DOI 10.12694/scpe.v17i3.1183
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 17, Number 3, pp. 237–249. http://www.scpe.org

SOLVING THE TABLE MAKER'S DILEMMA ON CURRENT SIMD ARCHITECTURES

CHRISTOPHE AVENEL[†§] PIERRE FORTIN[†] MOURAD GOUICEM[†‡] AND SAMIA ZAIDI[†]

Abstract. Correctly-rounded implementations of some elementary functions are recommended by the IEEE 754-2008 standard, which aims at ensuring portable and predictable floating-point computations. Such implementations require the solving of the Table Maker's Dilemma which implies a huge amount of computation time. These computations are embarrassingly and massively parallel, but present control flow divergence which limits performance at the SIMD parallelism level, whose share in the overall performance of current and forthcoming HPC architectures is increasing. In this paper, we show that efficiently solving the Table Maker's Dilemma on various multi-core and many-core SIMD architectures (CPUs, GPUs, Intel Xeon Phi) requires to jointly handle divergence at the algorithmic, programming and hardware levels in order to scale with the number of SIMD lanes. Depending on the architecture, the performance gains can reach 10.5x over divergent code, or be constrained by different limits that we detail.

Key words: floating-point arithmetic, Table Maker's Dilemma, SIMD, control flow divergence, OpenCL

AMS subject classifications. 68M07, 68W10

1. Introduction. Since 1985, the IEEE 754 standard specifies the implementation of floating-point operations in order to have portable and predictable numerical software. Its latest revision [15, 4] recommends the correct rounding of some elementary functions, like log, exp and the trigonometric functions. Since such functions are transcendental, one cannot evaluate them exactly but have to approximate their evaluation. However, it is hard to decide which intermediate precision is required in the function implementation to guarantee a correctly rounded result: 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* or TMD (see [25], chapter 12: Solving the Table Maker's Dilemma).

Solving the TMD involves finding the *hardest-to-round* arguments of the function [25], that is to say the arguments requiring the highest precision to be correctly rounded when the function is evaluated at. This precision guaranteeing the correct rounding for all arguments is named the *hardness-to-round* of the function [25]. The hardest-to-round cases can be found by *exhaustive search*, which implies to browse each floating-point number in the domain of definition of the function. This approach is however prohibitive since it leads to a $O(2^p)$ operation count when considering precision-p floating-point numbers as arguments.

In order to speed up the search for hardest-to-round arguments, the Lefèvre algorithm [21] uses local affine approximations of the targeted function. The domain of definition of the function is split into several domains D_i and an affine approximation of the function is computed for each D_i. Thanks to this affine approximation, one can isolate *hard-to-round cases* (HR-cases, see [25, 23]) with a $O(p^2)$ operation count for a domain D_i with precision-p floating-point numbers. The hardest-to-round cases are then found among the HR-cases with a localized exhaustive search. Higher degree approximations have been introduced since (SLZ algorithm [29]) in order to further reduce the asymptotic operation count for large values of p. However quadruple precision ($p = 113$) is still currently out of reach. We thus focus in this article on the double precision format ($p = 53$), for which the Lefèvre algorithm is as efficient as the SLZ algorithm in practice [25, 6]. Moreover, the Lefèvre algorithm has already been used to generate all known hardness-to-round in double precision [25], and it offers fine-grained parallelism which is suitable for massively parallel architectures like GPUs [8, 10]. We therefore study the Lefèvre algorithm here.

Even if the Lefèvre algorithm makes it possible to compute the hardness-to-round of elementary functions, it remains very computationally intensive. For example, it requires around five years of CPU time for the exponential function over all double precision arguments. Moreover, even if the hardest-to-round cases of some functions in double precision are known [25], this is still not the case for about half of the univariate functions

[†]Sorbonne Universités, UPMC Univ Paris 06, CNRS, UMR 7606, LIP6, F-75005, Paris, France
[‡]LIRMM, CNRS/Université Montpellier 2, UMR 5506, Montpellier, France
[§]Centre for Image Analysis, Uppsala University

recommended by the IEEE standard 754-2008. Furthermore, some scientific computations may require correctly-rounded implementations of other elementary functions, of specific compositions of elementary functions or even of elementary functions using non-standard formats or precisions. Being able to find the hardness-to-round of any elementary function in double precision in a reasonable amount of time would therefore be very useful.

In practice, both the affine approximation generation and the HR-case search are independent among the D_i domains. This data-parallel algorithm is thus embarrassingly and massively parallel which suits well to multi-core and many-core parallel architectures. However, such architectures rely heavily on SIMD (Single Instruction Multiple Data) parallelism. As far as CPUs are concerned, such parallelism is increasingly important in the overall CPU performance since the SIMD vector width has been constantly increasing from 64 bits (MMX [16] and 3DNow! [1]) to 128 bits (SSE [17], AltiVec [5]), then to 256 bits (AVX [18]), and to 512 bits on the Intel Xeon Phi as well as in the forthcoming AVX-512 instruction set [19]. As far as GPUs are concerned, they are now widely used in HPC and also present a partial SIMD execution since multiple GPU threads are processed in a SIMD fashion by groups of 32 or 64 threads. This increasing SIMD parallelism offers indeed important performance gains at a relatively low hardware cost. But efficiently exploiting such parallelism requires "regular" algorithms where the memory accesses and the computations are similar among the lanes of the SIMD vector unit. Unfortunately, as presented in [8], the original HR-case search of Lefèvre algorithm (*Lefèvre HR-case search*) presents multiple sources of control flow divergence which limit its performance on GPUs.

In this paper, we show that efficiently solving the TMD on various multi-core and many-core SIMD architectures (CPUs, GPUs, Intel Xeon Phi), and scaling performance with the number of SIMD lanes, requires to jointly handle this divergence at multiple levels: algorithm, programming and hardware. We start by describing a regular HR-case search algorithm, first presented in [9, 10], which has been shown to drastically reduce divergence in the execution flow on NVIDIA GPUs [9, 10]. We then extend the deployment of this HR-case search on other SIMD architectures: CPUs and the Intel Xeon Phi. We first compare C programming with the SPMD-on-SIMD (*Single Program Multiple Data*) programming model [26, 7], and show that the SPMD-on-SIMD model is well-suited for vectorizing the HR-case search on CPUs and on the Xeon Phi. Moreover, thanks to OpenCL this programming model enables code portability on various architectures, including AMD GPUs. Secondly, we present a survey of the deployment of this OpenCL implementation on various current SIMD architectures. We detail the performance results depending on how divergence is handled at the hardware level, and on the discrepancy between control flow (static) divergence and execution flow (dynamic) divergence. We obtain performance gains up to 10.5x on some architectures, and specify the performance bottlenecks on the other architectures. Finally, we present a performance comparison of these architectures for solving the TMD.

