Acceleration with GPUs and other RooFit news

Emmanouil Michalainas, Jonas Rembser, Stephan Hageboeck and Lorenzo Moneta

CERN, Esplanade des Particules 1, 1211 Geneva 23, Switzerland

E-mail: `emmanouil.michalainas@cern.ch`, `jonas.rembser@cern.ch`, `stephan.hageboeck@cern.ch`, `lorenzo.moneta@cern.ch`

Abstract. RooFit is a toolkit for statistical modeling and fitting, and together with RooStats it is used for measurements and statistical tests by most experiments in particle physics, particularly the LHC experiments. As the LHC program progresses, physics analysis becomes more computationally demanding. Therefore, RooFit development in recent years is focused on modernizing RooFit, improving its ease of use, and on performance optimization. This paper presents the new RooFit vectorized computation mode, which supports calculations on the GPU. Additionally, we discuss new features in the upcoming ROOT 6.26 release, highlighting the new pythonizations in particular.

1. Introduction

RooFit [1] is a C++ package for statistical modeling distributed with ROOT [2]. RooFit's development started in the year 2000 within the BaBar collaboration. Since then, RooFit has been a reliable tool for many experiments in high-energy physics at B factories and the Large Hadron Collider. With RooFit, one can build statistical models with observables, parameters, functions and PDFs[1], which can be fitted to data, plotted or used for statistical tests.

Recently, many of RooFit's core computation functions have been re-implemented to better use modern hardware, like GPUs or the vector instructions of CPUs, depending on the available hardware. This work started in ROOT 6.20, where auto-vectorizable implementations of many PDFs were introduced [3]. A dedicated computation library was designed based on this work, targeting CPUs, vector instructions, multithreading, and GPUs. An overview of this library will be given, illustrating how one can reuse the same code for both GPU and CPU computations and showing the necessary steps to implement user PDFs. Subsequently, new Python-exclusive RooFit features are presented. Finally, we will highlight other new RooFit features in the upcoming ROOT release.

2. The `RooBatchCompute` library

In ROOT 6.24, the vectorizable implementations of PDFs were transferred to a dedicated library called `RooBatchCompute`. This allows for compiling the same code multiple times, for example, for the SSE or AVX(1,2,512) instruction sets and using CUDA for GPU support. When the RooFit library is loaded, the available hardware is inspected, and `RooBatchCompute` with the fastest supported vector instruction set is loaded, and the CUDA version, if supported.

[1] Probability Density Functions

The `RooBatchCompute` library supports ROOT's implicit multithreading mode, in which case computations are distributed among the available CPU threads for data-level parallelism. As RooFit ships with multiple versions of the `RooBatchCompute` library, users can fully benefit from their hardware regardless of how ROOT was compiled or installed on their system. To use the optimal computation backend, users pass the argument `BatchMode("cpu")` or `BatchMode("cuda")` to the `RooAbsPdf::fitTo()` function.

The computation functions of custom user-implemented PDFs can also be easily transferred to `RooBatchCompute`. By following a specific signature, RooFit can centrally provide input data to each computation function, so users only need to implement the computation kernel, which will be compiled multiple times for different architectures, and with and without multithreading.

Since the `RooBatchCompute` library only comprises computation functions but no logic to manage a computation graph, a new class, `RooFitDriver`, was introduced in ROOT 6.26. It analyzes the computation graph, handles memory, submits computation kernels (CUDA or CPU), and synchronizes the intermediate results. The new class bypasses the old code path that heavily relied on caching of intermediate results in the fields of RooFit objects and thus enables a thread-safe way to evaluate PDFs. The old code path was kept as default, but the goal is to make the driver approach the primary way to evaluate RooFit models in the future.

2.1. Accelerating with GPUs

The improvements previously described were prerequisites for enabling GPU computations. Introducing GPU support evolved in four steps, described in the following.

Firstly, the same implementations for vectorized computations can be used in CUDA code by utilizing grid-stride loops. This is a pattern to assign input data elements to CUDA threads to ensure parallel computations on the GPU. C-preprocessor macros are used to compute the correct loop bounds depending on whether a GPU or CPU version of the kernel is compiled. Input data for the computations are provided by the RooFitDriver, which ensures the correct placement of data in device memory.

Secondly, the simple case of evaluating all model components on the GPU was implemented. The data set is entirely placed in the device memory and remains unchanged during the fitting process. All the intermediate results remain on the device, and only the final result is returned to the host. In this case, one can benefit from concurrent kernel execution by asynchronously launching computations that do not depend on each other. RooFitDriver uses CudaStreams and CudaEvents for synchronizing the results and managing the dependencies.

