Hierarchical Approach for Deriving a Reproducible LU factorization

Roman Iakymchuk, Stef Graillat, David Defour, Erwin Laure, Enrique Quintana-Ortí

To cite this version:

Roman Iakymchuk, Stef Graillat, David Defour, Erwin Laure, Enrique Quintana-Ortí. Hierarchical Approach for Deriving a Reproducible LU factorization. 2016. hal-01419813v1

HAL Id: hal-01419813
https://hal.archives-ouvertes.fr/hal-01419813v1

Submitted on 20 Dec 2016 (v1), last revised 18 Apr 2017 (v4)

Hierarchical Approach for Deriving a Reproducible LU factorization

Roman Iakymchuk[a], Stef Graillat[b], David Defour[c], Erwin Laure[a], Enrique S. Quintana-Ortí[d]

[a] *KTH Royal Institute of Technology, CSC, CST/PDC, Stockholm, Sweden*
[b] *Sorbonne Universités, UPMC Univ Paris 06, CNRS, UMR 7606, LIP6, Paris, France*
[c] *Université de Perpignan, DALI-LIRMM, Perpignan, France*
[d] *Universidad Jaime I, 12.071-Castellón, Spain*

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 into a (blocked) high performance and stable algorithm for the 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 formidable performance rate, getting *accurate* and *reproducible*[1] results in floating-point arithmetic will represent two considerable challenges [9,

Email addresses: `riakymch@kth.se` (Roman Iakymchuk), `stef.graillat@lip6.fr` (Stef Graillat), `david.defour@univ-perp.fr` (David Defour), `erwinl@pdc.kth.se` (Erwin Laure), `quintana@uji.es` (Enrique S. Quintana-Ortí)

[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.

26]. 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 component. Although there exist implementations of this factorization that deliver high performance on a variety of processor architectures –including general-puropose multicore processors, Intel Xeon Phi, and graphics processors (GPUs)– their *reproducibility* and, even more, *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 multi-core 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 [13, 27].

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* [10]) that serve as building blocks in the LU 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 know as *jik* or *jki* variant [31]). The unblocked version is important for the solution of "batched" linear systems [14], 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, i.e. reproducible and correctly-rounded, parallel reduction (EXSUM) [6] in order to derive an exact dot product (EXDOT). For this purpose, we combine the multi-level parallel reduction algorithm with the traditional EFT, called `TwoProd` [30], 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, 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) [18]. We tackle the accuracy problem by applying a few iterations of iterative refinement.

- Finally, we integrate partial pivoting [13] 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.

The paper is organized as follows. Section 2 reviews several aspects of computer arithmetic, in particular floating-point expansions and the Kulisch superaccumulator. Section 3 presents the ExBLAS library with its current set of routines. Section 4 is devoted to the presentation of a 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 paper, we consider the double precision format (`binary64`) as specified in the IEEE-754 standard. The 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 [21]. Hence, the accuracy and reproducibility of floating-point operations strongly depend on their order [16, 28]. 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 lie at the foundation of our present work. Concretely, floating-point expansions 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 [6] (Section 3.1) and dot product [17] (Section 3.2).

2.1. Floating-Point Expansion

Floating-point expansions (FPE) allows 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 [25] and [15], respectively. The conventional EFT for the addition (`TwoSum`) is given in Alg. 1 [23] and, for the multiplication (`TwoProd`), in Alg. 2 [30]. Alg. 1 computes the addition of two floating-point numbers a and b and returns the result r and the error e such that r and e do not overlap. Similarly, `TwoProd` performs the multiplication of two floating-point numbers a and b. 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: EFT for the summation of two floating-point numbers.

```
Function [$r, e$] = TwoSum($a, b$)
    $r := a + b$
    $z := r - a$
    $e := (a - (r - z)) + (b - z)$
```

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

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

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 `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 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 significantly 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 `binary64` elements, Kulisch [24] proposed to use a 4,288-bit accumulator. The addition of products of `binary64` values is performed without loss of information by accumulating every floating-point number in the superaccumulator; see Fig. 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, and turns vectorization difficult, because of indirect memory accesses.

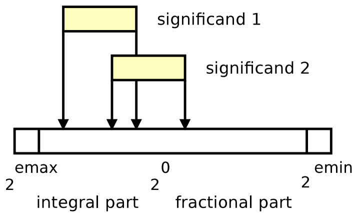


Figure 1: Kulisch Superaccumulator.

