Floating-Point Error Estimation Using Automatic Differentiation

141st Parallelism, Performance and Programming Model Meeting, Sept. 2022

Garima Singh

Motivation: Floating-Point Errors

- Floating-point (FP) errors can have terrible implications for programs with high-precision calculations or simple, but repetitive computations.

```
double c = -5e13;
for (unsigned int i = 0; i < 1e8; i++){
   if (i % 2) c = c - 1e-6;
   else c = c + 1e6;
}
```

Exact solution,
$c = -5 \times 10^{13} + ½ * 10^8 * 10^6 - ½ * 10^8 * 10^{-6}$
$= \mathbf{-50}$

*Rounding Mode	c
Rounded to the nearest	−0.02460
Rounded towards −∞	-207373.07020
Rounded towards +∞	−0.00820
Rounded towards 0	−0.00820

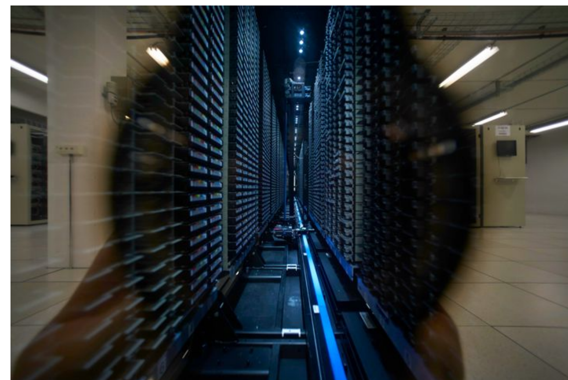
*IEEE-754 double precision. Example from: Problem — True North Floating Point

- Using higher precision is one solution, but this leads to larger program sizes. Other approaches require reimplemenation to prevent error accumulation (such as Kahan summation).

Floating Point Errors in Data Intensive Science

- Data intensive sciences must cope with a rapidly increasing data volume and a transition to a fully heterogeneous computing environment. Results are even more sensitive to floating-point errors in critical applications.
- For example: float vs. double can make a large difference in performance as many GPUs either do not provide double-precision float, do not follow the IEEE compliance or have fairly slow access.

- High Energy Physics is an example field seeing a large influx of data. Huge data rates (100s PB/yr this decade) require selectively saving data and to optimally save the data to optimize as much space as possible. As such, robust detection of floating-point errors can foster development of important new lossy compression algorithms.



In October 2017 the CERN data centre broke its own record for data storage when it collected 12.3 petabytes of data over a single month. Breaking data records bit-by-bit

Our Approach to Estimating FP Errors

- A general representation of floating point errors in arbitrary dimensions is then:

$$A_f \equiv \sum_{i=0}^{n} \frac{\partial f}{\partial x_i} \cdot |x_i| \cdot |\varepsilon_M| + E_L$$

We can get this from AD.

We know this - machine dependant.

Approximation error - hard to estimate.

A_f The absolute error in a function f.

x_i All input and intermediate variables.

E_L The error due to linearization of the Taylor series expansion.

ε_M The upper bound on the relative approximation error due to rounding. Machine dependent.

$\frac{\partial f}{\partial x_i}$ The derivative of f with respect to x_i.

- This formula gives a good upper bound estimate to A_f, and serves our general purpose use case. Automatic differentiation (AD) is an efficient means to compute the needed derivatives

AD Evaluates the Exact Derivative of a Function

- AD applies the chain rule of differential calculus throughout the semantics of the original program.
- For a complex nested function, two recursive relationships calculate the derivative.

$$y = f(g(h(x))) = f(g(h(w_0))) = f(g(w_1)) = f(w_2) = w_3$$

$$w_0 = x$$
$$w_1 = h(w_0)$$
$$w_2 = g(w_1)$$
$$w_3 = f(w_2) = y$$

Reverse-mode AD: $$\frac{dy}{dw_i} = \frac{dy}{dw_{i+1}} \frac{dw_{i+1}}{dw_i}$$

Forward-mode AD: $$\frac{dw_i}{dx} = \frac{dw_i}{dw_{i-1}} \frac{dw_{i-1}}{dx}$$

- There are multiple approaches to implementing AD on programs, including operator Overloading and Source Transformation.
- In the context of FP error estimation, we are more concerned with reverse-mode AD as it provides the derivative of the function with respect to all intermediate and input variables.

