Awkward Array: JSON-like data, NumPy-like idioms

Jim Pivarski[‡][*], Ianna Osborne[‡], Pratyush Das[¶], Anish Biswas[§], Peter Elmer[‡]

https://youtu.be/WlnUF3LRBj4

Abstract—NumPy simplifies and accelerates mathematical calculations in Python, but only for rectilinear arrays of numbers. Awkward Array provides a similar interface for JSON-like data: slicing, masking, broadcasting, and performing vectorized math on the attributes of objects, unequal-length nested lists (i.e. ragged/jagged arrays), and heterogeneous data types.

Awkward Arrays are columnar data structures, like (and convertible to/from) Apache Arrow, with a focus on manipulation, rather than serialization/transport. These arrays can be passed between C++ and Python, and they can be used in functions that are JIT-compiled by Numba.

Development of a GPU backend is in progress, which would allow data analyses written in array-programming style to run on GPUs without modification.

Index Terms—NumPy, Numba, Pandas, C++, Apache Arrow, Columnar data, AOS-to-SOA, Ragged array, Jagged array, JSON

Introduction

NumPy [np] is a powerful tool for data processing, at the center of a large ecosystem of scientific software. Its built-in functions are general enough for many scientific domains, particularly those that analyze time series, images, or voxel grids. However, it is difficult to apply NumPy to tasks that require data structures beyond N-dimensional arrays of numbers.

More general data structures can be expressed as JSON and processed in pure Python, but at the expense of performance and often conciseness. NumPy is faster and more memory efficient than pure Python because its routines are precompiled and its arrays of numbers are packed in a regular way in contiguous memory. Some expressions are more concise in NumPy's "vectorized" notation, which describe actions to perform on whole arrays, rather than scalar values.

In this paper, we describe Awkward Array [ak1], [ak2], a generalization of NumPy's core functions to the nested records, variable-length lists, missing values, and heterogeneity of JSON-like data. The internal representation of these data structures is columnar, very similar to (and compatible with) Apache Arrow [arrow]. But unlike Arrow, the focus of Awkward Array is to provide a suite of data manipulation routines, just as NumPy's role is focused on transforming arrays, rather than standardizing a serialization format.

* *Corresponding author: pivarski@princeton.edu*
‡ *Princeton University*
¶ *Institute of Engineering and Management*
§ *Manipal Institute of Technology*

Our goal in developing Awkward Array is not to replace NumPy, but to extend the set of problems to which it can be applied. We use NumPy's extension mechanisms to generalize its interface in a way that returns identical output where the applicability of the two libraries overlap (i.e. rectilinear arrays), and the implementation of non-structure-changing, numerical math is deferred to NumPy itself. Thus, all the universal functions (ufuncs) in the SciPy project [scipy] and its ecosystem can already be applied to Awkward structures because they inherit NumPy and SciPy's own implementations.

Origin and development

Awkward Array was intended as a way to enable particle physics analyses to take advantage of scientific Python tools. Particle physics problems are inherently structured, frequently needing nested loops over variable-length lists. They also involve big data, typically tens to hundreds of terabytes per analysis. Traditionally, this required physicists to do data analysis in Fortran (with custom libraries for data structures [hydra] before Fortran 90) and C++, but many physicists are now moving to Python for end-stage analysis [phypy]. Awkward Array provides the link between scalable, interactive, NumPy-based tools and the nested, variable-length data structures that physicists need.

Since its release in September 2018, Awkward Array has become one of the most popular Python libraries for particle physics, as shown in Figure 1. The Awkward 0.x branch was written using NumPy only, which limited its development because every operation must be vectorized for performance. We (the developers) also made some mistakes in interface design and learned from the physicists' feedback.

Spurred by these shortcomings and the popularity of the general concept, we redesigned the library as Awkward 1.x in a half-year project starting in August 2019. The new library is compiled as an extension module to allow us to write custom precompiled loops, and its Python interface is improved: it is now a strict generalization of NumPy, is compatible with Pandas [pandas] (Awkward Arrays can be DataFrame columns), and is implemented as a Numba [numba] extension (Awkward Arrays can be used in Numba's just-in-time compiled functions).

Although the Awkward 1.x branch is feature-complete, serialization to and from a popular physics file format (ROOT [root], which represents over an exabyte of physics data [root-EB]) is not. Adoption among physicists is ongoing, but the usefulness of JSON-like structures in data analysis is not domain-specific and should be made known to the broader community.

