Optimizing weight evaluations for convolution fits in RooFit using vectorization

Hanna Olvhammar
hanna.olvhammar@outlook.com

August 26, 2022

Abstract

Convolutions play an important role in high energy physics, since they are used when fitting probability density functions (PDFs) to data. When fitting convolutions of PDFs to data with RooFit, the PDFs are convoluted with a discrete fast Fourier transform and the result is stored in a histogram that is interpolated at its bin centres. This report concerns optimizations of weight evaluations in RooFit in the case of one dimensional histograms with up to second order interpolations. By implementing vectorized versions of the evaluation functions, this project resulted in approximately 2-5 times faster weight evaluations, depending on interpolation order and histogram properties. A suggested continuation of the project is to implement vectorization for the convolution operations.

1 Introduction

ROOT is a data analysis framework widely used by the experiments at CERN [1]. While it was created for high energy physics, ROOT also continues to be a powerful tool for physicist in many disciplines all over the world. RooFit is the part of ROOT concerned with fitting PDFs to data acquired from experiments. Since observables in HEP experiments are often sensitive to different effects, such as the underlying physics and detector resolution, the final PDF of an observable is often the convolution of several PDFs.

For the function that fits a PDF to data, two steps are repeated during the minimization; a convolution operation between two PDFs, and an interpolation between bin centres of the histogram resulting from the convolution. Since these steps are repeated many times in the fitting process, increasing the speed of any of these steps would significantly increase the speed of not only the fitting function, but any RooFit operation that uses these elements.

The aim of this project is to increase the speed of fitting a convolution to data using vectorization. With vectorization, the functions are more prepared to be ported to GPUs in the future. However, vectorization in itself also improves the speed immensely even using only CPU back-ends, since there are vector instructions on modern CPUs. This project concerns the interpolation part of the fitting, specifically speeding up the interpolation functions with vectorization.

2 Fitting a convolution

The convolution operation in RooFit is performed using the class `RooFFTConvPdf`. This class implements the fast Fourier transform (FFT) to evaluate a convolution, using that

$$\mathscr{F}[g(x) * h(x)] = G(k) \cdot H(k). \tag{1}$$

Since the fast Fourier transform is discrete, the number of sampling points for the FFT needs to be provided when performing the convolution. The result of the convolution is stored in a histogram object with the number of bins being the number of sampling points. To ensure better results when using minimization to find parameters that better fit the data, the histogram is made into a smoother PDF using interpolation between bin centres.

The user can choose three levels of interpolation: zeroth order interpolation, which means the histogram is unchanged; first order interpolation, where a straight line is drawn between each bin centre; second order interpolation, where the positions and weights of three neighbouring bins are used to create a quadratic function between two bin centres.

To determine which functions were most worth improving the speed of, the whole fitting procedure was profiled.

3 Profiling the fitting function

The starting point was to study the convolution tutorial in RooFit [2]. In the tutorial, a Gaussian and a Landau PDF are convoluted and observable values are generated using the convolution. Then the

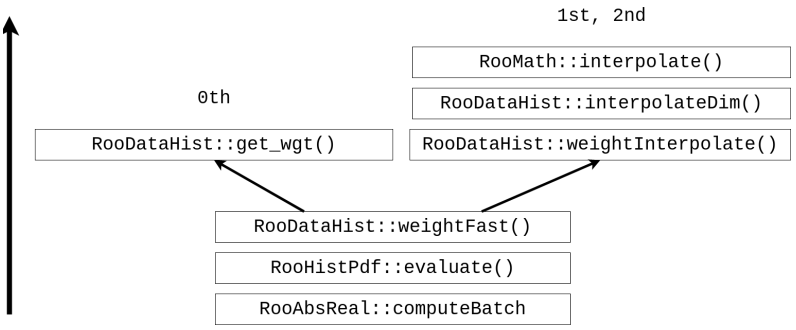

Figure 1: When using the function `RooAbsReal::computeBatch()`, the call stack is high and the evaluation of weights is time consuming. It calls `RooHistPdf::evaluate()` in each iteration of a for-loop over events. From `evaluate()`, the weight for a single event is evaluated in several steps depending on whether zeroth, first or second order interpolation is used.

function `RooAbsPdf::fitTo()` is called to fit the convolution to the generated data.

