CERN Summer Student Project

Comparative Benchmarks for RooFit in Python

Summer Student:	Daniel Werner
Supervisors:	Jonas Rembser, Lorenzo Moneta
Group:	EP-SFT, ROOT-Team
Project duration:	25.06 - 22.09.2023

Contents

1 Abstract

The statistical models that are used in modern High Energy Physics (HEP) research are independent of their specific implementations. As a consequence, many different tools have been developed to perform statistical analyses in HEP. These implementations differ both in their performance, but also in their usability. In this scenario, comparative benchmarks are essential to aid users in choosing a library and to support developers in resolving bottlenecks or usability shortcomings.
The goal of this project was to make a comparison of some of the most used tools in statistical analyses. The Python package `pyhf` focuses on template histogram fits, which are the backbone of many analyses at the ATLAS and CMS experiments. In contrast, `zfit` mainly focuses on unbinned fits to analytical functions. These frameworks are compared to the C++ library RooFit, which is used via Python bindings. RooFit can perform both template histogram fits and unbinned fits and is thus compared to both other packages.
The benchmarks show that the RooFit backends `'off'` and `'cpu'` perform best for template histogram fits with few bins, while the JAX backend of `pyhf` is the fastest backend for many bins. For multiple channels, all `pyhf` backends perform much worse. In unbinned fits, the `'cpu'` and `'cuda'` backends of RooFit perform much better than the `zfit` implementation on both CPU and GPU. The `'codegen'` backend, using AD, performs better than the default for many events.
The code to reproduce the results of this project can be seen in the Git repository [1].

2 Statistics background

2.1 Likelihood fitting

A likelihood function $L(\mathbf{n}|\boldsymbol{\theta})$ describes the probability of measuring a result $\mathbf{n}$ given a set of parameters $\boldsymbol{\theta}$ of the model. To find the best parameters $\hat{\boldsymbol{\theta}}$ for describing the measurement, the likelihood can be maximised. For practical purposes in numerical optimisations, the logarithm of the likelihood function is often maximised instead.
In many applications, one of the parameters is the parameter of interest (POI) μ, while the other parameters are called nuisance parameters. To express the likelihood only in terms of this POI, the profile likelihood $\lambda(\mathbf{n}|\mu)$ can be used:

$$\lambda\left(\mathbf{n}\,|\,\mu\right) = \frac{L\left(\mathbf{n}\,|\,\mu, \hat{\hat{\boldsymbol{\theta}}}\right)}{L\left(\mathbf{n}\,|\,\hat{\mu}, \hat{\boldsymbol{\theta}}\right)}\,. \tag{1}$$

In this formula, $\hat{\hat{\boldsymbol{\theta}}}$ are the best-fit nuisance parameters from a fit with μ being fixed. This new function has a maximum of 1 in the case that μ is identical to the best-fit value $\hat{\mu}$ and becomes close to 0 for unlikely values of the POI.

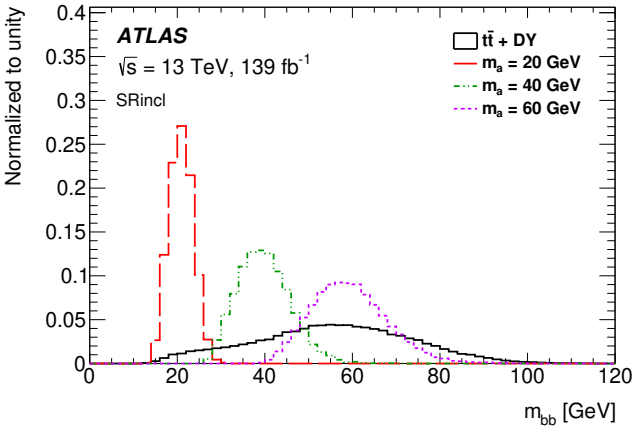

Figure 1: Template histograms from an ATLAS analysis [2].

