Future Analysis According to ROOT

Lorenzo Moneta
(CERN/EP-SFT)

12/7/2022

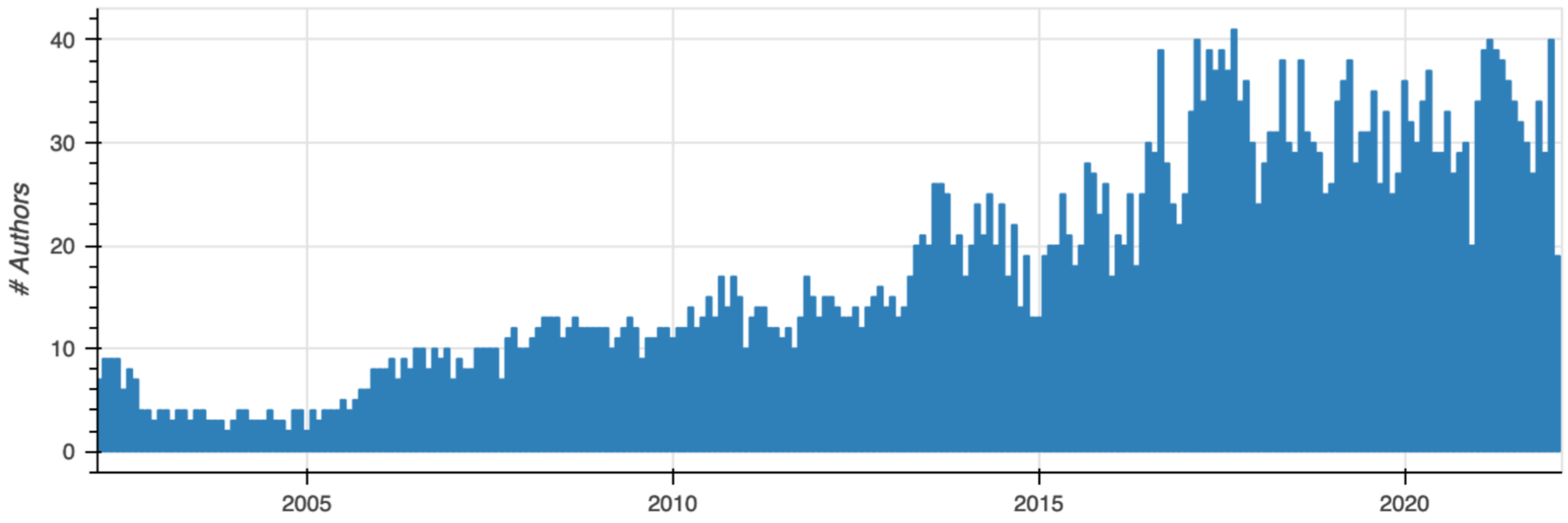
Outline

- Analysis landscape:
  - I/O developments
    - RNtuple
  - data exploration, declarative analysis
    - RDataFrame
  - machine learning, interoperability and analysis integration
    - TMVA/SOFIE
  - statistical analysis and modelling
    - RooFit
- What we have in ROOT now
- Planned future developments

A lot of material presented coming from recent ICHEP talks

Introduction

- ROOT is an open source, community driven project
- Despite existing for many years ROOT is still very active
  - high number of contributors/month



- ROOT is used by all HEP experiments
  - more than 1EB of data stored in ROOT format

ROOT Core Components

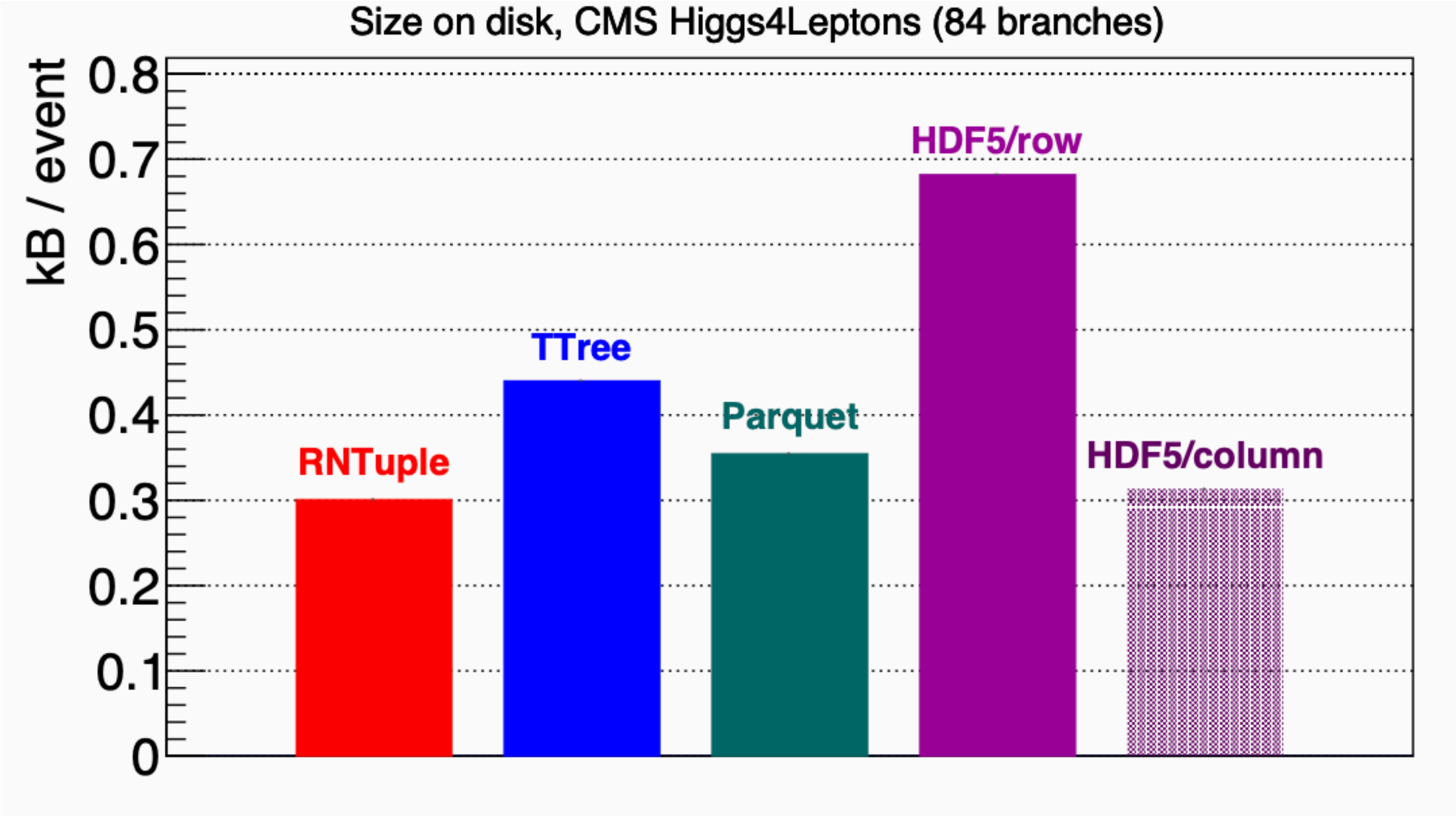
- I/O : reading and writing data efficiently
- Analysis interfaces: histograms and RDataFrame
- Interactivity: C++ interpreter and Python bindings
- Math libraries: PRNG, numerical algorithms (Minuit)
- Statistics and modeling: fitting
- Graphics: scientific visualization
- Machine learning: model evaluations and interoperability

Will show the major recent developments in some of these components and the future plans

