LLNL-SR-820918

LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

Technical Report: Accessor Realization

H. Anzt, T. Cojean, T. Gruetzmacher

March 26, 2021

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Technical report: Accessor Realization

Hartwig Anzt, Terry Cojean, Thomas Grützmacher

March 26, 2021

1. Introduction

As we have seen the arithmetic power of hardware architectures growing much faster than the memory bandwidth [1], there is an urgent need to rethink the way we operate, communicate, and store data. A promising strategy for overcoming the "memory wall" is the decoupling of memory precision from arithmetic precision as proposed in Anzt et al. [2]. Such a decoupling of memory precision and arithmetic precision requires a "memory accessor" that handles the conversion between formats (resp. compression of the data) on the fly in registers. For such a memory accessor to be useful in practice, it has to combine multiple characteristics. Most important, it should be

- general in the sense of being able to handle a wide range of floating point formats and compression strategies;
- extensible in the sense of adding new precision formats in the future;
- opaque in the sense of abstracting the implementation details from the user;
- portable across different hardware architectures;
- efficient in the sense of not introducing runtime overhead but hiding all conversion cost behind the memory access.

In this document we describe the technical realization of the memory accessor we deploy in the Ginkgo open source library [3]. We note that the realization tries to reduce the accessor dependency on other Ginkgo components, therewith allowing for easy integration and utilization in other software efforts. The rest of the document is structured as follows. We first provide some background motivating the development of a memory accessor in Section 2. We then detail in Section 3 the implementation of the memory accessor making heavy use of modern C++ features. In Section 4 we provide a comprehensive analysis on the performance characteristics and overhead of the memory accessor implementation on AMD GPUs (Section 4.1), NVIDIA GPUs (Section 4.2), and OpenMP-supporting CPUs (Section 4.3). In Section 5 we present details on how to integrate the memory accessor into two algorithms: in Section 5.1 the Compressed Basis Generalized Minimal Residual (CB-GMRES) algorithm that uses the memory accessor to store the Krylov basis vectors in compressed fashion, and in Section 5.2 how the accessor can be used to compress the values in a mixed precision sparse approximate inverse preconditioner.

2. Background

For decades, the compute performance in digital processors has been growing faster than the main memory bandwidth [2]. The consequence is an increasing number of applications becoming memory bound, that is, the applications not being able to leverage the full compute potential of the hardware as they are stalled and waiting for data. While the redesign of the algorithms to expose higher arithmetic intensity and make better use of the cache hierarchies is a promising strategy to prepare applications for future hardware, this concept may not be applicable to all applications. In particular, a good fraction of the scientific computing applications operate on sparse data structures, making it extremely difficult or even impossible to transform the algorithms into BLAS-3 operations. At the other end of the simulation pipeline, scientists are well-aware that writing the application results to permanent storage (disk or tape) can be the bottleneck in the workflow. In response, applications writing significant data volumes to disk have early on started to implement lossy and lossless compression techniques. On a higher level, the compression and decompression of datasets can be seen as an accessor acting on the disc level. Following a similar strategy, we design an accessor that acts on the main memory level. However, there exist significant differences in the requirements and design between the compression strategies for data compression and the accessor operating on main memory that become obvious when reviewing the traditional compression strategies. Lossless compression is based on the idea of detecting smart strategies to store the data in a more compact fashion, see, e.g., [4, 5]. Generally, lossless compression is rarely able to achieve compression ratios exceeding 1.5×, which implies only limited throughput increase. Lossy floating-point compression has shown to achieve much higher compression ratios, but its efficiency heavily depends on the input data [6].

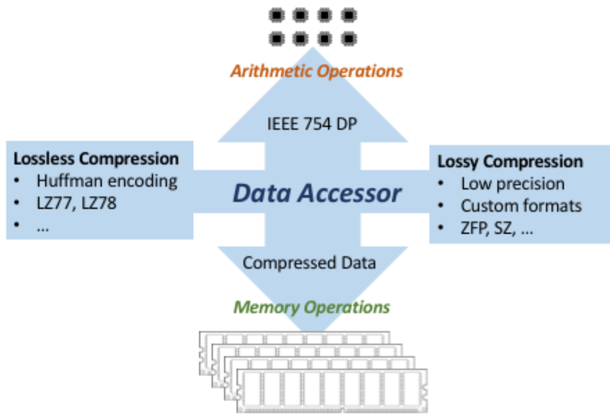

Figure 1: Memory accessor deployed in the GINKGO library. Currently only lossy compression in terms of precision format casting is supported.

