High Performance Numerical Validation using Stochastic Arithmetic

Pacôme Eberhart, Julien Brajard, Pierre Fortin, Fabienne Jézéquel

High Performance Numerical Validation using Stochastic Arithmetic*

P. Eberhart†[1], J. Brajard[3,4], P. Fortin[1], and F. Jézéquel[1,2]

[1]Sorbonne Universités, UPMC Univ Paris 06, CNRS, UMR 7606, Laboratoire d'Informatique de Paris 6 (LIP6), 4 place Jussieu, 75252 Paris CEDEX 05, France
[2]Université Panthéon-Assas, 12 place du Panthéon, 75231 Paris CEDEX 05, France
[3]Sorbonne Universités, UPMC Univ Paris 06, CNRS-IRD-MNHN, LOCEAN Laboratory, 4 place Jussieu, 75005 Paris, France
[4]Inria Paris-Rocquencourt, Domaine de Voluceau, 78150, Le Chesnay, France
{*Pacome.Eberhart,Pierre.Fortin,Fabienne.Jezequel*}*@lip6.fr,*
Julien.Brajard@locean-ipsl.upmc.fr

Abstract

In the context of high performance computing, numerical validation becomes increasingly important because of the higher level of parallelism and of the large number of operations. Our approach, Discrete Stochastic Arithmetic, implemented through the CADNA library, has however a high overhead on execution time, especially for very optimised applications, and does not enable the use of vector instructions. In this paper, we present a new CADNA version that will reduce this overhead by up to 85% for both simple and more realistic benchmarks. This new version also enables the use of vector instructions for an additional speedup between 2.5 and 3 times on the AVX2 instruction set extension.

Keywords: rounding errors, numerical validation, Discrete Stochastic Arithmetic, HPC, SIMD

AMS subject classifications: 65G99, 65Y04, 65Y05

*Submitted: February 20, 2015; Revised: November 27, 2015; Accepted: December 19, 2015.

†Corresponding author

1 Introduction

In floating-point arithmetic, rounding errors occur because of the finite representation of floating-point numbers in computers. When these rounding errors pile up, the result of a floating-point computation can greatly differ from its exact result. In the context of high performance computing, new architectures, becoming more and more parallel, offer higher floating-point computing power. Thus, the size of the problems considered (and with it, the number of operations) increases, becoming a possible cause for increased uncertainty. As such, estimating the reliability of a result at a reasonable cost is of major importance for numerical software.

Various approaches to estimate the propagation of rounding errors include, among others, interval arithmetic, backward error analysis and stochastic arithmetic.

Interval arithmetic [1, 10] replaces all operands in floating-point operations by intervals containing the exact value. These operations give a 100% certain result, represented as an interval containing the exact result. However, these intervals can grow very large as the compensation in rounding errors is not taken into account. To prevent intervals from expanding too much, specific algorithms and methods have been developed [1, 10], but usually require recoding the application. In terms of performance, recent implementations show a good scalability and a low overhead [14].

Backward error analysis [18] considers that instead of an approximate solution to an exact problem, we compute the exact solution to an approximate problem. By studying the behaviour of an application when its entry is perturbed, the direct error can be deduced by an estimation of the condition number [5]. This method has a low overhead in terms of execution time, but does not support every type of problem (it is used mainly for linear problems).

Our approach, Discrete Stochastic Arithmetic [17], considers several executions of the same computation with a randomly chosen rounding mode for each operation. From the samples obtained, we can get, through statistical analysis, a 95 % confidence interval on the number of exact significant digits of the result. The cost of instrumenting a code with CADNA [6], the library implementing discrete stochastic arithmetic, is low as it is mostly done through operator overloading (as it could be done with interval arithmetic).

However, CADNA is currently not adapted to high performance computing. Its overhead on execution time is usually between 10 and 100 times [8] and can go up to several orders of magnitude on highly optimised codes [12]. Furthermore, it cannot use one of the mainstay of parallel computing, vector instructions.

In this paper, we thus present a new CADNA version with several features which will improve its scalar performance and enable the use of vector instructions within the library.

Section 2 provides a more comprehensive explanation of Discrete Stochastic Arithmetic and its implementation through the CADNA library. In section 3, we detail our contributions for performance improvements and the use of vector instructions. In section 4, we provide performance results for both simplified and realistic benchmarks. Finally, we conclude and discuss future works in section 5.

2 Discrete Stochastic Arithmetic and the CADNA Library

2.1 Discrete Stochastic Arithmetic

When no overflow occurs, the exact result, r, of any non exact floating-point arithmetic operation is bounded by two consecutive floating-point values R^- and R^+. When the arithmetic operation is exact, we have $r = R^- = R^+$.

Stochastic arithmetic is based on the CESTAC method (*Contrôle et Estimation Stochastique des Arrondis de Calculs*) [16]. Each arithmetic operation is performed N times, with probability 0.5 to round to R^- or R^+, corresponding, respectively, to rounding towards $-\infty$ and $+\infty$ [4]. In a sequence of arithmetic operations, each rounding error is propagated differently N times, so we get a set of N samples R_i. The value of the computed result $\overline{R}$ is chosen to be the mean value of $\{R_i\}$ and the number of exact significant digits in $\overline{R}$, $C_{\overline{R}}$, is estimated by

$$C_{\overline{R}} = \log_{10}\left(\frac{\sqrt{N}\,|\overline{R}|}{\sigma\tau_\beta}\right) \tag{1}$$

where σ is the standard deviation of $\{R_i\}$ and τ_β is the value of Student's distribution for $N-1$ degrees of freedom and a confidence level $1-\beta$.