2.2 Template histograms

A probability distribution for a binned variable x can be described as a template histogram. In such a histogram, the number of expected events $E[n_i]$ in a bin i is the sum of the expected background events b_i and the signal events μs_i given a signal strength μ. The expectations for both background and signal generally depend on some nuisance parameters $\boldsymbol{\theta}$. Template histograms are generated by simulating events and their measured quantities. The uncertainties in the modelling and the statistical uncertainty are accounted for through the nuisance parameters. A measurement is described through the event counts $\mathbf{n}$ in all N bins. The number of events in each bin follows a Poisson distribution around the expected value. Additional measurements $\mathbf{m}$ independent of μ can be used to constrain the nuisance parameters. The likelihood function for a template histogram with constrained nuisance parameters is:

$$L\left(\mathbf{n}, \mathbf{m} \,|\, \mu, \boldsymbol{\theta}\right) = \prod_{j=1}^{N} \frac{\left(\mu s_j(\boldsymbol{\theta}) + b_j(\boldsymbol{\theta})\right)^{n_j}}{n_j!} e^{-\left(\mu s_j(\boldsymbol{\theta}) + b_j(\boldsymbol{\theta})\right)} \cdot \prod_{k=1}^{M} \frac{u_k^{m_k}(\boldsymbol{\theta})}{m_k!} e^{-u_k(\boldsymbol{\theta})} \tag{2}$$

An advantage of using template histograms in analyses is that no analytical function is needed to describe the shape of a distribution. However, one still needs to Monte Carlo simulations of events to generate the templates. An example of template histograms as used for the search of new particles in an ATLAS analysis can be seen in Figure 1.

2.3 Hypothesis tests

Hypothesis tests are used to decide whether to accept or reject a null hypothesis H_0 in favour of an alternative hypothesis H_1. In particle physics, these hypotheses would typically be the background-only (b) hypothesis $\mu = 0$ or the signal-and-background (sb) hypothesis $\mu = 1$. The decision is then based on a critical value of some test statistic. For excluding the signal hypothesis, the test statistic q_μ can be used [3]:

$$q_\mu = \begin{cases} -2\log(\lambda(\mu)) & \hat{\mu} \leq \mu \\ 0 & \hat{\mu} > \mu \end{cases} \tag{3}$$

This quantity grows large when λ is close to 0, corresponding to a disagreement between the model and the measurement. When a large excess of events over the background is found, leading to $\hat{\mu} > \mu$, the measurement does not disagree with the sb-hypothesis and the test statistic reaches its minimal value of 0.
The result $\hat{\mu} < 0$ cannot correspond to a negative signal strength in our physical model but only to statistical fluctuations. Therefore, such results should not have additional statistical power in rejecting the signal hypothesis. To solve this problem, the test statistic $\tilde{q}_\mu$ can be introduced with a modified profile likelihood [3]:

$$\tilde{q}_\mu = \begin{cases} -2\log\left(\frac{L\left(\mu,\hat{\hat{\theta}}\right)}{L\left(0,\hat{\hat{\theta}}\right)}\right) & \hat{\mu} < 0 \\ -2\log(\lambda(\mu)) & 0 \leq \hat{\mu} \leq \mu \\ 0 & \hat{\mu} > \mu \end{cases} \tag{4}$$

The probability of finding a disagreement with H_0 of at least $\tilde{q}_{\mu,\text{obs}}$ if H_0 were true is the p-value as seen in (5). To compute the p-value, the probability distribution function (pdf) of $\tilde{q}_\mu$ must be known. Asymptotic formulae for such pdfs in the case of many events are provided in [3].

$$p_\mu = \int_{\tilde{q}_{\mu,\text{obs}}}^{\infty} f\left(\tilde{q}_\mu|\mu\right) \mathrm{d}\tilde{q}_\mu \tag{5}$$

As a measure for excluding a hypothesis, the confidence level CL_s of any μ can be calculated as seen in (6). This quantity is a relation between the significance $\alpha = p_\mu$ of a test and its exclusion potential $1 - \beta$.

$$\text{CL}_s(\mu) = \frac{\int_{\tilde{q}_{\mu,\text{obs}}}^{\infty} f\left(\tilde{q}_\mu|\mu\right) \mathrm{d}\tilde{q}_\mu}{\int_{\tilde{q}_{\mu,\text{obs}}}^{\infty} f\left(\tilde{q}_\mu|0\right) \mathrm{d}\tilde{q}_\mu} = \frac{\alpha}{1-\beta} \tag{6}$$

In many analyses, the exclusion limit is determined as the μ value that leads to $\text{CL}_s \leq 0.05$. A visualisation of α and $1 - \beta$ with the pdfs from [3] is seen in Figure 2.

