CLUSTER INNOVATION CENTRE, UNIVERSITY OF DELHI

Computational upgrades to the high energy physics data analysis pipeline for future LHC/HL-LHC runs

By
Saransh Chopra

Under the supervision of
Dr. Jim Pivarski

Thesis submitted in fulfillment of the requirements for the degree of
Bachelor of Technology

Submission Date: July 11, 2024

Disclaimer

I confirm that this thesis is my own work and I have documented all sources and material used.

New Delhi, July 11, 2024 Saransh Chopa

Funding information

Support for this work was provided by NSF cooperative agreements OAC-1836650 and PHY-2323298 (IRIS-HEP), and in receipt of a grant from the Department of Physics, Princeton University.

Abstract

High Energy Physics experiments, such as the Large Hadron Collider at CERN, produce petabytes of data every year. Physicists require scalable and efficient scientific software to analyze and perform physics on the obtained data.elemedis The initial frameworks and scientific software developed for analyzing HEP data, such as ROOT (Brun et al. 2020), GEANT4 (Agostinelli et al. 2003), and BOOST (Boost 2015), were written in C and C++; hence, such software had and still have a steep learning curve, especially for physicists with no programming background. Multiple HEP ecosystems have emerged in languages that are comparatively easy to pick up, such as the IRIS-HEP (Elmer et al. 2018) ecosystem in Python. This thesis began as a bid to implement the remaining pieces of Automatic Differentiation in Awkward Arrays (Pivarski et al. 2018b), Vector (Schreiner et al. b), and Coffea (Gray et al. 2023) but soon expanded to work on multiple other computational upgrades to the IRIS-HEP ecosystem. More specifically, this thesis extends the support of AD in Awkward Arrays, implements the Unified Histogram Interface (Schreiner et al. 2023b) for rebinning in boost-histogram (Schreiner et al. 2023a), migrates Coffea's vector algebra backend to Scikit-HEP/vector, and implements a symbolic backend in Vector. The work also includes several computational upgrades specifically in Vector to meet its rapidly growing user base. Finally, this thesis also includes development of a new Python package, cuda-histogram, to support Histogramming on GPUs for HEP data analysis pipelines. The work carried out in the past six months has already been integrated into the data analysis pipelines of physicists all around the globe. Furthermore, the upcoming upgrade of the Large Hadron Collider to the High-Luminosity Large Hadron Collider demands an even fine-grained suite of software, and the work carried out during this thesis adds up to these upgrades.

Keywords: scientific computing, high energy physics, differentiable programming, analysis systems, columnar analysis, vector algebra, histogramming, symbolic programming, GPU programming

Dedicated to my late grandfather,
Chattar Sain Chopra

Acknowledgements

This thesis compiles the wide range of computational upgrades carried out on the Scikit-HEP or the broader IRIS-HEP ecosystem in the past six months. First and foremost, I express my immense gratitude to my supervisor, *Jim Pivarski*. This work was only possible with Jim guiding me in the right direction. I am deeply grateful for everything I could absorb through him during the past six months, and thankful to him for always being there as a mentor and a friend. The contract itself (and the extension in the contract) was only possible through *Peter Elmer*. I am thankful for his time, efforts, and consideration. I am lucky enough to have worked with an organization as unique as IRIS-HEP. I am grateful to everyone at IRIS-HEP, especially *David Lange*, for organizing the R2 meetups and *Matthew Feickert* for coordinating much of my spread-up work.

Given that my work encompassed multiple pieces or libraries of the data analysis pipeline, I was fortunate enough to be mentored by many other people over a short duration of time. I cannot miss thanking *Henry Schreiner* for mentoring me on the Unified Histogram Interface project and constantly acting as a source of inspiration. I would also like to thank *Lindsey Gray* and *Nicholas Smith* for supervising me and being patient while I migrated Coffea's internals to Scikit-HEP/vector. Even though *Alexander Held* could not supervise me directly to work on Automatic Differentiability for the Analysis Grand Challenge, I am grateful for all the always helpful discussions I have had with him.

Apart from supervisors, this work would not have been possible without the friends I made at CERN. Everything added up to this work, from talking about the bugs we are trying to solve to playing a game of cards at midnight. I will also take this opportunity to thank my friends and family back home. The ten days I spent in India were fantastic, only because I met these amazing people every day. The thesis itself was written through the constant pushes by *Mel*. Thank you everyone for supporting and being there for me while writing this.

Finally, I would like to thank my institution, Cluster Innovation Centre, University of Delhi, for allowing me to write a thesis at CERN. Their constant support and flexibility with the process is commendable. I am incredibly grateful to have studied and worked under *Prof. Shobha Bagai*, who has acted as an informal mentor throughout the four years of my degree.

Contents

Listings

List of Figures

List of Tables

Abbreviations

CERN	Conseil Européen pour la Recherche Nucléaire
HEP	High Energy Physics
LHC	Large Hadron Collider
HL-LHC	High Luminosity - Large Hadron Collider
ROOT	Rapid Object-Oriented Technology
AD	Automatic Differentiation
UHI	Unified Histogram Interface
Coffea	Columnar Framework For Effective Analysis

Table 1: Abbreviations/notations used across this thesis.

Preface

If you told middle school Saransh that he would one day not only get to work but also write his thesis at CERN, he would lose his mind. I know you wanted to be a physicist, and Indian society pushed you into engineering, but you still navigated your way. Now, look at your writing software for high energy and nuclear physics. I still do not know how I ended up here, but it all started with a Slack message to Peter, and the next thing I knew, I was sitting at R1 with Aaron, Aryaman, Vaibhav, and Manasvi having dinner. All the morning R2 meetups with croissants by David, climbing sessions with Prajwal, R1 lunches with Aaron, weekend night stays at Aryaman's place, *kabbo* games at the Schuman hostel with Alok, Amal, Aryan, and Vaibhav, and car rides with Sofian and Jordan have contributed to this thesis as much as my work in Building 8 has done.

I will forever be grateful to the entire IRIS-HEP team for funding me to work under such incredible people on such intriguing projects. Each computational upgrade mentioned in this thesis was a never-before-seen challenge for me, teaching me invaluable skills. The sheer amount of knowledge that I absorbed while performing these upgrades ranged from pure mathematics (what is the physical significance of a Jacobian?) to a little bit of Physics (why does Vector's new symbolic backend only work for positive time-like vectors??) to software engineering principles (what is the best way to migrate the vector algebra backend of Coffea to Scikit-HEP/vector without annoying the users?). This is the most diverse range of work I have carried out in such a short time, and I was able to do it because of a highly supportive mentor and an always-ready-to-help team.

I was a bit hesitant before moving to Geneva, and I remember how lonely and hard the first few weeks were, but now it's even harder to leave. I started liking Geneva so much that I ended up extending my contract by two months, pushing the joining date of an in-hand full-time role further. This thesis gave me an opportunity to travel around Europe at the age of 22, an experience I will never forget. When I look back at 2024, all I will remember is sipping Aperol in Italy, overcoming my fear of seafood in Spain, slowing down the time after having a space cake in the Netherlands, devouring kebabs and baklavas in Turkey, buttering a croissant in France, chugging a beer in Germany, dipping things in cheese in Switzerland, and taking a sauna in Finland. These experiences would not have been possible without the lifelong friends I made at CERN. Finally, Switzerland gave me something very special, my girlfriend. I will always have a reason to revisit Switzerland.

Any place I work in the remaining years of my life will never come close to CERN. It has been an absolute pleasure being at a place that is at the center of the modern scientific revolution. Leaving CERN after finishing this thesis will be one of the hardest things I have done in my life, but rest assured, I will be back someday. Younger Saransh will be so proud. I hope you have fun reading the rest of this thesis, which will get a bit more technical than this preface. Happy reading!

Part I

Fundamentals

Chapter 1

An introduction to Computation in High Energy Physics

What is a Computer Science student doing at CERN? This chapter introduces the computations required by experiments and the HEP research at CERN. Before reaching the computational part, it is imperative to outline how HEP research is carried out at CERN's flagship experiment, the Large Hadron Collider.

1.1 High Energy Physics at CERN

CERN or the Conseil européen pour la Recherche Nucléaire (European Organisation for Nuclear Research), was founded in 1952 with the aim of collecting researchers at a single, state-of-the-art facility to combat the after-effects of World War 2. Since then, CERN has grown exponentially and is now known for HEP and carrying out incredible research in engineering, computer science, and pure sciences other than physics. CERN houses multiple cutting-edge experiments with one goal: studying the universe at the atomic level. The flagship experiment of CERN, the LHC or the Large Hadron Collider, is a 27-kilometer ring situated 100 meters underground, capable of colliding particles such as protons as well as heavy ions. The LHC is the world's largest experiment and the most powerful particle accelerator ever built. The high-energy particle beams are accelerated at almost the speed of light in opposite directions and are guided in a circular track using superconducting electromagnets.

The particles start as a Hydrogen atom and are accelerated in Linac4 (Bertone et al. 2011), a linear accelerator, before passing into the Proton Synchrotron Booster (Reich). These high-energy beams are stripped off of their electrons when they leave Linac4 and are passed onto the Proton Synchrotron (ps 1962) next. Once accelerated at a threshold, the beams are redirected to the Super Proton Synchrotron (Doble et al. 2017). Until this step, the beams rotate in a single direction, but the beams split into two and go in opposite directions once they are released from the Super Proton Synchrotron into the Large Hadron Collider,

The collider houses four broad experiments or detectors: ATLAS (Aad et al. 2008), CMS (Compact Muon Solenoid) (Chatrchyan et al. 2008), LHCb (Large Hadron Collider beauty) (Alves et al. 2008), and ALICE (A Large Ion Collider Experiment) (Aamodt et al. 2008). ATLAS and CMS

are two general-purpose detectors built with the same goal but different approaches. The particles collide at the center of these detectors, and the detectors take a snapshot of the collision event for the physicists. ATLAS and CMS can register the created particles' paths, momentum, and energy. Additionally, CMS is also involved in searching for extra dimensions and particles that could make up dark matter. On the other hand, the LHCb and the ALICE experiments work differently from the two general-purpose detectors. LHCb investigates the differences between matter and antimatter by studying the beauty quark, and ALICE records the collisions of heavy ions (such as Lead and Oxygen ions) to examine the state of the universe that existed right after its birth by producing quark-gluon plasma.

The particle collisions transform energy to produce particles that might not readily exist in nature. The created particles travel in a straight line, but electromagnets are used to curve their trajectories and gather information about them. Physicists use several data points and types of equipment to gather evidence about the newly produced particles. For instance, a particle with a lower momentum is deflected more by the magnetic field than a particle with a higher momentum. Several other factors, including but not limited to energy loss (measured by calorimeters), the trajectory of the particles (reconstructed using tracking devices and machine learning algorithms), the emittance of Cherenkov radiation, particle's velocity, and particle's characteristic (muons can pass through most of the matter) are used at HEP experiments to gather information about the collision event. The unstable particles created during collisions further decay before being detected, thus creating a chain of decay that can be backtracked by applying physics to the gathered data.

1.2 Computational needs for High Energy Physics

The work at CERN operates at the intersection of Engineering, Computer Science, and pure Sciences like Physics and Mathematics. The particle collisions in the detectors generate tons of data. This data goes through advanced trigger systems that filter the events and flow them to the storage facilities. The collected data requires state-of-the-art computational methods for analyzing and extracting physics. CERN's computational infrastructure ranges from OpenStack (Rosado & Bernardino 2014), an open-source Infrastructure-as-a-Service cloud platform, to ROOT, an analysis framework for high-energy data, to Zenodo (European Organization For Nuclear Research & OpenAIRE 2013), and open source, open access, open science platform, to finally, Geant4, a simulation toolkit for high energy physics.

This thesis focuses on developing the data analysis tools or pipelines used at CERN to perform physics with the data captured by CERN's detectors. The particle collisions at the LHC produce a petabyte of data each second, which is stored in places like the CERN's data center. CERN has made numerous technological advancements to handle and process the sheer amount of data produced. For instance, the problem of sharing data and information amongst scientists led to the development of the World Wide Web at CERN. Analysis of this ever-increasing data has been challenging since the inception of CERN, and several efforts are in place to make this easier as time passes. Moreover, with the upcoming upgrade of the LHC to the HL-LHC, it has become imperative to improve the existing software suites to handle the surge in incoming data.

