Reproducible, Accurately Rounded and Efficient BLAS

Chemseddine Chohra[1,2,3(✉)], Philippe Langlois[1,2,3], and David Parello[1,2,3]

[1] Univ. Perpignan Via Domitia, Digits, Architectures et Logiciels Informatiques, 66860 Perpignan, France
{Chemseddine.Chohra,Philippe.Langlois,David.Parello}@univ-perp.fr
[2] Univ. Montpellier II, Laboratoire d'Informatique Robotique Et de Microélectronique de Montpellier, UMR 5506, 34095 Montpellier, France
[3] CNRS, Paris, France

Abstract. Numerical reproducibility failures rise in parallel computation because floating-point summation is non-associative. Massively parallel and optimized executions dynamically modify the floating-point operation order. Hence, numerical results may change from one run to another. We propose to ensure reproducibility by extending as far as possible the IEEE-754 correct rounding property to larger operation sequences. We introduce our RARE-BLAS (Reproducible, Accurately Rounded and Efficient BLAS) that benefits from recent accurate and efficient summation algorithms. Solutions for level 1 (asum, dot and nrm2) and level 2 (gemv) routines are presented. Their performance is studied compared to the Intel MKL library and other existing reproducible algorithms. For both shared and distributed memory parallel systems, we exhibit an extra-cost of 2× in the worst case scenario, which is satisfying for a wide range of applications. For Intel Xeon Phi accelerator a larger extra-cost (4× to 6×) is observed, which is still helpful at least for debugging and validation steps.

1 Introduction and Background

The increasing power of supercomputers leads to a higher amount of floating-point operations to be performed in parallel. The IEEE-754 [8] standard defines the representation of floating-point numbers and requires the addition operation to be correctly rounded. However because of errors generated by every addition, the accumulation of more than two floating-point numbers is non-associative. The combination of the non-deterministic behavior in parallel programs and the non-associativity of floating-point accumulation yields non-reproducible numerical results.

Numerical reproducibility is important for debugging and validating programs. Some solutions have been given in parallel programming libraries. Static data scheduling and deterministic reduction ensure the numerical reproducibility of the library OpenMP. Nevertheless the number of threads has to be set for all runs [15]. Intel MKL library (starting with 11.0 release) introduces CNR [15]

F. Desprez et al. (Eds.): Euro-Par 2016 Workshops, LNCS 10104, pp. 609–620, 2017.
DOI: 10.1007/978-3-319-58943-5_49

(Conditional Numerical Reproducibility). This feature limits the use of instruction set extensions to ensure numerical reproducibility between different architectures. Unfortunately this decreases significantly the performance especially on recent architectures, and requires the number of threads to remain the same from run to run to ensure reproducible results.

First algorithmic solutions are proposed in [4]. Algorithms *ReprodSum* and *FastReprodSum* ensure numerical reproducibility independently of the operation order. Therefore numerical results do not depend anymore on hardware configuration. The performance of these latter is improved with the algorithm *OneReduction* [6] by relying on indexed floating-point numbers [5] and requiring a single reduction operation to reduce the communication cost on distributed memory parallel platforms. However, those solutions do not improve accuracy. The computed result even if it is reproducible, it is still exposed to accuracy problems. Especially when we address an ill-conditioned problem.

Another way to guarantee reproducibility is to compute correctly rounded results. Recent works [1,2,11] show that a accurately rounded floating-point summation can be calculated with very little or even no extra-cost. With accurately rounded we mean that the result is either correctly rounded (the nearest floating-point number to the exact result) or faithfully rounded (one of the two floating-point numbers that surround the exact result). We have analyzed in [1] different summation algorithms, and identified those suited for an efficient parallel implementation on recent hardware. Parallel algorithms for correctly rounded *dot* and *asum* and for a faithfully rounded *nrm*2 have been designed relying on the most efficient summation algorithms. Their implementation exhibits interesting performance with 2× extra-cost in the worst case scenario on shared memory parallel systems [1].

