

Awkward Just-In-Time (JIT) Compilation:

A Developer's Experience

Ianna Osborne^{1,*}, *Jim Pivarski*¹, *Ioana Ifrim*¹, *Angus Hollands*¹, and *Henry Schreiner*¹

¹Princeton University, Princeton, NJ 08544, USA

Abstract. Awkward Array is a library for performing NumPy-like computations on nested, variable-sized data, enabling array-oriented programming on arbitrary data structures in Python. However, imperative (procedural) solutions can sometimes be easier to write or faster to run. Performant imperative programming requires compilation; JIT-compilation makes it convenient to compile in an interactive Python environment. Various functions in Awkward Arrays JIT-compile a user's code into executable machine code. They use several different techniques, but reuse parts of each others' implementations. We discuss the techniques used to achieve the Awkward Arrays acceleration with JIT-compilation, focusing on RDataFrame, cppy, and Numba, particularly Numba on GPUs: conversions of Awkward Arrays to and from RDataFrame; standalone cppy; passing Awkward Arrays to and from Python functions compiled by Numba; passing Awkward Arrays to Python functions compiled for GPUs by Numba; and header-only libraries for populating Awkward Arrays from C++ without any Python dependencies

1 Introduction

Awkward Array library [1] offers its users an array-oriented programming style in a dynamically typed language. The advantage of the array-oriented calculations is that they fit the data analysis workflow better, considering the large amounts of data which a physicist typically loads into an interactive environment. Moreover, the code that performs one operation on all data is easy enough to write, then observe the intermediate results on distributions, and reiterate. The imperative code, on the other hand, is better for the stage after the interactive exploration, but the performant imperative programming requires compilation. The loops may take longer to write, especially when fitting into a Just-In-Time (JIT)-compiler and getting the compile-time types correctly, but are often self-explanatory and faster. It is not uncommon for a user analysis code to be expressed in a traditional high-performance, statically typed language - C++, and a statically typed language requires compilation.

The JIT-compilation makes it convenient to compile in an interactive Python environment. Several JIT-compilation techniques are used to achieve the desired acceleration. The Awkward Array functions JIT-compile a user's code into executable machine code. They use different techniques, but reuse parts of each others' implementations.

The techniques discussed in this article are focusing on integrating RDataFrame [3], cppy [4], and Numba [5], particularly Numba on GPUs. These include Awkward Arrays to

*e-mail: iosborne@princeton.edu

and from `RDataFrame` conversions, the standalone `cppyy` integration, passing Awkward Arrays to and from Python functions compiled by Numba, passing Awkward Arrays to Python functions compiled for GPUs by Numba, and populating Awkward Arrays from C++ without any Python dependencies using a header-only library.

2 Awkward Arrays and `RDataFrame`

The Awkward Arrays can be converted to the `RDataFrame` columns and the `RDataFrame` columns can be converted to Awkward Arrays as described in detail in [6] and [7].

The user benefits from a faster execution of both the ROOT C++ functions and the user-defined pure C++ functions. Here is an example of such conversion shown beneath.

```
import awkward as ak

df = ak.to_rdataframe({"Events": array})

rdf = (
    df.Filter("Events.nMuon() == 2")
      .Filter("Events.Muon_charge()[0] != Events.Muon_charge()[1]")
      .Define("dimuon_mass", """
// this is C++
return std::sqrt(2 * Events.Muon_pt()[0] * Events.Muon_pt()[1]
    * (std::cosh(Events.Muon_eta()[0] - Events.Muon_eta()[1])
    - std::cos(Events.Muon_phi()[0] - Events.Muon_phi()[1])));
""")
)
```

The `ak.to_rdataframe` function presents a view of an Awkward Array as an `RDataFrame` source. The `ak.from_rdataframe` function converts the selected columns as native Awkward Arrays. The implementation leverages on a zero-copy Awkward Array view and implements all for-loops on data in C++.

The handle to this Array view is a lightweight 40-byte C++ object allocated on the stack. The generated `RDataSource` takes pointers into the original array data via this view. Next, the column readers are generated based on the run-time type of the views. Finally, the readers are passed to a generated source derived from `ROOT::RDF::RDataSource`.

