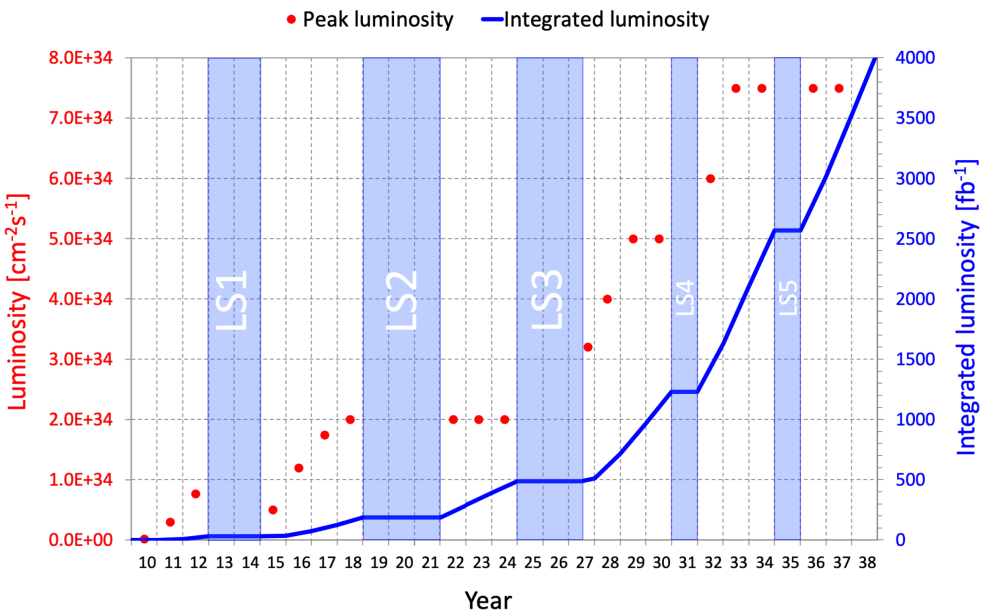
# High performance analysis with RDataFrame: Scaling and Interoperability

Josh Bendavid (CERN/EP-CMG)
with input from
M. Cipriani, M. Dünser (CERN),
J. Eysermans, K. Long (MIT)
E. Manca (UCLA)

May 11, 2022
ROOT User's Workshop

# Introduction




CMS-TDR-0122

- **Huge** amount of data to be collected in HL-LHC-era, 20x increase over today
- Interplay between integrated luminosity, physics program, trigger strategy, but ∼**all searches and measurements across all final states/phase space regions will have significantly more data and MC to analyze**

# Precision W measurements as a prelude to HL-LHC computing

- Personally working on precision W measurements in CMS
- Inclusive W production is among the highest cross section electroweak processes at the LHC $\rightarrow$ more than $3 \times 10^9 W \rightarrow \ell\nu$ produced per lepton flavour in LHC run 2 per experiment
- Example analysis **for** $1/4$ **of total run 2 integrated luminosity and one lepton flavour**:
  - 800M single lepton-triggered data events with little to no scope for skimming
  - 1.5B **Signal** Monte Carlo events with little to no scope for skimming
- For this type of analysis **HL-LHC is now**

# Introduction

- Broad analysis steps
  1. Production of NANOAOD (on the grid)
  2. Preparation/measurements of calibrations and corrections ("Auxiliary Workflows")
  3. **NANOAOD → histograms (nominal + systematic variations)**
  4. Statistical analysis (maximum likelihood fit)

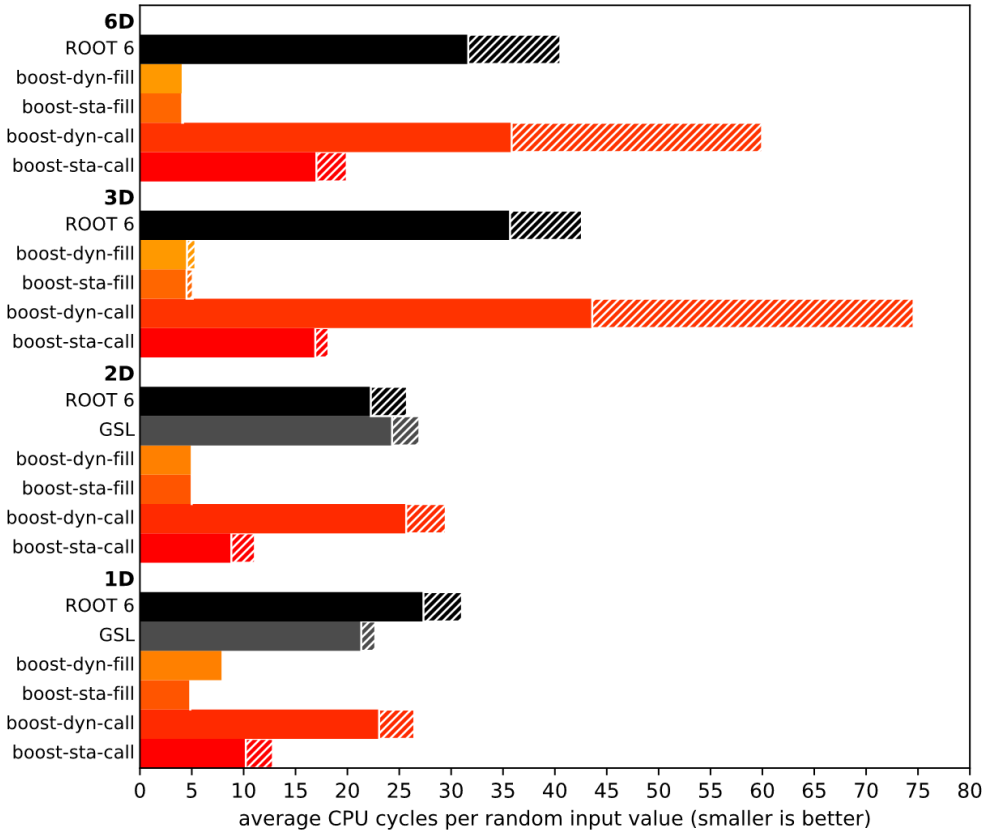
# ROOT-python ecosystem interoperability