ROOT, started in 1995, is CERN's first comprehensive and still functioning in-house solution

to the challenges faced by physicists while analyzing HEP data. ROOT provides physicists with the ability to read, write, analyze, mine, and plot the data, as well as interactively play with it. The original ROOT framework is written in C++, making it highly efficient and scalable, but also making it infamous amongst physicists for having an extremely steep learning curve. ROOT does provide bindings in other languages, such as Python and R, but the comfort of writing code using these bindings comes at the cost of speed and efficiency. Recent efforts have been made to write HEP software in interpreted and just-in-time compiled languages from scratch. These new frameworks and libraries are slowly being integrated at CERN, but ROOT remains the most used software for data analysis for HEP.

1.3 The Scikit-HEP ecosystem

The Scikit-HEP (Rodrigues et al. 2020) ecosystem provides tools for analyzing HEP data using Python. The project is primarily maintained by IRIS-HEP, a software institute developing and working on the challenges of data-intensive scientific research at the HL-LHC experiment at CERN. The Scikit-HEP ecosystem is a modular ecosystem providing users with an interface for datasets, aggregations, modeling, simulation, and visualization. The ecosystem blends well with the other Python ecosystems and even C++ high-energy frameworks like ROOT. The ecosystem is already used within multiple HEP experiments in and outside CERN. Further, CERN's software pipeline, including ROOT, readily adopts bits and pieces from the Scikit-HEP ecosystem for improvements.

Library	**Function**
Awkward Array	Manipulate JSON-like data with NumPy-like idioms
Vector	Manipulate Lorentz, 3D, and 2D vectors
Uproot (Pivarski et al. 2017)	ROOT I/O in pure Python and NumPy
Uproot Browser	Terminal browser and tools for ROOT files
Boost-histogram	Python bindings for the C++14 Boost::Histogram library
Hist (Schreiner et al. a)	Hist is a analyst friendly front-end for boost-histogram
Decay (Rodrigues & Schreiner a)	Describe and convert particle decays
Particle (Rodrigues & Schreiner b)	PDG particle data and identification codes
iminuit (Dembinski & et al. 2020)	Jupyter-friendly Python interface for the Minuit2
Cabinetry (cra 2021)	Design and steer profile likelihood fits
pyhf (Heinrich et al.)	pure-Python implementation of HistFactory models

Table 1.1: A few Scikit-HEP projects and their functionality.

Scikit-HEP also encompasses projects like ROOT's distribution on conda-forge, cibuildwheel, a utility for building all PyPI-supported binary wheels, and Pybind11, a C++11 API for CPython and PyPy. This thesis aims to upgrade the Scikit-HEP ecosystem for both, niche HEP and general programming use cases. Most of the work in this thesis was carried out on Scikit-HEP, preparing it for the challenges posed by the future of HEP experiments at CERN and other research institutes.

Part II

Computational Upgrades

Chapter 2

Automatic differentiation for the Scikit-HEP ecosystem

2.1 Automatic differentiation and JAX

Automatic differentiation, or AD, is a methodology used to evaluate derivatives using computer systems. AD uses the chain rule as the fundamental algorithm behind its work and is broadly categorized into two types: forward mode and reverse mode. In AD, the chain rule is repeatedly applied to the operations until the final derivative value is obtained. The usage of chain rule distinguishes AD from symbolic and numerical differentiation, making it a better choice for scientists. Numerical differentiation produces round-off and discretization errors, given that the obtained derivative value is generated using approximation algorithms, but AD outputs the exact derivative value. Unlike numerical differentiation, though symbolic differentiation can give exact gradients along with the gradient expressions, it is prone to phenomena like expression swell and does not scale well.

2.1.1 Chain rule

The chain rule allows computing the derivative of composite differentiable functions in terms of the functions itself. Following the *Langrage* notation, the derivative of a function $h(x)$, expressed as the composition of functions f and g ($h = f \circ g$), can be calculated as -

$$h' = (f' \circ g).g'$$

One can also express the rule using *Leibnitz* notation, where a depends on z, z depends on y, and y depends on x -

$$\frac{da}{dx} = \frac{da}{dz}.\frac{dz}{dy}.\frac{dy}{dx}$$

2.1.2 Forward mode

The forward mode AD computes the chained derivatives bottom-up, that is, evaluating the last expression in the chain rule first and the first expression at last. More formally, the forward mode

AD computes the following recursive relation -

$$\frac{\partial w_i}{\partial x} = \frac{\partial w_i}{\partial w_{i-1}} \frac{\partial w_{i-1}}{\partial x}$$

with the initial case, $w_i = y$.

Forward mode AD performs one pass for each variable, calculating the function value and the derivative in the same pass. Therefore, this type of AD is preferred for functions with a lower number of independent variables, allowing a lower number of passes. For a function defined as $f : \mathbf{R}^n \to \mathbf{R}^m$, the forward mode will compute the final result in n passes; hence, it is considered efficient if $n << m$.

2.1.3 Reverse mode

The reverse mode AD computes the chained derivatives top-down, that is, evaluating the first expression in the chain rule first and the last expression at last. More formally, the reverse mode AD computes the following recursive relation -

$$\frac{\partial y}{\partial w_i} = \frac{\partial y}{\partial w_{i+1}} \frac{\partial w_{i+1}}{\partial w}$$

with the initial case, $w_0 = x$.

Reverse mode AD performs one pass to evaluate the function and another pass to calculate the partial derivatives for every independent variable. For a function defined as $f : \mathbf{R}^n \to \mathbf{R}^m$, the forward mode will compute the final result in m passes; hence, it is considered efficient if $n >> m$.

2.1.4 Differentiable programming with JAX

Differentiable programming is a programming paradigm where the written program is differentiable using techniques like AD. Differentiable programming languages generally generate a compiled or a dynamic computational graph for the written code, which is used to differentiate the program using AD. Compiled graphs can use compiler optimizations to speed up and scale computations but are usually complex to reason or debug. On the other hand, dynamic graphs can miss important compiler optimizations and might not scale well with data, but this operation is much more readable and accessible to debug. Software like Tensorflow (Abadi et al. 2015) and Clad (Vassilev et al. 2015) generate a compiled graph for AD, whereas libraries like PyTorch (Ansel et al. 2024) (Paszke et al. 2019) and NumPy (Harris et al. 2020) rely on dynamic graphs for AD. Interestingly, Julia's (Bezanson et al. 2017) Zygote.jl (Innes 2018) extracts the best of both worlds and generates a graph on Julia's intermediate representation, leveraging the just-in-time compiler for compiler optimizations.

JAX (Bradbury et al. 2018) is a Python library for high-performance computing, program transformations, and differentiable programming. The language follows the functional and differentiable programming paradigm, allowing users to write safe and differentiable code. Like Julia, JAX can be compiled just-in-time, allowing its dynamic computation graph to be statically compiled.

The static compilation enables compiler optimizations to work on the graph while differentiating it. JAX also provides a way to extend its AD capabilities to user-defined data structures using the `register_pytree_node` function. `register_pytree_node` only requires the users to specify a way to flatten and unflatten the custom data structure, handling everything else on its own.

JAX is compatible with NumPy, the fundamental numerical computing library for the IRIS-HEP and Scikit-HEP ecosystem, making it a perfect choice for implementing AD. Furthermore, the existing efforts to introduce AD in the IRIS-HEP ecosystem, including the work done by gradhep (gradHEP), rely on JAX. The existence of JAX in the IRIS-HEP ecosystem makes it an even better choice for other libraries in the same ecosystem.

2.2 Need for automatic differentiation in HEP analysis pipelines

HEP data analysis pipelines require tuning several free parameters to either precisely measure the known measurements of the standard model or to detect new particles. For instance, (Simpson 2023a) shows how machine learning algorithms are utilized to extract relevant data or discriminate the required signal from background noise in HEP pipelines. Specialized machine learning algorithms such as gradient descent work well on specific tasks, but such algorithms are often optimized only for a single physics task. The specialization of these algorithms does not account for systematic uncertainties and removes several steps from the ultimate physical goal of searching for a new particle or testing a new physical theory (Guest et al. 2018).

(Simpson 2023a) further shows how having a differentiable program on the statistical side of HEP pipelines can help overcome the limitations of traditional loss functions. The work mentions how p-value can be a good objective function for some analysis tasks, requiring the function and the subset of the program to be differentiable. The existing program will have to be differentiated in its entirety to reach the p-value function, which is not possible or highly inefficient in most cases. This paves the way for making standalone modular chunks in the HEP pipelines differentiable. Therefore, allowing the fundamental libraries such as Awkward Arrays and Vector, which are required for manipulating and performing initial physics with HEP data, becomes imperative.

2.3 Automatic differentiation and Awkward Arrays

Awkward Arrays are designed to handle the non-uniform nature of HEP data by providing jagged array structures. The library offers NumPy-like idioms for jagged data, enhancing the speed, scalability, and efficiency of HEP analysis in Python. Rather than representing data as jagged arrays in memory, Awkward Arrays store data linearly while specifying a nested data structure with offsets as metadata. This approach allows users and the memory management system to switch between jagged and linear array representations. Figure 2.1 illustrates the memory representation of an Awkward Array.

Given that JAX requires users to specify methods for flattening and unflattening data structures for differentiation, an Awkward Array is initially flattened into `NumpyArray`s before being passed to the necessary JAX function. Once flattened, Awkward slices, *ufuncs*, and `behavior` are managed to

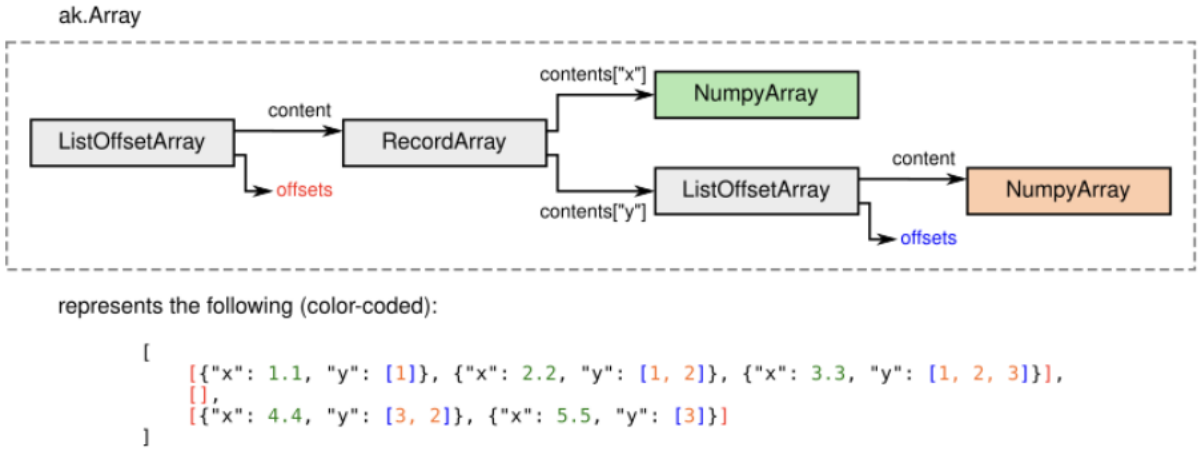

Figure 2.1: Representing an Awkward array in the memory (Pivarski).

ensure interoperability between the Awkward `Array` and JAX's `DeviceArray`. Although this data structure becomes differentiable via JAX, it is not yet in a user-ready format. After obtaining results from JAX, the flattened array is reassembled into an Awkward Array using the stored metadata. The unflattened array is then returned to the user, thereby abstracting the underlying JAX and Awkward interoperability. Figure 2.2 graphically illustrates the compatibility of an Awkward Array with JAX and its automatic differentiation system.