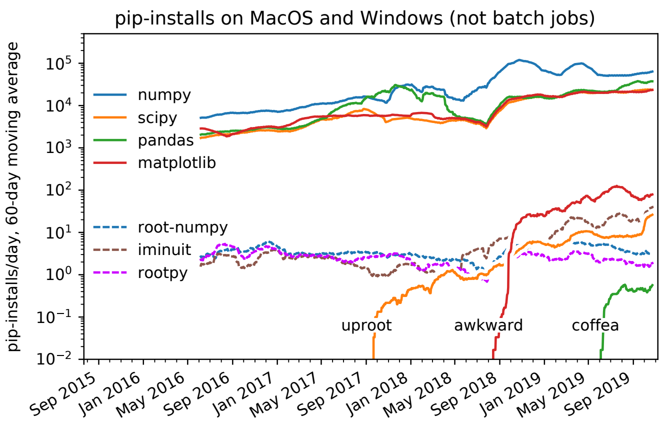

Fig. 1: *Adoption of Awkward 0.x, measured by PyPI statistics, compared to other popular particle physics packages (root-numpy, iminuit, rootpy) and popular data science packages.*

Demonstration using a GeoJSON dataset

To show how Awkward Arrays can be applied beyond particle physics, this section presents a short exploratory analysis of Chicago bike routes [bikes] in GeoJSON format. GeoJSON has a complex structure with multiple levels of nested records and variable-length arrays of numbers, as well as strings and missing data. These structures could not be represented as a NumPy array (without `dtype=object`, which are Python objects wrapped in an array), but there are reasons to want to perform NumPy-like math on the numerical longitude, latitude coordinates.

To begin, we load the publicly available GeoJSON file,

```
import urllib.request
import json

url = "https://raw.githubusercontent.com/Chicago/" \
      "osd-bike-routes/master/data/Bikeroutes.geojson"
bikeroutes_json = urllib.request.urlopen(url).read()
bikeroutes_pyobj = json.loads(bikeroutes_json)
```

and convert it to an Awkward Array. The two main data types are `ak.Array` (a sequence of items, which may contain records) and `ak.Record` (a single object with named, typed fields, which may contain arrays). Since the dataset is a single JSON object, we pass it to the `ak.Record` constructor.

```
import awkward1 as ak
bikeroutes = ak.Record(bikeroutes_pyobj)
```

The record-oriented structure of the JSON object, in which fields of the same object are serialized next to each other, has now been transformed into a columnar structure, in which data from a single field across all objects are contiguous in memory. This requires more than one buffer in memory, as heterogeneous data must be split into separate buffers by type.

The structure of this particular file (expressed as a Datashape, obtained by calling `ak.type(bikeroutes)`) is

```
{"type": string,
 "crs": {
     "type": string,
     "properties": {"name": string}},
 "features": var * {
     "type": string,
     "properties": {
         "STREET": string,
         "TYPE": string,
         "BIKEROUTE": string,
         "F_STREET": string,
         "T_STREET": option[string]},
     "geometry": {
         "type": string,
         "coordinates":
             var * var * float64}}}
```

We are interested in the longitude, latitude coordinates, which are in the `"coordinates"` field of the `"geometry"` of the `"features"`, at the end of several levels of variable-length lists (`var`). At the deepest level, longitude values are in coordinate `0` and latitude values are in coordinate `1`.

We can access each of these, eliminating all other fields, with a NumPy-like multidimensional slice. Strings in the slice select fields of records and ellipsis (`...`) skips dimensions as it does in NumPy.

```
longitude = bikeroutes["features", "geometry",
                       "coordinates", ..., 0]
latitude  = bikeroutes["features", "geometry",
                       "coordinates", ..., 1]
```

The `longitude` and `latitude` arrays both have type `1061 * var * var * float64`; that is, 1061 routes with a variable number of variable-length polylines.

At this point, we might want to compute the length of each route, and we can use NumPy ufuncs to do that, despite the irregular shape of the `longitude` and `latitude` arrays. First, we subtract off the mean and convert degrees into a unit of distance (`82.7` and `111.1` are conversion factors at Chicago's latitude).

```
km_east = (longitude - np.mean(longitude)) * 82.7
km_north = (latitude - np.mean(latitude)) * 111.1
```

Subtraction and multiplication defer to `np.subtract` and `np.multiply`, respectively, and these are ufuncs, overridden using NumPy's `__array_ufunc__` protocol [nep13]. The `np.mean` function is not a ufunc, but it, too, can be overridden using the `__array_function__` protocol [nep18]. All ufuncs and a handful of more generic functions can be applied to Awkward Arrays.

To compute distances between points in an array `a` in NumPy, we would use an expression like the following,

```
differences = a[1:] - a[:-1]
```

which views the same array without the first element (`a[1:]`) and without the last element (`a[:-1]`) to subtract "between the fenceposts." We can do so in the nested lists with

```
differences = km_east[:, :, 1:] - km_east[:, :, :-1]
```

even though the first two dimensions have variable lengths. They're derived from the same array (`km_east`), so they have the same lengths and every element in the first term can be paired with an element in the second term.

