BLAS-based Block Memory Accessor with Applications to Mixed Precision Sparse Direct Solvers

PATRICK AMESTOY, Mumps Technologies, France
ANTOINE JEGO, Sorbonne Université, CNRS, LIP6, France
JEAN-YVES L'EXCELLENT, Mumps Technologies, France
THEO MARY, Sorbonne Université, CNRS, LIP6, France
GRÉGOIRE PICHON, LIP (Université de Lyon, ENS de Lyon, UCBL, CNRS, Inria), France

Mixed precision "memory accessor" approaches decouple storage and compute precisions (data is stored and accessed in low precision, but computations are kept in higher precision) to reduce data accesses, improve accuracy, and simplify programming. In this work, we develop such a memory accessor aimed at accelerating sparse direct solvers and propose several new improvements. In particular, we propose a BLAS-based, block approach that can directly rely on BLAS libraries for efficiency and portability. When considering BLAS-2 memory-bound operations like triangular solves, we observe that the performance adequately matches the storage cost, in multiple parallel settings, provided that the conversion from storage to compute precision is efficient, and that the block size is suitably chosen. For the storage precision, we leverage custom floating-point types unsupported by hardware, and we take advantage of the recent AVX512-VBMI instruction set to reach an improved efficiency. We also consider rank-structured matrix representations such as the block low-rank (BLR) format, and explain how to optimize the memory accessor for such matrices. We present preliminary performance experiments using the sparse direct solver MUMPS with adaptive precision BLR approximations. Our results confirm the potential of these memory accessor approaches to achieve efficiency while optimizing storage.

CCS Concepts: • **Mathematics of computing** → **Mathematical software performance**; *Solvers*; • **Computer systems organization** → Single instruction, multiple data; • **Theory of computation** → **Numeric approximation algorithms**.

Additional Key Words and Phrases: Mixed precision, numerical linear algebra, sparse direct solver, triangular solve, memory accessor, floating-point arithmetic, block low-rank matrices

ACM Reference Format:
Patrick Amestoy, Antoine Jego, Jean-Yves L'Excellent, Theo Mary, and Grégoire Pichon. YYYY. BLAS-based Block Memory Accessor with Applications to Mixed Precision Sparse Direct Solvers. *ACM Trans. Math. Softw.* 1, 1 (April YYYY), 27 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The goal of this work is to accelerate the solution of sparse linear systems via mixed precision direct methods. In particular, we are specifically interested in strategies that decouple the precisions that are used to store the data and to perform the computations. Such strategies are beneficial in various contexts: for example, we may want to store the LU factors of the matrix in low precision but apply their inverse (that is, perform triangular solves) in high precision in the context of iterative refinement or preconditioned Krylov solvers [2, 4, 14]. This is also useful when the storage

Authors' Contact Information: Patrick Amestoy, patrick.amestoy@mumps-tech.com, Mumps Technologies, Lyon, France; Antoine Jego, antoine.jego@lip6.fr, Sorbonne Université, CNRS, LIP6, Paris, France; Jean-Yves L'Excellent, jean-yves.l.excellent@mumps-tech.com, Mumps Technologies, Lyon, France; Theo Mary, theo.mary@lip6.fr, Sorbonne Université, CNRS, LIP6, Paris, France; Grégoire Pichon, gregoire.pichon@ens-lyon.fr, LIP (Université de Lyon, ENS de Lyon, UCBL, CNRS, Inria), Lyon, France.

YYYY. Manuscript submitted to ACM

precision corresponds to a custom floating-point type with no hardware support, in which case we must use a higher compute precision corresponding to a standard type with native support.

In order to achieve efficiency, strategies that decouple the storage and compute precisions are based on so-called memory accessors [10]: the idea is to access the data in low precision and convert it on the fly to high precision before performing arithmetic operations on it. This allows for reducing the cost of data transfers and should therefore improve the performance of memory-bound computations, such as the triangular solves in the context of direct methods. This concept of memory accessor was first proposed in [10] and implemented in the Ginkgo library [25] to accelerate various memory-bound BLAS computations such as matrix–vector products and triangular solves. Memory accessors have also been developed for accelerating sparse matrix–vector products in [37] and later extended to their adaptive precision variant [23] in [24]. Finally, memory accessors for hierarchical matrix–vector products have also been proposed in [33, 34].

One limitation of most existing memory accessor approaches is that the data is accessed at the scalar (elementwise) level: the matrix coefficients are converted from the storage to the compute precision one at a time. This requires entirely handcoding the target kernel, which makes achieving efficiency and portability difficult, and goes against the traditional separation of concerns whereby high level linear algebra operations rely on low level BLAS kernels optimized for a particular architecture.

In this work, we take a different approach: we investigate a block memory accessor that loads into memory and converts entire blocks of scalars at a time, and relies on BLAS libraries to perform computations on these blocks. This approach avoids the heavy programming duty of rewriting and extending the BLAS, and allows for exploiting optimized uniform precision BLAS kernels. This block approach has been much less investigated than the scalar approach. It has been used in [33], where it is called "semi-on-the-fly", although it was not investigated in depth.

The main contribution of this article is therefore to perform a thorough investigation of this BLAS-based block memory accessor approach and its potential to accelerate mixed precision sparse direct solvers. This study is organized as follows. First, we describe in more detail the motivations for developing memory accessors in section 2. Then, in section 3, we describe our block memory accessor approach and its implementation for the triangular solve kernel. The rest of the study is then organized in an incremental fashion. We start in section 4 by evaluating the performance of our memory accessor in the simplest scenario, using only the natively supported floating-point types fp64 and fp32, and with a sequential implementation of the accessor that each thread uses to solve its own independent triangular system. Then, in each section, we extend and generalize the approach as follows. In section 5, we investigate the use of custom floating-point types and describe an efficient conversion operation. In section 6, we adapt the accessor for matrices compressed with block low-rank approximations [6]. In section 7, we discuss the parallelization of the memory accessor. Finally, in section 8, we describe the integration of the memory accessor within the sparse direct solver MUMPS [6, 7], and we report preliminary performance results.

2 Motivations for memory accessors in sparse direct solvers

We are interested in efficiently solving large sparse linear systems $Ax = b$, with a focus on direct methods. Direct methods rely on a factorization of A, such as $A = LU$, to directly solve for x. They are numerically robust and reliable, but also very computationally intensive. This has motivated the use of a variety of approximated computing methods to reduce their cost, such as low-rank approximations [6] or mixed precision arithmetic [30]. In the following, we explain why the use of these approximations creates various contexts where being able to decouple the storage and compute precisions becomes beneficial.

2.1 Mixed precision iterative refinement

One of the most common and successful methods for accelerating the solution of linear systems is by using mixed precision arithmetic. In the context of direct, LU factorization–based methods, the natural strategy is to compute the LU factorization in low precision, use the approximate LU factors to obtain a first approximation of the solution, and then refine it to high accuracy using high precision. This is outlined in Algorithm 1, which presents mixed precision iterative refinement where the correction system $Ad = r$ is solved with an unspecified solver. The simplest approach to solve this system is to directly use the approximate LU factors to solve $LUd = r$ by substitution. However, this LU-based iterative refinement approach can only converge for sufficiently well-conditioned matrices: specifically, it requires $\kappa(A)u_f \ll 1$, where u_f is the (low) factorization precision [14, 35].

Algorithm 1 Mixed precision iterative refinement for solving $Ax = b$.

```
Compute A ≈ LU in low precision.
Initialize solution by solving LUx_0 = b by substitution.
for j ∈ 0, 1, ... do
    Compute r = b − Ax_i in high precision.
    Solve Ad ≈ r.
    Update x_{i+1} = x_i + d in high precision.
end for
```

More robust approaches that can handle ill-conditioned matrices have been proposed: in particular, GMRES-based iterative refinement [4, 13, 14] solves $Ad = r$ by the Krylov method GMRES [38] preconditioned by the approximate LU factors. However, GMRES-IR requires the preconditioned matrix–vector product to be applied in a higher precision than the LU factorization precision; a typical setting in practice is to apply it in the working precision. In absence of a memory accessor approach, the baseline strategy to meet this requirement is to cast the entire LU factors from low to high precision after they have been computed. Amestoy et al. [2] carry out an experimental study on large sparse matrices that confirms the ability of this strategy to solve ill-conditioned problems in mixed precision; however, the cast of the LU factors and their application in high precision represent a significant cost, both in terms of runtime and storage. GMRES-IR thus constitutes a first setting in which a memory accessor would be beneficial, by allowing the low precision LU factors to be applied in high precision.

2.2 Custom floating-point types

While the number of floating-point formats supported in hardware has considerably grown in the last decade (with, in particular, the emergence of 16-bit and 8-bit formats), there still remains large gaps between the unit roundoffs of commonly supported formats. For example, there is no hardware support for intermediate formats between 64-bit and 32-bit. Yet, many applications may require these intermediate levels of accuracy; this is in particular common when other types of errors are introduced, such as when using low-rank approximations [6] (with a truncation threshold ε) or iterative solvers (with a stopping tolerance ε), where ε is an accuracy parameter that does not need to correspond to the unit roundoff of any floating-point format. In such cases, it is natural to optimize the size of the floating-point types by using custom formats whose unit roundoff is approximately ε.

One simple approach to define such custom types is to take a standard high precision format, such as fp64, and truncate the mantissa by the necessary number of bits. The efficient handling of the resulting datatype has been for example studied in [36] and [8], the latter proposing vectorization techniques based on SSE instruction sets for

improved efficiency. Other datatypes have been explored where both the mantissa and exponent are truncated such as in [39] and [24]. Moreover, some packages such as chop [31], FloatX [21], or cpfloat [19] allow for processing custom-defined floating-point types, although their focus is on simulation and not performance. Finally, posit numbers can also be viewed as custom datatypes given their marginal adoption on widely available hardware; see [26] for a detailed discussion and comparison with floating-point types.

While these custom formats may be useful to optimize the storage of floating-point types, since they are not supported in hardware, computing with them requires conversions to supported higher precision types; thus, an efficient memory accessor is also of interest in this context.

2.3 Adaptive precision methods

Memory accessors may be particularly beneficial when combining the two previous settings: adaptive precision methods [30, sect. 14] are a subclass of mixed precision methods that adapt the precisions used to the computation at hand. Such adaptive precision methods have in particular been developed for computing sparse matrix–vector products [23, 24], low-rank approximations [3], and block Jacobi preconditioners [9, 20]. Inexact (or relaxed) Krylov methods can also be seen as adaptive precision [22]. Adaptive precision methods can target any arbitrary level of accuracy ε (they have been called variable accuracy methods, as opposed to variable precision, for this reason [1]). They leverage small quantities (matrix coefficient, rank-one component, residual vector…) appearing in the computation by switching them to a precision inversely proportional to their magnitude: a quantity of magnitude smaller than ε/u_k can be switched to a precision with unit roundoff u_k. Therefore, the more floating-point formats are available, the more we can gradually adapt the precisions to the data at hand; this makes custom types particularly attractive for adaptive precision methods, and memory accessors become even more critical to achieve high performance.

3 Mixed precision BLAS-based block memory accessors

The goal of this work is to develop an efficient mixed precision memory accessor for accelerating sparse direct solvers and, more specifically, the triangular solve phase, which is memory bound.

In section 3.1, we present our approach, which operates at the block level, and describe its implementation for triangular solves. Then, in section 3.2, we present the experimental setting that will be used throughout the rest of this article for evaluating the performance of this approach.

3.1 Mixed precision triangular solve with a block memory accessor

As mentioned in the introduction, we propose a block memory accessor approach that loads and converts the data by blocks, and exploits the BLAS to perform the computations.