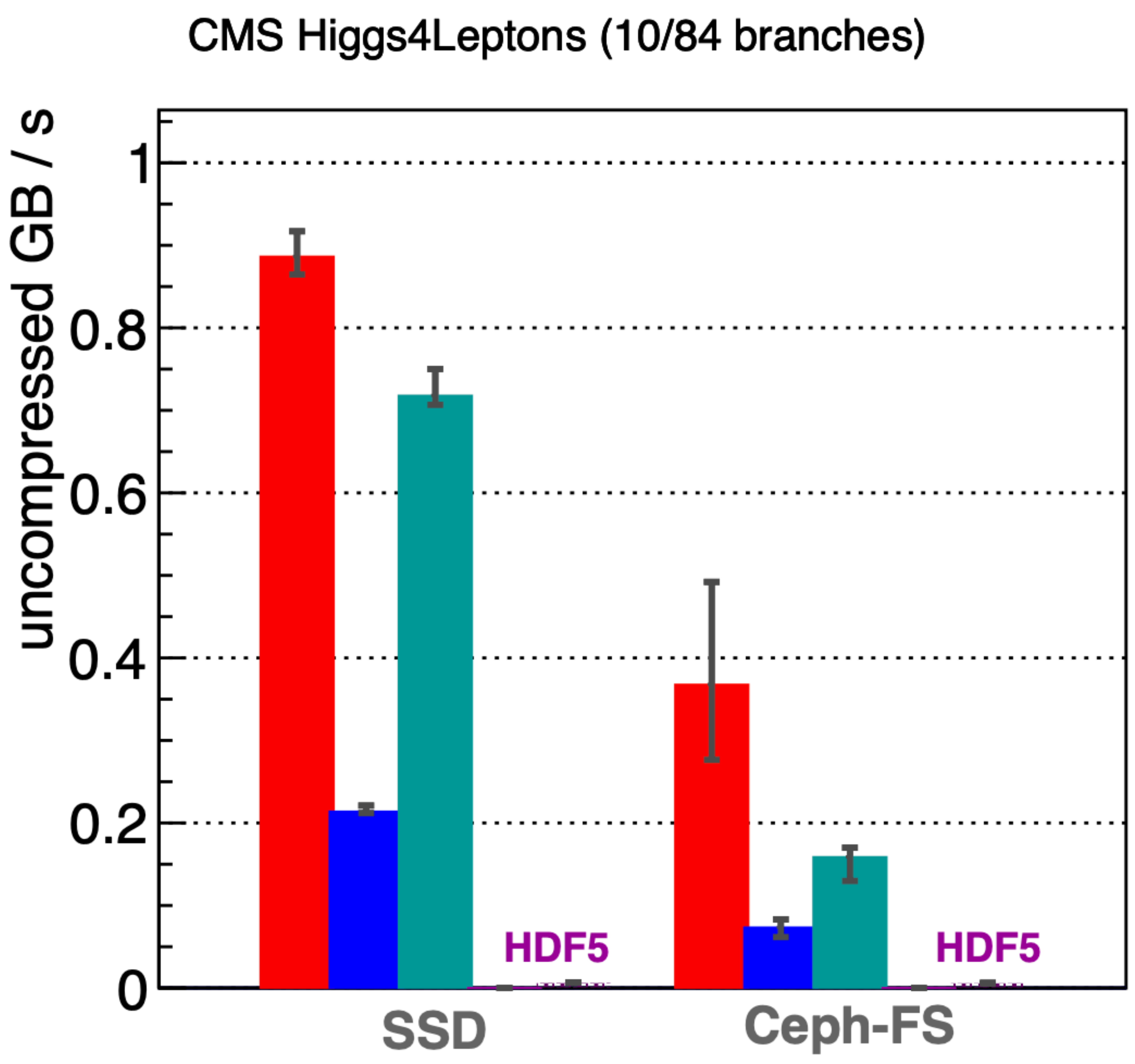
RNTuple

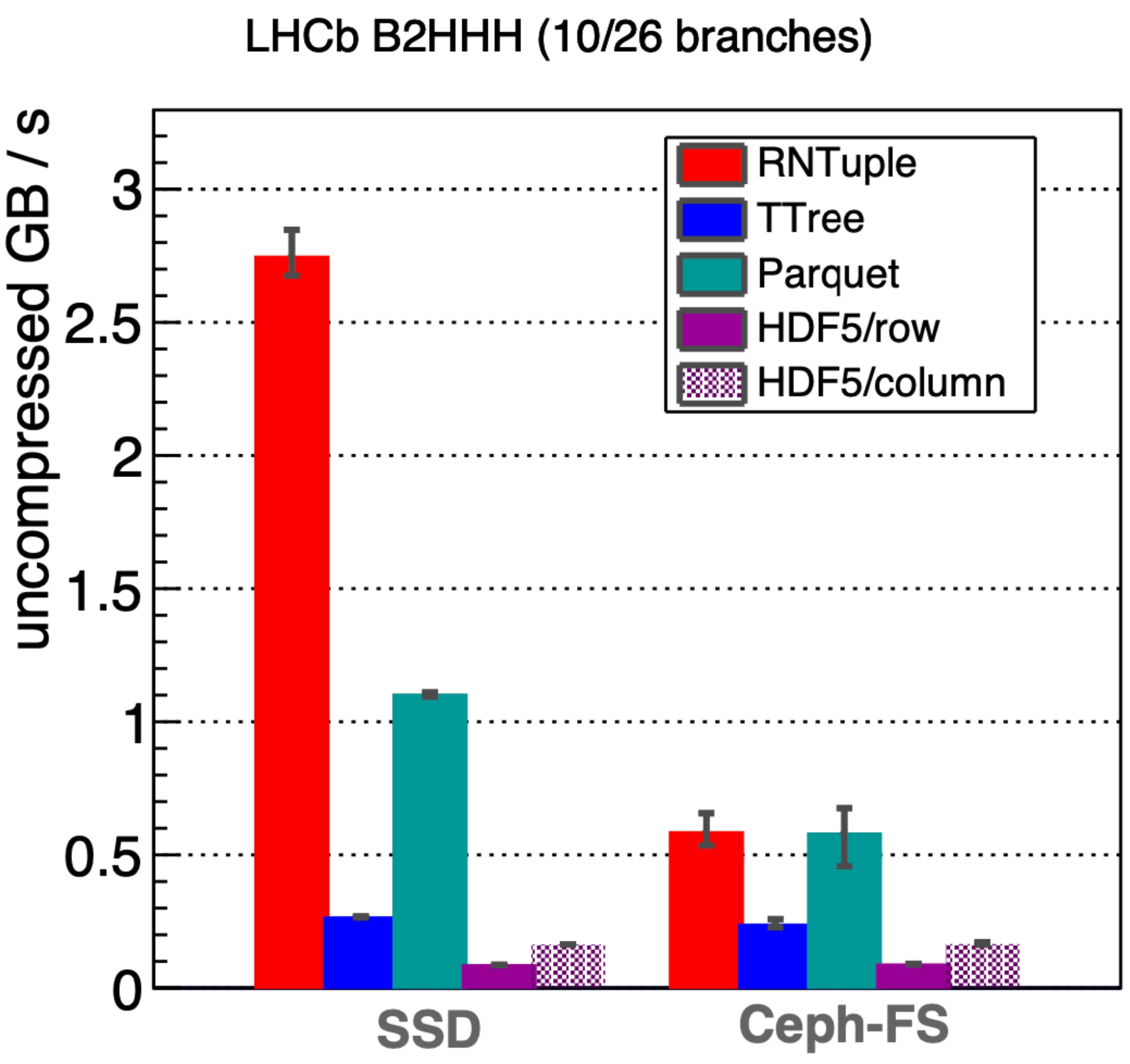
- Successor of TTree
  - columnar storage of event data optimised for selective reads
- Schedule for production for HL-LHC
- Based on 25+ years of TTree experience,
- Redesigned I/O subsystem providing:
  - Less disk and CPU usage for the same data content
  - 25% smaller files, × 2–5 better single-core performance
  - 10 GB/s per box and 1 GB/s per core sustained end-to-end throughput
- And also:
  - Systematic use of exceptions to prevent silent I/O errors
  - Efficient support of modern hardware (built for multi-threading and async I/O)
  - Native support for object stores

RNTuple Performances: I/O Size



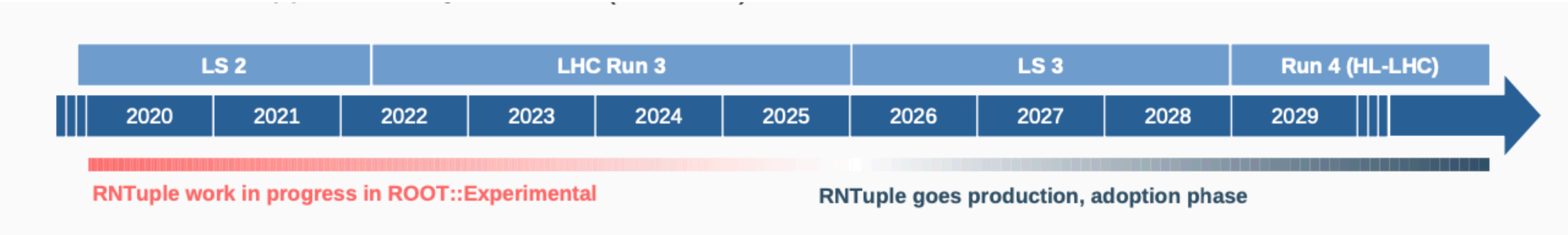
RNTuple Performances: I/O Speed





RNTuple Plans

- Smaller files and significantly faster reads compared to TTree
- Modern and robust API
- Capable of making efficient use of modern devices and storage systems (such as SSD, object stores, many cores)
- RNTuple is work in progress in *ROOT::Experimental*
  - the on-disk format is still subject to small changes!
- We are happy to get your feedback!



HEP Analysis Landscape

Analysis life cycle

skimming, ntuple production — quick exploration, first implementation — systematics, scale out — statistical analysis

Platforms

laptop or PC — many-core machine — computing cluster + job submission

Analysis languages

- ⬇ ~50% C++
- ⬆ ~50% Python

Storage

local disk
fast-access network storage
EOS or other not-so-fast backend

from E. Guiraud: RDF@ICHEP 2022

RDataFrame

A Swiss army knife for data analysis

Analysis life cycle

- skimming, ntuple production
- quick exploration, first implementation
- systematics, scale out

Platforms

- laptop or PC
- many-core machine
- computing cluster + job submission

Analysis languages

- ⬇ ~50% C++
- ⬆ ~50% Python

Storage

- local disk
- fast-access network storage
- EOS or other not-so-fast backend

ROOT.RDataFrame is a modern analysis interface that addresses all these use cases with **one high-level programming** model that performs well, scales well and enables **HEP-specific ergonomics**, in C++ and Python.

from E. Guiraud: RDF@ICHEP 2022

RDataFrame Code

Example of RDF code in Python

```
df = ROOT.RDataFrame(dataset)
df = df.Filter("x > 0")
       .Define("r2", "x*x + y*y")
rHist = df.Histo1D("r2")
df.Snapshot("newtree", "out.root")
```

`df = ROOT.RDataFrame(dataset)`	on this (ROOT, CSV, ...) dataset	
`df = df.Filter("x > 0")`	only accept events for which x > 0	**event selection**
`.Define("r2", "x*x + y*y")`	define r2 = $x^2 + y^2$	**derived quantities, object selections**
`rHist = df.Histo1D("r2")`	plot r2 for events that pass the cut	
`df.Snapshot("newtree", "out.root")`	write the skimmed data and r2 to a new ROOT file	**data aggregations**

Users can inject **arbitrary code** at all steps,
which makes this relatively simple API extremely versatile.

Physicists describe analysis ingredients, ROOT handles technicalities

RDataFrame Code: MT

Switch to multi-thread execution (Python):

`ROOT.EnableImplicitMT()`	Run a multi-thread event loop
`df = ROOT.RDataFrame(dataset)`	on this (ROOT, CSV, ...) dataset
`df = df.Filter("x > 0")`	only accept events for which x > 0
`.Define("r2", "x*x + y*y")`	define r2 = $x^2 + y^2$
`rHist = df.Histo1D("r2")`	plot r2 for events that pass the cut
`df.Snapshot("newtree", "out.root")`	write the skimmed data and r2 to a new ROOT file

from E. Guiraud: RDF@ICHEP 2022

RDataFrame Code: DistRDF

Switch to distributed execution (Python):

```
cluster = dask_jobqueue.HTCondorCluster(n_workers=64)
df = RDataFrame(dataset, daskclient=Client(cluster))
df = df.Filter("x > 0")
          .Define("r2", "x*x + y*y")
rHist = df.Histo1D("r2")
df.Snapshot("newtree", "out.root")
```

connect to HTCondor via Dask

other code stays the same

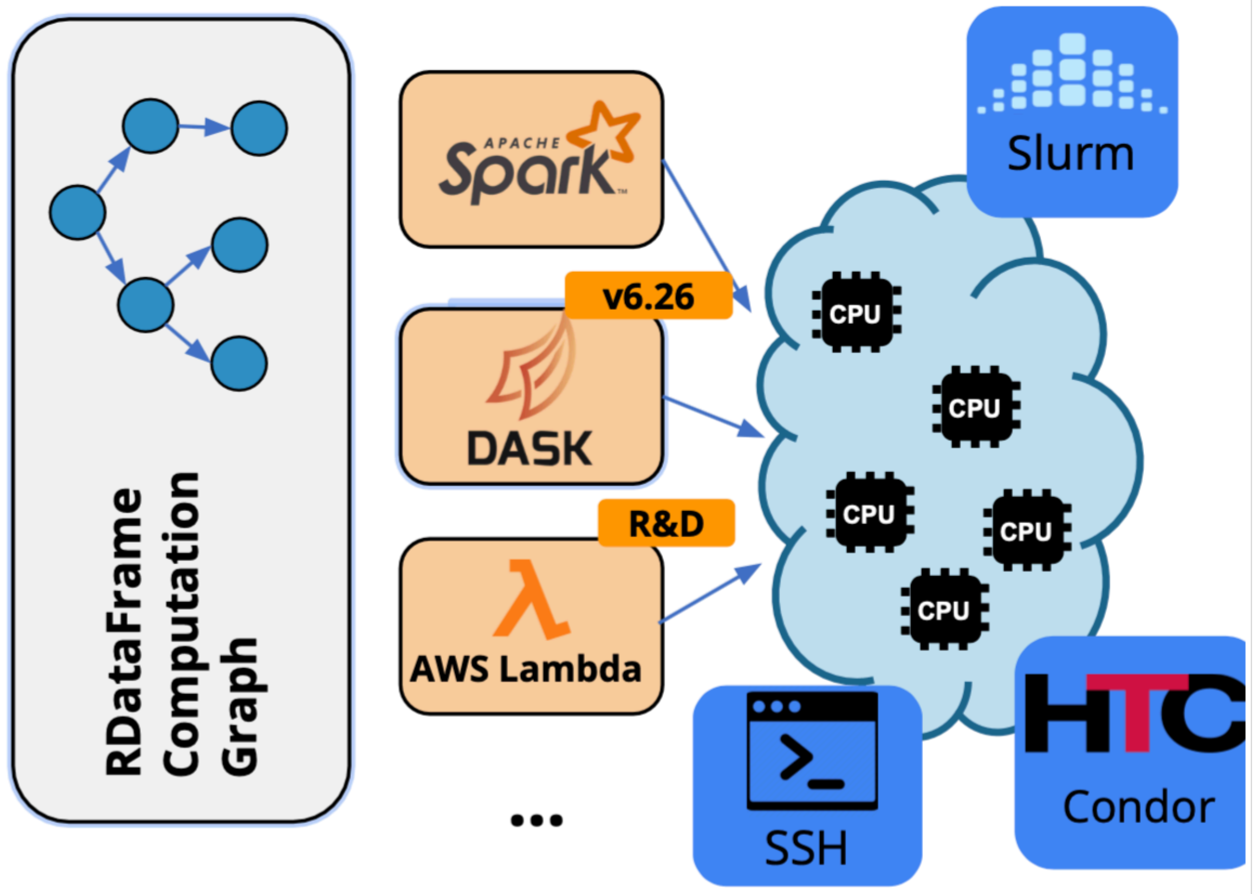
Available since v6.26 (experimental)