To identify what areas of the fitting process could be made faster, the `fitTo()` function was analysed in depth. First, the number of events generated was varied while the number of sampling points used for the convolution was constant. The execution time for `fitTo()` as a function of the number of events was measured for two different back-ends: An older, well tested CPU back-end and a more current, optimized CPU back-end (the latter uses batch evaluation, and is enabled with the `BatchMode` option in `fitTo()`) [3, 4]. The results are shown in Figure 2.

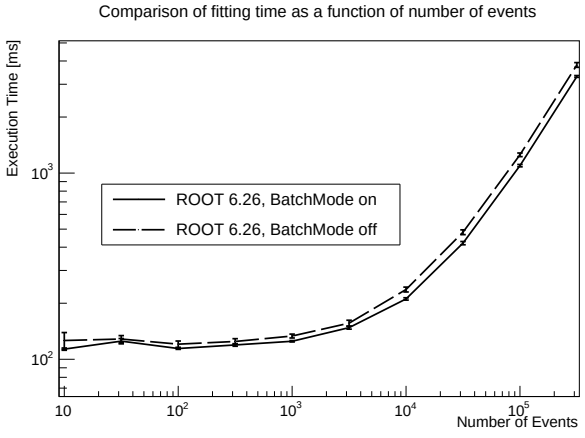

Figure 2: The execution time for the `fitTo()` function was measured for the ROOT 6.26 version with two different CPU back-ends. The back-end with the batch mode enabled is slightly faster. There is a constant time caused by the FFT and an increasing time with the number of events that is caused by the interpolation.

In the figure there is a constant part of the execution time that originates from the constant number of sampling points used for the FFT. Furthermore, the execution time increases when the number of events increases. This is a consequence of the interpolation between bin centres.

The next step in analysing the fitting process was to generate a flame graph , where the execution time of each function is shown on the horizontal axis of the flame graph and the call stack is shown on the vertical axis [5]. The flame graph revealed that the function `RooAbsReal::computeBatch()`, which evaluates the weights after the convolution, should be investigated for speed improvements. To measure the execution time for evaluating weights, `RooAbsPdf::getValues()` was called from a separate script since it uses `RooAbsReal::computeBatch()` for weight evaluation.

4 Vectorizing weight evaluations

In ROOT version 6.26, weights were evaluated in `RooAbsReal::computeBatch()`. This function evaluates the weight for a single observable value using the histogram produced in the convolution. The weight of one value is evaluated using a chain of function calls visualised in Figure 1. This means that in `RooAbsReal::computeBatch()`, these functions are all called in each iteration of a for-loop that fills an array of weights.

In a vectorized implementation of the weight evaluation, the weights for all events are calculated by passing vectors of observable values and weights through a series of functions and eventually evaluating all weights in a for-loop over events. To this end, a new function `RooHistPdf::computeBatch()` was implemented to override `RooAbsReal::computeBatch()` in the case of one dimensional histograms.

In this project, the new function `RooHistPdf::computeBatch()` was implemented.

