Towards Reproducible Blocked LU Factorization

Roman Iakymchuk*, Enrique S. Quintana-Ortí†, Erwin Laure* and Stef Graillat‡
*KTH Royal Institute of Technology, CSC, CST/PDC, 100 44 Stockholm, Sweden
Emails: {riakymch,erwinl}@kth.se
†Universidad Jaime I, 12.071-Castellón, Spain
Email: quintana@uji.es
‡Sorbonne Universités, UPMC Univ Paris 06, CNRS, UMR 7606, LIP6, Paris, France
Email: stef.graillat@lip6.fr

***Abstract*—In this article, we address the problem of reproducibility of the blocked LU factorization on GPUs due to cancellations and rounding errors when dealing with floating-point arithmetic. Thanks to the hierarchical structure of linear algebra libraries, the computations carried within this operation can be expressed in terms of the Level-3 BLAS routines as well as the unblocked variant of the factorization, while the latter is correspondingly built upon the Level-1/2 BLAS kernels. In addition, we strengthen numerical stability of the blocked LU factorization via partial row pivoting. Therefore, we propose a double-layer bottom-up approach for ensuring reproducibility of the blocked LU factorization and provide experimental results for its underlying blocks.**

***Keywords*-Reproducibility; LU factorization; BLAS; long accumulator; floating-point expansion; error-free transformation; GPUs.**

I. Introduction

The first Exascale computers, delivering 10^{18} operations per second, are expected to arrive by 2023, offering scientists the opportunity to perform simulations (related, e.g., to space weather forecast, human brain, supernova, etc.) at extreme scales. In order to efficiently utilize such systems, the runtimes in charge of orchestrating those simulations will employ various strategies to reduce the communication overhead as well as to equally and efficiently distribute computations and the associated data. However, those strategies pursuing excellent performance scaling may also impair the *accuracy* and *reproducibility*[1] of the floating-point arithmetic results [1], [2]. The bottom reason is that the order of operations impacts the accuracy of the final result, especially when there is a change in the thread execution order (dynamic scheduling), reduction trees, blocking, partitioning, instructions sets, etc. This narrows to the use of finite-precision computer arithmetic [3], [4] and, therefore, to floating-point operations that are commutative, but non-associative due to rounding errors. For instance, $\oplus$ the addition in `binary64` floating-point arithmetic, $(-1 \oplus 1) \oplus 2^{-53} \neq -1 \oplus (1 \oplus 2^{-53})$ since $(-1 \oplus 1) \oplus 2^{-53} = 2^{-53}$ and $-1 \oplus (1 \oplus 2^{-53}) = 0$. Thus, the usage of various optimization strategies and data access/partitioning patterns during the computation may potentially lead to differences in the final results.

The IEEE 754 standard [5], created in 1985 and then revised in 2008 (IEEE 754-2008), has led to considerable enhancements in the reliability of numerical computations by rigorously specifying the properties of floating-point arithmetic. This standard is now adopted by most processors, thus leading to a much better portability of numerical applications. The IEEE 754-2008 standard also contains the reproducibility clause that forwards the reproducibility issue to language standards, which then enforce all implementations of the language to produce the same result. Additionally, the IEEE standard introduced some suggestions to achieve reproducible results. Emerging attention to reproducibility strives to draw a more careful attention to the problem by the computer arithmetic community, leading to a potential inclusion of some mechanisms, which are under consideration, to assure numerical reproducibility of floating-point operations into the new version of the IEEE 754 standard due in 2018 [6].

Finding the solution of a linear system of equations often occurs in the large variety of scientific applications. The common practice engages the high-performance blocked LU factorization in this process. Despite the existence of various implementations of the blocked LU factorization, targeting an ample variety of architectures ranging from conventional CPUs to graphics processing units (GPUs), the accuracy and reproducibility of the produced results cannot be guaranteed. This is due to the non-associativity of floating-point operations, dynamic thread scheduling and concurrent execution on CPUs, as well as the non-determinism of warp scheduling on GPUs.

In this article, we aim to derive a reproducible algorithmic variant of the blocked LU factorization and provide the corresponding implementation on GPUs. Instead of developing this GPU implementation from scratch, we benefit from the modular and hierarchical structure of linear algebra libraries and, at first, construct and enhance reproducible OpenCL implementations of the corresponding underlying Basic Linear Algebra Subprograms (BLAS [7]) kernels.

[1]By accuracy, we mean the relative error between the exact result and the computed result. We define reproducibility as the ability to obtain a bit-wise identical floating-point result from multiple runs of the code on the same input data.

Proceeding in this manner, the blocked algorithmic variant of the LU factorization exhibits a double-layer structure:

- The first layer corresponds to the unblocked algorithmic variant with partial pivoting for stability that can be formulated in terms of the Level-1/2 BLAS kernels, namely the vector scaling (SCAL) and the rank-1 update of a matrix (GER). In addition, for pivoting we search for a maximum element in the column (MAX) and, if necessary, swap rows (SWAP).
- The second layer relies upon the Level-3 BLAS kernels for the matrix-matrix multiplication (GEMM) and the unit lower triangular system solve with multiple right-hand sides (TRSM).

We avoid rounding errors in SCAL and GER by carefully performing or reordering computations –preventing double rounding when the vector is scaled by the inverse of a diagonal element through computing this division during the scaling (INVSCAL), so that these routines yield both reproducible and correctly-rounded results. The remaining two operations (MAX and SWAP), which are involved in pivoting, are reproducible by nature. We extend our hierarchical approach [8], which leverages a long accumulator and error-free transformations (EFTs), to produce an exact dot product (EXDOT) by employing the `TwoProd` EFT [9] for the multiplication of two floating-point numbers. For EXTRSM and EXGEMM, we propose blocked variants that combine together high performance GPU kernels and EXDOT. EXTRSM relies upon small EXTRSM on diagonal blocks and EXGEMM on off-diagonal blocks. EXTRSM delivers reproducible, but not yet correctly rounded results. We outline a strategy for enhancing EXTRSM's accuracy up to achieving correctly-rounded results. In addition, we draw a strategy for improving performance of Level-3 BLAS routines, in particular of EXGEMM.

