Estimating Floating Point Errors, The Automatic Differentiation Way

Garima Singh[1&2], Baidyanath Kundu[1&2], Harshitha Menon[3], Alexander Penev[4], David J. Lange[1], Vassil Vassilev[1&2]

[1] Princeton Univ. (US), [2] CERN, [3] LLNL (US), [4] Univ. of Plovdiv (Bulgaria)

Floating point errors

	Value	Error
Input number:	0.3	-
Representation in float:	0.30000001192092895508	1.19e-08
Representation in double:	0.29999999999999998890	1.11e-17

Let's try a simple addition operation: 0.3 + 0.3

Operation output:	0.6	-
Representation in float:	0.60000002384185791016	2.38e-08
Representation in double:	0.59999999999999997780	2.22e-17

Link to code for these numbers

Floating point errors

	Value	Error
Input number:		-
Representation in float:		1.19e-08
Representation in double:		1.11e-17

Because floating point errors are additive, they can become significant in HPC or other large-scale applications!

Let's try a simple addition operation: 0.3 + 0.3 . . . billion times

Operation output:	3.00e+08	-
Representation in float:	8.39e+06	2.92e+08
Representation in double:	3.00e+08	5.65e+00

[Link to code]() for these numbers

Floating Point(FP) Error Estimation Tools

Automated search-based tools

Examples

CRAFT, Precimonious

Very expensive as the state space to search is large

Can be infeasible even for small benchmarks

Static analysis tools

Examples

SEESAW, FPTaylor

Provides rigorous estimates for FP errors

Limited to programs with a small no. of operations

Tools using Automatic Differentiation

Examples

ADAPT-FP (SoTA), FloatSmith

Works well with smaller HPC benchmarks

Require manual code changes and takes long to setup

CHEF-FP

Can handle both small and large HPC benchmarks

Requires minimal setup

Supports various error analysis through custom error models

Derivation of Error Estimation Formula

Let's assume an arbitrary function $f(x)$, and the floating-point error in x to be h, the **Taylor series** expansion is:

$$f(x+h) = f(x) + \frac{h}{1!} f'(x) + \frac{h^2}{2!} f''(x) + \frac{h^3}{3!} f'''(x) + \cdots$$

Since h is a very small value when compared to x we can assume h^2 to be insignificant and safely **drop higher order** terms:

$$f(x+h) \approx f(x) + h \cdot f'(x)$$

Therefore, the **absolute floating-point error (Δf_x)** in f due to x is:

$$\Delta f_x \approx |f(x+h) - f(x)| = |h \cdot f'(x)|$$

The maximum floating-point error (h_{max}) in x as allowed by IEEE is $|x| \cdot \varepsilon_M$, where ε_M is the machine epsilon. Thus,

$$\Delta f_x \approx |f'(x) \cdot |x| \cdot \varepsilon_M|$$

Classical Formula for Error Estimation

The general representation of the error estimation formula is:

Input and intermediate variables

First order partial derivatives

Machine Epsilon

$$\Delta f \equiv \sum_{i=0}^{n} \left| \frac{\partial f}{\partial x_i} \cdot |x_i| \cdot \varepsilon_M \right|$$

Derivative w.r.t that variable

Error in variable

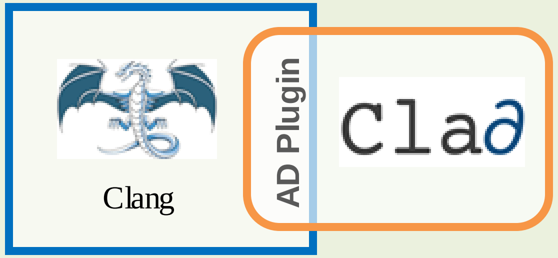
Automatic Differentiation (AD)

AD refers to a set of techniques that are used to calculate the exact derivatives of a given function by using the chain rule of differential calculus.

