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$^1$, Vasil Vassilev$^{1&2}$

$^1$Princeton Univ. (US), $^2$CERN, $^3$LLNL (US), $^4$Univ. of Plovdiv (Bulgaria)
## Floating point errors

<table>
<thead>
<tr>
<th>Input number:</th>
<th>Value</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Representation in float:</td>
<td>0.30000001192092895508</td>
<td>1.19e-08</td>
</tr>
<tr>
<td>Representation in double:</td>
<td>0.29999999999999998890</td>
<td>1.11e-17</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Operation output:</th>
<th>Value</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Representation in float:</td>
<td>0.60000002384185791016</td>
<td>2.38e-08</td>
</tr>
<tr>
<td>Representation in double:</td>
<td>0.59999999999999997780</td>
<td>2.22e-17</td>
</tr>
</tbody>
</table>

[Link to code](#) for these numbers
## Floating point errors

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

<table>
<thead>
<tr>
<th>Input number:</th>
<th>Representation in float:</th>
<th>Representation in double:</th>
<th>Value</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.39e+06</td>
<td>8.39e+06</td>
<td>3.00e+08</td>
<td>-</td>
<td>2.92e+08</td>
</tr>
<tr>
<td>3.00e+08</td>
<td>3.00e+08</td>
<td>3.00e+08</td>
<td>-</td>
<td>5.65e+00</td>
</tr>
</tbody>
</table>

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

[Link to code for these numbers]
# Floating Point (FP) Error Estimation Tools

<table>
<thead>
<tr>
<th>Automated search-based tools</th>
<th>Static analysis tools</th>
<th>Tools using Automatic Differentiation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Examples</strong></td>
<td><strong>Examples</strong></td>
<td><strong>Examples</strong></td>
</tr>
<tr>
<td>CRAFT, Precimonious</td>
<td>SEESAW, FPTaylor</td>
<td>ADAPT-FP (SoTA), FloatSmith</td>
</tr>
</tbody>
</table>

- **Very expensive as the state space to search is large**
- **Can be infeasible even for small benchmarks**
- **Provides rigorous estimates for FP errors**
- **Limited to programs with a small no. of operations**
- **Works well with smaller HPC benchmarks**
- **Require manual code changes and takes long to setup**
- **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 ($\Delta f_x$) 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 $\varepsilon_M$ is the machine epsilon. Thus,

$$\Delta f_x \approx |f'(x) \cdot |x| \cdot \varepsilon_M|$$
The general representation of the error estimation formula is:

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

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

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.
Define the kernel

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

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

Pass it to CHEF-FP

Let's see how this works internally when you compile the code!
CHEF-FP architecture

**Error Estimation Module**
- Source Info Capture
- Derivative Tracker
- Code Generator & Emitter
- Clad Interface

**Clad Interface**

**Error Model**

**Clang**

Code (from the previous slide)

**Clang**

Derivative code

**Derivative Tracker**

**Error Model Interface**

\[ \frac{\partial}{\partial x} \]

\[ \Delta x \text{ Code} \]

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

Customizing the error model:
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);

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();

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;
}
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:

<table>
<thead>
<tr>
<th>5 Benchmarks</th>
<th>Arc Length</th>
<th>Simpsons</th>
<th>k-Means</th>
<th>HPCCG</th>
<th>blackscholes</th>
</tr>
</thead>
</table>

Systems used:

**Princeton Tiger**
- **CPU**: 2.4GHz Intel Xeon Gold 6148
- **RAM**: 188 GB

**LLNL Quartz**
- **CPU**: 2.1GHz Intel Xeon E5-2695
- **RAM**: 128 GB
Performance Improvements vs ADAPT-FP

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arc Length</td>
<td>1.61x</td>
<td>1.95x</td>
</tr>
<tr>
<td>Simpsons</td>
<td>2.17x</td>
<td>1.44x</td>
</tr>
<tr>
<td>k-Means</td>
<td>2.02x</td>
<td>4.44x</td>
</tr>
<tr>
<td>HPCCG</td>
<td>1.03x</td>
<td>1.02x</td>
</tr>
<tr>
<td>Blackscholes</td>
<td>1.76x</td>
<td>6.32x</td>
</tr>
</tbody>
</table>

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.

Scan the QR code to access the GitHub repository for the benchmarks.
Beyond FP Error Estimation

Demonstrating the flexibility of CHEF-FP through custom error models
Sensitivity Analysis

Representation of the original HPCCG code:

```c
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!

Loop perforation-based optimization

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

```cpp
for (k = 1; k <= 50; k++) {
    // HPCCG loop code with all vars
    // in double
}
```

```cpp
for (; k <= 100; k++) {
    // HPCCG loop code with $r$, $p$, $x$, $Ap$
    // in float and rest in doubles
}
```
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:

\[ \Delta f_{mf} = \left| \frac{\partial f}{\partial mf(x)} |mf(x) - mf_{approx}(x)| \right| \]

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 $S: name \rightarrow function\ name$

1: $\Delta \leftarrow 0$
2: if $name$ is contained in $S$ then
3: $fName \leftarrow S.GETVALUE(name)$
4: $\Delta \leftarrow EVAL(fName, x) - EVALAPPROX(fName, x)$
5: $\Delta \leftarrow \Delta \div (\partial EVAL(fName, x)/\partial x)$
6: end if
7: $xApproxError \leftarrow |dx \times \Delta|$
8: REGISTERERROR($name$, $xApproxError$)
9: return $xApproxError$

Math equation to compute the approximation error:

$$\Delta_{mf} = \left| \frac{\partial f}{\partial mf(x)} \right| \left| mf(x) - mf_{approx}(x) \right|$$

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:

<table>
<thead>
<tr>
<th>App Configuration</th>
<th>Estimated Error</th>
<th>Speedup</th>
<th>Actual Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using FastApprox log &amp; pow</td>
<td>1.16e+01</td>
<td>1.14</td>
<td>1.16e+01</td>
</tr>
<tr>
<td>Using FastApprox log, pow &amp; exp</td>
<td>1.18e+02</td>
<td>1.65</td>
<td>5.88e+01</td>
</tr>
</tbody>
</table>
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.

Scan the QR code to get started with CHEF-FP

compiler-research.org
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;
}

\[ \Delta f_{mf} = \left| \frac{\partial f}{\partial mf(x)} \right| mf(x) - mf_{approx}(x) \]

where \( mf \) is a simple math library function such as \( \log \), \( \sqrt{\text{r}} \) or \( \exp \) and \( mf_{approx} \) is the approximate version of it.
The code:

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