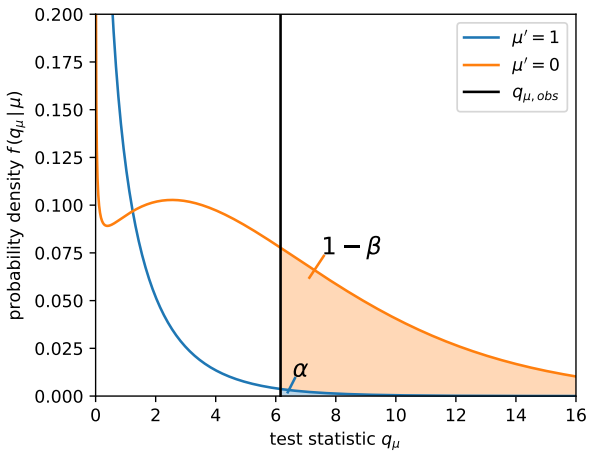

Figure 2: Visualisation of pdfs of q_μ [3] for different hypotheses and the quantities α and $1 - \beta$ from equation (6).

3 Computational background

3.1 Computation graph

The models used in likelihood fits can be represented as a computation graph. An example of such a graph can be seen in Figure 3. In such a graph, each primary node (depicted in blue) represents an input value. Nodes referencing other nodes via edges (red) represent functions with the referenced values as inputs. By calculating the value for each node step-by-step, a final pdf value can be computed from all input variables. The likelihood is then the product of pdf values for different observable values.

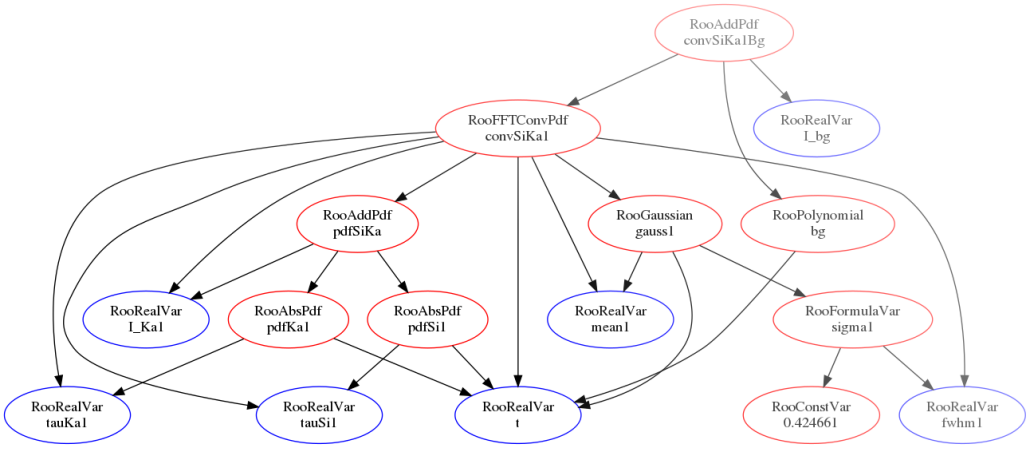

Figure 3: Visualisation of a pdf model as a computation graph.

3.2 Vectorization

When evaluating likelihoods for many different parameter values, the same mathematical operations are applied to a large number of values. This process can be sped up by treating the set of values as a vector with many entries and applying the operation to the entire vector at once. This technique is called the vectorization of the computation and can significantly reduce the runtime of calculations.
Some compilers can turn calculations done in loops into vectorized CPU instruction calls, which is called auto-vectorization.

