Towards solving the Table Maker's Dilemma on GPU

Pierre Fortin, Mourad Gouicem, Stef Graillat
UPMC Univ Paris 06 and CNRS UMR 7606, LIP6,
4 place Jussieu, F-75252, Paris cedex 05, France
Contact: mourad.gouicem@lip6.fr

***Abstract*—Since 1985, the IEEE 754 standard defines formats, rounding modes and basic operations for floating-point arithmetic. In 2008 the standard has been extended, and recommendations have been added about the rounding of some elementary functions such as trigonometric functions (cosine, sine, tangent and their inverses), exponentials, and logarithms. However to guarantee the exact rounding of these functions one has to approximate them with a sufficient precision. Finding this precision is known as the *Table Maker's Dilemma*. To determine this precision, it is necessary to find the *hardest-to-round* argument of these functions. Lefèvre et al. proposed in 1998 an algorithm which improves the exhaustive search by computing a lower bound on the distance between a line segment and a grid. We present in this paper an analysis of this algorithm in order to deploy it efficiently on GPU. We manage to obtain a speedup of 15.4 on a NVIDIA Fermi GPU over one single high-end CPU core.**

I. Introduction

The IEEE 754 standard specifies the implementation of floating-point operations in order to have portable and predictable numerical softwares. It defines formats (half, single, double and quadruple precision), rounding modes (to the nearest and toward 0, $-\infty$ and $+\infty$) and operations $(+, -, \times, /, \sqrt{})$.

In 2008 this standard has been revised, recommending correct rounding of some transcendental functions, like *log*, *exp* and the trigonometric functions. One way to compute efficiently these functions is to approximate them by polynomials. However, it is hard to decide which precision is required to guarantee a correctly rounded result – the rounded evaluation of the approximation polynomial must be equal to the rounded evaluation of the function with infinite precision. This problem is known as the *Table Maker's Dilemma* or TMD. A common way to solve the TMD is to find the floating-point arguments of these functions which require high precision for correct rounding. These *hard-to-round* floating-point numbers are also named *HR-cases*. Knowing the *hardest-to-round* floating-point number, one can determine the precision required to approximate the targeted function and to ensure the correct rounding of all returned evaluations.

For general functions, the first improvement over the prohibitive exhaustive search of HR-cases was proposed by Lefèvre in [1]. The main idea of his algorithm is to split the domain into several intervals and to "isolate" HR-cases. This isolation is efficiently performed using local affine approximations of the targeted function. Stehlé, Lefèvre and Zimmermann extended this method in 2003 [2], [3] (SLZ algorithm) for higher degree approximations, using the Coppersmith method for finding small roots of univariate modular equation[1].

Both algorithms are very computationally intensive (several months for double precision on a single CPU), as well as highly parallel since the HR-case searches on each interval are independent, and since the number of such intervals is huge. The purpose of this work is therefore to accelerate these computations on Graphical Processing Units (GPUs), which theoretically perform one order of magnitude better than CPUs thanks to their massively parallel architecture. We focus here on Lefèvre's algorithm, which is simpler than the SLZ algorithm and efficient for double precision rounding. We consider its deployment on latest NVIDIA Fermi GPUs which offer improved computation performance for 64-bit integers. There has been some recent work on solving the TMD on FPGA [4], but to our knowledge, this is the first deployment of Lefèvre's algorithm on GPU.

In this paper, we will present Lefèvre's algorithm in Section II after a brief recall on rounding transcendental functions. Then in Section III, we will detail how we have partially deployed this algorithm (namely, the HR-case search) on one NVIDIA Fermi GPU by minimizing the number of inactive threads and the divergence. Finally, in Section IV we will present and analyze our performance results, and we will compare performance results on multiple GPUs and on multicore CPUs.

II. Presentation of Lefèvre's algorithm for searching HR-cases

A. Background on correctly rounding transcendental functions

We first present the scheme of transcendental function evaluation: more details can be found in [5, Chap. 11]. Let f be such a transcendental function defined over a set D, and $\circ_p$ a given rounding mode with p bits of precision. The function f being transcendental, we cannot compute $f(x)$ with infinite precision. Hence, one way to compute $f(x)$ is to approximate it by a polynomial P. This is done in two steps.

[1] http://www.loria.fr/equipes/spaces/slz.en.html

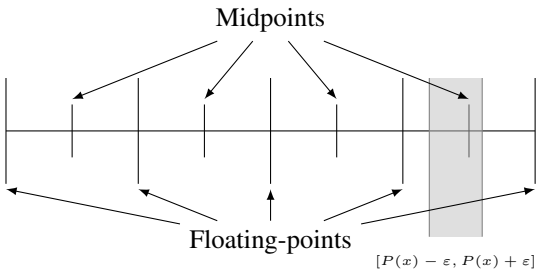

Figure 1. Hard-to-round case for rounding to nearest, where the rounding breakpoints are the midpoints of floating-point numbers.

