Estimation of numerical reproducibility on CPU and GPU

Fabienne Jézéquel[1], Jean-Luc Lamotte[1] & Issam Said[2]

[1]LIP6, Université Pierre et Marie Curie

[2]Total & LIP6, Université Pierre et Marie Curie

8th Workshop on Computer Aspects of Numerical Algorithms (CANA'15)
Lodz, Poland
13-16 September 2015

Numerical reproducibility

Numerical reproducibility failures:

- from one architecture to another
- inside the same architecture.

different orders in the sequence of instructions
$\Rightarrow$ different round-off errors

differences in results may be difficult to identify: round-off errors or bug?

Stochastic arithmetic can estimate which digits in the results are different from one execution to another because of round-off errors.

Outline

1. Reproducibility failures in a wave propagation code
2. Principles of stochastic arithmetic
3. Stochastic arithmetic for CPU simulations
4. Stochastic arithmetic for CPU-GPU simulations
5. The wave propagation code examined with stochastic arithmetic

Reproducibility failures in a wave propagation code

For oil exploration, the 3D acoustic wave equation

$$\frac{1}{c^2}\frac{\partial^2 u}{\partial t^2} - \sum_{b\in x,y,z}\frac{\partial^2}{\partial b^2}u = 0$$

where u is the acoustic pressure, c is the wave velocity and t is the time

is solved using a finite difference scheme

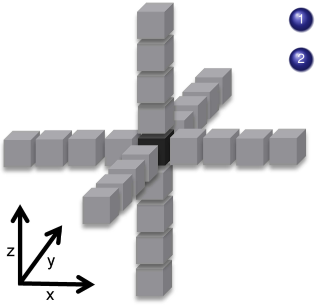
- time: order 2
- space: order p (in our case $p = 8$).

2 implementations of the finite difference scheme

1. $$u_{ijk}^{n+1} = 2u_{ijk}^{n} - u_{ijk}^{n-1} + \frac{c^2\Delta t^2}{\Delta h^2} \sum_{l=-p/2}^{p/2} a_l \left(u_{i+ljk}^{n} + u_{ij+lk}^{n} + u_{ijk+l}^{n} \right) + c^2 \Delta t^2 f_{ijk}^{n}$$

2. $$u_{ijk}^{n+1} = 2u_{ijk}^{n} - u_{ijk}^{n-1} + \frac{c^2\Delta t^2}{\Delta h^2} \left(\sum_{l=-p/2}^{p/2} a_l u_{i+ljk}^{n} + \sum_{l=-p/2}^{p/2} a_l u_{ij+lk}^{n} + \sum_{l=-p/2}^{p/2} a_l u_{ijk+l}^{n} \right) + c^2 \Delta t^2 f_{ijk}^{n}$$

where u_{ijk}^{n} (resp. f_{ik}^{n}) is the wave (resp. source) field in (i, j, k) coordinates and n^{th} time step and $a_{l \in -p/2, p/2}$ are the finite difference coefficients.



1. nearest neighbours first
2. dimension 1, 2 then 3

Reproducibility problems

- differences from one implementation of the finite difference scheme to another
- differences from one execution to another inside a GPU

  repeatability problem due to differences in the order of thread executions
- differences from one architecture to another

In binary 32, for $64 \times 64 \times 64$ space steps and 1000 time iterations:

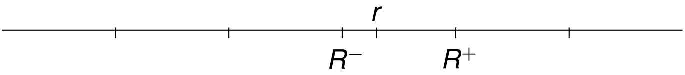
- any two results at the same space coordinates have 0 to 7 common digits
- the average number of common digits is about 4.

Results computed at 3 different points

scheme	point in the space domain		
	$p_1 = (0, 19, 62)$	$p_2 = (50, 12, 2)$	$p_3 = (20, 1, 46)$
AMD Opteron CPU with gcc			
1	**-1.110**479E+0	**5.454**238E+1	**6.141038**E+2
2	**-1.110**426E+0	**5.454**199E+1	**6.14103**5E+2
NVIDIA C2050 GPU with CUDA			
1	**-1.110**204E+0	**5.454**224E+1	**6.1410**46E+2
2	**-1.10**9869E+0	**5.454**244E+1	**6.1410**47E+2
NVIDIA K20c GPU with OpenCL			
1	**-1.10**9953E+0	**5.454**218E+1	**6.1410**44E+2
2	**-1.11**1517E+0	**5.454**185E+1	**6.1410**24E+2
AMD Radeon GPU with OpenCL			
1	**-1.10**9940E+0	**5.454**317E+1	**6.1410**38E+2
2	**-1.110**111E+0	**5.454**170E+1	**6.1410**44E+2
AMD Trinity APU with OpenCL			
1	**-1.110**023E+0	**5.454**169E+1	**6.1410**62E+2
2	**-1.110**113E+0	**5.454**261E+1	**6.1410**49E+2

How to estimate the impact of round-off errors?