Clad: Source-Transformation AD Tool

- Clad is implemented as a plugin to the clang compiler. It inspects the internal compiler representation of the target function to generates its derivative.

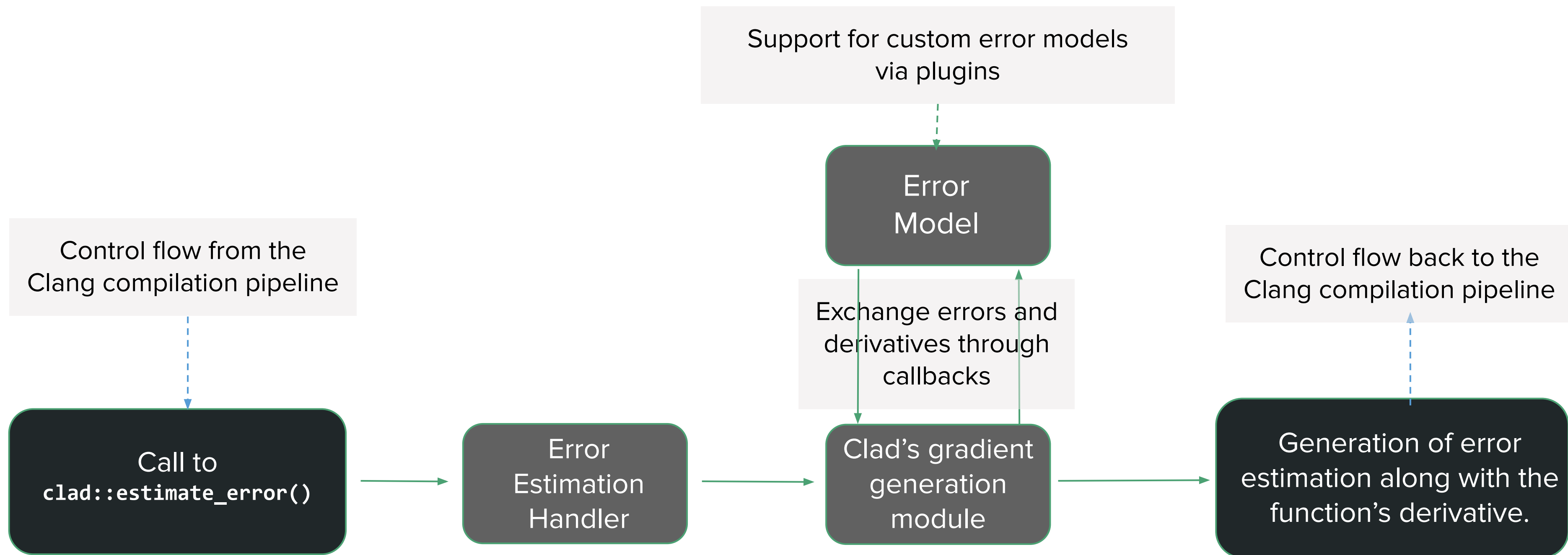
```
double sqr(double x){
      return x * x;
}
```

```
double sqr_darg0(double x){
      double _d_x = 1;
      return _d_x * x + x * _d_x;
}
```

Advantages of Clad’s approach

- Requires little or no code modification for computing derivatives of existing codebase.
- Supports a growing subset of C++ constructs, statements and data-types as well as differentiating CUDA-based programs.
- Enables efficient gradient computation (independent time complexity from inputs) in its reverse accumulation mode enabling scalable FP error-estimation.
- Well integrated into the compiler, allowing automatic generation of error estimation code.