Two-dimensional distances are the square root of the sum of squares of these differences,

```
segment_length = np.sqrt(
    (km_east[:, :, 1:] - km_east[:, :, :-1])**2 +
    (km_north[:, :, 1:] - km_north[:, :, :-1])**2)
```

and we can sum up the lengths of each segment in each polyline in each route by calling `np.sum` on the deepest `axis`.

```
polyline_length = np.sum(segment_length, axis=-1)
route_length = np.sum(polyline_length, axis=-1)
```

The same could be performed with the following pure Python code, though the vectorized form is shorter, more exploratory, and 8× faster (Intel 2.6 GHz i7-9750H processor with 12 MB cache on a single thread); see Figure 2.

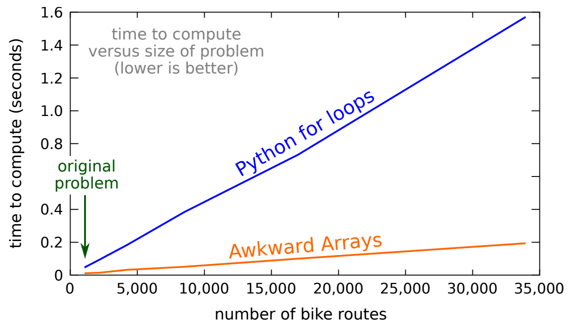

Fig. 2: *Scaling of Awkward Arrays and pure Python loops for the bike routes calculation shown in the text.*

```
route_length = []
for route in bikeroutes_pyobj["features"]:
    polyline_length = []
    for polyline in route["geometry"]["coordinates"]:
        segment_length = []
        last = None
        for lng, lat in polyline:
            km_east = lng * 82.7
            km_north = lat * 111.1
            if last is not None:
                dx2 = (km_east - last[0])**2
                dy2 = (km_north - last[1])**2
                segment_length.append(
                    np.sqrt(dx2 + dy2))
            last = (km_east, km_north)

        polyline_length.append(sum(segment_length))
    route_length.append(sum(polyline_length))
```

The performance advantage is due to Awkward Array's precompiled loops, though this is mitigated by the creation of intermediate arrays and many passes over the same data (once per user-visible operation). When the single-pass Python code is just-in-time compiled by Numba *and* evaluated over Awkward Arrays, the runtime is 250× faster than pure Python (same architecture).

Scope: data types and common operations

Awkward Array supports the same suite of abstract data types and features as "typed JSON" serialization formats—Arrow, Parquet, Protobuf, Thrift, Avro, etc. Namely, there are

- primitive types: numbers and booleans,
- variable-length lists,
- regular-length lists as a distinct type (i.e. tensors),
- records/structs/objects (named, typed fields),
- fixed-width tuples (unnamed, typed fields),
- missing/nullable data,
- mixed, yet specified, types (i.e. union/sum types),
- virtual arrays (functions generate arrays on demand),
- partitioned arrays (for off-core and parallel analysis).

Like Arrow and Parquet, arrays with these features are laid out as columns in memory (more on that below).

Like NumPy, the Awkward Array library consists of a primary Python class, `ak.Array`, and a collection of generic operations. Most of these operations change the structure of the data in the array, since NumPy, SciPy, and others already provide numerical math as ufuncs.

Awkward functions include

- basic and advanced slices (`__getitem__`) including variable-length and missing data as advanced slices,
- masking, an alternative to slices that maintains length but introduces missing values instead of dropping elements,
- broadcasting of universal functions into structures,
- reducers of and across variable-length lists,
- zip/unzip/projecting free arrays into and out of records,
- flattening and padding to make rectilinear data,
- Cartesian products (cross join) and combinations (self join) at `axis >= 1` (per element of one or more arrays).

Conversions to other formats, such as Arrow, and interoperability with common Python libraries, such as Pandas and Numba, are also in the library's scope.

Columnar representation, columnar implementation

Awkward Arrays are columnar, not record-oriented, data structures. Instead of concentrating all data for one array element in nearby memory (as an "array of structs"), all data for a given field are contiguous, and all data for another field are elsewhere contiguous (as a "struct of arrays"). This favors a pattern of data access in which only a few fields are needed at a time, such as the longitude, latitude coordinates in the bike routes example.

Additionally, Awkward operations are performed on columnar data without returning to the record-oriented format. To illustrate, consider an array of variable-length lists, such as the following toy example:

```
[[1.1, 2.2, 3.3], [4.4], [5.5, 6.6], [7.7, 8.8, 9.9]]
```

Instead of creating four C++ objects to represent the four lists, we can put all of the numerical data in one buffer and indicate where the lists start and stop with two integer arrays:

```
starts:  0, 3, 4, 6
stops:   3, 4, 6, 9
content: 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9
```

For an array of lists of lists, we could introduce two levels of `starts` and `stops` arrays, one to specify where the outer square brackets start and stop, another to specify the inner square brackets. Any tree-like data structure can be built in this way; the hierarchy of nested array groups mirrors the hierarchy of the nested data, except that the number of these nodes scales with the complexity of the data type, not the number of elements in the array. Particle physics use-cases require thousands of nodes to describe complex collision events, but billions of events in memory at a time. Figure 3 shows a small example.

