# Searching for sinks for the Hénon map using a multiple-precision GPU arithmetic library

Mioara Joldes* Valentina Popescu† Warwick Tucker ‡

## ABSTRACT

Today, GPUs represent an important hardware development platform for many problems in dynamical systems, where massive parallel computations are needed. Beside that, many numerical studies of chaotic dynamical systems require a computing precision higher than common floating point (FP) formats. One such application is locating invariant sets for chaotic dynamical systems. In particular, we focus on rigorously proving the existence of stable periodic orbits for the Hénon map for parameter values close to the classical ones. For that, we present a multiple-precision floating-point arithmetic library in CUDA programming language for the NVIDIA GPU platform. Our library extends the precision using so-called *FP expansions*, where a number is represented as the unevaluated sum of standard machine precision FP numbers. This format offers the advantage of using directly available and highly optimized hardware FP operations. We generalize algorithms used by multiple-precisions libraries such as Bailey's QD, or the analogue GPU version, GQD.

## Keywords

Floating-point arithmetic, multiple precision library, GPGPU computing, error-free transform, Hénon map, dynamical systems

## 1. INTRODUCTION

The advent of high-performance computing architectures allows for the numerical study of many problems in dynamical systems, like bifurcations analysis or periodic orbit computation. However, for long time iterations of chaotic systems, two issues occur: *(i)* one usually needs *more precision than the standard IEEE* 64*-bit floating-point arithmetic*; *(ii)* numerical observed phenomena have to be *rigorously proven in a computer assisted way* using validated numerics. A classical example is that of the Lorenz system [9], for which both *(i) and (ii)* were tackled in literature: the conjecture that the structure of the solution is that of a strange attractor was proven in a computed assisted way [22]; extended precision methods were developed for iterating this system numerically with several hundreds of digits [1].

In this article we focus on another well-known dynamical system: the Hénon map [6], which can be considered as one of the "classic" discrete dynamical systems, but for which several long-standing open questions remain. The Hénon map [6] is a 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), or a combination of these. It is conjectured that for 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.*

We present a method which uses parallel computations on GPUs and constitutes the main part of the process of disproving this conjecture for several parameters close to the classical ones. For that, we need to compute very long orbits for a large amount of initial points and parameters. This computation can be viewed as a SIMD parallel problem, so a GPU implementation is presented in Sec. 2. Using double precision only, several parameters (close to the classical ones) for which the conjecture is disproved have been reported [5]. But in order to tackle the conjecture for parameters even closer to the classical ones we need a higher precision than double. Existing multiple-precision GPU-tuned arithmetic libraries (recalled in Sec. 3) prove to be unusable or suboptimal for our application. Thus, we designed also *a multiple-precision floating-point arithmetic library* using the CUDA programming language [16] for the NVIDIA GPU platform that we detail in Sec. 4. Finally, we present and compare our implementation results for this application when using multiple precision in Sec. 5.

## 2. PARALLELIZATION OF THE SEARCH FOR SINKS OF HÉNON MAP

It is known [3] that there is a set of parameters (near $b = 0$) with positive Lebesgue measure for which the Hénon map has a strange (chaotic) attractor. The parameter space is believed to be densely filled with open regions, where the attractor consists of one or more stable periodic orbits (sinks). In light of this, it is probably impossible to verify that, given a specific point $(a, b)$ in parameter space, the dynamics of the map generates a strange attractor. On the

*LAAS-CNRS, 7 Avenue du Colonel Roche, 31077 Toulouse, France, joldes@laas.fr

†LAAS-CNRS, 7 Avenue du Colonel Roche, 31077 Toulouse, France, popescu@laas.fr

‡Department of Mathematics, Uppsala Univ., Box 480, 75106 Uppsala, Sweden, warwick@math.uu.se

This work was presented in part at the international symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART2014), Sendai, Japan, June 9-11, 2014.

other hand, it was recently proven using validated numerics [5] that for several parameter values close to the classical ones, what appears to be a strange attractor (Fig. 1(a)) is actually a stable periodic orbit (Fig. 1(b)). Specifically, in Fig. 1, 10000 iterations of the Hénon map $h(x, y)$, with fixed parameters $a = 1.399999486944$ and $b = 0.3$ are plotted. The iterates appearing in Fig. 1(a) start in a point $(x_0', y_0')$ and those for Fig. 1(b) in $(x_0'', y_0'')$, which are chosen in the following way: $5 \cdot 10^9$ iterations are performed and skipped (not plotted) before obtaining $(x_0', y_0')$; and respectively, for (b) $6 \cdot 10^9$ iterations are skipped before obtaining $(x_0'', y_0'')$. Clearly, Fig. 1(a) looks like the Hénon strange attractor, while Fig. 1(b) is just a periodic orbit. This means that what we observe in computer simulations is actually a transient behavior to the periodic steady state that we are actually interested in.

