Columnar data processing for HEP analysis

Jim Pivarski[1], Jaydeep Nandi[2], David Lange[1], and Peter Elmer[1]

[1]Princeton University
[2]National Institute of Technology, Silchar, India

Abstract. In the last stages of data analysis, physicists are often forced to choose between simplicity and execution speed. In High Energy Physics (HEP), high-level languages like Python are known for ease of use but also very slow execution. However, Python is used in speed-critical data analysis in other fields of science and industry. In those fields, most operations are performed on Numpy arrays in an array programming style; this style can be adopted for HEP by introducing variable-sized, nested data structures. We describe how array programming may be extended for HEP use-cases and an implementation known as awkward-array. We also present integration with ROOT, Apache Arrow, and Parquet, as well as preliminary performance results.

1 Introduction

The field of High Energy Physics (HEP) has always dealt with big data, large enough datasets that finite memory and processing speed cannot be ignored. Traditionally, physicists have relied on compiled languages like Fortran and C++ to analyze large datasets in batch, but this limits interactive exploration. Today, data analysts in other academic fields and industry are developing tools to analyze large datasets in high-level languages, but these tools treat data as rectangular tables and flat arrays. HEP analysis requires *nested* data— arbitrarily many tracks within jets or variable-sized collections of particle candidates— which do not fit into rectangular data without padding and truncation.

This paper describes how the concepts of array programming may be extended to more complex data structures. It also introduces awkward-array [1], an implementation of these "awkward" data structures as arrays, which is written in pure Python (depending only on Numpy [2]) for portability.

2 Array programming and its extension to nested data

Array programming is an interface in which individual user commands apply to whole arrays: each function call performs a regular operation on millions of data points. Pioneered by APL [3] in 1962, this programming style appears primarily in analyst-facing data processing languages such as S (1976), MATLAB (1984), S-PLUS (1988), R (1993), and Numpy (2005). Beyond its succinct syntax, which can be easier to read than the corresponding for loops, it encourages simple passes over columnar data that facilitates CPU cache-prefetching and vectorization. In fact, programs written in an array programming style are also appropriately organized for Single Instruction Multiple Data (SIMD) processing, which makes it easier to migrate to GPUs.

Array programming usually includes the following elements (expressed in Numpy syntax here):

- Multidimensional slices: `rgb_pixels[0, 50:100, ::3]`
- Elementwise operations: `all_pz = all_pt * sinh(all_eta)`
- Broadcasting: `all_phi - 2*pi`
- Array reduction: `array.sum()` → scalar
- Masking (list compaction): `data[trigger & (pt > 40)]`
- Fancy indexing ("gather"): `all_eta[argsort(all_pt)]`
- Row/column commutativity `table["column"][7]` (row 7 of column array)
  `table[7]["column"]` (field of row tuple 7)

The semantics of each of these is well established for rectangular tables and flat arrays, but they can be extended to arbitrarily nested data structures. The examples below illustrate common cases that are defined in detail later in the text. In these examples, `events["jets"]`, `jetpt`, etc., contain lists of arbitrarily many objects or values.

- Multidimensional slices: `events["jets"][:, 0]` → first jet per event
- Elementwise operations: `jetpt * sinh(jeteta)` → keep nested structure
- Broadcasting: `jetphi - metphi` → expand `metphi` from one-per-event to one-per-jet before operation
- Array reduction: `jetpt.max()` → array of max jet p_T per event
- Masking (list compaction): `data[trigger]` → drop whole events
  `data[jetpt > 40]` → drop jets from events
- Fancy indexing ("gather"): `a = argmax(jetpt)` → `[[2], [], [1], [4]]`
  `jeteta[a]` → `[[3.6], [], [-1.2], [0.4]]`
- Row/column commutativity `events["jets"]["pt"][7, 1]` is the same as
  `events["jets"][7]["pt"][1]` and
  `events["jets"][7, 1]["pt"]` etc.

