

Precision auto-tuning and control of accuracy in high performance simulations

Fabienne Jézéquel

LIP6, Sorbonne Université, France

REPRISES meeting
7-8 April 2022



Introduction

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 ☺

Outline

In this talk we aim at answering the following questions.

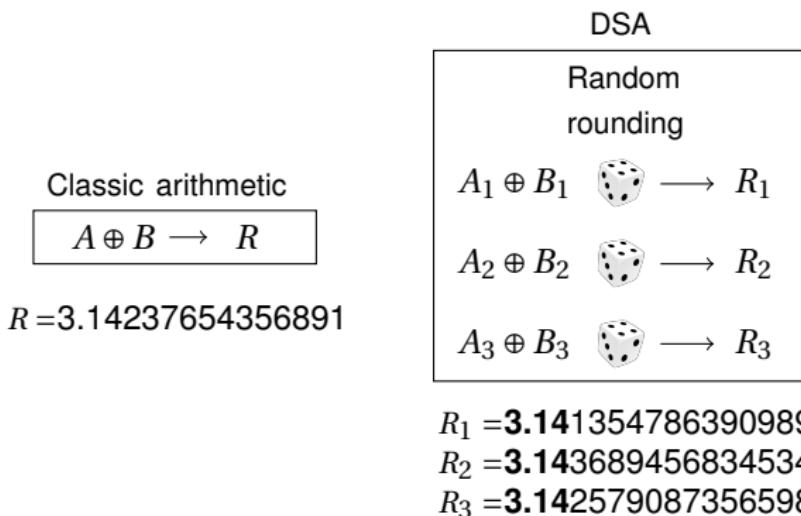
- ➊ How to control the validity of (mixed precision) floating-point results?
- ➋ How to determine automatically the suitable format for each variable?

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
 - requires no algorithm modification
 - can be used in HPC programs
 - ex: **CADNA** [Chesneaux'90], **SAM** [Graillat & al.'11],
VERIFICARLO [Denis & al.'16], **VERROU** [Févotte & al.'17]

Discrete Stochastic Arithmetic (DSA) [Vignes'04]

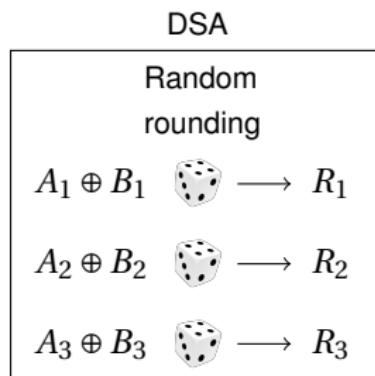


- each operation executed 3 times with a random rounding mode

Classic arithmetic

$$A \oplus B \rightarrow R$$

$$R = 3.14237654356891$$

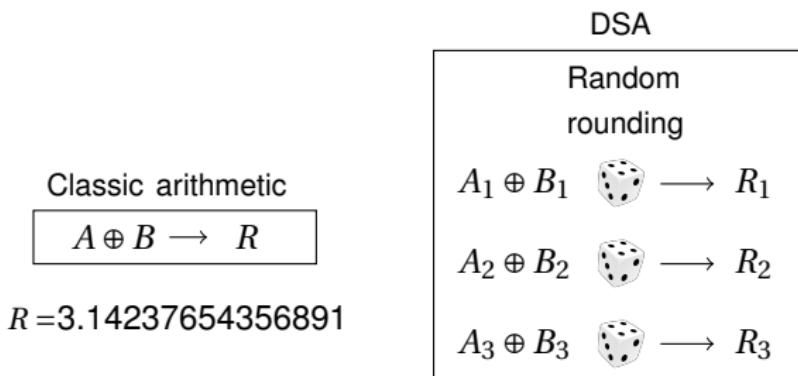


$$R_1 = \mathbf{3.141354786390989}$$

$$R_2 = \mathbf{3.143689456834534}$$

$$R_3 = \mathbf{3.142579087356598}$$

- 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%



$$R_1 = \mathbf{3.141354786390989}$$

$$R_2 = \mathbf{3.143689456834534}$$

$$R_3 = \mathbf{3.142579087356598}$$

- 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

The CADNA library

cadna.lip6.fr



- implements stochastic arithmetic for **C/C++ or Fortran codes**
- provides **stochastic types** (3 floating-point variables and an integer)
- 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

[Chesneaux'90], [Jézéquel & al'08], [Lamotte & al'10], [Eberhart & al'18],...