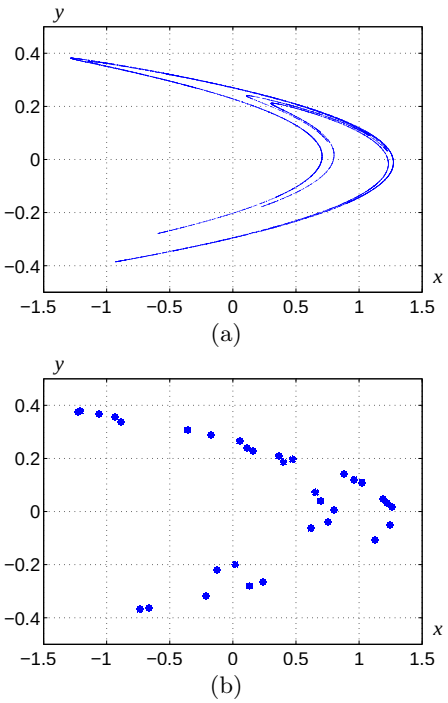


**Figure 1: Hénon map $h(x, y) = (1 + y - ax^2, bx)$ with $a = 1.399999486944$, $b = 0.3$; 10000 iterates are plotted after skipping (a) $5 \cdot 10^9$ and (b) $6 \cdot 10^9$ iterations.**

Proving the existence of such a stable periodic orbit involves a finite (yet challenging) amount of computations and we should theoretically be able to find them using high performance computing. We adapt the method in [5] where a CPU architecture is used. In brief, *(i)* for each considered point $(a, b)$ in parameter space, we perform a large amount of iterations of the Hénon map $h$ for many different initial points. The hope is that at least one of these trajectories will, after some initial transient behaviour, be attracted to what appears to be a periodic orbit; *(ii)* we use rigorous numerics to validate/falsify the existence of any sink found at step *(i)*. These main steps are explained in what follows.

Given a fixed $(a, b)$ together with a single initial point $(x_0, y_0)$, the subsequent computations are governed by two integers $N_t$ and $p_{max}$. First, we perform $N_t$ iterations of the map $h$ which now depends on $(a, b)$: $h(x_0, y_0)$, $h(h(x_0, y_0))$,..., $h^{N_t}(x_0, y_0)$. These are all discarded, except the final iterate $h^{N_t}(x_0, y_0)$, which we continue to follow for another $p_{max}$ iterates. At this stage, we examine the piece of orbit $h^{N_t+1}(x_0, y_0)$,..., $h^{N_t+p_{max}}(x_0, y_0)$ for any close returns. In other words, we attempt to find an integer $1 < k < p_{max}$ such that $\max_{i=1}^{k} \|h^{N_t+i}(x_0, y_0) - h^{N_t+i+k}(x_0, y_0)\|$ is small. If this succeeds, we may have found a period-$k$ sink, which we later attempt to verify using rigorous numerics.

The number $N_t$ of transient iterations which are discarded is usually chosen by trial-and-error since it depends on hidden intrinsic properties of the dynamics of the Hénon map. In practice, $N_t \sim 10^9$. In our search, we have used $p_{max} = 5000$. For each parameter we use $N_i \sim 10^3$ different initial points. Finally, we repeat the entire procedure for $N_p \sim 10^6$ parameters near $(1.4, 0.3)$.

If at the end of this search process we identify some "numerical periodic orbits", in a second step we rigorously prove their existence using methods from interval analysis [11, 14]. This part can be checked "off-line" on a CPU architecture, and we use the procedure described in [5], which is based on an interval Newton operator. This step is not detailed further here, since it is only the first part that is computationally expensive. Its complexity depends on two main factors: the precision used for computations, and the capability of exploiting the inherent parallelism available both in the parameter space and the initial points considered for each parameter.

