Reproducible Triangular Solvers for High-Performance Computing

Roman Iakymchuk, David Defour, Caroline Collange, Stef Graillat

Reproducible Triangular Solvers for High-Performance Computing

Roman Iakymchuk*†, David Defour‡, Sylvain Collange§, Stef Graillat*
*Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, LIP6, F-75005 Paris, France
Email: {stef.graillat, roman.iakymchuk}@lip6.fr
†Sorbonne Universités, UPMC Univ Paris 06, ICS, F-75005 Paris, France
‡DALI–LIRMM, Université de Perpignan, 52 avenue Paul Alduy, F-66860 Perpignan, France
Email: david.defour@univ-perp.fr
§INRIA – Centre de recherche Rennes – Bretagne Atlantique, Campus de Beaulieu, F-35042 Rennes Cedex, France
Email: sylvain.collange@inria.fr

***Abstract*—On modern parallel architectures, floating-point computations may become non-deterministic and, therefore, non-reproducible mainly due to non-associativity of floating-point operations. We propose an algorithm to solve dense triangular systems by leveraging the standard parallel triangular solver and our, recently introduced, multi-level exact summation approach. Finally, we present implementations of the proposed fast reproducible triangular solver and results on recent NVIDIA GPUs.**

***Keywords*—Triangular linear system, substitution algorithm, reproducibility, accuracy, superaccumulator, error-free transformations, GPU accelerators.**

I. Introduction

Exascale computing (10^{18} operations per second) is likely to be reached within a decade. Thanks to this increasing computational power of many- and multi-core architectures we are able to solve more complex and/or larger problems. That, consequently, leads to the higher number of floating-point operations to be performed, each of them potentially causes a round-off error. Floating-point operations like addition and multiplication are non-associative, so that, for instance, parallel implementations of the Basic Linear Algebra Subprograms (BLAS) routines become non-reproducible. Hence, the result may vary from one parallel machine to another or even from one run to another. These discrepancies worsen on heterogeneous architectures – such as clusters with accelerators like GPUs – which combine together programming environments that may obey various floating-point models and offer different intermediate precision or different operators [19]. Non-determinism of floating-point calculations in parallel programs causes validation and debugging issues and may even lead to deadlocks [3]. It is expected that these problems will get increasingly critical as the trend towards large-scale heterogeneous platforms continues [1].

Solving a dense triangular linear system (corresponds to the TRSV routine in BLAS) is an important building block for many numerical linear algebra problems, which arise in many science and engineering simulations. The solution of a system of linear equations with n equations and n unknowns is generally presented in a matrix form as $Ax = b$, where A is an $n \times n$ matrix and both x and b are vectors of length n. While this general system of equations requires $O(n^3)$ operations to get the solution, the special case where A is a triangular matrix (contains zeros either above or below the main diagonal) is considerably cheaper to solve, requiring $O(n^2)$ operations only. The triangular solver is usually treated as the second stage of the Gaussian elimination or the following step of the LU decomposition.

Unfortunately, the performance of the parallel naive triangular solver is notoriously poor and resilient to efficient parallelization. Because of long dependency chains, there is little inherent concurrency in the algorithm. Low data reuse also means the solver has a low arithmetic intensity. Thus, communication cost is high compared to computation, leaving headroom to perform extra operations at no cost, especially on modern architectures. Concurrency has been addressed through classical parallel implementations such as the "fan-in", "fan-out", or column sweep algorithm as well as the "wave-front" and "cyclic" algorithm, referred as the Li-Coleman algorithm [13], [4], [9]. However, none of these parallel implementations can ensure the reproducibility of results due to their parallel nature.

On this class of problems, reproducibility may be addressed using two solutions. The first one consists in providing a deterministic control of rounding errors by, for example, enforcing the execution order for each operation. However, this solutions is not portable and do not scale well with the number of processing cores. The second aims at avoiding cancellation and rounding errors by using, for example, a superaccumulator [11]. Indeed, superaccumulators guarantee the accuracy of the solution, but for the cost of more operations per data.

