CAMPARY: Cuda Multiple Precision Arithmetic Library and Applications

Mioara Joldes, Jean-Michel Muller, Valentina Popescu, Warwick Tucker

CAMPARY: Cuda Multiple Precision Arithmetic Library and Applications

Mioara Joldes[1], Jean-Michel Muller[2], Valentina Popescu[2], and Warwick Tucker[3]

[1] LAAS-CNRS, 7 Avenue du Colonel Roche, 31077 Toulouse, France
joldes@laas.fr,
[2] LIP Laboratory, ENS Lyon, 46 Allée d'Italie, 69364 Lyon Cedex 07, France
jean-michel.muller@ens-lyon.fr,
valentina.popescu@ens-lyon.fr,
[3] Department of Mathematics, Uppsala University, Box 480, 75106 Uppsala, Sweden
warwick@math.uu.se

Abstract. Many scientific computing applications demand massive numerical computations on parallel architectures such as Graphics Processing Units (GPUs). Usually, either floating-point single or double precision arithmetic is used. Higher precision is generally not available in hardware, and software extended precision libraries are much slower and rarely supported on GPUs. We develop CAMPARY: a multiple-precision arithmetic library, using the CUDA programming language for the NVidia GPU platform. In our approach, the precision is extended by representing real numbers as the unevaluated sum of several standard machine precision floating-point numbers. We make use of error-free transforms algorithms, which are based only on native precision operations, but keep track of all rounding errors that occur when performing a sequence of additions and multiplications. This offers the simplicity of using hardware highly optimized floating-point operations, while also allowing for rigorously proven rounding error bounds. This also allows for easy implementation of an interval arithmetic. Currently, all basic multiple-precision arithmetic operations are supported. Our target applications are in chaotic dynamical systems or automatic control.

Keywords: floating-point arithmetic, multiple precision library, GPGPU computing, error-free transform, floating-point expansions, dynamical systems, Hénon map, semi-definite programming

1 Introduction

CAMPARY is a multiple-precision arithmetic library which targets mainly 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 [1]) are freely available at `http://homepages.laas.fr/mmjoldes/campary/`. Our library provides extended precisions on the order of a few hundreds of bits. Currently, all basic multiple-precision arithmetic

operations ($+, -, *, /, \sqrt{}$) are supported. Our implementation is very flexible: we provide templated precision sizes and overloaded operators. The library contains two levels of algorithms: *(i)* certified algorithms with rigorous error bounds and output constraints; *(ii)* "quick-and-dirty" algorithms that perform well for the average case, but do not consider the corner cases (i.e. cancellation prone computations). The later one also comes with a code generation module that allows for optimal performance.

The library was initially developed and tuned for long time iteration of chaotic dynamical systems in extended precision. At present, we aim at handling applications which require both extended precision and high performance computing.

2 Context and related software

Currently most floating-point (FP) calculations are done using single-precision (also called binary32) or double-precision (also called binary64) arithmetic. The majority of today's available processors (including GPUs) offer very fast implementations of FP arithmetic in these two formats, and comply with the IEEE 754-2008 standard for FP arithmetic [2]. This standard defines five *rounding functions* (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 using *infinite precision* and then applying the rounding function. 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 requirement improves the portability of numerical software and also makes it possible and relatively easy to build an *interval arithmetic* (i.e., we get sure lower and upper bounds on the exact result).

However, several high-performance computing (HPC) problems require higher precision (also called *multiple precision*), up to a few hundred bits. For instance, in the field of chaotic dynamical systems, such problems appear in both mathematical questions (e.g., the study of strange attractors such as the Hénon attractor [3], in bifurcation analysis and stability of periodic orbits) and in applications to celestial mechanics (e.g., long-term stability of the solar system [4]). Multiple precision is also used in computational geometry (several techniques we use were initially developed for this domain) [6]. An example in *experimental mathematics* is 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 [9]. That approach is based on very accurate solving of numerically sensitive semi-definite optimization problems (SDP). A recent increased interest in high precision SDP libraries comes also from ill-conditioned problems in quantum chemistry or control theory [10].

