Neural Network Precision Auto-Tuning and PROMISE Improvement

Quentin Ferro Stef Graillat Thibault Hilaire Fabienne Jézéquel
LIP6/PEQUAN, Sorbonne Université, CNRS, France

8-9 June 2023

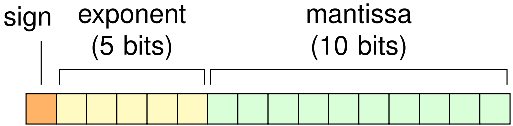
Table of Contents

Table of Contents

Introduction

IEEE754 Standard types

Format	Name	Length	Sign	Mantissa Length	Exponent Length
binary16	Half	16 bits	1 bit	11 bits	5 bits
binary32	Single	32 bits	1 bit	24 bits	8 bits
binary64	Double	64 bits	1 bit	53 bits	11 bits

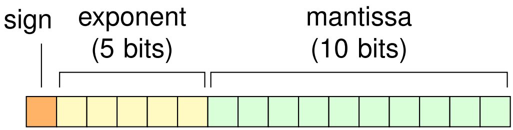


binary16 format

Introduction

IEEE754 Standard types

Format	Name	Length	Sign	Mantissa Length	Exponent Length
binary16	Half	16 bits	1 bit	11 bits	5 bits
binary32	Single	32 bits	1 bit	24 bits	8 bits
binary64	Double	64 bits	1 bit	53 bits	11 bits



binary16 format

Reduced precision:

- Shorter execution time ☺
- Less volume of results exchanged (less memory used) ☺
- Less energy consumption ☺

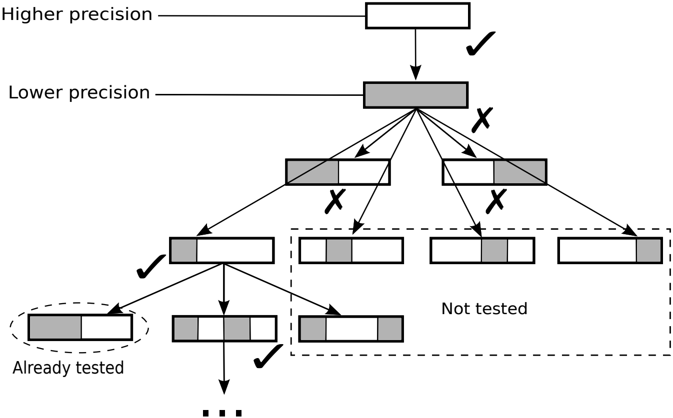
CADNA Software cadna.lip6.fr [Chesneaux, 1995]

- implements stochastic arithmetic for C/C++ or Fortran codes
- provides stochastic types: 3 values of a variable + 1 integer being the accuracy
- returns value with the exact number of correct digits

PROMISE promise.lip6.fr ([Graillat et al., 2019], [Jézéquel et al., 2021])

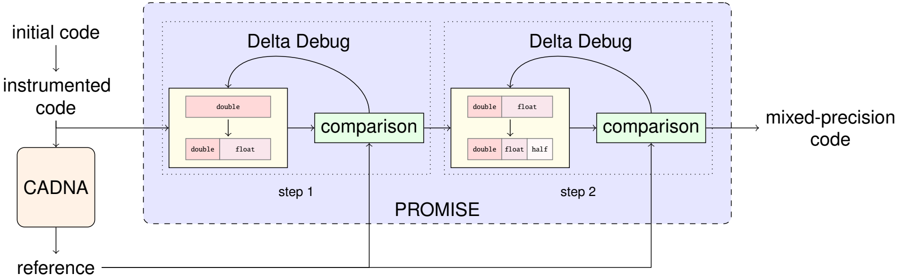
- Provides a mixed-precision code taking in account a required accuracy
- Uses CADNA to validate a configuration
- Uses the Delta-Debug algorithm to test the different configurations, not exhaustive but mean complexity in $O(nlog(n))$ for n variables [Zeller, 2019]

