Implementation and performance evaluation of an extended precision floating-point arithmetic library for high-accuracy semidefinite programming

Mioara Joldes, Jean-Michel Muller, Valentina Popescu

▶ To cite this version:

Mioara Joldes, Jean-Michel Muller, Valentina Popescu. Implementation and performance evaluation of an extended precision floating-point arithmetic library for high-accuracy semidefinite programming. 2017. hal-01491255v1

HAL Id: hal-01491255
https://hal.science/hal-01491255v1

Preprint submitted on 16 Mar 2017 (v1), last revised 23 Apr 2017 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Implementation and performance evaluation of an extended precision floating-point arithmetic library for high-accuracy semidefinite programming

Mioara Joldes*, Jean-Michel Muller† and Valentina Popescu†
* LAAS-CNRS, 7 Avenue du Colonel Roche, 31077 Toulouse, France
† LIP Laboratory, ENS Lyon, 46 Allée d'Italie, 69364 Lyon Cedex 07, France

Abstract

Semidefinite programming (SDP) is widely used in optimization problems with many applications, however, certain SDP instances are ill-posed and need more precision than the standard double-precision available. Moreover, these problems are large-scale and could benefit from parallelization on specialized architectures such as GPUs. In this article, we implement and evaluate the performance of a floating-point expansion-based arithmetic library (newFPLib) in the context of such numerically highly accurate SDP solvers. We plugged-in the newFPLib with the state-of-the-art SDPA solver for both CPU and GPU-tuned implementations. We compare and contrast both the numerical accuracy and performance of SDPA-GMP, -QD and -DD, which employ other multiple-precision arithmetic libraries against SDPA-newFPLib. We show that our newFPLib is a very good trade-off for accuracy and speed when solving ill-conditioned SDP problems.

Index Terms

floating-point arithmetic, multiple precision library, ill-posed semidefinite programming, GPGPU computing, error-free transform, floating-point expansions

1. Introduction

Nowadays most floating-point (FP) computations use *single*-precision (*binary32*) or *double*-precision (*binary64*) arithmetic. Arithmetic operations on these two formats are very efficiently implemented in currently available processors, either CPUs or GPUs, and are compliant with the IEEE 754-2008 standard for FP arithmetic [1]. Among other requirements, this standard defines five rounding modes (round downwards, upwards, towards zero, to the nearest "ties to even", and to the nearest "ties to away"). An arithmetic operation should return the result as if computed with infinite precision and then the rounding function is applied. Such an operation is said to be correctly rounded. The IEEE 754-2008 standard enforces the correct rounding of all basic arithmetic operations (addition, multiplication, division and square root). This improves the portability and reliability of numerical software.

While these two precision formats are ubiquitous in most software and applications, they are sometimes not sufficient. This happens for instance in some recently studied optimization problems, which come to solving in a very accurate way, numerically sensitive (and sometimes large-scale) semidefinite optimization problems (SDP). Examples include problems from experimental mathematics, like the high-accuracy computation of *kissing numbers*, i.e. the maximal number of non-overlapping unit spheres that simultaneously can touch a central unit sphere [2]; bounds from binary codes; control theory and structural design optimization (e.g., the wing of Airbus A380) [3]; quantum information and physics [4].

This resulted in a recent increased interest in providing both higher-precision (also called *multiple precision*) and high-performance SDP libraries. To accomplish this, multiple challenges occur. Firstly, one has to establish the core mathematical algorithm for numerically solving (say, in "real numbers") the SDP problem: most nowadays solvers employ primal-dual path-following interior-point method (PDIPM) [5]. This algorithm is considered in literature as theoretically mature, is widely accepted and implemented in most state-of-the-art SDP solvers like SDPA [6], CSDP [7], SeDuMi [8], SDPT3 [9]. This algorithm is recalled in Section 2 and we considered the SDPA implementation in this work, without any important modification.

Secondly, the underlying multiple-precision arithmetic operations have to be treated. Several multiple precision arithmetic libraries like GMP and QD were already ported inside SDPA [10]. The resulted implementations are more accurate, yet much more computationally expensive [10]. These libraries and

implementations with SDPA are described in Section 3. Finally, since most problems are large-scale, parallelization is also very important. We treat the case of highly parallel architectures of GPUs, for which most multiple precision libraries are not suitable.

The first contribution of this work is to propose the use of our multiple precision library (newFPLib) with SDPA, which results in a better performance vs. accuracy trade-off. Our underlying algorithms based on floating-point expansions are described in Section 4. The second contribution is a multiple precision GPU compatible general matrix multiplication routine that can be used in SDPA. This routine runs at up to 83% of the theoretical GPU peak-performance and allows for an average speedup of one order of magnitude for SDP instances run in multiple precision with SDPA-newFPLib and GPU support compared to SDPA-newFPLib on CPU only. Implementation details are provided in Section 5. Benchmarks were performed on well-known ill-conditioned examples from [11] and [3]. The results are discussed in Section 6.