As of today, higher precision, such as *quad-precision* (binary128) or more has not yet been implemented in hardware on widely distributed processors, and the most common solution is 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 Mathematica, Maple or Sage. Furthermore, GNU MPFR [7] is an open source library written in C, that provides, besides arbitrary precision, correct rounding for each atomic basic operation. However, the versatility of such multiple precision libraries, which are able to manipulate numbers with tens of thousands –or even more– of bits, can sometimes be a quite heavy alternative to use, when a precision up to a few hundred bits is sufficient and we have strong performance requirements. Moreover, these libraries are rather difficult to port to recent highly parallel architectures, such as GPUs, since they implement very complex arithmetic algorithms, and they employ non-trivial memory management. Their complexity also makes them very difficult to prove formally.

In order to take advantage of the availability and efficiency of standard floating-point operations, our approach consists in representing higher precision numbers as unevaluated sums of several FP numbers (of different magnitudes). This representation is called double-double when the numbers are made up with two double-precision numbers, triple-double for three double-precision numbers, etc., and *floating-point expansion* in the general case (an arbitrary number of terms). The arithmetic operations on such representations are based on the use of *error-free transforms*, namely algorithms that allow one to compute the error of a FP addition or multiplication exactly. For instance, the sum of two FP numbers can be represented *exactly* 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 computed at a very low cost by a simple sequence of standard precision FP operations. For instance, assuming that a and b are two FP numbers and that the rounding function, denoted RN, is one of the two round-to-nearest functions defined by IEEE 754-2008, a simple algorithm (called *2Sum* and due to Knuth [11]) computes the decomposition of $a + b$ using only 6 FP operations (see Algorithm 1). Similarly, if a FMA operator[4] is available, *2ProdFMA* returns π and the error e (namely $ab - \pi$) in 2 FP operations (see Algorithm 2). Algorithms like this can be extended to be used with arbitrary precision computations by chaining, resulting into the so called *distillation* algorithms [12].

Algorithm 1 2Sum(a, b)

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

Algorithm 2 2ProdFMA(a, b)

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

[4] A FMA (Fused Multiply-Add) operator evaluates an expression of the form $xy + t$ with one final rounding only.

It is thus possible to compute very accurate values even when rounding occurs at the intermediate operation's level. However, proving correctness and computing error bounds for this kind of algorithms is quite tricky and often their formal proof is necessary. Currently, the only available and easily portable code for manipulating such floating-point expansions is Bailey's QD library [8]. It provides double-double (DD) and quad-double (QD) arithmetic. 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 and no interval arithmetic can be constructed based on this library. However, the performance results of QD are very good on tested problems (e.g. on SDP instances [10]).

We generalize or modify this kind of algorithms in order to prove their correctness and keep good performances. We provide intermediary formats (such as triple-double) and also we generalize the use of expansions to those based on single-precision (for some processors which support only this format).

3 Key features

CAMPARY is fully supported on and suitable for GPUs. This is because most available GPUs are compliant with the IEEE 754-2008 standard for FP arithmetic for both single and double precision; all rounding modes are provided and dynamic rounding mode change is supported without penalties. The `fma` instruction is supported in all devices with CUDA compute capability ≥ 2.0.

We implemented and proved new algorithms for normalizing, adding, multiplying, dividing and square rooting FP expansions. The method we use for computing the reciprocal and the square root of a FP expansion is based on an adapted Newton-Raphson iteration, where the intermediate calculations are done using "truncated" operations (additions, multiplications) involving FP expansions. We gave a thorough error analysis showing that it allows for very accurate computations (see [13]). We also introduced a new multiplication algorithm for FP expansions with arbitrary precision, up to the order of tens of FP elements in mind. The main feature consists in the partial products being accumulated in a specially designed data structure that has the regularity of a fixed-point representation while allowing the computation to be naturally carried out using native FP types. This allows us to easily avoid unnecessary computation and to obtain a rigorous accuracy analysis. The correctness and accuracy proofs of the algorithm and performance comparisons with existing libraries are presented in [14].

