Exploring Different Data Analysis Approaches in EDM4hep

Konrad Helms,

Georg-August-Universität Göttingen,

Germany

September 7, 2022

Abstract

In the following report, eight analysis approaches for a Higgs recoil mass calculation in $e^+e^- \to Z^* \to ZH \to H\ell^+\ell^-$ events are presented. The analyses are performed using conventional event loops, RDataFrames, or Uproot. Afterwards, the different approaches are benchmarked and a representative runtime analysis of a python RDataFrame script is presented. Overall, the c++ macro using RDataFrames performs best in terms of runtime. In python, Uproot and RDataFrames show similar levels of performance. Lastly, several aspects of usability for the different approaches are discussed.

Contents

1. Introduction **3**

2. Software Framework **3**

3. Theory **4**
3.1. Physics 5
3.2. Amdahl's Law 5

4. Analysis Approaches **6**
4.1. Python 6
4.2. C++ 7
4.3. Performance Analysis 7
4.4. Runtime Analysis 9

5. Summary **10**

A. Runtimes **12**

1. Introduction

Since the Higgs boson's discovery in 2012 [1, 2], precision measurements of its properties were brought into focus. With the yet to be build Higgs factory, the International Linear Collider (ILC), software development on a new simulation, experiment orchestration and analysis framework began.

In modern high energy physics (HEP) experiments, computational analysis is the key to finding new physics, as well as doing precision measurements or probing theoretical predictions. When working with recorded or simulated data, a reliable and robust software framework is necessary. Additionally, the framework should deliver easy-to-use and fast tools to analyse the given data.

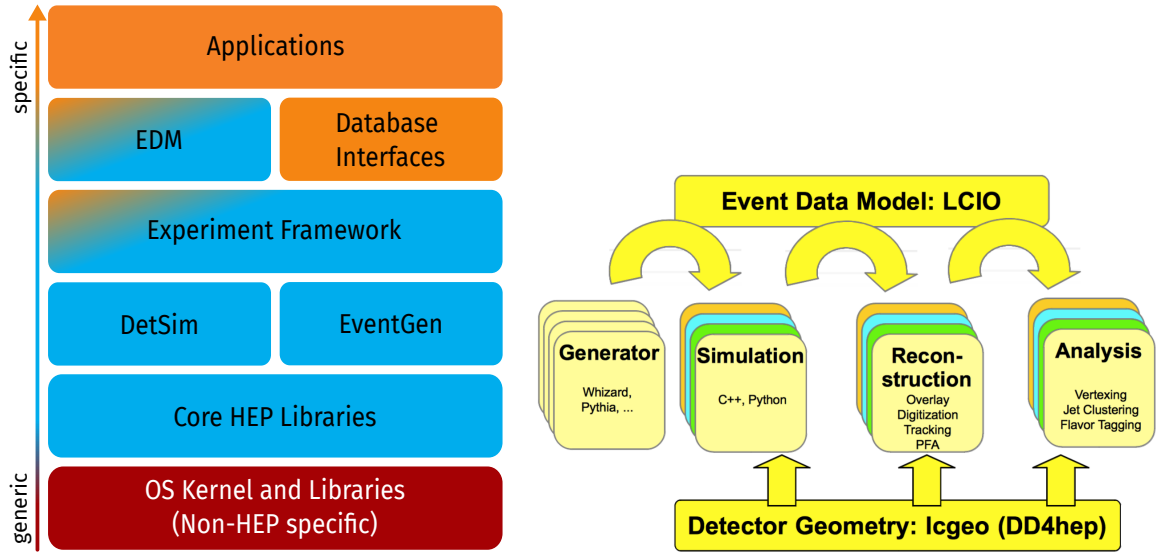
In the following, an example analysis calculating the Higgs recoil mass in an $e^+e^- \to Z^* \to ZH \to H\ell^+\ell^-$ process, see Fig. 2, at a resonance energy of 250 GeV is presented using eight different analysis approaches. This process can be measured with high precision and low background at the ILC. Lastly, the different approaches are benchmarked and analysed in terms of runtime as well as optimised in terms of their performance.