Usually, both lossy and lossless compression techniques are based on compressing moderate-sized chunks of data by detecting patterns in the data, approximating the data with functions, or applying sophisticated schemes based on Fast Fourier Transforms (FFT). Therefore, these techniques need data arrays of floating point numbers that exhibit spatial correlation to function properly and should not be used for unstructured data or unrelated numbers. In general, the larger the array of related data is, the more efficient is the compression without losing too much information. Conversely, it is much harder to realize a compression of small data chunks which is necessary for fast random accesses. Hence, most of the lossy and lossless data compression schemes are not optimized for random accesses. However, zfp is one example that is optimized for both random read and write performance with the help of a software cache [6]. zfp is spearheading also in terms of user friendliness as it provides an array interface which mimics STL vectors for easier adoption and use.

The memory accessor operating on the main memory level has to compress the data before invoking main memory operations, it has to operate on data that is located in processor registers. The limited number of processor registers makes it very difficult to use complex compression algorithms that operate on chunks of data and use features of the data set for a more compact data representation. In the extreme case, the memory accessor may compress each data element individually like suggested in [2]. There, the data is compressed by casting the independent values to a lower precision format. In consequence, the memory accessor can not only be interpreted as a customized version of a data compression algorithm acting on main memory, but also as a mixed precision algorithm converting values from the arithmetic precision format into a lower precision format for the memory operations. As this requires careful consideration of the impact on the application using the memory accessor, the accessor we realize in the GINKGO library is designed to not only support the use of a variety of precision formats for the memory operations, but also accept user input to select between these formats depending on the application's requirements. A simplified operating scheme of the memory accessor we realize in the GINKGO library is given in Figure 1.

3. Implementation

The accessor implementation is part of the open source library GINKGO[1], which is written in C++14. We made sure that there are no dependencies to other parts of GINKGO, allowing other software efforts to make use of the accessor by treating it as a header-only library without building or installing GINKGO itself. The

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

accessor itself is implemented thanks to static (compile-time) polymorphism, which allows for little to no overhead and full compiler optimizations.

The implementation consists of two to three layers, which we present in the following sections: (1) The range, acting as the interface for every accessor; (2) The accessor itself, which is responsible for translating the index information into read/write access; (3) (optional) A reference class, which is necessary if the accessor needs to perform computations to translate between memory and arithmetic format.

3.1 THE 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 implementation of the `range` is shown in Listing 1. In order to interface with the accessor, it takes an accessor type as a template parameter. At its core is the parenthesis operator, which is used for both read and write operations. A read from a two-dimensional range `mat` looks like `val = mat(2,3)`, while a write operation to the same location is done with `mat(2,3) =` 42. The implementation of the `range` is shown in Listing 1. We first look at the parenthesis operator defined in Line 19. It uses the C++ feature of perfect forwarding, which takes an arbitrary number of arguments, each of any type, and calls the accessor's overload of the parenthesis operator with the exact same arguments without modifying them. The return 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 have a parenthesis operator. Using these sort of techniques we are able to set interface requirements for the accessor, such that the `range` is able to use any type of accessor.

Additionally, all accessors need to define a `length` member function (according to Line 32), returning the length of the given dimension in order to have access to the size the accessor covers.

At last, the `range` also allows to directly interact with the underlying accessor in Line 36 and Line 42 to access functionality that may be specific to the accessor, for example, changing a scalar value or modifying the compression strategy. Direct access is important because having every possible functionality in one 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
```

```
    // `Returns a reference to the accessor.
    GKO_ACC_ATTRIBUTES constexpr const accessor &get_accessor() const noexcept
    {
        return accessor_;
    }

private:
    accessor accessor_;
};
```

Listing 1: Full implementation of the range

3.2 THE ACCESSOR: TRANSLATES INDEX INFORMATION INTO DATA ACCESS

A simple two-dimensional column-major accessor is shown in Listing 2 which fulfills all requirements from `range`. It uses as template parameter the `ValueType`, the type of the stored data pointer. We define the `range` class as a friend 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 do for all currently implemented accessors). These improve the debugging capabilities of the code and do not impact performance since the checks are only performed in debug mode. In release mode, the checks are not present.

