“Jim stuff”

Jim Pivarski

Princeton University – IRIS-HEP

June 19, 2019
Apologies for any misrepresentation

Event processing
- flat formulæ
- combinatorics
- reductions
- JIT/pre-compilation
- GPU transparency

Data formats
- ROOT I/O
- Parquet, Arrow

Scalable systems
- scheduling (Spark, Parsl)
- caching
- predicate push-down
- user scaling, security
- user interface
- monitoring, support
- real systems ($)

Event processing
- fast-filling
- combining (hadd)
- serialization, formats
- plotting, fitting interface
- multihistogram projection

Histogramming
- fast-filling
- combining (hadd)
- serialization, formats
- plotting, fitting interface
- multihistogram projection

DOMA
- worldwide distribution
- DDS, ServiceX
- column stores
- database-like indexing
- conversion
- replication

Data management
- global distribution
- DDS, ServiceX
- column stores
- database-like indexing
- conversion
- replication

Fitting
- simple fits (ergonomics)
- multihistogram (e.g. pyhf)
- fit-driven workflow (e.g. Gaussian process)

Reproducibility/recasting
- end-to-end encapsulation
- distributed deployment
- language?

Language/expression
- end-to-end analysis
- syntax vs embedded
- functional, declarative
- combinatorics, identity

Jim stuff $\approx$ event processing
Projects and collaborations
Year of teaching

- April 1: Software Carpentry at Fermilab (intensity frontier)
- April 8–10: PICSciE Numpy course at Princeton (mostly science)
- May 6: Language tools tutorial at ADL workshop (particle physics)
- May 28–29: Scientific Python and columnar analysis HATS (CMS)
- June 10: Software Carpentry at Argonne (U.S. ATLAS)
- July 22–26: CoDaS-HEP (particle physics)

- Pratyush Das (IRIS-HEP fellow)
- Charlie Escott (GSoC)
- Duy Nguyen (volunteer)
Uproot is wildly popular :)
Nobody (except Coffea) installs awkward-array on its own, only through uproot :(
The way to get new awkward features in front of a lot of physicists is to make it an option in uproot.
Uproot is wildly popular :) 
Nobody (except Coffea) installs awkward-array on its own, only through uproot :( 
The way to get new awkward features in front of a lot of physicists is to make it an option in uproot.
Lazy arrays have an interface like direct arrays but internally iterate in chunks.

```python
>>> ttree.array("Muon_pt")  # loads whole array, contiguously
<JaggedArray [[], [], [5.31576], ..., [26.35128], [], [44.2805, 6.699721]]>
>>> ttree.lazyarray("Muon_pt")  # loads first and last chunk (to print)
<ChunkedArray [[], [], [5.31567], ..., [26.35128], [], [44.2805, 6.699721]]>
```

The following splits data into 1 GB chunks, loading no more than 3 GB at a time, and computes $p_z$ from $Muon_{pt}$ and $Muon_{eta}$.

```python
>>> cache = uproot.ArrayCache("3 GB")
>>> events = uproot.lazyarrays("cms-nanoaod/*.root", "Events", entrysteps="1 GB", cache=cache)
>>> pz = events.Muon_pt * numpy.sinh(events.Muon_eta)
```
Lazy arrays have an interface like direct arrays but internally iterate in chunks.

```python
>>> ttree.array("Muon_pt")  # loads whole array, contiguously
<JaggedArray [[]] [[]] [5.31576] ... [26.35128] [[]] [44.2805 6.699721]]>
```

```python
>>> ttree.lazyarray("Muon_pt")  # loads first and last chunk (to print)
<ChunkedArray [[]] [[]] [5.31576] ... [26.35128] [[]] [44.2805 6.699721]]>
```

