Received: 31 May 2021 | Revised: 2 September 2021 | Accepted: 26 September 2021

DOI: 10.1002/spe.3041

SPECIAL ISSUE PAPER

WILEY

Using Ginkgo's memory accessor for improving the accuracy of memory-bound low precision BLAS

Thomas Grützmacher[1] | **Hartwig Anzt**[1,2] | **Enrique S. Quintana-Ortí**[3]

[1]Steinbuch Centre for Computing, Karlsruhe Institute of Technology, Karlsruhe, Germany

[2]Innovative Computing Lab, University of Tennessee, Knoxville, Tennessee, USA

[3]Departamento de Informática de Sistemas y Computadores, Universitat Politècnica de València, Valencia, Spain

Correspondence
Hartwig Anzt, Steinbuch Centre for Computing, Karlsruhe Institute for Technology, Karlsruhe, Germany.
Email: hartwig.anzt@kit.edu

Funding information
Helmholtz-Gemeinschaft, Grant/Award Number: VH-NG-1241; US Exascale Computing Project, Grant/Award Number: 17-SC-20-SC

Abstract

The roofline model not only provides a powerful tool to relate an application's performance with the specific constraints imposed by the target hardware but also offers a graphic representation of the balance between memory access cost and compute throughput. In this work, we present a strategy to break up the tight coupling between the precision format used for arithmetic operations and the storage format employed for memory operations. (At a high level, this idea is equivalent to compressing/decompressing the data in registers before/after invoking store/load memory operations.) In practice, we demonstrate that a "memory accessor" that hides the data compression behind the memory access, can virtually push the bandwidth-induced roofline, yielding higher performance for memory-bound applications using high precision arithmetic that can handle the numerical effects associated with lossy compression. We also demonstrate that memory-bound applications operating on low precision data can increase the accuracy by relying on the memory accessor to perform all arithmetic operations in high precision. In particular, we demonstrate that memory-bound BLAS operations (including the sparse matrix-vector product) can be re-engineered with the memory accessor and that the resulting accessor-enabled BLAS routines achieve lower rounding errors while delivering the same performance as the fast low precision BLAS.

KEYWORDS

accessor, floating-point formats, high performance, memory-bound algorithms, mixed precision, roofline model

1 | INTRODUCTION

Over the past decades, the arithmetic performance of computer processor architectures has steadily grown at a faster pace than the memory bandwidth[1-4] and, without groundbreaking changes in the chip technology, we can expect this difference to remain constant or even increase. The gap between processor and memory throughput exerts a severe

Abbreviations: BLAS, Basic Linear Algebra Subprogram; CSR, compressed sparse row; FLOP/s, floating-point operations per second; `GEMV`, general dense matrix-vector product; MB, machine balance; `SpMV`, sparse matrix-vector product; `TRSV`, triangular matrix-vector solve.

impact on the practical performance that applications can achieve on current (and future) processor architectures,[5] as the roofline model[6] graphically illustrates. This model provides a well-established tool for the performance analysis of applications, combining two architectural factors—the processor compute arithmetic peak and the memory bandwidth, with the algorithm's arithmetic intensity in order to yield an upper bound on the algorithm's performance. Several efforts have recently refined the basic formulation of the roofline model by taking into account specific architectural features.[7-9] All these variations of the model though, still distinguish between *compute-bound* and *memory-bound* algorithms, depending on their arithmetic intensity being, respectively, higher or lower than the hardware-specific *machine balance* (MB). Concretely, the performance of a compute-bound algorithm is limited by the number of arithmetic operations the processor can execute in a time interval, while the performance of a memory-bound algorithm is constrained by the speed at which data can be retrieved from the memory into the arithmetic units. (In reality, the situation is significantly more complicated, and many additional factors play a role, such as the arithmetic imbalance of the algorithm, caches, coalesced data access, etc.)

Generally speaking, on the same hardware architecture, a compute-bound algorithm will achieve higher performance in terms of floating-point operations per second (FLOP/s) than a memory-bound algorithm. (For brevity, in the following we will consider only floating-point data and floating-point operations when discussing the performance of an algorithm/architecture.) In Figure 1, we visualize this effect for a simple roofline model relating an algorithm's performance with its arithmetic intensity and the constraints imposed by the target architecture's memory bandwidth and arithmetic peak. For many applications, it is extremely difficult or even impossible to find an algorithm that exhibits high arithmetic intensity and, therewith, features the "appealing" compute-bound nature. Thus, already today a significant portion of the scientific applications running on flagship supercomputers are memory-bound.[10] Even more alarming: A faster growth of the arithmetic peak compared with that of the memory bandwidth hints that an algorithm that is compute-bound on a current processor may eventually become memory-bound on a future hardware architecture. In consequence, the performance boost that many relevant applications can obtain from newer supercomputers will progressively diminish.

The roofline model presumes a tight coupling between the precision format adopted by the arithmetic operations and the precision format used in the memory accesses. In Reference 11, the authors proposed to break up this coupling and allow for more compact precision formats in the memory operations to reduce the pressure on memory while preserving high precision in the arithmetic operations acting on data in registers. This idea of reducing the volume of the data retrieved from memory is similar to the techniques exploited in data compression;[12-14] however, it differs in that it occurs at the level of the processor registers and, therefore, it cannot take advantage of data regression techniques for efficient compression.[15] The strategy of simply converting the data to a lower precision format and storing the data in memory in that compact format yields an attractive and efficient compression strategy if the algorithm properties can cope with it. The expectation is that, by compressing the data, the bandwidth-induced performance bound in the roofline model is pushed upwards, increasing the performance of the algorithms while maintaining the arithmetic intensity; see Figure 1.

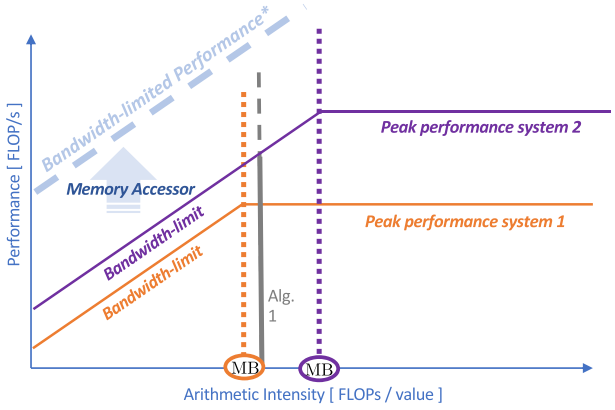

FIGURE 1 Roofline model predicting an algorithm's performance on two hardware systems with different machine balances. The "MB" mark identifies the threshold from which the algorithm is memory- or compute-bound. For example, Algorithm 1 is compute-bound on system 1 and memory-bound on system 2. The modular precision ecosystem realizing all data accesses via the memory accessor discussed in our work virtually pushes the bandwidth-induced performance roofline upwards, allowing the algorithm to achieve higher arithmetic performance without changing the underlying architecture

In this article, we take a significant step toward producing a portable modular precision ecosystem that decouples the memory precision from the arithmetic precision by making the following new contributions:

- Using the ideas proposed in Reference 11, we present a memory accessor realization that induces only a negligible interface overhead and hides the data conversion details behind the memory operations.
- We assess the performance characteristics of the memory accessor by presenting experimental performance rooflines for GPUs from AMD and NVIDIA, as well as a CPU from Intel (similar results were obtained for a CPU from AMD).
- We re-engineer the memory-bound Basic Linear Algebra Subprogram (BLAS) operations in order to leverage the memory accessor for format decoupling, and we elaborate error and performance analyses comparing the accessor-BLAS to the BLAS routines provided in vendor libraries.

The rest of the document is structured as follows. In Section 2, we motivate the development of a memory accessor and detail its implementation making heavy use of modern C++ features. After a brief discussion of related work next, in Section 3, we provide a comprehensive analysis on the performance characteristics and overhead of the memory accessor implementation on AMD GPUs (Section 3.1), NVIDIA GPUs (Section 3.2), and OpenMP-supporting CPUs (Section 3.3). In Section 4, we present details on how we use the memory accessor to reimplement memory-bound BLAS operations, and we elaborate on the error bounds and performance of the accessor-BLAS routines on recent processor architectures. We conclude in Section 5 with a summary of the findings and an outlook of which problems we plan to tackle using the accessor-enabled BLAS as building blocks.

1.1 | Related work

In the linear algebra domain, attacking the memory wall is a well-known target for many dense and sparse computational kernels. Consider, for example, a key numerical kernel for the iterative solution of sparse linear systems such as the sparse matrix-vector product.[16] Reducing the pressure on the memory bandwidth for this kernel has been pursued via the design of specialized data structures for the sparse matrices that reduce the volume of "indexing" information, introduce blocking to increase the number of cache hits, combine mixed precision with iterative refinement, and so forth. For a detailed examination of these efforts, see References 17 and 18 and the references therein. All these techniques though share the principle of maintaining the floating-point values in memory and perform the arithmetic operations in the floating-point units using the same precision format.

The memory accessor concept proposed in Reference 11 also addresses the memory constraint, but departs from these other conventional approaches by considering two decoupled precision formats: a "low" precision format to maintain the data in memory and a "high" precision format for the arithmetic operations. As a result, this approach can be viewed as maintaining the problem data compressed in memory, similar to the compression techniques for file systems.[12-14] (Alternatively, taking the low precision format as a reference point, the memory accessor can also be viewed as preserving the precision of the original data but performing the arithmetic in an extended precision.)

