BulkIO $\rightarrow$ Numpy, progress and performance

Jim Pivarski

Princeton University – DIANA

June 23, 2017

Purpose of this project

To give native Numpy support to ROOT.

Potential aspects:

1. TTree branches → Numpy arrays.
2. Numpy arrays → TTree branches.
3. PyROOT `ROOT.std.vector` (etc.) → Numpy.

This talk addresses only #1, but the others aren't off the table.

Why? Isn't there a root_numpy?

root_numpy is an external project that uses Cython and TTreeFormula to fill Numpy arrays.

We want Numpy support...

- to be a part of ROOT (to streamline interaction with machine learning libraries, for instance),
- without unnecessary dependencies (Numpy only),
- taking advantage of ROOT internals for performance.

Why? Isn't there a root_numpy?

root_numpy is an external project that uses Cython and TTreeFormula to fill Numpy arrays.

We want Numpy support...

- to be a part of ROOT (to streamline interaction with machine learning libraries, for instance),
- without unnecessary dependencies (Numpy only),
- taking advantage of ROOT internals for performance.

In fact, this is a great application of Brian's BulkIO.

Scope of TTree branches $\rightarrow$ Numpy arrays

- A single-leaf branch becomes a Numpy array.

Scope of TTree branches → Numpy arrays

- A single-leaf branch becomes a Numpy array.
  - Multidimensional leaves (with "`[...]`" in the title) affect the *shape* (dimensionality) of the array— one-to-one with TLeaf.

Scope of TTree branches → Numpy arrays

- A single-leaf branch becomes a Numpy array.
  - Multidimensional leaves (with "`[...]`" in the title) affect the *shape* (dimensionality) of the array— one-to-one with TLeaf.
  - Variable length leaves (with "`[counter]`" in the title) require the counter to be read and maybe returned to the user.

Scope of TTree branches → Numpy arrays

- A single-leaf branch becomes a Numpy array.
  - Multidimensional leaves (with “`[...]`” in the title) affect the *shape* (dimensionality) of the array— one-to-one with TLeaf.
  - Variable length leaves (with “`[counter]`” in the title) require the counter to be read and maybe returned to the user.
- A “leaf-list” branch becomes a Numpy *record array* (like an array of C structs: row-wise, can't have different lengths, can manually set the byte offsets).

Scope of TTree branches → Numpy arrays

- A single-leaf branch becomes a Numpy array.
  - Multidimensional leaves (with "`[...]`" in the title) affect the *shape* (dimensionality) of the array— one-to-one with TLeaf.
  - Variable length leaves (with "`[counter]`" in the title) require the counter to be read and maybe returned to the user.
- A "leaf-list" branch becomes a Numpy *record array* (like an array of C structs: row-wise, can't have different lengths, can manually set the byte offsets).
- A branch with subbranches becomes a Python dictionary of Numpy arrays.

Scope of TTree branches → Numpy arrays

- A single-leaf branch becomes a Numpy array.
  - Multidimensional leaves (with "`[...]`" in the title) affect the *shape* (dimensionality) of the array— one-to-one with TLeaf.
  - Variable length leaves (with "`[counter]`" in the title) require the counter to be read and maybe returned to the user.
- A "leaf-list" branch becomes a Numpy *record array* (like an array of C structs: row-wise, can't have different lengths, can manually set the byte offsets).
- A branch with subbranches becomes a Python dictionary of Numpy arrays.
- No attempt to reconstruct objects from the branch data; I have a separate project to do this in Python.

Implemented interface (low level)

```
ROOT._numpyinterface.iterate(*branches,
                             return_new_buffers=True,
                             swap_bytes=True)
```

- Returns an iterator over clusters, yielding
  `(entry_start, entry_end, array, array, array...)`
  for each cluster.

Implemented interface (low level)

```
ROOT._numpyinterface.iterate(*branches,
                             return_new_buffers=True,
                             swap_bytes=True)
```

- Returns an iterator over clusters, yielding `(entry_start, entry_end, array, array, array...)` for each cluster.
- `return_new_buffers` determines whether arrays should be read-only views of ROOT's internal data or copies. Default is to copy to discourage accidental abuse.

Implemented interface (low level)

```
ROOT._numpyinterface.iterate(*branches,
                             return_new_buffers=True,
                             swap_bytes=True)
```

- Returns an iterator over clusters, yielding
  `(entry_start, entry_end, array, array, array...)`
  for each cluster.
- `return_new_buffers` determines whether arrays should be read-only views of ROOT's internal data or copies. Default is to copy to discourage accidental abuse.
- If all baskets align per cluster, zero-copy is possible. Otherwise, we need to double-buffer to match entry ranges.

Implemented interface (low level)

```
ROOT._numpyinterface.iterate(*branches,
                             return_new_buffers=True,
                             swap_bytes=True)
```

- Returns an iterator over clusters, yielding `(entry_start, entry_end, array, array, array...)` for each cluster.
- `return_new_buffers` determines whether arrays should be read-only views of ROOT's internal data or copies. Default is to copy to discourage accidental abuse.
- If all baskets align per cluster, zero-copy is possible. Otherwise, we need to double-buffer to match entry ranges.
- `swap_bytes` transforms to little endian; in either case, the correct Numpy flag is set.

Implemented interface (low level)

```
ROOT._numpyinterface.dtypeshape(*branches,
                                swap_bytes=True)
```

- Just get the types and lengths and do not iterate.
- Useful for setting up allocate-then-fill with the iterator.

```
ROOT._numpyinterface.performance()
```

- Get a dictionary of performance counters, to aid performance-debugging without recompiling.