AD-Based FP Error Estimation Framework



AD-Based FP Error Estimation Framework

```
float func(float x, float y) {
 float z;
 z = x + y;
 return z;
}
```

A function we want to estimate the error of.

```
auto df = clad::estimate_error(func);
```

In the main function, we call a function that tells clad to estimate the error in ‘func’.

```
float x = 1.95e-5, y = 1.37e-7;
float dx = 0, dy = 0;
double fp_error = 0;
df.execute(x, y, &dx, &dy, fp_error);
std::cout << "FP error in func: " << fp_error;
```

Finally, call the generated function through the ‘execute’ interface of clad objects. After the execution, the last parameter will store the accumulated FP error in the function.

Sensitivity Analysis

- Our uncertainty framework enables a sensitivity analysis of all input and intermediate variables to floating point errors and reason about the numerical stability of algorithms..

$$S_{x_i} \equiv \left| \frac{\partial f}{\partial x_i} \cdot x_i \right|$$

- If the sensitivity of any variable is high, then the function is more prone to floating-point errors in computations involving that variable.
- This information can be useful when developing programs. The appropriate precision can be used throughout the program given a requirement on floating-point accuracy of the result.

One example use case is **mixed precision tuning**: “demoting” certain types to lower precision while still maintaining the desired accuracy can be beneficial for optimizing speed, size and precision.

Case Study: Simpson's Rule

- The program on the right side is used to to evaluate the integral of a function using Simpson's rule for numerical integration.

```
// defines f(x) = pi * sin(x * pi)
// integral = 2 over [0, 1]
long double f(long double x) {
 long double pi = M_PI;
 long double tmp = x * pi;
 long double tmp2 = sin(tmp) * pi;
 return tmp2;
}
```

The function is implemented with all variables in extended precision. Is that necessary?

```
long double simpsons(long double a, long double b) {
int n = 1000000;
// calculates the integral of a function f
// over the interval [a, b] for n iterations.
long double h = (b - a) / (2.0 * n);
long double x = a;
long double tmp;
long double fa = f(a), fb = f(b);
long double s1 = fa;

for(int l = 0; l < n; l++) {
  x = x + h;
  s1 = s1 + 4.0 * f(x);
  x = x + h;
  s1 = s1 + 2.0 * f(x);
}

s1 = s1 - fb;
tmp = h / 3.0;
s1 = s1 * tmp;
return s1;
}
```

Example adapted from ADAPT: Algorithmic Differentiation Applied to Floating-Point Precision Tuning.

Clad's Results

Precision configurations `simspons(0, 1)`	**Absolute Error**	**Clad Estimated Error**
10-byte extended precision (*long double*)	4.1e-14	3.1e-12
IEEE double-precision (*double*)	6.8e-11	6.2e-9
IEEE single-precision (*float*)	0.03812	3.31

- Clad provides a good estimate for the asymptotic error bound on the FP errors.
- More accurate estimates can be derived using custom error models with the FP error estimation framework.

```
long double simpsons(long double a, long double b) {
int n = 1000000;
// calculates the integral of a function f
// over the interval [a, b] for n iterations.
long double h = (b - a) / (2.0 * n);
long double x = a;
long double tmp;
long double fa = f(a), fb = f(b);
long double s1 = fa;

for(int l = 0; l < n; l++) {
  x = x + h;
  s1 = s1 + 4.0 * f(x);
  x = x + h;
  s1 = s1 + 2.0 * f(x);
}

s1 = s1 - fb;
tmp = h / 3.0;
s1 = s1 * tmp;
return s1;
}
```

Example adapted from ADAPT: Algorithmic Differentiation Applied to Floating-Point Precision Tuning.

Choosing Variables for Precision Tuning

- How do you pick the right variables to keep in higher precision?