The paper is organized as follows. Section II reviews several aspects of computer arithmetic, in particular the floating-point expansion and the long accumulator. Section III presents the ExBLAS library with the required set of routines for algorithmic variants of both the unblocked and blocked LU factorizations. Those reproducible algorithmic variants are presented in Sections IV and V, accordingly. Finally, we evaluate our implementations in Section VI and draw conclusions in Section VII.

II. Floating-Point Arithmetic

Floating-point arithmetic consists in an approximating of real numbers with a significand, an exponent, and a sign:

$$x = \pm \underbrace{x_0.x_1 \dots x_{M-1}}_{mantissa} \times b^e, \quad 0 \le x_i \le b-1, \quad x_0 \neq 0,$$

where b is the basis (2 in our case), M is the precision, and e stands for the exponent that is bounded ($e_{\min} \le e \le e_{\max}$).

In this paper, we consider the `binary64` or double-precision format of the IEEE-754-2008 standard. The standard requires the basic arithmetic operations $(+, -, \times, /, \sqrt{})$ to be correctly rounded that is to say that the operations are performed as if the result was first computed with infinite precision and then rounded to the current floating-point format. In the sequel of the paper, we assume that the rounding-m ode is rounding-to-nearest. It means that the basic operations return the closest floating-point number to the exact result, breaking ties by rounding to the floating-point number with the even significand.

To increase the accuracy of floating-point operations, we will use two strategies in order to deal with rounding errors. The first solution computes the rounding error which occurred during basic floating-point operations (when possible) with error-free transformation and then uses floating-point expansions (unevaluated sum of several floating-point numbers with little overlapping), see Section II-A. The second solution exploits the finite range of exponents of floating-point numbers by storing every bit in a long vector of bits (long accumulator), see Section II-B.

A. Floating-Point Expansion

Floating-point expansion (FPE) makes it possible to increase the precision of the computations at a moderate cost especially for floating-point additions. FPE are represented by an unevaluated sum of p floating-point numbers whose components are ordered in magnitude with minimal overlap to cover a wide range of exponents. The algorithms for computing with FPE rely on the use of EFT for the addition (`TwoSum`, see Alg. 1 [10]) and, for the multiplication (`TwoProd`, see Alg. 2 [9]). Alg. 1 computes the addition r of two floating-point numbers a and b and the rounding error e such that r and e do not overlap and $a + b = r + e$. Similarly, `TwoProd` computes the product of two floating-point numbers a and b as well as the rounding error. For `TwoProd`, we use the fused-multiply-and-add (FMA) instruction to track the error that computes $a \cdot b - r$ with only one rounding at the end.

Algorithm 1: Error-free transformation for the sum of two floating-point numbers.

Function $[r, s] = \mathrm{TwoSum}(a, b)$
$r \leftarrow a + b$
$z \leftarrow r - a$
$s \leftarrow (a - (r - z)) + (b - z)$

Algorithm 2: Error-free transformation for the product of two floating-point numbers.

Function $[r, e]$ = `TwoProd`(a, b)
$r := a \cdot b$
$e :=$ FMA$(a, b, -r)$

Adding a floating-point number to an expansion of size p is an iterative process. The floating-point number is first added to the head of the expansion and the rounding error is next recovered as a floating-point number using the `TwoSum` EFT. The error is then recursively accumulated to the remainder of the expansion. As long as the dynamic range of the sum is lower than $2^{53\times p}$ for `binary64`, the FPE approach computes the accumulation of numbers without loss of accuracy.

The main advantage of FPEs is that they can be stored in registers (after being fetched) during the computations. Nevertheless, their accuracy may be insufficient for large sums or for floating-point numbers with significantly variations in magnitude. Moreover, the complexity of FPEs grows linearly with their size.

B. Long accumulator

Another way to increase the precision of the computation is to use a long fixed-point accumulator (superaccumulator). A fixed-point representation stores numbers using an integral part and a fractional part of fixed size, or equivalently as a scaled integer. A long accumulator can be seen as a projection from the input floating-point format that can represent every bit of information of this format and covers all the numbers in the range from the minimum representable floating-point value to the maximum value, independently of the sign. As an example, Kulisch [11] proposed to use a 4288-bit long accumulator for the exact dot product of two vectors composed of `binary64` numbers. Fig. 1 illustrates the error-free accumulation of floating-point input numbers in the long accumulator. The superaccumulator is a convenient way to compute the exact result of a large amount of floating-point numbers of arbitrary magnitude. The main drawbacks of the superaccumulator are its very large memory overhead and indirect memory accesses.

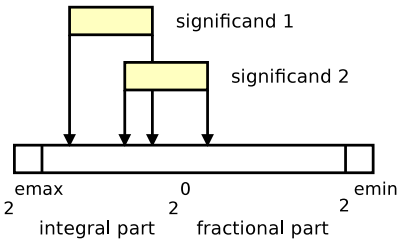


Figure 1: Long accumulator.

III. ExBLAS– Accurate and Reproducible BLAS

This section provides a brief overview of the prototype implementations of the Exact BLAS (ExBLAS) library routines [12] that are used within the studied unblocked and blocked LU factorizations. We begin with the parallel reduction and dot product that are two fundamental BLAS kernels. We then continue with the Level-1/2 BLAS routines, namely vector scaling and outer product, and show that reproducibility can be ensured by carefully rearranging arithmetic operations. We extend this approach to the Level-3 BLAS routines – such as the matrix multiplication and the triangular solve with multiple right-hand sides.

