



GPU Optimizations for HEP Analysis in ROOT

Kevin Nobel (University of Amsterdam)

Supervisors: Monica Dessole (CERN) & Jolly Chen (University of Twente)

Keywords: GPU, CUDA, ROOT, Histogram, High Energy Physics Analysis

Abstract

This project focuses on improving the runtime performance of High Energy Physics (HEP) analyses by leveraging GPUs, which are increasingly common coprocessors used to accelerate computationally intense operations. Histogramming is a core operation in HEP analysis, however the current GPU implementation is bounded by memory transfers. Hence, we implemented a generic batch histogramming kernel to fill multiple histograms at the same time. In future work, this implementation can be used to improve data reuse. Additionally, we explored offloading define actions to the GPU, which showed the potential for significant performance gains, up to speedups of $100\times$ compared against equivalent multi threaded CPU implementations.

Contents

1	Introduction	2
2	Background	2
2.1	Events	2
2.2	Filters and Defines	2
2.3	Histogramming	3
3	Batch Histogramming	3
3.1	Implementation	3
3.2	Results	5
4	Offloading Define Actions to GPUs	6
4.1	DiMuon Analysis	6
4.2	Folded W Mass	8
5	Future Work	8
6	Conclusion	10

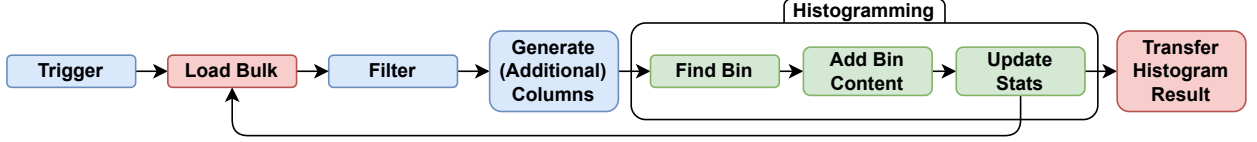


Figure 1: An abstract overview of a typical HEP analysis

1 Introduction

At CERN, all Large Hadron Collider (LHC) experiments produce petabytes of data that require efficient analysis to study the fundamental theory of physics. With the upcoming upgrade to the High Luminosity LHC, the volume of data is expected to increase significantly [3]. This surge in data demands an improvement in runtime performance to keep up with the pace of research. Furthermore, faster analysis will ultimately result in enhanced scientific output of physics experiments.

ROOT[1] is an open-source C++ data analysis framework widely used in High Energy Physics (HEP) to efficiently analyze event data. A typical HEP analysis in ROOT (see figure 1) generally consists of filtering data and defining new columns of data based on existing columns followed by computing histograms.

In this project we focus on utilizing GPU acceleration with CUDA to explore potential performance improvements in HEP analysis. GPUs are increasingly common coprocessors used to accelerate computationally intense operations. Specifically, we target the typical HEP analysis described previously. In this experiment, we achieve a speedup of over $100\times$ compared to an equivalent multithreaded CPU implementation. While these optimizations are currently focused on a specific scenario, our long-term goal is to generalize the approach to potentially benefit a wider range of analyses.

The code produced for this project is published on GitHub¹.

2 Background

2.1 Events

In ROOT, an event is a data object typically used to represent data from HEP experiments. Events correspond to individual experimental observations, such as a particle collision. An event is comparable to a row in a tabular data format. However, as opposed to a flat table, each field can contain multi dimensional values.

2.2 Filters and Defines

Filters can be used to exclude unwanted or irrelevant events by applying conditions to each row. These filters are defined either through a boolean statement provided as a string or implemented as a function.

¹github.com/tweska/cern-ssp/

Defines are used to generate new columns based on existing data. In ROOT, the user can specify a define action using a formula in the form of a string or by referencing a function. For example, a define action can be used to compute the invariant mass of particles in an event.

By combining filters and defines, users can refine their datasets before finalizing the analysis with histogramming.

2.3 Histogramming

Histogramming is a central operation in many HEP analyses. To fill a histogram with event data, we must first determine the appropriate bin to be filled based on the input coordinates of each event. Once the bin is identified, it is incremented by a specified weight, or by a default value of one if no weights are provided.