In this work, we address the problem of reproducibility for substitution algorithms due to cancellation and rounding errors that occur during multiplications and additions within the inner loop. We apply our multi-level summation approach based on superaccumulators [2] that allows us to perform accumulation in any order without loosing a bit of information. Therefore, the work can be distributed in any fashion (row- or column-wise or per block) that allows to exploit concurrency while achieving reproducibility.

The paper is organized as follows. Section II reviews floating-point expansions and superaccumulators. Section III introduces our multi-level approach to solve triangular systems.

Section IV presents implementations and results on GPUs. Finally, we discuss future work and conclusions in Section V.

II. Background

Floating-point (FP) arithmetic consists in approximating real numbers with a significand, an exponent, and a sign. The IEEE-754 standard, which was revised in 2008, specifies floating-point formats and operations. In this paper, we consider the `binary64` or double precision format, although our strategy applies to the other formats as well.

Non-associativity of floating-point addition comes from rounding errors while accumulating numbers with different exponents. It leads to the elimination of the lower bits of the sum. In contrast, the subtraction between numbers of the same sign and the same exponent is always exact. However, due to the cancellation, it amplifies the impact of previous errors. Thus, the accuracy of a floating-point summation depends on the order of evaluation. More detailed information can be found in the main references for floating-point arithmetic [6], [16].

Two approaches exist to perform floating-point addition without incurring round-off errors. The first approach aims at computing the error which occurs during rounding using error-free transformations, see Section II-A. The second approach exploits the finite range of representable floating-point numbers by storing every bit in a very long vectors of bits, see Section II-B.

A. Floating-Point Expansion

In order to perform summations exactly, one has to recover errors which occurred while rounding and keep track of them. FP *expansions* represent the result as an unevaluated sum of FP numbers, whose components are ordered in magnitude with minimal overlap to cover a wide range of exponents. Floating-point expansions of size 2 and 4 are described in [12] and [5], accordingly. They are based on error-free transformation (EFT). Indeed, when working with rounding-to-nearest, the rounding error in addition or multiplication can be represented as a floating-point number and can also be computed in floating-point arithmetic. The traditional EFT for the addition is `TwoSum` [10], Alg. 1, and for the multiplication is `TwoProduct` [17], Alg. 2. For `TwoProduct`, we use the fused-multiply-and-add (FMA) instruction that makes it possible to compute $a \times b + c$ with only one rounding.

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

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

One can notice that adding one FP number to an expansion is an iterative process. The FP number is first added to the head of the expansion and the rounding error is recovered as a floating-point number using an EFT such as `TwoSum`. The error is then recursively accumulated to the remainder of the expansion. With expansions of size n – that correspond to the unevaluated sum of n floating-point numbers – it is possible to accumulate floating-point numbers without losing accuracy as long as every intermediate result can be represented exactly as a sum of n FP numbers. This situation happen when the dynamic range of the sum is lower than $2^{53 \times n}$ (for `binary64`).

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

Function $[r, s]$ = *TwoProduct*(a, b)
$r \leftarrow a * b$
$s \leftarrow fma(a, b, -r)$

The main advantage of this solution is that the expansion can stay in registers during the computations. However, the accuracy is insufficient for the summation of numerous FP numbers or FP numbers with a huge dynamic range. Moreover, the complexity of this approach grows linearly with the size of the expansion.

B. Superaccumulator

An alternative to expansions is to use a very long fixed-point accumulator or *superaccumulator*. The length of the accumulator is chosen such that every bit of information of the input format can be represented (`binary64` in our case); this covers the range from the minimum representable floating-point value to the maximum value, independently of the sign. For instance, Kulisch [11] proposed to use an accumulator of size 4288 bits to handle the accumulation of products of `binary64` values. The addition is performed without loss of information by accumulating every floating-point input number in the superaccumulator, see Fig. 1. The superaccumulator is a solution to produce the exact result of a very large amount of floating-point numbers of arbitrary magnitude. However, for a long period this approach was considered impractical as it induces a very large memory overhead. Furthermore, without dedicated hardware support, its performance is limited by indirect memory accesses that make vectorization challenging.