As far as related work is concerned, general solutions have been proposed to handle divergence on SIMD architectures, at the hardware level [3, 12, 24] as well as at the software level [11, 13, 31, 28]. We target here currently available hardware, and our HR-case searches offer very fine computation grains: the overhead of software solutions to handle divergence would be too high here (see [8] for example). Up to our knowledge, there is no other specific work to reduce the SIMD divergence when solving the TMD. The reference C code of V. Lefèvre [21] is a CPU scalar code that can target multi-core and distributed multi-processor architectures, but does not exploit SIMD parallelism within each CPU core. It can be noticed that another implementation to solve the TMD has been designed for FPGA architectures [6], but this implementation relies on the exhaustive search.

In the rest of this paper, Sect. 2 introduces the Table Maker's Dilemma and Lefèvre algorithm. In Sect. 3 we present our regular algorithm for the HR-case search which reduces divergence in the execution flow. In Sect. 4 we compare two programming models to enable the CPU vectorization of the HR-case search code, and show the relevance of the SPMD-on-SIMD programming model. Section 5 presents performance results, detailed analyzes on various SIMD architectures, and the performance comparison among these architectures. Finally, concluding remarks will be presented in Sect. 6.

2. The Table Maker's Dilemma and Lefèvre algorithm. Floating-point arithmetic aims at approximating real arithmetic. It uses the property that any real number α can be uniquely represented in base β scientific notation as $\alpha = m \times \beta^e$, with $1 \leq m < \beta$ the *significand* of α and $e \in \mathbb{N}$ the *exponent* of α. A precision-p floating-point (abbreviated FP_p) number format will then represent real numbers with e in a specific range and m with a finite precision of p digits. Hence every real number α is either exactly representable as an

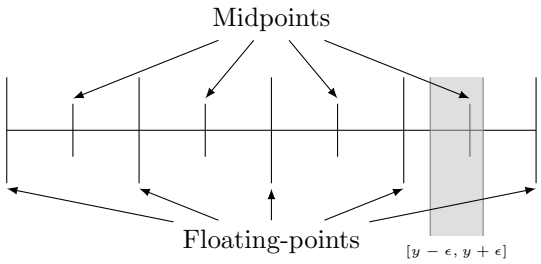


FIG. 2.1. *Example of undetermined correct rounding for a value y computed with precision ϵ in the case of rounding to nearest, where the rounding breakpoints are the midpoints of floating-point numbers.*

FP_p number, or is not, in which case it will be approximated using a *rounding function*.

When evaluating a function at a precision-p floating-point argument, the exact result is commonly not a precision-p floating-point number. The simple example of the inverse function $1/x$ evaluated at x different than a power of 2 (e.g. $x = 10$) yields an infinite sequence of digits in the fractional part of the result if represented in binary format. Hence, a typical implementation of a mathematical function will have to approximate the exact mathematical result $f(x)$ by $\hat{f}(x)$ with precision ϵ, and then round this approximation to p bits of precision. However, if for some argument x, $\hat{f}(x)$ is at a distance less than ϵ to a rounding breakpoint (where the result of the rounding function changes), it is impossible to determine the correct rounding of $f(x)$ from $\hat{f}(x)$ as illustrated in Fig. 2.1. Such an argument x is called a (p, ϵ) *hard-to-round* case (abbreviated as HR-case).

Given a function f, a rounding function, and a FP_p format, the *Table Maker's Dilemma* is hence defined as finding the necessary accuracy ε such that both $f(x)$ and an approximation $\hat{f}(x)$ with accuracy ε round to the same FP_p number for every FP_p number x in the definition domain of the function f. The largest ε verifying this property is called the *hardness-to-round* of f at precision p.

To find this hardness-to-round with a better complexity than exhaustive search, Lefèvre algorithm relies on a three step methodology based on searching (p, ϵ) HR-cases [21]:

- fix a "convenient" ϵ using probabilistic assumptions [25],
- find (p, ϵ) HR-cases with *ad hoc* methods,
- find the hardest-to-round case among the (p, ϵ) hard-to-round cases.

There are two key-points in that methodology. First, the statistical assumption states that the probability of an argument being an HR-case decreases exponentially with the precision of the approximation. This implies that the hardness-to-round of an elementary function evaluated to a precision-p is likely to be around 2^{-2p} [25], and that there are few $(p, 2^{-2p})$ HR-cases. This can dramatically reduce the use of exhaustive search, which is time consuming. But more importantly, the second key-point is that we can search for HR-cases in polynomial time in the format size, against an exponential time for the exhaustive search. This efficient HR-case search is obtained by using polynomials and the following two steps.

- *The generation of polynomial approximations*: we generate many local polynomial approximations P_i of the function f over independent domains D_i, with error $\epsilon_{approx} \approx \epsilon$.
- *The (p, ϵ') HR-case search*: for each polynomial approximation P_i, we search for (p, ϵ') HR-cases of P_i, which are the (p, ϵ) HR-cases of f, with $\epsilon' = \epsilon + \epsilon_{approx}$.

These two steps are massively parallel over the domains D_i since these numerous domains can all be processed independently. However, while the polynomial approximation generation has a regular control flow [9], the HR-case search presents divergence issues when executed on SIMD architectures. Moreover, the HR-case search is the most time consuming step when solving the TMD. In the rest of this paper, we will therefore focus on this HR-case search and its SIMD execution on various HPC architectures.

The HR-case search on the polynomials P_i is done using an isolation strategy [22] as described in Algorithm 1. An HR-case existence test `Exists?` is executed on each domain D_i. It returns `True` if there potentially exists an HR-case in the tested domain (false positives are possible), or `False` otherwise. If the test succeeds (that is to say, there might be an HR-case in the tested domain D_i), we split D_i into κ sub-domains $D_{i,j}$, upon which

we repeat the HR-case existence test. For each of these sub-domains succeeding the HR-case existence test, we finally perform exhaustive search. In [22], Lefèvre tested other variants of this strategy and concluded that in practice the most efficient strategy was this three phases refinement. This is mainly due to the fact that the number of arguments succeeding the existence test can be roughly predicted, and that the amount of time spent in the HR-case existence test has to be balanced with the amount of time spent in exhaustive search [9, 22].