A. EXSUM: Exact Parallel Reduction

The parallel reduction is in the core of many BLAS routines. So, at first, we derive a multi-level approach for this operation, aiming to address various modern architectures with their complex multi-level memory structures. From one side, we want this approach to be fast to ensure compatible performance of the reproducible version of the parallel reduction. From the other side, we want to preserve every bit of information before the final rounding to the desired format, e.g. `binary64`, to assure reproducibility. To accomplish our goal, we combine together, tune, and extend to new architectures – like GPUs and Intel Xeon Phi co-processors – the existing solutions [8], [12]: the floating-point expansion and the long accumulator.

Algorithm 3: Floating-point expansion a of size p.

Function *ExpansionAccumulate(x)*
  for $i = 0 \rightarrow p - 1$ **do**
    $(a_i, x) := \mathrm{TwoSum}(a_i, x)$
  end
  if $x \neq 0$ **then**
    Superaccumulate(x)
  end

For accumulating floating-point numbers using FPE with the `TwoSum` EFT we rely upon Alg. 1. Since FPE occupies only few words of memory we assign them to each thread and split computations among those threads; to note, no sorting or reordering are required during the entire process. Thus, each thread crunches numbers assigned to it and sends back this accumulated result. Alg. 3 extends the classic FPE of size 2 to a variable size p ($p = 8$ is the large size we test) and introduces the superaccumulator when the accuracy of the FPE is not sufficient to store every bit of the result. To reduce the memory usage, these superaccumulators are shared among multiple threads; contention among these threads is handled via atomic operations. This local accumulation stage is then followed by the reduction of the superaccumulators within the work group of threads and, therefore, among the work groups. Finally, the global superaccumulator, which stores the result, is correctly rounded to the target floating-point format.

B. EXDOT: Exact Dot Product

The inner product or dot product of two vectors is another crucial fundamental BLAS operation. After deriving the exact parallel reduction, the remaining challenge to build the exact dot product [13] lays in the exact multiplication of two floating-point numbers. For that purpose, we utilize the

`TwoProd` EFT, see Alg. 2, that returns two values: the result and the error. Therefore, the EXDOT algorithm is based on the EXSUM algorithm and the `TwoProd` EFT: the accumulation of both the result and the error to the FPEs followed by the reductions of these FPEs and superaccumulators on various levels as in EXSUM.

C. EXSCAL *and* EXINVSCAL*: Exact Vector Scaling*

Scaling a vector x by a scalar α is rather a trivial operation as it does not induce any dependencies among the vector elements and requires only one operation to be performed per element ($x_i := \alpha \cdot x_i$). Hence, in order to ensure correctly-rounded and reproducible results of this operation, which we name EXSCAL, we require only the IEEE 754-2008 compliance. But, in the studied variant of the unblocked LU factorization, see Alg. 4, EXSCAL scales a vector by the inverse of the diagonal element ($\alpha = 1/a_{ii}$), which does not assure the correct-rounding. That is due to the double rounding: one by the division while computing α and another by the actual vector scaling. To obtain the exact result, we propose an inverse version of EXSCAL (EXINVSCAL) – this operation directly performs the division of all the elements of the vector by the diagonal element, avoiding the redundant intermediate rounding. Therefore, EXINVSCAL not only ensures the exact result, but also reduces the amount of computations.

D. EXGER*: Exact Rank-1 Update*

We have already discussed the inner product (DOT) of two vectors. We also consider the outer product (GER) of two vectors x and y, which forms a matrix, and updates a matrix A: $A := A + \alpha \cdot x \cdot y^T$; this operation is often called as the rank-1 update. The corresponding element-wise operation to be performed is $a_{ij} := a_{ij} + \alpha \cdot x_i \cdot y_j$. Since we aim to derive the correctly-rounded and reproducible GER that underlies the unblocked LU factorization, see Alg. 4, here, we focus on a special case of GER when $\alpha := 1.0$. In this case, the element-wise outer product can be performed by invoking the fused-multiply-and-add (FMA) instruction. This instruction computes the intermediate result as in the infinite precision and, then, correctly-rounds the final result to the desired precision. Thus, by explicitly using the FMA instruction, we avoid the intermediate rounding and deliver the exact result of GER.

E. EXTRSM*: Exact Triangular Solve*

The triangular solve with multiple right-hand sides (TRSM) solves one of the matrix equations

$$op(A) \cdot X = \alpha \cdot B, \quad \text{or} \quad X \cdot op(A) = \alpha \cdot B,$$

where α is a scalar; X and B are $m \times n$ matrices; A is a unit, or non-unit, upper or lower triangular matrix; and $op(A)$ is one of $op(A) = A$ or $op(A) = A^T$. Once the computation progresses, the matrix-solution X overwrites B, so only one matrix is required.

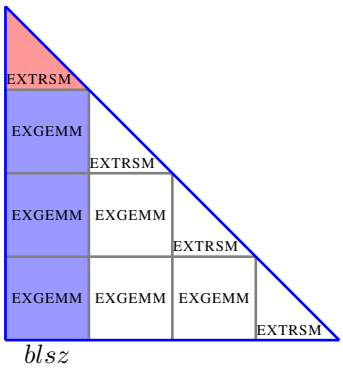


Figure 2: Partitioning of a lower triangular matrix A, where $blsz$ stands for the block size.

Our interest to TRSM lays in its employment within the studied blocked LU factorization, Alg. 5, where TRSM is applied to a *unit lower triangular matrix* A on the left from the matrix-solution X. Hence, our focus here is on this particular variant of TRSM.