Throughout this article, we focus on the BLAS kernel `trsv`, which solves the triangular system $Tx = y$, where T is an $m \times m$ triangular matrix, y is the right-hand side, and x is the solution vector to be computed. Without loss of generality, we assume that $T = U$ is upper triangular; the lower triangular case can be handled analogously.

We consider a mixed precision implementation of this kernel with two precisions: u_{low}, the low storage precision, and u_{high}, the high compute precision (note that, throughout this article, we use the metonymy "precision u" for "precision with unit roundoff u"). Algorithm 2 describes our implementation based on a block memory accessor, parameterized by the block size b. The triangular matrix U is partitioned into $b \times b$ blocks U_{ij} and the system $Ux = y$ is solved by blocks. Whenever a given block U_{ij} needs to be used, it is loaded from the main memory, where it is stored in precision u_{low},

and it is converted (upcasted) to precision u_{high} into a temporary workspace B. The computation is then performed with B in precision u_{high}.

Algorithm 2 Mixed precision triangular solve with a block memory accessor.

Input: U, an $m \times m$ upper triangular matrix stored in precision u_{low} and partitioned into $b \times b$ blocks U_{ij}, with $m = pb$; y, an m-vector stored in precision u_{high}; B, a $b \times b$ workspace stored in precision u_{high}.
Output: x, an m-vector stored in precision u_{high}, solution to $Ux = y$.

```
Initialize x = y.
for j = p, ..., 1 do
    Read U_jj, upcast it to precision u_high, and store it in B.
    Solve B x_j = x_j in precision u_high using trsv.
    for i = j − 1, ..., 1 do
        Read U_ij, upcast it to precision u_high, and store it in B.
        Compute x_i = x_i − B x_j in precision u_high using gemv.
    end for
end for
```

This block approach presents several advantages. The workspace B is reused for all blocks U_{ij} and therefore we only require an extra storage of controlled size $b \times b$. All BLAS calls are executed with standard-complying uniform precision routines; because such routines are often hand-optimized we expect them to be very efficient. Assuming that efficient conversion operations from precision u_{low} to precision u_{high} are also available, the performance of the algorithm should therefore be mainly driven by the data transfers, which are reduced thanks to the low storage precision.

With its block size parameter b, Algorithm 2 offers a flexible tradeoff between the efficiency of the BLAS calls and the efficiency of the data transfers – which happen from the slow memory holding the entire linear system to the fast memory close to the compute registers. In order for the data transfers to be efficient, it is important that the workspace B remains in fast memory before the BLAS call; if it spills into the slow memory then this mixed precision approach is essentially pointless as the high precision B will need to be reloaded into the fast memory. Therefore, the ideal is to set b sufficiently large for the BLAS calls to be efficient, but sufficiently small for B to fit into the fast memory. Note that what constitutes the "fast" and the "slow" memory is architecture dependent; we may expect the registers and highest levels of cache to be fast, and the main memory to be slow. We will analyze how to choose b in section 4.2.

Note that Algorithm 2 is written in a right-looking form; a left-looking form is also possible, as will be described in section 4.1.

3.2 Experimental setting of this study

Our end goal is to use the block memory accessor approach to accelerate the solve phase of sparse direct solvers. The computations for this solve phase follow the so-called elimination tree [17], whose nodes are associated with smaller dense matrices with which we must compute dense triangular solves, implemented as described in Algorithm 2.

Therefore, we will experiment with two types of configurations. The first is a tree parallelism configuration, where we perform in parallel many independent dense solves with relatively small matrices, each solve being sequential; this corresponds to the typical workload towards the bottom of the elimination tree. The second is a node+tree parallelism configuration, where we perform in parallel a few dense solves with bigger matrices, and each solve is also parallel; this corresponds to the typical workload towards the top of the tree. In our experiments, we first consider the tree

Table 3.1. Compute platform used in this work.

	chirop
CPU	2× Intel Xeon Platinum Ice Lake 8358
core / CPU	32
Memory channels / CPU	8
Memory / Node	512 GiB DDR4
L3 cache (MB)	48
L2 cache (KB)	1280
L1 cache (data, KB)	48

parallelism configuration in sections 4 through 6; we turn to the node+tree parallelism configuration in section 7, which requires parallelizing Algorithm 2.

Our experimental setup is an Intel Xeon Gold (Ice Lake 8358) dual-CPU compute node hosted on the `chirop` partition within the Grid5000 network[1]. Hardware details are reported in Table 3.1. We have used intel compilers 2023.0.0, MKL 2023.0.0 and intel mpi 2021.8.0.

For all experiments, we report the median performance across ten runs.

4 Performance evaluation with fp64 and fp32

We begin by evaluating the performance of our block memory accessor in the simplest scenario: we use fp64 as the compute precision u_{high} and fp32 as the storage precision u_{low}; both floating-point types have native hardware support and the conversion operations between them are highly efficient.

Moreover, we focus on the tree parallelism configuration: one thread is spawned per core and each thread solves an independent triangular system of order $m = 4096$, using a sequential implementation of Algorithm 2.

We use this simplest scenario to validate the potential of the approach and to analyze the effect of various design choices and parameters, notably the matrix layout and the block size.

4.1 Choice of matrix layout

In Figure 4.1, we compare the performance of eight variants depending on the following choices:

- contiguous vs tiled: in the contiguous case the entire matrix U is contiguous in memory, whereas in the tiled case each block U_{ij} is contiguous;
- row-major vs column-major layout: this distinction applies to either the entire matrix U (in the contiguous case) or to each individual tile (in the tiled case);
- right-looking vs left-looking: Algorithm 2 is written in a right-looking (eager) form: the updates $x_i = x_i - U_{ij}x_j$ for $i = 1, \ldots, j-1$ are all performed at step j, as soon as possible, by traversing the jth block-column of matrix U. Alternatively, the triangular solve can also be written in a left-looking (lazy) form, where all the updates $x_i = x_i - U_{ij}x_j$ for $j = i+1, \ldots, p$ are all performed at step i, as late as possible, by traversing the ith block-row of matrix U.

[1] https://www.grid5000.fr/w/Lille:Hardware

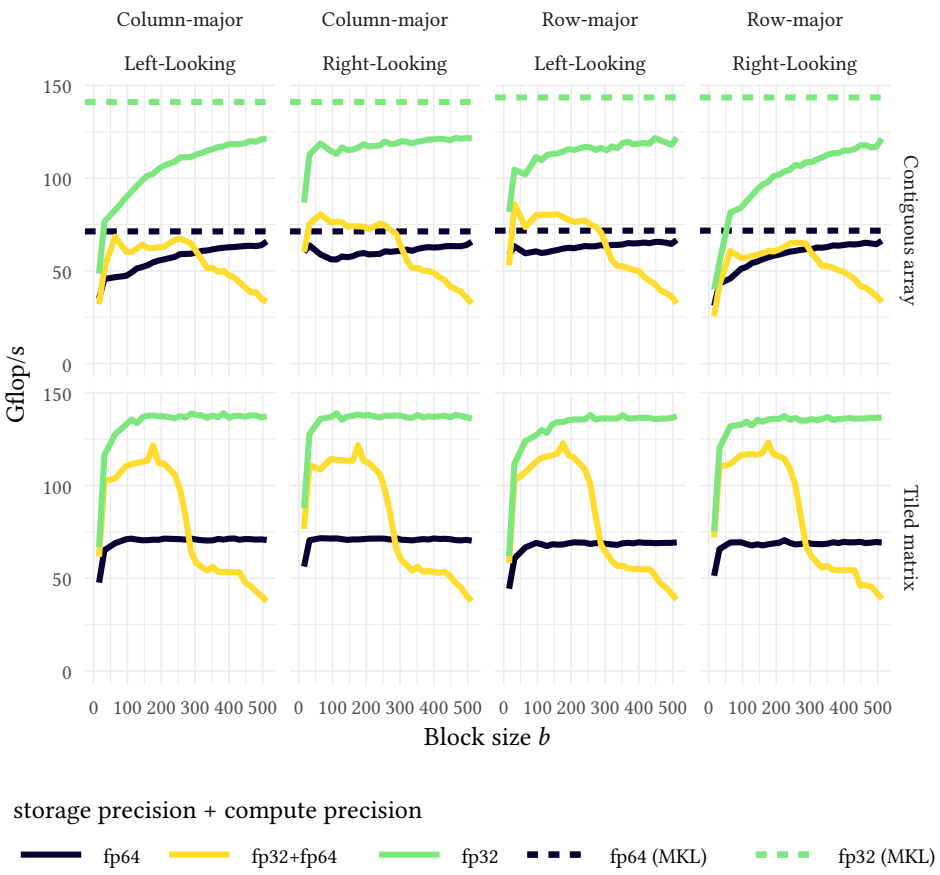

Fig. 4.1. Performance of triangular solve with fp64 and fp32 native types and tree parallelism, depending on the matrix layout and on the block size. Each core on the compute node is solving a system of order 4096.

For each of these eight variants, we measure their performance for three precision configurations: two uniform precision ones (either fp64 or fp32 as both storage and compute precisions, black and green solid lines in Figure 4.1) and a mixed precision one (fp32 as storage precision and fp64 as compute precision, yellow line). In the uniform precision cases, naturally, no conversion is required. Finally, we also report the performance obtained by the MKL implementation in uniform precision, both in fp64 and fp32 (black and green dashed lines). This provides a state-of-the-art reference performance for the uniform precision runs.

We can draw many interesting conclusions from the results shown in Figure 4.1.

- As a general rule, the performance of the uniform precision variants (black and green lines) tends to increase as the block size increases, thanks to more efficient BLAS calls, until it reaches a plateau. The mixed precision variant (yellow line) behaves very differently: while it also increases until it reaches a plateau, this plateau ends in an abrupt performance drop when the block size becomes too large. This is expected, since, as explained

previously, we require the workspace B to remain into the fast memory; we will analyze the precise value of the block size for which this performance drop happens in the next section.

- The results confirm the relevance of our BLAS-based block approach: the performance is much lower for very small block sizes, of which the scalar memory accessor is the special case $b = 1$. We emphasize that the comparison is not entirely fair: our code is not optimized for small b (or indeed $b = 1$); the point is that our block approach does not require such handcoded, architecture-specific optimizations and rather relies on BLAS for performance.
- In the case where the entire matrix is contiguous, the order of operations has a significant impact on performance: the right-looking variant with column-major layout and the left-looking variant with row-major layout perform noticeably better than the other two combinations. This effect is notably visible for the mixed precision variant, regardless of the choice of block size. This is explained by the fact that in left-looking we traverse the matrix by block-rows, whereas in right-looking we traverse it by block-columns. Using a matching layout presumably benefits from prefetching.
- On the contrary, if the matrix is stored with a tiled layout, all combinations of left/right-looking and row/column-major perform comparably and much better than the contiguous variants. Moreover, the performance of the tiled variants matches that of MKL (dashed lines) when the block size is sufficiently large, which validates our baseline uniform-precision triangular solve.

Overall, with the tiled layout and a suitable choice of block size, the mixed precision variant (yellow line) significantly outperforms the fp64 variant (black line) and almost matches the performance of the fp32 variant (green line), which is expectedly about twice higher. This suggests that the performance of the solve is mainly driven by the data transfers, and confirms the relevance of memory accessor approaches for accelerating such operations while preserving computations in high precision.

4.2 Choice of block size

The experiments reported in Figure 4.1 show that the performance of the block memory accessor drops abruptly when the block size b exceeds a certain value. In this section we attempt at correlating this value with the properties of the machine, which would allows us to assess what ranges of block sizes are suitable for a given hardware.

