume a peak memory bandwidth of 1.4TB/s, we end up with $2nnz \cdot \frac{1{,}400GB/s}{12B/nnz} \approx 230GFLOP/s$.

On the right-hand side of Figure 4, we show a performance profile [21] considering all `SpMV` kernels available in either cuSPARSE or Ginkgo for real matrices containing more than 1e5 nonzero elements. As this shows, the `cusparse_gcsr2` kernel has the largest share in terms of being the fastest kernel for a problem. However, `cusparse_gcsr2` does not generalize well: `cusparse_gcsr2` is more than $1.5\times$ slower than the fastest kernel for 20% of the problems, and more than $3\times$ slower than the fastest kernel for 10% of the problems. Although Ginkgo's CSR `SpMV` kernel is not the fastest choice for as many problems as the `cusparse_csr` and the `cusparse_gcsr2` kernels, it generalizes better, and is virtually never more than $2.5\times$ slower than the fastest kernels among all matrices. The kernels providing the best performance portability across all matrix problems are the COO `SpMV` kernels from Ginkgo and cuSPARSE: only for 10% of the problems are they more than $1.4\times$ slower than the fastest kernel. As expected, the SELLP `SpMV` kernel does not generalize well, but it is the fastest kernel for 20% of the problems.

We then evaluate the performance improvements when comparing the `SpMV` kernels on the newer NVIDIA A100 GPU and the older NVIDIA V100 GPU. Figure 5 visualizes the speedup factors for NVIDIA's cuSPARSE library (left-hand side) and the Ginkgo library (right-hand side). In the first row of Figure 5 we focus on the CSR performance. As expected, the CSR `SpMV` achieves higher performance on the newer A100 GPU. For both libraries, cuSPARSE and Ginkgo, the CSR kernels achieve for most matrices a $1.7\times$ speedup on the A100 GPU—which reflects the bandwidth increase. However, for many matrices, the speedup exceeds $1.7\times$. For Ginkgo, the acceleration of up to $5\times$ might be related to larger caches on the A100 GPU, allowing for more efficient caching of the input vector entries. For cuSPARSE, the speedup is up to $3\times$ in the large matrices. However, A100 cuSPARSE CSR is slower than V100 cusSPARSE CSR in some cases. The analysis of the performance improvement for the COO kernels is presented in the second row of Figure 5. For matrices with fewer than 500,000 nonzeros, the performance improvements are about $1.3\times$ for the Ginkgo library and the cuSPARSE library. For matrices with more than 500,000



(a) cuSPARSE CSR SpMV

(b) Ginkgo CSR SpMV

(c) cuSPARSE COO SpMV

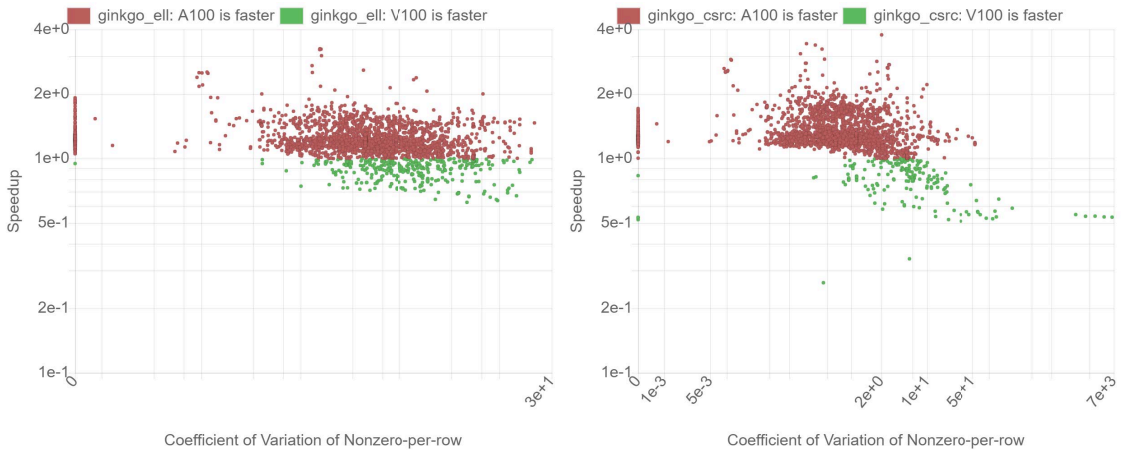
(d) Ginkgo COO SpMV

(e) Ginkgo ELL SpMV

(f) Ginkgo hybrid SpMV

Fig. 5: Performance improvement assessment of the A100 GPU over the V100 GPU for SpMV kernels from NVIDIA's cuSPARSE library and Ginkgo.

nonzero elements, the performance improvement can be up to $3\times$ in cuSPASRE and $4\times$ in Ginkgo library. In the third row of Figure 5, we visualize the performance improvements for Ginkgo's ELL and hybrid formats that do not have a direct counterpart in cuSPARSE 11.0. Again, we see that the A100 provides for most matrices about $1.4\times$ higher performance. However, especially for the ELL SpMV kernel, several matrices achieved higher performance on V100 than A100.



(a) Ginkgo Ell SpMV

(b) Ginkgo Classical Csr SpMV

Fig. 6: Performance improvement of the A100 GPU over the V100 GPU for Ell and Classical Csr from Ginkgo with coefficient of variance of nonzero-per-row.