- A few frustrating or confusing things for users
  - Uproot and PyROOT provide different python representations of the same ROOT objects (TFile, TH1, etc)
    - Uproot does provide from_pyroot and to_pyroot functions which can help bridge the gap, can also convert between python boost hists and root hists (though no THn support yet)
  - Not straightforward to use boost histogram python bindings in RDF
- PyROOT/cppyy can effectively create automatic python bindings to the C++ Boost histogram library just by including the header
  - **without** nice pythonic interface (UHI indexing etc)
  - **with** (much) more template flexibility
  - Not easily serialized (see **GitHub Issue**)
  - Objects are mutually incompatible with "official" python boost histogram bindings (pybind11-based)
- n.b. Recent improvements to Cling which properly allow O3, inlining, and removing runtime checks bring runtime performance on-par with pre-compiled code

# Why Boost Histograms?

- Analysis uses large, complex multidimensional histograms → effort to add HistoND support (THnT<double>) to RDF
- Encountered three serious technical bottlenecks using RDF with HistoND
  - **Memory limits**: Histogram bins use more memory than reasonably available per thread
  - **Long merging times** for per-thread histograms with 256 threads
  - 1GB absolute limit for writing to file
- **Solution:** Use std::atomic<double> (with CAS loop) for atomic aggregation
- C++ Boost Histogram templates provide convenient means to do this
- Python boost hists provide great convenience for indexing and metadata to manage the complexity of multi-dim histograms (and can be easily serialized with pickle)
- Being able to disable underflow/overflow bins leads to huge savings for charge, boolean, category axes

# Event Batches vs Event Loop




https://www.boost.org/doc/libs/1_77_0/libs/histogram/doc/html/histogram/benchmarks.html

- Boost histograms (in C++) provide an interesting example where static/inlining optimizations matter for the per-event case, but largely irrelevant for large batches
- **Make templates as specific as possible when using Boost histograms**

# Bridging python boost-histogram $< - >$ PyROOT divide

- **One option:** import identical version of the boost histogram headers used to compile pybind11 bindings into Cling (jitting can be thought of us a separate translation unit) and pass/cast the pointers between pybind11 and PyROOT/cppyy
  - **Pros:** Zero copy
  - **Cons:** Careful synchronization of headers needed, pushing the limits of ABI compatibility? constrained to precompiled template instances
- **Alternative:** Get the pointer to the storage and reinterpret the memory (the relevant accumulator classes are all standard-layout/layout-compatible)
  - In practice this actually works, but it's extremely difficult to do in a way which formally respects strict aliasing (depending on your interpretation of P0593R6 this **might** be possible in a standard compliant way)

**Defect reports**

The following behavior-changing defect reports were applied retroactively to previously published C++ standards.

| DR | Applied to | Behavior as published | Correct behavior |
|---|---|---|---|
| P0593R6 | C++98 | previous object model did not support many useful idioms required by the standard library and was not compatible with effective types in C | implicit object creation added |

# Bridging python boost-histogram $< - >$ PyROOT divide

- **Strategy:**
  - Carefully choose the appropriate axis type/template instance for each axis **link**
  - Carefully choose the appropriate storage type/template instance
  - atomic types are used for the storage where appropriate, using a generic atomic_adapter accumulator class **link**
  - A "mirror" histogram with compatible axes, layout, etc is created
  - The pybind11 histogram storage is encapsulated by a lightweight c++ class using the numpy array interface (pointer/shape/strides) **link**
  - Conversion functions are implemented for filling/reading C++ Root and/or Boost hists through this array_interface_view
- Conversion functions are provided in python: hist_to_root, root_to_hist, hist_to_pyroot_boost

# Boost Histograms in RDF

- Use pythonization to add a HistoBoost call to RInterface, which takes a list of python boost-histogram axis types **link**
- A helper class is used with RDF Book underneath to fill the C++ boost histogram, and then fill the array_interface_view in the Finalize call **link**
- The returned (cppyy proxy to) RResultPtr has its GetValue, etc methods swapped out at the python level to **return the python boost-histogram instead** (the actual RResultPtr just contains the array_interface_view with the pointer to the underlying memory)

```
axis_eta = hist.axis.Regular(48, -2.4, 2.4, name = "eta")
axis_pt = hist.axis.Regular(29, 26., 55., name = "pt")
hres = df.HistoBoost("muon_eta_pt", [axis_eta, axis_pt],
                                ["Muon_eta", "Muon_pt"])
h = hres.GetValue()
print(type(h)) # <class 'hist.hist.Hist'>
```

- n.b. there are also HistXDWithBoost calls which produce ROOT histograms as normal, but use C++ boost histograms (with templates instantiated on-the-fly as needed) to provide the performance/atomic aggregation benefits

