Accelerating Large Scientific Workflows Using Source Transformation Automatic Differentiation

Vassil Vassilev, Princeton
compiler-research.org

This work is partially supported by National Science Foundation under Grant OAC- 2311471, OAC- 1931408 and NSF (USA) Cooperative Agreement OAC-1836650
Motivation

Provide automatic differentiation for C/C++ that works without little code modification (including legacy code)
AD Basics

- $Y = \text{CONSTANT}$
- $Y = X^2$
- $Y = X^3$
- $Y = \cos X$
AD. Chain Rule

\[
\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}
\]

Intuitively, the chain rule states that knowing the instantaneous rate of change of \(z\) relative to \(y\) and that of \(y\) relative to \(x\) allows one to calculate the instantaneous rate of change of \(z\) relative to \(x\) as the product of the two rates of change.

“if a car travels twice as fast as a bicycle and the bicycle is four times as fast as a walking man, then the car travels \(2 \times 4 = 8\) times as fast as the man.” G. Simmons
AD. Algorithm Decomposition

\[ y = f(x) \]
\[ z = g(y) \]

\[ \frac{dy}{dx} = \frac{df}{dx}(x) \]
\[ \frac{dz}{dy} = \frac{dg}{dy}(y) \]
\[ \frac{dz}{dx} = \frac{dz}{dy} \times \frac{dy}{dx} \]

In the computational graph each node is a variable and each edge is derivatives between adjacent edges.

We recursively apply the rules until we encounter an elementary function such as addition, substraction, multiplication, division, sin, cos or exp.
AD. Chain Rule

\[ y = f(x_0, x_1) \]
\[ z = g(y) \]
\[ w_0, w_1 = l(z) \]

\[
\frac{\partial w_0}{\partial x_0} = \frac{\partial w_0}{\partial z} \frac{\partial z}{\partial y} \frac{\partial y}{\partial x_0}
\]
\[
\frac{\partial w_1}{\partial x_0} = \frac{\partial w_1}{\partial z} \frac{\partial z}{\partial y} \frac{\partial y}{\partial x_0}
\]
\[
\frac{\partial w_1}{\partial x_1} = \frac{\partial w_1}{\partial z} \frac{\partial z}{\partial y} \frac{\partial y}{\partial x_1}
\]

16-Aug-2023
V. Vassilev – Accelerating Large Scientific Workflows Using Source Transformation Automatic Differentiation
AD step-by-step. Forward Mode

dx0dx = \{1, 0\}
dx1dx = \{0, 1\}
y = f(x_0, x_1)
dydx = df(x_0, dx0dx, x_1, dx1dx)
z = g(y)
dzdx = dg(y, dydx)
w0, w1 = l(z)
dw0dx, dw1dx = dl(z, dzdx)
AD step-by-step. Reverse Mode

\[ y = f(x_0, x_1) \]
\[ z = g(y) \]
\[ w_0, w_1 = l(z) \]

\[ \text{dwdw}_0 = \{1, 0\} \]
\[ \text{dwdw}_1 = \{0, 1\} \]

\[ \text{dwdz} = d_l(\text{dwdw}_0, \text{dwdw}_1) \]

\[ \text{dwdy} = dg(y, \text{dwdz}) \]

\[ \text{dwx}_0, \text{dwx}_1 = df(x_0, x_1, \text{dwdy}) \]
AD. Cheap Gradient Principle

• The computational graph has common subpaths which can be precomputed.
• If a function has a single input parameter, no matter how many output parameters, forward mode AD generates a derivative that has the same time complexity as the original function.
• More importantly, if a function has a single output parameter, no matter how many input parameters, reverse mode AD generates derivative with the same time complexity as the original function.
AD tools can be categorized by how much work is done before program execution

• *Tracing/Operator Overloading/Dynamic Graphs/Taping* -- Records the linear sequence of computation operations at runtime into a tape

• *Source Transformation* -- Constructs the computation graph and produces a derivative at compile time
Automatic vs Symbolic Differentiation

\[ f(x) = e^{e^{e^{e^x}}} \]

\[ \frac{d}{dx} \left( e^{e^{e^{e^x}}} \right) = e^x + e^{e^x} + e^{e^x} + e^x \]

Figure out the analytical fn

Handcode

Symbolic via Wolfram Alpha

// f(x)=e^(e^(e^(e^(e^x))))
#include <cmath>
double f (double x, int N=5) {
    double result = x;
    for (unsigned i = 0; i < N; i++)
        result = std::exp(result);
    return result;
}

double f_dx(double x, int N=5) {
    double result = x;
    double d_result = 1;
    for (unsigned i = 0; i < N; i++)
        result = std::exp(result);
    d_result *= result;
    return d_result;
}
AD. Gradient Generation

• Control Flow and Recursion fall naturally in forward mode.