In a third step, mixed CPU/GPU computations were implemented, since the assumption of exclusive CUDA computations is not realistic: 1) not all of RooFit's computations have a CUDA implementation and 2) computations of custom user-made PDFs or formulas which have not been transferred to the `RooBatchCompute` library need to be supported. When CUDA and CPU computations are mixed, the need for data transfers between host and device memory arises, and in turn, more calculations might better be done on the CPU to avoid copying overhead. RooFitDriver solves this by using asynchronous device-to-host transfers and by tracking the progress of every computation or copying running at the moment. This information permits the dynamic scheduling and distribution of the computation tasks to minimize idle time.

In a fourth step, higher data-level parallelism was achieved by independently computing model components. For the moment, RooFit only calculates up to two different components concurrently (without counting concurrent kernel execution), assuming that the CPU can do a part of the work while the GPU is computing some other component. An optimal allocation of the computations between the CPU and the GPU is hard to predict, as it depends on many factors, such as problem size and available hardware. Since each iteration of the minimizing algorithm involves the same computations with altered values of model parameters, RooFitDriver measures the execution times of every component of the model in CPU and (if possible) in GPU in the first two iterations, with negligible overhead. Using these time measurements, a greedy algorithm tries to minimize the idle time of the hardware by assigning each computation to the device that has the highest performance for the given problem. In the future, the use of multiple GPUs can be investigated to compute more independent components in parallel.

2.2. Benchmark results

The new batch mode was benchmarked with several unbinned fit configurations. Table 1 shows the run times for RooFit's classic scalar computation mode and the new batch mode with vectorization, multithreading or GPU offloading. In Figure 1, the speedups relative to the scalar mode are visualized. In the single-threaded batch mode on the CPU, a speedup up to 10 times is observed, while on the GPU, the speedup can reach 50 times. The CPU-related speedup is a result of faster data loading, better cache locality, and AVX2 computations, which can process four double-precision numbers per instruction. The GPU results were obtained on a gaming GPU, which has much fewer double-precision registers than data-center-grade GPUs, so we expect further speedups with GPUs designed for scientific calculations. Even though the multithreaded CPU mode uses all the 24 threads available on the system, the speedup does not scale proportional to the number of threads, as each thread only receives a relatively small amount of work when fitting 1 million events. We hope to improve this in the future and also add benchmarks for binned fits.

There are two caveats in these benchmarks: in the batch mode, the Kahan summation for adding up the log-likelihoods is disabled, and RooFit's recovery from invalid parameters is switched off. Implementing these features also in the batch mode will slow it down slightly, but these can be vital for fit convergence.

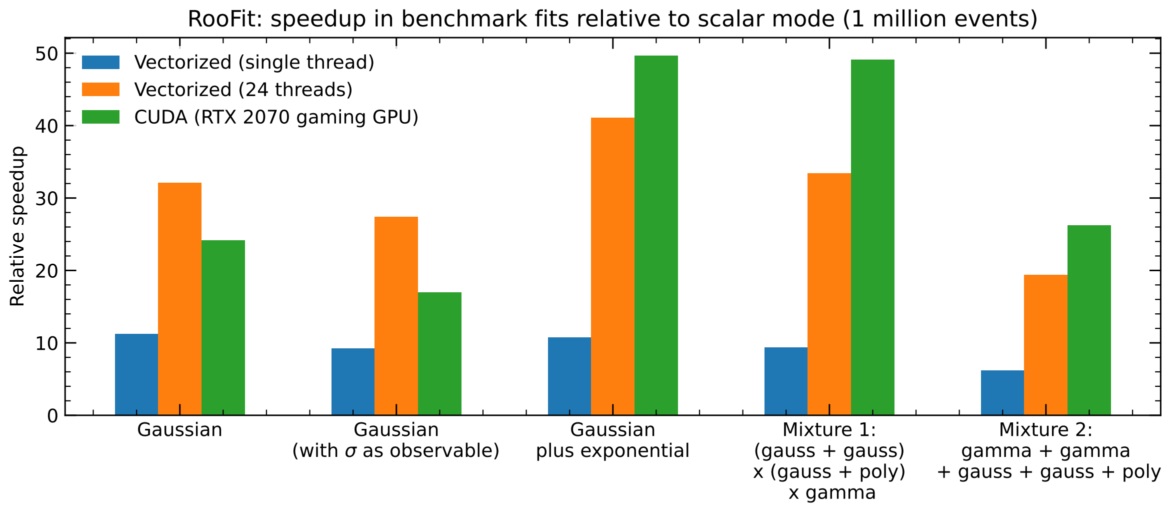
Table 1. Run times for several unbinned fit benchmarks using an AMD Ryzen 9 3900 12-Core Processor (24 threads), and an NVIDIA GeForce RTX 2070 GPU. The symbol g signifies Gaussian distributions, Γ are gamma distributions, and p are polynomials. In the benchmarks with multiple PDFs, each PDF is multiplied by a coefficient that is a fit parameter.