The main idea for the parallelization on GPU is that each thread computes the iterates of the map $h$ starting with one fixed initial point and fixed parameter $(a, b)$ (these iterations are inherently sequential). The initial points are generated in a suitable region close to the attractor by a single thread on the GPU. They are stored in a shared memory array that gives access to all the other threads in the same block. Each thread writes in a shared memory array the period (if any) of the orbit, and one point of the orbit in the affirmative case. Each block is bi-dimensional and corresponds to one parameter $(a, b)$. We grid the parameter space near $(1.4, 0.3)$ and apply the above process to each grid point. We also implemented some variants where several blocks correspond to the same parameter $(a, b)$ in order to be able to iterate on more initial points. Without this ability, we are limited by the size of the block shared memory to ca $10^3$ initial points/parameter.

As a first step we implemented this GPU-oriented method in double precision to compare the performance with respect to the CPU implementation in [5]. We re-checked the same orbits already found in [5]. For example, for $10^6$ grid points for $a \in [1.3999, 1.4001]$, $b = 0.3$ fixed, 1024 orbits/parameter and $10^6$ iterations/orbit we found 57 parameters which present stable periodic orbits in 2.94h on 2 Tesla C2075 GPUs. A 21.5x speedup is obtained by our CUDA C implementation on an NVIDIAGeForce Tesla C2075 graphics card with 448 cores, 1.15GHz vs, a C implementation with OpenMP, on an Intel(R) Core(TM) i7 CPU 3820, 3.6GHz, 4 cores, 8 threads computer; for other Intel(R) platforms like Xeon E5 series the speed-ups are similar.

But double precision does not suffice if we want to obtain sinks for parameters closer to the classical ones. So we focus in what follows on how to use of multiple precision for our GPU application.

## 3. MULTIPLE PRECISION ARITHMETIC LIBRARIES AND GPUS

Today most floating-point (FP) computations – on both CPUs and GPUs – are done in double precision (also called

*binary64*) and are compliant with the IEEE 754-2008 standard [8]. The standard requires correct rounding of basic arithmetic operations with several rounding modes, i.e., the returned result should be as if computed with infinite precision, then rounded. This brings portability to numerical code and also makes it possible to build a correct interval arithmetic [11] rather easily.

But quad or higher precision is seldom implemented in hardware, and the most common solution is to use software emulation for multiple precision. There are mainly two ways of representing numbers in extended precision. First, in *multiple-digit representation*, a number is represented by a sequence of digits coupled with a single exponent. An example is the representation used in GNU MPFR [4] which is an open-source C library providing arbitrary precision with correct rounding for standard operations and functions. Currently, GNU MPFR is not supported on GPUs. Another example is ARPREC [2], which has been ported to GPUs under the name GARPREC, by Lu et al. [10].

Second, in *multiple terms representation*, a number is expressed as an unevaluated sum of several standard FP numbers. This sum is usually called a *FP expansion*. Bailey's library QD [7] supports double-double (DD) and quad-double (QD) computations; a number is represented as the unevaluated sum of 2 or 4 double-precision FP numbers. This has been ported to CUDA under the name of GQD [10]. It is known [12] that the DD/QD formats are not compliant with the IEEE 754-2008 standard, and the algorithms for basic operations in [7] do not provide correct rounding. However, this multiple term format offers the simplicity of using directly available and highly optimized FP operations. Also, most multiple terms algorithms are straightforwardly portable to highly parallel architectures such as GPUs. GQD/QD *multiple terms representation* supports only DD and QD computations, which is equivalent to roughly up to 212 bits of significand. We remark that this is not exactly the same as 212 bits of significand in the *multiple digits representation* because the multiple terms format can represent more bits (some intermediary zero bits can be skipped in this representation [12, Chap.14]). When the same precision is required, the GARPREC computation cost is usually higher than GQD [10].

In [13] an improved *multiple-digit* library – CUMP – is presented. This library is based on the low-level integer arithmetic routines of GMP, and uses the 64-bit integer arithmetic internally on the GPU instead of the double FP arithmetic used by GARPREC. On the NVIDIA Tesla C2050, CUMP is reported to be up to 2.6 times faster than GARPREC. For the sake of brevity, we refer to related works given in [13] concerning integer multiple precision and earlier work for GPUs without double precision hardware support.