Lossy compression has been successfully leveraged in various applications with the purpose of storing either the initial or the final data.[19-21] More recently, in Reference 22, the authors examine the flexible GMRES iterative solver, showing that the vectors spanning the search space can be compressed using a variety of practical strategies. Unfortunately, the authors validate the numerical results but do not elaborate a high-performance implementation that allows assessing the performance impact of their approach.

Interestingly, a couple of hardware vendors have recently introduced a technology similar to the memory accessor in current GPUs: The NVIDIA tensor core[23] and the AMD matrix engine[24] can use a different precision format in the arithmetic operations than what they read in from main memory. For example, the first generation tensor cores deployed in the NVIDIA V100 GPU read in two matrices in `fp16`, compute their matrix product, accumulate the result with another matrix in `fp32`, and write out the result in `fp32`. The second generation tensor cores deployed in the NVIDIA A100 has more flexibility, for example, it allows to read only the first 19 bits of a `fp32`number and compute the product of two matrices on this truncated `fp32` format (that is denoted as "TF32 format"). The goal in both cases is to accelerate the computation of the dense matrix product as this operation forms the backbone of deep learning training and inference. Thus, the decoupling of memory precision and arithmetic precision in NVIDIA tensor cores (and AMD matrix engines) is not motivated by the faster memory access, but by faster computations in low precision. Also, compared to our software-based memory accessor, the hardware realization allows for much less flexibility in terms of supported kernels and precision formats.

2 | IMPLEMENTATION OF THE MEMORY ACCESSOR

To realize the idea of decoupling the precision format so that the arithmetic calculations are performed in a precision format that is different from that used for storing and retrieving data from main memory,[11] we need a technical realization of a "memory accessor" that on-the-fly handles the conversion (compression/decompression) between formats. For such a memory accessor to be useful in practice, it has to combine multiple characteristics. In particular, it has to be:

- general to handle a wide range of floating-point formats and compression strategies;
- flexible to accommodate new precision formats in the future;
- opaque to abstract the user from the implementation details;
- portable across different hardware architectures; and
- efficient to avoid introducing runtime overhead, hiding the conversion costs behind the memory access.

The memory accessor in this work presents all these alluring features. The actual realization is deployed in the Ginkgo open source library[25] (https://ginkgo-project.github.io/), which is written in C++14, but we reduce the dependency on other Ginkgo components, therewith allowing for the easy integration and utilization of the memory accessor in other software. The memory accessor itself is implemented using static (compile-time) polymorphism, which introduces negligible overhead and enables full compiler optimizations.

The implementation consists of three layers, which we describe in some detail in the following subsections:

1. The range acts as the common interface for all memory accessors.
2. The memory accessor itself is responsible for translating the index information into read/write accesses.
3. An optional reference class, which is necessary if the memory accessor needs to perform any computations to translate between the memory format and the arithmetic format.

While these technical details are important to realize a flexible and usable high-performance implementation of the memory accessor, in Figure 2, we visualize the high-level idea of the memory accessor and its use from a computational kernel.

2.1 | The memory accessor interface: Range

A `range` can be considered as the front-end or wrapper class for all accessors. Its purpose is to define a simple and minimal interface for memory accesses. The interface takes an accessor type as a template parameter; and the parenthesis operator, used for both read and write operations, is at its core. For example, a read from a two-dimensional

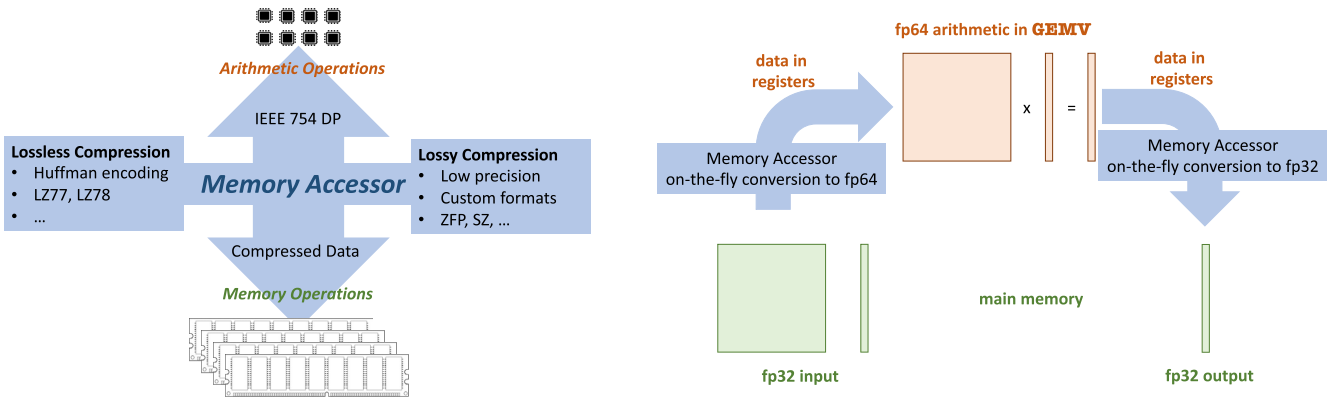

FIGURE 2 Overall idea of the memory accessor compressing data on-the-fly in memory access (left), and an example of use from a matrix-vector product (`GEMV`) kernel using `fp32` as memory format and `fp64` as arithmetic format (right)

range `mat` is performed as in `value = mat(2,3)`, while the write counterpart to the same location is done as `mat(2,3) = value`. The implementation of the `range` is shown in Listing 1. We first look at the parenthesis operator defined in Line 19. It leverages the C++ *perfect forwarding* mechanism, which takes an arbitrary number of arguments, of any type(s), and invokes the accessor's overloaded parenthesis operator with the same arguments without modifying them. The output value of the accessor's parenthesis call is then simply returned by the function. The purpose of this function is to force the accessor to offer a parenthesis operator. Using this type of techniques, we are able to specify interface requirements for the accessor, so that the `range` can accept any type of accessor.

Additionally, all accessors need to define a `length` member function (according to Line 32) that returns the length of the given dimension in order to gain access to the size covered by the accessor.

Finally, the `range` also allows to directly interact with the underlying accessor, in Lines 36 and 42, to offer functionality that may be specific to the accessor, for example, changing a scalar value or modifying the compression strategy. Direct access is important because offering every possible functionality in a single interface is simply impossible.

```
1  template <typename Accessor>
2  class range {
3  public:
4      using accessor = Accessor;
5
6      // The number of dimensions of the range.
7      static constexpr size_type dimensionality = accessor::dimensionality;
8
9      ~range() = default;
10
11     // Creates a new range by forwarding the arguments to the accessor constructor
12     template <typename... AccessorParams>
13     GKO_ACC_ATTRIBUTES constexpr explicit range(AccessorParams &&... params)
14         : accessor_{std::forward<AccessorParams>(params)...}
15     {}
16
17     // Returns a value (or a sub-range) with the specified indexes.
18     template <typename... DimensionTypes>
19     GKO_ACC_ATTRIBUTES constexpr auto operator()(DimensionTypes &&... dimensions) const
20        -> decltype(std::declval<accessor>()(std::forward<DimensionTypes>(dimensions)...))
21     {
22         static_assert(sizeof...(dimensions) <= dimensionality,
23                       "Too many dimensions in range call");
24         return accessor_(std::forward<DimensionTypes>(dimensions)...);
25     }
26
27     range(const range &other) = default;
28
29     // Returns the length of the specified dimension of the range.
30     GKO_ACC_ATTRIBUTES constexpr size_type length(size_type dimension) const
31     {
32         return accessor_.length(dimension);
33     }
34
35     // Returns a pointer to the accessor.
36     GKO_ACC_ATTRIBUTES constexpr const accessor *operator->() const noexcept
37     {
38         return &accessor_;
39     }
40
41     // 'Returns a reference to the accessor.
42     GKO_ACC_ATTRIBUTES constexpr const accessor &get_accessor() const noexcept
43     {
44         return accessor_;
45     }
46
47 private:
48     accessor accessor_;
49 };
```

Listing 1: Full implementation of the range

2.2 | The accessor functionality: Translating index information into data access

A simple two-dimensional column-major accessor that fulfills all the requirements from `range` is shown in Listing 2. The template parameter `ValueType` specifies the type of the stored data pointer. We define the `range` class as a friend class in Line 7, and declare the constructors of the accessor as protected, in Lines 10 and 16, to force its initialization through the `range` class. As a bonus, we add bound checks to the parenthesis operator in Line 32 (which we include for all currently implemented accessors). These improve the debugging capabilities of the code yet do not impair performance since the checks are only performed in debug mode but not in release mode.

```
1  template <typename ValueType>
2  class col_major_2d {
3  public:
4      static constexpr size_type dimensionality{2};
5      using value_type = ValueType;
6
7      friend class range<col_major_2d>;  // Allow range to instantiate this class
8
9  protected:
10     GKO_ACC_ATTRIBUTES constexpr col_major_2d(
11         std::array<size_type, dimensionality> size, value_type *data,
12         size_type stride)
13         : size_(size), data_{data}, stride_{stride}
14     {}
15     // Default stride is size[0], so without any padding
16     GKO_ACC_ATTRIBUTES constexpr col_major_2d(
17         std::array<size_type, dimensionality> size, value_type *data)
18         : col_major_2d{size, data, size[0]}
19     {}
20
21 public:
22     GKO_ACC_ATTRIBUTES constexpr size_type length(size_type dim) const
23     {
24         return assert(dim < dimensionality), size_[dim];
25     }
26
27     GKO_ACC_ATTRIBUTES constexpr value_type &operator()(size_type x,
28                                                         size_type y) const
29     {
30         // Validate that indices are inside the size constraints when building
31         // in Debug mode. This is not required, but easy to add.
32         return assert(x < size_[0]), assert(y < size_[1]),
33                data_[y * stride_ + x];
34     }
35
36 private:
37     const std::array<size_type, dimensionality> size_;
38     const size_type stride_;
39     value_type *const data_;
40 };
```