• Extra work is required for reverse mode in reverting the loop and storing the intermediaries in general.

```cpp
double f_reverse (double x, int N=5) {
    double result = x;
    std::stack<double> results;
    for (unsigned i = 0; i < N; i++) {
        results.push(result);
        result = std::exp(result);
    }
    double d_result = 1;
    for (unsigned i = N; i; i--) {
        d_result *= std::exp(results.top());
        results.pop();
    }
    return d_result;
}
```
Clad
Clad. Design Principles

• Look Ma’ I can make a compiler generate a derivative!
• Make AD a first-class citizen to a high-performance language such as C++
• Support idiomatic C++ (compile-time programming such as constexpr, consteval)
• Infrastructure reuse – employ our compiler engineering skills
• Lower contribution entry barrier
• Diagnostics
High-Level Data Flow

- Compiler module, very similar to the template instantiator by idea and design.
- Generates $f'$ of any given $f$ using source transformation at compile time.
Programming Model

// clang++ -fplugin libclad.so -Iclad/include ...

// Necessary for clad to work include
#include "clad/Differentiator/Differentiator.h"
double pow2(double x) { return x * x; }
double pow2_darg0(double);

int main() {
    auto dfdx = clad::differentiate(pow2, 0);
    // Function execution can happen in 3 ways:
    // 1) Using CladFunction::execute method.
    double res = cladPow2.execute(1);
    // 2) Using the function pointer.
    auto dfdxFnPtr = cladPow2.getFunctionPtr();
    res = cladPow2FnPtr(2);
    // 3) Using direct function access through fwd declaration.
    res = pow2_darg0(3);
    return 0;
}
User-defined substitution

Programming Model. Differential Operators

```
// MyCode.h
float custom_fn(float x);

namespace custom_derivatives {
  float custom_fn_dx(float x) {
    return x * x;
  }
}

float do_smth(float x) {
  return x * x + custom_fn(x);
}

int main() {
  clad::differentiate(do_smth, 0).execute(2); // will return 6
  return 0;
}
```

template <typename T1, typename T2>
CUDA_HOST_DEVICE ValueAndPushforward<decltype::std::pow(T1(), T2())),
  decltype::std::pow(T1(), T2()))>
pow_pushforward(T1 x, T2 exponent, T1 d_x, T2 d_exponent) {
  auto val = ::std::pow(x, exponent);
  auto derivative = (exponent * ::std::pow(x, exponent - 1)) * d_x;
  // Only add directional derivative of base^exp w.r.t exp if the directional
  // seed d_exponent is non-zero. This is required because if base is less than or
  // equal to 0, then log(base) is undefined, and therefore if user only requested
  // directional derivative of base^exp w.r.t base -- which is valid --, the result would
  // be undefined because as per C++ valid number + NaN * 0 = NaN.
  if (d_exponent)
    derivative += (::std::pow(x, exponent) * ::std::log(x)) * d_exponent;
  return {val, derivative};
}
Clad in High-Energy Physics

- We have seen some promising results (in ROOT) already!

**Performance Comparison of Generation in TFormula**

- Gaussian: 13.7
- Exponent: 16.2
- breitwigner: 9.88
- Chebyshev deg.0: 6.05
- Chebyshev deg.1: 4.57
- Chebyshev deg.2: 3.95

<table>
<thead>
<tr>
<th>Time (ns)</th>
<th>Numerical Differentiation</th>
<th>Clad AD</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>303</td>
<td>14.4 x</td>
</tr>
<tr>
<td>50</td>
<td>271</td>
<td>9.9 x</td>
</tr>
<tr>
<td>100</td>
<td>235</td>
<td>22.9 x</td>
</tr>
<tr>
<td>150</td>
<td>200</td>
<td>12.4 x</td>
</tr>
<tr>
<td>200</td>
<td>179</td>
<td>29.4 x</td>
</tr>
<tr>
<td>250</td>
<td>156</td>
<td>26.4 x</td>
</tr>
</tbody>
</table>

**Performance Speedup of a Multi-Gaussian Fit (10000 bins)**

- Speedup of 60x!