Benchmark (1 million events)	Scalar mode	Vectorized (single thread)	Vectorized (multithread)	CUDA (RTX 2070)
Gaussian	2632 ms	234 ms	82 ms	109 ms
Gaussian (both x and σ as observables)	1069 ms	116 ms	39 ms	63 ms
Gaussian plus exponential	9784 ms	908 ms	238 ms	197 ms
Mixture 1: $(g_1(x) + g_2(x)) \cdot (g_3(y) + p(y)) \cdot \Gamma(z)$	112 s	12 s	3.35 s	2.28 s
Mixture 2: $\Gamma_1(x) + \Gamma_2(x) + g_1 x + g_2(x) + p(x)$	93 s	15 s	4.80 s	3.55 s

3. A more pythonic RooFit

As part of the ROOT framework, RooFit comes with Python bindings that are generated automatically using the cppyy [5] package and ROOT's C++ interpreter cling. This is called PyROOT [4]. Owing to these, every C++ function or object part of the ROOT framework is reachable from Python. This, however, may lead to Python code that looks and feels like

Figure 1. A visualization of the speedup with the new vectorized RooFit evaluation mode corresponding to the results in Table 1, relative to the old scalar mode.



C++. Any class can be "pythonized", though, via a pythonization hook in cppyy. That is, an interface exclusive to the Python side can be designed, which takes care of managing the details of the C++ side appropriately. This allows for writing code that much more looks and feels like Python, while C++ is executed at the backend.

With Python becoming increasingly popular, the RooFit Python bindings were modernized. For RooFit, a higher-level abstraction of the cppyy pythonization feature was designed. In the backend, each RooFit class now has a pure Python *mirror class*, whose member functions will dispatch to their C++ equivalents. This makes it easy for the developer to define both pythonizations of existing functions and new member functions in one place, to allow for customizations that improve the user experience on the Python side. In the following, we will present the most relevant pythonizations, as well as new functions that were added for improved interoperability of the RooFit dataset classes with the Python ecosystem.

3.1. Pythonizations

Various new pythonizations are introduced to streamline RooFit code in Python. Some notable highlights are:

- **Implicit conversion from Python to RooFit collections**: Users can now benefit from implicit conversion from Python lists to `RooArgList`, and from Python sets to `RooArgSet`. When the Python collection contains numbers, they are implicitly converted to `RooConst`.

  Example: without pythonizations...

  ```
  pdf = ROOT.RooPolynomial(
      "p", "p", x, ROOT.RooArgList(
          ROOT.RooConst(0.01),
          ROOT.RooConst(-0.01),
          ROOT.RooConst(0.0004)))
  pdf.generate(
      ROOT.RooArgSet(x, y, cut), 10000)
  ```

  ...with new pythonizations in ROOT 6.26

  ```
  pdf = ROOT.RooPolynomial("p", "p", x,
      [0.01, -0.01, 0.0004])
  pdf.generate({x, y, cut}, 10000)
  ```

- **Keyword argument pythonizations**: all functions that take RooFit command arguments as parameters now accept equivalent Python keyword arguments, for example simplifying calls to `RooAbsPdf::fitTo()`.

Example: without pythonizations...

```
result = pdf.fitTo(data,
            ROOT.RooFit.Range("r1"),
            ROOT.RooFit.Save())
```

...with new pythonizations in ROOT 6.26

```
result = pdf.fitTo(data,
                    Range="r1",
                    Save=True)
```

- **String to enum pythonizations**: many functions that take an enum as a parameter now accept also a string with the enum label.

  Example: without pythonizations...

```
data.plotOn(frame,
    ROOT.RooFit.DataError(
        ROOT.RooAbsData.SumW2))
```

  ...with new pythonizations in ROOT 6.26

```
data.plotOn(frame, DataError="SumW2")
```

- **Allow for the use of Python collections instead of C++ STL containers**: some RooFit functions take STL map-like types such as `std::map` as parameters. Until now, users had to create the correct C++ class in Python, but now they can usually pass a Python dictionary instead. For example, a `RooCategory` can be created like this:

```
ROOT.RooCategory("sample", "sample", {"sig": 1, "bkg1": 2, "bkg2": 3})
```

- **RooWorkspace accessors**: in Python, one can now get objects stored in a `RooWorkspace` with the square-bracket operator, automatically cast to the correct type. Users don't have to use type-specific functions such as `var()` for variables or `pdf()` for PDFs, instead one can retrieve any object using square brackets.