Clad calculates the total FP error contribution of every variable, we can then ask clad to print this information to some file (let's assume the file is called 'errors'). Clad will print the error for each variable as follows:

variable-name: *error-value*

Note: It is also possible to configure what clad prints i.e. the value of 'error-value' can be customized to print derivatives or even sensitivities of variables.

With this information, it is trivial to filter out the variables whose error violates a certain boundary or threshold.

```
long double simpsons(long double a, long double b) {
int n = 1000000;
// calculates the integral of a function f
// over the interval [a, b] for n iterations.
long double h = (b - a) / (2.0 * n);
long double x = a;
long double tmp;
long double fa = f(a), fb = f(b);
long double s1 = fa;

for(int l = 0; l < n; l++) {
  x = x + h;
  s1 = s1 + 4.0 * f(x);
  x = x + h;
  s1 = s1 + 2.0 * f(x);
}

s1 = s1 - fb;
tmp = h / 3.0;
s1 = s1 * tmp;
return s1;
}
```

Example adapted from [ADAPT: Algorithmic Differentiation Applied to Floating-Point Precision Tuning.]

Case Study: Simpson's Rule

Results

Precision configurations	Absolute Error	Clad Estimated Error	Variables in lower precision (out of 11)
10-byte extended precision (*long double*)	4.07e-14	3.1e-12	0
Clad's mixed precision	4.08e-14	3.0e-12	6
IEEE double-precision (*double*)	6.8e-11	6.2e-9	-
IEEE single-precision (*float*)	0.03812	3.31	-

"Demoting" low-sensitivity variables to lower precision improves performance by ~10% in this example.

Here clad's estimate also agrees that there is no significant change in the final error. This can be useful in the cases where an accurate ground-truth comparison is not available.

```
long double f(long double x) {
 long double pi = M_PI;
 long double tmp = x * pi;
 long double tmp2 = sin(tmp) * pi;
 return tmp2;
}
```

```
long double simpsons(double a, double b) {
int n = 1000000;

double h = (b - a) / (2.0 * n);
long double x = a;
double tmp;
double fa = f(a), fb = f(b);
long double s1 = fa;

for(int l = 0; l < n; l++) {
  x = x + h;
  s1 = s1 + 4.0 * f(x);
  x = x + h;
  s1 = s1 + 2.0 * f(x);
}

s1 = s1 - fb;
tmp = h / 3.0;
s1 = s1 * tmp;
return s1;
}
```

Case Study: Numerical Stability

Overview

Let us Define a functions that approximate the value of pi.

```
double unstable_ApproximatePi(double Sn){
 double e = Sn * Sn;
 double tmp = std::sqrt(4 - e);
 double Sn1 = std::sqrt(2 - tmp);
 return Sn1;
}
```

```
int count = 30;
double Sn= sqrt(2);
double pi;
double n = 4;

for (int i = 1; i < count; i++) {
   Sn = ApproximatePi(Sn);
   n = 2*n;
   pi = n * Sn / 2;
}
```

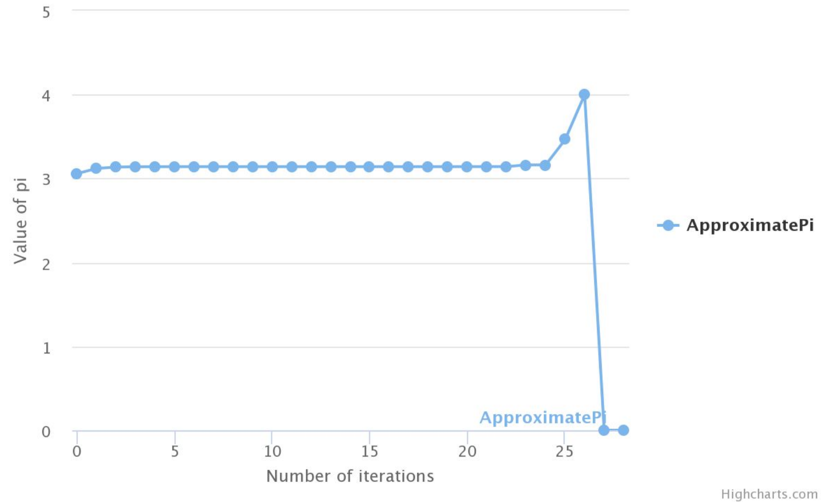
Reference: ADAPT: algorithmic differentiation applied to floating-point precision tuning