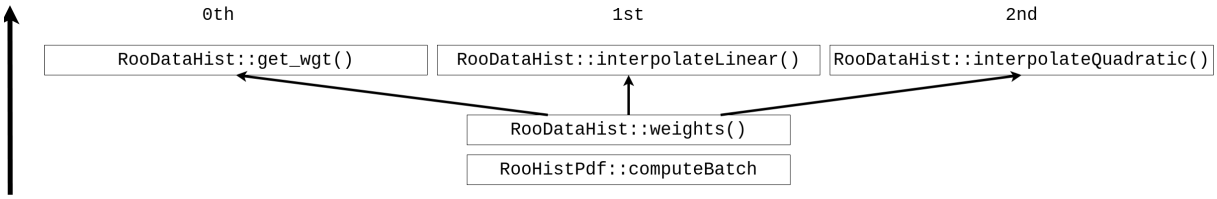

Figure 3: In contrast to `RooAbsReal::computeBatch()`, `RooHistPdf::computeBatch()` calls `RooDataHist::weights()` and passes the whole output vector. Depending on the order of the interpolation, weights are then explicitly evaluated in a for-loop over events in either of the three available functions.

It calls `RooAbsReal::computeBatch()` for histograms of dimension two or higher, but implements vectorized weight evaluation for zeroth, first and second order interpolation for one dimensional histograms. The interpolation concept is illustrated in Figure 4. These three orders of interpolation were implemented in three different merges to the ROOT developer branch in Git [6, 7, 8]. The structure of the weight evaluation in the one dimensional case is represented by Figure 3 and can be compared to Figure 1.

In this implementation of the weight evaluation, the cases of uniform bins and non-uniform bins were distinguished. Using that the bin widths and bin centre coordinates can be found directly in the case of uniform bins, the uniform binning case is faster than the non-uniform binning case in the vectorized implementation.

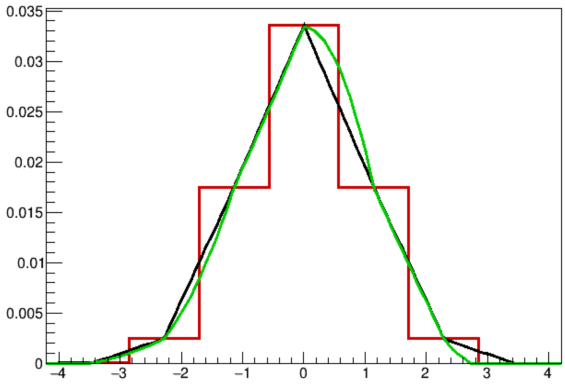

Figure 4: An illustration of the interpolation concept using a perfect Gaussian distribution. The red histogram is used for zeroth order interpolation, the black lines result from first order interpolation and the green curves result from second order interpolation. The quadratic interpolation performs at least as well as the linear interpolation.

4.1 No interpolation

The case of no interpolation was implemented explicitly in the new function `RooDataHist::weights()` that uses an array of observable values passed from `RooHistPdf::computeBatch()`. To evaluate the weights, the bin number corresponding to a certain observable value is needed. When the corresponding bin number is found, the bin contents can be used for weight evaluation. When no interpolation is performed, the returned weight is simply the bin contents for the bin corresponding to the value. Therefore, the implementation consisted in first finding an array of bin indices corresponding to the array of observable values, and then using a `RooDataHist` member function to get the corresponding bin contents.

4.2 First order interpolations

When evaluating the weight for an observable value using first order interpolation, `RooDataHist::weights()` calls the new function `RooDataHist::interpolateLinear()` instead of evaluating the weights explicitly.

In `RooDataHist::interpolateLinear()`, it is first determined whether the value is to the left or the right of the bin centre in the corresponding bin. If the value is to the left of the bin centre, its weight is given by the corresponding value of a straight line fitted between the bin centre to the left and the bin centre in the current bin. If it is to the right of the bin centre, a straight line is fitted between the current bin and the bin on the right instead.

The straight line is found by solving the coefficients in a system of two linear equations, where the two equations are evaluated at the neighbouring bin centres respectively. The coefficients can then be used together with the observable value to evaluate the interpolated weight.