Listing 2: Sample 2D column-major accessor implementation

Thanks to C++ Template Metaprogramming, we support arbitrary dimensionality for the following accessors without code duplication:

1. `row_major`: A very simple row-major accessor without separation of memory and arithmetic precision. The implementation is similar to that in Listing 2 with an additional template parameter specifying the dimensionality.
2. `block_col_major`: The lowest two dimensions are stored in column-major order, while all other dimensions are stored in row-major order. This is used to handle two-dimensional blocks stored in column-major order, as in a blocked compressed sparse row (CSR) format. This accessor uses the same memory and arithmetic precision as well.
3. `reduced_row_major`: Allows for separate memory and arithmetic precisions, and performs the conversion between the two for each access.
4. `scaled_reduced_row_major`: Similar to the `reduced_row_major`, with the addition of a scalar values used for scaling the stored values before reads and writes.

2.3 | The accessor extensibility: An additional layer for complex accessors

If the accessor needs to perform any arithmetic computations when writing, returning a plain reference in the overloaded parenthesis operation is not possible. Instead, a custom reference object is returned, which performs the necessary computations prior to returning the value/writing to memory.

The reference class used in `reduced_row_major` is shown in Listing 3. In Line 21, we define a cast operation to our arithmetic type where we incorporate the computations to perform when reading in memory precision. Here, the cast is from memory format to arithmetic precision, but it could also be a complex decompression algorithm. After the computation is done, the transformed value is returned. This reference object can now be treated as a variable of type `arithmetic_type` thanks to the C++ implicit conversions feature. If the reference object cannot be accepted in an operation, but an `arithmetic_type` value does, the compiler calls the conversion function implicitly.

Write operations are intercepted by overloading the assignment operator, as shown in Lines 28, 34, and 38 in Listing 3.

The reference class `reduced_storage` accepts any type that has a `static_cast` defined between `ArithmeticType` and `StorageType`. This includes the IEEE types `fp64`, `fp32`, and `fp16`, which have hardware support (at least the former two) for these conversions. However, it is also possible to use software types such as the `__float128` from `gcc`.

```
1  template <typename ArithmeticType, typename StorageType>
2  class reduced_storage
3      // This Mixin  defines the operators *,/,+,-, *=,/=,+=,-+
4      : public detail::enable_reference_operators<
5            reduced_storage<ArithmeticType, StorageType>, ArithmeticType> {
6  public:
7      using arithmetic_type = std::remove_cv_t<ArithmeticType>;
8      using storage_type = StorageType;
9
10     reduced_storage() = delete;
11     ~reduced_storage() = default;
12     reduced_storage(reduced_storage &&) = default;
13     // Forbid copy construction
14     reduced_storage(const reduced_storage &) = delete;
15
16     constexpr explicit GKO_ACC_ATTRIBUTES reduced_storage(storage_type *const ptr)
17         : ptr_{ptr}
18     {}
19
20     // Overload cast function to arithmetic_type for reads
21     constexpr GKO_ACC_ATTRIBUTES operator arithmetic_type() const {
22         // Important to properly apply the __restrict__ qualifier on GPUs
23         const storage_type *const GKO_ACC_RESTRICT r_ptr = ptr_;
24         return static_cast<arithmetic_type>(*r_ptr);
25     }
26
27     // Overload assignment operator for writes
28     constexpr GKO_ACC_ATTRIBUTES arithmetic_type operator=(arithmetic_type val) {
29         storage_type *const GKO_ACC_RESTRICT r_ptr = ptr_;
30         *r_ptr = static_cast<storage_type>(val);
31         return val;
32     }
33
34     constexpr GKO_ACC_ATTRIBUTES arithmetic_type operator=(const reduced_storage &ref) {
35         return *this = static_cast<arithmetic_type>(ref);
36     }
37
38     constexpr GKO_ACC_ATTRIBUTES arithmetic_type operator=(reduced_storage &&ref) {
39         return *this = static_cast<arithmetic_type>(ref);
40     }
41
42 private:
43     storage_type *const ptr_;
44 };
```

Listing 3: Shortened implementation of the reduced_storage reference class used for reduced_row_major

3 | THE EXPERIMENTAL ROOFLINE OF THE MEMORY ACCESSOR

In this section, we leverage the roofline model to assess the performance overhead of the memory accessor realization on a variety of recent processor architectures. In contrast with the conventional utilization of the roofline model as a prediction tool, the experimental roofline model is not based on hardware specifications and modeling, but on experimental results observed when running algorithms with predefined arithmetic intensity. For this evaluation, we implement a benchmark with user-defined arithmetic intensity. The test is inspired in the mixbench benchmark (https://openbenchmarking.org/test/pts/mixbench), from which we develop two variants: one that directly accesses the values in main memory, and one that relies on the memory accessor to retrieve/store values from/in main memory. For the variant retrieving the data through the memory accessor, we choose `reduced_row_major` for the data representation, as that is one of the most complex accessor configurations. The only difference between the two benchmark implementations lies in the read and write accesses while the access patterns, the order, floating-point format, and type of computations executed in the arithmetic units are identical in both cases. We note that, in order to calculate the FLOP/s rates for the variant that uses the memory accessor, we ignore the FLOPs needed for the additional computations necessary for the on-the-fly conversion between the precision formats. We consider those as "part of the data access," and they need to contribute a minor cost if we want the memory accessor to be competitive.

In summary, the benchmark implementations generate an experimental roofline model, achieving the full bandwidth for memory-bound computations and the arithmetic peak for compute-bound computations. In particular, for the standard memory access, the realization generates an experimental roofline that matches the behavior of mixbench. The variant that retrieves the data via the memory accessor will deliver different results if the memory precision differs from the arithmetic precision. Therefore, comparing the two benchmark variants in terms of actual compute performance for a certain arithmetic intensity allows us to identify the cost penalty of the type conversion (if data are compressed) as well as the overhead of the memory accessor (for all memory accessor uses).

In the experimental evaluation, we consider the following four precision configurations, using either the standard benchmark variant or its counterpart that employs the accessor for memory accesses:

1. `fp64` arithmetic and `fp64` memory access for the standard benchmark. This configuration reflects a conventional `fp64` scenario where all data is stored and processed using the standard (IEEE) 64-bit floating-point format.
2. `fp64` arithmetic and `fp64` memory access for the accessor-based benchmark. This case is conceptually identical to the standard `fp64` scenario, but employs the accessor for the data exchanges with main memory. Technically, the accessor converts between the `fp64` and `fp64`precision formats.
3. `fp32` arithmetic using `fp32` memory access for the standard benchmark. This configuration reflects a conventional `fp32` scenario.
4. `fp64` arithmetic using `fp32` memory access. This can only be realized using the accessor in the `Accessor<fp64,fp32>` configuration, working with `fp32` values in memory yet performing `fp64` arithmetic.

When comparing the performance between the test configurations, the confrontation of 1 and 2 quantifies the overhead of the memory accessor as, ideally, both configurations should achieve the same compute performance for all arithmetic intensities. Test 3 provides a reference roofline for `fp32` computations. The roofline exhibited by configuration 4 leverages the conceptual advantages of the memory accessor strategy, and should also expose the overheads of the type conversion plus that of the memory accessor.

In order to assess the performance of the memory accessor on different hardware architectures, we implemented the benchmarking module in different programming ecosystems: the HIP language for AMD GPUs, the CUDA language for NVIDIA GPUs, and as a multithreaded OpenMP C++ code for general-purpose CPUs. In the following subsections, we present the experimental rooflines for the four test configurations on representative processor architectures that we list along with some key characteristics in Table 1.

3.1 | Accessor's experimental roofline on AMD GPUs

To expose the performance characteristics of the memory accessor on AMD GPUs, we use the HIP version of the benchmark variants compiled using ROCm version 4.0.