2. Semidefinite programming formulation

Semidefinite programing (SDP) is a convex optimization problem, which can be seen as a natural generalization of linear programming to the cone of symmetric matrices with non-negative eigenvalues, i.e. positive semidefinite matrices. While linear programming optimizes a linear functional subject to linear inequality constraints, SDP optimizes a linear functional subject to linear matrix inequalities (LMIs). Many optimization problems in automatic control or signal processing can be formulated using LMIs. Denote by $\mathbb{R}^{n\times n}$ the space of size $n \times n$ real matrices, by $\mathbb{S}^n \subset \mathbb{R}^{n\times n}$ the subspace of real symmetric matrices, equipped with the inner product $\langle A, B\rangle_{\mathbb{S}^n} = \mathrm{tr}(A^T B)$, where $\mathrm{tr}(A)$ denotes the trace of the matrix A. Also, denote by $A \succcurlyeq O$ the fact that A is positive semidefinite (and respectively $A \succ O$ for positive definite). A typical SDP program is expressed in its primal-dual form as follows:

$$(\mathrm{P}) \quad \begin{aligned} p^* = &\sup_{X\in\mathbb{S}^n} \langle C, X\rangle_{\mathbb{S}^n} \\ &\text{s.t. } \langle A_i, X\rangle_{\mathbb{S}^n} = b_i, i = 1, \ldots, m, \\ &\quad X \succcurlyeq O, \end{aligned}$$

$$(\mathrm{D}) \quad \begin{aligned} d^* = &\inf_{y\in\mathbb{R}^m} b^T y \\ &\text{s.t. } Y := \sum_{i=1}^{m} y_i A_i - C \succcurlyeq O, \end{aligned}$$

where $C, A_i \in \mathbb{S}^{n\times n}, i = 1, \ldots m$ and $b \in \mathbb{R}^m$ are given.

It is however difficult in general to obtain an accurate optimum for a SDP problem. On the one hand, strong duality does not always hold, unlike for linear programs: weak duality is always satisfied, i.e. $p^* \leq d^*$, but sometimes, p^* is strictly less than d^*. Simpler instances are those where strong duality holds and this happens when the feasible set contains a positive definite matrix [12, Thm. 1.3]. In practice, the method of choice for SDP solving is based on interior-point algorithm. This relies on the existence of interior feasible solutions for problems (P) and (D). In such cases, these two problems are simultaneously solved in polynomial time in the size of parameters of the input problem using the well-established primal-dual path-following interior-point method (PDIPM) [5]. This method relies on the fact that when an interior feasible solution exits, one has the necessary and sufficient condition for X^*, Y^* and y^* to be an optimal solution:

$$X^*Y^* = 0,\ X^* \succ O,\ Y^* \succ O, \tag{1}$$

$$Y^* = \sum_{i=1}^{m} y_i^* A_i - C, \tag{2}$$

$$\langle A_i, X\rangle_{\mathbb{S}^n} = b_i, i = 1, \ldots, m, \tag{3}$$

The complementary slackness condition (1) is replaced by a perturbed one: $X_\mu Y_\mu = \mu I$. It is known that this perturbed system has a unique solution and that the central path, that is the set $\mathcal{C} = \{(X_\mu, Y_\mu, y_\mu) : \mu > 0\}$ forms a smooth curve converging to (X^*, Y^*, y^*) as $\mu \to 0$. So, the main idea is to numerically trace the central path $\mathcal{C}$. This algorithm, implemented among others by SDPA [6], [10] and its multiple precision versions (SDPA-GMP, SDPA-DD, SDPA-QD) is sketched in Algorithm 1 (adapted from [10]). Step 1 describes the procedure for computing the search direction based on Mehrotra type predictor-corrector [6]. The stopping criteria (Step 4) depends on several quantities: primal and dual feasibility error are defined as the maximum absolute error appearing in (3) and (2), respectively. The duality gap depends on the absolute or relative difference between the objectives of (P) and (D). Numerical results involving these parameters are given in Section 6.

However, problems which do not have an interior feasible point induce numerical instability and may result in inaccurate calculations or non-convergences. Recently, SPECTRA package [13] proposes to solve such problems with exact rational arithmetic, but the instances treated are small and this package does not aim to be a concurrent of general numerical solvers.

On the other hand, even for problems which have interior feasible solutions, numerical inaccuracies may

Algorithm 1 PDIPM Algorithm, adapted from SDPA implementation [10].

Step 0: Choose an initial point $X^0 \succ O$, y^0 and $Y^0 \succ O$. Set $h = 0$ and choose the parameter $\gamma \in (0, 1)$.
Step 1: Compute a search direction:
Evaluate the Shur Complement Matrix $B \in \mathbb{S}^n$ by the formula $B_{ij} = \langle (Y^h)^{-1} A_i X^h, A_j \rangle$.
Solve the linear equation $Bdy = r$. Using the solution dy, compute dX, dY and obtain the search direction (dy, dX, dY).
Step 2: Compute max step length α to keep the positive semidefiniteness: $\alpha = \max\{\alpha \in [0,1] : Y^h + \alpha dY \succ O, X^h + \alpha dX \succ O\}$.
Step 3: Update the current point: $(y^{h+1}, X^{h+1}, Y^{h+1}) = (y^h, X^h, Y^h) + \gamma\alpha(dy, dX, dY)$.
Step 4: If $(y^{h+1}, X^{h+1}, Y^{h+1})$ satisfies the stopping criteria, output it as a solution. Otherwise, set $h = h + 1$ and return to Step 1.

appear when solving with finite precision due to large condition numbers (higher than 10^{16}, for example) which appear when solving linear equations. This happens, as explained in [10], when approaching optimal solutions: suppose there exist $X^* \succ O$, $Y^* \succ O$ and y^* which satisfy all the constraints in (P) and (D); or better said, when $\mu \to 0$ on the central path. Then, $X^* Y^* = 0$. From this it follows that $\text{rank}(X^*) + \text{rank}(Y^*) \leq n$, which implies that these matrices are usually singular in practice.