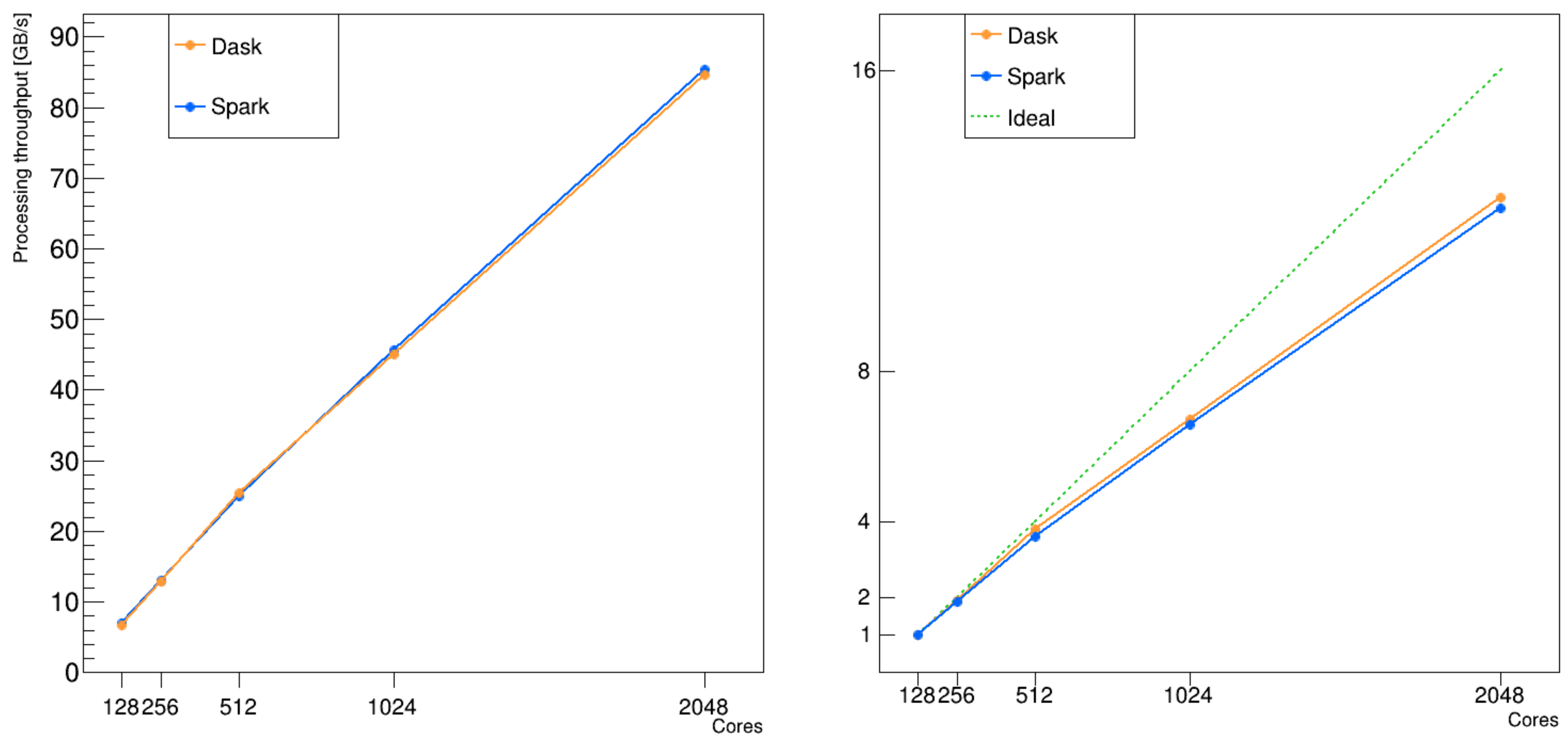
Also see [this tutorial], [the docs], the [recent ATTF talk]

from E. Guiraud: RDF@ICHEP 2022

Distributed RDF

- Enables **interactive large-scale distributed** data analysis
- Python RDF API, C++ event loop
- Full access to ROOT I/O
- Let **Spark/Dask/HTCondor/Slurm/SSH**…. take care of scheduling and resource management
- Transparently merges results coming from different computing nodes



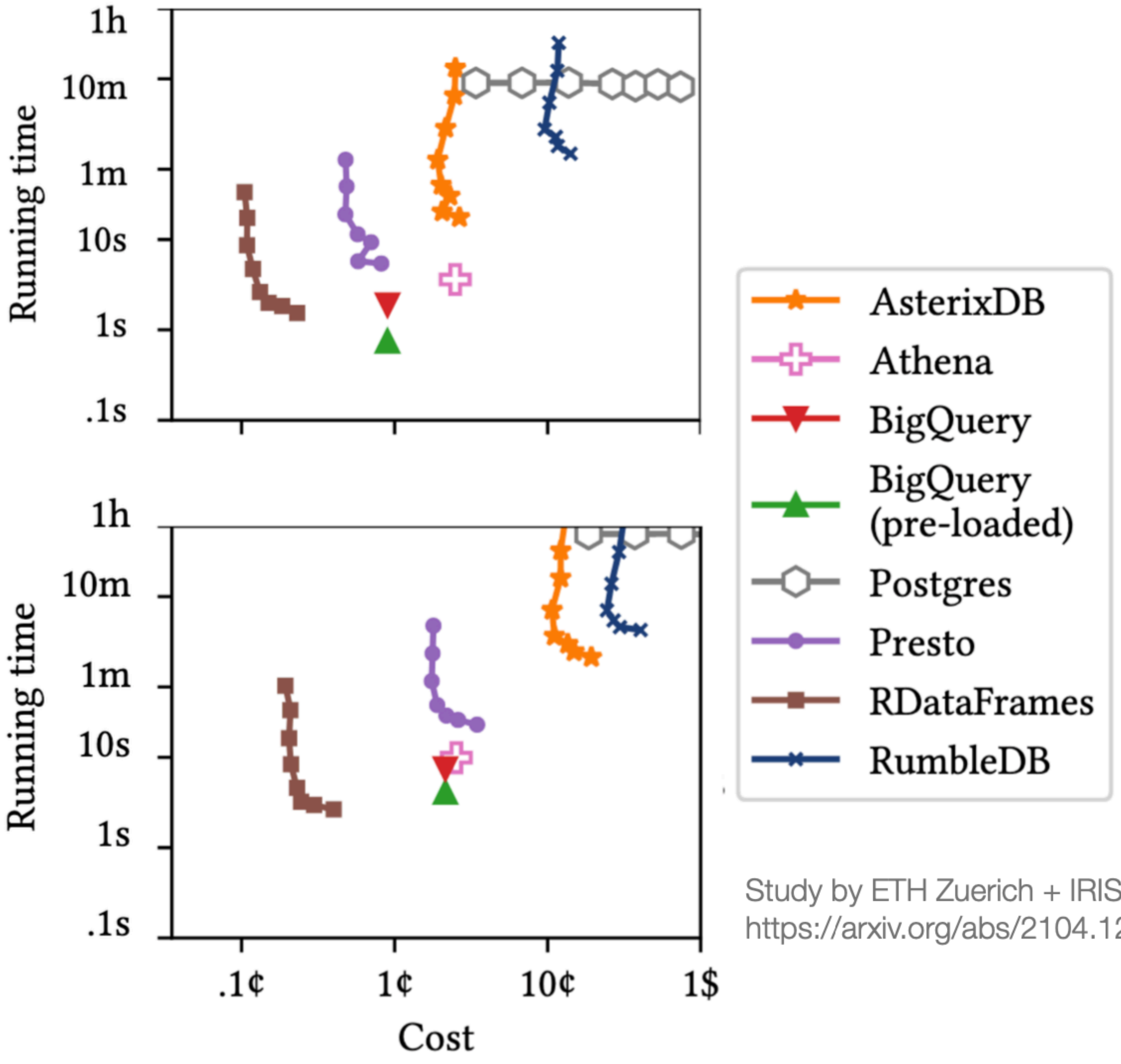


Distributed RDF benchmark (dimuon, 4000x data)
See this talk

from E. Guiraud: RDF@ICHEP 2022

RDF Performances

- RDataFrame enables fast turnaround for complex analysis use cases
- RDataFrame scales well to many cores, many nodes, many histograms
- Independent study shows RDataFrame analysis to be significantly **faster**
- Performance is always ongoing work: we are constantly looking for feedback/use cases



Study by ETH Zuerich + IRIS-HEP
https://arxiv.org/abs/2104.12615

RDF Benchmarks

Fully compiled C++ RDataFrame

query	1x data (s)	10x data (s)
Q1	0.37	1.50
Q2	0.46	3.70
Q3	0.73	6.23
Q4	0.65	5.92
Q5	0.84	7.45
Q6	3.08	27.99
Q7	2.56	22.27
Q8	1.17	10.22

Coffea 0.7.12 (using chunksize=2**19)

query	1x data (s)	10x data (s)
Q1	1.40	4.24
Q2	1.51	5.76
Q3	1.81	7.96
Q4	1.65	6.58
Q5	2.41	12.43
Q6	13.89	124.59
Q7	4.19	29.12
Q8	3.27	17.70

- note that these benchmarks are not representative of large analysis workloads
- see also this ACAT talk by Nick Smith

Benchmark from github.com/nsmith-/coffea-benchmarks
Setup: AMD EPYC 7702P, using 48 physical cores, data read from filesystem cache

RDF: Working with Collections

Select and fill: quick one-liner

```
h = df.Define("pt", "muon_pt[abs(muon_eta) < 2]").Histo1D("pt")
```

Compiled C++

```
RVecD selectPt(RVecD &pt, RVecD &eta) {
  return pt[abs(eta) < 2];
}

auto h = df.Define("pt", selectPt,
                   {"muon_pt", "muon_eta"})
           .Histo1D<RVecD>("pt");
```

Python+Numba

Current R&D

```
def select_pt(muon_pt, muon_eta):
  return muon_pt[np.abs(muon_eta) < 2]

h = df.Define("pt", select_pt).Histo1D("pt")
```

See docs about injecting Python into RDF in v6.26.

from E. Guiraud: RDF@ICHEP 2022

Systematic Variations

Python

attach an up/down variation to “pt”

```
nominal_hx =
   df.Vary("pt", "RVecD{pt*0.9, pt*1.1}",  ["down", "up"])
     .Filter("pt > k")
     .Define("x", someFunc, ["pt"])
     .Histo1D("x")

hx = ROOT.RDF.VariationsFor(nominal_hx)
hx["nominal"].Draw()
hx["pt:down"].Draw("SAME")
```

proceed as usual, as if working with nominal values only

obtain all variations

Variations automatically propagate to selections, derived quantities and results.
Multi-thread and **distributed** execution **just works.**
Only needed quantities are re-computed, all in **one event loop.**

from E. Guiraud: RDF@ICHEP 2022

NumPy Interoperability

- **TTree → NumPy** via RDataFrame

```
cols = df.Filter("x > 10").AsNumpy(["x", "y"])
```

- **NumPy → RDataFrame**

```
data = {"x": np.array(...), "y": np.array(...), ...}
df = ROOT.RDF.MakeNumpyDataFrame(data)
```

- Work in progress: RDF ⇔ Awkward arrays, see here
- Working on developing direct batch generator of NumPy arrays for training of ML models (see later)

from E. Guiraud: RDF@ICHEP 2022

Summary RDF

- and many more features, e.g.
  - transparent support for RNTuple, with no code changes
  - machine learning inference as part of the event loop
    - SOFIE, RBDT
- RDF keeps evolving
  - cooperation with HEP community
- Critical to focus on the right features
  - need your feedback and help

Lazy action	Description
Aggregate()	Execute a user-defined accumulation operation on the processed column values.
Book()	Book execution of a custom action using a user-defined helper object.
Cache()	Cache column values in memory. Custom columns can be cached as well, filtered entries are not cached. Users can specify which columns to save (default is all).
Count()	Return the number of events processed. Useful e.g. to get a quick count of the number of events passing a Filter.
Display()	Provides a printable representation of the dataset contents. The method returns a ROOT::RDF::RDisplay() instance which can print a tabular representation of the data or return it as a string.
Fill()	Fill a user-defined object with the values of the specified columns, as if by calling `Obj.Fill(col1, col2, ...)`.
Graph()	Fills a TGraph with the two columns provided. If multi-threading is enabled, the order of the points may not be the one expected, it is therefore suggested to sort if before drawing.
GraphAsymmErrors()	Fills a TGraphAsymmErrors. If multi-threading is enabled, the order of the points may not be the one expected, it is therefore suggested to sort if before drawing.
Histo1D(), Histo2D(), Histo3D()	Fill a one-, two-, three-dimensional histogram with the processed column values.
HistoND()	Fill an N-dimensional histogram with the processed column values.
Max()	Return the maximum of processed column values. If the type of the column is inferred, the return type is `double`, the type of the column otherwise.
Mean()	Return the mean of processed column values.
Min()	Return the minimum of processed column values. If the type of the column is inferred, the return type is `double`, the type of the column otherwise.
Profile1D(), Profile2D()	Fill a one- or two-dimensional profile with the column values that passed all filters.
Reduce()	Reduce (e.g. sum, merge) entries using the function (lambda, functor...) passed as argument. The function must have signature `T(T,T)` where `T` is the type of the column. Return the final result of the reduction operation. An optional parameter allows initialization of the result object to non-default values.
Report()	Obtain statistics on how many entries have been accepted and rejected by the filters. See the section on **named filters** for a more detailed explanation. The method returns a ROOT::RDF::RCutFlowReport instance which can be queried programmatically to get information about the effects of the individual cuts.
Stats()	Return a TStatistic object filled with the input columns.
StdDev()	Return the unbiased standard deviation of the processed column values.
Sum()	Return the sum of the values in the column. If the type of the column is inferred, the return type is `double`, the type of the column otherwise.
Take()	Extract a column from the dataset as a collection of values, e.g. a `std::vector<float>` for a column of type `float`.

