Hierarchical approach for deriving a reproducible unblocked LU factorization

Roman Iakymchuk[1] (ID)**, Stef Graillat[2], David Defour[3]
and Enrique S Quintana-Ortí[4]**

Abstract
We propose a reproducible variant of the unblocked LU factorization for graphics processor units (GPUs). For this purpose, we build upon Level-1/2 BLAS kernels that deliver correctly-rounded and reproducible results for the dot (inner) product, vector scaling, and the matrix-vector product. In addition, we draw a strategy to enhance the accuracy of the triangular solve via iterative refinement. Following a bottom-up approach, we finally construct a reproducible unblocked implementation of the LU factorization for GPUs, which accommodates partial pivoting for stability and can be eventually integrated in a high performance and stable algorithm for the (blocked) LU factorization.

Keywords
LU factorization, BLAS, reproducibility, accuracy, long accumulator, error-free transformation, GPUs

1. Introduction

The IEEE 754 standard, created in 1985 and then revised in 2008, has led to a considerable enhancement 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.

Exascale computing (10^{18} operations per second) is likely to be reached within a decade. For the type of systems yielding such performance rate, getting *accurate* and *reproducible* results in floating-point arithmetic will represent two considerable challenges (Dongarra et al., 2014; Lucas et al., 2014). 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. Reproducibility is also an important and useful property when debugging and checking the correctness of codes as well as for legal issues.

The solution of a linear system of equations is often at the core of many scientific applications. Usually, this process relies upon the LU factorization, which is also its most compute-intensive part. Although there exist implementations of this factorization that deliver high performance on a variety of processor architectures—including general-purpose multicore processors, Intel Xeon Phi, and graphics processors (GPUs)—their *reproducibility* and, even worst,

accuracy cannot be guaranteed. This problem is mainly due to the non-associativity of floating-point operations, combined with the concurrent thread-level execution of independent operations on conventional multicore processors or the non-determinism of warp scheduling on many-core GPUs. This last type of architecture is especially appealing for the acceleration of compute-intensive kernels, as those appearing in dense linear algebra (Golub and Loan, 1996; ICL Team, 2016).

In this work, we address the problem of reproducibility of the LU factorization on GPUs due to cancellations and rounding errors when dealing with floating-point arithmetic. Instead of developing a GPU implementation of the LU factorization from scratch, we rely on the hierarchical and modular structure of linear algebra libraries and start by creating and augmenting reproducible OpenCL kernels for the BLAS (*Basic Linear Algebra Subprograms*; Dongarra et al., 1990) that serve as building blocks in the LU

[1] KTH Royal Institute of Technology, Stockholm, Sweden
[2] Sorbonne Universités, Paris, France
[3] Université de Perpignan, Perpignan, France
[4] Universidad Jaime I, Castellón, Spain

Corresponding author:
Roman Iakymchuk, KTH Royal Institute of Technology, Lindstedtsvägen 5, 100 44 Stockholm, Sweden.
Email: riakymch@kth.se

factorization. In addition, we enhance the accuracy (in case of non-correctly-rounded results) of these underlying BLAS kernels for graphics accelerators.

We consider the unblocked left-looking algorithm for the LU factorization (also known as *jik* or *jki* variant) (Ortega, 1988). The unblocked version is important for the solution of "batched" linear systems (Haidar et al., 2015), where the goal is to solve a large sequence of independent small-size problems, and also as a building block to assemble high performance algorithms for the factorization. In addition, the left-looking version is especially appealing for fault tolerance, out-of-core computing, and the solution of linear systems when the coefficient matrix does not fit into the GPU memory. The unblocked left-looking algorithm can be formulated in terms of the Level-1 and Level-2 BLAS kernels for the dot product (DOT), vector scaling (SCAL), matrix-vector product (GEMV), and triangular system solve (TRSV). We prevent cancellations and rounding errors in these kernels by applying the following techniques:

- We leverage a long accumulator and error-free transformations (EFTs) designed for the exact, that is, reproducible and correctly-rounded, parallel reduction (EXSUM) (Collange et al., 2015) in order to derive an exact dot product (EXDOT). For this purpose, we combine the multilevel parallel reduction algorithm with the traditional EFT, called `TwoProd` (Ogita et al., 2005), for the multiplication of two floating-point numbers.
- By its nature, SCAL is both reproducible and correctly rounded. However, in the unblocked left-looking factorization, SCAL multiplies a vector by the inverse of a diagonal element, which causes two rounding errors (one to compute the inverse and one for the multiplication by the inverse). To address this issue, we provide an extension of SCAL (INVSCAL) that performs the division directly, ensuring correctly-rounded and reproducible results.
- We develop a reproducible and accurate implementation of GEMV by combining together a high performance GPU kernel of this operation with the exact DOT.
- To improve the parallel performance of TRSV, we use a blocked variant that relies upon small TRSV involving the diagonal blocks and rectangular GEMV with the off-diagonal blocks. This approach leads to a reproducible, but not yet correctly-rounded, triangular solve (EXTRSV) (Iakymchuk et al., 2015b). We tackle this accuracy problem by applying a few iterations of iterative refinement.
- Finally, we integrate partial pivoting (Golub and Loan, 1996) into unblocked left-looking algorithm for the LU factorization which, as part of future work, will allow us to employ this component in the solution of batched linear systems as well as a building block for high performance blocked factorization algorithms.

Algorithm 1. EFT for the summation of two floating-point numbers.

---
1 Function $[r, e] = \texttt{TwoSum}(a, b)$ $r := a + b$
2 $z := r - a$
3 $e := (a - (r - z)) + (b - z)$

The article is organized as follows. Section 2 reviews several aspects of computer arithmetic, in particular floating-point expansions (FPEs) and the Kulisch superaccumulator. Section 3 presents the ExBLAS library with its current set of routines. Section 4 is devoted to the presentation of the proposed reproducible LU algorithm. We evaluate our implementations in Section 5. Finally, we discuss related works and draw conclusions in Sections 6 and 7, respectively.

2. Background

In this article, we consider the double precision format (`binary64`) as specified in the IEEE-754 standard. This standard requires correctly-rounded results for the basic arithmetic operations $(+, -, \cdot, /, \sqrt{\ })$, which means that the operations are performed as if the result was first computed using infinite precision, and then rounded to the floating-point format. In this work, we assume the rounding-to-nearest mode.

Due to rounding errors, floating-point operations are non-associative and, therefore, non-reproducible (IEEE, 1985). Hence, the accuracy and reproducibility of floating-point operations strongly depend on their order (Higham, 2002; Muller et al., 2010). As a consequence, dynamic thread scheduling, which is often exploited to improve the performance of parallel algorithms, may lead to different results from one execution to another.

In the remainder of this section, we present a brief overview of algorithms that constitute the foundation of our present work. Concretely, FPEs with EFTs (Section 2.1) and a Kulisch superaccumulator (Section 2.2) have been proposed in the past in order to perform addition/subtraction of floating-point numbers without round-off errors; and these two algorithms have been efficiently combined to derive the hierarchical scheme for parallel summation (Collange et al., 2015) (Section 3.1) and dot product (Iakymchuk et al., 2015a) (Section 3.2).

2.1. Floating-point expansion

FPEs allow us to recover and track rounding error which occurs during floating-point additions. FPE represents the result as an unevaluated sum of p floating-point numbers whose components are ordered in magnitude with minimal overlap to cover a wide range of exponents. FPEs of sizes $p = 2$ and 4, based on the EFT, are described in the works of Li et al. (2002) and Hida et al. (2001), respectively. The conventional EFT for the addition (`TwoSum`) is given in Algorithm 1 (Knuth, 1997) and, for the multiplication (`TwoProd`), in Algorithm 2 (Ogita et al., 2005). Algorithm 1 computes the addition of two floating-point

Algorithm 2. EFT for the product of two floating-point numbers.