Delta-Debug with 2 Types



PROMISE

instrumented code = code with PROMISE variables, custom types variables that PROMISE recognizes and will consider tweaking



- step 1: lower from double to single precision
- step 2: lower from single to half precision

Table of Contents

Neural networks application

Application of PROMISE on 4 neural networks:

- Sine NN: 3 layers, densely-connected interpolation network approximating the sine function
- MNIST NN: 2 layers, densely-connected classification network based on the MNIST database
- CIFAR NN: 5 layers, convolutional classification network based on the CIFAR10 database
- Pendulum NN: 2 layers, densely-connected interpolation network approximating the Lyapunov function of an inverted pendulum [Chang et al., 2020]

Neural networks application

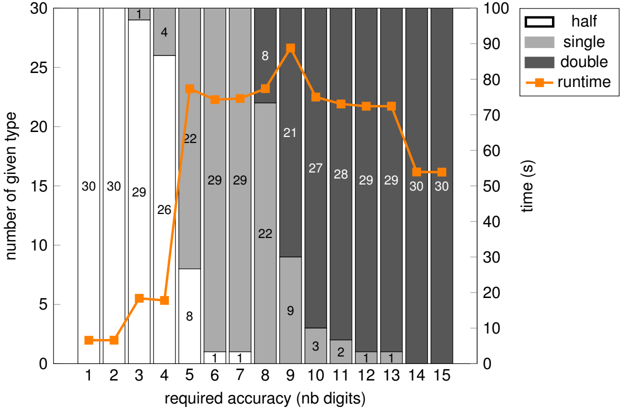
Application of PROMISE on 4 neural networks:

- Sine NN: 3 layers, densely-connected interpolation network approximating the sine function
- MNIST NN: 2 layers, densely-connected classification network based on the MNIST database
- CIFAR NN: 5 layers, convolutional classification network based on the CIFAR10 database
- Pendulum NN: 2 layers, densely-connected interpolation network approximating the Lyapunov function of an inverted pendulum [Chang et al., 2020]

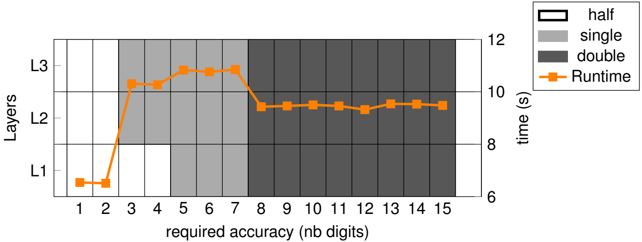
Gave us the different configurations with two approaches :