Stochastic types

exist in half, single, double, quadruple precision

Half precision in CADNA

control of fp16 computation with

- emulated half precision thanks to the library developed by C. Rau (<http://half.sourceforge.net>)
- native half precision on e.g., NVIDIA GPUs or ARM v8.2 processor (successful tests on Fugaku supercomputer)

CADNA cost

- memory: 4
- run time ≈ 10

Efficient rounding mode change

- implicit change of the rounding mode thanks to

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

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

$\circ_{+\infty}$ (resp. $\circ_{-\infty}$): floating-point operation rounded $\rightarrow +\infty$ (resp. $-\infty$)

The SAM library

www-pequan.lip6.fr/~jezequel/SAM

SAM (Stochastic Arithmetic in Multiprecision)

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

operator overloading ⇒ few modifications in user C/C++ programs

¹www.mpfr.org

The SAM library

www-pequan.lip6.fr/~jezequel/SAM

SAM (Stochastic Arithmetic in Multiprecision)

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

operator overloading \Rightarrow few modifications in user C/C++ programs

- uniform precision version [Graillat & al.'11]
- mixed precision version: control of operations mixing different mantissa lengths

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

$$C = A \oplus B$$

The diagram shows the expression $C = A \oplus B$. Below it, three lines point from the terms to their respective bit widths: '35 bits' under A , '23 bits' under B , and '47 bits' under C .

\Rightarrow 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-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;
}
```

```
#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);

    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;

    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-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_st rump(double_st x, double_st 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_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)

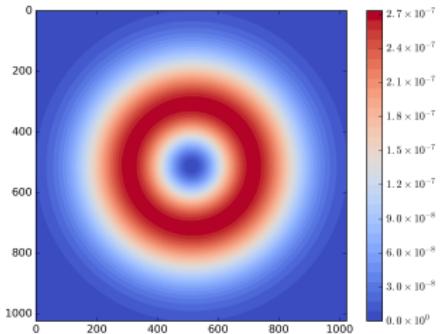
Numerical validation of a shallow-water (SW) simulation on GPU

- 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

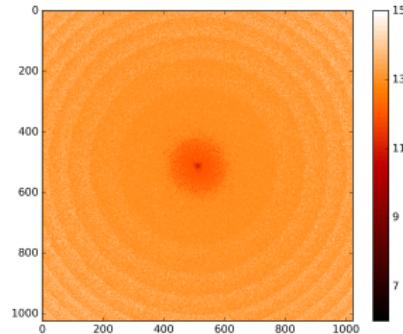


SW eddy simulation with CADNA-GPU

At the end of the simulation:



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



Number of correct digits estimated by CADNA

- 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**

Tools related to CADNA

available on cadna.lip6.fr

- CADNAIZER
 - automatically transforms C codes to be used with CADNA

- CADTRACE
 - identifies the instructions responsible for numerical instabilities

Example:

There are 12 numerical instabilities.

10 LOSS(ES) OF ACCURACY DUE TO CANCELLATION(S).

 5 in <ex> file "ex.f90" line 58

 5 in <ex> file "ex.f90" line 59

1 INSTABILITY IN ABS FUNCTION.

 1 in <ex> file "ex.f90" line 37

1 UNSTABLE BRANCHING.

 1 in <ex> file "ex.f90" line 37

Other numerical validation tools based on result perturbation

- **VERIFICARLO** [Denis & al.'16] based on LLVM
 - **VERROU** [Févotte & al.'17] based on Valgrind, no source code modification ☺
-
- **asynchronous approach:** 1 complete run → 1 result
 - several executions:
 - for rounding error analysis
 - to point out unstable tests
 - no support for GPU codes.