2. Software Framework

Modern high energy physics has to face a number of challenges. Since the lifetime of experiments is planned to be in the order of several decades, new technological developments could lead to a shift in priorities or lead to new paradigms. With long lifetimes come large amounts of data that need to be preserved and stored in a format that ensures longevity. Furthermore, as resources are limited, the usage of such should be minimised, therefore computing efforts need to be optimised. Historically, each research facility has their own software framework. Today, the general idea is to unify the scattered landscape of software frameworks and to create one ecosystem that can be used by every future HEP experiment.

The set of software packages to do event and detector simulation as well as data analysis and storage in high energy physics is called HEP software stack. A representative figure showing the layered structure of the software stack can be seen in Fig. 1 (a).

The base layer, that is the most generic layer, in the HEP software stack consists of the operating system (OS) and commonly used, non-HEP specific, libraries and tools, such as different programming languages, compilers etc. Following the base layer, the core HEP libraries that provide generic functionality are included. The data analysis framework *ROOT*[3], or the detector simulation platform *Geant4*[4, 5, 6] are for instance part of this layer. Next is a more specific layer that combines and extends the previous, consisting of components that are still shared by many experiments. Event generation software like *Pythia*[7, 8] could for example be part of this layer. The next



(a) Layers of a HEP software stack. ©Thomas Madlener

(b) Processing chain of a HEP experiment. ©Thomas Madlener

Figure 1: Main ingredients for the turnkey HEP framework Key4hep.

layer is the experiment framework layer which provides orchestration of the experiment, e.g. through *Marlin*[9] or *Gaudi*[10]. The layer on top consists of the event data model (EDM) for persistent, i.e. to be stored, and transient, i.e. not to be stored, data and the interfaces to databases. The most specific layer then holds algorithms and tools that implement the simulation and reconstruction logic to other packages making use of the features, like *Delphes*[11] or *FastJet*[12, 13].

As all the pieces of software are intertwined with each other, the goal for *Key4hep* is to connect and extend all packages towards a complete data generation and analysis framework. It should be easy to set up, use, maintain and extend as well as fast in runtime.

The main ingredients to this turnkey software framework are shown in Fig. 1 (b). Here, the data processing framework, Marlin for the ILC, is the base, everything else is constructed on. The detector geometry information is done by *DD4hep*[14, 15]. The EDM will be implemented by *podio*[16]. Podio is a generator for thread safe c++ code, taking a high level description of the EDM as a YAML file as an input.

3. Theory

In this section, a description of the underlying physics process that is analysed is presented, as well as Amdahl's law, which is observed in runtime of RDataFrame analyses, see Sec. 4.

3.1. Physics

In the following analyses, the $e^+e^- \to Z^* \to ZH \to H\ell^+\ell^-$ process at a centre of mass energy of 250 GeV is analysed, a representative leading order Feynman diagram can be seen in Fig. 2, and the Higgs recoil mass is calculated. The process can be measured precisely at the ILC.