To do so, we use the following performance model:

$$\text{time} = \beta\omega_{\text{low}} \cdot \frac{m^2}{2} + \beta \cdot \frac{m^2}{2b^2} \cdot \max\left((\omega_{\text{low}} + \omega_{\text{high}})b^2 - M, 0\right) + \gamma m^2 + o(m^2), \tag{1}$$

where m is the matrix size, b is the block size, ω_{low} and ω_{high} are the size in bytes of floating-point numbers in precision u_{low} and u_{low}, β is the inverse bandwidth to load data from the slow memory to the fast memory, M is the size in bytes of the fast memory, and γ is the inverse flops/s rate. Note that this is a simplified model: we only consider two levels of memory (fast and slow), and we neglect $O(m) = o(m^2)$ terms in the flop count and number of entries transferred.

Since we are dealing with a memory-bound computation, we expect the γm^2 term to be negligible. The term $\beta\omega_{\text{low}}m^2/2$ simply reflects the fact that we must load U, that is, $m^2/2$ entries stored in precision u_{low} (we load U by blocks U_{ij} of size b^2 at a time, but this does not change the total volume of transfers in precision u_{low}). Finally, the middle term in (1) can be derived as follows. For each block (there are $m^2/2b^2$ of them), we load b^2 entries in precision u_{low}, convert them to precision u_{high} storing the result in B, and perform a computation on B. Therefore, we need to store in the fast memory $s = (\omega_{\text{low}} + \omega_{\text{high}})b^2$ entries at the same time; if the fast memory is large enough ($s \leq M$), then we do not need to load anything more and this term is zero. However, if s is too large to fit into the fast memory

($s > M$), then the $s - M$ bytes that have overspilled are back into the slow memory, and we must load them again. Therefore, we can conclude that the optimal performance is obtained when $s \leq M$, that is, when

$$b \leq \sqrt{\frac{M}{\omega_{\text{low}} + \omega_{\text{high}}}}. \tag{2}$$

Let us now assess the relevance of this analysis by applying (2) to the setting used for the experiments of Figure 4.1. We have $\omega_{\text{low}} = 4$ (fp32), $\omega_{\text{high}} = 8$ (fp64), and, according to Table 3.1, $M = 1328 \times 1024$, assuming the "fast memory" to be composed of the L1 and L2 caches. Condition (2) then becomes $b \lesssim 337$, which roughly matches the values at which we observe the performance of the mixed precision dropping. In practice, the performance drop occurs for slightly smaller values than what the model predicts, which may simply be because it is too simplistic, or perhaps because we should not assume the entire cache size to be available; indeed, other elements are likely stored in cache, such as some frequently-accessed variables within the data structures that we use (such as block sizes) or other elements that we do not control.

5 Custom floating-point types

As explained in section 2.2, programs that require an accuracy that does not match the standard IEEE fp64 or fp32 formats may benefit from using custom types, in order to target specific user-defined accuracies. In this section, we explain how to extend our BLAS-based memory accessor approach to such custom types.

First, we define in section 5.1 the custom types that we will consider. Then, we describe in section 5.2 how to efficiently perform conversions from these custom types to higher precision standard types; in particular, we describe a method that can take advantage of the latest AVX512 instruction sets. Finally, we evaluate the performance of our accessor with these custom types in section 5.3.

5.1 Definition of custom types

The IEEE-754 standard defines binary floating-point numbers through three values: their sign s, their (biased) exponent e, and their mantissa m. While s is simply encoded on 1 bit, e and m can be encoded on various number of bits depending on the format; the number of bits for e affects the range of representable numbers and the number of bits for m affects the relative precision of representable numbers. To avoid any ambiguity when dealing with custom floating-point numbers, we will refer to them as eνmμ where ν is the number of bits for e and μ the number of bits for m (we do not include the implicit leading bit in μ). Thus, for example, the standard fp32 and fp64 formats correspond to `e11m52` and `e8m23`, respectively.

Custom floating-point types can be defined by taking standard types and reducing ν and/or μ. In this work, we will not explore reducing ν, the number of bits for the exponent, below 8—the value used by fp32. This is an interesting perspective to further optimize storage, which we leave for future work. We will thus only focus on reducing μ, the number of bits for the mantissa. Such reduced precision types can be easily defined by simply truncating some of the least significant bits of a standard type. For alignment and commodity reasons, we will only consider custom types whose total number of bits is a multiple of 8 (that is, is an integer number of bytes).

By removing some of the least significant bytes from the fp64 type, we can obtain a total of six different custom types: `e11m`μ, with $\mu \in \{44, 36, 28, 20, 12, 4\}$ bits for the mantissa. Alternatively, we can remove some of the least significant bytes from the fp32 type to obtain two more custom types: `e8m15` and `e8m7`. Note that `e8m15` and `e11m12` both have the same size (24 bits): the difference is that the former is obtained by truncating the mantissa of an fp32 number, and the

Table 5.1. List of custom floating-point types obtained by truncating the mantissa of the standard fp64 and fp32 types (we also include these two standard types for completeness).

Unambiguous name	Ambiguous abbreviation
`e11m4`	—
`e8m7`	fp16
`e11m12`	—
`e8m15`	fp24
`e11m20`	—
`e8m23`	fp32
`e11m28`	fp40
`e11m36`	fp48
`e11m44`	fp56
`e11m52`	fp64

latter by truncating that of an fp64 number; these two types thus offer different tradeoffs between precision and range. The same comment applies to `e8m7` and `e11m4`, which are both 16-bit types, and to `e8m23` (the standard, untruncated fp32 type) and `e11m20`.

Table 5.1 summarizes the list of custom types that we will consider. We anticipate that, in section 5.3, we will compare the performance of types truncated from fp32 with that of types of the same size but truncated from fp64, and show that the former matches the latter, so that we will subsequently discard the types `e11m4`, `e11m12`, and `e11m20`. We will thus be left with five custom types of different sizes, which we will ambiguously abbreviate to fp16, fp24, fp40, fp48, and fp56.

5.2 Conversion between custom and standard types

We now discuss how to implement efficient conversion operations between a custom floating-point number and a standard type of higher precision.

5.2.1 Scalar conversion. Since the size of the custom types that we consider is always a whole number of bytes, we can represent custom numbers as arrays of `uint8_t`, whose size corresponds to the number of bytes of the custom type. For example, the 48-bit `e11m36` type is represented as `uint8_t[6]`. We can then write the conversion in a relative lightweight fashion by only using bit shifts and additions on the standard integer types[2] `uint64_t`, `uint32_t`, `uint16_t`, and `uint8_t`.

As an example, Figure 5.1 describes how to perform conversions between the standard fp64 (`e11m52`) type and the `e11m36` custom type, represented as `uint8_t[6]`. Note that, for efficiency, we do not process each of these 6 bytes individually (as `uint8_t`). Rather, we decompose this 6-byte array as the concatenation of a 4-byte array and a 2-byte one, which we can process as `uint32_t` and `uint16_t`, respectively. We thus only need to manipulate two integers, which reduces the number of required bitshits and additions. Note that the number of integers that are needed depends on the target custom format: for example, for the 56-bit (7-byte) `e11m44` format, three integers (`uint32_t`, `uint16_t`, and `uint8_t`) are needed. Note that, in our actual implementation, we use the `union`[3] construct to simplify the code by getting rid of the standard `reinterpret_cast`[4].

[2] C fixed-width integer types. https://en.cppreference.com/w/cpp/header/cstdint
[3] Union declaration. https://en.cppreference.com/w/cpp/language/union
[4] Underlying bit pattern conversion. https://en.cppreference.com/w/cpp/language/reinterpret_cast

When converting between the standard fp64 type and the custom types `e8m7` and `e8m15`, we also need to handle the different number of bits used for the exponent. In order to do that efficiently, we use the standard fp32 (`e8m23`) type as intermediary. Thus, for converting from fp64 to `e8m7` (say), we first convert from fp64 to fp32 with a standard conversion and then convert from fp32 to `e8m7` in a similar fashion as described above. Analogously, converting from `e8m7` back to fp64 relies on a fp32 to fp64 standard conversion. Therefore, if the compute precision is fp64, using custom types with 8-bit exponent as storage precision requires an extra conversion between fp32 and fp64. As we will experimentally confirm in the next section, the cost of this extra conversion is however small and usually worth gaining 3 bits of precision.

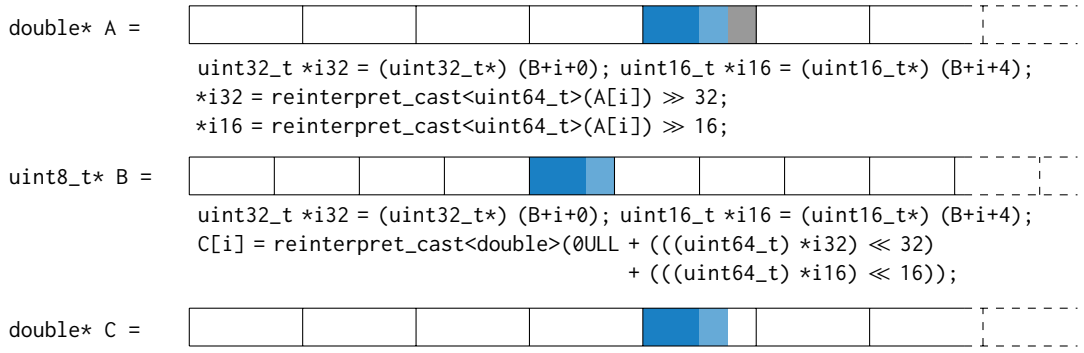

Fig. 5.1. Conversion of an array of numbers from fp64 to `e11m36` and back to fp64. The figure focuses on the conversion of the 5th element. Dark blue rectangle corresponds to the 32 bits that are casted to `uint32_t`. Light blue rectangle corresponds to the 16 bits that are casted to `uint16_t`. Gray rectangle corresponds to the 16 bits that are truncated.

5.2.2 Vectorized conversion. So far, we have discussed how to convert a single scalar number, but our goal is to convert an entire array (block) of such numbers. Naturally, we could simply loop on each element an apply the conversion described above. However, a more efficient conversion can be obtained by exploiting vectorized instructions. Specifically, we will use the `permutexvar` instruction, which is available on recent hardware that possess the `AVX512-VBMI` feature flag[5]. While the scalar conversion presented above remains the most portable method, this vectorized instruction enables to convert multiple elements in a single instruction, which we expect to be potentially much more efficient than using multiple instructions. The gain in efficiency of the vectorization will ensure that the conversion cost is negligible compared to the cost of loading elements from the slow memory. An extreme example is the 56-bit `e11m44` type that, as already mentioned, can only be represented as the concatenation of a `uint32_t`, a `uint16_t` and a `uint8_t` and therefore requires three conversions, two bit shifts, and three additions to convert a single element. As we explain in the following, the `permutexvar` instruction allows for performing the conversion in a single instruction, regardless of the target custom type.

The `permutexvar` instruction requires a permutation array [32, volume 2C, page 471-472]. This array contains as many elements as there are elements in the register: for example, to permute a 512-bit register byte-wise, the permutation array holds 64 integers. The permutation array specifies which parts of each element to extract and is thus different for each custom type. In Figure 5.2, we illustrate the principle behind this conversion by taking the `e11m36` format as an example. Note that, if we must convert a number of elements that does not exactly fit in the 512-, 256-, or

[5]AVX512-VBMI Wikichip entry. https://en.wikichip.org/wiki/x86/avx512_vbmi

128-bit registers, that is, that is not divisible by 8, 4, or 2, the last elements may not be processed via the `permutexvar` instruction. For such elements, our implementation falls back on the scalar conversion described previously.