The following splits data into 1 GB chunks, loading no more than 3 GB at a time, and computes $p_z$ from $\text{Muon}_\text{pt}$ and $\text{Muon}_\text{eta}$.

```python
>>> cache = uproot.ArrayCache("3 GB")
>>> events = uproot.lazyarrays("cms-nanoaod/*.root", "Events",
        entrysteps="1 GB", cache=cache)
>>> pz = events.Muon_pt * numpy.sinh(events.Muon_eta)
```
If your TTrees are CMS NanoAOD, you can get lazy arrays as an event hierarchy:

```python
>>> events = ttree.lazyarrays(profile="cms.nanoaod")
>>> events
<Table [<Event 0> <Event 1> ... <Event 499998> <Event 499999>]>  
>>> events.muons               # now it reads nMuon...
<ChunkedArray [[]] [[]] [<Muon 0>] ... [] [<Muon 537187] <Muon 537188]>]]>
>>> events.muons.pt            # now it reads Muon_pt...
<ChunkedArray [[]] [[]] [5.315762] ... [] [44.28051 6.6997213]>
Lazy profile: CMS NanoAOD

Since everything is loaded on demand, some fields can be defined in terms of others without unnecessary reading or memory use.

```python
>>> events.muons.p4  # now it reads Muon_eta, Muon_phi, Muon_mass
<ChunkedArray [[]] []
  [TLorentzVector(5.315, 1.127, 2.695, 0.1057)]
  ...
  [TLorentzVector(44.281, 0.7856, 2.3838, 0.10571)
   T LorentzVector(6.6997, 1.9197, -0.0706, 0.10571)]>
```
Lazy profile: CMS NanoAOD

Since everything is loaded on demand, some fields can be defined in terms of others without unnecessary reading or memory use.

```python
>>> events.muons.p4  # now it reads Muon_eta, Muon_phi, Muon_mass
<ChunkedArray [[]] [[]]
    [TLorentzVector(5.315, 1.127, 2.695, 0.1057)]
    ...
    [TLorentzVector(44.281, 0.7856, 2.3838, 0.10571)
     T LorentzVector(6.6997, 1.9197, -0.0706, 0.10571)]
>`

Cross-references are presented as objects, too.

```python
>>> events.muons.jet  # now it reads Muon_JetIdx and nJet
<ChunkedArray [[]] [[]] [<Jet 12>] ... [<Jet 4102602> <Jet 4102603>]]`
```
Since everything is loaded on demand, some fields can be defined in terms of others without unnecessary reading or memory use.

```python
>>> events.muons.p4  # now it reads Muon_eta, Muon_phi, Muon_mass
<ChunkedArray [[]] []
    [TLorentzVector(5.315, 1.127, 2.695, 0.1057)]
    ...
    [TLorentzVector(44.281, 0.7856, 2.3838, 0.10571)
     T LorentzVector(6.6997, 1.9197, -0.0706, 0.10571)]>
```

Cross-references are presented as objects, too.

```python
>>> events.muons.jet  # now it reads Muon_JetIdx and nJet
<ChunkedArray [[]] [] [<Jet 12>] ... [<Jet 4102602> <Jet 4102603>]]>
```

So we can compute things like “$\Delta \phi$ between every muon and its associated jet.”

```python
>>> events.muons.p4.delta_phi(events.muons.jet.p4)
<ChunkedArray [[]] [] [-0.118652344] ... [0.0009765625 0.1444397]>
With lazy arrays, uproot now uses half of the awkward-array classes

- **JaggedArray**: array containing variable-length subarrays
- **Table**: struct of arrays, presented as an array of structs
- **ObjectArray**: creates Python objects on demand, such as `TLorentzVector`
- **Methods**: vectorized methods, like `muons.delta_phi(muons.jet)`
- **StringArray**: strings (jagged array of characters)
- **IndexedArray**: “pointers” into another array (via integer indexes)
- **SparseArray**: inverse of IndexedArray: zero everywhere except specified indexes
- **MaskedArray**: marks elements as None with a byte array
- **BitMaskedArray**: marks elements as None with a bit array
- **IndexedMaskedArray**: indexes and masks in one array to avoid placeholders in the data array
- **UnionArray**: contains multiple (enumerated) types
- **ChunkedArray**: view discontiguous memory buffers as one array
- **AppendableArray**: efficiently grow an array in chunks
- **VirtualArray**: load data on demand

Used by uproot to read ROOT, used to read Parquet, used for both lazy arrays are ChunkedArrays containing VirtualArrays.
New method (experimental): make Pandas recognize jagged arrays as columns.