In the bike routes example, we computed distances using slices like `km_east[:, :, 1:]`, which dropped the first element from each list. In the implementation, list objects are not created for the sake of removing one element before translating back into a columnar format; the operation is performed directly on the columnar data.

For instance, to drop the first element from each list in an array of lists `a`, we only need to add `1` to the `starts`:

```
starts:  1, 4, 5, 7
stops:   3, 4, 6, 9
content: 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9
```

Without modifying the `content`, this new array represents

```
[[     2.2, 3.3], [   ], [     6.6], [     8.8, 9.9]]
```

because the first list starts at index `1` and stops at `3`, the second starts at `4` and ends at `4`, etc. The "removed" elements are still present in the `content` array, but they are now unreachable, much like the stride tricks used for slicing in NumPy.

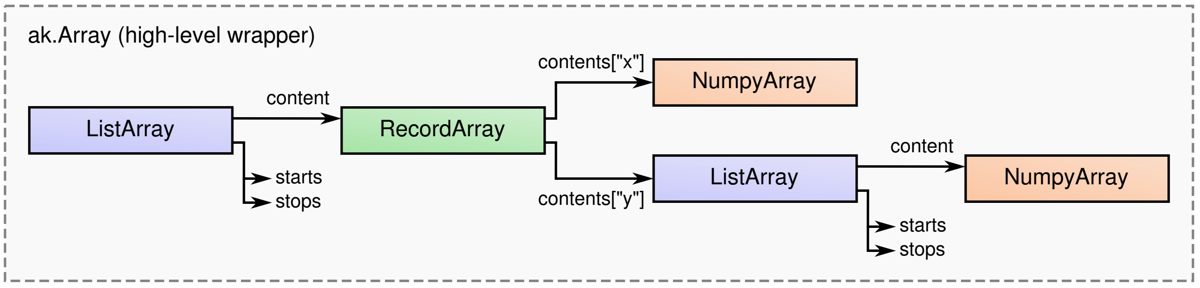

Fig. 3: Hierarchy for an example data structure: an array of lists of records, in which field `"x"` *of the records are numbers and field* `"y"` *of the records are lists of numbers. This might, for example, represent* `[[], [{"x": 1, "y": [1]}, {"x": 2, "y": [2, 2]}]]`*, but it also might represent an array with billions of elements (of the same type). The number of nodes scales with complexity, not data volume.*

Leaving the `content` untouched means that the precompiled slice operation does not depend on the `content` type, not even whether the `content` is a numeric array or a tree structure, as in Figure 3. It also means that this operation does not cascade down such a tree structure, if it exists. Most operations leave nested structure untouched and return views, rather than copies, of most of the input buffers.

Architecture of Awkward 1.x

In August 2019, we began a half-year project to rewrite the library in C++ (Awkward 1.x), which is now complete. Whereas Awkward 0.x consists of Python classes that call NumPy on internal arrays to produce effects like the slice operation described in the previous section, Awkward 1.x consists of C++ classes that perform loops in custom compiled code, wrapped in a Python interface through pybind11.

However, the distinction between slow, bookkeeping code and fast math enforced by Python and NumPy is a useful one: we maintained that distinction by building Awkward 1.x in layers that separate the (relatively slow) polymorphic C++ classes, whose job is to organize and track the ownership of data buffers, from the optimized loops in C that manipulate data in those buffers.

These layers are fully broken down below and in Figure 4:

- The high-level interface is in Python.
- The array nodes (managing node hierarchy and ownership/lifetime) are in C++, accessed through pybind11.
- An alternate implementation of array navigation was written for Python functions that are compiled by Numba.
- Array manipulation algorithms (without memory management) are independently implemented as "cpu-kernels" and "cuda-kernels" plugins. The kernels' interface is pure C, allowing for reuse in other languages.

The separation of "kernels" from "navigation" has two advantages: (1) optimization efforts can focus on the kernels, since these are the only loops that scale with data volume, and (2) CPU-based kernels can, in principle, be swapped for GPU-based kernels. The latter is an ongoing project.