To find the root of these outliers, we correlate the performance improvement in Figure 6 to the coefficient of variation of the nonzero-per-row metric—that is, the ratio between the standard deviation of the nonzero-per-row metric and the mean of the nonzero-per-row metric. Given the strategy Ginkgo's ELL kernel balances the work, larger coefficient of variation tend to introduce small data reads that perform poorly on the A100 GPU. Even more visible is this effect for the CSR_C SpMV kernel, the classical row-parallelized CSR SpMV kernel involving one subwarp in each row (see the right-hand side in Figure 6): irregular sparsity patterns resulting in frequent loads of small data arrays and reflected in a large coefficient of variation result in the poor performance of the A100 GPU. Remarkably, the V100 can better deal with the frequent access to small data arrays in Figure 2. Ginkgo's CSR SpMV automatically chooses between the classical CSR_C kernel providing good performance for regular sparsity patterns and the load-balancing CSR_I kernel [12] providing good performance for unbalanced sparsity patterns.

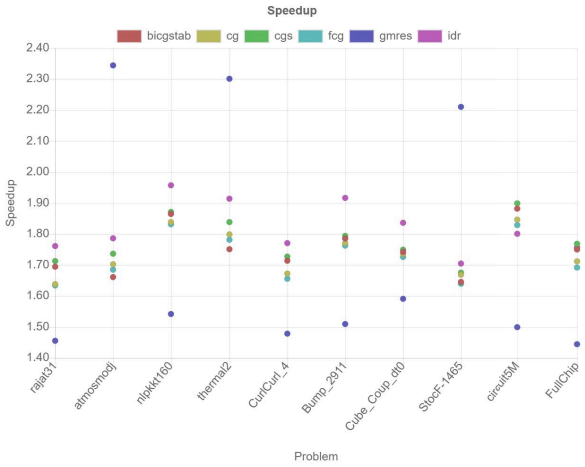


Fig. 7: Krylov solver acceleration when upgrading from the V100 GPU to the A100 GPU.

V. Krylov Solver Performance Assessment

Krylov methods are among the most efficient algorithms for solving large and sparse linear systems. When applied to a linear system $Ax = b$ (with the sparse coefficient matrix A, right-hand side b, and unknown x), Krylov solvers started with an initial guess x_0 to produce a sequence of vectors $x_1, x_2, x_3, \ldots$ that, in general, progressively reduce the norm of the residuals $r_k = b - Ax_k$, eventually yielding an acceptable approximation to the solution of the system [22].

Algorithmically, every iteration of a Krylov solver is composed of a (sparse) matrix vector product to generate the new search direction, an orthogonalization procedure, and the update of the approximate solution and the residual vector. In practice, Krylov methods are often enhanced with preconditioners to improve robustness and convergence [22]. Ignoring the preconditioner, every iteration can be composed of level-1 BLAS routines (vector operations) and a level-2 BLAS in the form of a sparse matrix vector product (SpMV). All these components—and in virtually all cases also the preconditioner application—are memory-bound operations.

When upgrading from the V100 to the A100 GPU, for the vector updates and reduction operations, we can expect performance improvements corresponding to the bandwidth improvements observed in Section II. For the basis-generating SpMV kernel, the improvements are problem-dependent and may even exceed the 1.7× bandwidth improvement (see Section IV). The total solver speedup then depends on how the Krylov method composes the SpMV and vector operations, and how these components contribute to the overall runtime.

In Figure 7, we visualize the performance improvement observed when upgrading from the V100 to the A100 GPU. We select 10 test matrices that are large in terms of size and nonzero elements, different in their characteristics, and representative for different real-world applications. The Krylov solvers are all taken from the Ginkgo library [3]; the `SpMV` kernel we employ inside the solvers is Ginkgo's COO `SpMV` kernel. For most test problems, we actually observe larger performance improvements than what the bandwidth ratios suggest. We also observe that if focusing exclusively on a single test problem, the speedup factors for the Krylov methods based on short recurrences (i.e., bicgstab, cg, cgs, fcg, idr) are all almost identical. These methods are all very similar in design, and the `SpMV` kernel takes a similar runtime fraction. The GMRES algorithm is not based on short recurrences but builds up a complete search space, and every new search direction has to be orthogonalized against all previous search directions. As this increases the cost of the orthogonalization step—realized via a classical Gram-Schmidt algorithm—the `SpMV` accounts for a smaller fraction of the algorithm runtime. As a result, the speedup for GMRES is often different than the speedup for the other methods. However, for GMRES the speedup can exceed the $1.7\times$ bandwidth improvement, as the A100 features larger caches that can be a significant advantage for the orthogonalization kernel. Overall, we observe that for most scenarios we tested, the Krylov solver executes on the A100 GPU more than $1.8\times$ faster than on the V100 GPU.