Cost comparison

C++ arithmetic benchmarks (compute/memory bound) [Picot'18]

	3 samples w.r.t classic exec.
CADNA	≈ 5 to 8
VERIFICARLO	≈ 300 to 600
VERROU	≈ 30

If the results accuracy is not satisfactory...

- higher precision: single → double → quad → arbitrary precision
⚠ numerical validation
- compensated algorithms
[Kahan '87], [Priest '92], [Ogita & al. '05], [Graillat & al. '09]
 - for sum, dot product, polynomial evaluation,...
 - results \approx as accurate as with twice the working precision
- accurate and reproducible BLAS
 - ReproBLAS [Demmel & al. '13]
 - ExBLAS [Collange & al. '15]
 - RARE-BLAS [Chohra & al. '16]
 - OzBLAS [Mukunoki & al. '19]

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

- mixed precision linear algebra algorithms
 - matrix-matrix and matrix-vector multiplication,
 - LU and QR matrix factorizations,
 - iterative refinement,
 - Krylov solvers,
 - least squares problems
- precision autotuning

Static tools

- FPTaylor/FPTuner [Solovyev & al.'15] symbolic Taylor expansions
- DAISY [Darulova & al.'18] mixed-precision with rewriting
- TAFFO [Cherubin & al.'19] auto-tuning for floating to fixed-point optimization
- POP [Ben Khalifa & al.'19] error analysis by constraint generation

not suited to large scale programs ☺

Precision autotuning

Dynamic tools

intend to deal with large codes

- **CRAFT** [Lam & al.'13] binary modifications on the operations
- **Precimonious** [Rubio-Gonzàlez & al.'13] source modification with LLVM
- **Blame Analysis** [Nguyen & al.'15] improves Precimonious
- **HiFPTuner** [Guo & al.'18] based on a hierarchical search algorithm
- **ADAPT** [Menon & al.'18] based on algorithmic differentiation
- **FloatSmith** [Lam & al.'19] combination of CRAFT & ADAPT
- Tools dedicated to GPUs (that pay attention to casts):
 - **AMPT-GA** [Kotipalli & al.'19]
 - **GPUMixer** [Laguna & al.'19]
 - **GRAM** [Ho & al.'21]

Precision autotuning

Dynamic tools 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: $P = 2.571784e+29$

Precision autotuning

Dynamic tools 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: $P = 2.571784\text{e+}29$

double: $P = 1.17260394005318$

Precision autotuning

Dynamic tools 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: $P = 2.571784\text{e+}29$

double: $P = 1.17260394005318$

quad: $P = 1.17260394005317863185883490452018$

Precision autotuning

Dynamic tools 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: $P = 2.571784\text{e+}29$

double: $P = 1.17260394005318$

quad: $P = 1.17260394005317863185883490452018$

exact: $P \approx -0.827396059946821368141165095479816292$



- provides a mixed precision code (half, single, double, quad) taking into account a required accuracy
- uses CADNA 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

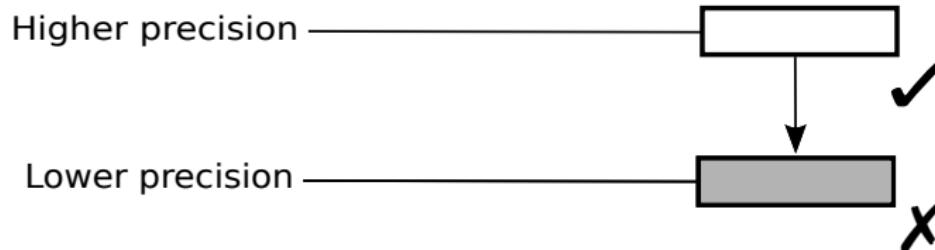
Method based on the Delta Debug algorithm [Zeller '09]