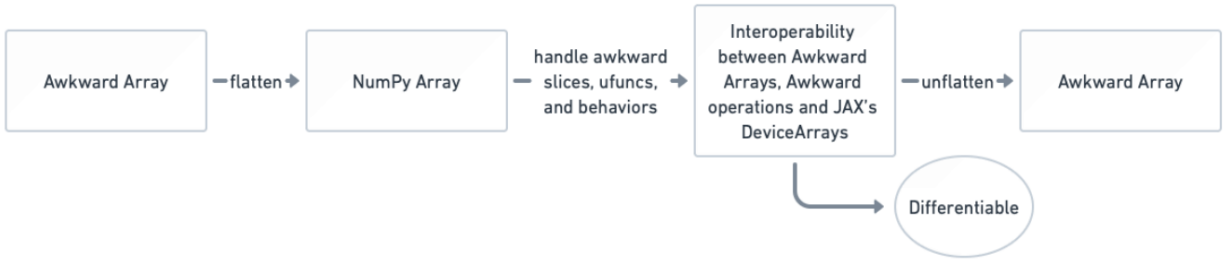

Figure 2.2: Interoperability between Awkward's Array and JAX's DeviceArray.

Example 2.1 shows how Awkward's JAX backend can be used to compute the jacobian-vector product and gradient of jagged data. The example highlights how the backend can handle NumPy and Awkward ufuncs, complex slicing, as well as jagged data without any issues.

```
import jax
import awkward as ak
import numpy as np

ak.jax.register_and_check()

def f(x):
    return np.power(x[[2, 2, 0], ::-1], 3)
```

```
primals = ak.Array([[1., 2., 3.], [], [5., 6.]], backend="jax")
tangents = ak.Array([[0., 1., 0.], [], [0., 0.]], backend="jax")

val, grad = jax.jvp(f, (primals,), (tangents,))
# <Array [[216.0, 125.0], [...], [27.0, 8.0, 1.0]]
# type='3 * var * float32'>
# <Array [[0.0, 0.0], [0.0, ...], [0.0, 12.0, 0.0]]
# type='3 * var * float32'>

print(jax.grad(np.sum)()(primals)())
# [[1.0, 1.0, 1.0], [], [1.0, 1.0]]
```

Listing 2.1: Awkward's JAX backend in code.

The support for AD through JAX existed in Awkward Arrays, but it was buggy and untested by physicists. This work aimed at polishing the existing JAX backend of Awkward Arrays and extending it to the other IRIS-HEP libraries. This work extended the Awkward's JAX backend to -

- work on any arbitrary combination of binary operations
- not clash with Numba functions
- allow differentiating through chained ufuncs,
- allow differentiating through `ak.count` and `ak.mean`

Therefore, as described in 2.2, Awkward users can now perform a much more comprehensive set of differentiable operations on the data using Awkward's JAX backend.

```
import jax
import awkward as ak

ak.jax.register_and_check()

def f(x):
    return ak.mean(ak.sum(x) * x)

a = ak.Array([[1., 2., 3.], [], [5., 6.]], backend="jax")

f(a), jax.grad(f)(a)
# Array(57.8, dtype=float32),
# <Array [[6.8, 6.8, 6.8], [], [6.8, 6.8]]
# type='2 * var * float32'>
```

Listing 2.2: Differentiating through a chain of *ufuncs*, Awkward's mean function, and binary operations using Awkward's JAX backend.

Furthermore, 2.3 shows how the JAX backend works well with Numba (Lam et al. 2015) functions and decorators, causing no clashes between Awkward, JAX, and Numba.

```
import jax
import awkward as ak
import numba

ak.jax.register_and_check()
behavior = {}

input_arr = ak.Array([1.0], backend="jax")

@numba.vectorize(
    [
        numba.float32(numba.float32, numba.float32),
        numba.float64(numba.float64, numba.float64),
    ]
)
def _some_kernel(x, y):
    return x * x + y * y

@ak.mixin_class(behavior)
class SomeClass:
    @property
    def some_kernel(self):
        return _some_kernel(self.x, self.y)

ak.behavior.update(behavior)

arr = ak.zip(
    {"x": input_arr, "y": input_arr}
    with_name="SomeClass"
)

arr.some_kernel
# [2.0]
# --------------------
# 1 * float32
```

Listing 2.3: Awkward's JAX backend does not clash with Numba decorators and functions.

All fixes and features to Awkward's JAX backend have been released and thoroughly tested within the repository's continuous integration pipeline, significantly enhancing its completeness and reliability for practical HEP analysis pipelines. To assess the readiness of Awkward's JAX backend, ongoing efforts aim to integrate automatic differentiation (AD) through Awkward in the Analysis

Grand Challenge (AGC) (Held et al. 2022), resulting in a prototype that showcases the backend's capabilities.

2.4 Automatic differentiation and the Analysis Grand Challenge

The AGC is designed to perform fundamental HEP analysis by constructing comprehensive data pipelines that demonstrate the capabilities of the IRIS-HEP ecosystem. This challenge facilitates the integration of various components of the ecosystem into a single code pipeline, encompassing data extraction, processing, modeling, and the final presentation of results. The inclusion of AD within the AGC pipeline, enabled by the work undertaken in this thesis, marks a significant advancement in the project's ability to successfully adopt AD through Awkward's JAX backend.

Neos (Simpson & Heinrich 2021) and Relaxed (Simpson 2023b) provide AD capabilities for AGC's modeling, inference, and plotting stages. However, the initial stages of data extraction and manipulation, involving Awkward Arrays, Vector (fundamental Scikit-HEP libraries), and Coffea (a columnar collider analysis framework), previously lacked AD support. This thesis has extended AD implementation in Awkward to Coffea and Vector, thereby enabling the integration of AD across the entire AGC pipeline. Examples illustrating the use of Awkward's JAX backend with Coffea and Vector are shown in 2.4 and 2.5, respectively.

```
import awkward as ak
from coffea.nanoevents import NanoEventsFactory, NanoAODSchema

ak.jax.register_and_check()
NanoAODSchema.warn_missing_crossrefs = False

ttbar_file = "https://github.com/scikit-hep/"\
    "scikit-hep-testdata/raw/main/src/skhep_testdata/data/"\
    "nanoAOD_2015_CMS_Open_Data_ttbar.root"

events = NanoEventsFactory.from_root(
    {ttbar_file: "Events"},
    schemaclass=NanoAODSchema
).events()
events = ak.to_backend(events.compute(), "jax")

evtfilter = ak.to_backend(ak.num(events.Jet.pt) >= 2, "jax")
jets = events.Jet[evtfilter]

(jets[:, 0] + jets[:, 1]).mass
# <Array [157.21956, 81.92088, ..., 32.363174, 223.94753]
# type='140 * float32'>
```

Listing 2.4: Coffea using Awkward's JAX backend to support AD.

```
import awkward as ak
import vector
import numpy as np
import uproot

vector.register_awkward()
ak.jax.register_and_check()

ttbar_file = "https://github.com/scikit-hep/"\
    "scikit-hep-testdata/raw/main/src/skhep_testdata/data/"\
    "nanoAOD_2015_CMS_Open_Data_ttbar.root"

with uproot.open(ttbar_file) as f:
    arr = f["Events"].arrays(
            [
                "Electron_pt",
                "Electron_eta",
                "Electron_phi",
                "Electron_mass",
                "Electron_charge"
            ]
        )

px = arr.Electron_pt * np.cos(arr.Electron_phi)
py = arr.Electron_pt * np.sin(arr.Electron_phi)
pz = arr.Electron_pt * np.sinh(arr.Electron_eta)
E = np.sqrt(arr.Electron_mass**2 + px**2 + py**2 + pz**2)


evtfilter = ak.num(arr["Electron_pt"]) >= 2

els = ak.zip(
    {
        "pt": arr.Electron_pt,
        "eta": arr.Electron_eta,
        "phi": arr.Electron_phi,
        "energy": E,
        "charge": arr.Electron_charge
    },
    with_name="Momentum4D"
)[evtfilter]
els = ak.to_backend(els, "jax")

(els[:, 0] + els[:, 1]).mass
# <Array [86.903534, 97.60412, ..., 62.408997, 50.49058]
```

```
# type='5 * float32'>
```

Listing 2.5: Vector using Awkward's JAX backend to support AD.

Consequently, this work has expedited the incorporation of AD into physicists' analysis pipelines, with a prototype implementation currently being developed at agc-autodiff (Held b).

Chapter 3

Unified Histogram Interface for variable axis rebinning

Almost every analysis in HEP ends with a histogram, and the irregularities or spikes in these plots help physicists identify particles or reconfirm measurements. Therefore, having fast, flexible, easy-to-use, and easy-to-plot solutions for physicists in any data analysis pipeline becomes essential. The Scikit-HEP ecosystem includes a modular mini-ecosystem of histogram libraries that are interoperable with each other, as well as with ROOT and PyROOT (Dawe et al. 2015).

3.1 The histogramming mini-ecosystem

The Scikit-HEP ecosystem consists of scalable and efficient histogramming libraries either written entirely in Python or providing Python bindings without compromising speed. This mini-ecosystem follows the ideology of Histogram-as-an-Object to create powerful histogramming functionalities, such as projects, advanced indexing, slicing, serialization, plotting, and much more. The ecosystem consists of boost-histogram, a Python package that provides Python bindings for Boost.Histogram, Hist, analyst-friendly front-end for boost-histogram, Histoprint (Koch et al. 2022), nicely displays histograms in the terminal, UHI, a protocol providing static tools and documentation for the expected behavior and interaction between histogram libraries, and dask-histogram, Dask support for boost-histogram. Other libraries that have been deprecated include aghast (Scikit-Hep) and histbook (Pivarski et al. 2018a).

Boost-histogram is one of the fastest histogramming libraries present in the HEP ecosystem and forms the basis for all other Scikit-HEP histogramming libraries. The library treats histograms as objects, and they are written internally in C++ to maintain speed. Moreover, the library is interoperable with ROOT histograms, PyROOT histograms, as well as histograms written by Uproot. Boost-histogram supports most of the UHI protocol, making it compatible with any library that supports the same. The entire histogramming mini-ecosystem is described in 3.2.

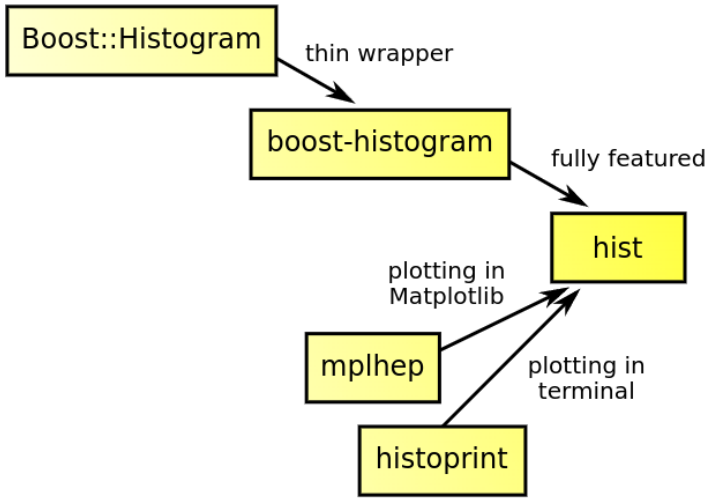

Figure 3.1: Scikit-HEP's mini-ecosystem for Histograms-as-an-Object.

3.2 Unified Histogram Interface

UHI or Unified Histogram Interface specifies protocols for the current and future histogram libraries of Scikit-HEP. These protocols allow the libraries and their histograms to interact with each other and have a consistent API signature. UHI was primarily developed at Scikit-HEP under IRIS-HEP but is now being adopted by other software frameworks, such as Coffea and ROOT.

The interface provides three major protocols for all the histogram libraries -

- UHI Indexing: powerful indexing system for histograms, designed to extend standard Array indexing for Histogram operations.
- UHI Indexing+: set of extensions to the standard indexing that make it easier to use on the command line.
- PlottableProtocol: minimal and complete set of requirements for a source library to produce and a plotting library to consume to plot a histogram, including error bars.