In order to construct an exact TRSM, which we name as EXTRSM, we combine together a high-performance implementation of TRSM and our multi-level reproducible approach. Regarding the former, this implementation involves blocking with the block size $blsz$, where both A and B are split into blocks of size $blsz \times blsz$. Thus, the computations are organized on those blocks. Each local triangular system, involving a diagonal block, is solved with the local TRSM, while the update, involving the entire panel underneath each diagonal block, is computed with the local GEMM. This strategy is depicted in Fig. 2. Since the local TRSM still computes the solution in the sequential order, the performance benefit originates in the local GEMM that is computed in parallel with the cloud of work items. Due to the dependency on the local TRSM, the TRSM algorithm proceeds with the panel-step from left to right. Finally, we integrate our exact multi-level algorithm, EXDOT to be precise, into these local TRSM and GEMM to derive the local EXTRSM and EXGEMM (see Section III-F), accordingly.

Although EXTRSM assures reproducibility of computed solution as a sequence of reproducible and correctly-rounded computations for all its elements, the overall computed solution is often not correctly-rounded compared to the exact solution. That is due to the cascaded of rounding errors from rounding each computed element to the target floating-point format and then using this value to calculate the proceeding elements. In order to enhance the accuracy of EXTRSM, we propose to apply a few iterations of refinement based on the ExBLAS routines.

F. EXGEMM: *Exact Matrix Multiplication*

The matrix-matrix multiplication (GEMM) is one of the building blocks for the triangular solver with multiple right-hand sides (used internally within this routine) as well as for the blocked LU factorization, Alg. 5.

For the sake of this article, we consider a general case of GEMM: $C := \alpha \cdot A \cdot B + \beta \cdot C$, where α and β are scalars that equal one and zero, accordingly; C is a square matrix; and both A and B are column- and row-panel matrices, accordingly. To derive the exact and efficient GEMM (EXGEMM) [14], we construct our approach by combining the blocked implementation of GEMM for the performance purpose and EXDOT to assure both the accuracy and reproducibility. Thanks to the usage of EXDOT, EXGEMM delivers correctly-rounded results for each element of the matrix C.

Since computing each element of the matrix C requires involvement of a superaccumulator, even in the case of our hierarchical approach, that leads to a large memory footprint, in [14] we propose to use only certain amount of superaccumulators that correspond to the currently computed blocks of the matrix C. Then, these superaccumulators can be reused for computing the remaining blocks of C. Here, we propose a lightweight approach for ensuring reproducibility of GEMM, aiming to improve the EXGEMM performance. This approach employs only floating-point expansions with the early-exit technique and, then, rounds each expansion to the desired format. The latter is the difficult part in this approach. We consider to utilize the `Add3` [15] algorithm, however we aim to derive our own algorithm for this rounding. This lightweight approach will reduce the memory pressure and assure reproducibility, but may not always lead to the correctly rounded results.

IV. REPRODUCIBLE UNBLOCKED LU FACTORIZATION

The LU factorization decomposes an $m \times n$ matrix A into the product of an $m \times r$ unit triagular factor L and an $r \times n$ upper triangular factor U, where $r = \min(m, n)$. For the numerical stability, a sequence of row permutations is applied during the factorization, yielding the decomposition $PA = LU$, where P is an $m \times m$ permutation matrix. Alg. 4 displays the (unblocked) right-looking (RL) algorithm for the LU factorization with partial pivoting using the FLAME notation [16], [17]. There, *size*(A) returns the number of columns of matrix A; for further details on the notation, see [16]. Before the computation commences, A is virtually partitioned into four blocks: A_{TL}, A_{TR}, A_{BL}, and A_{BR}, where A_{TL} is initially void (0×0). The matrix A is then traversed from its top-left to the bottom-right corners. At each iteration of the loop, A is repartitioned $2 \times 2 \rightarrow 3 \times 3$, see Fig. 3, identifying a scalar α_{11} on the diagonal of A, as well as the vectors $a_{01}, a_{10}^T, a_{12}^T$, and the matrix blocks A_{00}, A_{02}, A_{20}, and A_{22}. The operations in the loop body update a_{21} and A_{22} using, respectively, calls to Level-1 and Level-2 BLAS kernels SCAL and GER. Upon completion, the strictly lower triangle of A is overwritten with the corresponding entries of the the unit lower triangular factor L while the upper triangular part of A contain those of U. Furthermore, a vector p of pivots is constructed that implicitly stores the permutation matrix P applied during the factorization.

All the operations appearing in Alg. 4 are cast in terms of BLAS routines. Therefore, we can develop an entire reproducible unblocked RL algorithmic variant for the LU factorization by simply relying on the corresponding ExBLAS routines, namely EXINVSCAL and EXGER.

Regarding the application of the partial pivoting strategy in Alg. 4, we note that this technique is composed of two stages:

1) MAX – find the maximum element in absolute value in the part of a matrix column, starting from the diagonal element. This operation is always reproducible.
2) SWAP – exchange values of two rows. This operation is also reproducible by nature.

In conclusion, all computational steps of the proposed unblocked RL algorithm for the LU factorization with partial pivoting employ reproducible kernels, such as EXINVSCAL and EXGER, in conjunction with the reproducible strategy for partial pivoting. Thus, by removing all the sources of indeterminism in Alg. 4, and exploiting multi-threaded parallelism within its building blocks, we ensure the reproducibility and improve parallel efficiency for this algorithmic variant for the LU factorization.

V. REPRODUCIBLE BLOCKED LU FACTORIZATION

Despite being numerically stable (in practice) and reproducible, the unblocked algorithmic variant for the LU factorization discussed in Section IV will not deliver high performance on current processor architectures. For this scenario, it is more efficient to rely on a blocked formulation of the algorithm, as that illustrated in Alg. 5.

At each iteration of Alg. 5, matrix A is repartitioned $2 \times 2 \rightarrow 3 \times 3$, identifying the $n_b \times n_b$ block A_{11} on the diagonal of A and other matrix blocks of various shapes: row-panels, column-panels and square blocks. The operations in the loop body that perform the updates within Alg. 5 are split into four steps:

1) The LU factorization with partial row pivoting of the diagonal and subdiagonal matrices, computed via the unblocked RL algorithm in Alg. 4.
2) The permutation of multiple rows of the matrix A.
3) TRSM to solve a unit lower triangular system with multiple right-hand sides.
4) GEMM to compute a general matrix-matrix multiplication.

