Refactoring Awkward Array

Jim Pivarski

Princeton University – IRIS-HEP

May 26, 2021

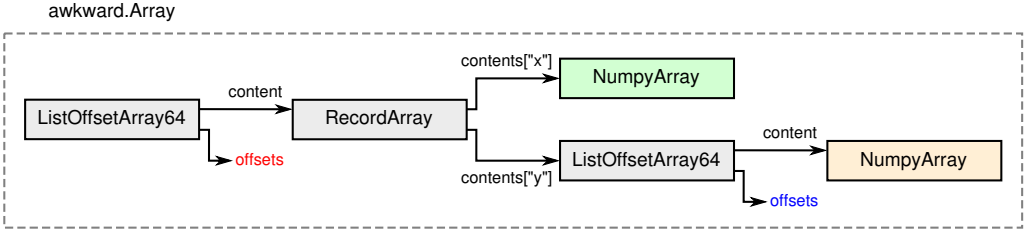
Quick reminder: Awkward Array manipulates trees of 1D buffers

```
array = ak.Array([
    [{"x": 1,  "y": [11]},
     {"x": 4,  "y": [12, 22]},
     {"x": 9,  "y": [13, 23, 33]}],
    [],
    [{"x": 16, "y": [14, 24, 34, 44]}]
])
```

```
outer offsets:  0,                      3,  3, 5
content for x:  1,  4,      9,             16
offsets for y:  0,  1,      3,              6,            10
content for y: 11, 12, 22, 13, 23, 33,     14, 24, 34, 44
```

Quick reminder: Awkward Array manipulates trees of 1D buffers

```
array = ak.Array([
    [{"x": 1,  "y": [11]},
     {"x": 4,  "y": [12, 22]},
     {"x": 9,  "y": [13, 23, 33]}],
    [],
    [{"x": 16, "y": [14, 24, 34, 44]}]
])
```



Quick reminder: Awkward Array manipulates trees of 1D buffers

```
array = ak.Array([
    [{"x": 1,  "y": [11]},
     {"x": 4,  "y": [12, 22]},
     {"x": 9,  "y": [13, 23, 33]}],
    [],
    [{"x": 16, "y": [14, 24, 34, 44]}]
])

>>> array[:, :, "y", 1:]

outer offsets:  0,                      3,  3, 5
content for x:  1,  4,      9,             16
starts  for y:  0,  1,      3,              6
stops   for y:      1,      3,              6,             10
content for y: 11, 12, 22, 13, 23, 33,     14, 24, 34, 44
```

Quick reminder: Awkward Array manipulates trees of 1D buffers

```
array = ak.Array([
    [{"x": 1,  "y": [  ]},
     {"x": 4,  "y": [    22]},
     {"x": 9,  "y": [    23, 33]}],
    [],
    [{"x": 16, "y": [    24, 34, 44]}]
])
```

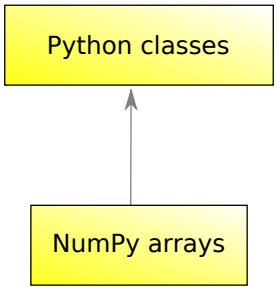
```
>>> array[:, :, "y", 1:]
```

```
outer offsets:  0,                      3,  3, 5
content for x:  1,  4,      9,             16
starts  for y:      1,  2,      4,                7
stops   for y:      1,      3,                6,             10
content for y: 11, 12, 22, 13, 23, 33,    14, 24, 34, 44
```