The validity of $C_{\overline{R}}$ is compromised if both operands in a multiplication or the divisor in a division are not significant (*i.e.* $C_{\overline{R}} \leq 0$). It is essential, therefore, to test each of these operands, since their lack of significance may invalidate the method. The need for this dynamic control of multiplications and divisions has led to the concept of computational zero. A computed result, $\{R_i\}$, is a computational zero, denoted by @.0, if and only if one of the following two conditions holds [15]:

1. $\forall i, R_i = 0 \quad i = 1, \ldots, N$
2. $C_{\overline{R}} \leq 0$

This means that a computational zero is either the mathematical zero or a number without any significance *i.e.* numerical noise. To establish consistency between the arithmetic operators and the relational operators, discrete stochastic relations have been defined. Let $X = \{X_i\}$ and $Y = \{Y_i\}$ be two results computed using stochastic arithmetic. Then discrete stochastic relations are defined as:

1. $X = Y$ *if and only if* $X - Y = @.0$,
2. $X > Y$ *if and only if* $\overline{X} > \overline{Y}$ *and* $X - Y \neq @.0$,
3. $X \geq Y$ *if and only if* $\overline{X} \geq \overline{Y}$ *or* $X - Y = @.0$.

These definitions take into account the numerical noise and allow the recovery of some coherence between relational and arithmetic operations [3, 2].

Discrete Stochastic Arithmetic (DSA) is defined as the combination of the CESTAC method, the concept of computational zero and the discrete stochastic relations [17].

2.2 The CADNA library

The CADNA (*Control of Accuracy and Debugging for Numerical Applications*)[1] [6] library is an implementation of DSA devoted to programs written in C/C++ and

[1] `http://www.lip6.fr/cadna`

Fortran. In the rest of this paper, we will focus on the C/C++ implementation that provides classes, functions and methods to easily instrument any C/C++ program.

In order to use CADNA, standard floating-point types must be replaced by corresponding stochastic classes (`float_st` in single precision and `double_st` in double precision). Each stochastic variable contains $N = 3$ values of the corresponding floating-point type, one for each sample R_i. In addition to these three fields, there is an integer field, used as a cache for the costly computation of the number of exact significant digits. In practice, the floating-point values R^- and R^+ are obtained using the rounding modes towards $-\infty$ and $+\infty$ defined in the IEEE 754 standard [4]. Arithmetic operators, comparison operators, all the mathematical functions have been overloaded to return a stochastic type when called with stochastic arguments. The C++ print operator `>>` has been overloaded to output the computed result, $\overline{R}$, with only its exact significant digits. Two additional functions `nb_significant_digits` and `is_computed_zero` allow respectively to get the estimated number of exact significant digits (within a $1 - \beta = 95\%$ confidence interval for the result) in the stochastic argument and to test if the argument is a computational zero.

During the execution, when a numerical anomaly is detected, dedicated CADNA counters are incremented. At the end of the run, the value of these counters along with appropriate warning messages are printed on standard output. These warnings are of four types.

1. Self-validation: both operands in a multiplication or the divisor in a division are not significant.
2. Mathematical instability: non significant argument in a mathematical function.
3. Branching instability: indeterminism in a branching test.
4. Cancellation instability: sudden loss of accuracy on an addition or a subtraction.

At the end of the run, each type of anomaly together with its number of occurrence are printed. If no anomaly has been detected, the computed results are reliable and their accuracy has been correctly estimated up to a certain probability. If anomalies have been detected, two cases need to be considered. Self-validation warnings indicate that the validity of $C_{\overline{R}}$ has been compromised and the CADNA results cannot be relied on. Other warnings mean that numerical instabilities have been detected. In both cases the messages need to be analyzed, the source of the anomaly identified and, if necessary, the code changed.

In practice, instrumentation of a C/C++ program with the CADNA library involves five steps:

1. inclusion of the header for the CADNA library with `#include <cadna.h>`
2. initialization of instability detection and internal parameters of CADNA via the `cadna_init` function
3. substitution of the types `float` and `double` with `float_st` and `double_st`, respectively
4. change of output statements to print stochastic results to exact accuracy using `strp` or `>>` operator
5. print of the results of the anomaly detection via the `cadna_end` function

2.3 Performance impact of the current CADNA library

A program that uses the CADNA library executes $N = 3$ times the arithmetic operations and a few additional operations if instability detection is activated. As such, we expect the computation time when using CADNA to be at least 3 times that of the same computation using standard floating-point arithmetic.

However, once a program has been instrumented with the CADNA library, there is a much more important overhead on computation time, up to about 2 orders of magnitude slower [8], depending on the program and on the level of anomaly detection. In highly optimised programs, such as BLAS routines, it can even go up to 1000 times [12].

There are two main factors that can explain this overhead:

- the cost of anomaly detection,
- the cost of stochastic operations.

Anomaly detection is based mostly on the test of whether a stochastic value is significant or not. In the case of cancellation detection, the test depends on the difference of number of exact significant digits between the operands and the result. The significance of a stochastic value can be tested with only arithmetic operations, however the computation of the number of exact significant digits relies on the `log10` function. This mathematical function is much more costly than floating-point arithmetic operations. In cancellation detection, it is used for both arguments and for the result, incurring a strong overhead.

Stochastic arithmetic operations are implemented with the help of standard operations and the explicit change of rounding mode in the FPU (Floating-Point Unit). Changing the rounding mode is of relatively low cost in itself (only a few assembler instructions required: reading, modifying and writing the control word of the FPU), but it flushes the pipelines of the FPU, requiring several processor cycles to refill them. This is especially disadvantageous for CADNA, as up to three rounding mode changes can occur in every stochastic operation. As HPC applications aim to fill these pipelines as much as possible to improve their performance, CADNA has an even more detrimental impact on high performance scientific applications.

Besides, the stochastic operations are implemented by overloading the arithmetic operators for stochastic types. As such, they are defined as functions or methods, instead of being processor instructions. In addition to executing the body of these functions, the function call also requires additional operations such as extending the stack and managing the arguments and can also prevent pipelining successive operations. Again, this overhead is particularly severe in high performance scientific codes.

Finally, as SIMD parallelism (*Single Instruction Multiple Data*) is increasingly important for high performance computing, CADNA should enable the use of vector instructions. However, the reliance on the rounding mode of the hardware makes it impossible to use SIMD parallelism with the current CADNA version. Indeed, vector units such as SSE (128 bits wide), AVX (256 bits wide), or the Xeon Phi ones (512 bits wide) only enable the control of the rounding on the whole vector, not on a lane by lane basis. This would result in the same rounding mode being selected for operations on the same vector, breaking the hypothesis of CESTAC that the rounding mode should be chosen independently for each operation. It is one of the reasons the current version of CADNA is unable to use vector instructions.

