GPU-Accelerated Generation of Correctly Rounded Elementary Functions

PIERRE FORTIN, Sorbonne Universités and CNRS
MOURAD GOUICEM, LIRMM, CNRS/Université Montpellier 2, Sorbonne Universités, and CNRS
STEF GRAILLAT, Sorbonne Universités and CNRS

The IEEE 754-2008 standard recommends the correct rounding of some elementary functions. This requires solving the Table Maker's Dilemma (TMD), which implies a huge amount of CPU computation time. In this article, we consider accelerating such computations, namely the Lefèvre algorithm on graphics processing units (GPUs), which are massively parallel architectures with a partial single instruction, multiple data execution.

We first propose an analysis of the Lefèvre hard-to-round argument search using the concept of continued fractions. We then propose a new parallel search algorithm that is much more efficient on GPUs thanks to its more regular control flow. We also present an efficient hybrid CPU-GPU deployment of the generation of the polynomial approximations required in the Lefèvre algorithm. In the end, we manage to obtain overall speedups up to 53.4× on one GPU over a sequential CPU execution and up to 7.1× over a hex-core CPU, which enable a much faster solution of the TMD for the double-precision format.

Categories and Subject Descriptors: G.1.0 [**Numerical Analysis**]: General—*Computer arithmetic*; G.4 [**Mathematical Software**]—Parallel and Vector Implementations

General Terms: Algorithms, Reliability, Performance

Additional Key Words and Phrases: Correct rounding, Table Maker's Dilemma, Lefèvre algorithm, GPU computing, SIMD, control flow divergence, floating-point arithmetic, elementary function

ACM Reference Format:
Pierre Fortin, Mourad Gouicem, and Stef Graillat. 2016. GPU-accelerated generation of correctly rounded elementary functions. ACM Trans. Math. Softw. 43, 3, Article 22 (December 2016), 26 pages.
DOI: http://dx.doi.org/10.1145/2935746

1. INTRODUCTION

1.1. Problem

Since 1985, the IEEE 754 standard has specified the implementation of floating-point operations to have portable and predictable numerical software. In its latest revision in 2008 [IEEE Computer Society 2008], it defines formats (in this article, we will consider only binary formats binary32, binary64, and binary128, even though the presented algorithms work for any radix), rounding modes (to nearest and toward 0, $-\infty$ and $+\infty$), and operations ($+$, $-$, $\times$, $/$, $\sqrt{}$, FMA) returning correctly rounded values.

This work was supported by the TaMaDi project of the French ANR (grant ANR 2010 BLAN 0203 01).
Authors' addresses: P. Fortin, Sorbonne Universités, UPMC Univ Paris 06, CNRS, UMR 7606, LIP6, F-75005 Paris, France; email: pierre.fortin@lip6.fr; M. Gouicem, LIRMM, CNRS/Université Montpellier 2, UMR 5506, Montpellier, France and Sorbonne Universités, UPMC Univ Paris 06, CNRS, UMR 7606, LIP6, F-75005 Paris, France; email: mourad.gouicem@intel.com; S. Graillat, Sorbonne Universités, UPMC Univ Paris 06, CNRS, UMR 7606, LIP6, F-75005 Paris, France; email: stef.graillat@upmc.fr.

DOI: http://dx.doi.org/10.1145/2935746

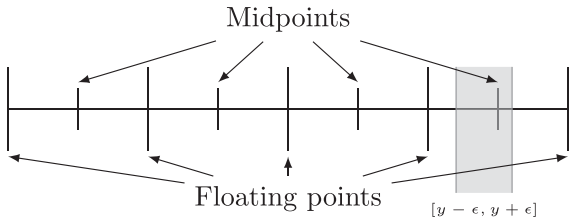


Fig. 1. Example of undetermined correct rounding for rounding to nearest, where the rounding breakpoints are the midpoints of floating-point numbers.

Furthermore, it recommends correct rounding of some elementary functions, such as *log*, *exp*, and the trigonometric functions. However, it is hard to decide which intermediate precision is required to guarantee a correctly rounded result for an elementary function—the rounded evaluation of the approximation must be equal to the rounded evaluation of the function with infinite precision. This problem is known as the Table Maker's Dilemma (TMD) [Muller et al. 2009].

1.2. State of the Art

There exist theoretical bounds on the intermediate precision required for correctly rounded functions [Muller et al. 2009], but these are not sharp enough for efficient floating-point implementations of elementary functions. For example, the Nesterenko and Waldschmidt [1996] bound for the exponential in double precision states that 7,290,678 bits of intermediate precision suffice to provide a correctly rounded result. However, according to probabilistic hypotheses, we expect this intermediate precision to be slightly more than twice the working precision (106 bits for double precision) [Muller et al. 2009]. Hence, ad hoc methods are needed to find a sharper bound for each function.

A first method introduced by Ziv [1991] was to compute an approximation y of a function value $f(x)$ with an error bounded by $\epsilon = |y - f(x)|$ (containing mathematical and round-off errors). As rounding modes are monotonic, if $y - \epsilon$ and $y + \epsilon$ round to the same floating-point number, then $f(x)$ does as well; otherwise, the correct rounding cannot be determined from y alone (Figure 1). Therefore, a correctly rounded result of $f(x)$ can be obtained by refining the approximation (y, ϵ), decreasing ϵ until $y - \epsilon$ and $y + \epsilon$ round to the same floating-point number. For the most common elementary functions, such an ϵ exists according to the Lindemann—Weierstrass theorem, when the function is evaluated at almost all floating-point numbers [Galochkin 2011].

However, precomputing an ϵ that guarantees correct rounding of the evaluation of f at any floating-point argument allows one to guarantee that the Ziv method will stop after no more than a few iterations and to simplify the implementation by avoiding to write unneeded high-precision Ziv iterations. This can be done by finding the hardest-to-round arguments of the function—that is to say, the arguments requiring the highest precision for the evaluation of $f(x)$ to be correctly rounded. This precision guaranteeing the correct rounding for all arguments is called the *hardness-to-round of the function*. The hardest-to-round cases can be found by invoking the Ziv algorithm at every floating-point number in the domain of definition of the function, but this is prohibitive (some multiple of 2^p operations when considering precision-p floating-point numbers as arguments).

The first improvement was proposed by Lefèvre et al. [1998] (the Lefèvre algorithm). The main idea of their algorithm is to split the domain of definition into several domains D_i, to "filter" hard-to-round cases (HR-cases) for a fixed extended precision, and then to use the Ziv algorithm to find the hardest-to-round cases among them. This filtering is efficiently performed using local affine approximations of the targeted function over $O(2^{2p/3})$ domains D_i. Stehlé et al. [2005] extended this method in 2003 (the

SLZ algorithm) for higher-degree approximations, using the Coppersmith method for finding small roots of a univariate modular equation over $O(2^{p/2})$ domains D_i.

1.3. Motivations and Contributions

Even if they are asymptotically and practically faster than an exhaustive search, the Lefèvre and SLZ algorithms remain very computationally intensive. For example, the Lefèvre algorithm required around 5 years of CPU time for the exponential function over all double-precision arguments, and the SLZ algorithm took around 9 years of CPU time for the function 2^x over extended double-precision arguments in the interval $[1/2, 1]$ [Stehlé et al. 2003]. Moreover, even if the hardest-to-round cases of some functions in double precision are known [Muller et al. 2009], it is still not the case for about half of the univariate functions recommended by IEEE standard 754-2008. Furthermore, we still have no efficient way to find the hardest-to-round cases of any elementary function in double precision, and quadruple precision is out of reach. We will hence be interested in accelerating the search for hardest-to-round cases in double precision (binary64).

As both algorithms split the domain of definition of the targeted function into domains D_i and search for HR-cases in them independently, these computations are embarrassingly and massively parallel. The purpose of this work is therefore to accelerate these computations on graphics processing units (GPUs), which theoretically perform one order of magnitude better than CPUs on suitable problems thanks to their massively parallel architectures.

We will focus here on the Lefèvre algorithm, which has been used to generate hardness-to-round values for all functions for which they are known, for double precision [Muller et al. 2009]. The Lefèvre algorithm is asymptotically less efficient than SLZ, as it considers more domains D_i ($O(2^{2p/3})$ against $O(2^{p/2})$). However, it performs fewer operations per domain D_i ($O(\log p)$ against $O(\text{poly}(p))$). Therefore, the Lefèvre algorithm is more efficient than SLZ in practice for finding the hardness-to-round of elementary functions for the double-precision format [Muller et al. 2009] and offers fine-grain parallelism, making it suitable for a GPU.

In Fortin et al. [2012], we discussed implementation techniques to deploy the original Lefèvre algorithm efficiently on GPUs, which led to an average speedup of 15.4× with respect to the reference CPU implementation on one CPU core. The major bottleneck of this GPU deployment was the control-flow divergence, which is penalizing considering the partial single instruction, multiple data (SIMD) execution of the GPU. Hardware [Brunie et al. 2012] and software [Frey et al. 2012; Han and Abdelrahman 2011] general solutions have been proposed to address this problem on GPUs. However, these solutions are not efficient in our context, as we have a very fine computation grain for each GPU thread. Hence, we focus here on algorithmic solutions to directly tackle the origin of this divergence issue.

In this article, we thus redesign the Lefèvre algorithm with the continued fraction formalism, which enables us to get a better understanding of it and to propose a much more regular algorithm for searching HR-cases. More precisely, we strongly reduce two major sources of divergence in the Lefèvre algorithm: loop divergence and branch divergence. We also propose an efficient hybrid CPU-GPU deployment of the generation of polynomial approximations D_i using fixed-sized multiprecision operations on the GPU. These contributions enable an overall speedup of 53.4× on a GPU over Lefèvre's original sequential CPU implementation and of 7.1× over six CPU cores (with two-way SMT). Finally, as we obtain in the end the same HR-cases as Lefèvre, de Dinechin, and Muller experiments [de Dinechin et al. 2011; Lefèvre and Muller 2001], we also strengthen the confidence in the generated HR-cases.

1.4. Outline

We first introduce some notions on GPU architecture and divergence in Section 2. Then in Section 3, we present some mathematical background on the TMD and properties of the set $\{a \cdot x \mod 1 \mid x < n\}$ with a fixed and x a positive integer. In Section 4, we detail the HR-case search step of the Lefèvre algorithm and of the new and more regular algorithm, along with their deployment on a GPU. In Section 5, we detail how to efficiently generate on a GPU the polynomial approximations D_i needed by the two HR-case searches. Finally, we present performance results in Section 6 and conclude in Section 7.

2. GPU COMPUTING

GPUs are many-core devices originally intended for graphics computations. However, since the mid-2000s, they became increasingly used for high-performance scientific computing, since their massively parallel architectures theoretically perform one order of magnitude better than CPUs, and since general-purpose languages adapted to GPUs (e.g., CUDA [NVIDIA 2012a] and OpenCL [Khronos Group 2011]) have emerged. In this section, we briefly describe the architecture of the NVIDIA GPU used to test our deployments (the Fermi architecture), GPU programming in CUDA, and the divergence problems arising from the partial SIMD execution on the GPU. Here we use the CUDA nomenclature.