3.3 The problem of non-uniform axis rebinning

Physicists often require rebinning or transforming the axes of a histogram in their analysis pipelines. Rebinning includes changing an axis of a histogram and redistributing its content accordingly. This transformation can either change the number of intervals, keeping them equidistant from each other (going from a `Regular` axis to a `Regular` axis) or change the distance between two intervals (going from a `Regular` axis to a `Variable` axis).

Boost-histogram supported rebinning a `Regular` axis to a `Regular` axis (uniform rebinning) but lacked the support for rebinning a `Regular` axis to a `Variable` axis (non-uniform rebinning).

For instance, consider the following `Regular` axis with ten bins placed at equal distances within the interval $[0, 10]$ -

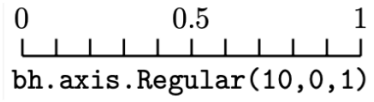

Figure 3.2: Regular axis with 10 bins.

A histogram with the axis described above can be constructed in boost-histogram using 3.1.

```
import boost_histogram as bh

h = bh.Histogram(bh.axis.Regular(10, 0, 1))
```

Listing 3.1: Constructing a 1D histogram with Regular axis using boost-histogram.

The axis described in 3.1 can be rebinned by a factor of 0.5 to increase the number of bins while keeping them at an equal distance within the same interval. Figure 3.3 shows how the final result will look like.

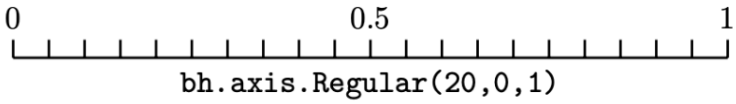

Figure 3.3: Regular axis with 20 bins.

Following UHI indexing, the rebinning described above can be carried out in boost-histogram using 3.2.

```
h[::rebin(0.5)]
```

Listing 3.2: Rebinning the histogram with a factor of 0.5.

Now consider the variable axis in 3.4 with three bins spread unequally within the same interval, $[1, 10]$.

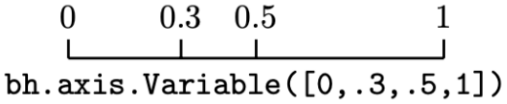

Figure 3.4: Variable axis with 3 bins.

Boost-histogram limited the rebinning process to Regular-Regular rebinning; hence, the histogram's `Regular` axis could not be rebinned to the described `Variable` axis. Multiple physicists

have requested this particular case of rebinning through different mediums, including (Chen), (Schreiner a), (Feikert), and (Held a). The protocol for variable axis or non-uniform rebinning is specified in the UHI, but its implementation needed to be added to boost-histogram. Adding complete UHI support will enable boost-histogram to support non-uniform rebinning, allowing physicists to use the efficient and UHI-compatible solution instead of writing their implementations

3.4 Implementing UHI for rebinning in boost-histogram

The implementation introduces a new `Rebinner` class, as highlighted in UHI, that is capable of accepting a `factor` and a list of `groups`, along with it being scalable or extendable for the future. An object of the `Rebinner` class can be passed in UHI-compatible indexing operations, making the API similar to the existing `rebin` function. The `Rebinner` class is also exported out of the library with an alias `rebin` to ensure backward compatibility. Listing 3.3 shows the implementation of the UHI-compatible `Rebinner` class. The new `groups` argument specifies how the existing bins should be grouped with each other. For instance, rebinning a histogram with axis `Regular(10, 0, 1)` using `Rebinner(groups=[3, 2, 5])` will result in a new histogram with axis `Variable([0, 0.3, 0.5, 1])`.

```
class Rebinner:
    __slots__ = (
        "factor",
        "groups",
    )

    def __init__(
        self,
        factor: int | None = None,
        *,
        groups: Sequence[int] | None = None,
    ) -> None:
        if not sum(i is None for i in [factor, groups]) == 1:
            raise ValueError()
        self.factor = factor
        self.groups = groups

    def __repr__(self) -> str:
        repr_str = f"{self.__class__.__name__}"
        args: dict[str, int | Sequence[int] | None] = {
            "factor": self.factor,
            "groups": self.groups,
        }
        for k, v in args.items():
            if v is not None:
                return_str = f"{repr_str}({k}={v})"
```

```
                break
        return return_str

    def __call__(self, axis: PlottableAxis) -> int | Sequence[int]:
        if self.factor is not None:
            return [self.factor] * (len(axis) // self.factor)

        if self.groups is not None:
            return self.groups

        raise NotImplementedError(axis)
```

Listing 3.3: A new Rebinner class following the UHI protocol.

Figure 3.5 outlines the code flow within boost-histogram after implementing variable axis rebinning. It can be observed that the algorithm for non-uniform rebinning is executed in pure Python instead of C++. At the moment, slicing and regular rebinning are written in C++ with bindings present in Python, but the new non-uniform rebinning has been written in Python while maintaining interoperability within the C++ and Python code blocks inside boost-histogram.

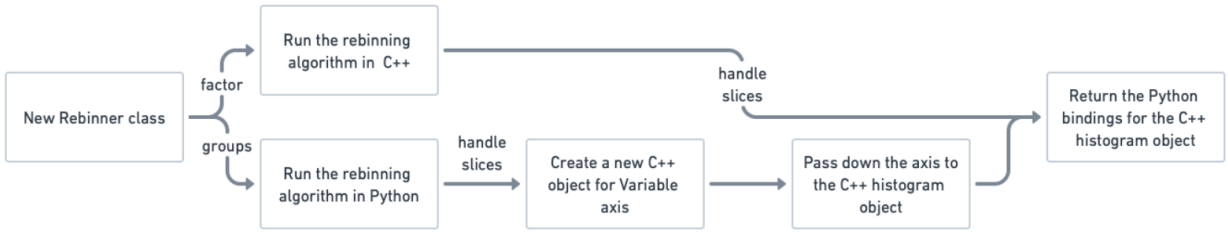

Figure 3.5: Implementation of UHI for non-uniform rebinning in boost-histogram.

Listing 3.4 describes the interface for rebinning a `Regular` axis to a `Regular` or a `Variable` axis after completing the UHI implementation in boost-histogram.

```
import boost_histogram as bh

h = bh.Histogram(bh.axis.Regular(10, 0, 1))

rebin = bh.tag.Rebinner(factor=0.5)
h[::rebin]

rebin = bh.tag.Rebinner(groups=[1, 2, 3])
h[::rebin]
# Histogram(
#     Variable([0, 0.1, 0.3, 0.6], metadata=...),
```

```
#     storage=Double()
# ) # Sum: 0.0
```

Listing 3.4: Rebinning the histogram with a factor of 0.5 and in groups of 1, 2, 3.

Similarly, 3.5 shows how one can rebin a particular axis of an *ND* (in this case $N = 2$) histogram with the newly implemented UHI-compliant `Rebinner` class.

```
import boost_histogram as bh

h = bh.Histogram(
    bh.axis.Regular(20, 1, 3), bh.axis.Regular(30, 1, 3),
    bh.axis.Regular(40, 1, 3)
)

s = bh.tag.Slicer()

s[:: bh.rebin(groups=[1, 2, 3])]}].axes.size
# (3, 30, 40)

h[
    {
        0: s[:: bh.rebin(groups=[1, 2, 3])],
        2: s[:: bh.rebin(groups=[1, 2 ,3])]
    }
].axes[2].edges
# array([1.  , 1.05, 1.15, 1.3 ])
```

Listing 3.5: Rebinning a 2D histogram with the new API.

Boost-histogram is now equipped with the ability to perform non-uniform rebinning following the standard UHI protocols. The work includes several tests that ensures the stability of the feature, but it is still under review. The feature is scheduled to be released with the upcoming boost-histogram release, making it available to the physicists before the end of this year.

Chapter 4

Computational upgrades to Vector

Performing vector algebra and physics on the collected HEP data is integral to HEP analysis pipelines. Given that HEP data is not uniform, the vector algebra frameworks or libraries should work readily on non-uniform or jagged data, allowing users to perform operations on an entire jagged array in minimum passes. The vector algebra systems allow physicists to go for raw data to some meaningful results, which can then be visualized using the histogramming libraries. The Scikit-HEP ecosystem maintains its own library for performing vector algebra. This library is compatible with the rest of the HEP ecosystem and is designed to be a general-purpose package instead of a HEP-specific one.

4.1 A quick introduction to vector

Vector is Python library for creating and manipulating 2D, 3D, and Lorentz vectors, especially arrays of vectors, to solve common physics problems in a NumPy-like way. Vector allows physicists to perform physics on the data extracted by HEP experiments using backend libraries like Awkward Arrays. Vectors may be represented in a variety of coordinate systems: Cartesian, cylindrical, pseudorapidity, and any combination of these with time or proper time for Lorentz vectors. In all, there are 12 coordinate systems: $x-y$ vs $\rho-\varphi$ in the azimuthal plane $\times$ z vs θ vs η longitudinally $\times$ t vs τ temporally. Further, the names and conventions in Vector are uniform with ROOT's `TLorentzVector` and `Math::LorentzVector`, as well as Scikit-HEP/math, uproot-methods, henryiii/hepvector (Schreiner b), and `coffea.nanoevents.methods.vector`.

Vector implements a variety of backends for several purposes. Although Vector was written with HEP in mind, it is a general-purpose library that can be used for any scientific or engineering application. Before my work on Vector, it housed 3+2 backends, a pure Python object backend, a NumPy backend, an Awkward backend, and implementations of the Object and the Awkward backend in Numba for just-in-time compilable operations. Finally, Vector also provides a distinction between geometrical vectors, which have a minimum of attribute and method names, and vectors representing momentum, which have synonyms like $pt = rho$, $energy = t$, and $mass = tau$.

For instance, 4.1 shows how vectors of different backends can be constructed using the library.

```
import vector; import sympy; import awkward as ak

# creating Object type vectors
vector.MomentumObject3D(px=1.1, py=2.2, pz=3.3)
vector.VectorObject4D(x=1.1, y=2.2, eta=3.3, tau=4.4)

# creating SymPy type vectors
x, y, px, py, pz, eta, tau = sympy.symbols(
    'x y px py pz eta tau',
    real=True,
)
vector.MomentumSympy3D(px=px, py=py, pz=pz)
vector.VectorSympy4D(x=x, y=y, eta=eta, tau=tau)

# creating NumPy type vectors
# NumPy-like arguments (literally passed through to NumPy)
vector.VectorNumpy2D(
    [(1.1, 2.1), (1.2, 2.2), (1.3, 2.3), (1.4, 2.4), (1.5, 2.5)],
    dtype=[("x", float), ("y", float)],
)

# Pandas-like arguments (dict from names to column arrays)
vector.VectorNumpy2D(
    {
        "x": [1.1, 1.2, 1.3, 1.4, 1.5],
        "y": [2.1, 2.2, 2.3, 2.4, 2.5]
    }
)

# creating Awkward type vectors
vector.awk(
    [
        [
            {"x": 1, "y": 1.1, "z": 0.1},
            {"x": 2, "y": 2.2, "z": 0.2}
        ],
        [{"x": 3, "y": 3.3, "z": 0.3}],
    ]
)

vector.register_awkward()

ak.Array(
    [
        [
```

```
            {"x": 1, "y": 1.1, "z": 0.1},
            {"x": 2, "y": 2.2, "z": 0.2}
        ],
        [{"x": 3, "y": 3.3, "z": 0.3}],
    ],
    with_name="Vector3D",
)
```

Listing 4.1: Vector constructors.

Similarly, 4.2 shows how one can manipulate or perform operations on the created vectors using the methods and functions provided by the library.

```
# Broadcasts a rotation angle of 0.1 to both elements of the
# first list, 0.2 to the empty list, and 0.3 to the only
# element of the last list.
vector.awk(
    [
        [
            {"rho": 1, "phi": 0.1}, {"rho": 2, "phi": 0.2}],
            [],
            [{"rho": 3, "phi": 0.3}
        ]
    ]
).rotateZ([0.1, 0.2, 0.3])

# binary operator equivalents
vector.obj(x=3, y=4).scale(10)
vector.obj(x=1, y=2) @ vector.obj(x=5, y=5)

# deltaphi is a planar operation (defined on the transverse
# plane)
vector.obj(rho=1, phi=0.5).deltaphi(
    vector.obj(rho=2, phi=0.3)
)

vector.obj(pt=1, phi=1.3, eta=2).deltaR(
    vector.obj(pt=2, phi=0.3, eta=1)
)
```

Listing 4.2: Vector methods.

