Third MODE Workshop on
Differentiable Programming for Experiment Design
Princeton University
24-26 July, 2023

COMPILER RESEARCH

Automatic Differentiation in RooFit

Jonas Rembser*, Vaibhav Thakkar+, Petro Zarytskyi+, Vassil Vassilev+
{ *CERN, +Princeton, compiler-research.org }

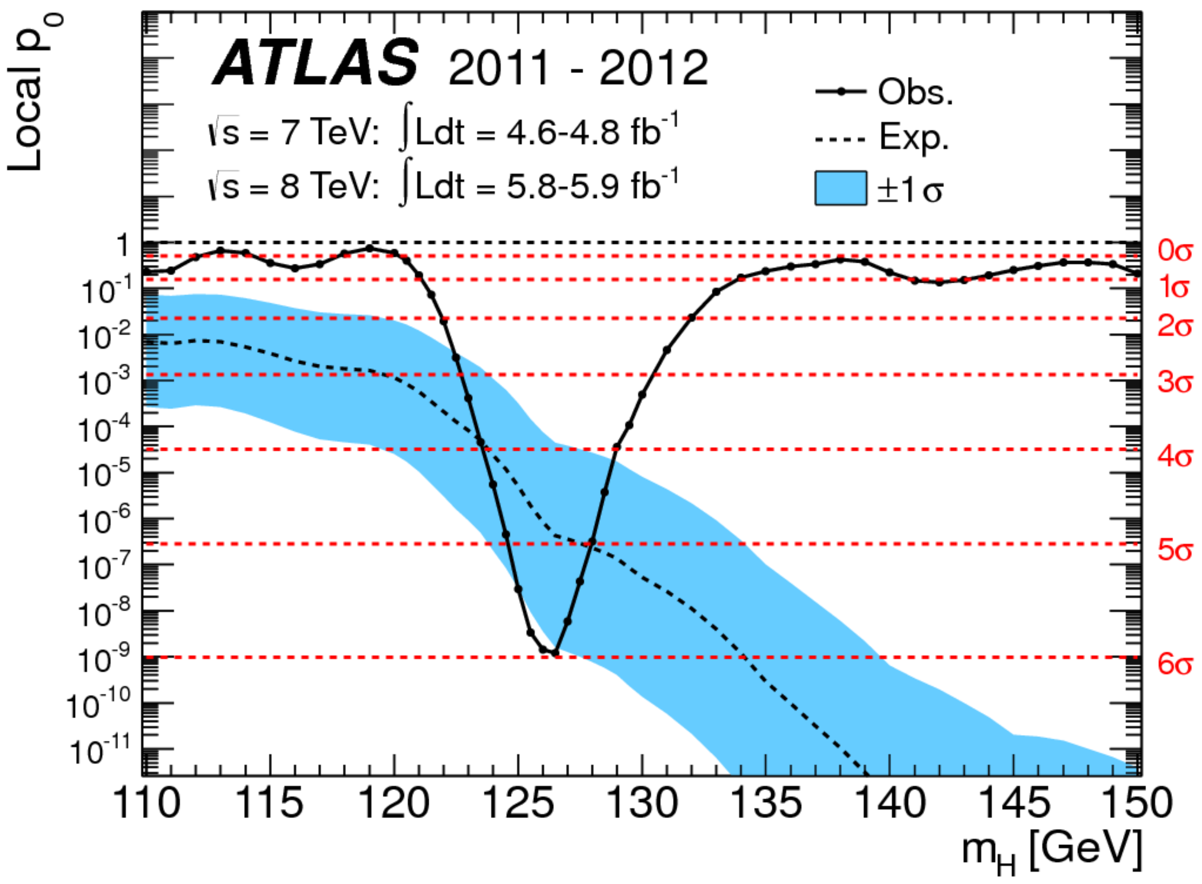
This work is partially supported by National Science Foundation under Grant OAC-2311471