- One type per neuron
- One type per layer

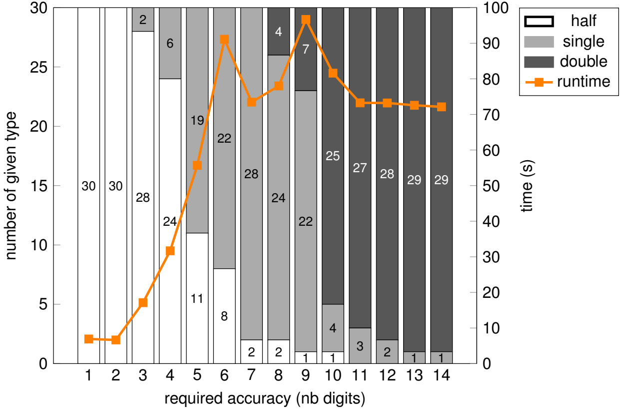
Sine NN w/ input=0.5



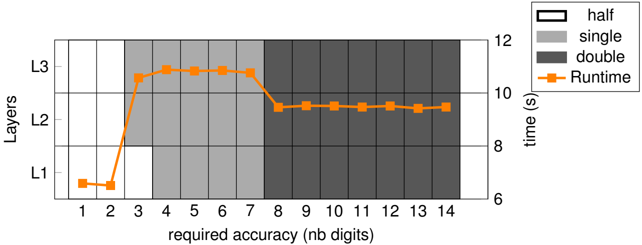
Sine NN w/ input=0.5



Sine NN w/ input=-2.37



Sine NN w/ input=-2.37

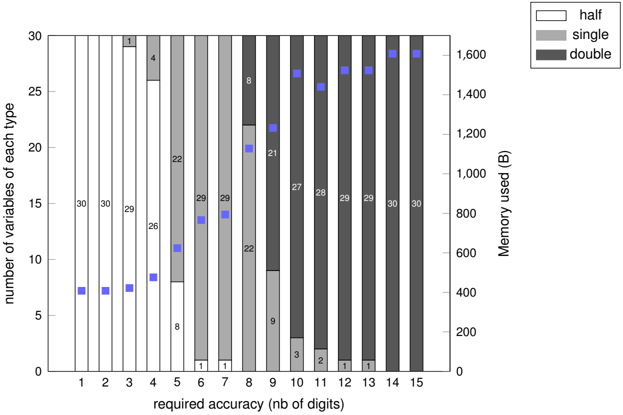


Memory gain with mixed precision

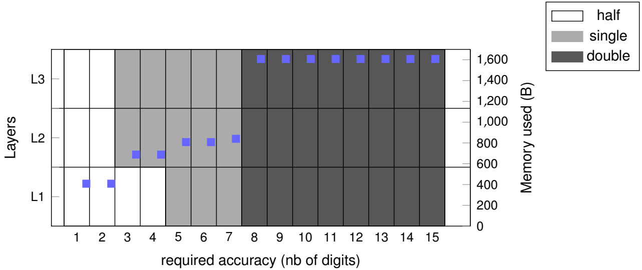
- First analysis done with Valgrind's Massif tool
- Consistent with theoretical values (+ some overhead when declaring pointers)

$\longrightarrow$ We consider the theoretical values

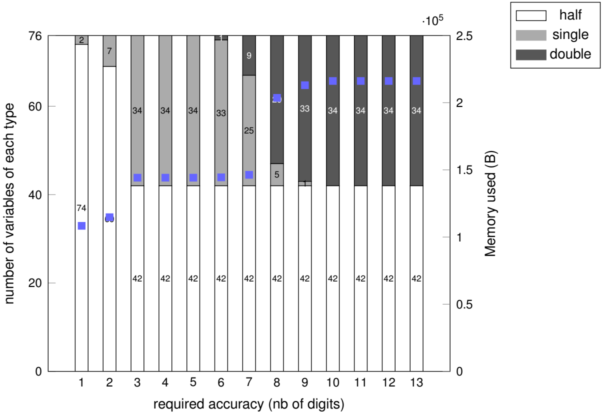
Memory gain for Sine NN w/ input=0.5

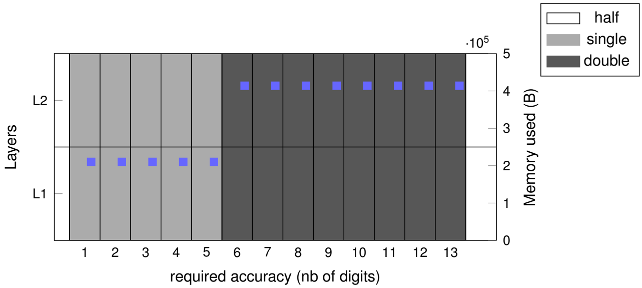


Memory gain for Sine NN w/ input=0.5

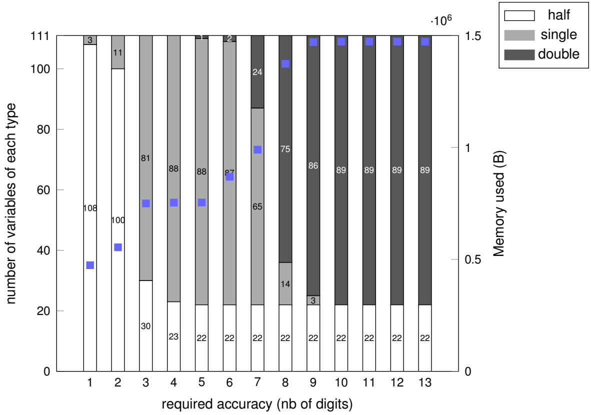


Memory gain for MNIST NN w/ input=test_data[61]