Algorithm 1: Lefèvre three phases isolation strategy for HR-case search.

```
1  foreach P_i over its domain D_i do
2      if Exists?(P_i, ε') then                                   /* Phase 1 */
3          (D_{i,1}, D_{i,2}, ..., D_{i,κ}) := SplitDomain(D_i, κ);
4          (P_{i,1}, P_{i,2}, ..., P_{i,κ}) := RefineApprox(P_i, κ);
5          foreach P_{i,j} over its domain D_{i,j} do
6              if Exist?(P_{i,j}, ε') then                        /* Phase 2 */
7                  ExhaustiveSearch(P_{i,j}, ε') ;                /* Phase 3 */
8              end
9          end
10     end
11 end
```

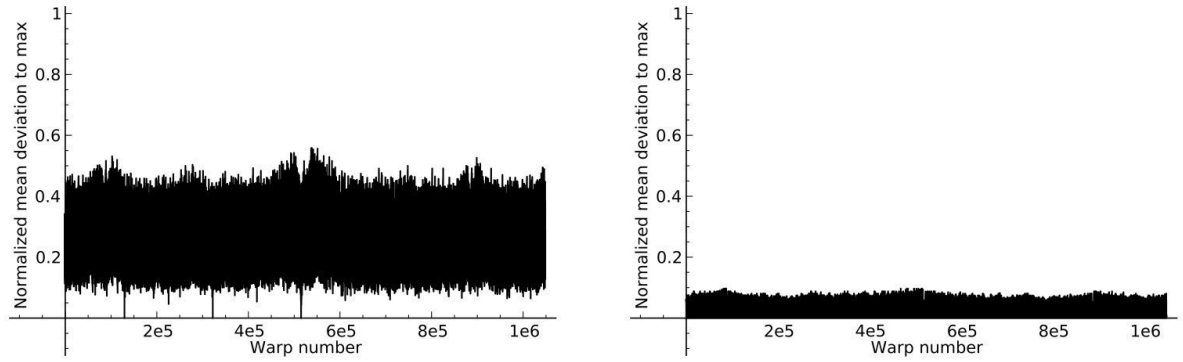
Lefèvre HR-case existence test takes as argument a degree one polynomial P_i or $P_{i,j}$. As we do not need the dynamic range of floating-point numbers, we use fixed-point arithmetic to avoid rounding errors, and we apply a suitable change of variable to write P_i or $P_{i,j}$ as a polynomial $b - a \cdot x$, while representing only the 64 bits after the p^{th} bit of the significands of a and b as 64-bit integers. Hence we also consider $x \in \mathbb{N}$. This HR-case existence test then returns a lower bound on the distance between the values of $b - a \cdot x$ for $x < N$ and the rounding breakpoints, with N the number of arguments to test in D_i or $D_{i,j}$. This is achieved by computing the continued fraction expansion of a with the Euclidean algorithm, and a particular decomposition of b in the sequence of partial remainders. Comparing this lower bound to ϵ', we can then determine whether there is potentially a (p, ϵ') HR-case in the domain or not. Lefèvre existence test is presented in Algorithm 2 and is explained more thoroughly in [9, 10].

Algorithm 2: Lefèvre HR-case existence test algorithm.

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

3. A regular algorithm for the HR-case search. In [8], we underlined a problem in Lefèvre algorithm execution on GPU architectures: the execution flow is highly divergent from one thread to another. There are three sources of divergence in Algorithm 2:

- the main unconditional loop, whose number of iterations depends on the value of the arguments;



(a) Lefèvre HR-case search (b) Regular HR-case search

FIG. 3.1. *Normalized mean deviation to the maximum of the number of main loop iterations per CUDA warp, on NVIDIA GPUs, among the* 2^{20} *CUDA warps required for the* exp *function in the domain* $[1; 1 + 2^{-13}]$.

- the main conditional statement, whose scope contains all the intructions within the main loop;
- and finally the divisions, which are computed using a hybrid implementation with a tunable parameter `LOGMS`. If we compute p/q and $p > 2^{\texttt{LOGMS}}q$, we call the division instruction, otherwise it is more efficient to use a loop to compute the quotient by repeated subtractions. As divisions operands are 64-bit integers, `LOGMS` $= 64$ implies all quotients are computed using repeated subtractions, and `LOGMS` $= 0$ implies all quotients are computed using the division intruction.

Even though the hybrid divisions affect the control flow, they do not lead to strong divergence issue at runtime since almost all the computed quotients are expected to be small in practice [9]. However when processing multiple instances of the Lefèvre HR-case existence test in parallel on GPUs, the main conditional statement and the main loop have a strong performance impact because of the partial SIMD execution of GPUs. Both are induced by conditioning the computation of the quotients of the continued fraction of a by the value of b. To our knowledge there is no *a priori* information on the number of loop iterations or on the branch executed at each iteration that would enable us to statically reorder the domains D_i in order to decrease this divergence. We also tried to use software solutions to reduce the impact of the loop divergence [8]: no performance gain was obtained because the computation is very fine-grained.

To highlight the impact of loop divergence during Lefèvre existence test execution, we introduced in [8] an indicator named the *normalized mean deviation to the maximum*. When processing concurrently n independent instances of a divergent loop on a SIMD unit with n lanes, the number of loop iterations issued in total is the maximum number of loop iterations issued among all the lanes of the SIMD unit. This indicator aims thus at giving the average percentage of loop iterations for which a lane remains idle during the SIMD execution. More formally, we denote ℓ_i the number of loop iterations to issue for the lane i and we number the lanes within a SIMD vector from 1 to n. If $\ell = \{\ell_i, i \in [\![1, n]\!]\}$, the Normalized Mean Deviation to the Maximum (NMDM) is defined as

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

In Fig. 3.1(a), we measured the NMDM of the main unconditional loop of Lefèvre HR-case search execution on a NVIDIA GPU ($n = 32$) on a set of domains D_i for the exponential function. We can see that the NMDM is uniformly high with an average NMDM of 25.6%, which means that a SIMD lane remains idle on average 25.6% of the number of loop iterations issued on its SIMD unit. This divergence in Lefèvre HR-case search is mainly due to the fact that the algorithm goes from the subtraction-based Euclidean algorithm to the division-based one depending on the value of b.