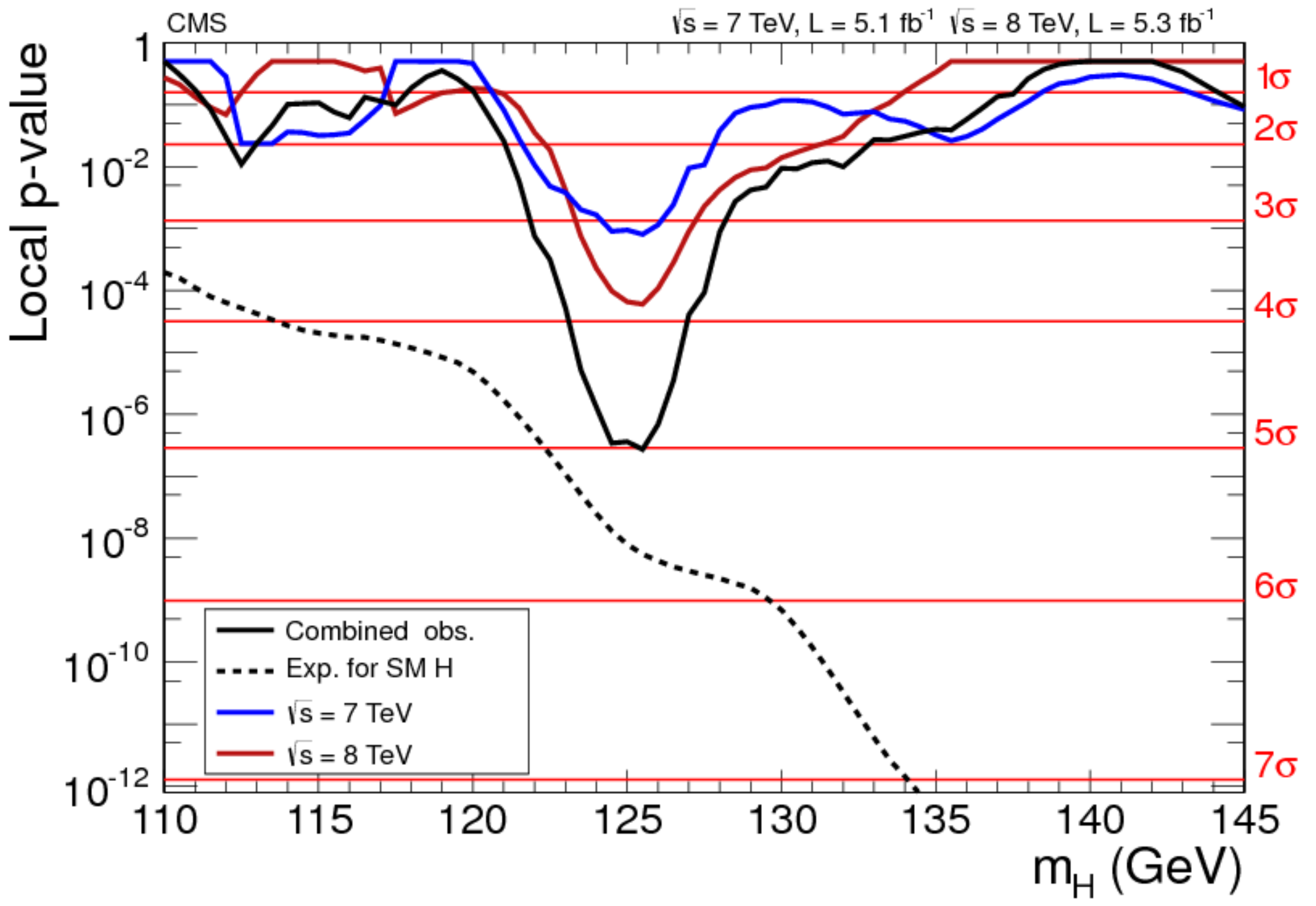
Introduction

If math is the language of science, the language of experimental science is statistics.

