Improving the performance of DistRDF tasks

Enric Tejedor, Vincenzo Eduardo Padulano, Enrico Guiraud

ROOT

Data Analysis Framework

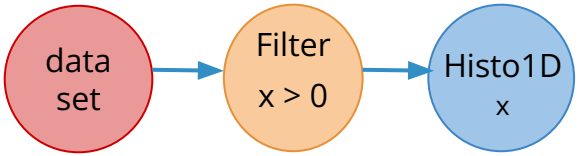
https://root.cern

Context

- DistRDF (and Python RDF) mostly rely on jitted code
  - E.g. `rdf.Filter("x > 0")`
- Jitted code is currently compiled at O0
  - Serious performance penalty for DistRDF!
- Possible solutions
  - Increase the optimization level of jitted code in RDF (PR and commit)
  - Generate a C++ workflow in DistRDF workers (PR)

How do DistRDF workers run tasks?

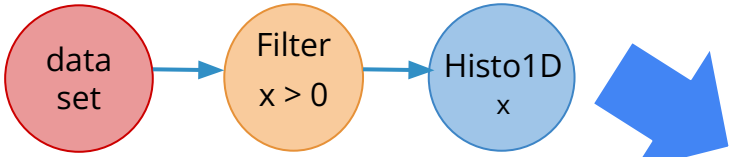
- DistRDF workers receive a Python graph object that represents the RDF computation
  - Nodes correspond to operations and their arguments
- *Default behaviour*: the graph is used to construct an RDF workflow, from Python, operation by operation



```
df = RDataFrame(dataset)
df2 = df.Filter("x > 0")
h = df2.Histo1D("x")
```

C++ workflow generation

- *New behaviour*: use the graph to generate the code of a C++ function that constructs the RDF workflow, then ACLIC it (with optimizations)



rdfworkflow_XYZ_cxx.so

```
Result GenerateWorkflow(RNode &headnode) {
  auto df1 = headnode.Filter("x > 0");
  auto res1 = df1.Histo1D("x");

  ...

}
```

- **Note**: the current implementation still involves jitting of the string arguments that contain C++ expressions!

Questions to ask ourselves

- What's the gain in jitting with optimizations (w.r.t. O0)?
  - And what's the price to pay in terms of jitting times?
- What's the gain in compiling the whole computation graph together (and with no jitting at all)?
  - Tells us if it's worth improving the C++ workflow generation

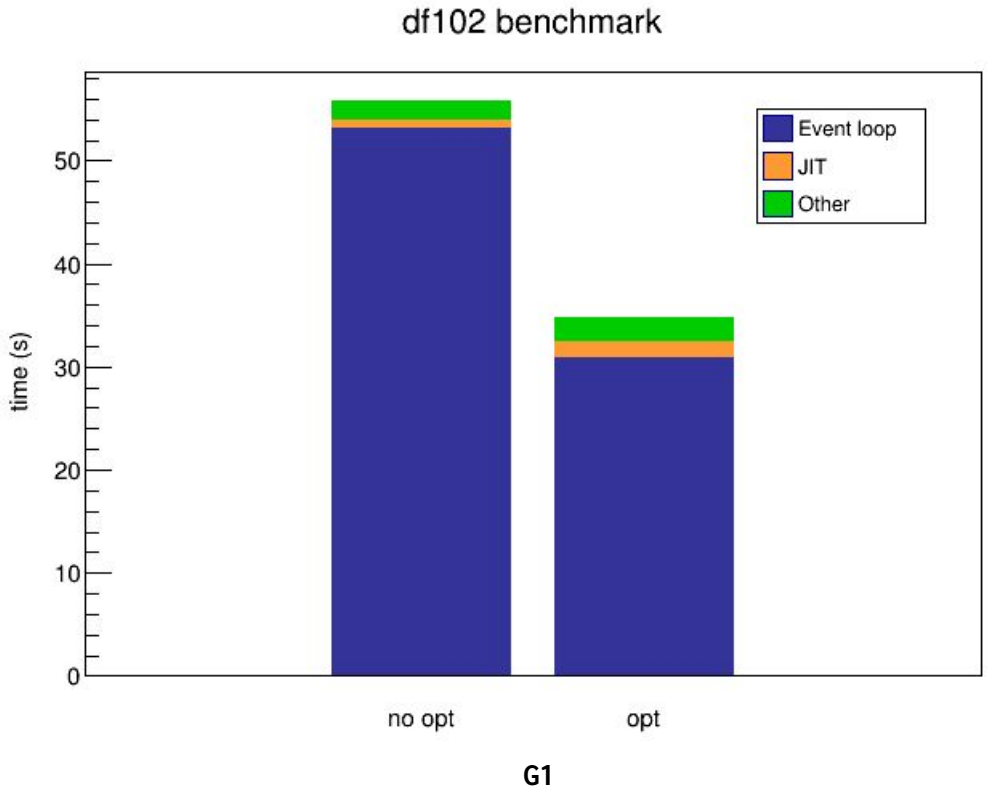
To help us answer... benchmarks!