3 High-level types and array programming interface

Arrays introduce a degree of static typing to dynamically typed programming languages, in the sense that all elements of an array have the same type and an operation only needs to perform a type-check once per array, rather than once per element. Conventional array processing environments like Numpy only support primitive types: numbers, booleans, and other fixed-size values (including records of fixed-size values). Fixed-size subarrays are presented as an array's multidimensional "shape," and may be considered part of the type. Introducing nested data requires more data types. (Below, **bold** keywords correspond to the bulleted lists of the previous section.)

The type of a flat array with length n can be denoted $[0, n) \to P$, as a function signature that takes integers between 0 (inclusive) and n (exclusive) to a primitive type P. An array of length n with variable-sized subarrays would be $[0, n) \to [0, \infty) \to P$, as any non-negative integer can be in the second domain. This is known as a jagged, or ragged, array. Jagged arrays may be nested: $[0, n) \to [0, \infty) \to [0, \infty) \to P$.

Extending the concept of **multidimensional slices** from subintervals of $[0, n)$ to subintervals of $[0, \infty)$ is straightforward, though the result is itself jagged because some subarrays may be smaller than the slice and are clipped. **Elementwise operations** also have a straightforward extension to jagged arrays, but not only must the total array lengths match among operands, their subarray lengths must match as well.

Conventional **broadcasting** allows scalar P data to be an operand in elementwise operations with arrays $[0, n) \to P$ by duplicating the one scalar to n array elements. Jagged broadcasting duplicates each element of a flat array $[0, n) \to P$ to the corresponding jagged array elements $[0, n) \to [0, \infty) \to P$. The same applies for any depth of jaggedness: broadcasting increases the jaggedness of operands to match the jaggedness of the most deeply nested array.

Reducing a flat $[0, n) \to P$ array, such as computing its sum, min, or max, returns a scalar P. Reducing a jagged $[0, n) \to [0, \infty) \to P$ array, such as computing the sum of each subarray, retuns $[0, n) \to P$, decreasing its jaggedness by one.

When **masking** a jagged array, we may want to remove subarrays (representing events) or we may want to remove subarray elements (representing particles). We can make that distinction by contrasting a boolean mask $[0, n) \to$ `bool` with a jagged boolean mask $[0, n) \to [0, \infty) \to$ `bool`. In keeping with the rules of conventional array programming, a boolean mask would remove subarrays from a jagged array $[0, n) \to [0, \infty) \to P$. Extending these rules, a jagged boolean mask can remove subarray elements if they have the same jagged structure.

Similarly, an integer array $[0, m) \to$ `int` selects m subarrays from a jagged array by the conventional rules, but a jagged integer array $[0, n) \to [0, \infty) \to$ `int` can select subarray elements. Since this **gather** feature is often used with functions like *argmax* and *argmin* to identify important indexes in one array and select elements at those indexes from another array, the jagged *argmax* and *argmin* should return jagged arrays with zero or one-element subarrays— the max or min if any elements exist.

Tables with named, differently typed columns can be included in this type system as a mapping from names to types. For instance, a table with three columns is $[0, n) \to \{\texttt{"one"} \to P_1,\ \texttt{"two"} \to P_2,\ \texttt{"three"} \to P_3\}$.

In our extension of array programming, tables and jaggedness can be composed. This is useful in HEP for describing event records that contain variable-sized lists of particles, each with a different multiplicity, each containing attributes of various types. In some cases, it's useful for the particle attributes to be variable-sized lists as well. Type descriptions may then be arbitrary trees, like the following:

$$[0, n) \to [0, \infty) \to \left\{ \begin{array}{lll} \texttt{"one"} & \to & [0, m) \to \left\{ \begin{array}{lll} \texttt{"x"} & \to & [0, \infty) \to P_1 \\ \texttt{"y"} & \to & P_2 \end{array} \right. \\ \texttt{"two"} & \to & P_3 \\ \texttt{"three"} & \to & P_4. \end{array} \right.$$

Rectangular tables can be sliced by row (integer index) or column (string index). So can jagged tables. A table described by $[0, n) \to [0, \infty) \to \{\texttt{"one"} \to T_1,\ \texttt{"two"} \to T_2\}$ sliced by a string index `"one"` returns a jagged array $[0, n) \to [0, \infty) \to T_1$. In general, this means that integer indexes **commute** with string indexes, though integer indexes do not commute with other integer indexes, nor do string indexes commute with other strings.