# Filling weight systematic variations in RDF

- Prior to new Vary functionality, several ways to fill systmatic variations "by-hand" using RDF
- Many (not all) systematics can be represented purely by a change of weight of Monte Carlo events (PDF variations, hard process scale variations, lepton efficiency uncertainties, etc)
- Naive way: One Histogram per variation

```
for ipdf in range(103):
      df = df.Define(f"pdfWeights_{ipdf}", f"weight*LHEPdfWeight[{ipdf}];")
      pdfNNPDF31 = df.HistoND((f"pdfNNPDF31_{ipdf}", "", 5,
        [48, 29, 2, 2, 2], [-2.4, 26.,-2.,-0.5, -0.5], [2.4, 55., 2., 1.5, 1.5]),
        ["goodMuons_eta0", "goodMuons_pt0", "goodMuons_charge0", "passIso", "passMT", f"pdfWeights_{ipdf}"])
```

- Yes the "nominal" histogram here is already an obnoxious 5D hist with $> 11,000$ bins +overflow/underflow (eta x pt x charge x pass_isolation x pass_mt) and we just made 103 of them for PDF variation

# Filling weight systematic variations in RDF

- **Much** better way: A single histogram with one additional axis for the PDF variation index

```
df = df.DefinePerSample("pdfsyst_idx", "std::array<int, 103> res;  std::iota(res.begin(), res.end(), 0); return res;")
df = df.Define("pdfWeights_rvec", "weight*LHEPdfWeight") # LHEPdfWeight is a 103-element RVec<float>

# all your bins are belong to us
pdfNNPDF31 = df.HistoND(("pdfNNPDF31", "", 6,
    [48, 29, 2, 2, 2, 103], [-2.4, 26.,-2.,-0.5, -0.5, -0.5], [2.4, 55., 2., 1.5, 1.5, 102.5]),
    ["goodMuons_eta0", "goodMuons_pt0", "goodMuons_charge0", "passIso", "passMT", "pdfsyst_idx", "pdfWeights_rvec"])
```

- Two graph nodes instead of 103 (the indices are a constant)
- Scalars are "broadcast" to be filled repeatedly with the index and weight vectors
- n.b. this is the reason why **PR7499** existed

# Benchmark

- Using 411M events of CMS NANOAOD ($W^+ \to \mu\nu$) and filling **10** copies of the pdf variation histograms
- 256 threads (2 x EPYC 7702)

| Hist Type | Hist Config | Evt. Loop | Total | CPUEff | RSS |
|---|---|---|---|---|---|
| ROOT THnD | 10 x 103 x 5D | 59m39s | 74m05s | 0.74 | 400GB |
| ROOT THnD | 10 x 6D | 7m54s | 25m09s | 0.27 | 405GB |
| Boost ("sta") | 10 x 6D | 7m07s | 7m17s | 0.90 | 9GB |
| Boost ("sta") | 10 x (5D + 1-tensor) | 1m54s | 2m04s | 0.81 | 9GB |
| Boost ("sta") | 1 x (5D + 2-tensor) | 1m32s | 1m42s | 0.77 | 9GB |

- Standard HistoND calls are bogged down by long single-threaded histogram-merging step with so many threads $\to$ long runtime outside of event loop and poor cpu efficiency
- Actual event loop is still slightly faster with Boost histograms $\to$ speed benefit of specific template instances outweighs overhead of atomic accumulation (small when nbins $>>$ nthreads as here)
- Memory usage is much lower with atomic accumulation by construction
- The last rows are 3.5x - 4.5x faster (4MHz!) despite containing the same number of bins in total...

# Tensor Accumulation

| Hist Type | Hist Config | Evt. Loop | Total | CPUEff | RSS |
|---|---|---|---|---|---|
| ROOT THnD | 10 x 103 x 5D | 59m39s | 74m05s | 0.74 | 400GB |
| ROOT THnD | 10 x 6D | 7m54s | 25m09s | 0.27 | 405GB |
| Boost ("sta") | 10 x 6D | 7m07s | 7m17s | 0.90 | 9GB |
| Boost ("sta") | 10 x (5D + 1-tensor) | 1m54s | 2m04s | 0.81 | 9GB |
| Boost ("sta") | 1 x (5D + 2-tensor) | 1m32s | 1m42s | 0.77 | 9GB |

- In the special case of systematic variations represented by weight variations only, filling a 6D histogram N times is wasteful because the first 5 axis indices are identical for every call
- Boost histograms are flexible enough to allow e.g. a std::array<double> as an accumulator type → stay with 5D histogram and move the systematic axis into the weight
- For complex systematics, to keep things organized, might want more than one axis for systematic variations (last row is 103 x 10) → use Eigen::TensorFixedSize
- Tensor accumulation implemented in generic tensor_accumulator adapter wrapping boost histogram accumulator types and Eigen tensors **link**

# Tensor Accumulation

```
ROOT.gInterpreter.Declare("""
template<typename T, std::ptrdiff_t N, typename V>
auto vec_to_tensor_t(const V &vec, std::size_t start = 0) {
  Eigen::TensorFixedSize<T, Eigen::Sizes<N>> res;
  std::copy(vec.begin() + start, vec.begin() + start + N, res.data());
  return res;
}
""")

df = df.Define("pdfWeights_tensor", "vec_to_tensor_t<double, 103>(LHEPdfWeight)")
pdfNNPDF31 = df.HistoBoost(f"pdfNNPDF31", [axis_eta, axis_pt, axis_charge, axis_passIso, axis_passMT],
                      ["goodMuons_eta0", "goodMuons_pt0", "goodMuons_charge0", "passIso", "passMT", "pdfWeights_tensor"])
```