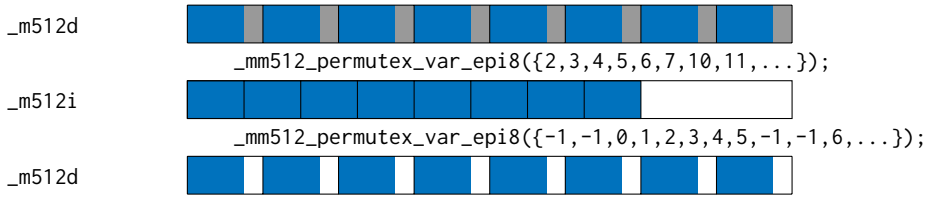

Fig. 5.2. Vectorized conversion of numbers from fp64 to `e11m36` and back to fp64, using the AVX512-VBMI instruction set. Blue rectangles correspond to the 48 bits that are kept. Gray rectangles correspond to the remaining 16 bits that are truncated. Casting from/to register of different types is omitted.

The `e8m7` and `e8m15` types with an 8-bit exponent once again require special handling, as illustrated in Figure 5.3 in the case of `e8m15`. Since the AVX512-VBMI instruction set provides AVX256 interfaces, our implementation once again uses the standard fp32 type as intermediary: the conversion between fp32 and the `e8m7` or `e8m15` type is performed on 256-bit registers instead of 512-bit ones. Moreover, the conversions between fp64 and fp32 can also be vectorized through the AVX512F instruction set using `cvtps_pd` and `cvtpd_ps` [32, volume 2A, page 281]. Once again, we do not expect a significant overhead from this additional conversion: the processor should be able to hide the cost of this extra conversion by pipelining the 512-bit array conversions across the whole array of elements of much larger size.

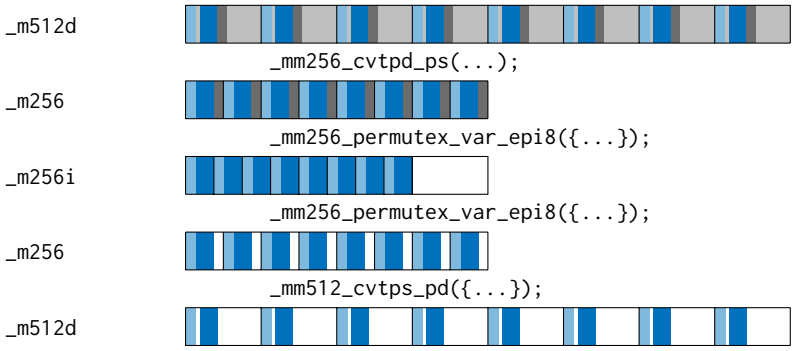

Fig. 5.3. Vectorized conversion of numbers from fp64 to `e8m15` and back to fp64, using the AVX512-VBMI and AVX512F instruction sets. Light blue rectangles correspond to the sign bit and the 8 exponent bits that are kept. Dark blue rectangles correspond to the 15 bits of mantissa that are kept. Light gray rectangles correspond to the 3 exponent bits and 29 mantissa bits that are truncated in the fp64 to fp32 conversion. Dark gray rectangles correspond to the 8 mantissa bits that are further truncated in the fp32 to `e8m15` conversion.

As a final word of warning, we note that the presented conversion scheme from high to low precision essentially uses round-to-zero (RTZ), because it removes the least significant bits. While, in principle, the worst-case rounding error bounds only differ by a factor of two when using directed rounding modes such as RTZ instead of round-to-nearest (RTN), the use of RTZ introduces a bias that can destroy the statistical distribution of rounding errors usually obtained with RTN, which typically benefits from significantly improved (probabilistic) bounds [15, 27, 28]; see [18] for a detailed discussion on this issue. Therefore, in practice, we avoid this pitfall by introducing a correction to the truncation error by setting the most significant bit of the truncature to 1.

5.3 Performance evaluation with custom types

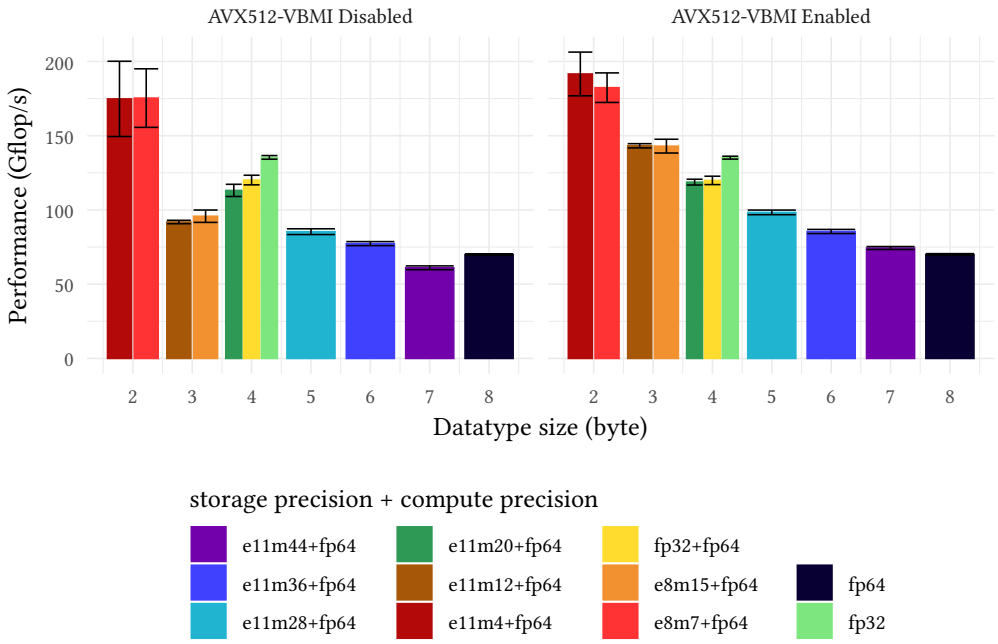

Fig. 5.4. Performance of the triangular solve with custom types and tree parallelism, depending on whether AVX512-VBMI is enabled or not. Each core on the compute node is solving a system of order 4096.

Figure 5.4 evaluates the performance of the memory accessor with each of the custom types previously defined as storage precision and with fp64 as compute precision. As in the experiments of the previous section, we still consider a tree parallelism configuration, that is, each core performs a triangular solve on an independent system of order $m = 4096$, using a sequential implementation of Algorithm 2. We report the median performance obtained on the range of block sizes $[112 : 16 : 196]$. The figure also showcases the impact of the AVX512-VBMI instruction by comparing the performance with the instruction enabled (Figure 5.4, right) or disabled (left); the instruction is disabled by compiling with the `-mno-avxvbmi` option. Each bar corresponds to a different storage floating-point type; types of the same size are grouped. We also report the performance of the uniform precision fp32 (green bar) and fp64 (black bar) solves to provide a reference performance: these two cases do not require conversion.

Comparing the left and right plots of Figure 5.4 shows that using the AVX512-VBMI instruction set can significantly improve the performance. This effect is less strong for types of size 2 and 4 bytes, which obtain relatively good performance even without AVX512-VBMI. This is most likely due to the fact that the decompression is done through a unique primitive type `uint16_t` or `uint32_t`. For the remaining types, while they may achieve better performance than fp64, a smaller type size does not always lead to higher performance without AVX512-VBMI. Indeed, the types of size 3 bytes (composed of a `uint16_t` and a `uint8_t`) achieve a lower performance than the types of size 4 bytes. Similarly, the `e11m44` type of size 7 bytes (composed of a `uint32_t`, a `uint16_t` and a `uint8_t`) achieves a lower performance than the fp64 type of size 8 bytes. As discussed in section 5.2, these complex compositions deter efficiency. Therefore,

without AVX512-VBMI, these types of size 3 and 7 bytes achieve both lower precision and lower performance than the types of size 4 and 8 bytes, and should not be used.

However, when the AVX512-VBMI instruction set is enabled, the picture changes: the performance of all types now follows their respective size—albeit not in a strictly proportional fashion. Indeed, the `e11m44` type (of size 7 bytes) outperforms fp64, and the `e8m15` and `e11m12` types (of size 3 bytes) also outperform fp32. In the case of the `e8m7` and `e11m4` types (of size 2 bytes), while the maximum performance (for an optimal choice of block size) of about 200 Gflops/s is almost unchanged when compared with the case where the instruction set is disabled, the average performance across the range of block sizes is much more consistent, as indicated by narrower intervals.

We now compare the performance of types of the same size (bars from the same group in Figure 5.4), depending on whether they are obtained by truncating an fp64 or fp32 type. Note that since the compute precision is fp64 in all cases, decompressing the fp32-truncated types requires to first decompress the custom type into an fp32 type and then an extra conversion from fp32 to fp64, as described in section 5.2. Thus, the goal of the following comparison is to check whether this extra conversion has a significant impact on the performance. The following comments apply both when the AVX512-VBMI instruction set is enabled or disabled. For the 2-byte types, there is somewhat of a tradeoff since `e8m7` is slightly slower than `e11m4`; however, the difference in performance does not seem to be worth losing 3 bits of precision. For the 3-byte types, the conclusion is clearer, since `e8m15` matches the performance of `e11m12`. The same applies to the 4-byte types, for which the fp32 type matches or even outperforms the `e11m20` type. Therefore, based on these results, we can discard the `e11m20`, `e11m12`, and `e11m4` types, in favor of the `e8m23` (fp32), `e8m15`, and `e8m7` types, respectively. We are thus left with five custom types of different sizes; in the rest of this manuscript, we abbreviate these types to fp16, fp24, fp40, fp48, and fp56, as indicated in Table 5.1.

So far with Figure 5.4, we have evaluated the performance of the custom types averaged over a range of suitable block sizes. We now perform a more detailed experiment to assess how we should choose the block size depending on the size of the custom type. The results are shown in Figure 5.5. As in the previous experiment of section 4, black and green dashed lines correspond to the MKL in fp64 and fp32, black and green solid lines correspond to our own implementation of uniform fp64 and fp32 variants with varying block size b; for a large enough block size, the performance of our blocked implementation matches that of the MKL. The rest of the lines correspond to the memory accessor approach with fp64 as compute precision and a lower precision as storage precision. The yellow line, which is the same as in the previous section, uses the standard fp32 type as storage precision; all the other lines are new and use custom types as storage precision.

Figure 5.5 shows that the continuum of sizes of the custom types translates well into a continuum of performance, as long as the block size is suitably chosen. Block sizes in the interval $[100, 200]$ perform relatively well across all variants; performance drops abruptly for block sizes larger than about 250. This behavior is consistent with the model that we used in section 4.2: interestingly, according to (2), the maximum suitable block size should slightly increase as the size of the storage precision type ω_{low} decreases, since reducing the size of the low precision blocks loaded in the fast memory creates more room for larger blocks. This is quantified in Table 5.2, which indicates that the maximum block size satisfying condition (2) increases from 302 for fp56 to 369 for fp16. Although, once again, in practice the performance drops for smaller block sizes than what the model indicates, the experiments do confirm that smaller storage types tend to continue to perfom well for larger block sizes than larger types.

