Precision auto-tuning and control of accuracy in numerical simulations

Fabienne Jézéquel

LIP6, Sorbonne Université, France

Compute Accelerator Forum
CERN
26 April 2023

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.

1. How to control the validity of (mixed precision) 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
  - 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]

Classic arithmetic

$A \oplus B \longrightarrow R$

$R =$3.14237654356891

DSA

Random rounding

$A_1 \oplus B_1 \longrightarrow R_1$

$A_2 \oplus B_2 \longrightarrow R_2$

$A_3 \oplus B_3 \longrightarrow R_3$

$R_1 =$**3.14**1354786390989
$R_2 =$**3.14**3689456834534
$R_3 =$**3.14**2579087356598

- each operation executed 3 times with a random rounding mode

Discrete Stochastic Arithmetic (DSA) [Vignes'04]

Classic arithmetic

$A \oplus B \longrightarrow R$

$R =$3.14237654356891

DSA

Random rounding

$A_1 \oplus B_1 \longrightarrow R_1$

$A_2 \oplus B_2 \longrightarrow R_2$

$A_3 \oplus B_3 \longrightarrow R_3$

$R_1 =$**3.14**1354786390989
$R_2 =$**3.14**3689456834534
$R_3 =$**3.14**2579087356598

- 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%

Discrete Stochastic Arithmetic (DSA) [Vignes'04]

Classic arithmetic

$A \oplus B \longrightarrow R$

$R =$3.14237654356891

DSA

Random rounding

$A_1 \oplus B_1$  $\longrightarrow R_1$

$A_2 \oplus B_2$  $\longrightarrow R_2$

$A_3 \oplus B_3$  $\longrightarrow R_3$

$R_1 =$**3.14**1354786390989
$R_2 =$**3.14**3689456834534
$R_3 =$**3.14**2579087356598

- 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
  - $\Rightarrow$ detection of numerical instabilities
    Ex: `if (A>B)` with `A-B` numerical noise
  - $\Rightarrow$ 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) of various precisions that can be mixed together or with classic types
- 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

`half_st` `float_st` `double_st` `float128_st`

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)

Quadruple precision in CADNA

control of fp128 computation based on

- `__float128` (with gcc)
- `_Quad` (with icc)

CADNA performance

CADNA cost

- memory: 4
- run time $\approx$ 10

Efficient rounding mode change

- implicit change of the rounding mode thanks to

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

$\bigcirc_{+\infty}$ (resp. $\bigcirc_{-\infty}$): 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 MPFR[1])
mp_st stochastic type

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

[1] www.mpfr.org

The SAM library

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

SAM (Stochastic Arithmetic in Multiprecision)

implements stochastic arithmetic in arbitrary precision (based on MPFR[1])

mp_st stochastic type

operator overloading ⇒ 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;

$$C = A \oplus B$$

35 bits   23 bits   47 bits

⇒ accuracy estimation on FPGA

[1] www.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=2.00000000000000e+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)

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

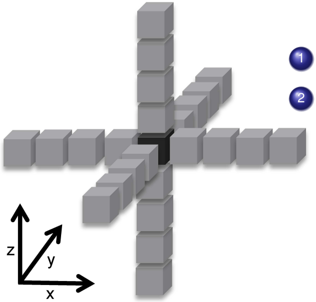
- time: order 2
- space: order p (in our case $p = 8$).

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

Results depend on:

- the implementation of the finite difference scheme
- the compiler / architecture (various CPUs and GPUs used)

In *binary32*, 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	**-1.11**0479E+0	**5.454**238E+1	**6.1410**38E+2
2	**-1.11**0426E+0	**5.454**199E+1	**6.1410**35E+2
NVIDIA C2050 GPU with CUDA			
1	**-1.11**0204E+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	**-1.11**0111E+0	**5.454**170E+1	**6.1410**44E+2
AMD Trinity APU with OpenCL			
1	**-1.11**0023E+0	**5.454**169E+1	**6.1410**62E+2
2	**-1.11**0113E+0	**5.454**261E+1	**6.1410**49E+2

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	**-1.11**0479E+0	**5.454**238E+1	**6.1410**38E+2
2	**-1.11**0426E+0	**5.454**199E+1	**6.1410**35E+2
NVIDIA C2050 GPU with CUDA			
1	**-1.11**0204E+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	**-1.11**0111E+0	**5.454**170E+1	**6.1410**44E+2
AMD Trinity APU with OpenCL			
1	**-1.11**0023E+0	**5.454**169E+1	**6.1410**62E+2
2	**-1.11**0113E+0	**5.454**261E+1	**6.1410**49E+2

How to estimate the impact of rounding errors?

