RDataFrame and jitting

Runtime comparison of jitted vs compiled code

Enrico Guiraud, Stefan Wunsch

ROOT

Data Analysis Framework

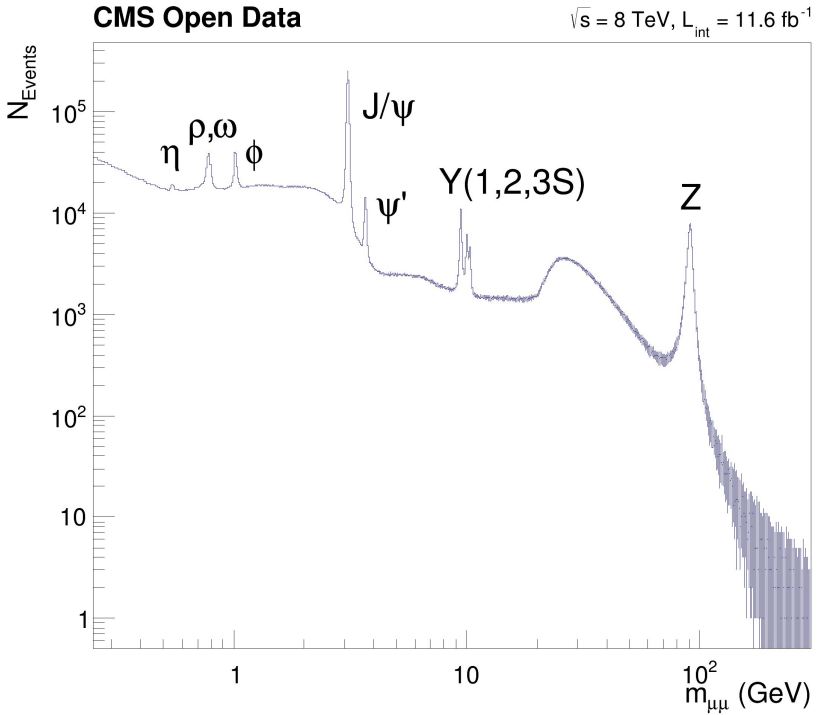
https://root.cern

What is the problem?

- RDF performs considerably worse in Python, which requires jitting parts of the computation graph
- Pressing issue since Python is the language of choice for most RDF users
- **Today:**
  Putting out reproducers and numbers to pin down the issue

The baseline

- Using the tutorial [df102](df102) as the benchmark
- All measurements are single threaded, repeated 5 times and the minimal runtime is reported
- Data read from local SSD with hot cache
- ROOT configuration:
  - `RelWithDebInfo`
  - All other settings on default
  - On state of Dec 17, 2020



C++: Compiled RDF

```
#include "ROOT/RDataFrame.hxx"
#include "ROOT/RVec.hxx"

int main()
{
   ROOT::RDataFrame df("Events", "Run2012BC_DoubleMuParked_Muons.root");
   auto df_2mu = df.Filter(
           [](unsigned int n){ return n == 2; },
           {"nMuon"});
   auto df_os = df_2mu.Filter(
           [](ROOT::RVec<int>& c){ return c[0] != c[1]; },
           {"Muon_charge"});
   auto df_mass = df_os.Define("Dimuon_mass",
           ROOT::VecOps::InvariantMass<float>,
           {"Muon_pt", "Muon_eta", "Muon_phi", "Muon_mass"});
   auto h = df_mass.Histo1D<float>({"", "", 30000, 0.25, 300}, "Dimuon_mass");
   h.GetValue();
}

void df102() {
    main();
}
```

Compiled with **`g++ -O3 df102.cxx $(root-config --cflags --libs)`**
or run interpreted with **`root -l -q df102.cxx`**

C++: Jitted RDF

```
#include "ROOT/RDataFrame.hxx"
#include "ROOT/RVec.hxx"

int main()
{
   ROOT::RDataFrame df("Events", "Run2012BC_DoubleMuParked_Muons.root");
   auto df_2mu = df.Filter("nMuon == 2");
   auto df_os = df_2mu.Filter("Muon_charge[0] != Muon_charge[1]");
   auto df_mass = df_os.Define("Dimuon_mass", "InvariantMass(Muon_pt, Muon_eta, Muon_phi, Muon_mass)");
   auto h = df_mass.Histo1D({"", "", 30000, 0.25, 300}, "Dimuon_mass");
   h.GetValue();
}

void df102() {
    main();
}
```