Hence, Alg. 5 is also composed of simpler linear algebra operations – such as the unblocked LU factorization and

Algorithm 4: The unblocked RL algorithmic variant with partial pivoting for the LU factorization.

Partition

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right), \; p \rightarrow \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right)$$

where A_{TL} is 0×0, p_T has 0 elements

While $size(A_{TL}) < size(A)$ **do**

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right), \; \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \rightarrow \left(\begin{array}{c} p_0 \\ \hline \pi_1 \\ \hline p_2 \end{array}\right)$$

where α_{11} and π_1 are scalars

$$\pi_1 := PivIndex\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right) \qquad \text{(MAX)}$$

$$\left(\begin{array}{c|c|c} a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right) := P(\pi_1)\left(\begin{array}{c|c|c} a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right) \qquad \text{(SWAP)}$$

$$a_{21} := a_{21}/\alpha_{11} \qquad \text{(SCAL/INVSCAL)}$$

$$A_{22} := A_{22} - a_{21}a_{12}^T \qquad \text{(GER)}$$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right), \; \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \leftarrow \left(\begin{array}{c} p_0 \\ \hline \pi_1 \\ \hline p_2 \end{array}\right)$$

endwhile

Algorithm 5: The blocked RL algorithmic variant with partial pivoting for the LU factorization.

Partition

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right), \; p \rightarrow \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right)$$

where A_{TL} is 0×0, p_T has 0 elements

While $size(A_{TL}) < size(A)$ **do**

Determine block size n_b

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \; \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \rightarrow \left(\begin{array}{c} p_0 \\ \hline p_1 \\ \hline p_2 \end{array}\right)$$

where A_{11} is $n_b \times n_b$ and p_1 has n_b elements

$$\left[\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right), p_1\right] := LUP_{unb}\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right) \qquad (Alg.\ 4)$$

$$\left(\begin{array}{c|c} A_{10} & A_{12} \\ \hline A_{20} & A_{22} \end{array}\right) := P(p_1)\left(\begin{array}{c|c} A_{10} & A_{12} \\ \hline A_{20} & A_{22} \end{array}\right)$$

$$A_{12} := trilu(A_{11})^{-1}A_{12} \qquad \text{(TRSM)}$$

$$A_{22} := A_{22} - A_{21}A_{12} \qquad \text{(GEMM)}$$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \; \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \leftarrow \left(\begin{array}{c} p_0 \\ \hline p_1 \\ \hline p_2 \end{array}\right)$$

endwhile

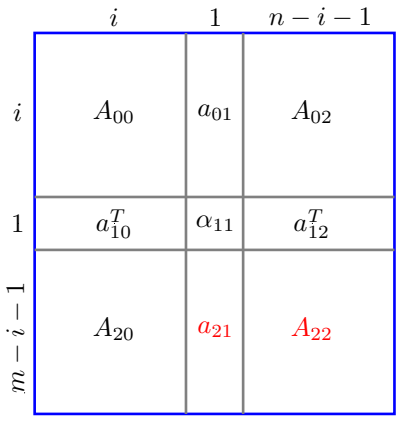


Figure 3: Partitioning of the matrix A.

several BLAS kernels. Consequently, we can construct a reproducible blocked algorithmic variant of the LU factorization on top of the unblocked algorithmic variant and the appropriate ExBLAS routines, namely EXTRSM and EXGEMM. To conclude, this algorithmic variant ensures reproducibility of the results and can be expected to deliver higher performance than its unblocked counterpart.

VI. Experimental Results

We verify the accuracy and evaluate the performance of the unblocked RL LU factorization with partial row pivoting and the underlying BLAS routines of the blocked variant on two different NVIDIA architectures; see Tab. I for the architectures details.

Table I: Hardware platforms employed in the experimental evaluation.

NVIDIA Quadro K420	192 CUDA cores	0.780 GHz
NVIDIA Tesla K80	4,992 CUDA cores with a dual-GPU design	0.560-0.875 GHz

In order to develop implementations for the studied reproducible algorithmic variants, we start by providing our vectorized, parallelized, and optimized non-deterministic double precision implementations of the general matrix-matrix multiplication, the triangular solve with multiple right-hand sides, and the unblocked LU factorization in Alg. 4; we refer to these implementations on figures as "GEMM", "TRSM", and "UNBLU", accordingly. We then integrate our reproducible solutions into these implementations. We tune our implementations by promoting loop unrolling and changing workgroup size, as well as by efficiently utilizing the GPU

resources – such as SIMD instructions, FMAs, private and local memory, and atomic instructions.

We verify the accuracy of both non-deterministic and reproducible implementations by comparing their results against the ones produced by the multiple precision sequential library MPFR.

In the comparison, we employ these non-deterministic double precision implementations that are natural candidates to assess the performance, accuracy and reproducibility of the results. Regardless of some performance penalties, we emphasize the importance of obtaining reproducible and, when possible, correctly-rounded results.