Instant action	Description
Foreach()	Execute a user-defined function on each entry. Users are responsible for the thread-safety of this callable when executing with implicit multi-threading enabled.
ForeachSlot()	Same as Foreach(), but the user-defined function must take an extra `unsigned int slot` as its first parameter. `slot` will take a different value, `0` to `nThreads - 1`, for each thread of execution. This is meant as a helper in writing thread-safe Foreach() actions when using RDataFrame after ROOT::EnableImplicitMT(). ForeachSlot() works just as well with single-thread execution: in that case `slot` will always be `0`.
Snapshot()	Write the processed dataset to disk, in a new TTree and TFile. Custom columns can be saved as well, filtered entries are not saved. Users can specify which columns to save (default is all). Snapshot, by default, overwrites the output file if it already exists. Snapshot() can be made *lazy* setting the appropriate flag in the snapshot options.

Queries

These operations do not modify the dataframe or book computations but simply return information on the RDataFrame object.

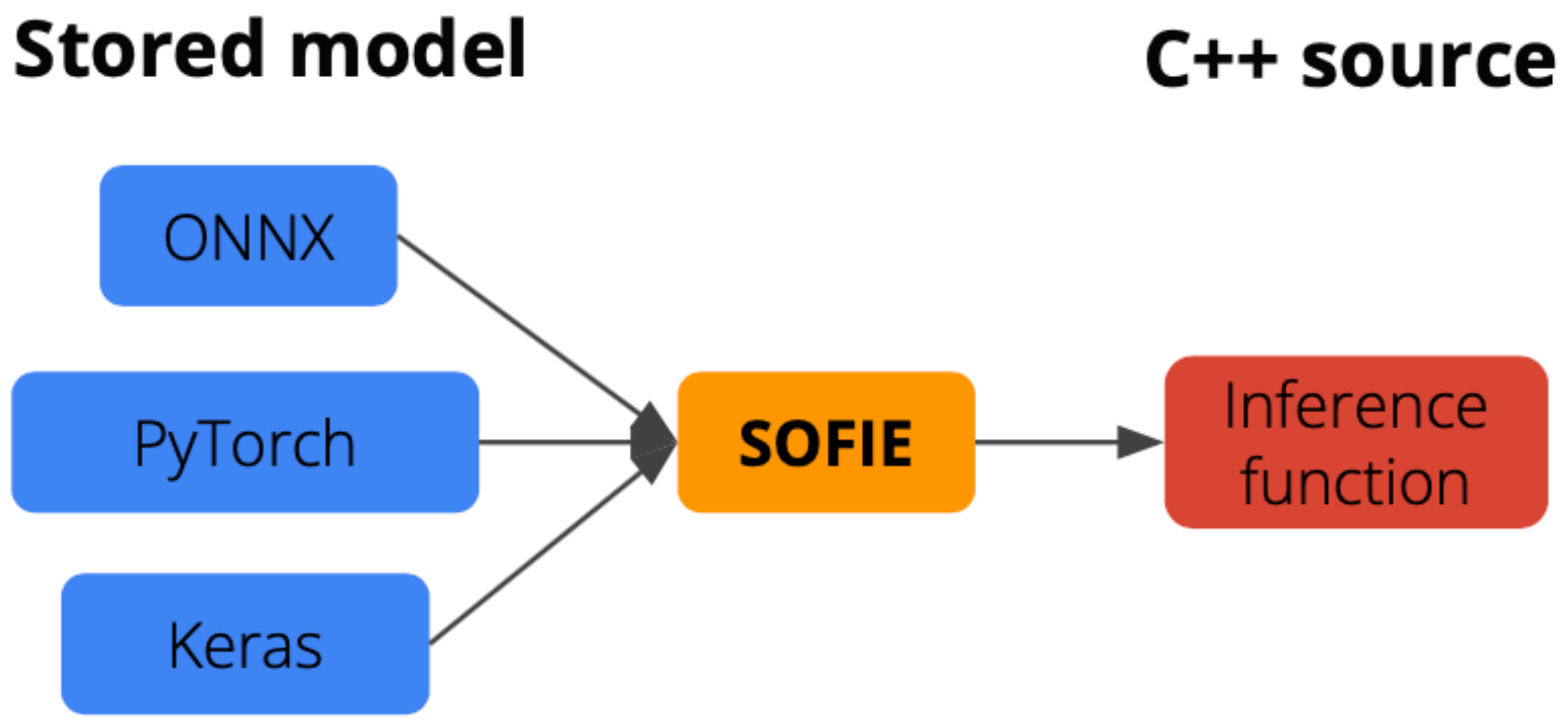
Operation	Description
Describe()	Get useful information describing the dataframe, e.g. columns and their types.
GetColumnNames()	Get the names of all the available columns of the dataset.
GetColumnType()	Return the type of a given column as a string.
GetColumnTypeNamesList()	Return the list of type names of columns in the dataset.
GetDefinedColumnNames()	Get the names of all the defined columns.
GetFilterNames()	Return the names of all filters in the computation graph.
GetNRuns()	Return the number of event loops run by this RDataFrame instance so far.
GetNSlots()	Return the number of processing slots that RDataFrame will use during the event loop (i.e. the concurrency level).
SaveGraph()	Store the computation graph of an RDataFrame in DOT format (graphviz) for easy inspection. See the **relevant section** for details.

from E. Guiraud: RDF@ICHEP 2022

SOFIE: Fast Inference in ROOT

- New ROOT tool for inference code generation for DL models
  - input, a trained ML model:
    - ONNX (new standard for ML)
    - TF/Keras
    - PyTorch
  - Output:
    - C++ code
      - minimal dependency (BLAS)
      - can be compiled on the fly (e.g. with Cling)

ROOT 6.26 (experimental)




```cpp
using namespace TMVA::Experimental::SOFIE;
RModelParser_ONNX parser;
RModel model = parser.Parse("Model.onnx");
```

```cpp
// generate text code internally (with some options)
model.Generate();
// write output header file and data weight file
model.OutputGenerated();
```

SOFIE with RDataFrame

- Model evaluation with SOFIE can be integrated in RDF event loop
  - SofieFunctor : adapter for using SOFIE within RDF

```
auto h1 = df.DefineSlot("DNN_Value",
          SofieFunctor<7,TMVA_SOFIE_higgs_model_dense::Session>(nslots),
          {"m_jj", "m_jjj", “m_lv", “m_jlv","m_bb","m_wbb","m_wwbb"}).
          Histo1D("DNN_Value");
```

  - support for multi-threaded evaluation using RDF slots
  - see full Example tutorial code in C++ or Python

Supported ONNX Operators

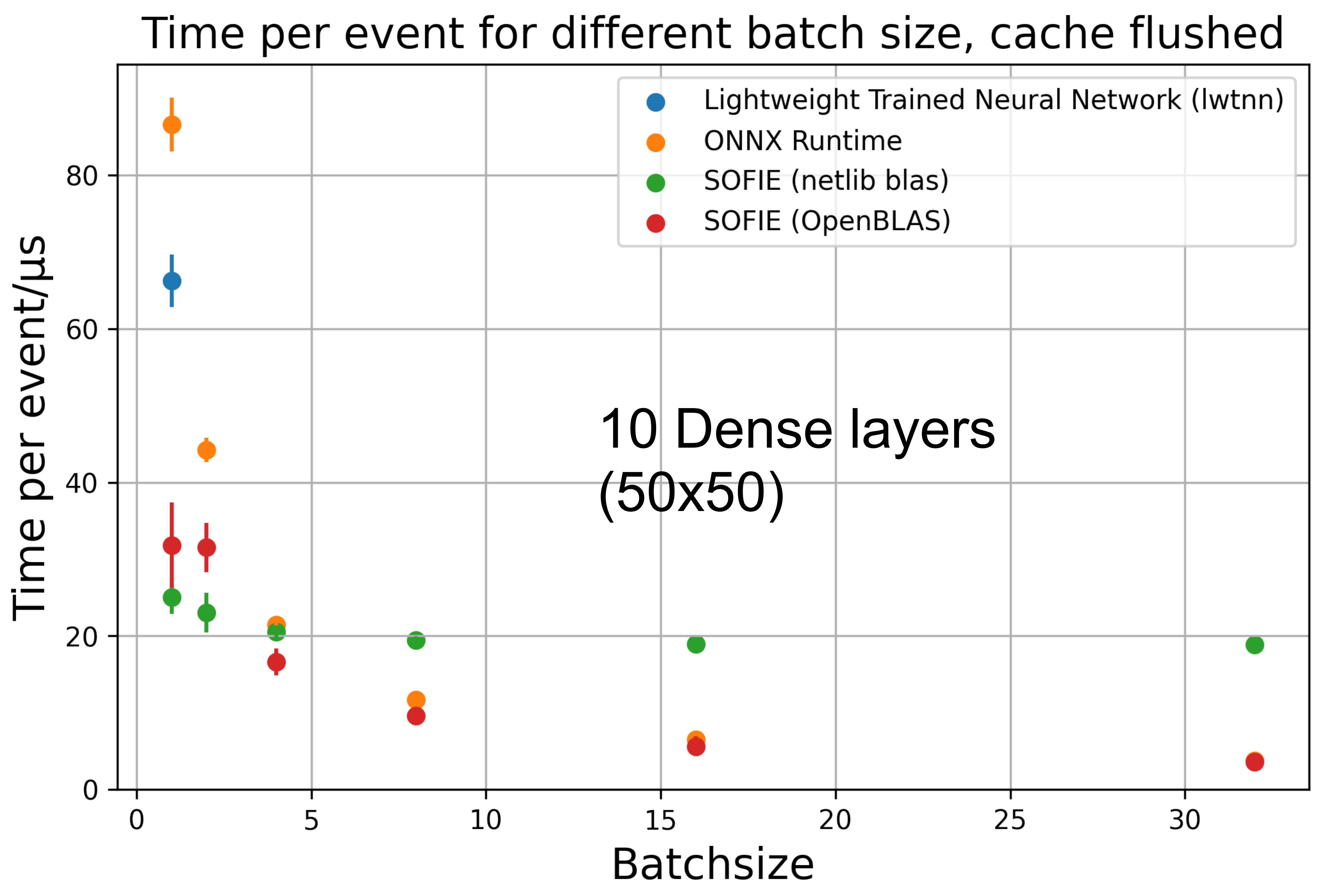
Perceptron: Gemm	Implemented and integrated (ROOT 6.26)
Activations: Relu, Seul, Sigmoid, Softmax, LeakyRelu	Implemented and integrated
Convolution (1D, 2D and 3D)	Implemented and integrated
Recurrent: RNN, GRU, LSTM	Implemented and integrated
BatchNormalization	Implemented and integrated
Pooling: MaxPool, AveragePool, GlobalAverage	Implemented and integrated
Layer operations: Add, Sum, Mul, Div, Reshape, Flatten, Transpose, Squeeze, Unsqueeze, Slice, Concat, Identity	Implemented and integrated
InstanceNorm	Implemented but to be integrated (PR #8885)
Deconvolution, Reduce operators (for generic layer normalisation), Gather (for embedding) and more…	Planned for next release
???	Depending on user needs

SOFIE Benchmarks

Comparison with ONNXRuntime and LWTNN

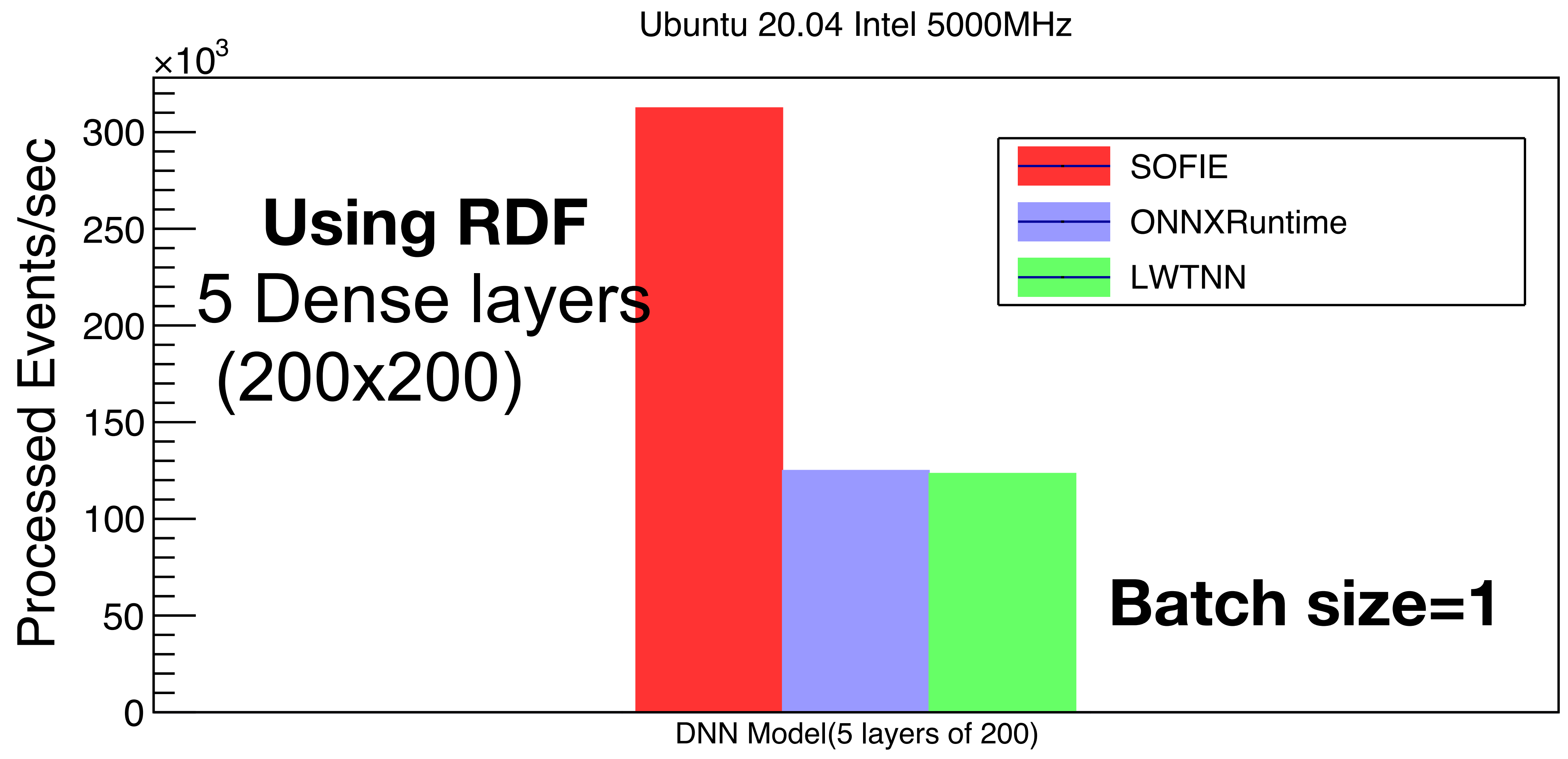
- 2-3 faster than ONNXRuntime
  for DNN with batch size=1
- 20% faster for RNN operators
- slower for CNN (20% for 2D, x2 for 3D)