Statistical modelling helps us define a scientific narrative by talking to our data sets

Introduction





Observation of a New Boson at a Mass of 125 GeV with the ATLAS and CMS Experiments at the LHC

Credits: ATLAS, CMS Collaborations

Motivation

Likelihoods are central for High Energy Physics

$$L(\vec{n},\vec{a}|\vec{\eta},\vec{\chi}) = \prod_{c\in unbinned\ ch}\ \prod_{i\in obs} \frac{f_c(\vec{x}_{ci}|\vec{\eta},\vec{\chi})}{\int f_c(\vec{x}_{ci}|\vec{\eta},\vec{\chi})\, d\vec{x}_c} \cdot \prod_{c\in binned\ ch(analytical)}\ \prod_{b\in obs} Pois(n_{cb}|\nu(\vec{\eta},\vec{\chi})) \cdot \prod_{\chi\in\vec{\chi}} c_\chi(a_\chi|\chi)$$

Numerical and analytic integrals

$\vec{n}$: $data$, $\vec{a}$: $auxilary\ data$, $\vec{\eta}$: $unconstrained\ parameters$, $\vec{\chi}$: $constrained\ parameters$

CMS Combine Paper https://arxiv.org/pdf/2404.06614

Object Oriented Math with RooFit

$$g_1(\mathrm{x}) = \frac{1}{\sigma_1\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma_1}\right)^2}$$

$$g_2(\mathrm{x}) = \frac{1}{\sigma_2\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma_2}\right)^2}$$

$$P_{bkg}(\mathrm{x}) = \frac{1 + a_0 * T_1(x) + a_1 * T_2(x)}{\int 1 + a_0 * T_1(x) + a_1 * T_2(x)}$$

$$S(\mathrm{x}) = f_{sig1} g_1(x) + \left(1 - f_{sig1}\right) g_2(x)$$

$$\mathrm{Model}(\mathrm{x}) = f_{bkg} P_{bkg}(x) + \left(1 - f_{bkg}\right) S(x)$$

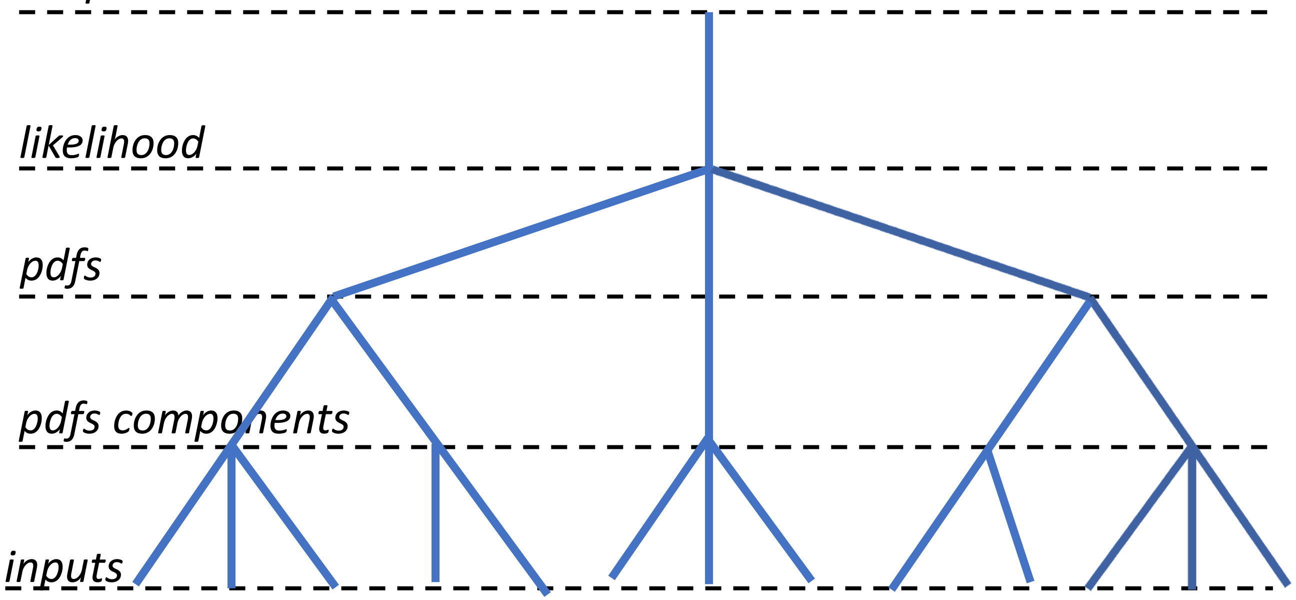
$a_0 = 0.5, a_1 = 0.2, f_{sig1} = 0.8, f_{bkg} = 0.5,$

