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

COMPILER C | R RESEARCH

Scaling RooFit's Automatic Differentiation Capabilities to CMS Combine

Jonas Rembser[*], David Lange[+], Vassil Vassilev[+]
{ [*]CERN, [+]Princeton, compiler-research.org }

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

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

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

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

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

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

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

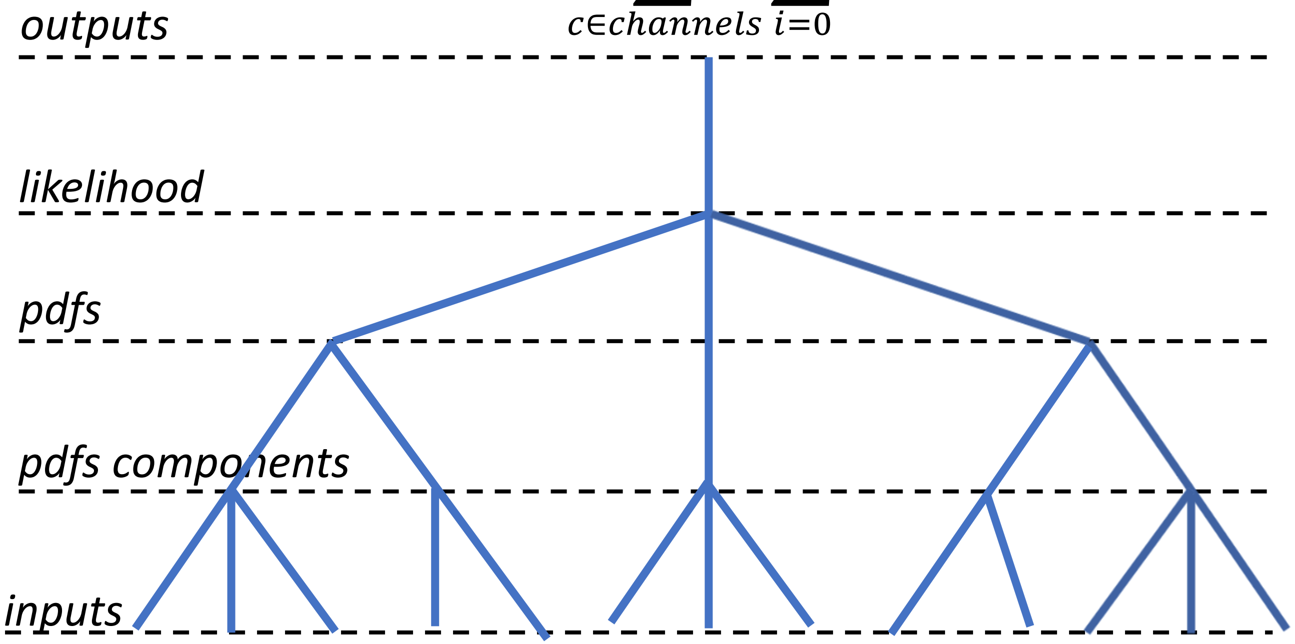
$a_0 = 0.5, a_1 = 0.2, f_{sig1} = 0.8, f_{bkg} = 0.5,$

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

```
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"))};
```