```python
>>> ttree.array("Muon_pt").pandas  # JaggedArray → JaggedSeries
<JaggedSeries [[]] [5.315762] ... [26.351288] [] [44.28051 6.6997213]]>

>>> pandas.Series(ttree.array("Muon_pt").pandas)
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[]</td>
</tr>
<tr>
<td>1</td>
<td>[]</td>
</tr>
<tr>
<td>2</td>
<td>[5.315762]</td>
</tr>
<tr>
<td>3</td>
<td>[47.05483 19.042616]</td>
</tr>
<tr>
<td>4</td>
<td>[15.776729]</td>
</tr>
<tr>
<td>5</td>
<td>[14.511511]</td>
</tr>
<tr>
<td>6</td>
<td>[]</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>499994</td>
<td>[36.54806]</td>
</tr>
<tr>
<td>499995</td>
<td>[10.061976]</td>
</tr>
<tr>
<td>499996</td>
<td>[]</td>
</tr>
<tr>
<td>499997</td>
<td>[26.351288]</td>
</tr>
<tr>
<td>499998</td>
<td>[]</td>
</tr>
<tr>
<td>499999</td>
<td>[44.28051 6.6997213]</td>
</tr>
</tbody>
</table>

Length: 500000, dtype: awkward
awkward-cpp: Charlie Escott (GSoC) and maybe Duy Nguyen

EscottC commented 17 hours ago

Just for reference @Danbo3004 @jpivarski:
These are the possible codes from pybind11::buffer_info.format, which outputs a std::string. This property comes from a py::buffer, py::array, or py::array_t when you call [array].request().format:

Also, I believe the strings are treated as Unicode; I couldn't find a way to make them into ASCII. The reference to numpy dtypes is here, and the reference for pybind11(buffer_info.h) is here.

EscottC added some commits 7 hours ago

- non-native endian support
- byteswap support for all systems

jpivarski commented 5 hours ago

Before you put too much into custom byte-swapping functions, note that there are built-in byte-swapping intrinsics: https://codereview.stackexchange.com/a/64827.

Boost has an endian library (https://www.boost.org/doc/libs/1_63_0/libs/endian/doc/index.html), which you can use if it's header-only. Installing non-header-only Boost libraries is a giant pain, but installing header-only libraries is a breeze. According to this, the "endian" library is included in the list of header-only—worth checking.
awkward-numba: my primary coding project; last touched Feb 27
Meddling in histogram design

Okay, I think I have the grand unified theory of projection and rebinning: suppose that the third argument of a slice is always just some callable (never confusable with an integer stride) that we apply to an axis. If the result is reduces the dimension by 1, we eliminate the axis; if it keeps the dimension the same, we don’t.

The histrogramming library can provide two conveniences for this with the following definitions:

```python
def project(x):
    return numpy.add.reduce(x)
def rebin(n):
    return lambda x: numpy.add.reduceat(x, numpy.arange(0, len(x), n))
```

and they would be used in slices like:

```python
h[:,:project, 10:20:project]  # to keep the first axis, completely project away the second, and
h[:,:rebin(3), 10:20:rebin(2)]  # to keep the first axis as-is, rebin the second by a factor of
```

For context, this is what the two convenience functions do to an array:

```python
>>> a = numpy.ones(12)
>>> project(a)
12.0
>>> rebin(3)(a)
array([3., 3., 3., 3.])
>>> rebin(4)(a)
array([4., 4., 4., 4.])
```

The advantage of this over the kinds of tags we were talking about before is that the *rule* applied in `__getitem__` doesn’t depend on a named function from a particular library, but the library can provide some useful functions. The *rule* is open-ended enough that future users or libraries can invent some funky things without the `__getitem__` code having to change.