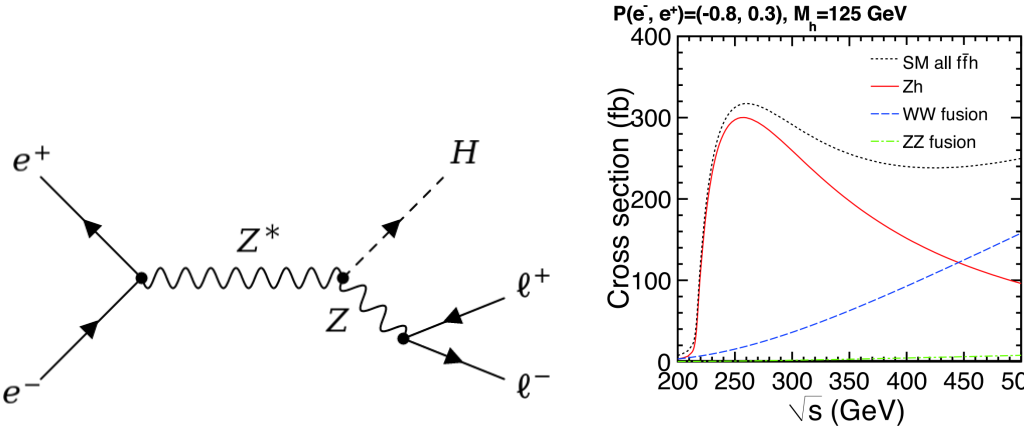


Figure 2: Leading order Feynman diagram (left) and cross section plot [17] (right) with a fixed polarisation of the electrons and positions and a fixed Higgs boson mass.

The square of the invariant mass of the object, in this case the Higgs boson H, that recoils against the Z boson, which subsequently decays into two leptons $\ell^+\ell^-$, is given by

$$M_H^2 = M_{\text{recoil}}^2 = s + M_Z^2 - 2E_Z\sqrt{s}, \tag{1}$$

where s is the centre of mass energy, M_Z is the Z boson mass and E_Z its energy. The cross section for this process peaks at a centre of mass energy of 250 GeV, as shown in Fig. 2. The recoil mass spectrum can be seen in Fig. 3. Here, a clear peak at the Higgs boson mass of $M_H = (125.10 \pm 0.14)\,\text{GeV}$ [18] is observed. The second peak approximately around the Z mass is due to the irreducible ZZ background.

3.2. Amdahl's Law

In computer science, Amdahl's law describes the theoretical speedup of the runtime of a task that can be expected by increasing the number of threads the task uses. It describes the fact that the overall runtime improvements gained by increasing the number of threads and parallelising part of the task is limited by the fraction of time the parallelised part of the task contributes to the total runtime. Amdahl's law [19] is given by

$$S(x) = \frac{1}{(1-p) + \frac{p}{x}}, \tag{2}$$

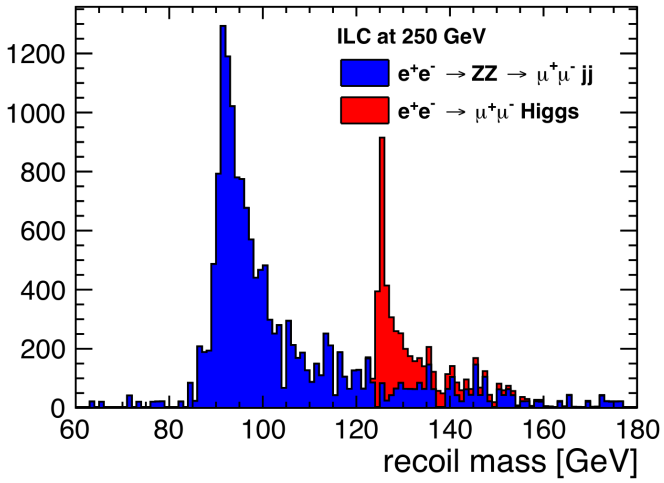


Figure 3: Recoil mass spectrum of toy Monte Carlo events in the $e^-e^+ \to \mu^+\mu^- H$ channel at a centre of mass energy of 250 GeV with $e^+e^- \to \mu^+\mu^-$jj background.

where S is the theoretical speedup of the runtime of the whole task, x is the speedup of the part of the task that is parallelised and p is the proportion of runtime that the parallelised part of the program originally occupied.

4. Analysis Approaches

This analysis has been carried out using eight different analysis approaches, all using EDM4hep, listed in Tab. 1. In the following, each approach is explained. Afterwards, the approaches are analysed in terms of their performance. In all versions, 292,806 background and 17,143 signal events where analysed. The events were simulated using Delphes.
All analyses can be found on GitLab[1].

4.1. Python

The python analysis is carried out in three different ways, characterised by the framework used in the analysis.