Fully certified algorithms like the aforementioned usually come with a performance cost. Thus, we chose to offer besides these, some so-called "quick-and-dirty" algorithms. *(i)* The certified ones [5] come with correctness proofs and they ensure the resulted expansion to be non-overlapping. Roughly speaking this means that an FP expansion carries sufficient information by ensuring that the

[5] Certified algorithms are available in *multi_prec_certified.h* file

each two consecutive terms, say u_i and u_{i+1} are sufficiently far apart; for example, $|u_{i+1}| \leq \text{ulp}(u_i)$, where ulp is the unit in the last place [11, Chap.2]. This is achieved by using different re-normalization algorithms, depending on the methods used for computing. Moreover, these algorithms offer a very tight error bound on the result. *(ii)* The "quick-and-dirty"[6] use faster versions of re-normalization algorithms. In most cases the result is going to be the same as obtained when computing with the certified level, even the non-overlapping condition can be achieved. The result may be uncertain if cancellation happens during intermediate computations; this can generate intermediate 0s or even non-monotonic expansions in the result. Also the worst case error bounds that we are able to prove are not as tight as in the certified level. We recommend the use of the "quick-and-dirty" if the performance requirements are strong, especially if there is a possibility to a posteriori check the correctness of the numerical result.

4 Applications

In what follows we briefly describe two applications (one achieved and one ongoing work) for CAMPARY.

4.1 Hénon map iteration

In [3], we studied the behavior of the Hénon map, a classical two-parameter, invertible map $h(x, y) = (1 + y - ax^2, bx)$. Depending on the two parameters a and b, this map can be chaotic, regular (the attractor of the map is a stable periodic orbit, also called sink), or a combination of these. We were interested in observing whether near the classical parameters $a = 1.4$ and $b = 0.3$, the Hénon map is chaotic and supports a strange attractor. This property has been observed numerically, but the question whether the Hénon attractor is indeed chaotic (trajectories belonging to the attractor are aperiodic and sensitive to initial conditions) or not remains open. In order to disprove this conjecture and find sinks for parameters close to the classical ones, we need to compute very long orbits for a large amount of initial points and parameters. Iterating the map for various initial points is a classical SIMD parallel problem, so a GPU implementation was done. For a double-precision implementation we obtain a significant speed-up of 21.5x compared to a multi-threaded CPU implementation. In order to tackle the conjecture, we used CAMPARY. The strategy for locating sinks is briefly the following: *(1)* We long term iterate the map ($10^6 \sim 10^9$ iterations) for various combinations of parameters and initial points in order to identify (using some additional tricks) some "numerical periodic orbits". A GPU code snippet for iterating the map is in Figure 1. This very computationally intensive search process is parallelized;*(2)*At the end of the search, we rigorously prove the existence of periodic orbits using methods from interval analysis. This part is checked "on-line" on a CPU architecture. A performance results comparison

[6] "Quick-and-dirty" algorithms are available in *multi_prec.h* file

with QD and MPFR is given in Table 1 (the "quick-and-dirty" version of the algorithms is used for CAMPARY).

```
#define prec 4
/*device fct to be run using prec*doubles precision*/
__host__ __device__ void henon_iterate(double x0, double y0,
     double a, double b, long int ITER) {
    /*init multi_prec template vars*/
    multi_prec<prec,double> x_i(x0);
    multi_prec<prec,double> y_i(y0);
    multi_prec<prec,double> x_old;
    for (long int i=1; i <= ITER; i++) {
        /*Compute iterates*/
        x_i = y_i + 1.0 - a*x_i*x_i;
        y_i = b*x_old;
    }
}
```

Fig. 1: Example of usage of template `multi_prec` types and operations with 4-doubles precision in a host or device code that performs Hénon map iterations

