Reproducible, Accurately Rounded and Efficient BLAS

Chemseddine Chohra, Philippe Langlois, David Parello

Reproducible, Accurately Rounded and Efficient BLAS

Chemseddine Chohra*, Philippe Langlois* and David Parello*

Univ. Perpignan Via Domitia, Digits, Architectures et Logiciels Informatiques, F-66860, Perpignan. Univ. Montpellier II, Laboratoire d'Informatique Robotique et de Microélectronique de Montpellier, UMR 5506, F-34095, Montpellier. CNRS. 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 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 to 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 same number of threads is still required [13]. Intel MKL library (starting with 11.0 release) introduces CNR [13]

* firstname.lastname@univ-perp.fr

(Conditional Numerical Reproducibility). This feature limits the use of instruction set extensions to ensure numerical reproducibility between different architectures. Unfortunately this decreases dramatically the performance especially on recent architectures.

First algorithmic solutions are proposed in [4]. Algorithms *ReprodSum* and *FastReprodSum* ensure numerical reproducibility independently from the operation order. Therefore numerical results do not depend 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.

Another way to guarantee reproducibility is to compute accurately rounded results. Recent works [2, 1, 11] show that an accurately rounded floating-point summation can be calculated with very little or even no extra-cost. We have analyzed in [1] different summation algorithms, and identified those suited for an efficient parallel implementation on recent hardware. Algorithms for correctly rounded *dot* and *asum* and for faithfully rounded $nrm2$ have been designed relying on the most efficient summation algorithms. Their implementation exhibits interesting performance with $2\times$ extra-cost in the worst case scenario on shared memory parallel systems.

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 solutions for distributed memory model, and confirm its scalability with tests on the Occigen supercomputer[1]. We also present tests on Intel Xeon Phi accelerator to illustrate the portability and appreciate the efficiency of our implementation on many-core accelerator. Our efficiency of our correctly rounded dot product scales well on distributed memory parallel systems. It has no substantial extra-cost on up to 128 sockets with 12 used cores on each socket. On Intel Xeon Phi accelerator the extra-cost of our algorithms is up to $6\times$. Still they could be useful for validation, debugging or applications that require supplemental 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 section 3. Section 4 is devoted to implementation and detailed results,

2 Sequential RARE BLAS

We present the algorithms for accurately rounded BLAS. This section starts with level 1 BLAS subroutines (*dot*, *asum* and $nrm2$). Then the accurately rounded matrix-vector multiplication is introduced.

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

2.1 Sequential Algorithms for the Level 1 BLAS

We only focus on the sum of absolute values ($asum$), the dot product (dot), and the euclidean norm ($nrm2$). Some other level 1 BLAS subroutines do not suffer of reproducibility problem. We recall our Sequential solutions already introduced in [1]

Sum of Absolute Values The condition number for the sum of absolute values is known to equal 1. This motivates to use algorithm $SumK$ [12] to compute a correctly rounded $asum$ as $SumK(p)$ where all p_i are non-negative. Its relative accuracy is bounded as:

$$\frac{|asum(p) - \sum p_i|}{|\sum p_i|} \leq \frac{(n \cdot u)^K}{1 - (n \cdot u)^K} + u, \tag{1}$$

where u is the computing precision ($u = 2^{53}$ for IEEE-754 binary64) and n the length of the sum. Picking carefully the value of K such that $(n \cdot u)^K < u/(u+2)$, $asum$ will be correctly rounded. The 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 occupies $4TB$ of data, the appropriate value for K is 4.

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 hold in high level cache and that can be reused with no memory extra-cost, the algorithm $FastAccSum$ [14] is used. Algorithms $HybridSum$ [16] or $OnlineExact$ [17] are preferred for large vectors. The performance of both algorithms is almost the same and the choice depends on the execution environment.