In Figure 3, we visualize the experimental roofline relating the compute performance to the arithmetic intensity in the [FLOPs/value] metric. In the top of Figure 3, we show the results for the MI100 server line GPU; at the bottom, we report

TABLE 1 Hardware architectures considered in the experimental evaluation of the memory accessor

Hardware	Bandwidth (GB/s)	`fp64` performance (GFLOP/s)	`fp32` performance (GFLOP/s)
NVIDIA V100	900	7800	15,700
NVIDIA A100	1555	9746	19,490
AMD Radeon VII	1024	3360	13,440
AMD MI100	1200	11,500	23,100
Intel Xeon Gold 6230	282	Not public	Not public

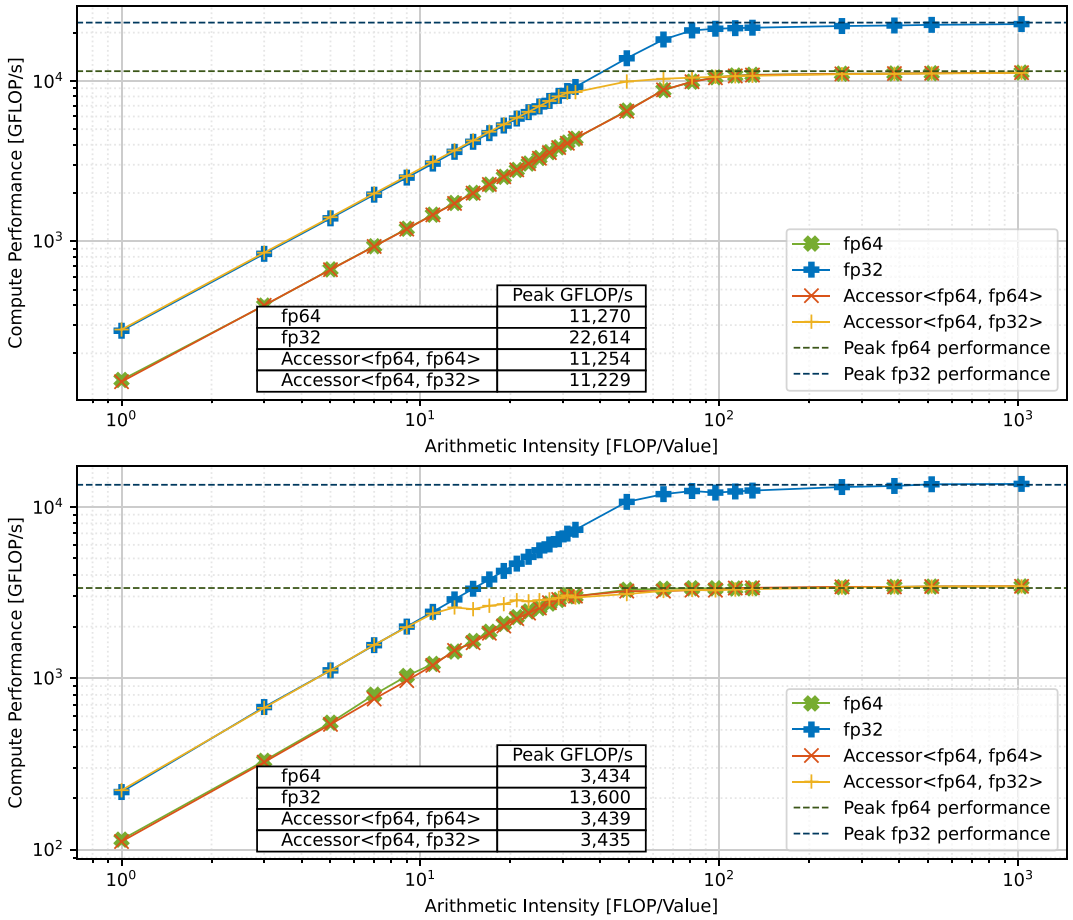

FIGURE 3 Experimental roofline performance of the memory accessor on an AMD MI100 GPU (top) and an AMD Radeon VII GPU (bottom)

the results for the Radeon VII consumer line GPU. The dashed lines indicate the (theoretical) arithmetic peaks for `fp32` (single precision) and `fp64` (double precision), as listed in the technical specifications. We notice that the benchmark using the standard memory access attains the arithmetic peak for high values of the arithmetic intensity while, in the memory-bound region, the performance line closely follows the main memory bandwidth. Also, the performance of the `Accessor<fp64,fp64>` is almost indistinguishable from the `fp64` performance. That indicates the quality of the accessor implementation as it incurs negligible overhead. As expected, for a fixed arithmetic intensity below the MB, the `fp32`configuration achieves higher performance than the `fp64` configuration. The key aspect is that, in that region, the `Accessor<fp64,fp32>` benchmark also delivers higher performance than the `fp64` configuration. The exclusive reason for this is that the accessor retrieves `fp32`values from main memory, while still relies on `fp64` arithmetic. The fact that the performance matches that of the `fp32` case reveals that the conversion between the two formats can be efficiently hidden behind the memory access. Once the arithmetic intensity equals the MB, the roofline performance of

the `Accessor<fp64,fp32>` asymptotically approaches the compute performance of the `fp64` configuration. This was expected as the arithmetic operations in `fp64`constrain the performance of the `Accessor<fp64,fp32>` benchmark for compute-bound algorithms.

3.2 | Accessor's experimental roofline on NVIDIA GPUs

In the case of NVIDIA GPUs, the performance behavior is exposed using the CUDA version of the benchmarks, compiled with CUDA version 11.0.

Figure 4 displays the experimental roofline performance for the CUDA realizations of the benchmarks on the NVIDIA V100 GPU (top) and NVIDIA A100 GPU (bottom), both belonging to the server line from NVIDIA. We also indicate the theoretical arithmetic peaks for `fp64` and `fp32` listed in the specifications. Consistently with the AMD results, the standard fixed precision benchmarks `fp32` and `fp64` follow the memory bandwidth for arithmetic intensities below the MB, and respectively, reach the `fp64` peak and `fp32` peak for arithmetic intensities superior to that. The performance of `Accessor<fp64,fp32>` again matches that of `fp32` in the memory-bound region, yet is limited by the `fp64` peak for arithmetic intensities above the MB. This reveals that the memory accessor incurs virtually no overhead, and succeeds in hiding all data conversion behind the memory accesses.

3.3 | Accessor's experimental roofline on CPUs

Implementing the benchmarks for CPUs is significantly more challenging as it has to reflect all aspects of sophisticated CPU execution: multithreading, vectorization, and compiler optimizations. We compile the benchmarks using g++ 10.2.0.

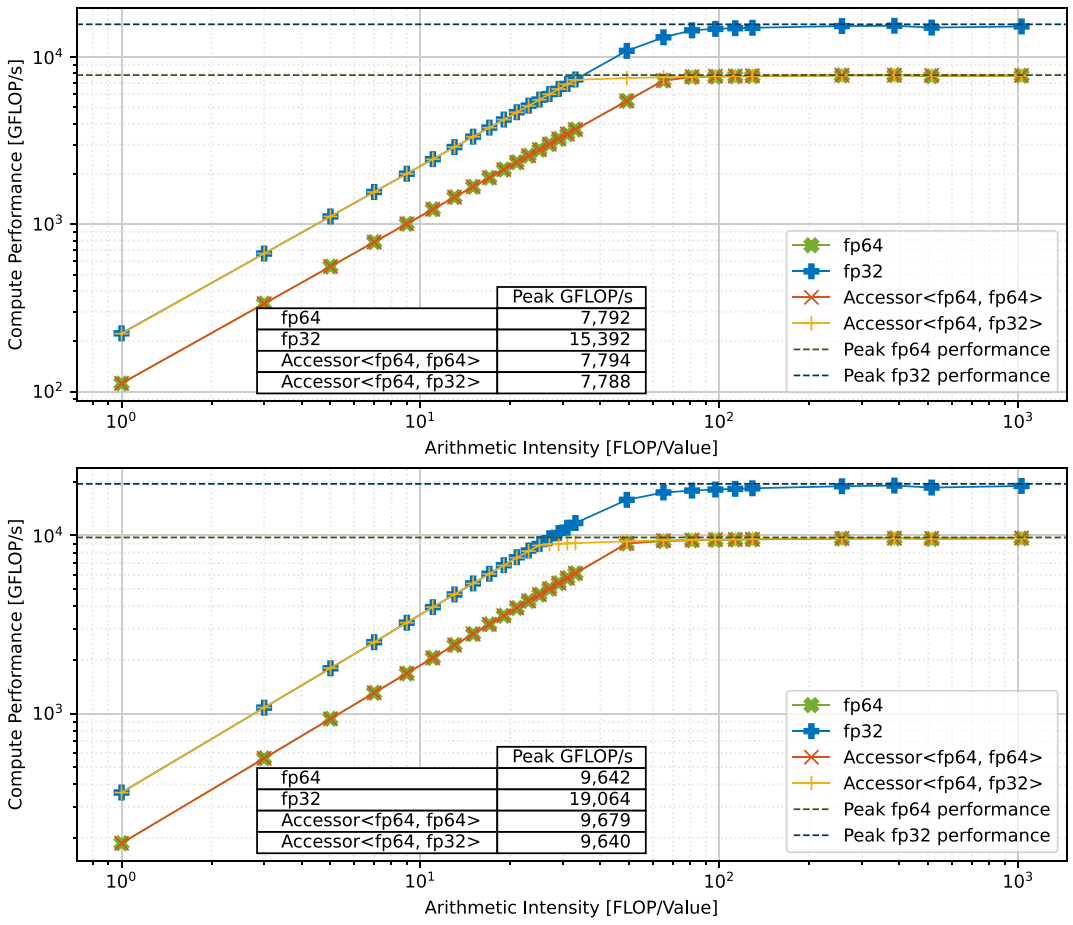

FIGURE 4 Experimental roofline performance of the memory accessor on an NVIDIA V100 GPU (top) and an NVIDIA A100 GPU (bottom)

By investigating the assembly code of the benchmark implementations for the CPU, we validate that the two benchmark variants produce very close assembly codes when using the same precision format for memory access and arithmetic operations. The main difference between them lies in the function call as one variant uses a structure (the memory accessor itself containing the size, data pointer, and strides), while its counterpart employs a plain data pointer (since the stride value was inlined). When ignoring register naming, the load, store, and vector operations are identical, but some loop conditions differ slightly (they are functionally identical, but the codes present some minor differences).

