Reproducible and Accurate Matrix Multiplication for GPU Accelerators

Roman Iakymchuk, David Defour, Caroline Collange, Stef Graillat

Reproducible and Accurate Matrix Multiplication for GPU Accelerators

Roman Iakymchuk[1,2], David Defour[3], Sylvain Collange[4], and Stef Graillat[1]

[1] Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, LIP6, F-75005 Paris, France
CNRS, UMR 7606, LIP6, F-75005 Paris, France
{stef.graillat, roman.iakymchuk}@lip6.fr
[2] Sorbonne Universités, UPMC Univ Paris 06, ICS, F-75005 Paris, France
[3] DALI–LIRMM, Université de Perpignan, 52 avenue Paul Alduy, F-66860 Perpignan, France
david.defour@univ-perp.fr
[4] INRIA – Centre de recherche Rennes – Bretagne Atlantique
Campus de Beaulieu, F-35042 Rennes Cedex, France
sylvain.collange@inria.fr

Abstract. Due to non-associativity of floating-point operations and dynamic scheduling on parallel architectures, getting a bitwise reproducible floating-point result for multiple executions of the same code on different or even similar parallel architectures is challenging. In this paper, we address the problem of reproducibility in the context of matrix multiplication and propose an algorithm that yields both reproducible and accurate results. This algorithm is composed of two main stages: a filtering stage that uses fast vectorized floating-point expansions in conjunction with error-free transformations; an accumulation stage based on Kulisch long accumulators in a high-radix carry-save representation. Finally, we provide implementations and performance results in parallel environments like GPUs.

Keywords: Matrix multiplication, reproducibility, accuracy, Kulisch long accumulator, error-free transformation, floating-point expansion, rounding-to-nearest, GPU accelerators.

1 Introduction

In many fields of science and engineering, the process of finding the solution for a specific problem requires to solve a system of linear equations, or a least squares problem, or eigenvalue problem. The common approach is to develop solvers for those tasks alone and then spend tremendous amount of time on tuning them. However, the best practice suggests to use already optimized solution-routines contained in linear algebra libraries.

The development of linear algebra libraries has its beginning in the early 1970s. From that time many libraries have been released. With the influence of common HPC computers, which were based on vector processors, in 1979 a first set of Basic Linear Algebra Subprograms (BLAS-1) [1] was designed as a set of *basic vector operations*. In 1988 the idea of BLAS was developed further yielding to a second set of routines for *matrix-vector operations* (BLAS-2) [2]. For those routines the amount of data required and floating point operations (Flops) performed have quadratic complexity.

When architectures with multiple layers of cache memory appeared, the performance for both BLAS-1 and BLAS-2 operations became an issue: for these routines the ratio between the numbers of Flops and memory accesses is only $O(1)$. In order to attain high performance on architectures with a hierarchical memory system, in 1990 the third level of BLAS (BLAS-3) [3] with *matrix-matrix operations* was defined. These routines perform $O(n^3)$ Flops over $O(n^2)$ data, giving the opportunity to hide the memory latency and offer performance close to the achievable peak.

A generic implementation of the BLAS specification is provided since the announcement of the library in 1979. This reference implementation is equipped with the complete functionality, but it is not optimized for any architecture. Thus, processor manufacturers as well as scientists developed tuned implementations of the BLAS for each architecture. Prominent examples of these implementations are Intel MKL, AMD ACML, IBM ESSL, ATLAS, and GotoBLAS (now OpenBLAS). ATLAS [4] is based on an auto-tuned empirical approach while GotoBLAS [5,6] is a hand-tuned machine-specific implementation of the BLAS. Due to the raising popularity of GPUs for high-performance computing, NVIDIA provided a GPU-version of the BLAS (cuBLAS).