In the remainder of this paper, we will therefore focus on overcoming the aforementioned problems, improving the performance of CADNA on one CPU core. It has

to be noticed that there exists a parallel CADNA version for distributed memory architectures based on MPI [12]. The current work will thus directly benefit this version of CADNA.

3 A New CADNA Version for HPC Applications

We will now present our improvements for the CADNA library in order to reduce the overhead and enable the vectorisation of CADNA. In subsection 3.1, we will replace the `log10` with an approximation. In subsection 3.2, we will propose to stop changing the rounding mode of the FPU for each operation and rather emulate one rounding from another thanks to integer and logical operations. In subsection 3.3, we will remove the overhead due to function calls by inlining them. In subsection 3.4, we will change the random number generator of CADNA to enable vectorisation. We will vectorise the CADNA library for a specific programming paradigm detailed in subsection 3.5. Finally, we will try to reduce the divergence in control flow introduced by CADNA in SIMD codes in section 3.6.

3.1 Approximation of `log10` in the cancellation detection

Cancellation detection relies on the estimated number of exact digits of the two stochastic arguments and the stochastic result of addition or subtraction operations. The computation of this number is given by Eq.1.

All the constants can be computed at compile time, thus leaving only the mean and standard deviation to depend on the execution. Equation 1 also requires the use of two transcendental functions: `sqrt` (for σ) and `log10`. Fortunately, due to the fact that $log_{10}(\sqrt{x}) = \frac{log_{10}(x)}{2}$, we only need to use the `log10` function.

However, even one call to a transcendental function in this computation can be very detrimental to the performance of cancellation detection. Indeed, this process must be applied three times (once for each argument and once for the result) for each addition or subtraction. Moreover, we only need the integer part of the result of the `log10` function (a digit can only be significant or non significant).

We therefore propose a faster method to get the integer part of the `log10` evaluation by considering the argument in its base 10 floating-point form $x = m_{10} \times 10^{e_{10}}$, with $1 \leq m_{10} < 10$ and $e_{10} \in \mathbb{Z}$. We then have

$$\log_{10}(x) = \log_{10}(m_{10} \times 10^{e_{10}}) = \log_{10}(m_{10}) + e_{10} \tag{2}$$

Since $e_{10} \in \mathbb{Z}$, and since $1 \leq m_{10} < 10$ implies $0 \leq \log_{10}(m_{10}) < 1$, we conclude that

$$\lfloor \log_{10}(x) \rfloor = e_{10} \tag{3}$$

Nevertheless, the base 10 exponent is difficult to obtain from the binary representation of the floating-point number. Denoting $x = m_2 \times 2^{e_2}$ as the base 2 floating-point form of the argument, we propose to approximate e_{10} by the quantity $\lfloor e_2 \times \log_{10}(2) \rfloor$ and show the following property.

Property 3.1 *Let $x \in \mathbb{R}$, $x > 0$.*
We denote $m_{10} \in \mathbb{R}$, $1 \leq m_{10} < 10$, and $e_{10} \in \mathbb{Z}$, such as $x = m_{10} \times 10^{e_{10}}$, the base 10 representation of x and $m_2 \in \mathbb{R}$, $1 \leq m_2 < 2$, and $e_2 \in \mathbb{Z}$, such as $x = m_2 \times 2^{e_2}$,

the base 2 representation of x.
Then, we have $e_{10} - 1 \leq \lfloor e_2 \times \log_{10}(2) \rfloor \leq e_{10}$.

Proof:

$$\log_2(x) - 1 < \log_2(x)$$

As $\log_2(x) = e_2 + \log_2(m_2)$, we obtain

$$\log_2(x) - 1 < e_2 + \log_2(m_2) = \log_2(x)$$

We multiply by $\log_{10}(2)$ and subtract $\log_{10}(m_2)$

$$\log_{10}(x) - \log_{10}(2) - \log_{10}(m_2) < e_2 \times \log_{10}(2) = \log_{10}(x) - \log_{10}(m_2) < \log_{10}(x)$$

We apply the integer part

$$\lfloor \log_{10}(x) - \log_{10}(2) - \log_{10}(m_2) \rfloor \leq \lfloor e_2 \times \log_{10}(2) \rfloor \leq \lfloor \log_{10}(x) \rfloor$$

Since $\log_{10}(2) + \log_{10}(m_2) < 1$, we have

$$\lfloor \log_{10}(x) \rfloor - 1 \leq \lfloor e_2 \times \log_{10}(2) \rfloor \leq \lfloor \log_{10}(x) \rfloor$$

From Eq. 3, we conclude

$$e_{10} - 1 \leq \lfloor e_2 \times \log_{10}(2) \rfloor \leq e_{10}$$

□

Our approximation gives a more pessimistic estimation (being less or equal to the exact evaluation), ensuring that it will not declare more exact significant digits than the exact evaluation. The approximation is also not overly pessimistic, having at most a difference of one exact significant digit with the `log10` based implementation. Moreover, we emphasize that our approximation is much faster than the `log10` evaluation, since we only need a few logical and integer instructions to extract the exponent, followed by a floating-point multiplication by a pre-computed constant ($log_{10}(2)$) and a cast to integer.

3.2 Changing the rounding mode

The random rounding mode of CADNA relies on changing the rounding mode of the FPU. As this flushes the pipelines of the FPU, it is very costly, especially when used repeatedly on fast operations, such as arithmetic operations. This way of implementing the random rounding mode also prevents vectorisation for codes instrumented with CADNA, the same rounding mode being assigned to every lane of the vector. Finally, it forces to disable any optimisation when compiling the CADNA code with the gcc compiler, as optimising may generate incorrect code with gcc when changing the rounding mode, even when using the -frounding-math option [2].