4.2 The recent surge in Vector's usage

Vector saw a sudden surge in its user base during my time working at CERN. This surge was attributed to various factors joining together near the start of LHC's run 3. The new upgrades to transform LHC into HL-LHC have sparked several efforts to improve the existing software infrastructure to handle a sudden and massive increase in HEP data. One such attempt, the 200 GBPS challenge (Bockelman et al.), outlines the requirements of the planned HL-LHC upgrades. The challenge aims to operate at 25% of the HL-LHC scale, allowing physicists to read 200TB of data in 30 minutes. This challenge and other LHC upgrades pushed several teams to Vector requesting various performance optimizations, bug fixes, and feature requests.

Besides the standalone Vector issues, the library also required a sync with the other HEP libraries in the ecosystem. For instance, given that physicists primarily rely on Vector's awkward backend, the new JAX and Dask (Dask Development Team 2016) backends of Awkward also required propagation in Vector. The new Awkward backends were already used by physicists in their analysis pipelines; hence, supporting them in Vector became a pressing issue. Furthermore, Coffea developers decided to move away from their internal vector modules to Scikit-HEP/vector. This internal switch of the vector algebra backend started a discussion about the mismatch in vector algebra frameworks in the HEP ecosystem and the process of making everything uniform.

4.3 Preparing Vector for future LHC/HL-LHC runs

Vector saw 5 new releases within a short span of six months, making it one of the most active years for the library. The releases included minor bug fixes, feature additions, and maintenance jobs, as well as, major design changes, new backend support, and a better cohesion with the rest of the HEP ecosystem.

The v1.2 release focused on specifying a uniform promotion and demotion scheme for the geometric coordinates of a vector of any backend. This discussion was prompted by the start of the migration of Coffea's vector modules to Scikit-HEP/vector. Along with a new scheme, sub-classing awkward mixins from Vector was made easier, bug-free, and more documented. Both these changes were requested directly by physicists working at CERN experiments, enabling Vector to adapt better to its intended use case. More specifically, the following non-maintenance and non-documentation changes were released in Vector in v1.2 -

- fix: result of an infix operation should be demoted to the lowest possible dimension
- fix: all infix operations should not depend on the order of arguments
- fix: return the correct awkward record when performing an infix operation
- fix: respect user defined awkward mixin subclasses and projection classes

4.3 shows the v1.2 updates and fixes in action.

```
import awkward as ak
import vector
from coffea.nanoevents.methods import vector

a = vector.zip(
    {
        "x": [10.0, 20.0, 30.0],
        "y": [-10.0, 20.0, 30.0],
        "z": [5.0, 10.0, 15.0],
        "t": [16.0, 31.0, 46.0],
    },
)
b = vector.zip(
    {
        "x": [-10.0, 20.0, -30.0],
        "y": [-10.0, -20.0, 30.0],
        "z": [5.0, -10.0, 15.0],
    },
)

{
    "x": [0.0, 40.0, 0.0],
    "y": [0.0, 0.0, 60.0],
    "z": [10.0, 0.0, 30.0],
    "t": [16.0, 31.0, 46.0],
},

# before fix
ak.parameters(a+b), ak.parameters(b+a)
# {'__record__': 'Momentum4D'}, {'__record__': 'Momentum3D'}

# after fix
ak.parameters(a+b), ak.parameters(b+a)
# {'__record__': 'Momentum3D'}, {'__record__': 'Momentum3D'}

# before fix
a + b, b + a
{
    "x": [0.0, 40.0, 0.0],
    "y": [0.0, 0.0, 60.0],
    "z": [10.0, 0.0, 30.0],
    "t": [16.0, 31.0, 46.0],
}, {
    "x": [0.0, 40.0, 0.0],
    "y": [0.0, 0.0, 60.0],
```

```
    "z": [10.0, 0.0, 30.0],
}

# after fix
a + b, b + a
{
    "x": [0.0, 40.0, 0.0],
    "y": [0.0, 0.0, 60.0],
    "z": [10.0, 0.0, 30.0],
}, {
    "x": [0.0, 40.0, 0.0],
    "y": [0.0, 0.0, 60.0],
    "z": [10.0, 0.0, 30.0],
}

a = ak.zip(
    {
        "x": [[1, 2], [], [3], [4]],
        "y": [[5, 6], [], [7], [8]],
        "z": [[9, 10], [], [11], [12]],
        "t": [[50, 51], [], [52], [53]],
    },
    with_name="LorentzVector",
    behavior=vector.behavior
)

# before fix
a.to_Vector3D()
# <MomentumArray3D [{rho: 10, phi: 0.1, ...}, ..., {...}]
# type='3 * Momentum3...'>

# after fix
a.to_Vector3D()
# <ThreeVectorArray [[{x: 1, y: 5, z: 9}, {...}], ...]
# type='4 * var * ThreeV...'>
```

Listing 4.3: Fixes and features introduced in Vector v1.2.

Vector v1.3 saw multiple new features, including Dask (Dask-Contrib) support for parallel computing, momentum coordinate support in coordinate transformation methods, and a new `like` method to help physicists adapt to a new strict promotion and demotion scheme for geometric coordinates. These updates were made to keep Vector compatible and in sync with the rest of the HEP ecosystem and to address new issues reported by physicists. Vector v1.3 included the following upgrades -

- feat: coordinate transformation functions with momentum names

- feat: allow momentum coords in to_VectorND methods + cleanup
- feat: like method for projecting vector into the coordinate space of another vector + better type errors and hints
- feat: add support for dask-awkward arrays in Vector constructors
- feat: short names for to_VectorND methods

4.4 shows the v1.3 updates and fixes in action.

```
import vector
import dask_awkward as dak

# coordinate support in momentum transformation methods
vec = vector.MomentumObject2D(pt=0, phi=1)
vec.to_ptphieta()

# momentum coordinate support in transformation methods
vec.to_Vector3D().to_Vector4D(M=4)

# short aliases
vec.to_4D()

# Dask support
x = dak.from_awkward(
    ak.Array(
     [{"x": 1, "y": 2}, {"x": 1.1, "y": 2.2}]
    ),
    npartitions=1
)
vec = vector.Array(x)

# the new like method
vec2 = vector.MomentumObject2D(pt=0, phi=1, eta=3)
vec + vec2.like(vec)
```

Listing 4.4: Fixes and features introduced in Vector v1.3.

v1.3.1 was a small release focusing on fixing issues with momentum vectors and supporting the latest dask-awkward version for distributed computing. The release included the following changes -

- feat: make momentum-ness infectious
- fix: support dask-awkward 2024.3.0
- fix: momentum coords should not be repeated with generic coords in subclasses

4.5 shows the v1.3.1 updates and fixes in action.

```
import vector
import awkward as ak
from coffee.nanoevents.methods import vector

v1 = vector.zip(
      {
          "px": [10.0, 20.0, 30.0],
          "py": [-10.0, 20.0, 30.0],
      },
 )
 v2 = vector.zip(
      {
          "x": [10.0, 20.0, 30.0],
          "y": [-10.0, 20.0, 30.0],
          "z": [5.0, 1.0, 1.0],
      },
 )

# infectious momentum-ness
# momentum + generic = momentum
# 2D + 3D.like(2D) = 2D
v1 + v2.like(v1)

a = ak.zip(
     {
          "pt": [[1, 2], [], [3], [4]],
          "eta": [[1.2, 1.4], [], [1.6], [3.4]],
          "phi": [[0.3, 0.4], [], [0.5], [0.6]],
          "energy": [[50, 51], [], [52], [60]],
     },
     with_name="PtEtaPhiELorentzVector",
     behavior=vector.behavior,
     )
b = ak.zip(
    {
          "rho": [10.0, 20.0, 30.0],
          "theta": [0.3, 0.6, 1.1],
          "phi": [-3.0, 1.1, 0.2],
    },
    with_name="SphericalThreeVector",
    behavior=vector.behavior,
)

# before fix
a.fields, a.like(b).fields
```

```
# ['pt', 'eta', 'phi', 'energy'],
# ['rho', 'phi', 'eta', 'pt', 'energy']

# after fix
# ['pt', 'eta', 'phi', 'energy'], ['rho', 'phi', 'eta']
```

Listing 4.5: Fixes and features introduced in Vector v1.3.1.

Vector v1.4 introduced SymPy as a new symbolic computation backend for the library. The backend is discussed in great detail in Chapter 6. Other than the new backend, the release also contained minor feature additions and bug fixes. More specifically, the release added the ability to pass momentum coordinates in coordinate transformation methods and fixed the implementation of squaring a vector.

- feat: a sympy backend
- feat: allow coord values in to_¡coord_names¿ methods
- fix: call the square implementation for power 2 on object vectors
- fix: use negfactor in negfactor scale test

4.6 shows the v1.4 updates and fixes in action.

```
import vector

# before fix
vector.array(
    {"E": [1], "px": [1], "py": [1], "pz": [1]}
) ** 2
# [-2.0]
vector.zip(
    {"E": [1], "px": [1], "py": [1], "pz": [1]},
) ** 2
# ak.Array([-2])
vector.obj(E=1, px=1, py=1, pz=1) ** 2
# 2.0000000000000004

# after fix
vector.array(
    {"E": [1], "px": [1], "py": [1], "pz": [1]}
) ** 2
# [-2.0]
vector.zip(
    {"E": [1], "px": [1], "py": [1], "pz": [1]},
) ** 2
# ak.Array([-2])
vector.obj(E=1, px=1, py=1, pz=1) ** 2
```

```
# -2

# coordinate support in momentum transformation methods
vec = vector.MomentumObject2D(pt=0, phi=1)
vec.to_ptphieta(eta=2)
```

Listing 4.6: Fixes and features introduced in Vector v1.4.

v1.4.1 was another small release that included multiple bug fixes for the SymPy, NumPy, and the Object backends of vector -

- fix: sympy backend on NumPy 2.0 (full NumPy 2.0 support)
- fix: add lower and upper bounds for deltaangle
- fix: maximum for SymPy backend is the identity function now
- fix: get coordinate classes to work for NumPy

4.7 shows the v1.4.1 updates and fixes in action.

```
import vector

v = vector.obj(x=1, y=1, z=1)

# before fix
v.deltaangle(v)
nan

# after fix
v.deltaangle(v)
0.0

# coordinate classes for NumPy vectors
vec4d = vector.VectorNumpy4D(
      {
          "x": [1.1, 1.2, 1.3, 1.4, 1.5],
          "y": [2.1, 2.2, 2.3, 2.4, 2.5],
          "z": [3.1, 3.2, 3.3, 3.4, 3.5],
          "t": [4.1, 4.2, 4.3, 4.4, 4.5],
      }
 )
vec4d.azimuthal, vec4d.longitudinal, vec4d.temporal
```

Listing 4.7: Fixes and features introduced in Vector v1.4.1.

Most of the updates made to Vector were computational, but several infrastructure, documentation, and maintenance changes were also added to the library. This thesis does not cover

non-computational changes but these changes are available in Vector's changelog. Moreover, all the upgrades and fixes in Vector were created to meet the expectations of its rapidly growing users. Vector is now much better prepared for the challenges posed by future upgrades to HEP experiments at CERN.

Chapter 5

Coffea's new backend for vector algebra

The existing Scikit-HEP ecosystem provides physicists with standalone libraries that can be used together in a script to achieve analysis of HEP data. This modularity as a design choice allows users to explore the libraries at a lower level, giving them complete control over their code. However, this modularity also makes the syntax alien-like and confusing for beginners. Moreover, the fundamental libraries of the ecosystem are as general as possible, allowing them to be used by physicists and other STEM people. This generalization makes it hard to integrate HEP or CERN experiment-specific schemas, vector classes, or data structures within the libraries. To combat this, Coffea provides essential tools and wrappers for enabling not-too-alien syntax when running columnar collider HEP analysis.