**TFormula benchmarks of gradient generation time from numerical differentiation and clad AD.**

**TF1 based benchmarks. TF1 is the TFormula fitting interface for fitting histograms.**

A data analysis framework used to process EB data
There and Back Again

Social Engineering, Progress, Social Engineering...

In the meanwhile: Cling, ROOT6, C++ Modules, IPCC-ROOT, compiler-research.org, Clang-Repl ...

Derivatives in C++ in HEP

- Relevant for building gradients used in fitting and minimization.
- Minimization of likelihood function with ~1000 parameters
Statistical Modelling
Automatic Differentiation in RooFit

RooFit represents all mathematical formulae as RooFit objects which are then brought together into a compute graph. This compute graph makes up a model on which further data analysis is run.

Programmers/users know this relationship. But how do we connect these two together when a connection is not obvious in code?

<table>
<thead>
<tr>
<th>Math Notations</th>
<th>RooFit Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable</td>
<td>x</td>
</tr>
<tr>
<td>function</td>
<td>f(x)</td>
</tr>
<tr>
<td>PDF</td>
<td>f(x)</td>
</tr>
<tr>
<td>space point</td>
<td>( \hat{x} )</td>
</tr>
<tr>
<td>integral</td>
<td>( \int_a^b f(x) )</td>
</tr>
<tr>
<td>list of space points</td>
<td>( \hat{x}_1, \hat{x}_1, \ldots )</td>
</tr>
</tbody>
</table>

\[ f(x) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{1}{2} \left( \frac{x - \mu}{\sigma} \right)^2} \]

\( \text{Gaussian Probability Distribution Function (pdf)} \)

//Obj represents \( f(x) \) here
RooGaussian \( \text{obj}(x, \mu, \sigma) \);
Equivalent Code in C++ with RooFit
Bottlenecks

- One goal - Make RooFit Faster. Results from a Higgs-combination fit:

Derivatives become bottleneck!

- Good results, but still use numerical differentiation.
- Potential next step – use Automatic Differentiation to compute the gradients.

Automatic Differentiation in RooFit

What that we want to differentiate

Some way to expose differentiable properties of the graph as code.

C++ code the AD tool can understand

C++ code the AD tool can understand

The AD tool

Derivative code of the model!
Automatic Differentiation in RooFit. Approach

What that we want to differentiate

C++ code the AD tool can understand

Define 2 Functions in RooFit

The “glue” function enabling graph squashing.

Stateless function enabling differentiation of each class.

double ADDetail::gauss(double x, double mean, double sigma) {
    const double arg = x - mean;
    const double sig = sigma;
    return std::exp(-0.5 * arg * arg / (sig * sig));
}

void RooGaussian::translate(...) override {
    result = "ADDetail::gauss" +
             _x->getResult() +
             " ," + _mu->getResult() +
             " ," + _sigma->getResult() + "\n";
}
Automatic Differentiation in RooFit. Approach

What we want to differentiate

C++ code the AD tool can understand

Define 2 Functions in RooFit

\texttt{RooGaussian::evaluate()}

The RooFit call to evaluate a gaussian

- Bookkeeping

& caching

\texttt{ADDetail::gauss(x, mu, sig)}

The equivalent code generated

\texttt{ADDetail::gauss(x, mu, sig) / ADDetail::gaussIntegral(...)}

The equivalent code generated
(given the class supports analytical integrals)

V. Vassilev – Accelerating Large Scientific Workflows Using Source Transformation Automatic Differentiation
Automatic Differentiation in RooFit: Approach

- What we want to differentiate
- ‘Squash’ the graph into code
- C++ code the AD tool can understand

C++ code the AD tool can understand

The AD tool

Derivative code of the model!
Basic RooFit Example With Binned Fit of Analytical Shapes