Primitives, variable-sized lists, and tables of records would be sufficient for most HEP cases, but sometimes polymorphic types and cross-references (pointers) are needed. Polymorphic types can be included with unions $T = T_1 \mid T_2$ (elements may be type T_1 or type T_2) and optional types $?T$ (elements may be type T or null). Since the possibilities of a polymorphic type are enumerated, operations must be legal on all enumerated possibilities, and operations on an optional type are only applied to non-null elements.

For cross-references, such as tagging jets with their associated leptons, we allow type trees to be arbitrary graphs. A list may contain elements of a list elsewhere in the graph or its own ancestors. As an example of the latter case,

$$[0, n) \to T := \left(\texttt{int} \;\middle|\; [0, \infty) \to T\right)$$

is an array that may contain integers, lists of integers, or any depth of lists of integers.

This set of types is as inclusive as most high-level programming languages, and it can be constructed entirely from columnar arrays.

4 Low-level data structures and their uses

The fundamental construct for variable-sized, nested structure is the jagged array. To represent a list of unequal-length sublists with the following logical structure, we store the flattened content in one array and the structure in either an “offsets” or a “parents” array.

```
Logical structure:   [[0, 1, 2], [], [3, 4], [5, 6, 7, 8], []  ]
Content:             [ 0, 1, 2,       3, 4,    5, 6, 7, 8]
Offsets:             [0,         3,  3,      5,            10, 10]
Parents:             [ 0, 0, 0         2, 2,   3, 3, 3, 3]
```

The offsets array may be constructed as the content index after every opening bracket (the “starts” array) appended by the total length, or it may be constructed as the content index after every closing bracket (the “stops” array) prepended by zero. It is useful for descending from outer levels to inner levels of structure.

The parents array has the same length as the contents array; it associates each content element with the index of the subarray that contains it. A parents array is incapable of expressing empty sublists at the end of a list (as in the above example), but it can specify empty lists in a sequence by skipping integers. Offsets are therefore more general than parents, but parents are useful in reducer operations that ascend from inner levels to outer levels of structure.

The awkward-array library has a `JaggedArray` implementation that allows arbitrary contents— nesting a `JaggedArray` within a `JaggedArray` constructs $[0, n) \to [0, \infty) \to [0, \infty) \to P$. The `JaggedArray.__getitem__` method overrides square brackets in Python to provide slicing, masking, and fancy indexing with all the features described in the previous section. Elementwise operations and broadcasting are implemented in `__array_ufunc__`, which overrides the behavior of all universal functions in Numpy [4].

Tables of records are implemented as a `Table` class that maps column names to arrays. This is like Numpy's “record arrays,” except that a Numpy record array must be a contiguous array of structs, while awkward-array's `Table` may be non-contiguous or a struct of arrays. `JaggedArray` and `Table` interact to let integer and string valued indexes commute, which hides the arrays-of-structs/structs-of-arrays distinction.

`UnionArray` and `MaskedArray` implement polymorphism and optional types. Numpy has a masked array type, but `MaskedArray` shares conventions with the rest of awkward-array's suite and additionally has a `BitMaskedArray` variant for compatibility with Apache Arrow [5] (next section).

In HEP applications, it is useful to have datasets representing arrays of objects, such as $[0, n) \rightarrow$ `LorentzVector`, which returns `LorentzVector` objects when given an index. Numpy has an "object array" type, which stores pointers to Python objects in memory, but awkward-array's `ObjectArray` constructs a Python object when given an index. This allows us to represent such data with $12\times$ less memory, as we will see in the section on performance below.