Compiled with **`g++ -O3 df102.cxx $(root-config --cflags --libs)`**
or run interpreted with **`root -l -q df102.cxx`**

Python

```
import ROOT

df = ROOT.RDataFrame("Events", "Run2012BC_DoubleMuParked_Muons.root")
df_2mu = df.Filter("nMuon == 2")
df_os = df_2mu.Filter("Muon_charge[0] != Muon_charge[1]")
df_mass = df_os.Define("Dimuon_mass", "InvariantMass(Muon_pt, Muon_eta, Muon_phi, Muon_mass)")
h = df_mass.Histo1D(("", "", 30000, 0.25, 300), "Dimuon_mass")
h.GetValue()
```

Run with **`python3 df102.py`**

Runtimes

C++ **compiled** with **templated RDF**: **29s**

C++ **compiled** with **jitted RDF**: **63s**

C++ **interpreted** with **templated RDF**: **190s**

C++ **interpreted** with **jitted RDF**: **66s**

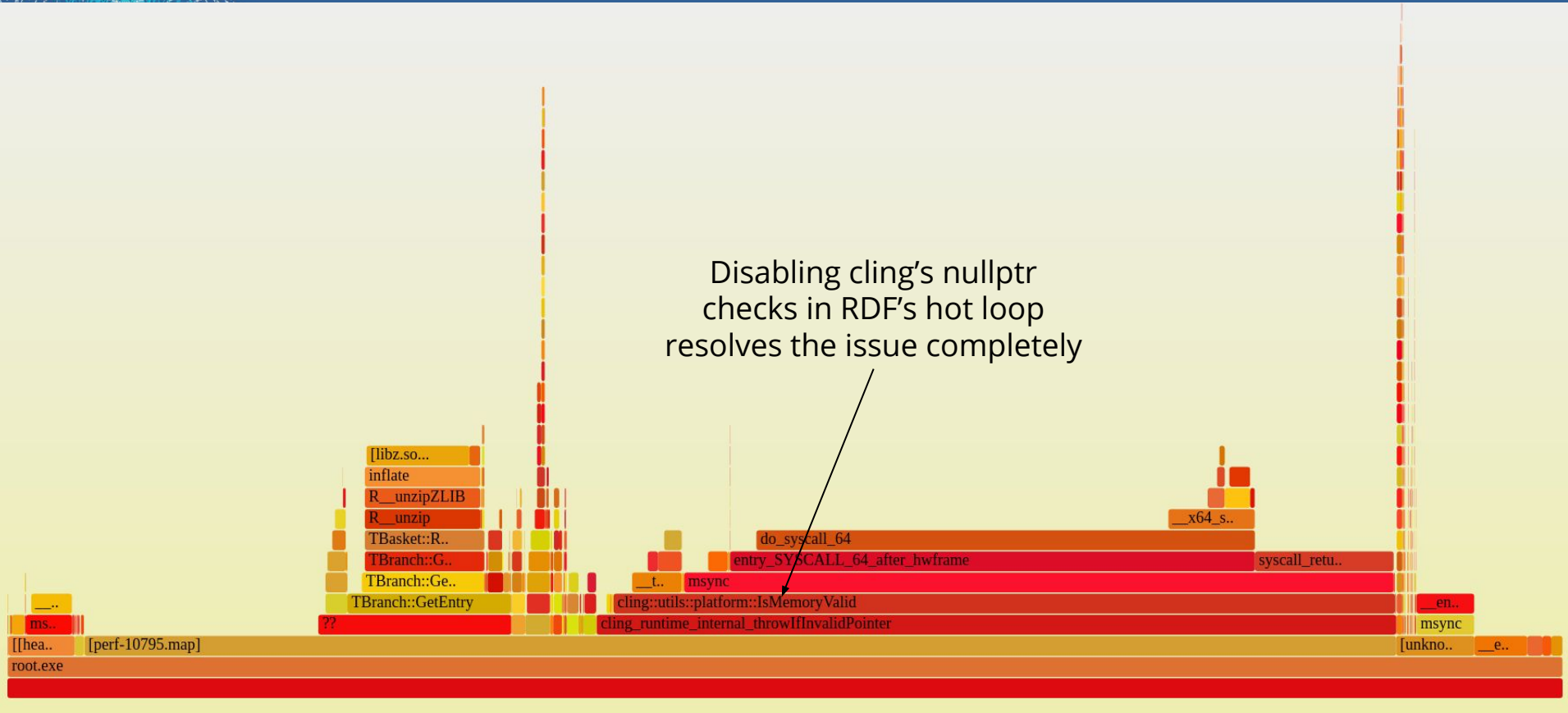
Python: **65s**

Investigated in a following slide

Jitted RDF has always a similar runtime

Python is 2x slower than the fully compiled C++ version

Interpreted C++ with templated RDF



Runtimes after fixing the nullptr checks

C++ **compiled** with **templated RDF** (O3): **29s**

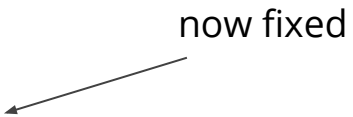
C++ **compiled** with **templated RDF** (O0): **61s**

C++ **compiled** with **jitted RDF**: **63s**

C++ **interpreted** with **templated RDF**: **61s**

C++ **interpreted** with **jitted RDF**: **66s**

Python: **65s**



Impact of compiler optimizations

- Investigate the impact of the compiler optimization level (00, 01, 02 and 03) on the runtime
- Use the version with **compiled** C++ and **templated RDF** as benchmark

- Runtime with 03: **29s**
- Runtime with 02: **30s**
- Runtime with 01: **30s**
- Runtime with 00: **60s**

About the same runtime as the jitted RDF version.
01 already improves the runtime by 50%

Jitted vs compiled RDF: Step by step

```
auto df_2mu = df.Filter("nMuon == 2");
    1) replaced by
auto df_2mu = df.Filter(
    [](unsigned int n){ return n == 2; },{"nMuon"});
```

```
auto df_os = df_2mu.Filter("Muon_charge[0] != Muon_charge[1]");
    2) replaced by
auto df_os = df_2mu.Filter(
    [](ROOT::RVec<int>& c){ return c[0] != c[1]; },
    {"Muon_charge"});
```

```
auto df_mass = df_os.Define("Dimuon_mass",