Numba for just-in-time compilation

Some expressions are simpler in "vectorized" form, such as the Awkward Array solution to the bike routes calculation. Others are simpler to express as imperative code. This issue arose repeatedly as physicists used Awkward Array 0.x in real problems, both

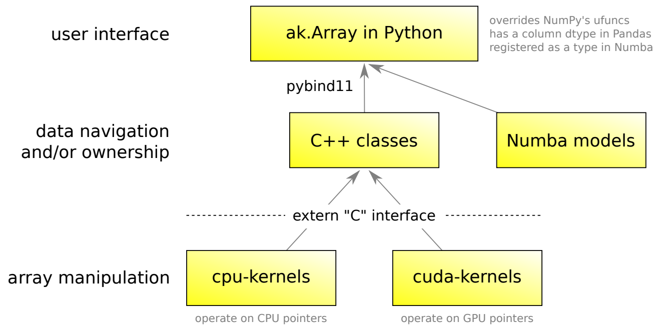

Fig. 4: Components of Awkward Array, as described in the text. All components have been implemented except for the "cuda-kernels."

because they were more familiar with imperative code (in C++) and because the problems truly favored non-vectorized solutions. For instance, walking up a tree, looking for nodes of a particular type (such as a tree of particle decays) is hard to express in vectorized form because some elements of a test array reach the stopping condition before others; preventing them from continuing to walk the tree adds complexity to a data analysis. Any problem that must "iterate until converged" is also of this form.

These problems are readily solved by Numba, a just-in-time compiler for Python, but Numba cannot compile code involving arrays from Awkward 0.x. To solve physics problems, we had to break the array abstraction described above. Ensuring that Numba would recognize Awkward 1.x arrays was therefore a high priority, and it is a major component of the final system.

Numba has an extension mechanism for registering new types and overloading operators for new types. We added Numba extensions for the `ak.Array` and `ak.Record` types, overloading `__getitem__` (square bracket) and `__getattr__` (dot) operators and iterators, so that users can walk over the data structures with conventional loops.

Returning to the bike routes example, the following performs the same calculation with Numba:

```
import numba as nb

@nb.jit
def compute_lengths(bikeroutes):
    # allocate output array
    route_length = np.zeros(len(bikeroutes["features"]))

    # loop over routes
    for i in range(len(bikeroutes["features"])):
        route = bikeroutes["features"][i]
        # loop over polylines
        for polyline in route["geometry"]["coordinates"]:
            first = True
            last_east = 0.0
            last_north = 0.0

            for lng_lat in polyline:
                km_east = lng_lat[0] * 82.7
                km_north = lng_lat[1] * 111.1

                # compute distances between points
                if not first:
                    dx2 = (km_east - last_east)**2
                    dy2 = (km_north - last_north)**2
                    distance = np.sqrt(dx2 + dy2)
                    route_length[i] += distance

                # keep track of previous value
                first = False
                last_east = km_east
                last_north = km_north

    return route_length
```

This expression is not concise, but it is 250× faster than the pure Python solution and 30× faster than even the Awkward Array (precompiled) solution. It makes a single pass over all buffers, maximizing CPU cache efficiency, and it does not allocate or fill any intermediate arrays. This is possible because `nb.jit` compiles specialized machine code for this particular problem.

Combining Awkward Array with Numba has benefits that neither has alone. Ordinarily, complex data structures would have to be passed into Numba as Python objects, which means a second copy of the data that must be "unboxed" (converted into a compiler-friendly form) and "boxed" (converted back). If the datasets are large, this consumes memory and time. Awkward Arrays use less memory than the equivalent Python objects (5.2× smaller for the bike routes) and they use the same internal representation (columnar arrays) inside and outside functions just-in-time compiled by Numba.

The disadvantage of Numba and Awkward Arrays in Numba is that neither support the whole language: Numba can only compile a subset of Python and the NumPy library and Awkward Arrays are limited to imperative-style access (no array-at-a-time functions) and homogeneous data (no union type). Any code that works in a just-in-time compiled function works without compilation, but not vice-versa. Thus, there is a user cost to preparing a function for compilation, which can be seen in a comparison of the code listing above with the pure Python example in the original bike routes section. However, this finagling is considerably less time-consuming than translating a Python function to a language like C or C++ and converting the data structures. It favors gradual transition of an analysis from no just-in-time compilation to a judicious use of it in the parts of the workflow where performance is critical.

ArrayBuilder: creating columnar data in-place

Awkward Arrays are immutable; NumPy's ability to assign elements in place is not supported or generalized by the Awkward Array library. (As an exception, users can assign fields to records using `__setitem__` syntax, but this *replaces* the inner tree with one having the new field.) Restricting Awkward Arrays to read-only access allows whole subtrees of nodes to be shared among different versions of an array.

To create new arrays, we introduced `ak.ArrayBuilder`, an append-only structure that accumulates data and creates `ak.Arrays` by taking a "snapshot" of the current state. The `ak.ArrayBuilder` is also implemented for Numba, so just-in-time compiled Python can build arbitrary data structures.