In the same spirit as integer/string index commutivity, methods of objects returned by `ObjectArray` may be applied to the entire array for vectorized HEP calculations. For instance, an `ObjectArray` of `LorentzVectors` may be boosted en masse:

```
subleading_jets.boost(leading_jets)
```

which works equally well if `subleading_jets` is a jagged array (several per event) and `leading_jets` is a flat array (one per event), thanks to jagged broadcasting. An array of strings is an `ObjectArray` of a `JaggedArray`, converting subarrays of characters into `str` objects on demand.

In addition, awkward-array provides some array classes for dealing with data structures without affecting their high-level data types. Some file formats provide data in discontiguous chunks, such as ROOT's baskets [6] or Parquet's pages and row groups [7]. `ChunkedArray` lets us view discontiguous chunks of data as a single logical array without copying. `VirtualArray` uses a given function to generate an array on demand, rather than holding it in memory (optionally passing it to a temporary cache). A chunked, virtual array corresponds to Dask's concept of a delayed array [8].

Any of these can be cross-referenced in Python, as the implementation is careful to avoid infinite recursion. To cross-reference array elements, to associate a single lepton with a jet for instance, `IndexedArray` represents a deferred indirection. An `index` array stores locations `j[i]` to look up in a `content` array: given `i`, it returns `content[j[i]]`. `JaggedArray` is like `IndexedArray` except that it represents ranges of `content` indexes, rather than individual indexes.

`SparseArray` is the conceptual opposite of `IndexedArray`: its `index` and `content` must be the same length and each `index` is the logical position of the corresponding `content`. Other indexes are presumed to be zero or some default value. As a sparse matrix, this is known as the COOrdinate format [9]. Whereas an `IndexedArray` represents a smaller logical array than its `content` (or one with duplicates), a `SparseArray` represents a larger logical array than its `content`. It is useful in HEP for looking up indexes in a table that is far too large to fit in memory, such as a mapping from ten-digit detector id numbers to measurement corrections.

5 ROOT/Arrow/Parquet compatibility and persistence

The awkward-array classes were not designed to be a new format, but to give existing formats new behaviors, so that they can be used in an array programming style. As such, they are configurable to fit a variety of representations.

The scope of awkward-array was chosen such that ROOT, Apache Arrow, and Parquet data may be viewed through configurations of the awkward classes. Numpy natively supports the full range of numeric types, big and little endian byte orders, and fixed-size subarrays, which awkward-array inherits. `ByteJaggedArray` can use ROOT's basket format directly, which specifies the positions of variable-sized entries by byte index. All the baskets in a branch may be viewed as a `ChunkedArray`, while lazy loading and decompression are provided by `VirtualArray`. A ROOT TTree is a (possibly jagged) `Table`, and pointer references can be emulated with `IndexedArray`. Arbitrary C++ types are encoded in ROOT's streamers, which uproot [10] translates to Python classes. Data with this general type are then presented as `ObjectArrays` with `TLorentzVector` being the primary example.

Apache Arrow has a simpler data model consisting only of primitives, arbitrary-length lists, records, unions, strings, date/time types, and "dictionary encodings," in which a small set of large values, such as the names of categories (strings), are represented by fixed-width integers for efficiency. All of these arrays are bit-masked to allow missing (null) data anywhere. `JaggedArrays` represent the lists, `Tables` the records, `UnionArrays` the unions, `StringArrays` (which are just `JaggedArrays` of characters) represent the strings, and `ObjectArrays` represent the date/time types. Dictionary encodings are `IndexedArrays` without cross-references. The Parquet file format is wildly different, beyond the scope of awkward-array to wrap directly, but pyarrow (the Python front-end for Arrow-C++ and Parquet-C++) converts Parquet into Arrow.