3.2. New PyROOT functions for interoperability with the Python ecosystem
RooFit has two distinct dataset classes: `RooDataSet` for unbinned data, and `RooDataHist` for binned data. These dataset classes are usually created either from other ROOT data structures like TTree, or filled either manually or using `RDataFrame`. With ROOT 6.26, new Python-exclusive functions are added to import or export NumPy arrays. A RooDataSet is mapped to a Python dictionary of NumPy arrays, where the dictionary keys are the variable names. When importing from NumPy, the name for the weight variable can be specified. Because of the tabular nature of RooDataSet, there are also functions for conversion to Pandas dataframes and back. For RooDataHist, the import/export was inspired by `numpy.histogramdd` [6]: it can be converted to a (multidimensional) array of histogram counts and a list of bin edge arrays. Another new Python-exclusive function is `RooRealVar.bins`, which returns the bin edges of a RooFit variable as a NumPy array.

4. Other new RooFit features and developments
Besides the new vectorized evaluation mode with GPU support and the new PyROOT features, there are other noteworthy RooFit features added in ROOT 6.26.

4.1. Parallelized gradient calculation
For models with many parameters, most of the fitting time is spent for the numeric gradient computation, as the model has to be re-evaluated after changing the parameters one at a time. Hence, distributing the gradient calculation over multiple processes is a general way to parallelize fitting [7]. With ROOT 6.26, the code for this parallel gradient calculation is included in the release. The necessary classes to build and minimize likelihood objects are available to advanced users. However, there is yet no high-level user interface to enable it (i.e., a way to switch it on in `RooAbsPdf::fitTo`). This will be part of the next release, for which it is also foreseen to combine the parallel gradient calculation with the vectorized evaluation in a single code path.

4.2. Experimental JSON import/export for RooFit workspaces
RooFit now implements serialization and deserialization of `RooWorkspace` objects to and from JSON and YML. For now, this functionality is not feature complete with respect to all available functions and PDFs available in RooFit, but provides an interface that is easily extensible by users [8]. So far, the JSON conversion is experimental and development will continue in the future with the help of user feedback.

4.3. Creating RooFit datasets from RDataFrame
RooFit now contains two RDataFrame action helpers, called `RooDataSetHelper` and `RooDataHistHelper`, which allows for creating RooFit datasets directly from RDataFrames.

4.4. Global observables in RooFit datasets
RooFit groups model variables into *observables* and *parameters*. For fits with parameter constraints, there are additional *global observables*. These represent the results of auxiliary measurements that constrain nuisance parameters. Prior to ROOT 6.26, the global observable values were always taken from the model/pdf. With this release, a mechanism is added to store a snapshot of global observables in any RooDataSet or RooDataHist. This simplifies toy studies, where the global observables have to be varied in each iteration, as the bookkeeping of global observable values is done internally.

5. Conclusions
In this paper, the new RooFit features arriving with ROOT 6.26 were presented. In the spotlight was the updated vectorized RooFit interface (aka. batch mode). The computations are now offloaded to a new library that also supports GPU computations with CUDA. With the vectorized CPU version, one can get up to 10 times faster fits, while with the CUDA version, the speedup can be as high as 50 on a gaming GPU. The new RooFit Python features were presented in detail as well, showcasing both new pythonic wrappers around existing functions (pythonizations), and new Python-exclusive functions for better interoperability with the scientific Python ecosystem. Finally, the other new RooFit features that are expected to improve the user experience were presented.

Future developments will focus on improved user experience and automatic differentiation in RooFit, which is expected to result in significant speedups for many-parameter fits. With three attack vectors on the optimization front (vectorized computations, gradient parallelization, and automatic differentiation), RooFit will be readied for the HL-LHC analysis demands, while the user experience is continuously improved, especially on the Python side.

References

[1] W. Verkerke, D. Kirkby *The RooFit toolkit for data modeling*, eCONF, CHEP 2003, PSN MOLT07

[2] R. Brun, et al., *ROOT: An object oriented data analysis framework*, Nucl. Instrum. Methods Phys. Res. A389 81–86 (1997)

[3] S. Hageboeck, *What the new RooFit can do for your analysis*, Proceedings of Science, ICHEP2020 (2021), [doi:`10.22323/1.390.0910`]

[4] M. Galli, et al., *A New PyROOT: Modern, Interoperable and More Pythonic*, EPJ Web Conf. 245 06004 (2020)

[5] W. Lavrijsen and A. Dutta, *High-Performance Python-C++ Bindings with PyPy and Cling*, 2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC), [doi:`10.1109/PyHPC.2016.008`]

[6] *NumPy Manual - `numpy.histogramdd`*, `https://numpy.org/doc/stable/reference/generated/numpy.histogramdd.html#numpy.histogramdd`, Accessed: 2022-02-20

[7] E. G. P. Bos et al., *Faster RooFitting: Automated parallel calculation of collaborative statistical models*, EPJ Web Conf. 245 (2020), [doi:`10.1051/epjconf/202024506027`]

[8] *RooFit HS3 - README*, `https://github.com/root-project/root/blob/master/RooFit\protect\let\futurelet\@let@token\let\let\@xspace@maybespace\relax/hs3/README.md`, Accessed: 2022-02-20