Implemented interface (high level)

```
ROOT.numpyinterface.arraydict(*branches,
    allocate = lambda shape, dtype:
                     numpy.empty(shape, dtype=dtype),
    trim = lambda array, length: array[:length],
    swap_bytes = True)
```

- High-level interface to filling arrays with overridable allocators.
- Have to trim `dtypeshape`'s overestimate.

Implemented interface (high level)

```
ROOT.numpyinterface.arraydict(*branches,
    allocate = lambda shape, dtype:
                    numpy.empty(shape, dtype=dtype),
    trim = lambda array, length: array[:length],
    swap_bytes = True)
```

- High-level interface to filling arrays with overridable allocators.
- Have to trim `dtypeshape`'s overestimate.

```
ROOT.numpyinterface.recarray(*branches,
                             swap_bytes = True)
ROOT.numpyinterface.iterate_pandas(*branches)
ROOT.numpyinterface.pandas(*branches)
```

- Maybe also PyTables (for HDF5), etc.
- All implemented in Python for import-flexibility.

Performance measurements

Performance measurements

Test file: flat ntuple of `px`, `py`, `pz`, `mass` for 751 919 dimuons.

Performance measurements

Test file: flat ntuple of `px`, `py`, `pz`, `mass` for 751 919 dimuons.

`px`, `py`, and `pz` are basket-aligned, but `mass` is not. Thus,

$$p = \sqrt{{p_x}^2 + {p_y}^2 + {p_z}^2}$$

doesn't involve any double-buffering but the following does:

$$E = \sqrt{{p_x}^2 + {p_y}^2 + {p_z}^2 + m^2}$$

Performance measurements

Test file: flat ntuple of `px`, `py`, `pz`, `mass` for 751 919 dimuons.

`px`, `py`, and `pz` are basket-aligned, but `mass` is not. Thus,

$$p = \sqrt{{p_x}^2 + {p_y}^2 + {p_z}^2}$$

doesn't involve any double-buffering but the following does:

$$E = \sqrt{{p_x}^2 + {p_y}^2 + {p_z}^2 + m^2}$$

Three compression cases:

- uncompressed
- LZ4 level 7 (future default); this file doesn't gain much from compression (1.0), but it is in the headers
- deflate level 1 (old default); still not much advantage (1.07)

Performance measurements

Numpy: each step— squaring, adding, square root— creates intermediate arrays; calculations performed one column at a time in precompiled code.

Numba: Python code is JIT-compiled with LLVM, basically what one would do in C, but with Python syntax.

view/copy: compare direct views of internal ROOT data with making intermediate copies.

root_numpy: calls TTreeFormula to fill an array, then do Numpy method.

SetBranchAddress: the traditional method, entirely in C++.

TTreeReader: the ROOT 6 method, entirely in C++.

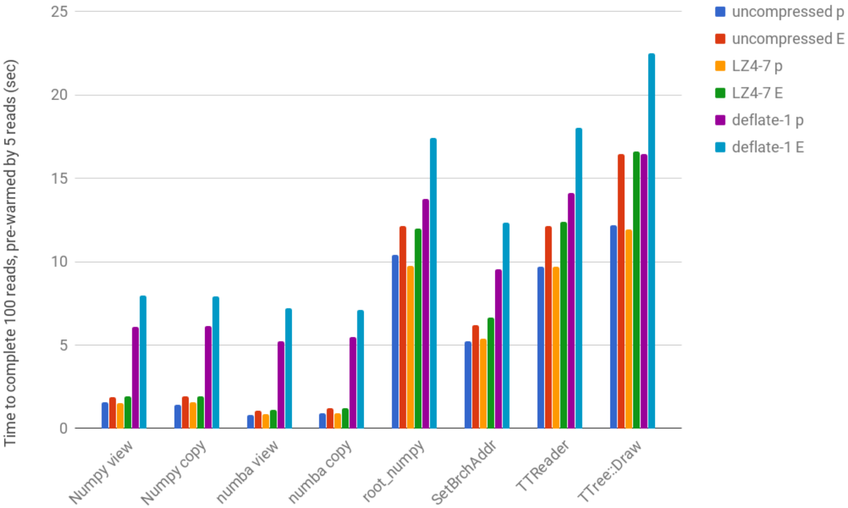
TTree::Draw: use TTree's histogram-filling method.

BulkIO in C++: **not tested**, couldn't get it to work (yet).

TDataFrame: **not tested**

Performance measurements

(Lower is better.)



Conclusions

- BulkIO is $\sim 5\times$ faster than SetBranchAddress
- At this new rate, decompression is a bottleneck but LZ4 handles poorly compressed data gracefully.
- Number of memory copies is not as relevant:
  - view vs. copy does not show much difference (15%)
  - Numpy makes many copies and is only $\sim 2\times$ worse
- Not shown here, but byte-swapping has negligible effect.

Status and next steps

- I need to handle variable-length branches, add a formal test suite, and handle all the cases on page 10.
- Functions currently take filePath, treePath, *branches as arguments, should accept PyROOT TBranches!
- Should be integrated into PyROOT in general.

  Could someone help me with that? It could be the way I get introduced to the internals of PyROOT.
- Should be integrated into the standard ROOT build system, should be code-reviewed, agree on name and style conventions (remembering that this is for use in Python).
- Aiming for ROOT 6.12 in December.