- Find an interval I such that, for all $x \in D$ there exists $y \in I$ for which $f(x)$ can be "deduced" from $f(y)$.
- Compute a polynomial approximation P of f such that $\circ_p(f(y)) = \circ_p(P(y))$ with $y \in I$.

The first step, named *range reduction*, aims at reducing the domain of the function in order to have a better polynomial approximation. In fact, this step introduces some problems for functions which are hard to approximate by a polynomial for some arguments. In particular, this is the reason why we cannot find efficiently all the HR-cases for sin, cos and tan for large arguments yet. The second step guarantees that the evaluation of the polynomial P gives the correctly rounded evaluation of the function f. More formally, P is computed such that $|f(x) - P(x)| < \varepsilon$ for all $x \in I$. This means, when we compute $P(x)$ we do not obtain the value $f(x)$ but the center of an interval of length 2ε, containing $f(x)$.

Now let us call *rounding breakpoints* the values where $\circ_p$ changes and *hard-to-round case* an x such that the interval centered on $P(x)$ of length 2ε contains such a breakpoint. For these hard-to-round cases, we cannot round correctly as we do not know on which side of the breakpoint $f(x)$ is located (see Fig. 1). This problem is known as the Table Maker's Dilemma and is defined more formally in problem 1.

Problem 1 (Table Maker's Dilemma)**.** *For a given rounding mode $\circ_p$ and a function f defined over I, find an ϵ, if it exists, such that $\circ_p(f(x) - \epsilon) = \circ_p(f(x) + \epsilon)$ for all $x \in I$. The greater ϵ satisfying the TMD is called the* hardness-to-round *of f.*

However, such an ϵ may not exist if there exists x such that $f(x)$ equals a rounding breakpoint. Fortunately, for the mathematical elementary functions, these points are easily determined and can be treated separately. Furthermore, we can have a probabilistic estimation of the *hardness-to-round* for these functions. Let us consider f defined over $[a, b]$ with a and b two consecutive powers of 2 and a target precision p. As shown in [5, Chap. 12], if the output bits after the p^{th} digit are uniformly distributed, the expected *hardness-to-round* is around 2^{-2p} (which is a good estimation in practice).

Knowing such an ϵ, we can approximate f by P such that P satisfies $|f(x) - P(x)| < \varepsilon$ with ε greater than ϵ plus the error of the polynomial evaluation. Though, as ε increases, the degree and the size of the coefficients of P decrease. Since the complexity of the evaluation of a degree n polynomial directly depends on these two parameters ($O(n)$ arithmetic operations with Horner method), finding the *hardness-to-round* of f enables us to minimize the degree and the size of the coefficients of P, and hence to have an efficient evaluation of P. This is why we search the *hardness-to-round* of these functions.

B. Description of the algorithm

A common way to find the *hardness-to-round* of an elementary function is to search for HR-cases for a given sufficiently small ε and to find among them the *hardest-to-round*. If the considered function takes precision-p floating-point numbers as arguments, we will use an approximation error of $2^{-(p+p')}$, where p' is the extension of precision. As the expected *hardness-to-round* is around 2^{-2p}, we choose p' between 0 and p. A small p' implies a fast HR-case search but outputs many HR-cases. A p' close to p implies fewer HR-cases at output but a more costly HR-case search.

A naive algorithm to find the HR-cases is the exhaustive search. If we want to exhaustively test an unary function with precision-p floating-point numbers, we will have to evaluate a univariate polynomial 2^p times. The most efficient algorithm to evaluate a polynomial for arguments in arithmetic progression is the difference table method [4]. The bit complexity of such an algorithm would be $O(2^p \cdot n \cdot A(p+p'))$ with n the degree of the approximation and $A(p+p')$ the cost of an addition of two numbers of $p+p'$ bits. The computation time of this algorithm is therefore prohibitive for binary64 format, and intractable for binary128.

V. Lefèvre presented in [6] an improved algorithm to find the HR-cases of an elementary function. The main idea of his algorithm is to build local affine approximations in order to use several "filtering" phases on sub-intervals to "isolate" HR-cases. These filters compute a lower bound on the distance between a regular grid and a line. This way, we can exhaustively search the HR-cases in few small intervals. We will now describe the two major steps of this algorithm. To simplify notations, we consider ε a variable encompassing all computation and approximation errors.

The first step of his algorithm is the *generation of affine approximations*. Computing accurate affine approximations is not possible for large intervals as the degree of the approximation grows with the size of the interval. To build accurate affine approximations, we have to split the targeted interval I into smaller intervals J, containing $|J|$ floating-point numbers. Then, for each of these "sufficiently" small intervals J, we compute an affine approximation polynomial P_J. This can be achieved efficiently by first generating the Taylor expansion of the function f for a given precision over I. By this mean, we obtain a polynomial P such as $|f(x) - P(x)| < \varepsilon$ for all $x \in I$. Then by using a hierarchical algorithm based on the difference table method [1], we can generate P_J from P such that $|f(x) - P_J(x)| < \varepsilon$ for all $x \in J$ and such that the degree of P_J is less than the degree of P.