In this second case, having an efficient underlying multiple-precision arithmetic is crucial to detect (at least numerically) whether the convergence issue came simply from numerical errors due to lack of precision.

3. Multiple precision arithmetic libraries

Most SDP solvers are based on *single* or *double* floating-point arithmetic, which is very efficient as it is implemented in hardware. An increase in precision, for example to *quad*-precision (*binary128*) or more, comes with a significant performance drawback since this is currently done by software emulation. Arbitrary precision, i.e., the user's ability to choose the precision for each calculation, is now available in most computer algebra systems such as Maple. SPECTRA [13] is implemented with Maple's rational arithmetic. SDPA extended precision versions are implemented using either GMP [14] or QD [15] libraries. GMP offers arbitrary precision by representing floating-point numbers in a so-called *multiple-digit* format. This is sufficiently general for being able to manipulate numbers with tens of thousands –or even more– bits, but it is a quite heavy alternative for precisions of few hundreds bits. QD choses to better optimize precisions of 2 or 4 doubles using a different *multiple-component* format. It is known that most operations implemented in this library do not come with proven error bounds and correct or directed rounding is not supported. It is thus usually impossible to assess the final accuracy of these operations. However, the performance results of QD are very good on tested problems (e.g. on SDP instances [6]). Moreover, only QD is ported on parallel GPU architectures, since floating-point algorithms in the *multiple-digit* format are very complex, and they employ non-trivial memory management. QD code with SDPA shows in practice that it is possible to compute very accurate values even when rounding occurs at the intermediate operation's level.

We generalized or modified this kind of algorithms in order to prove their correctness and keep good performances.

4. Our floating-point expansions based arithmetic library

For extending the precision, we represent numbers as unevaluated sums of floating-point numbers. When the sum consists of two terms, one has a *double-double* (DD) number, *triple-double* (TD) for three terms, *quad-double* for four, and *floating-point expansion* when made up with an arbitrary number of terms. Arithmetic operations with such numbers are based on so-called *error-free transforms* algorithms, which allow for computing the exact result of a FP addition or multiplication by computing also the rounding error. For example, the sum of two FP numbers can be represented *exactly* (in the sense of dyadic numbers) as a FP number which is the correct rounding of the sum, plus a second FP number corresponding to the rounding error. Under certain assumptions, this decomposition can be efficiently computed by a simple sequence of standard precision FP operations. This is the key feature of this representation: only very optimized hardware operations with standard FP numbers are employed.

Algorithms 2 and 3 are state-of-the-art for error-free transform of the sum and respectively the product of two FP numbers.

Algorithm 2 2Sum(a, b)

$s \leftarrow \text{RN}(a+b)$
$a' \leftarrow \text{RN}(s-b)$
$b' \leftarrow \text{RN}(s-a')$
$\delta_a \leftarrow \text{RN}(a-a')$
$\delta_b \leftarrow \text{RN}(b-b')$
$t \leftarrow \text{RN}(\delta_a+\delta_b)$
return s, t

Algorithm 3 2ProdFMA($a,\ b$)

$\pi \leftarrow \mathrm{RN}(ab)$
$e \leftarrow \mathrm{RN}(ab - \pi)$
return $\pi,\ e$

Assuming that a and b are two FP numbers and that the rounding function, denoted RN, is round-to-nearest "ties to even", Knuth's *2Sum* algorithm [16] computes the decomposition of $a+b$ using only 6 FP operations (see Algorithm 2). When the operands are ordered, Dekker's *Fast2Sum* algorithm [17] is used, which takes only 3 FP operations. Similarly, if a fused multiply-add [1] (FMA) operator is available, *2ProdFMA* [16] returns π, the correct rounded product, and e, the rounding error (namely $ab-\pi$), in 2 FP operations (see Algorithm 3). When chaining together such error-free transforms (resulting in the so called *distillation* algorithms [18]) one can design efficient basic operations with FP expansions. Such algorithms were recently improved and proved to be correct in [19].

Based on these, our newFPLib is a multiple-precision arithmetic library which targets both CPU applications and applications deployed on NVIDIA GPU platforms (compute capability 2.0 or greater). Both a CPU version (in C++ language) and a GPU version (written in CUDA C programming language [20]) are freely available. This library supports both the *binary64* and the *binary32* formats as basic bricks for the multiple component representation. This allows for extended precisions on the order of a few hundreds of bits, with constraints given by the exponent range of the underlying FP format used: the maximum expansion size is 39 for *double*-precision, and 12 for *single*-precision. Currently, all basic multiple-precision arithmetic operations $(+,-,*,/,\surd)$ are supported. In the design phase we chose to make the implementation very flexible: the precision (FP expansion size) is given as a template parameter and we provide overloaded operators.