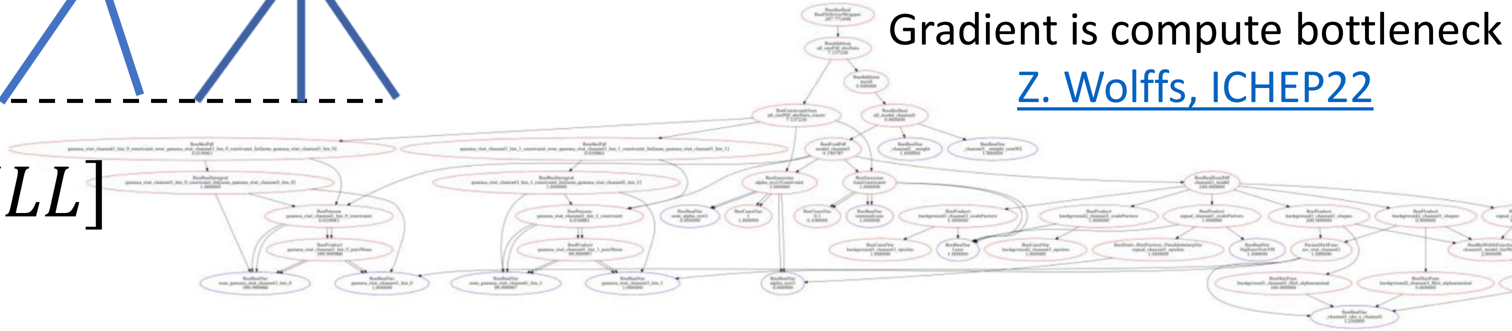
Object Oriented Math. Compute Cost

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



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

serial old	
roofit_setup	313
migrad_seed	230
migrad_gradient	6289
migrad_descent	323



Gradient is compute bottleneck
[Z. Wolffs, ICHEP22]

Statistical Modelling in CMS

- CMS Combine is the flagship tool for statistical modelling in CMS. It is based on RooFit but has many customizations.
- The workflows run for days once the statistical model is constructed
- Most workflows are dominated by the gradient part of the minimization step
- ClaↃ is a compiler-based source transformation automatic differentiation tool integrated in RooFit. It is capable of generating cheap gradients whose asymptotic computational time complexity is independent on the size of the inputs

Integration in CMS Combine

Work steered mostly via CAT hackathons. Thank you Aliya Nigamova and Piergiulio Lenzi!

First RooFit AD integration #1019

Merged **lenzip** merged 10 commits into `cms-analysis:main` from `guitargeek:roofit_ad_dev` on Apr 4

Conversation 12 | Commits 10 | Checks 9 | Files changed 38

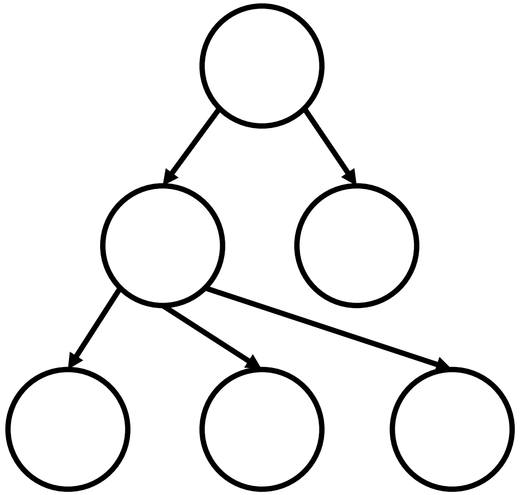
guitargeek commented on Nov 18, 2024 • edited

Contributor

Enable the `"codegen"` backend with Automatic Differentiation for an initial set of Combine models.