For our problem, we could have used either the GQD or CUMP library. However, these turn out to be suboptimal for our purpose as detailed in Sec. 5. CUMP is based on the *multiple-digit* format, for which the basic operations is slower than those on *multiple-terms*, while GQD is *multiple terms* but limited to 4 doubles. In our work, we exploit the *multiple terms* format, and generalize Bailey's algorithms to $n$-double multiple-terms, i.e., we use Bailey's FP expansions.

# 4. LIBRARY IMPLEMENTATION

Let us fix some notation. A normal binary precision-$p$ floating-point (FP) number is a number of the form $x = M_x \cdot 2^{e_x-p+1}$, with $2^{p-1} \leq |M_x| \leq 2^p - 1$. The integer $e_x$ is called the *exponent* of $x$, and $M_x \cdot 2^{-p+1}$ is called the *significand* of $x$. We denote according to Goldberg's definition $\mathrm{ulp}(x) = 2^{e_x-p+1}$ [12, Chap. 2]. We also denote the machine epsilon by $\varepsilon = 2^{-p+1}$.

A natural extension of the notion of DD or QD is the notion of *floating-point expansion*. A *floating-point expansion* $u$ with $n$ terms is the unevaluated sum of $n$ floating-point numbers $u_0, u_1, \ldots, u_{n-1}$, in which all nonzero terms are ordered by magnitude (i.e., $u_i \neq 0 \Rightarrow |u_i| \geq |u_{i+1}|$). Arithmetic on FP expansions was introduced by Priest [17], and later by Shewchuk [21]. To ensure that such an expansion carries significantly more information than one FP number only, it is required that the $u_i$'s do not "overlap". This notion of overlapping varies depending on the authors. An expansion $u_0, u_1, \ldots, u_{n-1}$ is *$\mathcal{B}$-non-overlapping* (that is, non-overlapping according to Bailey's definition [7]) if for all $0 < i < n$, we have $|u_i| \leq \frac{1}{2}\,\mathrm{ulp}(u_{i-1})$.

Many algorithms exist for the addition and multiplication of FP expansions [18, 21, 19, 20, 7, 15]; all are based on various atomic *error-free transforms*. These are known under the names of *Fast2Sum*, *2Sum*, *2Prod*, and – when a `fma` (Fused Multiply-Add) instruction is available – *2ProdFMA*, see [12, Chap.4-5] for details. These atomic error-free algorithms use only native precision operations, but keep track of all accumulated rounding errors, ensuring that no information is lost. We mainly make use of Algorithm 1 *2Sum* and Algorithm 2 *2ProdFMA* which require 6 and 3 double precision operations, respectively.

**Algorithm 1** 2Sum $(a, b)$.

$s \leftarrow \mathrm{RN}(a + b)$
{ RN stands for performing the operation in rounding to nearest mode.}
$t \leftarrow \mathrm{RN}(b - \mathrm{RN}(s - \mathrm{RN}(s - b)))$
$e \leftarrow \mathrm{RN}(s + t)$
**return** $(s, e)$ such that $s = \mathrm{RN}(a + b)$ and $s + e = a + b$

**Algorithm 2** 2ProdFMA $(a, b)$.

$p \leftarrow \mathrm{RN}(a \cdot b)$
{ RN stands for performing the operation in rounding to nearest mode.}
$e \leftarrow \mathtt{fma}(a, b, -p)$
**return** $(p, e)$ such that $p = \mathrm{RN}(a \cdot b)$ and $p + e = a \cdot b$

We also make use of *VecSum*, given in Fig. 2 and Algorithm 3, which is simply a chain of *2Sum*, performing an error free transform on the sum of $n$ FP numbers [21, 19].Our library is implemented in CUDA – an extension of the C language developed by NVIDIA [16] for their GPUs. The algorithms mentioned above are very suitable for the GPU because all basic operations $(+, -, *, /, \sqrt{})$ conform to the IEEE 754-2008 standard for FP arithmetic for single and double precision. Support for the four rounding modes is provided and dynamic rounding mode change is supported without any penalties. The `fma` instruction is supported for all devices with CUDA Compute Capability at least 2.0.