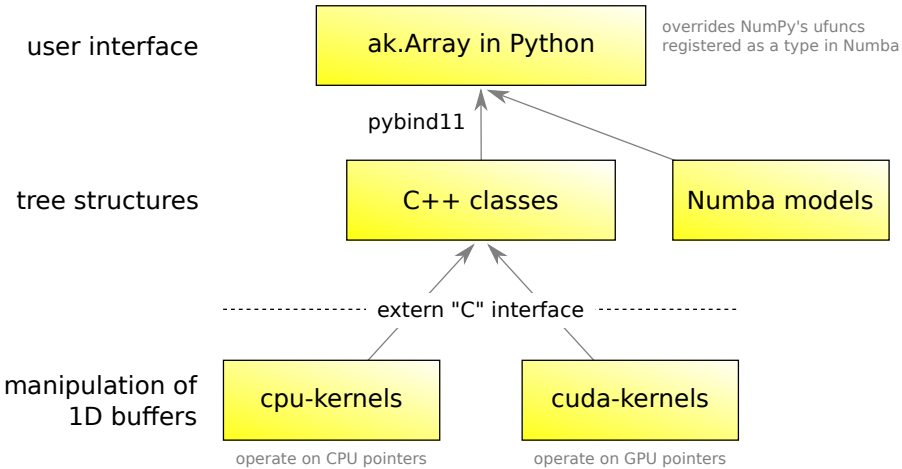
Awkward 0.x did it with NumPy arrays

tree structures,
user interface

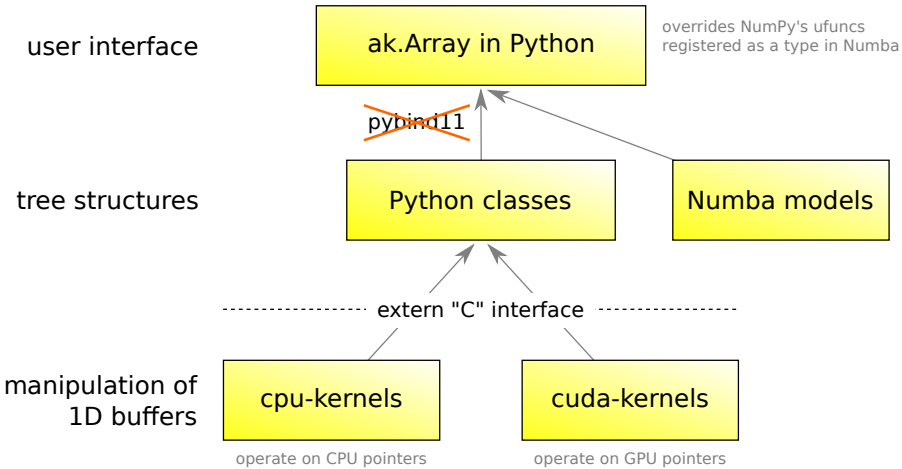
manipulation of
1D buffers



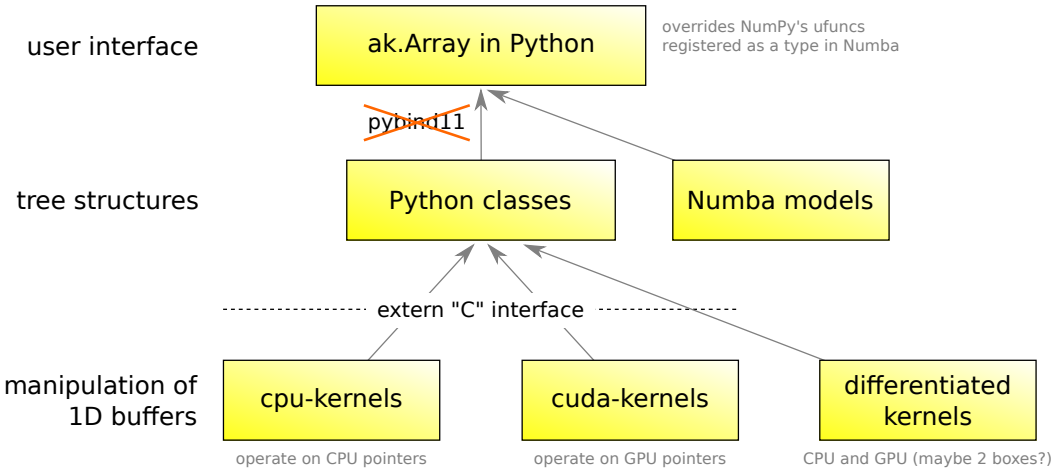
Awkward 1.x has a compiled part



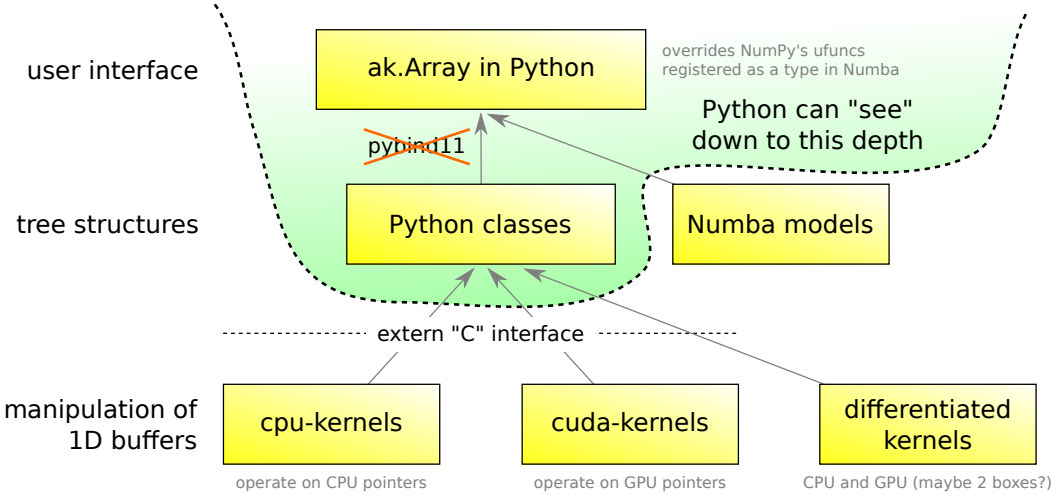
But only the low-level kernels *must be* compiled



Mid-level Python would make autodiff easier to implement



Mid-level Python would make autodiff easier to implement



Why is this coming up now?

Anish put enormous effort into autodifferentiating Awkward Array, but could only handle ufuncs and slices without scalars because of technical issues unrelated to autodifferentiation.

Before the Slice:

```
listoffsetarray.layout
```

```
<ListOffsetArray64>
    <offsets><Index64 i="[0 3 3 5]" offset="0" length="4" at="0x0000029c3770"/></offsets>
    <content><NumpyArray format="d" shape="5" data="1 2 3 4 5" at="0x000002a1bdb0"/></content>
</ListOffsetArray64>
```

After the Slice:

```
listoffsetarray[[2, 2, 0], ::-1].layout
```

```
<ListOffsetArray64>
    <offsets><Index64 i="[0 2 4 7]" offset="0" length="4" at="0x000002aa7420"/></offsets>
    <content><NumpyArray format="d" shape="7" data="5 4 5 4 3 2 1" at="0x000002bb6720"/></content>
</ListOffsetArray64>
```

Before the slice, the linear buffer of `[1, 2, 3, 4, 5]` had a **JAX `Tracer` associated with it**. It now becomes important to **slice this `Tracer`**(since JAX passes these `Tracers` objects, throughout the function computation and it needs to know how our function maps **`A -> B`**) in such a way that it **starts representing our Array after the slice**. Since, slices can be complex in Awkward Arrays and most of them don't map onto linear buffers, we make use of **`Identities`**.

9

Why is this coming up now?

He got around it (for slices with no scalars) by enabling `ak::Identities`, finding out which output corresponds to which input and slicing the JAX Tracers accordingly. This probably isn't generalizable to all Awkward operations.

```
ak.to_list(listoffsetarray)
```

```
[[1.0, 2.0, 3.0], [], [4.0, 5.0]]
```

```
np.asarray(listoffsetarray.layout.content.identities)
```

```
array([[0, 0],
       [0, 1],
       [0, 2],
       [2, 0],
       [2, 1]], dtype=int64)
```

```
ak.to_list(listoffsetarray[[2, 2, 0], ::-1])
```