In this paper we extend our approach to an other type of parallel platforms and to higher BLAS level. We consider the matrix-vector multiplication from the level 2 BLAS. We complete our shared memory parallel implementation with solution for a distributed memory model, and confirm its scalability with tests on the Occigen supercomputer[1]. We also present tests on the Intel Xeon Phi accelerator to illustrate the portability and appreciate the efficiency of our implementation on a many-core accelerator. The efficiency of our correctly rounded dot product scales well on distributed memory parallel systems. Compared to optimized but not reproducible implementations, it has no substantial extra-cost up to about 1600 threads (128 sockets, 12 cores). On Intel Xeon Phi accelerator the extra-cost increases up to 6× mainly because our solution benefits less from the high memory bandwidth of this architecture compared to MKL's implementation. Nevertheless they still could be useful for validation, debugging or for applications that require precision or reproducible results.

This paper is organized as follows. Section 2 presents our sequential algorithms for reproducible and accurate BLAS. Parallel versions are presented in Sect. 3. Section 4 is devoted to implementation and detailed results, and Sect. 5 includes some conclusions and the description of future work.

[1] https://www.cines.fr/en/occigen-the-new-supercomputeur/.

2 Sequential RARE BLAS

We present the algorithms for accurately rounded BLAS. This section starts briefly recalling our sequential level 1 BLAS subroutines (*dot*, *asum* and *nrm*2) already introduced in [1]. Then the accurately rounded matrix-vector multiplication is introduced.

2.1 Sequential Algorithms for the Level 1 BLAS

In this section we focus on the sum of absolute values (*asum*), the dot product (*dot*), and the euclidean norm (*nrm*2).

Sum of Absolute Values. The condition number of a sum is defined as $cond(\sum p_i) = \sum |p_i| / \sum p_i$. For the sum of absolute values the condition number is known to equal 1. This justifies the use of algorithm *SumK* [13].

Picking carefully the value of K ensures that computing $asum(p)$ as $SumK(p)$ is faithfully rounded. Such appropriate value of K only depends on the vector size. We have $K = 2$ for $n \leq 2^{25}$, and $K = 3$ for $n \leq 2^{34}$. For $n \leq 2^{39}$ which represents $4TB$ of data, $K = 4$ is sufficient [1].

Dot Product. Using Dekker's *TwoProd* [3], the dot product of two n-vectors can be transformed without error to a sum of a $2n$-vector. The sum of the transformed vector is correctly rounded using a mixed solution. For small vectors that fit in high level cache and that can be reused with no memory extra-cost, the algorithm *FastAccSum* [14] is used, the algorithms *HybridSum* [17] or *OnlineExact* [18] are preferred for large vectors (both algorithms exhibit barely the same performance). The idea of these algorithms is to add elements that share the same exponent to a dedicated accumulator —in practice one or two floating-point numbers respectively. Therefore, the $2n$-vector is error-free replaced by a smaller accumulator vector (of size 4096 or 2048 respectively). Here the result and the error calculated with *TwoProd* are directly accumulated. Finally we apply the distillation algorithm *iFastSum* [17] to the accumulator vector to compute the correctly rounded dot product.

Euclidean Norm. The euclidean norm of a vector p is defined as $(\sum p_i^2)^{1/2}$. The sum $\sum p_i^2$ can be correctly rounded using the previous dot product. Finally, we apply a square root that returns a faithfully rounded euclidean norm [7]. numbers that enclose the exact result). This does not allow us to compute a correctly rounded norm-2 but this faithful rounding is reproducible.

2.2 Sequential Algorithms for the Level 2 BLAS

Matrix-vector multiplication is defined in the BLAS as $y = \alpha A \cdot x + \beta y$. In the following, we denote $y_i = \alpha a^{(i)} \cdot x + \beta y_i$, where a$^{(i)}$ is the ith row of matrix A.