```
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 are able to 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 Listing 2 with an additional template parameter specifying the dimensionality.

2. block_col_major: The lowest two dimensions are stored column-major, while all other dimensions are stored in row major. It is used to handle two-dimensional blocks stored in column-major like 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 a separate memory and arithmetic precision and performs a conversion between the two for every access. This accessor is used in CB-GMRES and SAI for IEEE floating point storage precisions.

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. This accessor is used in CB-GMRES to allow for integer memory precisions.

3.3 CUSTOM REFERENCES: AN ADDITIONAL LAYER FOR COMPLEX ACCESSORS

If the accessor needs to perform 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 first before returning the value / writing to memory.

The reference class used in reduced_row_major is shown exemplary in Listing 3. In Line 21, we define a cast operation to our arithmetic type where we incorporate the computations we need to perform when reading in memory precision. Here, it is the cast from memory to arithmetic precision, but it could also be a complex de-compression algorithm. After the computation is done, the resulting value is returned. This reference object can now be treated like a variable of type arithmetic_type thanks to the C++ implicit conversions feature. Whenever the reference object would not be accepted in an operation, but a arithmetic_type value would be, the compiler calls the conversion function implicitly.

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

The shown reference class reduced_storage accepts any type that has a static_cast defined between ArithmeticType and StorageType. Part of that are the IEEE types fp64, fp32 and fp16 (at least on GPUs), which have hardware support for these conversions. However, it is also possible to use software types like the __float128 from GCC. Software types would achieve subpar performance, but could be used to experiment with various precision formats without changing the algorithm implementation.

```
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) {
```

```
        return *this = static_cast<arithmetic_type>(ref);
    }

    constexpr GKO_ACC_ATTRIBUTES arithmetic_type operator=(reduced_storage &&ref) {
        return *this = static_cast<arithmetic_type>(ref);
    }

private:
    storage_type *const ptr_;
};
```

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

4. Evaluation

To assess the performance overhead of the accessor, we implement a benchmark able to achieve the peak memory bandwidth and the peak compute performance by successively increasing the arithmetic intensity of the benchmark algorithm. We develop two implementations of this benchmark: one using direct access to the values in main memory, one going through the memory accessor to retrieve values from main memory. We choose `reduced_row_major` as the representative of the accessors since it is one of the most complex with its custom reference class which performs the conversion between memory and arithmetic precision on-the-fly. The only difference between the two implementations is the read and write accesses: the access patterns, the order and kind of computations are identical. We emphasize that the benchmark using the memory accessor does not account for the FLOPs for the additional computations necessary for the on-the-fly conversion between the precision formats. The implementations are designed to generate an experimental roofline model, achieving the full bandwidth for memory bound computations and the compute peak for compute bound computations. Running these two benchmarks and comparing their compute performance for a certain arithmetic intensity allows us to identify the performance penalty of the type conversion (if compressing data) and the overhead of the memory accessor (for all memory accessor use).

In the experimental evaluation, we run four precision configurations using either the standard benchmark algorithm or the benchmark algorithm employing the accessor for the memory access.

1. `fp64` arithmetic using `fp64` memory access using the standard benchmark. This configuration reflects standard `fp64` operations.

2. `fp64` arithmetic using `fp64` memory access using the accessor-based benchmark. This `Accessor<fp64,fp64>` is conceptually identical to standard `fp64` computations, but uses the accessor for the main memory access. Technically, the accessor is converting between `fp64` and `fp64` precision formats.

3. `fp32` arithmetic using `fp32` memory access. This configuration reflects the standard `fp32` operations. We assess this configuration using standard memory access.

4. `fp64` arithmetic using `fp32` memory access. This scenario can only be realized using the accessor in the `Accessor<fp64,fp32>` configuration, reading `fp32` values and performing `fp64` arithmetic.

Comparing the performance between the test configurations (1) and (2) allows us to quantify the overhead of the memory accessor. Ideally, both benchmarks achieve the same compute performance for all arithmetic intensities. Test (3) provides a reference roofline for `fp32` computations. The roofline exhibited by configuration (4) reveals both the conceptual advantages of the memory accessor strategy and the overhead of the type conversion.