In Figure 5, we visualize the experimental roofline performance we attain with the OpenMP version of the memory accessor on an Intel Xeon Gold 6230 Processor (codename Cascade Lake). Compared with the GPU performance results, we note that we are further away from the theoretical peak specified in the technical brief. Consistently with the GPU results, we do not observe any performance degradation when accessing the values in main memory via the memory accessor, even though the operation mode of CPUs is fundamentally different from the operating mode of streaming accelerators. Interestingly, despite the additional format conversion step, the `Accessor<fp64,fp32>` benchmark outperforms the `fp64` benchmark for arithmetic intensities beyond the MB. The analysis of the assembly code in Figure 6 reveals an additional unrolling performed by the compiler in this case (which may be subject to the specific compiler): For the selected compiler, every load operation is 256-bit wide. Since we convert from `fp32` to `fp64`, we now have two 256-bit wide vector registers (`ymm4` and `ymm3` in Figure 6B) compared with only one (`ymm2` in Figure 6A). This yields a more efficient utilization of the vector pipeline, resulting in higher performance. We recall that the additional computations which are needed for the format conversion are not accounted for in the FLOP count. However, they are negligible compared with the computational load.

We close this section by noting that analogous results were obtained using the OpenMP-based benchmarks for CPUs on an AMD EPYC 7742 processor. To avoid repetition, we omit them from the article.

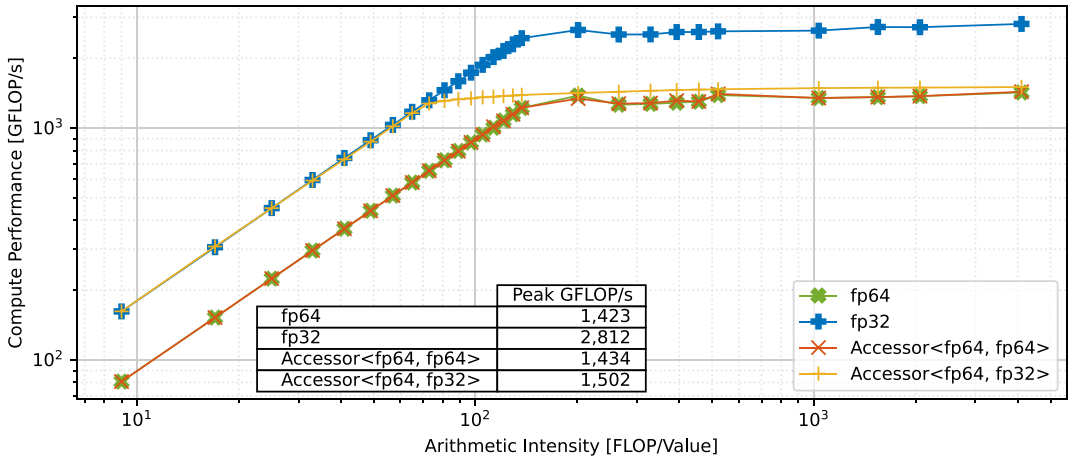

FIGURE 5 Experimental roofline performance of the memory accessor on an Intel Xeon Gold 6230 processor (codename Cascade Lake)

```
// Load 256bit vector, containing 4 fp64 values
vmovupd ymm2, ymmword ptr [rdi+rsi]
mov     eax, 0x40
vaddpd  ymm3, ymm0, ymm2
vaddpd  ymm4, ymm6, ymm2
vaddpd  ymm1, ymm5, ymm2
```

(A) Load operation for `Accessor<fp64,fp64>`

```
// Load 256bit vector, containing 8 fp32 values
vmovups ymm1, ymmword ptr [rdi+rsi]
mov     eax, 0x40
vcvtps2pd ymm4, xmm1  // Convert 4 fp32 to fp64
vextractf128 xmm1, ymm1, 0x1
// Convert the remaining 4 fp32 to fp64
vcvtps2pd ymm3, xmm1
vaddpd  ymm6, ymm0, ymm4
vaddpd  ymm5, ymm0, ymm3
vaddpd  ymm8, ymm10, ymm4
vaddpd  ymm7, ymm10, ymm3
vaddpd  ymm1, ymm9, ymm4
vaddpd  ymm2, ymm9, ymm3
```

(B) Load operation for `Accessor<fp64,fp32>`

FIGURE 6 CPU assembly codes generated by the compiler when loading with `Accessor<fp64,fp64>` or `Accessor<fp64,fp32>`, including a small part of the computation

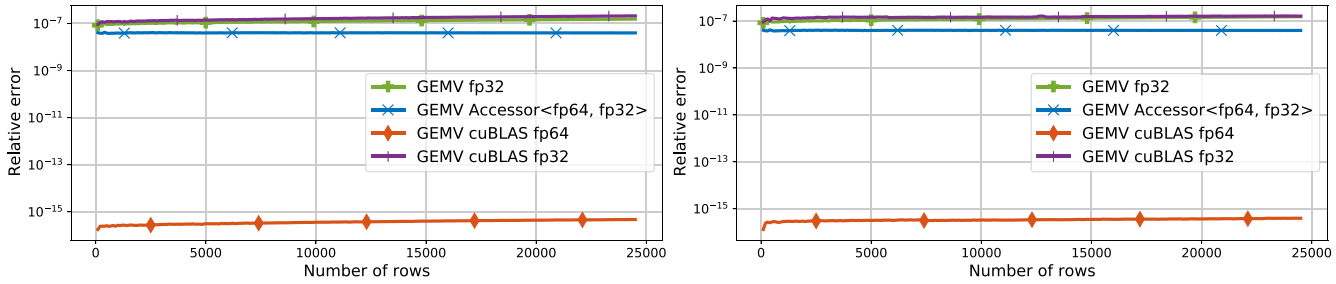

FIGURE 7 Analysis of the average relative rounding error of GEMV kernels on the NVIDIA V100 GPU (left) and the NVIDIA A100 GPU (right)

4 | ACCESSOR-BLAS

After having evaluated the performance of the memory accessor with a synthetic benchmark, we now turn our attention to the practical use of the accessor in memory-bound BLAS operations. The goal is to preserve the performance of low precision (in particular, `fp32`) BLAS, but to increase the accuracy of the routines by performing all arithmetic and in-register accumulations in higher precision (`fp64`). The accuracy improvements can be expected to depend on the input data, and therefore we detail the value distribution for each experiment. Furthermore, as the specific value distribution has a significant impact on the error accumulation, we always report the average error results over 10 executions, with the data for each run being randomly generated using a distinct initial seed. In the experimental evaluation, for brevity, we refrain from considering all hardware architectures covered in Section 2, and instead focus on the two NVIDIA GPUs (V100 and A100), which belong to two distinct hardware generations and differ in some hardware parameters (number of multiprocessors, available registers, etc.). The performance and error characteristics of this analysis can be expected to be representative for other hardware architectures.

4.1 | General matrix-vector product

We start the evaluation with the reimplementation of the general dense matrix-vector product (GEMV).[26] We generate both the input matrix and input vector with random values uniformly distributed in the interval $[-1, 1]$*and we set the scaling factors for this operation to $\alpha = 1.0$ and $\beta = 0.0$. In Figure 7, we compare the relative errors taking the `fp64` matrix-vector kernel as reference. The average relative errors for the fixed precision GEMV implementations are 10^{-7} for `fp32`and 10^{-16} for `fp64`, respectively, and they slowly grow with the matrix size. These error bounds are conformal with the unit round-off of the `fp32` and `fp64` formats. For the accessor-based GEMV, the error is about half an order of magnitude smaller than the error of the `fp32` GEMV. The higher accuracy of the accessor-based GEMV comes from the use of `fp64` in the accumulation of the partial sums in registers. To investigate the performance impact of the use of higher precision in the arithmetic operations, in Figure 8, we visualize the FLOP/s achieved by the different implementations of GEMV on the V100 and A100 GPUs. We notice that, on both architectures, the accessor-based GEMV delivers the same throughput rate as our `fp32` GEMV implementation. This reveals the high quality of the memory accessor implementation and the validity of the approach which adds a value conversion and carries out all arithmetic in `fp64`. While outside the goal of this article, we notice that the accessor-based GEMV implementation outperforms the cuBLAS GEMV on the V100 GPU while, for large matrix sizes, it is competitive with the cuBLAS realization of this kernel on the A100 GPU.

4.2 | DOT product

Acknowledging that the higher accuracy we observe for the accessor-based GEMV comes from handling the accumulation in `fp64`, we can expect the accuracy gains to grow more significantly for longer accumulations. Therefore, we next turn

*We show in the Appendix a more comprehensive accuracy evaluation where we consider a uniform distribution of the values and a normal distribution of the values.