Smaller = Better

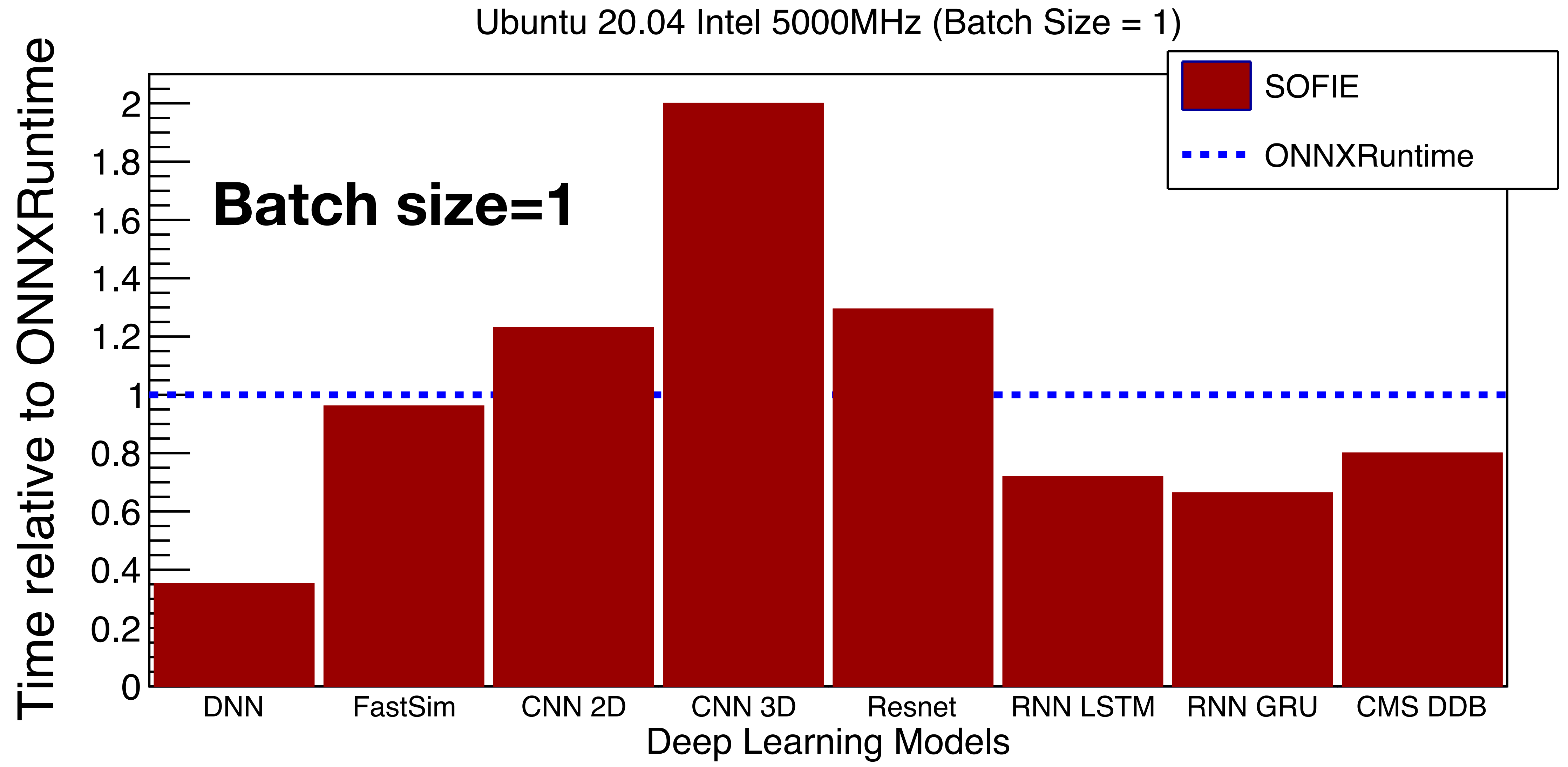


varying batch size…

Larger = Better



Smaller = Better



using different models (DNN, CNN, RNN)

Fast Decision Tree Inference

- Inference engine taking model parameters from externally trained models
- Features:
  - Simple to use from Python and C++
  - Thread-safe
  - Zero-copy
  - Fast for single event and batch inference

- External training and model conversion

```
xgb = xgboost.BDTClassifier(options)
xgb.fit(x, y)

ROOT.TMVA.SaveXGBoost(xgb, "myBDT", "model.root")
```

- Python application

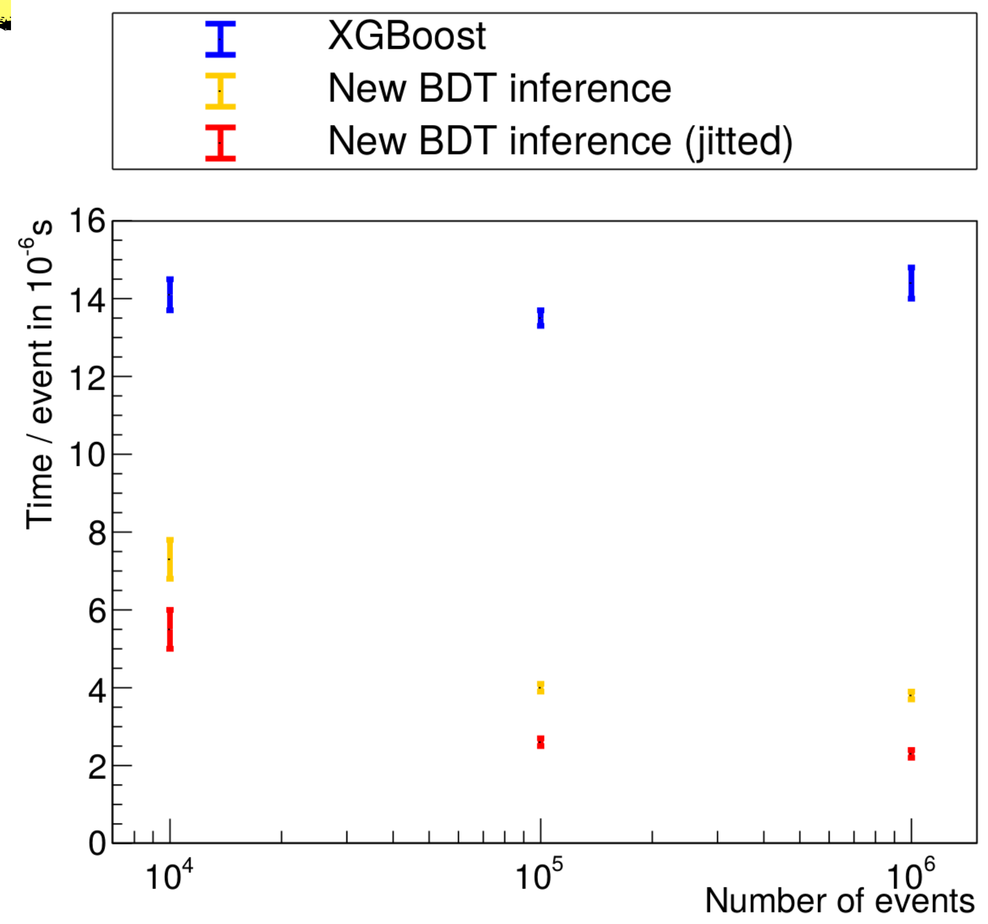
```
xgb = xgboost.BDTClassifier(options)
xgb.fit(x, y)

ROOT.TMVA.SaveXGBoost(xgb, "myBDT", "model.root")
```

- C++ application

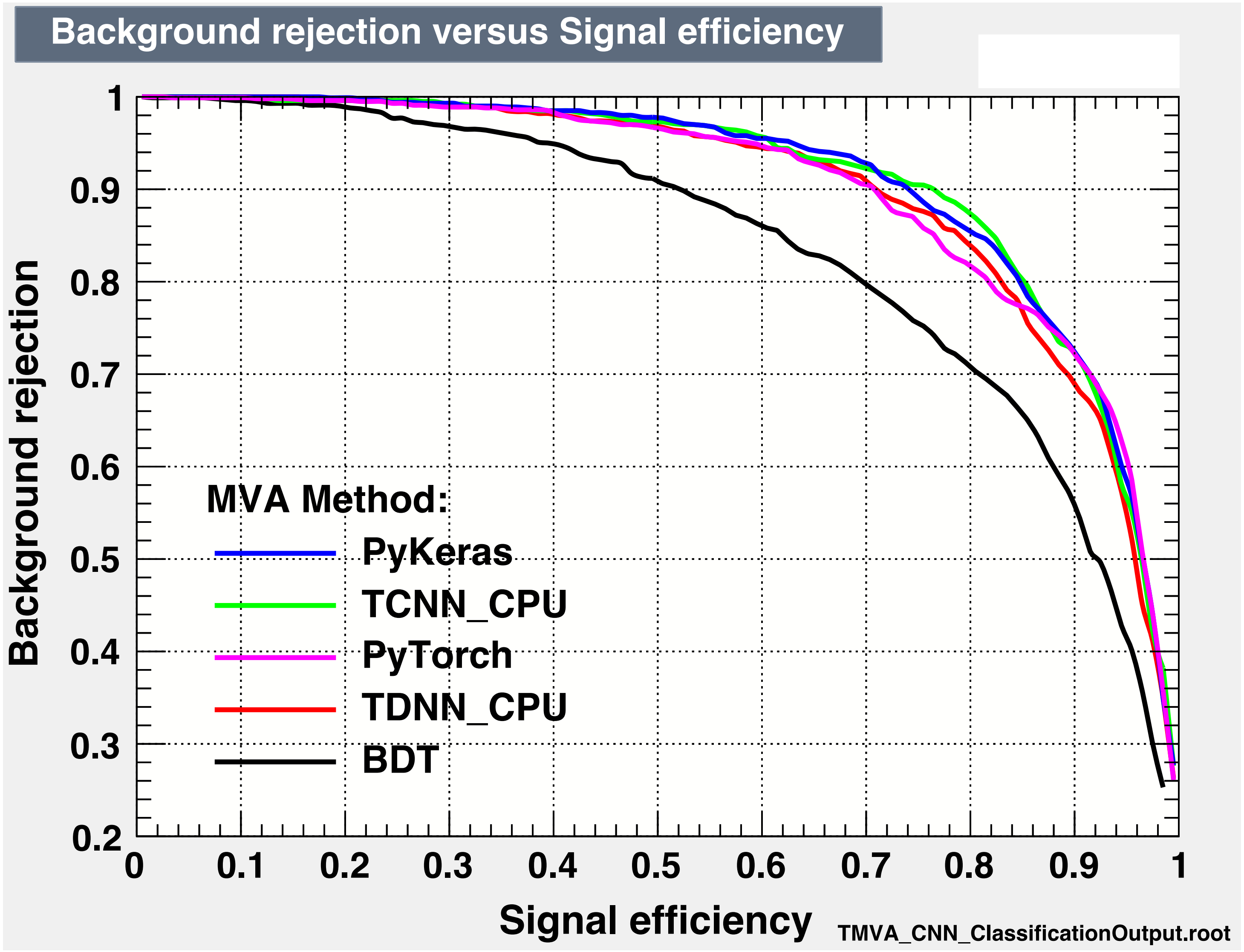
```
TMVA::RBDT bdt("myBDT", "model.root");
auto y1 = bdt.Compute({1.0, ...});

auto x = TMVA::RTensor<float>(data, shape);
auto y2 = bdt.Compute(x);
```