Conventional Event Loop (`higgs_recoil_with_bkg_edm4hep_python.py`)
This approach uses the podio generated application programming interface (API) of EDM4hep. In this example, the events are read using the `EventStore` function of podio, and looped over using a conventional python loop. Within the loop, the recoil mass is calculated.

Uproot (`higgs_recoil_with_bkg_edm4hep_uproot(_slow,_conc).py`)
Uproot is a columnar analysis framework, thus the 'event loop' consists of indexing a high

[1] gitlab.desy.de/ftx-sft-key4hep/edm4hep-analysis

dimensional array. There are three versions of the analysis using Uproot. Two versions, using `lazy`, which is a lazy read in of the files and on demand executions for calculations, the other one using `concatenate`, which concatenates all input files and loads them into memory. The difference in the two `lazy` versions is the time at which the di-muon cut in the analysis is applied. `lazy` is scheduled to be decommissioned in December 2022.

RDataFrame (`higgs_recoil_with_bkg_edm4hep_RDataframe(_leafs).py`)
Analogous to Uproot, the RDataFrame analysis is also a columnar analysis. The input files are read into a dataframe. Every action that is performed on the dataframe is tracked in a computation graph. When processing the dataframe's content further, for example when plotting, the computation graph is executed. Currently, RDataFrames only support indexing on leaf level content of ROOT files, thus, to index branch level objects, separate utilities functions, written in c++, are used as a workaround. This has been reported to the ROOT developers. Two analyses were done using RDataFrames, one indexing on leaf, the other one indexing on branch level.

4.2. C++

To compare the c++ performance to the python performance, one conventional, using a conventional event loop, and one columnar analysis is done in this programming language.

Conventional Event Loop (`higgs_recoil_with_bkg_edm4hep.C`)
Analogous to the corresponding python example, the files are read with podio `Reader`, then the events are accessed though podio's `EventStore`. Afterwards they are looped over using a conventional c++ loop.

RDataFrame (`higgs_recoil_with_bkg_edm4hep_RDataFrame.cc`)
The goal with the RDataFrame versions in c++ is to eliminate the just-in-time compilation (JIT) when running the program, as compared to python. Again, analogous to the python analysis a c++ analysis using RDataFrames is done. Here, the indexing of the data frame is done on branch level using separate utilities functions. Analogous to the python version, computation graphs are created when operations on the dataframe are performed.

4.3. Performance Analysis

The average runtimes of representative macros for each analysis type are shown in Tab. 1. Here, `MT0` is short hand for the code line `EnableImplicitMT(0)`, which enables implicit multithreading, with ROOT managing the number of threads automatically.

Fig. 4 shows a runtime comparison of the python scripts using Uproot or a conventional event loop. Shown are the mean runtime, averaged over five executions, minimum and maximum runtime during the five executions. The analysis using Uproot `concatenate`, which loads the whole data set to memory, is slowest. The applicability of Uproot `concatenate` depends on the file size and hardware of the executing machine. The hardware of the machine used to benchmark and run the analysis is shown in Tab. 2.
The other versions shown in the figure use `lazy`, which provides a lazy read in of the files to

Uproot. The two versions differ by the time at which the event selection is applied. In the `late cut` version, the recoil mass calculations are done before the event selection is applied. In the `early cut` version, the event selection is applied first, then the recoil mass is calculated. Although the calculation of the recoil mass is not computationally expensive, applying the event selection before the calculation results in a speedup by a factor of about two.
The runtime of the conventional event loop using python is in between the two Uproot `lazy` versions.
The dotted lines in the plot indicate the runtimes of the c++ RDataFrame (red), the conventional c++ event loop (black) and the fastest python RDataFrame (magenta) macros. Here, the conventional event loop in c++ is faster than the corresponding python one. The fastest python RDataFrame macro, using 16 threads and indexing on leaf level, is approximately as fast as the Uproot version using an early event cut. The overall fastest execution of the recoil mass analysis is provided by the c++ executable, using RDataFrames and an automatic management of the number of threads by ROOT.