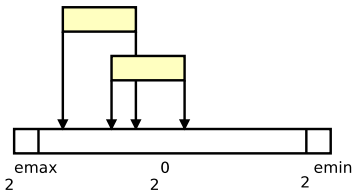


Fig. 1: Superaccumulator.

C. Multi-level exact summation

The multi-level approach to summation splits the computation in the substitution algorithm into five stages: filtering, private superaccumulation, scalar superaccumulation, rounding, scatter/gather [2]. This decomposition is suitable for the nested parallelism of modern architectures and maps efficiently to GPUs.

The first stage uses floating-point expansions with error-free transformations for the multiplication and the addition

of two floating point numbers, see Algs. 1 and 2. Each thread maintains its own floating-point expansion that is stored in registers. In order to maintain expansions of size n, we extend the approach from Alg. 1 and derive Alg. 3. A GPU implementation will allocate one expansion per thread.

Algorithm 3: Error-free transformation of size n.

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

In case the accuracy provided by floating-point expansions is not enough, then the remaining rounding error is accumulated to private superaccumulators. At the end of the summation, the contents of the expansion is also accumulated to the same superaccumulators. Depending on the amount of memory available, private superaccumulators are stored in either fast local memory, e.g. cache or shared memory, or global memory.

In the third stage, k private superaccumulators are merged into a single scalar superaccumulator. The rounding of the scalar superaccumulator back to the desired floating-point format is performed in the fourth stage in order to obtain the correctly rounded results. And, the division by diagonal element is calculated in the same floating-point precision. Finally, the computed elements of the solution are scattered to the other computing units. At the end of the computation, the solution is constructed by gathering the computed parts.

III. Reproducible Triangular Solvers

In this section we describe our multi-level reproducible approach for solving triangular systems. At first we present the sequential triangular solvers and, then, we derive our parallel reproducible substitution algorithm.

The system $Lx = b$, where L is a non-unit lower triangular matrix, can be solved using the formula $x_i = (b_i - \sum_{j=1}^{i-1} l_{ij}x_j)/l_{ii}$. In this case, elements of x are computed from first to last. A sequential algorithm for computing forward substitution is presented in Alg. 4.

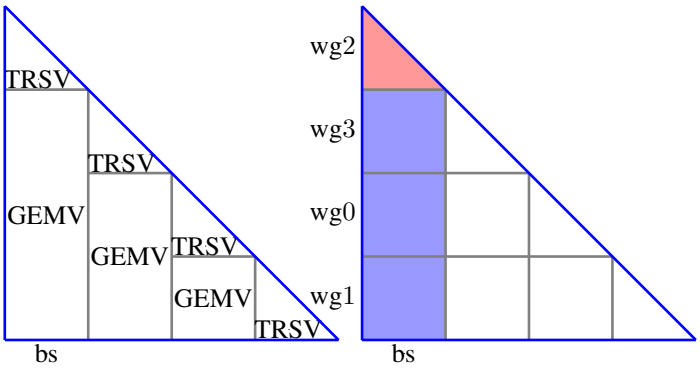
Algorithm 4: Forward substitution. $L \in R^{n \times n}$ is a nonsingular lower triangular matrix.

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

We focus on an algorithm for lower triangular systems as the results for back substitution have obvious analogues for forward substitution. In the sequel of this paper, we denote T as a matrix that can be either upper or lower triangular.

A. Hierarchical Scheme for Triangular Solvers

In order to parallelize Alg. 4, the work should be equally (optimistic scenario) distributed among parallel compute units such as processes (could be work-groups in case of GPUs) and threads. A common approach, which is used in the GotoBLAS (now OpenBLAS) library and is described in [7] for GPUs, is to split the matrix and both the right-hand side and the solution vectors into blocks and conduct the computations on those blocks. Fig. 2a represents Goto's implementation of TRSV in which the matrix is divided into diagonal blocks and panels underneath these blocks. In this case the computation is organized as follows: each diagonal block of size $bs \times bs$ ($bs = 32, 48, 64, ...$) is solved using the private routine `xTRSV_NLN` that finds the solution of a linear system with a non-transpose (N) non-unit (U) lower (L) triangular matrix; the remainingpart of each panel is computed by standard GEMV – a matrix-vector multiplication routine of the BLAS.