- We took three of the RDF benchmarks in rootbench and adapted them for DistRDF
  - NanoAOD dimuon (**df102**), NanoAOD Higgs (**df103**), Higgs to two photons (**df104**)
  - Code is available here
- For each benchmark, we have two versions:
  - DistRDF in Python, jitted strings
  - C++ no jitting, lambdas

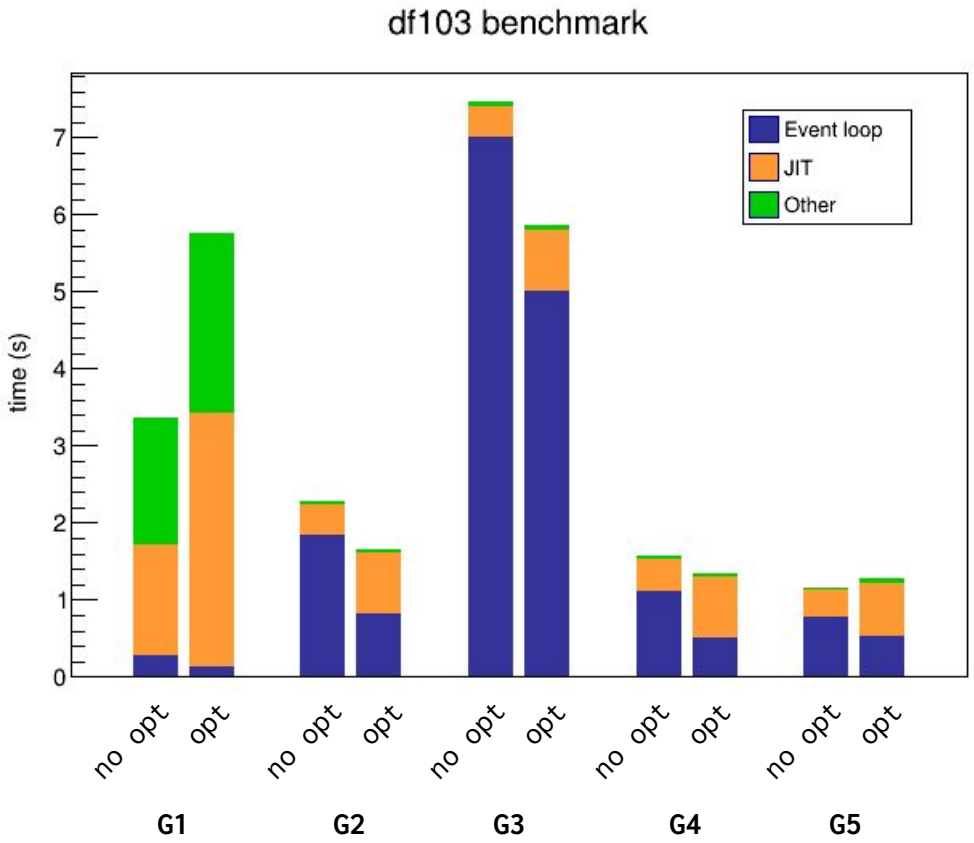
Test #1: No opt vs opt in PyDistRDF

- Performance of Python DistRDF, one partition (i.e. one task), in two jitting modes:
  - **O0** as default (master)
  - **O1** as default (PR) + **O3** activated in RDF jitting (commit)