Another flexibility is on the trade-off between proven output accuracy in worst case versus highly efficient average case. This results in levels of algorithms (two different classes): *(i)* "certified" algorithms with rigorously proven error bounds and correctness proofs; *(ii)* "quick-and-dirty" algorithms which are very fast, but do not consider accuracy issues for corner cases (i.e. cancellation prone computations). The later one also comes with a code generation module, that allows for the user to code generate the algorithms function of the needed expansion size. This module provides increased performance by custom unrolling some complex loops (which are usually not optimized by GCC or NVCC compilers).

1. FMA operator evaluates an expression of the form $xy+t$ with only one final rounding.

For double-double, the algorithms initially proposed in QD library, were recently rediscussed, improved and proved in [21]. For example, addition of two DD numbers takes 20 FP operations and multiplication 9, when an FMA is available (see Algorithms 6 and 11 in [21]). For higher precisions, we use a generalization of these algorithms from [19], with the technicality that the renormalization of FP expansions is composed of the first two levels of chained 2Sum of Algorithm 6 in [19]. Their generality comes with a higher operation count. The number of FP operations for addition and multiplication of two operands represented on n-double, for a result on n-double is given in Table 1. For addition, Algorithm 4 [19], is on the order of $3n^2 + 10n - 4$ double FP additions, while multiplication, Algorithm 5 [19], takes $2n^3+2n^2+6n-4$ double FP operations.

n	Addition	Multiplication
2D	20	9
3D	53	86
4D	84	180
5D	121	326
6D	164	536
8D	268	1196

Table 1: Number of operations for addition and multiplication (FMA compatible) with n-doubles formats.

5. Implementation details of SDPA-newFPLib

The SDPA-newFPLib package is built from the SDPA-QD/DD package, where QD/DD library is replaced with newFPLib at the compilation step of SDPA. This can be done efficiently since both SDPA and newFPLib are written in C/C++. Starting with version 6.0, SDPA incorporated LAPACK [22] for dense matrix computations, but also exploits sparsity of data matrices and solves large scale SDPs [23]. More recently, MPACK [24] was developed and integrated with SDPA. This is a multiple precision linear algebra package which is based on BLAS and LAPACK [22]. For this package, the major change is, as in our case, the underlying arithmetic format, such that users can easily switch from a double precision BLAS/LAPACK code to a multiple precision one, in order to obtain better accuracy. MPACK supports various multiple precision arithmetic libraries like GMP, MPFR, and QD. We integrated newFPLib with MPACK.

Moreover, MPACK also provided a GPU tuned implementation in double-double of the Rgemm routine:

this is the multiple precision *Real* version of Dgemm, the general double matrix multiplication [25]. This routine is central for other linear algebra operations such as solving linear equations, singular value decomposition or eigenvalue problems. For this, MPACK authors re-implemented parts of DD library for CUDA-compliant code. In our implementation, we used instead our GPU version of newFPLib.

This routine's implementation is reported in [25] with best practical performance and was intensively tuned for GPU-based parallelism: classical blocking algorithm is employed, and for each element of a block a thread is created; a specific number of threads is allocated per block also. For example, for the NVIDIA(R) Tesla(TM) C2050 GPU, best performance of 16.4GFlops (Giga FP operations per second) is obtained in [25] for $A \times B$, the product of two matrices A and B with block size of: 16×16 for A and 16×64 for B; 256 threads are allocated per block. Shared memory is used for each block. Also, reading is done from texture memory.

In our implementation, we use a similar algorithm, except that reading is done from global memory instead of texture memory, since a texture memory element (texel) size is limited to *int4* i.e., 128 bits and our implementation is generic for n-double. Matrix block sizes and thread block sizes are similar, since our bench GPU is similar. Specifically, our GPU NVIDIA(R) Tesla(TM) C2075 is part of the same Fermi architecture, with 448 cores, 1.15 GHz, 32KB of register, 64KB shared memory/L1 cache set by default to 48KB for shared memory and 16KB for L1 cache. The difference is that our GPU has 6GB of global memory, compared to 3GB of C2050. However, this has little importance for the performance results on kernel execution once the global memory has been loaded. With our newFPLib in double-double, the peak kernel performance obtained is 14.8GFlops as shown in Figure 1. The theoretical peak performance can be obtained as follows: first, consider that in Rgemm operations consist of mainly multiply-add type, so the theoretical peak for multiply-add is of 1.15[GHz]$\times 14$[SM]$\times 32$[CUDA cores]$\times (2$[Flop]$/2$[cycle]$) = 515$[Gflops]. Now, since addition and multiplication in our newFPLib take $(20+9)$ Flops for double-double, one obtains theoretically: $515.2/29 = 17.8$[GFlops]. For double-double, our implementation is slower by $\sim 10\%$ than the implementation in [25]. This can be explain by the generality of our code. Although we tested our implementation also using texture memory, we observed no speedup. On the other hand, in our case, higher precision RGEMM is straightforward. Performance results for n-double RGEMM are shown in Figure 2: one observes that the increase in number of additions and multiplications reported in Table 1 fits the decrease of performance when precision is increased.