We thus propose in this new CADNA version to emulate the rounding modes toward infinity taking advantage of the following properties:

- $a \oplus_{+\infty} b = -(-a \oplus_{-\infty} -b)$ (similarly for $\ominus$)
- $a \otimes_{+\infty} b = -(a \otimes_{-\infty} -b)$ (similarly for $\oslash$)

[2] GCC bug 34678 - optimization generates incorrect code with -frounding-math option. `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=34678`

where $\oplus_{+\infty}$ and $\otimes_{+\infty}$ (resp. $\oplus_{-\infty}$ and $\otimes_{-\infty}$) are the floating-point operations rounded towards $+\infty$ (resp. $-\infty$). Since the results of each rounding mode can be obtained from computation made in the other rounding mode, there is no need to change the rounding mode of the FPU during the execution of the program. We only require to set the rounding mode towards $+\infty$ or $-\infty$ once, in the `cadna_init` function.

As our goal is also to enable SIMD parallelism, we will implement this solution without introducing divergence in the execution flow. To do so, instead of using `if` blocks depending on the chosen rounding mode, we could multiply the operands and the results, according to the aforementioned properties, by 1 or -1. Although it avoids the divergence, this would comes at the cost of two or three floating-point multiplications for each sample of the stochastic value. Instead, we will apply a random mask to the sign bit of the binary representation of the floating-point numbers to change their sign as required, without relying on the multiplication.

As there is no more rounding mode change in the computation part of the application code, we can use optimisation options of `gcc` without the risk of floating-point instructions being moved and executed in an unintended rounding mode. This enables the optimisation of the CADNA library for high performance.

3.3 Inlining

CADNA functionalities are implemented by overloading arithmetic operators and mathematical functions. Although the overhead for functions can be negligible, the additional cost of a function call for each arithmetic operator can be high, not to mention the impact on the filling of the pipelines. As the code of the stochastic operators of CADNA version 1.1.9 is compiled in the library, it is not available in the CADNA header for inlining in the application code. To enable inlining, we have moved the code of these operators to the header of the CADNA library. However, it would be inconvinient to apply this optimisation without first removing the change of the rounding mode during the execution of the stochastic operations, as the required assembly code would imply a different header for each architecture.

By inlining the arithmetic operators, we can get rid of much of the overhead of CADNA due to function calls. Moreover, it enables optimisations such as pipelining several stochastic operations, or interleaving their instructions, contrary to the previous version where it would require much more difficult interprocedural optimisation.

3.4 Random generator

The selection of the rounding mode for stochastic operations is based on the value of a randomly generated bit. The 1.1.9 version of CADNA uses an intrinsically sequential, and difficult to vectorise, method. An array is pre-filled with randomly generated numbers during the CADNA initialisation, and bits are picked by sequentially reading each number bit per bit. In a SIMD context, this implies to compute a different bit index for each lane and to increment the index according to the vector width.

To account for any possible width of vector and to have a straightforward and efficient vectorisation, we have chosen to replicate the random generator for each lane. However, instead of pre-generating an array that would be duplicated, the random number generation will now be executed on the fly. As such, an integer will be randomly generated and read bit by bit for each random pick. When every bit has been picked, every lane will produce a new number at the same time, as the bits were consumed at the same rate on each lane. Where the previous version of CADNA used a

16-bit `short` integer generator, we now use the generator presented in Mohanty and al. [11] which produces 32-bit `int` values. This new generator also has a longer period, a good statistical distribution, and uses only integer addition, multiplication, and logical operations, whereas the previous also used integer division. Integer division is usually relatively inefficient and most vector instructions sets (such as SSE, AVX and the Xeon Phi ones) do not contain integer division.

The dynamic generation will have the added benefit of reducing the memory footprint and memory accesses of CADNA, at the cost of slightly more computation. As computation is becoming increasingly cheaper than memory accesses on current and future HPC architectures, this should also yield an improvement in the performance of CADNA instrumented applications.

3.5 Vectorising

Now that the main prerequisites have been achieved, we can deal with vectorisation. For this, several programming paradigms are possible: intrinsics, automatic vectorisation, compiler directives and the SPMD-on-SIMD (*Single Program Multiple Data*) programming model.

Intrinsics enable the use of vector instructions without using assembly language by relying on vector-specific functions. Using such intrinsics is always possible, but rather tedious in general: one has to write specific code for each intrinsic set (SSE, AVX, ...) and each vector width. With CADNA, this is even more tedious as we have to handle the composite data types and replace each intrinsic call with a corresponding CADNA version.

Automatic vectorising is a compilation technique in which the compiler analyses the code and decides whether it is possible and efficient to vectorise it. Such automatic vectorising, due to its automatic nature, needs to ensure that the dependencies of the scalar code are respected when vectorising. For instance, in IEEE floating-point arithmetic, the iterations of the loop of Program 1 are independent from each other and can be automatically vectorised. However, with CADNA, even though the variables in these operations are completely different, the process of choosing the rounding mode creates a new dependency. Indeed, the random bit chosen for one iteration is necessarily picked after the previous iteration. As such, automatic vectorising can never be achieved for any code instrumented with CADNA.

Program 1 Loop of independent floating-point operations

```
for (int i = 0; i < n; i++)
  c[i] = a[i] + b[i];
```

The new version of OpenMP[3] (OpenMP 4.0), as well as the Intel compiler, contain compilation directives aimed at vectorising loops. Using such directive-assisted vectorisation, we could force the vectorisation and circumvent the limitation of automatic vectorisation. To ensure that the randomness of the rounding mode is retained, we must duplicate the random generator on each lane. However, there is no lane identifier, necessary to access each generator independently.

We thus focus on the SPMD-on-SIMD model, where all computations are written as scalar ones and it is up to the compiler to merge such scalar computations in SIMD

[3] `http://openmp.org/wp/`

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 zeroes according to this vector width. Such programming paradigm is increasingly used in HPC: first on GPU with CUDA [4] and then on various devices with OpenCL[5]. Moreover, there is a lane identifier that enables us to easily replicate the random generator (with a different seed for each lane). On CPU, such programming model is available in OpenCL (OpenCL implicit vectorization), as well as in the Intel SPMD Program Compiler (`ispc`)[6] [13]. We have elected to choose `ispc` over OpenCL, as OpenCL does not currently support the overloading of operators necessary for CADNA. Nevertheless, the same process could be applied for other SPMD-on-SIMD languages, as long as operator overloading is supported.