$\mu = 5, \sigma_1 = 0.5, \sigma_1 = 1.0$

```
RooGaussian sig1("sig1", "Signal component 1", x, mu, sigma1);
RooGaussian sig2("sig2", "Signal component 2", x, mu, sigma2);

// Build Chebychev polynomial pdf
RooChebychev bkg("bkg", "Background", x, {a0, a1});

// Sum the signal components into a composite signal pdf
RooRealVar sig1frac("sig1frac", "fraction of c 1 in signal", 0.8, 0.,
1.);
RooAddPdf sig("sig", "Signal", {sig1, sig2}, sig1frac);

// Sum the composite signal and background
RooRealVar bkgfrac("bkgfrac", "fraction of background", 0.5, 0., 1.);
RooAddPdf model("model", "g1+g2+a", {bkg, sig}, bkgfrac);

// Create NLL function
std::unique_ptr<RooAbsReal> nll{model.createNLL(*data,
EvalBackend("codegen"))};
```

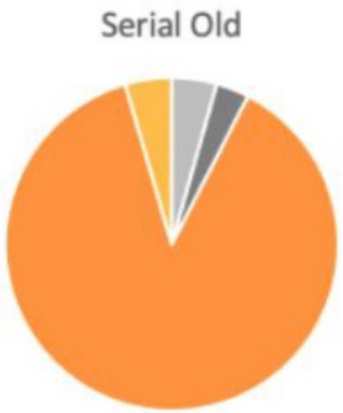
Object Oriented Math. Compute Cost

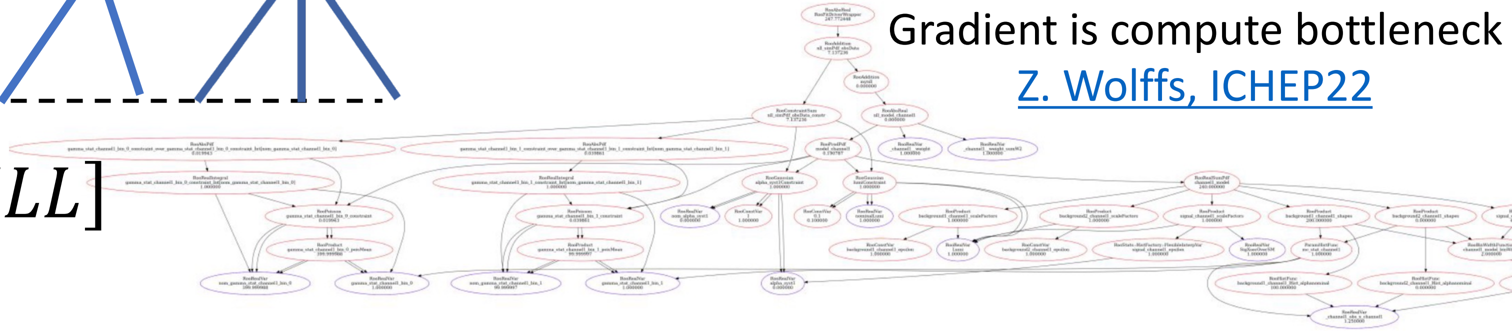
$$NLL = -\sum_{c \in channels} \sum_{i=0}^{N_c} \log(Model(x_i))$$



$$(\hat{\eta}, \hat{\chi}) = \arg\min_{\eta,\chi}[NLL]$$

serial old	
roofit_setup	313
migrad_seed	230
migrad_gradient	6289
migrad_descent	323