Tested on ROOT master as of May 2023.
*Excludes the seed generation time

V. Vassilev – Accelerating Large Scientific Workflows Using Source Transformation Automatic Differentiation

16-Aug-2023
# Large Analysis Benchmark Describing Workflows in HEP

## Fitting Time (s)*

<table>
<thead>
<tr>
<th>N Channels</th>
<th>RooFit ND</th>
<th>RooFit AD</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.03</td>
<td>0.01</td>
<td>2x</td>
</tr>
<tr>
<td>5</td>
<td>1.19</td>
<td>0.26</td>
<td>2.5x</td>
</tr>
<tr>
<td>10</td>
<td>2.22</td>
<td>0.36</td>
<td>5.2x</td>
</tr>
<tr>
<td>20</td>
<td>7.38</td>
<td>1.17</td>
<td>5.3x</td>
</tr>
</tbody>
</table>

*Excludes the seed generation time, more info

Large Analysis Benchmark Compile Times

<table>
<thead>
<tr>
<th>Mode</th>
<th>JIT</th>
<th>gcc10</th>
<th>clang-13</th>
</tr>
</thead>
<tbody>
<tr>
<td>-O0</td>
<td>16s</td>
<td>1.15s</td>
<td>0.82</td>
</tr>
<tr>
<td>-O1</td>
<td>17s</td>
<td>4.46s</td>
<td>6.00s</td>
</tr>
<tr>
<td>-O2</td>
<td>17s</td>
<td>9.24s</td>
<td>8.57s</td>
</tr>
<tr>
<td>-O3</td>
<td>17s</td>
<td>10.69s</td>
<td>8.88s</td>
</tr>
</tbody>
</table>

The generated code is suboptimal for the optimization pipelines. We know how to fix this.
Floating Point Error Analysis
# Floating point errors

<table>
<thead>
<tr>
<th>Input number:</th>
<th>0.3</th>
<th>Error</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>Representation in float:</td>
<td>0.30000001192092895508</td>
<td></td>
<td>1.19e-08</td>
</tr>
<tr>
<td>Representation in double:</td>
<td>0.29999999999999998890</td>
<td></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>0.6</th>
<th>Error</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>Representation in float:</td>
<td>0.60000002384185791016</td>
<td></td>
<td>2.38e-08</td>
</tr>
<tr>
<td>Representation in double:</td>
<td>0.59999999999999997780</td>
<td></td>
<td>2.22e-17</td>
</tr>
</tbody>
</table>

[Link to code](#) for these numbers

---

16-Aug-2023

V. Vassilev – Accelerating Large Scientific Workflows Using Source Transformation Automatic Differentiation
Classical Formula for Error Estimation

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

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

The general representation of the error estimation formula is:

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

Clad in FP Error Analysis: CHEF-FP

CHEF-FP

Error Estimation Module

Source Info Capture

Derivative Tracker

Code Generator & Emitter

Clad Interface

Error Model Interface

Error Model

δ

Δx

assignments

Δx Code

Code

Clang

Clad

Derivative code

Clang

16-Aug-2023
V. Vassilev – Accelerating Large Scientific Workflows Using Source Transformation Automatic Differentiation
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);

doate x = 1.95e-5, y = 1.37e-7;
doate dx = 0, dy = 0;
doate 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;
}
Plans

- Grey box AD
  - Enhance the pushforward/pullback mechanisms to avoid common AD pitfalls
- Further advancements and applications on floating point error estimation
  - Controlling the error limits helps the energy efficiency of algorithms
- Robust activity analysis
- A research platform AD in C/C++
  - Combines all power of Clang Static Analyzer, LLVM Optimization Passes, Control Flow Graphs
Thank you!

Violeta Ilieva
Initial prototype, Forward Mode

Vassil Vassilev
Conception, Mentoring, Bugs, Integration, Infrastructure

Martin Vassilev
Forward Mode,CodeGen

Alexander Penev
Conception, CMake, Demos, Jupyter

Aleksandr Efremov
Reverse Mode

Jack Qui
Hessians

Roman Shakhov
Jacobians

Oksana Shadura
Infrastructure, Co-mentoring

Pratyush Das
Infrastructure

Garima Singh
FP error estimation, RooFit, Bugs

Ioana Ifrim
CUDA AD

Parth Arora
Initial support classes, functors, pullbacks

Baidyanath Kundu
Array Support, ROOT integration

Vaibhav Thakkar
Forward Vector Mode