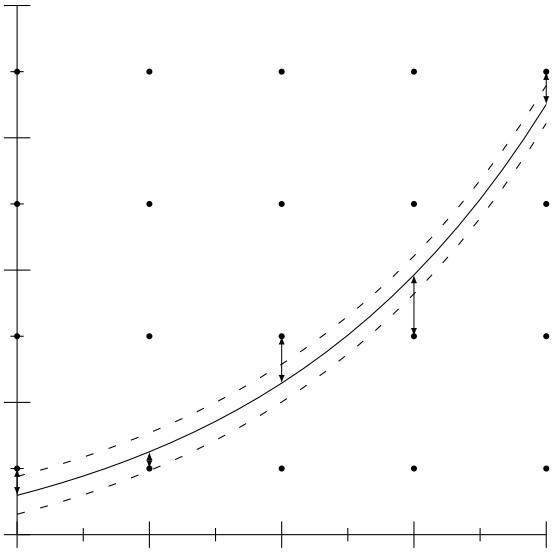

Figure 2. Distances between a curve and the breakpoint grid. The solid line is the curve $P(x)$ and the dashed ones are the curves $P(x)+\varepsilon$ and $P(x)-\varepsilon$.

The second step is the *HR-case search* in each of the J intervals. Before presenting this step, it has to be noticed that the TMD can be described as computing the distances between the regular grid of breakpoints and a curve defined by $P_J(x)$ as presented in Fig. 2. The grid is defined by points (x, y) such that x is a floating-point number in J and y a breakpoint. In order to have the grid regular in ordinate, we consider d the smallest distance between two breakpoints in the codomain of the polynomial P_J and add points accordingly. Finding the HR-cases in J then becomes similar to find all $x \in J$ and $y \in \mathbb{Z}$ satisfying

$$|P_J(x) - dy| < \varepsilon.$$

One straightforward method to find these x is the exhaustive search. If we write $|J|$ the number of floating-points in J, the exhaustive search can be performed in $O(|J|)$ arithmetic operations, which is prohibitive as $|J|$ grows exponentially with the targeted precision.

To minimize this exhaustive search, Lefèvre [6] uses a test to isolate HR-cases by computing a lower bound on the distance between the curve defined by $P_J(x)$ and the grid. If the degree of P_J is one, computing a lower bound can indeed be done efficiently in $O(\log|J|)$ arithmetic operations [7]. If the lower bound is greater than ε, there is no HR-case in J. Else, as the lower bound can be reached, there may exist an HR-case in J.

Based on this test, Lefèvre specified a filtering strategy to minimize the exhaustive search. For each given J we use the three following phases.

- Phase 1: we compute a first lower bound in J in $O(\log|J|)$ operations.
- Phase 2: if the lower bound computed on J is less than ε, we refine the affine approximation and split J into eight intervals K_i. Then, for each of these K_i, we compute a refined lower bound in $O(\log|K_i|)$ operations.
- Phase 3: for each K_i whose refined lower bound is less than ε, we search exhaustively for HR-cases in $O(|K_i|)$ operations.

C. Computing a lower bound on the distance between a line segment and a regular grid

Algorithm 1: Lower bound computation and test algorithm with subtractive division.

```
   input : P(x) = ax + b, ε, N
1  initialisation: x ← a;  y ← 1;  d ← b;
                   u ← 1;  v ← 1;
2  if d < ε then return Failure;
3  while True do
4      if d < x then
5          while x < y do
6              if u + v ≥ N then return Success;
7              y ← y − x; u ← u + v;
8          if u + v ≥ N then return Success;
9          x ← x − y; v ← v + u;
10     else
11         d ← d − x;
12         if d < ε then return Failure;
13         while y < x do
14             if u + v ≥ N then return Success;
15             x ← x − y; v ← u + v;
16         if u + v ≥ N then return Success;
17         y ← y − x; u ← u + v;
```

Now, we present in algorithm 1 the lower bound computation and test algorithm presented by Lefèvre in [8] and improved in [7]. Let $ax+b$ be an approximation of P of precision ε on an interval $[0, N-1]$ containing N floating-points. If we consider the floating-points $\{k{\cdot}a \mod d \mid 0 \leq k < N\}$ on a line segment of length d, these points partition the line segment into N intervals. For any $0 \leq k < N$, these intervals have at most three different lengths, which is known as the three-distance theorem [9]. There exists some configurations where the intervals have two lengths x and y. Lefèvre algorithm is based on computing these configurations with two possible lengths. The main idea of the algorithm is then, to go through these configurations and locate b to the closest point. If the interval containing b is split, we change the value of d which represent the distance between b and the closest point. Else, we continue splitting the segment. The variable u (respectively v) counts the number of intervals of size x (resp. y): we stop when $u + v > N$.