The `ak.from_rdataframe` function converts the selected columns as native Awkward Arrays. The templated C++ header-only implementation and the dynamically generated C++ code are used to extract the columns' types and data.

```
import ROOT

df = ROOT.RDataFrame("Events",
    "root://eospublic.cern.ch//eos/opendata/cms/derived-data/"
    "AOD2NanoAOD0OutreachTool/Run2012BC_DoubleMuParked_Muons.root")

array = ak.from_rdataframe(
    df, columns=("Muon_charge", "Muon_eta", "Muon_mass",
                "Muon_phi", "Muon_pt", "nMuon")
)
```

This array has type

```

type: 61540413 * {
    Muon_charge: var * int32,
    Muon_eta: var * float32,
    Muon_mass: var * float32,
    Muon_phi: var * float32,
    Muon_pt: var * float32,
    nMuon: uint32
}

```

and some of its values look like

```

[{'Muon_charge': [-1, -1], 'Muon_eta': [1.07, -0.564], ...},
 {'Muon_charge': [1, -1], 'Muon_eta': [-0.428, 0.349], ...},
 {'Muon_charge': [1], 'Muon_eta': [2.21], 'Muon_mass': [0.106], ...},
 ...,
 {'Muon_charge': [-1, 1], 'Muon_eta': [-2.15, 0.291], ...}]

```

On the one hand, the users who do their analysis entirely in Python ecosystem can benefit from faster execution using ROOT C++ functions, or pure C++, in an otherwise Awkward analysis at full speed. The performance cost of converting Awkward Arrays into RDataFrame is negligible, since it is a zero-copy view.

On the other hand, those who prefer C++, ROOT, and RDataFrame, have an ability to convert their data into Awkward Arrays in order to leverage the tools available in the wider Scientific Python ecosystem.

3 Standalone cppyy

Like ROOT and RDataFrame, cppyy allows a user to write C++ and JIT-compile it to use it from Python. The C++ code can be included to Python from a C++ file or written directly as a Python string. But cppyy can be installed without the entire ROOT package and Awkward Array's interface to it is more low-level: users can write C++ functions that operate on whole arrays, multiple arrays, and return arrays.

The `ak.Array`, the Python class for all Awkward Arrays, implements a magic function `__cast_cpp__` that is called by cppyy to determine a C++ type of the array. The Numba implementation [8] is reused here to generate a C++ array view on demand. The generated `ArrayView` C++ class hashed type is registered as a `cpp_type` Python string attribute of the `ak.Array` class. The cppyy maps the C++ class type as a string to a Python type. The down side is that the user cannot redefine the function. Each function must have a unique name. This is similar to the PyROOT interpreter, because an earlier version of cppyy is used to bind Python and C++ in PyROOT.

The user does not need to know what `cpp_type` is - the `cpp_type` is generated on demand when the array needs to be passed to the C++ function.

```

array = ak.Array([
    [{"x": 1, "y": [1.1]}, {"x": 2, "y": [2.2, 0.2]}],
    [],
    [{"x": 3, "y": [3.0, 0.3, 3.3]}],
])

```

```

source_code_cpp = """

```

```

template<typename T>
double go_fast_cpp(T& awkward_array) {
    double out = 0.0;
    for (auto list : awkward_array) {
        for (auto record : list) {
            for (auto item : record.y()) {
                out += item;
            }
        }
    }
    return out;
}
"""
cppyy.cppdef(source_code_cpp)

out = cppyy.gbl.go_fast_cpp[array.cpp_type](array)
assert out == ak.sum(array["y"])

```

The cppy version that is used must be 3.1 or later. The cppy library can construct an object of a previously declared type based on an arguments. This new feature is not available in the earlier versions. cppy implements an implicit instantiation from `__cast_cpp__` returning a tuple.

4 Awkward Arrays and Numba

Numba is a JIT-compiler of functions with Python syntax, and Awkward Arrays can be passed to and from Numba-compiled functions with a similar interface as cppy. Numba can be used in contexts in which acceleration is needed, but C++ is not—it allows users to write accelerated code in the same Python language as the rest of their code, albeit in a subset of that language (not all Python code can be compiled).

Numba infers the argument types at call time, and generates optimized code based on this information. Numba also compiles separate specializations depending on the input types.

```

import numba as nb, numpy as np

@nb.jit
def path_length(array):
    result = np.zeros(len(array), dtype=np.float32)
    for i, row in enumerate(array):
        result[i] = 0
        for j, val in enumerate(row):
            result[i] += val
    return result

```

The implementation to pass Awkward Arrays to and from Python functions compiled by Numba defines a `numba_type` property of an `ak.Array` that is a type of a generated C++ Array view, as discussed before. Awkward Arrays can be iterated over in Numba-compiled functions using zero-copy views.

5 Python functions compiled for GPUs by Numba

Numba can also compile code for Nvidia GPUs through LLVM's CUDA backend, and Awkward Arrays can be passed as arguments into such functions. Currently, the extension needs to be defined by a user in a function decorator (as `extensions=[ak.numba.cuda]` below), but this requirement will be removed in a future version of Numba.

```
import cupy as cp

# make an Awkward Array of variable-length lists on the GPU with CuPy
N = 2**20
counts = cp.random.poisson(1.5, N).astype(np.int32)
content = cp.random.normal(0, 45, int(counts.sum())).astype(np.float32)
array = ak.unflatten(content, counts)

@nb.cuda.jit(extensions=[ak.numba.cuda])
def path_length(out, array):
    tid = nb.cuda.grid(1)
    if tid < len(array):
        out[tid] = 0
        for i, x in enumerate(array[tid]):
            out[tid] += x

blocksize = 256
numblocks = (N + blocksize - 1) // blocksize

result = cp.empty(len(array), dtype=np.float32)
path_length[numblocks, blocksize](result, array)
```