Thanks to our previous contributions, very few changes are necessary to adapt the CADNA library to `ispc`. Indeed, adding relevant ispc attributes to variables (`varying` for lane specific variables, `uniform` for vector shared ones) and initialising the seed for each lane were the only necessary adjustments. Like C++ code, ispc code can be instrumented with CADNA by simply changing the types of the variables to stochastic ones.

3.6 Execution masks

When a vectorised code that contains conditional branches (`if`) is compiled, the compiler usually uses an execution mask, so that each lane still executes the same instruction, but does not commit to memory or registers when it is in a branch it should not execute. However, this process is generally implemented through the software (e.g. for SSE and AVX) and can be costly in terms of performance.

In the current version of CADNA, the types of instabilities that are detected are chosen during the execution, by using a parameter of the `cadna_init` function. As such, detection flags are set up once and checked dynamically, creating conditional branches on the part of the CADNA code dedicated to anomaly detection. However, when a given anomaly is not detected, these branches can still produce execution masks when vectorising and compiling.

To try to reduce the impact of execution masks, we will replace the tests in these branches by preprocessor directives that can be evaluated at compile time. We can still change the type of instabilities to detect by changing compilation options, however this method disables the possibility of changing it during execution.

4 Performance Results

4.1 Experimental setup

To assess the impact of our different improvements on the performance of CADNA, we will first measure the overhead of several scalar benchmarks instrumented with different versions of the library. Then we will use the same benchmarks, that we will vectorise with `ispc`, and measure the performance speedups with respect to the scalar versions.

[4] `https://developer.nvidia.com/cuda-zone`
[5] `http://www.khronos.org/opencl/`
[6] `http://ispc.github.io/`

In the context of HPC, applications are generally classified in two categories depending on which resource limits their performance:

- compute-bound applications are limited by computational power;
- memory-bound applications are limited by memory bandwidth.

This classification mainly depends on the ratio of number of floating-point operations over number of memory accesses, and on the underlying architectures. To examine the behaviour of CADNA on high performance computations, our benchmarks must include both compute-bound and memory-bound applications.

We have chosen two different types of benchmarks. The first is aimed at showing the performance of the CADNA library from a purely arithmetic point of view. As such, we will perform only one arithmetic operation and repeat it with a high number of floating-point numbers. To do so, we will add (and multiply) long vectors (several millions of elements). We will have both compute-bound versions (see Program 2) and memory-bound versions (see Program 3). For both programs, n is the size of the array and k is the number of times we repeat the operation on the same element. We have chosen $k = 128$, to ensure that the execution time is stable and that Program 3 is actually compute-bound. In practice, we have chosen $n = 2^{24}$ so that our array would not fit in the memory caches and distort our results. The second type of benchmark is aimed at showing the performance of CADNA in more realistic applications. Therefore, we will use a Mandelbrot set computation (compute-bound) and a 3D finite difference stencil computation (memory-bound) representing a wave propagation simulation. For the Mandelbrot computation, the use of CADNA will allow us to better determine if the sequence corresponding to a specific point of the 2D plane is bounded or not. The stencil computation will also benefit from the use of CADNA, as the wave propagation can lose up to all digits in accuracy on successive iterations [9]. For all these benchmarks, all floating-point numbers will be single precision (`float` and `float_st`). The codes for the Mandelbrot and stencil computations have been taken from the examples distributed with `ispc`.

Program 2 Compute-bound code for the addition benchmark

```
for (int i = 0; i < n; i++)
  for (int j = 0; j < k; j++)
    a[i] = b[i] + a[i];
```

Program 3 Memory-bound code for the addition benchmark

```
for (int j = 0; j < k; j++)
  for (int i = 0; i < n; i++)
    a[i] = b[i] + a[i];
```

4.2 Scalar performance

It is not necessary to test each possible scalar optimisation in an isolated way. Firstly, applying inlining without first removing the change of rounding mode during computation would probably yield limited gain in performance, as the main restricting

factor, the flushing of the FPU pipelines, would still hinder performance. Secondly, it is not safe to apply compiler optimisations on CADNA version 1.1.9. Finally, the new random generator is not meant to improve performance, it aims to prepare for vectorisation. As such, we will only apply it to the best performing version. We will thus measure our scalar performance in the following incremental manner.

For each benchmark, we will compare a version implemented with IEEE arithmetic (*IEEE*) to several versions of the same code, instrumented with different versions of the CADNA library:

- the current version (named *1.1.9*, according to the last version number),
- using the logarithm approximation (*log approx*),
- using *log approx* and basing the computation on one rounding mode and masks using the `gcc -O0` flag for no optimisation (*mask O0*),
- using *mask O0* and using a high level of compiler optimisation with the `-O3 gcc` flag (*mask 03*)
- using *mask O3* and adding the inlining (*inline*),
- using *inline* and changing the random generation to be dynamic (*dyn*).

The platform for our scalar performance tests is an Intel Xeon E3-1275 CPU clocked at 3.5 GHz. The benchmarks will be compiled with `gcc` version 4.9.2 and optimised with -O3. The CADNA libraries will be compiled with `gcc` version 4.9.2; versions *1.1.9* and *log approx* will not be optimised (using the -O0 flag) due to the aforementioned `gcc` bug, whereas other CADNA versions will be optimised with the -O3 flag. When using the `-O3` flag, we will disable the optimisation for automatic vectorisation to ensure that performance is measured on a scalar code.

We start with the performance test for the cancellation detection, where only the addition benchmark will be used and the library versions will be *1.1.9* and *log approx*. The CADNA library will enable here every instability detection.

As seen in Fig. 1, the overhead of executing using CADNA while enabling every instability detection is very high. However, with our logarithm approximation, we managed to reduce the overhead over the IEEE computation by 43%.