The `ak.ArrayBuilder` is a dynamically typed object, inferring its type from the types and order of data appended to it. As elements are added, the `ak.ArrayBuilder` builds a tree of columns *and* their types to refine the inferred type.

```
                   # type of b.snapshot()
b                  # 0 * unknown
b.begin_record()   # 0 * {}
b.field("x")       # 0 * {"x": unknown}
b.integer(1)       # 0 * {"x": int64}
b.end_record()     # 1 * {"x": int64}
b.begin_record()   # 1 * {"x": int64}
b.field("x")       # 1 * {"x": int64}
b.real(2.2)        # 1 * {"x": float64}
b.field("y")       # 1 * {"x": float64, "y": ?unknown}
b.integer(2)       # 1 * {"x": float64, "y": ?int64}
b.end_record()     # 2 * {"x": float64, "y": ?int64}
b.null()           # 3 * ?{"x": float64, "y": ?int64}
b.string("hello")  # 4 * ?union[{"x": float64,
                   #              "y": ?int64}, string]
```

In the above example, an initially empty `ak.ArrayBuilder` named `b` has unknown type and zero length. With `begin_record`, its type becomes a record with no fields. Calling `field` adds a field of unknown type, and following that with `integer` sets the field type to an integer. The length of the array is only increased when the record is closed by `end_record`.

In the next record, field `"x"` is filled with a floating point number, which retroactively updates previous integers to floats. Calling `b.field("y")` introduces a field `"y"` to all records, though it has option type because this field is missing for all previous records. The third record is missing (`b.null()`), which refines its type as optional, and in place of a fourth record, we append a string, so the type becomes a union.

Internally, `ak.ArrayBuilder` maintains a similar tree of array buffers as an `ak.Array`, except that all buffers can grow (when the preallocated space is used up, the buffer is reallocated and copied into a buffer 1.5× larger), and `content` nodes can be replaced from specialized types to more general types. Taking a snapshot *shares* buffers with the new array, so it is a lightweight operation.

Although `ak.ArrayBuilder` is compiled code and calls into it are specialized by Numba, its dynamic typing has a runtime cost: filling NumPy arrays is faster. `ak.ArrayBuilder` trades runtime performance for convenience; faster array-building methods would have to be specialized by type.

High-level behaviors

One of the surprisingly popular uses of Awkward 0.x has been to add domain-specific methods to records and arrays by subclassing their hierarchical node types. These can act on scalar records returning scalars, like a C++ or Python object,

```
# distance between points1[0] and points2[0]
points1[0].distance(points2[0])
```

or they may be "vectorized," like a ufunc,

```
# distance between all points1[i] and points2[i]
points1.distance(points2)
```

This capability has been ported to Awkward 1.x and expanded upon. In Awkward 1.x, records can be named (as part of more general "properties" metadata in C++) and record names are linked to Python classes through an `ak.behavior` dict.

```
class Point:
    def distance(self, other):
        return np.sqrt((self.x - other.x)**2 +
                       (self.y - other.y)**2)

class PointRecord(Point, ak.Record):
    pass

class PointArray(Point, ak.Array):
    pass

ak.behavior["point"] = PointRecord
ak.behavior["*", "point"] = PointArray

points1 = ak.Array([{"x": 1.1, "y": 1},
                    {"x": 2.2, "y": 2},
                    {"x": 3.3, "y": 3}],
                   with_name="point")

points2 = ak.Array([{"x": 1, "y": 1.1},
                    {"x": 2, "y": 2.2},
                    {"x": 3, "y": 3.3}],
                   with_name="point")

points1[0].distance(points2[0])
# 0.14142135623730964

points1.distance(points2)
# <Array [0.141, 0.283, 0.424] type='3 * float64'>

points1.distance(points2[0])    # broadcasting
<Array [0.141, 1.5, 2.98] type='3 * float64'>
```

When an operation on array nodes completes and the result is wrapped in a high-level `ak.Array` or `ak.Record` class for the user, the `ak.behavior` is checked for signatures that link records and arrays of records to user-defined subclasses. Only the name `"point"` is stored with the data; methods are all added at runtime, which allows schemas to evolve.

Other kinds of behaviors can be assigned through different signatures in the `ak.behavior` dict, such as overriding ufuncs,

```
# link np.absolute("point") to a custom function
def magnitude(point):
    return np.sqrt(point.x**2 + point.y**2)

ak.behavior[np.absolute, "point"] = magnitude

np.absolute(points1)
# <Array [1.49, 2.97, 4.46] type='3 * float64'>
```

as well as custom broadcasting rules, and Numba extensions (typing and lowering functions).

As a special case, strings are not defined as an array type, but as a parameter label on variable-length lists. Behaviors that present these lists as strings (overriding `__repr__`) and define per-string equality (overriding `np.equal`) are preloaded in the default `ak.behavior`.

