

Numerical precision: just what's needed

Fabienne Jézéquel
LIP6, Sorbonne Université, France

EuroHPC Summit
Green HPC session
25 Mar. 2021



Floating-point arithmetic:

Sign	Exponent	Mantissa
------	----------	----------

Various floating-point formats:

	#bits Mantissa (p)	Exp.	Range	$u = 2^{-p}$
bfloat16 (half)	8	8	$10^{\pm 38}$	$\approx 4 \times 10^{-3}$
fp16 (half)	11	5	$10^{\pm 5}$	$\approx 5 \times 10^{-4}$
fp32 (single)	24	8	$10^{\pm 38}$	$\approx 6 \times 10^{-8}$
fp64 (double)	53	11	$10^{\pm 308}$	$\approx 1 \times 10^{-16}$
fp128 (quad)	113	15	$10^{\pm 4932}$	$\approx 1 \times 10^{-34}$

↘ precision:

- ↘ execution time ☺
- ↘ volume of results exchanged ☺
- ↗ energy efficiency ☺

energy consumption proportional to p^2

energy ratio	
fp64/fp32	≈ 5
fp32/fp16	≈ 5
fp32/bfloat16	≈ 9

- But **computed results may be invalid** because of rounding errors ☹

In this talk we aim at answering the following questions.

- 1 How to control the validity of floating-point results?
- 2 How to determine automatically the suitable format for each variable?

Rounding error analysis

Several approaches

- Interval arithmetic
 - guaranteed bounds for each computed result
 - the error may be overestimated
 - specific algorithms
 - ex: **INTLAB** [Rump'99]
- Static analysis
 - no execution, rigorous analysis, all possible input values taken into account
 - not suited to large programs
 - ex: **FLUCTUAT** [Goubault & al.'06], **FLDLib** [Jacquemin & al.'19]
- Probabilistic approach
 - estimates the number of correct digits of any computed result
 - can be used in HPC programs
 - requires no algorithm modification
 - ex: **CADNA** [Chesneaux'90], **VERIFICARLO** [Denis & al.'16], **VERROU** [Févotte & al.'17]

Classic arithmetic

$$A \oplus B \rightarrow R$$

$R = 3.14237654356891$

Stochastic arithmetic

Random
rounding

$$A_1 \oplus B_1 \rightarrow R_1$$

$$A_2 \oplus B_2 \rightarrow R_2$$

$$A_3 \oplus B_3 \rightarrow R_3$$

$R_1 = \mathbf{3.141354786390989}$

$R_2 = \mathbf{3.143689456834534}$

$R_3 = \mathbf{3.142579087356598}$

- each operation executed 3 times with a random rounding mode

Classic arithmetic

$$A \oplus B \rightarrow R$$

$R = 3.14237654356891$

Stochastic arithmetic

Random
rounding

$$A_1 \oplus B_1 \rightarrow R_1$$

$$A_2 \oplus B_2 \rightarrow R_2$$

$$A_3 \oplus B_3 \rightarrow R_3$$

$R_1 = \mathbf{3.141354786390989}$

$R_2 = \mathbf{3.143689456834534}$

$R_3 = \mathbf{3.142579087356598}$

- each operation executed 3 times with a random rounding mode
- number of correct digits in the results estimated using Student's test with the confidence level 95%

Classic arithmetic

$$A \oplus B \rightarrow R$$

$R = 3.14237654356891$

Stochastic arithmetic

Random
rounding

$$A_1 \oplus B_1 \rightarrow R_1$$

$$A_2 \oplus B_2 \rightarrow R_2$$

$$A_3 \oplus B_3 \rightarrow R_3$$

$R_1 = \mathbf{3.141354786390989}$

$R_2 = \mathbf{3.143689456834534}$

$R_3 = \mathbf{3.142579087356598}$

- each operation executed 3 times with a random rounding mode
- number of correct digits in the results estimated using Student's test with the confidence level 95%
- operations executed synchronously
 - ⇒ detection of numerical instabilities
Ex: if $(A > B)$ with $A - B$ numerical noise
 - ⇒ optimization of stopping criteria



- implements stochastic arithmetic for C/C++ or Fortran codes
- provides **stochastic types** (3 floating-point variables and an integer)
half_st float_st double_st quad_st
- all operators and mathematical functions overloaded
⇒ **few modifications in user programs**
- support for MPI, OpenMP, GPU, **vectorised** codes
- In **one CADNA execution**: accuracy of any result, complete list of numerical instabilities

