Automatic Differentiation in RooFit for fast and accurate likelihood fits

Jonas Rembser [1] David Lange [2+] Petro Zarytsky [2+] Lorenzo Moneta [1]

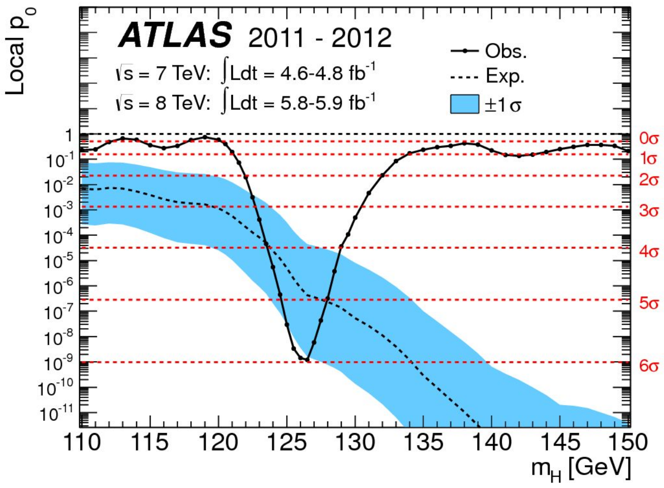
Vaibhav Thakkar [2+] Vassil Vassilev [2+]

1 - CERN 2+ - Princeton University (US) and supported by the National Science Foundation under Grant OAC-2311471.

RooFit

RooFit: C++ library for statistical data analysis in ROOT.

- Used for modelling and normalization of probability density functions (p.d.f)
- Fitting likelihood models to the event data set.
    - Minimizing both binned and unbinned likelihoods
- Used most prominently by the LHC experiments, also for discovering the Higgs boson in 2012
    - Example of profile likelihood scan on the right

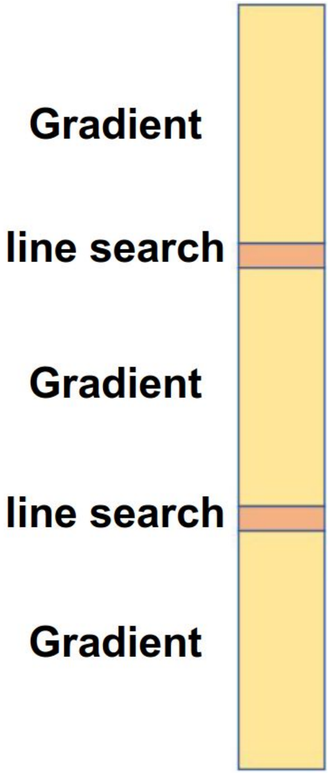


Minimization

- For optimizing parameters, we minimize the likelihood using **Minuit 2** (implements a minimization algo similar to BFGS)

- The minimization time for many-parameter models is dominated by gradient evaluation time
  (see also the ICHEP 2022 RooFit presentation)

- Our goal: make evaluating gradients cheap again with **Automatic differentiation (AD)** using **source code transformation**



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

```
// 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;
}
```

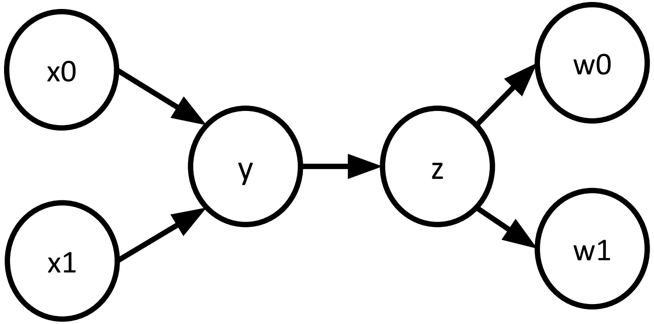
AD

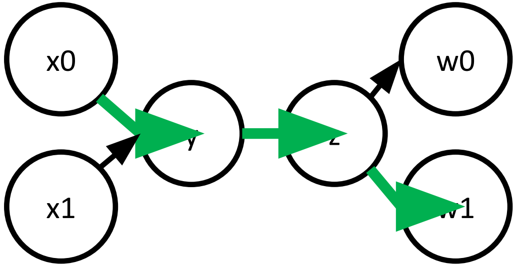
```
double f_dx(double x, int N=5) {
  double result = x;
  double d_result = 1;
  for (unsigned i = 0; i < N; i++) {
     result = std::exp(result);
     d_result *= result;
  }
  return d_result;
}
```