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

Source Transformation AD

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



CHEF-FP uses Clad, a source transformation AD tool that is developed as a Clang plugin

CHEF-FP Usage

Define the kernel

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

Pass it to CHEF-FP

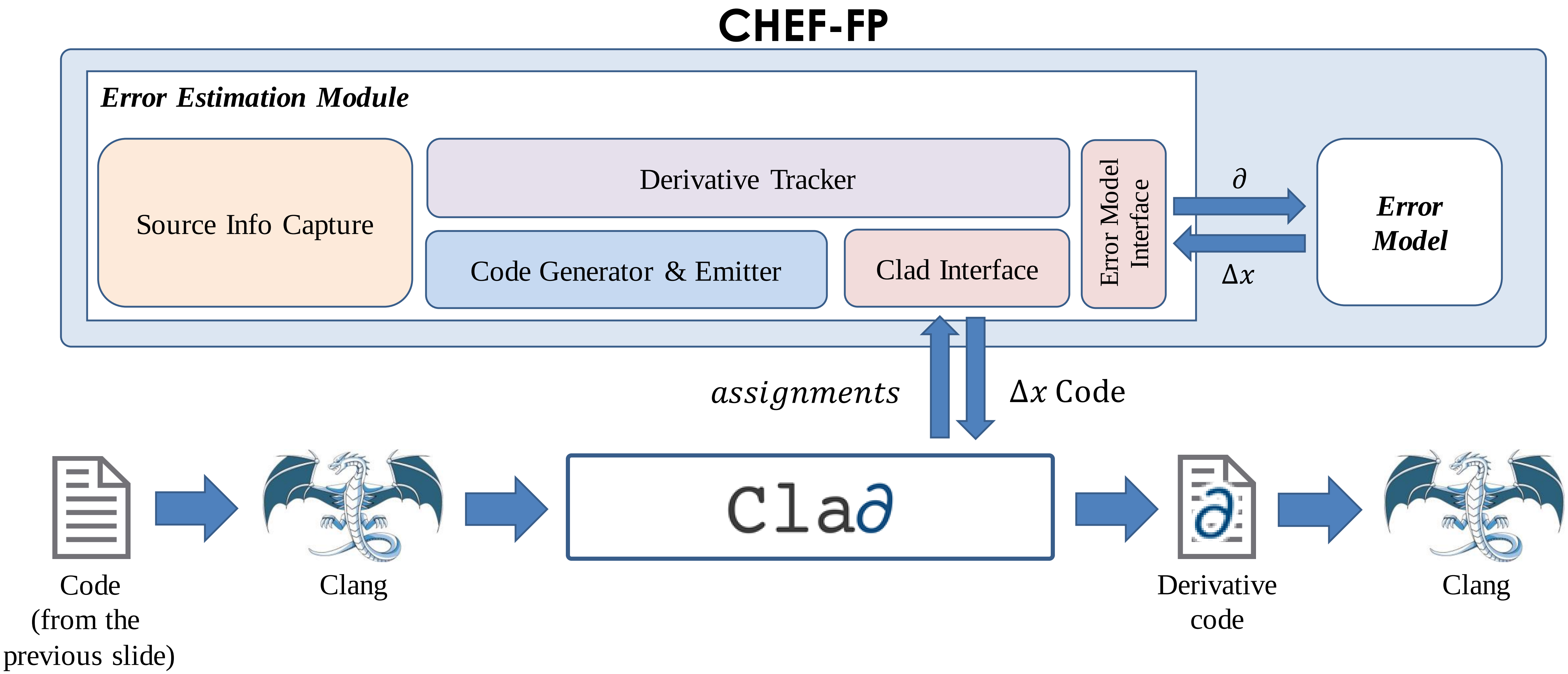
```
#include “clad/Differentiator/Differentiator.h”
#include “../PrintModel/ErrorFunc.h”

// Call CHEF-FP on the function
auto df = clad::estimate_error(func);
```

Let’s see how this works internally when you compile the code!

USER GENERATED CODE

CHEF-FP architecture



CHEF-FP Error Model

```
double getErrorVal(double dx, double x,
                   const char *name) {
  double error = std::abs(dx * (x - (float)x));
  ErrorStorage.store_error(name, error);
  return error;
}
```

Error Model

Error Storage data structure:

Name	*Max*	*Total*	*Count*
var0	2.94e-04	6.77e-01	100000
var1	1.16e-05	9.18e-01	100000

<u>Customizing the error model:</u>

The above code can be modified to carry out different types of analysis. Examples of such analyses will be shown later in the presentation.