---
1 Function $[r, e] = \mathtt{TwoProd}(a, b)$ $r := a \cdot b$
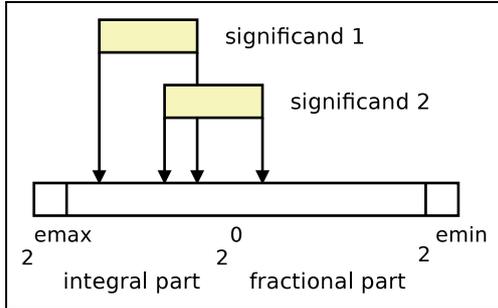2 $e := \mathrm{FMA}(a, b, -r)$

Figure 1. Kulisch superaccumulator.

numbers a and b and returns the result r and the error e such that r and e do not overlap. Similarly, $\mathtt{TwoProd}$ performs the multiplication of two floating-point numbers a and b. For $\mathtt{TwoProd}$, 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.

Adding one 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 $\mathtt{TwoSum}$ 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 $\mathtt{binary64}$), the FPE approach computes the accumulation of numbers without loss of accuracy.

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

2.2. Kulisch superaccumulator

The Kulisch superaccumulator covers the range from the minimum representable floating-point value to the maximum value in absolute value. For the dot product of two vectors composed of $\mathtt{binary64}$ elements, Kulisch (Kulisch and Snyder, 2011) proposed to use a 4288-bit accumulator. The addition of products of $\mathtt{binary64}$ values is performed without loss of information by accumulating every floating-point number in the superaccumulator (see Figure 1). The superaccumulator can produce the exact sum or dot product of a very large amount of floating-point numbers with arbitrary dynamic ranges. However, the superaccumulator incurs a large memory overhead, due to the required storage hindering vectorization due to indirect memory accesses.

Algorithm 3. Floating-point expansion of size p.

---
1 Function ExpansionAccumulate(x) **for**
   $i = 0 \rightarrow p - 1$ **do**
2 $\quad | \quad (a_i, x) := \mathrm{TwoSum}(a_i, x)$
3 **end**
4 **if** $x \neq 0$ **then**
5 $\quad |$ Superaccumulate(x)
6 **end**
---

3. Exact BLAS library (ExBLAS) on GPUs

This section briefly reviews the prototype implementation of the Exact BLAS (ExBLAS) library (Iakymchuk et al., 2016a) that underlies our LU factorization. It starts with the parallel reduction and dot product, as they are two fundamental BLAS kernels, and then continues with the triangular solver, outlining the improvements present in this article compared with prior work. In addition, we extend the set of routines to the scaling of a vector by the inverse of a scalar and the matrix-vector product, which represents new reproducible algorithms and implementations. While presenting these routines, we also expose how they are used within the discussed unblocked LU factorization.

3.1. Exact parallel reduction: EXSUM

In our previous works (Collange et al., 2015; Iakymchuk et al., 2016), we introduced a multilevel approach to the summation problem. It combines the FPE and superaccumulator algorithms. The procedure splits the computation into five phases: filtering, private superaccumulation, local superaccumulation, global superaccumulation, and rounding. This five-phase decomposition exploits the nested parallelism of modern parallel architectures such as GPUs.

The first phase of this hierarchical summation approach relies on FPEs with EFTs for the accumulation of floating-point numbers (see Algorithms 1 and 2). Each thread maintains its own FPE. In order to enable expansions of size p, we apply Algorithm 3, based on Algorithm 1, enhanced with superaccumulators in case the result of the accumulation cannot be represented with a FPE of size p.

In the second phase, if the last rounding error x is non-zero (see Algorithm 3), x is forwarded to the private superaccumulator, the FPE is flushed to the superaccumulator, and the accumulation process is continued. At the end of the summation, all FPEs are forwarded to superaccumulators. Depending on the amount of memory that is available, private superaccumulators (shared among a small number of threads) are allocated in either fast local memory, for example, cache or shared memory, or global memory.

In the third phase, k private superaccumulators are merged into a single local superaccumulator, one per group of threads. In the fourth phase, all local superaccumulators within a GPU are combined together into a global superaccumulator. Finally, the global superaccumulator is

rounded back to the target floating-point format, in the fifth phase, in order to obtain the correctly-rounded result.

We developed hand-tuned variants of a single OpenCL implementation for NVIDIA and AMD GPUs. These variants use 16 superaccumulators per workgroup of 256 threads and employ local memory to store these superaccumulators. In order to avoid bank conflicts, superaccumulators are interleaved to spread their digits among different memory banks. Concurrency between 16 threads that share one superaccumulator is settled thanks to atomic operations while scattering the input data into the corresponding digits of the superaccumulator.

3.2. Exact dot product: EXDOT

We apply the multilevel parallel reduction algorithm (Collange et al., 2015) to the dot product (Iakymchuk et al., 2015a) by additionally utilizing the `TwoProd` EFT for the exact multiplication of two floating-point numbers. The GPU implementation of EXDOT is based on the same concept that underlying EXSUM with an auxiliary function call to `TwoProd` and the corresponding treatment of its outputs. The performance results of the exact dot product on GPUs present a small overhead induced by the summation of two numbers (the result and the error) after each call to `TwoProd`. In contrast, for large array sizes, EXDOT delivers both numerically reproducible and correctly-rounded results with comparable performance to its standard non-deterministic version.

3.3. Exact vector scaling: EXSCAL *and* EXINVSCAL

Multiplying a vector x by a scalar α is a rather simple kernel that requires only one operation to be performed for each element of the vector ($x_i := \alpha \cdot x_i$). When the IEEE 754-2008 compliance is ensured, this operation (EXSCAL) is both correctly rounded and reproducible. However, the unblocked LU factorization (Algorithm 7) requires a call to the SCAL routine in order to scale the vector by the inverse of the diagonal element. In that case, the result is not correctly rounded due to the two rounding errors coming from the division followed by the multiplication. To ensure correctly-rounded and reproducible results, we propose a dedicated version of SCAL (EXINVSCAL). This version directly performs division of the vector elements by the diagonal element without the intermediate rounding error. Both EXSCAL and EXINVSCAL are easy to implement on GPUs as the update of the vector elements can be performed in parallel by a team of threads.

3.4. Exact matrix-vector product: EXGEMV

The matrix-vector product (GEMV) is one of the building blocks for the triangular solver as well as for the unblocked LU factorization. Therefore, we next present its correctly-rounded and reproducible implementation. The GEMV kernel computes one of the following matrix-vector operations
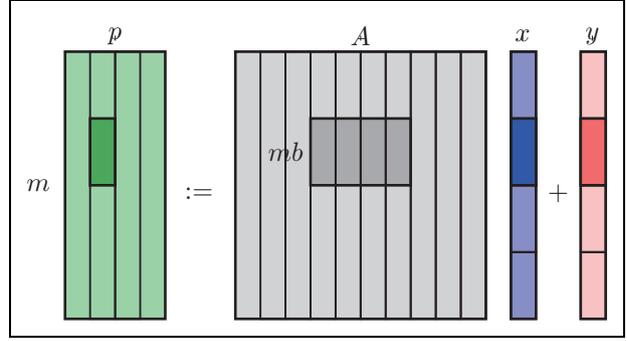

Figure 2. Work distribution in the matrix-vector product.

$$y := \alpha A \cdot x + \beta y \quad \text{or} \quad y := \alpha A^T \cdot x + \beta y \qquad (1)$$

where α and β are scalars, x and y are vectors, and A is a matrix of size $m \times n$.