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)
```





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

Essentially, a generalization of backpropagation (from deep learning).

Clad

- ***Source transformation based AD tool for C++***
    - *Runs at compile time - clad generates a readable (and easily debuggable) code for derivatives.*
    - *Optimization capabilities of the Clang/LLVM Infrastructure enabled by default.*
- ***Support for control flow expression - difficult with operator overloading approaches.***
    - *Better handling of complex control flow logic handling compared to machine-learning frameworks like Tensorflow and Pytorch, hence more suitable for scientific computing.*
- ***Integrated with ROOT infrastructure.***
    - *Clad's compiler research team has integration in High Energy Physics (HEP), and making significant improvements for RooFit use case.*

About Clad - usage example

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

double f (double x, double y) {
  return x*y; // <— Function to be differentiated
}

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();
}
```

Compilation (+ execution)

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

Produces

```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;
}
```

New capabilities for customization and improving the efficiency of the generated code

Providing custom derivatives

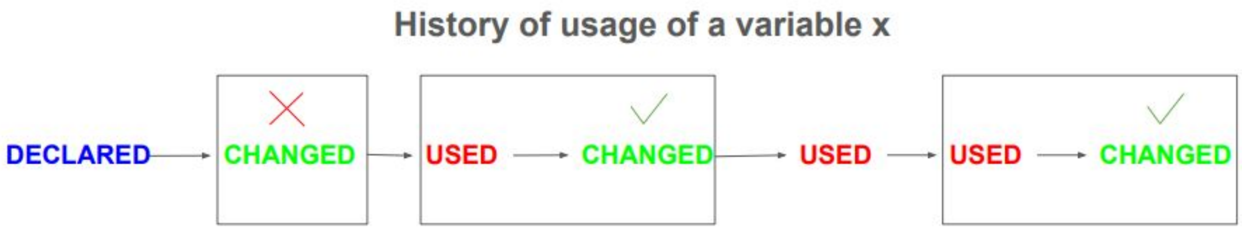
```
double my_pow (double x, double y) {
  // … custom code here for computing xʸ …
}

namespace clad {
namespace custom_derivatives {
// Providing custom code for derivative computation of my_pow.
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.
}}
```

- 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

To Be Recorded (TBR) analysis in reverse mode

Reverse-mode automatic differentiation requires storing intermediate values of variables that have impact on derivatives to restore those in the backward pass.

However, we don’t actually have to store all of them.



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.