Higher precision



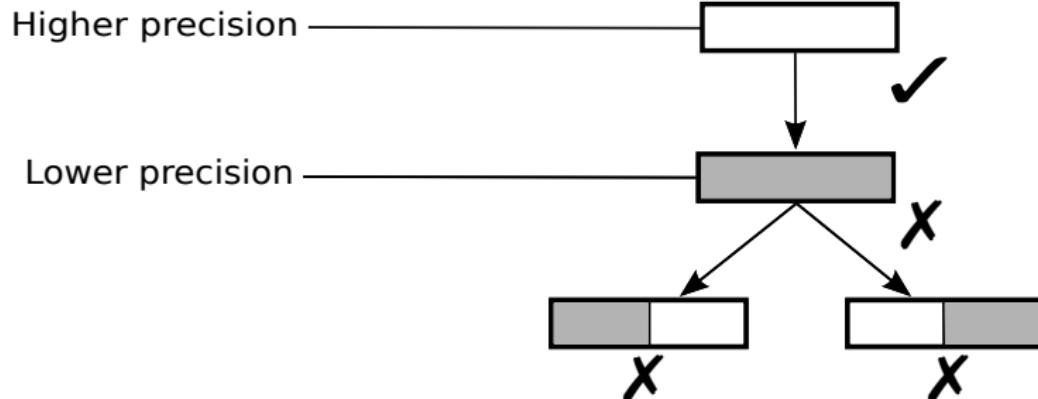
Searching for a valid configuration with 2 types

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



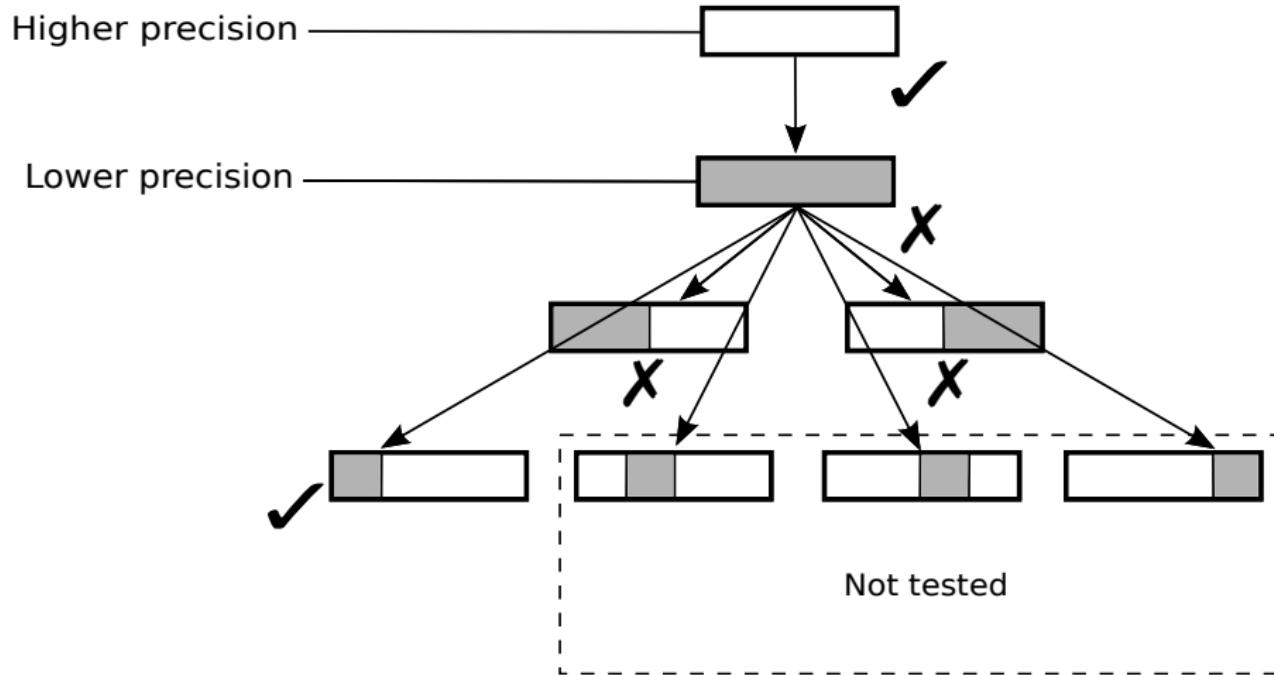
Searching for a valid configuration with 2 types