To assess the performance of the memory accessor on different hardware architectures, we have to implement the benchmarking concept in different programming ecosystems. For now, we deploy the benchmark pair (via direct access and via the memory accessor) in the CUDA language for NVIDIA GPUs, the HIP language for AMD GPUs, and as multithreaded OpenMP C++ code for general purpose CPUs. In the following sections, we present the experimental rooflines for the four test configurations on different hardware architectures.

4.1 AMD GPUS

To assess the performance characteristics of the memory accessor on AMD GPUs, we use the HIP version of the benchmarks. We compile using ROCm version 4.0.

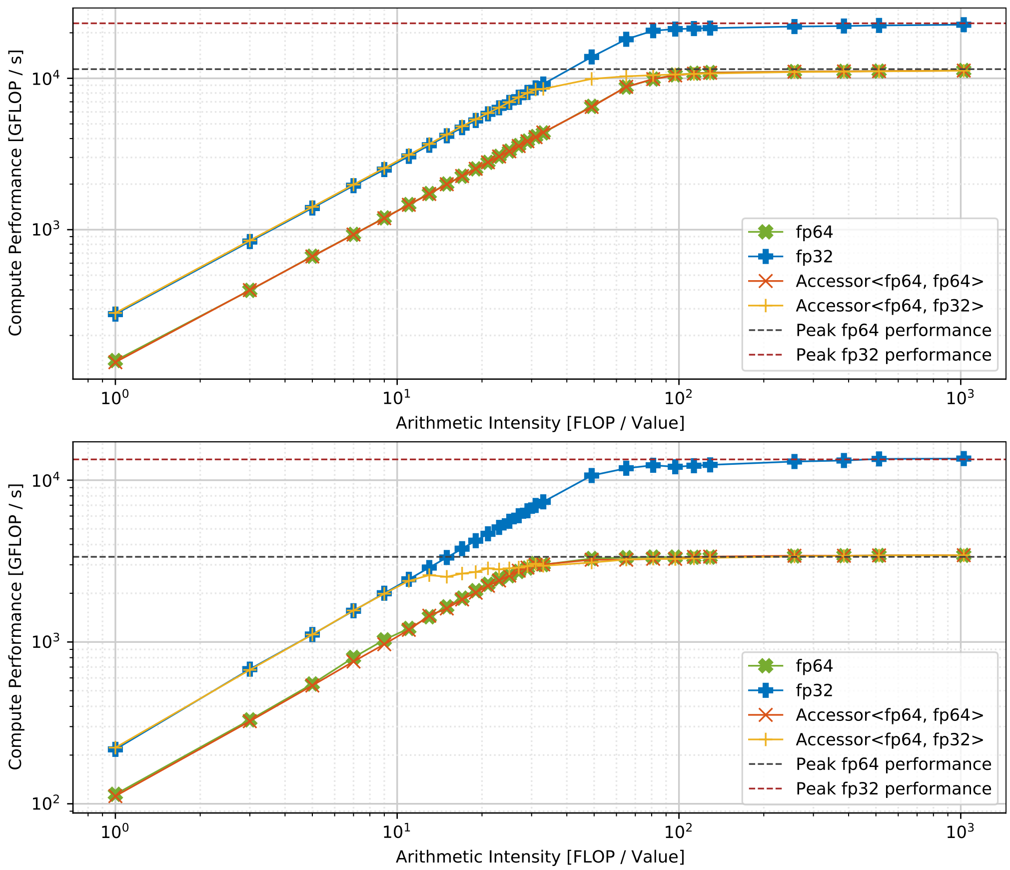

Figure 2: Experimental roofline performance of the memory accessor on an AMD MI100 GPU (top) and an AMD Radeon VII GPU (bottom).