The exact result r of an arithmetic operation is approximated by a floating-point number R^- or R^+.



The random rounding mode

Approximation of r by R^- or R^+ with the probability 1/2

The CESTAC method

The same code is run several times with the random rounding mode. Then different results are obtained.

Briefly, the part that is common to all the different results is assumed to be reliable and the part that is different in the results is affected by round-off errors.

Implementation of the CESTAC method

The implementation of the CESTAC method in a code providing a result R consists in:

- performing N times this code with the random rounding mode to obtain N samples R_i of R,
- choosing as the computed result the mean value $\overline{R}$ of R_i, $i = 1, ..., N$,
- estimating the number of exact significant decimal digits of $\overline{R}$ with

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

where

$$\overline{R} = \frac{1}{N}\sum_{i=1}^{N} R_i \quad \text{and} \quad \sigma^2 = \frac{1}{N-1}\sum_{i=1}^{N}\left(R_i - \overline{R}\right)^2.$$

τ_β is the value of Student's distribution for $N - 1$ degrees of freedom and a probability level β.

In pratice, $N = 3$ and $\beta = 95\%$.

Self-validation of the CESTAC method

The CESTAC method is based on a 1st order model.

- A multiplication of two insignificant results
- or a division by an insignificant result

may invalidate the 1st order approximation.

Therefore the CESTAC method requires a dynamical control of multiplications and divisions, during the execution of the code.

The concept of computed zero

J. Vignes, 1986

Definition

Using the CESTAC method, a result R is a computed zero, denoted by @.0, if

$$\forall i, R_i = 0 \ \text{ or } \ C_{\overline{R}} \leq 0.$$

It means that R is a computed result which, because of round-off errors, cannot be distinguished from 0.

The stochastic definitions

Definition

Let X and Y be two results computed using the CESTAC method (N-sample), X is stochastically equal to Y, noted $X \ s\!= \ Y$, if and only if

$$X - Y = @.0.$$

Definition

Let X and Y be two results computed using the CESTAC method (N-sample).

- X is stochastically strictly greater than Y, noted $X \ s\!> \ Y$, if and only if

$$\overline{X} > \overline{Y} \quad \text{and} \quad X \ s\!\neq \ Y$$

- X is stochastically greater than or equal to Y, noted $X \ s\!\geq \ Y$, if and only if

$$\overline{X} \geq \overline{Y} \quad \text{or} \quad X \ s\!= \ Y$$

Discrete Stochastic Arithmetic (DSA) is defined as the joint use of the CESTAC method, the computed zero and the stochastic relation definitions.

The CADNA library http://www.lip6.fr/cadna

The CADNA library implements Discrete Stochastic Arithmetic.

CADNA allows to estimate round-off error propagation in any scientific program written in Fortran or in C++.

More precisely, CADNA enables one to:

- estimate the numerical quality of any result
- control branching statements
- perform a dynamic numerical debugging
- take into account uncertainty on data.

The CADNA library http://www.lip6.fr/cadna

The CADNA library implements Discrete Stochastic Arithmetic.

CADNA allows to estimate round-off error propagation in any scientific program written in Fortran or in C++.

More precisely, CADNA enables one to:

- estimate the numerical quality of any result
- control branching statements
- perform a dynamic numerical debugging
- take into account uncertainty on data.

CADNA provides new numerical types, the stochastic types, which consist of:

- 3 floating point variables
- an integer variable to store the accuracy.

All operators and mathematical functions are redefined for these types.
$\Rightarrow$ CADNA requires only a few modifications in user programs.