Algorithm 1 details our proposed reproducible computation: (1) The first step transforms the dot product $a^{(i)} \cdot x$ into a sum of non-overlapping floating-point numbers. This error-free transform uses a minimum extra storage: the transformed result is stored in one array of maximum size 40 (the floating-point number range divided by the mantissa size). This process is done in different ways depending on the vector size. For small vectors we use $TwoProd$ to create a $2n$-vector. The distillation algorithm iFastSum [17] is then used to reduce the vector size. For large ones we do not create the $2n$-vector. The result and the error of $TwoProd$ are directly accumulated in accordance to their exponent as requested by $HybridSum$ or $OnlineExact$. After the dot product has been error-free transformed to a smaller vector, the same distillation process is applied. Let us remark that this step does not compute the dot product $a^{(i)} \cdot x$ but transforms it without error in a small floating point vector. (2) The second step evaluates multiplications by the scalars α and β using $TwoProd$. Again data is transformed with no error. (3) Finally we distillate the results of the previous steps to get a correctly rounded result of $y_i = \alpha a^{(i)} \cdot x + \beta y_i$. The same process is repeated for each row of the matrix A.

3 Parallel RARE BLAS

This section presents our parallel reproducible version of Level 1 and 2 BLAS.

3.1 Parallel Algorithms for the Level 1 BLAS

Sum of Absolute Values. The natural parallel version of algorithm $SumK$ introduced in [16] is used for parallel $asum$. Two stages are required. (1) The first one consists in applying the sequential algorithm $SumK$ on local data without performing the final error compensation. So we end with K floating point numbers per thread. (2) The second stage gathers all these numbers in a single vector. Afterwards the master thread applies a sequential $SumK$ on this vector.

Dot Product and Euclidean Norm. Figure 1 illustrates our correctly rounded dot product. Note that for step 1, the two entry vectors of the dot product are equally split between the threads. We use the same transformation as the one presented in Sect. 2.2 to error-free transform the local dot product. The accumulation of elements with the same exponent is only done for large vectors. As before C' vector size equals 4096 or 2048. For small vectors we create a $2n$-vector using only $TwoProd$. Distillation in step 2 mainly aims at reducing the communication cost of the union that yields the vector C. Since all transformations up to C are error-free, the final call to $iFastSum$ in step 3 returns the correctly rounded result for the dot product.

The euclidean norm is faithfully rounded as explained for the sequential case. Even if we do not calculate a correctly rounded result for euclidean norm, it is guaranteed to be reproducible because it only depends on a reproducible dot product.

```
Data: $A : m \times n$-matrix; $x : n$-vector; $y : m$-vector; $\alpha, \beta$ :double precision float;
Result: the input vector $y$ updated as $y = \alpha A \cdot x + \beta y$;
for $row$ in $1 : m$ do
    $currentrow = A[row, 1 : n]$;
    if $currentrow$ and $x$ fit in cache then
        declare $2n$-vector $C$;
        for $column$ in $1 : n$ do
            $(result, error) = TwoProd(currentrow[column], x[column])$;
            $C[column] = result$; $C[n + column] = error$;
        end
    else
        declare the accumulator vector $C$;
        for $column$ in $1 : n$ do
            $(result, error) = TwoProd(currentrow[column], x[column])$;
            accumulate $result$ and $error$ to corresponding accumulator in $C$;
        end
    end
    declare a vector $distil$;
    $distil = distillationProcess(C)$;
    declare a vector $finalTransformation$;
    size = sizeOf(distil);
    /* Step 2 : multiply by the scalars $\alpha$ and $\beta$ */
    for $i$ in $1 : size$ do
        $(result, error) = TwoProd(distil[i], \alpha)$;
        $finalTransformation[i] = result$;
        $finalTransformation[size + i] = error$;
    end
    $(result, error) = TwoProd(y[row], \beta)$;
    $finalTransformation[size \times 2 + 1] = result$;
    $finalTransformation[size \times 2 + 2] = error$;
    /* Step 3 : use iFastSum to calculate the correctly rounded
       result */
    $y[row] = iFastSum(finalTransformation)$;
end
```

Algorithm 1: Correctly rounded matrix-vector multiplication

3.2 Parallel Algorithms for the Level 2 BLAS

For matrix-vector multiplication, several algorithms are available according to the matrix decomposition. The three possible ones are: row layout, column layout and block decomposition. We opt for row layout decomposition because the algorithms we use are more efficient when working on large vectors. This choice also avoids the additional cost of reduction.