The method of finding the correct bin depends on how the bin edges are distributed over the axis. In case of a fixed bin axis, the formula below is used to calculate the bin based on the limits of the axis and the number of bins. When variable bin sizes are chosen, a binary search is used to locate the appropriate bin, as the bins might not be evenly spaced.

$$FindBin(x) = \begin{cases} 0, & \text{if } x < x_{min} \\ n_{bins} + 1, & \text{if } x \geq x_{max} \\ 1 + \left\lfloor n_{bins} \times \frac{x - x_{min}}{x_{max} - x_{min}} \right\rfloor, & \text{otherwise} \end{cases}$$

In a parallel implementation, special care is required to handle potential race conditions when multiple threads attempt to update the same bin simultaneously.

3 Batch Histogramming

The new approach builds upon an existing GPU-based histogramming kernel [2]. This implementation processes one event per thread. The kernel is launched with bulks of multiple events. The original kernel was designed to handle only one histogram at a time. We introduced modifications that enable processing multiple histograms concurrently in a single kernel, we call this batch histogramming.

The motivation behind these changes was to offer a more efficient solution for the case in which the user wants to compute multiple histograms in parallel, and some of the input data is shared between the histograms. However, we later realized that such a case is relatively rare in practice.

3.1 Implementation

To implement this, we modified the data structure on the GPU to manage multiple histograms in parallel. Figure 2 illustrates the architecture of the GPU data structure. We