2.1. GPU Architecture and CUDA Programming

From a hardware point of view, a GPU is composed of several *streaming multiprocessors* (SMs; there are 14 SMs on Fermi C2070), each being an SIMD unit [NVIDIA 2011]. An SM is composed of multiple execution units or *CUDA cores* (32 on Fermi) sharing the same pipeline and many registers (32,768 on Fermi). GPU memory is organized into two levels: *device memory*, which can be accessed by any SM on the device, and *shared memory*, which is local to each SM. The device memory accesses are cached on the Fermi architecture.

From a software point of view, the developer writes in CUDA a scalar code for one function designed to be executed on the device, namely a *kernel*. At runtime, many *threads* are created by *blocks* and bundled into a *grid* to run the same kernel concurrently on the device. Each block is assigned to an SM. Within each block, threads are executed in groups of 32, called *warps*. The ratio of the number of resident warps (the number of warps that an SM can process at the same time for a given kernel) to the maximum number of resident warps per SM is called the *occupancy*. To increase the occupancy, the number of blocks and their sizes have to be tuned.

2.2. Divergence

As threads are executed by warps on the GPU SIMD units, applications should have regular patterns for memory accesses and control flow.

The regularity of memory access patterns is important to achieve high memory throughput. As the threads within a warp load data from memory concurrently, the developer has to coalesce accesses to device memory and avoid bank conflicts in the shared memory (Chapter 6 in NVIDIA [2012a]). This can be done by reorganizing data storage.

The regularity of control flow is important to achieve high instruction throughput and is obtained when all threads within a warp execute the same instruction concurrently (Chapter 9 in NVIDIA [2012a]). In fact, when the threads of a same warp diverge (i.e., they follow different execution paths), the different execution paths are serialized. For an *if* statement, the *then* and *else* branches are serially executed. For a loop, any thread

exiting the loop has to wait until all threads of its warp exit the loop. In the following, we will distinguish branch divergence due to *if* statements and loop divergence due to loop statements.

The impact of branch divergence can be statically estimated by counting the number of instructions issued within the scope of the *if* statement. Let us consider the case where the *then* branch issues n_{then} instructions and the *else* branch issues n_{else} instructions. If the warp does not diverge, either n_{then} or n_{else} instructions are issued depending on the evaluation of the condition. If the warp diverges, $n_{then} + n_{else}$ instructions are issued.

Unlike branch divergence, measuring the impact of loop divergence requires a dedicated indicator and profiling. In Fortin et al. [2012], we introduced the mean deviation from the maximum of a warp. This indicator is similar to the standard deviation, which is the mean deviation from the mean value. However, as the number of loop iterations issued for a warp is equal to the maximum number of loop iterations issued by any thread within the warp, it is relevant to consider the mean deviation from the maximum value. This gives the mean number of loop iterations a thread remains idle within its warp. More formally, we denote ℓ_i as the number of loop iterations of the thread i and number the threads within a warp from 1 to 32. If $\ell = \{\ell_i, i \in [\![1, 32]\!]\}$, the mean deviation from the maximum (MDM) of a warp is defined as

$$\mathrm{MDM}(\ell) = \max(\ell) - \mathrm{mean}(\ell).$$

We can normalize the MDM by $\max(\ell)$ to compute the average percentage of loop iterations for which a thread remains idle within its warp. Hence, the normalized mean deviation from the maximum (NMDM) is

$$\mathrm{NMDM}(\ell) = 1 - \frac{\mathrm{mean}(\ell)}{\max(\ell)}.$$

3. MATHEMATICAL PRELIMINARIES

In this section, we give some definitions to introduce the TMD more formally. We also recall some known properties on the distribution of the elements of the set $\{a \cdot x \mod 1 \mid x < n\}$ with a fixed and x a positive integer [Slater 1967], as well as the corresponding continued fraction formalism.

3.1. The TMD

Before defining the TMD, we introduce some notations and definitions. We denote $\{X\}$ or $X \mod 1$ as the positive fractional part of X. We write $X \bmod 1$ as the centered modulo, which is the real Y such that $X - Y \in \mathbb{Z}$ and $Y \in (-1/2, 1/2]$ (Y equals $X - \lfloor X \rfloor$ or $X - \lceil X \rceil$ depending on which has the lowest absolute value). We also write $\mathbb{F}_p$ as the set of precision-p floating-point numbers and $|E|_p$ as the number of precision-p floating-point numbers in the set E (namely, $|E \cap \mathbb{F}_p|$). We recall that we consider only binary floating-point format in this article (even though all definitions can be generalized to any radix).

Definition 3.1. The significand $m(x)$ and the exponent $e(x)$ of a nonzero real number x are defined by $|x| = m(x) \cdot 2^{e(x)}$ with $1/2 \leq m(x) < 1$. We will also denote $M_p(x) = 2^p \cdot m(x)$ as the scaled significand.

Definition 3.2. The scaled distance between a real number x and the closest precision-p floating-point number is defined by $\mathrm{dist}_p(x) = |M_p(x) \bmod 1|$.

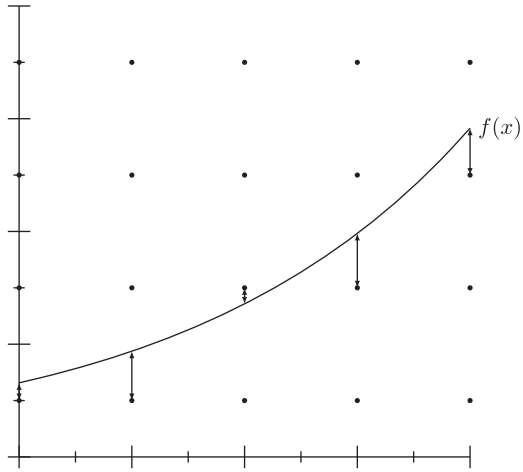


Fig. 2. Distances between the curve defined by f and the rounding breakpoints for rounding-to-nearest. The points of the grid are formed by the floating points on the x-axis and the rounding breakpoints on the y-axis.

Definition 3.3. We now define a (p, ϵ) *hard-to-round case* (or *HR-case*) of a real-valued function f as a precision-p floating-point number x solution of the inequality

$$\text{dist}_p(f(x)) < \epsilon.$$

The given definition of an HR-case only applies for directed rounding. However, this definition can be extended to all IEEE-754 rounding modes, as rounding-to-nearest (p, ϵ) HR-cases are directed rounding $(p + 1, 2\epsilon)$ HR-cases. To simplify notations, we will then focus on directed rounding HR-cases. It has to be noticed that if x is a (p, ϵ) HR-case, it also satisfies $M_p(f(x)) + \epsilon < 2\epsilon \mod 1$. The latter inequality is used to test whether an argument is a (p, ϵ) HR-case, as it avoids the computation of absolute values and cmod.

Hence, a $(p, 2^{-p'})$ HR-case x is a precision-p floating-point number for which $f(x)$ is at a scaled distance (as defined in Definition 3.2) less than $2^{-p'}$ from the closest precision-p floating-point number. In other words, more than $p + p'$ bits of accuracy are necessary to correctly round $f(x)$ at precision-p.

Definition 3.4 (*Table Maker's Dilemma*). Given a real valued function f defined over a domain D and a precision p, the TMD is defined as the problem of finding the smallest integer p' such that there are no $(p, 2^{-p'})$ HR-cases in D.

We call *hardest-to-round* cases the arguments $x \in D$ minimizing $\text{dist}_p(f(x))$. Knowing the hardest-to-round cases gives us a lower bound on the distances between the function f and the rounding breakpoints (Figure 2) and therefore a solution to the TMD, as they are $(p, 2^{-p'-1})$ HR-cases.

The general method to find the hardest-to-round cases of a function is the following:

(1) Fix a "convenient" ϵ using probabilistic assumptions [Muller et al. 2009].
(2) Find (p, ϵ) HR-cases with ad hoc methods such as the Lefèvre or the SLZ algorithms.
(3) Find the hardest-to-round among the (p, ϵ) HR-cases using the Ziv [1991] method.

The most compute-intensive step in this method is to find the (p, ϵ) HR-cases. To this purpose, the Lefèvre and SLZ algorithms both rely on the following three major steps:

(1) *The splitting of the domain of definition of the function*: We split the domain of definition of the function into d domains $D_i = [X_i, X_{i+1}) \cap \mathbb{F}_p$ such that $\forall x, y \in D_i\ e(x) = e(y)$ and $\forall i, j \in [\![0, d-1]\!]\ |D_i|_p = |D_j|_p$.
(2) *The generation of polynomial approximations*: We approximate the function $f(X)$ with $X \in D_i$ by polynomials $P_i(x)$ with $x \in [\![0, |D_i|_p - 1]\!]$ such that $|P_i(x) - f(X)| < \epsilon_{approx} 2^{e(f(X))-p}$. We thus proceed to a change of variable, enabling us to test the floating-point arguments $X \in D_i$ by testing the integers $x \in [\![0, |D_i|_p - 1]\!]$. Each polynomial P_i is first centered on the domain D_i by applying the change of variable $\phi_1 : X \mapsto X - X_i$. Then, as the exponent is constant over each domain D_i, we will consider integer arguments by applying the change of variable $\phi_2 : X \mapsto X \cdot 2^{p-e(X_i)}$. All in all, $x = \phi_2 \circ \phi_1(X) = 2^{p-e(X_i)}(X - X_i)$.
(3) *The HR-case search*: We find the (p, ϵ') HR-cases of P_i with $\epsilon' = \epsilon + \epsilon_{approx}$ that comprise all (p, ϵ) HR-cases for f in D_i.

In the HR-case search of both algorithms, a Boolean test is used to isolate HR-cases. It succeeds if there are no (p, ϵ') HR-cases for P_i in D_i and fails otherwise. It has to be noticed that ϵ' should be small enough such that very few HR-cases exist in each domain to test (based on probabilistic assumptions [Muller et al. 2009]).

In this article, we focus on the Lefèvre algorithm, which truncates the polynomials P_i to degree 1 for the Boolean test. We denote $Q_i(x) = P_i(x) \mod x^2$ as the truncation of P_i to degree 1. As the Boolean test requires a fixed scaled error bound over each domain to test against, we define $q = \min_{x \in [\![0, |D_i|_p - 1]\!]} \{e(P_i(x))\}$ such that $|Q_i(x) - P_i(x)| < \epsilon_{trunc} 2^{q-p}$, and

$$2^{p-q} Q_i(x) + \epsilon'' = b - a \cdot x,$$

with $\epsilon'' = \epsilon' + \epsilon_{trunc}$. A consequence of this scaling is to create false HR-cases where a change of exponent occurs in the codomain—these false HR-cases can easily be eliminated subsequently. Hence, the Boolean test of the Lefèvre algorithm consists of testing whether following inequality holds:

$$\min \{b - a \cdot x \mod 1 \mid x < |D_i|_p\} < 2\epsilon''. \tag{1}$$

More precisely, if the inequality (1) does not hold, then there are no (p, ϵ'') HR-cases for Q_i in D_i, which implies that there is no (p, ϵ') HR-case for P_i in D_i. Therefore, the Boolean test returns Failure if the inequality (1) holds, and Success or Failure (false HR-case) if it does not hold. In the case of Failure, further tests are needed, and in the case of Success, one can deduce that there are no HR-cases in the domain. Moreover, we remark that computing the minimum of the set $\{b - a \cdot x \mod 1 \mid x < |D_i|_p\}$ is similar to finding the multiple of a that is the closest to the left of b modulo 1 on the unit segment $[0, 1)$.