Case Study: Numerical Stability

Initial Results

```
double unstable_ApproximatePi(double Sn){
 double e = Sn * Sn;
 double tmp = std::sqrt(4 - e);
 double Sn1 = std::sqrt(2 - tmp);
 return Sn1;
}
```

After executing the code for ~30 iterations, we get the following results:



What is happening at the ~25th iteration?

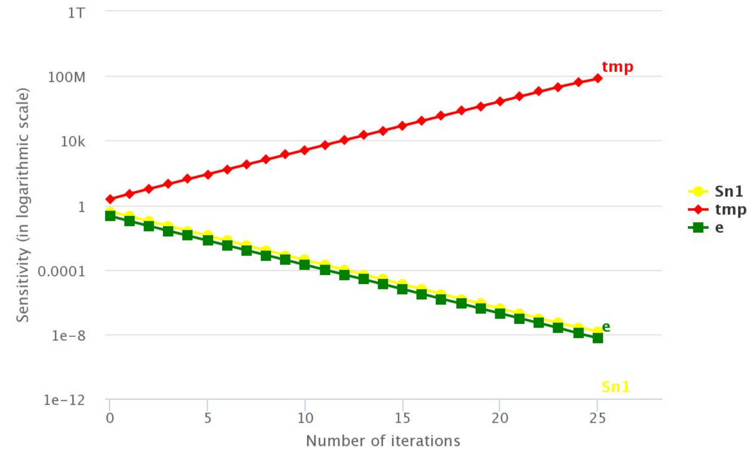
Case Study: Numerical Stability

Sensitivity Analysis

```
double unstable_ApproximatePi(double Sn){
 double e = Sn * Sn;
 double tmp = std::sqrt(4 - e);
 double Sn1 = std::sqrt(2 - tmp);
 return Sn1;
}
```

To figure out what is wrong, we can user clad's error estimation to analyse the sensitivity of each intermediate variable. Which (in our case) is given as follows:

$$S_{x_i} = |\frac{\partial f}{\partial x_i} * x_i|$$



Case Study: Numerical Stability

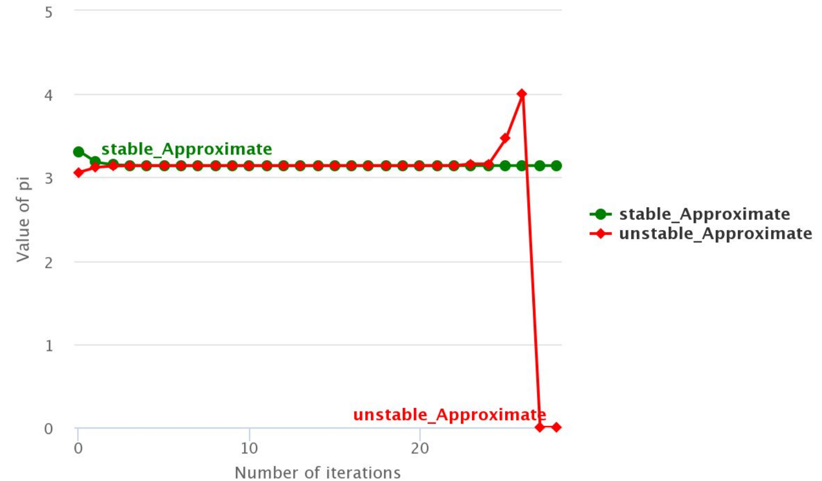
Verifying Results

```
double unstable_ApproximatePi(double Sn){
 double e = Sn * Sn;
 double tmp = std::sqrt(4 - e);
 double Sn1 = std::sqrt(2 - tmp);
 return Sn1;
}
```

```
double stable_ApproximatePi(double Tn) {
double e = Tn * Tn;
double tmp = std::sqrt(4 + e);
double Tn1 = 2 * Tn / (2 + tmp);
return Tn1;
}
```