Being similar to the Euclidean greatest common divisor algorithm, we have to compute remainders (while loops at lines 5 and 13) at each iteration. In practice, we can make use of different division implementations to compute these remainders. We can apply a subtractive division, a division instruction, or combine both in an hybrid approach as presented in [7].

Let write C_{div} (resp. C_{sub}) the cost of the division instruction (resp. the subtraction instruction). The subtractive division cost is $q{\cdot}C_{sub}$ with q the computed quotient. The cost of the division instruction is constant and equals C_{div}. Then, if the computed quotient is less than $\frac{C_{div}}{C_{sub}}$, subtractive division is more efficient than division instruction, else division instruction is more efficient than subtractive division.

Lefèvre's hybrid division consists in choosing the best division implementation each time we compute a quotient. As we cannot use the quotient itself as a criteria, it uses the expected size of the quotient. If we divide a by b, the size of the expected quotient can be estimated by the difference of size between a and b. Let k be a threshold on the size of the expected quotient. If $a > 2^k b$ (namely, a is rather big compared to b) we use the division instruction, else we use the subtractive division. Setting the threshold k to a relevant value directly depends on the architecture and is determined by extensive testings.

Finally, it has to be noticed that in practice Lefèvre adds specific computations for special instances where early partial quotients are large. We have omitted them here for clarity but they are present in our implementations on GPU.

III. Deployment on GPU

Graphical Processing Units (GPUs) are many-core devices originally intended to graphical computation. However, they recently became general purpose devices especially with CUDA [10] and OpenCL [11]. We use here NVIDIA GPUs along with CUDA version 4.0 [10] and target NVIDIA Fermi GPUs with improved 64-bit integer arithmetic performance and cache support.

From a hardware point of view, a GPU is composed of several *multi-processors*, each being a SIMD unit. These multi-processors execute *threads* by groups of 32, which are called *warps* in CUDA. From a software point of view, the threads are organized by *block*, and a group of blocks forms a *grid*. At execution time, all the threads of a given grid run the same program, namely a *kernel*. The threads are assigned to a multi-processor by block, and are executed by warps.

In this paper, we only deploy on GPU the HR-case search step for double precision in each interval I (containing 2^{40} floating-point numbers divided in 2^{25} independent J intervals of 2^{15} arguments in practice). The generation of affine approximations has not been deployed as multi-precision arithmetic is needed. To our knowledge, no efficient multi-precision arithmetic library is indeed available on GPU at the time of writing this paper. Hence, we compute affine approximations on the host CPU, we transfer the required coefficients to the GPU device via the PCI bus, and we search HR-cases on GPU. In the following, we will thus focus only on the computation time of the HR-case search on GPU. Indeed we plan to implement in the future the generation of affine approximations on GPU: the data volume transferred on the PCI bus will then be very low and the transfer time negligible.

A. HR-case search on GPU

The exhaustive search algorithm perfectly takes advantage of the GPU massive parallelism and of its (partial) SIMD execution. However, Lefèvre's algorithm is faster than the exhaustive search by several orders of magnitude in practice on CPU. We have therefore decided to deploy Lefèvre's algorithm on GPU, and we present here two possible deployments. For each deployment, we changed the data layout to a "structure of arrays" in order to have coalesced memory accesses [12, Sect. 3.2.1]. We also avoided as much as possible consecutive dependent instructions in order to increase the instruction-level parallelism within each thread.

A first straightforward way to deploy Lefèvre's algorithm on GPU is to build one kernel where each thread computes the three phases for one interval J (*one kernel* deployment). This kernel is executed over a grid of 2^{25} threads. As these phases are filtering phases, we will have few threads executing phase 2, and fewer executing phase 3. Table I shows the number of intervals involved in each phase for an interval I containing 2^{40} floating-point numbers. As we can see, very few intervals lead to the exhaustive search step. Consequently, depending on the distribution of the intervals involved in phases 2 and 3, we can have very few active threads within each warp.

Phase	Number of intervals	Per thousand
1	$2^{25} \approx 33.55 \cdot 10^6$	1000‰
2	109048	3.25‰
3	2182	0.07‰
HR-cases	243	0.007‰

Table I
Details of interval filtering through Lefèvre's algorithm, for *exp* in $[1, 1 + 2^{-13}]$.

To tackle this problem, we propose another deployment where we use three CUDA kernels, one for each phase (*three kernel* deployment). This allows us to re-build 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 intervals[2] of phase 1 and 2 in consecutive memory locations as we prepare coalesced reads for the next phase. As we have very few HR-cases, this is done with atomic operations on the GPU global memory and not with parallel prefix sum [13]. Second, between two phases, we have to transfer back to CPU the number of failing intervals to compute on CPU the optimal grid size for the next phase. This optimal grid size enables to minimize the number of useless threads running in the next phase.

Moreover, for these two possible deployments, we have very fine grain computations. We have therefore introduced the possibility of having each thread computing more than one interval J to increase the computation grain. We have also considered a double buffering technique in order to overlap the memory accesses with computations between two consecutive J intervals.