3. Exact BLAS Library (ExBLAS) on GPUs

This section briefly reviews the prototype implementation of the Exact BLAS (ExBLAS) library [19] that underlies our LU factorization. We begin with the parallel reduction and dot product, as they are two fundamental BLAS kernels, and continue then with the triangular solver, outlining the improvements in this paper compared with prior work. We also extend our ideas to scaling a vector by the inverse of a scalar and the matrix-vector product, contributing novel 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 [6, 19] we introduced a multi-level approach to summation that combines the FPE and superaccumulator algorithms. The procedure splits the computation into five levels: filtering, private superaccumulation, local superaccumulation, global superaccumulation, and rounding. This 5-stage decomposition is advantageous for modern parallel architectures such as GPUs, and it can accommodate nested parallelism.

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

Algorithm 3: Floating-point expansion of size p.

```
Function ExpansionAccumulate(x)
    for $i = 0 \to p - 1$ do
        $(a_i, x) :=$ TwoSum$(a_i, x)$
    end
    if $x \neq 0$ then
        Superaccumulate$(x)$
    end
```

In the second level, if the last rounding error x is non-zero (see Alg. 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, e.g. cache or shared memory, or global memory.

In the third level, k private superaccumulators are merged into a single local superaccumulator, one per group of threads. In the fourth level, 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 level, in order to obtain the correctly-rounded result.

We develop 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 superaccumualtor.

3.2. Exact Dot Product: EXDOT

We apply the multi-level parallel reduction algorithm [6] to the dot product [17] 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, in the call to SCAL from the unblocked LU factorization, see Alg. 7, the vector is scaled by the inverse of the diagonal element, so that the result is not correctly-rounded due to the two rounding errors induced by one division and one multiplication. To ensure correctly-rounded and, therefore, reproducible results in this case, we propose a version of SCAL (EXINVSCAL) that 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 an 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:

$$y := \alpha A \cdot x + \beta y, \quad \text{or} \quad y := \alpha A^T \cdot x + \beta y, \tag{1}$$

where α and β are scalars, x and y are vectors, and A is a matrix.

We derive a reproducible and accurate algorithm for EXGEMV by combining its two-kernel algorithmic variants from [3] to render 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 detailed explanation of this merge into a GPU kernel.

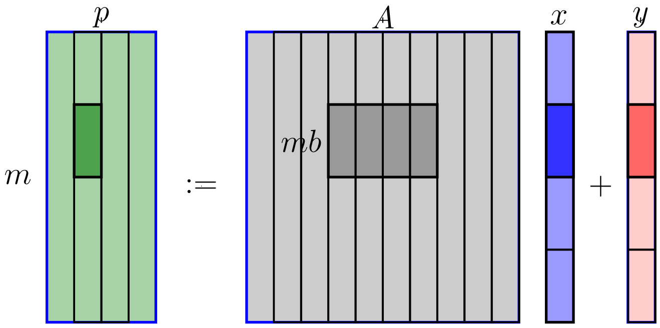


Figure 2: Work distribution in the matrix-vector product.

The proposed OpenCL implementation of the matrix-vector product splits the computations into blocks, so that each workgroup of threads compute a certain part of the output vector y. Fig. 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 pre-loading 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 that multiple workgroups per multiprocessor proceed concurrently. This strategy also 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 (1).

3.5. Reproducible Triangular Solver

The triangular solve involving a lower triangular coefficient matrix is one of the building blocks for the unblocked LU factorization. We next provide the multi-level reproducible approach for triangular systems with a single right-hand side [18] 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 Alg. 4, where the elements of x are computed from first to last.

Algorithm 4: Forward substitution.

```
$x_1 := b_1/l_{11}$
for $i = 2 : n$ do
    $s := b_i$
    for $j = 1 : i - 1$ do
        $s := s - l_{ij}x_j$
    end
    $x_i := s/l_{ii}$
end
```

Alg. 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 ($\widehat{s} := RNDN(acc(k))$). To note, rounding to double is performed only once – at the end of computing each element of the solution.
4. Perform the division by the corresponding diagonal element ($\widehat{x_i} := \widehat{s}/l_{ii}$).