There are two special cases for first order interpolation, namely interpolation close to the two boundaries of the histogram. The borders can be handled in two different ways. If the histogram describes a cumulative distribution function, a straight line will be fitted between the leftmost boundary with weight zero and the first bin centre, as well as between the last bin centre and the rightmost boundary with weight one. If the his-

Table 1: The average execution time for `RooAbsPdf::getValues()` (averaged over 100 function calls) was found for the three different interpolation orders as well as in the case of uniform binning and non-uniform binning. The time was then compared for ROOT version 6.26 and the same version with the new vectorized functions. The weight evaluation was made for 10 million events and 100 sampling points.

Implementation	Bin width	ROOT 6.26 [ms]	Vectorized [ms]	Times faster
No interpolation	Uniform	630	200	3.2
	Non-uniform	970	520	1.9
Linear interpolation	Uniform	1090	250	4.4
	Non-uniform	1510	570	2.6
Quadratic interpolation	Uniform	1160	270	4.3
	Non-uniform	1620	580	2.8

togram does not describe a cumulative distribution function, the first and last bins are mirrored outside the histogram boundaries and are used for the linear interpolation.

It was tested whether allocating an array containing the coefficients of each straight line between the bin centres would be faster, where the elements of this array were called depending on which bin the observable value was contained in. However, that resulted in a negligible increase in speed for 10 million events and 100 sampling points and might even be slower for a smaller bin number to event number ratio.

4.3 Second order interpolations

In the case of second order interpolation, `RooDataHist::weights()` calls the new function `RooDataHist::interpolateQuadratic()`.

The method of quadratic interpolation is very similar to that of linear interpolation; the coefficients in a system of three quadratic equations are found. For an observable value between two bin centres, two bin centres to the left and one bin center to the right of the observable value are used for the interpolation. In the same way as for linear interpolation, the weight is then found by evaluating a quadratic equation using the coefficients and the observable value.

The case of mirroring is easily extended to the quadratic case. In the case of the cumulative distribution function, the same points and weights are used for the interpolation as in the linear case, with the extra condition that the boundary point is the minimum of the quadratic function that is evaluated.

5 The total increase in speed

To determine the effect of the vectorized implementations on the time for evaluating weights of one dimensional histograms, the time for executing `RooAbsPdf::getValues()` was measured and averaged over 100 executions. The execution time was studied for all three interpolation cases both for uniform and non-uniform binning, and the time for ROOT version 6.26 and the vectorized implementation was compared for 10 million events and 100 sampling points. The difference between the two versions is that `RooAbsPdf::getValues()` calls `RooAbsReal::computeBatch()` in the older version (see figure 1), and `RooHistPdf::computeBatch()` in the new implementation (see Figure 3).

In Table 1, the averaged execution time is presented as well as a summary of how many times faster the new implementation is than the older version. It is clear that the vectorized implementation is several times faster than the older implementation, especially in the case of uniform bins.

Furthermore, a comparison between versions and back-ends for an increasing number of events was performed. In Figure 5, the dashed lines represent the execution time for the ROOT 6.26 version and the new implementation when the new CPU batch mode is turned off, and the filled lines represent the time for when the batch mode is enabled.

The figure shows two important results. Firstly, the vectorized implementation with the batch mode enabled is significantly faster than the other three cases. In this case, the execution time is not sensitive to the interpolation calculations for up to 10^4 events. For more than 10^4 events, the vectorized version with the batch mode on is much faster than the other implementations. Secondly, the figure confirms that the new batch mode is essential for fast calculations on the CPU,

6 Further optimizations

This project was concerned with speeding up weight evaluations for one dimensional histograms using vectorizations. A natural extension of this project is to implement the vectorization for higher dimensional histograms. It would be interesting to compare a GPU implementation of these vectorized functions, but it might be even more beneficial to continue optimizing the existing CPU implementations in other parts of the fitting procedure.