CHEF-FP Usage

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

```
#include "clad/Differentiator/Differentiator.h"
#include "../PrintModel/ErrorFunc.h"

// Call CHEF-FP on the function
auto df = clad::estimate_error(func);
```

Execute the CHEF-FP object to get the error

```
double x = 1.95e-5, y = 1.37e-7;
double dx = 0, dy = 0;
double fp_error = 0;

df.execute(x,y, &dx, &dy, fp_error);

std::cout << "FP error in func: " << fp_error;
// FP error in func: 8.25584e-13

// Print mixed precision analysis results
clad::printErrorReport();
```

USER GENERATED CODE

```
void func_grad(double x, double y,
     clad::array_ref<double> _d_x,
     clad::array_ref<double> _d_y,
     double &_final_error) {
  double _d_z = 0, _delta_z = 0, _EERepl_z0;
  double z = x + y;
  _EERepl_z0 = z;
  double func_return = z;
  _d_z += 1;
  * _d_x += _d_z;
  * _d_y += _d_z;
  _delta_z +=
      clad::getErrorVal(_d_z, _EERepl_z0, "z");
  double _delta_x = 0;
  _delta_x +=
      clad::getErrorVal(* _d_x, x, "x");
  double _delta_y = 0;
  _delta_y +=
      clad::getErrorVal(* _d_y, y, "y");
  _final_error +=
      _delta_y + _delta_x + _delta_z;
}
```

The function generated by CHEF-FP to estimate the errors

AUTO GENERATED CODE

Evaluation

How does CHEF-FP fare against the state of the art?

Experiments – Mixed Precision Analysis

Compared against:

ADAPT-FP	Mixed precision tuning tool based on operator-overloading AD

Evaluated on:

5 Benchmarks	
Arc Length	Simpsons
k-Means	HPCCG
Blackscholes	

Systems used:

Princeton Tiger	LLNL Quartz
CPU: 2.4GHz Intel Xeon Gold 6148	CPU: 2.1GHz Intel Xeon E5-2695
RAM: 188 GB	RAM: 128 GB

CHEF-FP vs ADAPT-FP

Performance Improvements vs ADAPT-FP

Benchmark	*Time*	*Memory*
Arc Length	1.61x	1.95x
Simpsons	2.17x	1.44x
k-Means	2.02x	4.44x
HPCCG	1.03x	1.02x
Blackscholes	1.76x	6.32x

Scan the QR code to access the GitHub repository for the benchmarks

Why is CHEF-FP more efficient?

- CHEF-FP inserts the error estimation code into the derivative, so it is calculated in the same step while ADAPT-FP calculates them in two different steps.
- CHEF-FP uses Clad, a source-code transformation AD tool, whereas ADAPT-FP uses CodiPack which is an operator overloading AD.
- The code produced by CHEF-FP is optimized by the compiler to gain much more performance.

Beyond FP Error Estimation

Demonstrating the *flexibility* of CHEF-FP through **custom error models**

Sensitivity Analysis

Representation of the original HPCCG code:

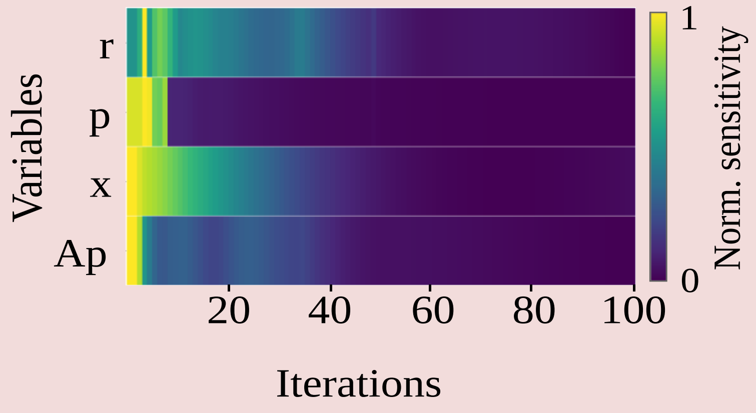
```
for (int k = 1; k <= 100; k++) {
  // HPCCG loop code using doubles
}
```

CHEF-FP provides customizable error models which can be modified to dump the sensitivities of all variables into their respective files.