In general, each error-free transform applied to two FP numbers, returns two FP numbers. So, an algorithm that performs addition of two expansions $x$ and $y$ with $n$ and $m$ terms, respectively, will return a FP expansion with at

**Algorithm 3** VecSum $(a_0, \ldots a_{n-1})$.

**Ensure:** $r_0 + \ldots + r_{n-1} = a_0 + \ldots + a_{n-1}$.
  $r_0 \leftarrow a_0$
  **for** $i \leftarrow 1$ to $n-1$ **do**
    $(r_i, r_{i-1}) \leftarrow$ *2Sum*$(a_i, r_{i-1})$
  **end for**
  **return** $r_0, \ldots, r_{n-1}$

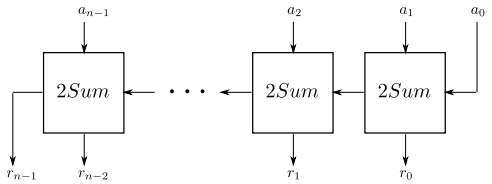


**Figure 2: VecSum with $n$ terms. Each *2Sum* box performs Algorithm 1, the sum is output to the left and the error downwards.**

most $n+m$ terms. Similarly, for multiplication, the product is a FP expansion with at most $2nm$ terms [17]. So-called normalization algorithms are used to render the result non-overlapping, and this implies also a potential reduction in the number of terms. Bailey's QD algorithms return only the most significant 2 or 4 FP terms for DD and QD formats, respectively. Analogously, we only present the *"input k - output k terms"* variant for our algorithms, although we have implemented fully customizable FP expansions algorithms for addition/subtraction and multiplication.

*Addition.* Algorithm 4, based on a chain of *2Sum* transformations (see Figure 3), generalizes Bailey's QD addition; given two FP expansions $a = a_0 + \ldots + a_{k-1}$ and $b = b_0 + \ldots + b_{k-1}$, it produces the $k$ most significant FP components of the sum $r = a + b$. In order to compute one more error correction term, that will be present in the renormalization at the end, we perform our "error free transformation scheme" $k+1$ times.

At step $n = 0$ we compute the exact sum $a_0 + b_0 = r_0 + e_0$, where roughly speaking, $r_0$ is of order $\mathcal{O}(1)$ and $e_0$ is of order $\mathcal{O}(\varepsilon)$. At each step $n = 1, \ldots, k$ we compute the exact result of $a_n + b_n = s_n + e_n$, where $s_n$ and $e_n$ are of order $\mathcal{O}(\varepsilon^n)$ and $\mathcal{O}(\varepsilon^{n+1})$, respectively. From previous steps we have already obtained $n$ error terms of order $\mathcal{O}(\varepsilon^n)$ that we add together with $s_n$ to obtain the term $r_n$ "of order" $\mathcal{O}(\varepsilon^n)$ before the renormalization step. This addition is done with the *VecSum* (see Algorithm 3). The $(k+1)$-th component $r_k$ is obtained by a simple summation of the previously obtained terms of order $\mathcal{O}(\varepsilon^k)$.

Note that, in this setting, subtraction is much simpler than for the multiple digit case, and can be performed simply by negating the FP terms in $b$.

*Multiplication.* Algorithm 5 (see also Figure 4) generalizes Bailey's QD multiplication. Again, we consider two FP expansions $a = a_0 + \ldots + a_{k-1}$ and $b = b_0 + \ldots + b_{k-1}$ and we want to compute the $k$ most significant FP components of the product $r = a \cdot b$. For the product $(p, e) =$ *2ProdFMA*$(a_i, b_j)$, $p$ is of order $\mathcal{O}(\varepsilon^n)$ and $e$ of $\mathcal{O}(\varepsilon^{n+1})$, where $n = i + j$, and we consider only the terms for which $0 \leq n \leq k$. This implies that for each $n$ we have $n+1$ products to compute (see line 4 of Algorithm 5). Next, we need to add all terms of the same order of magnitude. By induction, it can be easily shown that beside the $n+1$ products,