3.3 Automatic differentiation

Differentiation of functions can be done in multiple ways with different disadvantages. Numerical derivatives work by using finite differences between function values, which introduces rounding errors. In contrast, symbolic differentiation turns the function into a mathematical expression that has an analytical derivative, which can be computationally expensive. Both of these methods have in common, that they take long for higher level derivatives or functions with many inputs.
An alternative method is automatic differentiation (AD), which uses the chain rule of derivatives as seen in equation (7). Since all functions are implemented in code as a

combination of elementary operations and functions, this rule reduces any derivative to a product of known derivatives of these elementary functions.

$$f(x) = g(h(x)) \Rightarrow \frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial h(x)}\frac{\partial h}{\partial x} \tag{7}$$

4 Statistical analysis frameworks

4.1 RooFit

ROOT is a C++ framework for data analysis that was developed at CERN in 1997 [4]. RooFit is a special library for statistical analyses within ROOT [5]. The functionality of RooFit is to define observables and parameters and to build pdfs from them. These can be combined to more complex models that are either unbinned or binned. The minimisation software MINUIT2 is used within RooFit to fit the models to datasets [6, 7].
By default, likelihoods in RooFit are determined by evaluating the computation graph of the likelihood for each event in an outer loop. This method does not allow for vectorization to improve the performance. However, the intermediate results of the computation graph are cached, so that nodes are only recomputed when one of their parameters changes. This default mode can enforced through the option `EvalBackend('off')`.
An alternative method was introduced as `EvalBackend('cpu')` [8]. This backend introduces vectorization by executing vectorizable loops over each node of the computation graph.
Another backend was introduced to ROOT to allow for the use of GPUs in likelihood evaluation [9]. The `EvalBackend('cuda')` option tells ROOT to launch CUDA kernels to compute the likelihoods. In addition, the CPU can be used to perform some of the computations.
The newest of these backends that was introduced to ROOT is the `EvalBackend('codegen')` [10]. Before evaluating the likelihood, this backend compiles the likelihood into minimal C++ code. Through the use of the automatic differentiation software Clad [11], a derivative of this code is then compiled.

4.2 pyhf

Histograms can be turned into template histogram pdfs via the HistFactory tool [12] that was designed for the use in ROOT. `pyhf` is a pure-Python implementation of this tool that allows to build template histograms and perform fits of them in Python [13, 14]. The likelihoods can be evaluated through one of several Python packages that are available as backends in `pyhf`.
The NumPy backend saves all values in NumPy-arrays to vectorize the operations on these values. In addition to running compiled C++ code in the background, this accounts for the performance of NumPy. All the other backends use data formats that are similar to NumPy-arrays.
Another package that is used as a backend in `pyhf` is the machine learning framework TensorFlow. TensorFlow uses AD through `tf.GradientTape` to compute the gradient of

the likelihood. This AD method records all used functions in the likelihood and combines their derivatives into the gradient. The data format is called `tf.Tensor`. All computation graphs in TensorFlow can be optimised through Accelerated Linear Algebra (XLA).
A similar machine learning framework is PyTorch that uses `Autograd` for automatic differentiation and saves data in its own `torch.tensor` format.
The last backend of `pyhf`, JAX combines the updated version of Autograd with XLA to achieve efficient gradient computations for machine learning.

4.3 zfit

In 2019, `zfit` was developed to provide a fitting framework for HEP in Python [15, 16]. Similarly to RooFit, `zfit` is used by defining observables, parameters, and finally models. The likelihoods and gradients of these models are then provided and optimised by the underlying package TensorFlow. In the end, the minimisation itself is performed by typical minimizers used in HEP like the Python wrapper around MINUIT2 called `iminuit` [17].

5 Benchmarks

5.1 Implementation (pytest-bench)

The plugin pytest-benchmark for the Python package pytest provides a wrapper function for benchmarking functions in Python. To use this plugin, a function with the naming scheme `test_<name>(benchmark, *args)` is used. Within this function, the benchmarking wrapper is called with the function to be benchmarked and all its arguments as can be seen in line 12 of the example code in Listing 1.

```
1  import pytest
2
3  backends = ["off", "cpu", "numpy", "jax"]
4  backend_ids = ["RooFit_off", "RooFit_cpu", "pyhf_numpy",
5                 "pyhf_jax"]
6
7  @pytest.mark.parametrize("backend", backends, ids=backend_ids)
8  def test_fitting(benchmark, backend, n_bins):
9      set_backend(backend)
10     model = get_model(n_bins)
11     data = get_data(n_bins)
12     result = benchmark(perform_fit, data, model)
```