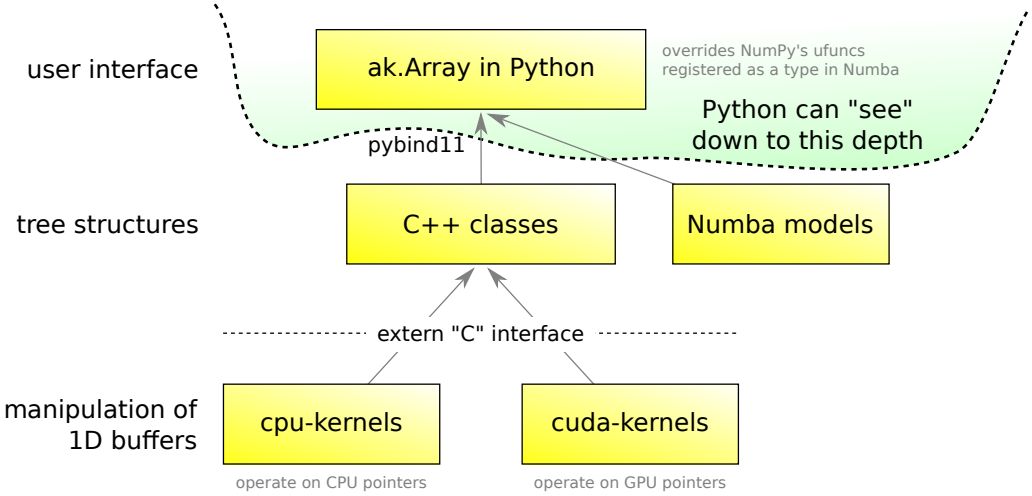
```
[[5.0, 4.0], [5.0, 4.0], [3.0, 2.0, 1.0]]
```

```
np.asarray(listoffsetarray[[2, 2, 0], ::-1].layout.content.identities)
```

```
array([[2, 1],
       [2, 0],
       [2, 1],
       [2, 0],
       [0, 2],
       [0, 1],
       [0, 0]], dtype=int64)
```

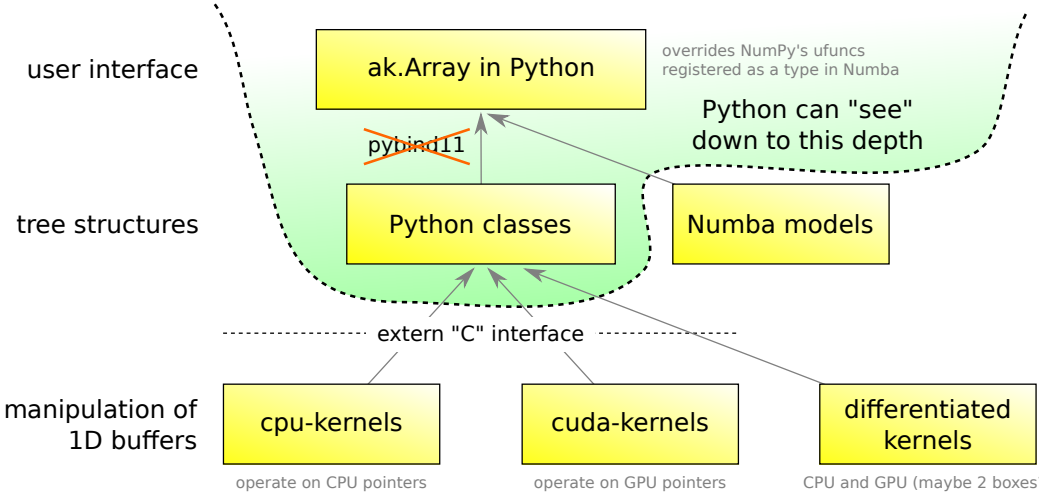
The fundamental problem

Ultimately, the problem is that JAX can't "see" what happens to 1D arrays. It sees tree structures whose nodes get replaced and 1D buffers get shuffled in complex ways.