Interoperability of TMVA

- Working on Pythonization of TMVA interfaces
  - moving from string API to python keyword args
- Possible training of Python ML within TMVA workflow
  - interfaces to scikit-learn, Keras and PyTorch



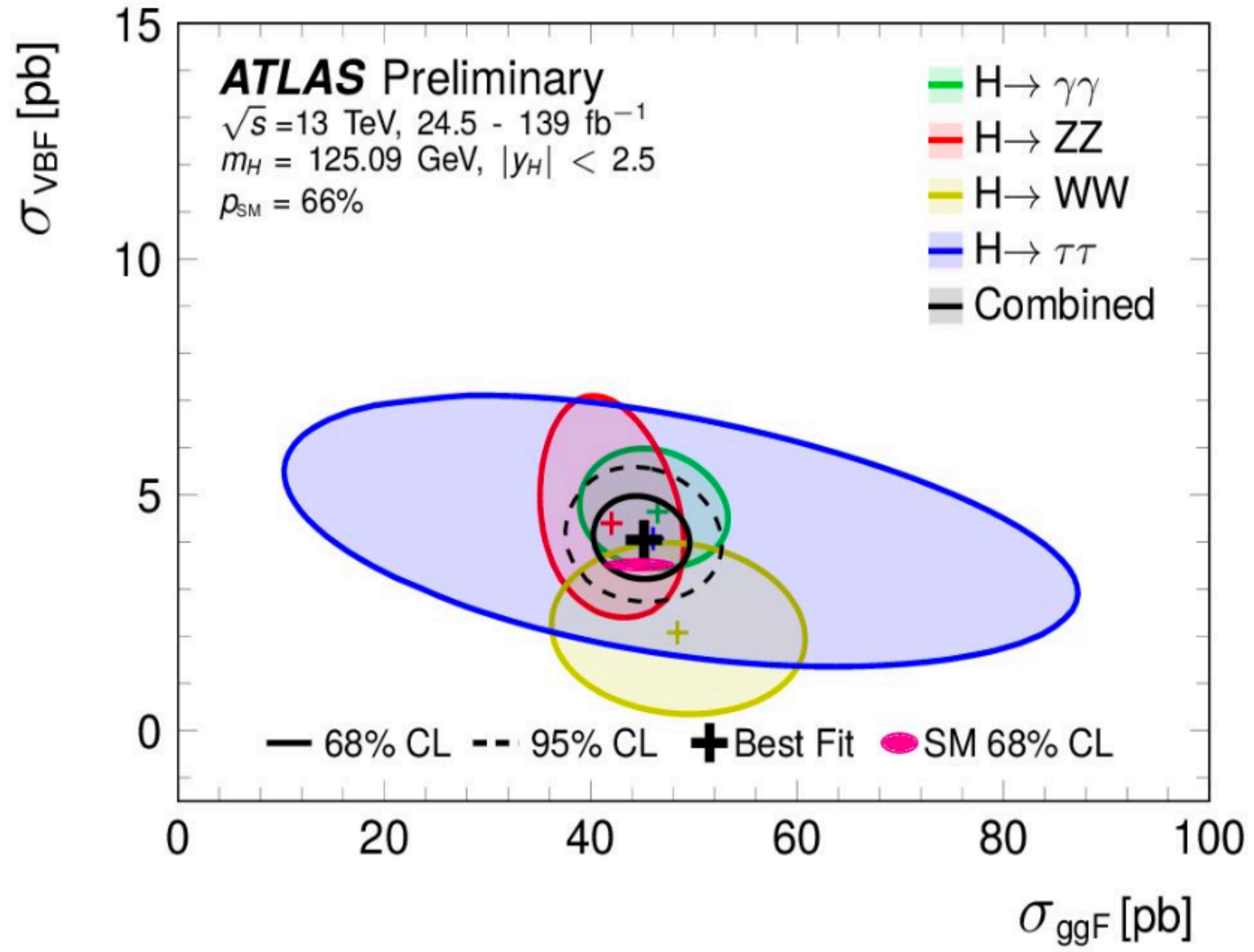
- Working on generic data-loader for ML workflows
  - Generator doing batching and shuffling from ROOT files on the fly
  - Allows for training on huge datasets
  - Direct feeding of data from disk to GPU

Example of a possible ML workflow loading batches

```
df = ROOT.RDataFrame("Events", "http://file.root")
generator = TMVA.BatchGenerator(df, cols, batchSize)
for step in gradientSteps:
    x = generator()
    model.fit(x)
```

Statistical Modeling

- Physics analyses continuously generate increasingly complex likelihood models to describe their data
  - Higgs combination fits, EFT interpretations
    - O(1000) parameters
    - O(100) likelihood components
    - O(100) datasets



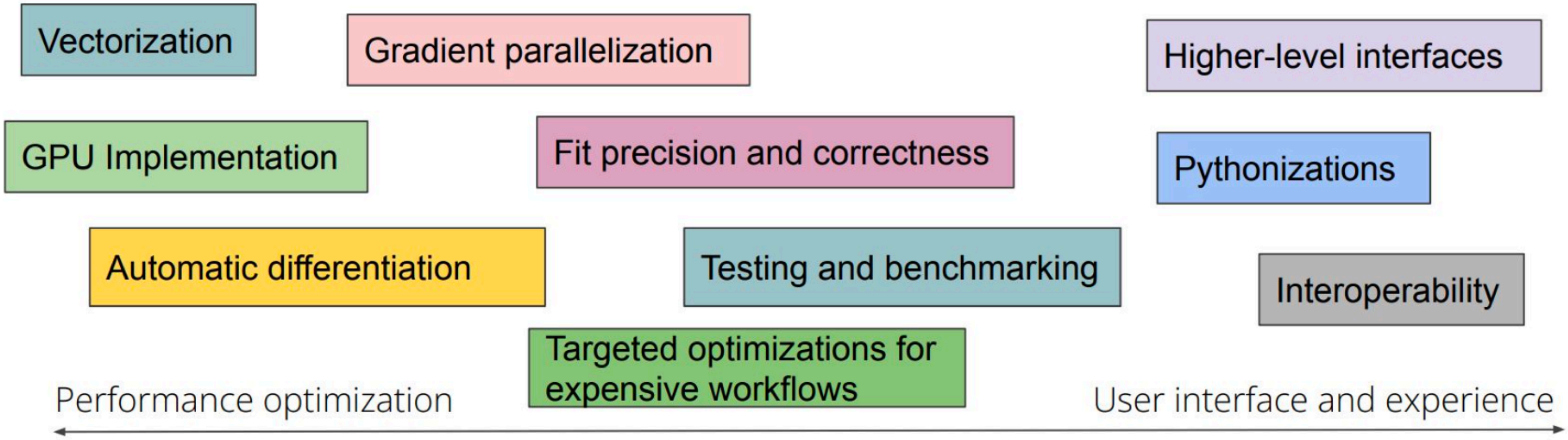
Recent ATLAS Higgs combination fit result [1]. This fit took about five hours to complete.

- Only one tool capable of handling such models: RooFit
- Recent development give possibility to bring down fitting time from hours to minutes
  - from a work day to a coffe break!

from Z. Wolffs: RooFit@ICHEP 2022

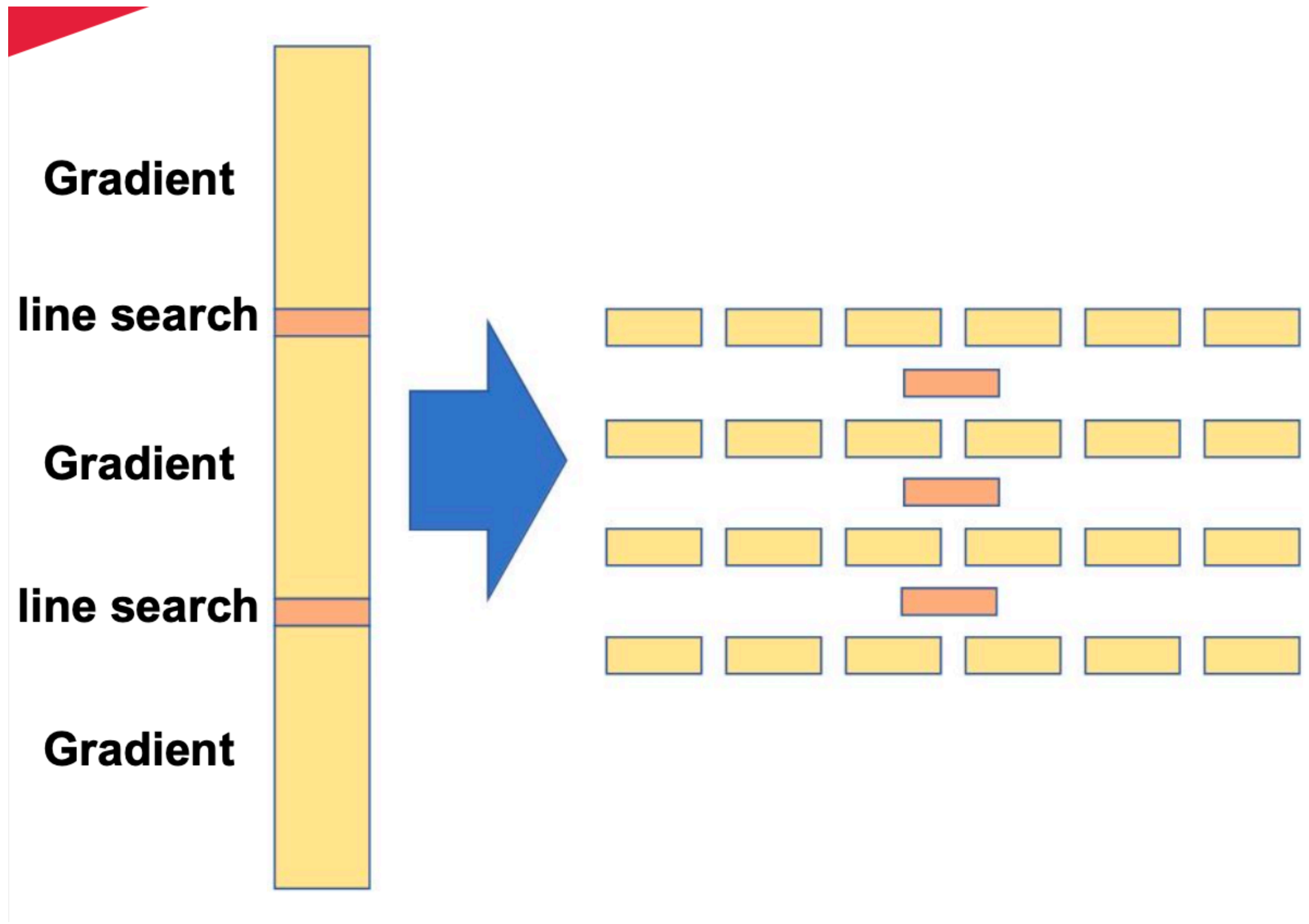
RooFit Evolution

RooFit is evolving on many different areas:



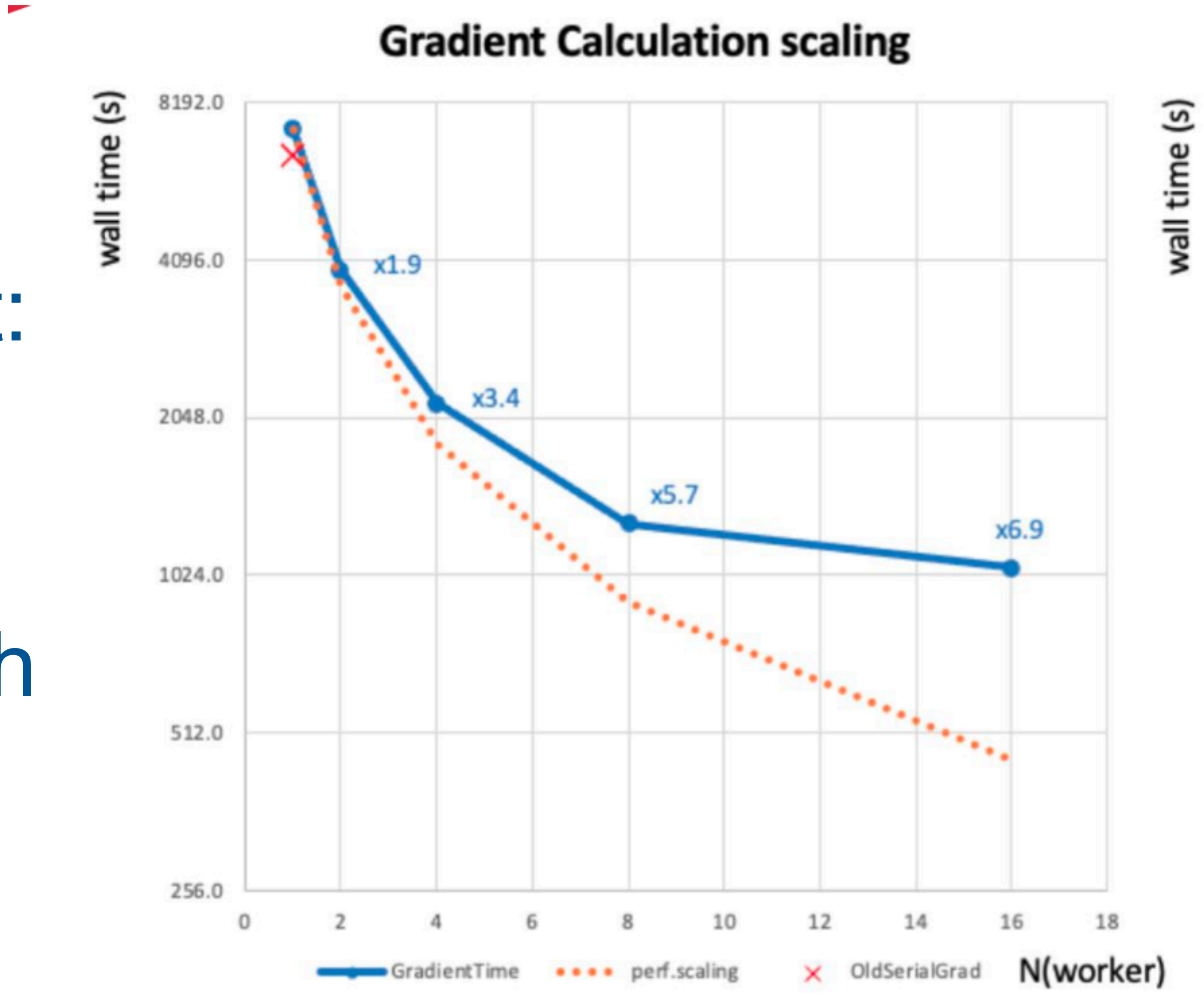
from Z. Wolffs: RooFit@ICHEP 2022

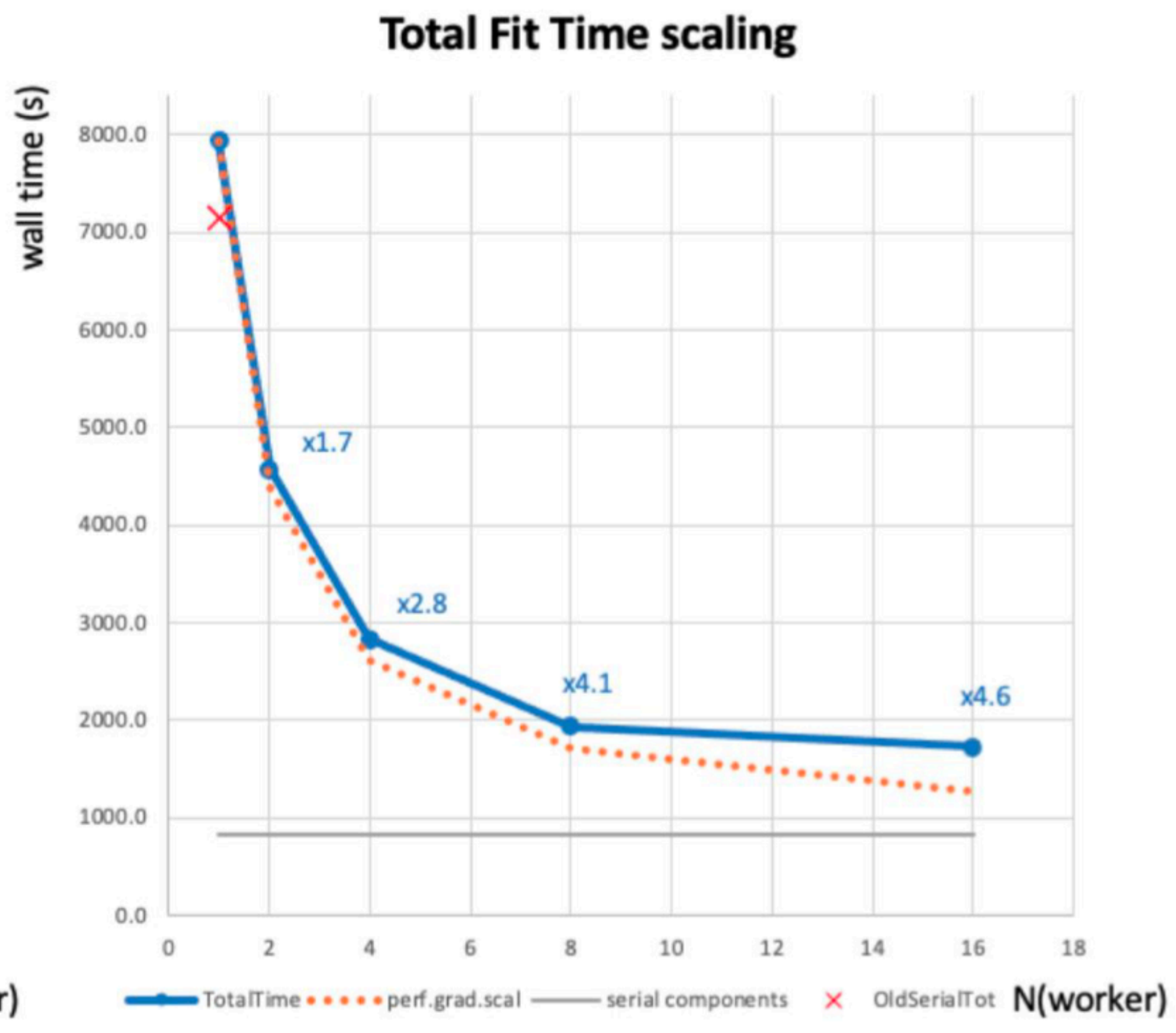
Gradient parallelization



- Parallelize at gradient calculation level
  - N parameters: ~O(2N) function evaluations for computing numerical derivatives
  - Line search: serial part ~O(3) function evaluation
- Dynamic load balancing over workers through random work stealing algorithm
  - complexity of derivative calculation varies
- Designed to have maximum speed impact of complex fits with many parameters

- Wall time decrease in Higgs combination fit:
  - from 2h12m26s → 28m52s (~4/5 times faster)
- Result validated: all parameters agreed with serial fit within 1% of their uncertainties !