The core of the BLAS library is xGEMM[5], which is a BLAS-3 routine, that computes the matrix-matrix products as

$$C := \alpha op(A) op(B) + \beta C, \tag{1}$$

where α and β are scalars; $op(A), op(B)$, and C are general matrices with $op(A)$ a $m \times k$ matrix, $op(B)$ a $k \times n$ matrix, and C a $m \times n$ matrix; $op(X)$ represents either a non-transposed X or a transposed X^T matrix. xGEMM performs $2mnk$ floating-point operations over $mk + kn + mn$ data. When $m = n = k$ the ratio between floating-point operations (Flops) and memory accesses is $\frac{2n}{3}$. This means that most memory accesses can be hidden in the background while the processor performs the computation. All the other BLAS-3 routines can be expressed in terms of xGEMM. Moreover, when different implementations of BLAS are compared, the first criteria used for this comparison is the performance of xGEMM.

The profitable ratio between the computation and the memory references of the BLAS-3 routines has a strong impact on the design and automatic generation of linear algebra algorithms. For instance, in order to exploit the optimized BLAS implementations, the Linear Algebra PACKage (LAPACK) builds its blocked algorithms on top of the BLAS-3 operations. Furthermore, scientists either try to generate algorithms relying more on the BLAS-3 routines, in particular xGEMM, or try to rewrite their algorithms in order to benefit from the performance provided by the BLAS-3 routines [7,8].

In general, matrix-matrix products relies on optimized version of parallel reduction and dot-product involving floating-point additions and multiplications which are non-associative operations. Hence, as the order of operations may vary from one parallel machine to another or even from one run to another [9], reproducibility of results is not guaranteed. These discrepancies worsen on heterogeneous architectures – such as clusters composed of standard CPUs in conjunction with GPUs and/or accelerators like Intel Xeon Phi – which combine together different programming environments that

[5] In general, x stands for four different formats, but in the scope of this article we consider x to correspond to single (S) or double (D) precision.

may obey various floating-point models and offer different intermediate precision or different operators [10,11]. In some cases, such non-reproducibility of floating-point computations on parallel machines causes validation and debugging issues, and may even lead to deadlocks [12].

By reproducibility, we mean getting a bitwise identical floating-point result from multiple runs of the same code on the same data. Numerical reproducibility can be addressed by targeting either the order of operations or the error resulting from finite arithmetic. One solution consists in providing the deterministic control over rounding errors by, for example, enforcing the execution order for each operation. However, these approach is not portable and/or does not scale well with the number of processing cores. The other solution aims at avoiding cancellation and rounding errors by using, for instance, a long accumulator such as the one proposed by Kulisch [13]. This solution increases the accuracy at the price of more operations and memory transfers per output data. Because of that, for a long time, it was considered too expensive for the little benefit it was providing.

Recently, we introduced in [14] an approach to compute deterministic sums of floating-point numbers. Our approach is based on a multi-level algorithm that combines efficiently floating-point expansions and long accumulators. The proposed implementations on recent Intel desktop and server processors, on Intel Xeon Phi accelerator, and on both AMD and NVIDIA GPUs, showed that the numerical reproducibility and bit-perfect accuracy can be achieved at no additional cost for large sums that have dynamic ranges of up to 90 orders of magnitude. This speed-up is possible thanks to arithmetic units that are left underused by the standard reduction algorithms.

In this article, we propose an approach to ensure both the reproducibility and the accuracy (rounding-to-nearest) of the product of two matrices composed of floating-point numbers. The derived algorithm is based on the standard non-deterministic xGEMM and our deterministic summation algorithm. Moreover, we provide implementations of this algorithm on GPU accelerators. To our knowledge, this is the first work on reproducible matrix-matrix multiplication.

The paper is organized as follows. Section 2 reviews related aspects of floating-point arithmetic in particular floating-point expansions and long accumulators. Section 3 presents our approach to derive exact, meaning both reproducible and accurate, matrix-matrix product. In Section 4, we expose implementations and performance results on GPU accelerators. Finally, we discuss related works and draw conclusions in Sections 5 and 6, respectively.

2 Background

Without loss of generality, in the rest of this article, we will consider double precision format (`binary64`) from the IEEE-754 standard [15]. Floating-point representation of numbers allows to cover a wide *dynamic range*. Dynamic range refers to the absolute ratio between the number with the largest magnitude and the number with the smallest non-zero magnitude in a set. For instance, `binary64` can represent positive numbers from 4.9×10^{-324} to 1.8×10^{308}, so it covers a dynamic range of 3.7×10^{631}.