We derive a reproducible and accurate algorithm for EXGEMV by combining its two-kernel algorithmic variants from Bainville (2010) to achieve high performance; and the exact dot product, which is described in Section 3.2, to guarantee both reproducibility and accuracy of the results. We next provide a description of the implementation as a GPU kernel.

The proposed OpenCL implementation of the matrix-vector product splits the computations into blocks, so that each workgroup of threads computes a certain part of the output vector y. Figure 2 shows how a part of the vector y is computed using p workgroups of size mb. Each thread from a workgroup is engaged in preloading a part of x (colored in dark blue) into local memory, making it available to the whole workgroup. Then, each thread computes its partial dot product using its own FPE, which is kept in private memory; the computation on the thread-owned FPEs is very fast. The FPEs with the partial results are flushed to local superaccumulators that are stored in the corresponding cells of the $m \times p$ matrix (dark green). Concerning the superaccumulators, we adhere to one of the following two scenarios:

1. Hold a matrix of $m \times p$ superaccumulators in global memory and then perform a reduction on them in order to obtain a vector of m superaccumulators that corresponds to the output vector y.
2. Hold only a vector of m superaccumulators, where each superaccumulator is shared among p threads, and solve the contention for shared superaccumulators using atomic operations.

The second scenario reduces the EXGEMV implementation to one kernel and releases the pressure on the global memory. That is also beneficial for the $n - 1$ (n is a matrix size) calls to EXGEMV within the unblocked LU factorization. The first scenario is left for future work.

Each workgroup holds only a part of the vector x in its local memory. This allows multiple workgroups per multiprocessor to proceed concurrently. This strategy also

Algorithm 4. Forward substitution.

```
1  x₁ := b₁/l₁₁
2  for i = 2 : n do
3  |   s := bᵢ
4  |   for j = 1 : i − 1 do
5  |   |   s := s − lᵢⱼxⱼ
6  |   end
7  |   xᵢ := s/lᵢᵢ
8  end
```

maximizes the reuse of cached-shared parts of the vector x as there are 256 or 512 threads per workgroup.

In the current version of ExBLAS, we provide implementations that cover all possible cases of EXGEMV as depicted in equation (1).

3.5. Reproducible triangular solver: EXTRSV

The triangular solve involving a lower triangular coefficient matrix is one of the building blocks for the unblocked LU factorization. We next provide the multilevel reproducible approach for triangular systems with a single right-hand side (Iakymchuk et al., 2015b) and reveal our strategies for improving its accuracy.

Let us consider the system $Lx = b$, where L is a non-singular (square) lower triangular matrix. This system can be solved using (sequential) forward substitution, as depicted in Algorithm 4, where the elements of x are computed from first to last.

Algorithm 5 represents the blocked algorithm for the reproducible triangular solver with a lower triangular matrix. Our strategy depicted in this algorithm is the following:

1. Use a blocked version of the triangular solver.
2. Apply the exact dot product ($s := s − l_{ij}x_j$) with FPEs and superaccumulators. We apply this procedure through the blocked EXTRSV: within the non-blocked (called local) EXTRSV and the local blocked EXGEMV. The array of superaccumulators is shared between these two routines. Since the computations in the blocked EXTRSV are performed in sequential order (local EXTRSV, local EXGEMV, local EXTRSV, etc.), there is no contention for the array of superaccumulators.
3. Correctly round the accumulated result ($\hat{s} := RNDN(acc(k))$). One can notice that rounding to double is performed only once—at the end of the computation of each element of the solution.
4. Perform the division by the corresponding diagonal element ($\hat{x}_i := \hat{s}/l_{ii}$).

Theorem 3.1. The previous strategy yields a reproducible solution for the triangular solve.

Proof. We give a proof by induction. As $x_1 := b_1/l_{11}$ and as the division is correctly-rounded, the latter always returns

Algorithm 5. Blocked EXTRSV: *blsz* stands for a block size; *acc* refers to an array of superaccumulators of size *n*; *acc(k)* corresponds to one superaccumulator at the position *k* in this array; *ExpansionAccumulate()* accumulates a number using Algorithm 3; *Round()* rounds a superaccumulator to the target floating-point format.

```
1  for i = 0 : blsz : n do
2  |   Local EXTRSV  for k = i : i + blsz do
3  |   |   for j = 1 : k − 1 do
4  |   |   |   [r, e] := TwoProd(lₖⱼ, −xⱼ)
5  |   |   |   ExpansionAccumulate(r)
6  |   |   |   ExpansionAccumulate(e)
7  |   |   end
8  |   |   ExpansionAccumulate(bₖ)
9  |   |   ŝ := Round(acc(k))
10 |   |   xₖ := ŝ/lₖₖ
11 |   end
12 |   Local EXGEMV  for k = i + blsz : n do
13 |   |   for j = i : i + blsz do
14 |   |   |   [r, e] := TwoProd(lₖⱼ, −xⱼ)
15 |   |   |   ExpansionAccumulate(r)
16 |   |   |   ExpansionAccumulate(e)
17 |   |   end
18 |   end
19 end
```

the same result, and, therefore, it is reproducible. Let us now assume that $x_1, \ldots, x_{i-1}$ are reproducible. As the computation $s := s − l_{ij}x_j$ is done with a large accumulator, there is no rounding error and the result is exact, independently of the order of computation and, consequently, reproducible. The operation $\hat{s} := RNDN(s)$ is reproducible as it is a correctly-rounded operation. Finally, $x_i := \hat{s}/l_{ii}$ is reproducible as it is the result of a correctly-rounded division between two reproducible quantities. □

With the proposed approach, the matrix, both the right-hand side and the solution vectors are first split into blocks. Then computations on those blocks are organized as in Figure 3(a) and (b). With this organization, each diagonal block of size $blsz \times blsz$ participates in EXTRSV, and the panel underneath this block is involved in EXGEMV. The challenge lies in ensuring a balanced workload distribution among workgroups on GPUs, and a light synchronization overhead, as some parts of the algorithm (the local EXTRSV on the diagonal blocks, see Figure 3(b)) still need to be executed in the sequential order.

Each of the four aforementioned parts of the proposed triangular solver is reproducible and, therefore, the computed solution is reproducible as well. Compared with our previous work (Iakymchuk et al., 2015b), in the new version of ExBLAS we covered all the four variants of the triangular solver as follows

$$T \cdot x = b \quad \text{or} \quad T^T \cdot x = b \quad (2)$$
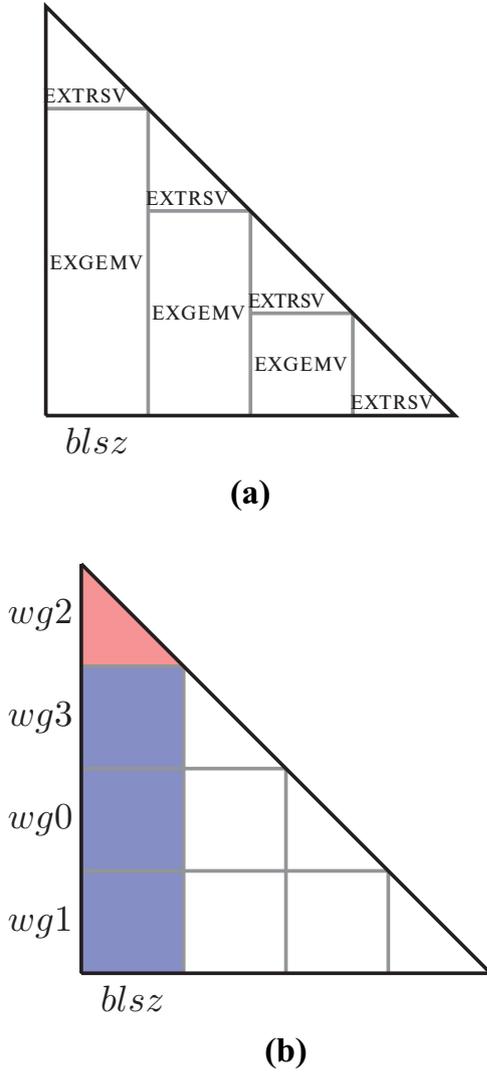
Figure 3. Partitioning of a lower triangular matrix L, where blsz stands for a block size and WGx is the number of a workgroup x: (a) kernel wise and (b) work group wise.