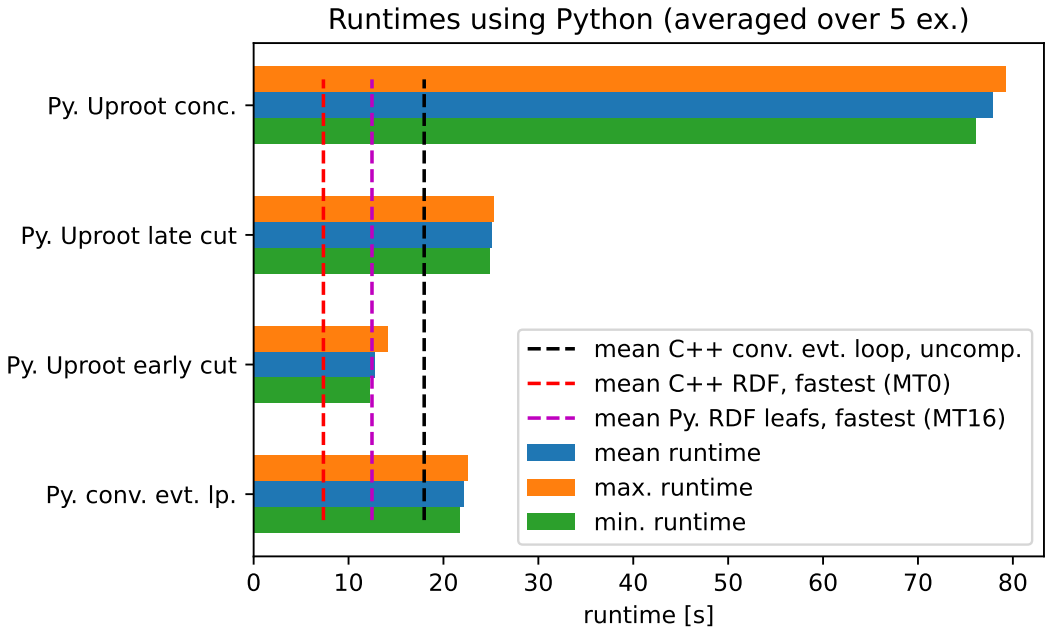


Figure 4: Runtime comparison of python macros using Uproot concatenate and lazy, as well as a conventional event loop.

Fig. 5 shows the runtimes of the macros using RDataFrames. Here, two python versions are shown, one using indexing on the branch level, the other one using indexing on leaf level, in the ROOT files, as well as a c++ version using RDataFrames with a branch level indexing. Here, it is observed, that all the c++ macros perform better than the python versions.
Furthermore, all RDataFrame analysis approaches show the same behaviour in terms of runtime decrease with an increase of used threads. Runtimes 'converge' at a thread number of approximately eight, this is due to Amdahl's law, see Sec. 3.2.
Additionally, runtimes at `MT=0` are fastest in the c++ and python branch level versions. Solely in the python leaf level indexing version the runtime of the macro using 16 threads is lower than the one of the macro using an automatic determination for the number of threads by ROOT.