[2]Intervals for which the computed lower bound is less than ε.

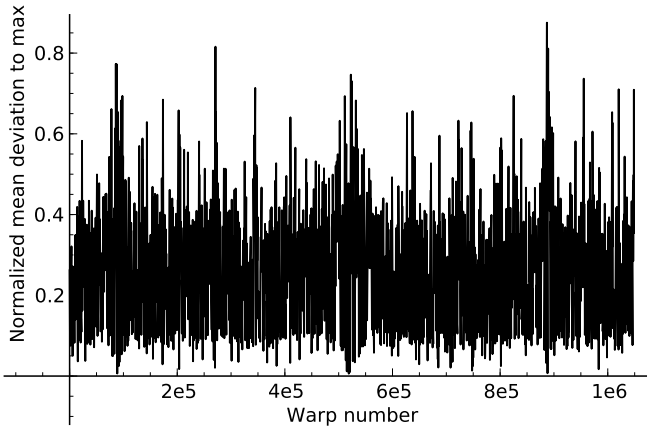

Figure 3. Normalized mean deviation to the maximum of the number of main loop iterations per warp of 32 threads for the *exp* function in the interval $[1; 1 + 2^{-13}]$.

B. Divergence minimization

GPU multi-processors are SIMD units. It is therefore more efficient to have as much as possible all the threads of a warp following the same execution path. When the threads of a warp follow different execution paths, we have a *divergent* warp [12, Sect. 6.1]. This happens when the threads within a warp do not evaluate a conditional instruction to the same value. For an *if* statement, *then* and *else* branches are in this case serially executed. For a *while* loop, a thread exiting the loop has to wait until all the threads of the warp exit the loop.

Contrary to the exhaustive search of phase 3 which is perfectly SIMD, the lower bound test in phases 1 and 2 presents two sources of divergence that we will now detail.

1) Divergence on the main loop: The lower bound test (see algorithm 1) presents an unconditional main loop which is executed in parallel by many threads, each thread treating one J or K_i interval. This introduces divergence as we can have two (or more) threads running a different number of iterations in the same warp. Figure 3 presents the normalized mean deviation to the maximum number of iteration of this main loop among the 32 threads of each warp, that is to say :

$$1 - \frac{\sum_{i=1}^{32} \frac{n_i}{32}}{\max\limits_{1 \leq i \leq 32}(n_i)},$$

where n_i is the number of main loop iterations for the thread number i. The divergence on the main loop varies thus greatly from one warp to another and can be very important.

A way to minimize this divergence is to re-organize the data. Unfortunately, we have no *a priori* information that would enable us to estimate the number of loop iterations. However, when considering one thread computing several intervals, we can make each thread load the next interval coefficients and continue looping, instead of exiting and waiting for the other threads of its warp to finish their current interval.

But there are two side effects to this method. First, the reads in GPU global memory are no more coalesced. Indeed, all the threads do not read concurrently their next interval J coefficients as they do not finish at the same time treating their current interval J. However, as Fermi GPUs have one L1 cache, global memory reads results in the load of the full 128-byte cache line. This way, the extra cost of the non-coalesced accesses may be offset by the L1 cache. Second, we add divergence within the loop as we replace a loop exit instruction by load instructions for the next interval J.

2) Divergence within the main loop: The second source of divergence is on the main conditional statement on the value d (see line 4 in algorithm 1). We aim at reducing the number of instructions controlled by the branch condition, and, if reduced enough, benefit from the GPU branch predication [12, Sect. 6.2].

By looking carefully at the content of each branch, we can notice that they contain the same instructions, except that the variables x (respectively u) and y (resp. v) are interchanged, and that x is subtracted to b. We therefore swap the two values x and y (resp. u and v) to remove the common instructions from the conditional scope as described in algorithm 2. The swap implies a small extra cost but we thus reduce the portion of divergent code, and hence the number of divergent instructions.

Algorithm 2: Lefèvre's algorithm with swap.

```
   input : P(x) = ax + b, ε, N
1  initialisation:
     x ← a;   y ← 1;   d ← b;
     u ← 1;   v ← 1;   are_swapped ← false;
2  if d < ε then return Failure;
3  if (d ≥ x) then
4  |  SWAP(x, y); SWAP(u, v);
5  |  are_swapped ← true;
6  while True do
7  |  if are_swapped then
8  |  |  d ← d − x;
9  |  |  if d < ε then return Failure;
10 |  while x < y do
11 |  |  if u + v ≥ N then return Success;
12 |  |  y ← y − x; u ← u + v;
13 |  if u + v ≥ N then return Success;
14 |  x ← x − y; v ← v + u;
15 |  if are_swapped xor (d ≥ x) then
16 |  |  SWAP(x, y); SWAP(u, v);
17 |  |  are_swapped ← not(are_swapped);
```

To minimize the extra cost of the swap, we swap the values only when this is required, that is to say we swap the values only if the evaluation of the condition $d < x$ changes at line 15 of algorithm 2. This enables us to minimize the number of swap operations. In practice, each swap is performed thanks to an auxiliary variable.