3.2. Properties of the Set $\{a \cdot x \mod 1 \mid x < n\}$

Here we will detail some properties on the configurations of the points $\{a \cdot x \mod 1 \mid x < n\}$ over the unit segment, with x a nonnegative integer. These properties are necessary to efficiently locate the closest point to $b \mod 1$ in these configurations.

THEOREM 3.5 (THREE DISTANCE THEOREM [SLATER 1950]). *Let $0 < a < 1$ be an irrational number. If we place on the unit segment $[0, 1)$ the points $\{0\}, \{a\}, \{2a\}, \ldots, \{(n-1)a\}$, then these points partition the unit segment into n intervals having at most three lengths with one being the sum of the two others.*

Actually, the lengths, and the distribution of these lengths, heavily rely on the continued fraction expansion of a [Slater 1967], which we will denote by

$$a = \cfrac{1}{k_1 + \cfrac{1}{k_2 + \ddots}}.$$

We denote by $(\theta_i)_{i\in\mathbb{N}}$ the sequence of the remainders computed during the continued fraction expansion of a using the Euclidean algorithm, and by $(p_i/q_i)_{i\in\mathbb{N}}$ the sequence of the convergents of a, defined by the following recurrence relations:

$$\begin{array}{lll} \theta_{-1} = 1, & \theta_0 = a, & \theta_{i+1} = \theta_{i-1} - k_{i+1} \cdot \theta_i; \\ p_{-1} = 1, & p_0 = 0, & p_{i+1} = p_{i-1} + k_{i+1} \cdot p_i; \\ q_{-1} = 0, & q_0 = 1, & q_{i+1} = q_{i-1} + k_{i+1} \cdot q_i; \end{array}$$

with $k_{i+1} = \lfloor \theta_{i-1}/\theta_i \rfloor$. Note that $(\theta_i)_{i\in\mathbb{N}}$ is a decreasing real-valued positive sequence, whereas $(p_i)_{i\in\mathbb{N}}$ and $(q_i)_{i\in\mathbb{N}}$ are increasing integer-valued positive sequences. We also define $\theta_{i-1,t} = \theta_{i-1} - t \cdot \theta_i$ and $q_{i-1,t} = q_{i-1} + t \cdot q_i$ with $t \in [\![0, k_{i+1}[\![$. The lengths obtained when adding multiples of a over the unit segment are therefore the elements of the sequence $(\theta_{i,t})_{i\in\mathbb{N},t\in[\![0,k_{i+1}[\![}$ [Slater 1967]. An example is provided in Figure 3 and Table I. All of the properties provided in this section are valid when a is irrational. However, they are also valid for a rational as long as $\theta_i \neq 0$ (that is to say, until the last quotient of the continued fraction expansion is computed).

In the following, we will use some properties of the configurations $\{a \cdot x \mod 1 \mid x < n\}$ that contain intervals of only two different lengths. They are of algorithmic interest, as there are only $O(\log n)$ such configurations when n tends to infinity. Each label (i, t), with $i \in \mathbb{N}$ and $t \in [\![0, k_{i+1}[\![$, denotes one two-length configuration that satisfies the following equation:

$$q_i \cdot \theta_{i-1,t} + q_{i-1,t} \cdot \theta_i = 1. \tag{2}$$

Equation (2) gives details on the number of intervals of each length. After adding $q_i + q_{i-1,t}$ multiples of $a \mod 1$ over the unit segment, there are exactly two different lengths of intervals over the unit segment: q_i intervals of length $\theta_{i-1,t}$ and $q_{i-1,t}$ intervals of length θ_i.

A special and noticeable subset of the two-length configurations corresponds to the configurations produced using the division-based Euclidean algorithm. These are the $(i, 0)$ configurations, satisfying

$$q_i \cdot \theta_{i-1} + q_{i-1} \cdot \theta_i = 1. \tag{3}$$

Furthermore, we have a way to construct the two-length configurations, which follows.

Property 1 (Two-length Configurations Construction [Slater 1967]). *Given the (i, t) two-length configuration, the next two-length configuration is*

$$\begin{cases} (i, t+1) & \text{if } t < k_{i+1} - 1, \\ (i+1, 0) & \text{if } t = k_{i+1} - 1. \end{cases}$$

To simplify the notation, given the (i, t) configuration, we will write $(i, t + 1)$ for the next two-length configuration and assimilate the configuration (i, k_{i+1}) to $(i + 1, 0)$. Property 1 implies that for going from a two-length configuration to the next one, the intervals of length $\theta_{i-1,t}$ are split. The way intervals are split is described by the following *directed reduction* property and illustrated in Figure 3.

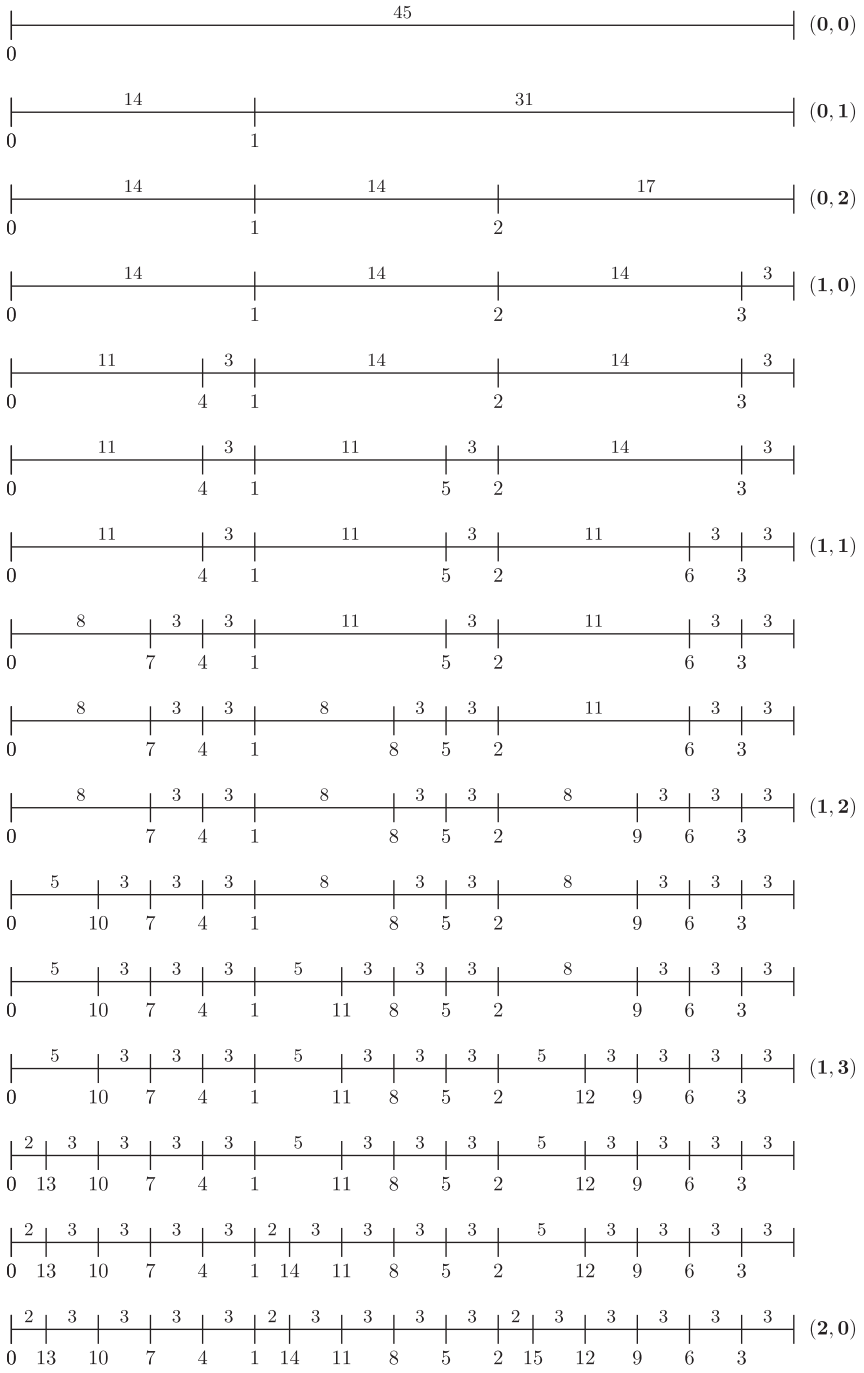


Fig. 3. Example of $\{a \cdot x \mod 1 \mid x < n\}$ configurations generated by $a = 14/45$. The unit segment is scaled by a factor of 45 for clarity. Each two-length configuration is labeled by its index (i, t) on the right. Each segment is labeled by its length above, and each multiple of a is labeled by its index below.

Table I. Values of θ_i, $\theta_{i-1,t}$, q_i, $q_{i-1,t}$ and k_{i+1} for Each Two-Length Configuration of the Example of Figure 3

i	t	$q_{i-1,t}$	q_i	$\theta_{i-1,t}$	θ_i	k_{i+1}
0	0	0	1	45	14	3
	1	1		31		
	2	2		17		
1	0	1	3	14	3	4
	1	4		11		
	2	7		8		
	3	10		5		
2	0	3	13	3	2	1
3	0	13	16	2	1	2
	1	29		1		
4	0	16	45	1	0	1

Note: As in Figure 3, the lengths θ_i and $\theta_{i-1,t}$ are scaled by a factor of 45.

PROPERTY 2 (DIRECTED REDUCTION [VAN RAVENSTEIN 1988]). *Given the two-length configuration* (i, t)*, when constructing the next two-length configuration, intervals of length* $\theta_{i-1,t}$ *are split into two intervals in this order from left to right:*

—*one of length* θ_i *and one of length* $\theta_{i-1,t+1}$ *if* i *is even,*
—*one of length* $\theta_{i-1,t+1}$ *and one of length* θ_i *if* i *is odd.*

4. HR-CASE SEARCH ON THE GPU

In this section, we describe two algorithms for HR-case search: Lefèvre [2005] HR-case search, as originally described, and our new HR-case search, which is more regular in the sense of reducing divergence. Both algorithms make use of Boolean tests that rely on the properties described in Section 3.2. Hence, we will describe both of them with continued fraction expansions, which give a uniform formalism to explain and compare their different behaviors. Then we will describe how they have been deployed on the GPU and the benefit on divergence provided by our new algorithm.