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 the same result, and, therefore, it is reproducible. Let us

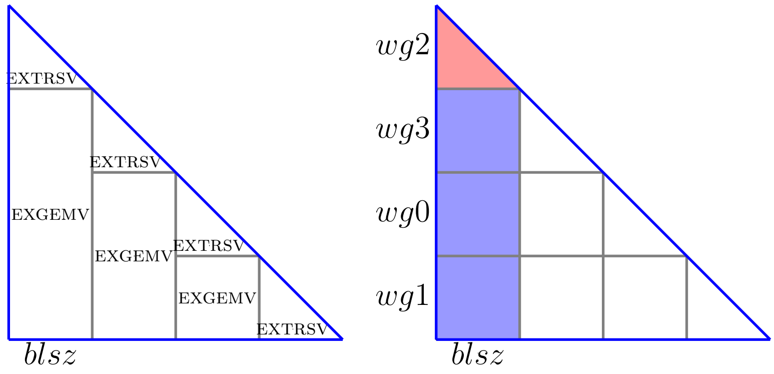
Algorithm 5: Blocked EXTRSV: *blsz* stands for a block size; *acc* refers to an array of superaccumulators of size n; $acc(k)$ corresponds to one superaccumulator at the position k in this array.

```
for $i = 0 : blsz : n$ do
    Local EXTRSV
        for $k = i : i + blsz$ do
            for $j = 1 : k - 1$ do
                $[r, e] := \texttt{TwoProd}(l_{kj}, -x_j)$
                $ExpansionAccumulate(r)$          accumulate $r$ using Alg. 3
                $ExpansionAccumulate(e)$          accumulate $e$ using Alg. 3
            end
            $ExpansionAccumulate(b_k)$
            $\widehat{s} := RNDN(acc(k))$            round to double a superaccumulator
            $x_k := \widehat{s}/l_{kk}$
        end
    Local EXGEMV
        for $k = i + blsz : n$ do
            for $j = i : i + blsz$ do
                $[r, e] := \texttt{TwoProd}(l_{kj}, -x_j)$
                $ExpansionAccumulate(r)$
                $ExpansionAccumulate(e)$
            end
        end
end
```

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 $\widehat{s} := RNDN(s)$ is reproducible as it is a correctly-rounded operation. Finally $x_i := \widehat{s}/l_{ii}$ is reproducible as it is the result of a correctly-rounded division between two reproducible quantities. □

Our approach is based on partitioning the matrix and both the right-hand side and the solution vectors into blocks, and then organizing computations on those blocks as in Figs. 3a and 3b; so that 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 Fig. 3b) 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 [18], in the new version of ExBLAS we covered all the four variants of the triangular



(a) Kernel-wise **(b)** Work-group-wise.

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.

solver as follows

$$T \cdot x = b, \quad \text{or} \quad T^T \cdot x = b, \tag{2}$$

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 refinement based on the ExBLAS routines, as described in Alg. 6. The overhead of the iterative refinement can be diminished by locating the origin of the accuracy problem and, consequently, applying iterative refinement directly on the diagonal blocks right after the local EXTRSV.

Algorithm 6: The reproducible triangular solver with iterative refinement.

```
$\widehat{x} := L^{-1}b$            (EXTRSV)
for $i = 1 : nbiter$ do
    $r := b - L\widehat{x}$          (EXGEMV)
    $d := L^{-1}r$                   (EXTRSV)
    $\widehat{x} := \widehat{x} + d$ (EXAXPY)
end
```

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

PROOF. The first step uses EXTRSV to compute $\widehat{x} := L^{-1}b$, so $\widehat{x}$ 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. □

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 $m \times m$ permutation P such that $PA = LU$. In order to compute this decomposition, we consider an unblocked left-looking variant, illustrated in Alg. 7 using the FLAME notation [4, 12]. This notation makes it easier to identify which regions of the matrix are updated and used; see Fig. 4. In Alg. 7, *size*(A) indicates the number of columns of the matrix A. Before the computation commences, 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.

Alg. 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. We benefit from this layered hierarchy to construct a reproducible algorithmic variant of the unblocked LU factorization, Alg. 7, on top of the underlying ExBLAS routines: EXTRSV, EXDOT, EXINVSCAL, and EXGEMV.