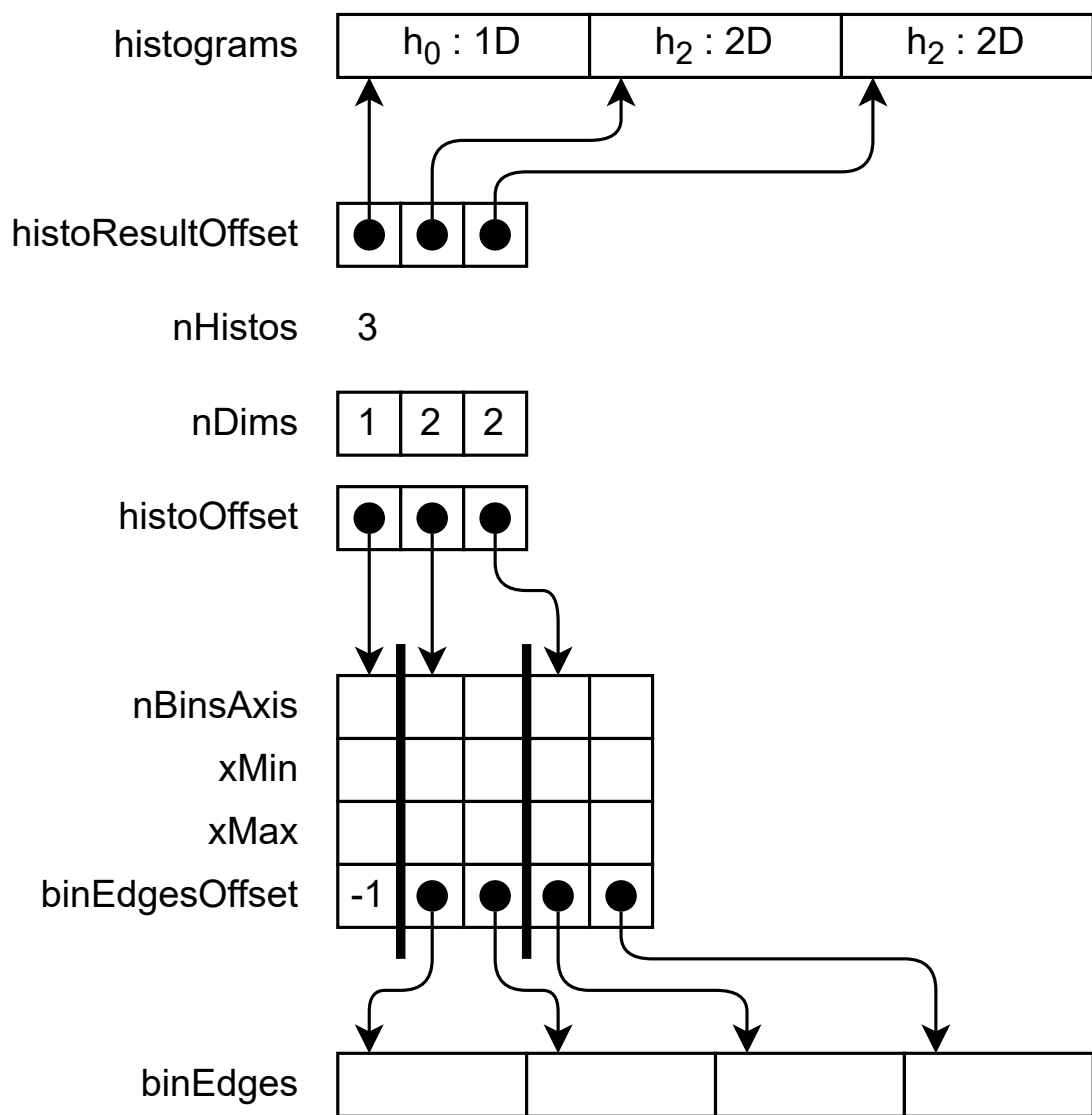


Figure 2: GPU datastructure that enables the batch histogramming

introduced arrays with offsets to efficiently calculate the correct index in each array during histogramming.

Our implementation is encapsulated in a C++ class, providing an interface to interact with. The implementation is compiled with CUDA but any program interacting with the class can be compiled with a normal C++ compiler. The class manages the entire lifecycle of the batch histogramming process as follows:

- **Init** more efficiently use the available hardware realization: When the class is instantiated, the required buffers on both the CPU and GPU will be allocated and initialized.
- **Filling histograms**: A fill method can be called multiple times to launch the histogramming kernel. The method will first transfer the data before launching the kernel.
- **Retrieving results**: At any time, the resulting histograms can be retrieved from the GPU back to the CPU.
- **Deletion**: When the class is destroyed, the buffers allocated in the initialization are automatically freed.

The final implementation is capable of handling multiple histograms with mixed dimensions, axis sizes and both fixed and variable bin sizes.

3.2 Results

The final implementation is tested on machine with a **AMD Ryzen 7 5700g** CPU and **NVIDIA GeForce RTX 3060** GPU. The code is compiled with the GNU Compiler Collection version 12.3.0 with the `-O3` flag and CUDA Toolkit version 12.5 with the `-arch` flag set to `sm_75`. Our implementation is compared for correctness against **ROOT** version 6.33.01. We report the runtime of an average of 10 executions after a single warmup round. The same experimental setup is used in further experiments.

When filling 3 histograms (1D, and two 2D) with 100 million events, we obtained a speedup of $6.1\times$ over a single-threaded CPU implementation. If we have a closer look at our results in table 1, we see a $281.9\times$ speedup in the fill method. However, we spend 97.8% of the runtime on transferring data from the CPU to the GPU.

	CPU Runtime	GPU Runtime	GPU Percentage	Speedup
Transfer	N/A	543ms	97.8%	N/A
Fill	3383ms	12ms	2.2%	$281.9\times$
Result	N/A	0ms	0.0%	N/A
Total	3383ms	555ms	100.0%	$6.1\times$

Table 1: Runtime performance results for batched histogram implementation.

4 Offloading Define Actions to GPUs

During the implementation of the batch histogramming kernel, we had an observation: while overlapping input data for histogramming is rare, overlapping input data for defines is more common. By shifting the define actions to the GPU, we can potentially reduce the data transfer from the CPU to the GPU while at the same time increasing the arithmetic intensity on the GPU, leading to more efficient utilization of its computational resources. This idea was also presented in a poster presentation².

To explore this optimization, we begin with a simple use-case which creates only a single histogram before moving on to a use-case with many defines and histograms. This allows us to observe how this optimization behaves in different situations.

Note that we do not create generic kernels for these experiments, as this is only a feasibility study for now. A generic solution would require integration with ROOT RDataFrame, which is outside the scope of this project. We are aware that we are investigating a best-case scenario, and expect a generic solution to perform slightly worse due to overhead of the framework.

4.1 DiMuon Analysis

The first experiment focuses on a modified version of the DiMuon analysis from the ROOT tutorials³. In this analysis, a filter is used to find all events with two muons of opposite charge. After filtering, the invariant mass of the two muons is calculated. Once the invariant mass is computed, it is used to fill a histogram.

In the GPU version of the program, we decided to keep the filter on the CPU as more than half of the events are discarded in this step. Moreover, after filtering, the data is very predictable as there are always exactly 2 muons described in the event. After transferring the data to the GPU, the invariant mass is calculated and the histogram is filled with the mass value on the GPU. The program is visualized in figure 3a.