- To make use of this, just need to make the weight column an appropriate TensorFixedSize type
- HistoBoost will automatically detect this and create the histogram with the appropriate tensor_accumulator type (can be combined with atomic as well)
- The python boost-histogram that you get back has additional Integer axes created which correspond to the tensor dimensions
- The user can also provide custom defined axes of the correct size in order to keep track of systematics metadata (pdf indices along the axis, binning for efficiency corrections, etc)

```
pdfNNPDF31 = df.HistoBoost(f"pdfNNPDF31", [axis_eta, axis_pt, axis_charge, axis_passIso, axis_passMT],
                      ["goodMuons_eta0", "goodMuons_pt0", "goodMuons_charge0", "passIso", "passMT", "pdfWeights_tensor"],
                       tensor_axes = [hist.axis.Integer(306000, 103, name = "lhapdfid")])
```

# Axis ordering and Cache Locality

| Hist Type | Hist Config | Evt. Loop | Total | CPUEff | RSS |
|---|---|---|---|---|---|
| ROOT THnD | 10 x 103 x 5D | 59m39s | 74m05s | 0.74 | 400GB |
| ROOT THnD | 10 x 6D back | 7m54s | 25m09s | 0.27 | 405GB |
| ROOT THnD | 10 x 6D front | 13m52s | 30m27s | 0.42 | 406GB |
| Boost ("sta") | 10 x 6D back | 7m07s | 7m17s | 0.90 | 9GB |
| Boost ("sta") | 10 x 6D front | 3m22s | 3m33s | 0.86 | 9GB |
| Boost ("sta") | 10 x (5D + 1-tensor) | 1m54s | 2m04s | 0.81 | 9GB |
| Boost ("sta") | 1 x (5D + 2-tensor) | 1m32s | 1m42s | 0.77 | 9GB |

- In the tensor/array weight-case the weights for the different systematic idxs are contiguous in memory by construction
- In the N+1-d histogram case it depends on the array ordering
- TH1/2/3 and boost-histograms have fortran array ordering $\rightarrow$ systematic idx axis is best at the **front**
- THn has C array ordering $\rightarrow$ systematic idx axis is best at the **back**
- The difference is about a factor of 2 for both root and boost hists (but still $>$ 50% additional gain from tensor filling)
- largely accounted simply by skipping the extra FDIVs needed for redundant value-to-index conversion for the 5 axes

# Optimized weight systematic filling vs RDF Vary

| Hist Type | Hist Config | Evt. Loop | Total | CPUEff | RSS |
|---|---|---|---|---|---|
| ROOT THnD | 10 x 103 x 5D | 59m39s | 74m05s | 0.74 | 400GB |
| ROOT THnD | 10 x 6D back | 7m54s | 25m09s | 0.27 | 405GB |
| ROOT THnD | 10 x 6D front | 13m52s | 30m27s | 0.42 | 406GB |
| Boost ("sta") | 10 x 6D back | 7m07s | 7m17s | 0.90 | 9GB |
| Boost ("sta") | 10 x 6D front | 3m22s | 3m33s | 0.86 | 9GB |
| Boost ("sta") | 10 x (5D + 1-tensor) | 1m54s | 2m04s | 0.81 | 9GB |
| Boost ("sta") | 1 x (5D + 2-tensor) | 1m32s | 1m42s | 0.77 | 9GB |

- **TODO:** test where new RDF Vary functionality fits in here (likely somewhere in between)
- Vary covers more general case than just weight variations → can it detect and/or optimize for the weight systematic case?
- Will Vary provide the option to fill systematics along an additional axes instead of filling vectors of histograms? (very relevant for O(100) or O(1000) variations)
- Vary is currently tied to ROOT histograms(?), can it be interfaced with custom helper/filler classes via RDF Book or similar?

# Aside on Threading Efficiency

| Hist Type | Hist Config | Evt. Loop | Total | CPUEff | RSS) |
|---|---|---|---|---|---|
| ROOT THnD | 10 x 103 x 5D | 59m39s | 74m05s | 0.74 | 400GB |
| ROOT THnD | 10 x 6D | 7m54s | 25m09s | 0.27 | 405GB |
| Boost ("sta") | 10 x 6D | 7m07s | 7m17s | 0.90 | 9GB |
| Boost ("sta") | 10 x (5D + 1-tensor) | 1m54s | 2m04s | 0.81 | 9GB |
| Boost ("sta") | 1 x (5D + 2-tensor) | 1m32s | 1m42s | 0.77 | 9GB |

- Reaching this level of threading efficiency required some additional improvements to ROOT **PR9486**, **PR10318**
- This is related to use of global lists to manage TFiles and TChains and therefore take the global write lock during the event loop
- a.k.a "fully scalable mode" from Phillipe's talk, which is now the default for TTreeProcessorMT (and hence multithreaded RDF reading TTrees)