An example proposed by S. Rump

Computation of $f(10864, 18817)$ and $f(\frac{1}{3}, \frac{2}{3})$ with $f(x, y) = 9x^4 - y^4 + 2y^2$

```
      program ex1
      implicit double precision (a-h,o-z)
      x = 10864.d0
      y = 18817.d0
      write(*,*)'P(10864,18817) = ', rump(x,y)
      x = 1.d0/3.d0
      y = 2.d0/3.d0
      write(6,100) rump(x,y)
 100  format('P(1/3,2/3) = ',e24.15)
      end

      function rump(x,y)
      implicit double precision (a-h,o-z)
      a=9.d0*x*x*x*x
      b=y*y*y*y
      c=2.d0*y*y
      rump = a-b+c
      return
      end
```

An example proposed by S. Rump (2)

The results:

```
P(10864,18817) =    2.00000000000000
P(1/3,2/3) =    0.802469135802469E+00
```

```
program ex1

implicit double precision  (a-h,o-z)

x = 10864.d0
y = 18817.d0
write(*,*)'P(10864,18817) = ', rump(x,y)
x = 1.d0/3.d0
y = 2.d0/3.d0
write(*,*)'P(10864,18817) = ', rump(x,y)

end

function rump(x,y)

implicit double precision  (a-h,o-z)
a = 9.d0*x*x*x*x
b = y*y*y*y
c = 2.d0*y*y
rump = a-b+c
return
end
```

```fortran
program ex1
use cadna
implicit double precision  (a-h,o-z)

x = 10864.d0
y = 18817.d0
write(*,*)'P(10864,18817) = ', rump(x,y)
x = 1.d0/3.d0
y = 2.d0/3.d0
write(*,*)'P(10864,18817) = ', rump(x,y)

end

function rump(x,y)
use cadna
implicit double precision  (a-h,o-z)
a = 9.d0*x*x*x*x
b = y*y*y*y
c = 2.d0*y*y
rump = a-b+c
return
end
```

```
program ex1
use cadna
implicit double precision  (a-h,o-z)
call cadna_init(-1)
x = 10864.d0
y = 18817.d0
write(*,*)'P(10864,18817) = ', rump(x,y)
x = 1.d0/3.d0
y = 2.d0/3.d0
write(*,*)'P(10864,18817) = ', rump(x,y)

end

function rump(x,y)
use cadna
implicit double precision  (a-h,o-z)
a = 9.d0*x*x*x*x
b = y*y*y*y
c = 2.d0*y*y
rump = a-b+c
return
end
```

```
program ex1
use cadna
implicit double precision  (a-h,o-z)
call cadna_init(-1)
x = 10864.d0
y = 18817.d0
write(*,*)'P(10864,18817) = ', rump(x,y)
x = 1.d0/3.d0
y = 2.d0/3.d0
write(*,*)'P(10864,18817) = ', rump(x,y)
call cadna_end()
end

function rump(x,y)
use cadna
implicit double precision  (a-h,o-z)
a = 9.d0*x*x*x*x
b = y*y*y*y
c = 2.d0*y*y
rump = a-b+c
return
end
```

```
program ex1
use cadna
implicit double precision  (a-h,o-z)
call cadna_init(-1)
x = 10864.d0
y = 18817.d0
write(*,*)'P(10864,18817) = ', rump(x,y)
x = 1.d0/3.d0
y = 2.d0/3.d0
write(*,*)'P(10864,18817) = ', rump(x,y)
call cadna_end()
end

function rump(x,y)
use cadna
implicit double precision  (a-h,o-z)
a = 9.d0*x*x*x*x
b = y*y*y*y
c = 2.d0*y*y
rump = a-b+c
return
end
```

```
program ex1
use cadna
implicit type(double_st)  (a-h,o-z)
call cadna_init(-1)
x = 10864.d0
y = 18817.d0
write(*,*)'P(10864,18817) = ', rump(x,y)
x = 1.d0/3.d0
y = 2.d0/3.d0
write(*,*)'P(10864,18817) = ', rump(x,y)
call cadna_end()
end

function rump(x,y)
use cadna
implicit type(double_st)  (a-h,o-z)
a = 9.d0*x*x*x*x
b = y*y*y*y
c = 2.d0*y*y
rump = a-b+c
return
end
```

```
program ex1
use cadna
implicit type(double_st)  (a-h,o-z)
call cadna_init(-1)
x = 10864.d0
y = 18817.d0
write(*,*)'P(10864,18817) = ', rump(x,y)
x = 1.d0/3.d0
y = 2.d0/3.d0
write(*,*)'P(10864,18817) = ', rump(x,y)
call cadna_end()
end

function rump(x,y)
use cadna
implicit type(double_st)  (a-h,o-z)
a = 9.d0*x*x*x*x
b = y*y*y*y
c = 2.d0*y*y
rump = a-b+c
return
end
```

```
program ex1
use cadna
implicit type(double_st)  (a-h,o-z)
call cadna_init(-1)
x = 10864.d0
y = 18817.d0
write(*,*)'P(10864,18817) = ',str(rump(x,y))
x = 1.d0/3.d0
y = 2.d0/3.d0
write(*,*)'P(10864,18817) = ',str(rump(x,y))
call cadna_end()
end

function rump(x,y)
use cadna
implicit type(double_st)  (a-h,o-z)
a = 9.d0*x*x*x*x
b = y*y*y*y
c = 2.d0*y*y
rump = a-b+c
return
end
```