For the rest of the benchmarks, only self-validation (unstable multiplications and divisions detection) will be activated. Indeed cancellation detection is not necessary to ensure the validity of the CESTAC method. As such, in the following, we will not detail the performance of the *log approx* library.

For the compute-bound arithmetic benchmarks, we see in Fig. 2 that the overhead on multiplications is higher than for additions, because self-validation has an impact only on multiplications and not on additions. An instability is detected in a multiplication if both operands have no exact significant digit. If one operand is significant, the accuracy of the other one is not computed. In the best case scenario, the first operand we check has not been modified since we checked it last, and we can confirm whether the multiplication is stable by reading only the already computed accuracy field of this first operand. However, so as to not underestimate the cost of self-validation, we have chosen the worst case scenario, that first checks the operand that has been modified and needs thus additionnal computation. We also note that our successive modifications to the CADNA library significantly improve the performance. Most of the total gain in performance is gained from the *mask O3* version, due to the combined effects of compiler optimisation and the absence of change in the rounding mode of the FPU. Performance further increases with the *inline* version, that, when combined

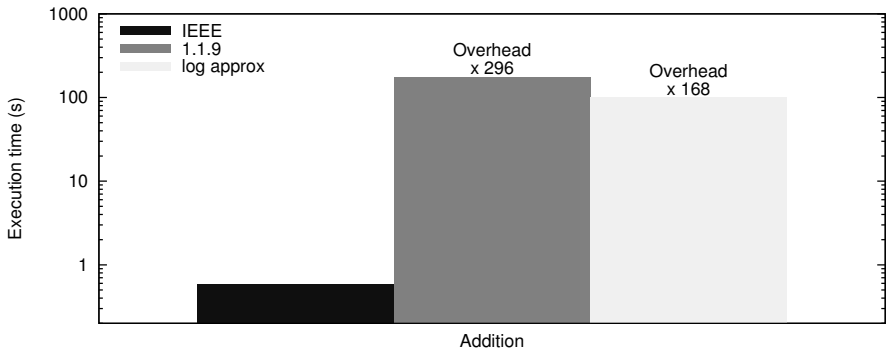

Figure 1: Scalar performance (logarithmic scale) for cancellation detection on the compute-bound addition benchmark and overhead over IEEE performance for CADNA versions

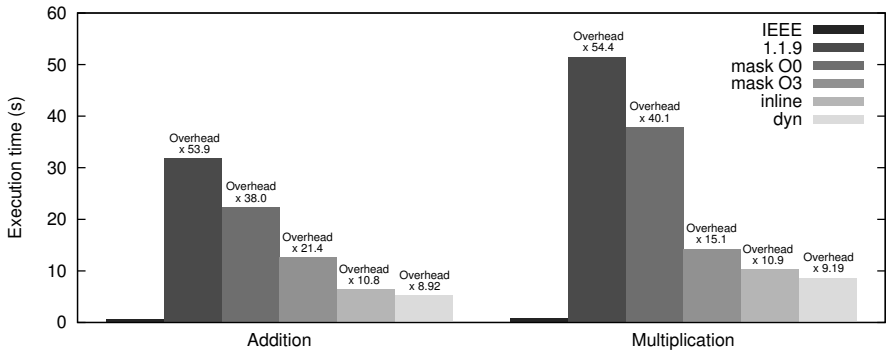

Figure 2: Scalar performance for operations (compute-bound) and overhead over IEEE performance for CADNA versions

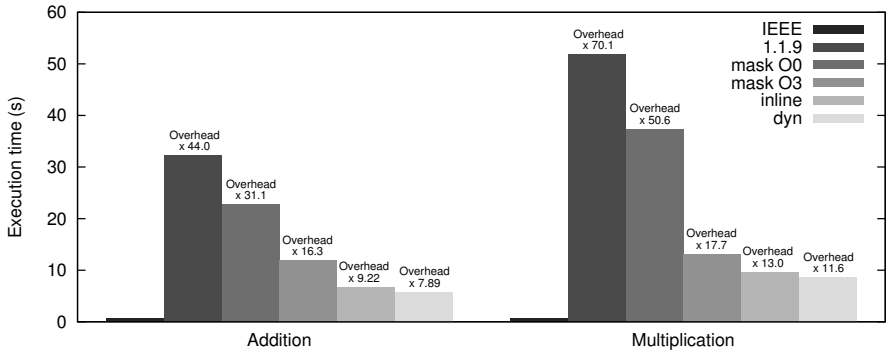

Figure 3: Scalar performance for operations (memory-bound) and overhead over IEEE performance for CADNA versions

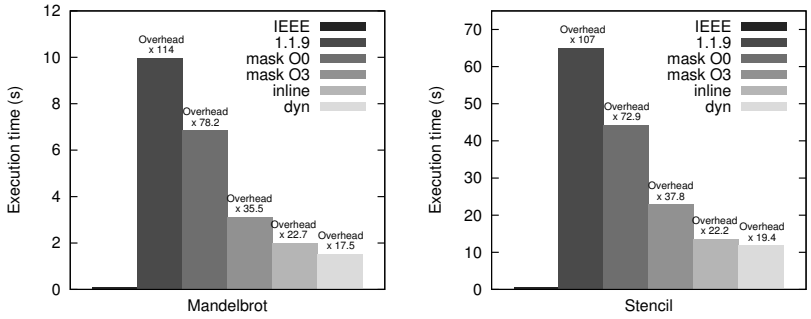

Figure 4: Scalar performance for applications and overhead over IEEE performance for CADNA versions

with previous improvements, allows a tighter integration of the CADNA code in the application code. Moreover, the *dyn* version also slighlty improves performance, even though its main focus was to prepare the random generator for vectorisation. Overall, we reduced the overhead both on the addition benchmark and on the multiplication benchmark by 83%.