# Full Analysis Performance

- "Full analysis" running on 330M data events, 720M signal events and 360M background events
- Nominal hist: 5D with >11,000 bins
- Up to 2800 variations depending on the process
- Total runtime: 9 minutes

```
Samples: 1M of event 'cycles:u', Event count (approx.): 233539313904768
  Children      Self  Command          Shared Object                                  Symbol
+   59.55%     0.94%  python           libTree.so                                     [.] TBranch::GetEntry
+   57.68%     0.00%  python           libCore.so                                     [.] R__unzip
+   57.68%     0.00%  python           libCore.so                                     [.] R__unzipLZMA
+   57.65%     0.00%  python           liblzma.so.5.2.5                               [.] lzma_code
+   57.58%     0.00%  python           liblzma.so.5.2.5                               [.] 0x00007f2a6a5db76b
+   57.35%     0.14%  python           libTree.so                                     [.] TBranch::GetBasketAndFirst
+   57.31%     0.00%  python           libTreePlayer.so                               [.] ROOT::Detail::TBranchProxy::Read (inlined)
+   57.21%     0.01%  python           libTree.so                                     [.] TBranch::GetBasketImpl
+   57.17%     0.01%  python           libTree.so                                     [.] TBasket::ReadBasketBuffers
+   56.98%     0.00%  python           libTreePlayer.so                               [.] At (inlined)
+   56.31%     0.00%  python           liblzma.so.5.2.5                               [.] 0x00007f2a6a5d9b9f
+   56.15%     0.00%  python           liblzma.so.5.2.5                               [.] 0x00007f2a6a5df205
+   55.99%     0.00%  python           liblzma.so.5.2.5                               [.] 0x00007f2a6a5e7b2c
+   48.67%     0.00%  python           [JIT] tid 1996                                 [.] 0x00007f29b12e700c
+    5.86%     0.00%  python           [JIT] tid 1996                                 [.] 0x00007f29b1000fdc
+    5.59%     5.57%  python           [JIT] tid 1996                                 [.] 0x00007f29b128289a
+    5.57%     0.00%  python           libImt.so                                      [.] tbb::detail::d1::start_for<tbb::detail::d1::blocked_range<unsigned int>, tbb:
+    5.57%     0.00%  python           libImt.so                                      [.] tbb::detail::d1::partition_type_base<tbb::detail::d1::auto_partition_type>::e
+    5.57%     0.00%  python           libImt.so                                      [.] tbb::detail::d1::dynamic_grainsize_mode<tbb::detail::d1::adaptive_mode<tbb::d
+    5.57%     0.00%  python           libImt.so                                      [.] tbb::detail::d1::start_for<tbb::detail::d1::blocked_range<unsigned int>, tbb:
+    5.57%     0.00%  python           libImt.so                                      [.] tbb::detail::d1::parallel_for_body_wrapper<std::function<void (unsigned int)>
+    5.57%     0.00%  python           libImt.so                                      [.] std::function<void (unsigned int)>::operator() (inlined)
+    5.43%     0.00%  python           libtbb.so.12.5                                 [.] 0x00007f2a69704abd
```

- TLDR from profiler: 58% on R__unzipLZMA
- Probably should just use zstd
- We've taken a lot of care to minimize malloc
- Debug symbols for jitted code would be great...

# Initialization Time

- Initializing scale factors/corrections and corresponding helpers, plus building the graphs takes about 40s, a lot of it is jitting time

```
Samples: 5K of event 'cycles:u', Event count (approx.): 123375124439
  Children      Self  Command          Shared Object                                  Symbol
+   74.76%     0.00%  python           [unknown]                                      [k] 0xffffffffffffffff
+   31.59%     0.20%  python           libpython3.10.so.1.0                           [.] _PyEval_EvalFrameDefault
+   30.42%     1.04%  python           libCling.so                                    [.] clang::RedeclarableTemplateDecl::loadLazySpecializationsImpl
+   29.71%     0.00%  python           libCling.so                                    [.] llvm::legacy::PassManagerImpl::run
+   29.31%     0.03%  python           libCling.so                                    [.] clang::TemplateArgumentList::ComputeODRHash
+   29.26%     0.00%  python           libpython3.10.so.1.0                           [.] _PyFunction_Vectorcall
+   28.79%     0.00%  python           libCling.so                                    [.] TClingCallFunc::compile_wrapper
+   27.73%     2.99%  python           libCling.so                                    [.] clang::ODRHash::AddQualType
+   27.71%     4.98%  python           libCling.so                                    [.] clang::ODRHash::AddDecl
+   27.71%     0.00%  python           libCling.so                                    [.] clang::ODRHash::AddType (inlined)
+   27.21%     0.00%  python           libCling.so                                    [.] clang::ODRHash::AddTemplateArgument (inlined)
+   27.05%     0.00%  python           libCling.so                                    [.] TClingCallFunc::make_wrapper
+   26.94%     0.27%  python           libCling.so                                    [.] llvm::FPPassManager::runOnFunction
+   26.92%     0.00%  python           libCling.so                                    [.] TClingCallFunc::IFacePtr
+   26.85%     0.00%  python           libcppyy_backend3_10.so                        [.] WrapperCall
+   26.39%     0.00%  python           libCling.so                                    [.] cling::IncrementalParser::Compile
+   26.39%     0.00%  python           libcppyy3_10.so                                [.] CPyCppyy::CPPMethod::ExecuteFast
+   26.32%     0.04%  python           libcppyy3_10.so                                [.] CPyCppyy::(anonymous namespace)::mp_call
+   26.01%     0.01%  python           libcppyy3_10.so                                [.] CPyCppyy::CPPMethod::ExecuteProtected
+   25.98%     0.00%  python           libcppyy3_10.so                                [.] CPyCppyy::CPPMethod::Execute
```

- 29% in ComputeODRHash which has been **discussed before**
- Templates here are getting pretty complex...