(a) GotoBLAS **(b)** GPU's

Fig. 2: Partitioning of a lower triangular matrix L, where bs stands for a block size and wg$x$ corresponds to a work group x.

We construct our multi-level reproducible substitution algorithm in a similar manner to Goto's and Hogg's [7] (depicted in Fig. 2b) TRSV by, in addition, integrating our hierarchical summation approach [2]. Hence, due to the dependency on the solution of diagonal blocks, the parallel substitution algorithm proceeds with the panel-step, meaning it is a loop over n/bs panels. Thus, the optimal amount of threads to perform computations would be $n \times k$, where $k \leq bs$. By following this assumption, each group of threads of size $bs \times k$ would be responsible of computing bs values of the resulting vector x. In order to avoid conflicts and collisions, TRSV on the diagonal block is performed using only $bs \times 1$ threads. This actually advocates the usage of GEMV which utilizes $bs \times k$ threads.

B. Accuracy for the general case

The classic way to compute the condition number of matrix A is $cond(A) = \|A^{-1}\|\|A\|$. Skeel introduced an alternative method to compute the condition number of linear system $Ax = b$ with real coefficients as [18], [6]

$$\text{cond}(A, x) = \frac{\||A^{-1}||A||x|\|_\infty}{\|x\|_\infty}. \quad (1)$$

In the case of triangular systems $Tx = b$, we have the following estimate of the relative forward error

$$\frac{\|x - \widehat{x}\|}{\|x\|} \leq nu\text{cond}(T, x) + \text{O}(u^2), \tag{2}$$

where u denotes the rounding unit, e.g. $u = 2^{-53}$ for `binary64`, and n is a size of the triangular system.

C. Accuracy when x is exactly representable

In this section we will show that the proposed method provides full accuracy independently of the condition number of the problem when the result x is exactly representable and neither overflow nor underflow occurs.

Theorem 3.1: Let the triangular system $Tx = b$, where $T \in R^{n\times n}$ is nonsingular, be solved by substitution, with any ordering. In addition, every element of the matrix T as well as the vector b is exactly representable in the destination format, i.e. *binary64*. If we know that the resulting vector x is exactly representable in the destination format as well, then using our multi-level reproducible algorithm, we can recover vector x in one pass independently of the condition number.

Proof: As the proposed method is insensitive to the order of operations, let us consider without loss of generality the sequential algorithm given in Alg. 4 to solve a triangular system $Lx = b$. We will prove by induction that if x_{i-1} is exactly recovered, then x_i will be exactly recovered using our algorithm.

Both b_1 and l_{11} are exactly representable, so by definition x_1 is exactly representable as well. This means that the division b_1/l_{11} is exact without any rounding error. Therefore, the theorem is valid for x_1.

Let all x_j be recovered exactly using our algorithm for every j such that $0 \leq j \leq i-1$. By definition, we know that l_{ij} is exactly representable, so the result of the multiplication $l_{ij} \times x_j$ is computed exactly using error-free transformations for every j such that $0 \leq j \leq i-1$. Then, the results is accumulated exactly in the superaccumulator s as long as no overflow occurs.

By definition we know that x_i is exactly representable. This means that the division $s/l_{ii} = x_i$ is exact if we consider s as a long vector. We now have to prove that

$$fl\left(fl\left(\frac{fl(s)}{l_{ii}}\right) + fl\left(\frac{s - fl(s)}{l_{ii}}\right)\right) = x_i \tag{3}$$

is correct, where $fl(y)$ is a floating-point number such that $fl(y) = y(1+\delta)$ and $|\delta| \leq u$ (u corresponds to the unit roundoff, $u = 2^{-53}$ in double precision).