Although most data would be drawn from a standard format like ROOT (in HEP) or Parquet (beyond HEP), it is often necessary to save derived quantities after some analysis. Derived quantities might use features not available in the source format. Therefore, awkward-array has its own persistence format: a JSON string describing how awkward-array classes are nested with cross-references and a binary blob for each array. These may be placed in any container of named binary blobs, such as a ZIP file, an HDF5 file, or a distributed object store. By default, `VirtualArrays` representing data from ROOT or Parquet are saved as instructions for reading from the original file, rather than the data themselves. Thus, an analyst can read ROOT or Parquet data, add or replace fields in its `Tables`, and save the ensemble in a file representing only the changes, which can be considerably smaller than the original data.

6 Performance measurements

The data structures and operations described here were designed for performance, but a full study has not yet been carried out. Such a study would require a target HEP analysis, since performance differs in detail from one analysis to the next. However, arrays have a natural order parameter: their size, which should be chosen to significantly exceed any metadata in the awkward-array classes. It also has an obvious hotspot: stepping through Python instructions is significantly slower than any numerical operations. This prescribes a simple rule— Python code should never iterate over all elements of an array; any operation that scales with the number of elements in an array must be performed in compiled code (i.e. Numpy).

As a rough indicator of performance trade-offs, we studied one operation in a variety of contexts, the jagged elementwise calculation

```
pz = pt * numpy.sinh(eta)
```

where `pt` and `eta` are jagged arrays with the same structure: 552,056 muons in 701,716 events. These tests were performed on CERN's SWAN service and all files to

Table 1. Horizontal study of `pz = pt * numpy.sinh(eta)` in many types of systems.

MB	RAM memory occupied by data	time to complete load, compute, or both	sec
311.95	Python list of lists of dicts	PyROOT load and compute	45.9
215.11	root_numpy's array of arrays	JaggedArray compute in Python for loops	13.4
139.79	Python list of lists of `__slots__` classes		
		root_numpy compute in loop over ufuncs	1.96
37.19	serialized JSON text	Python list of lists of dicts in Python for loops	1.24
		Python list of lists of `__slots__` classes in Python for loops	1.23
22.38	`std::vector<std::vector<struct>>`	root_numpy load	0.635
11.67	JaggedArray of Table of pt, eta, phi	ROOT RDataFrame load and compute	0.163
		ROOT TTreeReader load and compute	0.091
		ROOT TBranch::GetEntry load and compute	0.046
		uproot load	0.031
		JaggedArray compute as Numpy-like ufunc	0.023
		JaggedArray compute in Numba-accelerated Python for loops	0.023

reproduce the study are shared publicly on CERNBox [11]. Table 1 shows the memory used to store these data before calculation and the time to compute in awkward-array, ROOT, pure Python, and instrumented Python like awkward-array and PyROOT in which Python code is iterating over all elements of the arrays. The differences span orders of magnitude.

If this style of programming is adopted for HEP analysis, it presents an opportunity to optimize the array primitives independently of the analysis code. Array-centric data structures are ripe for vectorization, but some of the naïve algorithms, such as jagged reduction, have loop-carried dependencies that prevent vectorization. One of us (Nandi) developed vectorized algorithms for jagged operations [12], such as this CUDA implementation of jagged reduction (fully described in a notebook [13]):

```
int i = threadIdx.x;
for (int d = 1;  d < blockSize.x;  d *= 2) {
    if (i >= d  &&  parents[i] == parents[i - d]) {
        content[i] += content[i - d];
    }
    __syncthreads();
}
```

The rate of this algorithm on a Tesla P100 GPU and vector instructions on a Xeon Silver 4110 CPU are shown in Figure 1 as a function of the Poisson average number of items per group. For comparison, two sequential algorithms (nested for loops with all compiler optimizations, same CPU) are overlaid.