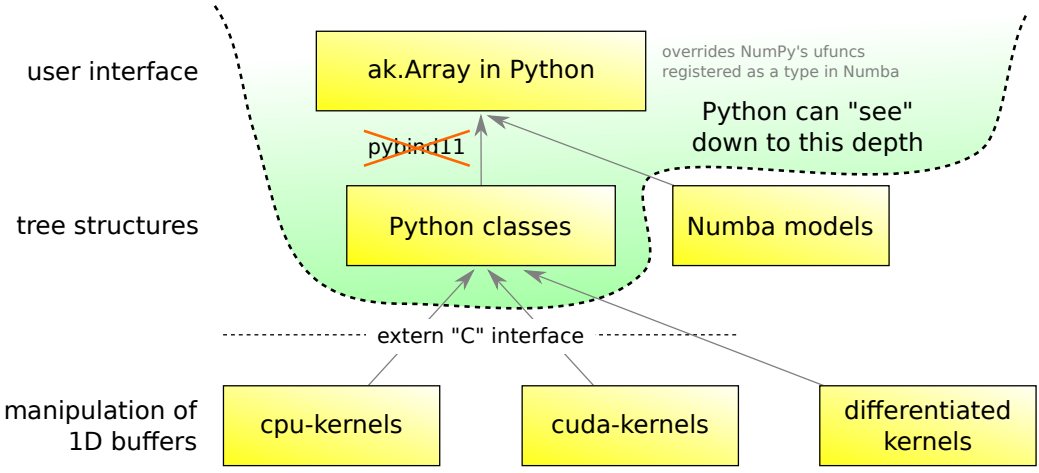
The fundamental problem

But it could be alleviated if JAX could "see" down to the level of kernel manipulation. Then it would only need to be told how each kernel shuffles 1D buffers.



The fundamental problem

Then, Awkward Array would look like an application on arrays (like pyhf) with some extra array-functions we need to tell JAX about (the kernels).



It's not just JAX; JAX is what finally convinced me

1. JAX/autodiff: patched (incompletely) with `ak::Identities`.
2. Garbage collector reference cycles: `gc.collect()` can't see cycles that go Python → C++ → Python. Patched with `weakref` (brittle).
3. Future GPU optimization: to avoid putting unnecessary synchronization points between kernels, we need to give CUDA a DAG of kernels (rather than calling them eagerly). A general way to make such a DAG is with Dask, but only if Dask can see all the way down to the kernel level.
4. Python profilers and debugger: currently, the C++ is opaque. Patched by (manually!) attaching line number references to every exception, but still we don't see the full stack trace.
5. **Also, the C++ layer does not serve the function it was created for.**

Why does Awkward Array have a C++ layer?

It is not for performance.

(The *kernels* are compiled for performance; the C++ layer never performs any operations whose time complexity is above $\mathcal{O}(1)$ in array length n.)

Why does Awkward Array have a C++ layer?

It is not for performance.

(The *kernels* are compiled for performance; the C++ layer never performs any operations whose time complexity is above $\mathcal{O}(1)$ in array length n.)

It's for interoperability with C++ libraries.

Quintessential example: bindings to FastJet (C++)

The idea was that we would accept many-event sets of particles as jagged record arrays and return many-even sets of jets as jagged record arrays:

```
ak::Index64 offsets(num_events + 1);
ak::ContentPtr content = std::make_shared<ak::NumpyArray>(...);

// fill offsets and content with jet results

return ak::ListOffsetArray64(offsets, content);
```

Quintessential example: bindings to FastJet (C++)

The idea was that we would accept many-event sets of particles as jagged record arrays and return many-even sets of jets as jagged record arrays:

```
ak::Index64 offsets(num_events + 1);
ak::ContentPtr content = std::make_shared<ak::NumpyArray>(...);

// fill offsets and content with jet results

return ak::ListOffsetArray64(offsets, content);
```

But compiling and linking other projects to Awkward Array is brittle: #778, #774, #483, #562, #476, #430, #316, #281, #233, #217, #211, #209.

Quintessential example: bindings to FastJet (C++)

The idea was that we would accept many-event sets of particles as jagged record arrays and return many-even sets of jets as jagged record arrays:

```
ak::Index64 offsets(num_events + 1);
ak::ContentPtr content = std::make_shared<ak::NumpyArray>(...);

// fill offsets and content with jet results

return ak::ListOffsetArray64(offsets, content);
```

But compiling and linking other projects to Awkward Array is brittle: #778, #774, #483, #562, #476, #430, #316, #281, #233, #217, #211, #209.