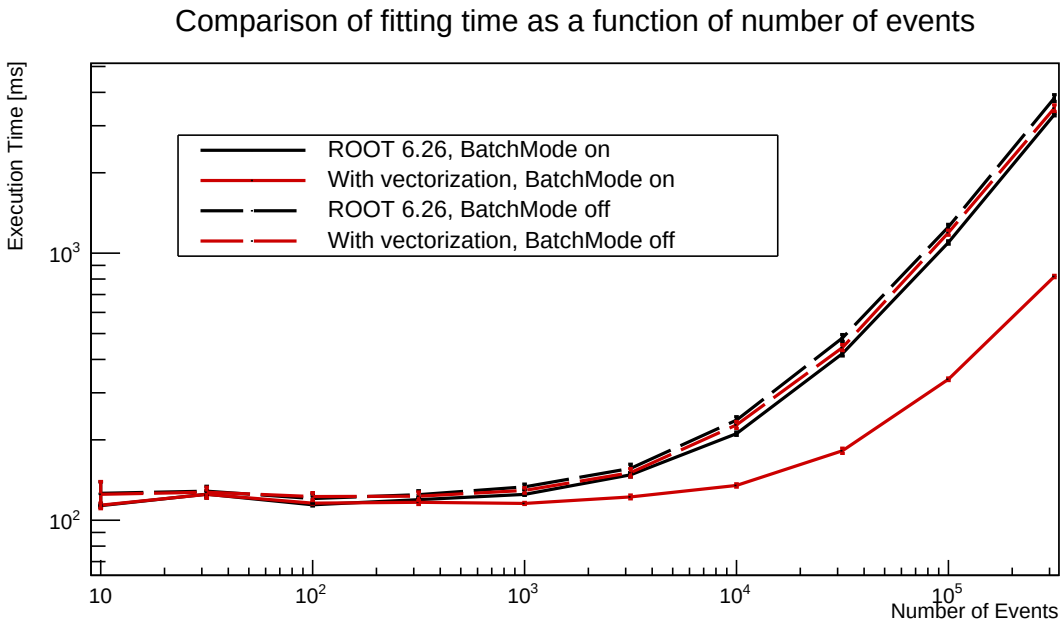

Figure 5: The execution time for an increasing number of events was measured for four different versions of `fitTo()`. It is clear that the vectorized version together with the new CPU back-end is much faster than the other three versions. The vectorized interpolation needs about ten times more events than the other versions before it impacts the total execution time significantly.

In the beginning of this project, flame graphs and results such as Figure 2 were used to identify where further optimizations can be performed. From Figure 2 we noted that while the interpolation caused the increase in execution time for an increasing number of events and constant number of sampling points, while the FFT caused the large time constant. It would therefore be interesting to study how the execution time behaves for a constant number of events and an increasing number of sampling points, and then vectorize time consuming functions used in `RooFFTConvPdf`.

References

[1] *ROOT website*. URL: `https://root.cern.ch/`.

[2] *RooFit convolution tutorial*. URL: `https://github.com/root-project/root/blob/master/tutorials/roofit/rf208_convolution.C`. (accessed: 24.08.2022).

[3] Jonas Rembser et al. *Accelerating RooFit with GPUs*. URL: `https://indico.cern.ch/event/855454/contributions/4596763/`.

[4] Zef Wolffs et al. *New RooFit developments to speed up your analysis*. URL: `https://indico.cern.ch/event/855454/contributions/4596763/`.

[5] *Flame graph resources*. URL: `https://www.brendangregg.com/flamegraphs.html`. (accessed: 24.08.2022).

[6] *Code contribution: Vectorize weight evaluation with `RooDataHist::weights()`*. URL: `https://github.com/root-project/root/pull/11171`.

[7] *Code contribution: Enable vectorized first order interpolation for weight evaluation*. URL: `https://github.com/root-project/root/pull/11211`.

[8] *Code contribution: Enable vectorized second order interpolation for weight evaluation*. URL: `https://github.com/root-project/root/pull/11224`.