**Algorithm 4** Algorithm of addition of FP Expansions with $k$ terms.

**Require:** FP expansion $a = a_0 + \ldots + a_{k-1}$; $b = b_0 + \ldots + b_{k-1}$.
**Ensure:** FP expansion $r = r_0 + \ldots + r_{k-1}$.
1: $(r_0, e_0) \leftarrow$ *2Sum*$(a_0, b_0)$
2: **for** $n \leftarrow 1$ to $k-1$ **do**
3: $\quad(s_n, e_n) \leftarrow$ *2Sum*$(a_n, b_n)$
4: $\quad r_n, e_0, \ldots, e_{n-1} \leftarrow$ *VecSum*$(s_n, e_0, \ldots, e_{n-1})$
5: **end for**
6: $r_k \leftarrow 0$
7: **for** $i \leftarrow 0$ to $k-1$ **do**
8: $\quad r_k \leftarrow r_k + e_i$
9: **end for**
10: $r[0 : k-1] \leftarrow$ *Renormalize*$(r[0 : k])$
11: **return** FP expansion $r = r_0 + \ldots + r_{k-1}$.

we also have $n^2$ terms resulting from the previous iteration. This addition is performed using *VecSum* to obtain $r_n$ in line 6. The remaining terms are concatenated with the errors from the $n+1$ products, and the entire $e_0, \ldots, e_{(n+1)^2-1}$ array is used in the next iteration. The $(k+1)$-th component $r_k$ is obtained by simple summation of all remaining errors with the simple products of order $\mathcal{O}(\varepsilon^k)$. *2ProdFMA* is not needed in the last step since the errors are not reused.

**Algorithm 5** Algorithm of multiplication of FP Expansions with $k$ terms.

**Require:** FP expansion $a = a_0 + \ldots + a_{k-1}$; $b = b_0 + \ldots + b_{k-1}$.
**Ensure:** FP expansion $r = r_0 + \ldots + r_{k-1}$.
1: $(r_0, e_0) \leftarrow$ *2ProdFMA*$(a_0, b_0)$
2: **for** $n \leftarrow 1$ to $k-1$ **do**
3: $\quad$**for** $i \leftarrow 0$ to $n$ **do**
4: $\quad\quad(p_i, \hat{e}_i) \leftarrow$ *2ProdFMA*$(a_i, b_{n-i})$
5: $\quad$**end for**
6: $\quad r_n, e[0 : n^2 + n - 1] \leftarrow$ *VecSum*$(p[0 : n], e[0 : n^2 - 1])$
7: $\quad e[0 : (n+1)^2 - 1] \leftarrow e[0 : n^2 + n - 1], \hat{e}[0 : n]$
8: **end for**
9: **for** $i \leftarrow 1$ to $k-1$ **do**
10: $\quad r_k \leftarrow r_k + a_i \cdot b_{k-i}$
11: **end for**
12: **for** $i \leftarrow 0$ to $k^2 - 1$ **do**
13: $\quad r_k \leftarrow r_k + e_i$
14: **end for**
15: $r[0 : k-1] \leftarrow$ *Renormalize*$(r[0 : k])$
16: **return** FP expansion $r = r_0 + \ldots + r_{k-1}$.

The division algorithm is also implemented, but not discussed in this article, since we do not use it in our Hénon code. In our library we use templates for both the number of terms in the expansion and the native type for the terms. In other words, we allow static generation of any input-output precision combinations (e.g. add a double-double with a quad-double and store the result on triple-double) and operations with types like single-single, quad-single etc., are supported. All the functions are defined using __host__ __device__ specifiers, which allows for the library to be used on both CPU and GPU. We also favor in-place algorithms to avoid register spill/loads. Basic operations for interval arithmetic are also supported in a similar templated class. For this class, we implemented Priest's [18] and Shewchuk's [21] algorithms,

which are generally slower than those presented in this article, but they allow for enclosing correctly in an outward rounding mode the rounding errors.

Understandably, each multiple-precision arithmetic operation presented is not parallelized. Some parallelized algorithms for addition and dot product for large arrays of numbers have been proposed in [23], but for our purpose – and this is the case for all existing multiple precision libraries – the parallelization takes place at the problem level.