In [9, 10], we proposed a new HR-case existence test which presents a regular execution, as illustrated in Algorithm 3. This is enabled by getting rid of the dependence between the computation of the continued

Algorithm 3: Regular HR-case existence test algorithm.

```
   input  : b − ax, ϵ′, N
1  initialisation:  p ← {a};   q ← 1;   d ← {b};
                    u ← 1;     v ← 0;
2  while True do
3  |  k = ⌊q/p⌋;
4  |  q = q − k ∗ p; u = u + k ∗ v;
5  |  d = d  mod p;
6  |  if u + v ≥ N then return d > ϵ′;
7  |  k = ⌊p/q⌋;
8  |  p = p − k ∗ q; v = v + k ∗ u;
9  |  if d ≥ p then
10 |  |  d = d − p  mod q;
11 |  end
12 |  if u + v ≥ N then return d > ϵ′;
13 end
```

fraction expansion of a and the value of b. It first turns the unpredictable main conditional statement of Lefèvre algorithm into a deterministic test, which can be removed by unrolling two loop iterations as in Algorithm 3. And second, a full quotient of the Euclidean algorithm is entirely computed at each loop iteration in the regular HR-case search, which is not the case in the Lefèvre existence test. As the number of quotients to compute is almost constant from one domain D_i to the next, we reduce the mean NMDM per SIMD computation on NVIDIA GPUs from 25.6% to 0.1% (cf. Fig. 3.1(b)). However, even though the execution flow is now regular, the control flow in the source code remains divergent: the algorithm still exhibits the main unconditional loop, a few conditional statements, inner loops for the hybrid implementation of the divisions, and outer loops (over the D_i or $D_{i,j}$ domains as presented in Algorithm 1).

In practice, such regular existence test offers performance gains on NVIDIA GPUs up to 3.4x over Lefèvre existence test [9, 10]. When comparing an high-end hex-core CPU with an high-end NVIDIA GPU, the GPU deployment delivers a 6.6x speedup for the regular existence test. Such speedup is mainly due to the lack of SIMD computations on the CPU. That is why we will now consider the vectorization of the two existence tests on CPUs and on Xeon Phi.

For the sake of shortness, we will name in the remainder of this paper *Lefèvre HR-case search* the combination of the isolation algorithm with Lefèvre existence test, and *regular HR-case search* the combination of the isolation algorithm with the regular existence test.

4. The relevant programming paradigm. In order to deploy the HR-case search on CPUs and on the Xeon Phi, one could first consider to rely on C programming. We thus start by considering the vectorization of the reference scalar C code implementing the Lefèvre and regular HR-case searches [9, 10].

4.1. Vectorization with C compiler. When considering the vectorization of a C program, one can use several programming paradigms.

Manual SIMD programming with intrinsics is a first possibility. However, this is generally a tedious task which requires for example array padding and which leads to non-portable code: the program must be re-written when moving to another SIMD instruction set or to another vector width. In the case of the HR-case search, this would be an especially tedious task because of the multiple nested `while` loops and conditional branches. Each one is a divergence source which implies a different mask to handle this divergence on CPU SIMD units. Therefore, with intrinsics we would have to set and update all these masks in the source code which represents a very important programming effort.

Another possibility is to rely on the C compiler. Automatic vectorization is provided for example in `icc` (Intel C/C++ Compiler) and `gcc` (GNU C Compiler). The programmer can let the compiler perform the dependency analysis of the targeted loop, in order to determine whether the loop is parallel or not, hence vectorizable or not. This dependency analysis is however limited by the compiler capacity [20]. Therefore, some compilers support compiler directives, which enable the programmer to indicate (and ensure) that the loop is parallel: no dependency analysis is then required by the compiler. Such compiler directives are available in `icc`

(`#pragma simd`), and have recently been standardized in the last versions of OpenMP (OpenMP 4.X).

As far as the HR-case search is concerned, we aim at vectorizing multiple iterations of the outermost loop which browses 2^{25} D_i domains. We use here `icc` (version 15.0.2).

4.1.1. Exhaustive search. In order to start with a simpler code, we first consider only the exhaustive search (phase 3): phases 1 and 2 are here removed from the code. In the reference C code, the original implementation used to rely on two inner `do..while` loop for each D_i domain: the 2^{15} arguments of the D_i domain being browsed as 8 sub-domains of 2^{12} arguments (to match phase 2). In order to minimize the number of nested loops, and to ease the compiler vectorization, these two `do..while` loops have been merged in one single `for` loop.

When considering automatic vectorization of a loop nest without compiler directive, the compiler starts with the innermost loop [20], which corresponds here to this new inner `for` loop within the outer `for` loop over the D_i domains. However, as the exhaustive search is performed using the tabulated differences algorithm [21], this inner `for` loop presents `flow` and `anti` data dependencies among its iterations: the next polynomial evaluation is computed from the current one. This used to prevent former versions of `icc` from vectorizing the inner loop and hence the outer loop. The latest version of `icc` (15.0.2) can override this vector dependency on the inner loop, and attempt to vectorize the outer loop. But, this outer loop vectorization fails due to an `output` dependency. Indeed, the 2^{25} D_i domains provided as input lead in practice to very few HR-cases (e.g. 243 for the first set of 2^{25} D_i domains). These HR-cases are written consecutively in memory thanks to a counter incremented each time an HR-case is found, which results in an `output` data dependency between successive iterations.

When considering vectorization hinted by compiler directives (here `#pragma simd` on the outer loop), either the compiler does generate vector code, which leads to wrong results because of this `output` data dependency, or the compiler detects the dependency and refuses the vectorization.

4.1.2. Complete HR-case search. We now consider the complete HR-case search with the three phases and study the regular HR-case search. In order to enable the vectorization [20], we had to strongly rewrite our C code. Function calls from the loop bodies have first been replaced by preprocessor macros. The corresponding `return` statements have been replaced by boolean tests: this ensures one single entry and one single exit in each loop [20]. Likewise, `goto` statements among the different phases have been replaced by boolean tests.

When considering the complete HR-case search, the compiler faces the same data dependencies as with the exhaustive search only. Moreover, there are in phases 1 and 2 inner `while` loops with unknown iteration numbers, which cannot be vectorized and can thus prevent the outer loop vectorization. All this leads to the same conclusion: the outer loop vectorization fails without compiler directives. When forcing vectorization with compiler directives, the compiler can manage to vectorize the code but this results again in wrong results because of the `output` dependency.

It has to be noticed that the same conclusions would also apply to the Lefèvre HR-case search which presents the same data dependencies and also outputs its HR-cases consecutively.

4.2. Implicit vectorization in OpenCL. Another possibility to exploit the SIMD units is to rely on the SPMD-on-SIMD (*Single Program Multiple Data*) programming model [26, 7]. All computations are written as scalar ones and it is up to the compiler to merge such scalar computations in SIMD instructions. The main advantages are the ease of programming and the portability: the programmer needs neither to write the specific SIMD intrinsics for each architecture, nor to know the vector width, nor to implement data padding with zeros according to this vector width. The vector width will indeed be determined only at compile time (depending on the targeted hardware). Moreover, like compiler directives, no data dependency analysis is required by the compiler: it is up to the user to ensure that the scalar computations can be processed correctly in parallel. Such programming paradigm is increasingly used in HPC: first on GPUs with CUDA and then on various compute devices with OpenCL. On CPU, such programming model is available in OpenCL (OpenCL implicit vectorization), as well as in the Intel SPMD Program Compiler (`ispc`) [26]. We choose here OpenCL over `ispc` since OpenCL enables us to maintain one single source code for both CPUs and GPUs, and to target other GPUs like the AMD ones. On multi-core CPUs, we use the Intel OpenCL SDK[1] which provides OpenCL

[1] See: `https://software.intel.com/en-us/intel-opencl`

TABLE 5.1
Times in seconds for both HR-case searches over I_0 on one NVIDIA C2070 GPU.