Without loss of generality, let consider x_i and l_{ii} such that $1 \leq x_i, l_{ii} < 2$. Then $1 \leq x_i \times l_{ii} < 4$ with $x_i \times l_{ii} = s$ exactly. Let introduce p and e such that $s = p + e$ with $p = fl(s)$ and $e = s - fl(s)$. Then we have $x_i = \frac{p}{l_{ii}} + \frac{e}{l_{ii}}$ exactly. We now have three cases to consider. We should mention that thanks to Theorem 8.4 from [15], we do not have to consider the case when the quotient of two floating-point numbers lies exactly between two representable floating-point numbers.

When $1 \leq x_i \times l_{ii} < 2$. In that case $|e| < u$. Therefore

$$\left|x_i - \frac{p}{l_{ii}}\right| \leq \frac{|e|}{l_{ii}} < \frac{u}{l_{ii}} < u$$

which means that $fl\left(\frac{p}{l_{ii}}\right) = x_i$. As $\frac{|e|}{l_{ii}} < u$ we have $fl\left(fl\left(\frac{p}{l_{ii}}\right) + fl\left(\frac{e}{l_{ii}}\right)\right) = x_i$ and equation 3 is true.

When $2 \leq x_i \times l_{ii} < 4$ and $|e| < u \times l_{ii}$. In that case

$$\left|x_i - \frac{p}{l_{ii}}\right| \leq \frac{|e|}{l_{ii}} < \delta$$

and equation 3 is true as it is similar to the first case.

When $2 \leq x_i \times l_{ii} < 4$ and $\delta \times l_{ii} \leq |e| < 2 \times l_{ii}$. Let consider the case $e > 0$. We have

$$u \leq \left|x_i - \frac{p}{l_{ii}}\right| < \frac{2 \times u}{l_{ii}}$$

and $\frac{p}{l_{ii}} \in]pred(x_i), x_i - u[$ with $pred(y)$ corresponding to the floating-point number which is immediately before y. Therefore $fl(\frac{p}{l_{ii}}) = pred(x_i)$. In addition, we have $\frac{e}{l_{ii}} \geq u$ which implies that $fl\left(fl\left(\frac{p}{l_{ii}}\right) + fl\left(\frac{e}{l_{ii}}\right)\right) = succ(pred(x_i)) = x_i$ with $succ(y)$ corresponding to the floating-point number which is immediately after y. Therefore, equation 3 is true. The same method applies to the case $e \leq 0$. ■

IV. Implementations and Results

This section presents our implementations of the multi-level reproducible triangular solver and their evaluation on NVIDIA GPUs, see Tab. I for the detailed description of these architectures. We verify the accuracy of our implementations by comparing the computed results with the ones produced by the multiple precision MPFR library on CPUs; the library is not multi-threaded and does not support GPUs.

TABLE I: Hardware platforms employed in the experimental evaluation.

NVIDIA Tesla K20c	13 SMs × 192 CUDA cores	0.705 GHz
NVIDIA Quadro K5000	8 SMs × 192 CUDA cores	0.706 GHz

A. Implementation

Our implementations attempt to obtain the maximum performance through utilizing all resources of the considered GPU architectures: SIMD instructions, fused-multiply-and-add, private and local memory as well as atomic instructions.

We developed both unique and hand-tuned OpenCL implementations for NVIDIA GPUs. One way to implement the substitution algorithm is to create two kernels: one for TRSV on diagonal blocks; the other for GEMV on the remaining parts of each panel. This approach induces a kernel launch overhead. Instead, we combine two routines into one kernel and perform synchronization between work-groups through global memory. This strategy brings two benefits: overlap between the substitution of the diagonal block on one SM/CU and matrix-vector multiplications on the other SMs/CUs; a possibility to mask prefetching of matrix blocks.