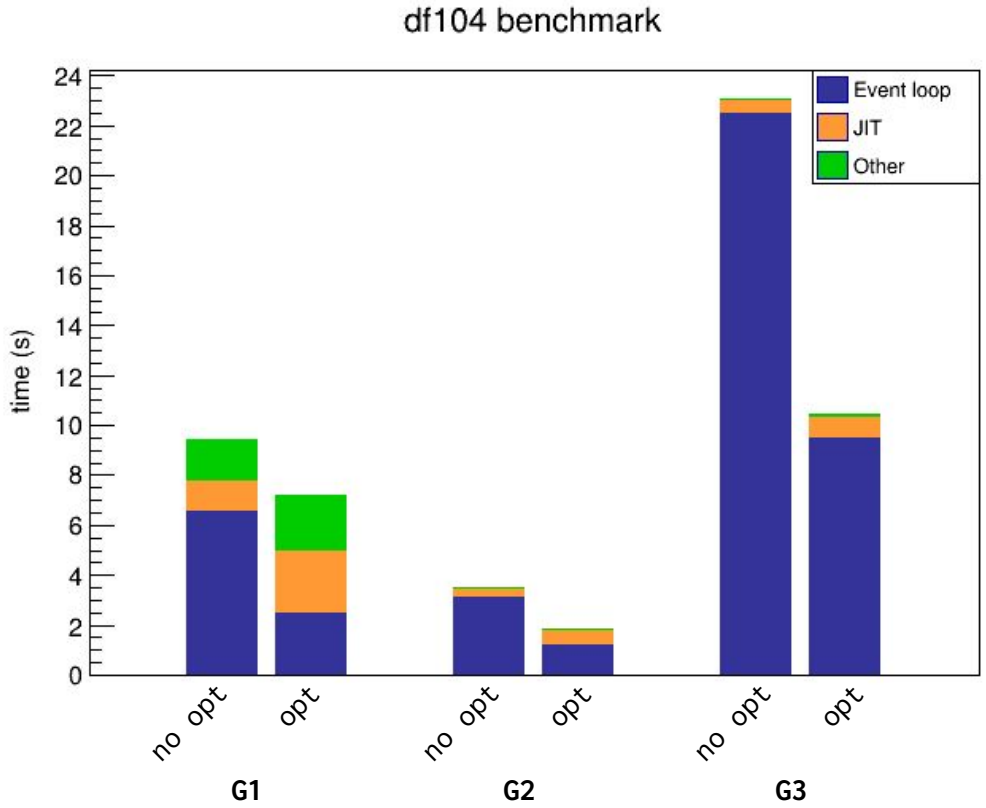
No opt vs opt: df102



No opt vs opt: df103



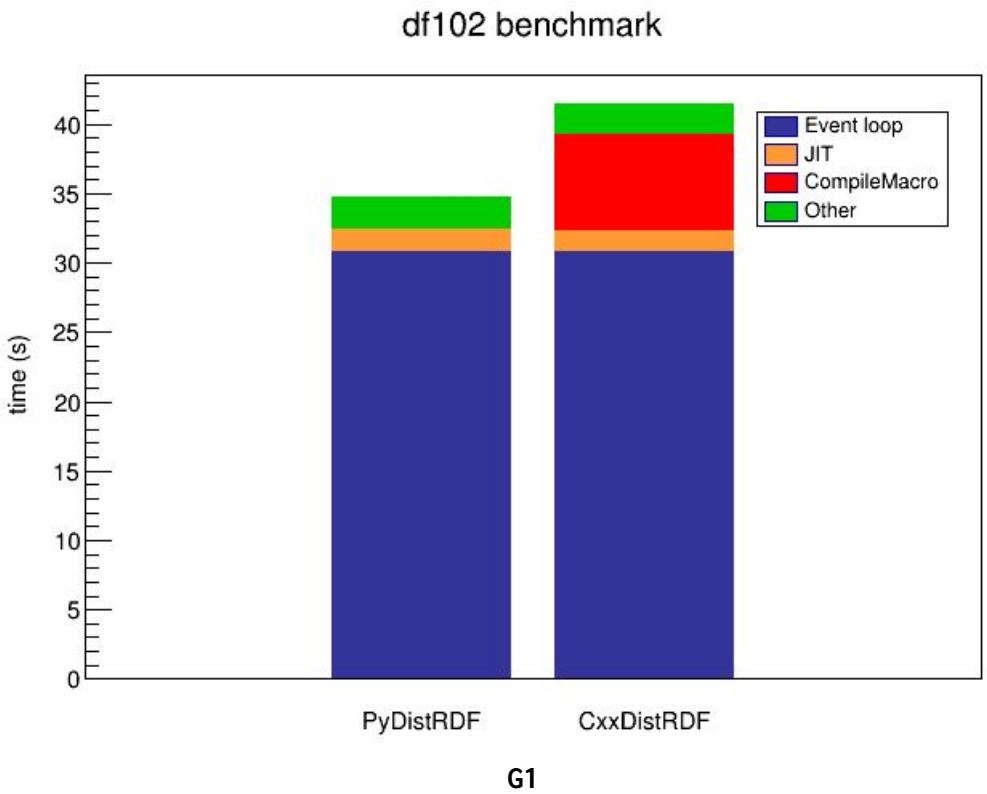
No opt vs opt: df104



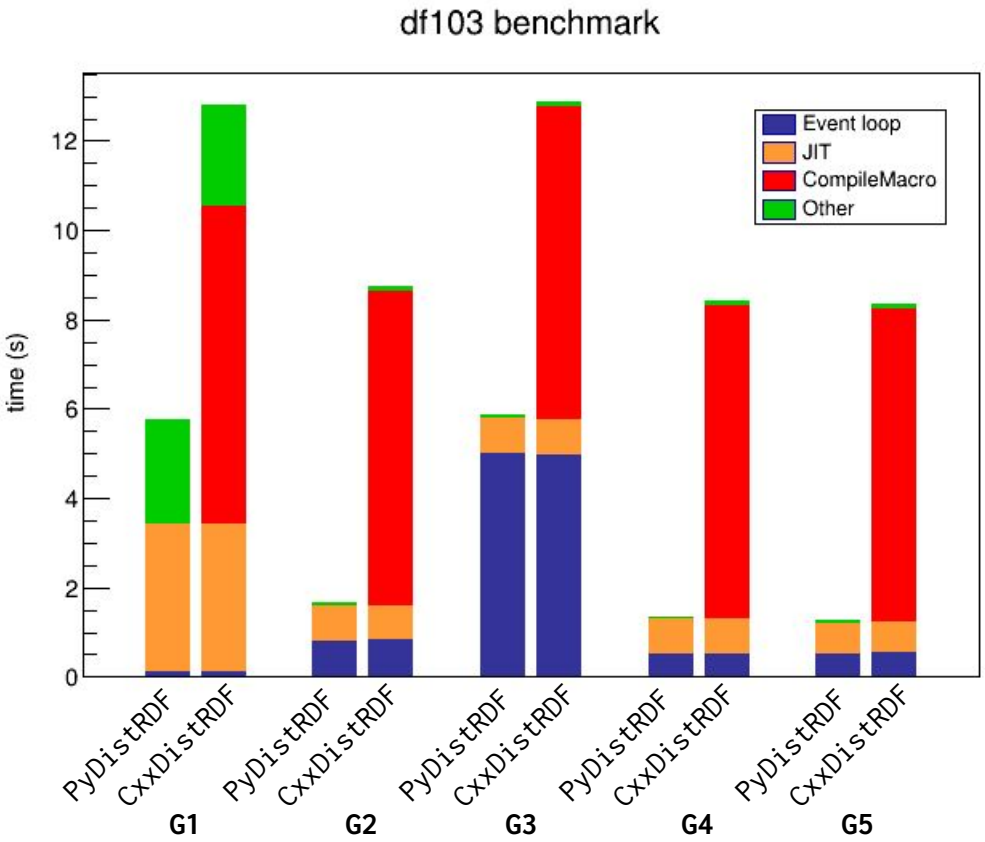
Test #2: PyDistRDF vs CxxDistRDF

- Performance of DistRDF, one partition, **with jitting optimizations**, in two modes:
  - Generation of the graph in **Python** (default)
  - Generation and compilation of **C++** workflow (new)
    - Generated C++ code for each benchmark can be found here
- For the C++ workflow mode, show also a multi-partition (multi-task) run, to see how the CompileMacro cost is paid only once