The run with CADNA

CADNA software — University P. et M. Curie — LIP6
Self-validation detection: ON
Mathematical instabilities detection: ON
Branching instabilities detection: ON
Intrinsic instabilities detection: ON
Cancellation instabilities detection: ON

P(10864,18817) = @.0
P(1/3,2/3) = 0.802469135802469E+000

CADNA software — University P. et M. Curie — LIP6
There are 2 numerical instabilities
0 UNSTABLE DIVISION(S)
0 UNSTABLE POWER FUNCTION(S)
0 UNSTABLE MULTIPLICATION(S)
0 UNSTABLE BRANCHING(S)
0 UNSTABLE MATHEMATICAL FUNCTION(S)
0 UNSTABLE INTRINSIC FUNCTION(S)
2 LOSS(ES) OF ACCURACY DUE TO CANCELLATION(S)

CADNA on CPU

- **Rounding mode change**: the *rnd_switch* function
  - switches the rounding mode from $+\infty$ to $-\infty$, or from $-\infty$ to $+\infty$.
  - is written in assembly language
  - changes two bits in the FPU Control Word.

CADNA on CPU

- **Rounding mode change**: the *rnd_switch* function
  - switches the rounding mode from $+\infty$ to $-\infty$, or from $-\infty$ to $+\infty$.
  - is written in assembly language
  - changes two bits in the FPU Control Word.
- **Instability detection**:
  - dedicated counters are incremented
  - the occurrence of each kind of instability is given at the end of the run.

CADNA for CPU-GPU simulations

Rounding mode change

An arithmetic operation on GPU can be performed with a specified rounding mode.

CPU

```
if (RANDOM) rnd_switch();
res.x=a.x*b.x;


if (RANDOM) rnd_switch();
res.y=a.y*b.y;
rnd_switch();
res.z=a.z*b.z;
```

GPU

```
if (RANDOMGPU())
  res.x=__fmul_ru(a.x,b.x);
else
  res.x=__fmul_rd(a.x,b.x);

if (RANDOMGPU()) {
  res.y=__fmul_rd(a.y,b.y);
  res.z=__fmul_ru(a.z,b.z);
}
else {
  res.y=__fmul_ru(a.y,b.y);
  res.z=__fmul_rd(a.z,b.z);
}
```

2 types: `float_st` for CPU computation and `float_gpu_st` for GPU

CADNA for CPU-GPU simulations

Instability detection

- No counter: would need more memory (shared) and would need a lot of atomic operations
- An unsigned char is associated with each result (each bit is associated with a type of instability).

CPU +GPU

```
class float_st {
protected:
float x,y,z;
private:
mutable unsigned int accuracy;
unsigned char accuracy;
mutable unsigned char error;
unsigned char pad1, pad2;
}
```

GPU

```
class float_gpu_st {
public:
float x,y,z;
public:
mutable unsigned char accuracy;
mutable unsigned char error;
unsigned char pad1, pad2; }
```

CADNA for CPU-GPU simulations

Instability detection

- No counter: would need more memory (shared) and would need a lot of atomic operations
- An unsigned char is associated with each result (each bit is associated with a type of instability).

CPU +GPU

```
class float_st {
protected:
float x,y,z;
private:
mutable unsigned int accuracy;
unsigned char accuracy;
mutable unsigned char error;
unsigned char pad1, pad2;
}
```

GPU

```
class float_gpu_st {
public:
float x,y,z;
public:
mutable unsigned char accuracy;
mutable unsigned char error;
unsigned char pad1, pad2; }
```

CADNA for CPU-GPU simulations

Instability detection

- No counter: would need more memory (shared) and would need a lot of atomic operations
- An unsigned char is associated with each result (each bit is associated with a type of instability).