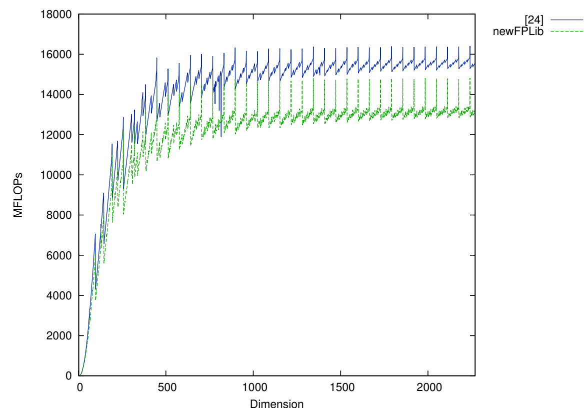

Figure 1: Performance of RGEMM with newFPLib vs [25] in double-double on GPU. Maximum performance was 14.8GFlops for newFPLib and 16.4GFlops for [25].

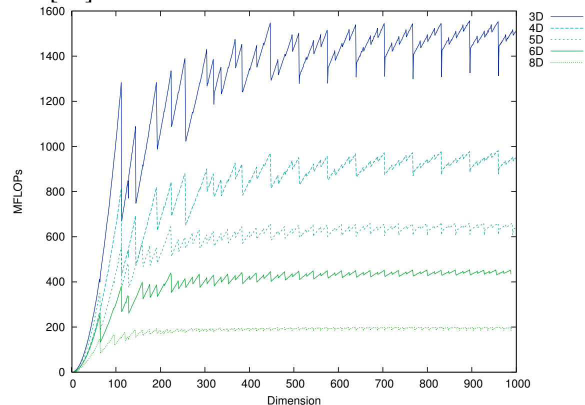

Figure 2: Performance of RGEMM with newFPLib for n-double on GPU. Maximum performance was 1.6GFlops for TD, 976MFlops for QD, 660MFlops for 5D, 453MFlops for 6D, 200MFlops for 8D.

Note that cuBLAS, the NVIDIA GPU linear algebra package does not support precisions higher than double and it is not open source, so we consider it difficult to extend it in the context of multiple precision linear algebra for GPUs. For Fermi architecture GPUs like C2050 or C2075, the peak performance of DGEMM is reported in [26] to be 302GFlops with cuBlas and 362GFlops with further optimizations which is 58% and 70% of the theoretical peak performance, so the RGEMM implementation we have is quite efficient with: 83% of theoretical peak performance for DD, respectively 43% for TD, 50% for 4D, 57% for 5D, 61% for 6D, 57% for 8D.

Problem	SDPA-DD	SDPA-QD	SDPA-newFPLib		
			(DD)	(TD)	(QD)
gpp124-1	optimal: -7.3430762652465377				
relative gap	$7.72e-04$	$6.75e-13$	$7.72e-04$	$8.33e-12$	$5.42e-18$
p.feas.error	$5.42e-20$	$4.37e-45$	$5.42e-20$	$1.57e-30$	$1.14e-41$
d.feas.error	$4.40e-14$	$1.54e-30$	$3.99e-14$	$2.58e-16$	$3.43e-22$
iteration	24	40	24	38	49
time (s)	3.27	66.37	2.63	35.56	83.92
gpp250-1	optimal: $-1.5444916882934067e+01$				
relative gap	$5.29e-04$	$4.32e-13$	$5.19e-04$	$1.03e-12$	$1.89e-17$
p.feas.error	$3.89e-20$	$2.27e-45$	$1.35e-20$	$1.38e-30$	$5.73e-42$
d.feas.error	$9.78e-14$	$1.97e-30$	$1.12e-13$	$3.04e-17$	$3.25e-22$
iteration	25	41	25	44	52
time (s)	26.68	538.39	21.57	321.35	698.34
gpp500-1	optimal: $-2.5320543879075787e+01$				
relative gap	$1.008e-03$	$4.72e-13$	$3.72e-04$	$8.89e-12$	$9.30e-18$
p.feas.error	$1.01e-20$	$1.92e-45$	$2.71e-20$	$8.81e-31$	$1.43e-42$
d.feas.error	$5.29e-14$	$2.95e-30$	$3.51e-13$	$1.49e-16$	$5.13e-22$
iteration	25	42	26	41	51
time (s)	207.63	4340.95	172.95	2396.43	5369.79
qap10	optimal: $-1.0926074684462389e+03$				
relative gap	$3.84e-05$	$3.18e-14$	$6.94e-05$	$1.08e-09$	$7.22e-14$
p.feas.error	$2.54e-21$	$7.32e-47$	$6.56e-21$	$6.62e-35$	$2.18e-47$
d.feas.error	$4.91e-14$	$8.78e-30$	$1.83e-13$	$2.98e-21$	$3.45e-29$
iteration	20	36	20	27	35
time (s)	27.75	647.96	21.01	262.03	629.81
hinf3	optimal: $5.6940778009669388e+01$				
relative gap	$1.35e-08$	$7.93e-3$	$3.25e-05$	$3.98e-26$	$1.98e-31$
p.feas.error	$2.75e-24$	$1.18e-54$	$1.69e-21$	$3.32e-39$	$3.18e-55$
d.feas.error	$3.82e-14$	$6.82e-35$	$3.41e-13$	$2.83e-29$	$4.08e-42$
iteration	30	90	24	48	48
time (s)	0.00	0.18	0.00	0.05	0.09

Table 2: The optimal value, relative gaps, primal/dual feasible errors, iterations and time for solving some problems from SDPLIB by SDPA-QD, -DD, -newFPLib.