IV. Performance results

We now present performance results obtained on a server composed of two high-end hex-core Intel Xeon X5650 processors (twelve cores in total) running at 2.67 GHz, two high-end NVIDIA Fermi C2070 GPUs and 48 GB of DDR3 memory.

Implementations have been compiled with gcc-4.4.5 for CPU code and nvcc (CUDA 4.0) for GPU code. The reference CPU code is the one provided by V. Lefèvre.

All the results given in this section are issued from the HR-case search on the *exp* function for double precision. The extension of precision used is $p' = 32$. Some tests are performed over the I interval $I_0 = [1; 1 + 2^{-13}]$ and some over the 1024 I intervals $I_{0..1023} = [1; 1 + 2^{-3}]$.

A. HR-case search on GPU

We first want to find which division version is the fastest one on GPU among the subtractive division, the hybrid division or the division instruction presented in Section II-C. We varied the parameter k, refering to the threshold on the size of the quotient, from 0 (division instruction) to 64 (subtractive division as we use 64-bit integers). Since all of the computed quotients except the first one have very small values (most of the time 0, 1 or 2), we expect to compute a lot of subtractive divisions.

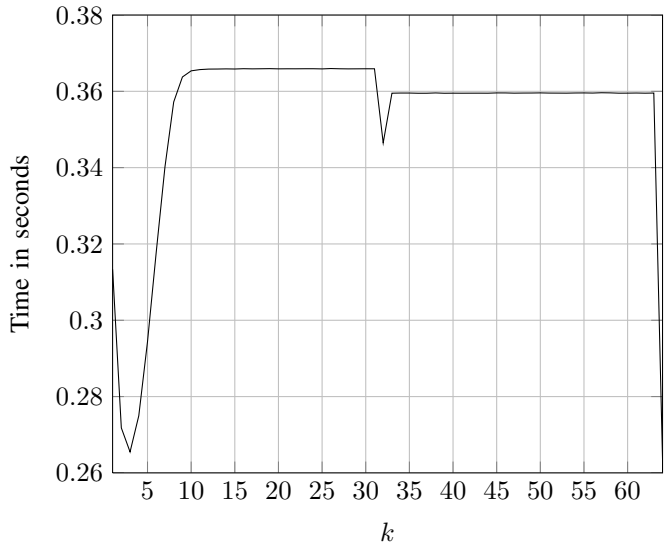

Figure 4. Execution time of HR-case search in the interval I_0 according to k threshold on a C2070 GPU.

As shown in Fig. 4, on a C2070 GPU with the *three kernel deployment*, the optimal k is 3. This result is similar to the one observed by Lefèvre [7], which had an optimal k of 3 on CPU. We also run the same test on our X5650 CPU and obtain an optimal k of 3. This means that the ratio $\frac{C_{div}}{C_{sub}}$ mentioned in Section II-C is the same on CPU and on GPU. We therefore use the hybrid division with $k = 3$ in all the following testings.

After finding the best fitted way to compute remainders on GPU, we focus on deploying the three phases strategy of the HR-case search step.

Version			Registers	Time (s)
Exhaustive	CPU (1 core)		-	1079.41
	GPU		21	33.23
Lefèvre's algorithm	CPU (1 core)		-	4.52
	One kernel		31	0.348
	Three kernels	Phase 1	30	0.258
		Phase 2	37	0.006
		Phase 3	20	0.001
		Total	-	0.265

Table II
Times and register consumptions per thread for searching HR-cases in the interval I_0

In Table II, we present the register consumptions and timings for different CPU implementations and GPU deployments for I_0. As expected, the exhaustive search is more efficiently performed on GPU, and we obtain a speedup of 32.5 over one CPU core. However, Lefèvre's algorithm on CPU remains more efficient than exhaustive search on GPU, which justifies its deployment on GPU. With the *one kernel deployment* we obtain a speedup of 13.0 over one CPU core for the HR-case search, whereas the *three kernel deployment* offers a speedup of 17.1. To measure the overhead implied by the use of atomic operations, we replaced atomic operations by non-atomic operations : this did not decrease the computation time mainly because the atomic operations are infrequent in this computation. The extra cost introduced by the atomic operations is thus negligible, as well as those introduced by the additional CPU-GPU transfers, the computation on CPU of the next grid size, and the launching of a new kernel. Besides, trying to modify Lefèvre's strategy in order to balance the computation time of the three phases (for example by changing the number of K_i intervals or by increasing the exhaustive search part) did not improved the performance. We therefore use the *three kernel deployment* in our following tests.