4.1. The Lefèvre HR-Case Search

Lefèvre [2005] presented an algorithm to search for (p, ϵ') HR-cases of a polynomial $P_i(x)$. This algorithm relies on a Boolean test on $Q_i(x)$ (the truncation of $P_i(x)$ to degree 1) that computes a lower bound of the set $\{b - a \cdot x \mod 1 \mid x < |D_i|_p\}$ and returns Failure if the inequality (1) holds and Failure or Success otherwise.

In Section 3.2, we described some properties of the configurations $\{a \cdot x \mod 1 \mid x < n\}$. According to these properties, computing the interval lengths of a subset of the two-length configurations can be done efficiently in $O(\log |D_i|_p)$ arithmetic operations by computing the continued fraction expansion of a [Brent and Zimmermann 2010]. However, if we use the continued fraction expansion, we will place more points than $|D_i|_p$ on the unit segment (at most $2 \cdot |D_i|_p$ if we use the subtraction-based Euclidean algorithm). To take advantage of the efficient construction of the two-length configurations, the Lefèvre HR-case search computes the minimum of $\{b - a \cdot x \mod 1 \mid x < n\}$ with n the number of multiples of a placed and $n \geq |D_i|_p$. This gives a lower bound on $\{b - a \cdot x \mod 1 \mid x < |D_i|_p\}$. Then the minimum of $\{\text{dist}_p(P_i(x)) < \epsilon' \mid x < |D_i|_p\}$ is exactly computed by exhaustive search in $O(|D_i|_p)$ arithmetic operations only if required. To minimize this exhaustive search, we use a filtering strategy in three phases as in Lefèvre [2005]:

ALGORITHM 1: Lefèvre Lower Bound Computation and Test Algorithm (Simplified).

```
   input: b − a · x, ε″, N
1  initialization: p ← {a}; q ← 1 − {a}; d ← {b};
                   u ← 1;   v ← 1;
2  if d < ε″ then return Failure;
3  ;
4  while True do
5  |  if d < p then
6  |  |  k ← ⌊q/p⌋;
7  |  |  q ← q − k ∗ p; u ← u + k ∗ v;
8  |  |  if u + v ≥ N then return Success;
9  |  |  ;
10 |  |  p ← p − q; v ← v + u;
11 |  else
12 |  |  d ← d − p;
13 |  |  if d < ε″ then return Failure;
14 |  |  ;
15 |  |  k ← ⌊p/q⌋;
16 |  |  p ← p − k ∗ q; v ← v + k ∗ u;
17 |  |  if u + v ≥ N then return Success;
18 |  |  ;
19 |  |  q ← q − p; u ← u + v;
```

—*Phase 1*: We compute a lower bound on $\{b - a \cdot x \mod 1 \mid x < |D_i|_p\}$ and test if this lower bound matches a (p, ϵ'') HR-case of Q_i. If not, there are no (p, ϵ') HR-cases for P_i in D_i. Otherwise, go to the next phase.

—*Phase 2*: We split D_i into subdomains $D_{i,j}$, we refine the approximation $Q_i(x)$ as $Q_{i,j}(x)$, and we compute a lower bound on $\{b_j - a_j \cdot x \mod 1 \mid x < |D_{i,j}|_p\}$ for each $D_{i,j}$. For each $D_{i,j}$ where the lower bound on $\{b_j - a_j \cdot x \mod 1 \mid x < |D_{i,j}|_p\}$ matches a (p, ϵ''_j) HR-case of $Q_{i,j}$, go to the next phase.

—*Phase 3*: We search exhaustively for (p, ϵ') of P_i in $D_{i,j}$ using the table difference method (see Section 5).

The cornerstone of the Lefèvre algorithm strategy is therefore the computation of the minimum of $\{b - a \cdot x \mod 1 \mid x < n\}$. In other words, it computes the distance between $\{b\}$ and the closest point to the left of $\{b\}$ in the configuration $\{a \cdot x \mod 1 \mid x < n\}$. We write N for the number of floating-point numbers in the considered subdomain ($n \geq N$ as we compute a lower bound). Depending on how we generate the two-length configurations (using the subtraction-based or the division-based Euclidean algorithm), we can derive from Property 2 two ways to compute this distance. The first one is the Lefèvre HR-case search, and the second one is the new HR-case search proposed in Section 4.2.

In the lower bound computation of the Lefèvre HR-case search, the way the two-length configurations are computed depends on the length of the interval containing $\{b\}$. When adding points in the interval containing $\{b\}$ and in the direction of $\{b\}$, Lefèvre uses a subtraction-based Euclidean algorithm (he moves from the (i, t) configuration to $(i, t + 1)$). Otherwise, he uses a division-based Euclidean algorithm (he moves from the (i, t) configuration to $(i + 1, 0)$). Algorithm 1 describes the lower bound computation of the Lefèvre HR-case search and the corresponding test with respect to ϵ'', which is the sum of all errors involved.

In this algorithm, the variables u and v count the number of intervals as in Equation (2) to exit when $n = u + v \geq N$, where u and v respectively store q_i and $q_{i-1,t}$ for i

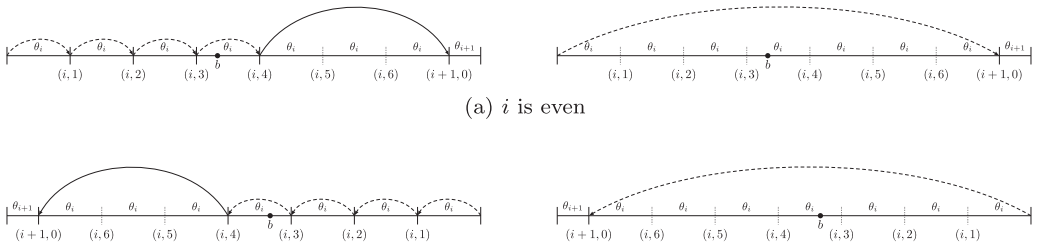


Fig. 4. Behavior of the Lefèvre (left) and the new (right) HR-case searches when b is in an interval of length θ_i (solid lines) and when b is in an interval of length $\theta_{i-1,t}$ (dashed lines). Each point is labeled by the index (i, t) of the two-length configuration in which it is added.

even and $q_{i-1,t}$ and q_i for i odd. The variables p and q respectively store the lengths θ_i and $\theta_{i-1,t}$ for i even and the lengths $\theta_{i-1,t}$ and θ_i for i odd. The variable d contains the distance between $\{b\}$ and the closest multiple $\{a \cdot x\}$ to its left.

Hereafter, we detail the relations between the two-length configurations and the execution paths of Algorithm 1, Lefèvre's lower bound computation and test. This algorithm starts with the configuration $(0, 1)$ and then considers the (i, t) configurations. Note that the condition at line 4 is false only if we have added one point directly to the left of $\{b\}$ during the previous loop iteration, and it is true otherwise. Hence, it has to be interpreted as "does d need to be updated?" This interpretation is allowed by the fact that the value of d at line 4 corresponds to the previous configuration (the configuration $(0, 0)$ at start) and that at least one point was already added (line 8 or 15). This condition enables us to handle the next four cases (illustrated in Figure 4):

—If i is even:
  —If $\{b\}$ is in an interval of length θ_i, then $d < \theta_i$ (this happens if the point previously added is just at the right of $\{b\}$). In this case, no point is added in the interval containing $\{b\}$, so we go directly to the configuration $(i + 1, 0)$ (lines 5 and 6) and $(i + 1, 1)$ (line 8).
  —If $\{b\}$ is in an interval of length $\theta_{i-1,t}$, then $d > \theta_i$ (this case happens if the point previously added is just at the left of $\{b\}$). In this case, d is updated by subtracting θ_i (line 10), $k = 0$ since $\theta_{i-1,t} > \theta_i$ (lines 12 and 13), and we go to configuration $(i, t+1)$ (line 15), as other points can be added to the left of $\{b\}$ in the next two-length configuration under Property 2.
—If i is odd:
  —If $\{b\}$ is in an interval of length θ_i, then $d > \theta_{i-1,t}$ (this case happens if the point previously added is just at the left of $\{b\}$). In this case, d is updated (line 10), and we go to the configuration $(i + 1, 0)$ (lines 12 and 13) and $(i + 1, 1)$ (line 15).
  —If $\{b\}$ is in an interval of length $\theta_{i-1,t}$, then $d < \theta_{i-1,t}$ (this case happens if the point previously added is just at the right of $\{b\}$). In this case, $k = 0$ since $\theta_{i-1,t} > \theta_i$ (lines 5 and 6), and we go to the configuration $(i, t + 1)$ (line 8), as a point can be added to the left of $\{b\}$ in the next two-length configuration according to Property 2.

Note that the Lefèvre algorithm always reduces d by using subtractions at line 10, as points are added one by one at the left of $\{b\}$. In practice, Lefèvre adds specific instructions to partly compute these reductions with divisions to avoid computing large quotients with subtractions. We have omitted these instructions here for clarity, but they are present in our implementations of the Lefèvre algorithm.

Furthermore, the algorithm computes divisions (lines 5 and 12). In practice, we can make use of different division implementations. We can apply a subtractive division, a

ALGORITHM 2: New Regular Lower Bound Computation and Test Algorithm.

```
   input: b − a·x, ε″, N
1  initialization: p ← {a}; q ← 1; d ← {b};
                   u ← 1;    v ← 0;
2  if d < ε″ then return Failure;
3  ;
4  while True do
5  |  if p < q then
6  |  |  k ← ⌊q/p⌋;
7  |  |  q ← q − k ∗ p; u ← u + k ∗ v;
8  |  |  d ← d mod p;
9  |  else
10 |  |  k ← ⌊p/q⌋;
11 |  |  p ← p − k ∗ q; v ← v + k ∗ u;
12 |  |  if d ≥ p then
13 |  |  |  d ← (d − p) mod q;
14 |  if u + v ≥ N then return d > ε″;
15 |  ;
```

division instruction, or combine both in an hybrid approach as presented and analyzed in Lefèvre [2005] and Fortin et al. [2012].

4.2. New Regular HR-Case Search

We now propose a new algorithm for the HR-case search where we use the same filtering and division strategy as in the Lefèvre algorithm, but we introduce a more regular algorithm—in the sense that it strongly reduces divergence on the GPU—to compute a lower bound on $\{b - a \cdot x \mod 1 \mid x < |D_i|_p\}$. Hereafter, we will refer to this new algorithm as the regular HR-case search.

The regular HR-case search is described in Algorithm 2. In it, we only consider configurations satisfying Equation (3) in order to use only the division-based Euclidean algorithm. The variables p and q respectively store the lengths θ_i and θ_{i-1} for i even and the lengths θ_{i-1} and θ_i for i odd. The variables u and v respectively store q_i and q_{i-1} for i even and q_{i-1} and q_i for i odd.