Math equation to compute the sensitivity:

$$S_x = \left| \frac{\partial f}{\partial x} \cdot |x| \right|$$

Sensitivity Analysis



The sensitivities of **r**, **p**, **x**, **Ap** become negligible around the 50th iteration!

```
for (int k = 1; k <= 100; k++) {
  // HPCCG loop code using doubles
}
```

Loop perforation-based optimization

```
for (k = 1; k <= 50; k++) {
  // HPCCG loop code with all vars
  // in double
}
for (; k <= 100; k++) {
  // HPCCG loop code with r,p,x,Ap
  // in float and rest in doubles
}
```

8% Speedup

Cost Analysis of Approximation

Goal: Estimate the error introduced into Black-Scholes algorithm by replacing standard math functions with their respective approximate versions

Solution: This can be easily achieved by modifying the CHEF-FP's error model to compute the approximation error.

Math equation to compute the approximation error:

$$\Delta f_{\mathrm{mf}} = \left| \frac{\partial f}{\partial\, \mathrm{mf}(x)} \left| \mathrm{mf}(x) - \mathrm{mf}_{approx}(x) \right| \right|$$

where mf is a simple math library function such as log, sqrt or exp and mf_{approx} is the approximate version of it.

Error Model Algorithm : Cost Analysis of Approximation

Algorithm for estimating approximation-based errors:

Require: input variable as x and its name as $name$, the partial derivative of x w.r.t. the function as dx, and a map of variables of interest as $S : name \rightarrow function\ name$

```
Δ ← 0
if name is contained in S then
    fName ← S.GETVALUE(name)
    Δ ← EVAL(fName, x) − EVALAPPROX(fName, x)
    Δ ← Δ ÷ (∂EVAL(fName, x)/∂x)
end if
xApproxError ← |dx × Δ|
REGISTERERROR(name, xApproxError)
return xApproxError
```

Math equation to compute the approximation error:

$$\Delta f_{\mathrm{mf}} = \left| \frac{\partial f}{\partial\, \mathrm{mf}(x)} \left| \mathrm{mf}(x) - \mathrm{mf}_{approx}(x) \right| \right|$$

where mf is a simple math library function such as log, sqrt or exp and mf_{approx} is the approximate version of it.

We used the approximate math functions in the FastApprox library for this analysis

P. Mineiro, "Fastapprox," [Google Archive Link], 2011

Cost Analysis of Approximation

Analysis of errors due to approximation:

App Configuration	*Estimated Error*	*Speedup*	*Actual Error*
Using FastApprox log & pow	1.16e+01	1.14	1.16e+01
Using FastApprox log, pow & exp	1.18e+02	1.65	5.88e+01

Conclusion

- AD can be used to create an effective, scalable, and easy-to-use error estimation tool.
- Source transformation AD is an ideal candidate for such a tool because the error estimation code can be inlined into generated derivatives thus benefitting from compiler optimizations and reduced memory usage.

COMPILER
C | R
RESEARCH

compiler-research.org

Scan the QR code to get started with CHEF-FP

Thank You

To appear in the proceedings of IPDPS'23

Title: **Fast And Automatic Floating Point Error Analysis With CHEF-FP**

https://arxiv.org/abs/2304.06441

Benchmarks Repo

compiler-research.org

CHEF-FP Tutorial

Backup

Cost Analysis of Approximation: Error Model Code

```
double getErrorVal(double dx, double x, const char* cname) {
  char name[50];
  int i = 0;
  while (cname[i++] != '\0’)
    name[i - 1] = cname[i - 1];
  char *token = strtok(name, "_");
  if (strcmp(token, "clad"))
    return 0;
  token = strtok(NULL, "_");
  double error;
  if (!strcmp(token, "exp"))
    error = std::fabs(dx * (exp(x) - fastexp(x)) / exp(x));
  else if (!strcmp(token, "log"))
    error = std::fabs(dx * (log(x) - fastlog(x)) * exp(x));
  else if (!strcmp(token, "sqr"))
    error = std::fabs(dx * (sqrt(x) - fastpow(x, 0.5)) * 2 * sqrt(x));
  else return 0;
  ErrorStorage::getInstance().set_error(cname, error);
  return error;
}
```

$$\Delta f_{\text{mf}} = \left| \frac{\partial f}{\partial\, \text{mf}(x)} \left| \text{mf}(x) - \text{mf}_{approx}(x) \right| \right|$$

where mf is a simple math library function such as log, sqrt or exp and mf_{approx} is the approximate version of it.

Error Model Code:
FP Error Analysis

The code:

```
double getErrorVal(double dx, double x, const char *name) {
  double e_M = std::numeric_limits<T>::epsilon();
  double error = std::abs(dx * std::abs(x) * e_M);
  return error;
}
```