6. Numerical results

In order to asses the performance of SDPA-newFPLib solver we look at the results obtained for some standard SDP problems both on CPU and GPU. In particular, on GPU we use the Rgemm routine, with our newFPLib implementation explained above.

All CPU tests were performed on an Intel(R) Xeon(R) CPU E3-1270 v3 @ 3.50GHz processor, with Haswell micro-architecture which supports hardware implemented FMA instructions. For the GPU tests, we used a NVIDIA(R) Tesla(TM) C2075 processor board.

Table 2 shows the results and performance obtained for five well-known problems from the SDPLIB package [11]. We compare the QD library implementation, both double- and quad-double precisions, with the double-, triple-, and quad-double offered by newFPLib, on CPU. One can observe that our double-double implementation outperforms the one of QD's by far, while our quad-double precision performs the same. The triple-double proves that it can be a good alternative for problems for which double-double does not suffice, but for which quad-double is too expensive.

While QD library offers only two extended precisions, GMP offers arbitrary precision. In newFPLib we have several extended formats of n-double, this is why we chose to show the comparison between these two in a different table, Table 3. For this we treated three problems from the SDPLIB with corresponding precisions of $106, 159, 212, 265, 318$ and 424 bits. It is easily seen that for small precisions, up to quad-double, our library performs much better or the same on CPU. Performance decreases gradually on CPU when precision increases due to the overhead caused by error propagation and handling inside the FP expansions. However, since newFPLib is generic and ported on GPU, this decrease in performance can be overcome when launching SDPA-newFPLib with GPU support on heavier instances as seen in Table 3. In Figure 3 we show the speedup obtain for newFPLib with GPU support, when varying precision, for several problems from the SDPLIB.

For testing the accuracy of our library, we considered several examples from Sotirov's collection [3], which are badly conditioned numerically, and cannot be tackled with double precision only. The first one comes from a classical problem in coding theory: finding the largest set of binary words with n letters, such that the Hamming distance between two words

is at least some given value d. This is reformulated as a maximum stable set problem, which is solved with SDP, according to the seminal work of Schrijver [27], followed by Laurent [28].

Problem	SDPA-newFPLib CPU	SDPA-newFPLib GPU	SDPA-GMP CPU
`gpp124-1`	optimal: -7.3430762652465377		
precision	DD	DD	$2*53$ bits
iteration	24	24	38
time (s)	2.63	1.19	56.31
precision	TD	TD	$3*53$ bits
iteration	38	39	48
time (s)	35.56	15.1	78.30
precision	QD	QD	$4*53$ bits
iteration	49	57	59
time (s)	83.92	27.9	103.32
precision	5 D	5 D	$5*53$ bits
iteration	62	62	77
time (s)	178.23	46.5	149.76
precision	6 D	6 D	$6*53$ bits
iteration	77	77	77
time (s)	330.08	101	149.76
precision	8 D	8 D	$8*53$ bits
iteration	77	77	77
time (s)	666.60	217	186.32
`gpp250-1`	optimal: $-1.5444916882934067e+01$		
precision	DD	DD	$2*53$ bits
iteration	25	25	39
time (s)	21.57	7.65	455.36
precision	TD	TD	$3*53$ bits
iteration	44	39	46
time (s)	321.35	50.6	591.12
precision	QD	QD	$4*53$ bits
iteration	52	52	64
time (s)	698.34	100.3	880.90
precision	5 D	5 D	$5*53$ bits
iteration	62	62	73
time (s)	1404.48	180.4	1117.60
precision	6 D	6 D	$6*53$ bits
iteration	73	73	73
time (s)	2465.30	350.6	1116.72
precision	8 D	8 D	$8*53$ bits
iteration	73	73	73
time (s)	4987.770	1047.7	1392.71
`hinf3`	optimal: $5.6940778009669388e+01$		
precision	DD	DD	$2*53$ bits
iteration	24	24	39
time (s)	0.00	0.51	0.06
precision	TD	TD	$3*53$ bits
iteration	48	48	47
time (s)	0.05	0.82	0.07
precision	QD	QD	$4*53$ bits
iteration	48	48	47
time (s)	0.09	2.01	0.08
precision	5 D	5 D	$5*53$ bits
iteration	48	48	47
time (s)	0.16	3.52	0.09
precision	6 D	6 D	$6*53$ bits
iteration	48	48	47
time (s)	0.25	5.13	0.09
precision	8 D	8 D	$8*53$ bits
iteration	48	48	47
time (s)	0.53		0.10

Table 3: The optimal value, iterations and time for solving some problems from SDPLIB by SDPA-GMP, -newFPLib

In Table 4 we show the performance obtained on the CPU for the Schrijver and Laurent instances from [3]. The comparison is done between SDPA-DD, SDPA-GMP (run with 106 bits of precision) and SDPA-newFPLib-DD. The results obtained were very close and performance-wise our library performs better. Some instances do not converge when DD precision is used. We also include the results obtained with the SDPA-newFPLib with triple-double (TD) precision, which has a better performance comparing to SDPA-GMP with 106 bits of precision.