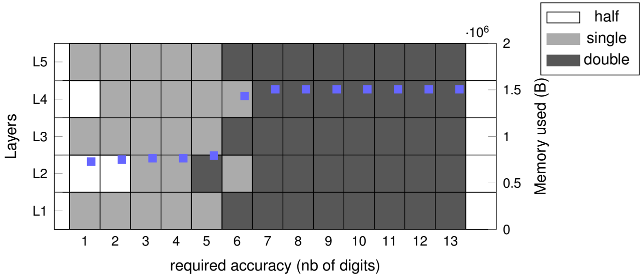



half
single
double
$\cdot 10^5$
5
4
3
2
1
0
Memory used (B)
Layers
L2
L1
1
2
3
4
5
6
7
8
9
10
11
12
13
required accuracy (nb of digits)

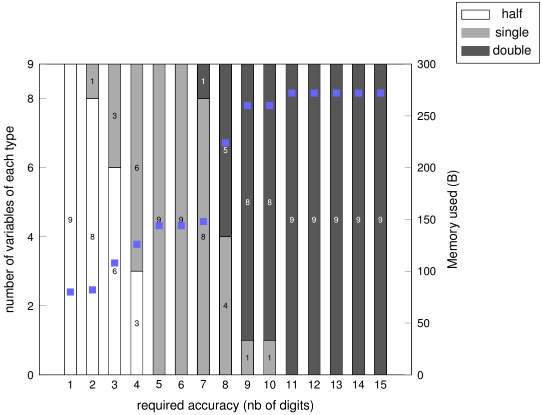
Memory gain for CIFAR NN w/ input=test_data[386]



Memory gain for CIFAR NN w/ input=test_data[386]



Memory gain for PENDULUM NN w/ input=(0.5,0.5)



Memory gain for PENDULUM NN w/ input=(0.5,0.5)

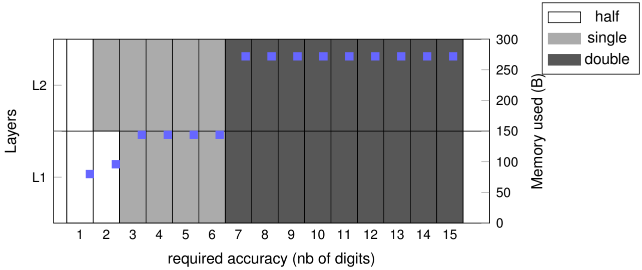
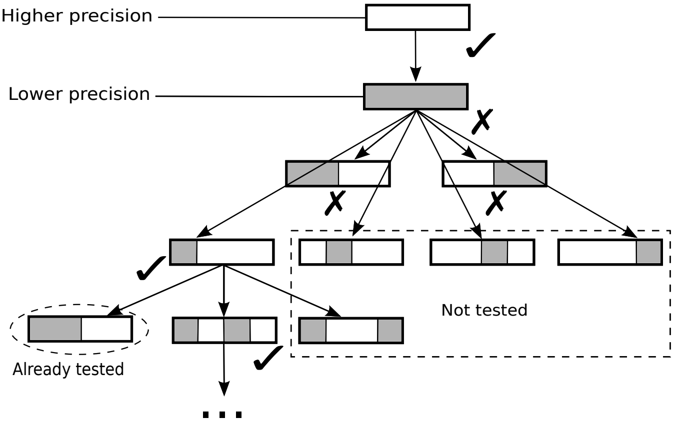