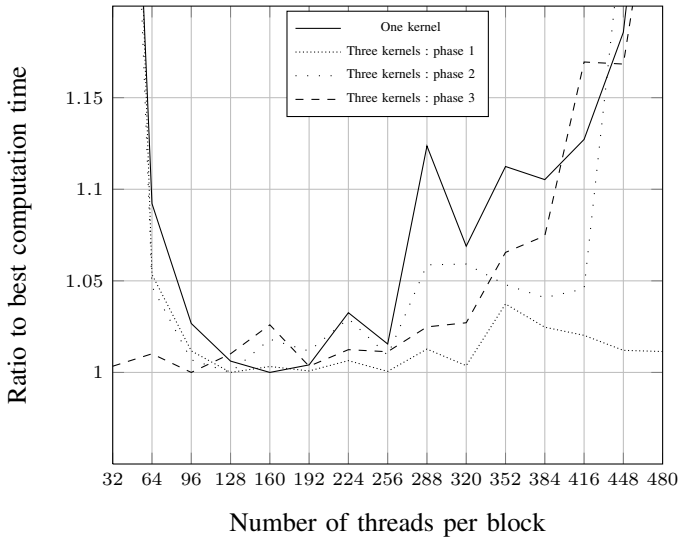

Figure 5. Overhead for the computation time of each kernel for I_0 with respect to the optimal block size.

Moreover, we observe in Table II that each phase of the *three kernel deployment* does not have the same register consumption. This difference implies that each kernel may have a different optimal number of threads per block. Figure 5 shows for each phase the overhead induced by non-optimal block sizes. We observe that the three kernels have different optimal block sizes. With the *three kernel deployment* we can therefore optimize the block size of each phase, which explains, along with the higher number of active threads per warp, why this deployment outperforms the *one kernel deployment*.

We also tried to increase the number of J intervals per thread. The best number of J intervals per thread is 6 for phase 1, but the performance gain is only 0.38% since we have a good occupancy (higher than 50%, see [12, Sect. 4.1]). Likewise, the double-buffering technique did not improved the performance, since the memory accesses are already overlapped with computation thanks to the good occupancy.

B. Divergence minimization

We tried to reduce the divergence on the main loop of the lower bound test as described in Section III-B1. The HR-case search then requires 1.504 seconds for I_0, which is only 3.2 times faster than CPU. This loss in performance may be explained by two reasons. First, the C2070 cache seems to be too small to offset the non-coalesced memory accesses. Second, the overhead of loading the next interval coefficients in the loop is greater than the benefit of reducing the divergence on the main loop.

As explained in Section III-B2, we also tried to decrease the number of divergent instructions executed within the main loop by swapping values. This way, the HR-case search requires 0.249 seconds for I_0, which represents a speedup of 18.2 over one CPU core (corresponding to a 6.2% improvement). It has to be noticed that over $I_{0..1023}$ the swap offers a maximum gain of 9.8% and an average gain of 6.0%. Such varying gains are directly due to the number of required swaps in each interval. We also tried storing variables in arrays and swapping the indexes or swapping pointers instead of the variables. Even if they save instructions, they did not improved performance since indirections which cannot be statically determined by the compiler are inefficient on GPU.

C. Comparison with multi-core CPU

We now compare our best GPU version (three kernels with swap) on multiple GPUs and on multi-core CPUs for the 1024 I intervals $I_{0..1023}$. We use the MPI standard with OpenMPI version 1.4.3. For the tests with multiple CPU cores, we distribute equally the 1024 intervals among the available CPU cores thanks to a cyclic decomposition which offers a better load balancing than a block decomposition. We do not bind processes with MPI, nor to core, neither to socket, since these bindings do not improve performance. For the tests with two GPUs, we use the same cyclic decomposition as for two cores. Figure 6 shows the performance results of these tests. The code scales very good on the multi-core CPUs. We obtain for example a speedup of 11.3 on 12 cores. We do not obtain a perfect speedup for this embarrassingly parallel application due to internal memory bandwidth limitations for these multi-core processors: the efficiency starts to decrease with more than two processes on the same socket. Moreover it can be noticed that when activating the two-way SMT (Simultaneous multithreading, or *Hyper-threading* for Intel) capability of the X5650 CPUs we can obtain a speedup of 13.7 with 24 threads.

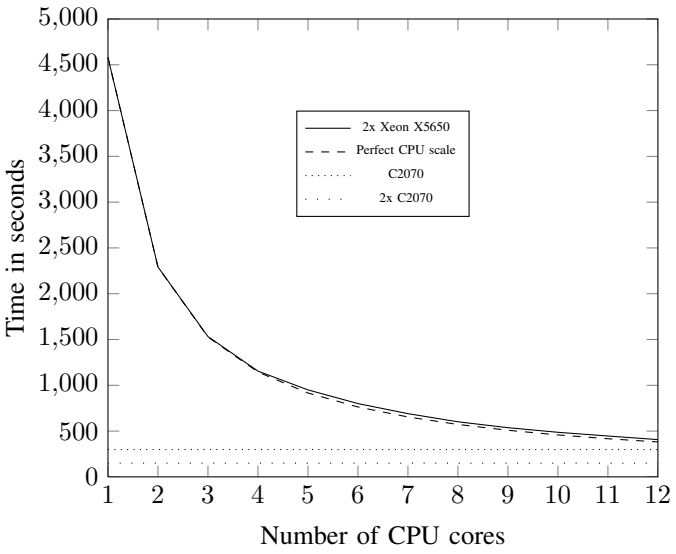

Figure 6. Comparison of multi-core CPUs and multiples GPUs.