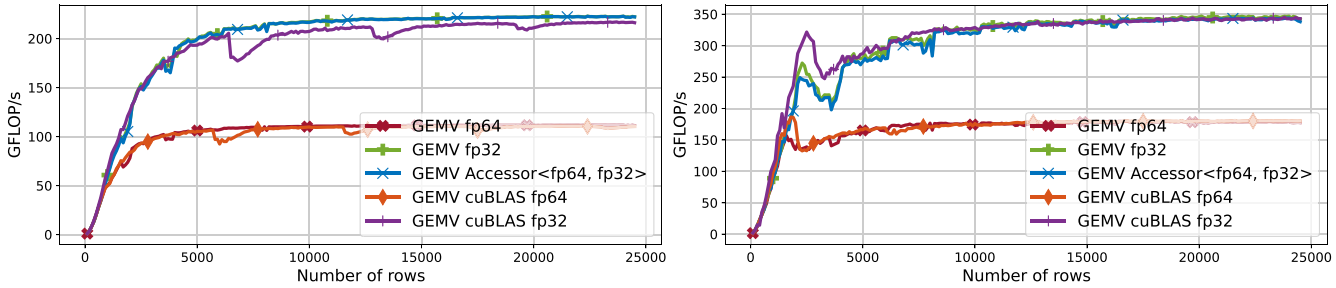

FIGURE 8 Performance analysis of `GEMV` kernels on the NVIDIA V100 GPU (left) and the NVIDIA A100 GPU (right)

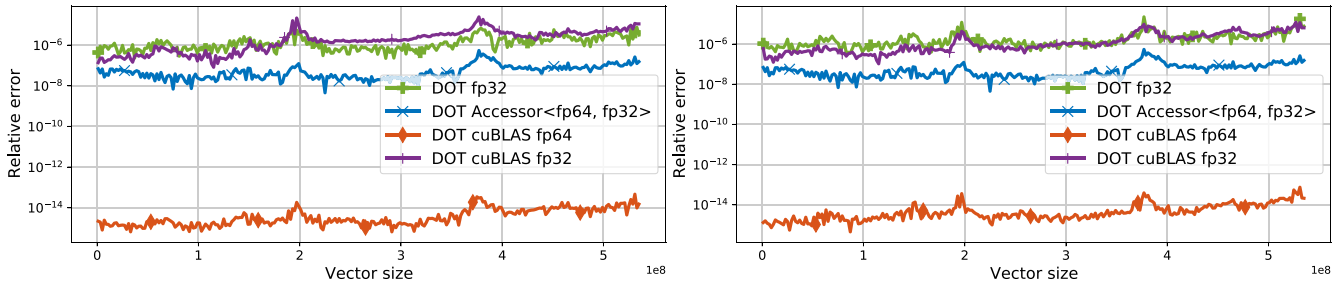

FIGURE 9 Analysis of the average relative rounding error of `DOT` kernels on the NVIDIA V100 GPU (left) and the NVIDIA A100 GPU (right) using uniform random values in the interval $[-1, 1]$

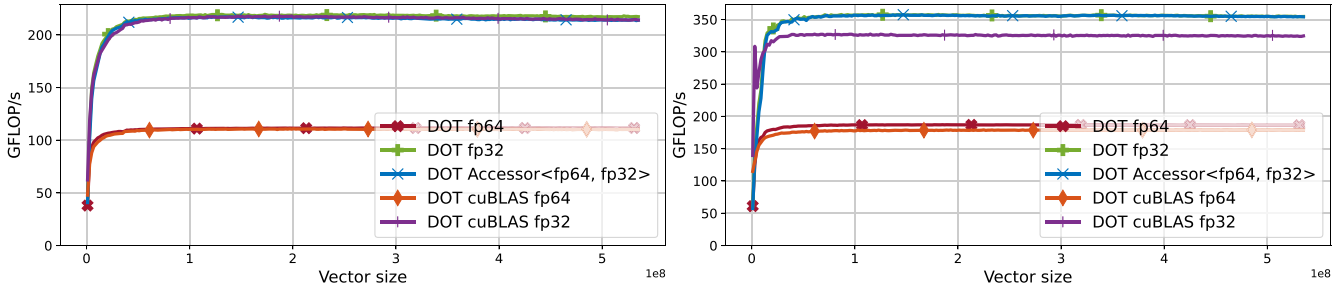

FIGURE 10 Performance analysis of `DOT` kernels on the NVIDIA V100 GPU (left) and the NVIDIA A100 GPU (right)

to the `DOT` kernel from BLAS, computing the dot (or scalar) product of two vectors. In Figure 9, we again use input vectors with random values uniformly distributed in the interval $[-1, 1]$. We again observe a mild accuracy advantage of the accessor-based `DOT` over the `fp32 DOT` kernels. At the same time, the performance of the `DOT` kernel does not suffer from the use of higher precision in the arithmetic operations, achieving almost twice the arithmetic rate of the `DOT` kernel using `fp64`; see Figure 10. We note that the accessor-based `DOT` is slightly slower than our `fp32 DOT` kernel on the A100 GPU; however, it still outperforms the `fp32 DOT` kernel from the cuBLAS library. Thus, using the accessor-based `DOT` can render accuracy improvements while incurring only a small runtime overhead.

4.3 | Dense triangular system solve

Next, we turn our attention to the dense triangular matrix-vector solve (`TRSV`) from BLAS. Accuracy plays a critical role for this kernel as it is the last component in the direct solution of a linear system of equations via, for example, an LU factorization,[27] and any error introduced in the kernel impacts the accuracy of the solution of the original problem.

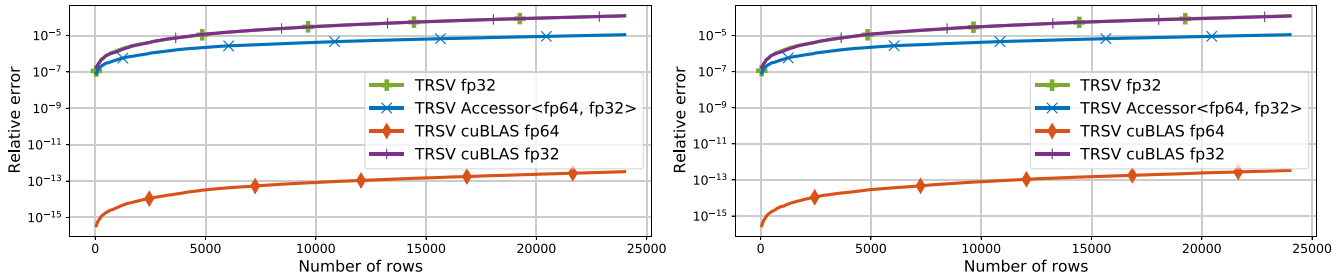

FIGURE 11 Analysis of the average relative rounding error of TRSV kernels on the NVIDIA V100 GPU (left) and the NVIDIA A100 GPU (right)

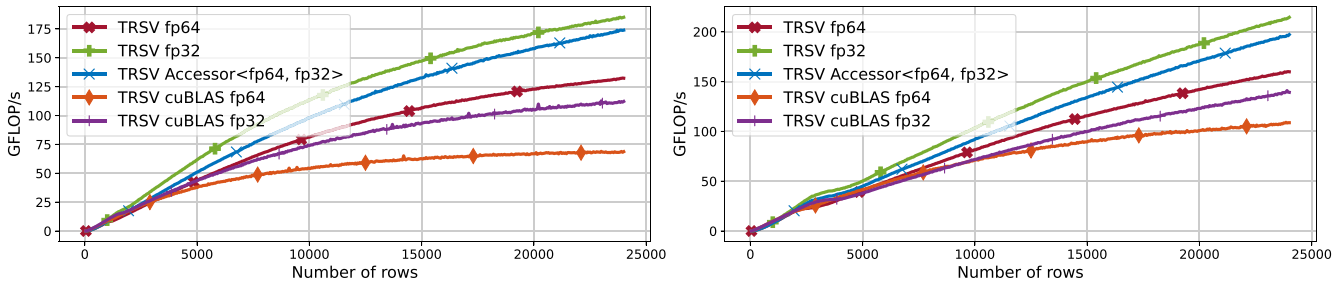

FIGURE 12 Performance analysis of TRSV kernels on the NVIDIA V100 GPU (left) and the NVIDIA A100 GPU (right)

Linear system solvers often employ `fp64`routines, but an iterative refinement method that operates in lower precision for both the factorization and the triangular solves has shown to be an appealing alternative that can, in many cases, accelerate the solution process.[28,29] Thus, any scheme that improves the accuracy of the `fp32 TRSV` kernel while maintaining its performance is highly appreciated by the community. The `TRSV` kernel we implement largely follows the ideas presented in Reference 30. For the error analysis, we generate a matrix with random entries uniformly distributed in the interval $[-1, 1]$. We then factorize the matrix using the `fp64` LU factorization routine from NVIDIA's cuSOLVER library. In Figure 11, we visualize the average error bounds that the distinct `TRSV` kernels achieve for this problem. The accessor-based `TRSV` kernel delivers about an order of magnitude higher accuracy than the `fp32TRSV` kernel. Unfortunately, Figure 12 reveals that this accuracy gain does not come for free in this case: on the NVIDIA V100 GPU (left-hand side in Figure 12), the accessor-based `TRSV` kernel achieves about 10 GFLOP/s lower performance than our `fp32 TRSV` kernel; however it still outperforms the cuBLAS `TRSV` and our `fp64 TRSV` kernel. This can be explained by the design of our `TRSV` kernel based on Reference 30. Concretely, the kernel inverts the panel (diagonal block) of the LU factor, using shared memory and registers for this step. Performing this inversion in `fp64` arithmetic doubles the requirements for registers and shared memory, therewith reducing the GPU occupancy. The kernel still outperforms the `fp64` variant due to the faster memory access. We identify a similar effect on the NVIDIA A100 GPU: the performance of the accessor-based `TRSV` is about 20 GFLOPs/s lower than that of the `fp32 TRSV`. On the A100 GPU, the register-per-core ratio is lower than on the V100 GPU. Thus, the performance penalty is more significant when compared with the `fp64 TRSV`. This reveals that, for BLAS kernels which are not entirely bandwidth-bound but also put pressure on other resources, the re-engineered BLAS kernels using the memory accessor can mildly suffer from the increased register usage. Nonetheless, we still augment the accuracy of the `TRSV` kernel while outperforming the `fp64 TRSV` kernels.