According to our algorithm, the matrix is divided into blocks of size bs. Thus, a triangular solver can be associated with diagonal blocks while a matrix-vector multiplication routine with off-diagonal blocks. Such work distribution induces dependency between diagonal and off-diagonal blocks – a diagonal solver cannot begin until all off-diagonal blocks in the row have completed and a matrix-vector multiplication with off-diagonal blocks cannot start until a diagonal solver in that column has completed. We overcome this dependency by assigning to each work-group of threads a block-row of the matrix, see Fig. 2b. Thus, each work-group performs GEMV on each block within a block-row from left to right and then executes TRSV. In case the required data for a GEMV is not ready, the execution waits until it is available and transmitted.

To establish synchronization among work-groups, we use a single scalar in global memory that tracks a number of completed row. This is due to the fact that TRSV on diagonal blocks has to be computed in order, thus it is enough to store the latest completed row. To ensure that the vector of solution is visible to all work-groups we apply `barrier(CLK_GLOBAL_MEM_FENCE)` before incrementing the counter.

An extra synchronization step is required to dynamically assign matrix rows to work-groups since there is no guarantee that work-groups will run in index order. Static allocation would potentially lead to deadlocks. Thus, dynamic allocation is preferred that is accomplished using a global variable in conjunction with atomic increment. The proposed two global memory synchronizations need to be initialized before the execution of TRSV on the whole matrix though a trivial kernel run with a single thread.

B. Relative Forward Error

In all our tests, we rely on the IEEE-754 double precision format. To carry out the experiments, we wrote a random generator of very ill-conditioned triangular systems using the algorithm proposed in [14]; the random generator ensures that both the matrix T and the right-hand side vector b are composed of double precision floating-point numbers. In order to compute the exact solution x and our reproducible solution $\widehat{x}_r$, we benefited from Matlab Simulink symbolic computations and rounded the result back to double precision.

Fig. 3 presents the relative accuracy, meaning relative forward error computed by (2), of the classic substitution algorithm performed in double precision DTRSV and our reproducible substitution algorithm ExDTRSV versus the condition number. The Skeel condition number (1) varies from 10^5 to 10^{35} for matrices of size 40×40. For those relative errors that are greater than one, we set them to one, which means that no accuracy is left; that also makes it clearly visible on the plot. The results of both double precision and our reproducible substitutions show that the relative accuracy is proportional to the rounding unit u; for our substitution algorithm sometimes the relative accuracy is proportional to u^2. However, our substitution algorithm demonstrates the fluctuation in the relative accuracy. We believe that such behavior is related to the rounding errors caused by rounding-to-nearest after the reproducible and accurate summation with superaccumulators and the final division by diagonal elements of the matrix.

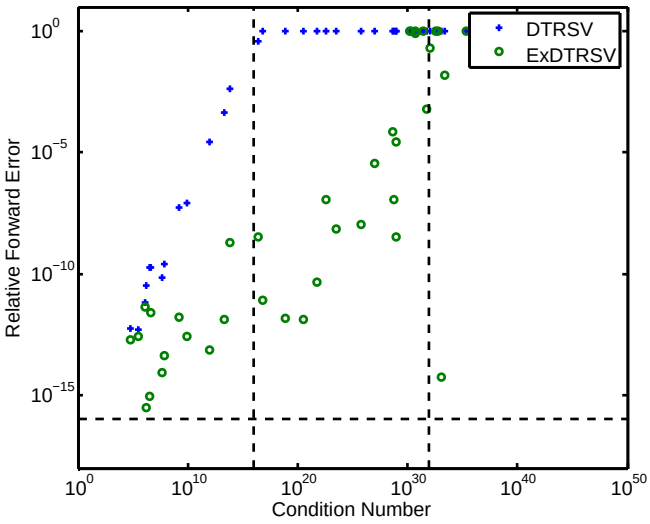


Fig. 3: Accuracy of `DTRSV` and `ExDTRSV` (our substitution algorithm) with respect to the condition number.