Awkward Arrays and Pandas

Awkward Arrays are registered as a Pandas extension, so they can be losslessly embedded within a `Series` or a `DataFrame` as a column. Some Pandas operations can be performed on them—particularly, NumPy ufuncs and any high-level behaviors that override ufuncs—but best practices for using Awkward Arrays within Pandas are largely unexplored. Most Pandas functions were written without deeply nested structures in mind.

It is also possible (and perhaps more useful) to translate Awkward Arrays into Pandas's own ways of representing nested structures. Pandas's MultiIndex is particularly useful: variable-length lists translate naturally into MultiIndex rows:

```
ak.pandas.df(ak.Array([[[1.1, 2.2], [], [3.3]],
                       [],
                       [[4.4], [5.5, 6.6]],
                       [[7.7]],
                       [[8.8]]]))
#                              values
# entry subentry subsubentry
# 0     0        0                1.1
#                1                2.2
#       2        0                3.3
# 2     0        0                4.4
#       1        0                5.5
#                1                6.6
# 3     0        0                7.7
# 4     0        0                8.8
```

and nested records translate into MultiIndex column names:

```
ak.pandas.df(ak.Array([{"I":
                          {"a": _, "b": {"c": _}},
                        "II":
                          {"x": {"y": {"z": _}}}}
                       for _ in range(0, 50, 10)]))
#          I      II
#          a   b   x
#              c   y
#                  z
# entry
# 0        0   0   0
# 1       10  10  10
# 2       20  20  20
# 3       30  30  30
# 4       40  40  40
```

In the first of these two examples, empty lists in the Awkward Array do not appear in the Pandas output, though their existence may be inferred from gaps between entry and subentry indexes. When analyzing both lists and non-list data, or lists of different lengths, it is more convenient to translate an Awkward Array into multiple DataFrames and `JOIN` those DataFrames as relational data than to try to express it all in one DataFrame.

This example highlights a difference in applicability between Pandas and Awkward Array: Pandas is better at solving problems with long-range relationships, joining on relational keys, but the structures that a single DataFrame can represent (without resorting to Python objects) is limited. Awkward Array allows general data structures with different length lists in the same structure, but most calculations are elementwise, as in NumPy.

GPU backend

One of the advantages of a vectorized user interface is that it is already optimal for calculations on a GPU. Imperative loops have to be redesigned when porting algorithms to GPUs, but CuPy, Torch, TensorFlow, and JAX demonstrate that an interface consisting of array-at-a-time functions hides the distinction between CPU calculations and GPU calculations, making the hardware transparent to users.

Partly for the sake of adding a GPU backend, all instances of reading or writing to an array's buffers were restricted to the "array manipulation" layer of the project (see Figure 4). The first implementation of this layer, "cpu-kernels," performs all operations that actually access the array buffers, and it is compiled into a physically separate file: `libawkward-cpu-kernels.so`, as opposed to the main `libawkward.so`, Python extension module, and Python code.

In May 2020, we began developing the "cuda-kernels" library, provisionally named `libawkward-cuda-kernels.so`

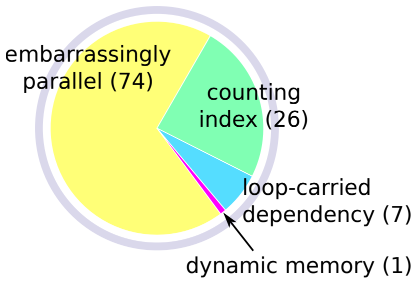

***Fig. 5:** CPU kernels by algorithmic complexity, as of February 2020.*

(to allow for future non-CUDA versions). Since the main codebase (`libawkward.so`) never dereferences any pointers to its buffers, main memory pointers can be transparently swapped for GPU pointers with additional metadata to identify which kernel to call for a given set of pointers. Thus, the main library does not need to be recompiled to support GPUs and it can manage arrays in main memory and on GPUs in the same process, which could be important, given the limited size of GPU memory. The "cuda-kernels" will be deployed as a separate package in PyPI and Conda so that users can choose to install it separately as an "extras" package.

The kernels library contains many functions (428 in the `"extern C"` interface with 124 independent implementations, as of May 2020) because it defines all array manipulations. All of these must be ported to CUDA for the first GPU implementation. Fortunately, the majority are easy to translate: Figure 5 shows that almost 70% are simple, embarrassingly parallel loops, 25% use a counting index that could be implemented with a parallel prefix sum, and the remainder have loop-carried dependencies or worse (one used dynamic memory, but it has since been rewritten). The kernels were written in a simple style that may be sufficiently analyzable for machine-translation, a prospect we are currently investigating with pycparser.