We will analyse 2 things:

- The comparison of results for 30 iterations
- Sensitivity in all intermediate variables

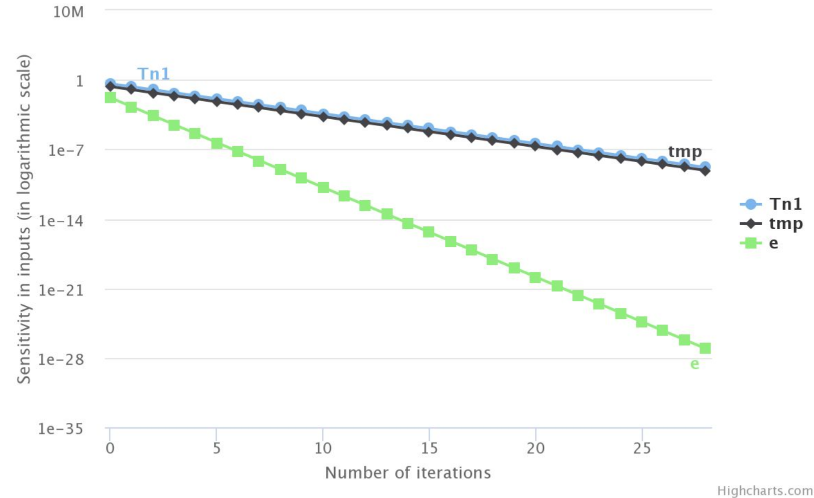


Case Study: Numerical Stability

Verifying Results

```
double unstable_ApproximatePi(double Sn){
 double e = Sn * Sn;
 double tmp = std::sqrt(4 - e);
 double Sn1 = std::sqrt(2 - tmp);
 return Sn1;
}
```

```
double stable_ApproximatePi(double Tn) {
double e = Tn * Tn;
double tmp = std::sqrt(4 + e);
double Tn1 = 2 * Tn / (2 + tmp);
return Tn1;
}
```



Using Custom Models

Clad supports the usage of custom defined models:

1. Implement the `clad::FPErrorEstimationModel` class, a generic interface that provides the error expressions for clad to generate.
2. Override the `AssignError()` function. This function is called for all LHS of every assignment expression in the target function.

The function `AssignError()` essentially represents the mathematical formula of an error model in a form that clang can understand and convert to code. It provides users with the reference to the variable of interest and its derivative. The user in turn has to return an expression which will be used to accumulate the error.

While filling in these functions requires some clang API knowledge, it is possible to bypass this by just creating a function call expression as the error expression and then implement the function as you like!

For more information on how to build a custom model from scratch, check here! And for more information the FP error estimation framework, check here!

Summary

- AD based FP error analysis enables understanding the largest contributions to FP errors and enables mixed-precision program optimization.
- Clad has a novel, customizable, AD-based error estimation framework that automates FP error analysis, available with `conda install clad`.
- We demonstrated a case study for sensitivity analysis of a Simpson's rule program and verified the numerical stability of an approximate pi function.
- We illustrated how to create a custom FP error analysis model capable of incorporating domain-specific knowledge