Table of Contents

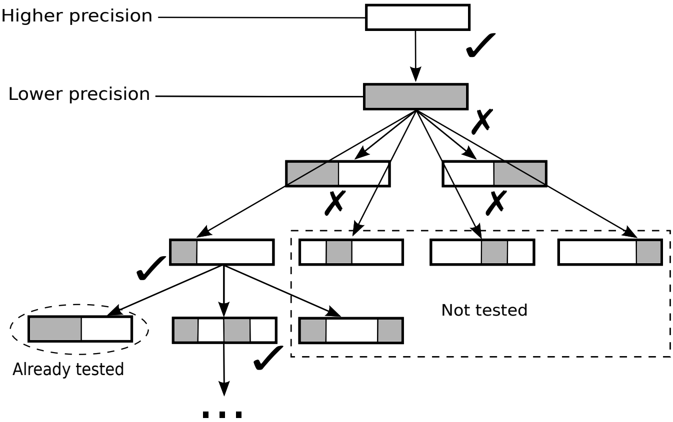
Parallelization of Delta-Debug

- Based on the parallelization described in [Hodován and Kiss, 2016] and developed in the tool Picire (github.com/renatahodovan/picire)
- Use Python's multiprocessing module to launch multiple processes in parallel
- Test multiple configurations in parallel at one level of the Delta-Debug

Parallelization of Delta-Debug

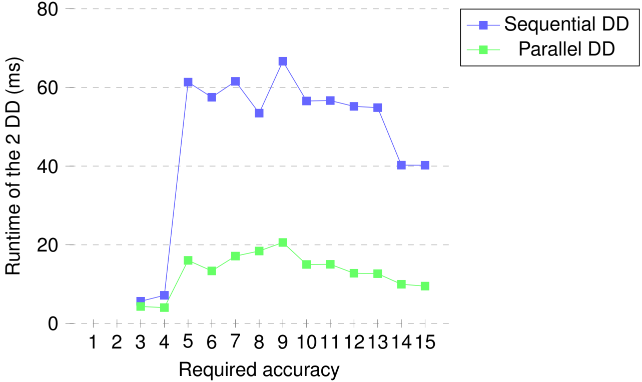


Parallelization of Delta-Debug

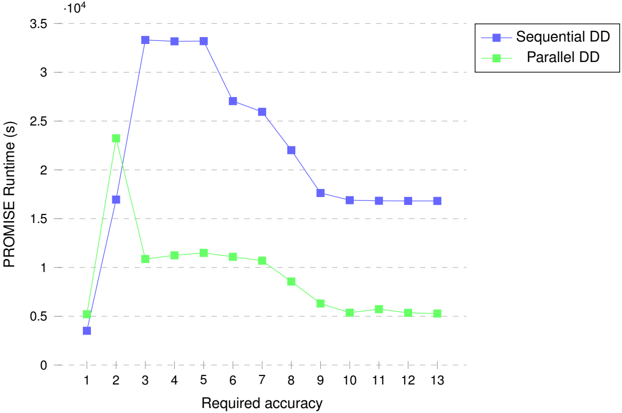


- 6 core CPU - 6 processes in parallel

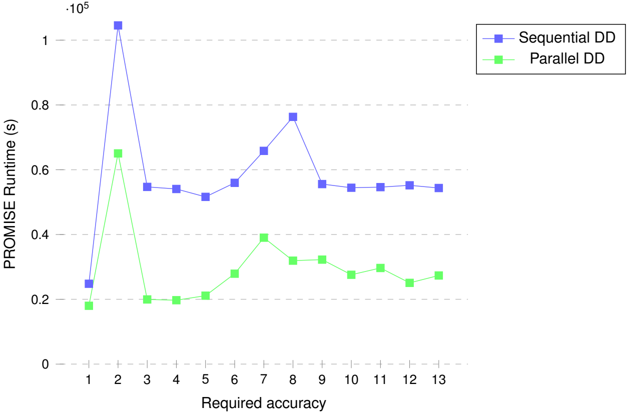
Delta-Debug on Sine NN



PROMISE Parallel results on MNIST



PROMISE Parallel results on CIFAR



Instrumentation: before

- Use of a 'cadnaizer' which turns every floating points types into stochastic type
- Perl script

