Clad, compile-time automatic differentiation for C++

Vaibhav Thakkar
(Princeton University)

Brief Intro of Automatic Differentiation

$$f(x) = e^{e^{e^{e^{e^x}}}}$$

Symbolic via Wolfram Alpha

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

Figure out the analytical fn

Handcode

```cpp
// 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;
}
```

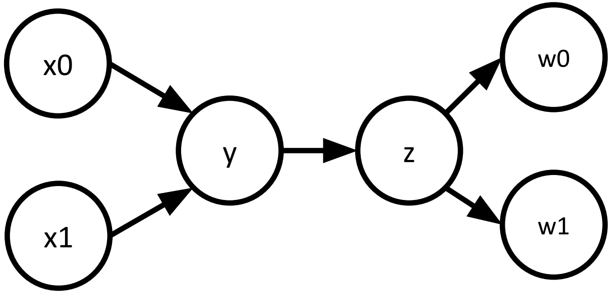
AD

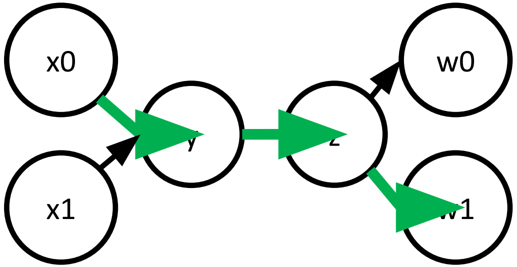
```cpp
double f_dx(double x, int N=5) {
  double result = x;
  double d_result = 1;
  for (unsigned i = 0; i < N; i++) {
     result = std::exp(result);
     d_result *= result;
  }
  return d_result;
}
```

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

Crux of AD - Computational graph + Chain rule

```
y = f(x0, x1)
z = g(y)
w0, w1 = l(z)
```





$$\frac{\partial w1}{\partial x0} = \frac{\partial w1}{\partial z}\frac{\partial z}{\partial y}\frac{\partial y}{\partial x0}$$

About Clad

- ***Source transformation based AD tool for C++***
  - *Runs at compile time* - clad generates the code for derivatives using the Abstract Syntax Tree (AST) of the original / primal function as the computational graph.
  - *Implemented as a Clang plugin* - uses the APIs and robust infrastructure of LLVM/Clang for traversing over the parsed graph and generating the derivative code.
  - Aims to enable differentiable programming utilizing all of the features and power of C++.
- ***Supports both forward and reverse mode, also provide functionality for higher order derivatives, Jacobians and Hessians.***

About Clad - usage example

```cpp
// Source.cpp
#include "clad/Differentiator/Differentiator.h"
#include <iostream>

double f (double x, double y) {
  return x*y;
}

double main() {
  // Call clad to generate the derivative of f wrt x.
  auto f_dx = clad::differentiate(f, "x");

  // Execute the generated derivative function.
  std::cout << f_dx.execute(/*x=*/3, /*y=*/4) << std::endl;
  std::cout << f_dx.execute(/*x=*/9, /*y=*/6) << std::endl;

  // Dump the generated derivative code to stdout.
  f_dx.dump();
}
```

```
clang++ -I clad/include/ -fplugin=clad.so Source.cpp
```

```cpp
4 // df/dx for (x,y) = (3, 4)
6 // df/dx for (x,y) = (9, 6)

double f_darg0 (double x, double y) {
  double _d_x = 1;
  double _d_y = 0;
  return _d_x * y + x * _d_y;
}
```

Benefits of Clad

- Readable (hence easily debuggable) generated code for gradient computation.

- Compile time generation of differentiation code enables:
    - Support for control flow expression - *not possible with operator overloading approaches.*
    - Optimization capabilities of the Clang/LLVM Infrastructure enabled by default.
    - Diagnostic messages when differentiation fails.
    - Compile time evaluation - templates, consteval

- Easy integration with cling and ROOT.

- Extra capabilities for customization, experimentation and improving the efficiency of the generated code ….

Providing custom derivatives

- Some use cases:
    - Calling a library function whose definition is not available.
    - Efficiency reasons - you have a better way.
    - Implicit function to be differentiated - for ex. requires solving some maximization problem

```
double my_pow (double x, double y) {
  // … custom code here …
}

namespace clad {
namespace custom_derivatives {

double my_pow_darg0(double x, double y) {return y * my_pow(x, y - 1);} // ∂f/∂x.
double my_pow_darg1(double x, double y) {return my_pow(x, y) * std::log(x);} // ∂f/∂y.
}}
```

To Be Recorded (TBR) analysis in reverse mode

Original function

```
double f_exp(double x, size_t N) {
    for (int i=0; i < N; ++i)
        x = 2 * x;
    return x;
}
```

In RooFit, more than 30% code size reduction.

3x speedup in jit time.

TBR analysis off

```
void f_exp_grad(...) {
    // forward pass
    …
    clad::tape<double> _t1 = {}; // used to store x
    _t0 = 0;
    for (i = 0; i < N; ++i) {
        _t0++;
        clad::push(_t1, x); // x is only transformed linearly so it's
        x = 2 * x;          // value is not needed in the reverse pass
    }
    …
    // reverse pass
    for (; _t0; _t0--) {
        --i;        // i is never used to compute the derivatives
        x = clad::pop(_t1); // no need to restore x
        …
    }
}
```

TBR analysis on

```
void f_exp_grad(...) {
    // forward pass
    …
    _t0 = 0;
    for (i = 0; i < N; ++i) {
        _t0++;
        x = 2 * x;
    }
    …
    // reverse pass
    for (; _t0; _t0--) {
        …
    }
}
```

Live Demo - online service to try out Clad

AD Tutorial - CLAD & Jupyter Notebook

xeus-cling provides a Jupyter kernel for C++ with the help of the C++ interpreter cling and the native implementation of the Jupyter protocol xeus.

Within the xeus-cling framework, Clad can enable automatic differentiation (AD) such that users can automatically generate C++ code for their computation of derivatives of their functions.

[1]:
```
#include "clad/Differentiator/Differentiator.h"
#include <iostream>
```

Forward Mode AD

For a function *f* of several inputs and single (scalar) output, forward mode AD can be used to compute (or, in case of Clad, create a function) computing a directional derivative of *f* with respect to a single specified input variable. Moreover, the generated derivative function has the same signature as the original function *f*, however its return value is the value of the derivative.

[2]:
```
double fn(double x, double y) {
  return x*x*y + y*y;
}
```

[3]:
```
auto fn_dx = clad::differentiate(fn, "x");
```

[4]:
```
fn_dx.execute(5, 3)
```

[4]:
```
30.000000
```

Binder - Jupyter Notebook with Clad

Future Work

- Adding support for stl containers - *std::vector, std::array, std::queue*
- Activity analysis to further improve the generated code, only differentiating statements which contribute towards the final result.
- Improving pointer support - especially tricky in reverse mode AD.
- Better support for compile time evaluations of generated code - *consteval and constexpr*.
- Many more ...

Thank you

Questions or Comments ?