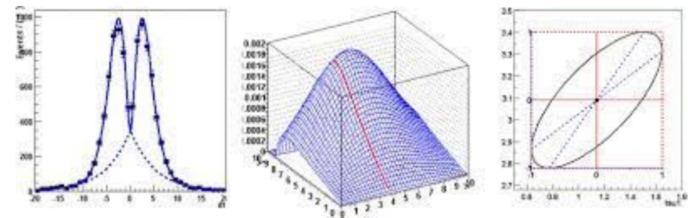
We plan add functionality to use the mixed precision recommendations to automatically generate optimized code which will allow further research in the area of lossy compression.

https://github.com/vgvassilev/clad | Binder - Jupyter Notebook

backup

Motivation: FP errors in High Energy Physics

- High energy physics (HEP), also sees many applications of floating point errors as physicists often have to deal with complex algorithms that may accumulate errors that could be proportional to the large amounts of input data.
- An example - ROOT, a data analysis framework, is used widely by physicists at CERN. It provides RooFit, a toolkit for modeling the expected distribution of events in a physics analysis. RooFit is often used for minimization probability density functions (pdfs) with respect to a set of specific inputs. Here, sometimes floating point errors can cause these minimizations to not converge correctly, and hence, it might be useful to automatically detect and warn users if the floating point errors could be playing a role in the convergence problems.

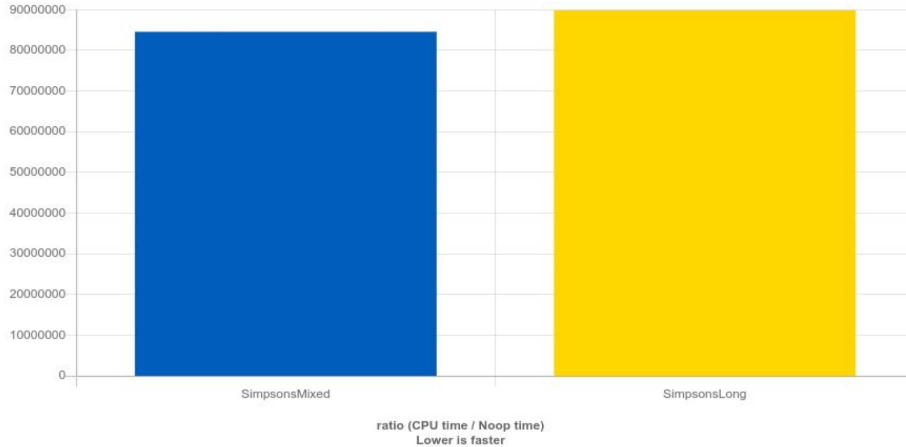


Analysis data from RooFit, src: http://roofit.sourceforge.net/

Alternative: Motivation: FP errors in High Energy Physics

- The event field of High Energy Physics re-processes exabytes of data in its quest for “interesting” physics, organized into independent “events” allowing embarrassingly parallelism
- Each re-processing phase (simulation, reconstruction, analysis) consist of a long pipeline of systems mostly written in C++
- How we can automatize our understanding about FP representation effects and use it to our benefit?
    - How much we can trust our results?
    - Can we process less data while keeping an acceptable accuracy?

Example - Simpsons



Benchmarked with QuickBench:https://quick-bench.com/q/x0WmkQvqdMR3e8BeqqyHhLxvOtw