Gradient is compute bottleneck
Z. Wolffs, ICHEP22

Lower Compute Cost of Gradients

- Automatic/Algorithmic differentiation (AD) employs the chain rule to decompose the compute graph into atomic operations.
- Top-down decomposition is called forward and bottom up -- reverse mode
- Reverse mode provides independent time complexity of the gradient from input parameters at the cost of adding extra code to enable functions to be run bottom-up (reverse) requiring extra memory
- Operation record-and-replay (operator overloading) or source code transformation are the two common approaches to implement AD

Automatic/Algorithmic Differentiation

$$f(x) = e^{e^{e^{e^{e^{x}}}}} \xrightarrow{\text{Symbolic via Wolfram Alpha}} \frac{d}{dx}\left(e^{e^{e^{e^{e^{x}}}}}\right) = e^{x+e^{e^{e^{e^{x}}}}+e^{e^{e^{x}}}+e^{e^{x}}+e^{x}}$$

Figure out the analytical fn

Handcode

Handcode, optimized by expert

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

AD →

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

Source Code Transformation with Clad

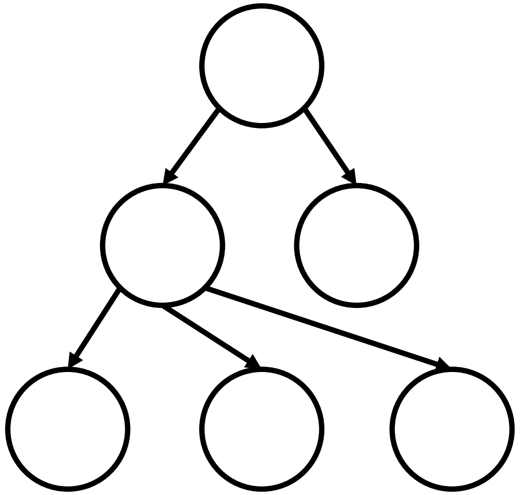
Atell's talk

Extensible Clang/LLVM plugin that runs at compile time to produce readable C++ source code and apply advanced AD high-level analyses

Max's talk

Cla∂ as RooFit’s AD Engine

RooFit Compute Graph



CodeGen/Flatten

Standalone Simplified Compute Graph C++

```
...
double gauss(double *x) {
   using namespace RooFit::Detail;

   return gEvaluate(x[3], (x[0] + x[1]),
(x[2] * 1.5)) /
      gIntegral(-10., 10., (x[0] +
x[1]), (x[2] * 1.5));
}
...
```

AD

Δ

Optimize

FCN

```
pdf.fitTo(data, RooFit::EvalBackend("codegen"))
pdf.createNLL(data, RooFit::EvalBackend("codegen"))
```

Most of HistFactory RooFit primitives are supported. Please reach out if you need additional primitive support

What was a discovery yesterday is a test case today

ATLAS Benchmark Models

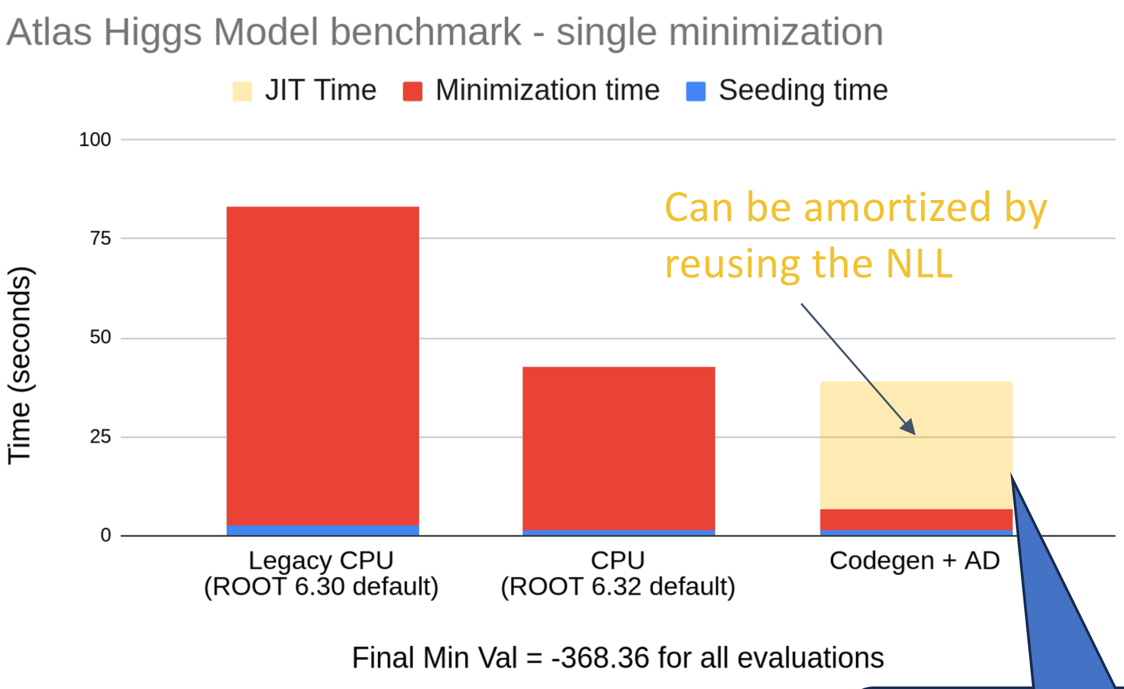
49 HistFactory channels, 739 parameter in total, in rootbench, toy data