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

```
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 $0 \times 0$, $p_T$ 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  $\alpha_{11}$ and $\pi_1$ are scalars

    $\left(\begin{array}{c} a_{01} \\ \hline \alpha_{11} \\ \hline a_{21} \end{array}\right) := P(p_0) \left(\begin{array}{c} a_{01} \\ \hline \alpha_{11} \\ \hline a_{21} \end{array}\right)$
    $a_{01} := L_{00}^{-1} a_{01}$                    $(trsv)$
    $\alpha_{11} := \alpha_{11} - a_{10}^T a_{01}$             $(dot)$
    $a_{21} := a_{21} - A_{20} a_{01}$               $(gemv)$
    $\pi_1 := PivIndex \left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)$
    $\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right) := P(\pi_1) \left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)$
    $a_{21} := a_{21} / \alpha_{11}$                   $(scal/invscal)$
    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
```

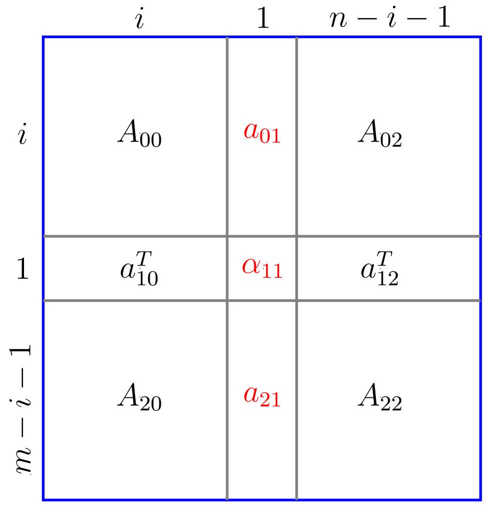


Figure 4: Partitioning of matrix A.

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

1. Searching for the maximum element in absolute value within the subdiagonal 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. In conclusion, all computational steps of the proposed unblocked LU factorization rely upon their reproducible counterparts, such as EXTRSV, EXDOT, EXINVSCAL, and EXGEMV, plus the reproducible strategy implementing partial pivoting. Therefore, we have successfully removed all sources of indeterminism, while efficiently exploiting data-parallelism within each basic block, and the result computed by Alg. 7 is reproducible.

5. Performance 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 Tab. 1.

Table 1: Hardware platforms employed in the experimental evaluation.

NVIDIA Quadro K420	192 CUDA cores	0.780 GHz
NVIDIA Tesla K20c	2,496 CUDA cores	0.706 GHz
NVIDIA Tesla K80	4,992 CUDA cores with a dual-GPU design	0.560-0.875 GHz

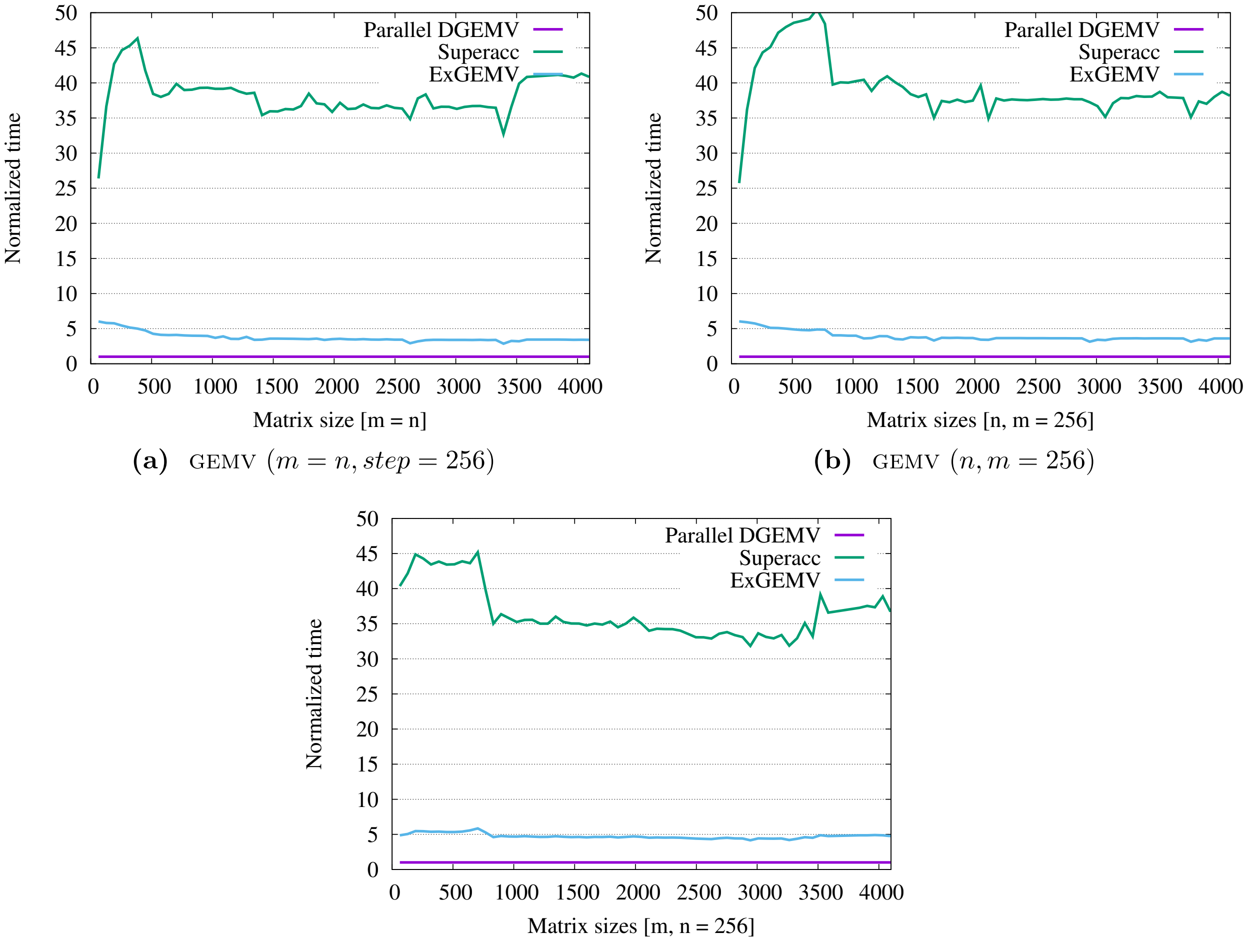
We develop unique OpenCL implementations of each algorithm on GPUs and tuned these implementations – e.g. by promoting loop unrolling or changing workgroup size, etc. – for a particular architecture in order to optimize performance. Moreover, our implementations strive to deliver the best performance through utilizing all the underlying resources of the target GPUs: SIMD instructions, FMAs, private and local memory, as well as atomic instructions. We verify the accuracy of our implementations by comparing the computed results with those produced by the multiple precision sequential library MPFR for CPUs.