Algorithm 6. The reproducible triangular solver with iterative refinement.

1	$\widehat{x} := L^{-1}b$	EXTRSV
2	**for** $i = 1 : nbiter$ **do**	
3	$r := b - L\widehat{x}$	EXGEMV
4	$d := L^{-1}r$	EXTRSV
5	$\widehat{x} := \widehat{x} + d$	EXAXPY
6	**end**	

where x and b are n elements vectors, and T is an $n \times n$ unit, or non-unit, upper or lower triangular matrix.

In order to enhance the accuracy of the reproducible TRSV, we propose to apply a few iterations of iterative refinement based on the ExBLAS routines, as described in Algorithm 6. The overhead of the iterative refinement can be diminished by locating the source of the accuracy

Algorithm 7. The left-looking algorithmic variant of the unblocked algorithm LU factorization with partial pivoting.

Partition

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

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

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

    Repartition

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

    where α_{11} and π_1 are scalars

$$\left(\begin{array}{c} a_{01} \\ \hline \alpha_{11} \\ \hline a_{21} \end{array}\right) := P(p_0) \left(\begin{array}{c} a_{01} \\ \hline \alpha_{11} \\ \hline a_{21} \end{array}\right)$$

$$\begin{array}{ll} a_{01} := L_{00}^{-1} a_{01} & (trsv) \\ \alpha_{11} := \alpha_{11} - a_{10}^T a_{01} & (dot) \\ a_{21} := a_{21} - A_{20} a_{01} & (gemv) \end{array}$$

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

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

$$a_{21} := a_{21}/\alpha_{11} \qquad\qquad (scal/invscal)$$

    Continue with

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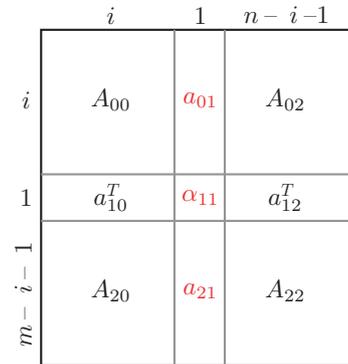
endwhile

Figure 4. Partitioning of the matrix A: $a_{01}, \alpha_{11}, a_{21}$ are updated using Algorithm 7.

problem and, consequently, applying iterative refinement directly on the diagonal blocks right after the local EXTRSV.

Theorem 3.2. Algorithm 6, for the triangular solve with iterative refinement, is reproducible.

Proof. The first step uses EXTRSV to compute $\hat{x} := L^{-1}b$, so $\hat{x}$ is reproducible. Inside the loop, we sequentially compute r, d, and x, with each of them being reproducible due to the use of EXTRSV, EXTRSV, and EXAXPY, accordingly. As a consequence, the whole algorithm is reproducible. $\qquad\square$

4. Reproducible LU factorization

The LU factorization decomposes an $m \times n$ matrix A into the product of an $m \times r$ unit lower triangular matrix L, with $r = \min(m, n)$; an $r \times n$ upper triangular matrix U; and an $r \times r$ permutation P such that $PA = LU$. In order to compute this decomposition, we consider an unblocked left-looking variant, illustrated in Algorithm 7 using the FLAME notation (Bientinesi et al., 2005; van de Geijn and Quintana-Ortí, 2008). This notation makes it easier to identify which regions of the matrix are updated and used (see Figure 4). In Algorithm 7, $size(A)$ represents the number of columns of the matrix A. Before the computation starts, A is virtually partitioned into four blocks A_{TL}, A_{TR}, A_{BL}, and A_{BR}, where A_{TL} is a 0×0 matrix. The matrix is then traversed from the top-left to the bottom-right corner. At each iteration of the loop, A is repartitioned from 2×2 to 3×3 form, where A_{00}, A_{02}, A_{20}, and A_{22} are matrices; $a_{01}, a_{10}^T, a_{12}^T$, and a_{21} are vectors; and α_{11} is a scalar. The algorithm updates a_{01}, α_{11}, and a_{21} using TRSV, DOT, and INVSCAL/GEMV, respectively. At the end of the computation, matrix A is overwritten by the upper triangular matrix U and the unit lower triangular matrix L. Additionally, the vector p of pivots (representing the permutations contained in P) is created.

Algorithm 7 shows that, thanks to the modular and hierarchical structure of linear algebra libraries, computations of more complex algorithms—such as the LU factorization—can be entirely expressed and built on top of the lower level fundamental routines—in particular, the BLAS one. We benefit from this hierarchical organization to propose a reproducible algorithmic variant of the unblocked LU factorization, Algorithm 7, on top of the underlying ExBLAS routines: EXTRSV, EXDOT, EXINVSCAL, and EXGEMV.

To enable support for (reproducible) partial pivoting in Algorithm 7, we split this process into two stages:

1. Searching for the maximum element in absolute value within the sub-diagonal part of a matrix column. This operation is always reproducible.
2. Swapping two rows. This operation is also reproducible by nature.

In addition, for this particular variant of the unblocked LU factorization, we apply pivoting from the previous iteration right before the computations update the current one.

Theorem 4.1. Algorithm 7, for the unblocked LU factorization with partial pivoting, is reproducible.

Proof. The first and last steps consist in partitioning the matrix A that does not bring any reproducibility issues. All computational steps of the unblocked LU factorization rely upon their reproducible counterparts, such as EXTRSV, EXDOT, EXINVSCAL, and EXGEMV, plus the reproducible strategy implementing partial pivoting. As a consequence, the

Table 1. Hardware platforms employed in the experimental evaluation.

NVIDIA Quadro K420	192 cores	0.780 GHz
NVIDIA Tesla K20c	2496 cores	0.706 GHz
NVIDIA Tesla K80	4992 cores	0.560–0.875 GHz

result computed by Algorithm 7 is numerically reproducible.

Therefore, we have successfully removed all sources of indeterminism, while efficiently exploiting data parallelism within each basic block. Even though the accuracy of the result computed by Algorithm 7 is improved, it is not correctly rounded. This is due to the fact that a rounding step is executed between each reproducible BLAS operations. A solution to increase the accuracy is to perform some iteration of iterative refinement. However, we believe it is not possible to prove that correct rounding can be achieved even with iterative refinement. This is due to the fact that there exist obvious cases (division by 3 or multiplications by small numbers, for example) where numerical information that could be useful for rounding could be lost due to underflow.

5. Experimental results

In this section, we evaluate our implementations of the unblocked LU factorization with partial pivoting and its underlying ExBLAS routines on three NVIDIA architectures (see Table 1). Note that NVIDIA Tesla K80 cores are with a dual-GPU design.

We develop unique OpenCL implementations for each algorithm on GPUs and tuned these implementations—for example, by promoting loop unrolling or changing work-group size, and so on—for each particular architecture in order to optimize performance. Moreover, our implementations strive to deliver the best performance through utilizing all the underlying resources of the target GPUs: SIMD instructions, FMAS, private and local memory, as well as atomic instructions. We check the correctness of our implementations by comparing the computed results with those produced by the multiple precision sequential library MPFR for CPUs on randomly generated test sets.