The idea of algorithms $HybridSum$ and $OnlineExact$ is to add elements that share a same exponent to a dedicated accumulator — in practice one or two floating-point numbers respectively. Therefore, the $2n$-vector is replaced by a smaller accumulator vector. The result and the error calculated with $TwoProd$ can be directly accumulated in accordance with their exponents. Finally we apply $iFastSum$ [16] algorithm on the accumulator vector to compute the correctly rounded dot product.

Euclidean Norm The euclidean norm of a vector p is defined as $\sqrt{\sum p_i^2}$. The sum $\sum p_i^2$ can be accurately rounded using the previous dot product. Finally, we apply a square root to return a faithfully rounded euclidean norm [7]. This algorithm does not round correctly according to the IEEE-754 standard. We recall that a faithfully rounded value is one of the two floating-point numbers that enclose the exact result. Nevertheless we mention here that this computation is reproducible, thanks to IEEE-754 standard.

2.2 Sequential Algorithm for the Level 2 BLAS

We consider the matrix-vector multiplication problem 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 i^{th} row of matrix A. Algorithm 1 details the following process.
(1) The first step is to transform the dot product $a^{(i)} \cdot x$ into a sum of non-overlapping floating-point numbers. This error free transform the dot product uses a minimum extra storage: the transform result is stored in one array of maximum size 40 (the range of floating-point numbers 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. A distillation algorithm **??** 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 have been error free transformed to a smaller vector, the same distillation process is applied. (2) The second step evaluates multiplications by the scalers α and β using $TwoProd$. Note that until now data have been transformed with no error. (3) Finally we apply a summation algorithm on the results of the previous step 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. Blocking is not used in our algorithm. For small datasets that hold in $L1$ cache blocking does not improve performance. For larger datasets algorithms $HybridSum$ or $OnlineExact$ are used. The accumulators (2048 and 4096 floating-point numbers respectively) need to be stored in memory for each dot product. If we use blocks, the accumulator memory can not be freed until one row of blocks is processed. This would increase the memory extra-cost and prevents the efficient use of cache when blocking.

3 Parallel RARE BLAS

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

3.1 Parallel Level 1 BLAS

Sum of Absolute Values The natural parallel version of algorithm $SumK$ introduced in [15] is used for parallel $asum$. Two stages are required. (1) The first one consists in applying the sequential algorithm $SumK$ on local data. But the final error compensation is not performed. Therefore, we end with K floating point numbers per thread. (2) The second stage gathers all these floating point numbers in a single vector. Afterwards the master thread applies a sequential $SumK$ on this vector.

Dot Product and Euclidean Norm Fig. 1 illustrates our correctly rounded dot product algorithm. Note that for step 1, the two entry vectors of the dot product are equally split between the threads. We use the same transformation

Data: $A : m \times n$-matrix; $x : n$-vector; $y : m$-vector; α, β :double precision float;
Result: the input vector y updated as $y = \alpha A \cdot x + \beta y$.
for *row in* $1 : m$ **do**
  $currentline = A[row, 1 : n]$;
  if *currentrow and x hold 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**
      $(result, error) = TwoProd(currentline[column], x[column])$;
      accumulate *result* and *error* to corresponding accumulator in C;
    end
  end
  declare a vector *distil*;
  $distil = distillationProcess(C)$;
  declare a vector $finalTransformation$;
  for *i in* $1 : size$ **do**
    $(result, error) = TwoProd(distil[i], \alpha)$;
    $finalTransformation[i] = result$;
    $finalTransformation[size + i] = error$;
  end
  $(result, error) = TwoProd(y[row], \beta)$;
  $finalTransformation[size \times 2 + 1] = result$;
  $finalTransformation[size \times 2 + 2] = error$;
  return $iFastSum(finalTransformation)$;
end

Algorithm 1: Correctly rounded matrix-vector multiplication

as the one presented in Section 2.2 to error free transform the local dot product. The accumulation of elements with the same exponent is only done for large vectors. In this case the size of vector C' is either 2048 if $HybridSum$ transformation is used, or 4096 for $OnlineExact$. For small vectors we create a $2n$-vector using only $TwoProd$. Distillation in step 2 mainly aims at reducing the communication cost. Since all done transformations to compute C are error free, the final call to $iFastSum$ in step 3 returns the correctly rounded result for the dot product.