VI. Batched Linear Algebra Computations

Many scientific computing applications such as finite element simulations or circuit simulation problems result in sparse or block-sparse matrices. For example, in finite element discretizations, the unknowns associated with the same (finite) element are strongly connected and thus form a dense block in the overall sparse matrix. Depending on the size of the dense blocks, it can pay off to handle the systems as a collection of small dense systems, rather than as one sparse matrix. For the efficient processing of these problems, the so-called "batched" routines have been developed, which realize the data-parallel processing of the blocks on multi- and many-core architectures [23]. In this context, the qualifier "batched" identifies a procedure that applies the same operation to a large collection of data entities. In general, the subproblems (i.e., the data entities) are all small and independent, asking for a parallel formulation that simultaneously performs the operation on several/all subproblems in order to yield more efficient exploitation of the computational resources [24]. Batched routines are especially attractive for reducing the overall kernel launch overhead on GPUs, as they replace a sequence of kernel calls with a single kernel invocation. In addition, if the data for the subproblems is conveniently stored in the GPU memory, a batched routine can orchestrate more efficient (coalesced) memory access.

We next assess the performance batched routines achieve on NVIDIA's A100 GPU. In this performance evaluation, we focus on routines that are among the most relevant for scientific computing applications and sparse linear algebra: batched matrix matrix multiply (Batched DGEMM) [25], [26], batched LU factorization (Batched DGETRF) [27], [28], batched QR factorization (Batched DGEQRF) [29], and batched triangular solve (Batched DTRSV).

VII. Batched Routines Performance Assessment

For the performance assessment of the batched routines, we consider two scenarios: collections containing 10,000 dense problems of very small size (≤ 32), and collections containing 500 problems of moderate size (up to 1024). Figure 8 and Figure 9 show the two scenarios on the left-hand side and the right-hand side, respectively. The batched routines we benchmark are taken from NVIDIA's cuBLAS library (v11.0.167) and the MAGMA open-source library (master commit - 9ce41ca)[1], which is version 2.5.3 with cuda 11 compilation fix, on both gpu.

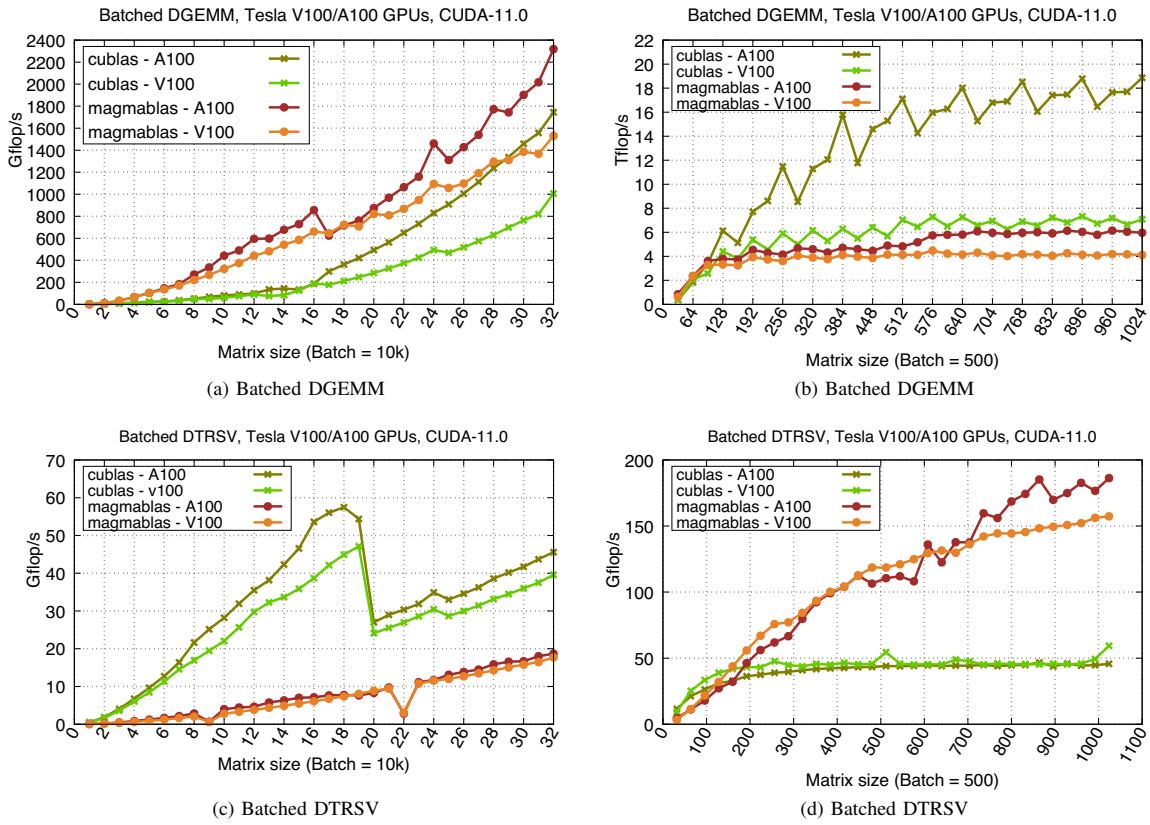
Figure 8a and Figure 8b show the performance of the batched DGEMM routine. For the small-sized problems, MAGMA's batched DGEMM reaches up to $1.6/2.4$ teraFLOP/s on the V100/A100 GPUs, respectively. This is $60\%/33\%$ faster than cuBLAS DGEMM on both GPUs. This is due to special optimizations in MAGMA that target extremely small matrices [26]. For larger problems, the situation is reversed, and cuBLAS's batched DGEMM reaches up to $6/18$ teraFLOP/s on the two GPUs, which is several times faster than MAGMA. We also notice that the gap between cuBLAS and MAGMA on the A100 GPUs is much bigger than on the V100 GPU. This is because cuBLAS takes advantage of the A100's new tensor core units, which can now accelerate FP64 arithmetic (as opposed to the V100). The MAGMA batched DGEMM routine does not currently use tensor cores. Comparing the performance of the batched DGEMM on the two GPU generations, we notice that for both MAGMA and cuBLAS, the performance advantages tend to increase with the problem size. Overall, MAGMA and cuBLAS run up to $1.5\times$ and up to $3\times$ faster on the new A100 GPU, respectively.