As a baseline for comparison purposes, we provide our vectorized, parallelized, and optimized non-deterministic double precision implementations of the matrix-vector product, triangular solver, and the unblocked LU factorization (Algorithm 7). We denote these implementations on figures as "Parallel GEMV," "Parallel TRSV," and "Parallel LU," accordingly. We use these implementations as starting points for the integration of our reproducible solutions. Hence, these baseline implementations are very relevant in order to assess the performance, accuracy, and reproducibility of the results. Despite some performance penalties, we would like to emphasize the importance of obtaining reproducible and, if possible, correctly-rounded results.
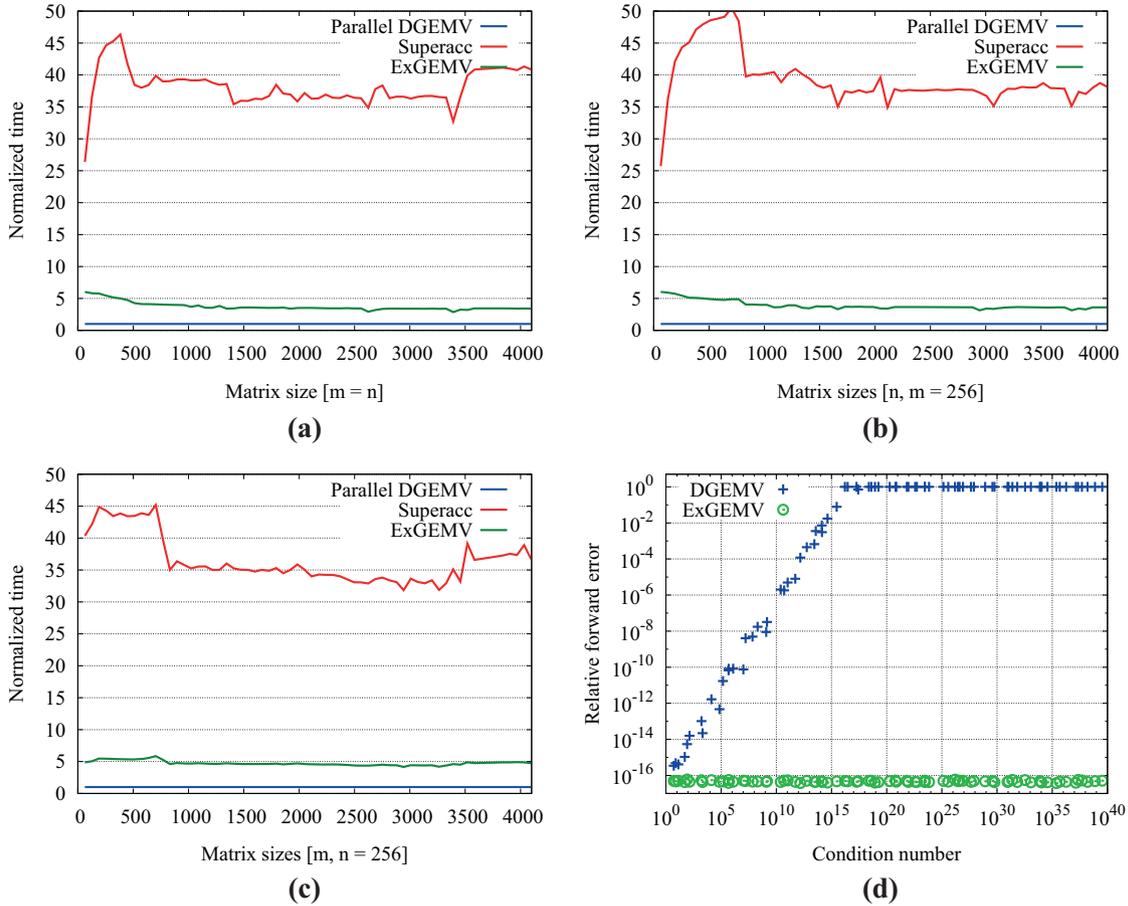
Figure 5. Performance and accuracy results of GEMV: (a) GEMV (m = n, step = 256) on K80, (b) GEMV (m = 256, n) on K80, (c) GEMV (m, n = 256) on K80, and (d) Accuracy of GEMV.

5.1. EXGEMV results

While decomposing a matrix into a lower and an upper triangular matrix using the LU factorization, the matrix-vector product involves matrices of various shapes, starting from column-panels, through squarish, and then row-panels. In our experiments, we aim to evaluate these scenarios by considering the following three test cases:

1. GEMV with square matrices;
2. GEMV with row-panel matrices, where the number of rows is fixed to $m = 256$ and the number of columns n varies; and
3. GEMV with column-panel matrices, where the number of columns is fixed to $n = 256$ and the number of rows varies.

Figure 5(a) presents the performance results achieved by the matrix-vector algorithms for square matrices $m \times n (m = n)$ composed of double precision floating-point numbers. In the caption of the following plots, "Superacc" corresponds to the accurate and reproducible matrix-vector algorithm that is solely based on superaccumulators; "EXGEMV" stands for our exact implementation,

which delivers the best performance, with FPEs of size n ($n = 2 : 8$) in conjunction with the TwoSum and TwoProd EFTs, and superaccumulators when required. The "Superacc" implementation, which can be classified as a brute-force approach to ensure reproducibility, suffers from its extensive memory usage and is an order of magnitude slower than our solution.

The performance experiments using the latter two test cases are depicted in Figure 5(b) and (c), respectively. Figure 5(c) shows rather constant execution time for all GEMV implementations. This is due to the use of a column-wide array of threads that performs local dot products within GEMV. Figure 5(b) demonstrates the outcome of applying this pool of threads iteratively over columns of the matrices of size $n \times 256$. This turns to be beneficial from the point of view of performance, as the overhead decreases with the number of columns. Figure 5(a) and (b) exhibits very close relative performance results. The best performance among the EXGEMV implementations for all the test cases is delivered by "FPE 3 + Superacc"—an FPE of size 3 using both TwoSum and TwoProd EFTs and superaccumulators, if necessary. For instance, its overhead is 4.26 times for $m = n = 4096$ compared to the non-deterministic GEMV

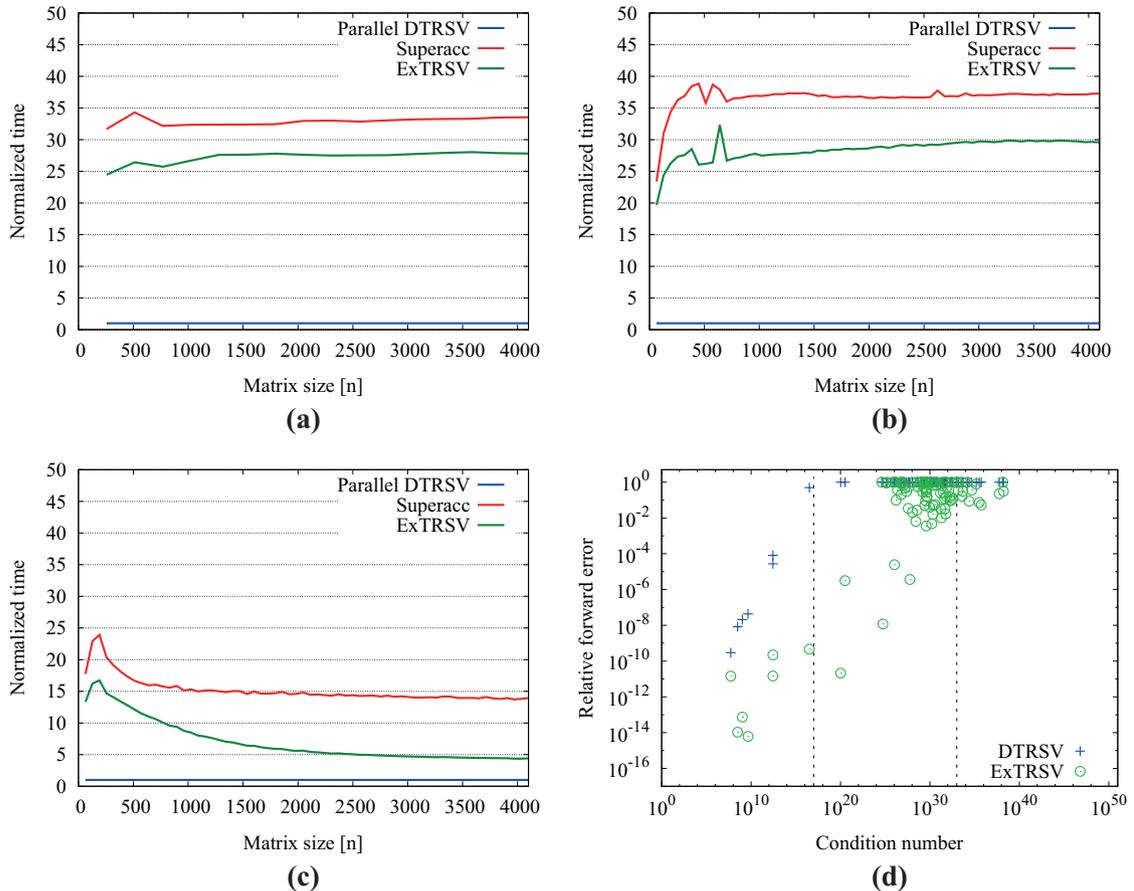**Figure 6.** Performance and accuracy results of TRSV: (a) On K20c, (b) On K80, (c) On K420, and (d) Accuracy of TRSV.