Clad 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

Combine Compute Graph

No need to recompile RooFit

Visit each node

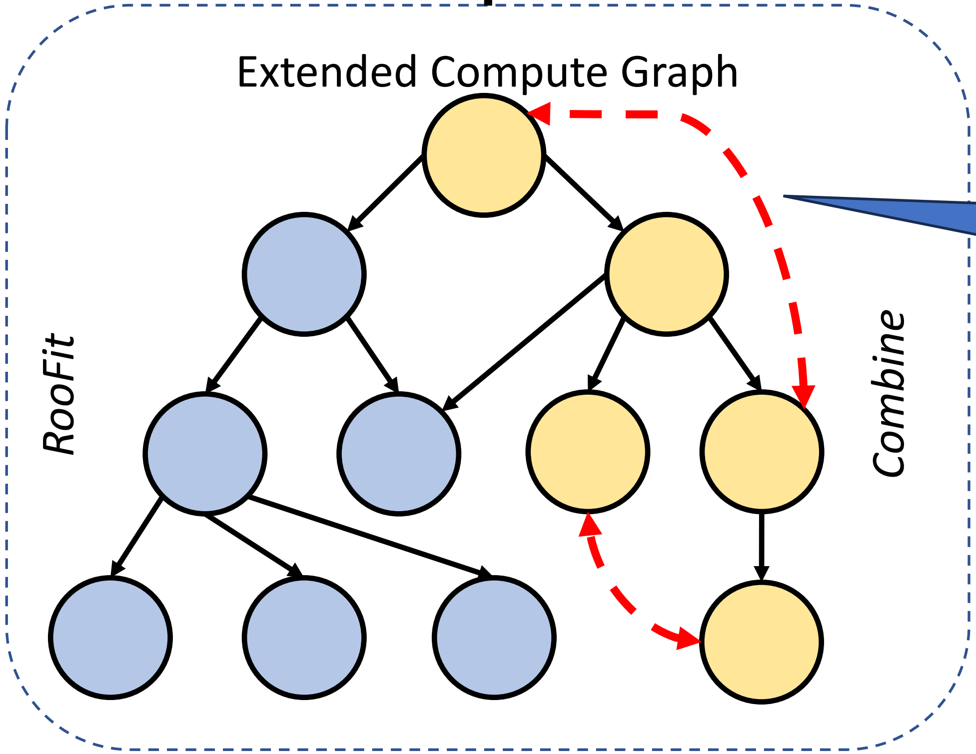
```
void codegenImpl(RooAddPdf&, CodegenContext&);
void codegenImpl(RooChebychev&, CodegenContext&);
...
void codegenImpl(ProcessNormalization&, CodegenContext&);
void codegenImpl(FastVerticalInterpHistPdf2&, CodegenContext&);
```

AD

Δ

Optimize

FCN



More cleanup is needed to avoid layering violations

Annotated Combine Compute Graph

Semantic Meaning

RooAbsArg Name

```
// ProcessNormalization::n_exp_bindijet_proc_qqH[ thetaList=(pdf_qqbar) asymmThetaList=()
otherFactorList=(r_qqH) ] = 0.95
const double t20 = RooFit::Detail::MathFuncs::processNormalization(
      0.950000, 1, 0, 1, t19, xlArr + 6, nullptr, xlArr + 6, xlArr + 6, t18);

// RooAddition::n_exp_bindijet[ n_exp_bindijet_proc_ggH + n_exp_bindijet_proc_qqH +
n_exp_bindijet_proc_bkg ] = 4.55
  const double t21 = (t17 + t20 + params[4]);


// RooNLLVar[ pdf=model_s weightVar=_weight _weight_sumW2=_weight_sumW2 ] = 0
for (int loopIdx1 = 0; loopIdx1 < 1; loopIdx1++) {
  nll_result += RooFit::Detail::MathFuncs::nll(t25, obs[3], 0, 0);
}
```

zero because of offsetting

Crosscheck with RooFit evaluate

Combine Supported Primitives

- Some of the optimisations/tricks implemented at the time are now bottlenecks
- For example, Crystal Balls

	Combine (`RooDoubleCBFast`) (per loop)	Native (`RooCrystalBall`) (per loop)
Object creation	28.5 μs ± 7.74 μs (7 runs, 10,000 loops each)	28.4 μs ± 1.69 μs (7 runs, 10,000 loops each)
Event generation (100k events)	292 ms ± 19.9 ms (7 runs, 10 loops each)	241 ms ± 15.2 ms (7 runs, 10 loops each)
Minimization	10.3 s ± 1.64 s (7 runs, 2 loops each)	5.89 s ± 840 ms (7 runs, 2 loops each)