Thus, instead of testing if $\{b\}$ went from a split interval to an unsplit one like in the Lefèvre HR-case search, we test here which length is reduced, as in the classical Euclidean algorithm, and then we reduce it and update d accordingly (as illustrated in Figure 4). In practice, the quotients are computed like in the Lefèvre HR-case search with a subtractive division, a division instruction, or the hybrid approach. Now we detail the execution of Algorithm 2. Let $(i, 0)$ be a two-length configuration:

—If i is even, then the test $p < q$ is true since $\theta_i < \theta_{i-1}$, so we go to the configuration $(i + 1, 0)$ (lines 5 and 6), and:
  —If $\{b\}$ was in an interval of length θ_i, no point was added in the interval containing $\{b\}$ and d is not updated as $d < \theta_i$ and $d = d \mod \theta_i$ (line 7).
  —If $\{b\}$ was in an interval of length θ_{i-1}, points were potentially added to the left of $\{b\}$. Hence the distance d is updated by reduction modulo θ_i (line 7) since intervals are split from the left under Property 2.
—If i is odd, then the test $p < q$ (line 4) is false, so we go to the configuration $(i + 1, 0)$ (lines 9 and 10), and:
  —If $\{b\}$ was in an interval of length θ_i, no point was added in the interval containing $\{b\}$. However, we subtract θ_{i+1} from d if $d \geq \theta_{i+1}$ (line 12), which is similar to

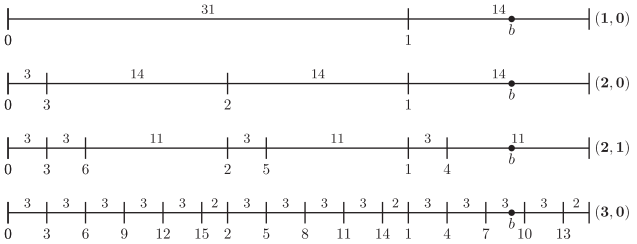


Fig. 5. Example where Algorithm 2 considers a configuration $(i + 1, 1)$ for $i = 1$.

considering the configuration $(i + 1, 1)$. Note that this would have been done in the next loop iteration at line 7 (Figure 5).

—If $\{b\}$ was in an interval of length θ_{i-1}, points were potentially added to the left of $\{b\}$. According to Property 2, intervals are split from the right. Then the distance d is updated if $d > \theta_{i+1}$ by reducing $d - \theta_{i+1} \mod \theta_i$ (lines 11 and 12).

Note that we do not test the divisors for division by zero in Algorithm 2, as it is very unlikely to occur: we only catch the division-by-zero exception and recover from it. The likelihood of a division by zero to happen is directly correlated to the probability of encountering a large quotient (which follows the Gauss-Kuzmin distribution [Khinchin 1997]) at a given iteration. For example, while searching for $(53, 2^{-32})$ HR-cases, the probability of having a division by zero using 64-bit arithmetic is much smaller than 2^{-64}. Until now, we have never encountered a division-by-zero in all of our tests.

4.3. Deployment on the GPU

The exhaustive search algorithm perfectly takes advantage of the GPU's massive parallelism and of its (partial) SIMD execution. Hence, we will focus on the deployment of the lower bound computation. In this section, we present the GPU deployment of the Lefèvre HR-case search as detailed in Fortin et al. [2012] and the GPU deployment of the new regular HR-case search. We particularly study the divergence in both HR-case searches at three levels: the filtering strategy, the main loop, and the main conditional statement. For these deployments, we first changed the data layout to a "structure of arrays" to have coalesced memory accesses (Section 6.2.1 in NVIDIA [2012a]). We also avoided (as much as possible) consecutive dependent instructions to increase the instruction-level parallelism within each thread. In addition, we mention that the computation of the continued fraction expansion is done using the Lehmer double-digit GCD algorithm [Brent and Zimmermann 2010] (we reduce the remainders by 32-bit chunks using 64-bit integers).

Throughout this section, we will consider the example domain $[1, 1 + 2^{-13})$ in the binade $[1, 2)$ for the exponential function in double precision, as this binade is considered by Lefèvre [2005] as the general case.

4.3.1. Filtering Strategy Divergence. As a consequence of the filtering strategy, we will have few threads executing phase 2 and fewer executing phase 3. Table II shows the number of subdomains involved in each phase for 2^{25} domains D_i containing 2^{15} floating-point numbers each. As we can see, very few subdomains are involved in the exhaustive search step (phase 3). Hence, executing one kernel computing the three phases leads to an important divergence, as we have fewer and fewer active threads within each warp from one phase to the next [Fortin et al. 2012].

Table II. Details of Argument Filtering During HR-Case Search in $[1, 1 + 2^{-13}]$

	Number of Arguments	
	Lefèvre	*Regular*
Phase 1	$2^{40} \approx 1.1 \cdot 10^{12}$	$2^{40} \approx 1.1 \cdot 10^{12}$
Phase 2	$\approx 3.6 \cdot 10^{9}$	$\approx 1.8 \cdot 10^{10}$
Phase 3	$\approx 8.9 \cdot 10^{6}$	$\approx 5.9 \cdot 10^{7}$
HR-Cases	243	243

To tackle this problem, we propose to use three kernels, one for each phase. This allows us to rebuild the grid of threads between each phase and to run the exact number of threads required by each phase. However, this implies two additional costs.

First, we have to write failing subdomains[1] of phases 1 and 2 into consecutive memory locations as we prepare coalesced reads for the next phase. In Fortin et al. [2012], this was done with atomic operations on the GPU global memory, since we had few failing subdomains. For some specific binades, however, the number of failing subdomains can be much greater and the numerous atomic operations result in lower performance. Hence, inspired by the algorithm of Sengupta et al. [2008], we have implemented optimized compact operations based on parallel prefix sums, which are specific to the binary values used. Our compact operations are as efficient as the atomic operations when we have few failing subdomains, and they are more efficient when there are many.

Second, between each two successive kernels, we have to transfer the number of failing subdomains back to the CPU to compute (on the CPU) a suitable CUDA grid size for the next kernel.

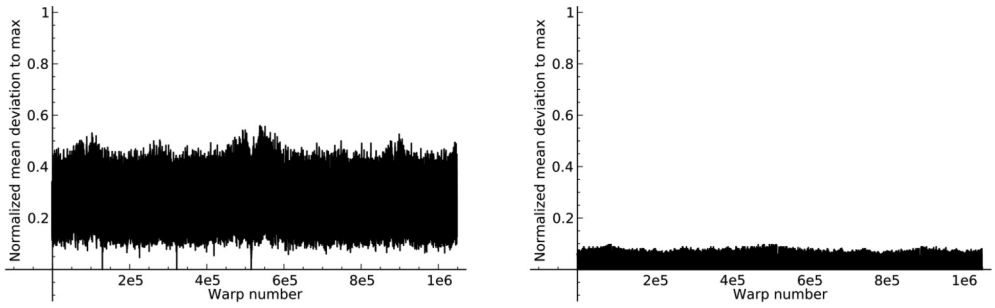
It can be noticed in Table II that the Lefèvre HR-case lower bound computation filters out a little more of the arguments than the new algorithm. The Lefèvre HR-case lower bound computation uses the subtraction-based Euclidean algorithm when splitting the interval containing $\{b\}$. This results in several considered arguments less than $2N$. On the contrary, we always use the division-based Euclidean algorithm in the regular HR-case search. If we consider i such that $q_i + q_{i-1} < N < q_{i+1} + q_i$, then $q_{i+1} + q_i = q_{i-1} + k_{i+1}q_i + q_i < (k_{i+1} + 1)N$ – by considering $q_i < N$. However, the geometric mean of the quotients k_i of the continued fraction of almost all real numbers equals Khinchin's constant (≈ 2.69) [Khinchin 1997]. Hence, using the regular HR-case search, we consider an average of less than $3.69 \cdot N$ arguments.

4.3.2. Loop Divergence. The second source of divergence is the main unconditional loop (see line 3 in Algorithms 1 and 2). Figure 6 shows the NMDM of the number of loop iterations by warp, for the different HR-case searches, when testing 2^{25} domains D_i containing 2^{15} double-precision floating-point arguments. Table III summarizes statistical information on the NMDM and the number of iterations for both the Lefèvre and regular HR-case searches.

For the Lefèvre HR-case search, the main unconditional loop is an important source of divergence with a mean NMDM of 25.6%. In other words, a thread remains idle on average 25.6% of the number of loop iterations executed by its warp. To the best of our knowledge, there is no a priori information on the number of loop iterations that would enable us to statically reorder the subdomains to decrease this divergence. We also tried to use software solutions to reduce the impact of the loop divergence [Fortin et al. 2012], although to no avail because the computation grain is very fine.

This divergence in the Lefèvre HR-case search is mainly due to the fact that the quotients are either entirely or partially computed at each iteration depending on

[1]Subdomains for which the computed lower bound is less than ϵ'' in Algorithms 1 and 2.



(a) Lefèvre HR-case search with specific instructions

(b) Regular HR-case search

Fig. 6. Normalized mean deviation to the maximum of the number of main loop iterations per warp among the 2^{20} warps required for the *exp* function in the domain $[1; 1 + 2^{-13}]$.

Table III. Comparison of the Main Loop Behavior Among the 2^{20} Warps Required for the Different HR-Case Searches on the *Exp* Function in the Domain $[1, 1 + 2^{-13}]$

HR-Case search	Min Iteration Number	Max Iteration Number	Mean Iteration Number	Mean NMDM
Lefèvre	5	328	24	25.6%
with specific instructions	5	31	16	25.7%
Regular	8	19	12	0.1%

the position of b, even with the specific instructions (see Section 4.1). Thanks to these specific instructions, the pathological cases are avoided (see Table III), but the mean NMDM is still around 25.6%.

In the new regular HR-case search, the key point is that a quotient of the continued fraction expansion of a is entirely computed at each loop iteration, which is not the case in the Lefèvre HR-case search. Hence, the number of loop iterations only depends on the number of quotients of the continued fraction expansion of a computed to reach $|D_i|_p$ points on the segment. As the number of quotients to compute is very close from one subdomain to the next, we reduce the mean NMDM by warp to 0.1%.

4.3.3. Branch Divergence. The third source of divergence is on the main conditional statement (see line 4 in Algorithms 1 and 2). We aim at reducing the number of instructions controlled by the branch condition, and if they are reduced enough, we benefit from the GPU branch predication (Section 9.2 in NVIDIA [2012a]). This branch predication enables the pipelines to be filled at best, for short sections of divergent code, by scheduling both *then* and *else* branches for all threads: thanks to a per-thread predicate, only the relevant results are actually computed and finally written.

As observed in Fortin et al. [2012], both branches of the Lefèvre HR-case search contain the same instructions, except the variables p (respectively, u) and q (respectively, v) are interchanged, and p is subtracted from d in the *else* branch. We therefore swap the two values p and q (respectively, u and v) to remove the common instructions from the conditional scope. This is described in Algorithm 3. The swap implies a small extra cost, but we thus reduce the number of divergent instructions.