In Figure 2 we visualize the experimental roofline performance relating the compute performance to the arithmetic intensity in the [FLOP/value] metric. In the top of Figure 2 we show the results for the MI100 server line GPU[2], in the bottom we show the results for the Radeon VII consumer line GPU. With dashed lines we indicate the compute peak for `fp32` (single precision) arithmetic and `fp64` (double precision) arithmetic, respectively, that are listed in the technical specifications. We notice that the benchmark using standard memory access reaches the compute peak for high arithmetic intensity. In the memory-bound region, the performance closely reflects the main memory bandwidth. We note that the performance of the `Accessor<fp64,fp64>` is almost indistinguishable from the `fp64` performance. That indicates the high quality of the accessor implementation in terms of incurring only negligible overhead. As expected, for a fixed arithmetic intensity smaller than the machine balance, the `fp32` configuration achieves higher performance than the `fp64` configuration. The key aspect is that in this region, also the `Accessor<fp64,fp32>` benchmark achieves higher performance than the `fp64` configuration. This is exclusively because of the accessor retrieving `fp32` values from main memory, while still using `fp64` in the arithmetic operations. The performance matching the `fp32` performance reveals that the conversion between the formats can efficiently be hidden behind the memory access. Once the arithmetic intensity equals the machine balance, the roofline performance of the `Accessor<fp64,fp32>` asymptotically approaches the compute performance of the `fp64` configuration. This is expected as the arithmetic operations in `fp64` limit the performance of the

[2]The AMD MI100 GPU is part of the Tulip ECP early access system.

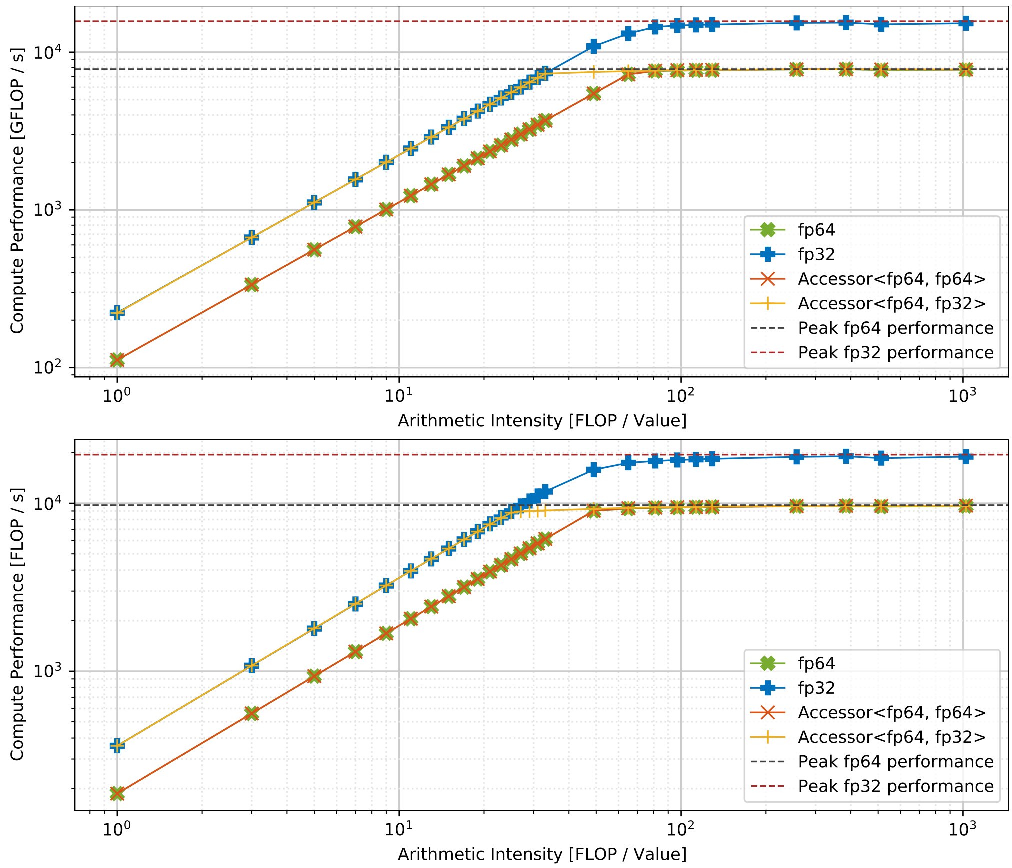

Figure 3: Experimental roofline performance of the memory accessor on an NVIDIA V100 GPU (top) and an NVIDIA A100 GPU (bottom).

`Accessor<fp64,fp32>` benchmark for compute-bound algorithms.

4.2 NVIDIA GPUS

To assess the performance characteristics of the memory accessor on NVIDIA GPUs, we use the CUDA version of the benchmarks. We compile using CUDA version 11.0.