Non-associativity of floating-point addition implies that the result depends on the order of the operations. For example in double precision $(-1 \oplus 1) \oplus 2^{-100}$ is different from $-1 \oplus (1 \oplus 2^{-100})$ where $\oplus$ denotes the result of a floating-point addition. Thus, the accuracy of a floating-point summation depends on the order of evaluation. More details about this phenomenon can be found in the main references [16,17].

Two approaches exist to execute one floating-point addition without introducing rounding error. The first solution aims at computing the error which occurred during rounding using floating-point expansions in conjunction with error-free transformations, see Section 2.1. The second solution exploits the finite range of representable floating-point numbers by storing every bit in a very long vector of bits, see Section 2.2.

2.1 Floating-Point Expansion

Floating-point expansions represent the result as an unevaluated sum of floating-point numbers, whose components are ordered in magnitude with minimal overlap to cover a wide range of exponents. Floating-point expansions of sizes 2 and 4 are described in [18] and [19], accordingly. They are based on error-free transformation. 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 error-free transformation for the addition is `TwoSum` [20], Alg. 1, and for the multiplication is `TwoProduct`, Alg. 2. For `TwoSum`, it means that $r + s = a + b$ with $r = a \oplus b$ and s, which is a floating-point number that corresponds to rounding error. For `TwoProduct`, we use the fused multiply and add (FMA) instruction that is widely available on modern architectures. $\mathrm{FMA}(a, b, c)$ makes it possible to compute $a \times b + c$ with only one rounding. Thus, we have $r + s = a \times b$ with $r = a \otimes b$ and $s = \mathrm{FMA}(a, b, -r)$, where $\otimes$ stands for the floating-point multiplication.

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

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

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

Function $[r, s]$ = `TwoProduct`(a, b)
- $r \leftarrow a \times b$
- $s \leftarrow \mathrm{FMA}(a, b, -r)$

Adding one floating-point number to an expansion is an iterative operation. The floating-point number is first added to the head of the expansion and the rounding error is recovered as a floating-point number using an error-free transformation 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 floating-point numbers. This situation occurs when the dynamic range of the sum is lower than $2^{53 \cdot n}$ (for `binary64`).

The main advantage of this solution is that expansions can be placed in registers during the whole computation. However, the accuracy is insufficient for the summation of numerous floating-point numbers or sums with a large dynamic range. Moreover, the complexity of this algorithm grows linearly with the size of expansion.

2.2 Long accumulator

An alternative algorithm to floating-point expansions uses very long fixed-point accumulators. The length of the accumulator is selected in such a way that it represents every bit of information of the input format, e.g. `binary64`; this covers the range from the smallest representable floating-point value to the largest one, independently of the sign. For instance, Kulisch [13] proposed to use an accumulator of 4288 bits to handle the dot product of two vectors composed of `binary64` values. The summation is performed without loss of information by accumulating every floating-point input numbers in the long accumulator, see Fig. 1. The long accumulator is the perfect 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 makes vectorization challenging.

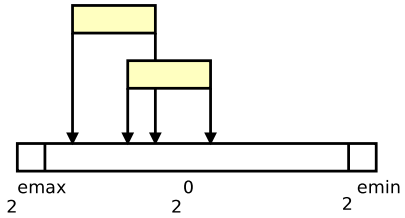


Fig. 1: Kulisch long accumulator.

3 Exact Matrix-Matrix Multiplication

In order to achieve best performance for linear algebra kernels, machine-specific hand tuning of those kernels is often applied; a good example is the Goto's implementation of xGEMM. Scientists aim at optimizing this process for existing and upcoming architectures through the automatic generation of linear algebra kernels. As the matrix-matrix multiplication is the core of the BLAS library, in several works [4,21] the problem of optimizing this routine for a given architecture was tackled by applying the automatic generation approach. For instance, the ATLAS project [4] provides a very good implementation of BLAS by tuning routines for various architectures; those are centralized

around a highly tuned matrix-matrix product that is automatically optimized for different levels of memory hierarchy. The idea of auto-tuning was extended to GPUs architectures applying different programming models such as CUDA and OpenCL. Apart from both code generation and heuristic search in conjunction with OpenCL, Matsumoto et. al. [21] proposed to store data in memory not only in a standard row-/column-major order, but also in a block-major order. We revise these ideas and employ it with modifications in our implementations of exact xGEMM, which is described in Section 4.1. Therefore, we combine together two approaches: auto-tuning for standard non-deterministic xGEMM and machine-specific hand tuning for our reproducible approach.