To sum up, the solution computed by our approach has the accuracy at least of the same order as the standard double precision substitution or often better (up to $nu^2\text{cond}(T,x)$ for systems with condition numbers smaller than $1/(nu^2)$). Moreover, when at least one bit of accuracy is guaranteed, our substitution algorithm always delivers reproducible solution.

C. Performance Results

As a baseline we consider the vectorized and parallelized non-deterministic double precision substitution algorithm [7]; referred as "Parallel DTRSV" on figures. Figs. 4 and 5 present the measured time achieved by the substitution algorithms as a function of the matrix size n on two GPUs, see Tab. I. In the keys of figures, "Superacc" corresponds to our algorithm that relies solely on superaccumulators and it is the slowest due to its extensive memory usage; "FPEn + Superacc" stands for our algorithm with floating-point expansions of size n ($n = 2 : 8$) in conjunction with error-free transformations and superaccumulators when needed; "FPE6EE + Superacc" represents our algorithm with the expansion of size 6 and the early-exit optimization technique [2]. In general, implementations with expansions deliver better performance than with superaccumulators only. However, due to switching to superaccumulators at the end of computing each element of the solution as well as when the accuracy provided by expansions is not enough, the gain from expansions is limited. Thus, there is a space for improvement in these preliminary results. Nevertheless, the solution computed by our substitution algorithms is constantly reproducible.

V. Conclusions and Future Work

We presented a multi-level approach to achieve reproducible and accurate solutions of triangular systems composed of floating-point numbers along with implementations on

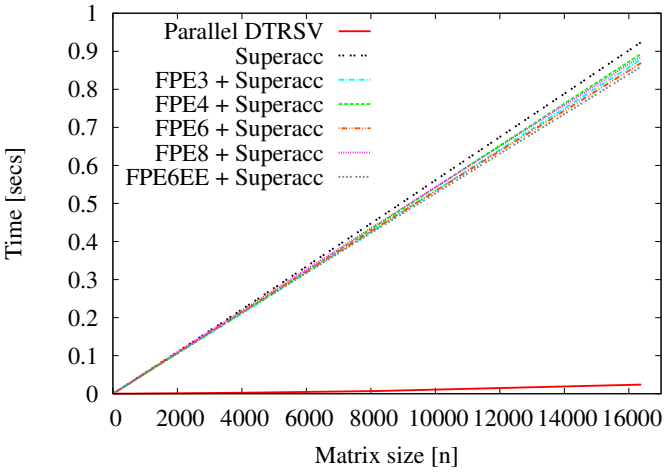


Fig. 4: Performance of substitution algorithms on NVIDIA K20c.

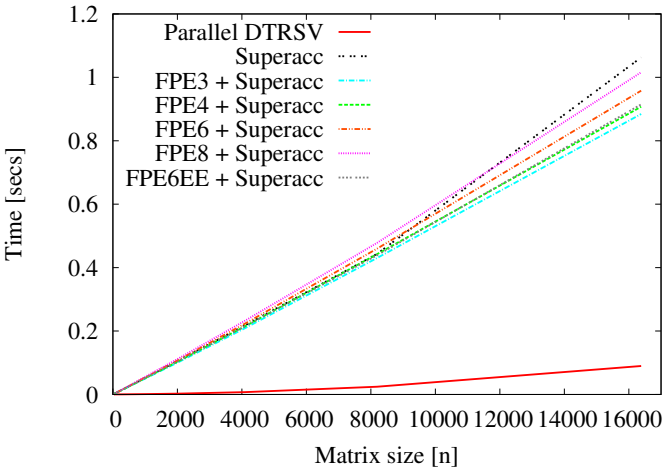


Fig. 5: Performance of substitution algorithms on NVIDIA K5000.

many-core architectures such as GPUs. Our approach certainly delivers the same accuracy as double precision and, moreover, its accuracy is often better in practice. In order to increase the accuracy, one may consider using double-double precision. However, this approach is a factor of 9 slower and it is non-reproducible. Thus, even though the delivered performance of our preliminary implementations is at least 11 times slower, which is considered to be improved, the experiments yielded the reproducible solution at any scenario.