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 (Alg. 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.

While decomposing a matrix into a lower and an upper triangular matrices 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;
3. GEMV with column-panel matrices, where the number of columns is fixed to $n = 256$ and the number of rows varies.

Fig. 5a 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



(a) GEMV $(m = n, step = 256)$

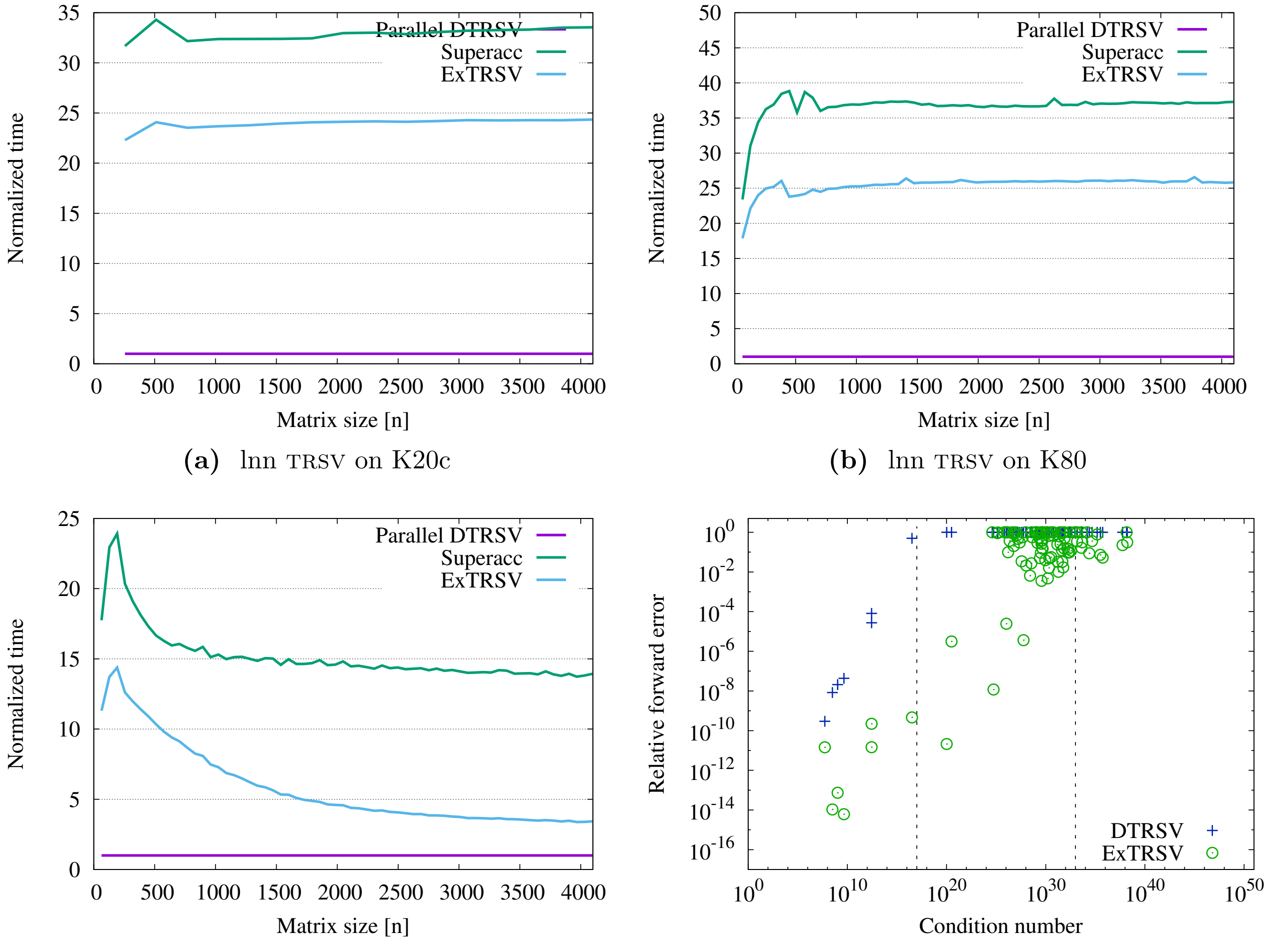
(b) GEMV $(n, m = 256)$

(c) GEMV $(m, n = 256)$

Figure 5: Performance of GEMV on K80.

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 `TwoProd` EFT 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 Figs. 5b and 5c, respectively. Fig. 5c 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. Fig. 5b 