In Figure 3 we show the experimental roofline performance we observe for benchmarks on the NVIDIA V100 GPU (top) and NVIDIA A100 GPU (bottom), both belonging to the server line from NVIDIA. We also indicate the compute peak for `fp64` and `fp32` that are listed in the specifications. Consistent with the AMD results, the standard fixed precision benchmarks `fp32` and `fp64` follow the memory bandwidth for arithmetic intensities smaller than the machine balance, and reach the single precision peak and double precision peak, respectively, for arithmetic intensities larger than the machine balance. The `Accessor<fp64,fp32>` performance again matches the `fp32` performance in the memory bound region, and is limited by the `fp64` peak for arithmetic intensities larger than the machine balance. This reveals that the memory accessor incurs virtually no overhead, and succeeds in hiding all data conversion behind the memory accesses.

4.3 CPUS

Implementing the benchmarks for CPUs is significantly more challenging as it has to reflect all aspects of sophisticated CPU execution: multithreading, vectorization on each core, compiler optimizations. We compile the benchmarks using g++ 10.2.0. By investigating the assembly code of the benchmark implementations for the CPU, we validate that the benchmark using direct memory access and the benchmark accessing the data via the accessor converge to the same assembly code when using the same precision format for

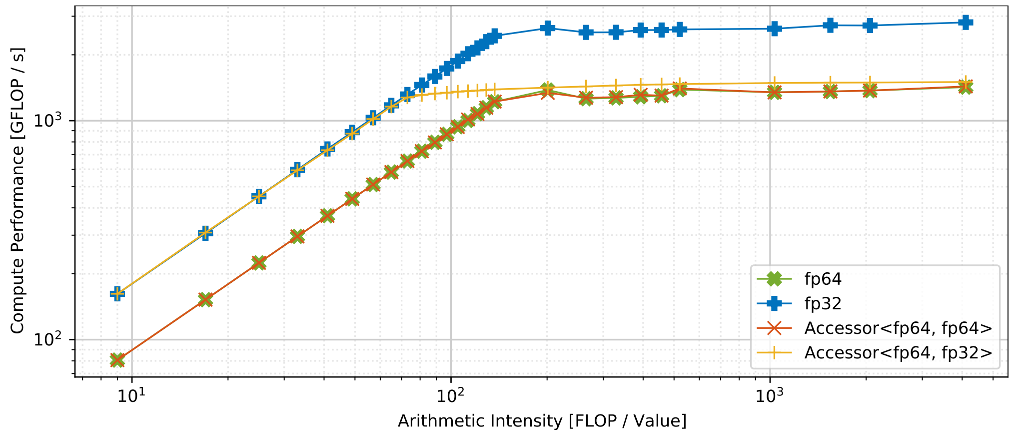

Figure 4: Experimental roofline performance of the memory accessor on an Intel Xeon Gold 6230 Processor (codename Cascade Lake).

```
// 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 5: Comparing generated CPU assembly between loading with Accessor<fp64,fp64> and Accessor<fp64,fp32>, including a small part of the computation.

memory access and arithmetic operations. We note that the assembly code is not completely identical. The function call is slightly different as one uses a structure (the accessor itself containing the size, data pointer and strides), the other one a plain data pointer (since the stride value was inlined). The load, store and vector operations are (when ignoring the register naming) identical, but some loop conditions differ slightly (functionally identical, but the code is slightly different).

In Figure 4, we visualize the experimental roofline performance we get for the OpenMP version of the memory accessor on an Intel Xeon Gold 6230 Processor (codename Cascade Lake). Compared to the GPU performance results, we note that we are further away from the theoretical peak specified in the technical brief. Consistent with the GPU results, we 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 processors. Interestingly, despite the additional format conversion step, the Accessor<fp64,fp32> benchmark outperforms the fp64 benchmark for arithmetic intensities larger than the machine balance. The analysis of the assembly code in Figure 5 reveals an additional unrolling from the compiler in this case (which may be subject to the specific compiler): For the selected compiler, every load operation is 256bit wide. Since we convert from fp32 to fp64, we now have two 256bit wide vector registers ymm4 and ymm3 in Figure 5b compared to only ymm2 in Figure 5a. Two independent vector registers utilize the vector pipeline better than a single register, resulting in better performance. We recall that additional computations needed for the format conversion are not accounted for in the FLOP count. However, they are negligible compared to the computational load.