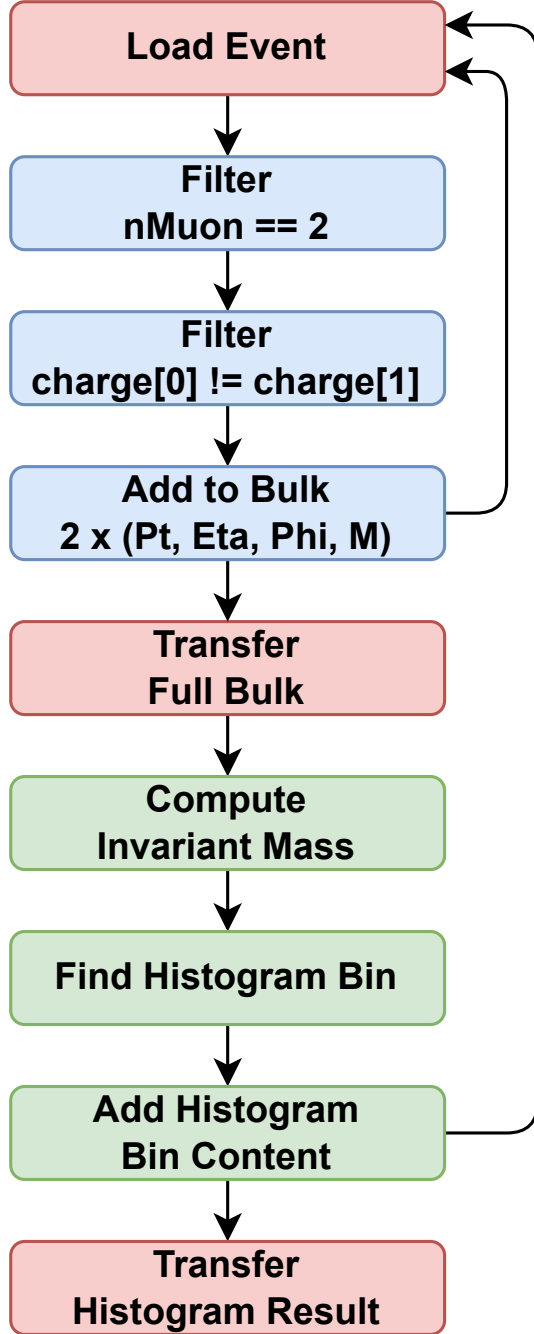
In this experiment we obtained a speedup of $2.8\times$ over a multi-threaded (16 threads) CPU implementation. The transfer time is reduced to 58.3% of the total runtime. The full results can be found in table 2.

	CPU Runtime	GPU Runtime	GPU Percentage	Speedup
Transfer	N/A	163ms	58.3%	N/A
Fill + Define	788ms	116ms	41.7%	$6.8\times$
Result	N/A	0ms	0.0%	N/A
Total	788ms	278ms	100.0%	$2.8\times$

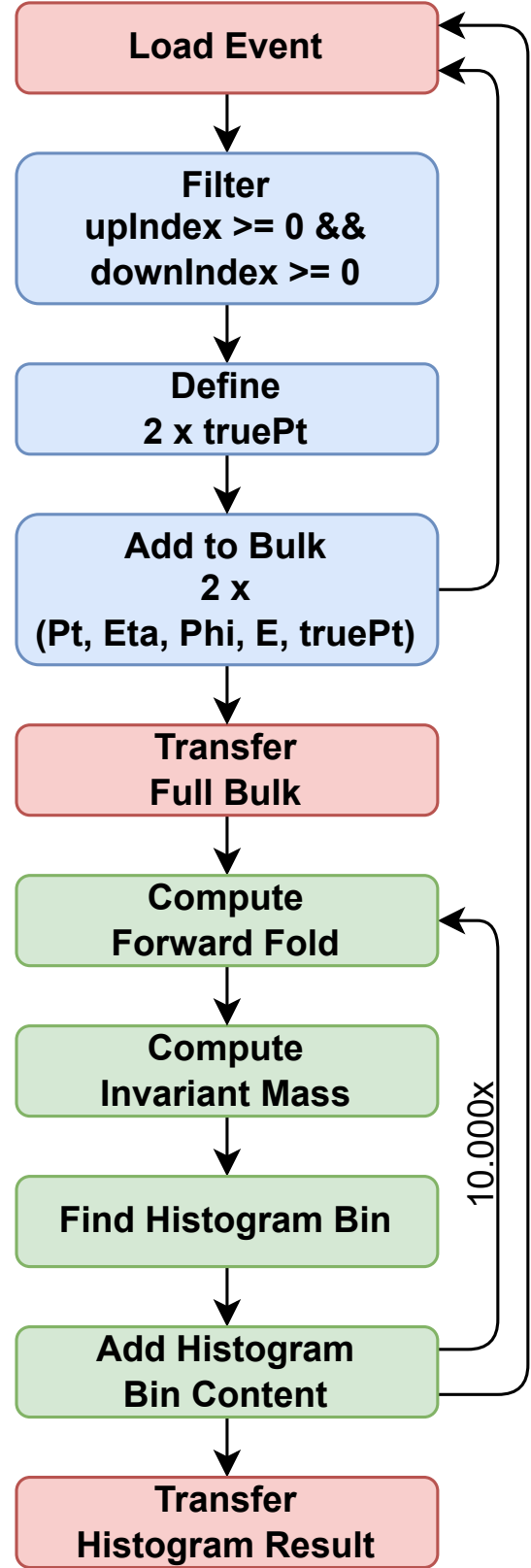
Table 2: Runtime performance results for the DiMuon experiment.