Prec	CAMPARY	QD
double	102398	
2-d	7608	4539
4-d	1788	618

Prec	CAMPARY	MPFR
2 doubles (106 bits)	227	11.8
3 doubles (159 bits)	76	10.6
4 doubles (212 bits)	37	10.1
6 doubles (318 bits)	15	8.9
8 doubles (424 bits)	8	7.9

Table 1: Peak number of Hénon map orbits/second for double vs. extended precision obtained using 10^6 iterations/orbit: (left) CAMPARY vs. QD library on a Tesla GPU[C2075]; (right) CAMPARY vs. MPFR (both parallelized with OpenMP on 8 threads) on Intel i7-3820 @3.60GHz.

4.2 SDP programming

We currently consider the large-scale numerically sensitive semi-definite programs (SDP) on linear matrix inequalities (LMI). SDP can be seen as an extension of linear programming to the cone of symmetric matrices with positive eigenvalues, and where the linear vector inequalities are replaced by LMI. LMI are an important modeling tool in various areas of signal processing or automatic control. Currently, SDPA [10] is the leading multiple precision HPC SDP solver. Versions of SDPA (SDPA-GMP, SDPA-DD, SDPA-QD) use different multiple-precision libraries for performing accurate computations. Among those, SDPA-DD and SDPA-QD are reported to be the fastest on the market. In our present study, we replaced QD with CAMPARY at the compilation step of SDPA (no other tuning performed). We considered test problems from [15] where the performances of the previously mentioned libraries are compared. Preliminary results given in Table 2 show that CAMPARY is competitive for this kind of application also.

Problem	SDPA-DD	SDPA-QD	SDPA-CAMPARY		
			(2D)	(3D)	(4D)
`gpp124-1`	optimal: -7.3430762652465377				
relative gap	$7.72e-04$	$1.91e-18$	$7.46e-04$	$6.72e-12$	$1.43e-18$
p.feas.error	$5.42e-20$	$2.86e-41$	$2.71e-20$	$2.72e-29$	$6.88e-41$
d.feas.error	$4.40e-14$	$3.48e-21$	$1.25e-14$	$2.72e-16$	$6.41e-21$
iteration	24	57	24	39	66
time (s)	3.580	94.58	13.25	55.57	127.52
`gpp250-1`	optimal: $-1.5444916882934067e+01$				
relative gap	$5.29e-04$	$4.75e-18$	$5.22e-04$	$5.42e-12$	$5.03e-18$
p.feas.error	$3.89e-20$	$2.58e-41$	$1.35e-20$	$1.18e-30$	$6.43e-42$
d.feas.error	$9.78e-14$	$1.64e-21$	$3.52e-14$	$5.92e-16$	$1.14e-21$
iteration	25	58	25	46	56
time (s)	28.93	762.89	132.1	527.33	856.16
`gpp500-1`	optimal: $-2.5320543879075787e+01$				
relative gap	$1.008e-03$	$2.13e-18$	$3.67e-04$	$5.78e-12$	$8.52e-18$
p.feas.error	$1.01e-20$	$5.73e-39$	$1.35e-20$	$2.01e-28$	$3.76e-42$
d.feas.error	$5.29e-14$	$1.70e-21$	$1.47e-13$	$1.03e-16$	$3.81e-21$
iteration	25	55	26	42	58
time (s)	230.05	5738.42	1027	3759.72	7146.72
`qap10`	optimal: $-1.0926074684462389e+03$				
relative gap	$3.84e-05$	$2.06e-14$	$9.82e-05$	$2.40e-10$	$3.86e-14$
p.feas.error	$2.54e-21$	$1.09e-46$	$8.27e-22$	$2.64e-34$	$9.85e-47$
d.feas.error	$4.91e-14$	$2.97e-30$	$2.62e-13$	$1.98e-22$	$1.18e-29$
iteration	20	36	19	29	36
time (s)	30.46	645.28	115.3	371.88	762.53
`hinf3`	optimal: $5.6940778009669388e+01$				
relative gap	$1.35e-08$	$5.30e-31$	$2.59e-06$	$2.47e-24$	$1.98e-31$
p.feas.error	$2.75e-24$	$1.18e-54$	$1.65e-23$	$7.10e-39$	$2.37e-55$
d.feas.error	$3.82e-14$	$1.74e-38$	$3.66e-14$	$1.79e-29$	$7.89e-42$
iteration	30	47	24	46	48
time (s)	0.02	0.11	0.03	0.07	0.12