The 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 wave propagation code examined with CADNA

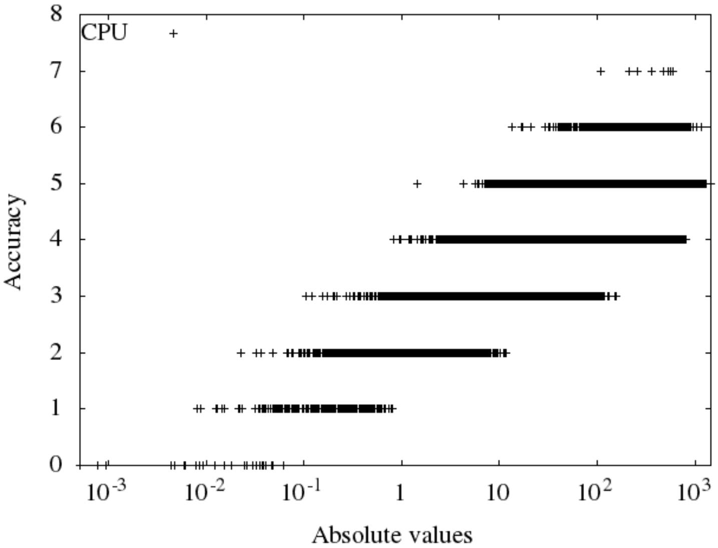
Results computed at 3 different points using scheme 1:

	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

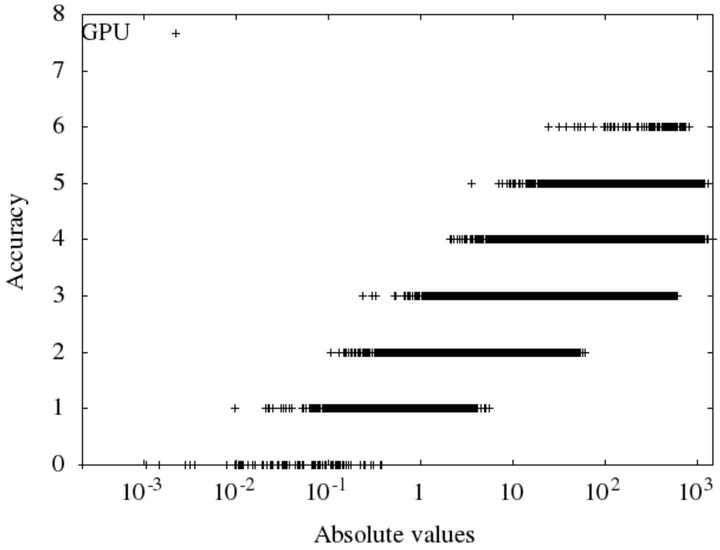
Despite differences in the estimated accuracy, the same trend can be observed on CPU and on GPU.

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

Accuracy distribution on CPU

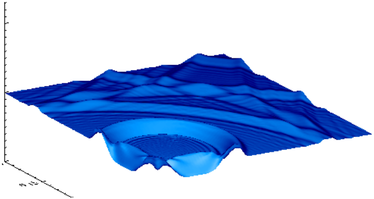


Accuracy distribution on GPU



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

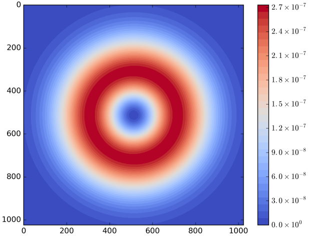
- Simulation of the evolution of water height and velocities in a 2D oceanic basin
- CUDA GPU code in double precision



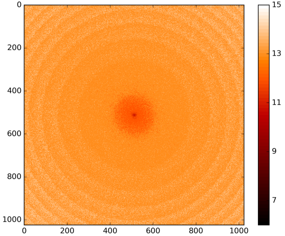
- Focusing on an eddy evolution: 20 time steps (12 hours of simulated time) on a 1024×1024 grid

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**

Tools related to CADNA

available on `cadna.lip6.fr`

- CADNAIZER
  - automatically transforms C/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	$\approx$ 5 to 8
VERIFICARLO	$\approx$ 300 to 600
VERROU	$\approx$ 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 ≈ as accurate as with twice the working precision

- accurate and reproducible BLAS
  - ExBLAS [Collange & al.'15]
  - RARE-BLAS [Chohra & al.'16]
  - Repro-BLAS [Ahrens & 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

  survey: [Higham & Mary'22]