Figure 8c and Figure 8d show the performance of the batched TRSV routine. Overall, the performance increase for the batched TRSV routine on the A100 is rather small compared to the expectations. The MAGMA library does not currently implement specific optimization for very small problems, which explains the clear cuBLAS advantage. For medium-size problems, MAGMA has better performance than cuBLAS with CUDA v11.0.167 library.[2]

The performance of the batched LU and QR factorization routines are shown in Figure 9. The MAGMA routines have clear advantages at almost every size on both GPUs. Very small problems are specifically targeted in MAGMA with special kernels that use optimal memory traffic [29]. Medium-to-large sizes benefit from a blocked implementation that

[1]We emphasize that the MAGMA routines are currently optimized for NVIDIA's V100 GPU.

[2]We notice that CUDA fixes or improves batched_trsv on medium-size problems in the later version (11.0.229) on V100. MAGMA still has a slight advantage over it. To keep the same CUDA version on both machines, we still use CUDA library v11.0.167.



(a) Batched DGEMM

(b) Batched DGEMM

(c) Batched DTRSV

(d) Batched DTRSV

Fig. 8: Performance of the batched GEMM and TRSV routines, applied to many (10k) small-sized problems (left) and few (500) moderate-sized problems (right).

leverages the performance of the batched GEMM routine in cuBLAS. For small sizes, the batched LU factorization is $3.7\times/3.6\times$ faster than cuBLAS on the V100/A100 GPUs, respectively. The asymptotic speedup for larger sizes is about $4\times/6\times$. For the batched QR factorization, MAGMA is about $6.7\times/5.8\times$ faster than cuBLAS for very small sizes on the V100/A100 GPUs, while the asymptotic speedups are more than $30\times$ on both GPUs. Comparing the two GPU generations, small problem sizes are handled up to $1.6\times$ faster—which correlates to the bandwidth improvement—while MAGMA processes large problems about twice as fast.

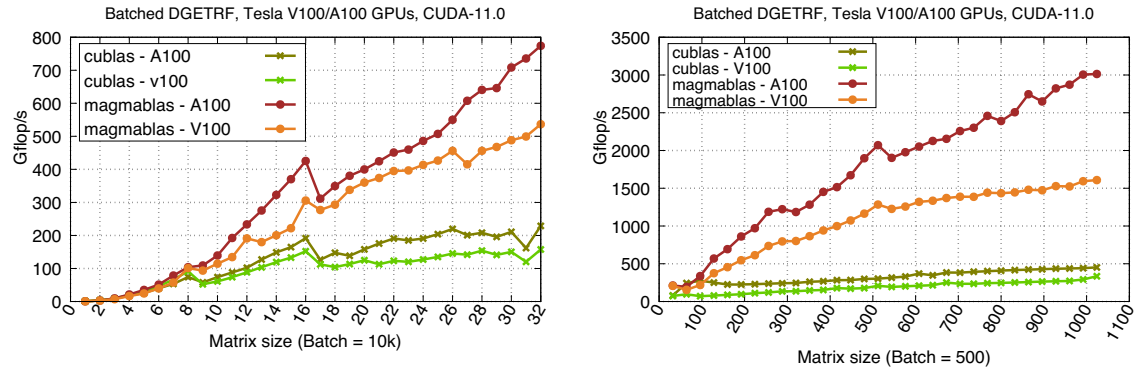
VIII. Improvement Overview

We visualize all improvement over A100 against V100 by boxplot in Figure 10 and Figure 11. We add two reference lines to help recognizing the performance improvement, one red line is the base line (1) and the other blue line is the general memory improvement (1.7). In Figure 10a, ginkgo `SpMV` does not always touch the memory improvement because the matrix sparsity leads memory movement is not enough to achieve the highest bandwidth. Except for gmres, other solver gives similar improvement around 1.7 in Figure 10b. In Figure 11a and Figure 11b, cublas gets a little better improvement than magma does except for batched_dtrsv on 500 batch and batched_dgeqrf on 10k batch.

We also summarizedthe average performance of all routines and corresponding performance improvement in Figure 12. The huge improvement of batched_dgemm on 500 batch is related to that A100 supports double precision in tensor core. On average, every routine except for cublas batched_dtrsv on 500 batch gets the new architecture benefit from higher bandwidth, performance, or more cache.