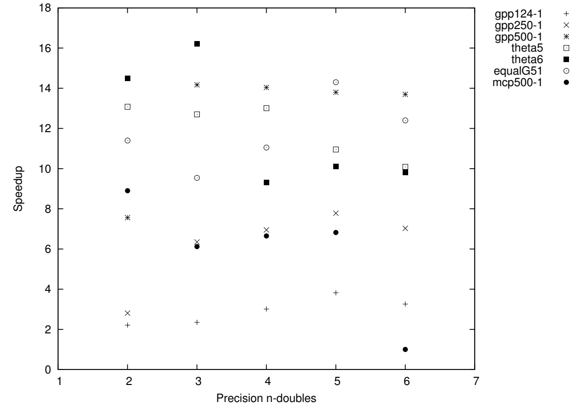

Figure 3: Speedup of SDPA-newFPLib for n-double with GPU vs CPU. Maximum speedup was 16.2.

7. Conclusion and future works

We implemented and evaluated the performance of a floating-point expansions-based arithmetic library (newFPLib) in the context of highly accurate computations needed by SDP solvers for ill-conditioned and performance demanding problems. The underlying arithmetic operations of state-of-the-art multiple-precision solver SDPA were replaced with our algorithms. This was possible since both SDPA and newFPLib are written in open-source and freely available C/C++. We compare and contrast both the numerical accuracy and performance of SDPA-GMP, -QD and -DD, which employ other multiple-precision arithmetic libraries, against SDPA-newFPLib. We consider both CPU and GPU implementations. We show that our newFPLib is a very good trade-off for accuracy and speed when solving ill-conditioned SDP problems.

One current limitation is that compared to the tuned GPU implementation of double-double RGEMM of [25], our implementation is 10% slower. However, the improvement is that our RGEMM is generic for n-double precision. This allows us to solve very ill-conditioned problems with SDPA and GPU support

Problem	SDPA-DD	SDPA-newFPLib-DD	SDPA-newFPLib-TD	SDPA-GMP
Laurent_A(19,6)	optimal: $-2.4414745686616550e-03$			
iteration	92	94	71	73
time (s)	4.3	3.1	18.65	29.16
Laurent_A(26,10)	optimal: $-1.3215201241629400e-05$			
iteration	80	80	123	125
time (s)	12.8	8.68	109.54	173.42
Laurent_A(28,8)	optimal: $-1.1977477306795422e-04$			
iteration	93	100	76	113
time (s)	47.8	36.85	219.46	541.19
Laurent_A(48,15)	optimal: $-2.229e-09$			
iteration	134	134	165	145
time (s)	2204.61	1569.48	14691.92	21695.08
Laurent_A(50,15)	optimal: $-1.9712e-09$			
iteration	142	142	191	154
time (s)	3463.2	2421.86	25773.96	35173.79
Laurent_A(50,23)	optimal: $-2.5985e-13$			
iteration	124	124	155	140
time (s)	342.73	221.32	2333.74	3426.17
Schriver_A(19,6)	optimal: $-1.2790362700180910e+03$			
iteration	40	40	66	95
time (s)	1.59	1.14	14.65	32.21
Schriver_A(26,10)	optimal: $-8.8585714285713880e+02$			
iteration	54	54	127	108
time (s)	7.75	5.2	100.73	134.48
Schriver_A(28,8)	optimal: $-3.2150795825792913e+04$			
iteration	45	45	69	97
time (s)	21.05	15.06	182.25	422.78
Schriver_A(37,15)	optimal: $-1.40069999999999886e+03$			
iteration	58	58	132	116
time (s)	54.86	36.35	683.07	988.21
Schriver_A(40,15)*	optimal: $-1.9e+04$			
iteration	23	23	23	23
time (s)	53.99	35.99	285.3	471.870
Schriver_A(48,15)*	optimal: $-2.56e+06$			
iteration	27	27	27	27
time (s)	432.13	307.88	2260.24	3862.29
Schriver_A(50,15)**	optimal: $-7.6e+06$			
iteration	29	29	29	29
time (s)	694.07	471.57	3695.95	677.830
Schriver_A(50,23)**	optimal: $-5.2e+03$			
iteration	29	29	29	29
time (s)	76.55	47.84	413.31	6370.97

Table 4: The optimal value, iterations and time for solving some ill-posed problems for binary codes by SDPA-DD, -newFPLib-DD, -newFPLib-TD, and -GMP-DD. $*$ problems that converge only when using quad-double precision. $**$ problems that do not converge. The digits written with blue were obtained only when triple-double precision was employed.

in a more efficient way: before, these problems could be tackled only with high precision GMP support on CPU only. Our new results show an order of magnitude ($\simeq 10$ times) average speedup when newFPLib is used with GPU support. As future work, this should allow us to solve more complicated and large scale very ill-conditioned SDP problems appearing in experimental mathematics, like for instance, the "kissing numbers" problem [2].