SAM (Stochastic Arithmetic in Multiprecision) [Graillat & al.'11]

- implements stochastic arithmetic in arbitrary precision (based on MPFR¹)
 `mp_st` stochastic type

¹www.mpfr.org

SAM (Stochastic Arithmetic in Multiprecision) [Graillat & al.'11]

- implements stochastic arithmetic in arbitrary precision (based on MPFR¹)
mp_st stochastic type
- recent improvement: control of operations **mixing different precisions**

Ex: mp_st<23> A; mp_st<47>B; mp_st<35> C;

$$C = A \oplus B$$

35 bits 23 bits 47 bits

⇒ accuracy estimation on FPGA

¹www.mpfr.org

An example without/with CADNA

Computation of $P(x, y) = 9x^4 - y^4 + 2y^2$ [Rump'83]

```
#include <iostream>
using namespace std;
double rump(double x, double y) {
    return 9.0*x*x*x*x - y*y*y*y + 2.0*y*y;
}
int main() {
    cout.precision(15);
    cout.setf(ios::scientific, ios::floatfield);
    double x, y;
    x = 10864.0;
    y = 18817.0;
    cout<<"P1="<<rump(x, y)<< endl;
    x = 1.0/3.0;
    y = 2.0/3.0;
    cout<<"P2="<<rump(x, y)<< endl;
    return 0;
}
```

An example without/with CADNA

Computation of $P(x, y) = 9x^4 - y^4 + 2y^2$ [Rump'83]

```
#include <iostream>
using namespace std;
double rump(double x, double y) {
    return 9.0*x*x*x*x - y*y*y*y + 2.0*y*y;
}
int main() {
    cout.precision(15);
    cout.setf(ios::scientific, ios::floatfield);
    double x, y;
    x = 10864.0;
    y = 18817.0;
    cout<<"P1="<<rump(x, y)<< endl;
    x = 1.0/3.0;
    y = 2.0/3.0;
    cout<<"P2="<<rump(x, y)<< endl;
    return 0;
}
```

P1=2.000000000000000e+00

P2=8.02469135802469e-01

```
#include <iostream>

using namespace std;

double rump(double x, double y) {
    return 9.0*x*x*x*x-x*y*y*y+2.0*y*y;
}

int main() {
    cout.precision(15);
    cout.setf(ios::scientific,ios::floatfield);

    double x, y;
    x=10864.0; y=18817.0;
    cout<<"P1="<<rump(x, y)<<endl;
    x=1.0/3.0; y=2.0/3.0;
    cout<<"P2="<<rump(x, y)<<endl;

    return 0;
}
```

```
#include <iostream>
#include <cadna.h>
using namespace std;
double rump(double x, double y) {
    return 9.0*x*x*x*x-y*y*y*y+2.0*y*y;
}
int main() {
    cout.precision(15);
    cout.setf(ios::scientific,ios::floatfield);
    cadna_init(-1);
    double x, y;
    x=10864.0; y=18817.0;
    cout<<"P1="<<rump(x, y)<<endl;
    x=1.0/3.0; y=2.0/3.0;
    cout<<"P2="<<rump(x, y)<<endl;
    cadna_end();
    return 0;
}
```

```

#include <iostream>
#include <cadna.h>
using namespace std;
double  rump(double  x, double  y) {
    return 9.0*x*x*x*x-x*y*y*y+2.0*y*y;
}
int main() {
    cout.precision(15);
    cout.setf(ios::scientific,ios::floatfield);
    cadna_init(-1);
    double  x, y;
    x=10864.0; y=18817.0;
    cout<<"P1="<<rump(x, y)<<endl;
    x=1.0/3.0; y=2.0/3.0;
    cout<<"P2="<<rump(x, y)<<endl;
    cadna_end();
    return 0;
}

```

```

#include <iostream>
#include <cadna.h>
using namespace std;
double_st rump(double_st x, double_st y) {
    return 9.0*x*x*x*x-x*y*y*y+2.0*y*y;
}
int main() {
    cout.precision(15);
    cout.setf(ios::scientific,ios::floatfield);
    cadna_init(-1);
    double_st x, y;
    x=10864.0; y=18817.0;
    cout<<"P1="<<rump(x, y)<<endl;
    x=1.0/3.0; y=2.0/3.0;
    cout<<"P2="<<rump(x, y)<<endl;
    cadna_end();
    return 0;
}