Figure 6: Accuracy (top) and performance relative to the standard double precision GMRES (bottom) of the CB-GMRES algorithm employing the memory accessor and using different memory precisions. The GMRES algorithms are enhanced with a scalar Jacobi preconditioner and use a restart parameter of 100. The target architecture is an NVIDIA V100 GPU.

5. Memory Accessor Usage Examples

We now present two examples showcasing the use of the memory accessor and its benefits in scientific applications, the Compressed Basis GMRES and Mixed Precision Sparse Approximate Inverse.

5.1 COMPRESSED BASIS GMRES

The Compressed Basis (CB-) GMRES is based on the idea of compressing the Krylov basis vectors in memory for accelerating the iterations. Without detailing the numerical effects in this report, we mention that for moderate rounding effects, the perturbations incurred by rounding the vector values to lower precision formats can be compensated with a few additional GMRES iterations [7]. In the CB-GMRES algorithm deployed in the Ginkgo library, the user can preset the precision format for the memory operations. Each generated Krylov basis vector is then converted to the memory precision format and stored in compressed form. All further access to the Basis vector is then handled by the memory accessor, retrieving the compressed values, and converting them back to working precision for the arithmetic operations. Accessing the Krylov basis in lower precision reduces the memory access cost, therewith accelerating the GMRES iterations. For completeness, we visualize in Figure 6 the accuracy (top) and performance (bottom) of the CB-GMRES algorithm using different memory precision relative to the performance of the standard double precision GMRES. We note that these speedups account for all numerical effects, including the need for additional GMRES iterations.

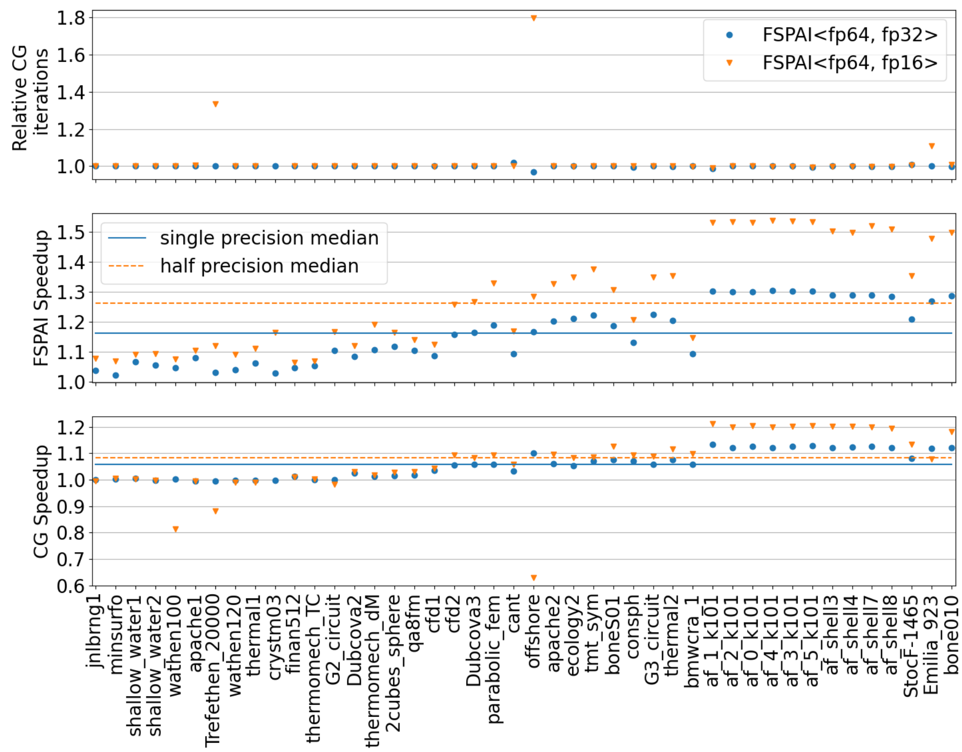

Figure 7: Preconditioned CG iterations (top), mixed precision SAI speedup (center), and resulting CG speedup (bottom) for a mixed precision SAI compressing the preconditioner data using the memory accessor. The target architecture is an NVIDIA V100 GPU.

5.2 MIXED PRECISION SPARSE APPROXIMATE INVERSE PRECONDITIONERS

Sparse Approximte Inverse (SAI) preconditioners are an efficient preconditioning for the iterative solution of sparse linear systems. For most problems, these preconditioners are memory bound on virtually all architectures. This motivates the strategy of storing SAI preconditioners in compressed form, therewith reducing the pressure on the main memory bandwidth. Generally, this strategy reduces the preconditioner quality, and potentially even destroys the preconditioner regularity, which results in the algorithmic breakdown. However, SAI preconditioners are often only a rough approximation of the system's inverse, and in many cases, the precision reduction does not hurt the preconditioner quality.

