Fully **Pythonic** RDataFrame Applications

Pawan Johnson, V. E. Padulano, E. Tejedor

ROOT
Data Analysis Framework
https://root.cern
Pythonized ROOT RDF

- RDataFrame in a Nutshell
- Current Interface v/s Fully Pythonic Interface
- Implementation
- Performance Comparison
- Future Direction
RDataFrame in a Nutshell

- High Level Interface for Data Analysis.
- Data from TTrees, TChains, CSV... all accessible from one place.
- A modern interface for operations like Filter and Defines.

Datasource

- ROOT (e.g. NanoAOD)
- CSV
- Apache Arrow
- ATLAS’ xAOD
- Numpy Arrays

Range Filter

Define

<table>
<thead>
<tr>
<th>px</th>
<th>py</th>
<th>pz</th>
<th>eta</th>
<th>p</th>
</tr>
</thead>
</table>

- histograms, profiles
- new ROOT files
- cut-flow reports
- data reductions (mean, sum,..)
- any user-defined operation
ROOT.gInterpreter.Declare(""")

std::array<double, 3> p(double px, double py, double pz) {
    std::array<double, 3> p{px, py, pz};
    return p;
}

bool momentum_cut(std::array<double, 3> p) {
    double p2 = 0.0;
    for (auto&& x : p) {
        p2 += x*x;
    }
    return sqrt(p2) < 10.0;
"")

rdf.Define("p", "p(px,py,pz)").Filter("momentum_cut(p)")
```python
rdf.Define("p", lambda px, py, pz: np.array([px, py, pz]))\n  .Filter(lambda p: np.linalg.norm(p)<10)
```

```
lambda px, py, pz: np.array([px, py, pz])
```
1. Infers the data type from the user input or makes use RDataFrame’s GetColumnType method
2. The function is now JIT-ed using Numba’s compiler
3. The JIT-ed function’s memory address is extracted and recast into a C++ function
4. The C++ Function is declared
5. After the function is declared, we now invoke the operations using the new “C++” Function
Function Jitter WorkFlow

- Infer function input and types
- Generate Numba Compatible signature
- Generate function Call
- Jit function Using Numba Declare
- Invoke Operation using Function Call

RDataFrame
What it looks like in code

**Python Function**
```python
def add(a, b):
    return a + b
```

**New Callable**
```cpp
Numba::add(x, y)
```

**C++ Function**
```cpp
double add(double x_0, double x_1) {
    const auto funcptr = reinterpret_cast<double(*)(double, double)>(140431539101712); // Memory Address
    return funcptr(x_0, x_1);
}
```

Numba’s JIT

Low Level LLVM Function stored in memory

Memory Address Extracted
How it Performs

Key Takeaways
1. JIT-ing occurs only once. The function is now in cache.

2. Time taken increases linearly with number of operations. (0.84s for 1000 Operations)
Variation of Time with No. of Operations

Size of RDF: $2^{20}$
Key Takeaways
1. The Python version is slower than the C++ version.
2. The slowness can be attributed to the function signature being checked against the cache and recreating the function call on each iteration.

Code for Benchmarks: https://github.com/Pawan-Johnson/RDF_Pythonization_Defines_and_Filters_Benchmarks/
Future Directions

- Test performance in Distributed RDataFrames
- Bring up the speed closer to C++ Levels
- Improve error handling for nopython mode failures
- Support for multidimensional RVec
- Bring down to single JIT by interfacing with NumPy directly
Thank you for your attention