CPU +GPU

```
class float_st {
protected:
float x,y,z;
private:
mutable unsigned int accuracy;
unsigned char accuracy;
mutable unsigned char error;
unsigned char pad1, pad2;
}
```

GPU

```
class float_gpu_st {
public:
float x,y,z;
public:
mutable unsigned char accuracy;
mutable unsigned char error;
unsigned char pad1, pad2; }
```

Example: matrix multiplication

```
#include "cadna.h"
#include "cadna_gpu.cu"

__global__ void matMulKernel(
          float_gpu_st* mat1,
          float_gpu_st* mat2,
          float_gpu_st* matRes,
          int dim) {

  unsigned int x = blockDim.x*blockIdx.x+threadIdx.x;
  unsigned int y = blockDim.y*blockIdx.y+threadIdx.y;

  cadna_init_gpu();

  if (x < dim && y < dim){
    float_gpu_st temp;
    temp=0;
    for(int i=0; i<dim;i++){
      temp = temp + mat1[y * dim + i] * mat2[i * dim + x];
    }
   matRes[y * dim + x] = temp;
  }
}
```

Example: matrix multiplication

```
...
float_st mat1[DIMMAT][DIMMAT], mat2[DIMMAT][DIMMAT],
         res[DIMMAT][DIMMAT];
...
  cadna_init(-1);
  int size = DIMMAT * DIMMAT * sizeof(float_st);
  cudaMalloc((void **) &d_mat1, size);
  cudaMalloc((void **) &d_mat2, size);
  cudaMalloc((void **) &d_res,  size);
  cudaMemcpy(d_mat1, mat1, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_mat2, mat2, size, cudaMemcpyHostToDevice);

  dim3 threadsPerBlock(16,16);
  int nbbx = (int)ceil((float)DIMMAT/(float)16);
  int nbby = (int)ceil((float)DIMMAT/(float)16);
  dim3 numBlocks(nbbx , nbby);
  matMulKernel<<< numBlocks , threadsPerBlock>>>
  (d_mat1, d_mat2, d_res, DIMMAT);
  cudaMemcpy(res, d_res,  size, cudaMemcpyDeviceToHost);
  ...
  cadna_end();
```

Output

```
mat1=
 0.0000000E+000  0.1000000E+001  0.2000000E+001  0.3000000E+001
 0.4000000E+001  0.5000000E+001  0.6000000E+001  0.6999999E+001
 0.8000000E+001  @.0             0.1000000E+002  0.1099999E+002
 0.1199999E+002  0.1299999E+002  0.1400000E+002  0.1500000E+002

mat2=
 0.1000000E+001  0.1000000E+001  0.1000000E+001  0.1000000E+001
 0.1000000E+001  @.0             0.1000000E+001  0.1000000E+001
 0.1000000E+001  0.1000000E+001  0.1000000E+001  0.1000000E+001
 0.1000000E+001  0.1000000E+001  0.1000000E+001  0.1000000E+001

res=
 0.5999999E+001  @.0                     0.5999999E+001  0.5999999E+001
 0.2199999E+002  @.0                     0.2199999E+002  0.2199999E+002
 @.0             @.0       MUL           @.0             @.0
 0.5399999E+002  @.0                     0.5399999E+002  0.5399999E+002
----------------------------------------------------------------
CADNA GPU software --- University P. et M. Curie --- LIP6
No instability detected on CPU
----------------------------------------------------------------
```

The acoustic wave propagation code examined with CADNA

The code is run on:

- an AMD Opteron 6168 CPU with gcc
- an NVIDIA C2050 GPU with CUDA.

With both implementations of the finite difference scheme, the number of exact digits varies from 0 to 7 (single precision).

Its mean value is:

- 4.06 with both schemes on CPU
- 3.43 with scheme 1 and 3.49 with scheme 2 on GPU.