    "InvariantMass(Muon_pt, Muon_eta, Muon_phi, Muon_mass)");
    3) replaced by
auto df_mass = df_os.Define("Dimuon_mass",
    ROOT::VecOps::InvariantMass<float>,
    {"Muon_pt", "Muon_eta", "Muon_phi", "Muon_mass"});
```

```
auto h = df_mass.Histo1D({"", "", 30000, 0.25, 300},
    "Dimuon_mass");
    4) replaced by
auto h = df_mass.Histo1D<float>({"", "", 30000, 0.25, 300},
"Dimuon_mass");
```

- Each jitted RDF node is replaced one by one until the full graph is compiled
- Always run with compiled C++

- Runtime **compiled C++** with **fully jitted RDF**: **63s**
- Runtime **compiled C++** with **fully templated RDF**: **29s**

1. New runtime: **57s**
2. New runtime: **49s**
3. New runtime: **32s**
4. New runtime: **29s**
   Same as the **fully templated RDF** version

What about `pragma cling optimize`?

```
diff --git a/tree/dataframe/src/RDFUtils.cxx b/tree/dataframe/src/RDFUtils.cxx
index bb12b3db6d..6199dd8b98 100644
--- a/tree/dataframe/src/RDFUtils.cxx
+++ b/tree/dataframe/src/RDFUtils.cxx
@@ -297,9 +297,11 @@ std::vector<std::string> ReplaceDotWithUnderscore(const std::vector<std::string>
    return newColNames;
 }

+static const auto opt = "#pragma cling optimize(3)\n";
+
 void InterpreterDeclare(const std::string &code)
 {
-   if (!gInterpreter->Declare(code.c_str())) {
+   if (!gInterpreter->Declare((opt + code).c_str())) {
       const auto msg =
          "\nRDataFrame: An error occurred during just-in-time compilation. The lines above might indicate the cause of "
          "the crash\n All RDF objects that have not run an event loop yet should be considered in an invalid state.\n";
@@ -310,7 +312,7 @@ void InterpreterDeclare(const std::string &code)
 Long64_t InterpreterCalc(const std::string &code, const std::string &context)
 {
    TInterpreter::EErrorCode errorCode(TInterpreter::kNoError);
-   auto res = gInterpreter->Calc(code.c_str(), &errorCode);
+   auto res = gInterpreter->Calc((opt + code).c_str(), &errorCode);
    if (errorCode != TInterpreter::EErrorCode::kNoError) {
       std::string msg = "\nAn error occurred during just-in-time compilation";
       if (!context.empty())
```