We conclude this section with an experiment measuring separately the time spent in the conversion and in the BLAS calls; we focus on the `gemv` calls whose cost dominates that of the `trsv` calls. Figure 5.6 shows this time breakdown corresponding to the runs with $b = 128$ in Figure 5.5; times are sampled across four runs and we report the results as

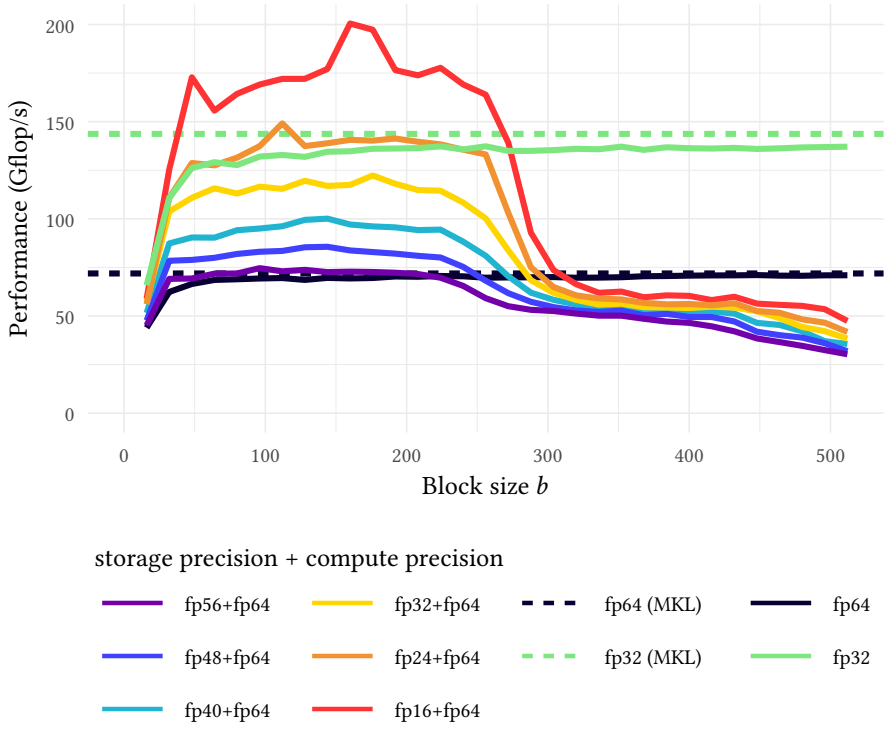

Fig. 5.5. Performance of the triangular solve with custom types and tree parallelism, depending on the block size. Each core on the compute node is solving a system of order 4096.

Table 5.2. Maximal values of b for which condition (2) is satisfied, with fp64 as compute precision ($\omega_{\text{high}} = 8$) and depending on the custom type used as storage precision. We have taken $M = 1328 \times 1024$, which corresponds to the size in bytes of the L1 and L2 caches of the machine (see Table 3.1).

custom type	ω_{low}	maximum value of b
fp16	2	369
fp24	3	352
fp32	4	337
fp40	5	323
fp48	6	312
fp56	7	302

boxplots. The figure shows that the time spent in the BLAS `dgemv` call is not impacted by the storage precision, and is always aout 5µs. This is entirely expected as it is the conversion operation that performs the expensive transfer of the blocks from the central memory to the fast memory. The transfer plus conversion times (computed as the total time minus the time spent in the BLAS) is close to being linear with respect to the size of the storage type, which confirms that its cost is mainly related to the volume of data loaded. Note that while the 5µs spent in `dgemv` is negligible when the storage precision is high, its relative incompressible cost grows as the size of the storage type decreases. For the

fp16 storage precision, about half the time is spent in the dgemv. This explains why the fp16+fp64 variant is "only" about 2.5× faster than fp64, rather than 4× as one may naively have expected.

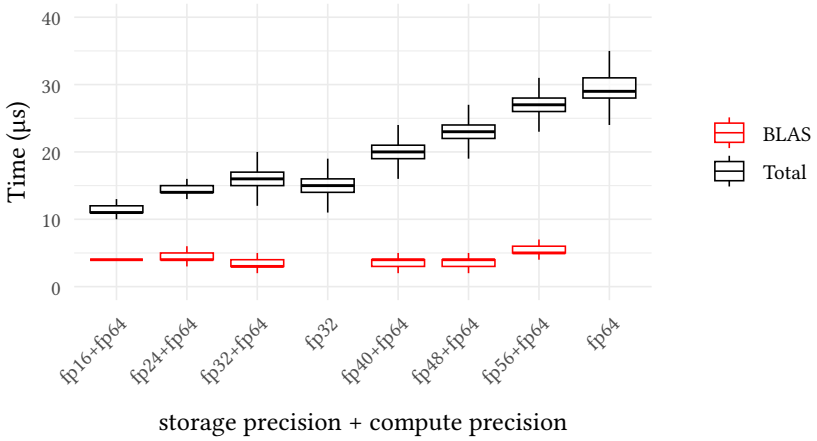

Fig. 5.6. Breakdown of the time spent computing mixed-precision matrix-vector multiply where the time of the uniform-precision BLAS call is extracted. The reminder of the time corresponds to the transfer and conversion of the low-precision matrix. Times have been extracted for the case $b = 128$ of Figure 5.5.

6 Block low-rank triangular solve

The objective of this section is to adapt the memory accessor for dense full-rank triangular solves presented in section 3 to matrices with a block-low rank (BLR) structure. We first review the basic properties of BLR matrices in section 6.1. Then we present an extension of Algorithm 2 to BLR matrices in section 6.2. Finally we evaluate its performance in section 6.3.

6.1 Basic reminders on BLR matrices

A block low-rank (BLR) approximation of a matrix $A \in \mathbb{R}^{m \times m}$ has the block $p \times p$ form

$$\widetilde{A} = \begin{bmatrix} A_{11} & \widetilde{A}_{12} & \cdots & \widetilde{A}_{1p} \\ \widetilde{A}_{21} & \cdots & \cdots & \vdots \\ \vdots & \cdots & \cdots & \vdots \\ \widetilde{A}_{p1} & \cdots & \cdots & A_{pp} \end{bmatrix}, \tag{3}$$

where $A_{ij} \in \mathbb{R}^{t_i \times t_j}$ and most of the off-diagonal blocks A_{ij} are approximated by low-rank products $\widetilde{A}_{ij} = X_{ij} Y_{ij}^T$ with $X_{ij} \in \mathbb{R}^{t_i \times r_{ij}}$ and $Y_{ij} \in \mathbb{R}^{t_j \times r_{ij}}$. In practice the ranks r_{ij} are computed such that $\|\widetilde{A}_{ij} - A_{ij}\| \leq \varepsilon \|A\|$, which yields a BLR approximation satisfying a relative accuracy of order ε, $\|\widetilde{A} - A\| \leq c\varepsilon \|A\|$, where c is a constant that depends on the choice of norm (for example, for the Frobenius norm, $c = p$). If the rank r_{ij} is small enough so that $r_{ij}(t_i + t_j) < t_i t_j$, storing the low-rank factors X_{ij} and Y_{ij} requires less entries than storing the full block A_{ij}, and A_{ij} is referred to as a low-rank block. Otherwise, it is more efficient to keep A_{ij} as is and it is referred to as a full-rank block.

Hereinafter, for the simplicity of the discussion and without loss of generality, we assume that all blocks have the same size $t_i = t_j = t$ (and thus $m = pt$) and all low-rank blocks have the same rank $r_{ij} = r$. Assume that there are at most q full-rank blocks on any block-row or block-column (note that $q \geq 1$ since the diagonal blocks are always full-rank). Then the entire BLR matrix requires storing

$$pqt^2 + 2p^2tr = qtm + 2m^2r/t \tag{4}$$

entries. By taking the block size $t = \sqrt{2mr/q}$, (4) attains its minimal value $\sqrt{2qrm^3}$. This shows two things: the block size t should increase with both m and r, and the optimal BLR storage is proportional to $m^{3/2}$ instead of m^2 for the full uncompressed matrix. Importantly, in several applications such as the solution of certain classes of discretized partial differential equations, both q and r can be shown to be small constants independent of m [5, 11, 12]. Therefore, BLR compression allows for an asymptotic complexity reduction, which means that the larger the matrix, the larger the storage gain.

BLR matrices have been particularly successful to accelerate sparse direct solvers: the LU factorization can be computed with reduced storage and flops complexities [5], and thus improved performance [6], with a backward error controlled by ε [29]. The cost of the triangular solve phase is proportional to the size of the LU factors and is thus also reduced with BLR approximations. In our mixed precision memory accessor context, we are especially interested in accelerating this BLR triangular solve operation, which can be the bottleneck in some applications requiring many solves, such as when BLR is used as a preconditioner.

6.2 Memory accessor for BLR triangular solve

This section introduces a memory accessor approach for the BLR triangular solve, that is, a triangular solve with a blocked matrix where the off-diagonal blocks are low-rank.

At first sight, extending Algorithm 2 to BLR matrices may seem straightforward: a naive implementation would be to set $b = t$ (that is, use the same block size for the BLR approximation and the accessor) and, whenever a low-rank block must be accessed, load its low-rank factors into the fast memory and convert them. However, the BLR block size t, while under our control, must be set to a sufficiently large value to achieve high compression rates. As explained in the previous section, this is because t should increase with the matrix size m; in practice, even for medium-sized matrices, BLR block sizes of order 500 are typically necessary to obtain good performance [5, 6]. Based on the experiments with the dense triangular solve of the previous sections, this naive approach would therefore not be efficient since the memory accessor requires much smaller block sizes that fit into the fast memory.

Thus, the challenge is to ensure that the loaded blocks fit into the fast memory of size M. Importantly, the difficulty to satisfy this condition comes mainly from the full-rank blocks. Indeed, a low-rank block only consists of $2rt$ entries and, moreover, we can easily load its X and Y^T factors separately, which yields the condition $rt \leq M$. In contrast, loading a full-rank block would require the condition $t^2 \leq M$ (or $t^2/2 \leq M$ in the case of the diagonal blocks, which are triangular). Table 6.1 reports the maximum value of t that satisfies these conditions, depending on the storage type and the block type (low-rank with various compression ratios t/r, diagonal, or full-rank off-diagonal). For the low-rank blocks, the table shows that if the compression ratio t/r is large enough, the condition $rt \leq M$ is met even for large BLR block sizes t. In contrast, the conditions $t^2/2 \leq M$ for diagonal blocks and especially $t^2 \leq M$ for off-diagonal full-rank blocks are much more restrictive and limit the maximum BLR block size that the memory accessor can handle efficiently. Note that blocks with $r = t/2$ are not really low-rank: storing them in low-rank form costs $2tr = t^2$ entries, the same as storing them in full-rank form.

Table 6.1. Maximal values of t for which a given $t \times t$ block can be loaded and converted to fp64 precision while fitting in a fast memory of size M, depending on its storage precision and its type: low-rank of rank r, diagonal, or off-diagonal full-rank. We have taken $M = 1328 \times 1024$, which corresponds to the size in bytes of the L1 and L2 caches of the machine (see Table 3.1).

storage precision	maximum value of t				
	Low-rank blocks			Full-rank blocks	
	$r = t/2$	$r = t/4$	$r = t/8$	Diagonal	Off-diagonal
fp16	522	738	1043	522	369
fp24	497	703	994	497	352
fp32	476	673	952	476	337
fp40	457	647	915	457	323
fp48	441	623	882	441	312
fp56	426	602	852	426	302

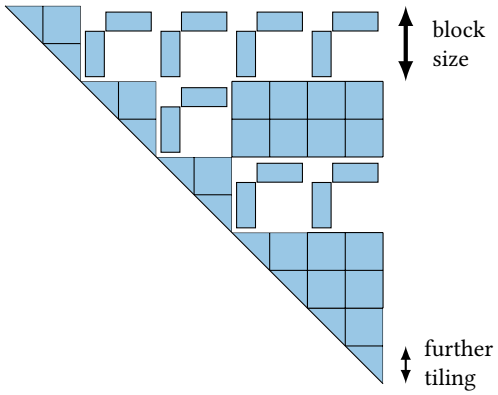

Fig. 6.1. Full-rank $t \times t$ blocks are further partitioned into $b \times b$ subblocks.