There’s also an aesthetic nicety that we don’t fundamentally distinguish projections from rebinnings except by inspecting the shape of the array returned by the function. And then these integrate with slices by always being found in the third argument of a slice object and never being confused for an integer stride.

This doesn’t address data coordinates in the slice versus integer coordinates in the slice. We could do this and Henry’s...
Toy languages: pattern matching for particle combinatorics

```javascript
// define a join pattern in a function
higgs(flavor1, flavor2) = join {
  z1 ~ {
    lep1 ~ flavor1 // lep1, lep2 from flavor1 collection
    lep2 ~ flavor1 // leptons are NOT double-counted
    mass = (lep1.p4 + lep2.p4).mass
  }
  z2 ~ {
    lep1 ~ flavor2 // lep1, lep2 from flavor2, which
    lep2 ~ flavor2 // might be the same as flavor1
    mass = (lep1.p4 + lep2.p4).mass
  }
}.filter(h => h.z1.lep1.charge != h.z1.lep2.charge and
           h.z2.lep1.charge != h.z2.lep2.charge)
           .sort(h => (h.z1.mass - 91)**2 + (h.z2.mass - 91)**2)

higgs4e = higgs(electrons, electrons) // use the function
higgs4mu = higgs(muons, muons) // to match patterns
higgs2e2mu = higgs(electrons, muons)
```

// use the function
// to match patterns
public class JaggedArray extends Array {
    PrimitiveArray.Int4 offsets;
    Array content;

    public JaggedArray(Interpretation interpretation, int length, PrimitiveArray.Int4 offsets, Array content) {
        super(interpretation, length);
        this.offsets = offsets;
        this.content = content;
    }
}
```plaintext
include "interpretation.fbs";

enum Compression: int {
    none = 0,
    zlib = 1,
    lzma = 2,
    old = 3,
    lz4 = 4
}

table Branch {
    local_offsets: [ulong] (required);
    page_seeks: [ulong] (required);
    compression: Compression;
    iscompressed: [bool];
    compressedbytes: [uint];
    uncompressedbytes: [uint] (required);
    basket_page_offsets: [uint] (required);
    basket_keylens: [uint];
    basket_data_borders: [uint];
}

table Column {
    interp: uproot_skyhook
        .interpretation_generated
        .Interpretation (required);
    title: string;
}

table File {
    location: string (required);
    uuid: string (required);
    branches: [Branch] (required);
}

table Dataset {
    name: string (required);
    treepath: string (required);
    colnames: [string] (required);
    columns: [Column] (required);
    files: [File] (required);
    globalOffsets: [ulong] (required);
    location_prefix: string;
}
```
Rename RForest to RNTuple #3839

Axel-Naumann merged 21 commits into root-project/master from jblomer/rename-forest-ntuple 25 days ago

jblomer commented 29 days ago

No description provided.

jblomer added some commits 29 days ago

- [forest -> ntuple] rename source files
- [forest -> ntuple] fix file header comments
- [forest -> ntuple] fix ingroup comment field
- [ntuple] rename RForestEntry --> REntry
- [ntuple] rename RForestView --> RNTupleView
- [ntuple] rename RForestDescriptor --> RNTupleDescriptor
- [ntuple] rename RForestOS --> RNTupleOS
- [ntuple] rename RForestModel --> RNTupleModel
- [ntuple] rename RForest to RNTuple in README and user-visible messages
- [ntuple] rename RForest --> RNTuple
- [ntuple] rename RForestStructure --> RNTupleStructure
Conclusion: just trying to keep all these fires burning
1. Create an environment in which particle physics analysis is convenient in open-ecosystem Python (i.e. easily mix with industry-standard tools).

2. Ensure that this is not a performance penalty by encouraging columnar processing, JIT-compilation, and precompiled kernels.

3. Reduce complexity with optional high-level layers (e.g. declarative languages).

4. Integrate these with scalable systems: eventually want simple, fast, Spark-like analysis.

... and death to private skims!