- Minimisation is slower as function evaluation is less stable
  - For example: $\frac{e^n}{e^m} = e^{n-m}$ can be non-NaN, even if e^n, e^m are individually very large. Combine computes each term separately, then takes the ratio

7 / 16

Tom Runting, AD in Combine, 23rd April, 2025

Combine Supported Primitives

To our estimation ~40% of the core Combine classes are supported:

- ProcessNormalization, AsymPow, FastVerticalInterpHistPdf2, FastVerticalInterpHistPdf2D2
- VerticalInterpPdf after PR1060

Classes in RooFit upstream to support combine:

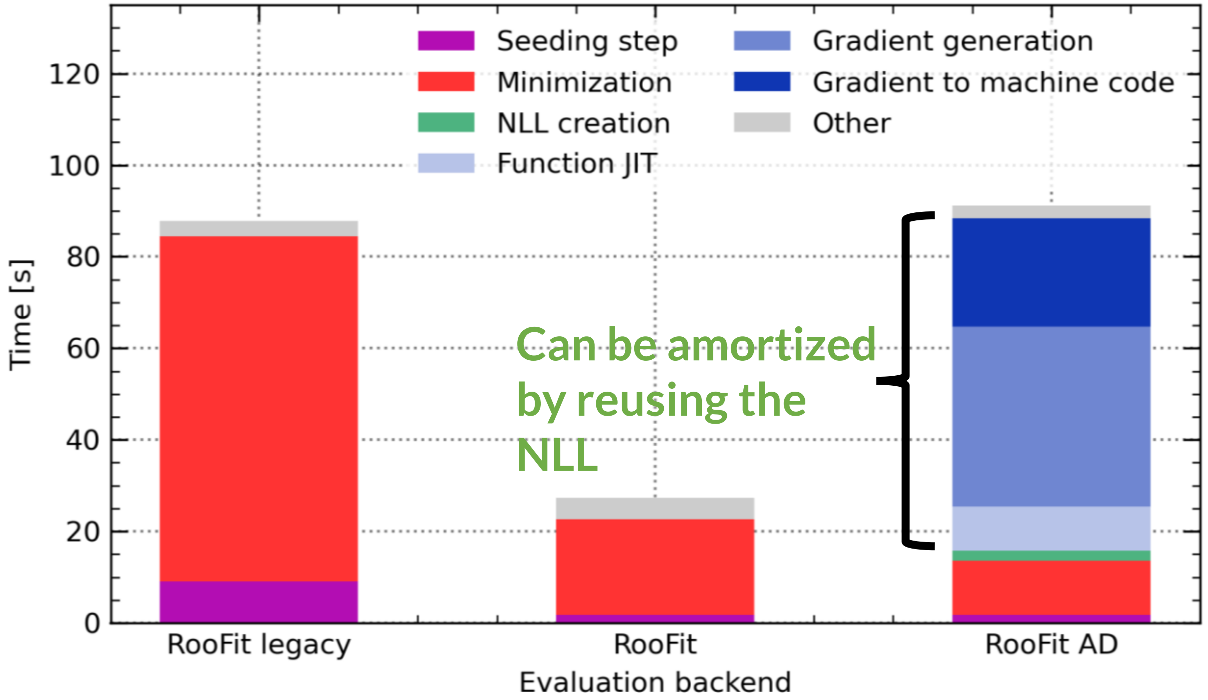
- RooParametricHist, RooHistPdf

Track progress in real time here

CMS Higgs Combination Benchmark

CMS published its Higgs likelihood observation model [Higgs observation likelihood]

- Very heterogeneous likelihood: **672 parameters** in **102 channels** with
  - Template histogram fits
  - Analytical shape fits, numerical integration necessary in some cases
- **Perfect example** to test the new Combine developments



CMS Higgs Observation Models. Numerical Stability

In this 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:

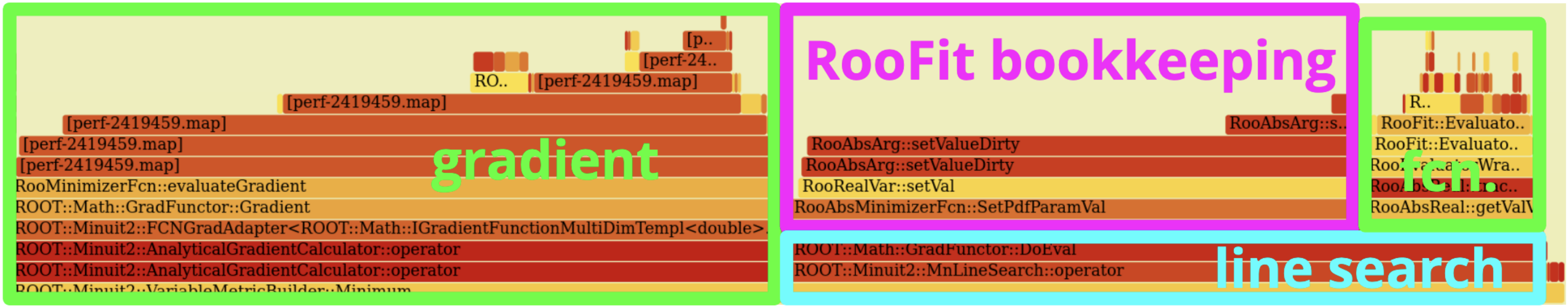
- Offsetting might fail if initial value is far from the minimum
- Bookkeeping of offsets is error-prone

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

Profile of CMS Higgs Combination Benchmark



- Profiling **CMS** minimization ([full flamegraph]()). **Gradient not the bottleneck anymore!**
- Likelihoods in CMS Combine are very optimized, so the **RooFit bookkeeping overhead** is relatively larger
- Once RooFit bookkeeping overhead is gone, further optimizing the gradient could be worth it

Extensive study by Jonas Rembser at [*https://compiler-research.org/meetings/#caas_05June2025*](https://compiler-research.org/meetings/#caas_05June2025)

Better Continuous Integration

To scale development we needed to enhance several infrastructure parts of Combine:

- Update the building Combine logic outside of CMSSW
- Enhanced static analysis on pull requests with clang-tidy (Matthew Barton)
- Formatting consistency with clang-format (Matthew Barton)
- Improved tests and validation that's run on every pull requests (Keila Moral)

Open Challenges

- Reduce jitting cost
  - Persistify likelihoods across multiple runs on the grid.
- Static RooFit computation graphs
  - No update operations from one end of the graph to the other (eg rework RooMultiPdf-like classes, analytic minimization of nuisance parameters)
- CI infrastructure for advanced testing and validation
- Ultimately Combine should reuse the generated gradient for all points in profile likelihood scans even distributed on the grid

Conclusion

Source-code transformation AD with Clad fits naturally into the ROOT, RooFit and Combine 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

A Less-Boring Conclusion

Data → **Likelihood** → **Fit** → **EFT constraints**.

RooFit/Combine likelihoods 2–10x faster would have a major positive impact on EFT analyses in both practical and strategic ways:

- Expand the scope of EFT analyses
- Improve the quality and precision of constraints
- Enable new techniques and collaborationss
- Shorten the time from theory to results

Thank you!

Offsetting

```
double chan1 = 1e-2 * nll_channel(params);
double chan2 = 1e3 * nll_channel(params + 2);
return chan1 + chan2; // 0.01 + 1000

if (DoOffset) {
   static double offset = 0.0;
   if (offset == 0.0) {
      offset = ret;   // Save initial value (1e6)
   }
   ret -= offset;     // Now ret is closer to 0
}
```

- Numerical differentiation becomes **more accurate**.
- Only Helps When Initial Value Dominates
- Makes Debugging and Logging Confusing
- Fails if Input Changes Too Much
  - If you move far from the original parameter values:
  - The offset is no longer meaningful.
  - The difference between ret and offset becomes large again, so **numerical instability returns**.