Figure 2 shows how our parallel matrix-vector multiplication is performed. The vector x must be attainable for all threads. On the other side the matrix A and the vector y are split into p parts where p is the number of threads. Each

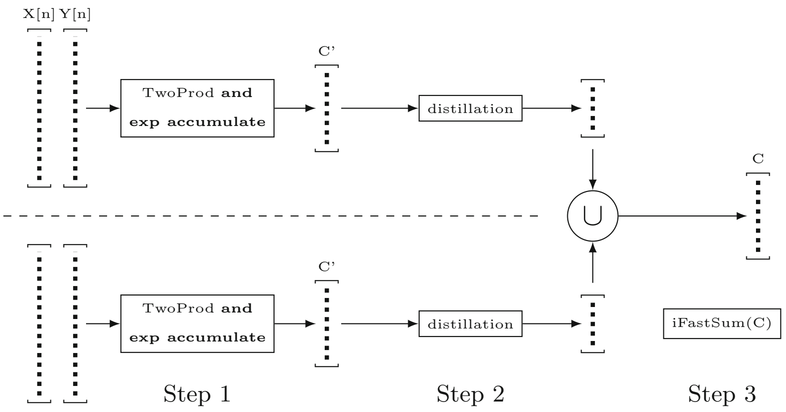


Fig. 1. Parallel algorithm for correctly rounded dot product

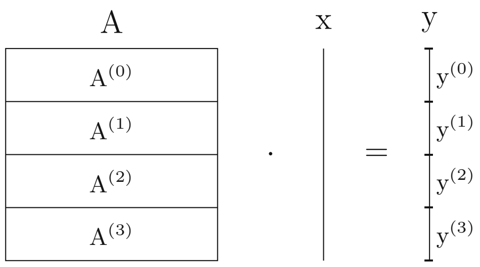


Fig. 2. Parallel algorithm for correctly rounded matrix-vector multiplication

thread handles the panel $A^{(i)}$ of A and the sub-vector $y^{(i)}$ of y. $y^{(i)}$ is updated with $\alpha A^{(i)} \cdot x + \beta y^{(i)}$ as described in Sect. 2.2.

4 Test and Results

In this section, we illustrate the performance and accuracy results of our proposed solution to accurate and reproducible level 1 and level 2 BLAS.

4.1 Experimental Framework

We consider the three frameworks described in Table 1. They are significant of today's practise of floating-point computing.

We test the efficiency of the sequential and the shared memory parallel implementation on platform **A**. Platform **B** illustrates the many core accelerator use. The scalability of our approach on large supercomputers is exhibited on platform **C** (Occigen supercomputer). Only the dot product has been tested on platform **C**. Data for dot product are generated as in [13]. The same idea is used

Table 1. Experimental frameworks

A	Processor	dual Xeon E5-2650 v2 16 cores (8 per socket), No hyper-threading. L1/L2 = 32/256 KB per core. L3 = shared 20 MB per socket.
	Bandwidth	59,7 GB/s
	Compiler	Intel ICC 16.0.0
	Options	-O3 -xHost -fp-model double -fp-model strict -funroll-all-loops
	Libraries	Intel OpenMP 5. Intel MKL 11.3.
B	Processor	Intel Xeon Phi 7120 accelerator, 60 cores, 4 threads per core. L1/L2 = 32/512 KB per core.
	Bandwidth	352 GB/s
	Compiler	Intel ICC 16.0.0
	Options	-O3 -mmic -fp-model double -fp-model strict -funroll-all-loops
	Libraries	Intel OpenMP 5. Intel MKL 11.3.
C	Processor	4212 Xeon E5-2690 v3 (12 cores per socket), No hyper-threading. L1/L2 = 32/256 KB per core. L3 = shared 30 MB per socket.
	Bandwidth	68 GB/s
	Compiler	Intel ICC 15.0.0
	Options	-O3 -xHost -fp-model double -fp-model strict -funroll-all-loops
	Libraries	Intel OpenMP 5. Intel MKL 11.2. OpenMPI 1.8

to generate condition dependent data for matrix-vector multiplication (multiple dot products with a shared vector).

