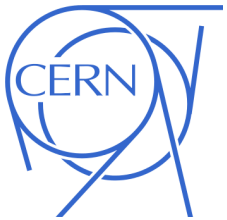
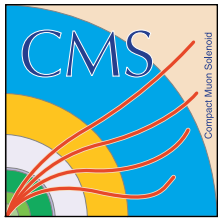
RDF Performance/Functionality Improvements

Josh Bendavid (CERN)





Oct. 28, 2021
ROOT PPP Meeting

Introduction

- Working on precision W measurements in CMS $\rightarrow$ large number of data and Monte Carlo events with little scope for skimming $\rightarrow$ event loop speed is critical for analysis development/fast iteration
- Main analysis workflow (with RDataFrame): CMS NANOAOD $\rightarrow$ histograms + systematic variations for binned maximum likelihood fit
- Several important auxiliary workflows as well for corrections/calibrations
- Using RDataFrame from python is extremely convenient (and allows mixing C++ and numba-jitted python functions)
- Encountered some functionality limitations, and discussed some potential performance improvements with Enrico (related also to some points from last week:

`https://eguiraud.web.cern.ch/eguiraud/decks/20211014_ppp_new_rdf_interfaces/`

RDF Define and Filter from python

- In C++, Define and Filter can work with callable objects (with non-overloaded call operator), (non-overloaded) free functions, and string expressions
- Currently in python, callable objects work, but not free functions
- n.b. calling Define or Filter with string expressions leads to an additional layer of indirection/virtual function calls in the graph via RJittedDefine/Filter

Improving Define/Filter from python

- Implement a "pythonization" (supported feature in pyroot/cppyy to overlay a C++ function with a python function, which would typically do some processing of the arguments and then call the underlying C++ function in order to improve the python interface)
- For Define/Filter this can accomplish several things:
  - Wrap free functions, or class member functions in appropriate callable objects
  - Infer types of columns from the graph and directly call the fully typed version of Define/Filter to avoid jitting from C++ and get directly an appropriate RDefine or RFilter object instead of RJittedDefine or RJittedFilter with extra layer of indirection (Enrico was studying the performance impacts of this in the context of DistRDF: `https://indico.cern.ch/event/1084843/contributions/4561167/attachments/2324353/3958709/PPP%20Improving%20the%20performance%20of%20DistRDF%20tasks.pdf`)
  - Infer column names from function arguments where possible

Improving Define/Filter from python

- Prototype for the Define case implemented in `https://github.com/root-project/root/pull/9174` (some small fixes needed, will be pushed ASAP)
- Aside from pythonization, also improves the C++ interface to allow (optionally) specifying the column types explicitly rather than inferring them from the Callable (allows use of overloads, and implicit conversions, ie float vs double don't have to match exactly in function argument vs column type, etc)
- Targeted interface extensions to TCling and cppyy TemplateProxy to allow needed information to be extracted (for column name inference from function arguments, and to access the bound class instance for a templated member function respectively)
- Can be easily extended to Filter

Improving Define/Filter from python

given the following defined in c++

```
float squared(float x) { return x*x; }

double squared(double x) { return x*x; }

template<typename T>
T squared(T x) { return x*x; }

float squared2(float x) { return x*x; }

double squared2(double y) { return y*y; }

class Callable {
public:
  float operator() (float x) { return x*x; }
  double operator() (double x) { return x*x; }

  template<typename T>
  T operator() (T x) { return x*x; }

  float squared(float x) { return x*x; }

  double squared(double x) { return x*x; }

  template<typename T>
  T squared(T x) { return x*x; }

  static float mul(float x, float y) { return x*y; }

};
```

Improving Define/Filter from python

then in python one can do

```
d = ROOT.ROOT.RDataFrame(chain)

#overload resolved by type of column
d = d.Define("overloadedFreePtsq", ROOT.squared, ["Muon_pt"])

#works as long as there is a column "x" with type implicitly convertible to float or double
d = d.Define("overloadedFreePtsq", ROOT.squared)

#will fail because argument name determination from overloads is ambiguous
d = d.Define("overloadedFreePtsq", ROOT.squared2)

#templated free function
d = d.Define("templatedFreeNmuonsSq", ROOT.squared["int"], ["nMuons"])

#static class member
d = d.Define("staticPtsq", ROOT.Callable.mul, ["Muon_pt", "Muon_pt"])

#class member
d = d.Define("overloadedMemberPtsq", ROOT.Callable().squared, ["Muon_pt"])

#templated class member
d = d.Define("templatedMemberNmuonsSq", ROOT.Callable().squared["int"], ["nMuons"])

#overloaded operator()
d = d.Define("overloadedCallPtsq", ROOT.Callable(), ["Muon_pt"])

#string expression
d = d.Define("lambdaPtsq", "Muon_pt*Muon_pt")

#complete lambda expression (direct jitting without parsing)
d = d.Define("lambdaPtsq", "[](float x) { return x*x; }", ["Muon_pt"])

#complete lambda expression with inferred column names and argument types (direct jitting without parsing, a
d = d.Define("lambdaAutoPtsq", "[](auto Muon_pt) { return Muon_pt*Muon_pt; }")
```