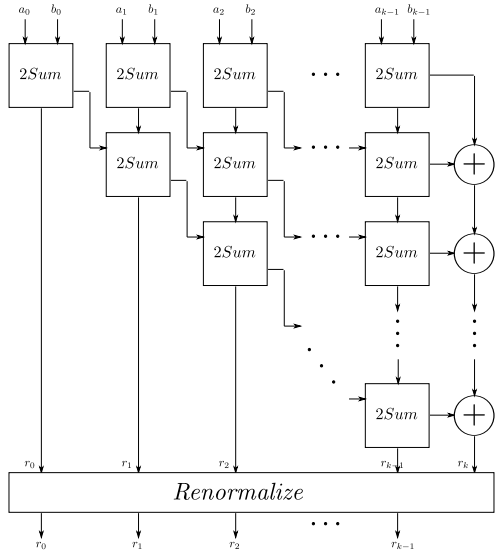


**Figure 3: Addition of FP Expansions with $k$ terms. Each *2Sum* box performs Algorithm 1, the sum is output downwards and the error to the right.**

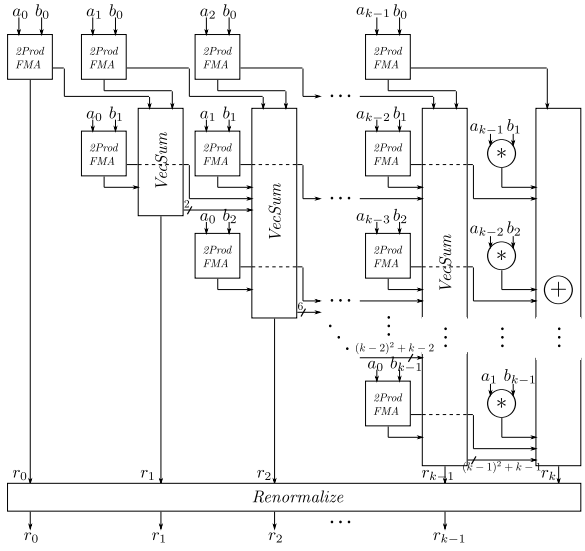


**Figure 4: Multiplication of FP Expansions with $k$ terms. Each *2ProdFMA* box performs Algorithm 2, the product is output downwards and the error to the right; *VecSum* stands for Algorithm 3.**

In our implementation we use a sequential memory layout for the expansion terms, whereas CUMP/GARPREC libraries use an interval memory layout where the terms of multiple precision numbers are interleaved. Specifically, for an array of $n$ multiple precision numbers, each with $m$ terms, the $j$th term of the $i$th number is stored in the position $jn+i$ in the array. This format is better suited for operations with large arrays of multiple precision numbers, since it favors coalesced accesses of off-chip memory. In what follows, our parallelization scheme is different for the Hénon map study, and does not require a special memory layout. Our library code will be made freely available.

# 5. RESULTS AND CONCLUSIONS

Our multiple precision library supports data and arithmetic operations for both host and device code. As such, only minor changes to the CUDA code written for double precision Hénon map iterations are required to allow the usage of our library. A code snippet from a Hénon like GPU kernel using 4-double precision from our `multi_prec` templated class is given in Fig. 5.

```
#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_old = x_i;
        x_i = y_i + 1.0 - a*x_i*x_i;
        y_i = b*x_old;
    }
}
```

**Figure 5: 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**

A performance comparison between `multi_prec` versus double computations on GPU for Hénon iterations is given in Table 1. In the same table, we also compare the GPU performance of our library versus QD (which supports only 2 and 4-doubles precision).

Currently, we are not able to compare GARPREC/CUMP performance for our Hénon map application, because they were both tuned for big array operations where the data is generated on the host, and only the operations are performed on the device. In our case, each thread needs to generate and allocate multiple precision data on the device. With these constraints, we were not able to implement our application using these libraries so far. However, in [10] it is stated that GQD should be faster that GARPREC for double-double and quad-double computations. Moreover, it is also known [7, 10] that multiple-terms operations are faster than multiple-digit ones for precisions in the range of up to several hundreds of bits. This is confirmed in our case, in Table 2.

More precisely, for the same benchmark Hénon map code we also compare the performances on CPU for our library versus MPFR in Table 2. This comparison is not entirely fair seeing that the multiple-digit format is not equivalent to the multiple-terms format. Indeed, we do not guarantee the correct rounding for each basic operation, but we present this comparison from a prospective point of view. As future work we intend to provide error bounds for our *FP expansions algorithms*; with this improvement, *multiple terms* format could become an interesting alternative even for the CPU architectures. One limitation is the precision supported with FP expansions, which is about 2000 bits (39

| Precision | Mprec | QD |
|---|---|---|
| **double** | 102398 | |
| **2 doubles** | 7608 | 4539 |
| **3 doubles** | 5200 | * |
| **4 doubles** | 1788 | 618 |
| **5 doubles** | 758 | * |
| **6 doubles** | 374 | * |
| **7 doubles** | 205 | * |
| **8 doubles** | 122 | * |