Listing 1: Example code of using pytest-benchmark

The code also shows how some preparations like creating the model and data are performed outside the benchmarked code. In addition to the benchmarking wrapper, pytest provides the option to parametrize parameters of the benchmark. An example for this is given with the backends in the example code. When pytest is called, the function `test_fitting` is called once for every possible value for `backend` that is provided through

the parameterization. In the case of the argument `n_bins`, the list of all values is given in a config file and can even be provided as a command line argument to pytest.

5.2 HistFactory Benchmark

For benchmarking the different implementations of HistFactory, a rather simple template histogram is chosen. In each bin, the expected signal count is a fixed value, while the number of background events is affected by a statistical uncertainty on the template. This Poisson distributed statistical effect is described by one nuisance parameter for each bin, leading to N nuisance parameters in addition to the single signal strength.
To ensure a fair comparison, the settings of MINUIT2 are set to identical values in both RooFit and `pyhf`. Additionally, it was checked that the parameter bounds, initial values and the data are identical in both cases. The number of calls to the minimizer is verified to be identical in all frameworks when the same data is used. A ratio between the CL_s result in both implementations is shown in the plots to show that the results are identical.
The benchmark is performed for templates with different numbers of bins to see how the different backends perform in different analyses. The results of these runs can be seen in Figure 4, with the RooFit code shown in shades of red and the `pyhf` backends in shades of blue. This benchmark was run on an Intel Core i7-6700 CPU with 4 cores that were only used in multithreading by the PyTorch and JAX backends.

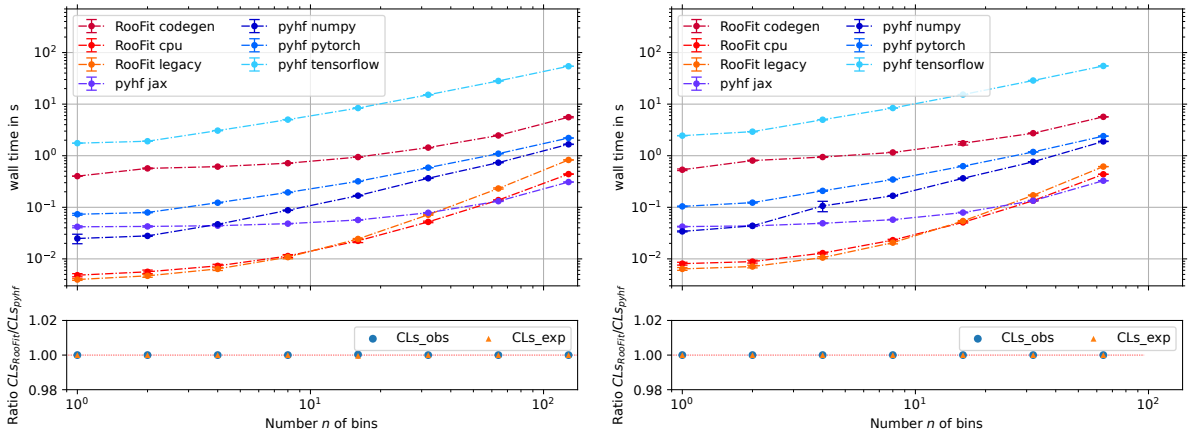

Figure 4: Benchmark of HistFactory implementations for different number n of bins in 1 channel (left) and in 2 channels (right).

One can see that the backends without AD are the fastest ones for few bins but get closer to the other backends for many bins. This is because the number of parameters is proportional to the number of bins, and AD shines for many parameters while for few parameters the overload dominates.
The fastest backend for around 100 bins is JAX. In TensorFlow the overhead is by far the largest, followed by `'codegen'`. The vectorization of `'cpu'` compared to `'off'` leads to an improved performance at around 10 bins.
Another run of the benchmark was done, varying the number of channels in the analysis as it is common for analyses to use multiple channels. This result is shown in Figure 5.