- PROMISE instrumentation has to be done by hand

Instrumentation: before

```
double fun(double x){
  int k, n = 5;
  double t1;
  double d1 = 1.0;

  t1 = x;
  for ( k = 1; k <= n; k++ )
    {
      d1 = 2.0 * d1;
      t1 = t1+ sin(d1 * x)/d1;
    }
  return t1;
}

int main( int argc, char **argv) {

  int i,n = 1000000;
  double h;
  double t1, t2, dppi;
  double s1;
  std::ofstream res;
  std::cout.precision(15);

  t1 = -1.0;
  dppi = acos(t1);
  s1 = 0.0;
  t1 = 0.0;
  h = dppi / n;

  for ( i = 1; i <= n; i++)
    {
      t2 = fun(i * h);
      s1 = s1 + sqrt(h*h + (t2 - t1) * (t2 - t1));
      t1 = t2;
      //if (i%1000==0) PROMISE_CHECK_VAR(t1);
    }

  std::cout << s1 << std::endl;

  return 0;
}
```

Instrumentation: before

```
double fun(double x){
  int k, n = 5;
  double t1;
  double d1 = 1.0;

  t1 = x;
  for ( k = 1; k <= n; k++ )
    {
      d1 = 2.0 * d1;
      t1 = t1+ sin(d1 * x)/d1;
    }
  return t1;
}

int main( int argc, char **argv) {

  int i,n = 1000000;
  double h;
  double t1, t2, dppi;
  double s1;
  std::ofstream res;
  std::cout.precision(15);

  t1 = -1.0;
  dppi = acos(t1);
  s1 = 0.0;
  t1 = 0.0;
  h = dppi / n;

  for ( i = 1; i <= n; i++)
    {
      t2 = fun(i * h);
      s1 = s1 + sqrt(h*h + (t2 - t1) * (t2 - t1));
      t1 = t2;
      //if (i%1000==0) PROMISE_CHECK_VAR(t1);
    }

  std::cout << s1 << std::endl;

  return 0;
}
```

```
__PR_fun__ fun(__PR_1__ x){
  int k, n = 5;
  __PR_fun__ t1;
  __PR_1__ d1 = 1.0;

  t1 = x;
  for ( k = 1; k <= n; k++ )
    {
      d1 = 2.0 * d1;
      t1 = t1+ sin(d1 * x)/d1;
    }
  return t1;
}

int main( int argc, char **argv) {

  int i,n = 1000000;
  __PR_1__ h;
  __PROMISE__ t1, t2, dppi;
  __PROMISE__ s1;
  std::ofstream res;
  std::cout.precision(15);

  t1 = -1.0;
  dppi = acos(t1);
  s1 = 0.0;
  t1 = 0.0;
  h = dppi / n;

  for ( i = 1; i <= n; i++)
    {
      t2 = fun(i * h);
      s1 = s1 + sqrt(h*h + (t2 - t1) * (t2 - t1));
      t1 = t2;
      //if (i%1000==0) PROMISE_CHECK_VAR(t1);
    }

  std::cout << s1 << std::endl;
  PROMISE_CHECK_VAR(s1);
  return 0;
}
```

Instrumentation: now

'Instrumentizer' using clang AST (Abstract Syntax Tree)

- 'Cadnaizer' using instrumentizer

- PROMISE instrumentation using instrumentizer
- Instrumentation can be done on chosen parts of the code

How does it works?

- Clang AST contains the information of the code as nodes in a tree structure
- A node can be a declaration of function (FuncDecl), an expression in parenthesis (ParenExpr) or even just a literal (FloatingLiteral, IntegerLiteral)

From the AST, we can match some nodes and directly change it, or run from there through the AST to replace what we want

Examples

```
double a = 3.14;
```

Examples

```
double a = 3.14;
```

```
|-DeclStmt 0x14c0408c0 <line:2:5, col:20>
| `-VarDecl 0x14c040838 <col:5, col:16> col:12 a 'double' cinit
|   `-FloatingLiteral 0x14c0408a0 <col:16> 'double' 3.140000e+00
```

Examples

```
double a = 3.14;
```

```
|-DeclStmt 0x14c0408c0 <line:2:5, col:20>
| `-VarDecl 0x14c040838 <col:5, col:16> col:12 a 'double' cinit
|   `-FloatingLiteral 0x14c0408a0 <col:16> 'double' 3.140000e+00
```