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 depends on a reproducible correctly rounded dot product.

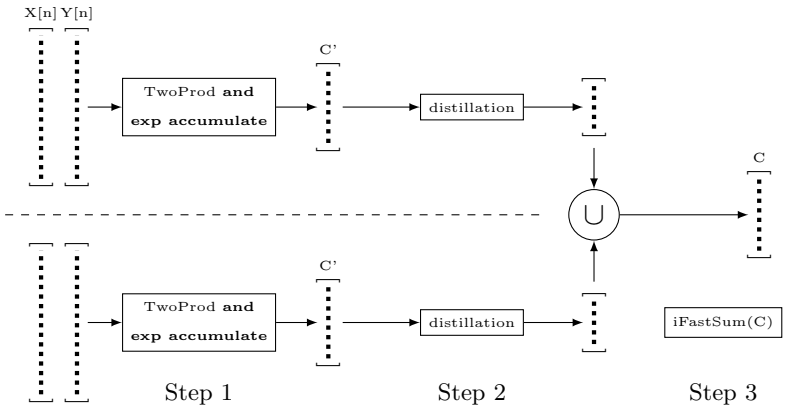


Fig. 1: Parallel Algorithm for Correctly Rounded Dot Product

3.2 Parallel 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.

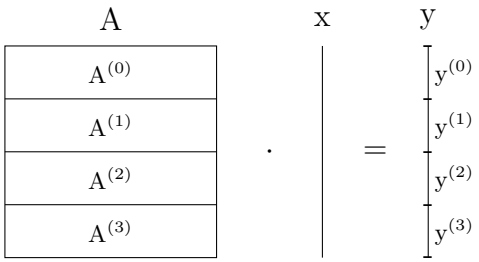


Fig. 2: Parallel Algorithm for Correctly Rounded Matrix-Vector Multiplication

We show in Figure 2 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 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 section 2.2.
Note that a classical parallel algorithm which uses this decomposition does not suffer from reproducibility problem. Although block algorithm should be more efficient but suffers from reproducibility problem.

4 Test and Results

In this section, we illustrate the efficiency of our proposed solution to reproducible level 1 and level 2 BLAS.

4.1 Experimental framework

We make performance tests on three frameworks significant of today's practice of floating-point computations. These frameworks are detailed in Table 1.

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.

Table 1: Experimental frameworks

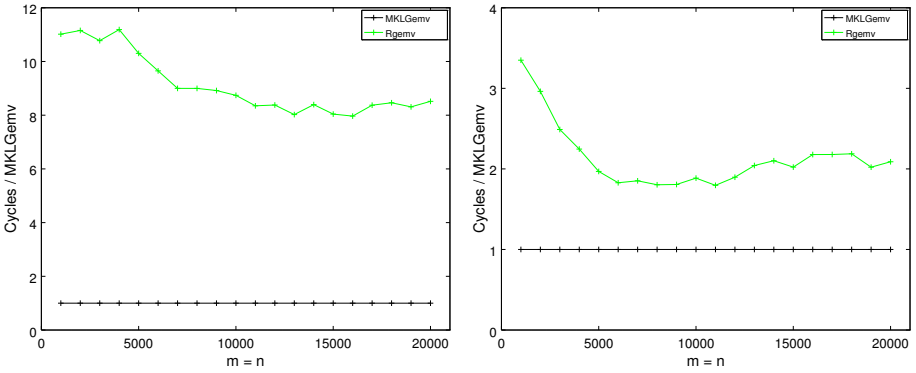
We test the efficiency of the sequential and the shared memory parallel implementation on platform A. Platform B illustrates the many core accelerator use. Finally the scalability of our approach on large supercomputers is exhibited on platform C (Occigen supercomputer). Unfortunately due to a limited access time to this latter we are only able to present results for dot product. Data for dot product are generated as in [12]. The same idea is used to generate condition dependant data for matrix-vector multiplication (multiple condition dependant dot products with a shared vector).