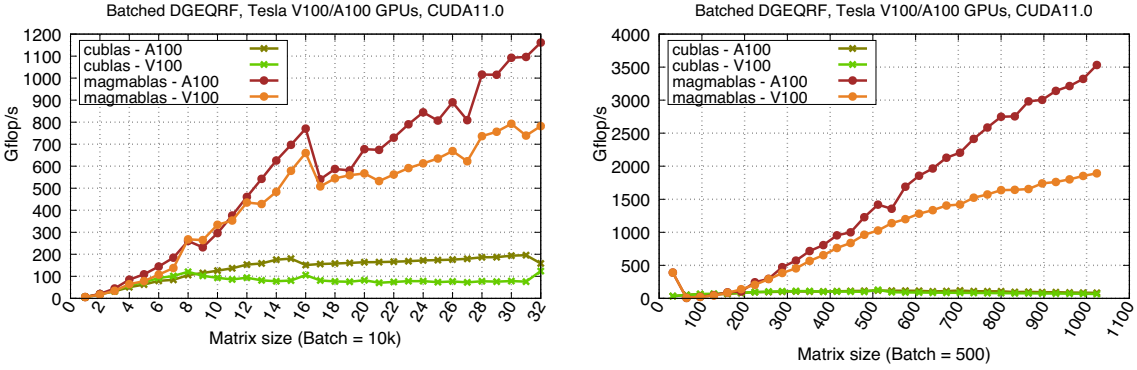
IX. Conclusion

In this paper, we assessed the performance NVIDIA's new A100 GPU achieves for sparse and batched computations. As most sparse linear algebra algorithms are memory bound, we initially present results for the STREAM bandwidth benchmark, then provide a very detailed assessment of the sparse matrix vector product performance for both NVIDIA's cuSPARSE library and the Ginkgo open-source library, and ultimately run complete Krylov solvers combining vector operations with sparse matrix vector products and orthogonalization routines. As many sparse problems coming from finite element simulations carry an inherent bock structure, we also assess the performance the A100 delivers for batched routines. We consider both the batched routines available in



(a) Batched DGETRF

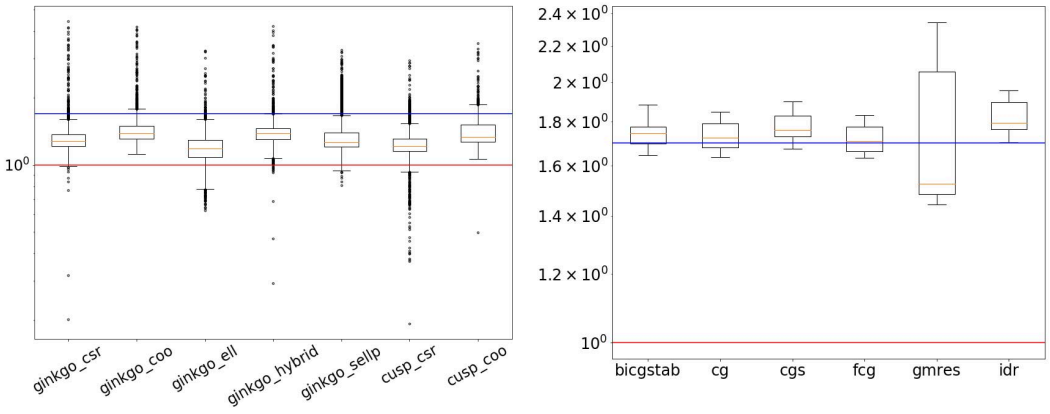
(b) Batched DGETRF



(c) Batched DGEQRF

(d) Batched DGEQRF

Fig. 9: Performance of batched LU and QR factorizations, applied to many (10,000) small-sized problems (left) and few (500) moderate-sized problems (right).

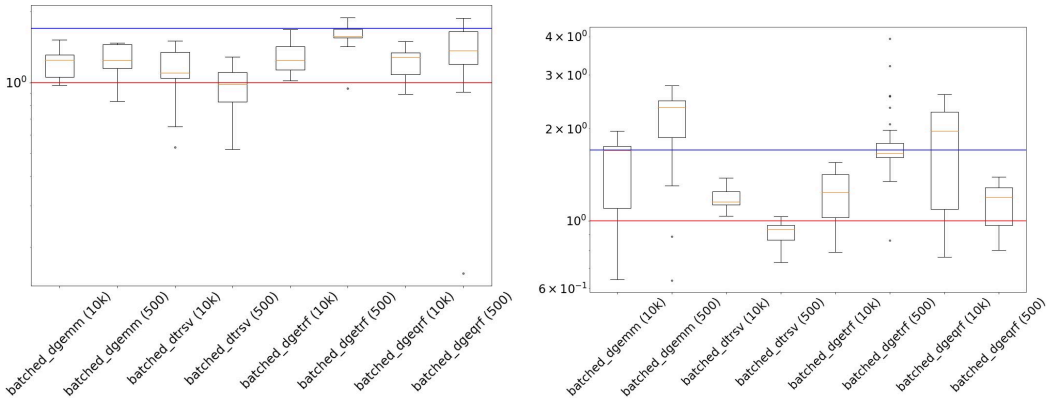


(a) SpMV Improvement

(b) Solver Improvement

Fig. 10: SpMV and Solver Performance Speedup Boxplot of A100 against V100. There are two reference lines, red line is the base = 1 and blue line is the speedup of memory in general = 1.7