- Match VarDecl

Examples

```
double a = 3.14;
```

```
|-DeclStmt 0x14c0408c0 <line:2:5, col:20>
| `-VarDecl 0x14c040838 <col:5, col:16> col:12 a 'double' cinit
|   `-FloatingLiteral 0x14c0408a0 <col:16> 'double' 3.140000e+00
```

- Match VarDecl
- Replace the type 'double'

Examples

```
double_st a = 3.14;
```

Examples

```
double_st a = 3.14;
```

```
__PR_1__ a = 3.14;
```

Examples

"Smart" replacement approach, keeping some consistency

```
float a;
double d;
d = (double) 4*a;
```

Examples

"Smart" replacement approach, keeping some consistency

```
float a;
double d;
d = (double) 4*a;
```

```
__PR_1__ a;
__PR_2__ d;
d = (__PR_2__) 4*a;
```

Stochastic Arithmetic in LLVM backend (Work in progress)

- Based on NSan - Numerical Stability Sanitizer [Courbet, 2021] and INSanE - Interface for Numerical Stability Sanitizer Extension, developed by Mathys Jam, Pablo Oliveira, Eric Petit)
- Modifies LLVM-IR to do the desired computation in shadow memory

Stochastic Arithmetic in LLVM backend (Work in progress)

- Based on NSan - Numerical Stability Sanitizer [Courbet, 2021] and INSanE - Interface for Numerical Stability Sanitizer Extension, developed by Mathys Jam, Pablo Oliveira, Eric Petit)
- Modifies LLVM-IR to do the desired computation in shadow memory

- Use INSANE interface to implement stochastic arithmetic (CADNA)

Table of Contents

Conclusion

- Reduction of memory consistent with theoretical values thanks to PROMISE
- Improvement of PROMISE in speed thanks to parallelization
- New tool to create PROMISE version of code and CADNA version of code

Future works

- Time gain of the neural networks thanks to PROMISE (work in progress)
- Instrumentizer and cadnaizer on MPI code (work in progress)
- Comparison of results with loop-splitting and multi-precision programs
  Exchange with Youssef Fakhreddine (DALI - Perpignan) (12/06-16/06)

Table of Contents

References I

[Chang et al., 2020] Chang, Y.-C., Roohi, N., and Gao, S. (2020).
Neural Lyapunov Control.
33rd Conference on Neural Information Processing Systems (NeurIPS 2019).
arXiv: 2005.00611.

[Chesneaux, 1995] Chesneaux, J.-M. (1995).
L'arithmétique stochastique et le logiciel CADNA, Habilitation à diriger des recherches.
Université Pierre et Marie Curie, Paris, France.

[Courbet, 2021] Courbet, C. (2021).
NSan: a floating-point numerical sanitizer.
In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, pages 83–93, Virtual Republic of Korea. ACM.

References II

[Graillat et al., 2019] Graillat, S., Jézéquel, F., Picot, R., Févotte, F., and Lathuilière, B. (2019).
Auto-Tuning for Floating-Point Precision with Discrete Stochastic Arithmetic.
Journal of Computational Science, 36:101017.

[Hodován and Kiss, 2016] Hodován, R. and Kiss, A. (2016).
Practical Improvements to the Minimizing Delta Debugging Algorithm:.
In *Proceedings of the 11th International Joint Conference on Software Technologies*, pages 241–248, Lisbon, Portugal. SCITEPRESS - Science and Technology Publications.

[IEEE Computer Society, 2008] IEEE Computer Society (2008).
IEEE Standard for Floating-Point Arithmetic.
IEEE Standard 754-2008.

References III

[Vignes, 2004] Vignes, J. (2004).
Discrete Stochastic Arithmetic for Validating Results of Numerical Software.

Numerical Algorithms, 37(1–4):377–390.

[Zeller and Hildebrandt, 2002] Zeller, A. and Hildebrandt, R. (2002).
Simplifying and Isolating Failure-Inducing Input.
IEEE Trans. Softw. Eng., 28(2):183–200.