$\Rightarrow$ consistent with our previous observations

Instabilities detected: > 270 000 cancellations

The acoustic wave propagation code examined with CADNA

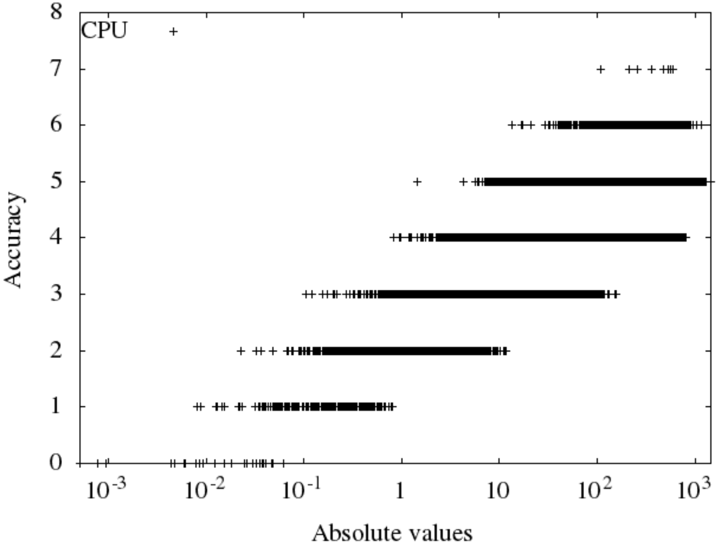
Results computed at 3 different points using scheme 1:

	Point in the space domain		
	$p_1 = (0, 19, 62)$	$p_2 = (50, 12, 2)$	$p_3 = (20, 1, 46)$
IEEE CPU	-1.110479E+0	5.454238E+1	6.141038E+2
IEEE GPU	-1.110204E+0	5.454224E+1	6.141046E+2
CADNA CPU	-1.1E+0	5.454E+1	6.14104E+2
CADNA GPU	-1.11E+0	5.45E+1	6.1410E+2
Reference	-1.108603879E+0	5.454034021E+1	6.141041156E+2

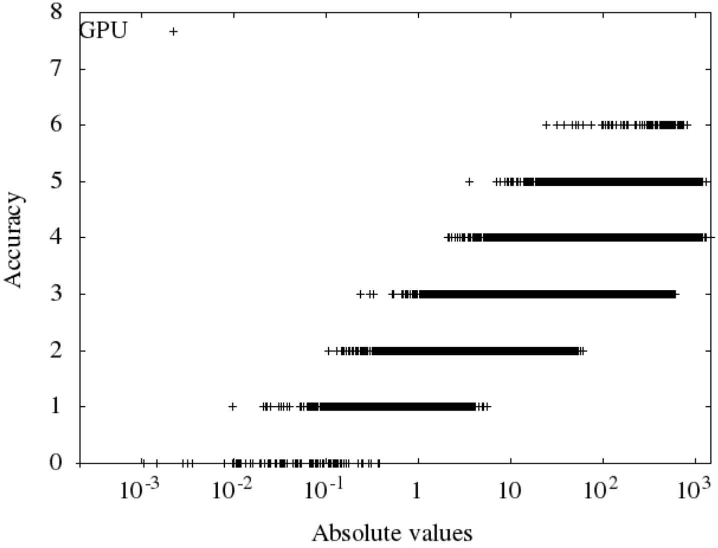
Despite differences in the estimated accuracy, the same trend can be observed on CPU and on GPU.

- Highest round-off errors impact negligible results.
- Highest results impacted by low round-off errors.

Accuracy distribution on CPU



Accuracy distribution on GPU



Execution times

CPU			
execution	instability detection	execution time (s)	ratio
IEEE	-	110.8	1
CADNA	all instabilities	4349	39.3
	no instability	1655	14.9
	mul., div., branching	1663	15.0

GPU			
execution	instability detection	execution time (s)	ratio
IEEE	-	0.80	1
CADNA	mul., div., branching	5.73	7.2

Conclusion

Stochastic arithmetic can estimate which digits are affected by round-off errors and possibly explain reproducibility failures.

Related works :

- taking advantage of SIMD instructions (SSE, AVX, Xeon Phi)
- CADNA for MPI codes
- CADNA for OpenMP codes.

On the probability of the confidence interval

With $\beta = 95\%$ and $N = 3$,

- the probability of overestimating the number of exact significant digits of at least 1 is 0.054%
- the probability of underestimating the number of exact significant digits of at least 1 is 29%.

By choosing a confidence interval at 95%, we prefer to guarantee a minimal number of exact significant digits with high probability (99.946%), even if we are often pessimistic by 1 digit.