Efficient C++ Derivatives Through Source Transformation AD With Clad

Vassil Vassilev, Princeton
compiler-research.org

This work is partially supported by National Science Foundation under Grant OAC-1931408
Motivation

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

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

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

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

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

\[ \frac{\partial w_0}{\partial x_0} = \frac{\partial w_0}{\partial z} \frac{\partial z}{\partial y} \frac{\partial y}{\partial x_0} \]
\[ \frac{\partial w_0}{\partial x_1} = \frac{\partial w_0}{\partial z} \frac{\partial z}{\partial y} \frac{\partial y}{\partial x_1} \]
\[ \frac{\partial w_1}{\partial x_0} = \frac{\partial w_1}{\partial z} \frac{\partial z}{\partial y} \frac{\partial y}{\partial x_0} \]
\[ \frac{\partial w_1}{\partial x_1} = \frac{\partial w_1}{\partial z} \frac{\partial z}{\partial y} \frac{\partial y}{\partial x_1} \]
AD step-by-step. Forward Mode

\[
\begin{align*}
dx0dx &= \{1, 0\} \\
dx1dx &= \{0, 1\} \\
y &= f(x0, x1) \\
dydx &= df(x0, dx0dx, x1, dx1dx) \\
z &= g(y) \\
dzdx &= dg(y, dydx) \\
w0, w1 &= l(z) \\
dw0dx, dw1dx &= dl(z, dzdx)
\end{align*}
\]
AD step-by-step. Reverse Mode

\[ y = f(x_0, x_1) \]
\[ z = g(y) \]
\[ w_0, w_1 = l(z) \]
\[ dwdw_0 = \{1, 0\} \]
\[ dwdw_1 = \{0, 1\} \]
\[ dwdz = dl(dwdw_0, dwdw_1) \]
\[ dwdy = dg(y, dwdz) \]
\[ dwx_0, dwx_1 = df(x_0, x_1, dwdy) \]
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. Implementation Approaches

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

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

Symbolic via Wolfram Alpha

\[ \frac{d}{dx} \left( e^{e^{e^{e^x}}} \right) = e^x + e^{e^x} + e^{e^x} + e^x + 1 \]

Double-precision AD

```cpp
#include <cmath>

double f(double x) {
    double result = x;
    for (unsigned i = 0; i < 5; i++)
        result = std::exp(result);
    return result;
}

double f_dx(double x) {
    double result = x;
    double d_result = 1;
    for (unsigned i = 0; i < 5; i++)
        d_result *= std::exp(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.

```c++
double f_reverse (double x) {
    double result = x;
    std::stack<double> results;
    for (unsigned i = 0; i < 5; i++) {
        results.push(result);
        result = std::exp(result);
    }
    double d_result = 1;
    for (unsigned i = 5; i; i--) {
        d_result *= std::exp(results.top());
        results.pop();
    }
    return d_result;
}
```
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;
}
Programming Model. Differential Operators

User-defined substitution

```cpp
// 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;
}
```

```cpp
template <typename T1, typename T2>
CUDA_HOST_DEVICE ValueAndPushforward<decltype(decltype::std::pow(T1(), T2()))>,
  decltype(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

Clad is available in ROOT:

```cpp
TF1* h1 = new TF1("f1", "formula");
TFormula* f1 = h1->GetFormula();
f1->GenerateGradientPar(); // clad

// clad
f1->GradientPar(x, result);
// numerical
h1->GradientPar(x, result);
```

gaus: Npar = 3, expo: Npar = 2, crystalball: Npar = 5, breitwigner: Npar = 5, cheb2: Npar = 4

25-July-2023
V. Vassilev -- Efficient C++ Derivatives Through Source Transformation AD With Clad – The 3rd MODE Workshop
Clad in FP Error Analysis: CHEF-FP

25-July-2023

V. Vassilev -- Efficient C++ Derivatives Through Source Transformation AD With Clad – The 3rd MODE Workshop
There and Back Again

Social Engineering, Progress, Social Engineering...

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

• 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!