3.1 Hierarchical Approach for Matrix-Matrix Multiplication

We introduced in [14] a hierarchical superaccumulation scheme for the summation of floating-point numbers (parallel reduction) that relies on floating-point expansions with error-free transformations and long accumulators as described in Section 2. Thanks to the latter, this approach guarantees both reproducible and accurate results. This allows us to propose a reproducible and accurate matrix-matrix multiplication scheme which divides computations into three stages: filtering, private superaccumulation, and rounding. This decomposition is suitable for the nested parallelism of modern architectures and it makes a full use of SIMD and multi-threads.

In the first stage, each partial product is computed using error-free transformation. In order to ensure accuracy, this steps generates two floating-point numbers, see Alg. 2. Both resulting floating-point numbers are accumulated using algorithm Alg. 1 in an expansion of size $n (n \geq 3)$ that is stored in registers or private memory for each threads. This step benefits from vectorization and pipelining by maintaining one expansion per GPU thread.

In case the accuracy provided by floating-point expansions for product and/or summation is not enough, a non-zero residue x remains after this first accumulation. Each residue x is added to a long accumulator. We also propose an optimized version of floating-point expansions of size n that relies on the stopping criteria ($x \equiv 0$) in the accumulation loop. This technique is called *early-exit* and exhibits performance which depends on the distribution of input numbers and the ability of the architecture to handle irregular branches.

A trade-off between speed and usage of the hardware resources lies in the proper choice of the size n of the floating-point expansion. A small value of n will lead to numerous transfers from the expansion towards the long accumulators, which will slow down the computation. A large value of n will lead to the overuse of registers and ultimately the register spilling.

Once all the input number are accumulated, each floating-point expansion is flushed to long accumulators, independently of the parameter n. Hence, the second stage is based on superaccumulation, meaning summation to long accumulators, and it is involved either when the accuracy provided by expansions is not enough or at the end of the computation. Depending on the amount of memory available, long accumulators are stored in either fast local memory, e.g. cache or shared memory, or global memory.

In the third stage, the rounding of the private long accumulator back to the desired floating-point format is performed in order to obtain the correctly rounded results.

4 Implementations and Experimental Results

This section presents our implementations of the multi-level reproducible matrix multiplication and their evaluation on both NVIDIA and AMD GPUs, see Tab. 1 for the detailed description of these GPU architectures. We compared the accuracy of our implementations with results produced by the multiple precision library MPFR on CPUs. We should mention that this library is not multi-threaded and does not support GPUs. In case of `binary64`, we used 4196 bits ($2\times$ (emin + emax + mantissa) $= 2\times(1022+1023+53)$) within MPFR in order to guarantee the bit-wise reproducibility as well as the accuracy of the results independently of rounding errors and dynamic ranges.

Table 1: Hardware platforms used for the experiments.

A	NVIDIA Tesla K20c	13 SMs $\times$ 192 CUDA cores	0.705 GHz
B	AMD Radeon HD 7970	32 CUs $\times$ 64 units	0.925 GHz

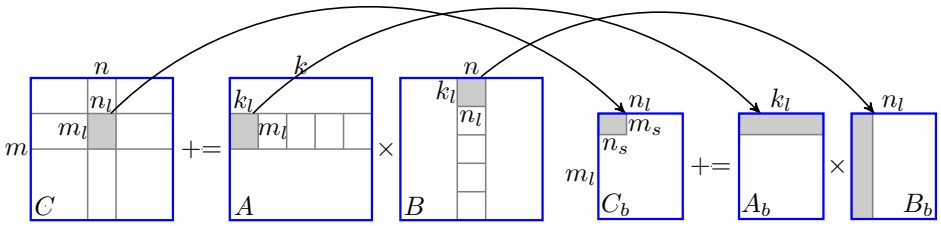
4.1 Implementations

We follow the strategy proposed by Matsumoto et al. [21] regarding their matrix partitioning technique in order to exploit multi-level memory hierarchies on GPU architectures, see Fig. 2. An adequate matrix partitioning improves significantly the reuse of data and keeps the computational units busy while performing memory transfers.