4.2 Implementation and Performance Results

We compare the performance results to the Intel MKL library which is highly optimized but neither accurately rounded nor reproducible.

Sequential Performance Performance tests for the sequential case are done on platform A. Results for *dot*, *asum* and *nrm*2 are presented in [1]. Accurately rounded versions have respectively 5×, 2× and 9× extra-cost.

In Figure 3a, we compare our correctly rounded matrix-vector multiplication to the classical one provided by MKL. Our algorithm is not compared to any reproducible solution. Indeed MKL matrix-vector multiplication does not suffer from reproducibility problem. *Rgemv* computes a correctly rounded matrix-vector multiplication using $iFastSum$ for small matrices and $HybridSum$ for large ones, this latter being slightly more efficient than On both platforms A and B. In the sequential case *Rgemv* costs about 8 times more compared to $MKLGemv$.



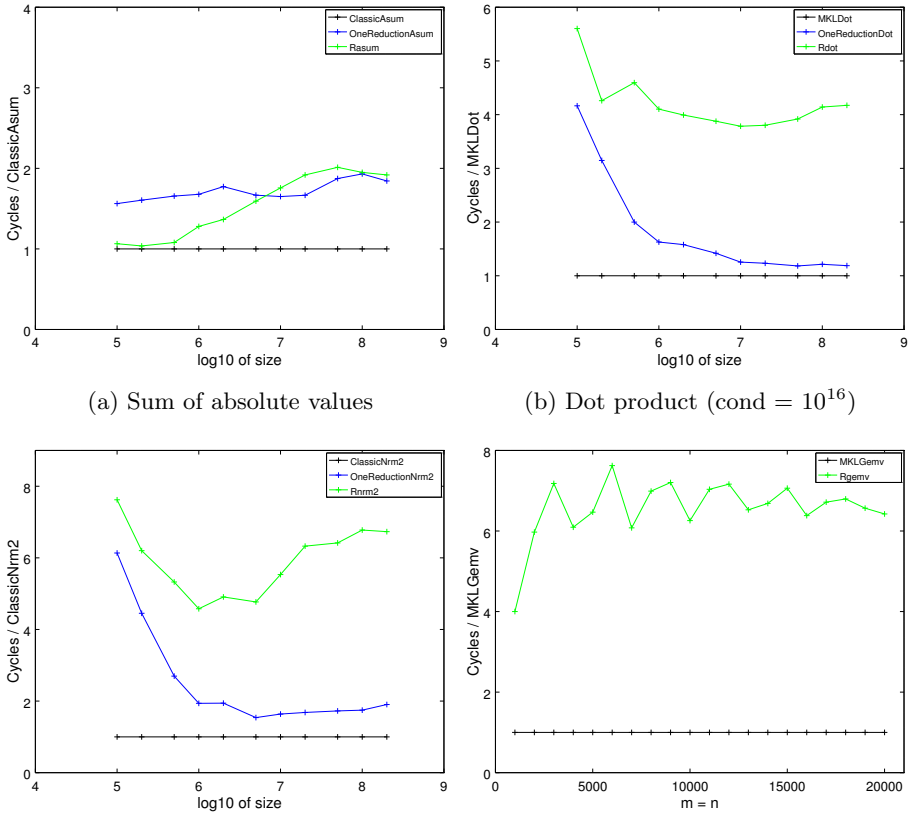
(a) Sequential (b) Parallel

Fig. 3: Extra-cost of correctly rounded Matrix vector multiplication (Cond = 10^8)

Shared Memory Parallel Performance Tests for shared memory parallel systems 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 sequential implementation, results for *dot*, *asum* and *nrm*2 are presented in [1]. The extra-cost of accurately rounded versions is respectively 1×, 1× and 2×.

For the matrix-vector multiplication, correctly rounded algorithm cost about twice compared to classical algorithms. $MKLGemv$ would use blocking algorithms to benefit from a better memory bandwidth use. This approach can reduce memory access operations to about half compared to algorithms that do not use blocking [9]. Since our algorithms do not benefit from blocking, they perform twice more memory access. This justifies the gap in performance between *Rgemv* and $MKLGemv$ even with enough CPU power.