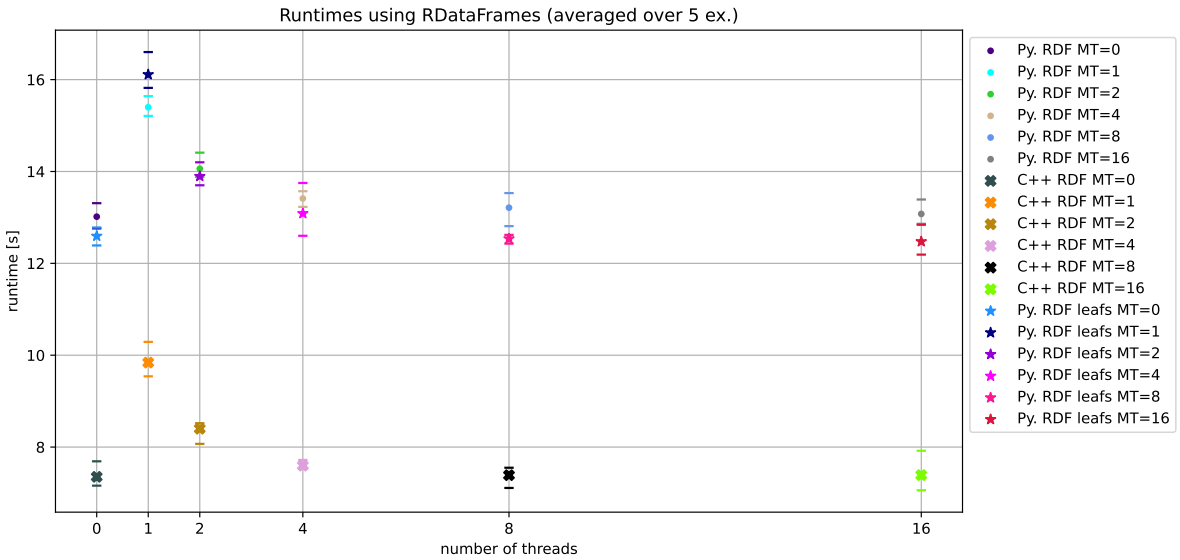


Figure 5: Mean, minimum and maximum runtimes of the RDataFrame macros.

However, differences are small and the runtimes using `MT=0` and `MT=16` are not significantly different.
When comparing the two python versions, the leaf level indexing version almost always performs better than the branch level indexing versions, although differences are insignificant and runtimes often overlap, as indicated by the 'fluctuations in runtime' showing the minimum and maximum runtimes during the five executions.

4.4. Runtime Analysis

In this section, a representative runtime analysis using a python RDataFrame macro is presented. Implementing

```
verbosity =
ROOT.Experimental.RLogScopedVerbosity(ROOT.Detail.RDF.RDFLogChannel(),
ROOT.Experimental.ELogLevel.kInfo)
```

in the macro yields a console output that displays the times each analysis step needs. Runtimes for each individual process, the total runtime and a description of the process are shown in Tab. 3.
Here, it is observed that the just-in-time compilation time that is needed for building the computation graph for the first time, i.e. for the signal files, contributes significantly to the overall runtime. Information is stored in the cache, such that building the computation graph for the second time, i.e. for the bkg. events, takes less than half of the runtime of the first build.
Furthermore, it is also observed, that executing the computation graph takes less than 10% of the time needed to build it, when working with approximately 310,000 events.
Importing the necessary modules as well as plotting and saving an output file also takes a significant fraction of the total runtime.

5. Summary

In this section, the most important results from the performance analysis, see Sec. 4.3, and runtime analysis, see Sec. 4.4, are summarised. Furthermore, the different analysis approaches are briefly discussed.

It is observed, that the c++ analyses are always faster than their python counterparts. This applies to the analyses using a conventional event loop as well as to the analyses using RDataFrames. The overall best performance in terms of runtime is achieved using RDataFrames with ROOT automatically determining the number of threads to use, in c++.

The runtime analysis of the scripts showed, that the just-in-time compilation and importing the necessary modules to the python analyses contribute significantly to their runtime. This can be avoided using a pre compiled c++ version of the analyses.

Although using RDataFrames significantly speeds up the analyses, as of September 2022 it is not possible to index on branch, rather than on leaf, level. A workaround to this problem is to implement separate utilities functions. Indexing on leaf level is marginally faster than using the workaround and indexing on branch level. In python, runtimes of macros using Uproot or RDataFrames do not differ. In Uproot, no separate utilities are needed, making it a more user friendly experience. Additionally, the podio/EDM4hep output files contain special characters which have to be avoided in RDataFrames, thus aliases have to be introduced. Uproot can handle these special characters.

When working with modest amounts of data, runtimes of python and c++ macros using a conventional event loop do not differ significantly. The podio `EventStore` interfaces are used in the conventional event loop examples. Here, a wide variety of functions, like `p4` which constructs a Lorentz vector, as well as an intuitive interface to access branches in events, for example `event.get()`, are provided.