```


Results with CADNA

only correct digits are displayed

CADNA_C software

Self-validation detection: ON

Mathematical instabilities detection: ON

Branching instabilities detection: ON

Intrinsic instabilities detection: ON

Cancellation instabilities detection: ON

P1= @.0 (no correct digits)

P2= 0.802469135802469E+000

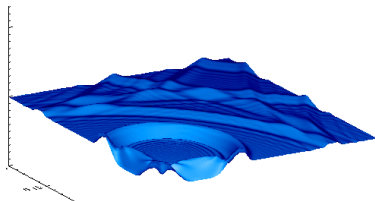
There are 2 numerical instabilities

2 LOSS(ES) OF ACCURACY DUE TO CANCELLATION(S)

Numerical validation of a shallow-water (SW) simulation on GPU

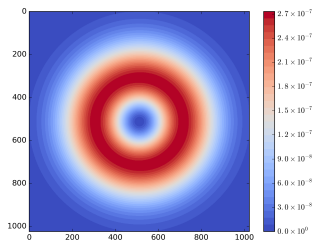
- Simulation of the evolution of water height and velocities in a 2D oceanic basin
- CUDA GPU code in double precision

- Focusing on an eddy evolution: 20 time steps (12 hours of simulated time) on a 1024×1024 grid

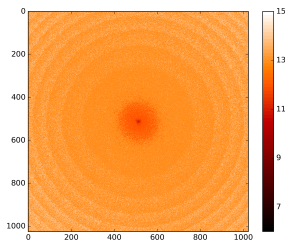


SW eddy simulation with CADNA-GPU

At the end of the simulation:



Square of water velocity in $m^2.s^{-2}$



Number of correct digits estimated by CADNA-GPU

- at eddy center: great accuracy loss due to cancellations
- point at the very center: 9 digits lost
⇒ **no correct digits in single precision**
- fortunately, velocity values close to zero at eddy center
→ negligible impact on the output
→ **satisfactory overall accuracy**

Can we use reduced or mixed precision to improve performance and energy efficiency?


- mixed precision linear algebra algorithms
ex: solution of linear systems using iterative refinement [Carson & al.'18]
- precision autotuning

- floating-point autotuning tools that intend to deal with large codes:
 - **Precimonious** [Rubio-González & al.'13]
 - source modification with LLVM
 - **CRAFT** [Lam & al.'13]
 - binary modifications on the operations
 - **ADAPT** [Menon & al.'18]
 - based on algorithmic differentiation
 - CRAFT & ADAPT now combined in **FloatSmith** [Lam & al.'19]

They rely on comparisons with the highest precision result.


- floating-point autotuning tools that intend to deal with large codes:
 - **Precimonious** [Rubio-González & al.'13]
 - source modification with LLVM
 - **CRAFT** [Lam & al.'13]
 - binary modifications on the operations
 - **ADAPT** [Menon & al.'18]
 - based on algorithmic differentiation
 - CRAFT & ADAPT now combined in **FloatSmith** [Lam & al.'19]

They rely on comparisons with the highest precision result.

 [Rump'88] $P = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + x/(2y)$
with $x = 77617$ and $y = 33096$


- floating-point autotuning tools that intend to deal with large codes:
 - **Precimonious** [Rubio-González & al.'13]
 - source modification with LLVM
 - **CRAFT** [Lam & al.'13]
 - binary modifications on the operations
 - **ADAPT** [Menon & al.'18]
 - based on algorithmic differentiation
 - CRAFT & ADAPT now combined in **FloatSmith** [Lam & al.'19]

They rely on comparisons with the highest precision result.

 [Rump'88] $P = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + x/(2y)$
with $x = 77617$ and $y = 33096$
float: $P = 2.571784e+29$

- floating-point autotuning tools that intend to deal with large codes:
 - **Precimonious** [Rubio-González & al.'13]
 - source modification with LLVM
 - **CRAFT** [Lam & al.'13]
 - binary modifications on the operations
 - **ADAPT** [Menon & al.'18]
 - based on algorithmic differentiation
 - CRAFT & ADAPT now combined in **FloatSmith** [Lam & al.'19]

They rely on comparisons with the highest precision result.

 [Rump'88] $P = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + x/(2y)$


with $x = 77617$ and $y = 33096$

float: $P = 2.571784e+29$

double: $P = 1.17260394005318$

- floating-point autotuning tools that intend to deal with large codes:
 - **Precimonious** [Rubio-González & al.'13]
 - source modification with LLVM
 - **CRAFT** [Lam & al.'13]
 - binary modifications on the operations
 - **ADAPT** [Menon & al.'18]
 - based on algorithmic differentiation
 - CRAFT & ADAPT now combined in **FloatSmith** [Lam & al.'19]

They rely on comparisons with the highest precision result.