As far as the new regular HR-case search is concerned, there is as much branch divergence within the unconditional loop in Algorithm 2 as there is in Algorithm 1. However, the main conditional statements of the two algorithms are rather different. In the Lefèvre HR-case search, this test (line 4) depends on the position of point b at each iteration. In the regular HR-case search, it depends on the length to reduce.

ALGORITHM 3: Lefèvre's Lower Bound Computation and Test Algorithm with Swap.

input: $b - a \cdot x$, ϵ'', N
initialization: $p \leftarrow \{a\}$; $q \leftarrow 1 - \{a\}$; $d \leftarrow \{b\}$; $u \leftarrow 1$; $v \leftarrow 1$; $are\_swapped \leftarrow False$;
if $d < \epsilon''$ **then return** Failure;
;
if $(d \geq p)$ **then**
  SWAP(p, q); SWAP(u, v);
  $are\_swapped \leftarrow True$;
while *True* **do**
  if $are\_swapped$ **then**
    $d \leftarrow d - p$;
    if $d < \epsilon''$ **then return** Failure;
    ;
  $k \leftarrow \lfloor q/p \rfloor$;
  $q \leftarrow q - k * p$; $u \leftarrow u + k * v$;
  if $u + v \geq N$ **then return** Success;
  ;
  $p \leftarrow p - q$; $v \leftarrow v + u$;
  if $are\_swapped$ *xor* $(d \geq p)$ **then**
    SWAP(p, q); SWAP(u, v);
    $are\_swapped \leftarrow not(are\_swapped)$;

ALGORITHM 4: New Regular Lower Bound Computation and Test Algorithm Unrolled.

input: $b - ax$, ϵ'', N
initialization: $p \leftarrow \{a\}$; $q \leftarrow 1$; $d \leftarrow \{b\}$; $u \leftarrow 1$; $v \leftarrow 0$;
while *True* **do**
  $k \leftarrow \lfloor q/p \rfloor$;
  $q \leftarrow q - k * p$; $u \leftarrow u + k * v$;
  $d \leftarrow d \mod p$;
  if $u + v \geq N$ **then return** $d > \epsilon''$;
  ;
  $k \leftarrow \lfloor p/q \rfloor$;
  $p \leftarrow p - k * q$; $v \leftarrow v + k * u$;
  if $d \geq p$ **then**
    $d \leftarrow d - p \mod q$;
  if $u + v \geq N$ **then return** $d > \epsilon''$;
  ;

Unlike the test on the position of b, the test on the length to reduce is deterministic, as the regular HR-case search computes a quotient of the continued fraction expansion of a at each loop iteration. Hence, the evaluation of the condition switches at each loop iteration, and it first evaluates to *True* as p is initialized to $\{a\}$ and q to 1. Therefore, by unrolling two loop iterations, we can avoid this test and strongly reduce the branch divergence. This is done in Algorithm 4.

5. POLYNOMIAL APPROXIMATION GENERATION ON THE GPU

In this section, we detail how we have deployed on the GPU the generation of the polynomial approximations P_i required for the HR-case search algorithms described in Section 4. We recall from Section 3.1 that the change of variable $x = 2^{p-e(X_i)}(X - X_i)$

enables testing of the floating-point arguments $X \in D_i$ of $f(X)$ by testing the integer arguments $x \in [\![0, |D_i|_p - 1]\!]$ of $P_i(x)$.

Although individual Taylor approximations could be used on each domain D_i, the number of domains makes this prohibitive. The principle here is therefore to consider the union of τ domains $D_{t\tau}, \ldots, D_{(t+1)\tau-1}$—denoted by $\mathcal{D}_t$—and to approximate the function f by a polynomial R_t of degree δ (e.g., with a Taylor approximation) such that $|R_t(x) - f(X)| < \epsilon'_{approx} 2^{e(f(X))-p}$ for all $X \in \mathcal{D}_t$ with $x = 2^{p-e(X_t)}(X - X_t)$.

If τ is chosen such that $e(x) = e(y)$ for all $x, y \in \mathcal{D}_t$, then $P_{t\tau+i}(x)$ is defined as an approximation of $R_t(x + iN)$ for $0 \leq i < \tau$ with $N = |D_t|_p$ (as $|D_i|_p = |D_j|_p$, for all $i, j \in [\![t\tau, (t+1)\tau - 1]\!]$). The shifts of the form $R_t(x + iN)$ are called *Taylor shifts* [von zur Gathen and Gerhard 1997]. If we denote by ϵ_{shift} the error propagated by the shift such that $|P_{t\tau+i}(x) - R_t(x + iN)| < \epsilon_{shift}$, then we set ϵ_{approx} to $\epsilon_{approx} = \epsilon'_{approx} + \epsilon_{shift}$ (see Section 3.1).

We now consider how to compute these polynomials $P_{t\tau+i}$. We first present the hierarchical method originally designed by Lefèvre [2000] to change one Taylor shift by N into several Taylor shifts by 1. Then, we present two existing Taylor shift algorithms:

—the *tabulated difference shift*, which, starting with $P_{t\tau}(x) = R_t(x)$, sequentially iterates a shift of the polynomial $P_{t\tau+i}$ to obtain $P_{t\tau+i+1}$ with only multiprecision additions [de Dinechin et al. 2011], and
—the *straightforward shift*, which computes the $P_{t\tau+i}$'s from R_t in parallel but requires multiprecision multiplications and additions [von zur Gathen and Gerhard 1997].

Finally, we propose a hybrid CPU-GPU Taylor shift algorithm that efficiently combines these two shifts with the hierarchical method and requires only fixed-size multiprecision additions on GPUs. More details on these algorithms and their error propagation can be found in de Dinechin et al. [2011].

5.1. Hierarchical Method

We first describe the hierarchical method originally described in Lefèvre [2000], which transforms one shift by N of a polynomial $R_t(x)$ of degree δ into $\delta + 1$ shifts by 1. This is of interest, as shifting by 1 can be done with only additions (see Section 5.2). This method requires the input polynomial to be interpolated in the binomial basis $\binom{x}{j} = \frac{\prod_{l=0}^{j-1}(x-l)}{j!}$. Therefore, we define the forward difference operator and its application to interpolate a polynomial in the binomial basis.

Definition 5.1. The forward difference operator, denoted by Δ_h, is defined as $\Delta_h[P](x) = P(x + h) - P(x)$. We write Δ_h^j for the composition of Δ_h j times and define $\Delta = \Delta_1$.

Using this forward difference operator, one can efficiently interpolate the polynomial R_t of degree δ in the binomial basis [de Dinechin et al. 2011], given the values $\{R_t(x), x \in [\![0, \delta]\!]\}$ as $R_t(x) = \sum_{j=0}^{\delta} \Delta^j[R_t](0) \cdot \binom{x}{j}$. An example is shown in Figure 7. This interpolation is computed using the definition of Δ and with initial values $\Delta^0[R_t](x) = R_t(x)$. This algorithm is a particular case of the Newton interpolation but with the forward difference operator used instead of the forward divided difference operator.

Now we describe the hierarchical method [Lefèvre 2000]. Given a polynomial $R_t(x)$, we want to build a scheme to shift this by polynomial in consecutive arguments following an arithmetic progression with common difference N. Let us consider the univariate polynomial R_t as a bivariate polynomial $R_t(x + iN)$ in the variables x and i. By interpolation in the binomial basis with respect to the variable x, we obtain a polynomial in

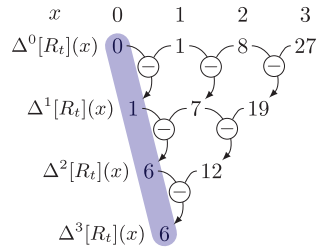


Fig. 7. Newton interpolation of polynomial x^3. The coefficients of the interpolated polynomial are highlighted.

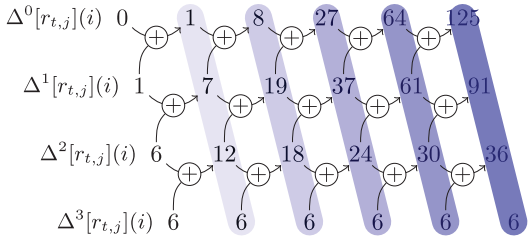


Fig. 8. Tabulated difference shift for evaluating the polynomial $r_{t,j}(i) = i^3$. The coefficients of the shifted polynomials are highlighted.

the variable x, with polynomial coefficients $r_{t,j}(i) = \Delta^j[R_t](iN)$ in the variable i, defined as follows:

$$R_t(x + iN) = \sum_{j=0}^{\delta} r_{t,j}(i)\binom{x}{j}.$$

Using the hierarchical method, we thus obtain the polynomials $r_{t,j}(i)$. From these, one can compute the shifts $P_{t\tau+i}(x)$ of the polynomial $R_t(x)$ by iN by evaluating $r_{t,j}(i)$ with $0 \leq j \leq \delta$. If we consider the polynomials $r_{t,j}$ in the binomial basis, these evaluations $r_{t,j}(i)$ can be obtained with the consecutive Taylor shifts of $r_{t,j}$ by 1—by taking the coefficients of degree 0 of these shifts, which are the $\Delta^0[r_{t,j}](i)$.

5.2. Taylor Shift Algorithms

Taylor shifts by 1 can be performed efficiently with the tabulated difference shift [de Dinechin et al. 2011; Lefèvre et al. 1998]. According to the forward difference operator definition, $\Delta^l[r_{t,j}](i) = \Delta^{l-1}[r_{t,j}](i+1) - \Delta^{l-1}[r_{t,j}](i)$—in other words, $\Delta^{l-1}[r_{t,j}](i+1) = \Delta^{l-1}[r_{t,j}](i) + \Delta^l[r_{t,j}](i)$. Furthermore, if $deg(r_{t,j}) = \gamma$, then $\Delta^\gamma[r_{t,j}](i)$ is constant for any integer $i \geq 0$, as it is the γ^{th} discrete derivative of $r_{t,j}$ times $\gamma!$. Hence, the only needed operations to obtain the consecutive evaluations of the polynomials $r_{t,j}$ are multiprecision additions of the coefficients. An example of this algorithm can be found in Figure 8.

Obtaining the consecutive evaluations of $r_{t,j}$ can also be performed with the straightforward shift. This algorithm multiplies the vector of the $r_{t,j}$ polynomial coefficients by a matrix constructed using Newton's binomial theorem. If we consider the polynomials $r_{t,j}$ expressed in the binomial basis, this multiplication exactly corresponds to applying the tabulated difference algorithm i times to the polynomial $r_{t,j}$. This matrix is upper triangular and Toeplitz, which can be used to speed up the matrix-vector multiplication

for high degree. It is constructed as

$$\begin{pmatrix} \binom{i}{0} & \binom{i}{1} & \cdots & \binom{i}{\gamma} \\ 0 & \binom{i}{0} & \cdots & \binom{i}{\gamma-1} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & \binom{i}{0} \end{pmatrix}.$$

Therefore, to construct this matrix, only the first $\binom{i}{l}$ with $0 \leq l < \gamma + 1$ are needed to compute its coefficients.