# Initialization Time

- For fun, an example of a template function instance (this one gets compiled implicitly by cppyy, but the explicit declaration would look like this)

```
template ROOT::RDF::RResultPtr<narf::array_interface_view<boost::histogram::accumulators::weighted_sum<double>,9,void>>
narf::book_helper<ROOT::RDF::RInterface<ROOT::Detail::RDF::RJittedFilter,void>,narf::FillBoostHelperAtomic<narf::array_interface_view<boost::h
istogram::accumulators::weighted_sum<double>,
9,void>,boost::histogram::histogram<tuple<boost::histogram::axis::regular<double,boost::use_default,boost::use_default,boost::histogram::axis:
:option::bitset<3> >,boost::histogram::axis::regular<double,boost::use_default,boost::use_default,boost::histogram::axis::option::bitset<3>
>,boost::histogram::axis::regular<double,boost::use_default,boost::use_default,boost::histogram::axis::option::bitset<0>
>,boost::histogram::axis::boolean<boost::use_default>,boost::histogram::axis::boolean<boost::use_default>
>,boost::histogram::storage_adaptor<vector<narf::atomic_adaptor<narf::tensor_accumulator<boost::histogram::accumulators::weighted_sum<double>,
Eigen::Sizes<48,13,2,2> >,void> > > >
>,float,float,int,bool,bool,Eigen::TensorFixedSize<double,Eigen::Sizes<48,13,2,2>,0,long>>(ROOT::RDF::RInterface<ROOT::Detail::RDF::RJittedFil
ter,void>&, narf::FillBoostHelperAtomic<narf::array_interface_view<boost::histogram::accumulators::weighted_sum<double>,
9,void>,boost::histogram::histogram<tuple<boost::histogram::axis::regular<double,boost::use_default,boost::use_default,boost::histogram::axis:
:option::bitset<3> >,boost::histogram::axis::regular<double,boost::use_default,boost::use_default,boost::histogram::axis::option::bitset<3>
>,boost::histogram::axis::regular<double,boost::use_default,boost::use_default,boost::histogram::axis::option::bitset<0>
>,boost::histogram::axis::boolean<boost::use_default>,boost::histogram::axis::boolean<boost::use_default>
>,boost::histogram::storage_adaptor<vector<narf::atomic_adaptor<narf::tensor_accumulator<boost::histogram::accumulators::weighted_sum<double>,
Eigen::Sizes<48,13,2,2> >,void> > > > >&&, const std::vector<std::string>&);
```

# Aside about Eigen Tensors/Arrays

- Need to be a bit careful using Eigen Tensors/Arrays within RDF since they implement expression templates with some interesting semantics for forcing evaluation
- This is also an opportunity...
- Consider the following idiomatic RDF defintion (counts gen leptons)

```
    for i in range(10):
        df = df.Define(f"n_fiducial_leptons_{i}", "Sum((abs(GenPart_pdgId) == 11 || abs(GenPart_pdgId) == 13) && GenPart_status
== 1 && abs(GenPart_eta)<2.4 && GenPart_pt > 25.)")
        sum_fiducial = df.Sum(f"n_fiducial_leptons_{i}")
```

- Since there are many gen particles per event, this may exceed the SmallVector optimization and trigger allocations
- Can directly use e.g. Eigen Arrays to do the same thing (adopting the memory from the RVec using the Eigen::Map)

```
ROOT.gInterpreter.Declare("""
template<typename V>
auto array_view(const V &vec) {
  return Eigen::Map<const Eigen::Array<typename V::value_type, Eigen::Dynamic, 1>>(vec.data(), vec.size());
}
""")
for i in range(10):
    df = df.Define(f"n_fiducial_leptons_{i}", "((array_view(GenPart_pdgId).abs() == 11 || array_view(GenPart_pdgId).abs() ==
13) && array_view(GenPart_status) == 1 && array_view(GenPart_statusFlags).unaryExpr([](int x) {return x & 0x1;}) &&
array_view(GenPart_eta).abs() < float(2.4) && array_view(GenPart_pt) > float(25.)).count()")
    sum_fiducial = df.Sum(f"n_fiducial_leptons_{i}")
```

- Running this 10x per event on 430M events, RVec version is ∼1m30s, Eigen version is ∼1m00s (comparable to by-hand loop)

# Aside about Eigen Tensors/Arrays

- Some question marks about best use of expression templates in a computational graph: when to store unevaluated expressions vs forcing evaluation?
- How to force evaluation without triggering allocation in the function?
  - Eigen::Array with dynamic size has std::vector-like allocation semantics, so a simple solution is to return unevaluated expressions from the function, but store an evaluated Array in the results vector of the RDF::Define object $\rightarrow$ memory can be reused on assignment and will only reallocate when encountering (much) larger events than previously
  - Requires being able to set the result_type of Define calls independent of the function return type (but could also detect this specific case automatically if expression templates were more deliberately supported), also related to **PR9174**

# Numba+Numpy Performance

- Modest overhead from memory allocation/intermediate RVecs visible here (this example has arrays which are usually larger than the small vector optimization in RVec)
- Numba + Numpy has **terrible** performance
  - Enrico and Ivan tracked this down to atomic operations in memory allocation/deallocation functions of Numba's numpy implementation → contention between threads/broken scaling
  - Hopefully possible to fix/avoid

| Implementation | Time |
|---|---:|
| C++ for loop | 62s |
| C++ RVec | 86s |
| C++ Eigen::Array | 58s |
| python numba for loop | 63s |
| python numba + numpy | >40 minutes!? |