In case when the solution x is exactly representable, we proved that our algorithm delivers full accuracy regardless of the condition number of triangular systems. In general case this may not hold, therefore, we plan to engage one step of iterative refinement in order to enhance and guarantee the accuracy of the results.

Our ultimate goal is to apply the multi-level approach to derive reproducible, accurate, and fast library for fundamental linear algebra operations – like those included in the BLAS library – on new parallel architectures such as Intel Xeon Phi co-processors and GPU accelerators. Moreover, we plan to conduct a priori error analysis of the derived ExBLAS (Exact BLAS) routines. More information on the ExBLAS project as well as its sources can be found in [8].

ACKNOWLEDGEMENT

This work undertaken (partially) in the framework of CALSIMLAB is supported by the public grant ANR-11-LABX-0037-01 overseen by the French National Research Agency (ANR) as part of the "Investissements d'Avenir" program (reference: ANR-11-IDEX-0004-02). This work was also granted access to the HPC resources of ICS financed by Region Île-de-France and the project Equip@Meso (reference ANR-10-EQPX-29-01) overseen by ANR as part of the "Investissements d'Avenir" program.

REFERENCES

[1] Keren Bergman and al. Exascale computing study: Technology challenges in achieving exascale systems. DARPA Report, September 2008.

[2] Sylvain Collange, David Defour, Stef Graillat, and Roman Iakymchuk. Full-Speed Deterministic Bit-Accurate Parallel Floating-Point Summation on Multi- and Many-Core Architectures. Technical Report HAL: hal-00949355, INRIA, DALI–LIRMM, LIP6, ICS, February 2014.

[3] K. Doertel. Best known method: Avoid heterogeneous precision in control flow calculations. Technical report, Intel, August 2013.

[4] M. T. Heath and C. H. Romine. Parallel solution of triangular systems on distributed memory multiprocessors. *SIAM Journal on Scientific and Statistical Computing*, 9:558–588, 1988.

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

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

[7] J.D. Hogg. A fast triangular solve on GPUs. Technical report, Science and Technology Facilities Council, 2012. RAL-P-2012-002.

[8] Roman Iakymchuk, Sylvain Collange, David Defour, and Stef Graillat. ExBLAS – Exact BLAS. https://exblas.lip6.fr/.

[9] S. L. Johnsson. Communication effect basic linear algebra computations on hypercube architectures. *J. Parallel Distrib. Comput.*, 4(2), 1987.

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

[11] Ulrich Kulisch and Van Snyder. The Exact Dot Product As Basic Tool for Long Interval Arithmetic. *Computing*, 91(3):307–313, March 2011.

[12] Xiaoye S. Li, James W. Demmel, David H. Bailey, Greg Henry, Yozo Hida, Jimmy Iskandar, William Kahan, Suh Y. Kang, Anil Kapur, Michael C. Martin, Brandon J. Thompson, Teresa Tung, and Daniel J. Yoo. Design, implementation and testing of extended and mixed precision BLAS. *ACM Trans. Math. Softw.*, 28(2):152–205, 2002.

[13] Li, G. and Coleman, T. F. A new method for solving triangular systems on distributed-memory message-passing multiprocessor. *SIAM J. Sci and Stat. Comp.*, 10:382–396, 1989.

[14] Nicolas Louvet. *Algorithmes compensés en arithmétique flottante : précision, validation, performances*. PhD thesis, UPVD, 2007.

[15] Peter Markstein. *IA-64 and elementary functions: speed and precision*. Prentice Hall, 2000.

[16] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser, 2010.

[17] Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi. Accurate sum and dot product. *SIAM J. Sci. Comput*, 26, 2005.

[18] Robert D. Skeel. Scaling for numerical stability in Gaussian elimination. *J. Assoc. Comput. Mach.*, 26(3):494–526, 1979.

[19] Nathan Whitehead and Alex Fit-Florea. Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs. Technical report, NVIDIA, 2011.