To overcome this issue, one solution is to further partition the full-rank blocks into smaller subblocks of size $b \times b$. This allows for loading the $t \times t$ full-rank blocks $b \times b$ subblocks at a time, where b can be chosen to satisfy $b^2 \leq M$. For the diagonal blocks, with which we must perform a `trsv`, we can then rely on the dense memory accessor approach presented previously, that is, we can use Algorithm 2. For the off-diagonal full-rank blocks, with which we must perform a `gemv`, we can implement a block memory accessor–based `gemv` in the same spirit as the `trsv` one. For the sake of conciseness, we omit the description of this algorithm since it is very similar to Algorithm 2. The subblocking approach of full-rank blocks is illustrated in Figure 6.1. We expect an overall performance improvement through this mechanism but it should be noted that it will lead to BLAS calls that process smaller number of elements, which could reduce their efficiency. The potential efficiency loss should not, however, outweigh the overall performance improvement.

We summarize our approach in Algorithm 3, which is an adaptation of Algorithm 2 to handle BLR matrices. The algorithm requires a workspace B in precision u_{high}, which is used to store the upcasted X and Y^T factors (of size tr) of the off-diagonal blocks, as well as the $b \times b$ subblocks of the full-rank blocks. Hence B must be of size $\max(tr, b^2)$. We also require an r-vector z in precision u_{high} to store the intermediate result $Y_{ji}^T x_i$. Note that, for the sake of simplicity, Algorithm 3 assumes that every off-diagonal block is low-rank.

Algorithm 3 Mixed precision BLR triangular solve with a block memory accessor.

Input: U, an $m \times m$ upper triangular matrix stored in precision u_{low} and partitioned into $t \times t$ blocks U_{ij}, with $m = pt$; if $i \neq j$, $U_{ij} = X_{ij} Y_{ij}^T$ is a low-rank block of rank r; the diagonal blocks U_{jj} are full-rank and partitioned into $b \times b$ subblocks; y, an m-vector stored in precision u_{high}; B, a workspace of size $\max(tr, b^2)$ stored in precision u_{high}; z, an r-vector stored in precision u_{high}.
Output: x, an m-vector stored in precision u_{high}, solution to $Ux = y$.

```
Initialize x = y.
for j = p, ..., 1 do
    Solve U_jj x_j = x_j using Algorithm 2 with block size b.
    for i = j − 1, ..., 1 do
        Read Y_ij^T, upcast it to precision u_high, and store it in B.
        Compute z = B x_j in precision u_high using gemv.
        Read X_ij, upcast it to precision u_high, and store it in B.
        Compute x_i = x_i − B z in precision u_high using gemv.
    end for
end for
```

6.3 Performance evaluation with BLR matrices

Figure 6.2 evaluates the performance of Algorithm 3 depending on various parameters. In all cases, we use a triangular BLR matrix with block size t and where all off-diagonal blocks are low-rank of rank r. For each plot, the BLR block size t varies on the x-axis, and lines of different colors correspond to a different storage precision types. The top, middle, and bottom plots consider different BLR compression ratios: $r = t/2$, $t/4$, and $t/8$, respectively; these correspond to the ratios considered in Table 6.1. Finally, we also assess the impact of further partitioning the full-rank blocks (which, in this experiment, only consist of the diagonal blocks): in the left plots we do not use any subblocking, whereas in the right plots the diagonal blocks are partitioned into $t/4 \times t/4$ subblocks. As in the experiments of the previous sections, we are still using a tree parallelism configuration with each core solving an independent system of order $m = 4096$ with a sequential implementation of Algorithm 3.

We can draw several interesting conclusions from Figure 6.2. First of all, we recover the conclusion of the previous experiments regarding the effect of the storage type: types of smaller size lead to higher performance, provided the block size belongs to a suitable interval. Second, comparing the left and right plots confirms that subblocking the diagonal full-rank blocks significantly extends the range of block sizes for which performance is satisfactory. This is a key result since, as previously mentioned, we cannot afford to choose the block size t only based on the memory accessor performance; in practice it is necessary to use large BLR block sizes to attain optimal compression rates. Moreover, comparing the top, middle, and bottom plots, we can see that the impact of subblocking the diagonal block becomes much stronger when the BLR compression ratio increases. Indeed, in the case $r = t/2$ (top plots), which corresponds to no BLR compression (since low-rank blocks require the same storage as if they were stored as full-rank), the subblocking of the diagonal blocks has almost no impact. This is expected since in this case the size of the X and Y^T factors, $tr = t^2/2$, is the same as the size of a diagonal block (since it is triangular). In contrast, in the middle and especially in the bottom plots, subblocking the diagonal blocks has a much stronger impact. For example, with fp16 as storage precision, subblocking extends the maximal block size t that maintains good performance from about 250 to 750. This experiment therefore confirms that subblocking is crucial to handle large BLR block sizes, and qualitatively supports the performance model used in Table 6.1. As in all previous experiments, the actual values are smaller in practice, but our model correctly captures the effect of the BLR compression on the performance.

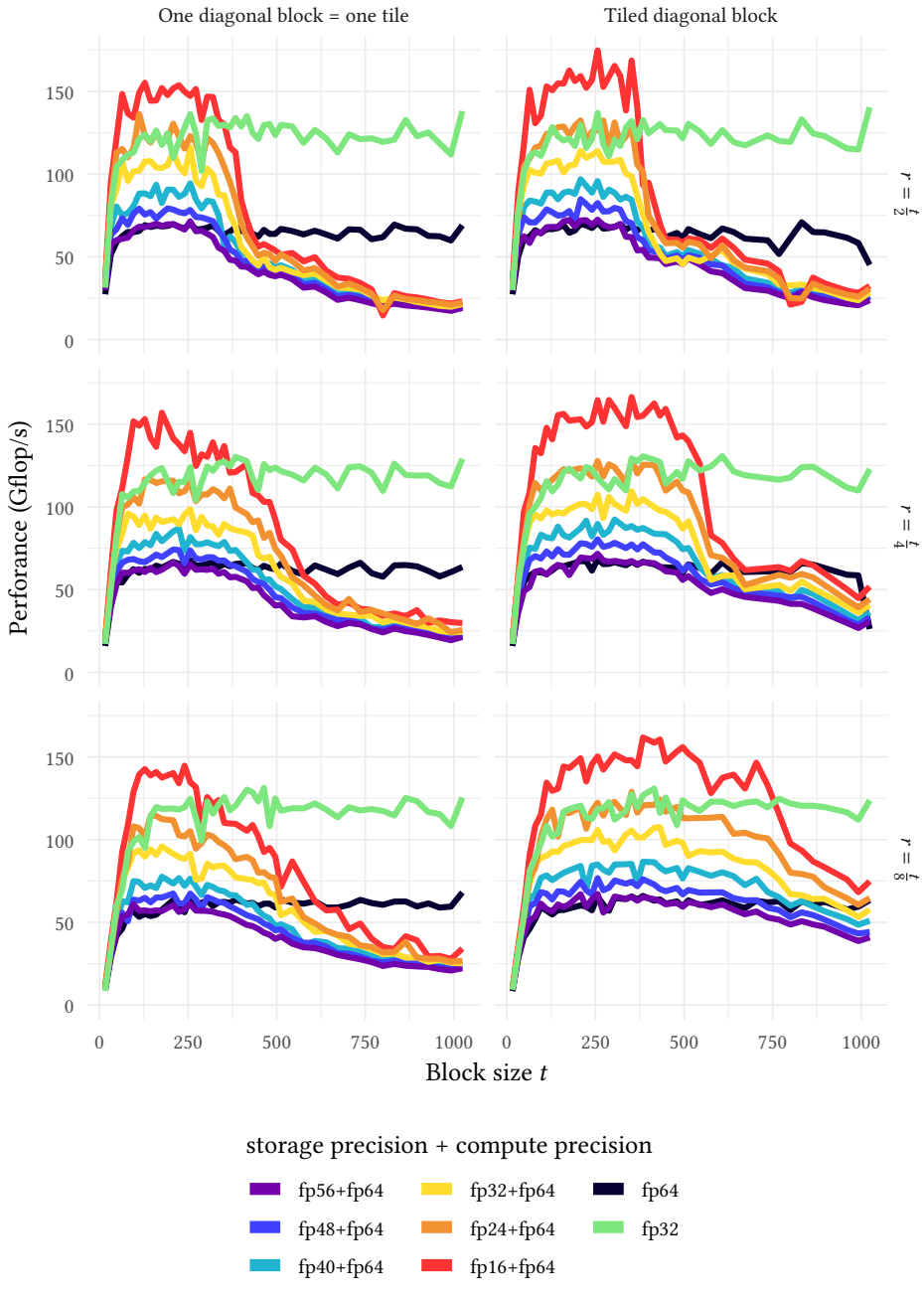

Fig. 6.2. Performance of the BLR triangular solve with custom types and tree parallelism, depending on the BLR compression ratio and on whether the full-rank blocks are further partitioned into subblocks or not.

7 Parallel triangular solve

All previous experiments have focused on tree parallelism configurations, where each thread solves an independent system of medium size using a sequential implementation of the memory accessor; as explained in section 3.2, this corresponds to the typical workload towards the bottom of the elimination tree in sparse direct solvers. In this section, we now turn to a node+tree parallelism configuration, corresponding to the top of the elimination tree: we consider a few independent systems of large size, each using a parallel implementation of the memory accessor. We first discuss how to parallelize Algorithm 2 in section 7.1; we consider an OpenMP parallelization, comparing fork-join and task-based approaches. Then, we evaluate the performance of this parallel memory accessor in section 7.2.

7.1 Parallelization of Algorithm 2

The goal of this section is to parallelize Algorithm 2 from section 3.1. Parallelizing this algorithm is essentially a matter of respecting the dependencies of the computation. For example, the updates $x_i = x_i - Bx_j$ are all independent and so a very simple parallelization strategy is to use an `omp for` directive on the inner `for` loop (on i). This strategy is usually referred to as a fork–join approach, because at each step j of the outer loop, the threads are forked over the inner loop, before joining again for the triangular solve with the diagonal block U_{jj}. This fork–join approach is therefore not optimal, because it requires synchronization between all threads at each outer step j (and so load unbalance will lead to idle threads), and because the diagonal triangular solve is sequential (all threads except one are idle during it). Importantly, we can expect idle time to have a stronger impact on the memory accessor performance than on the standard uniform precision solve. This is because idle threads do not saturate the memory bandwidth and thus the computation is less likely to be memory bound.

An alternative parallelization strategy is to follow more closely the dependencies of the computation by using the `omp task` directive. Such a task-based approach allows for a more dynamic parallel execution and reduces the idle time. Indeed, with this approach the tasks processing the diagonal triangular solves (`trsv`) and the tasks processing the off-diagonal updates (`gemv`) can be intertwined. This is illustrated in Figure 7.1 (left), which depicts the dependencies of a tiled dense triangular solve. Reducing idle time allows for better saturating the memory bandwidth and so we expect this task-based approach to better take advantage of the memory accessor.

However, one issue that one must pay attention to when using task-based approaches is the task generation and scheduling overhead. If the granularity of the tasks (that is, the amount of work performed by each task) is too small, this overhead can become the limiting factor on the performance. This risk is considerable in our context since we are dealing with BLAS-2 operations that perform relatively little work. One simple solution would be to increase the task granularity by increasing the block size. Importantly, in our context, this solution is not satisfactory because, as discussed in the previous sections, the memory accessor requires small block sizes to be efficient. Therefore, we adopt a different solution to increase task granularity: we batch tasks together so that a single thread executes multiple tasks together. This can be implemented by duplicating the inner loop so that the first β_{fine} iterations process individual tiles and the subsequent iterations process batches of β_{coarse} tiles as depicted in Figure 7.1 (right).