Our solution is different from Matsumoto's one, as we divide memory space among matrices, floating-point expansions, and superaccumulators. The latter may requires 78 times more storage than the matrix C in the non-optimized case (when superaccumulators are not reused). Thus, we use two levels of blocking in our matrix multiplication algorithms to amortize the cost of data accesses to the three levels of memory on GPUs, namely private (registers), local or data caches, and global. The first level focuses on enhancing the access latency between the global and local memories for each group of threads (or warp or work-group on GPUs). Suppose that m_l, n_l, and k_l are three block size multiple of m, n, and k respectively. Fig. 2a represents the partitioning of the matrices C, A, and B into blocks of sizes $m_l \times n_l, m_l \times k_l$, and $k_l \times n_l$, accordingly. Each $m_l \times n_l$ block of C is computed by a work-group that involves $m_l \times k$ blocks of A and $k \times n_l$ blocks of B.

This panel-panel multiplication iterates k/k_l times in the outermost loop of our xGEMM algorithm using the block-block multiplication. Thus, on each iteration the work-group updates each resulting $m_l \times n_l$ block of C with the product of an $m_l \times k_l$ block of A by a $k_l \times n_l$ block of B. This second level of blocking optimizes the use of private memory for each thread (work-item on GPUs). Fig. 2b shows further partitioning of matrices within their blocks in such a way that each work-item in the work-group is responsible for updating an $m_s \times n_s$ sub-block of C through the multiplication of an $m_s \times k_l$ sub-panel of A by a $k_l \times n_s$ sub-panel of B.

In order to ensure both reproducibility and accuracy of xGEMM, we use one floating-point expansion with error-free transformation per thread. When the accuracy provided by floating-point expansions is not enough, we switch to long accumulators that are allocated for each thread of a given work-group. However, this induces pressure on the memory hierarchy due to the required storage. So, we reuse both floating-point



(a) Work-group blocking.

(b) Work-item blocking.

Fig. 2: Partitioning of matrix-matrix multiplication.

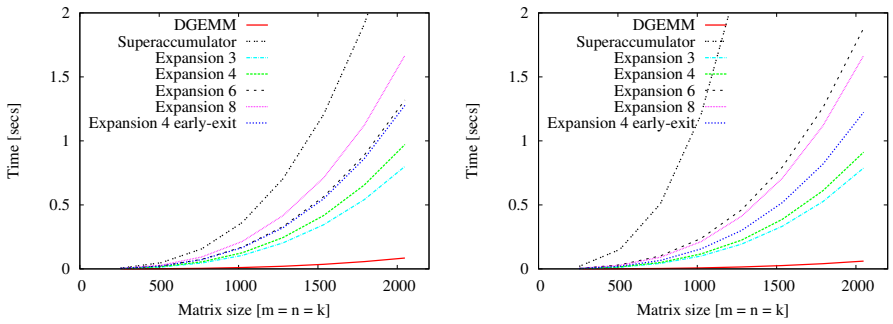
expansions and superaccumulators and aim at computing multiple elements of the resulting matrix with the same thread.

Our implementations attempt to get the maximum performance by using all resources of the considered GPU architectures: SIMD instructions, fused multiply-add, private and local memory as well as atomic instructions. We developed both unique and hand-tuned OpenCL implementations for NVIDIA and AMD GPUs.

We use a long accumulator of finite length that corresponds to the whole range of double precision floating-point numbers (4196 bits in case of `binary64`). We use such a long accumulator to avoid partial over/underflow that may occurs while accumulating partial product of the same sign. For instance, for matrices of size $n \times n$, only n partial-products need to be summed per resulting element leading to only $log_2(n)$ carry bits. With matrix size of $2^{20} \times 2^{20}$ that requires 8 Terabytes, only 20 extra bits are necessary to ensure that this phenomena will not occur.