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 perspective of performance, as the overhead decreases with the number of columns. Fig. 5a



(a) lnn TRSV on K20c

(b) lnn TRSV on K80

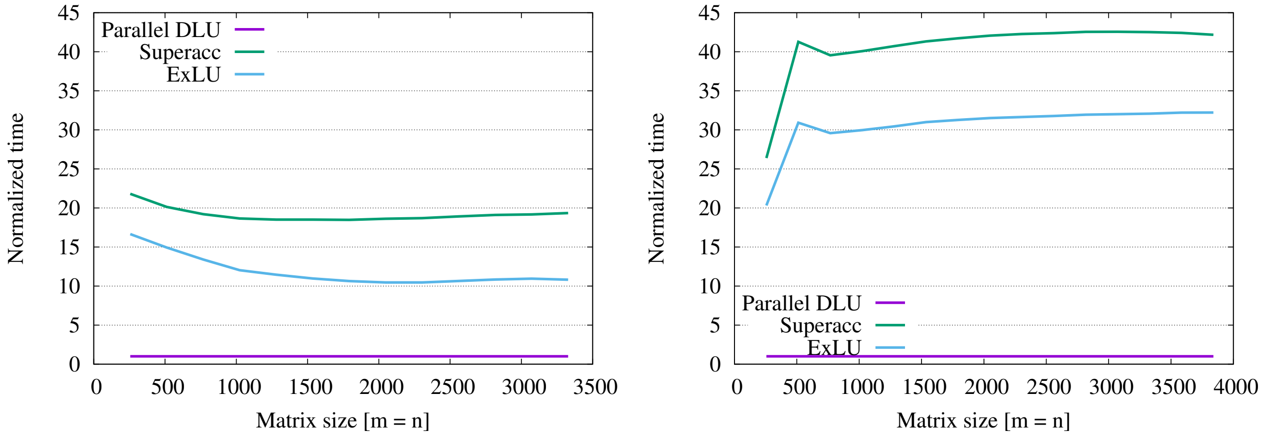
(c) lnu TRSV on K420

(d) Accuracy of TRSV with respect to the condition number of the matrix A

Figure 6: Performance and accuracy results of TRSV.

and Fig. 5b show very close relative performance results. The best performance among the EXGEMV implementations for all the test cases is delivered by “FPE3 + Superacc”. For instance, its overhead is 4.26 times for $m = n = 4096$ compared to the non-deterministic GEMV implementation. To summarizes, EXGEMV delivers both reproducible and correctly-rounded results for different shapes of both transpose and non-transpose matrices.