HR-case search	Lefèvre		Regular	
	CUDA	OpenCL	CUDA	OpenCL
Phase 1	0.258	0.246	0.074	0.069
Phase 2	0.006	0.006	0.010	0.008
Phase 3	0.001	0.001	0.003	0.003
Total	0.265	0.253	0.086	0.081

implicit vectorization while supporting conditional statements as well as `while` loops in the OpenCL kernels [27]. It has to be noticed that OpenCL also provides parallelism at the thread level in order to exploit multi-core processors in shared memory. This is however not a key-point here since such parallelism is straightforward to implement in the HR-case search [10].

Our OpenCL kernels are thus a translation of our CUDA kernels [8, 9]. The OpenCL implementation therefore uses the same code structure as in the CUDA implementation [8] where the three phases of the HR-case search have been separated in three distinct GPU kernels. Like in CUDA, atomic operations are used in OpenCL to consecutively write the outputs of each phase in memory. This includes the HR-cases resulting from phase 3 and leading to the `output` dependency with the C compiler vectorization. Here this issue is easily solved thanks to the SPMD-on-SIMD programming model, and these atomic operations are the only synchronizations required among the work-items. We thus emphasize that the HR-case search of the Table Maker's Dilemma fits naturally with the SPMD-on-SIMD programming model: each work-item processes one (or a few) D_i domain(s), and only a few atomic operations are required for correct work-item synchronization. We then fully exploit the data parallelism of this massively parallel application to process concurrently the numerous work-items on the SIMD units (as well as on all the available CPU cores).

However, as far as performance is concerned, the divergence in the SIMD processing of consecutive D_i domains will be a key-factor and will impact performance differently depending on the algorithm (Lefèvre or regular HR-case search) and on the underlying hardware, as detailed in the next section.

5. Performance results of HR-case searches on various current SIMD architectures. For the following performance results, each OpenCL implementation is tuned in order to determine, for each OpenCL kernel, the optimal value for the work group size, for the number of intervals processed by each work-item, and for the LOGMS value (for phases 1 and 2, cf. Sects. 2 and 3).

All the results given in this section are issued from the HR-case search on the *exp* function for double precision. Except otherwise mentioned, all tests are performed over the 1024 first intervals $I_{0..1023} = [1; 1+2^{-3}[$ of the binade $[1; 2[$ ($I_{0..1023}$ containing 2^{50} *doubles*). The parameter tuning has been performed only on the interval $I_0 = [1; 1+2^{-13}]$ (containing 2^{40} *doubles*).

5.1. GPUs and SIMD width impact. We first present performance results of our OpenCL implementation on both NVIDIA and AMD GPU architectures. Table 5.1 shows that for both HR-case searches the performance of our OpenCL implementation matches, and even slightly outperforms, the one of our original CUDA code on NVIDIA GPUs. This validates the choice to move from CUDA to OpenCL even on NVIDIA GPUs.

We now consider in Fig. 5.1 performance results on various high-end GPUs. On one NVIDIA C2070, the regular HR-case search delivers a 2.7x performance gain over the Lefèvre HR-case search thanks to its regular execution flow. On a newer NVIDIA GPU (K20c),the two HR-case searches are 2.2 or 2.3 times faster compared to their execution on the C2070, which shows that our HR-case search GPU implementation scales well on the newer Kepler GPU architectures which offers a much higher number of GPU cores. Besides, the performance gain of the regular HR-case search over the Lefèvre HR-case search is almost the same. The SIMD width is indeed the same in both Fermi (C2050) and Kepler (K20c) architectures: work-items are processed in a SIMD fashion by groups of 32.

Starting from the Southern Islands family (which includes our Radeon HD 7970), AMD GPUs present a scalar architecture (*Graphics Core Next* - GCN) that enables the programmer to reach best performance with

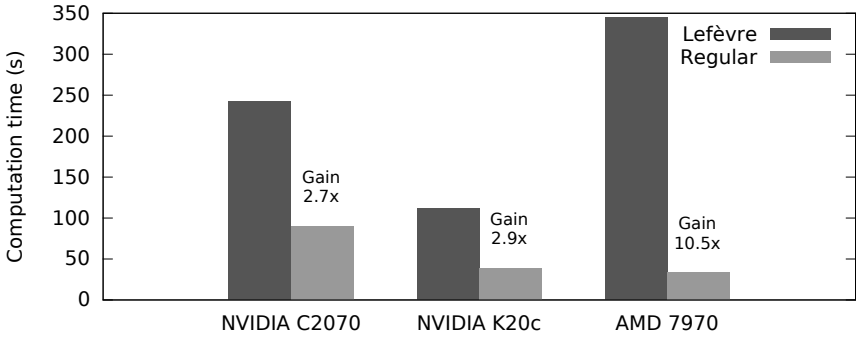


FIG. 5.1. *HR-case searches computation times over $I_{0..1023}$ on various GPUs, using OpenCL from CUDA 7.5 (352.68 driver) and from the AMD SDK 2.9.1 (Catalyst Omega 14.12 driver).The performance gains on top of the bars correspond to the Lefèvre over regular ratios.*

scalar work-items. Contrary to previous AMD GPU generations, no explicit vector programming is required. On these scalar GPUs, the ALUs are arranged in four SIMD arrays consisting of 16 processing elements each [2] (using OpenCL terminology), and work-items are processed in a SIMD fashion by groups of 64. Dynamic divergence among work-items can therefore have an even more detrimental impact on AMD GPUs than on NVIDIA GPUs. This explains that the gain of the regular HR-case search over the Lefèvre's one is more important on AMD Radeon HD 7970 GPU than on NVIDIA GPUs. Comparing this AMD 7970 and the NVIDIA K20c, whose hardware compute powers are similar[2], one can see that only thanks to the regular HR-case search similar application performance can be achieved on these two GPUs.

5.2. AVX and SSE CPUs. We now consider the OpenCL deployment of the two HR-case searches on standard CPUs with either SSE4.2 or AVX2 SIMD instruction sets. SSE4.2 vector units can process two 64-bit integers in a SIMD fashion, whereas AVX2 ones can process four 64-bit integers. There are however multiple issues when considering the deployment of the HR-case searches on such SIMD instruction sets.