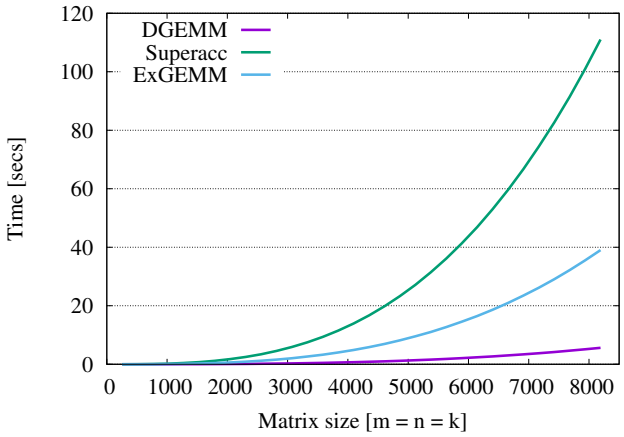
A. EXGEMM

For these tests, we consider two cases, depending on the matrices shapes:

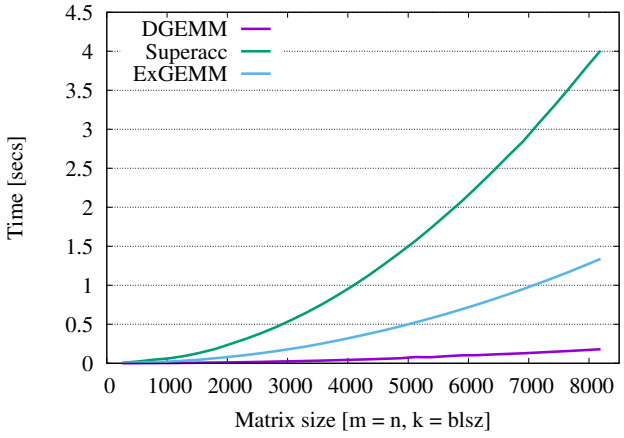
1) A general case where all the matrices are square.
2) The blocked LU factorization case, addressing the shapes of the matrices as inquired in the studied algorithmic variant, Alg. 5.

Figs. 4a and 4b display the performance results for these two cases. In the captions of both plots, "Superacc" corresponds to the exact matrix-matrix multiplication algorithm that is solely based on superaccumulators, while "EXGEMM " stands for our exact implementation of GEMM. The latter efficiently combines floating-point expansions and superaccumulators in contrast with the former, which could be classified as a reliable but brute-force approach to ensure reproducibility. Thus, EXGEMM reduces considerably the performance overhead of superaccumulators – from 20–22× to 7–8× only for the considered two cases compared to the non-deterministic double-precision GEMM – through the efficient usage of private memory for the expansions.

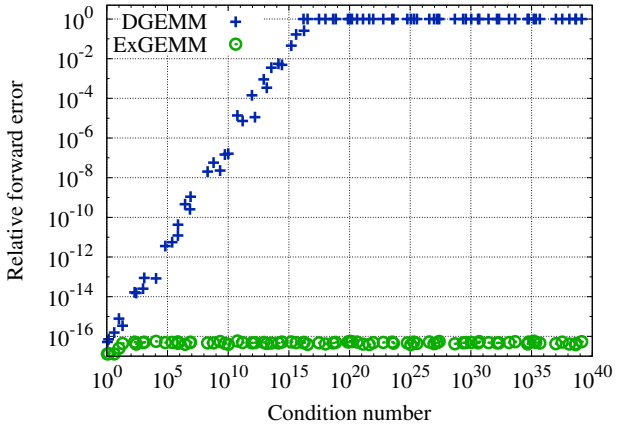
We also validate the accuracy of the computed results by GEMM and EXGEMM. Fig. 4c shows the relative forward error $\|C^* - C\|/\|C^*\|$ against the condition number of the problem, $A \cdot B$ in our case; C^* corresponds to the exact matrix-matrix multiplication and C is computed using either GEMM or EXGEMM. We compute the condition number of the problem as $\|exact(|A| \cdot |B|)\|/\|exact(A \cdot B)\|$. To generate the ill-conditioned matrix-matrix multiplication, we rely on the ill-conditioned dot product: The entries of $n-1$ rows of the matrix A and $n-1$ columns of the matrix B are generated following a random uniform distribution, and the remaining row in A and column in B represent vectors x and y, the result of the ill-conditioned dot product. For visual representation, those errors that exceed 1 are replaced by 1 since there is no single accurate digit left. As the relative forward error strongly depends on the condition number of the problem, the error of GEMM equals 1 for all condition numbers higher than 10^{16}. EXGEMM still ensures both correct-rounding and reproducibility of the results independently of the condition number because EXGEMM preserves every bit of the result until its final rounding to the target floating-point format, `binary64` in this case.



(a) General case with square matrices on K80



(b) The blocked LU factorization case on K80; $k = blsz = 256$; $blsz$ stands for the block size



(c) Accuracy of GEMM

Figure 4: Performance and accuracy results of GEMM.

B. EXTRSM

Fig. 5a shows the execution time of the double-precision and reproducible TRSM versus the matrix size. This test mirrors the scenario of using TRSM within the blocked LU factorization, Alg. 5, where the matrix A_{11} is of the fixed size $blsz \times blsz$ while the number of rows in the matrix A_{12} varies. The performance overhead of our reproducible approach within TRSM is higher – EXTRSM is roughly $14\times$ slower compared with the non-deterministic double-precision TRSM– than for GEMM due to the dependencies while computing the local EXTRSM as well as the limited possibility of using the local EXGEMM. However, we still benefit from the local EXGEMM as the internal block size is smaller than $blsz = 256$.

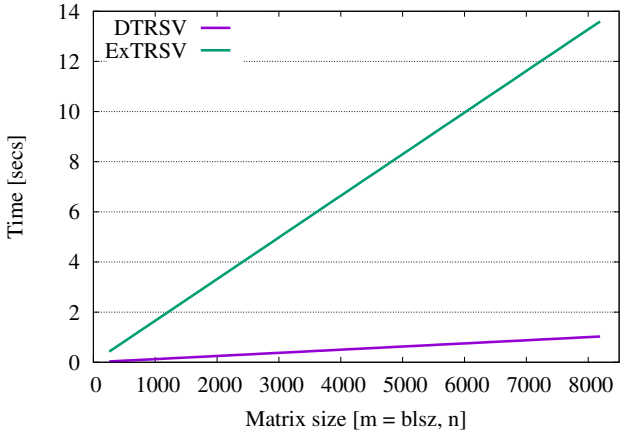
Regarding the accuracy of both double-precision and reproducible TRSM implementations, we carry out tests using ill-conditioned unit triangular matrices. To generate those matrices, we modify the algorithm described in [18] for the unit triangular matrices. The results of our tests are reported in Fig. 5b. We benefit from the MPFR library to compute exactly the relative forward error and the condition number of the problem; for the later we employ the Skeel formula [19], [3]: $cond(A, x) = \||A^{-1}||A||x|\|/\|x\|$. As the forward error reveals, both implementations are affected by the increase in the condition number, delivering no correct digit after a certain value; these errors were set to 1. This effect occurs later for the reproducible TRSM, as each element of the solution is computed in the reproducible and correctly-rounded way with only one rounding to double at the end.