4.2 Performance Results

As a baseline we consider the vectorized and parallelized non-deterministic double precision matrix multiplication. Figs. 3a and 3b present the measured time achieved by the matrix multiplication algorithms as a function of the matrix size n on two GPUs, see Tab. 1. Apart from "DGEMM", all implementations are ours: "Superaccumulator" is an implementation that relies solely on long accumulators and it is the slowest due to its extensive memory usage; "Expansions n" stand for implementations with floating-point expansions of various sizes; "Expansion 4 early-exit" is an optimized version of the expansion of size 4. The implementations with expansions deliver better performance than with superaccumulators only. Due to switching to the superaccumulator at the final stage of computing each resulting element as well as when the accuracy of expansions is not enough, the performance of implementations with expansions is bounded and it is at most 12 and 16 times off the DGEMM's performance on NVIDIA and AMD GPUs, respectively. We think that there is a possibility to improve these preliminary implementations in order to be within 10 times slower. Nevertheless, the computed results by our matrix multiplication algorithm are both reproducible and accurate.



(a) Platform A.

(b) Platform B.

Fig. 3: The matrix-matrix multiplication performance results on NVIDIA and AMD GPUs, see Tab. 1.

5 Related Works

To enhance reproducibility – defined as getting a bitwise identical floating-point result from multiple runs of the same code – Intel proposed a "Conditional Numerical Reproducibility" (CNR) in its MKL (Math Kernel Library). However, CNR is slow and does not give any guarantees on the accuracy of the result. Demmel and Nguyen recently introduced a family of algorithms for reproducible summation in floating-point arithmetic [22]. These algorithms always returns the same answer. They first compute an absolute bound of the sum and then round all numbers to a fraction of this bound. So, the addition of the rounded quantities is exact. Since the computed sum may be less accurate than the non-deterministic one, this solution offers no guarantees on the accuracy. It also induces a twofold slowdown as data transfers and reductions need to be performed twice: for computing the bound and the sum. As Section 4 shows, our algorithm is faster in the bandwidth-constrained scenarios with moderate dynamic ranges. Demmel and Nguyen have also improved the previous results [23] by using one single reduction step among nodes. Such an improvement yielded roughly $15\,\%$ overhead on 2048 processors compared to the Intel MKl's `dasum()`, but it shows 4.5 times slowdown on 32 processors. Demmel and Nguyen have extended their concept to reproducible BLAS routines, distributed in their ReproBLAS library[6]. For the moment of writing, the ReproBLAS library does not contain reproducible matrix multiplication.

6 Conclusions and Future Work

xGEMM is the core of the BLAS library and all the other BLAS-3 routines are virtually built on top of xGEMM. Furthermore, the development and automatic generation of linear algebra algorithms are driven by the goal of achieving best performance on various architectures. One step towards this goal is made by using blocked versions of algorithms

[6] `http://bebop.cs.berkeley.edu/reproblas/`

that are capable to obtain much higher performance compared to non-blocked algorithmic variants. This is achieved thanks to the usage of BLAS-3 routines, in particular xGEMM. Understanding such importance of the matrix multiplication routine, we target xGEMM and for the first time deliver both a multi-level reproducible and accurate approach as well as implementations of the same. Even though the performance can be argued (we think that a 10 times overhead at most for reproducible compute-bound algorithms is reasonable), the output of xGEMM is consistently reproducible and accurate, in terms of rounding-to-nearest, independently of threads scheduling and data partitioning.

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 many-core 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 [24].

Acknowledgement

We thank to Kazuya Matsumoto for kindly sharing with us the generated xGEMM code for the AMD GPU.

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).