The first issue lies at the hardware level, where divergence among SIMD lanes is handled differently on CPU and on GPU [14]. When the control flow diverges on a GPU SIMD unit, a mask register is set according to the condition evaluation: each processing element then either performs the following instruction or remains idle. A stack of mask registers is used to handle nested divergence levels. This is handled dynamically by the GPU hardware, which can then skip at runtime branches where all processing elements would be idle (e.g. for a `if-then-else` statement: when all mask bits are zero the `then` branch can be skipped, and when all mask bits are one the `else` branch can be skipped). When control flows diverge within a CPU SIMD unit, mask registers are also used to handle divergence among the SIMD lanes. On AVX and SSE however, all computations are always performed by all the SIMD lanes. The masks are used to prevent committing results in memory for computations that should not have been performed (*predication*). Moreover, on CPUs all this is handled explicitly in software by the compiler. This implies a general overhead compared to the GPU hardware management, and can also be crucial for the SIMD performance of our specific application. Both HR-case searches show indeed important static divergence (at compile time, in their control flow), but the regular HR-case search presents low dynamic divergence (at runtime, in its execution flow). This low execution flow divergence can thus be handled efficiently by the GPU hardware, while the CPU compiler has to set all the required masks for predication according to the control flow divergence of the source code. As far as masks with all zeros or all ones are concerned, it has to be noticed that recent work can detect these cases at runtime in order to avoid using code with predication when possible on CPUs [30].

The second issue with the vectorization of the HR-case search on x86 CPUs is the lack of vector integer

[2]We are not aware of the exact 64-bit integer compute power of these GPUs, but their floating-point peak performances are similar : 3520 SGflop/s (single precision) and 1170 DGflop/s (double precision) for the K20c, against 3789 SGflop/s and 947 DGflop/s for the 7970.

division instruction in SSE, in AVX2 [18] and even in the forthcoming AVX-512 [19]. The compiler therefore uses the scalar integer division instruction, or emulates the vector integer division: e.g. with vector subtraction instructions or with optimized intrinsics such as `_mm_div_epu64`, `_mm256_div_epu64` or `_mm512_div_epu64` (from the Intel Short Vector Math Library - SVML).

As far as our performance tests are concerned, the AVX2 server hosts an Intel Xeon E3-1275 v3 CPU, with 4 physical cores running at 3.50 GHz and 2-way SMT, and we use on this server the Intel SDK for OpenCL 2016 and the OpenCL Runtime 15.1. The SSE4.2 server hosts two Intel Xeon E5-2660 CPUs, totalizing 16 physical cores running at 2.20 GHz with 2-way SMT (and using SSE4.2 for integer SIMD operations), as well as a Xeon Phi coprocessor: we use here the Intel SDK for OpenCL 2016 with the OpenCL Runtime 14.2 (latest version for Xeon Phi coprocessors). On AVX and SSE, the OpenCL compiler relies on a heuristic[3] to determine if it is worth generating vector code. We use here the `CL_CONFIG_CPU_VECTORIZER_MODE` environment variable to explicitly force or prevent the OpenCL implicit vectorization.

Figure 5.2(a) shows the performance results of both HR-case searches on the AVX2 server with both vector and scalar codes generated. When inspecting the assembly code generated by the OpenCL compiler, one can see that the vector codes contain AVX2 vector instructions for additions, multiplications and subtractions but only scalar 64-bit integer divisions. No vector division, such as `_mm256_div_epu64`, are generated. Along with the overhead required for masking, this leads to the vectorized versions being slower than the scalar ones. When comparing the regular and Lefèvre HR-case searches, the regular HR-case search is then 1.5x faster in scalar mode, and 1.4x faster in SIMD mode, which nevertheless shows the benefit of the regular HR-case search on CPUs.

The same conclusions apply to the SSE4.2 server[4] (cf. Fig. 5.2(b)): the use of scalar 64-bit integer division and the cost of masking inhibit SIMD performance gains. On the SSE4.2 server, the regular HR-case search delivers in the end the same performance gains over the Lefèvre one as on the AVX2 server.

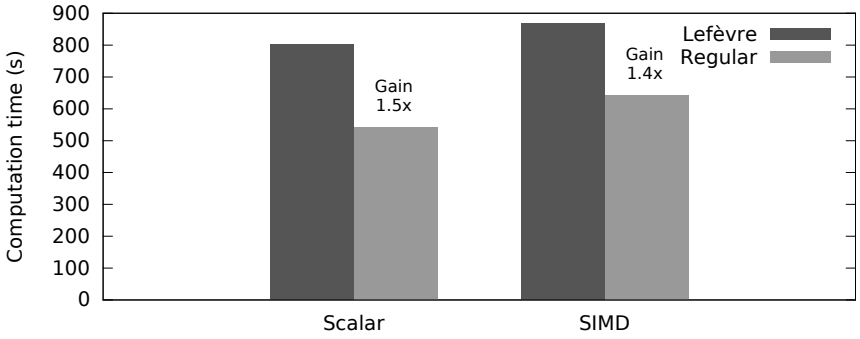
As far as the SIMD division issue is concerned, it can be noticed that we also tried to use `LOGMS` = 64 (cf. Sect. 3) instead of the optimal `LOGMS` in order to implement divisions with SIMD subtractions and thus try to avoid this issue: the scalar execution time will not be optimal, but the SIMD speedup could improve performance in the end. This however only leads to the vector versions being almost as fast as the scalar ones on both servers, the overhead of masking still inhibiting SIMD performance gains, and this thus results in an overall performance loss.

5.3. The Xeon Phi coprocessor. Since masking hinders SIMD performance gains on CPUs, we now target a Xeon Phi coprocessor (Knights Corner 5110P, with 60 cores with 4-way SMT at 1.053 GHz). Indeed the overhead of masking for SIMD control flow divergence is lower on Xeon Phi than on AVX2 or SSE4.2 CPUs since all Xeon Phi SIMD instructions directly support a 16-bit mask to control which lanes are active or not during the instruction execution. This avoids the predication required for SSE or AVX, but still requires a software management of the masks. However, there are no 64-bit integer SIMD arithmetic operations on the current Xeon Phis. The compiler must therefore emulate these 64-bit SIMD operations with 32-bit integer SIMD operations.