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 intrinsics are used to benefit from extensions of the instruction set available on Xeon Phi. In our scope, the main architectural difference comes from the larger memory bandwidth provided by the Xeon Phi. Therefore, the classical implementations that do not require huge CPU power become less memory bounded. A FMA (Fused Multiply and Add) operation is also available. Therefore $TwoProd$ is replaced by $2MultFMA$ [10] which only requires two FMAs to compute the product and its error, and so improves performance. We also include the recent reproducible and efficient algorithm $OneReduction$ [6]. This latter originally applies to sum, but we derive and test a reproducible versions using it for dot, $asum$ and $nrm2$. These versions are denoted $OneReductionDot$, $OneReductionAsum$ and $OneReductionNrm2$.



(a) Sum of absolute values

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

(c) Euclidean norm

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

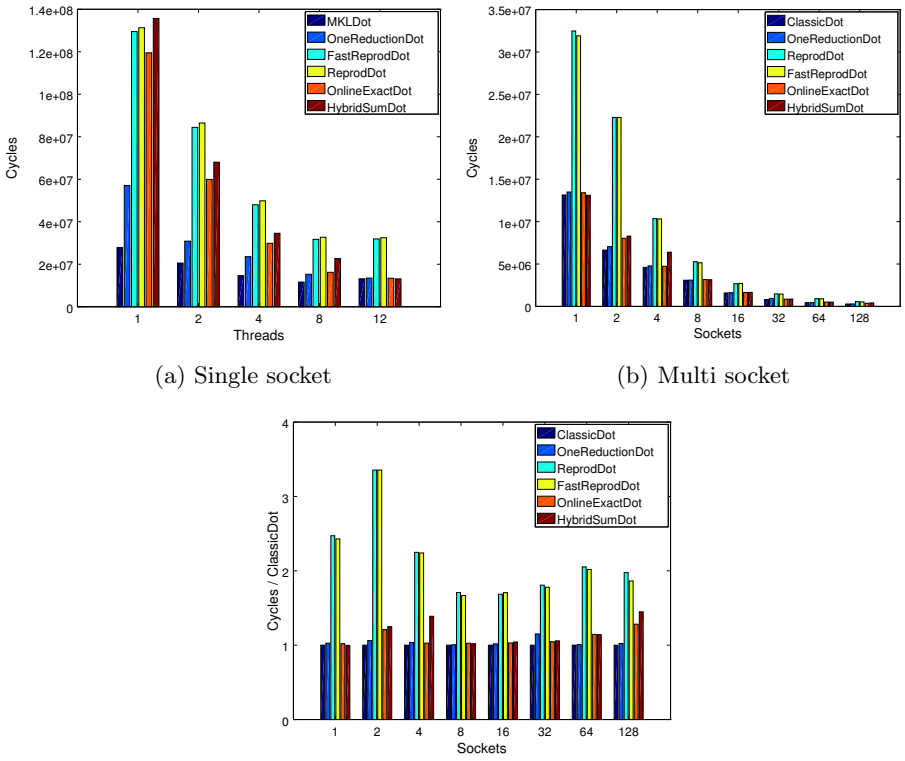
Fig. 4: Extra-cost of Xeon Phi implementation compared to classical algorithms

Fig. 4a shows that for sum of absolute values, both reproducible and correctly rounded solutions cost about twice compared to the classical one. For dot product, euclidean norm and matrix-vector multiplication respectively shown in Figures 4b, 4b and 4d, the gap between the accurately rounded and the classical implementations is larger than for CPU. We note a $4\times$ ratio for dot product, and a $6\times$ times one for both euclidean norm and matrix-vector multiplication. This worse ratio is due to (1) larger memory bandwidth, so classical implementations become less memory bounded, (2) algorithm $HybridSum$ suffer from operations that can not be vectorized, which breaks down performance on an accelerator.