5.1 Coffea and its vector modules

Coffea makes use of Scikit-HEP libraries like Uproot and Awkward Arrays but also implements its own histogramming, plotting, and vector functionalities. For instance, 5.1 outlines a short example (Peruzzi et al. 2020) displaying how Coffea can be used in a fundamental HEP analysis. The example uses Uproot and Awkward Arrays underneath to read, write, and manipulate HEP data, but on the surface, it provides users with wrapper functionalities that can be extended to perform other niche physics tasks. Furthermore, it is possible to scale a HEP analysis to a large multi-core server, computing clusters, and super-computers with Coffea. The scaling is enabled via other IRIS-HEP projects as well as the Dask backends of the array and histogram libraries of Scikit-HEP.

```
import awkward as ak
from coffea.nanoevents import NanoEventsFactory, NanoAODSchema

NanoAODSchema.warn_missing_crossrefs = False

fname = (
    "https://raw.githubusercontent.com/CoffeaTeam/coffea/" +
    "master/tests/samples/nano_dy.root"
)
events = NanoEventsFactory.from_root(
```

```
    {fname: "Events"},
    schemaclass=NanoAODSchema,
    metadata={"dataset": "DYJets"},
).events()

# compute the energy of GenJet
events.GenJet.energy.compute()

# find distance between leading jet and all electrons in each event
dr = events.Jet[:, 0].delta_r(events.Electron)
dr.compute()

# find minimum distance
ak.min(dr, axis=1).compute()

# computing the pt coordinate of Muons in events with
# 1 or more Muons
events.Muon[ak.num(events.Muon)>0].matched_jet.pt.compute()

# choose events with exactly 2 muons
# sum along the axis and compute the mass
mmevents = events[ak.num(events.Muon) == 2]
zmm = mmevents.Muon[:, 0] + mmevents.Muon[:, 1]
zmm.mass.compute()
```

Listing 5.1: A short analysis example using Coffea.

Coffea has deprecated its plotting and histogramming code since Scikit-HEP now consists of HEP-specific plotting and histogramming libraries. Coffea now depends on these libraries and extends their functionalities internally for the HEP data analysis use case.

5.2 Coffea's vector to Scikit-HEP/vector

Coffea's vector module pre-dates Scikit-HEP/vector, but now that Vector has achieved maturity, it made sense to migrate Coffea's internals to Scikit-HEP/vector. Scikit-HEP/vector is now much more sophisticated and functional than Coffea's vector sub-package, including support for third-party libraries, such as JAX, Dask, and SymPy (Meurer et al. 2017). Although Scikit-HEP/vector does not offer as many HEP functionalities as the current vector modules of Coffea, the basic HEP functionalities offered by Vector can be extended by inheriting the `VectorAwkward` mixin classes.

The migration will ensure that the IRIS-HEP ecosystem does not provide users with repetitive code or code that looks similar but performs differently. Additionally, the functionality and conventional differences between Coffea vector modules and Scikit-HEP/vector should be minimized to give users a single interface with the best of both worlds. Further, this will also unite physicists,

given that some were currently using Scikit-HEP/vector in their data analysis pipeline, and the remaining were using the internal vector modules of Coffea.

5.3 Implementing Scikit-HEP/vector as a backend for Coffee's vector classes

Bringing together both codebases to agree on a single convention started several discussions in both Scikit-HEP and Coffea. These discussions formalized multiple changes in Vector and Awkward Arrays such that adopting Vector as a vector algebra backend in Coffea can be as smooth as possible. The niche HEP-specific functionalities were agreed to be kept in Coffea; hence, Coffea could depend on Vector and extend its functionality by inheriting the Vector mixins or adding functions that accept Vector objects as arguments. Ultimately, Coffea was scheduled to deprecate its vector modules entirely and only depend on Vector to create and manipulate vectors.

The switch from Coffea's internal vector modules to Scikit-HEP/vector was a drastic and breaking change that was supposed to be rolled out in parts to let physicists migrate their data pipelines. Therefore, in the initial phase, Coffea vector classes were made to inherit Scikit-HEP/vector's mixin classes, allowing developers to remove Coffea vector methods and depend on the superclass's methods. The inheritance approach kept the Coffea's vector interface intact, giving more time to the physicists. Figure 5.1 describes how the migration was carried out internally. The migration also added a new feature in Awkward, the `copy_behaviors` function. The HEP-specific vector classes in Coffea required registering Awkward *ufuncs* for each of them individually. This process involved repetitive code, given that Coffea consists of multiple vector classes for multiple use cases. The passing down of behavior and *ufuncs* to subclasses was also an open issue in Awkward; hence, a new function was introduced to copy the *ufuncs* of a superclass to the subclass entries in the `behavior` dictionary.

Figure 5.1: Switching Coffea's vector algebra backend.

Examples 5.2 and 5.3 show how seamlessly one can switch from Coffea's vector modules to Scikit-HEP/vector without losing performance or any functionality.

```
import awkward as ak
from coffea.nanoevents.methods import vector
from coffea.nanoevents import NanoEventsFactory, BaseSchema
```

```
filename = "file://Run2012B_DoubleMuParked.root"
events = NanoEventsFactory.from_root(
    {filename: "Events"},
    steps_per_file=2_000,
    metadata={"dataset": "DoubleMuon"},
    schemaclass=BaseSchema,
).events()

muons = ak.zip(
    {
        "pt": events.Muon_pt,
        "eta": events.Muon_eta,
        "phi": events.Muon_phi,
        "mass": events.Muon_mass,
    },
    with_name="LorentzVector",
    behavior=vector.behavior,
)
```

Listing 5.2: Constructing a 4D momentum vector with Coffea.

```
import awkward as ak
import vector
from coffea.nanoevents import NanoEventsFactory, BaseSchema

filename = "file://Run2012B_DoubleMuParked.root"
events = NanoEventsFactory.from_root(
    {filename: "Events"},
    steps_per_file=2_000,
    metadata={"dataset": "DoubleMuon"},
    schemaclass=BaseSchema,
).events()

muons = vector.zip(
    {
        "pt": events.Muon_pt,
        "eta": events.Muon_eta,
        "phi": events.Muon_phi,
        "mass": events.Muon_mass,
    }
)
```

Listing 5.3: Creating a 4D momentum vector with Scikit-HEP/vector

The internal migration of Coffea's vector algebra backend is complete but is still under review. Coffea's interface will not change after the internal migration, making the old Coffea code perfectly

valid. After the internal switch, the next step will be to remove Coffea's vector modules and redirect physicists to use Scikit-HEP/vector in their data analysis pipelines.

Chapter 6

Vector's symbolic backend for theoretical computations

6.1 Symbolic programming and SymPy

Symbolic programming is a programming paradigm in which computer systems operate on formulas and data containers without interacting with numerical data. This type of programming is used within the theoretical Sciences and Mathematics to write proofs, solve equations, or even differentiate expressions using pre-defined rules. Multiple programming languages, such as Lisp (Steele 1990) and Mathematica (Inc.), are explicitly designed for symbolic programming. Other languages often have third-party libraries or packages for symbolic programming, such as SymPy in Python and Symbolics.jl (Gowda et al. 2022) in Julia.

Focusing on the Python ecosystem, SymPy is the most widely used computer algebra system (CAS) by scientists and programmers. SymPy has a full suite of symbolic operations and is maintained by a vast community. The operations and functionalities include symbolic calculus, combinatorics, discrete mathematics, geometry, physics, statistics, and even printing mathematical expressions in LaTeX. Since SymPy is written in pure Python, is well maintained, and is widely used in the Python ecosystem, it was the top choice for implementing a symbolic backend in Vector.

6.2 Need for a symbolic backend

The Scikit-HEP ecosystem is primarily meant to be used by experimental physicists to manipulate and perform physics on numerical data. Theoretical physicists are largely aloof from Scikit-HEP and other experimental Physics frameworks. Moreover, only MC generators like Pythia (Sjöstrand et al. 2015) are routinely used by both experimental and theoretical physicists.

A new SymPy backend in Vector (Pivarski & Chopra 2024a) (Pivarski & Chopra 2024b) will allow symbolic computations on HEP vectors. Along with experimental physicists using Vector for numerical computations, the SymPy backend will enable theoretical physicists to utilize the library for symbolic computations. Since the SymPy vector classes and their momentum equivalents will operate on SymPy expressions, all of the standard SymPy methods and functions will work on the

vectors, vector coordinates, and the results of operations carried out on vectors. Moreover, Vector's SymPy backend will create a stronger connection between software used by experimentalists and software used by theorists.

Furthermore, Vector's compute functions were written to operate only on data containers and not the numerical values of the data containers. The symbolic behavior of Vector's compute functions is tested in the continuous integration pipeline using uncompyle6. Given that uncompyle6 is supported till Python 3.8, which is reaching its end of life soon, finding other methods to test the specialized behavior of compute functions became important. The tests of the SymPy backend will allow vector developers to remove the uncompyle6 dependency because running the compute functions on SymPy vectors will ensure that they are operating only on the data containers.

6.3 Implementing a symbolic backend in Vector

Vector was designed from the ground-up to have multiple computational backends. The duck typing of compute functions allows them to be shared within the backends without introducing repetitive code. Implementing the SymPy backend included adding coordinate and vector classes capable of constructing vectors using SymPy's data containers and wrapping compute function results as a SymPy expression.

Figure 6.1 shows how the SymPy backend and other Vector backends interact with the computing functions. Each backend has its own coordinate and vector classes that can accept numerical (for the case of Object/NumPy/Awkward backend) or symbolic (for the case of SymPy backend) arguments. The classes, as well as the method written directly under them, are compatible with the respective backend libraries. The Object backend uses NumPy to perform all the arithmetic and Awkward functions work on NumPy vectors without any performance degradation.

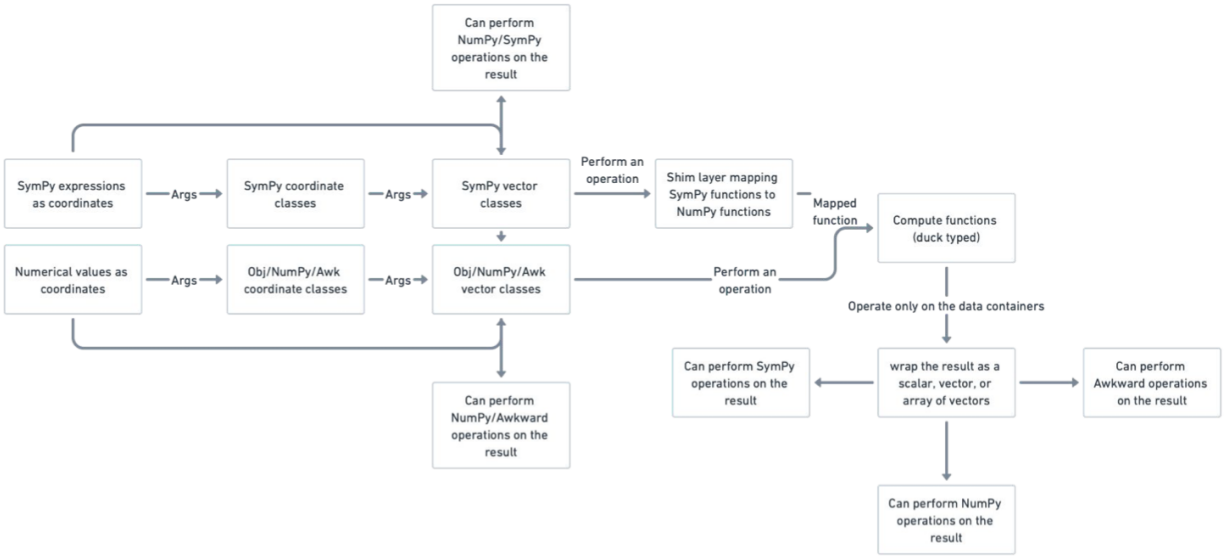

Figure 6.1: Implementation of Vector's SymPy backend.