With two GPUs, we obtain a perfect speedup of 2.0 over one GPU since the computation load is the same on the two GPUs. Finally, we obtain a speedup of 15.4 for one GPU over one CPU core, a speedup of 2.7 for one GPU with respect to one six-core CPU, and a speedup of 2.7 for two GPUs with respect to two six-core CPUs (2.2 with two-way SMT). The speedup is lower than with I_0 for two reasons. First, some intervals do not benefit from the swap as they involve the specific computations mentioned in Section II-C. Second, the divergence penalty can be stronger for some intervals which lowers the computation gain.

V. Conclusion and future work

In this paper, we have proposed a first step in deploying on GPU an algorithm to solve the Table Maker's Dilemma. This made it possible to obtain a speedup of 15.4 compared to the reference implementation on one CPU core, and a speedup of 2.7 over six CPU cores for finding HR-cases over the interval $[1; 1 + 2^{-3}]$.

Nevertheless, only one half of the full algorithm has been deployed on GPU. Indeed, the generation of the affine approximations of the function is still performed on the CPU (in 9.3 seconds for the interval $[1; 1 + 2^{-13}]$). Moreover the transfer on the PCI bus of all coefficients of the 2^{25} J intervals requires 0.25 seconds which is greater than our HR-case search on GPU. This deployment on GPU is therefore mandatory to accelerate the generation of affine approximations and to avoid

such costful transfers on the PCI bus. This is planed as a future work and is challenging since this requires an efficient multi-precision library on GPU.

As also mentioned in the introduction, there exists another algorithm named SLZ to search HR-cases. The algorithm heavily relies on the use of the LLL algorithm. The deployment of this algorithm on GPU is far from trivial if one wants to obtain good performances. The main advantage of the SLZ algorithm is to have fewer intervals to analyse (but this requires better approximations for the function) which may imply a different deployment on GPU. Porting this algorithm on GPU will be the next step of this work.

VI. Acknowledgement

This work was supported by the TaMaDi project of the french ANR (grant ANR 2010 BLAN 0203 01). The authors want to thank Vincent Lefèvre and Sylvain Collange for helpful discussions on the HR-case search and on GPU computing. We also thank Polytech-Paris UPMC and their system administrator team for allowing us to use their CPU-GPU server.

References

[1] V. Lefèvre, *Moyens arithmétiques pour un calcul fiable*. PhD thesis, École normale supérieure de Lyon, 2000.

[2] D. Stehlé, V. Lefèvre, and P. Zimmermann, "Worst cases and lattice reduction," in *16th IEEE Symposium on Computer Arithmetic (ARITH-16 '03)*, (Los Alamitos, CA, USA), pp. 142–147, IEEE Computer Society, 2003.

[3] D. Stehlé, V. Lefèvre, and P. Zimmermann, "Searching worst cases of a one-variable function using lattice reduction," *IEEE Transactions on Computers*, vol. 54, pp. 340–346, 2005.

[4] F. Dinechin, J.-M. Muller, B. Pasca, and A. Plesco, "An FPGA architecture for solving the Table Maker's Dilemma," in *22nd IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP 2011)*, pp. 187–194, IEEE Computer Society, 2011.

[5] J.-M. Muller, N. Brisebarre, F. D. Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of Floating-point Arithmetic*. Birkhauser, 2010.

[6] V. Lefèvre, "An algorithm that computes a lower bound on the distance between a segment and $\mathbb{Z}^2$," Tech. Rep. 97-18, Laboratoire de l'Informatique du Parallélisme, ENS Lyon, 1997.

[7] V. Lefèvre, "New Results on the Distance Between a Segment and $\mathbb{Z}^2$. Application to the exact Rounding," *17th IEEE Symposium on Computer Arithmetic (ARITH-17 '05)*, pp. 68–75, 2005.

[8] V. Lefèvre, J.-M. Muller, and A. Tisserand, "Toward correctly rounded transcendentals," *IEEE Transactions on Computers*, vol. 47, no. 11, pp. 1235–1243, 1998.

[9] P. Alessandri and V. Berthé, "Three distance theorems and combinatorics on words," *Enseignement Mathématique*, vol. 44, pp. 103–132, 1998.

[10] NVIDIA, *CUDA C Programming Guide Version 4.0*, May 2011.

[11] Khronos Group, *The OpenCL Specification Version 1.2*, November 2011.

[12] NVIDIA, *CUDA C Best Practices Guide Version 4.0*, May 2011.

[13] M. Harris, "Parallel prefix sum (scan) with CUDA," tech. rep., NVIDIA, April 2007.