4.4 | Sparse matrix-vector product

We now look back at the matrix-vector multiplication, but shift the focus from dense to sparse matrices. The sparse matrix-vector product (`SpMV`) kernel is an important component in scientific computing as it reflects how a discretized linear operator acts on a vector, and therewith plays the central role in the iterative solution of linear systems and

eigenvalue problems.[16] Popular methods based on the iterative application of the SpMV kernel include Krylov subspace solvers such as conjugate gradient (CG), generalized minimal residual (GMRES), or biconjugate gradient stabilized method (BiCGSTAB),[31] as well as the PageRank algorithm based on the Power Iteration.[32] The SpMV kernel is also a key routine in graph analytics as it can be used there to identify all immediate neighbors of a node or a set of nodes.

To evaluate the accuracy and performance trends of an accessor-based SpMV kernel, we choose the ELL format,[33] which explicitly stores the same number of nonzero elements for all rows of the sparse matrix, and accompanies the explicitly-stored nonzero values with column indices. The motivation for integrating and evaluating the memory accessor in an ELL -based SpMV kernel is that ELL has been compared to other sparse matrix formats, such as CSR or COO , with a smaller indexing overhead.[34] Thus, the selection of ELL highlights the differences induced by the accessor. The baseline ELL kernel we leverage for the design of the reference and accessor-based SpMV variants in the following evaluation is that available in the GINKGO library.[35] For the evaluation of the accessor-based SpMV kernel, we do not rely on artificial test matrices, but select benchmark matrices from the Suite Sparse Matrix Collection.

Even though we decide to focus on the ELL SpMV kernel to minimize the effect of the indexing, we acknowledge that the performance differences between fp32 SpMV and fp64 SpMV are much smaller than those observed for the analogous dense BLAS operation. For clarity, in the performance evaluation in Figure 13, we thus do not report the FLOP/s rates, but the speedup over the fp64 SpMV. We recognize that the accessor-based SpMV generally achieves marginally-lower speedups over the fp32 SpMV. This indicates that the accessor does not succeed in completely hiding the cost of the format conversion behind the memory access for irregular memory accesses. The reason is likely the doubling of the shared memory requirements for the accessor ELL kernel. However, as the performance differences are negligible for all test cases (lower than 3% on both GPU architectures), we consider this degradation acceptable. In Figure 14, we visualize the relative error of the different ELL SpMV kernels. As expected, both GPU architectures provide the same results (despite differences in the architecture and the execution). Finally, the accuracy improvements rendered by the accessor-SpMV over the fp32 SpMV are very problem-dependent. Although all these sparse matrices contain only

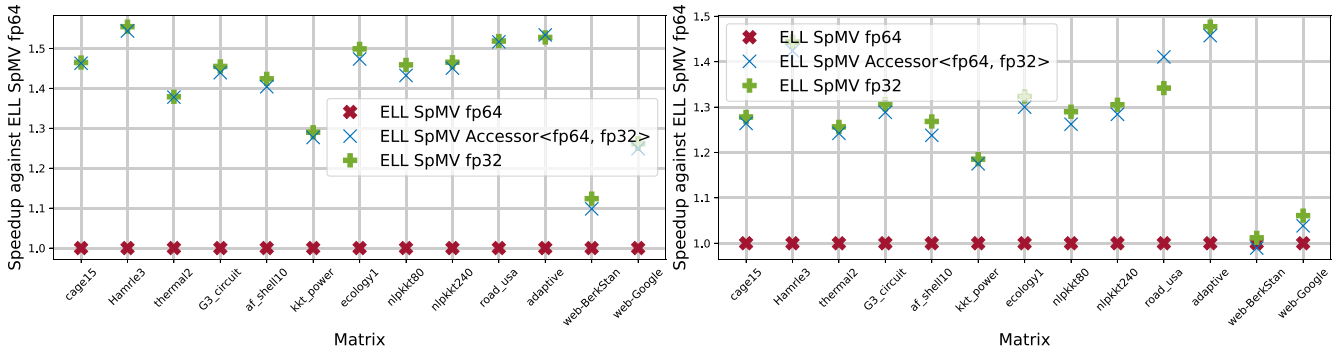

FIGURE 13 Performance analysis of the ELL SpMV kernels on the NVIDIA V100 GPU (left) and the NVIDIA A100 GPU (right). The baseline performance is that of fp64 ELL SpMV . The test matrices are taken from the Suite Sparse Matrix Collection

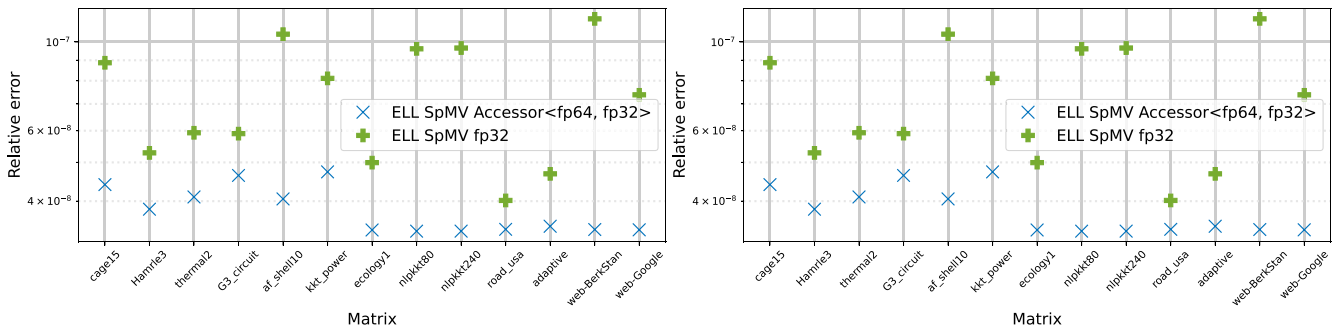

FIGURE 14 Analysis of the average relative rounding error of ELL SpMV kernels on the NVIDIA V100 GPU (left) and the NVIDIA A100 GPU (right). The test matrices are taken from the Suite Sparse Matrix Collection

a few nonzero elements per row, which are then accumulated in registers and shared memory, the accessor-`SpMV` can increase the accuracy over the `fp32 SpMV` by up to an order of magnitude. To close this section, we emphasize that the test matrices in this last experiment are derived from real-world problems, which demonstrates the practical benefits that the accessor-based `SpMV` renders over the standard `SpMV`kernels.

5 | SUMMARY AND FUTURE WORK

We have presented a realization for a memory accessor that combines single precision memory access with on-the-fly conversion to double-precision in registers to enable double-precision arithmetic without degrading the performance for memory-bound kernels. We have demonstrated that implementations of this memory accessor realization for GPUs from AMD and NVIDIA, as well as a CPU from Intel, succeed in hiding the value conversion behind the memory access. We have then re-engineered memory-bound BLAS kernels by using the memory accessor to perform all arithmetic in double precision while using single precision for all memory operations. The accessor-based BLAS routines provide higher accuracy than the single precision BLAS routines while incurring at most mild runtime overhead for routines where the registers are a scarce resource. Future work will look into how the integration of the accessor-based BLAS kernels can improve the time-to-accuracy performance of mixed precision algorithms such as mixed precision iterative refinement using low precision BLAS as part of an iterative process.

ACKNOWLEDGMENTS

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. This work was performed on the HoreKa supercomputer funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research. The authors acknowledge support by the state of Baden-Württemberg through bwHPC.

AUTHOR CONTRIBUTION

All authors contributed to this paper equally.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available from the corresponding author upon reasonable request.

ORCID

Thomas Grützmacher https://orcid.org/0000-0001-9346-2981

REFERENCES

1. International Technology Roadmap for semiconductors 2.0; 2015. http://www.itrs2.net/itrs-reports.html
2. Wulf WA, McKee SA. Hitting the memory wall: implications of the obvious. *SIGARCH Comput Archit News*. 1995;23(1):20-24. doi:10.1145/216585.216588
3. McKee SA, Wisniewski RW. *Memory Wall*. Springer; 2011:1110-1116.
4. McCalpin JD. Memory bandwidth and machine balance in current high performance computers. *IEEE Comput Soc Techn Committee Computr Arch Newslett*. 1995;2:19-25.
5. Dongarra J, Beckman P, Moore T, et al. The international ExaScale software project roadmap. *Int J High Perform Comput Appl*. 2011;25(1):3–60.
6. Williams S, Waterman A, Patterson D. Roofline: an insightful visual performance model for multicore architectures. *Commun ACM*. 2009;52(4):65-76. doi:10.1145/1498765.1498785
7. Ofenbeck G, Steinmann R, Caparros V, Spampinato DG, Püschel M. Applying the roofline model. Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS); 2014:76-85; IEEE.
8. Ilic A, Pratas F, Sousa L. Cache-aware roofline model: upgrading the loft. *IEEE Comput Archit Lett*. 2014;13(1):21-24. doi:10.1109/L-CA.2013.6
9. Cabezas VC, Püschel M. Extending the roofline model: bottleneck analysis with microarchitectural constraints. Proceedings of the IEEE International Symposium on Workload Characterization (IISWC); 2014:222-231; IEEE
10. Diamond J, Burtscher M, McCalpin JD, Kim B, Keckler SW, Browne JC. Evaluation and optimization of multicore performance bottlenecks in supercomputing applications. Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS); 2011:32-43; IEEE.