implementation. To summarize, EXGEMV delivers both reproducible and correctly-rounded results for different shapes of both transpose and non-transpose matrices.

In addition, we tested the accuracy of the double precision GEMV and the reproducible EXGEMV. Figure 5(d) demonstrates the relative forward error ($\| y^* - \hat{y} \| / \| y^* \|$, where $\hat{y}$ and y^* are the computed and exact vectors, accordingly) against the condition number of the problem. This condition number is computed with the MPFR library as a ratio of norms of the exact matrix-vector products, when the matrix and the vector are in their absolute and original values: $\| |A| \cdot |x| \| / \| A \cdot x \|$. In order to generate the ill-conditioned matrix-vector, we rely on the ill-conditioned dot product. So that, the $n - 1$ rows of the matrix A are created as random uniformly distributed numbers, for example, between 0 and 1, while one row and the vector x correspond to the ill-conditioned dot product. The exact vector y^* is computed with the MPFR library and rounded to the double precision vector. For visual representation, we replaced all the errors that exceed 1 by 1 as there is zero digit of accuracy left. Naturally, the relative forward error of GEMV strongly depends on the condition number and indicates the incorrectness of the computed results once the condition number reaches 10^{16}. Instead, our reproducible EXGEMV ensures both correctly-rounded

and reproducible results as our approach preserves every bit of the computed result until its final rounding.

5.2. EXTRSV *results*

Figure 6(a) shows a performance improvement around 35%, compared with the results of our previous work (Iakymchuk et al., 2015b), delivered by EXTRSV on K20c. This improvement is due to a new way we are handling errors produced by one of the two outputs of TwoProd. In our previous work (Iakymchuk et al., 2015b), these errors were accumulated to the FPE starting from its head. However, most of the time this solution was suboptimal as the error was much smaller that the head of the FPE, leading to further error propagation along the FPE. Based on that report, we decided to add these errors closer to the tail of the expansion, namely in the last two or three slots of the expansion. The difference in terms of performance among FPEs of various sizes is less apparent as they follow the same pattern with respect to the error propagation.

Figure 6(a) and (b) shows that there is still work to do on architecture-specific implementations, since for large matrices, EXTRSV on K420 only reports a 4× overhead, while on both K20c and K80 the overhead reaches 25×. Although OpenCL ensures portability, unfortunately, it

does not come with performance portability. We have already tuned our code on higher level for various GPU architectures. However, it is clear that, for some kernels as EXTRSV, additional architecture-specific implementations should be provided.

Figure 6(d) provides a comparison for the relative forward error versus the condition number of the double precision, TRSV, with the proposed reproducible, EXTRSV, substitution algorithms. The relative forward error corresponds to $\| x^* - \hat{x} \| / \| x^* \|$, where $\hat{x}$ and x^* are, respectively, the computed and exact solutions. We use the following Skeel formula (Higham, 2002; Skeel, 1979) to calculate the condition number of a linear system $Ax = b$ with real coefficients
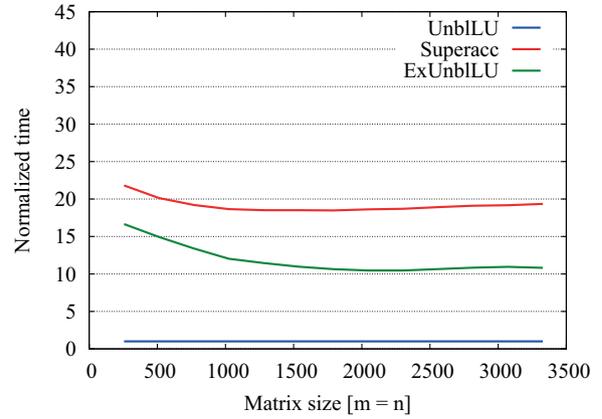
$$\text{cond}(A, x) = \frac{\| \, |A^{-1}| |A| |x| \, \|}{\| x \|}$$

To perform those tests, we wrote a random generator of ill-conditioned triangular systems using the algorithm proposed by Louvet (2007); the random generator ensures that both the matrix L and the right-hand side vector b are composed of double precision floating-point numbers. In order to compute the exact solution x^*, we rely on the MPFR library and rounded the result back to double precision. For the errors greater than 1, we cut them to 1 as the results are obviously incorrect. This experiment reveals that the relative forward error is proportional to the condition number and most importantly to the rounding error $u = 2^{-53}$. Altogether EXTRSV delivers most of the time better accuracy as the double precision triangular solver. Indeed, the accuracy of the classic triangular solver can be improved via double-double precision. However, this approach is already $9\times$ slower than TRSV and, moreover, it does not provide any guarantees on the reproducibility of the results. Therefore, we propose to apply iterative refinement, see Algorithm 6, in order to improve the accuracy of EXTRSV.
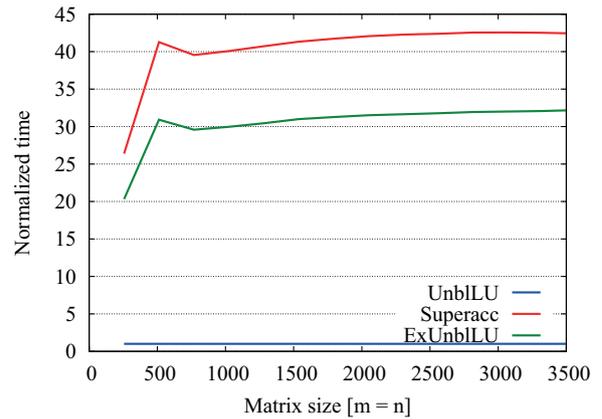
5.3. EXLU results