How to read this plot:

- **Seeding time**: initial Hessian estimate (num. second derivatives)
- **Minimization time**: finding the minimum
- JIT time: time to generate and compile the gradient code
  - The gradient can be be reused across different minimizations, amortizing the JIT time
  - For example, possible reuse in **profile likelihood scans**

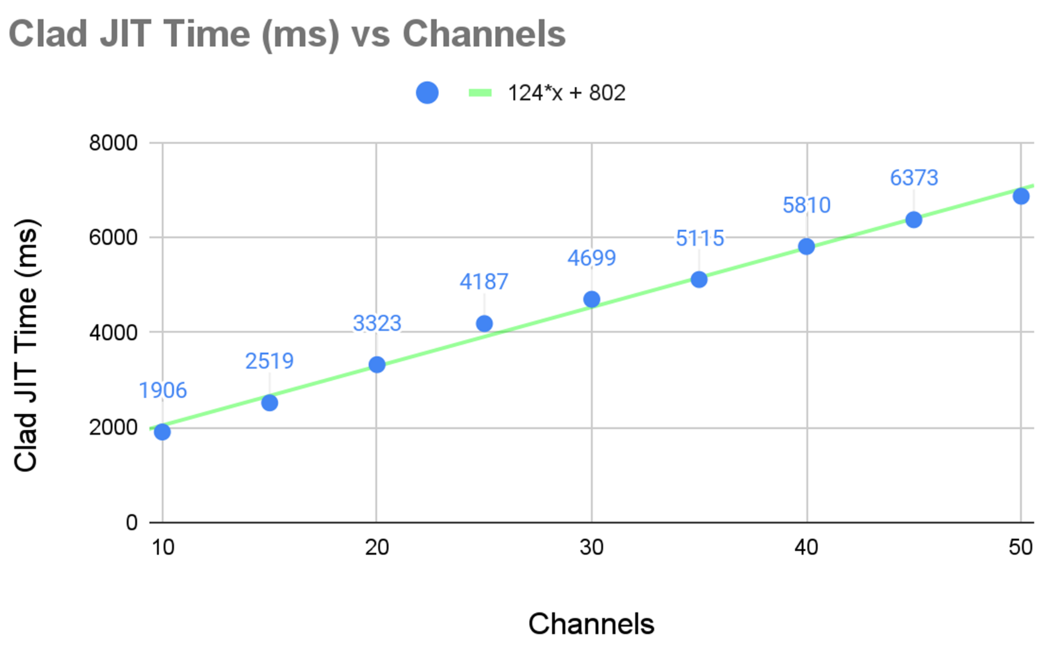
Using AD drastically reduces minimization time on top of the new CPU backend in ROOT 6.32.