Finally, all the compute functions valid on the dimension of the constructed vector work with all the backends. At the moment, the compute functions switch between backend libraries using a shim layer. This shim layer is not required for the Object, NumPy, and Awkward backends because NumPy works with all of them. On the other hand, due to different naming conventions between SymPy and NumPy, the NumPy functions are mapped to the respective SymPy functions in the shim layer and are flown down to the compute functions. The results from these compute functions are wrapped with values or appropriate data structures in the vector classes. This wrapped result can then be used in any of the functions provided by the backend libraries, making a strong cohesion between Vector backends and the backend libraries.

Consider the example 6.1 performing `deltaR` operation on two Object type 4D Momentum vectors.

```
import vector

muon_1_obj = vector.MomentumObject4D(px=1, py=2, pz=3, E=10)
muon_2_obj = vector.MomentumObject4D(px=2, py=3, pz=4, E=11)

muon_1_obj.deltaR(muon_2_obj)
# 0.19249147660266414
```

Listing 6.1: Performing deltaR on Object vectors.

The exact same operation can be carried out using the SymPy backend with an almost identical syntax. 6.2 shows how SymPy `symbols` can be passed into `MomentumSymPy` constructors as arguments just like numerical values are passed into `MomentumObject` constructors. The `deltaR` operation on SymPy vector returns a SymPy expression instead of a numerical value. The obtained SymPy expression is compatible with every SymPy method and function; hence, one can substitute (`subs`) and evaluate (`evalf`) the resultant expression to validate the theoretical expression.

```
import vector; import sympy

px_1, py_1, pz_1, E_1 = sympy.symbols(
    "px_1 py_1 pz_1 E_1", real=True
)
px_2, py_2, pz_2, E_2 = sympy.symbols(
    "px_2 py_2 pz_2 E_2", real=True
)

muon_1_sympy = vector.MomentumSympy4D(
    px=px_1, py=py_1, pz=pz_1, E=E_1
)
muon_2_sympy = vector.MomentumSympy4D(
    px=px_2, py=py_2, pz=pz_2, E=E_2
)
```

```
muon_1_sympy.deltaR(muon_2_sympy)
# sqrt((Mod(atan2(py_1, px_1) - atan2(py_2, px_2) + pi, 2*pi) -
# pi)**2 + (asinh(pz_1/sqrt(px_1**2 + py_1**2))
# - asinh(pz_2/sqrt(px_2**2 + py_2**2)))**2)

# take the values from object type vectors
muon_1_sympy.deltaR(muon_2_sympy).subs(
    {
        px_1: muon_1_obj.px,
        py_1: muon_1_obj.py,
        pz_1: muon_1_obj.pz,
        E_1: muon_1_obj.E,
        px_2: muon_2_obj.px,
        py_2: muon_2_obj.py,
        pz_2: muon_2_obj.pz,
        E_2: muon_2_obj.E,
    }
).evalf()
# 0.192491476602664
```

Listing 6.2: Performing deltaR on SymPy vectors.

The SymPy backend has two significant and intentional caveats. SymPy internally uses mpmath to perform complex floating-point arithmetic, which has led to minor disagreements between the results obtained through the Object and the SymPy backend. This disagreement can be minimized by specifying more decimal points in the precision. Further, operations on SymPy vectors are only 100% compatible with numeric vectors (Python, NumPy, and Awkward backends) if the vectors are positive time-like, that is, if -

$$t^2 > x^2 + y^2 + z^2$$

The space-like and negative time-like cases have different sign conventions; hence, to make SymPy's simplification work, these sign conventions are ignored in the shim layer. Given that most of the HEP analysis deals with positive time-like vectors, this caveat does not hinder the ability to use the Vector's SymPy backend in theoretical calculations.

Chapter 7

Bringing histograms to GPUs: cuda-histogram

Awkward Arrays provides NumPy-like functionality for jagged data produced by HEP experiments. The library intertwines naturally with several fundamental scientific computing Python libraries, allowing physicists to utilize the entire ecosystem to its full potential. The support for third-party libraries enables physicists to perform various tasks on their analysis pipelines, including, but not limited to, differentiation (JAX), parallelization (Dask), and just-in-time compilation (Numba). There are ongoing efforts to make Awkward Arrays compatible with CUDA, allowing physicists to accelerate their analysis using GPUs. The recent work of putting Awkward Arrays on GPUs has garnered interest from high energy physicists, especially Coffea developers, to accelerate the analysis framework for future HL-LHC runs. Though the work on Awkward Arrays is being carried out, more pieces are required to perform a complete analysis of HEP data on GPUs. One of the major pieces is the ability to generate and manipulate histograms on CUDA, allowing physicists to perform a complete analysis leveraging the entire Scikit-HEP ecosystem on GPUs.

Before this thesis, Coffea had a prototype implementation of histograms on CUDA, but the implementation was specific to Coffea. This thesis aimed to generalize and broaden the existing implementation of histograms on CUDA by developing a UHI-compliant stand-alone library – cuda-histogram. Furthermore, Coffea has been trying to incorporate broader libraries to replace its internal, more focused, modules. One such instance is the ongoing migration of Coffea's vector classes to Scikit-HEP/vector. This project will tie up with keeping the cuda-histogram functionality outside of Coffea for a broader audience.

At the time of writing this thesis, the cuda-histogram project is still in progress and under scrutiny. In the upcoming months, the project will be refined, made UHI-compatible, transformed into a standalone Scikit-HEP package, go through hyperoptimization routines, and finally used within Coffea as a dependency.

Bibliography

1962. The CERN proton synchrotron.

2021. *Building and steering template fits with cabinetry*. Zenodo. URL `https://doi.org/10.5281/zenodo.4627038`.

AAD, G., ET AL. 2008. The ATLAS Experiment at the CERN Large Hadron Collider. *JINST* 3.S08003.

AAMODT, K., ET AL. 2008. The ALICE experiment at the CERN LHC. *JINST* 3.S08002.

ABADI, MARTÍN; ASHISH AGARWAL; PAUL BARHAM; EUGENE BREVDO; ZHIFENG CHEN; CRAIG CITRO; GREG S. CORRADO; ANDY DAVIS; JEFFREY DEAN; MATTHIEU DEVIN; SANJAY GHEMAWAT; IAN GOODFELLOW; ANDREW HARP; GEOFFREY IRVING; MICHAEL ISARD; YANGQING JIA; RAFAL JOZEFOWICZ; LUKASZ KAISER; MANJUNATH KUDLUR; JOSH LEVENBERG; DANDELION MANÉ; RAJAT MONGA; SHERRY MOORE; DEREK MURRAY; CHRIS OLAH; MIKE SCHUSTER; JONATHON SHLENS; BENOIT STEINER; ILYA SUTSKEVER; KUNAL TALWAR; PAUL TUCKER; VINCENT VANHOUCKE; VIJAY VASUDEVAN; FERNANDA VIÉGAS; ORIOL VINYALS; PETE WARDEN; MARTIN WATTENBERG; MARTIN WICKE; YUAN YU; und XIAOQIANG ZHENG. 2015. TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org. URL `https://www.tensorflow.org/`.

AGOSTINELLI, S., ET AL. 2003. GEANT4: A simulation toolkit. *Nucl. Instrum. Meth.* A506.250–303.

ALVES, A. AUGUSTO, JR., ET AL. 2008. The LHCb Detector at the LHC. *JINST* 3.S08005.

ANSEL, JASON; EDWARD YANG; HORACE HE; NATALIA GIMELSHEIN; ANIMESH JAIN; MICHAEL VOZNESENSKY; BIN BAO; PETER BELL; DAVID BERARD; EVGENI BUROVSKI; GEETA CHAUHAN; ANJALI CHOURDIA; WILL CONSTABLE; ALBAN DESMAISON; ZACHARY DEVITO; ELIAS ELLISON; WILL FENG; JIONG GONG; MICHAEL GSCHWIND; BRIAN HIRSH; SHERLOCK HUANG; KSHITEEJ KALAMBARKAR; LAURENT KIRSCH; MICHAEL LAZOS; MARIO LEZCANO; YANBO LIANG; JASON LIANG; YINGHAI LU; CK LUK; BERT MAHER; YUNJIE PAN; CHRISTIAN PUHRSCH; MATTHIAS RESO; MARK SAROUFIM; MARCOS YUKIO SIRAICHI; HELEN SUK; MICHAEL SUO; PHIL TILLET; EIKAN WANG; XIAODONG WANG; WILLIAM WEN; SHUNTING ZHANG; XU ZHAO; KEREN ZHOU; RICHARD ZOU; AJIT MATHEWS; GREGORY CHANAN; PENG WU; und SOUMITH CHINTALA. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. *29th acm international conference on architectural support for*

programming languages and operating systems, volume 2 (asplos '24). ACM. URL https://pytorch.org/assets/pytorch2-2.pdf.

BERTONE, C., ET AL. 2011. The Linac4 Project at CERN. *Conf. Proc. C* 110904.900–902.

BEZANSON, JEFF; ALAN EDELMAN; STEFAN KARPINSKI; und VIRAL B SHAH. 2017. Julia: A fresh approach to numerical computing. *SIAM review* 59.65–98. URL https://doi.org/10.1137/141000671.

BOCKELMAN, BRIAN; WLCG/HSF; FERMILAB; MORGRIDGE; NOTRE DAME; PRINCETON; U. CHICAGO; NATIONAL CENTER FOR SUPERCOMPUTING APPLICATIONS; U. NEBRASKA; UT-AUSTIN; U. WASHINGTON; U. WISCONSIN; MORGRIDGE; IRIS-HEP; WLCG; USATLAS; USCMS; U. CHICAGO; CERN; ATLAS; CMS; ATLAS PHYSLITE; CMS NANOAOD; und XCACHE. IRIS-HEP 200Gbps challenge. Tech. rep. URL https://indico.cern.ch/event/1369601/contributions/5924000/attachments/2856630/4998936/IRIS-HEP%20200Gbps%20-%20WLCG%20Workshop%20-%20v1.pdf.

BOOST. 2015. Boost C++ Libraries. http://www.boost.org/. Last accessed 2024-07-08.

BRADBURY, JAMES; ROY FROSTIG; PETER HAWKINS; MATTHEW JAMES JOHNSON; CHRIS LEARY; DOUGAL MACLAURIN; GEORGE NECULA; ADAM PASZKE; JAKE VANDERPLAS; SKYE WANDERMAN-MILNE; und QIAO ZHANG. 2018. JAX: composable transformations of Python+NumPy programs. URL http://github.com/google/jax.

BRUN, RENE; 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; WLAV; ENRIC TEJEDOR; JAKOB BLOMER; ANDREI GHEATA; STEPHAN HAGEBOECK; STEFAN ROISER; MARSUPIAL; STEFAN WUNSCH; OKSANA SHADURA; ANIRUDHA BOSE; CRISTINACRISTESCU; XAVIER VALLS; und RAPHAEL ISEMANN. 2020. root-project/root: v6.18/02. URL https://doi.org/10.5281/zenodo.3895860.

CHATRCHYAN, S., ET AL. 2008. The CMS Experiment at the CERN LHC. *JINST* 3.S08004.

CHEN, YI-MU. [FEATURE] Non-uniform rebinning · Issue 345 · scikit-hep/hist. URL https://github.com/scikit-hep/hist/issues/345.

DASK-CONTRIB. GitHub - dask-contrib/dask-awkward: Native Dask collection for awkward arrays, and the library to use it. URL https://github.com/dask-contrib/dask-awkward.

DASK DEVELOPMENT TEAM. 2016. *Dask: Library for dynamic task scheduling*. URL http://dask.pydata.org.

DAWE, NOEL; PETER WALLER; EVAN K. FRIIS; CHRISTOPH DEIL; SCHMITTS; RUGGERO TURRA; JEFF KLUKAS; SCOTT STEVENSON; QUENTIN BUAT; CHRISBOO; ALESSANDRO; MAURO VERZETTI; LUKE KRECZKO; JEROEN HEGEMAN; und MATT HOLLINGSWORTH. 2015. rootpy: 0.8.0. URL https://doi.org/10.5281/zenodo.18897.