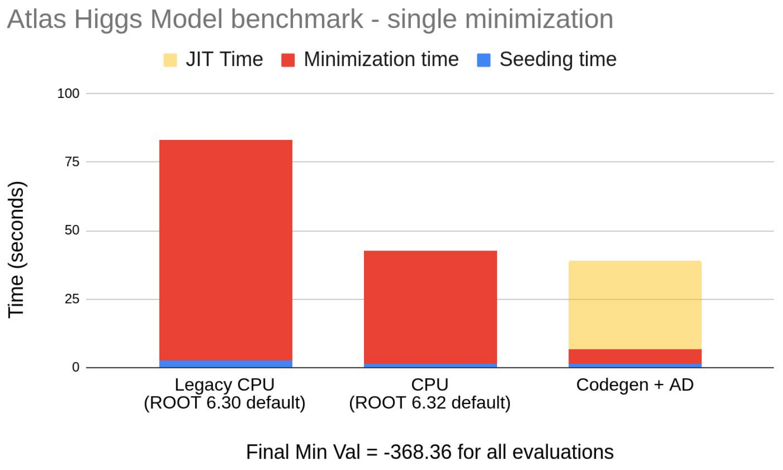
TBR analysis off

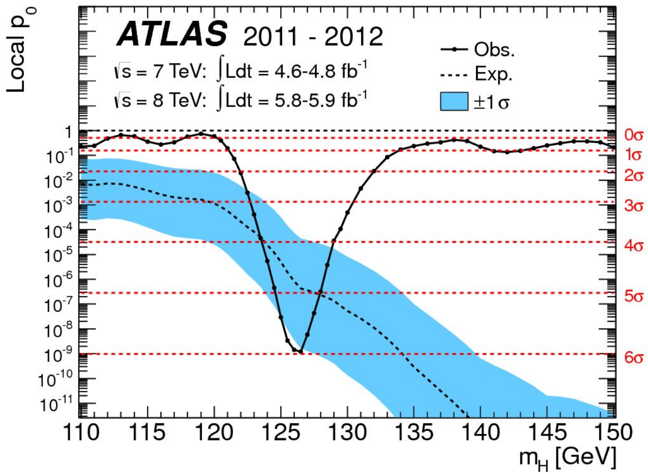
```
void f_exp_grad(...) {
    // forward pass
    …
    clad::tape<double> _t1 = {}; // used to store x
    _t0 = 0;
    for (i = 0; i < N; ++i) {
        _t0++;
        clad::push(_t1, x); // x is only transformed linearly so it's
        x = 2 * x;          // value is not needed in the reverse pass
    }
    …
    // 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--) {
        …
    }
}
```

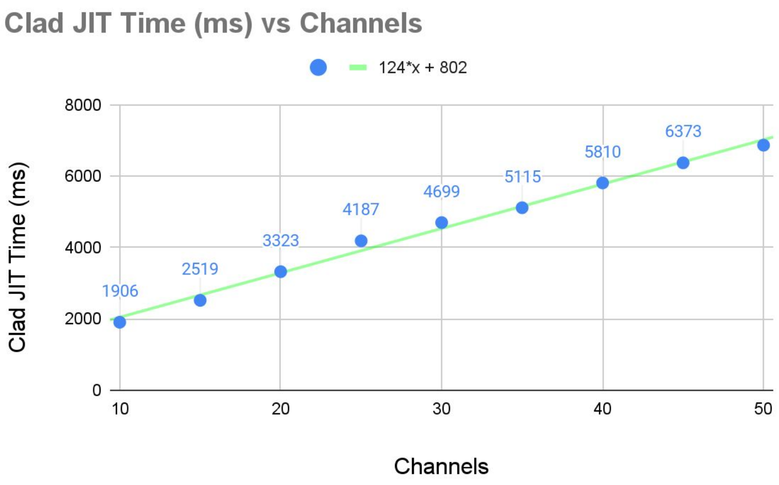
Experiments with Atlas Benchmark models

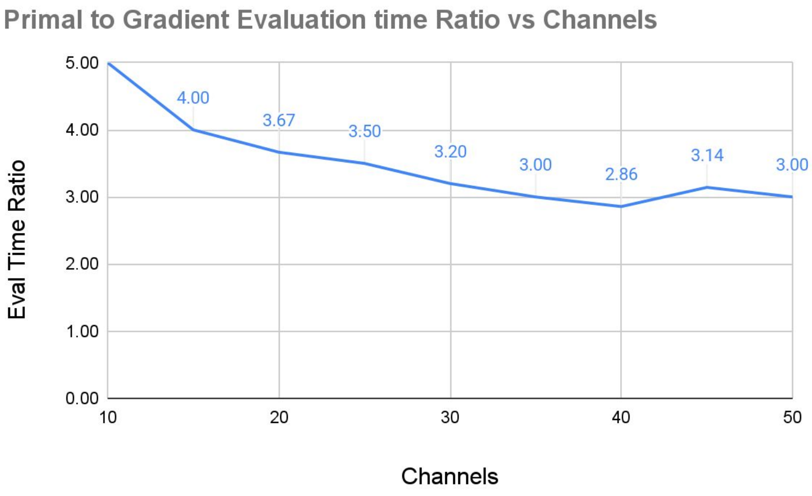




- For multiple minimizations w.r.t different constant parameters, the likelihood gradient can be reused.
    - Amortizing the JIT time across multiple minimizations.

Experiments with Atlas Benchmark models





- Memory consumption of gradient evaluation is very low compared to the python/ML based frameworks.
    - Constant factor of the consumption by primal function.

Some changes that led to these improvements

- Improvement in CPU evaluation backend in RooFit - vectorizing operations, efficient code generation backend : made default in ROOT version 6.32.

- Handling constant pointers for reverse mode AD : #919

- Reducing tape storage operations inside Clad for reverse mode AD : #655

- Dynamic capturing of differentiation plans - capturing and traversing the call graph : #766, #873

Planned improvements to further speedup RooFit

- Using Automatic Differentiation for computing Hessians
  - Computing only the diagonal entries of Hessians.

- Further improvements in Clad to remove redundant computations for Gradients.
  - Advanced analysis for improving the efficiency of Gradient computations.

- Experimenting with make the gradient computation parallelizable.
  - Trying vector forward mode for Hessians.

Thank you

Questions?