²indico.cern.ch/event/1435014/

³root.cern/doc/master/df102__NanoAODDimuonAnalysis_8C.html



(a) DiMuon



(b) Folded W Mass

Figure 3: Visualizations of the DiMuon and Folded W Mass computations on the GPU.

4.2 Folded W Mass

In the Folded W Mass experiment we look at a more complex use-case involving a large number of define operations. In this use-case, we have 10.000 defines applied on the same input data. There is a very large overlap as each define operates on the same 8 columns.

In the GPU version we filter the data first on the CPU again. To avoid unnecessary transfers we generate two `truePT` values on the CPU. The program operates on two indices in larger arrays. By only transferring the data at the relevant indices, we can significantly reduce the data transfer. For each event we only send 10 floats to the GPU. The program is visualized in figure 3b.

The bulksizes used for the GPU kernel has an impact on the runtime performance. In figure 4 we show the results of an experiment comparing multiple bulksizes. For the final performance experiment we choose a bulksizes of 2^{17} as it yields the highest performance.

Due to the high overlap in data and relatively large amount of computation, the data transfer from the CPU to the GPU is negligible. In our final experiment we obtained a speedup of $102.3\times$ over an equivalent multi-threaded (16 threads) CPU implementation. The full results of this experiment are shown in table 3.

	CPU Runtime	GPU Runtime	GPU Percentage	Speedup
Transfer	N/A	0ms	0.0%	N/A
Fill	17792ms	172ms	98.9%	$103.4\times$
Result	N/A	1ms	0.6%	N/A
Total	17792ms	174ms	100.0%	$102.3\times$

Table 3: Runtime performance results for the Folded W Mass experiment.

5 Future Work

We see a significant potential in generalizing our kernel so that arbitrary filters and defines can be executed on the GPU. This would allow users to apply GPU optimizations to a wider variety of analyses. However, several challenges need to be addressed to realize this.

One of the primary challenges is to give users control over where specific computations are executed. For efficient execution, some parts of the analysis may still need to be processed on the CPU, while others can benefit from being offloaded to the GPU. A solution should give the user control of making this distinction between CPU and GPU computation, as managing this tradeoff is essential for optimizing performance.

However, this may render the optimization only effective to advanced users who are more familiar with the strengths of the underlying hardware. To make these features accessible to a broader user base, it is important to invest in educational resources such as guidelines and tutorials.

A more technical challenge is generating arbitrary code for the GPU. ROOT uses Just-In-Time (JIT) compilation to generate code based on user input. Right now, ROOT can