NVIDIA's cuBLAS library and the MAGMA open-source library. Compared to the predecessor, the A100 GPU provides a 1.7× higher memory bandwidth, achieving almost 1.4 TB/s for large input sizes. The larger caches on the A100 allow for even higher performance improvements in complex applications like the sparse matrix vector product. Ginkgo's Krylov iterative solver runs in most cases more than 1.8× faster on the A100 GPU than on the V100 GPU. The batched routines



(a) Magma Batched Routine Improvement

(b) cuBLAS Batched Routine Improvement

Fig. 11: Batched Routine Performance Speedup Boxplot of A100 against V100. There are two reference lines, red line is the base = 1 and blue line is the speedup of memory in general = 1.7

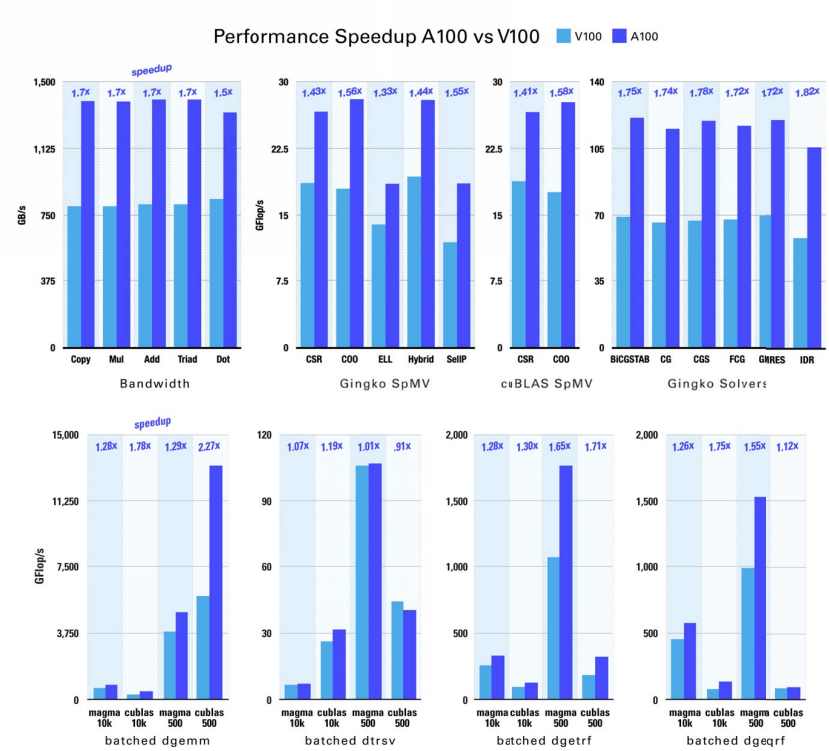


Fig. 12: Overview of the average performance of A100 against V100.

available in cuBLAS and MAGMA can run $1.45\times$ to $1.8\times$ faster on the A100 GPUs. With double-precision support in A100 tensor core, the batched DGEMM (mid-size problem batch 500) achieves up to 18 teraFLOP/s which is $3\times$ the performance on NVIDIA's V100 GPU.

ACKNOWLEDGMENTS

This work was supported by the "Impuls und Vernetzungsfond" of the Helmholtz Association under grant VH-NG-1241 and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. The authors would like to thank the Steinbuch Centre for Computing (SCC) of the Karlsruhe Institute of Technology for providing access to an NVIDIA A100 GPU.

REFERENCES

[1] The Top 500 List, http://www.top.org/.

[2] SuiteSparse, "Matrix Collection," https://sparse.tamu.edu, 2018, Accessed in April 2018.

[3] H. Anzt, T. Cojean, G. Flegar, F. Göbel, T. Grützmacher, P. Nayak, T. Ribizel, Y. M. Tsai, and E. S. Quintana-Ortí, "Ginkgo: A modern linear operator algebra framework for high performance computing," 2020.

[4] Nvidia, "NVIDIA A100 Tensor Core GPU Architecture ," https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf, June 2020.

[5] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, "Gpu-stream v2.0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models," in *High Performance Computing*, M. Taufer, B. Mohr, and J. M. Kunkel, Eds. Cham: Springer International Publishing, 2016, pp. 489–507.

[6] G. M. D. Corso, "Estimating an eigenvector by the power method with a random start," *SIAM J. Matrix Anal. Appl.*, vol. 18, no. 4, pp. 913–937, Oct. 1997. [Online]. Available: https://doi.org/10.1137/S0895479895296689

[7] A. N. Langville and C. D. Meyer, *Google's PageRank and Beyond: The Science of Search Engine Rankings*. Princeton, NJ, USA: Princeton University Press, 2012.

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

[9] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 18:1–18:11. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654078

[10] H. Anzt, T. Cojean, C. Yen-Chen, J. Dongarra, G. Flegar, P. Nayak, S. Tomov, Y. M. Tsai, and W. Wang, "Load-Balancing Sparse Matrix Vector Product Kernels on GPUs," *ACM Trans. Parallel Comput.*, vol. 7, no. 1, Mar. 2020. [Online]. Available: https://doi.org/10.1145/3380930

[11] G. Flegar and H. Anzt, "Overcoming load imbalance for irregular sparse matrices," in *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA3'17. New York, NY, USA: ACM, 2017, pp. 2:1–2:8. [Online]. Available: http://doi.acm.org/10.1145/3149704.3149767

[12] G. Flegar and E. S. Quintana-Ortí, "Balanced csr sparse matrix-vector product on graphics processors," in *Euro-Par 2017: Parallel Processing*, F. F. Rivera, T. F. Pena, and J. C. Cabaleiro, Eds. Cham: Springer International Publishing, 2017, pp. 697–709.