6 Header-only libraries

To create Awkward Arrays in C++ without any dependence on Python, we provide a separate set of header-only libraries that implement an `awkward::layoutbuilder` [9] that builds up an array by appending elements and then shares that array through basic C types (raw array buffers and a JSON-formatted string the convey structure).

```
#include "awkward/LayoutBuilder.h"
using awkward::layoutbuilder;

// instantiating layoutbuilder types (type aliases omitted for brevity)
enum Field : std::size_t {one, two};
using UserDefinedMap = std::map<std::size_t, std::string>;
UserDefinedMap fields_map({
    {Field::one, "one"},
    {Field::two, "two"}
});

RecordBuilder<
    RecordField<Field::one, NumpyBuilder<double>>,
    RecordField<Field::two, ListOffsetBuilder<int64_t,
```

```

    NumpyBuilder<int32_t>>>
> builder(fields_map);

// executable code to create and fill the layoutbuilder
auto& one_builder = builder.field<Field::one>();
auto& two_builder = builder.field<Field::two>();

one_builder.append(1.1);
auto& two_subbuilder = two_builder.begin_list();
two_subbuilder.append(1);
two_builder.end_list();

one_builder.append(2.2);
two_builder.begin_list();
two_subbuilder.append(1);
two_subbuilder.append(2);
two_builder.end_list();

```

7 Summary and future plans

The Awkward Array library was designed around array-oriented programming, in part because imperative programming in Python is impractical at large scales. However, this library also provides many ways to access the same data—without copying it—in JIT-compiled contexts, enabling imperative computing at large scales.

While these bridges may be used directly by users, they are also pathways to future features. For example, automatic bindings from Kaitai Struct [10] to Awkward Arrays are in development, and these rely heavily on the header-only `awkward::layoutbuilder`. JIT-compilation in Julia [11] is also in development, borrowing from the patterns set by C++ and Numba integration.

8 Acknowledgment

This work is supported by NSF cooperative agreement OAC-1836650 (IRIS-HEP) and NSF cooperative agreement PHY-2121686 (US-CMS LHC Ops).

References

- [1] Pivarski, J., Osborne, I., Ifrim, I., Schreiner, H., Hollands, A., Biswas, A., Das, P., Roy Choudhury, S., Smith, N., & Goyal, M. (2018). Awkward Array [Computer software]. <https://doi.org/10.5281/zenodo.4341376>
- [2] Harris, C.R., Millman, K.J., van der Walt, S.J. et al. Array programming with NumPy. *Nature* 585, 357–362 (2020). DOI: 10.1038/s41586-020-2649-2
- [3] RDataFrame: Easy Parallel ROOT Analysis at 100 Threads Danilo Piparo, Philippe Canal, Enrico Guiraud, Xavier Valls Pla, Gerardo Ganis, Guilherme Amadio, Axel Naumann, Enric Tejedor EPJ Web Conf. 214 06029 (2019) DOI: 10.1051/epj-conf/201921406029
- [4] cppyy [Online] <https://cppyy.readthedocs.io/en/latest/>

- [5] Siu Kwan Lam, stuartarchibald, Antoine Pitrou, Mark Florisson, Stan Seibert, Graham Markall, esc, Todd A. Anderson, rjenc29, Guilherme Leobas, luk-f-a, Michael Collison, Jay Bourque, Aaron Meurer, Travis E. Oliphant, Nick Riasanovsky, Kaustubh, Michael Wang, densmirn, ... James Bourbeau. (2022). numba/numba: Version 0.56.4 (0.56.4). Zenodo. <https://doi.org/10.5281/zenodo.7289231>
- [6] Osborne, I., Pivarski, J. 2022 Awkward to RDataFrame and back [*arXiv 2302.09860*] DOI: [10.48550/arXiv.2302.09860](https://doi.org/10.48550/arXiv.2302.09860)
- [7] Osborne, I., Pivarski, J., (2022, September 15). Awkward RDataFrame Tutorial. PyHEP 2022 (virtual) Workshop. Zenodo. <https://doi.org/10.5281/zenodo.7081586>
- [8] Jim Pivarski, Peter Elmer and David Lange, (16 November 2020). Awkward Arrays in Python, C++, and Numba, EPJ Web of Conferences 245, 05023 (2020). <https://doi.org/10.1051/epjconf/202024505023>
- [9] Goyal, M., Osborne, I., Pivarski, J. The Awkward World of Python and C++. ACAT 2022 Workshop. <https://doi.org/10.48550/arXiv.2303.02205>
- [10] Kaitai Struct [Online]. <https://kaitai.io>
- [11] Julia [Online] <https://julialang.org>