 [Rump'88] $P = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + x/(2y)$

with $x = 77617$ and $y = 33096$


float: $P = 2.571784e+29$

double: $P = 1.17260394005318$

quad: $P = 1.17260394005317863185883490452018$

- floating-point autotuning tools that intend to deal with large codes:
 - **Precimonious** [Rubio-González & al.'13]
 - source modification with LLVM
 - **CRAFT** [Lam & al.'13]
 - binary modifications on the operations
 - **ADAPT** [Menon & al.'18]
 - based on algorithmic differentiation
 - CRAFT & ADAPT now combined in **FloatSmith** [Lam & al.'19]

They rely on comparisons with the highest precision result.

 [Rump'88] $P = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + x/(2y)$

with $x = 77617$ and $y = 33096$

float: $P = 2.571784e+29$

double: $P = 1.17260394005318$

quad: $P = 1.17260394005317863185883490452018$

exact: $P \approx -0.827396059946821368141165095479816292$

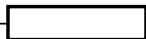
PROMISE

- provides a mixed precision code (half, single, double, quad) taking into account a required accuracy
- uses GADNA to validate a type configuration
- uses the Delta Debug algorithm [Zeller'09] to search for a valid type configuration with a mean complexity of $O(n \log(n))$ for n variables.

Searching for a valid configuration with 2 types

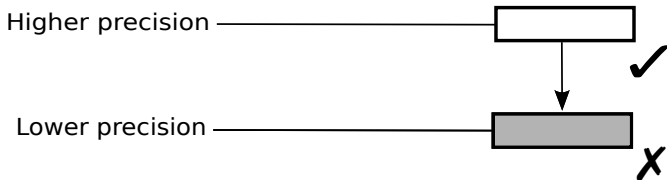
Method based on the Delta Debug algorithm [Zeller'09]

Higher precision



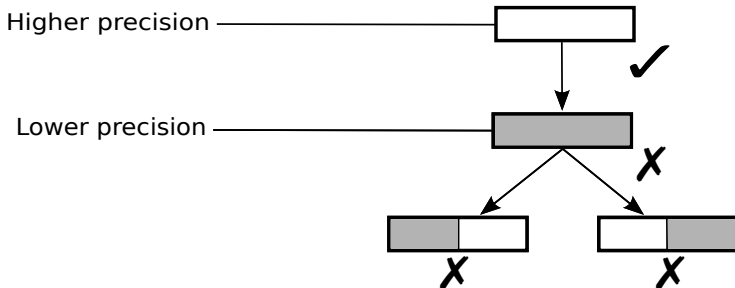
Searching for a valid configuration with 2 types

Method based on the Delta Debug algorithm [Zeller'09]



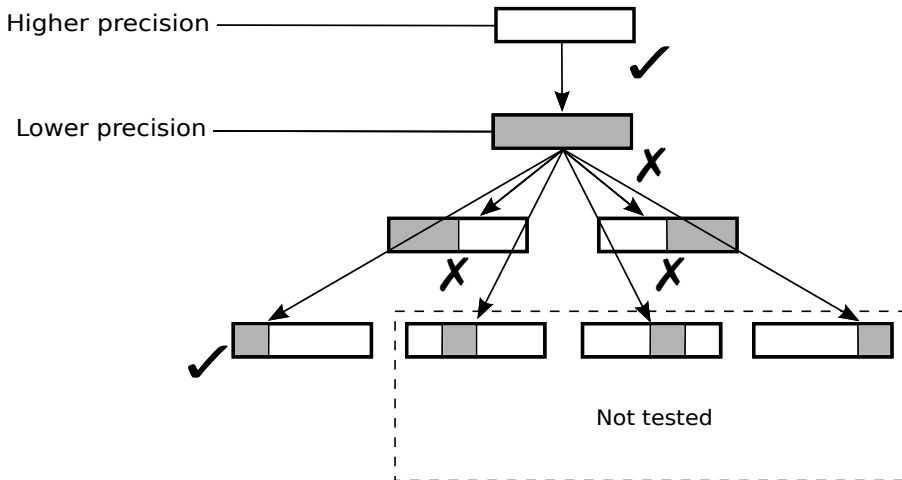
Searching for a valid configuration with 2 types