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]

Higher precision



Lower precision



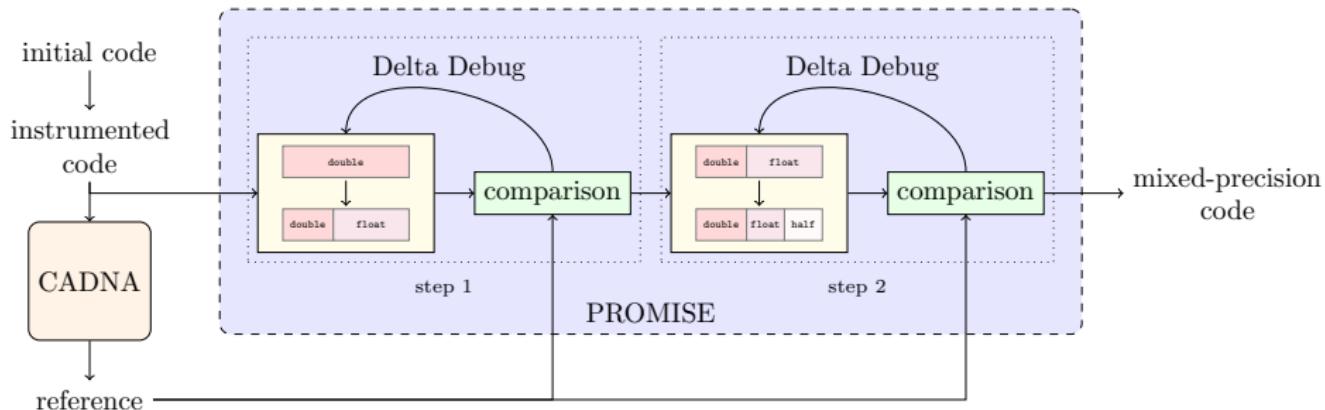
Not tested

Already tested



...

PROMISE in double, single and half precision



- step 1: code in double → variables relaxed to single precision
- step 2: single precision variables → variables relaxed to half precision

Conjugate Gradient code

Sequential version of a CG code from SNU NPB suite

(<http://aces.snu.ac.kr/software/snu-npb>).

The code solves a linear system with a matrix of size 7,000 with 8 non-zero values per row.

After 15 CG iterations:

# req. digits	# exec	# half-# single-# double	time (s)
1	44	19-6-0	212.71
2	55	18-7-0	235.07
3	53	17-8-0	241.90
4	69	14-11-0	209.08
5	67	12-13-0	197.04
6-7	74	12-13-0	204.96
8	100	10-13-2	256.29
9	89	11-9-5	225.77
10	89	12-5-8	219.10

time: total execution time of PROMISE (compilations, executions, and time spent in PROMISE routines)

MICADO: simulation of nuclear cores

code developed by EDF (French energy supplier)

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

# req. digits	# single - # double	speed up	memory gain
10	32-19	1.01	1.00
8	33-18	1.01	1.01
6	38-13	1.20	1.44
5	51-0	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