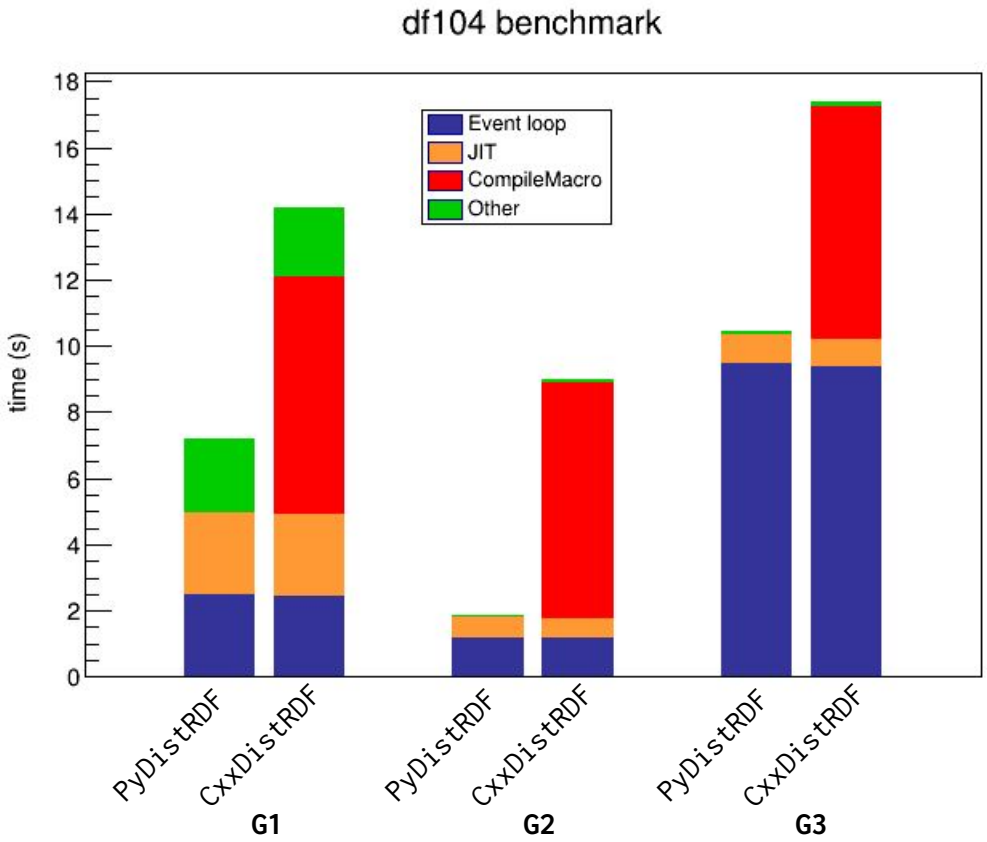
Py DistRDF vs C++ DistRDF: df102



Py DistRDF vs C++ DistRDF: df103

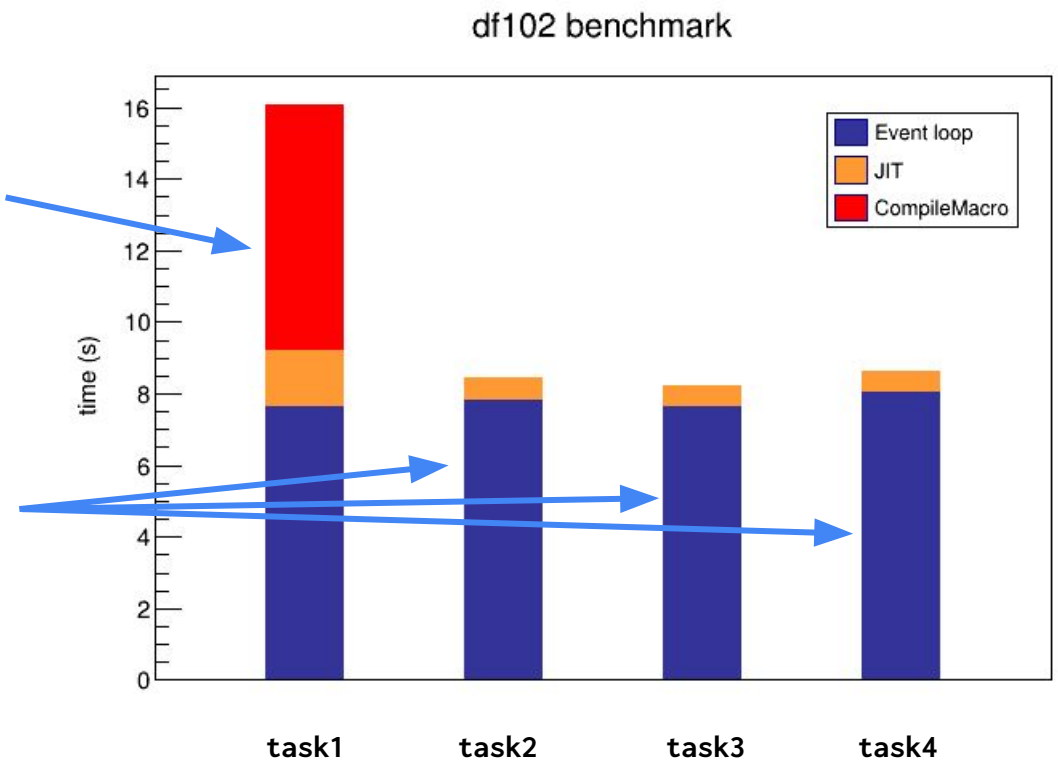


Py DistRDF vs C++ DistRDF: df104



C++ DistRDF: df102 multi-task

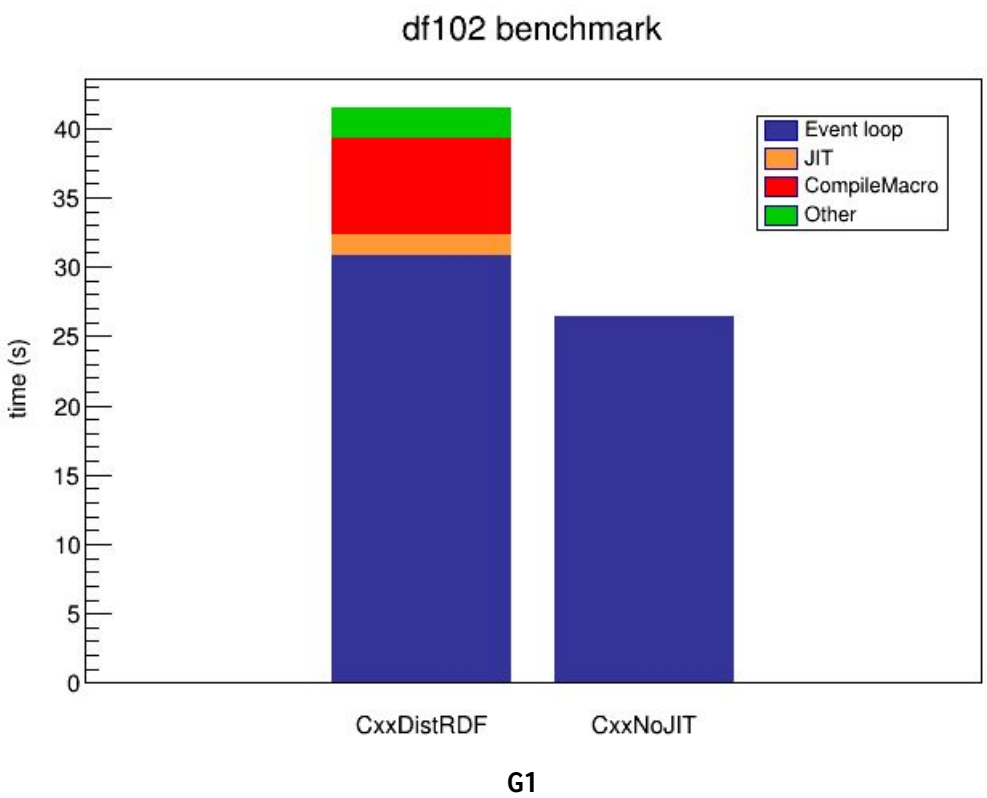
- Only the first mapper task in the worker pays the *CompileMacro* price
- The generated library is **reused** afterwards by tasks on other ranges of the dataset