C. EXUNBLU

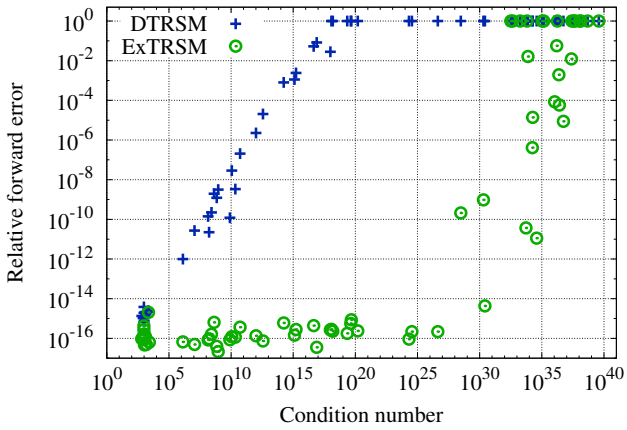
For the unblocked RL LU factorization, see Alg. 4, we do not employ our hierarchical approach for reproducibility, but rather carefully leverage the IEEE 754-2008 standard. Hence, Fig. 6 shows only two lines of results: UNBLU and EXUNBLU for the non-deterministic double-precision and exact double-precision implementations, respectively.

Figs. 6a and 6b report the execution time obtained by the unblocked RL algorithmic variant for the LU factorization with partial row pivoting as a function of the matrix size ($m = n$) on K420 and K80. UNBLU is roughly by 3–4 % faster than EXUNBLU on K80. In contrast, on K420 we can clearly see that the reproducible implementation outperforms the non-deterministic one by at least 10 %. The reason is that we compute α as an inverse of the diagonal element and then use this value in local SCAL (UNBLU), while in EXINVSCAL (EXUNBLU) we perform division directly. Therefore, as the error

To verify the accuracy of the RL unblocked LU factorization implementations, we conduct a set of tests on ill-conditioned matrices ($cond(A) \in [10^2, 10^{41}]$); the results of these tests are depicted in Fig. 6c. In order to create these matrices, we extend the generator for triangular systems [18]



(a) The blocked LU factorization case on K420; $m = blsz = 256$; $blsz$ stands for the block size



(b) Accuracy of TRSM

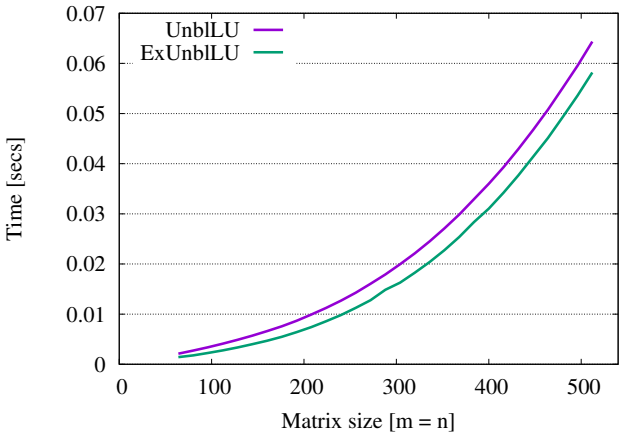
Figure 5: Performance and results of TRSM.

to a general case, when the entire matrix is in use. In order to compute the error as $\|PA - LU\|$, we rely on the MPFR library, especially for the product of two matrices. We filter errors and round those that exceed 1 to 1. This experiment demonstrates that both UNBLU and EXUNBLU deliver roughly the same accuracy, however the later is reproducible.

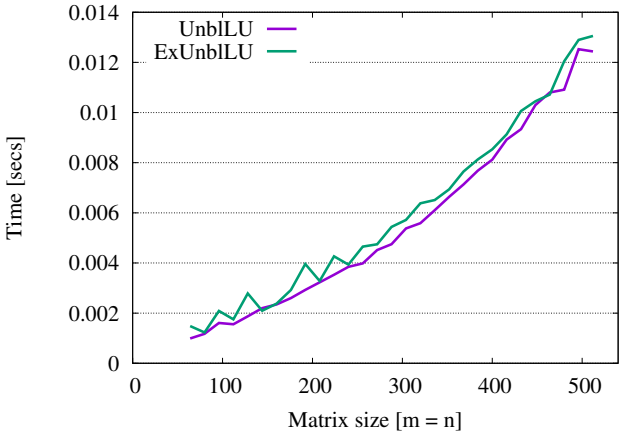
Taking into account the performance equivalence of both implementations (as the difference is within the time measurement fluctuation range of 3 %, especially for small problems) and the performance gain of EXUNBLU on K420, EXUNBLU is a competitive alternative to UNBLU as as it ensures numerical reproducibility of the results.