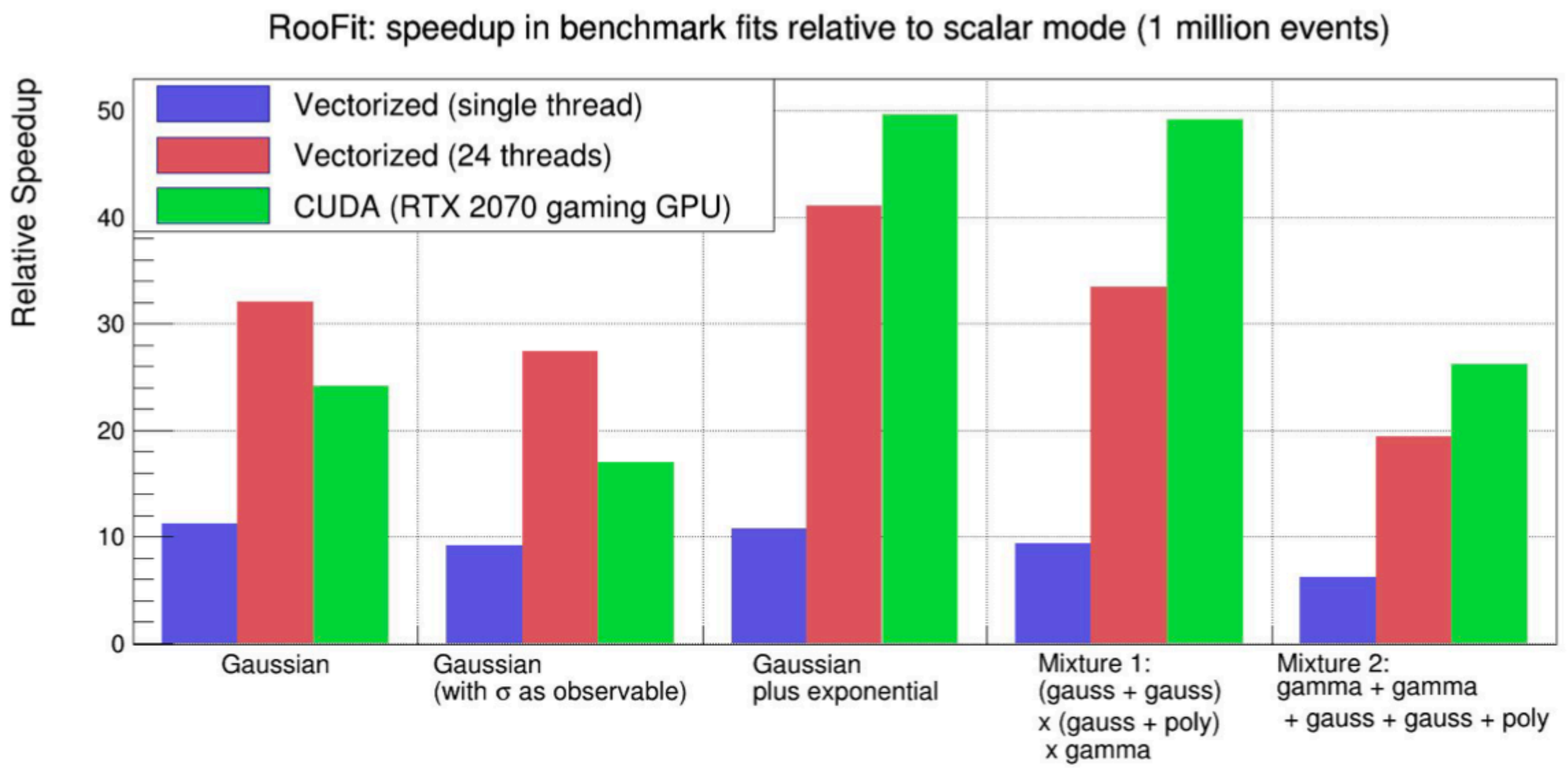


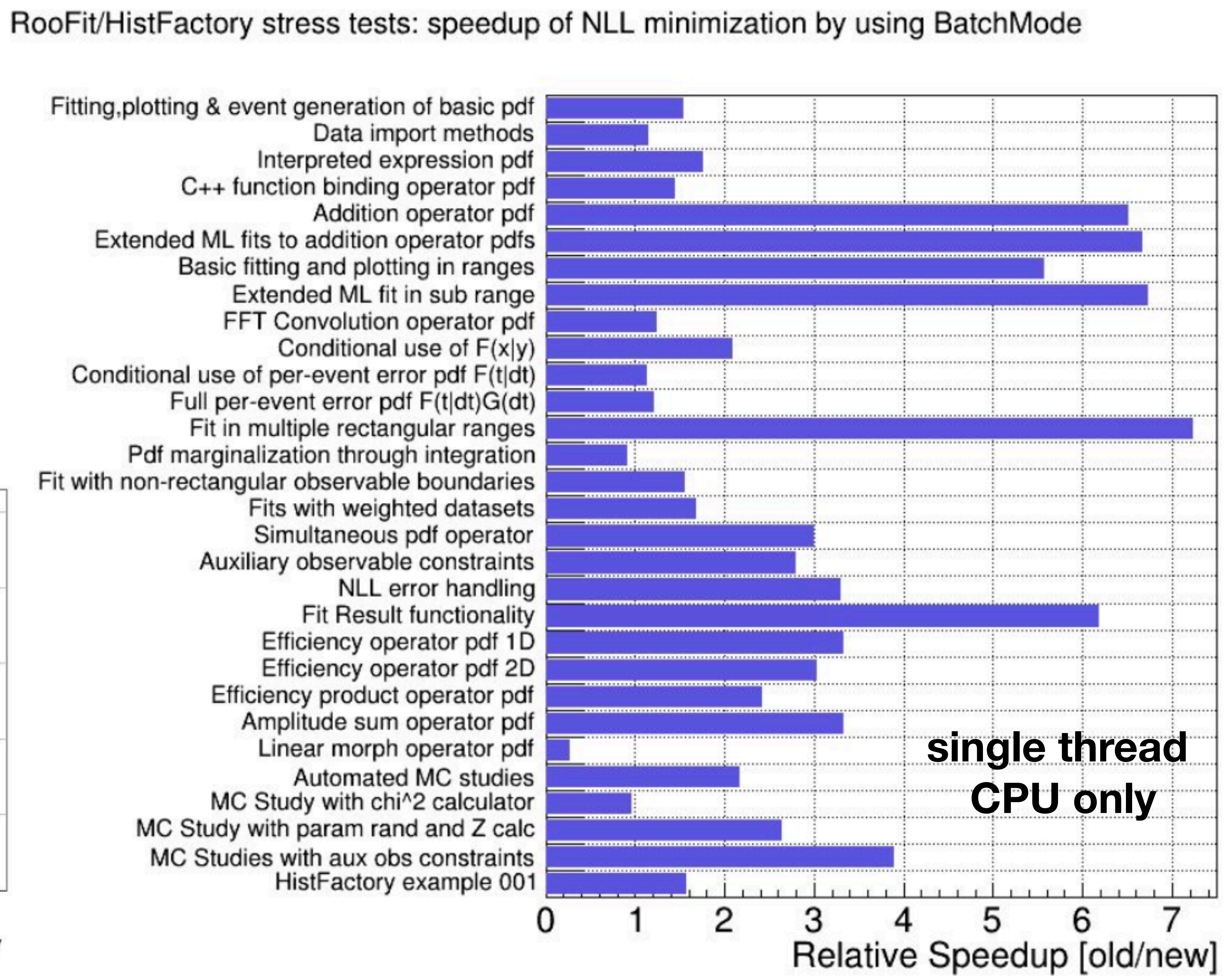


from Z. Wolffs: RooFit@ICHEP 2022

Vectorisation and GPU Computation

- Batch evaluation of RooFit models:
  - allow code vectorisation
  - optionally allow multi-threading or GPU computation (with CUDA)
- possible due to a re-structure of RooFit computational graph.
  - from event-by event to batch node evaluation





➡ Additional speed-up obtained in Batch mode by better CPU caching and vectorisation !

from J. Rembser@PPP meeting

RooFit Pythonizations

- PyROOT bindings more pythonic in latest ROOT (6.26)
- Now you can:
  - use Python keyword arguments instead of RooFit command arguments
  - pass around Python sets or lists instead of RooArgSet or RooArgList
  - pass Python dictionaries instead of `std::map<>`
  - implicitly convert floats to RooConstVar in RooArgList/Set constructors
- All Pythonizations are documented in the reference guide

Example code from a RooFit tutorial

```python
# Create background pdf poly(x)*poly(y)*poly(z)
px = ROOT.RooPolynomial("px", "px", x, [-0.1, 0.004])
py = ROOT.RooPolynomial("py", "py", y, [0.1, -0.004])
pz = ROOT.RooPolynomial("pz", "pz", z)
bkg = ROOT.RooProdPdf("bkg", "bkg", [px, py, pz])

# Create composite pdf sig+bkg
fsig = ROOT.RooRealVar("fsig", "signal fraction",
                       0.1, 0., 1.)
model = ROOT.RooAddPdf("model", "model",
                       [sig, bkg], [fsig])

data = model.generate((x, y, z), 20000)

# Make plain projection of data and pdf on x observable
frame = x.frame(Title="Projection on X", Bins=40)
data.plotOn(frame)
```

from J. Rembser@ACAT 2021

Interoperability with NumPy

- New **converters** between **NumPy** arrays/ **Pandas** dataframes and **RooDataSet/ RooDataHist**:
  - `RooDataSet.from_numpy()`
  - `RooDataSet.to_numpy()`
  - `RooDataSet.from_pandas()`
  - `RooDataSet.to_pandas()`
  - `RooDataHist.from_numpy()`
  - `RooDataHist.to_numpy()`
- New **RooRealVar.bins()** function to get RooFit bin boundaries as NumPy array

```
from ROOT import RooRealVar, RooCategory, RooGaussian

x = RooRealVar("x", "x", 0, 10)
cat = RooCategory("cat", "cat",
                  {"minus": -1, "plus": +1})
mean = RooRealVar("mean", "mean",
                  5, 0, 10)
sigma = RooRealVar("sigma", "sigma",
                   2, 0.1, 10)
gauss = RooGaussian("gauss", "gauss",
                    x, mean, sigma)


data = gauss.generate((x, cat), 100)
df = data.to_pandas()
```

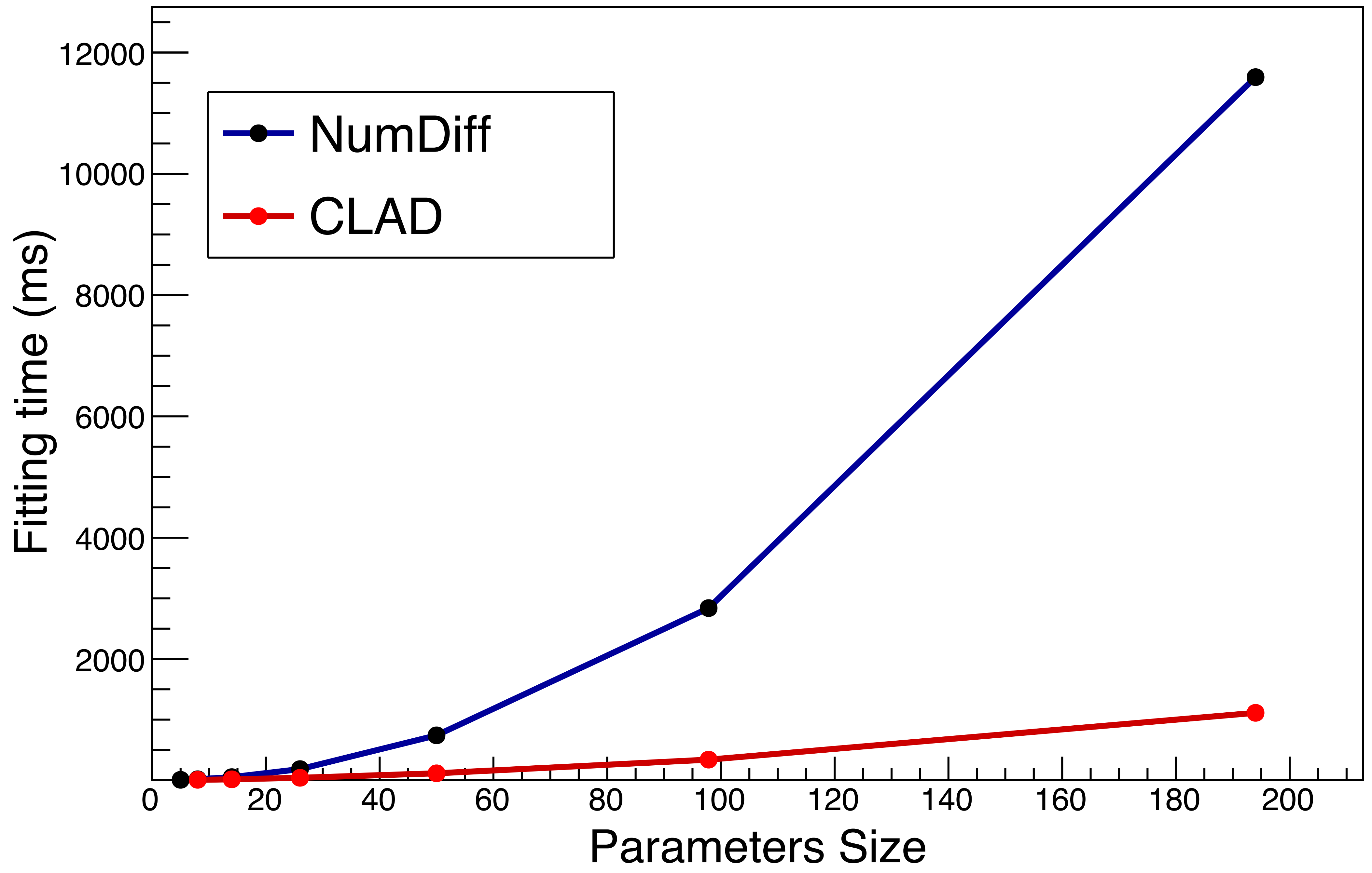
	x	cat
0	6.997865	-1
1	7.211196	-1
2	3.198248	1
3	5.015824	1
4	7.782388	1
...	...	...
95	6.878027	-1
96	0.475900	1
97	4.451101	-1
98	3.481015	-1
99	4.010105	-1

100 rows × 2 columns

from J. Rembser@ACAT 2021

Auto-differentiation (AD)

- AD available in ROOT **TFormula** for fitting
- Large speedup can be obtained with respect to using numerical differentiation in case of large number of parameters
  - ~ 10 speedup for 100 parameters fit
- Integrating also AD for Hessian
  - need to add support in Minuit



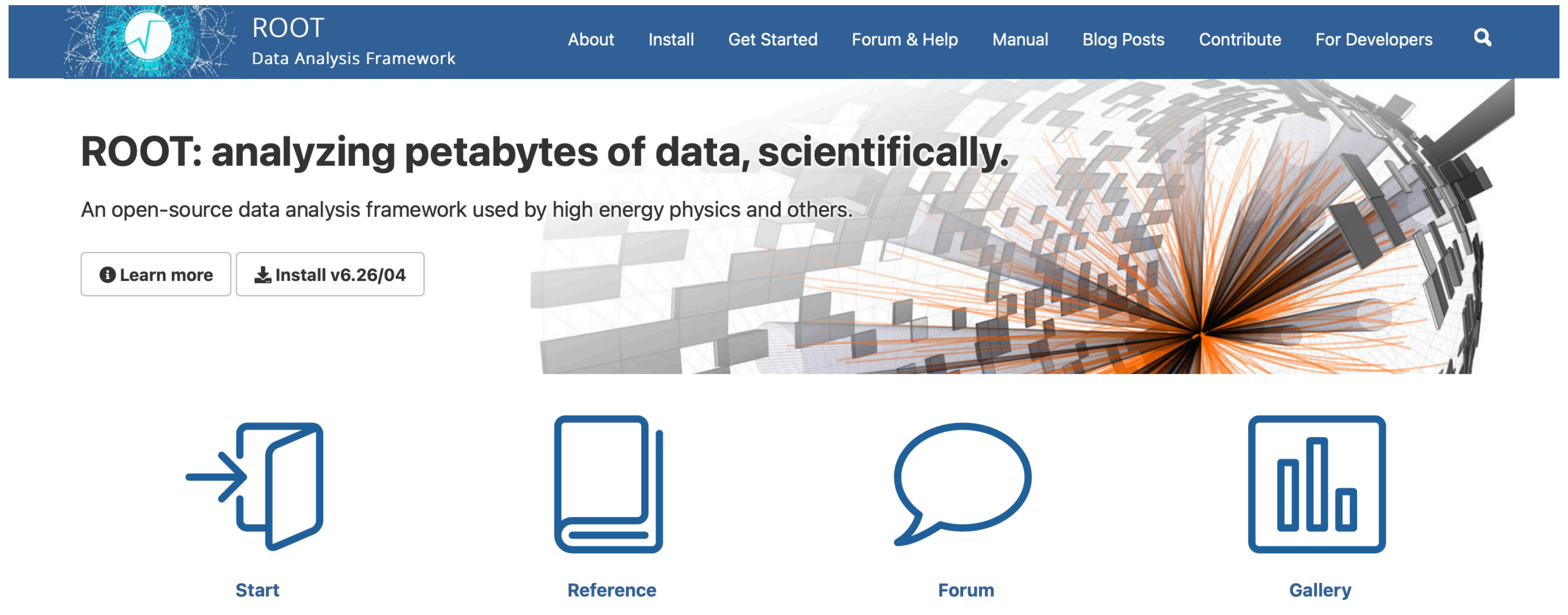
- Working on integrating AD in RooFit
  - starting with a prototype implementation for HistFactory models

Summary

- Future analyses in HEP require tools that are
  - efficient
  - with simple interfaces
  - reliable and supported
  - interoperability (e.g. with Python ecosystem)
- ROOT is evolving, large effort from the whole team
  - aiming to fulfil requirements of future HEP analysis
- Need your help for continuous feedback and support

References

- root.cern
- https://cern.ch/forum
- https://github.com/root-project



- For more information of current developments see presentations at latest ROOT Users Workshop
- see also the ICHEP 22 presentations in the Computing session

Backup Slides