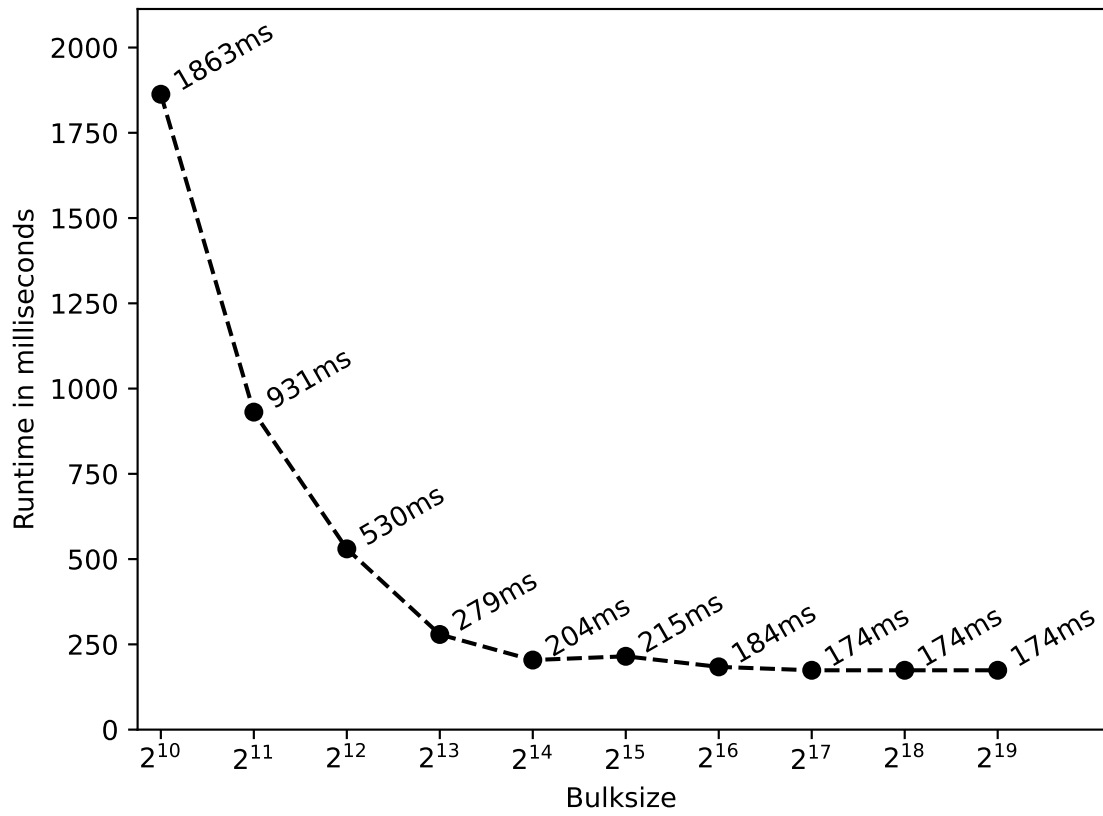


Figure 4: Effect on runtime (in milliseconds) for different bulksizes in GPU implementation of the Folded W Mass analysis.

generate C++ code but to handle arbitrary defines and filters on the GPU, we need to be able to generate CUDA code as well.

6 Conclusion

In this project, we successfully implemented a generic batch histogramming GPU kernel. This effort was motivated by the increasing data volume expected from the upcoming high-luminosity upgrade to the LHC, which will require faster data processing. By batching histograms on the GPU we hoped to reduce the data transfer from the CPU to the GPU when there is an overlap in the input data. However, we found that this use-case is relatively rare and decided to explore different optimizations instead.

We also explored the potential of executing define actions to the GPU. The overlap in input data for multiple defines provides an opportunity to offload these calculations to the GPU, increasing arithmetic intensity and reducing data transfers. Our experiments showed that in use-cases with many defines such as the Folded W Mass analysis, offloading certain defines and histogramming to the GPU yields a substantial performance gain. Even in a case with just a single histogram, there is no performance drawback when using the GPU.

With further development, executing arbitrary define actions and filters on GPUs could significantly accelerate histogramming in ROOT and enable physicists to handle the growing amounts of data generated in future experiments.

References

- [1] Rene Brun, Fons Rademakers, Philippe Canal, Axel Naumann, Olivier Couet, Lorenzo Moneta, Vassil Vassilev, Sergey Linev, Danilo Piparo, Gerardo GANIS, Bertrand Bellenot, Enrico Guiraud, Guilherme Amadio, wverkerke, Pere Mato, TimurP, Matevž Tadel, wlv, Enric Tejedor, Jakob Blomer, Andrei Gheata, Stephan Hageboeck, Stefan Roiser, marsupial, Stefan Wunsch, Oksana Shadura, Anirudha Bose, Cristina Cristescu, Xavier Valls, and Raphael Isemann. ROOT, June 2020.
- [2] Jolly Chen, Monica Dessole, and Ana Lucia Varbanescu. Lessons learned migrating CUDA to SYCL: A HEP case study with ROOT RDataFrame, 2024.
- [3] HEP Software Foundation, Johannes Albrecht, Antonio Augusto Alves, Guilherme Amadio, Giuseppe Andronico, Nguyen Anh-Ky, Laurent Aphecetche, John Apostolakis, Makoto Asai, Luca Atzori, et al. A roadmap for HEP software and computing R&D for the 2020s. *Computing and software for big science*, 3:1–49, 2019.