# Numba+Numpy Performance

```
@ROOT.Numba.Declare(["RVec<int>", "RVec<int>", "RVec<int>", "RVec<float>", "RVec<float>" ], "int")
def count_leptons(pdgId, status, statusFlags, pt, eta):
    abspdg11 = np.abs(pdgId) == 11
    abspdg13 = np.abs(pdgId) == 13
    return np.count_nonzero((abspdg11 | abspdg13) & status == 1 & (np.abs(eta)<2.4) & (pt > 25.))

@ROOT.Numba.Declare(["RVec<int>", "RVec<int>", "RVec<int>", "RVec<float>", "RVec<float>" ], "int")
def count_leptons_loop(pdgId, status, statusFlags, pt, eta):
    count = 0
    for i in range(len(pdgId)):
        abspdg11 = abs(pdgId[i]) == 11
        abspdg13 = abs(pdgId[i]) == 13
        if (abspdg11 or abspdg13) and status[i] == 1 and (abs(eta[i])<2.4) and (pt[i] > 25.):
            count += 1
    return count
```

# Aside: Some other PyROOT/cppyy issues/complaints

- Some classes of template functions with auto return type can't be called from PyROOT → Boost histogram factory functions in particular had to be wrapped with versions not using auto
- Error messages from template instantiations in PyROOT are obscure and/or hidden → for debugging I had to explicitly declare template instances with TInterpreter::Declare, which gives sensible error messages (equivalent to compiling with clang)
- Above could be related to different calls used to trigger the jitting? (follow-up from discussion **here**?)

# Some take-aways

- A fair bit of tools/utilities have been implemented for interoperability between ROOT and Boost histograms both in python and C++
- Huge benefits from atomic accumulation in certain circumstances (ie very large histograms)
- Major performance gains for weight-based systematics by using array-like accumulators
- Eigen::TensorFixed size is a convenient realization of this since it allows to naturally organize the variations along multiple axes
- Some other interesting possibilities related to the use of Eigen with RDF (expression templates)
- Jitting these templates is slow, but we know why (can it be improved?)

# Next Steps

- Some of what I've shown/written could be upstreamed to C++ Boost Histogram library (helpers/adapters for accumulation etc)
- Interoperability between python boost-histogram and PyROOT is not perfect (requires a copy), worth implementing full pythonization of boost histograms in PyROOT? (but then how to serialize them? hook into boost serialization? implement something using numpy a la boost-histogram for python pickling?), other ideas?
- On the ROOT side one could take this either as a set of lessons/examples/desired functionality for future histogram and RVec/related development OR as a push to more centrally support use of "foreign" objects with ROOT and RDF (push HistoBoost RDF functionality into ROOT for example?)

# How are we using this for analysis in practice?

- Basic utilities and extensions of RDataFrame functionality are implemented in `https://github.com/bendavid/narf/`
  - Histogram conversion functions:
    - **hist_to_root, root_to_hist, hist_to_pyroot_boost**
    - Some overlap with uproot functionality (but more complete support for Nd histograms)
  - RDataFrame extended functionality
    - **HistoBoost:** Produce python hist histograms directly (C++ boost histograms used underneath for filling)
    - **Histo1DWithBoost, Histo2DWithBoost, Histo3DWithBoost, HistoNDWithBoost:** Produce (pyroot) Root histograms, but using C++ boost histograms underneath for filling
  - "framework"-like functionality
    - **Dataset class:** Simple dataset metadata
    - Luminosity counting and json filtering tools
    - **build_and_run**: Simple helper function to build RDF graphs given a list of datasets and a user-provided function, run all the graphs, collect the output (also handles luminosity counting and json filtering)

# Not an RDF Framework

- Basic utilities and extensions of RDataFrame functionality are implemented in `https://github.com/bendavid/narf/`
- Basic design principles:
  - Very little abstraction on top of RDF: user-provided python function for graph definition which calls RDF Filter, Define, HistoXD, etc directly
  - Provide only minimal extra functionality which is not available in RDF
    - Metadata for individual samples/datasets
    - Manage the loop over datasets when building the graph
    - Help organize the outputs
  - Run directly on NANOAOD (no post-processing), though anything that works with RDF can also be used as input
- One thing which would be nice is a more coherent treatment of collections (a la nanoevents in Coffea) $\rightarrow$ already on RDF development plan presented by Enrico

# One RDF Graph Per Sample vs One Graph To Rule Them All?

- Currently using one RDF graph per sample (data, W$\rightarrow \mu\nu$, W$\rightarrow \tau\nu$, Z$\rightarrow \mu\mu$...)
- Use of multi-dimensional histograms naturally accommodates one monolithic graph, with "sample index" as an additional histogram axis
- DefinePerSample is an essential ingredient to enable this (but default values for missing branches are really required to fully exploit this $\rightarrow$ also in the RDF development plan)
- if/else logic in graph construction can be moved to DefinePerSample logic, though possibly with some memory overhead for redundant histogram bins (unfilled systematics for some samples, etc)