DEMBINSKI, HANS, und PITI ONGMONGKOLKUL ET AL. 2020. scikit-hep/iminuit. URL https://doi.org/10.5281/zenodo.3949207.

DOBLE, NIELS; LAU GATIGNON; KURT HÜBNER; und EDMUND WILSON. 2017. The Super Proton Synchrotron (SPS): A Tale of Two Lives. *Adv. Ser. Direct. High Energy Phys.* 27.135–177.

ELMER, PETER; MARK NEUBAUER; und MICHAEL D. SOKOLOFF. 2018. Strategic plan for a scientific software innovation institute (s2i2) for high energy physics. URL https://arxiv.org/abs/1712.06592.

EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH, und OPENAIRE. 2013. Zenodo. URL https://www.zenodo.org/.

FEIKERT, MATTHEW. Slack. URL https://iris-hep.slack.com/archives/C025BUK9V16/p1705438456376999.

GOWDA, SHASHI; YINGBO MA; ALESSANDRO CHELI; MAJA GWÓŹZDŹ; VIRAL B. SHAH; ALAN EDELMAN; und CHRISTOPHER RACKAUCKAS. 2022. High-performance symbolic-numerics via multiple dispatch. *ACM Commun. Comput. Algebra* 55.92–96. URL https://doi.org/10.1145/3511528.3511535.

GRADHEP. gradHEP. URL https://github.com/gradhep.

GRAY, LINDSEY; NICHOLAS SMITH; ANDRZEJ NOVAK; PETER FACKELDEY; BENJAMIN TOVAR; YI-MU CHEN; GORDON WATTS; und IASON KROMMYDAS. 2023. coffea. URL https://github.com/CoffeaTeam/coffea.

GUEST, DAN; KYLE CRANMER; und DANIEL WHITESON. 2018. Deep Learning and its Application to LHC Physics. *Ann. Rev. Nucl. Part. Sci.* 68.161–181.

HARRIS, CHARLES R.; K. JARROD MILLMAN; STÉFAN J. VAN DER WALT; RALF GOMMERS; PAULI VIRTANEN; DAVID COURNAPEAU; ERIC WIESER; JULIAN TAYLOR; SEBASTIAN BERG; NATHANIEL J. SMITH; ROBERT KERN; MATTI PICUS; STEPHAN HOYER; MARTEN H. VAN KERKWIJK; MATTHEW BRETT; ALLAN HALDANE; JAIME FERNÁNDEZ DEL RÍO; MARK WIEBE; PEARU PETERSON; PIERRE GÉRARD-MARCHANT; KEVIN SHEPPARD; TYLER REDDY; WARREN WECKESSER; HAMEER ABBASI; CHRISTOPH GOHLKE; und TRAVIS E. OLIPHANT. 2020. Array programming with NumPy. *Nature* 585.357–362. URL https://doi.org/10.1038/s41586-020-2649-2.

HEINRICH, LUKAS; MATTHEW FEICKERT; und GIORDON STARK. pyhf: v0.7.6. URL https://github.com/scikit-hep/pyhf/releases/tag/v0.7.6.

HELD, ALEXANDER. a. Add rebinning functionality · Issue 412 · scikit-hep/cabinetry. URL https://github.com/scikit-hep/cabinetry/issues/412.

HELD, ALEXANDER. b. GitHub - alexander-held/agc-autodiff: AD for the Analysis Grand Challenge. URL https://github.com/alexander-held/agc-autodiff.

HELD, ALEXANDER; OKSANA SHADURA; MATTHEW FEICKERT; JAYJEET CHAKRABORTY; MASON PROFFITT; KYUNGEON CHOI; ANDRZEJ NOVAK; DAVID KOCH; MAT ADAMEC; SARANSH CHOPRA; und STORM LIN. 2022. iris-hep/analysis-grand-challenge: v0.1.0. URL https://doi.org/10.5281/zenodo.7274937.

INC., WOLFRAM RESEARCH,. Mathematica, Version 14.0. Champaign, IL, 2024. URL https://www.wolfram.com/mathematica.

INNES, MICHAEL. 2018. Don't unroll adjoint: Differentiating ssa-form programs. *CoRR* abs/1810.07951. URL http://arxiv.org/abs/1810.07951.

KOCH, LUKAS; HENRY SCHREINER; EDUARDO RODRIGUES; MICHAEL HALL; und MATTHEW FEICKERT. 2022. scikit-hep/histoprint: v2.4.0. URL https://doi.org/10.5281/zenodo.6600707.

LAM, SIU KWAN; ANTOINE PITROU; und STANLEY SEIBERT. 2015. Numba: a llvm-based python jit compiler. *Proceedings of the second workshop on the llvm compiler infrastructure in hpc*, LLVM '15. New York, NY, USA: Association for Computing Machinery. URL https://doi.org/10.1145/2833157.2833162.

MEURER, AARON; CHRISTOPHER P. SMITH; MATEUSZ PAPROCKI; ONDŘEJ ČERTÍK; SERGEY B. KIRPICHEV; MATTHEW ROCKLIN; AMIT KUMAR; SERGIU IVANOV; JASON K. MOORE; SARTAJ SINGH; THILINA RATHNAYAKE; SEAN VIG; BRIAN E. GRANGER; RICHARD P. MULLER; FRANCESCO BONAZZI; HARSH GUPTA; SHIVAM VATS; FREDRIK JOHANSSON; FABIAN PEDREGOSA; MATTHEW J. CURRY; ANDY R. TERREL; ŠTĚPÁN ROUČKA; ASHUTOSH SABOO; ISURU FERNANDO; SUMITH KULAL; ROBERT CIMRMAN; und ANTHONY SCOPATZ. 2017. Sympy: symbolic computing in python. *PeerJ Computer Science* 3.e103. URL https://doi.org/10.7717/peerj-cs.103.

PASZKE, ADAM; SAM GROSS; FRANCISCO MASSA; ADAM LERER; JAMES BRADBURY; GREGORY CHANAN; TREVOR KILLEEN; ZEMING LIN; NATALIA GIMELSHEIN; LUCA ANTIGA; ALBAN DESMAISON; ANDREAS KOPF; EDWARD YANG; ZACHARY DEVITO; MARTIN RAISON; ALYKHAN TEJANI; SASANK CHILAMKURTHY; BENOIT STEINER; LU FANG; JUNJIE BAI; und SOUMITH CHINTALA. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems 32*, 8024–8035. Curran Associates, Inc. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

PERUZZI, MARCO; GIOVANNI PETRUCCIANI; ANDREA RIZZI; und FOR THE CMS COLLABORATION. 2020. The nanoaod event data format in cms. *Journal of Physics: Conference Series* 1525.012038. URL https://dx.doi.org/10.1088/1742-6596/1525/1/012038.

PIVARSKI, JIM. Awkward array hierarchy example. URL https://github.com/scikit-hep/awkward/blob/main/docs-img/diagrams/example-hierarchy.png.

PIVARSKI, JIM, und SARANSH CHOPRA. 2024a. A new sympy backend for vector: uniting experimental and theoretical physicists. *Proceedings of the 7th python in high energy physics workshop*.

PIVARSKI, JIM, und SARANSH CHOPRA. 2024b. A new sympy backend for vector: uniting experimental and theoretical physicists. *Proceedings of the 27th international conference on computing in high energy physics*.

PIVARSKI, JIM; LUKAS; EDUARDO RODRIGUES; DMITRY KALINKIN; und CLEMENS LANGE. 2018a. scikit-hep/histbook: 1.2.5. URL https://doi.org/10.5281/zenodo.1478738.

PIVARSKI, JIM; IANNA OSBORNE; IOANA IFRIM; HENRY SCHREINER; ANGUS HOLLANDS; ANISH BISWAS; PRATYUSH DAS; SANTAM ROY CHOUDHURY; NICHOLAS SMITH; und MANASVI GOYAL. 2018b. Awkward Array.

PIVARSKI, JIM; HENRY SCHREINER; ANGUS HOLLANDS; PRATYUSH DAS; KUSH KOTHARI; ARYAN ROY; JERRY LING; NICHOLAS SMITH; CHRIS BURR; und GIORDON STARK. 2017. Uproot.

REICH, K H. The cern proton synchrotron booster. *IEEE (Inst. Elec. Electron. Eng.), Trans. Nucl. Sci., NS-16: 959- 61(June 1969)*. URL https://www.osti.gov/biblio/4773934.

RODRIGUES, EDUARDO, und HENRY SCHREINER. a. DecayLanguage. URL https://github.com/scikit-hep/decaylanguage.

RODRIGUES, EDUARDO, und HENRY SCHREINER. b. Particle. URL https://github.com/scikit-hep/particle.

RODRIGUES, EDUARDO, ET AL. 2020. The Scikit HEP Project – overview and prospects. *EPJ Web Conf.* 245.06028.

ROSADO, TIAGO, und JORGE BERNARDINO. 2014. An overview of openstack architecture. *Proceedings of the 18th international database engineering & applications symposium*, IDEAS '14, 366–367. New York, NY, USA: Association for Computing Machinery. URL https://doi.org/10.1145/2628194.2628195.

SCHREINER, HENRY. a. Full UHI · Issue 208 · scikit-hep/boost-histogram. URL https://github.com/scikit-hep/boost-histogram/issues/208.

SCHREINER, HENRY. b. GitHub - henryiii/hepvector: Redesigned as Scikit-HEP: vector! URL https://github.com/henryiii/hepvector.

SCHREINER, HENRY; HANS DEMBINSKI; AMAN GOEL; JAY GOHIL; N!NO; JONAS ESCHLE; CHANCHAL MAJI; ANDRZEJ NOVAK; CHRIS BURR; DOUG DAVIS; KILIAN LIERET; KONSTANTIN GIZDOV; KYLE CRANMER; MATTHEW FEICKERT; und PIERRE GRIMAUD. 2023a. scikit-hep/boost-histogram: Version 1.4.0. URL https://doi.org/10.5281/zenodo.8336454.

SCHREINER, HENRY; HANS DEMBINSKI; KILIAN LIERET; und PIETER DAVID. 2023b. scikit-hep/uhi: Version 0.4.0. URL https://doi.org/10.5281/zenodo.10014713.

SCHREINER, HENRY; SHUO LIU; und AMAN GOEL. a. hist. URL https://github.com/scikit-hep/hist.

SCHREINER, HENRY; JIM PIVARSKI; und SARANSH CHOPRA. b. vector. URL https://github.com/scikit-hep/vector.

SCIKIT-HEP. GitHub - scikit-hep/aghast: Aghast: aggregated, histogram-like statistics, sharable as Flatbuffers. URL https://github.com/scikit-hep/aghast.

SIMPSON, NATHAN. 2023a. Data Analysis in High-Energy Physics as a Differentiable Program. Presented 03 Feb 2023. URL https://cds.cern.ch/record/2846434.

SIMPSON, NATHAN. 2023b. relaxed: version 0.3.0. URL https://github.com/gradhep/relaxed.

SIMPSON, NATHAN, und LUKAS HEINRICH. 2021. neos: version 0.2.0. URL https://github.com/gradhep/neos.

SJÖSTRAND, TORBJÖRN; STEFAN ASK; JESPER R. CHRISTIANSEN; RICHARD CORKE; NISHITA DESAI; PHILIP ILTEN; STEPHEN MRENNA; STEFAN PRESTEL; CHRISTINE O. RASMUSSEN; und PETER Z. SKANDS. 2015. An introduction to pythia 8.2. *Computer Physics Communications* 191.159–177. URL http://dx.doi.org/10.1016/j.cpc.2015.01.024.

STEELE, GUY. 1990. *Common lisp: the language*. Elsevier.

VASSILEV, V.; M. VASSILEV; A. PENEV; L. MONETA; und V. ILIEVA. 2015. Clad – Automatic Differentiation Using Clang and LLVM. Ausg. 608, 012055. IOP Publishing. URL https://iopscience.iop.org/article/10.1088/1742-6596/608/1/012055/pdf.