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}. We also test the two algorithms $ReprodDot$ and $FastReprodDot$, that we derived from $ReprodSum$ and $FastReprodSum$ respectively. They illustrate the effect of reduction and memory bandwidth on performance.

Fig. 5a shows scalability on the single socket configuration. It is not a surprise that $MKLDot$ does not scale since it is quickly limited by the memory bandwidth. $HybridSumDot$, $OnlineExactDot$ and $OneReductionDot$ scale very well up to exhibit no extra-cost compared to optimized $MKLDot$. This happens since we have enough CPU power and a limit on the memory bandwidth. On the other hand algorithms $ReprodDot$ and $FastReprodDot$ cost twice compared to $MKLDot$ in the best case. Indeed those algorithms run twice through the vector, and make twice more memory access operations than other algorithms.

Performance for the multi socket configuration is presented in Figures 5b and 5c. In Figure 5b we show the scalability of all tested algorithms, And Figure 5c presents the same data, but normalized compared to $ClassicDot$. Note that X-axis shows the number of sockets, and that all available 12 cores on each socket are used. Algorithms $HybridSumDot$, $OnlineExactDot$ and $OneReductionDot$ are almost as efficient as $ClassicDot$ up to used sockets. $FastReprodDot$ and $ReprodDot$ suffer from communication extra-cost at this level, both algorithms rely on two communications. This degrades dramatically their performance on this kind of large systems where communication cost is critical.



(a) Single socket

(b) Multi socket

(c) Multi socket / ClassicDot

Fig. 5: Extra-cost of distributed memory parallel implementation compared to classical algorithms

5 Conclusion and Future Work

We have presented algorithms that compute reproducible and accurately rounded results for BLAS. Level 1 and level 2 subroutines have been addressed in this paper. Implementations of these algorithms have been tested on three platforms significant of the floating-point computations practice. Our proposed solutions aim at ensuring reproducibility and best precision. It shows pleasant performance on CPU based parallel environments. Over-cost on CPU when all available cores are used is at worst twice. On the other hand, performance on Xeon Phi accelerator is lagging behind. The over-cost is between 4 times and 6 times more compared to classical implementations. Although, our algorithms remain efficient enough to be used for validation or debugging programs, and also for applications that can sacrifice performance to increase the accuracy and the reproducibility of their results.

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., Iakimchuk, R.: Reproducible and Accurate Matrix Multiplication in ExBLAS for High-Performance Computing. In: SCAN'2014. Würzburg, Germany (2014)
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: Proc. 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 (Sep 2014)
6. Demmel, J.W., Nguyen, H.D.: Parallel Reproducible Summation 64(7), 2060–2070 (July 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 (Oct 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 (Aug 2008)
9. Kabir, K., Haidar, A., Tomov, S., Dongarra, J.: High Performance Computing: 30th International Conference, ISC High Performance 2015, Frankfurt, Germany, July 12-16, 2015, Proceedings, chap. On the Design, Development, and Analysis of Optimized Matrix-Vector Multiplication Routines for Coprocessors, pp. 58–73. Springer International Publishing, Cham (2015)
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. Ogita, T., Rump, S.M., Oishi, S.: Accurate sum and dot product. SIAM J. Sci. Comput. 26(6), 1955–1988 (2005)
13. Rosenquist, T.: Run-to-Run Numerical Reproducibility with the Intel Math Kernel Library and Intel Composer XE 2013. Tech. rep., Intel Corporation (2013)
14. Rump, S.M.: Ultimately fast accurate summation. SIAM J. Sci. Comput. 31(5), 3466–3502 (2009)
15. Yamanaka, N., Ogita, T., Rump, S., Oishi, S.: A parallel algorithm for accurate dot product. Parallel Comput. 34(68), 392 – 410 (2008)
16. 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)
17. Zhu, Y.K., Hayes, W.B.: Algorithm 908: Online exact summation of floating-point streams. ACM Trans. Math. Software 37(3), 37:1–37:13 (2010)