Method based on the Delta Debug algorithm [Zeller'09]



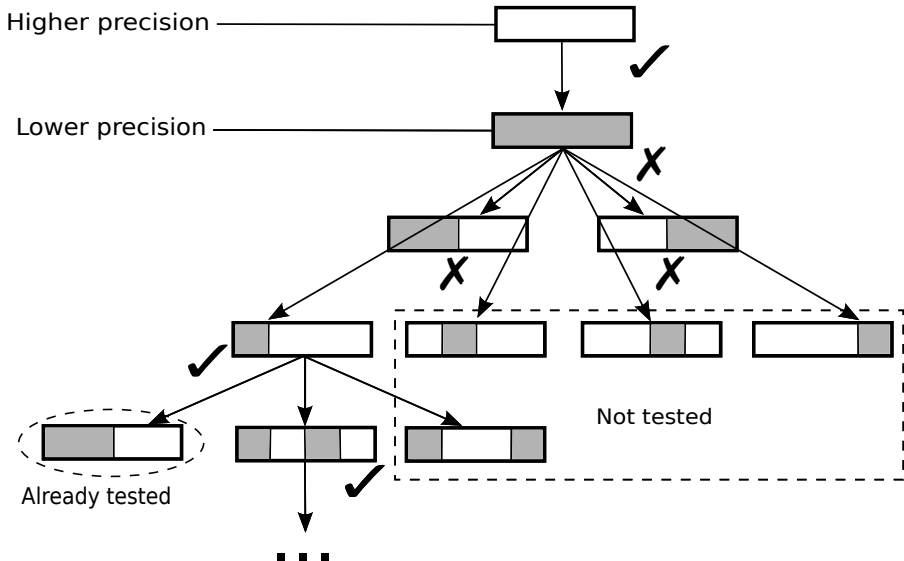
Searching for a valid configuration with 2 types

Method based on the Delta Debug algorithm [Zeller'09]



Searching for a valid configuration with 2 types

Method based on the Delta Debug algorithm [Zeller'09]



Searching for a valid type configuration

PROMISE with 2 types (ex: double & single precision)

From a code in double, the Delta Debug (DD) algorithm finds which variables can be relaxed to single precision.



Searching for a valid type configuration

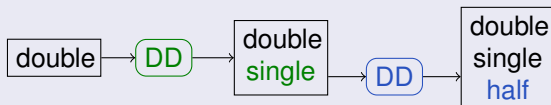
PROMISE with 2 types (ex: double & single precision)

From a code in double, the Delta Debug (DD) algorithm finds which variables can be relaxed to single precision.



PROMISE with 3 types (ex: double, single & half precision)

The Delta Debug algorithm is applied twice.



Precision autotuning using PROMISE

MICADO: simulation of nuclear cores (code developed by EDF)

- neutron transport iterative solver
- 11,000 C++ code lines

# Digits	# double - # float	Speed up	memory gain
10	19-32	1.01	1.00
8	18-33	1.01	1.01
6	13-38	1.20	1.44
5	0-51	1.32	1.62
4			

- Speedup, memory gain w.r.t. the double precision version
- Speed-up up to 1.32 and memory gain 1.62
- Mixed precision approach successful: speed-up 1.20 and memory gain 1.44

To optimize precision and so improve energy efficiency

- numerical validation tools such as CADNA
- precision autotuning tools such as PROMISE
- mixed precision algorithms

Perspectives

- floating-point autotuning in arbitrary precision
- combine mixed precision algorithms and floating-point autotuning

Thanks to the CADNA/SAM/PROMISE contributors:

Julien Brajard, Romuald Carpentier, Jean-Marie Chesneaux, Patrick Corde, Pacôme Eberhart, François Févotte, Pierre Fortin, Stef Graillat, Thibault Hilaire, Sara Hoseininasab, Jean-Luc Lamotte, Baptiste Landreau, Bruno Lathuilière, Romain Picot, Jonathon Tidswell, Su Zhou, ...

Thanks to the CADNA/SAM/PROMISE contributors:

Julien Brajard, Romuald Carpentier, Jean-Marie Chesneaux, Patrick Corde, Pacôme Eberhart, François Févotte, Pierre Fortin, Stef Graillat, Thibault Hilaire, Sara Hoseininasab, Jean-Luc Lamotte, Baptiste Landreau, Bruno Lathuilière, Romain Picot, Jonathon Tidswell, Su Zhou, ...

Thank you for your attention!