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

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

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

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

ATLAS Benchmark Models

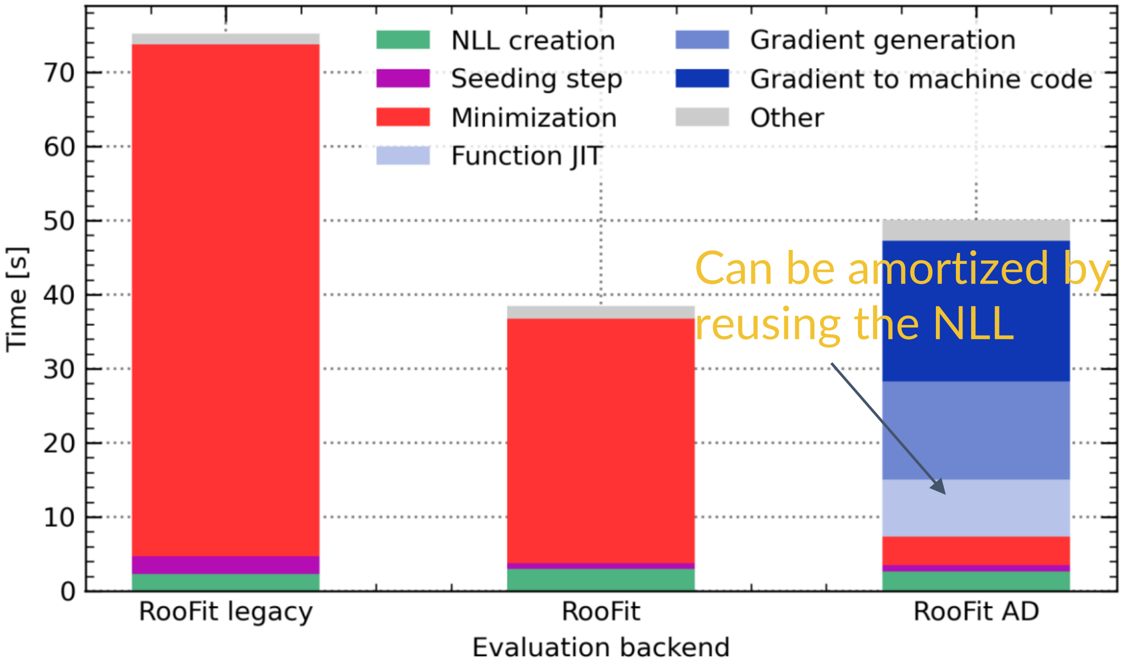
49 HistFactory channels, 739 parameter in total, in rootbench, toy data

How to read this plot:

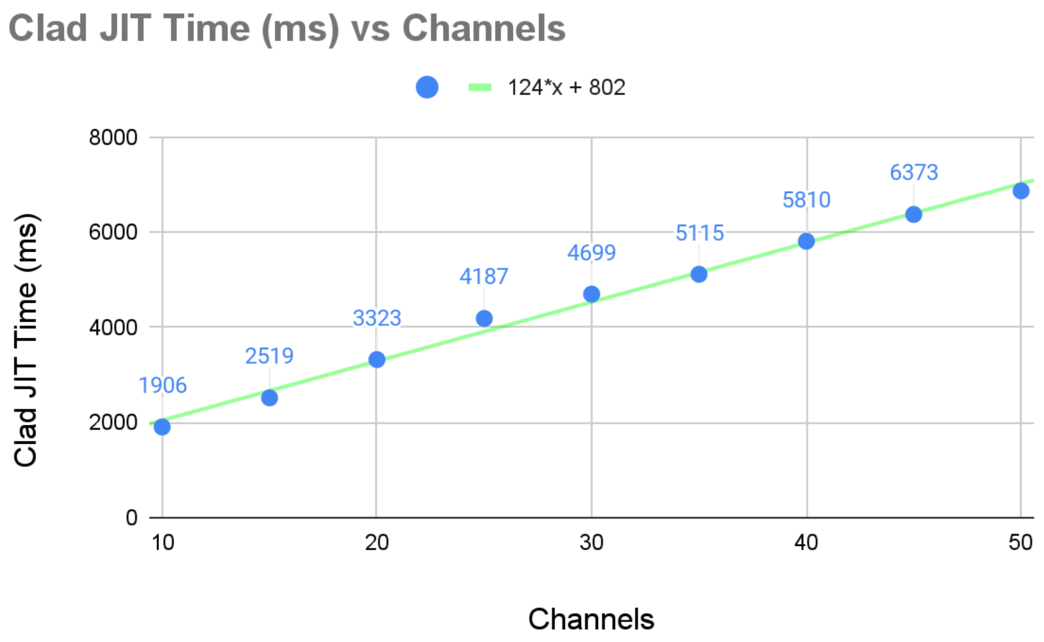
- **Seeding time**: initial Hessian estimate (num. second derivatives)
- **Minimization time**: finding the minimum
- JIT time: time to generate and compile the gradient code
  - The gradient can be be reused across different minimizations, amortizing the JIT time
  - For example, possible reuse in **profile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

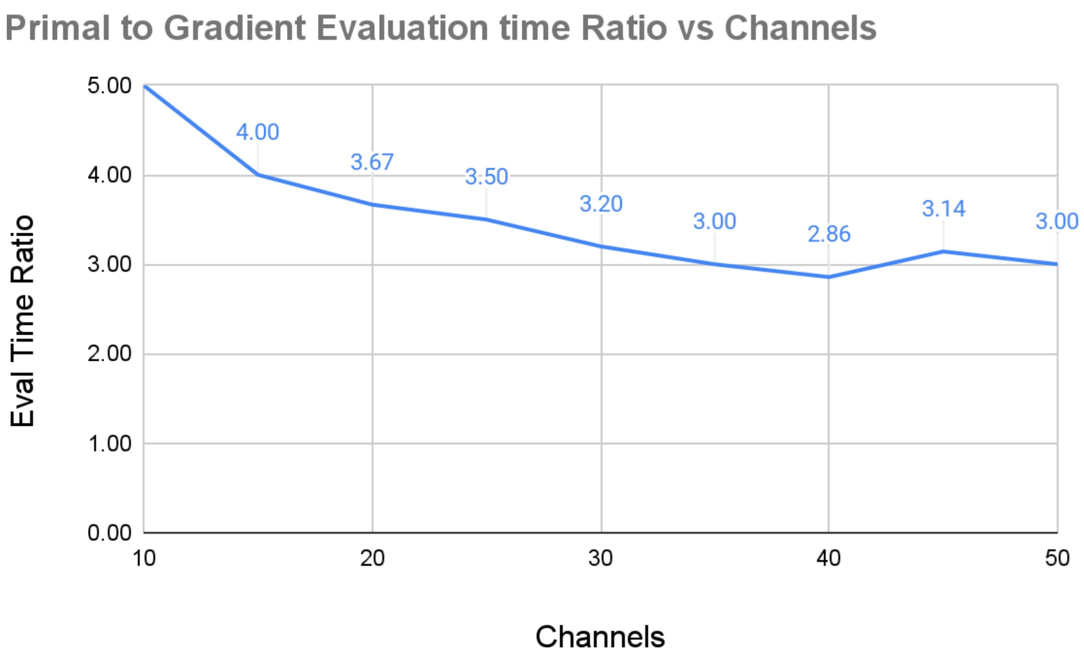
Using AD drastically reduces minimization time on top of the new CPU backend in ROOT 6.32.

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



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.