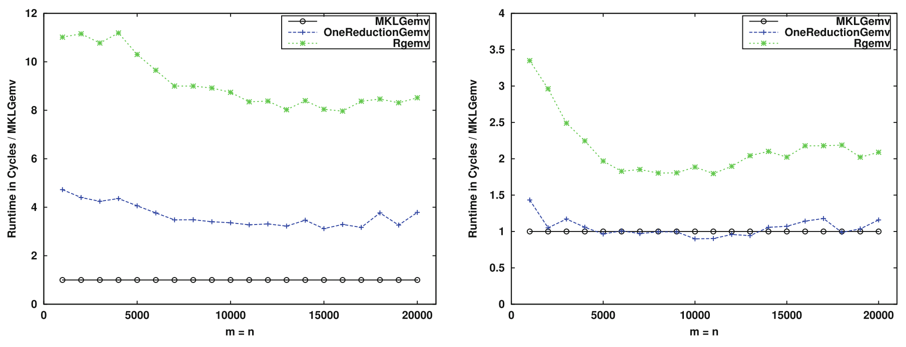
4.2 Implementation and Performance Results

We compare the performance results of our implementation to the highly optimized Intel MKL library, and to implementations based on algorithm OneReduction used on the library ReproBLAS [12]. We have implemented an OpenMP parallel version of this algorithm since ReproBLAS offers only an MPI parallel version. We derive reproducible version of *dot*, *nrm*2, *asum* and *gemv* by replacing all non-associative accumulations by the algorithm OneReduction [6]. These versions are denoted *OneReductionDot*, *OneReductionAsum*, *OneReductionNrm*2 and *OneReductionGemv*.

CNR feature [15] is not considered because it does not guarantee reproducibility between sequential and parallel runs. Running time is measured in cycles using the RDTSC instruction. In the parallel case, RDTSC calls have been made out of parallel region before and after function calls. We take the minimum running time over 8 executions for *gemv* and 16 executions for other routines to improve result consistency. We note up to 3% difference in number of cycles between different runs. This difference is due to turbo boost and operating system interruption and it is known that performance results can not be exactly reproduced.

Sequential Performance. Tests are run on platform **A**. Results for *dot*, *asum* and *nrm*2 are presented in [1]. These accurately rounded versions exhibit respectively 5×, 2× and 9× extra-cost.

Our *Rgemv* matrix-vector multiplication computes a correctly rounded result using *iFastSum* for small matrices and *HybridSum* for large ones, this latter being slightly more efficient than *OnlineExact* on both platforms **A** and **B**. As shown in Fig. 3a, *Rgemv* costs 8 times more compared to MKL in this sequential case.



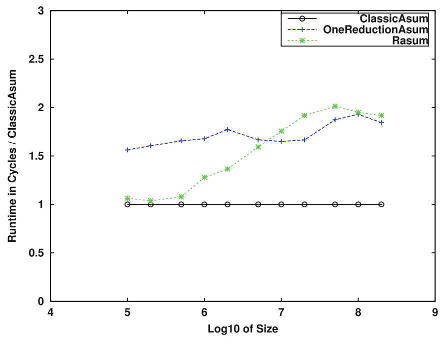
(a) Sequential (b) Parallel

Fig. 3. Extra-cost of correctly rounded matrix-vector multiplication (cond=10^8)