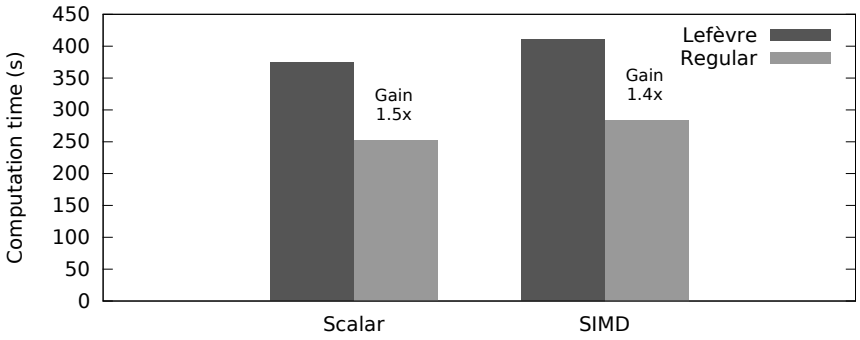
As far as SIMD 64-bit integer division is concerned, the assembly SIMD code generated from the OpenCL kernel shows both scalar 64-bit integer divisions and calls to SVML 64-bit integer functions. Since we are unable to determine which ones are indeed executed at runtime, we performed some micro-benchmarks of the 64-bit integer division on the Xeon Phi as presented on table 5.2. All tests on the Xeon Phi have been performed with the Intel SDK for OpenCL 2016 with the OpenCL Runtime 14.2 and the Intel Manycore Platform Software Stack 3.4. Scalar code is obtained thanks to the `CL_CONFIG_USE_VECTORIZER` environment variable. One can see that the vectorized OpenCL kernel offers the same SIMD speedup and performance as the C+SVML code. Even if our HR-case search kernels are thus likely to also use this SVML 64-bit integer division, the SIMD speedup for this operation is actually low: only up to 2.7x, whereas we have obtained SIMD speedups between 7.2x and 7.7x for 64-bit additions and multiplications (8x being the maximum theoretical speedup for 64-bit SIMD operations on the Xeon Phi). This shows that the SIMD 64-bit integer division is currently a potential bottleneck for SIMD performance on the Xeon Phi.

[3] See: `https://software.intel.com/en-us/node/540483`

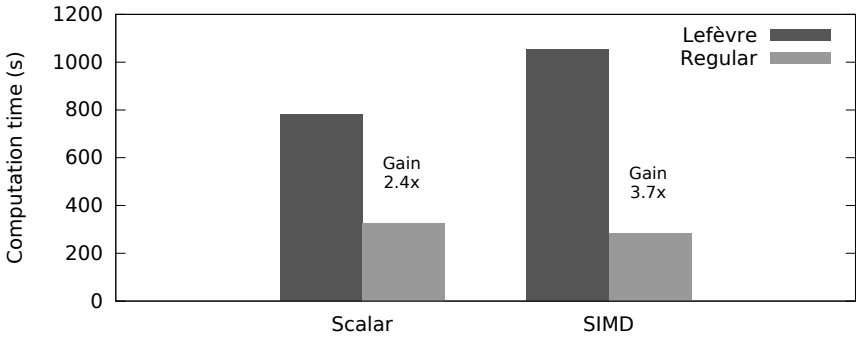
[4] Computation times are smaller on the SSE4.2 server than on the AVX2 server because of its higher number of CPU cores.



(a) AVX2 server



(b) SSE4.2 server



(c) Xeon Phi coprocessor

FIG. 5.2. *Performance results over $I_{0..1023}$ for Lefèvre and regular HR-case searches, in scalar and SIMD modes.*

Figure 5.2(c) shows the performance results of our HR-case searches on the Xeon Phi. Thanks to the more efficient masking on the Xeon Phi, we notice a 13% performance gain for SIMD code over scalar code with the regular HR-case search. This is an improvement over SSE and AVX CPUs, but the SIMD gain is very low: this can be explained by the emulation of 64-bit SIMD operations by 32-bit SIMD operations (while 64-bit scalar operations are not emulated), by the SIMD 64-bit integer division performance and by the software management of the masks. Again, using `LOGMS=64` to avoid the SIMD division issue does not improve these performance results. The SIMD version of the Lefèvre HR-case search shows however a performance loss with respect to its scalar version, probably due to its higher dynamic divergence. When comparing the two HR-case searches, the SIMD regular HR-case search is 3.7x faster than the SIMD Lefèvre HR-case search, and 2.8x faster than the

TABLE 5.2
Micro-benchmarks of the 64-bit integer division on the Xeon Phi. Computations are repeated 10^6 times on two input arrays of 1024 64-bit integers, with a scalar C code, a SIMD C+SVML code and an OpenCL kernel (vectorized or not).

	C [+ SVML]	OpenCL
Scalar	97.9 s	94.2 s
SIMD	36.7 s	39.9 s
SIMD speedup	2.7x	2.4x

scalar Lefèvre HR-case search on this architecture. This once again shows the interest of the regular HR-case search.

5.4. Architecture comparison. Thanks to the OpenCL portability, we can now compare the following different architectures for solving the TMD: the NVIDIA K20c and AMD 7970 GPUs, the SSE4.2 CPU server[5] and the Intel Xeon Phi. We first emphasize that the maximum power comsumptions are nearly the same: 225W for the K20c GPU, 250W for the 7970 GPU, 190W of TDP for the SSE4.2 server (with two Intel Xeon E5-2660 CPUs), and 225W of TDP for the Xeon Phi. In order to compute the 1024 intervals $I_{0..1023}$, with the regular HR-case search which performs best on all architectures, both GPUs require less than 40 s, whereas the SSE4.2 server and the Xeon Phi require at least 250 s. This 6.25x performance gap is clearly due to the inefficient SIMD execution of the HR-case search on the SSE4.2 CPUs and on the Xeon Phi.

6. Conclusion. In this paper, we have shown that handling efficiently the divergence on SIMD architectures for the most time consuming step of the Table Maker's Dilemma solving requires to use regular algorithms, with the relevant programming model, but also depends on the hardware. Using algorithmic changes, we can strongly reduce the divergence in the conditional statements within the main loop, as well as reduce the execution flow divergence on this main loop. Using OpenCL with its SPMD-on-SIMD programming model and its implicit vectorization feature, our massively parallel algorithm can be easily implemented and deployed on various GPUs and CPUs, as well as on the Intel Xeon Phi coprocessor.

Compared to the previous CUDA implementation, our OpenCL implementation shows similar performance gains (2.9x) for our regular algorithm on NVIDIA GPUs. This regular algorithm is even more decisive on AMD GPUs, with 10.5x performance gains, since their SIMD execution width is larger. However, when considering the SIMD units of CPUs and of the Xeon Phi, we show no or low performance gains for the SIMD execution over the scalar one. This is due the SIMD integer division implementation, to the lack of SIMD 64-bit integer instructions on the Xeon Phi, as well as to the static software handling of divergence on CPUs. This latter implies indeed an overhead compared to the dynamic hardware handling of divergence on GPUs, and cannot take full advantage of our regular algorithm, which presents important divergence in its control flow, but low divergence in its execution flow. However, it has to be noticed that the regular HR-case search still offers performance gains ranging between 1.5x and 2.8x on CPUs and on the Xeon Phi.