Test #3: CxxDistRDF vs CxxNoJIT

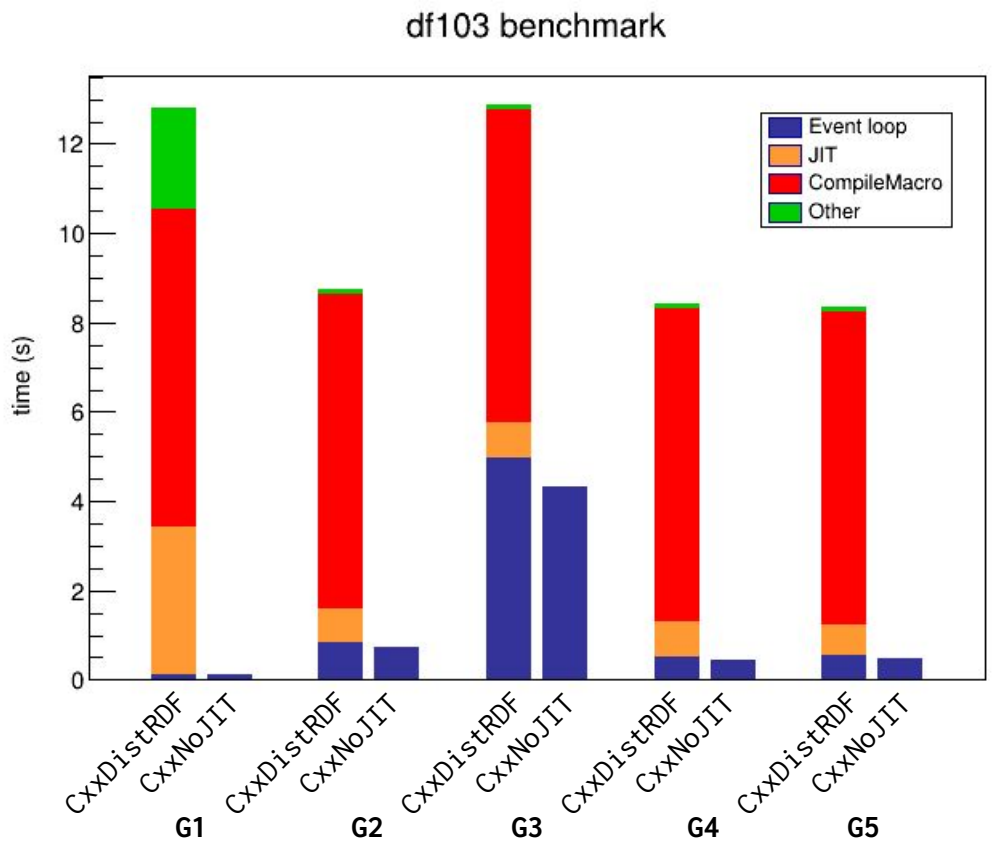
- Performance of two benchmark versions:
  - Python **DistRDF**, one partition, with jitting optimizations, generation and compilation of the graph in **C++**
  - **C++**, **no jitting** (use of lambdas), compiled at **O3**