- precision autotuning

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 =$2.571784e+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 =$2.571784e+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 =$2.571784e+29
- double: $P =$1.17260394005318
- quad: $P =$1.17260394005317863185883490452018
- exact: $P \approx$-0.827396059946821368141165095479816292

PROMISE (PRecision OptiMISE) [Graillat & al'19], [F.J. & al'21]

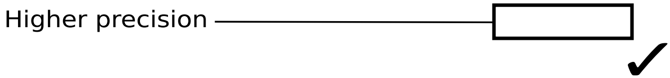
promise.lip6.fr

PROMISE

- provides a mixed precision code (half, single, double) 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.

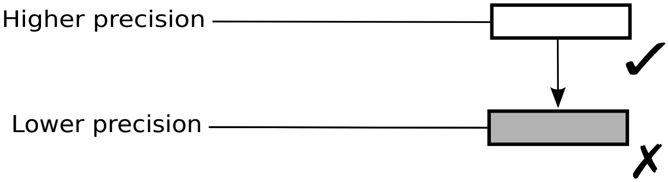
Searching for a valid configuration with 2 types

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



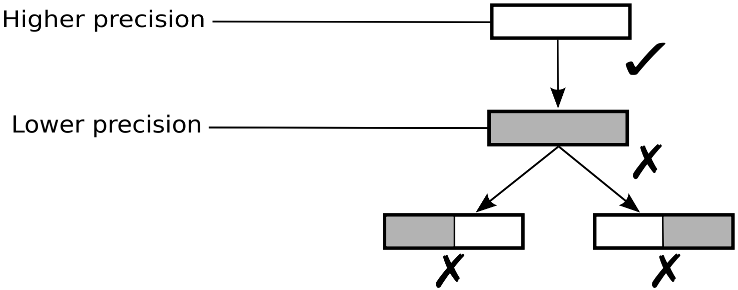
Searching for a valid configuration with 2 types

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



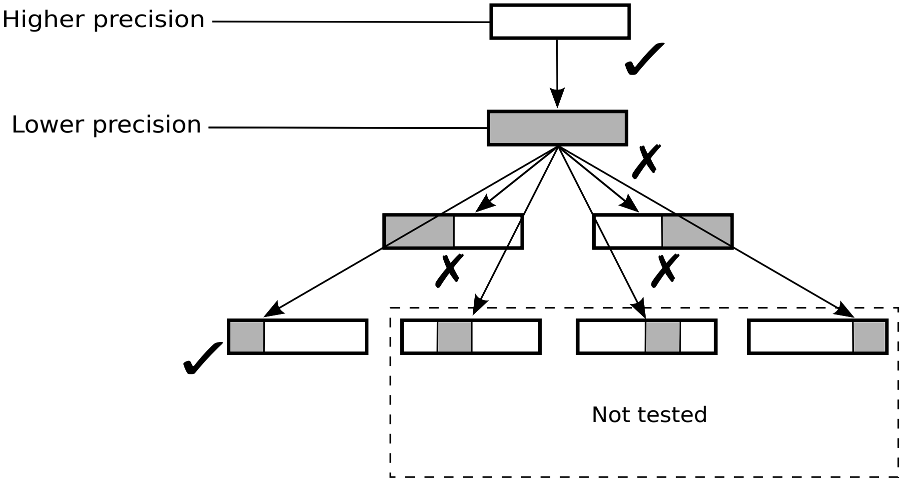
Searching for a valid configuration with 2 types

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



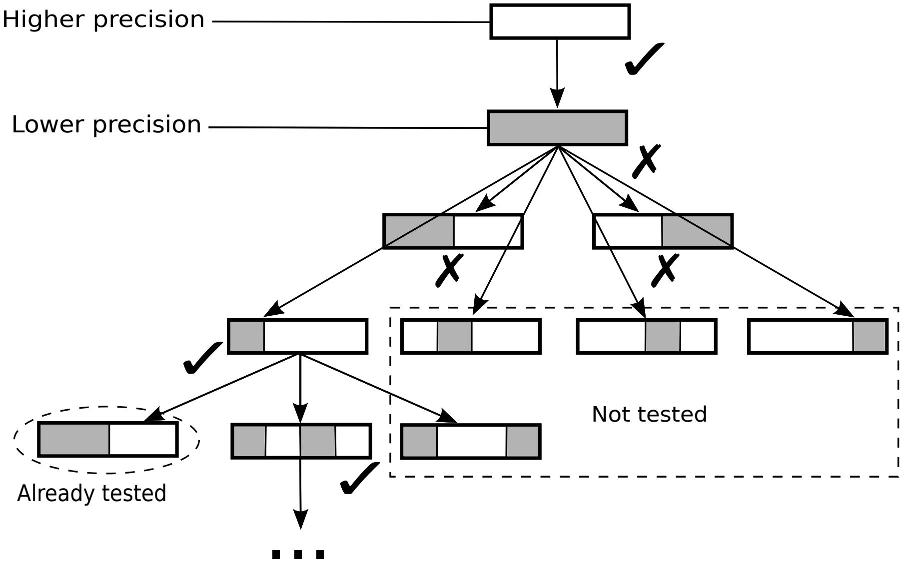
Searching for a valid configuration with 2 types