Currently, the solving of the Table Maker's Dilemma is thus more efficiently performed on GPU architectures due to their divergence handling. However, more efficient emulations of the SIMD integer division could be possible, and the forthcoming Xeon Phi processors (Knights Landing with AVX-512 SIMD instruction set) will support SIMD 64-bit integer instructions while maintaining a lower masking overhead than CPUs. This could lead to better SIMD performance gains on this architecture, especially for our regular algorithm.

Finally, it can also be noticed that FPGA vendors like Xilinx and Altera now support OpenCL. Our OpenCL implementation could thus straightforwardly be deployed on FPGA, tailoring the FPGA hardware to our algorithm. The performance of this FPGA deployment could then be compared with GPU and CPU deployments, as well as with other FPGA implementations for solving the TMD.

Acknowledgments. This work was supported by the TaMaDi project of the French ANR (grant ANR 2010 BLAN 0203 01). The authors would like to thank A. Zaks and A. Narkis from Intel for helpful discussions on the Xeon Phi and on OpenCL. The authors would also like to thank Pierre-Emmanuel Le Roux (LIP6) for

[5]The AVX2 server has only 4 CPU cores, and no speedup is obtained on CPU thanks to vectorization (in AVX2 as in SSE4.2): we therefore only consider the SSE4.2 server here.

managing the compute servers, and the SFPN specialty (numerical security, reliability and performance) of the master in computer science at *Université Pierre et Marie Curie* for providing access to the AVX2 server.

REFERENCES

[1] AMD, *3DNow! Technology Manual,* 2000.
[2] AMD, *Accelerated Parallel Processing OpenCL Programming Guide,* revision 2.7, November 2013.
[3] N. Brunie, S. Collange and G. Diamos, *Simultaneous Branch and Warp Interweaving for Sustained GPU Performance,* in Proceedings of the International Symposium on Computer Architecture (ISCA'12), 4960, 2012.
[4] M. Cornea, IEEE 754-2008 Decimal Floating-Point for Intel, ARITH, IEEE Symposium on Computer Arithmetic, 2009, pp. 225-228.
[5] K. Diefendorf, *Altivec extension to Power PC accelerates media processing,* 2001.
[6] F. de Dinechin, J.-M. Muller, B. Pasca and A. Plesco, *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, 187194, 2011.
[7] B. Gaster, L. Howes, D.R. Kaeli, P. Mistry and D. Schaa, *Heterogeneous Computing with OpenCL: Revised OpenCL 1.2 Edition,* Newnes, 2012.
[8] P. Fortin, M. Gouicem and S. Graillat, *Towards solving the Table Maker's Dilemma on GPU,* in Proceedings of the 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, pp. 407-415, 2012.
[9] P. Fortin, M. Gouicem and S. Graillat, *GPU-accelerated generation of correctly-rounded elementary functions,* ACM Transactions on Mathematical Software (to appear).
[10] P. Fortin, M. Gouicem and S. Graillat, *GPU-accelerated generation of correctly-rounded elementary functions,* Research Report hal-00751446 v2, `https://hal.archives-ouvertes.fr/hal-00751446`
[11] S. Frey, G. Reina and T. Ertl, *SIMT Microscheduling: Reducing Thread Stalling in Divergent Iterative Algorithms,* in Proceedings of the 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing, 399406, 2012.
[12] W. W. L. Fung, I. Sham, G. Yuan and T. M. Aamodt, *Dynamic Warp Formation: Efficient MIMD Control Flow on SIMD Graphics Hardware,* ACM Trans. Archit. Code Optim. 6(2), Article 7, 2009.
[13] T.D. Han and T.S. Abdelrahman, *Reducing branch divergence in GPU programs,* in Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units, 3:13:8, 2011.
[14] J.L. Hennessy, D.A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach,* The Morgan Kaufmann Series in Computer Architecture and Design, 2011
[15] IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic,* 2008.
[16] Intel Developer Services, *MMX Technology Technical Overview,* 1996.
[17] Intel, *Intel SSE4 Programming Reference,* Reference number: D91561-003, 2007.
[18] Intel, *Intel Architecture Instruction Set Extensions Programming Reference,* Number: 319433-012A, 2012.
[19] Intel, *Intel Architecture Instruction Set Extensions Programming Reference,* Number: 319433-024, 2016.
[20] Intel, *A Guide to Vectorization with Intel C++ Compilers,* 2012
[21] V. Lefèvre, J.-M. Muller and A. Tisserand, *Toward correctly rounded transcendentals,* IEEE Transactions on Computers, 47(11), pp. 1235-1243, 1998.
[22] V. Lefèvre, *New Results on the Distance between a Segment and $\mathbb{Z}^2$. Application to the Exact Rounding,* Proceedings of the 17th IEEE Symposium on Computer Arithmetic, pp. 68-75, 2005.
[23] L. D. McFearin, D. W. Matula, *Generation and Analysis of Hard to Round Cases for Binary Floating Point Division,* ARITH, IEEE Symposium on Computer Arithmetic, 2001.
[24] J. Meng, D. Tarjan and K. Skadron, *Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance,* in Proceedings of the International Symposium on Computer Architecture (ISCA'10), 2010.
[25] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé and S. Torres, *Handbook of floating-point arithmetic,* Springer, 2010.
[26] M. Pharr and W.R. Mark, *ispc: A SPMD compiler for high-performance CPU programming,* In Innovative Parallel Computing (InPar), pp. 1-13, 2012.
[27] N. Rotem, *Intel OpenCL Implicit Vectorization Module,* 2011 LLVM Developers' Meeting
[28] T. Schaub, S. Moll, R. Karrenberg and S. Hack, *The Impact of the SIMD Width on Control-Flow and Memory Divergence,* ACM Trans. Archit. Code Optim., 11(4), 2015.
[29] D. Stehlé, V. Lefèvre and P. Zimmermann, *Searching Worst Cases of a One-Variable Function Using Lattice Reduction,* IEEE Trans. Comput., 54, 340346, 2005
[30] S. Timnat, O. Shacham and A. Zaks, *Predicate Vectors If You Must,* Workshop on Programming Models for SIMD/Vector Processing, 2014, colocated with the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2014
[31] E. Z. Zhang, Y. Jiang, Z. Guo and X. Shen, *Streamlining GPU Applications On the Fly: Thread Divergence Elimination through Runtime Thread-Data Remapping,* in Proceedings of the 24th ACM International Conference on Supercomputing (ICS '10), 2010.

Edited by: Dana Petcu
Received: Apr 7, 2016
Accepted: May 2, 2016