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

5 Conclusion and Future Developments

Although initially used as a prototype for extended precision iterations of dynamical systems, CAMPARY has become a self-contained multiple precision arithmetic library mainly tuned for NVidia GPUs. We provide support for all arithmetic operations, so the first extension is to use and test it in the context of programs that make use of linear algebra, like SDP programming. Preliminary CPU implementations show good results. On short term, we intend to provide a GPU implementation for SDPA-CAMPARY. Concerning the certified part, a current ongoing work aims at formally proving our arithmetic algorithms using the Coq proof assistant [16]. A first proof of the renormalization algorithm is almost completed. A long term goal is to provide elementary functions also.

References

1. NVIDIA, *NVIDIA CUDA Programming Guide 5.5*, 2013.
2. IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic*, *IEEE Standard 754-2008*, Aug. 2008, `http://ieeexplore.ieee.org/servlet/opac?punumber=4610933`
3. M. Joldes, V. Popescu, and W. Tucker. *Searching for sinks for the Hnon map using a multiple-precision GPU arithmetic library.* SIGARCH Comput. Archit. News, 42(4):6368, December 2014.
4. J. Laskar and M. Gastineau. *Existence of collisional trajectories of Mercury, Mars and Venus with the Earth.* Nature 459, 7248 (June 2009), 817819.
5. D.H. Bailey, J.M. Borwein, P.B. Borwein and S. Plouffe. *The Quest for Pi. Mathematical Intelligencer*, No. 1, Vol. 19, 1997, pp. 50–57.
6. D.H. Priest. *Algorithms for arbitrary precision floating point arithmetic.* In Proceedings of the 10th IEEE Symposium on Computer Arithmetic (Arith-10) (June 1991), P. Kornerup and D. W. Matula, Eds., IEEE Computer Society Press, Los Alamitos, CA, pp. 132144.
7. L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier and P. Zimmermann. *MPFR: A multiple-precision binary floating-point library with correct rounding.* ACM Transactions on Mathematical Software 33, 2 (2007), Art.13, 115.
8. 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) (Vail, CO, June 2001), N. Burgess and L. Ciminiera, Eds., pp. 155162.
9. H.D. Mittelmann and F. Vallentin. *High-Accuracy Semidefinite Programming Bounds for Kissing Numbers*, Experimental Mathematics, 19:2 (2010), 175-179.
10. 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 Miguel F.Anjos and Jean B. Lasserre, editors, Handbook on Semidefinite, Conic and Polynomial Optimization, vol.166 of International Series in Operations Research & Management Science, pages 687713. Springer US, 2012.
11. 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.
12. 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.
13. M. Joldes, O. Marty, J.-M. Muller and V. Popescu. *Arithmetic algorithms for extended precision using floating-point expansions*, IEEE Transactions on Computers, Vol. 65, Issue 4, April 2016.
14. J.-M. Muller, V. Popescu and P. Tak P. Tang. *A new multiplication algorithm for extended precision using floating-point expansions*, to appear in Proceedings of the 23rd Symposium on Computer Arithmetic (ARITH-23), July 2016.
15. M. Nakata. *A numerical evaluation of highly accurate multiple-precision arithmetic version of semidefinite programming solver: SDPA-GMP,-QD and-DD.* Computer-Aided Control System Design (CACSD), 2010 IEEE International Symposium on. IEEE, 2010.
16. The Coq Development team. *The Coq proof assistant reference manual*, Inria, 2004. Version 8.0, `http://coq.inria.fr`.