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

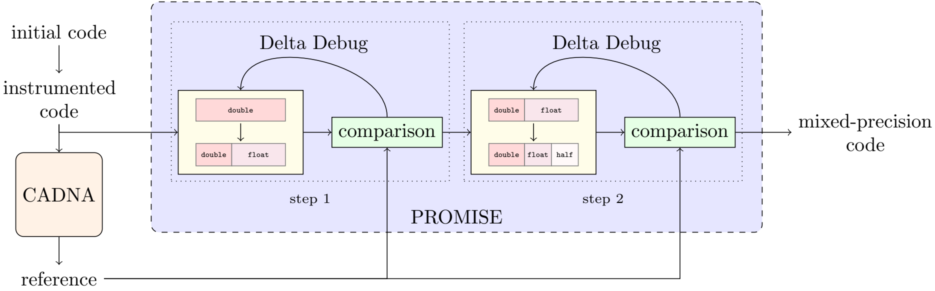


Searching for a valid configuration with 2 types

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



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[2]
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)

[2] http://aces.snu.ac.kr/software/snu-npb

MICADO: simulation of nuclear cores

code developed by EDF (French energy supplier)

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

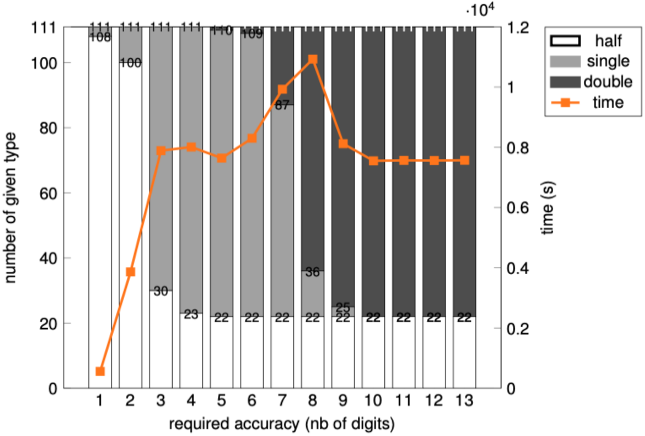
# req. digits	# single - # double	speed up	memory gain
10	32-19	1.01	1.00
8	33-18	1.01	1.01
6	38-13	1.20	1.44
5 4	51-0	1.32	1.62

- 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

Precision autotuning of neural networks

Work in progress

Example: classification network for CIFAR (5 layers, 111 types to set)



Q. Ferro, S. Graillat, T. Hilaire, F. Jézéquel, B. Lewandowski, Neural Network Precision Tuning Using Stochastic Arithmetic, 15th Int. Workshop on Numerical Software Verification, 2022.
`https://hal.archives-ouvertes.fr/hal-03682645`

Numerical validation of C++ codes related to AGATA

Work in progress

AGATA: European gamma-ray spectrometer used for nuclear structure studies
`https://www.agata.org`

Collaboration with IJCLab (Orsay, France)

- impact of precision (half, single, double) on accuracy
- mixed precision version

D. Chamont, R. Molina, V. Lafage, F. Jézéquel, Investigating mixed-precision for AGATA pulse-shape analysis, 26th International Conference on Computing in High Energy and Nuclear Physics (CHEP 2023), Norfolk, USA, May 2023.

`https://hal.archives-ouvertes.fr/hal-03981310`

Conclusion/Perspectives

To optimize precision

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

Perspectives

- extension of CADNA/PROMISE to other formats such as bf16
- extension of PROMISE to GPUs
- floating-point autotuning in arbitrary precision
- 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, Quentin Ferro, 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, Quentin Ferro, 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!