Put a **`#pragma cling optimize(3)`** everywhere in the RDF internal jitting

Runtimes with #pragma cling optimize

C++ **compiled** with **templated RDF**: **29s**

C++ **compiled** with **jitted RDF**: **48s**

C++ **interpreted** with **templated RDF**: **48s**

C++ **interpreted** with **jitted RDF**: **47s**

Python: **48s**

Same runtime also with `clang-5.0.2`

Stays the same, as expected

pragma is added in the script itself, **we could reach 29s with improved jitting optimizations?**

Improvement visible in all cases with jitted RDF code, **runtime decreases by ~25% compared to version without the pragma**

Python could be as fast as compiled C++?

cling + O3: integration benchmarks

	no pragma	pragma
gbenchmark-df102_NanoAODDimuonAnalysis_noimt	47.2	41.9
gbenchmark-df102_NanoAODDimuonAnalysis/8	11.3	11.2
BM_RDataFrame_h1Analysis	4.4	6.1
BM_RDataFrame_h1Analysis_MT/8	1.4	2.8
pytest_df102_NanoAODDimuonAnalysis_noimt	65.6	48.9
pytest_df102_NanoAODDimuonAnalysis_imt	17.9	16.7
df103_NanoAODHiggsAnalysis_noimt	16.0	16.3
df103_NanoAODHiggsAnalysis_imt/8	6.0	6.7
df103_NanoAODHiggsAnalysis_rungraphs/8	4.5	5.3
test_df103_NanoAODHiggsAnalysis_noimt	16.6	16.4
pytest_df103_NanoAODHiggsAnalysis_imt	11.0	14.1
pytest_df103_NanoAODHiggsAnalysis_rungraphs	5.2	6.3
pytest_df104_HiggsToTwoPhotons_noimt	51.8	39.4
pytest_df104_HiggsToTwoPhotons_imt	14.2	11.9
pytest_df104_HiggsToTwoPhotons_rungraphs	15.2	12.4
LoopSUSYFrame	6.5	12.1
wmass_noimt	28.3	31.2
wmass_imt	29.7	30.1

Jitting time goes from 4s to 10s

Take aways

- cling's nullptr checks are expensive - must remember to disable them in our internal templated code!
- Compiler optimization levels improve the runtime significantly (in this example 01 is 50% faster than 00)
- (at least part of) Python's performance loss seems to be due to missing compiler optimizations (03 vs 00)
- `pragma cling optimize(3)` restores the inefficiency only partially (just 25% faster than vanilla jitting with 00). Why?
- speeding up jitted RDF == speeding up RDF in Python ?

Food for thought

- Optimized jitted code improves the hot loop runtimes but adds a constant offset.
  - Jitting optimized code is probably a better default.
    Should this be a knob users can turn?

- Exploratory analyses on just few events might feel very slow.
  RDF+Python benchmarks below ~20s will be skewed by jitting times.
  An issue for initial adoption?

Interpreted C++ with templated RDF

```
$ date && /usr/bin/time root -l -q df102_compiled.cxx

Di 5. Jan 12:14:42 GMT 2021 # from date

Processing df102_compiled.cxx… # entering the macro in ROOT
Enter RLoopManager::Run
Date/Time = Tue, 05 Jan 2021 12:14:44 +0000 (GMT) +480968000 nsec
Start event loop
Date/Time = Tue, 05 Jan 2021 12:14:44 +0000 (GMT) +549038000 nsec
Leave RLoopManager::Run
Date/Time = Tue, 05 Jan 2021 12:17:55 +0000 (GMT) +556587000 nsec

151.86user 40.68system 3:12.73elapsed # from /usr/bin/time
```

Start of the macro

Start of the hot loop

End of the hot loop,
spent 191s in the hot loop

Total runtime of 192s
matches timestamps