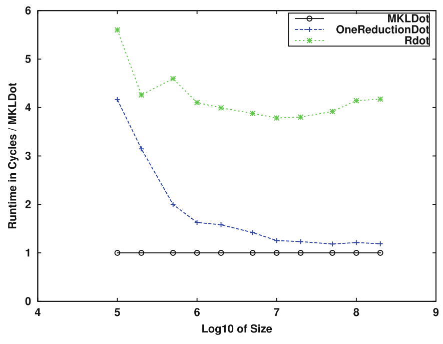
Shared Memory Parallel Performance. Tests have also been done on platform **A** where 16 cores are used with no hyper-threading. We use OpenMP to implement our parallel algorithms. As for the sequential case, results for *dot*, *asum* and *nrm*2 are presented in [1]. The *dot* and *asum* do not exhibit any extra-cost compared to classic versions, and *nrm*2 has 2× extra-cost.

For the matrix-vector multiplication, the correctly rounded algorithm costs about twice more compared to MKL as shown in Fig. 3b. As in the sequential case, *MKLGemv* certainly use cache blocking and so benefits from a better memory bandwidth use. Nevertheless our parallel implementation scales well and its extra-cost now reaches the 2× ratio.

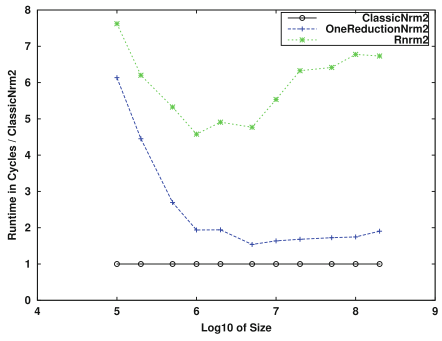
Xeon Phi Performance. There is not much difference between implementation for Xeon Phi and previous CPU ones. Thread level parallelism is implemented using OpenMP and intrinsic functions are used to benefit from the available instruction set extensions. A FMA (Fused Multiply and Add) is also available. Therefore *TwoProd* is replaced by 2*MultFMA* [10] which only requires two FMAs to compute the product and its error, and so improves performance.



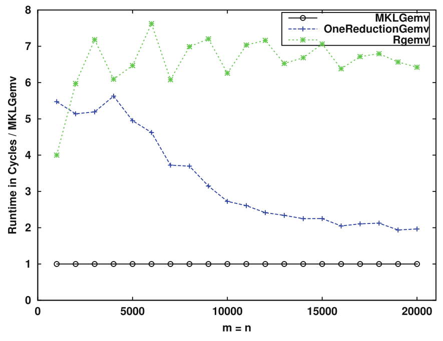
(a) Sum of absolute values



(b) Dot product (cond = 10^{16})



(c) Euclidean norm



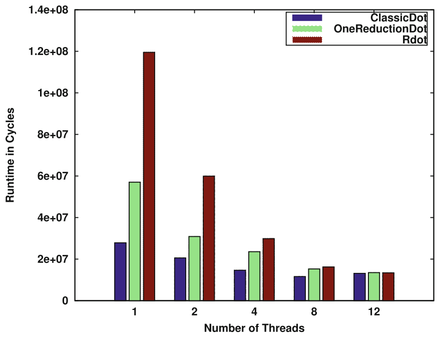
(d) Matrix-vector multiply (cond = 10^{8})

Fig. 4. Extra-cost of Xeon Phi implementation compared to classical algorithms

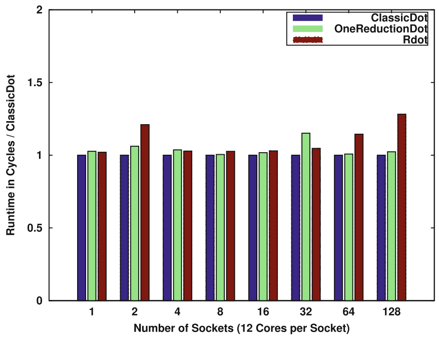
Figure 4 exhibits respective ratios of 2×, 4×, 6× and 6× for asum, dot product, euclidean norm and matrix-vector multiplication. So the extra-cost of accurately rounded implementations is larger for this accelerator than for the CPU. Indeed MKL based implementations of these memory bounded routines benefit from both higher memory bandwidth and large vector capabilities (AVX-512) provided by the Xeon Phi more than our accurate ones. Note that on our correctly rounded dot product algorithms there is no efficient way to vectorize the accumulation to the elements of vector C since the access to those elements is not contiguous (see Algorithm 1 and Fig. 1).