For the memory-bound arithmetic benchmarks, we see in Fig. 3 the same behaviour as with the compute-bound benchmarks. Each optimisation (*mask O0*, *mask 03*, *inline*) leads to a significant improvement in performance. We can also see that the overhead of the addition and multiplication benchmarks are similar to their compute-bound equivalent. This is due to the fact that the arithmetic intensity of those benchmarks is altered by the use of CADNA. Indeed, we have at least 3 times more computations (for each sample and for the mask operations), while needing 4 memory accesses. But these memory accesses can be performed at once with the same cache line (as the members of the stochastic types are contiguous in memory): the arithmetic intensities of our memory-bound benchmarks thus increase with CADNA, which brings them closer to the compute-bound benchmarks. In the end, similarly to the compute-bound case, the overhead was reduced by 82% on addition and by 83% on multiplication.

On realistic applications, we see from Fig. 4 that performance has much improved too. However, the overhead is higher than for our arithmetic benchmarks. This can be explained by the nature of our applications. The Mandelbrot set computation is even more arithmetic intensive than our arithmetic benchmarks. Indeed, for each iteration, there are more floating-point instructions than in the arithmetic benchmarks, and when the code is instrumented with CADNA, the overhead increases more. The stencil computation is memory-bound but contrary to the memory-bound arithmetic benchmarks, the 3D memory access pattern lowers the effectiveness of the memory cache and prefetch especially for the CADNA versions. Nevertheless, the overhead was reduced by 85% on the Mandelbrot set computation and by 82% on the finite difference stencil.

Overall, we see that we decrease the overhead on our different benchmarks between 82% and 85% with the *dyn* version. We will thus use this version as the basis for our vectorised versions.

4.3 Vectorised performance

For the vectorised performance tests, we will use a single core of the Intel Xeon E3-1275 CPU, and use its AVX2 instruction set for vector instructions (256-bits wide, 8 `float` per vector).

We will keep the same benchmarks. Vectorised versions of the memory-bound arithmetic benchmarks however, show little to no gain over the corresponding scalar versions (tests not presented here). Indeed, the performance is limited here by the bandwidth of the caches and the memory prefetch: fetching several elements at a time in the cache (vector load) rather than one at a time (scalar load) does not significantly improve performance here.

The versions of the CADNA library tested will be:

- a vectorised version of the scalar *dyn* version (also called *dyn*),
- a version using *dyn* and anomaly detection tests with `#define` evaluated at compile time (*define*) as presented in subsection 3.6.

The vectorised benchmarks for the AVX2 instruction set are compiled and optimised with `ispc` version 1.8.2.

On the AVX2 instruction set, we can see from Fig. 5 that the IEEE speedup on vectorisation is almost maximum, AVX registers containing 8 `float`. We have also achieved vectorisation for CADNA with speedups up to 3.64.We emphasize here that no vectorisation was possible with the previous CADNA version. The CADNA speedups are however lower than the IEEE ones. After having analysed the execution of these benchmarks with the Intel VTune Amplifier profiler[7], we have found that this is due to the memory accesses that are much more costly with CADNA. This can be explained by the composite nature of our stochastic types. Indeed, when the IEEE version is vectorised, bringing data from the memory to the registers can be done in a single vector load, the elements being contiguous in memory. However, with stochastic types, when we load a field of our structure, the values are separated by the other fields of the structure. This layout (Array of Structures, or AoS) requires special memory loads (*gather*) and stores (*scatter*) that are less efficient than simple vector loads and stores. Finally, we also see the beneficial effect of removing the execution masks from our code with the *define* version. As self-validation is the only instability detection

[7] `https://software.intel.com/en-us/node/529213`

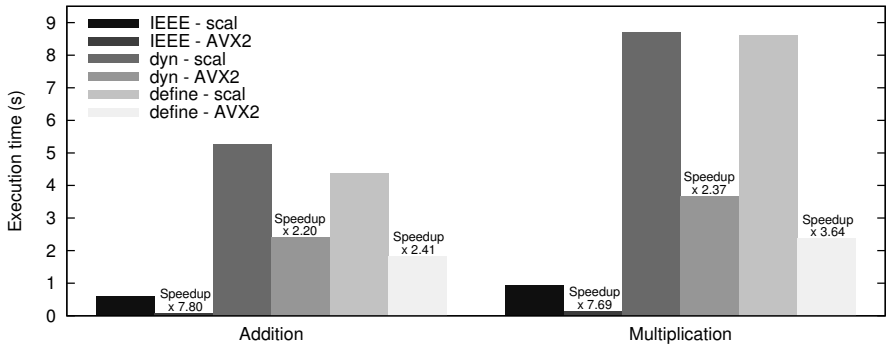

Figure 5: Vectorised performance for operations on AVX2 (compute-bound)
Speedup: speedup on scalar version

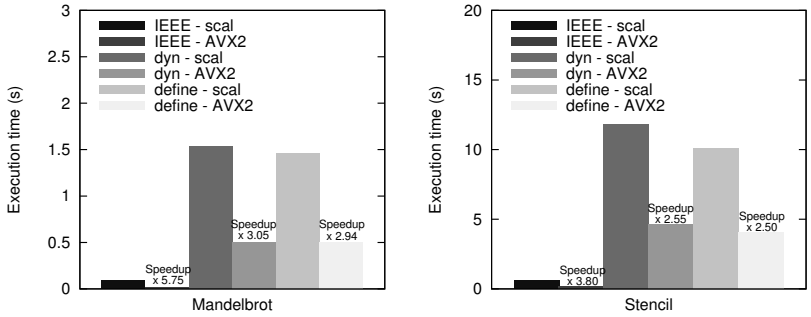

Figure 6: Vectorised performance for applications on AVX2
Speedup: speedup on scalar version

activated, the multiplication benchmark is the only one that creates execution masks during the computation. As such, it benefits from this improvement much more than the addition.

On realistic applications, we see from Fig. 6 we achieve a speedup of up to 3.05. We also observe that the speedup on the Mandelbrot set computation with the *dyn* version is slightly higher than for our other benchmarks. This confirms that the AoS memory layout is partially responsible for the lower CADNA speedup, as this application is the only one among those we used that does not need to load stochastic values from memory. We can also notice that the *define* version improves performance on both benchmarks, without improving the speedup due to vectorisation.