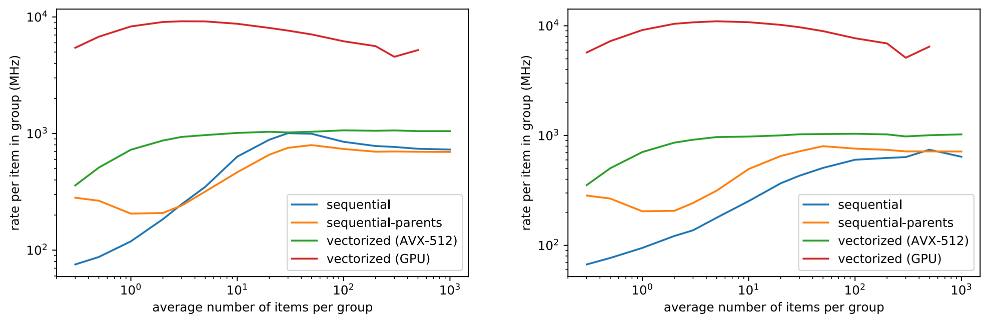

Figure 1. Jagged reduction `jaggedarray.sum()` (left) and `jaggedarray.max()` (right) for two sequential algorithms (CPU) and our vectorized algorithm on CPU and GPU.

7 Future directions

Numpy arrays are the common language of Python's scientific ecosystem. Now that structures necessary for HEP have been cast as arrays, they can be more easily integrated into this ecosystem. As a next step, awkward-arrays will be added as a Pandas [14] extension type, so that columns in a Pandas DataFrame may be structured. Following that, `ChunkedArrays` of `VirtualArrays` will be wrapped as Dask delayed arrays so that calculations can be distributed without moving data. Problems that require conventional loops and conditionals can be optimized by wrapping awkward-arrays as Numba extensions. Finally, awkward-array's exclusive dependence on Numpy should make it transparently portable to GPUs with CuPy [15], but that remains to be tested.

8 Acknowledgements

This work was supported by the National Science Foundation under grants ACI-1450377 and PHY-1624356.

References

[1] Jim Pivarski, "awkward-array" [software], Release 0.4.1, Zenodo, 26 October, 2018. https://zenodo.org/record/1472437
[2] Travis E, Oliphant, "A guide to NumPy," USA: Trelgol Publishing, (2006).
[3] Kenneth E. Iverson, *A programming language* (John Wiley & Sons, Inc., New York, 1962).
[4] Blake Griffith, "NEP 13: a mechanism for overriding ufuncs" [Numpy Enhancement Proposal (accepted)], 10 July 2017. https://www.numpy.org/neps/nep-0013-ufunc-overrides.html
[5] Arrow Development Team, "Apache Arrow" [website]. https://arrow.apache.org
[6] Rene Brun and Fons Rademakers, "The ROOT Object I/O System" [specification], 26 August, 1996. https://root.cern.ch/root/InputOutput.html
[7] Julien Le Dem, "Apache Parquet" [specification], 29 April 2015. https://parquet.apache.org/documentation/latest
[8] Dask Development Team, "Dask: Library for dynamic task scheduling" 2016. https://dask.org
[9] Youcef Saad, "SPARSKIT: a basic tool kit for sparse matrix computations" [technical report], 21 May 1990. https://ntrs.nasa.gov/search.jsp?R=19910023551
[10] Jim Pivarski et al., "uproot" [software], Release 3.2.6, Zenodo, 22 October, 2018. https://zenodo.org/record/1469102
[11] Performance test 1: https://cernbox.cern.ch/index.php/s/uhAH75uGCddFlFL
[12] Jaydeep Nandi, "Vectorized proof of concepts" [notebooks] August 2018. https://gitlab.com/Jayd_1234/GSoC_vectorized_proof_of_concepts
[13] Performance test 2: https://github.com/jpivarski/jupyter-performance-studies/blob/master/2018-09-10-jagged-reduction.ipynb
[14] Wes McKinney, "Data Structures for Statistical Computing in Python," Proceedings of the 9th Python in Science Conference, 51-56 (2010) (publisher link).
[15] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido and Crissman Loomis, "CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations." Proceedings of Workshop on Machine Learning Systems (LearningSys), 31st Annual Conference on Neural Information Processing Systems (NIPS), (2017) (publisher link).