5.3. Hybrid CPU-GPU Deployment

Now we propose a hybrid CPU-GPU deployment of the polynomial approximation generation step. The polynomials R_t are Taylor polynomials of "high" degree δ approximating the targeted function over $\tau = 2^{25}$ domains D_i as in Lefèvre et al. [1998]. We interpolate them in the binomial basis using the hierarchical method (Section 5.1) with $N = 2^{15}$, as we want to use the Boolean tests described in Section 4 on intervals containing 2^{15} arguments. We choose these parameter values so that the error analysis of the Boolean test in Lefèvre [2000] applies. As this interpolation in the binomial basis is done only once, it is precomputed on the CPU. Then, we shift these polynomials and truncate them to degree 2 to make the domain D_i splitting at phase 2 of the HR-case search more efficient. More formally, we have

$$P_{t\tau+i}(x) = R_t(i2^{15} + x) = \sum_{j=0}^{2} r_{t,j}(i)\binom{x}{j} \mod \binom{x}{3}.$$

Hence, to obtain all polynomials $P_{t\tau+i}$ for $0 \leq i < \tau$, we have to deploy on the GPU the computation of the consecutive evaluations of $r_{t,j}(i)$ for $0 \leq i < \tau$ and $0 \leq j \leq 2$.

On the one hand, the tabulated difference shift is efficient, as it requires only multiprecision additions. This method is thus used in the reference CPU implementation [Lefèvre et al. 1998]. However, this is an intrinsically sequential algorithm, which prohibits its direct efficient deployment on the GPU. On the other hand, the straightforward shift is embarrassingly parallel, but it requires multiprecision multiplications and divisions to compute the binomial coefficients and multiprecision multiplications and additions to compute the matrix-vector products.

To benefit from the efficiency of the tabulated difference shift on the GPU, we therefore use a hybrid strategy that relies on both the CPU and the GPU: we compute the shifts $r_{t,j,u}(i) = r_{t,j}(u\nu + i)$ to form μ packets of size ν such that $\mu\nu = \tau$. We vary u from 0 to $\mu - 1$ and construct the polynomials $r_{t,j,u}$ sequentially on the CPU with the straightforward shift.[2] All multiprecision operations on the CPU are computed efficiently by using the GMP library [Granlund and the GMP Development Team 2010].

The μ polynomials $r_{t,j,u}$ are then transferred to the GPU. We run a CUDA kernel of μ threads wherein each thread of ID u processes the polynomial $r_{t,j,u}$ and computes the evaluations $r_{t,j,u}(i)$ with $0 \leq i < \nu$ using the tabulated difference shift.

Furthermore, as there are $\delta + 1$ independent $r_{t,j}$ polynomials, we can run one kernel per $r_{t,j}$ polynomial and overlap the GPU tabulated difference shift for the polynomial $r_{t,j}$ with the CPU straightforward shift of the polynomial $r_{t,j+1}$. The only algorithm deployed on the GPU is therefore the tabulated difference shift, which is sequential

[2]This computation on the CPU could thus be parallelized, but the corresponding computation times are sufficiently small in practice that this is not worthwhile.

within each GPU thread but performed concurrently by multiple threads on multiple polynomials $r_{t,j,u}$.

As the coefficients of the considered polynomials are large, we need multiprecision addition on the GPU. Here, only fixed-size multiprecision additions are required, as bounds on the required precision, depending on the targeted function and the exponent of the targeted domain, can be computed before compile time [Lefèvre 2000; de Dinechin et al. 2011]. Multiprecision libraries on the GPU [Nakayama and Takahashi 2011; Lu et al. 2010] have been developed. However, we preferred to have our own implementation of this operation for two main reasons: to use PTX (NVIDIA assembly language) [NVIDIA 2012b] and the *addc* instruction to have an efficient carry propagation, and to benefit from the fixed size of the multiprecision words at compile time to unroll inner loops. As the *addc* instruction operates only on 32-bit words, multiprecision words are arrays of 32-bit chunks. The multiprecision addition function is implemented as a C++ template with the size of the multiprecision words given as a parameter, which enables automatic generation of addition functions for each size of fixed multiprecision word required by each union of domains $\mathcal{D}_t$. As a consequence, the inner loop on the number of chunks can easily be unrolled as the number of loop iterations is known at compile time. Furthermore, to have coalesced memory accesses, the word chunks are interleaved in global memory and loaded chunk by chunk in registers.

Finally, note that this algorithm is completely regular: there is therefore no divergence issue among the GPU threads here.

6. PERFORMANCE RESULTS

In this section, we present the performance analysis of our different deployments. All results are obtained on a server composed of one Intel Xeon X5650 hex-core processor running at 2.67GHz, one NVIDIA Fermi C2070 GPU, and 48GB of DDR3 memory.

We compare three implementations. The first one is the sequential implementation (named *Seq.*), which is Lefèvre's reference optimized code. The second one is the parallel implementation on the CPU (referred to as *MPI*), which is the sequential implementation with an MPI layer added (OpenMPI version 1.6) to distribute the domains composing a binade equally among the available CPU cores. We made the choice of MPI over OpenMP because the processes do not communicate with each other (the computations over each domain are independent), and MPI favors data locality. In addition, we use a cyclic decomposition, which offers better load balancing than a block decomposition, and we run 12 MPI processes to take advantage of the two-way SMT (simultaneous multithreading or hyperthreading for Intel) of each core. The third implementation (named *CPU-GPU*) relies on the GPU and CPU-GPU deployments presented in this article. We searched for the optimal block sizes on the GPU and tried to increase the number of domains computed per thread in every GPU kernel to optimize occupancy and computation granularity. The implementations have been compiled with gcc-4.6.4 for the CPU code and nvcc (CUDA 5.0) for the GPU code. All of the following timings are obtained for the problem of searching $(53, 2^{-32})$ HR-cases of the *exp* function. In other words, binary64 FP arguments for which 32 extra bits of precision during the function evaluation do not suffice to guarantee correct rounding. The timings include all computations and data transfers between the GPU and the CPU.

6.1. Multicore Deployments

As a first step, in Table IV, we compare the sequential implementation with our multicore one. As sequential executions are very time consuming, we run this test only over the binade $[1, 2)$. All MPI deployments scale very well on the multicore CPU, as we have a parallel speedup higher than the number of cores. This is mainly because they take advantage of the two-way SMT execution, which can partly offset the high

Table IV. Timings Comparison (in seconds) of Sequential and Multicore Deployments in the Binade [1, 2)

	Seq.	MPI	$\frac{\text{Seq.}}{\text{MPI}}$
Pol. approx.	43,300.81	5,251.53	8.25
Lefèvre HR-case search	36,816.10	5,292.67	6.96
Regular HR-case search	34,039.94	4,716.97	7.22
Lef. /Reg.	1.08	1.12	

Table V. Timings (in seconds) for Binades $[2^k, 2^{k+1})$ with $k \in \{0, 4, 7\}$ for exp and $k = -1$ for sin

		exp			sin
		[1, 2)	[16, 32)	[128, 256)	[1/2, 1)
Pol. approx. generation	MPI	5,336.81	8,475.11	11,243.26	5,469.77
	CPU-GPU	785.14	1,619.72	1,612.03	1,065.011921
	speedup	6.74	5.23	6.97	5.14
Lefèvre	MPI	5,292.67	29,595.04	169,911.90	5,408.10
	CPU-GPU	2,446.78	14,009.58	57,021.67	2,758.05
	speedup	2.16	2.11	2.98	1.96
Regular	MPI	4,716.96	—	—	—
	CPU-GPU	711.8	7,766.16	47,356.01	895.37
	speedup*	7.44	3.81	3.59	6.04

Note: Timings labeled by "—" have been omitted because they are prohibitively long. An asterisk (*) denotes speedups over the MPI Lefèvre HR-case search.

latency operations. These high latencies are caused by the carry propagation during the multiprecision addition for the polynomial approximation generation and the division instructions for the HR-case search. We also note that the regular HR-case search is slightly faster than the Lefèvre one in both sequential (8%) and MPI (12%) executions. This is partly due to its regularity, as removing the main conditional statement avoids wrong branch predictions.

6.2. HR-Case Search on the GPU

As a second step, in Table V, we compare performance results of the HR-case search deployments on the GPU with the MPI ones. We first discuss the results for the *exp* function over the binades [1, 2), [16, 32), and [128, 256). The binade [1, 2) corresponds to the case where the exp function is well approximated by a polynomial of degree 1; the binade [128, 256) corresponds to the last entire binade before overflow, where the exp function is hard to approximate by a polynomial of degree 1; and the binade [16, 32) is an intermediate binade. We note here that to minimize the amount of exhaustive search and balance the time spent in phases 2 and 3, we suitably set the number of subdomains to consider in phase 2 for both HR-case search algorithms.

The deployment of the Lefèvre HR-case search on the GPU offers good speedups of about 2.1× over the hex-core CPU. The fact that this speedup is constant from one binade to the next is to be expected, as the algorithm is run with the same number of subdomains in phase 2. In fact, here we are limited on the GPU by the irregularity of the Lefèvre Boolean test. We cannot decrease the number of arguments to test in phase 3—which is well accelerated on the GPU—by increasing the number of subdomains, as it would lead to lower overall performance.

The deployment of the regular HR-case search on the GPU offers very good speedups over the MPI Lefèvre HR-case search: from 7.44× in the binade [1, 2) to 3.59× in the binade [128, 256). Here the speedup is greater in the binade [1, 2) than in the binade [128, 256).

Table VI. Details on Each Phase for the Lefèvre and Regular HR-Case Searches on the GPU

	Lefèvre		Regular	
	Arguments	Time (s)	Arguments	Time (s)
Phase 1	$9.01 \cdot 10^{15}$	2,372.60	$9.01 \cdot 10^{15}$	583.97
Phase 2	$3.19 \cdot 10^{13}$	61.31	$1.62 \cdot 10^{14}$	91.41
Phase 3	$7.65 \cdot 10^{10}$	11.02	$5.14 \cdot 10^{11}$	35.17

(a) $[1, 2)$

	Lefèvre		Regular	
	Arguments	Time (s)	Arguments	Time (s)
Phase 1	$9.01 \cdot 10^{15}$	2,750.58	$9.01 \cdot 10^{15}$	1,726.06
Phase 2	$8.97 \cdot 10^{15}$	21,162.87	$9.01 \cdot 10^{15}$	20,872.52
Phase 3	$9.55 \cdot 10^{14}$	33,106.43	$9.00 \cdot 10^{14}$	24,756.96

(b) $[128, 256)$

Table VII. Timings (in seconds) of the exp Function for Binades $[2^k, 2^{k+1})$ with $-1 \leq k \leq 7$ and of the Log Function for $[1, 3.17)$

Binade Exponent	log	–1	0	1	2	3	4	5	6	7
Pol. approx. gen.	1,293.15	1,029.84	785.14	910.58	1,240.63	1,239.36	1,619.72	2,282.22	2,891.29	2,961.42
Lef. HR-case search	13,015.16	3,287.35	2,446.78	3,104.59	3,901.89	6,768.26	14,009.58	20,175.37	31,484.65	57,021.67
Reg. HR-case search	12,566.92	1,060.72	711.8	1,061.39	1,777.57	3,691.28	7,766.16	16,359.27	29,124.52	47,356.01
Lef. / Reg.	1.04	3.10	3.44	2.93	2.20	1.83	1.80	1.23	1.08	1.20

In the binade $[1, 2)$, phase 1 is the most time consuming on a multicore CPU. Hence, we benefit from the regular behavior of the new HR-case Boolean test on the GPU, which results in a speedup of $7.44\times$ over the MPI deployment.