On the whole, our vectorised CADNA versions have a global overhead on the IEEE vectorised versions that varies from a factor 19.2 to 32.4 depending on the benchmark. Although they are higher than for the scalar versions, there is still a net improvement over the former (non-vectorisable) version of CADNA which has an overhead between 407 and 651 over the vectorised IEEE benchmarks.

It can be noticed that we have also run tests on a single core of the Intel Xeon Phi 5110P co-processor, clocked at 1.053 GHz (512-bits wide, 16 `float` per vector)

with ispc. As `ispc` cannot directly compile for the Xeon Phi, its support being in beta status, it instead generates code that is compiled with the Intel C++ compiler (`icc`) version 15.0.1. Unfortunately, the use of the `-fp-model strict` flag, necessary for CADNA because the rounding mode is not toward nearest, currently forces the generation of scalar x87 floating-point instructions[8]. As soon as `icc` is able to generate vector code with the `-fp-model strict` flag or `ispc` support of the Xeon Phi enables direct compilation of binary objects, we believe that we will get better SIMD speedups on this new HPC architecture. Its vector units are indeed wider and the hardware supports execution masks for SIMD divergence, as well as scatter and gather operations [9].

Overall, the vectorisation of CADNA brings a significant additional speedup. The *define* version can perform better than the *dyn* version, but prevents the dynamical change of the anomaly detection mode.

5 Conclusion

Through our successive modifications to the CADNA library, we have improved the scalar performance significantly, reducing its overhead by up to 85 have also enabled vectorisation with the SPMD-on-SIMD programming paradigm with an additional speedup between 2.5 and 3. With vectorisation enabled, we make numerical validation possible for a wider variety of architectures and codes used in high performance computing.

This work could be straightforwardly extended to the Xeon Phi as soon as the Intel C++ compiler enables the generation of vector code with its strict floating-point model, needed when the rounding mode is not toward the nearest. This could also benefit to other methods based on rounding mode change, such as interval arithmetic.

Our future prospects include the development of a new CADNA version supporting shared memory (OpenMP). For this, we will need to enable instability detection in a multithreaded environment and ensure that the rounding mode is the same between the threads. We expect to implement our vectorised version in OpenCL when operator overloading and rounding mode change become supported. This will offer a new SPMD CADNA version portable on both CPU and GPU. This will also enable the study of its performance impact on GPUs (which rely on partial SIMD execution) and its comparison to the existing CUDA prototype [7] based on fixed rounding mode instructions and on the previous CADNA version.

References

[1] G. Alefeld and J. Herzberger. *Introduction to Interval Analysis*. Academic Press, 1983.

[2] J.-M. Chesneaux. The equality relations in scientific computing. *Numerical Algorithms*, 7(2):129–143, September 1994.

[8] Differences in Floating-Point Arithmetic Between Intel Xeon Processors and the Intel Xeon Phi Coprocessor, Intel White Paper, available at: `https://software.intel.com/file/420203/download`

[9] Intel Xeon Phi Coprocessor Vector Microarchitecture, available at: `https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-vector-microarchitecture`

[3] Jean-Marie Chesneaux. Stochastic arithmetic properties. In *Computational and Applied Mathematics, I. Algorithms and Theory. Selected and revised papers from the IMACS 13th World Congress, Dublin, Ireland, July 1991*, pages 81–91. North-Holland, 1992.

[4] IEEE P-754 Working Group. *ANSI/IEEE Std 754-1985: IEEE Standard for Binary Floating-Point Arithmetic*. IEEE, 1985.

[5] N.J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, second edition, 2002.

[6] F. Jézéquel and J.-M. Chesneaux. CADNA: a library for estimating round-off error propagation. *Computer Physics Communications*, 178(12):933–955, 2008.

[7] F. Jézéquel and J.-L. Lamotte. Numerical validation of Slater integrals computation on GPU. In *14th International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics (SCAN 2010)*, pages 78–79, 2010.

[8] F. Jézéquel, J.-L. Lamotte, and O. Chubach. Parallelization of discrete stochastic arithmetic on multicore architectures. In *Tenth International Conference on Information Technology: New Generations (ITNG)*, pages 160–166, April 2013.

[9] F. Jézéquel, P. Langlois, and N. Revol. First steps towards more numerical reproducibility. In *SMAI'2013: 6ème biennale des Mathématiques Appliquées et Industrielles*, pages 001–010, 2013.

[10] U.W. Kulisch. *Advanced Arithmetic for the Digital Computer*. Springer-Verlag, Wien, 2002.

[11] S. Mohanty, A. K. Mohanty, and F. Carminati. Efficient pseudo-random number generation for Monte Carlo simulations using graphic processors. *Journal of Physics: Conference Series*, 368(1), June 2012.

[12] S. Montan, J.-M. Chesneaux, C. Denis, and J.-L. Lamotte. Towards an efficient implementation of CADNA in the BLAS : Example of DgemmCADNA routine. In *15th GAMM - IMACS International Symposium on Scientific Computing, Computer Arithmetic, and Validated Numerics (SCAN)*, December 2012.

[13] M. Pharr and W.R. Mark. ispc: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing (InPar), 2012*, pages 1–13, May 2012.

[14] N. Revol and P. Théveny. Parallel implementation of interval matrix multiplication. *Reliable Computing*, 19(1):91–106, 2013.

[15] J. Vignes. Zéro mathématique et zéro informatique. *Comptes Rendus de l'Académie des Sciences - Series I - Mathematics*, 303:997–1000, 1986. also: La Vie des Sciences, 4 (1) 1-13, 1987.

[16] J. Vignes. A stochastic arithmetic for reliable scientific computation. *Mathematics and Computers in Simulation*, 35(3):233–261, September 1993.

[17] J. Vignes. Discrete stochastic arithmetic for validating results of numerical software. *Numerical Algorithms*, 37(1-4):377–390, December 2004.

[18] J. H. Wilkinson. *Rounding Errors in Algebraic Processes*, volume 32. HMSO, London, 1963.