References

1. Lawson, C.L., Hanson, R.J., Kincaid, D.R., Krogh, F.T.: Basic linear algebra subprograms for Fortran usage. ACM Transactions on Mathematical Software **5**(3) (September 1979) 308–323
2. Dongarra, J.J., Croz, J.D., Hammarling, S., Hanson, R.J.: An extended set of FORTRAN basic linear algebra subprograms. ACM Transactions on Mathematical Software **14**(1) (March 1988) 1–17
3. Dongarra, J.J., Croz, J.D., Hammarling, S., Duff, I.: A set of level 3 basic linear algebra subprograms. ACM Transactions on Mathematical Software **16**(1) (March 1990) 1–17
4. Whaley, R.C., Dongarra, J.J.: Automatically tuned linear algebra software. In: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM). Supercomputing '98, IEEE Computer Society (1998) 1–27
5. Goto, K., van de Geijn, R.A.: Anatomy of high-performance matrix multiplication. ACM Trans. Math. Softw. **34**(3) (May 2008)
6. Goto, K., van de Geijn, R.A.: High-performance implementation of the level-3 BLAS. ACM Trans. Math. Softw. **35**(1) (July 2008)
7. Fabregat-Traver, D., Aulchenko, Y., Bientinesi, P.: Solving sequences of generalized least-squares problems on multi-threaded architectures. Applied Mathematics and Computation **234** (May 2014) 606–617
8. Fabregat-Traver, D., Bientinesi, P.: Computing petaflops over terabytes of data: The case of genome-wide association studies. ACM Trans. Math. Softw. (2014) Accepted.
9. Bergman, K., al.: Exascale computing study: Technology challenges in achieving exascale systems. DARPA Report (September 2008)

10. Whitehead, N., Fit-Florea, A.: Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs. Technical report, NVIDIA (2011)
11. Corden, M.: Differences in floating-point arithmetic between Intel® Xeon® processors and the Intel® Xeon Phi™ coprocessor. Technical report, Intel (March 2013)
12. Doertel, K.: Best known method: Avoid heterogeneous precision in control flow calculations. Technical report, Intel (August 2013)
13. Kulisch, U., Snyder, V.: The Exact Dot Product As Basic Tool for Long Interval Arithmetic. Computing **91**(3) (March 2011) 307–313
14. Collange, S., Defour, D., Graillat, S., Iakymchuk, R.: 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)
15. IEEE Computer Society: IEEE Standard for Floating-Point Arithmetic. IEEE Standard 754-2008 (August 2008)
16. Higham, N.J.: Accuracy and stability of numerical algorithms, second ed. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA (2002)
17. Muller, J.M., Brisebarre, N., de Dinechin, F., Jeannerod, C.P., Lefèvre, V., Melquiond, G., Revol, N., Stehlé, D., Torres, S.: Handbook of Floating-Point Arithmetic. Birkhäuser (2010)
18. Li, X.S., Demmel, J.W., Bailey, D.H., Henry, G., Hida, Y., Iskandar, J., Kahan, W., Kang, S.Y., Kapur, A., Martin, M.C., Thompson, B.J., Tung, T., Yoo, D.J.: Design, implementation and testing of extended and mixed precision BLAS. ACM Trans. Math. Softw. **28**(2) (2002) 152–205
19. Hida, Y., Li, X.S., Bailey, D.H.: 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) 155–162
20. Knuth, D.E.: The Art of Computer Programming, Volume 2: Seminumerical Algorithms, third ed. Addison-Wesley (1997)
21. Matsumoto, K., Nakasato, N., Sakai, T., Yahagi, H., Sedukhin, S.G.: Multi-level optimization of matrix multiplication for gpu-equipped systems. In: ICCS. Volume 4 of Procedia Computer Science., Elsevier (2011) 342–351
22. Demmel, J., Nguyen, H.D.: Fast reproducible floating-point summation. In: Proceedings of the 21st IEEE Symposium on Computer Arithmetic, Austin, Texas, USA. (2013) 163–172
23. Demmel, J., Nguyen, H.D.: Numerical Reproducibility and Accuracy at ExaScale (invited talk). In: Proceedings of the 21st IEEE Symposium on Computer Arithmetic, Austin, Texas, USA. (April 2013) 235–237
24. Iakymchuk, R., Collange, S., Defour, D., Graillat, S.: ExBLAS – Exact BLAS `https://exblas.lip6.fr/`.