DistRDF C++ vs C++ No JIT: df102



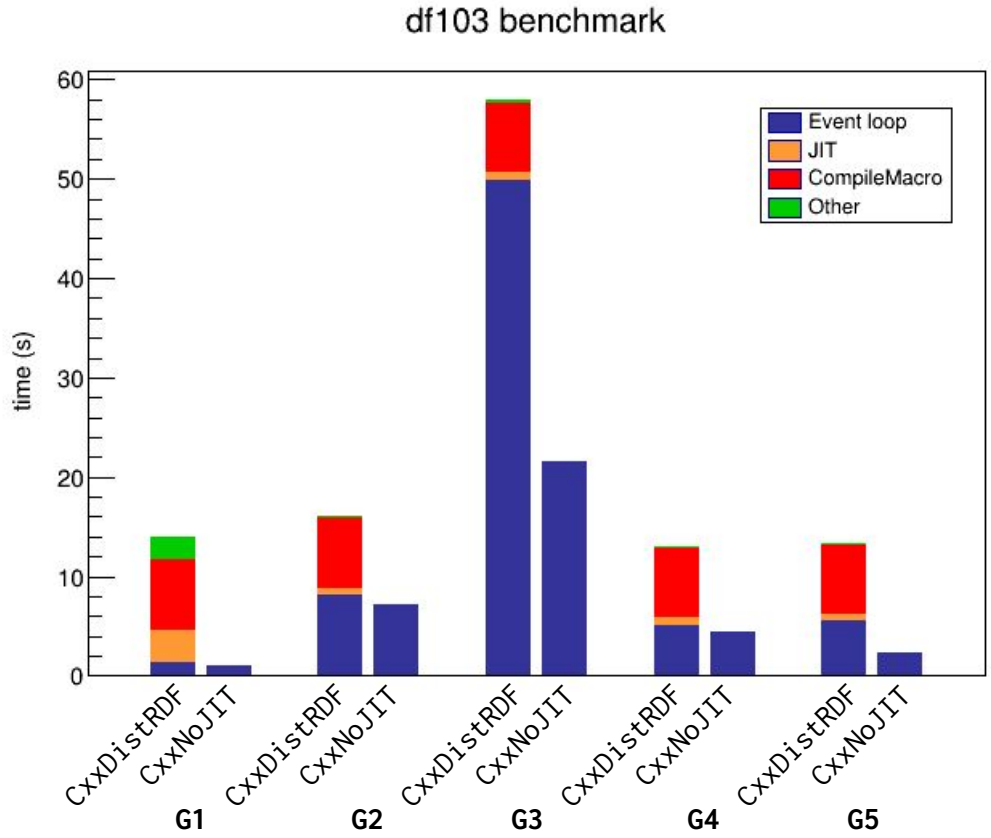
CxxNoJIT
avg compilation time:
3.43 s

DistRDF C++ vs C++ No JIT: df103



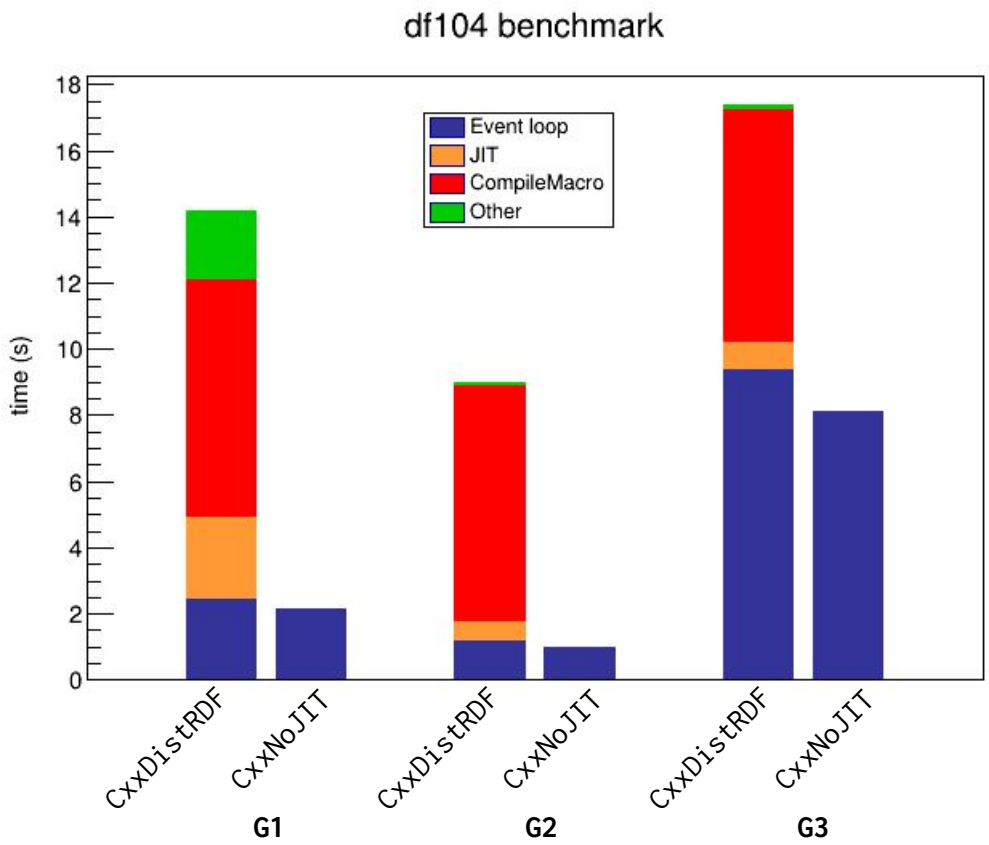
CxxNoJIT
avg compilation time:
7.38 s

DistRDF C++ vs C++ No JIT: df103 10x data



CxxNoJIT
avg compilation time:
7.39 s

DistRDF C++ vs C++ No JIT: df104



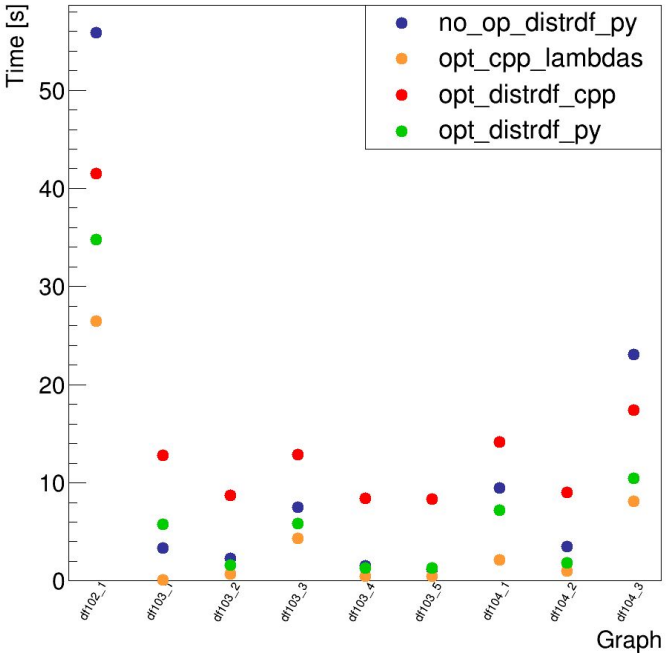
CxxNoJIT
avg compilation time:
5.09 s

Test #4: all in one plot, more data

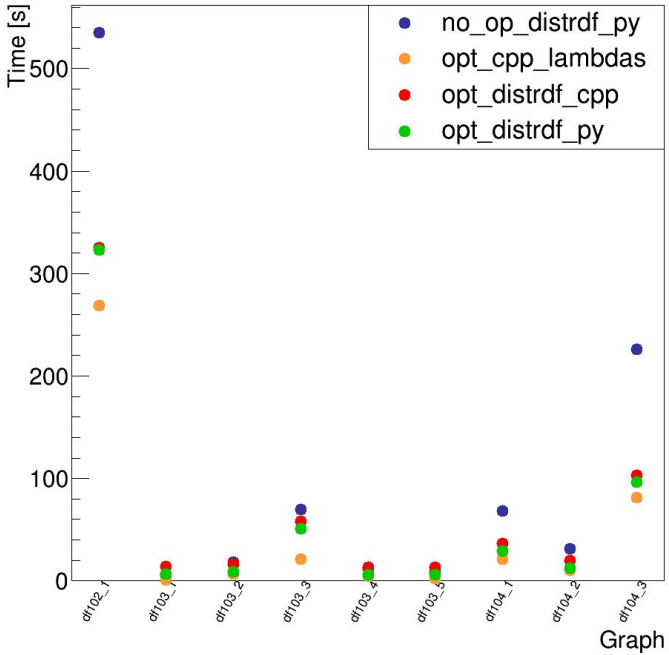
- Using both **1x** and **10x more data**, compare all the configurations seen so far:
  - Python DistRDF with no jitting optimizations
  - Python DistRDF with jitting optimizations
  - C++ DistRDF with jitting optimizations
  - C++ no jitting, compiled at O3