7.2 Performance evaluation of the parallel memory accessor

We now evaluate the performance of memory accessor in a node+tree parallelism configuration: we spawn 4 MPI ranks and each of them solves its own independent system of order $m = 32768$, using a parallel implementation of Algorithm 2 that exploits 16 threads. In total, we are thus using the full $4 \times 16 = 64$ cores of the machine. Figure 7.2

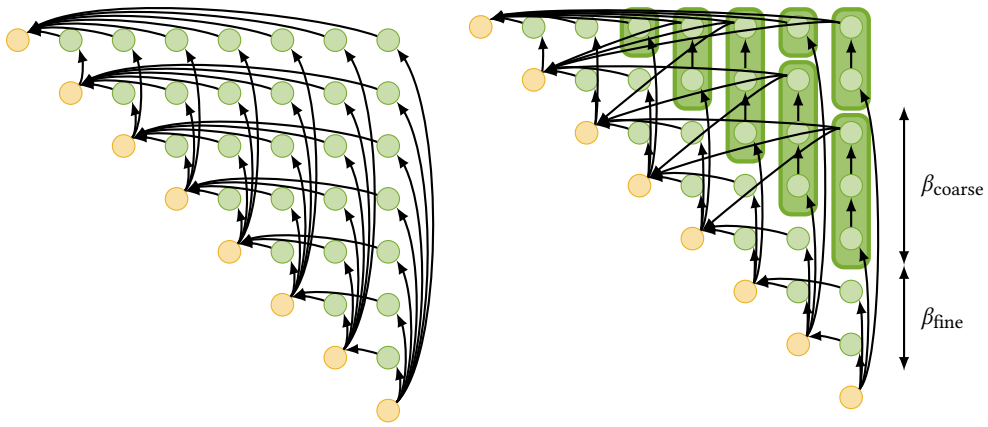

Fig. 7.1. Dependencies in a tiled dense triangular solve, and possible task-based parallelization approaches. On the left, each tile is associated to an individual task, leading to small task granularity. Rather than increasing the tile size, task granularity can be increased by batching tasks together, as illustrated on the right.

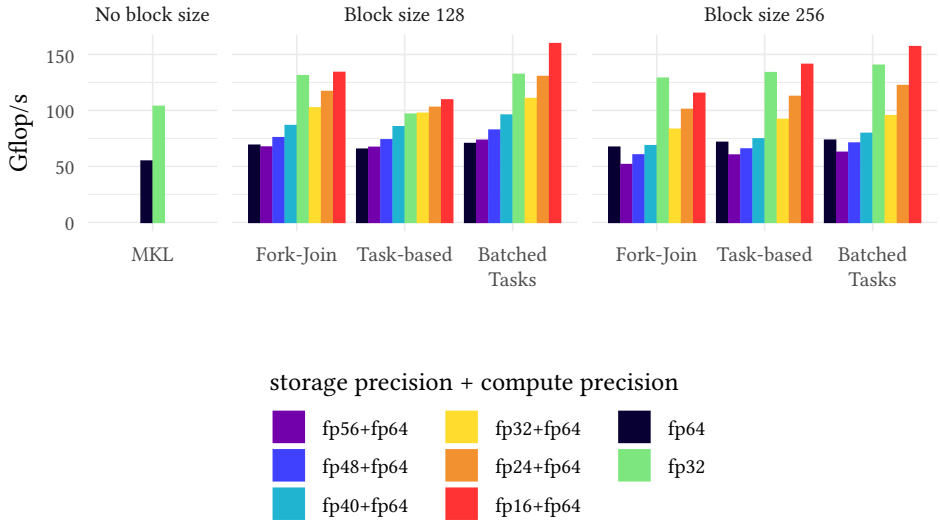

Fig. 7.2. Performance of the triangular solve with custom types and node+tree parallelism, depending on the parallelization strategy and on the block size ($b = 128$ or 256). A single compute node is used by four MPI processes. Each MPI process is solving a system of order 32,768 against a single vector.

compares the performance for various storage precision types, two block size values $b = 128$ or $b = 256$, and for three different parallelization strategies of Algorithm 2 described in the previous of section: the fork–join approach and the task-based approach with or without batching (for the batched version we have used $\beta_{\text{fine}} = \beta_{\text{coarse}} = 3$). In addition, we also provide as reference the performance of the multithreaded MKL `trsv` in fp64 or fp32 uniform precision.

First of all, the figure shows that the parallelization of `trsv` by MKL (leftmost black and green bars) is not very efficient. Our own tiled implementation of the uniform precision variants (black and green bars) significantly outperforms MKL, which certainly validates our implementation.

Let us first focus on the fork–join approach (first group of 8 bars after the MKL ones). As in previous experiments, we recover the good property that reducing the storage precision improves performance. Moreover, the absolute performance of the memory accessor variants is significantly better for a block size $b = 128$ (center plot) than for $b = 256$ (right plot), which is expected and consistent with the conclusions of previous experiments.

Next, let us turn to the task-based approach with no batching (second group of 8 bars). With $b = 256$ (right plot), this task-based approach achieves better performance than the fork–join one. Moreover, and quite interestingly, the performance improvement is noticeably better for the mixed precision memory accessor variants than for the uniform precision ones. For example, by switching from the fork–join to the task-based parallelization, the fp16+fp64 variant becomes more effiant than the uniform fp32 variant. However, as previously noted, the case $b = 256$ is not optimal for the memory accessor, for which $b = 128$ (center plot) should be preferred. Unfortunately, with this task-based approach, reducing the block size to $b = 128$ leads to a huge performance drop, to the point where it is actually less efficient than the fork–join approach. As discussed in the previous section, this is likely due to the task granularity being too small and leading to a high overhead.

This finally brings us to the task-based approach with batching (third group of 8 bars). The figure shows that batching successfully solves the issue of the task overhead and can achieve high performance even for the smaller block size $b = 128$. This almost does not benefit the uniform precision variants (black and green bars), which attain their optimal performance with $b = 256$, for which the task batching only slightly improves performance. However, task batching is really critical for the mixed precision memory accessor variants, which attain their optimal performance with $b = 128$. Overall, the best parallel memory accessor implementation is the task-based approach with batching and with a small block size $b = 128$. For this optimal implementation, the memory accessor successfully translates a lower storage precision into significant performance gains.

We conclude this section by mentioning that we have also implemented a parallel version of Algorithm 3 for BLR matrices. We draw the same conclusions as for dense systems with the fork–join approach. However, with task-based approaches (with or without batching), our implementation attains very poor performance; we did not investigate in depth the reasons for this behavior, and decide to rather focus our efforts on the integration of our approach within the MUMPS solver.

8 Preliminary results with MUMPS

In this section we provide some preliminary results with the MUMPS solver [6, 7]. MUMPS provides several approximate computing methods, including BLR compression [6], mixed precision iterative refinement [2], and adaptive precision BLR [3]. In this preliminary study we focus on using the memory accessor to accelerate the adaptive precision BLR triangular solve. In this context, the LU factors are compressed using the BLR format, and each low-rank block is further compressed by partitioning it into components stored in gradually lower precisions depending on the decay of its singular values [3]. As explained in section 2.3, such an adaptive precision algorithm can exploit a continuum of precisions; MUMPS supports the same types fp64, fp56, fp48, fp40, fp32, fp24, and fp16 as defined in Table 5.1.

MUMPS implements this approach to reduce the storage cost of the LU factors, and maintains the use of fp64 for the triangular solve. Before the integration of this work, this was achieved via a naive “memory accessor” which converted each adaptive precision low-rank block back to fp64, without any vectorized conversion or cache-aware strategies such

Table 8.1. Preliminary results with MUMPS (Queen_4147 matrix, BLR truncation threshold $\varepsilon = 3 \times 10^{-9}$).

	Compute precision	Storage precisions	Accessor method	Backward error	LU factor storage (GB)	Solve time (ms)
Full-rank	fp64	fp64	—	10^{-16}	112	871
BLR	fp64	fp64	—	10^{-11}	57	505
BLR	fp64	fp64, fp32	Naive	10^{-11}	44	922
BLR	fp64	fp64, fp32	This work	10^{-11}	44	509
BLR	fp64	fp64, fp56, …, fp16	Naive	10^{-11}	37	1089
BLR	fp64	fp64, fp56, …, fp16	This work	10^{-11}	37	479

as subblocking. We have implemented a much more optimized memory accessor based on the lessons learned in the previous sections of this work. In particular, we employ the efficient conversion routines from custom types described in section 5.2, and we use subblocking for the off-diagonal full-rank blocks as described in section 6.2. Note that we do not use subblocking for diagonal blocks, and the triangular solve is parallelized with a fork–join approach rather than a task-based one, so further optimizations are possible. We leave them for future work as they would require complex and intrusive modifications of the MUMPS data structures and kernels that are outside of the scope of this preliminary study.

We present our preliminary results in Table 8.1, taking matrix Queen_4147[6] from the SuiteSparse matrix collection [16] as an example. We have set the BLR truncation threshold to $\varepsilon = 3 \times 10^{-9}$. We compare several approaches: the fp64 full-rank and BLR (without any adaptive precision) baselines, and four variants of the adaptive precision BLR solver depending on the storage precisions (either only the natively supported fp64 and fp32 types, or the full continuum of types including the custom ones) and on the memory accessor implementation (naive one originally provided by MUMPS, or the new one based on this work).

First of all, the table shows that the adaptive precision BLR variants achieve a normwise backward error $\frac{\|Ax-b\|}{\|A\|\|x\|+\|b\|}$ of the same order 10^{-11} as the fp64 BLR baseline, as guaranteed by the method [3]. Second, the storage cost of the uniform fp64 BLR solver (which is, at 57 GB, already much lower than the 112 GB of the full-rank solver) can be significantly further reduced with the use of adaptive precision. Using only fp64 and fp32 as storage precisions leads to a 44 GB storage cost, whereas using the full continuum of precisions further reduces this cost to 37 GB. Third and most importantly, with the naive accessor originally implemented in MUMPS, these storage reductions do not lead to corresponding time reductions but are on the contrary achieved at the expense of time. With our new accessor from this work, we obtain significant speedups (greater than 2×), which brings the adaptive precision solve back to being as fast as the uniform precision one, or even slightly faster with the full continuum of precisions. These positive preliminary results confirm the potential of memory accessors to minimize the storage cost of the solver while maintaining its efficiency.

[6] https://sparse.tamu.edu/Janna/Queen_4147

9 Conclusions

We have proposed a memory accessor approach for accelerating memory-bound triangular solves, based on a mixed precision decoupling of the storage (low) precision and the compute (high) precision. Our approach differs from previous ones by proposing a BLAS-based block accessor that loads and converts the matrix by blocks and relies on BLAS kernels for the computations (Algorithm 2). This approach is minimally intrusive and easily portable, as it only requires wrapping the uniform precision BLAS kernels with conversion routines. Moreover, it can be highly efficient provided that the block workspace fits into the fast memory and that the conversion from the storage to the compute precision is efficient.

We have performed a thorough study of this approach by incrementally complexifying it to incorporate support for custom floating-point types, compressed BLR matrix representations, and parallelization. We can draw the following key lessons from this study.

- The matrix should be represented using a tiled storage layout. With such a layout, right- and left-looking algorithms perform comparably.
- The block size should be chosen sufficienly small so that the block workspace fits into the fast memory and sufficiently large so that the BLAS calls are efficient.
- Custom floating-point storage types can be efficiently handled by truncating some mantissa bytes from a standard format. For truncating an fp64 number to a type of size less than 32 bits, using fp32 as intermediate allows for reducing the exponent and gaining 3 bits of precision, without any significant performance penalty.
- The conversion from a custom type to a standard one can be efficiently vectorized using the AVX512-VBMI instruction set available on recent CPUs.
- The block size used for BLR matrices should be decoupled from the block size used for the accessor by further partitioning the full-rank blocks into smaller subblocks that fit into the fast memory.
- The optimal parallelization strategy is a task-based approach with task batching, which allows to define tasks with sufficiently small blocks that fit into the fast memory, while limiting the task generation and scheduling overhead.
- With the above optimizations, the performance adequately matches the storage cost of the matrix, that is, reducing the storage precision increases the performance.