Fig. 6a shows a performance improvement around 35 %, compared with the results in [18], delivered by EXTRSV on K20c. This is achieved thanks to some optimisation strategies, concretely the optimized handling of errors as one of the two outputs of `TwoProd`. In [18] these errors were accumulated to the FPE starting from its head. However, this may not be practical as the error may have different exponent and, therefore, may not be added to the head, requiring that it is propagated further in the FPE. Because of that, 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



(a) EXLU on K420 **(b)** EXLU on K80

Figure 7: Performance of EXLU.

apparent as they follow the same pattern with respect to the error propagation.

Figs. 6a to 6c 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 to be provided.

Concerning the accuracy of EXTRSV, Fig. 6d presents the relative forward error of the double precision (TRSV) and our reproducible (EXTRSV) substitution algorithms versus the condition number. For those errors greater than one, we cut them to one as the results are obviously incorrect. This experiment reveals that the relative forward error is proportional to the condition number, but even more to the rounding error $u = 2^{-53}$. Altogether EXTRSV delivers the same or often 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 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 Alg. 6, in order to improve the accuracy of EXTRSV.

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.

Fig. 7 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 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 matrices of sizes from 1×1 till $(n-1) \times (n-1)$. In order to ensure reproducibility, our algorithms underlying EXLU require more computations to be performed and utilize a larger amount of memory. The outcome is visible in the form of 11× and 32× 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. Due to the division by the diagonal element in EXTRSV, the correct rounding of the results may not be 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 producibility, which can be classified with respect to their implementations into sequential, vectorized, multithreaded on multicores, and distributed. Those solutions, at first, addressed sequential and parallel summation. For example, [32, 33] 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 [11] – 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 [22], 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 roughly $2x$ slower than both the conventional MKL's `dasum()` and EXSUM.

There are two other academic efforts to guarantee reproducibility. The first [7], which is built upon [32], proposed a family of algorithms for reproducible summation in FP arithmetic. Demmel and Nguyen improved their algorithms [8] by passing over the input data only once. This approach reduces the overhead to roughly 20 %. Arteaga et al. [2] 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 [1] new concepts – such as slice (significant bits in a bin) and indexed type/sum – and reformulated the algorithms [8] 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 in [5] ensures reproducibility of both sequential and parallel summations. Chohra et. al. proposed to use a combination of existing solutions such as FastAccSum [32] and OnlineExact [33] 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. [6] proposed a multi-level approach to compute the reproducible summation. Neal [29] integrated this concept in its scalar superaccumulators of different sizes for the summation problem in the R package. This approach is based on FP expansions 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 variety of computer architectures, including conventional clusters, GPUs, and co-processors. By tracking every bit of information for a product of two FP numbers, we extended this approach to the dot product, blocked triangular solver [18], and matrix-matrix multiplication [20]. The initial release (v0.1) of the Exact BLAS (ExBLAS) library [19] 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 afore-mentioned unblocked LU factorization is also available at the same location as a separate package.

7. Conclusions and Future Work

In numerical linear algebra, algorithms accommodate virtual a modular and hierarchical structure. This property permits to assemble higher-level algorithms on top of fundamental kernels. We leverage this layered organization of linear algebra algorithms to derive reproducible algorithmic variants for the LU factorization by building them on top of the underlying BLAS operations. 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 obtaining correctly-rounded results.

Although some of the performance results can be argued, in some scenarios we should not trade off numerical stability and reproducibility for performance. Instead, we should do inverse, when not aiming to attain both goals. The development of reproducible BLAS kernels, including the enhancement of the EXTRSV performance, led us to the derivation of the reproducible unblocked LU factorization that integrates partial pivoting. Finally, we presented the initial evidence that a class of reproducible higher-level linear algebra operations can be constructed by following the bottom-up approach.

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). This work was also granted access to the HPC resources of The Institute for Scientific

Computing and Simulation financed by Region Île-de-France and the project Equip@Meso (reference ANR-10-EQPX-29-01) overseen by the French National Agency for Research (ANR) as part of the "Investissements d'Avenir" program. This work was also partly supported by the FastRelax (ANR-14-CE25-0018-01) project of ANR.