[13] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. 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*, vol. 36, no. 5, pp. C401–C423, 2014. [Online]. Available: http://dx.doi.org/10.1137/130930352

[14] H. Anzt, S. Tomov, and J. Dongarra, "Implementing a Sparse Matrix Vector Product for the SELL-C/SELL-C-σ formats on NVIDIA GPUs," University of Tennessee, Tech. Rep. ut-eecs-14-727, March 2014.

[15] S. Dalton, S. Baxter, D. Merrill, L. Olson, and M. Garland, "Optimizing sparse matrix operations on gpus using merge path," in *2015 IEEE International Parallel and Distributed Processing Symposium*, May 2015, pp. 407–416.

[16] D. Merrill, M. Garland, and A. S. Grimshaw, "High-performance and scalable GPU graph traversal," *TOPC*, vol. 1, no. 2, pp. 14:1–14:30, 2015. [Online]. Available: https://doi.org/10.1145/2717511

[17] D. Merrill and M. Garland, "Merge-based parallel sparse matrix-vector multiplication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 58:1–58:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=3014904.3014982

[18] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan, "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, February 16-20, 2019*, 2019, pp. 300–314. [Online]. Available: https://doi.org/10.1145/3293883.3295712

[19] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the Web," in *Proceedings of the 7th International World Wide Web Conference*, Brisbane, Australia, 1998, pp. 161–172. [Online]. Available: citeseer.nj.nec.com/page98pagerank.html

[20] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009. [Online]. Available: http://doi.acm.org/10.1145/1498765.1498785

[21] N. Gould and J. Scott, "A note on performance profiles for benchmarking software," *ACM Trans. Math. Softw.*, vol. 43, no. 2, Aug. 2016. [Online]. Available: https://doi.org/10.1145/2950048

[22] H. Anzt, M. Gates, J. Dongarra, M. Kreutzer, G. Wellein, and M. Köhler, "Preconditioned Krylov solvers on GPUs," *Parallel Computing*, vol. 68, pp. 32–44, oct 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819117300777

[23] A. Haidar, T. Dong, P. Luszczek, S. Tomov, and J. J. Dongarra, "Batched matrix computations on hardware accelerators based on GPUs," *IJHPCA*, vol. 29, no. 2, pp. 193–208, 2015. [Online]. Available: https://doi.org/10.1177/1094342014567546

[24] H. Anzt, J. Dongarra, G. Flegar, E. S. Quintana-Ortí, and A. E. Tomás, "Variable-size batched gauss-huard for block-jacobi preconditioning," *Procedia Computer Science*, vol. 108, pp. 1783 – 1792, 2017, international Conference on Computational Science, {ICCS} 2017, 12-14 June 2017, Zurich, Switzerland. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050917307731

[25] A. Abdelfattah, A. Haidar, S. Tomov, and J. J. Dongarra, "Performance, design, and autotuning of batched GEMM for gpus," in *High Performance Computing - 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*, 2016, pp. 21–38. [Online]. Available: https://doi.org/10.1007/978-3-319-41321-1_2

[26] I. Masliah, A. Abdelfattah, A. Haidar, S. Tomov, M. Baboulin, J. Falcou, and J. J. Dongarra, "High-Performance Matrix-Matrix Multiplications of Very Small Matrices," in *Euro-Par 2016: Parallel Processing - 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings*, 2016, pp. 659–671. [Online]. Available: https://doi.org/10.1007/978-3-319-43659-3_48

[27] A. Abdelfattah, A. Haidar, S. Tomov, and J. J. Dongarra, "Factorization and Inversion of a Million Matrices using GPUs: Challenges and Countermeasures," in *International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland*, ser. Procedia Computer Science, vol. 108. Elsevier, 2017, pp. 606–615. [Online]. Available: https://doi.org/10.1016/j.procs.2017.05.250

[28] A. Abdelfattah, S. Tomov, and J. J. Dongarra, "Progressive Optimization of Batched LU Factorization on GPUs," in *2019 IEEE High Performance Extreme Computing Conference, HPEC 2019, Waltham, MA, USA, September 24-26, 2019*. IEEE, 2019, pp. 1–6. [Online]. Available: https://doi.org/10.1109/HPEC.2019.8916270

[29] A. Abdelfattah, A. Haidar, S. Tomov, and J. J. Dongarra, "Batched one-sided factorizations of tiny matrices using gpus: Challenges and countermeasures," *J. Comput. Sci.*, vol. 26, pp. 226–236, 2018. [Online]. Available: https://doi.org/10.1016/j.jocs.2018.01.005

APPENDIX A
NVIDIA V100/A100 ARCHITECTURE

Features	V100	A100
GPU Architecture	NVIDIA Volta	NVIDIA A100
SMs	80	108
TPcs	40	54
FP32 Cores/SM	64	64
FP64 Cores/SM	32	32
INT32 Cores/SM	64	64
GPU Boost Clock	1530 MHz	1410 MHz
Peak FP16 TFLOPS	31.4	78
Peak FP32 TFLOPS	15.7	19.5
Peak FP64 TFLOPS	7.8	9.7
Texture Units	320	432
Memory Interface	4096-bit HBM2	5120-bit HBM2
Memory Data Rate	877.5 MHz DDR	1215 MHz DDR
Memory Bandwidth	900 GB/sec	1555 GB/sec
L2 Cache	6144 KB	40960 KB
Shared Memory Size/SM	96 KB	164 KB
Register File Size	256 KB	256 KB
Transistors	21.1 billion	54.2 billion

TABLE I: NVIDIA GPU architecture comparison [4].