Further Improvements

- Using pythonization to fully specify types for Histo1D, Fill, Book, etc is straightforward (already have this implemented, PR soon)
- Currently even with fully typed versions of Define, Histo1D, Fill, Book, etc in C++, RDefine and RAction objects are used via RDefineBase and RActionBase virtual calls $\rightarrow$ prevents inlining
- "Column readers" which access values from defined columns or from the tree/datasource are also accessed through RColumnReader base virtual calls
- Filters are directly accessed with the fully templated RFilter type
- Can this be improved?

Further Improvements: Fully Inlined graphs

- "Minimal" change: Filter, Define, Histo1D, Fill, Book, etc calls can be extended to optionally accept additional template parameters for the specific column reader types
- Minor extension to RDefineReader to optionally accept template parameter for specific define type
- Since columns in RDF are accessed by name, actually using this interface from C++ is **extremely** inconvenient since complex types have to be specified by hand (avoiding this would require major changes to the user-facing interface in RDF)
- From pythonizations these types can all be inferred at run time and passed as template parameters automatically
- Implementation of the above in progress (and mostly working already)

Further Improvements: Fully Inlined graphs

- Last remaining step: The loop over actions in the final execution of the graph calls the actions through RActionBase (no way to avoid this in C++ for the "lazy" triggering of the graph over all filters)
- But the final loop over the actions can be jitted with specific RAction types, implementation also in progress (this jitting needs to be done from C++ to preserve the "lazy" automatic triggering of graph execution however)
- example of jitting in C++ with type casting of function pointer (better/more efficient way to do this?)

Further Improvements: Fully Inlined graphs

```
const unsigned int jitcounter = RDFInternal::GetJitCounter()++;

std::ostringstream nsname;
nsname << "RInterfaceJitted_" << jitcounter;

std::ostringstream tojit;
tojit << "namespace ROOT {" << std::endl;
tojit << "  namespace " << nsname.str() << " {" << std::endl;
tojit << "    std::unique_ptr<ROOT::Internal::RDF::RActionBase> BuildActionJitted(const std::vector<std::string> &cols, const " << helperArgClassName << "
tojit << "      return ROOT::Internal::RDF::BuildAction<" << colTypeString.str() << ">(cols, helper, nSlots, prevNode, actionTag, defines);" << std::endl;
tojit << "    }" << std::endl;
tojit << "  };" << std::endl;
tojit << "};" << std::endl;

const bool jitstatus = gInterpreter->Declare(tojit.str().c_str());
if (!jitstatus) {
  throw std::runtime_error("Jitting failed!");
}

std::ostringstream funcaddrexpr;;
funcaddrexpr << "&ROOT::" << nsname.str() << "::BuildActionJitted";

using ActionType = std::unique_ptr<ROOT::Internal::RDF::RActionBase>;

using fptype = ActionType(*)(const ColumnNames_t&, const std::shared_ptr<HelperArgType>&, const unsigned int, decltype(fProxiedPtr), ActionTag, const ROOT
fptype fp = reinterpret_cast<fptype>(gInterpreter->Calc(funcaddrexpr.str().c_str()));

auto action = fp(validColumnNames, helperArg, nSlots, fProxiedPtr, ActionTag{}, fDefines);
```

- example of jitting in C++ with type casting of function pointer (better/more efficient way to do this? would need something similar for jitted loop over actions)

Jitting Performance

- All of the above assumes high performance of jitted code at O2 or O3
- With Axel's PR at `https://github.com/root-project/root/pull/7283` to fix inlining, plus a fix to avoid null ptr checks (especially when instantiating templates from cppyy) have gotten jitted code performance essentially equivalent to precompiled code at O3 in test cases with complex template instantiations (boost histograms)
- Once graph inlining functionality is consolidated, will test performance in this configuration (compilation, jitting, runtime comparisons)
- Will definitely be some jitting-time vs runtime tradeoff here, level/mechanism of configurability, and defaults to be discussed

Conclusions

- Advanced implementation of improvements to usability of calls to RDF Define (and soon filter) from python
- WIP on performance improvements towards allowing compilation (or more likely jitting) of a fully templated/inlined graph
- n.b. I haven't forgotten about `https://github.com/root-project/root/pull/7499` to enable ND histograms and more flexible mixed scalar/vector filling of histograms from RDF, will update this to take into account comments and adopt some streamlining enabled by C++14, but this probably shouldn't be merged before jitting performance improvements, because compiling that STL stuff without optimizations would be suboptimal