During the development of the reproducible LU factorization, we discovered a limitation of OpenCL and, therefore, the ExBLAS routines. This drawback refers to the possibility of passing a reference to a location within a matrix ($A[i * (lda + 1)]$) as a kernel argument. We overcame this issue by passing an offset from the beginning of the matrix to the desired location. That introduces a change in the API of the BLAS to include a few new arguments.
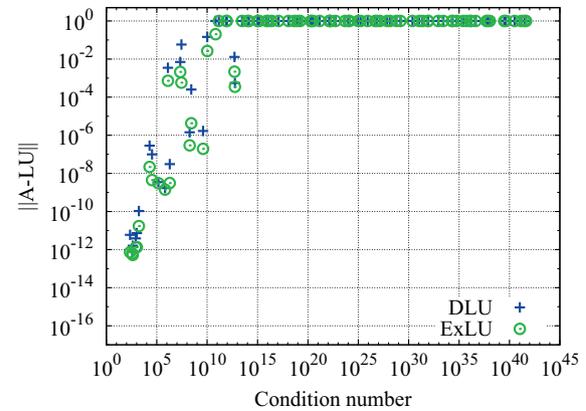
Figure 7(a) and (b) reports the execution times obtained by the unblocked algorithmic variant for the LU factorization with partial pivoting as a function of the matrix size ($m = n$). The execution times of the EXLU implementations are much larger than those of each ExBLAS routine for a specific problem size. That is because EXLU requires $n - 1$ executions of each underlying ExBLAS routine for a matrix of size n. For instance, EXTRSV finds the solution of a triangular system using

Figure 7. Performance and accuracy results of the unblocked LU factorization: (a) LU on K420, (b) LU on K80, and (c) Accuracy of the LU factorization.

matrices of sizes from 1×1 till $(n - 1) \times (n - 1)$. In order to ensure reproducibility, the proposed algorithm based on EXLU requires extra computations and memory. The outcome is visible in the form of $11\times$ and $32\times$ performance overheads on K420 and K80, respectively, compared to the double precision unblocked LU factorization; the latter overhead is the result of the large performance overhead caused by EXTRSV.

Furthermore, we study the accuracy of the non-deterministic and reproducible implementations of the unblocked LU factorization (Algorithm 7) on matrices with various condition numbers, covering a range from 10^2 to 10^{41}; these results are depicted in Figure 7(c). To produce these matrices, we modify the ill-conditioned triangular matrix generator to cover a more general case. We compute the error $\| PA - LU \|$ as an infinity norm using the MPFR library. For those errors that exceed 1, we round them down to 1. As for EXTRSV, EXLU delivers more accurate results than the double precision LU factorization. However, due to the division by the diagonal element in EXTRSV, the correct rounding of the results is not guaranteed in EXLU. We foresee to study and enhance this by improving the accuracy of the underlying EXTRSV using iterative refinement. Nevertheless, the EXLU implementations always deliver reproducible results.

6. Related work

There already exist some solutions to ensure numerical reproducibility, which can be classified with respect to their implementations into sequential, vectorized, multi-threaded on multicores, and distributed. Those solutions, at first, addressed sequential and parallel summation. For example, Rump (2009) and Zhu and Hayes (2010) focus on hybrid solutions that store the sum as floating-point numbers of fixed exponent without completely avoiding the previous drawbacks. Arbitrary precision libraries—like MPFR (Fousse et al., 2007)—are able to provide correct rounding. However, they are not designed to achieve acceptable performance for reproducible results. Moreover, MPFR is also not multi-threaded. As a result, EXSUM is three orders of magnitude faster than MPFR.

To ensure reproducibility, Intel introduced a Conditional Numerical Reproducibility (CNR) option in its MKL library (Intel®, 2012), but this does not provide any warranty on the accuracy. In addition, a CNR-enabled execution of MKL kernels incurs a large performance overhead compared to the existing alternatives. For instance, MKL's `dasum()` with CNR enabled is two times slower than both the conventional MKL's `dasum()` and EXSUM.

There are two other academic efforts to guarantee reproducibility. The first by Demmel and Nguyen (2013), which is built upon Rump (2009), proposed a family of algorithms for reproducible summation in FP arithmetic. Demmel and Nguyen improved their algorithms (Demmel and Nguyen, 2015) by passing over the input data only once. This approach reduces the overhead to roughly 20%. Arteaga et al. (2014) used this approach with improved communication and obtained the same accuracy with roughly 10% overhead. Demmel and Nguyen applied their approach to the absolute value summation, dot product, and 2-norm in the ReproBLAS library. Recently, they introduced Ahrens et al.'s (2016) new concepts—such as slice (significant bits in a bin) and indexed type/sum—and reformulated the algorithms (Demmel and Nguyen, 2015) in terms of these concepts. They introduced the conversion algorithm that leads to improved accuracy of their reproducible summation, handling of exceptions (not implemented in software), propagation of overflows. ReproBLAS was extended to include sequential matrix-vector and matrix-matrix products. However, ReproBLAS targets only CPU and does not exploit the data parallelism available in BLAS routines.

The approach by Chohra et al. (2016) ensures reproducibility of both sequential and parallel summations. Chohra et al. proposed to use a combination of existing solutions such as FastAccSum (Rump, 2009) and OnlineExact (Zhu and Hayes, 2010) depending on the size of the input vector. They cover absolute value summation, 2-norm, dot product and matrix-vector product, provided as the RARE-BLAS library. RARE-BLAS, which is not publicly available, runs on Intel server CPUs and Intel Xeon Phi coprocessors. The reproducibility of 2-norm is an open question as the returned result is faithfully rounded.

Alternatively, Collange et al. (2015) proposed a multi-level approach to compute the reproducible summation. Neal (2015) integrated this concept in its scalar superaccumulators of different sizes for the summation problem in the R package. This approach is based on FPEs and Kulisch superaccumulators, discussed in Section 2. We showed that the numerical reproducibility and bit-perfect (correct rounding) accuracy can be achieved without performance degradation for large sums with dynamic ranges of up to 90 orders of magnitude on a large variety of computer architectures, including conventional clusters, GPUs, and coprocessors. By tracking every bit of information for a product of two FP numbers, we extended this approach to the dot product, blocked triangular solver (Iakymchuk et al., 2015b), and matrix-matrix multiplication (Iakymchuk et al., 2016b). The initial release (v0.1) of the Exact BLAS (ExBLAS) library (Iakymchuk et al., 2016a) included implementations of only one algorithmic variant of GEMM and TRSV. The latest release (v1.0), which is available at `https://exblas.lip6.fr`, covers all algorithmic variants of GEMV and TRSV. The aforementioned unblocked LU factorization is also available at the same location as a separate package.

7. Conclusions and future work

In numerical linear algebra, algorithms and their corresponding implementation exhibit a modular and hierarchical structure. This property permits to assemble higher-level algorithms on top of fundamental kernels. In this article, we leverage this layered organization of linear algebra algorithms to derive reproducible algorithmic variants for the LU factorization. We made the demonstration that a class of reproducible higher-level linear algebra operations can be constructed by following the bottom-up approach. Through our development, we noticed that in the context of reproducibility, this process is not "automatic" and should be handled carefully especially if accuracy is also a major concern.

As a case study, we considered the unblocked *jik* variant of the LU factorization that relies upon the Level-1/2 BLAS routines DOT, SCAL, GEMV, and TRSV. In particular, we derived the accurate and reproducible matrix-vector product and provided performance results on NVIDIA GPUs. Additionally, we have improved the performance of EXTRSV and drew a strategy to enhance its accuracy via iterative refinement, potentially reaching correctly-rounded results.

Although some of the performance results can be argued in some scenarios, we advocate that we should not trade-off numerical stability and reproducibility to performance. As both goals are conflicting, users should start considering that some of the performance benefit brought by today's multicore has to be devoted to the later one.

Declaration of Conflicting Interests

Funding

ORCID iD

Roman Iakymchuk https://orcid.org/0000-0003-2414-700X

References

Ahrens P, Nguyen HD and Demmel J (2016) *Efficient reproducible floating point summation and BLAS*. Technical Report UCB/EECS-2016-121, EECS Department, University of California, Berkeley.