In Figure 7 we visualize the speedup a mixed precision SAI preconditioner achieves over the double precision counterpart when used inside a Conjugate Gradient (CG) iterative solver. We notice that despite the occasional need for additional CG iterations, the memory accessor compressing the preconditioner data for faster preconditoner application on average reduces the overall CG execution time. Employing the memory accessor for SAI preconditiong is in particular attractive as it can easily be integrated into the SAI application step without having to modify the top-level CG iterative solver. In fact, no modifications to the SAI preconditioner itself are needed as the preconditioner application boils down to a sparse matrix vector multiplication (SpMV) with the sparse inverse approximation, and the mixed precision SAI simply replaces the standard high precision SpMV with the SpMV that retrieves the values from the approximate inverse preconditioner via the memory accessor.

6. Conclusion

We described in this document the technical realization of the memory accessor, a modern compile-time C++ based facility which enables on the fly conversion between memory and arithmetic storage format. The memory accessor is designed to allow both lossless and lossy compressions of data on the fly in registers and to have negligible overhead as well as be extensible to new architectures and new memory compression schemes.

We have evaluated the memory accessor through synthetic benchmarks on high-end hardware coming from several vendors (namely NVIDIA, AMD and Intel) and have shown that not only does the accessor show no overhead, it allows to benefit from the much higher memory bandwidth of lower storage precision types. We have also shown concrete use cases and real-life performance benefits of the accessor in two mixed-precision algorithms, a GMRES implementation and a Sparse Approximate Inverse preconditioner implementation.

Acknowledgments

This work was supported by 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 under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Disclaimer

References

[1] Ahmad Abdelfattah, Hartwig Anzt, Erik G. Boman, Erin Carson, Terry Cojean, Jack Dongarra, Mark Gates, Thomas Grützmacher, Nicholas J. Higham, Sherry Li, Neil Lindquist, Yang Liu, Jennifer Loe, Piotr Luszczek, Pratik Nayak, Sri Pranesh, Siva Rajamanickam, Tobias Ribizel, Barry Smith, Kasia Swirydowicz, Stephen Thomas, Stanimire Tomov, Yaohung M. Tsai, Ichitaro Yamazaki, and Urike Meier Yang. A survey of numerical methods utilizing mixed precision arithmetic, 2020.

[2] Hartwig Anzt, Goran Flegar, Thomas Grützmacher, and Enrique S Quintana-Ortí. Toward a modular precision ecosystem for high-performance computing. *The International Journal of High Performance Computing Applications*, 33(6):1069–1078, 2019.

[3] Hartwig Anzt, Terry Cojean, Goran Flegar, Fritz Göbel, Thomas Grützmacher, Pratik Nayak, Tobias Ribizel, Yuhsiang Mike Tsai, and Enrique S. Quintana-Ortí. Ginkgo: A modern linear operator algebra framework for high performance computing, 2020.

[4] M. Burtscher and P. Ratanaworabhan. High throughput compression of double-precision floating-point data. In *2007 Data Compression Conference (DCC'07)*, pages 293–302, 2007.

[5] Vadim Engelson, Dag Fritzson, and Peter Fritzson. Lossless compression of high-volume numerical data from simulations. In *Proc. Data Compression Conference*, 2000.

[6] Peter Lindstrom. Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics*, 20, 08 2014.

[7] José I. Aliaga, Hartwig Anzt, Thomas Grützmacher, Enrique S. Quintana-Ortí, and Andrés E. Tomás. Compressed basis gmres on high performance gpus, 2020.