**Table 1: Peak number of Hénon map orbits/second for double vs. extended precision obtained with our library Mprec vs. QD library on Tesla GPU[C2075] using $10^6$ iterations/orbit. *precision not supported**

| Precision | Mprec | 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 2: Peak number of Hénon map orbits/second for obtained with our library Mprec vs. MPFR library (both parallelized with OpenMP on 8 threads) on Intel i7-3820 @3.60GHz using $10^6$ iterations/orbit.**

components), which occurs only if the first component is near overflow and the last near underflow.

It is clear from a mathematical point of view that only a very small amount of sinks can be found using double precision. At the same time, any sink can be resolved using a sufficiently high precision. Currently, we are searching for a sink at the classical parameters using 4-doubles `multi_prec` operations.

# 6. REFERENCES


[1] A. Abad, R. Barrio, and A. Dena. Computing periodic orbits with arbitrary precision. *Phys. Rev. E*, 84:016701, Jul 2011.

[2] D. H. Bailey, Y. Hida, X. S. Li, and B. Thompson. ARPREC: an arbitrary precision computation package. Technical report, Lawrence Berkeley National Laboratory, 2002. Available at `http://crd.lbl.gov/~dhbailey/dhbpapers/arprec.pdf`.

[3] M. Benedicks and L. Carleson. The dynamics of the Hénon map. *Annals of Mathematics*, 133(1):pp. 73–169, 1991.

[4] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding. 33(2), 2007. available at `http://www.mpfr.org/`.

[5] Z. Galias and W. Tucker. Combination of exhaustive search and continuation method for the study of sinks in the Hénon map. In *Proc. IEEE Int. Symposium on Circuits and Systems, ISCAS'13*, pages 2571–2574, Beijing, May 2013.

[6] M. Hénon. A two-dimensional mapping with a strange attractor. *Communications in Mathematical Physics*, 50:69–77, 1976. 10.1007/BF01608556.

[7] 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)*, pages 155–162, June 2001.

[8] 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`.

[9] E. N. Lorenz. Deterministic Nonperiodic Flow. *J. Atmos. Sci.*, 20(2):130–141, Mar. 1963.

[10] M. Lu, B. He, and Q. Luo. Supporting extended precision on graphics processors. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, DaMoN '10, pages 19–26, New York, NY, USA, 2010. ACM.

[11] R. Moore. *Interval Analysis*. Prentice Hall, Englewood Cliffs, NJ, 1966.

[12] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torrès. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, Nov. 2009.

[13] T. Nakayama and D. Takahashi. Implementation of multiple-precision floating-point arithmetic library for GPU computing. In *Proceedings of the 23rd IASTED International Conference on Parallel and Distributed Computing and Systems*, PDCS 2011, pages 343–349, December 2011.

[14] A. Neumaier. *Interval methods for systems of equations*. Cambridge University Press, 1990.

[15] H. D. Nguyen, S. Graillat, and J.-L. Lamotte. Extended precision with a rounding mode toward zero environment. Application to the Cell processor. *International Journal of Reliability and Safety*, 3(1):153–173, 2009.

[16] NVIDIA. *NVIDIA CUDA Programming Guide 5.5*. 2013.

[17] D. M. Priest. Algorithms for arbitrary precision floating point arithmetic. In *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 132–145. IEEE Computer Society Press, 1991.

[18] D. M. Priest. *On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*. Thesis (Ph.D. in mathematics), Department of Computer Science, University of California, Berkeley, Berkeley, CA, USA, 1992.

[19] S. M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part I: Faithful rounding. *SIAM J. Sci. Comput.*, 31(1):189–224, Oct. 2008.

[20] S. M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part II: Sign, k-fold faithful and rounding to nearest. *SIAM J. Scientific Computing*, 31(2):1269–1302, 2008.

[21] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18:305–363, 1996.

[22] W. Tucker. A rigorous ODE solver and Smale's 14th problem. *Foundations of Computational Mathematics*, 2(1):53–117, 2002.

[23] N. Yamanaka, T. Ogita, S. M. Rump, and S. Oishi. A parallel algorithm for accurate dot product. *Parallel Comput.*, 34(6-8):392–410, 2008.