In the binade $[128, 256)$, phases 2 and 3 become the critical phases, as the Boolean test fails often. Moreover, the new regular HR-case search filters fewer arguments than the Lefèvre HR-case search, as stated in Section 4.3.1. This results in more arguments to test exhaustively, as detailed in Table VI(a). However, this can be partly offset by increasing the number of subdomains in phase 2 whose regular HR-case search is efficiently performed on the GPU. For example, we thus use 32 subdomains for the binade $[128, 256)$ instead of 8 for the Lefèvre HR-case search. In the end, we therefore manage to obtain a shorter exhaustive search (see Table VI(b)) and a good speedup of $3.59\times$.

We also tested our deployments on the sine function over the interval $[0, \pi)$, and the speedups were similar to those of the exp function. In Table V, we detail the timings over the binade $[1/2, 1)$ and obtain speedups similar to those of the exp function over the same binade: the regular HR-case search on the GPU offers a speedup of $6.04\times$ over the Lefèvre HR-case search on our hex-core CPU. These speedups thus seem to be related to the number of approximations and the approximation error. Therefore, we can expect very good speedups for functions that can be locally well approximated by degree 1 polynomials.

As a third step, in Table VII, we compare both GPU deployments on the positive domain of definition of the exp function. As the overflow occurs early in the binade $[256, 512)$, we do not show any timings for this binade, as they are not significant. In addition, instead of testing many binades near 0, we test about one binade and a half ($[1, 3.17)$) of the inverse function (the log function), as suggested by Lefèvre [2005].

The regular Boolean test enables varying speedups up to $3.44\times$ over the Lefèvre. This speedup decreases as we test arguments further away from 0. As already mentioned, this is due to the increasing truncation error on the tested degree 1 polynomials. The best approach here might be to consider a Boolean test that uses higher degree polynomials like those in the SLZ algorithm.

Finally, we finish with a qualitative remark: all tested implementations return the same HR-cases in the considered binades, and they are identical to those of Lefèvre [Muller et al. 2009] and de Dinechin et al. [2011], which strengthens the confidence in the validity of the computed hardness-to-round.

6.3. Polynomial Approximation Generation on the GPU

Now we detail the performance of the polynomial approximation generation. First, we recall that the cost of generating the Taylor approximations is negligible, although they are generated with Maple [Maplesoft 2016]. In Table V, we show performance results of the polynomial approximation generation step over three chosen binades. We can observe that the hybrid CPU-GPU deployment greatly takes advantage of the GPU, as all threads perform independent computations and the control flow is perfectly regular among the GPU threads. It thus offers good speedups up to 55.15× over the one CPU core execution and up to 6.97× over the six-core execution.

We can also observe in Table VII that the timings of the CPU-GPU polynomial approximation generation increase more slowly compared to the HR-case search ones, even though we use longer multiprecision words in the large binades (the maximum coefficient sizes vary from 320 bits for the binade $[1, 2)$ to 448 for the binade $[128, 256)$) and the highest polynomial degree ($\max_j(\deg r_{t,j}(x))$ vary from 6 for the binade $[1, 2)$ to 10 for the binade $[128, 256)$). This makes the relative amount of time spent in this step vary from 5.9% at least to 52.4% at most. Hence, when the function is well approximated by a degree 1 polynomial, the HR-case search is fast and requires about half of the global time. In the opposite situation, the HR-case search becomes by far the bottleneck, and the time spent in the polynomial approximation generation is negligible.

7. CONCLUSION AND FUTURE WORK

In this article, we have proposed a new algorithm based on continued fraction expansions for HR-case searches. This new algorithm improves the Lefèvre HR-case search algorithm by strongly reducing loop and branch divergence, which is a problem inherent to GPUs because of their partial SIMD architecture. We have also proposed an efficient GPU deployment of these two HR-case search algorithms and a hybrid CPU-GPU deployment of the generation of polynomial approximations.

When searching for HR-cases of the exp function in double precision, these deployments enable an overall speedup of up to 53.4× on one GPU over a sequential execution on one CPU core and a speedup of up to 7.1× on one GPU over one hex-core CPU.

In the future, we plan to investigate whether the regular HR-case search can benefit from other SIMD architectures like vector units (SSE, AVX, ...) on multicore CPU and Intel Xeon Phi architectures. This will require an OpenCL [Khronos Group 2011] implementation and an effective automatic vectorization by the OpenCL compiler.

We also plan to provide formal proofs of the deployed algorithms. This task is eased by the continued fraction expansion formalism and would enable a validated generation of the hardness-to-round, which is necessary to improve the confidence in the produced results. This is desirable before computing the hardness-to-round of all functions recommended by IEEE standard 754.

Finally, it is our hope to tackle the TMD for quadruple precision by deploying on the GPU the SLZ algorithm, which tests the existence of HR-cases with higher-degree polynomials. This algorithm heavily relies on the use of the LLL algorithm. The deployment of this algorithm on the GPU is therefore far from trivial if one wants to obtain good performance. Porting the LLL algorithm to the GPU will be the next step of this work.

ACKNOWLEDGMENT

The authors thank Vincent Lefèvre for helpful discussions and Polytech Paris-UPMC for the CPU-GPU server.

REFERENCES

Richard Brent and Paul Zimmermann. 2010. *Modern Computer Arithmetic*. Cambridge University Press.

Nicholas Brunie, Sylvain Collange, and Gregory Diamos. 2012. Simultaneous branch and warp interweaving for sustained GPU performance. In *Proceedings of the International Symposium on Computer Architecture*. 49–60.

Florent de Dinechin, Jean-Michel Muller, Bogdan Pasca, and Alexandru Plesco. 2011. An FPGA architecture for solving the Table Maker's Dilemma. In *Proceedings of the 22nd IEEE International Conference on Application-Specific Systems, Architectures, and Processors*. 187–194.

Pierre Fortin, Mourad Gouicem, and Stef Graillat. 2012. Towards solving the Table Maker's Dilemma on GPU. In *Proceedings of the 20th International Euromicro Conference on Parallel, Distributed, and Network-Based Processing*. 407–415.

Steffen Frey, Guido Reina, and Thomas Ertl. 2012. SIMT microscheduling: Reducing thread stalling in divergent iterative algorithms. In *Proceedings of the 20th International Euromicro Conference on Parallel, Distributed, and Network-Based Processing*. 399–406.

Aleksandr Ivanovich Galochkin. 2011. Lindemann theorem. *Encyclopedia of Mathematics*. Available at http://www.encyclopediaofmath.org/index.php?title=Lindemann_theorem&oldid=14026.

Torbjörn Granlund and the GMP Development Team. 2010. GNU MP: The GNU Multiple Precision Arithmetic Library. 5.0.1. http://gmplib.org/.

Tianyi David Han and Tarek S. Abdelrahman. 2011. Reducing branch divergence in GPU programs. In *Proceedings of the 4th Workshop on General Purpose Processing on Graphics Processing Units*. 3:1–3:8.

IEEE Computer Society. 2008. IEEE Standard for Floating-Point Arithmetic.

Aleksandr Yakovlevich Khinchin. 1997. *Continued Fractions*. Dover, Mineola, NY.

Khronos Group. 2011. *The OpenCL Specification Version 1.2*.

Vincent Lefèvre. 2000. *Moyens Arithmétiques Pour un Calcul Fiable*. Ph.D. Dissertation. École normale supérieure de Lyon, France.

Vincent Lefèvre. 2005. New results on the distance between a segment and $\mathbb{Z}^2$. Application to the Exact Rounding. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*. 68–75.

Vincent Lefèvre and Jean-Michel Muller. 2001. Worst cases for correct rounding of the elementary functions in double precision. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*. 111–118. DOI: http://dx.doi.org/10.1109/ARITH.2001.930110

Vincent Lefèvre, Jean-Michel Muller, and Arnaud Tisserand. 1998. Toward correctly rounded transcendentals. *IEEE Transactions on Computers* 47, 11, 1235–1243. DOI: http://dx.doi.org/10.1109/12.736435

Mian Lu, Bingsheng He, and Qiong Luo. 2010. Supporting extended precision on graphics processors. In *Proceedings of the 6th International Workshop on Data Management on New Hardware*. 19–26.

Maplesoft. 2016. *Maple User Manual*.

Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. 2009. Solving the Table Maker's Dilemma. In *Handbook of Floating-Point Arithmetic*. Birkhauser, Boston, MA, 405–460.

Takatoshi Nakayama and Daisuke Takahashi. 2011. 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*. 343–349.

Yuri Valentinovich Nesterenko and Michel Waldschmidt. 1996. On the approximation of the values of exponential function and logarithm by algebraic numbers. *Mat. Zapiski*, 23–42. English version available at http://arxiv.org/abs/math/0002047.

NVIDIA. 2011. *CUDA C Programming Guide, version 4.1*. NVIDIA.

NVIDIA. 2012a. *CUDA C Best Practices Guide, version 4.1*. NVIDIA.

NVIDIA. 2012b. *Parallel Thread Execution ISA Version 3.0*. NVIDIA.

Shubhabrata Sengupta, Mark Harris, and Michael Garland. 2008. *Efficient Parallel Scan Algorithms for GPUs*. Technical Report NVR-2008-003. NVIDIA.

Noel Bryan Slater. 1950. The distribution of the integers N for which $\{\theta N\} < \phi$. *Proceedings of the Cambridge Philosophical Society* 46, 525–534.

Noel Bryan Slater. 1967. Gaps and steps for the sequence $n\theta \mod 1$. *Proceedings of the Cambridge Philosophical Society* 73, 1115–1123.

Damien Stehlé, Vincent Lefèvre, and Paul Zimmermann. 2003. Worst cases and lattice reduction. In *Proceedings of the 16th IEEE Symposium on Computer Arithmetic*. 142–147.

Damien Stehlé, Vincent Lefèvre, and Paul Zimmermann. 2005. Searching worst cases of a one-variable function using lattice reduction. *IEEE Transactions on Computers*, 340–346. DOI:http://dx.doi.org/10.1109/TC.2005.55

Tony Van Ravenstein. 1988. The three gap theorem (Steinhaus conjecture). *Australian Mathematical Society* A, 45, 360–370.

Joachim von zur Gathen and Jürgen Gerhard. 1997. Fast algorithms for Taylor shifts and certain difference equations. In *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation*. 40–47.

Abraham Ziv. 1991. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software* 17, 3, 410–423.

Received July 2014; revised May 2016; accepted May 2016