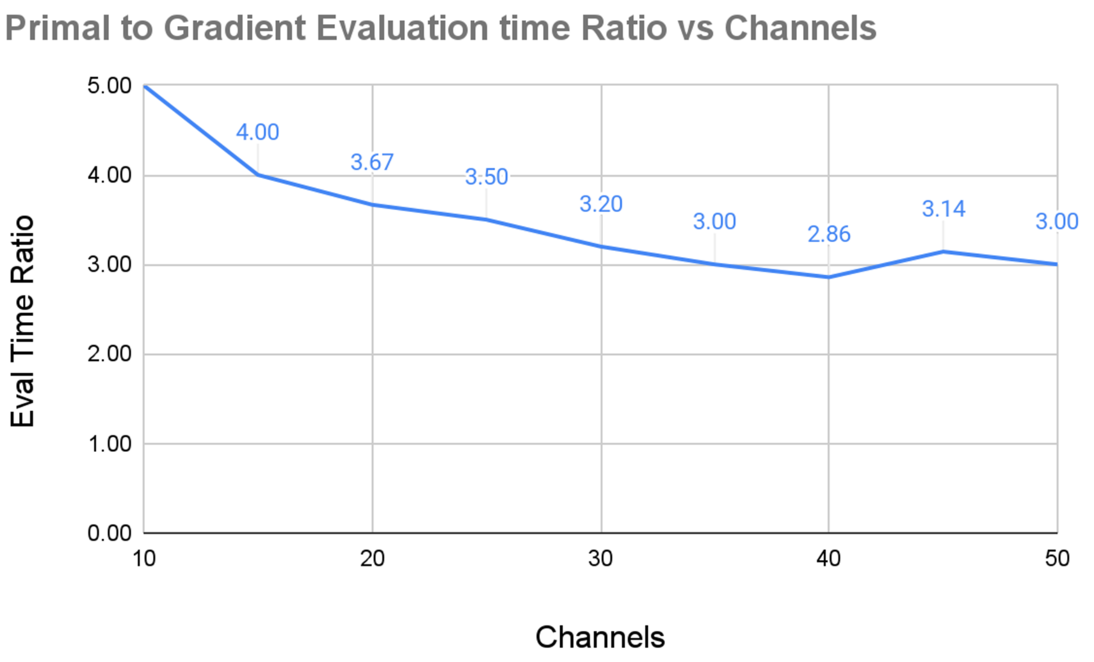
Bottom line: **10x faster minimization** compared to ROOT 6.30.



Max’s talk

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.

CMS Higgs Observation Open Data Models

CMS published RooFit-based Higgs observation likelihood, 672 parameters, 102 channels, real data

Very heterogeneous likelihood:

- Template histogram fits line in the ATLAS benchmark
- Analytical shape fits, **numerical integration** necessary in some cases

Perfect example to test the new RooFit developments

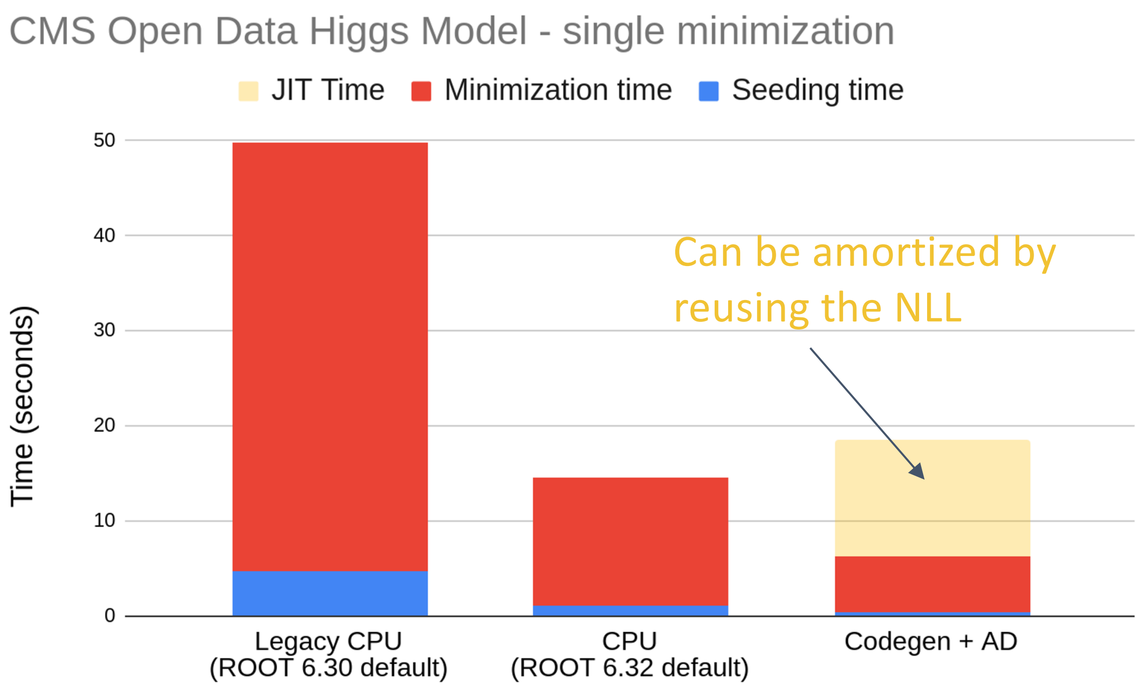
See also the presentation on CMS analysis tools at ICHEP.