- **Sum all times** (Event loop, JIT, CompileMacro, Other) **for each computation graph** of each benchmark

Time to plot: all benchmarks

Original dataset size

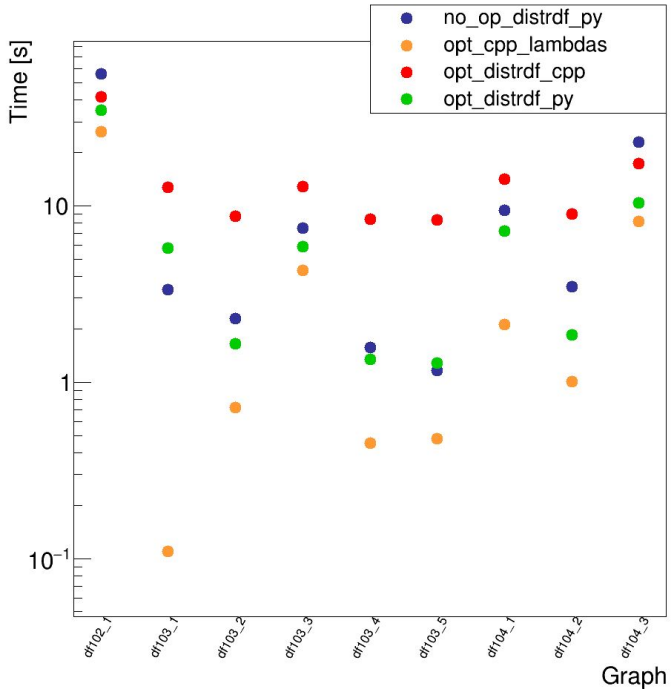


10x dataset size

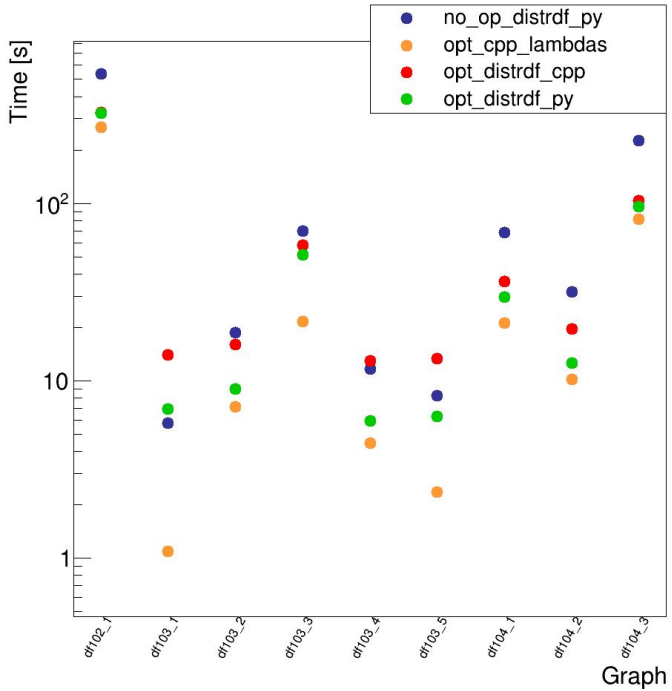


Time to plot: all benchmarks [Log Scale]

Original dataset size



10x dataset size



Conclusions

- Enabling optimizations pays off
- PyDistRDF and CxxDistRDF (in its current form) have the same performance
- The performance of CxxDistRDF could still improve further
  - By compiling all graphs together in a multi-graph application (via DistRDF RunGraphs) -> to reduce CompileMacro cost
  - By generating C++ code that does not jit: can pay off for big datasets
- Still CxxDistRDF does not seem a good default
  - CompileMacro times penalize too much for small datasets

Backup