We have leveraged these conclusions to improve the MUMPS direct solver [6, 7], which provides an adaptive precision BLR compression method to reduce storage [3]. Preliminary performance results with this new memory accessor in MUMPS indicate significant speedups, which allows for optimizing storage while maintaining efficiency. This represents a first major application of our accessor approach. A second possible application, that we will tackle in future work, consists in using it in mixed precision iterative refinement [2]. Moreover, while this study has focused on the triangular solve step of direct methods, it could be possible to extend it to the factorization step, provided that the latter remains memory bound, as might be the case, for example, with BLR compression.

Acknowledgments

This work was partially supported by the MixHPC (ANR-23-CE46-0005-01), NumPEx ExaMA (ANR-22-EXNU-0002), and InterFLOP (ANR-20-CE46-0009) projects of the French National Agency for Research (ANR).

References

[1] Emmanuel Agullo, Franck Cappello, Sheng Di, Luc Giraud, Xin Liang, and Nick Schenkels. 2020. *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. https://hal.inria.fr/hal-02572910

[2] Patrick Amestoy, Alfredo Buttari, Nicholas J. Higham, Jean-Yves L'Excellent, Theo Mary, and Bastien Vieublé. 2023. Combining Sparse Approximate Factorizations with Mixed-precision Iterative Refinement. *ACM Trans. Math. Software* 49, 1 (March 2023), 4:1–4:29. https://doi.org/10.1145/3582493

[3] P. R. Amestoy, O. Boiteau, A. Buttari, M. Gerest, F. Jézéquel, J.-Y. L'Excellent, and T. Mary. 2022. Mixed Precision Low Rank Approximations and their Application to Block Low Rank LU Factorization. *IMA J. Numer. Anal.* 43, 4 (2022), 2198–2227. https://doi.org/10.1093/imanum/drac037

[4] P. R. Amestoy, A. Buttari, N. J. Higham, J.-Y. L'Excellent, T. Mary, and B. Vieublé. 2024. Five-precision GMRES-based iterative refinement. *SIAM J. Matrix Anal. Appl.* 45, 1 (2024), 529–552. https://doi.org/10.1137/23M1549079

[5] P. R. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary. 2017. On the Complexity of the Block Low-Rank Multifrontal Factorization. *SIAM J. Sci. Comput.* 39, 4 (2017), A1710–A1740. https://doi.org/10.1137/16M1077192

[6] Patrick R. Amestoy, Alfredo Buttari, Jean-Yves L'Excellent, and Theo Mary. 2019. Performance and Scalability of the Block Low-Rank Multifrontal Factorization on Multicore Architectures. *ACM Trans. Math. Software* 45, 1 (2019), 2:1–2:26. https://doi.org/10.1145/3242094

[7] Patrick R Amestoy, Iain S Duff, Jean-Yves L'Excellent, and Jacko Koster. 2001. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J. Matrix Anal. Appl.* 23, 1 (2001), 15–41.

[8] Andrew Anderson and David Gregg. 2016. Vectorization of Multibyte Floating Point Data Formats. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation* (Haifa, Israel) *(PACT '16)*. Association for Computing Machinery, New York, NY, USA, 363–372. https://doi.org/10.1145/2967938.2967966

[9] Hartwig Anzt, Jack Dongarra, Goran Flegar, Nicholas J. Higham, and Enrique S. Quintana-Ortí. 2019. Adaptive Precision in Block-Jacobi Preconditioning for Iterative Sparse Linear System Solvers. *Concurrency Computat. Pract. Exper.* 31, 6 (2019), e4460. https://doi.org/10.1002/cpe.4460

[10] Hartwig Anzt, Goran Flegar, Thomas Grützmacher, and Enrique S. Quintana-Ortí. 2019. Toward a Modular Precision Ecosystem for High-Performance Computing. *Int. J. High Perform. Comput. Appl.* 33, 6 (2019), 1069–1078. https://doi.org/10.1177/1094342019846547

[11] Mario Bebendorf. 2008. *Hierarchical Matrices: A Means to Efficiently Solve Elliptic Boundary Value Problems.* Lecture Notes in Computational Science and Engineering (LNCSE), Vol. 63. Springer-Verlag.

[12] Mario Bebendorf and Wolfgang Hackbusch. 2003. Existence of $\mathcal{H}$-matrix approximants to the inverse FE-matrix of elliptic operators with L^∞-coefficients. *Numer. Math.* 95, 1 (2003), 1–28.

[13] Erin Carson and Nicholas J. Higham. 2017. A New Analysis of Iterative Refinement and its Application to Accurate Solution of Ill-Conditioned Sparse Linear Systems. *SIAM J. Sci. Comput.* 39, 6 (2017), A2834–A2856. https://doi.org/10.1137/17M1122918

[14] Erin Carson and Nicholas J. Higham. 2018. Accelerating the Solution of Linear Systems by Iterative Refinement in Three Precisions. *SIAM J. Sci. Comput.* 40, 2 (2018), A817–A847. https://doi.org/10.1137/17M1140819

[15] Michael P. Connolly, Nicholas J. Higham, and Theo Mary. 2021. Stochastic Rounding and Its Probabilistic Backward Error Analysis. *SIAM J. Sci. Comput.* 43, 1 (Jan. 2021), A566–A585. https://doi.org/10.1137/20m1334796

[16] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Software* 38, 1 (2011), 1:1–1:25. https://doi.org/10.1145/2049662.2049663

[17] I. S. Duff, A. M. Erisman, and J. K. Reid. 2017. *Direct Methods for Sparse Matrices.* Oxford University Press. https://doi.org/10.1093/acprof:oso/9780198508380.001.0001

[18] Massimiliano Fasi, Nicholas J. Higham, Florent Lopez, Theo Mary, and Mantas Mikaitis. 2023. Matrix Multiplication in Multiword Arithmetic: Error Analysis and Application to GPU Tensor Cores. *SIAM J. Sci. Comput.* 45, 1 (Feb. 2023), C1–C19. https://doi.org/10.1137/21m1465032

[19] Massimiliano Fasi and Mantas Mikaitis. 2023. CPFloat: A C Library for Simulating Low-precision Arithmetic. *ACM Trans. Math. Softw.* 49, 2, Article 18 (June 2023), 32 pages. https://doi.org/10.1145/3585515

[20] Goran Flegar, Hartwig Anzt, Terry Cojean, and Enrique S. Quintana-Ortí. 2021. Adaptive Precision Block-Jacobi for High Performance Preconditioning in the Ginkgo Linear Algebra Software. *ACM Trans. Math. Software* 47, 2 (April 2021), 1–28. https://doi.org/10.1145/3441850

[21] Goran Flegar, Florian Scheidegger, Vedran Novaković, Giovani Mariani, Andrés E Tom´ s, A Cristiano I Malossi, and Enrique S Quintana-Ortí. 2019. FloatX: A C++ Library for Customized Floating-Point Arithmetic. *ACM Transactions on Mathematical Software (TOMS)* 45, 4, Article 40 (Dec. 2019), 23 pages. https://doi.org/10.1145/3368086

[22] Luc Giraud, Serge Gratton, and Julien Langou. 2007. Convergence in Backward Error of Relaxed GMRES. *SIAM J. Sci. Comput.* 29, 2 (2007), 710–728. https://doi.org/10.1137/040608416

[23] Stef Graillat, Fabienne Jézéquel, Theo Mary, and Roméo Molina. 2024. Adaptive precision sparse matrix-vector product and its application to Krylov solvers. *SIAM J. Sci. Comput.* 46, 1 (2024), C30–C56. https://doi.org/10.1137/22M1522619

[24] Stef Graillat, Fabienne Jézéquel, Theo Mary, Roméo Molina, and Daichi Mukunoki. 2024. Reduced-Precision and Reduced-Exponent Formats for Accelerating Adaptive Precision Sparse Matrix-Vector Product. In *Euro-Par 2024: Parallel Processing*. Springer Nature Switzerland, Cham, 17–30. https://doi.org/10.1007/978-3-031-69583-4_2

[25] Thomas Grützmacher, Hartwig Anzt, and Enrique S. Quintana-Ortí. 2021. Using Ginkgo's Memory Accessor for Improving the Accuracy of Memory-Bound Low Precision BLAS. *Software—Practice and Experience* (Oct. 2021). https://doi.org/10.1002/spe.3041

[26] Gustafson and Yonemoto. 2017. Beating Floating Point at its Own Game: Posit Arithmetic. *Supercomput. Front. Innov.: Int. J.* 4, 2 (June 2017), 71–86. https://doi.org/10.14529/jsfi170206

[27] Nicholas J. Higham and Theo Mary. 2019. A New Approach to Probabilistic Rounding Error Analysis. *SIAM J. Sci. Comput.* 41, 5 (2019), A2815–A2835. https://doi.org/10.1137/18M1226312

[28] Nicholas J. Higham and Theo Mary. 2020. Sharper Probabilistic Backward Error Analysis for Basic Linear Algebra Kernels with Random Data. *SIAM J. Sci. Comput.* 42, 5 (2020), A3427–A3446. https://doi.org/10.1137/20M1314355

[29] Nicholas J. Higham and Theo Mary. 2021. Solving Block Low-Rank Linear Systems by LU Factorization is Numerically Stable. *IMA J. Numer. Anal.* 42, 2 (2021), 951–980. https://doi.org/10.1093/imanum/drab020

[30] Nicholas J. Higham and Theo Mary. 2022. Mixed Precision Algorithms in Numerical Linear Algebra. *Acta Numerica* 31 (May 2022), 347–414. https://doi.org/10.1017/s0962492922000022

[31] Nicholas J. Higham and Srikara Pranesh. 2019. Simulating Low Precision Floating-Point Arithmetic. *SIAM J. Sci. Comput.* 41, 5 (2019), C585–C602. https://doi.org/10.1137/19M1251308

[32] Intel. 2024. *Intel 64 and IA-32 Architectures Software Developer's Manual.* Intel. Available at https://cdrdv2.intel.com/v1/dl/getContent/671200.

[33] Ronald Kriemann. 2023. Hierarchical Lowrank Arithmetic with Binary Compression. arXiv:2308.10960 [cs.MS] https://arxiv.org/abs/2308.10960

[34] Ronald Kriemann. 2024. Performance of H-Matrix-Vector Multiplication with Floating Point Compression. arXiv:2405.03456 [cs.DC] https://arxiv.org/abs/2405.03456

[35] Julie Langou, Julien Langou, Piotr Luszczek, Jakub Kurzak, Alfredo Buttari, and Jack Dongarra. 2006. Exploiting the Performance of 32 bit Floating Point Arithmetic in Obtaining 64 bit Accuracy (Revisiting Iterative Refinement for Linear Systems). In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. IEEE. https://doi.org/10.1109/SC.2006.30

[36] Daichi Mukunoki and Toshiyuki Imamura. 2016. Reduced-Precision Floating-Point Formats on GPUs for High Performance and Energy Efficient Computation. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. 144–145. https://doi.org/10.1109/CLUSTER.2016.77

[37] Daichi Mukunoki, Masatoshi Kawai, and Toshiyuki Imamura. 2023. Sparse matrix-vector multiplication with reduced-precision memory accessor. In *2023 IEEE 16th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. IEEE, 608–615. https://doi.org/10.1109/MCSoC60832.2023.00094

[38] Yousef Saad. 2003. *Iterative Methods for Sparse Linear Systems* (second ed.). Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898718003

[39] H. Vandierendonck. 2022. Software-defined floating-point number formats and their application to graph processing. In *Proceedings of the 36th ACM International Conference on Supercomputing* (Virtual Event) *(ICS '22)*. Association for Computing Machinery, New York, NY, USA, Article 8, 17 pages. https://doi.org/10.1145/3524059.3532360

Received NA