We implemented CMS-specific primitives in a custom **CMS combine branch**

Showing **17 changed files** with **1,704 additions** and **113 deletions.**

CMS Higgs Observation Models. Benchmarks

- The new CPU code path default in **ROOT 6.32** is a big improvement to the old RooFit, possibly making many custom improvements in combine *obsolete*
- The AD backend further reduces minimization time
- Printing out the generated NLL code helps a lot to **understand** what's actually fitted
- Work in progress to improve the produced code and its gradient



CMS Higgs Observation Models. Numerical Stability

In the CMS model we observed that the derivatives are small compared to the NLL value

- Numerical differentiation often fails because the finite differences are smaller than numerical precision on the NLL
- Essential workaround for the Higgs model is to offset the NLL by initial value with:

```
pdf.createNLL(data, RooFit::Offset(true))
```

Problems with this:

- Offsetting might fail if initial value is far from the minimum
- Bookkeeping of offsets is error-prone

With AD, the offsetting is not necessary anymore!

```
36 - FCN = -9801946.549 Edm = 0.01129396511
37 - FCN = -9801946.566 Edm = 0.01497173883
38 - FCN = -9801946.574 Edm = 0.007242353199
39 - FCN = -9801946.583 Edm = 0.004954953322
40 - FCN = -9801946.589 Edm = 0.005774308843
41 - FCN = -9801946.596 Edm = 0.004695329674
42 - FCN = -9801946.602 Edm = 0.004558156748
43 - FCN = -9801946.615 Edm = 0.008141300763
44 - FCN = -9801946.625 Edm = 0.004861879849
45 - FCN = -9801946.628 Edm = 0.003472778648
46 - FCN =  -9801946.63 Edm = 0.001782083931
47 - FCN = -9801946.631 Edm = 0.0007515760698
```

Minimizer output, showing the small changes wrt. large NLL value

Possible next steps and perspectives

- Make the codegen backend default for RooFit
- Work together with experiments to **support your usecases** and help out in **integration RooFit AD in experiment frameworks**
- **Extend RooFit's interfaces** so it will be easy to get out the generated code and gradients to use them outside the RooFit minimization routines
- R & D on **analytic higher-order derivatives** that are used in Minuit
- Implement advanced clad-based analyses to remove the redundant computation

Conclusion

Source-code transformation AD with Clad fits naturally into the ROOT ecosystem and RooFit benefits from it in many ways:

- **Faster** likelihood **gradients**
- No need for tricks to get **numerically stable** gradients
- Likelihoods can be expressed in **plain C++** without need for aggressive **caching** by the user or in frameworks like RooFit
  - **Good for understanding** the math: optimization gets decoupled from logic - simple code
  - **Good for collaboration**: simple C++ can easily be shared and used in other contexts

Thank you!