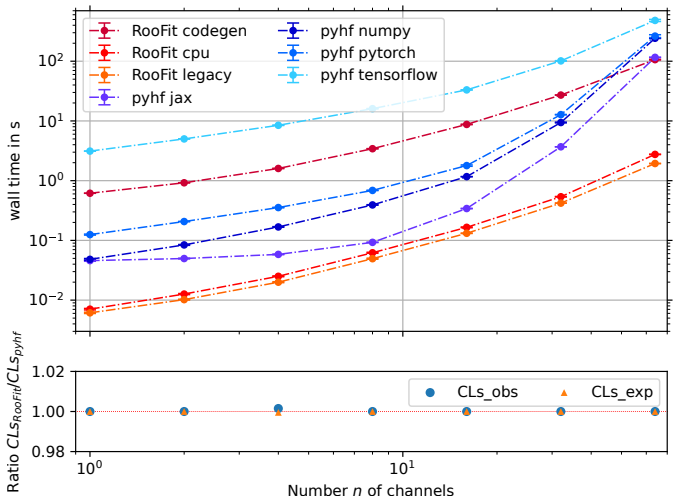

Figure 5: Benchmark of HistFactory implementations for different number n of channels with 4 bins per channel.

The main observation from this plot is that all `pyhf` backends become very slow for many channels. Investigating this effect showed that in `pyhf`, a loop over all channels is performed in Python. An additional insight from this benchmark is that the relative difference between `'cpu'` and `'off'` is unaffected by the number of channels, as the likelihood evaluation is only vectorized within each channel and then combined.

5.3 Unbinned Fitting Benchmark

The model used to benchmark the fitting of unbinned pdfs is the sum of nine Gaussian distributions that each have a different central value. An example plot resulting from this fit of 18 parameters is shown in Figure 6.
In `zfit`, one cannot specify any backends for the likelihood evaluation. Instead the user can choose whether the gradient should be provided by TensorFlow (`'grad'`) or by MINUIT2 (`'no_grad'`). Additionally, TensorFlow can optimise a computation graph (`'graph'`) or run in "Eager mode" without this optimisation (`'no_graph'`).

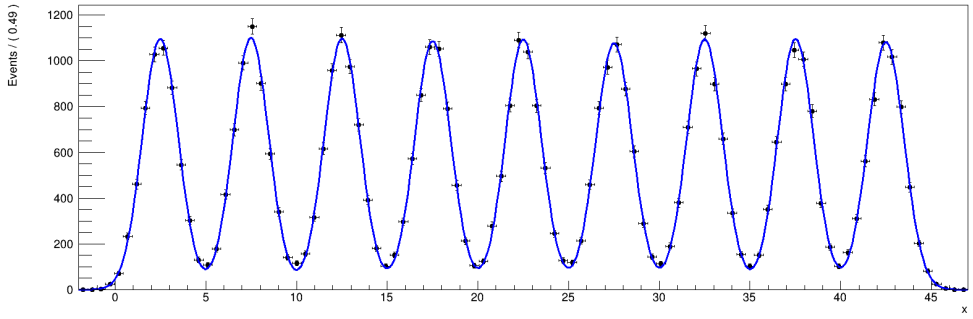

Figure 6: Example plot of a model of nine added Gaussian distributions fitted to data.

As in the HistFactory benchmark, it is verified that all settings, parameters, and data are identical. Again, the number of calls to the minimizer is the same in both implementations.

To show the identical results, the ratio between the central value of the first Gaussian in both frameworks is shown in the resulting plots.
The variable that was changed while performing this benchmark is the number of events. By running the benchmark on both a CPU and a GPU, the results in Figure 7 are obtained. The used CPU is an AMD Ryzen 9 3900 and the GPU is an NVIDIA RTX A4500. All of the CPU backends used only one core.