References

[1] IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008, Aug. 2008, available at http://ieeexplore.ieee.org/servlet/opac?punumber=4610933.

[2] H. D. Mittelmann and F. Vallentin, "High-accuracy semidefinite programming bounds for kissing numbers," *Experimental Mathematics*, vol. 19, no. 2, pp. 175–179, 2010.

[3] E. de Klerk and R. Sotirov, "A new library of structured semidefinite programming instances," *Optimization Methods and Software*, vol. 24, no. 6, pp. 959–971, 2009. [Online]. Available: http://dx.doi.org/10.1080/10556780902896608

[4] D. Simmons-Duffin, "A Semidefinite Program Solver for the Conformal Bootstrap," *JHEP*, vol. 06, p. 174, 2015.

[5] R. D. Monteiro, "Primal–dual path-following algorithms for semidefinite programming," *SIAM Journal on Optimization*, vol. 7, no. 3, pp. 663–678, 1997.

[6] M. Yamashita, K. Fujisawa, M. Fukuda, K. Kobayashi, K. Nakata, and M. Nakata, "Latest developments in the SDPA family for solving large-scale SDPs," in *Handbook on semidefinite, conic and polynomial optimization*. Springer, 2012, pp. 687–713.

[7] B. Borchers, "CSDP, a C library for semidefinite programming," *Optimization methods and Software*, vol. 11, no. 1-4, pp. 613–623, 1999.

[8] J. F. Sturm, "Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones," *Optimization methods and software*, vol. 11, no. 1-4, pp. 625–653, 1999.

[9] K. Toh, M. Todd, and R. Tutuncu, "SDPT3 — a Matlab software package for semidefinite programming," *Optimization methods and software*, vol. 11, no. 1-4, pp. 545–581, 1999.

[10] M. Nakata, "A numerical evaluation of highly accurate multiple-precision arithmetic version of semidefinite programming solver: SDPA-GMP,-QD and -DD." in *2010 IEEE International Symposium on Computer-Aided Control System Design*. IEEE, 2010, pp. 29–34.

[11] B. Borchers, "SDPLIB 1.2, a library of semidefinite programming test problems," *Optimization Methods and Software*, vol. 11, no. 1-4, pp. 683–690, 1999. [Online]. Available: http://dx.doi.org/10.1080/10556789908805769

[12] M. Laurent, "Sums of squares, moment matrices and optimization over polynomials," in *Emerging applications of algebraic geometry*. Springer, 2009, pp. 157–270.

[13] D. Henrion, S. Naldi, and M. Safey El Din, "SPECTRA - a Maple library for solving linear matrix inequalities in exact arithmetic." arXiv:1611.01947, 2016.

[14] T. Granlund, "GMP, the GNU multiple precision arithmetic library, version 4.1.2," December 2002, http://www.swox.com/gmp/.

[15] 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 (ARITH-16)*, N. Burgess and L. Ciminiera, Eds., Vail, CO, Jun. 2001, pp. 155–162.

[16] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010, ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-0-8176-4704-9.

[17] T. J. Dekker, "A floating-point technique for extending the available precision," *Numerische Mathematik*, vol. 18, no. 3, pp. 224–242, 1971.

[18] S. M. Rump, T. Ogita, and S. Oishi, "Accurate floating-point summation part I: Faithful rounding," *SIAM Journal on Scientific Computing*, vol. 31, no. 1, pp. 189–224, 2008. [Online]. Available: http://link.aip.org/link/?SCE/31/189/1

[19] M. Joldeş, O. Marty, J.-M. Muller, and V. Popescu, "Arithmetic algorithms for extended precision using floating-point expansions," *IEEE Transactions on Computers*, vol. 65, no. 4, pp. 1197–1210, 2016.

[20] NVIDIA, *NVIDIA CUDA Programming Guide 5.5*, 2013.

[21] M. Joldes, V. Popescu, and J.-M. Muller, "Tight and rigourous error bounds for basic building blocks of double-word arithmetic," Jul. 2016, working paper or preprint. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01351529

[22] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999.

[23] M. Yamashita, K. Fujisawa, and M. Kojima, "Implementation and evaluation of sdpa 6.0 (semidefinite programming algorithm 6.0)," *Optimization Methods and Software*, vol. 18, no. 4, pp. 491–505, 2003.

[24] K. Nakata, "The MPACK (MBLAS/MLAPACK) a multiple precision arithmetic version of BLAS and LAPACK," http://mplapack.sourceforge.net/, 2008–2012.

[25] M. Nakata, Y. Takao, S. Noda, and R. Himeno, "A fast implementation of matrix-matrix product in double-double precision on NVIDIA C2050 and application to semidefinite programming," in *2012 Third International Conference on Networking and Computing*, Dec 2012, pp. 68–75.

[26] G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao, and N. Sun, "Fast implementation of DGEMM on Fermi GPU," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 35.

[27] A. Schrijver, "New code upper bounds from the Terwilliger algebra and semidefinite programming," *IEEE Transactions on Information Theory*, vol. 51, no. 8, pp. 2859–2866, 2005.

[28] M. Laurent, "Strengthened semidefinite programming bounds for codes," *Mathematical programming*, vol. 109, no. 2-3, pp. 239–261, 2007.