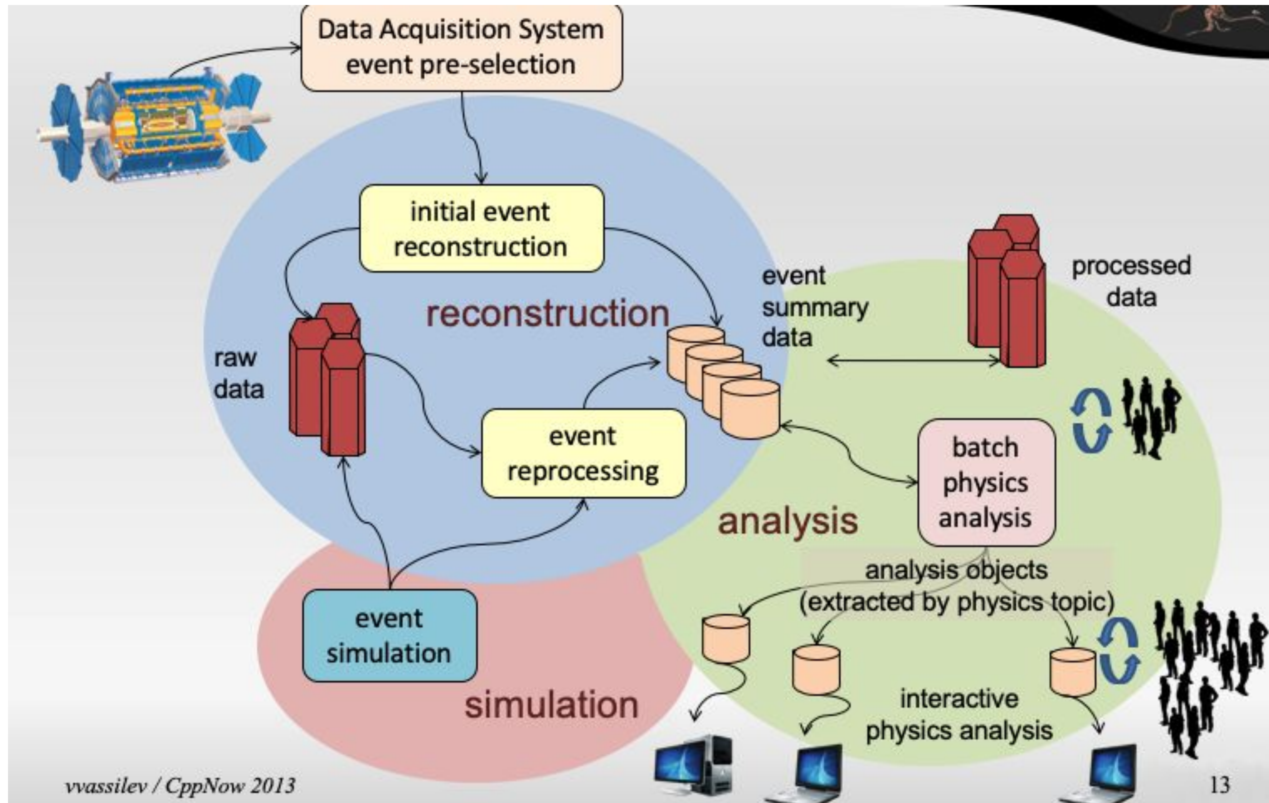
- Clad's mixed precision configuration is upto 10% faster than the 10-byte extended precision configuration.

```
long double f(double x) {
 double pi = M_PI;
 long double tmp = x * pi;
 long double tmp2 = sin(tmp) * pi;
 return tmp2;
}
```

```
long double simpsons(double a, double b) {
int n = 1000000;

double h = (b - a) / (2.0 * n);
long double x = a;
double tmp;
double fa = f(a), fb = f(b);
long double s1 = fa;

for(int l = 0; l < n; l++) {
  x = x + h;
  s1 = s1 + 4.0 * f(x);
  x = x + h;
  s1 = s1 + 2.0 * f(x);
}

s1 = s1 - fb;
tmp = h / 3.0;
s1 = s1 * tmp;
return s1;
}
```

Example adapted from ADAPT: Algorithmic Differentiation Applied to Floating-Point Precision Tuning[3]

Data Flow in the Field of High Energy Physics



Mixed Precision Tuning

- As mentioned in the previous slides, mixed precision tuning can be extremely useful for programs that want to optimize both speed, size and precision.
- As such, “demoting” certain types to lower precision while still maintaining the desired accuracy can be beneficial.
- Let’s talk about a simple case study in the upcoming slides.