11. Anzt H, Flegar G, Grützmacher T, Quintana-Ortí ES. Toward a modular precision ecosystem for high-performance computing. *The International Journal of High Performance Computing Applications*. 2019;33:(6):1069–1078. https://doi.org/10.1177/1094342019846547
12. Lindstrom P. Fixed-Rate Compressed Floating-Point Arrays. *IEEE Transactions on Visualization and Computer Graphics*. 2014;20:(12):2674–2683. https://doi.org/10.1109/tvcg.2014.2346458
13. Di S, Cappello F. Fast error-bounded lossy HPC data compression with SZ. Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016; May 23-27, 2016:730-739; IEEE Computer Society, Chicago, IL.
14. Tao D, Di S, Liang X, Chen Z, Cappello F. Optimizing lossy compression rate-distortion from automatic online selection between SZ and ZFP; 2019.
15. Venkataramanan R, Sarkar T, Tatikonda S. Lossy Compression via Sparse Linear Regression: Computationally Efficient Encoding and Decoding. *IEEE Transactions on Information Theory*. 2014;60:(6):3265–3278. https://doi.org/10.1109/tit.2014.2314676
16. Saad Y. *Iterative Methods for Sparse Linear Systems*. 2nd ed. SIAM; 2003.
17. Williams S, Oliker L, Vuduc R, Shalf J, Yelick K, Demmel J. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. Proceedings of the 2007 ACM/IEEE Conference on Supercomputing SC '07; 2007:1-12; IEEE.
18. Higham NJ. *Accuracy and Stability of Numerical Algorithms*. 2nd ed. Society for Industrial and Applied Mathematics; 2002.
19. Tao D, Di S, Chen Z, Cappello F. Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization. Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, May 29 - June 2, 2017:1129-1139; IEEE Computer Society, Orlando, FL.
20. Lindstrom P, Isenburg M. Fast and efficient compression of floating-point data. *IEEE Trans Vis Comput Graph*. 2006;12(5):1245-1250. doi:10.1109/TVCG.2006.143
21. Calhoun J, Cappello F, Olson LN, Snir M, Gropp WD. Exploring the feasibility of lossy compression for PDE simulations. *Int J High Perform Comput Appl*. 2019;33(2):397-410. doi:10.1177/1094342018762036
22. Agullo E, Cappello F, Di S, Giraud L, Liang X, Schenkels N. Exploring variable accuracy storage through lossy compression techniques in numerical linear algebra: a first application to flexible GMRES. Research report RR-9342, Inria Bordeaux Sud-Ouest; 2020.
23. Markidis S, Chien SWD, Laure E, Peng IB, Vetter JS. NVIDIA tensor core programmability, performance & precision. *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2018;522–531. https://doi.org/10.1109/IPDPSW.2018.00091
24. AMD. AMD CDNA architecture; 2020.
25. Anzt H, Cojean T, Flegar G, et al. Ginkgo: a modern linear operator algebra framework for high performance computing; 2020.
26. Blackford LS. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans Math Softw*. 2002;28(2):135-151. doi:10.1145/567806.567807
27. Golub GH, Van Loan CF. *Matrix Computations*. 3rd ed. The Johns Hopkins University Press; 1996.
28. Kurzak J, Dongarra J. Implementation of mixed precision in solving systems of linear equations on the cell processor. *Concurr Comput Pract Exper*. 2007;19(10):1371-1385.
29. Abdelfattah A, Anzt H, Dongarra J, et al. Linear algebra software for large-scale accelerated multicore computing. *Acta Numer*. 2016;25:1-160. doi:10.1017/S0962492916000015
30. Hogg JD. A fast dense triangular solve in CUDA. *SIAM J Sci Comput*. 2013;35(3):C303-C322. doi:10.1137/12088358X
31. Saad Y, Schultz MH. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J Sci Stat Comput*. 1986;7:856-869. doi:10.1137/0907058
32. Grützmacher T, Cojean T, Flegar G, Anzt H, Quintana-Ortí ES. Acceleration of PageRank with customized precision based on mantissa segmentation. *ACM Trans Parallel Comput*. 2020;7(1). doi:10.1145/3380934
33. Bell N, Garland M. Efficient sparse matrix-vector multiplication on CUDA; 2008.
34. Anzt H, Cojean T, Yen-Chen C, et al. Load-balancing sparse matrix vector product kernels on GPUs. *ACM Trans Parallel Comput*. 2020;7(1). doi:10.1145/3380930
35. Anzt H, Cojean T, Chen YC, et al. Ginkgo: a high performance numerical linear algebra library. *J Open Source Softw*. 2020. doi:10.21105/joss.02260

How to cite this article: Grützmacher T, Anzt H, Quintana-Ortí ES. Using Ginkgo's memory accessor for improving the accuracy of memory-bound low precision BLAS. *Softw Pract Exper*. 2021;1-18. doi: 10.1002/spe.3041

APPENDIX

For comprehensiveness, in this appendix, we report accuracy results obtained when using a uniform a normal distribution a mean of zero and a standard deviation of one. All datapoints presented here are obtained as median over running 10 experiments the respective routines on an NVIDIA V100 GPU. The left-hand side figures replicate the results for the uniform distribution in the interval $[-1, 1]$ while the right-hand side results present a normal distribution.

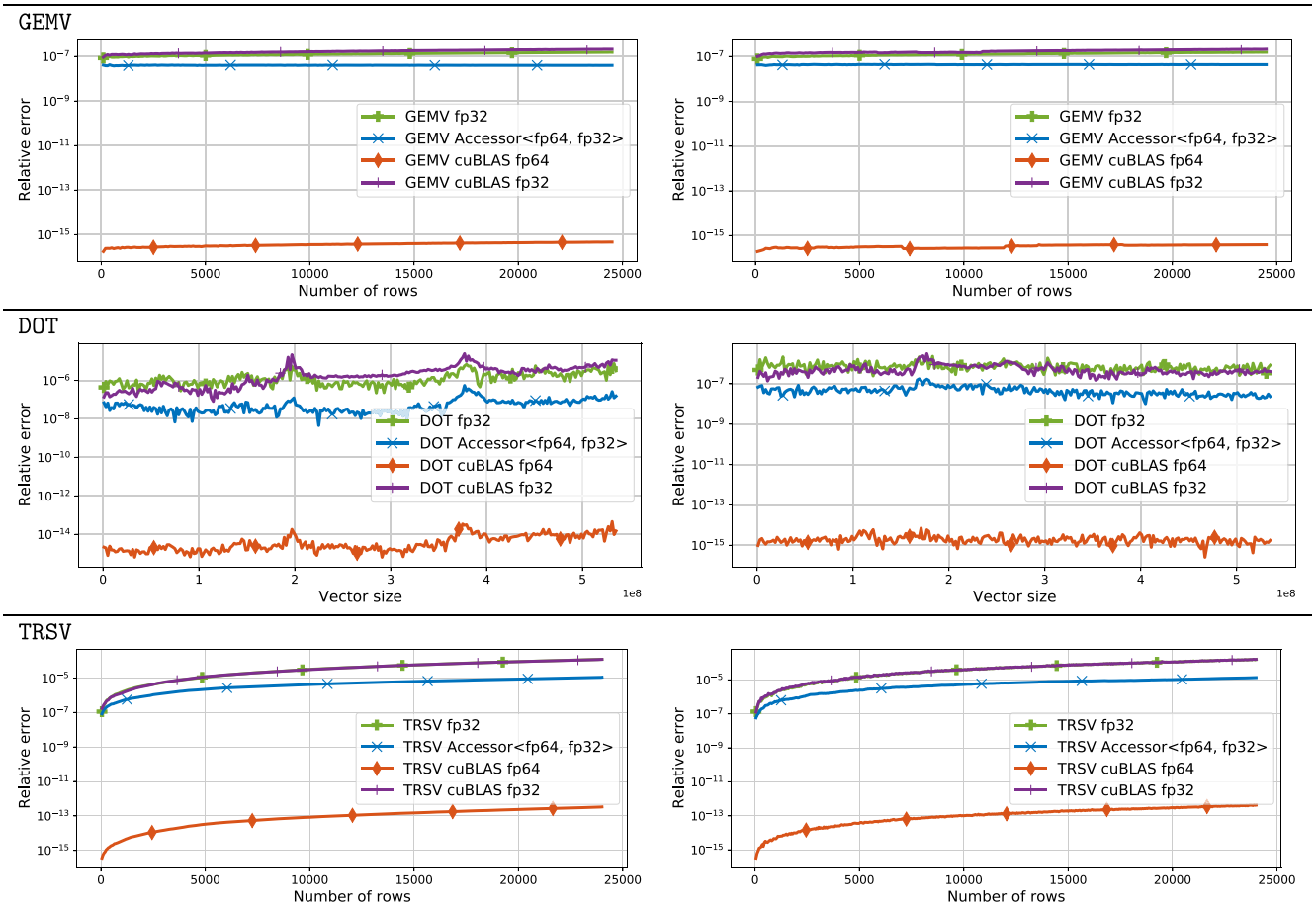

FIGURE A1 Analysis of the average relative rounding error of different kernels on the NVIDIA V100 GPU using either a uniform distribution for the input values (left) or a normal distribution of the input values (right)