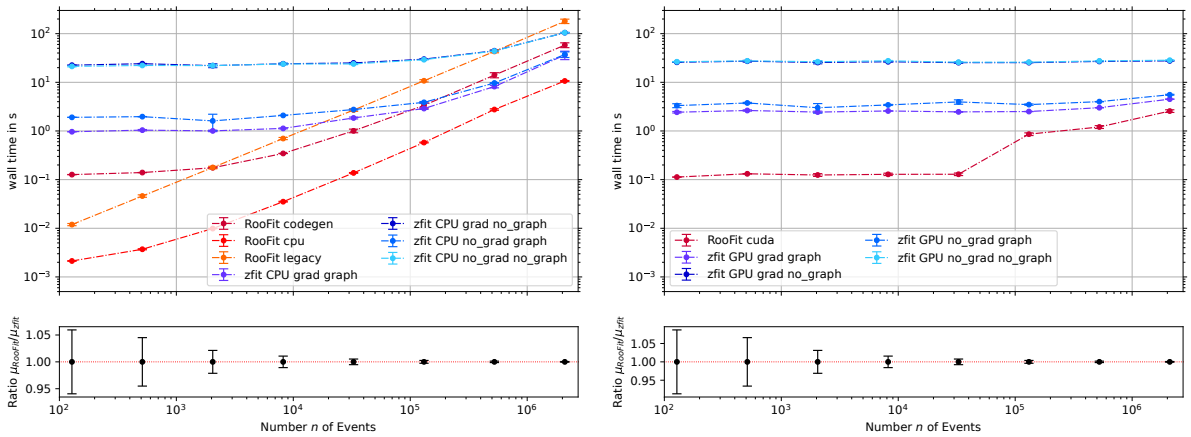

Figure 7: Benchmark of an unbinned fit for different number n of events run on a CPU (left) and a GPU (right). The curves with the "no_graph" option are hard to distinguish as they are nearly identical. The error bars in the ratio plot are determined from the uncertainty of μ as estimated by the fit.

From the results on a CPU one can again see the overhead of TensorFlow and `'codegen'`. The advantage of the vectorization can easily be seen from the comparison of `'cpu'` and `'off'`, which is more than one order of magnitude slower for many events. After around 2000 events, even `'codegen'` is faster than `'off'`, as the gradient and function evaluation become more complex with more events.
The "Eager mode" of TensorFlow is significantly slower as would be expected due to the missing optimisation of the graph. Using the gradient provided by Tensorflow is only slightly faster than using the one from MINUIT2.
While `zfit` is faster than the `'codegen'` and `'off'` backends of RooFit for many events, the `'cpu'` backend is clearly even faster.
From the benchmark run on a GPU, one can see that the `'cuda'` backend has much less overhead than TensorFlow. The `'cuda'` backend can handle 512 events at once with each of its 56 stream processors. As a consequence, serialisation is only necessary when more than $\approx 29\,000$ events are processed. This is a possible explanation for the sudden increase in duration from $\approx 32\,000$ to $\approx 128\,000$ events.
The GPU performance becomes better than the CPU performance around $2 \cdot 10^6$ events and the duration does not change significantly for `zfit` up to this point. This suggests that the time only grows with n at higher values and is dominated by the approximately constant overhead before. As a consequence, higher n should be benchmarked to estimate which of the frameworks performs best in the limit of many events.

6 Summary and Outlook

In this project, the first comparative benchmarks for the studied frameworks were created. A special focus was to ensure the fairness in comparing the different fits.
The developers of `pyhf` and `zfit` were informed about this project and given the opportunity to have a look at the code to ensure that their frameworks are not run inefficiently. The feedback from the developers was very positive.
On a personal level, I had the opportunity to learn a lot about statistical analysis and how it is performed computationally. I had a very detailed insight in how the different frameworks work behind the high-level functions. Finally, I want to thank Jonas Rembser for the supervision and his help in this project.

Future goals for these benchmarks are to add more models and more frameworks. The `zfit`-developer Jonas Eschle had several suggestions for models that can be run in both RooFit and `zfit`. One idea would be to use custom pdfs whose analytical integrals are unknown so that the integrals must be calculated numerically, potentially changing the performance of the different backends.
For adding more frameworks, one could consider adding the C++ framework GooFit or including `zfit` in the HistFactory benchmark, as `zfit` supports binned fits as well.
Finally the code will be included in some public repository like `rootbench` [18]. This will allow other users to add models and frameworks to the benchmark or to repeat the benchmarks with new versions of the different frameworks.