VII. CONCLUSIONS AND FUTURE WORK

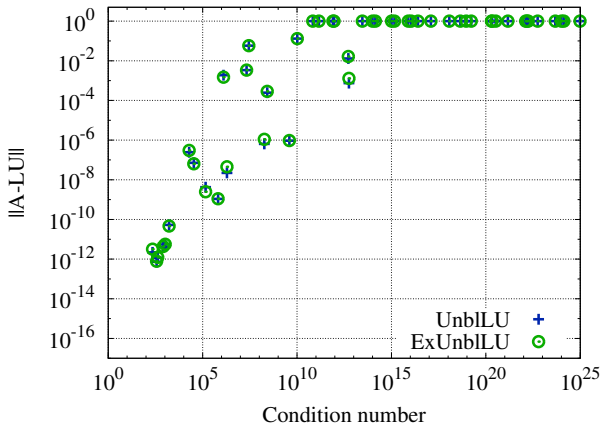
Dense linear algebra libraries virtually assemble a modular and hierarchical structure, where higher level operations –such matrix factorizations– can be entirely constructed on top of the lower level fundamental kernels – such as the BLAS routines. In this work, we exploited this property in order to derive its reproducible algorithmic variant of



(a) General case with square matrices on K420



(b) General case with square matrices on K80



(c) Accuracy of Alg. 4

Figure 6: Performance and accuracy results of the unblocked RL LU factorization, Alg. 4.

the blocked LU factorization. As a concrete case study, we considered the right-looking variant of the blocked LU factorization that is built on top of the Level-3 BLAS routines TRSM and GEMM as well as the right-looking unblocked LU factorization, which in turn relies upon the Level-1/2 BLAS kernels SCAL and GER. As the first step towards ensuring reproducibility of the blocked LU factorization, we proposed strategies to guarantee reproducibility of all these building blocks. We ensured both reproducible and correctly-rounded results for SCAL and GER by omitting the intermediate rounding, for example, through the explicit use of the FMA instruction. Moreover, we enhanced EXGEMM and proposed an initial version of the reproducible TRSM with blocking; and we introduced iterative refinement to enhance accuracy. All these underlying buildings blocks were implemented on NVIDIA GPUs reporting preliminary experimental results on two state-of-the-art accelerators. Our codes provide numerical stability and reproducibility. at the cost of some performance overheads that we plan to address in the future.

As part of future work, we plan to improve the performance of the compute-intensive BLAS kernels via a light-weight strategy, which we briefly outlined here, and complete the entire blocked LU factorization. In conjunction with the experimental evidences of the numerical reproducibility, we aim to provide theoretical proofs for EXTRSM and both unblocked and blocked LU factorizations along with their error analysis.

ACKNOWLEDGEMENT

The simulations were performed on resources provided by the Swedish National Infrastructure for Computing (SNIC) at PDC Centre for High Performance Computing (PDC-HPC).

REFERENCES

[1] J. Dongarra and al., "Applied Mathematics Research for Exascale Computing," U.S. Department of Energy, 2014.

[2] R. Lucas and al., "Top Ten Exascale Research Challenges," DOE ASCAC Subcommitee Report, 2014.

[3] N. J. Higham, *Accuracy and stability of numerical algorithms, second ed.* Philadelphia, PA: Society for Industrial and Applied Mathematics (SIAM), 2002.

[4] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of Floating-Point Arithmetic*. Birkhäuser, 2010.

[5] IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008, Aug. 2008.

[6] ——, *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-201x, Jan. 2017, draft 2.16. Available on the WWW, http://speleotrove.com/misc/P754-216.pdf.

[7] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff, "A set of level 3 basic linear algebra subprograms," vol. 16, no. 1, pp. 1–17, Mar. 1990.

[8] S. Collange, D. Defour, S. Graillat, and R. Iakymchuk, "Numerical reproducibility for the parallel reduction on multi- and many-core architectures," *Parallel Computing*, vol. 49, pp. 83–97, 2015.

[9] T. Ogita, S. M. Rump, and S. Oishi, "Accurate sum and dot product," *SIAM J. Sci. Comput*, vol. 26, 2005.

[10] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms, third ed.* Addison-Wesley, 1997.

[11] U. Kulisch and V. Snyder, "The Exact Dot Product As Basic Tool for Long Interval Arithmetic," *Computing*, vol. 91, no. 3, pp. 307–313, Mar. 2011.

[12] R. Iakymchuk, S. Collange, D. Defour, and S. Graillat, "ExBLAS (Exact BLAS) library," Available on the WWW, https://exblas.lip6.fr/, 2016, accessed 24-AUG-2016.

[13] R. Iakymchuk, S. Collange, D. Defour, and S. Graillat, "ExBLAS: Reproducible and accurate BLAS library," in *Proceedings of the Numerical Reproducibility at Exascale (NRE2015) workshop held as part of the Supercomputing Conference (SC15). Austin, TX, USA, November 15-20, 2015*, Oct. 2015.

[14] R. Iakymchuk, D. Defour, S. Collange, and S. Graillat, "Reproducible and Accurate Matrix Multiplication," in *LNCS. Scientific Computing, Computer Arithmetic, and Validated Numerics: SCAN 2014, Würzburg, Germany, September 21-26, 2014. Revised Selected Paper. Vol. 9553*, 2016, pp. 126–137.

[15] S. Boldo and G. Melquiond, "Emulation of a FMA and Correctly Rounded Sums: Proved Algorithms Using Rounding to Odd," *IEEE Transactions on Computers*, vol. 57, no. 4, pp. 462–471, 2008.

[16] P. Bientinesi, E. S. Quintana-Ortí, and R. A. van de Geijn, "Representing linear algebra algorithms in code: The FLAME application programming interfaces," *ACM Transactions on Mathematical Software*, vol. 31, no. 1, pp. 27–59, Mar. 2005.

[17] R. A. van de Geijn and E. S. Quintana-Ortí, *The Science of Programming Matrix Computations*. www.lulu.com, 2008.

[18] N. Louvet, "Algorithmes compensés en arithmétique flottante : précision, validation, performances," Ph.D. dissertation, UPVD, 2007. [Online]. Available: http://perso.ens-lyon.fr/nicolas.louvet/

[19] R. D. Skeel, "Scaling for numerical stability in Gaussian elimination," *J. Assoc. Comput. Mach.*, vol. 26, no. 3, pp. 494–526, 1979.