Arteaga A, Fuhrer O and Hoefler T (2014) Designing bit-reproducible portable high-performance applications. In: *Proceedings of the 2014 IEEE 28th international parallel and distributed processing symposium*, *IPDPS '14*, Washington, DC, USA, pp. 1235–1244. IEEE Computer Society.

Bainville E (2010) OpenCL GEMV. Available at: http://www.bealto.com/gpu-gemv_v3.html (accessed 12 February 2018).

Bientinesi P, Quintana-Ortí ES and van de Geijn RA (2005) Representing linear algebra algorithms in code: the FLAME application programming interfaces. *ACM Transactions on Mathematical Software* 31(1): 27–59.

Chohra C, Langlois P and Parello D (2016) Efficiency of reproducible Level 1 BLAS. In: *LNCS. Scientific Computing, Computer Arithmetic, and Validated Numerics: SCAN 2014*, Würzburg, Germany, 21–26 September 2014, Revised Selected Paper. Vol. 9553, pp. 99–108. Springer-Verlag.

Collange S, Defour D, Graillat S, et al. (2015) Numerical reproducibility for the parallel reduction on multiband many-core architectures. *Parallel Computing* 49: 83–97.

Demmel J and Nguyen HD (2013) Fast reproducible floating-point summation. In: *Proceedings of the 21st IEEE symposium on computer arithmetic*, Austin, Texas, USA, pp. 163–172.

Demmel J and Nguyen HD (2015) Parallel reproducible summation. *IEEE Transactions on Computers* 64(7): 2060–2070.

Dongarra JJ, DuCroz J, Hammarling S, et al. (1990) A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software* 16(1): 1–17.

Dongarra J, Hittinger J, Bell J, et al. (2014) *Applied Mathematics Research for Exascale Computing*. Washington, DC: U.S. Department of Energy.

Fousse L, Hanrot G, Lefèvre V, et al. (2007) MPFR: a multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software* 33(2): 13.

Golub GH and Loan CFV (1996) *Matrix Computations*, 3rd ed. Baltimore: The Johns Hopkins University Press.

Haidar A, Dong T, Luszczek P, et al. (2015) Batched matrix computations on hardware accelerators based on GPUs. *International Journal of High Performance Computing Applications* 29(2): 193–208.

Hida Y, Li XS and Bailey DH (2001) Algorithms for quad-double precision floating point arithmetic. In: *Proceedings of the 15th IEEE symposium on computer arithmetic*, Los Alamitos, CA, USA, pp. 155–162. IEEE Computer Society Press.

Higham NJ (2002) *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Philadelphia: Society for Industrial and Applied Mathematics (SIAM).

Iakymchuk R, Collange S, Defour D, et al. (2015a) 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, 15–20 November 2015.

Iakymchuk R, Collange S, Defour D, et al. (2015b) Reproducible triangular solvers for high-performance computing. In: *Proceedings of the 12th international conference on information technology: new generations (ITNG 2015), special track on: wavelets and validated numerics*, Las Vegas, Nevada, USA, 13–15 April 2015, pp. 353–358. HAL: hal-01116588v2.

Iakymchuk R, Collange S, Defour D, et al. (2016a) ExBLAS (Exact BLAS) library. Available at: https://exblas.lip6.fr/ (accessed 15 February 2018).

Iakymchuk R, Defour D, Collange S, et al. (2016b) Reproducible and accurate matrix multiplication. In: *LNCS. Scientific Computing, Computer Arithmetic, and Validated Numerics: SCAN 2014*, Würzburg, Germany, 21–26 September 2014, Revised Selected Paper. Vol. 9553. pp. 126–137.

ICL Team (2016) MAGMA project home page. Available at: http://icl.cs.utk.edu/magma/ (accessed 02 February 2017).

IEEE (1985) *IEEE standard for binary floating-point arithmetic. ANSI/IEEE Standard, Std 754-1985*. New York.

Intel® (2012) Conditional Numerical Reproducibility (CNR) in Intel MKL 11.0. Available at: https://software.intel.com/en-us/articles/introduction-to-the-conditional-numerical-reproducibility-cnr (accessed 12 February 2018).

Knuth DE (1997) *The Art of Computer Programming (Vol. 2). Seminumerical Algorithms*, 3rd ed. Boston: Addison-Wesley.

Kulisch U and Snyder V (2011) The exact dot product as basic tool for long interval arithmetic. *Computing* 91(3): 307–313.

Li XS, Demmel JW, Bailey DH, et al. (2002) Design, implementation and testing of extended and mixed precision BLAS. *ACM Transactions on Mathematical Software* 28(2): 152–205.

Louvet N (2007) *Algorithmes compensés en arithmétique flottante: Précision, validation, performances*. PhD Thesis, UPVD. Available at: http://perso.ens-lyon.fr/nicolas.louvet/.

Lucas R, et al. (2014) Top Ten Exascale Research Challenges. DOE ASCAC Subcommitee Report.

Muller JM, Brisebarre N, de Dinechin F, et al. (2010) *Handbook of Floating-Point Arithmetic*. Birkhäuser.

Neal RM (2015) *Fast exact summation using small and large super-accumulators*. Technical Report, University of Toronto.

Ogita T, Rump SM and Oishi S (2005) Accurate sum and dot product. *SIAM Journal on Scientific Computing* 26.

Ortega JM (1988) The *ijk* forms of factorization methods I. Vector computers. *Parallel Computing* 7: 135–147.

Rump SM (2009) Ultimately fast accurate summation. *SIAM Journal on Scientific Computing* 31(5): 3466–3502.

Skeel RD (1979) Scaling for numerical stability in Gaussian elimination. *Journal of the Association for Computing Machinery* 26(3): 494–526.

van de Geijn RA and Quintana-Ortí ES (2008) The science of programming matrix computations. Available at: www.lulu.com.

Zhu YK and Hayes WB (2010) Algorithm 908: online exact summation of floating-point streams. *ACM Transactions on Mathematical Software* 37(3): 1–37.

Author biographies

Roman Iakymchuk is a Researcher at KTH Royal Institute of Technology. He obtained a Dr.rer.nat. (equiv. to PhD) in computer science from the RWTH Aachen University. During his PhD, he did an internship at the National Institute of Informatics, Tokyo, Japan. Between 2012 and 2015, he was a post-doctoral researcher in France at the Institut Mines-Télécom–Télécom SudParis and the University Pierre and Marie Curie (now the Sorbonne University). His research interests span the fields of programming models, high-performance computing, and numerical linear algebra.

Stef Graillat received his PhD degree in 2005 from Université de Perpignan, France. He is a Professor of computer science at Sorbonne Université and a deputy director of LIP6 laboratory. His research interests are in computer arithmetic, floating-point arithmetic and validated computing.

David Defour is an Associate Professor at the University of Perpignan, Perpignan, France. He received his PhD degree in computer science from ENS, Lyon, France, in 2003. He is currently leading the evaluation committee of the High Performance Computing Center (HPC@LR). Defour is primarily interested in floating-point arithmetic, numerical algorithms, computer architecture, and microarchitecture for high performance computing and specifically GPUs.

Enrique S Quintana-Ortí received the bachelor's and PhD degrees in computer sciences from the UPV, Spain, in 1992 and 1996, respectively. After spending the last 23 years as Assistant, Associate and Full Professor at Universidad Jaume I, Castellón de la Plana, Spain, he recently came back to UPV as Full Professor in computer architecture. He has published more than 200 papers in international conferences and journals, and has contributed to software libraries such as PLiC/SLICOT, MAGMA, FLARE, BLIS, and libflame for control theory and parallel linear algebra. His current research interests include parallel programming, linear algebra, energy consumption, transprecision computing and bioinformatics, and advanced architectures and hardware accelerators.