References

[1] P. Ahrens, H.D. Nguyen, J. Demmel, Efficient reproducibile floating point summation and BLAS, Technical Report UCB/EECS-2016-121, EECS Department, University of California, Berkeley, 2016.

[2] A. Arteaga, O. Fuhrer, T. Hoefler, Designing bit-reproducible portable high-performance applications, in: Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium, IPDPS '14, IEEE Computer Society, Washington, DC, USA, 2014, pp. 1235–1244.

[3] E. Bainville, OpenCL GEMV, Available on the WWW, `http://www.bealto.com/gpu-gemv_v3.html`, 2010. Accessed 24-AUG-2016.

[4] P. Bientinesi, E.S. Quintana-Ortí, R.A. van de Geijn, Representing linear algebra algorithms in code: The FLAME application programming interfaces, ACM Transactions on Mathematical Software 31 (2005) 27–59.

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

[6] S. Collange, D. Defour, S. Graillat, R. Iakymchuk, Numerical reproducibility for the parallel reduction on multi- and many-core architectures, Parallel Computing 49 (2015) 83–97.

[7] J. Demmel, H.D. Nguyen, Fast reproducible floating-point summation, in: Proceedings of the 21st IEEE Symposium on Computer Arithmetic, Austin, Texas, USA, pp. 163–172.

[8] J. Demmel, H.D. Nguyen, Parallel Reproducible Summation, IEEE Transactions on Computers 64 (2015) 2060–2070.

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

[10] J.J. Dongarra, J. Du Croz, S. Hammarling, I. Duff, A set of level 3 basic linear algebra subprograms, ACM Trans. Math. Software 16 (1990) 1–17.

[11] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, P. Zimmermann, MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding, ACM Trans. Math. Softw. 33 (2007) 13.

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

[13] G.H. Golub, C.F.V. Loan, Matrix Computations, The Johns Hopkins University Press, Baltimore, 3rd edition, 1996.

[14] A. Haidar, T. Dong, P. Luszczek, S. Tomov, J. Dongarra, Batched Matrix Computations on Hardware Accelerators Based on GPUs, Int. J. High Perform. Comput. Appl. 29 (2015) 193–208.

[15] Y. Hida, X.S. Li, D.H. Bailey, Algorithms for quad-double precision floating point arithmetic, in: Proceedings of the 15th IEEE Symposium on Computer Arithmetic, IEEE Computer Society Press, Los Alamitos, CA, USA, 2001, pp. 155–162.

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

[17] R. Iakymchuk, S. Collange, D. Defour, 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.

[18] R. Iakymchuk, S. Collange, D. Defour, S. Graillat, 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, April 13-15, 2015, Las Vegas, Nevada, USA, pp. 353–358. HAL: hal-01116588v2.

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

[20] R. Iakymchuk, D. Defour, S. Collange, 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, pp. 126–137.

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

[22] Intel®, Conditional Numerical Reproducibility (CNR) in Intel MKL 11.0, Available on the WWW, `https://software.intel.com/en-us/articles/conditional-numerical-reproducibility-cnr-in-intel-mkl-110`, 2012.

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

[24] U. Kulisch, V. Snyder, The Exact Dot Product As Basic Tool for Long Interval Arithmetic, Computing 91 (2011) 307–313.

[25] X.S. Li, J.W. Demmel, D.H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S.Y. Kang, A. Kapur, M.C. Martin, B.J. Thompson, T. Tung, D.J. Yoo, Design, implementation and testing of extended and mixed precision BLAS, ACM Trans. Math. Softw. 28 (2002) 152–205.

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

[27] MAGMA project home page, MAGMA project home page, `http://icl.cs.utk.edu/magma/`, ???? Accessed 14-OCT-2016.

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

[29] R.M. Neal, Fast exact summation using small and large superaccumulators, Technical Report, University of Toronto, 2015.

[30] T. Ogita, S.M. Rump, S. Oishi, Accurate sum and dot product, SIAM J. Sci. Comput 26 (2005).

[31] J.M. Ortega, The ijk forms of factorization methods I. Vector computers, Parallel Computing 7 (1988) 135–147.

[32] S.M. Rump, Ultimately fast accurate summation, SIAM J. Scientific Computing 31 (2009) 3466–3502.

[33] Y.K. Zhu, W.B. Hayes, Algorithm 908: Online Exact Summation of Floating-Point Streams, ACM Trans. Math. Softw. 37 (2010) 37:1–37:13.