References

[1] D. Werner, *statanabenchmark*, Git repository https://gitlab.cern.ch/dawerner/statanabenchmark/ (2023)

[2] ATLAS Collaboration, *Search for Higgs boson decays into a pair of pseudoscalar particles in the bbμμ final state with the ATLAS detector in pp collisions at* $\sqrt{s}$ = 13 *TeV*, Phys. Rev. D **105**, 012006 (2021)

[3] G. Cowan, K. Cranmer, E. Gross, O. Vitells, *Asymptotic formulae for likelihood-based tests of new physics*, Eur. Phys. J. C **71**, 1554 (2011), [Erratum: Eur.Phys.J.C 73, 2501 (2013)]

[4] R. Brun, F. Rademakers, *ROOT: An object oriented data analysis framework*, Nucl. Instrum. Meth. A **389**, 81 (1997)

[5] W. Verkerke, D. Kirkby, *The RooFit toolkit for data modeling*, in *Statistical Problems in Particle Physics, Astrophysics and Cosmology*, pages 186–189, World Scientific (2006)

[6] F. James, M. Roos, *Minuit: A System for Function Minimization and Analysis of the Parameter Errors and Correlations*, Comput. Phys. Commun. **10**, 343 (1975)

[7] M. Hatlo, F. James, P. Mato, L. Moneta, M. Winkler, A. Zsenei, *Developments of mathematical software libraries for the LHC experiments*, IEEE Trans. Nucl. Sci. **52**, 2818 (2005)

[8] S. Hageboeck, L. Moneta, *Making RooFit Ready for Run 3*, J. Phys. Conf. Ser. **1525**, 012114 (2020)

[9] E. Michalainas, J. Rembser, S. Hageboeck, L. Moneta, *Acceleration with GPUs and other RooFit news*, Journal of Physics: Conference Series **2438(1)**, 012066 (2023), URL https://dx.doi.org/10.1088/1742-6596/2438/1/012066

[10] G. Singh, J. Rembser, L. Moneta, D. Lange, V. Vassilev, *Making Likelihood Calculations Fast: Automatic Differentiation Applied to RooFit*, in *26th International Conference on Computing in High Energy & Nuclear Physics*, EPJ Web Conf., **unpublished**

[11] V. Vassilev, M. Vassilev, A. Penev, L. Moneta, V. Ilieva, *Clad — Automatic Differentiation Using Clang and LLVM*, Journal of Physics: Conference Series **608(1)**, 012055 (2015), URL https://dx.doi.org/10.1088/1742-6596/608/1/012055

[12] K. Cranmer, G. Lewis, L. Moneta, A. Shibata, W. Verkerke (ROOT), *HistFactory: A tool for creating statistical models for use with RooFit and RooStats*, Technical report, New York U., New York (2012), URL https://cds.cern.ch/record/1456844

[13] L. Heinrich, M. Feickert, G. Stark, K. Cranmer, *pyhf: pure-Python implementation of HistFactory statistical models*, Journal of Open Source Software **6(58)**, 2823 (2021), URL https://doi.org/10.21105/joss.02823

[14] L. Heinrich, M. Feickert, G. Stark, *pyhf: v0.7.4*, https://github.com/scikit-hep/pyhf/releases/tag/v0.7.4, URL https://doi.org/10.5281/zenodo.1169739

[15] J. Eschle, A. Puig Navarro, R. Silva Coutinho, N. Serra, *zfit: Scalable pythonic fitting*, SoftwareX **11**, 100508 (2020), URL https://www.sciencedirect.com/science/article/pii/S2352711019303851

[16] J. Eschle, A. Puig, R. S. Coutinho, *zfit: scalable pythonic fitting* (2019), URL https://doi.org/10.5281/zenodo.2602043

[17] H. Dembinski, P. O. et al., *scikit-hep/iminuit* (2020), URL https://doi.org/10.5281/zenodo.3949207

[18] ROOT-project, *rootbench*, GitHub repository https://github.com/root-project/rootbench (2017)