# Not an RDF Framework

- Output from narf build_and_run is a python dictionary organizing the outputs by dataset, which can e.g. be directly pickled
- Good:
  - Can mix python hists and pyroot objects
  - python hists pickle without ROOT IO (no 1GB limit)
- Bad:
  - Full set of histograms, etc are written (or read back) all at once $\rightarrow$ time/memory implications
  - Would be nice to have a generic solution for writing/reading histograms one at a time, like with root files (something hdf5-based is probably not too difficult)

# Luminosity filtering and counting tools

- Helpers implemented in `https://github.com/bendavid/narf/blob/main/narf/lumitools.py`
- **jsonhelper**
  - construct with a json file
  - resulting helper returns a boolean given run and lumi section (to be used with RDF Filter)
  - narf inserts this into the graph for the event loop automatically using build_and_run when a json file is provided as part of the sample metadata
- **lumihelper**
  - construct with a csv file containing integrated luminosity per run and lumi section (from brilcalc with –byls option)
  - resulting helper returns the integrated luminosity given a run and lumi section
  - narf automatically constructs a graph using build_and_run running on the LuminosityBlocks tree in NANOAOD and computes the integrated luminosity for the analyzed data on the fly
    - guarantees consistent luminosity calculation and **extremely** convenient for running on partial statistics
    - at least for data on local ssd this takes only a few seconds
- These helpers can also be used standalone in principle

# How are we using this for analysis in practice?

- Analysis specific stuff can be implemented in a separate repository
  - narf is used as a git submodule
  - Where external data is needed (scale factors/corrections/etc) C++ helper classes are implemented holding Root or C++ boost histograms, etc
    - these can be used directly in an RDF Filter or Define as long as they implement a non-overloaded operator ()
    - Plans/WIP to significantly extend this functionality in RDF, see **ROOT PPP presentation**
    - could also use correctionlib for some things (but all of our corrections are custom/somewhat involved for the moment)
  - "Top-level" analysis implemented from a main python file, but of course utilities and re-usable graph subsets are factorized into other files, functions, etc

# Software Environment/Packaging

- So far have required custom Root builds to have all the needed patches, and very recent Eigen and C++ Boost versions
- Singularity image available at
  `/cvmfs/unpacked.cern.ch/gitlab-registry.cern.ch/bendavid/cmswmassdocker/wmassdevrolling\:latest`

  (also with relatively complete set of python stuff)
- Getting good performance also requires using O2 or O3 for jitting: (this can be forced from environment variables → will probably set it globally in the singularity image in the future)
  - `import ROOT; ROOT.gInterpreter.ProcessLine(".O3")`
  - can also be forced from environment variables → will probably set it globally in the singularity image in the future
  - Will also become less important when LLVM is updated in ROOT in the future
- All needed improvements to ROOT are now upstream (in both master and v6-26-00-patches though not yet in 6.26.02)
- Once next 6.26 patch release is out, can probably do things in a conda environment (or native Arch packages) as well

# Hardware and I/O

- Using a (very) big single machine for the moment provided by CERN IT for analysis and R&D:
  - 2 x EPYC 7702 64 core (128 cores/256 threads total)
  - 1TB memory
  - 20 x 3.84 TB Gen4 NVME SSDs
    - Main storage array has 16 Gen4 NVME SSDs in raid0 $\rightarrow$ 100Gbytes/sec sequential read and 60Gbytes/sec sequential write in synthetic benchmarks
  - 100gbps Network Interface (+ connection to eos)
    - n.b. currently limited to $\sim$ 50gbps in parallel xrdcp from eos due to only 8 receive queues/interrupt handling threads $\rightarrow$ improve through kernel configuration parameters?

# Hardware and I/O: Next Steps

- Would like to (eventually) explore distRDF with Spark/Dask etc for multi-node scaling e.g. at CERN or MIT subMIT system
  - Ability to use multithreaded tasks and RDF Book for custom aggregation (Boost histograms) essential to stay within memory constraints
- Planning in the near future to repeat direct benchmarks for local SSD array vs xrootd+eos vs CephFS (vs XCache?) at CERN to highlight remaining bottlenecks for typical eos-based workflows
- Can consider tests with RNTuple and/or object stores as well following Monday discussion
- Is EOS at CERN sufficient for high performance analysis moving forward, or do we need CephFS and/or (Ceph?) object store? (speaking "only" of final NanoAOD or similar analysis formats for "final" analysis step)

# Benchmark: Gradient Aggregation: IO Limits

| CPU | Storage | Avg. Rate (GBytes/sec) |
|---|---|---|
| 2 x Xeon (32C/64T) | eos/xrootd (eoscms) (25gbps) | 1.64 |
| 2 x Xeon (32C/64T) | eos/xrootd (test inst.) (25gbps) | 2.62 |
| 2 x Xeon (32C/64T) | CephFS HDD (25gbps) | 2.60 |
| 2 x Xeon (32C/64T) | CephFS SSD (25gbps) | 2.64 |
| 2 x Xeon (32C/64T) | Local SSD (16xSATA) | 5.21 |

thanks to IT-ST group for help setting up some of these tests

- Need to set e.g. XRD_PARALLELEVTLOOP=16 to get good eos performance
- EOS+xrootd standard production instance not quite scaling up to network limits (possible xrootd client/ROOT bottlenecks?)
- Extremely good performance of EOS test instance, and CephFS (CentOS 8 kernel client), approaching limits of ethernet connection
- **Reach 5.2GBytes/sec from local SSDs, approaching limits of disk array - PCIE 3.0 8x SATA controller)**, to be tested on newer machine (**much** faster disks and 100gbps network)