Now that we're actually writing Awkward-FastJet, we're doing it through Python.

The `ak.from_buffers` function can be a standard interface

Any library that can produce a Form (JSON), length (int), and buffers (string names → 1D buffers map) has effectively produced an Awkward Array.

ak.from_buffers

Defined in awkward.operations.convert on line 4741.

ak.from_buffers(*form*, *length*, *container*, *partition_start=0*, *key_format='part{partition}-{form_key}-{attribute}'*, *lazy=False*, *lazy_cache='new'*, *lazy_cache_key=None*, *highlevel=True*, *behavior=None*)

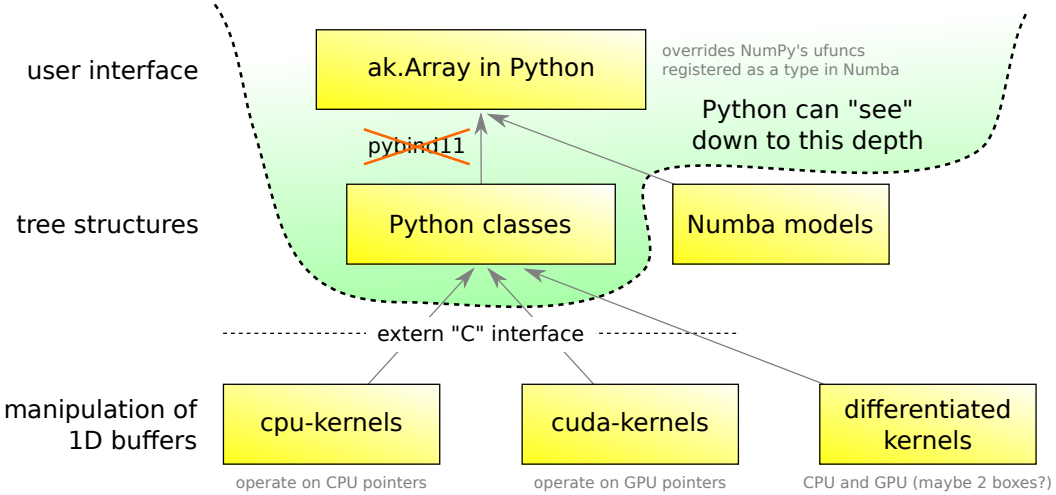
Parameters

form (`ak.forms.Form` or str/dict equivalent) – The form of the Awkward Array to reconstitute from named buffers.

length (*int or iterable of int*) – Length of the array to reconstitute as a non-partitioned array or the lengths (plural) of partitions in a partitioned array.

container (*Mapping, such as dict*) – The str → Python buffers that represent the decomposed Awkward Array. This `container` is only assumed to have a `__getitem__` method that accepts strings as keys.

So this is the plan: zero-interface change refactoring



Ioana Ifrim will be taking it on as a 25% project, ∼6 months

Ioana Ifrim

Research Staff
email: ioana.ifrim@cern.ch

- Research Fellowship Generative Deep Learning - CERN (2018-2020)
- Research student part of Morphological Computation and Learning Lab, Imperial College; Biologically Inspired Robotics Laboratory University of Cambridge (2016-2018)
- MPhil Advanced Computer Science, University of Cambridge (2018)
- BSc Computer Science with Robotics, King's College London (2017)

Details

- Not all of the mid-level C++ will be translated. ArrayBuilder, TypedArrayBuilder, and AwkwardForth are not based on kernels and must remain in C++.
- Therefore, the build procedure will not change (other than the plans Henry has for it already). There will still be a `libawkward.so`, but it will be smaller.
- 60k lines of C++ will be translated to (maybe) 30k lines of Python.
- The unit tests that will be needed to compare old C++ and new Python will significantly increase coverage.
- When we finally remove the old C++, that will be called Awkward 2.0.
- However, the *interface* will not change!