Further tests to the proposed analysis approaches can be done. Multithreading, benchmarking and analysing the runtime of the python and c++ macros using the conventional event loop is yet to be done. If they show a significant increase in performance, future analysis strategies can benefit from that information. This might guide the decision making process on future analysis approaches of particle physics collaborations.

References

[1] ATLAS Collaboration. Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC. *Phys. Lett. B*, 716(1):1–29, 2012.

[2] CMS Collaboration. Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC. *Phys. Lett. B*, 716(1):30–61, 2012.

[3] R. Brun and F. Rademakers. ROOT: An object oriented data analysis framework. *Nucl. Instrum. Meth. A*, 389:81–86, 1997.

[4] J. Allison et al. Geant4 developments and applications. *IEEE Trans. Nucl. Sci.*, 53(1):270–278, 2006.

[5] S. Agostinelli et al. Geant4—a simulation toolkit. *Nucl. Instrum. Methods Phys. Res. A*, 506(3):250–303, 2003.

[6] J. Allison et al. Recent developments in geant4. *Nucl. Instrum. Methods Phys. Res. A*, 835:186–225, 2016.

[7] Christian Bierlich et al. A comprehensive guide to the physics and usage of PYTHIA 8.3. *LU-TP 22-16, MCNET-22-04, FERMILAB-PUB-22-227-SCD*, Mar 2022.

[8] T. Sjostrand, S. Mrenna, and P. Z. Skands. PYTHIA 6.4 Physics and Manual. *JHEP*, 05:026, 2006.

[9] F. Gaede. Marlin and lccd—software tools for the ilc. *Nucl. Instrum. Methods Phys. Res. A*, 559(1):177–180, 2006.

[10] G. Barrand et al. Gaudi — a software architecture and framework for building hep data processing applications. *Comput. Phys. Commun.*, 140(1):45–55, 2001. CHEP2000.

[11] J. de Favereau et al. DELPHES 3, A modular framework for fast simulation of a generic collider experiment. *JHEP*, 02:057, 2014.

[12] M. Cacciari, G. P. Salam, and G. Soyez. FastJet User Manual. *Eur. Phys. J. C*, 72:1896, 2012.

[13] Matteo Cacciari and Gavin P. Salam. Dispelling the N^3 myth for the k_t jet-finder. *Phys. Lett. B*, 641:57–61, 2006.

[14] M. Frank et al. DD4hep: A detector description toolkit for high energy physics experiments. *J. Phys. Conf. Ser.*, 513(2):022010, Jun 2014.

[15] M. Petrič et al. Detector simulations with DD4hep. *J. Phys. Conf. Ser.*, 898:042015, Oct 2017.

[16] F. Gaede, B. Hegner, and P. Mato. PODIO: An event-data-model toolkit for high energy physics experiments. *J. Phys. Conf. Ser.*, 898:072039, Oct 2017.

[17] Jacqueline Yan et al. Measurement of the Higgs boson mass and $e^+e^- \to ZH$ cross section using $Z \to \mu^+\mu^-$ and $Z \to e^+e^-$ at the ILC. *Phys. Rev. D*, 94(11):113002, 2016.

[18] P. Zyla et al. Review of Particle Physics. *Prog. Theor. Exp. Phys.*, 2020.

[19] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective, 3rd edition*. Pearson, 2016.

A. Runtimes

Langauage	Technique/Framework	Avg. Runtime	Min. Runtime	Max. Runtime
Python	Conventional event loop	22.17 s	21.72 s	22.60 s
	Uproot (lazy, early cut)	12.74 s	12.23 s	14.12 s
	Uproot (lazy, late cut)	25.10 s	24.93 s	25.36 s
	Uproot (concatenate)	77.83 s	76.06 s	79.31 s
	RDataFrame (branch level) MT0	13.02 s	12.76 s	13.31 s
	RDataFrame (leaf level) MT16	12.47 s	12.43 s	12.62 s
C++	Conventional event loop	17.97 s	17.54 s	18.08 s
	RDataFrame MT0	7.35 s	7.16 s	7.69 s

Table 1: Average runtime, averaged over five executions, minimum and maximum runtimes of the python and c++ macros, using a conventional event loop, Uproot or RDataFrames, with various subversions.