APPENDIX B
ARTIFACT DESCRIPTION/ARTIFACT EVALUATION

A. Summary of the Experiments Reported

We ran SpMV, Krylov solver, and Batched Routine on Summit (V100 GPU) and KIT HoreKa (A100 GPU) using CUDA library v11.0.167, Ginkgo library v1.3.0 with IDR branch, MAGMA master branch (commit 9ce41ca, which fixes cuda11 compilation issue based on MAGMA v2.5.3)

B. Artifact Availability

- Software Artifact Availability: All author-created software artifacts are maintained in a public repository under an OSI-approved license.
- Hardware Artifact Availability: There are no author-created hardware artifacts.
- Data Artifact Availability: Some author-created data artifacts are NOT maintained in a public repository or are NOT available under an OSI-approved license
- Proprietary Artifacts:No author-created artifacts are proprietary.

C. Author artifacts

- Ginkgo: https://ginkgo-project.github.io
  (repo: https://github.com/ginkgo-project/ginkgo)
  @misc{anzt2020ginkgo,
  title = {Ginkgo: A Modern Linear Operator Algebra Framework for High Performance Computing},
  author = {Hartwig Anzt and Terry Cojean and Goran Flegar and Fritz Göbel and Thomas Grützmacher and Pratik Nayak and Tobias Ribizel and Yuhsiang Mike Tsai and Enrique S. Quintana-Ortí},
  year = {2020},
  eprint = {2006.16852},
  archivePrefix = {arXiv},
  primaryClass = {cs.MS}
  }
- MAGMA: https://icl.cs.utk.edu/magma
  (repo: https://bitbucket.org/icl/magma/src/master)
  @article{tdb10,
  title = {{Towards dense linear algebra for hybrid GPU accelerated manycore systems}},
  author = {Stanimire Tomov and Jack Dongarra and Marc Baboulin},
  booktitle = {Parallel Matrix Algorithms and Applications},
  doi = {10.1016/j.parco.2009.12.005},
  issn = {0167-8191},
  journal = {Parallel Computing},
  month = jun,
  number = {5-6},
  pages = {232–240},
  posted-at = {2010-12-17 09:48:58},
  priority = {2},
  volume = {36},
  year = {2010}
  }

D. V100 Experimental Setup

- Relevant hardware details: System name - Summit; POWER9 CPUs ; NVIDIA VOLTA 100 GPUs;
- Operating systems and versions
- Compilers and versions: gcc/g++ v6.4.0; nvcc v11.0.167
- Applications and versions: Ginkgo v1.3.0 with IDR branch, MAGMA master branch (commit 9ce41ca, which fixes cuda11 compilation issue based on MAGMA v2.5.3)
- Libraries and versions: CUDA Library v11.0.167
- Key algorithms: SpMV(Coo, Csr, Ell, Hybrid, Sellp), Krylov solver(BiCGSTAB, CG, CGS, FCG, GMRES, IDR), Batched routine(gemm, trsv, getrf, geqrf)
- Input datasets and versions: all real matrices from SuiteSparse

E. A100 Experimental Setup

- Relevant hardware details: System name - HoreKa; AMD EPYC 7742 64-Core CPUs ; NVIDIA Ampere 100 GPUs;
- Operating systems and versions: Ubuntu 18.04.4 LTS
- Compilers and versions: gcc/g++ v7.5.0; nvcc v11.0.167
- Applications and versions: Ginkgo v1.3.0 with IDR branch, MAGMA master branch (commit 9ce41ca, which fixes cuda11 compilation issue based on MAGMA v2.5.3)
- Libraries and versions: CUDA Library v11.0.167
- Key algorithms: SpMV(Coo, Csr, Ell, Hybrid, Sellp), Krylov solver(BiCGSTAB, CG, CGS, FCG, GMRES, IDR), Batched routine(gemm, trsv, getrf, geqrf)
- Input datasets and versions: all real matrices from SuiteSparse

F. Artifact Evaluation

- Performed verification and validation studies: In SpMV, we check the result with Ginkgo's Coo; In Krylov solver, we use NaN as the required residual to enforce the same maximum iteration, so do not use the checks; In Batched routine, we compare the MAGMA results with cuBLAS's.
- Validated the accuracy and precision of timings: In SpMV and Batched routine, we ran several times to get the average time for more reliable time. However, the time should be still reliable because we have warmup iteration to avoid the initial issue and solvers involving many kernels to reduce the effect of unbalanced time of each kernel.
- Used manufactured solutions or spectral properties: N/A
- Quantified the sensitivity of your results to initial conditions and/or parameters of the computational environment: We apply some warmup iterations to avoid the initial issue.
- Describe controls, statistics, or other steps taken to make the measurements and analyses robust to variability and unknowns in the system: N/A