Distributed Memory Parallel Performance. Finally we present performance on distributed memory systems. Only dot product tests have been run on the Occigen supercomputer. In this case we have two levels of parallelism: OpenMP is used for thread level parallelism on a single socket, and OpenMPI library for socket communication. The algorithm scalability is tested on a single data set with input vectors of length 10^7 and condition number is 10^{32}.

Figure 5a shows the scalability for a single socket configuration. It is not a surprise that *MKLDot* does not scale so far since it is quickly limited by the memory bandwidth. *OneReductionDot* and *Rdot* scale well up to exhibit no extra-cost compared to optimized *MKLDot*. Again such scaling occurs until being limited by the memory bandwidth.



(a) Single socket



(b) Multi socket normalized by ClassicDot

Fig. 5. Performance of the distributed memory parallel implementations.

Performance for the multi socket configuration is presented in Fig. 5b. X-axis shows the number of sockets where all the 12 available cores are used. Y-axis shows execution time normalized to *ClassicDot* (socket local *MKLDot*s followed by a MPI sum reduction).

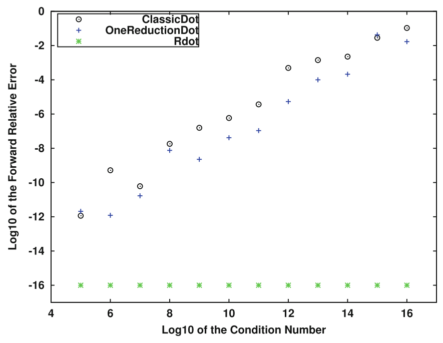
Algorithms *Rdot* and *OneReductionDot* stay almost as efficient as *ClassicDot*. All algorithms exhibit similar performance because they rely all on a single communication.

4.3 Accuracy Results

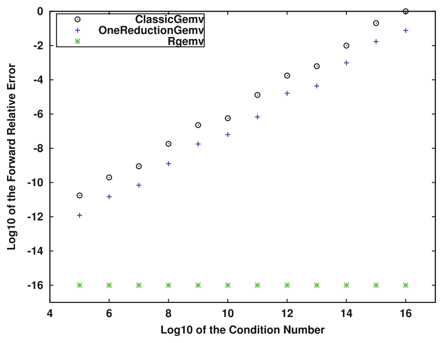
We present here accuracy results for *dot* and *gemv* variants. In both Fig. 6a and b, we show the relative error according to the condition number of the problem. Relative errors are calculated according to MPFR library [9] results. The two subroutines *nrm*2 and *asum* are excluded from this test because condition number is fixed for both of them. The condition number for the dot product is defined as $cond(\sum X_i \cdot Y_i) = \sum |X_i| \cdot |Y_i| / |\sum X_i \cdot Y_i|$. In almost all cases, solutions based on algorithm *OneReduction* besides being reproducible are more accurate than MKL. However, for ill-conditioned problems both MKL and *OneReduction* derived implementation give worthless results. On the other side RARE-BLAS subroutines ensure that results are always correctly rounded independently from the condition number.