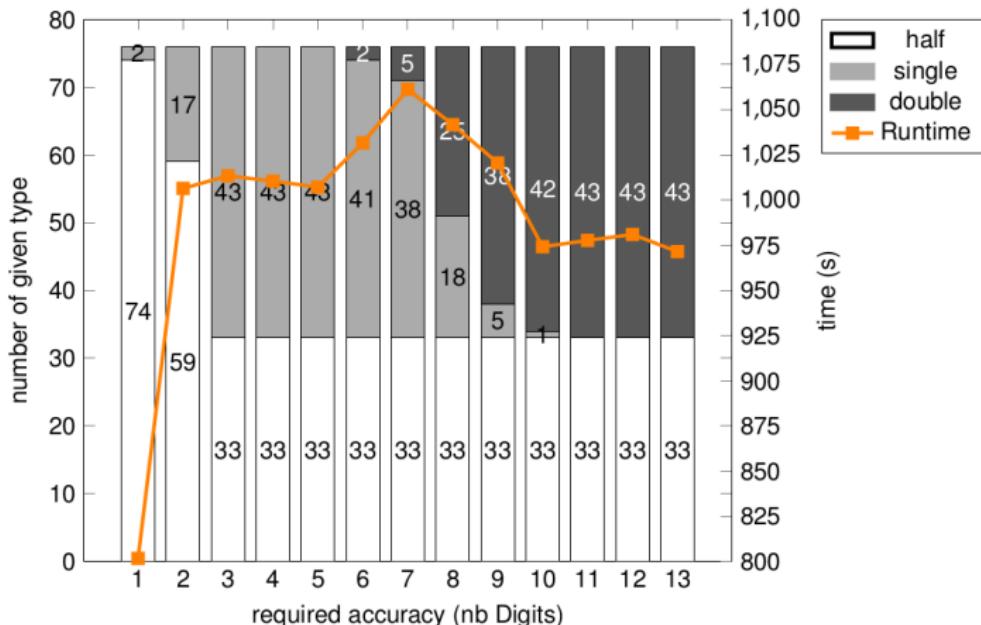
In progress: precision autotuning of neural networks

Keras/PyTorch models → mixed precision C++ codes

suitable precision for each node or each layer

various models studied: interpolator network, MNIST, CIFAR, pendulum

results for MNIST:



Perspectives

- CADNA/PROMISE extension to other formats such as bfloat16
- precision autotuning:
 - extension of PROMISE to GPUs
 - parallel search for a suitable type configuration
 - floating-point autotuning in arbitrary precision (possible target: FPGAs)
- combination of mixed precision algorithms and floating-point autotuning

References

- ❑ J. Vignes, Discrete Stochastic Arithmetic for Validating Results of Numerical Software, *Num. Algo.*, 37, 1–4, p. 377–390, 2004.
 - ❑ P. Eberhart, J. Brajard, P. Fortin, and F. Jézéquel, High Performance Numerical Validation using Stochastic Arithmetic, *Reliable Computing*, 21, p. 35–52, 2015.
<https://hal.archives-ouvertes.fr/hal-01254446>
 - ❑ S. Graillat, F. Jézéquel, R. Picot, F. Févotte, and B. Lathuilière, Auto-tuning for floating-point precision with Discrete Stochastic Arithmetic, *J. Computational Science*, 36, 2019.
<https://hal.archives-ouvertes.fr/hal-01331917>
 - ❑ F. Jézéquel, S. sadat Hoseininasab, T. Hilaire, Numerical validation of half precision simulations, 1st Workshop on Code Quality and Security (CQS 2021), WorldCIST'21, 2021.
<https://hal.archives-ouvertes.fr/hal-03138494>
-
- **CADNA**: <http://cadna.lip6.fr>
 - **SAM**: <http://www-pequan.lip6.fr/~jezequel/SAM>
 - **PROMISE**: <http://promise.lip6.fr>

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, Antoine Quedeville, 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, Antoine Quedeville, Jonathon Tidswell, Su Zhou, ...

Thank you for your attention!