Transition from Awkward 0.x

Awkward 0.x is popular among physicists, and some data analyses have come to depend on it and its interface. User feedback, however, has taught us that the Awkward 0.x interface has some inconsistencies, confusing names, and incompatibilities with NumPy that would always be a pain point for beginners if maintained, yet ongoing analyses must be supported. (Data analyses, unlike software stacks, have a finite lifetime and can't be required to "upgrade or perish," especially when a student's graduation is at stake.)

To support both new and ongoing analyses, we gave the Awkward 1.x project a different Python package name and PyPI package name from the original Awkward Array: `awkward1` versus `awkward`. This makes it possible to install both and load both in the same process (unlike Python 2 and Python 3). Conversion functions have also been provided to aid in the transition.

We are already recommending Awkward 1.x for new physics analyses, even though serialization to and from the popular ROOT file format is not yet complete. Nevertheless, the conversion functions introduce an extra step and we don't expect widespread adoption until the Uproot library natively converts ROOT data to and from Awkward 1.x arrays.

Eventually, however, it will be time to give Awkward 1.x "official" status by naming it `awkward` in Python and PyPI. At that time, Awkward 0.x will be renamed `awkward0`, so that a single

```
import awkward0 as awkward
```

would be required to maintain old analysis scripts.

As an incentive for adopting Awkward 1.x in new projects, it has been heavily documented, with complete docstring and doxygen coverage (already exceeding Awkward 0.x).

Summary

By providing NumPy-like idioms on JSON-like data, Awkward Array satisfies a need required by the particle physics community. The inclusion of data structures in array types and operations was an enabling factor in this community's adoption of other scientific Python tools. However, the Awkward Array library itself is not domain-specific and is open to use in other domains. We are very interested in applications and feedback from the wider data analysis community.

Acknowledgements

Support for this work was provided by NSF cooperative agreement OAC-1836650 (IRIS-HEP), grant OAC-1450377 (DIANA/HEP) and PHY-1520942 (US-CMS LHC Ops).

References

[np] Stéfan van der Walt, S. Chris Colbert and Gaël Varoquaux. *The NumPy Array: A Structure for Efficient Numerical Computation*, Computing in Science & Engineering, 13, 22-30 (2011), DOI:10.1109/MCSE.2011.37

[ak1] Jim Pivarski, Jaydeep Nandi, David Lange, Peter Elmer. *Columnar data processing for HEP analysis*, Proceedings of the 23rd International Conference on Computing in High Energy and Nuclear Physics (CHEP 2018). DOI:10.1051/epjconf/201921406026

[ak2] Jim Pivarski, Peter Elmer, David Lange. *Awkward Arrays in Python, C++, and Numba*, CHEP 2019 proceedings, EPJ Web of Conferences (CHEP 2019). arxiv:2001.06307

[arrow] Apache Software Foundation. *Arrow: a cross-language development platform for in-memory data*, https://arrow.apache.org

[scipy] Pauli Virtanen et al. *SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python*, SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nature Methods, in press. DOI:10.1038/s41592-019-0686-2

[hydra] R. K. Böck. *Initiation to Hydra*, https://cds.cern.ch/record/864527 (1974), DOI:10.5170/CERN-1974-023.402

[phypy] Jim Pivarski. *Programming languages and particle physics*, https://events.fnal.gov/colloquium/events/event/pivarski-colloq-2019 (2019).

[pandas] Wes McKinney. *Data Structures for Statistical Computing in Python*, Proceedings of the 9th Python in Science Conference, 51-56 (2010), DOI:10.25080/Majora-92bf1922-00a

[numba] Siu Kwan Lam, Antoine Pitrou, Stanley Seibert. *Numba: a LLVM-based Python JIT compiler*, LLVM '15: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, 7, 1-6 (2015), DOI:10.1145/2833157.2833162

[root] Rene Brun and Fons Rademakers, *ROOT: an object oriented data analysis framework*, Proceedings AIHENP'96 Workshop, Lausanne, (1996), Nucl. Inst. & Meth. in Phys. Res. A 389 (1997) 81-86.

[root-EB] Axel Naumann. *ROOT as a framework and analysis tool in run 3 and the HL-LHC era*, https://indico.cern.ch/event/913205/contributions/3840338 (2020).

[bikes] City of Chicago Data Portal, https://data.cityofchicago.org

[nep13] Pauli Virtanen, Nathaniel Smith, Marten van Kerkwijk, Stephan Hoyer. *NEP 13 — A Mechanism for Overriding Ufuncs*, https://numpy.org/neps/nep-0013-ufunc-overrides.html

[nep18] Stephan Hoyer, Matthew Rocklin, Marten van Kerkwijk, Hameer Abbasi, Eric Wieser. *NEP 18 — A dispatch mechanism for NumPy's high level array functions*, https://numpy.org/neps/nep-0018-array-function-protocol.html