5 Conclusion and Future Work

We have presented algorithms that compute reproducible and accurately rounded results for BLAS. Level 1 and 2 subroutines have been addressed in



(a) Accuracy results for *dot*



(b) Accuracy results for *gemv*

Fig. 6. Accuracy results for *dot* and *gemv*

this paper. Implementations of these algorithms have been tested on three platforms significant of the floating-point computing practice. While existing solutions tackle only the reproducibility problem, our proposed solutions aim at ensuring both reproducibility and the best precision. We compare them to optimized Intel MKL implementations. We measure interesting performance on CPU based parallel environments. Extra-cost on CPU when all available cores are used is at worst twice. Nevertheless performance on Xeon Phi accelerator is lagging behind: extra-cost is between 4 and 6 times more. Nevertheless, our algorithms remain efficient enough to be used for validation or debugging programs, and also for parallel applications that can sacrifice performance to increase the accuracy and the reproducibility of their results.

Our plan for future development includes achieving reproducibility and precision for other BLAS subroutines. We are currently designing an accurate and reproducible version of triangular solver. Other Level 3 BLAS routines will be addressed even if the performance gap with optimized libraries will enforce the previously identified restriction of the application scope.

References

1. Chohra, C., Langlois, P., Parello, D.: Implementation and Efficiency of Reproducible Level 1 BLAS (2015). http://hal-lirmm.ccsd.cnrs.fr/lirmm-01179986
2. Collange, S., Defour, D., Graillat, S., Iakymchuk, R.: Numerical reproducibility for the parallel reduction on multi- and many-core architectures. Parallel Comput. **49**(C), 83–97 (2015). http://dx.doi.org/10.1016/j.parco.2015.09.001
3. Dekker, T.J.: A floating-point technique for extending the available precision. Numer. Math. **18**, 224–242 (1971)
4. Demmel, J.W., Nguyen, H.D.: Fast reproducible floating-point summation. In: Proceedings of 21th IEEE Symposium on Computer Arithmetic, Austin, Texas, USA (2013)
5. Demmel, J.W., Nguyen, H.D.: Toward Hardware Support for Reproducible Floating-Point Computation. In: SCAN 2014, Würzburg, Germany, September 2014

6. Demmel, J.W., Nguyen, H.D.: Parallel reproducible summation. IEEE Trans. Comput. **64**(7), 2060–2070 (2015)
7. Graillat, S., Lauter, C., Tang, P.T.P., Yamanaka, N., Oishi, S.: Efficient calculations of faithfully rounded l2-norms of n-vectors. ACM Trans. Math. Softw. **41**(4), 24:1–24:20 (2015). http://doi.acm.org/10.1145/2699469
8. IEEE Task P754: IEEE 754–2008, Standard for Floating-Point Arithmetic. Institute of Electrical and Electronics Engineers, New York, August 2008
9. The MPFR library (2004). http://www.mpfr.org/. Accessed 8 July 2016
10. 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, Boston (2010)
11. Neal, R.M.: Fast exact summation using small and large superaccumulators. CoRR abs/1505.05571 (2015). http://arxiv.org/abs/1505.05571
12. Nguyen, H.D., Demmel, J., Ahrens, P.: ReproBLAS: Reproducible BLAS. http://bebop.cs.berkeley.edu/reproblas/
13. Ogita, T., Rump, S.M., Oishi, S.: Accurate sum and dot product. SIAM J. Sci. Comput. **26**(6), 1955–1988 (2005)
14. Rump, S.M.: Ultimately fast accurate summation. SIAM J. Sci. Comput. **31**(5), 3466–3502 (2009)
15. Todd, R.: Run-to-Run Numerical Reproducibility with the Intel Math Kernel Library and Intel Composer XE 2013. Intel Corporation, Technical report (2013)
16. Yamanaka, N., Ogita, T., Rump, S., Oishi, S.: A parallel algorithm for accurate dot product. Parallel Comput. **34**(68), 392–410 (2008)
17. Zhu, Y.K., Hayes, W.B.: Correct rounding and hybrid approach to exact floating-point summation. SIAM J. Sci. Comput. **31**(4), 2981–3001 (2009)
18. Zhu, Y.K., Hayes, W.B.: Algorithm 908: online exact summation of floating-point streams. ACM Trans. Math. Softw. **37**(3), 37:1–37:13 (2010)