Specification	Value
Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	20
Thread(s) per core:	2
Core(s) per socket:	10
Socket(s):	1
NUMA node(s):	1
CPU family:	6
Model:	85
Model name:	Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz
Stepping:	4
CPU MHz:	799.890
CPU max MHz:	3000,0000
CPU min MHz:	800,0000
L1d cache:	32 K
L1i cache:	32 K
L2 cache:	1024 K
L3 cache:	14080 K

Table 2: System hardware specifications on which the analyses where executed and benchmarked.

Process	Total Runtime	Description
importing modules	2.5 s	importing necessary modules to python
Run()>: Starting event loop number 0.		starting to build computation graph (CG) for signal event loop
Jit()>: Just-in-time compilation phase completed in 3.573096 seconds.	6.07 s	completed just-in-time compilation (JIT) for CG for signal event loop
RunTreeReader()>: Processing trees in files signal: entry range [0,17142], using slot 0 in thread 139735803369280.		reading `ROOT` file for CG for signal event loop
Run()>: Finished event loop number 0 (0.03s CPU, 0.028059s elapsed).	6.10 s	finished CG for signal event loop
Run()>: Starting event loop number 0.		starting to build CG for bkg. event loop
Jit()>: Just-in-time compilation phase completed in 1.389356 seconds.	7.49 s	completed JIT for CG for bkg. event loop
RunTreeReader()>: Processing trees in files bkg: entry range [0,292805], using slot 0 in thread 139735803369280.		reading `ROOT` file for CG for bkg. event loop
Run()>: Finished event loop number 0 (0.31s CPU, 0.308021s elapsed).	7.80 s	finished CG for bkg. event loop
Run()>: Starting event loop number 1.		starting to execute CG for signal
Jit()>: Just-in-time compilation phase completed in 0.252379 seconds.	8.05 s	finished JIT for execution of signal CG
RunTreeReader()>: Processing trees in files signal: entry range [0,17142], using slot 0 in thread 139735803369280.		reading in the signal files
Run()>: Finished event loop number 1 (0.22s CPU, 0.215331s elapsed).	8.27 s	finished executing the CG for the signal files
Run()>: Starting event loop number 1.		starting to execute CG for bkg.
Jit()>: Nothing to jit and execute.		nothing to JIT
RunTreeReader()>: Processing trees in files bkg: entry range [0,292805], using slot 0 in thread 139735803369280.		reading in the bkg. files
Run()>: Finished event loop number 1 (2.51s CPU, 2.50069s elapsed).	10.77 s	finished executing the CG for the bkg. files
plotting and saving the file as `.pdf`	13.02 s	stacking the histograms creating a plot and saving the plot as a file

Table 3: Representative runtime analysis of the python RDataFrame macro and description of the (most likely) process that is executed.

Acknowledgements

I would like to thank the whole FTX group for the very warm welcome to the group, I enjoyed every minute of working in the group. A special thank you to Frank, Felix and Andrea for making my stay at DESY possible. Of course also big thanks to my office mates Peter, Engin and Finn, it was always nice working in this office and of course I especially enjoyed every trip to the canteen. For sure I will miss this office!

Thank you so much for your support Thomas, day and night, weekdays and weekends! I learned so much from you! Being your summer student was always fun and it definitely shaped my interest in software and programming and possibly the next steps in my life! You were always supportive and encouraging. I will miss this time!

Also a big thank you to Olaf, Frederike, Andreas and Ute for making this summer school so fun and interesting and the time here at DESY so amazing. We summies loved this summer school!

Lastly, a final thanks must go to the summer students, of whom there are too many to acknowledge. The trips we made, the days, evenings and nights we spend together in the hostel kitchen, the memories we created, these are things I will never forget. I am certain I have gained some friends for life.