Updates on ROOT I/O

Brian Bockelman

14 October 2018

Today’s Updates

- Activities around ROOT I/O funded by DIANA/HEP:
  - Bulk I/O and odds-’n’-ends improvements. (This presentation)
  - Progress on compression algorithms.
  - I/O for uproot.

Improving I/O

- ROOT strives to maintain backward compatibility across decades.
  - Forward compatibility - ability to read files created in new versions of ROOT with old versions of ROOT - is also desired.
  - Forward compatibility is more difficult: very limiting on the file format changes. **Cannot make any changes older code cannot handle**!
    - No hard policy of when forward-compatibility breaks are allowed: evaluated on a case-by-case basis.
- ROOT 6.12 introduced “TIOFeatures”:
  - Bitset indicating file-format features used that cause forward-compatibility breaks.
  - Users can control which of these features are in use, avoiding forward compatibility breaks.
    - All the features we are considering are disabled by default - users must explicitly opt-in.
  - If ROOT encounters an unknown feature, it will return a clear error message instead of crashing or — worse! — returning incorrect data.
  - Versions prior to 6.12 will fail to read the file but will encounter a lousy error message.
- **This provides us with much more freedom to experiment with the file format than in the past!**

Offset Arrays

- Each time you call TTree::Fill, ROOT will serialize the current C++ object to serialized bytes in a buffer.
  - The memory buffer, when full, is compressed and written to disk.
  - The event range of serialized objects in a single branch is a basket.
- If the objects in the branch can be variable-sized, ROOT will save the "offset array" - value of entry N in the offset array is the location of event N's data in the basket.
  - Allows random access of events in the basket — once decompression is done!

Introduced in 6.12

Offset Arrays

- Offset arrays are useful and are minimal cost… usually.
- For “simple” objects - C-style arrays of fixed-style objects - they duplicate other information:
  - The number of entries in the array are often stored as a separate branch; **we can regenerate the offset array from that branch!**
  - Often, array sizes are small enough that the offset array is a significant portion of the total data size.
- If we cannot regenerate offset arrays, we convert them to “skip arrays” before writing: much higher compression levels!
  - On read, these are converted back to offset arrays.
- These offset array changes were the first use of the new ‘I/O features’ forward compatibility breaking mechanism.

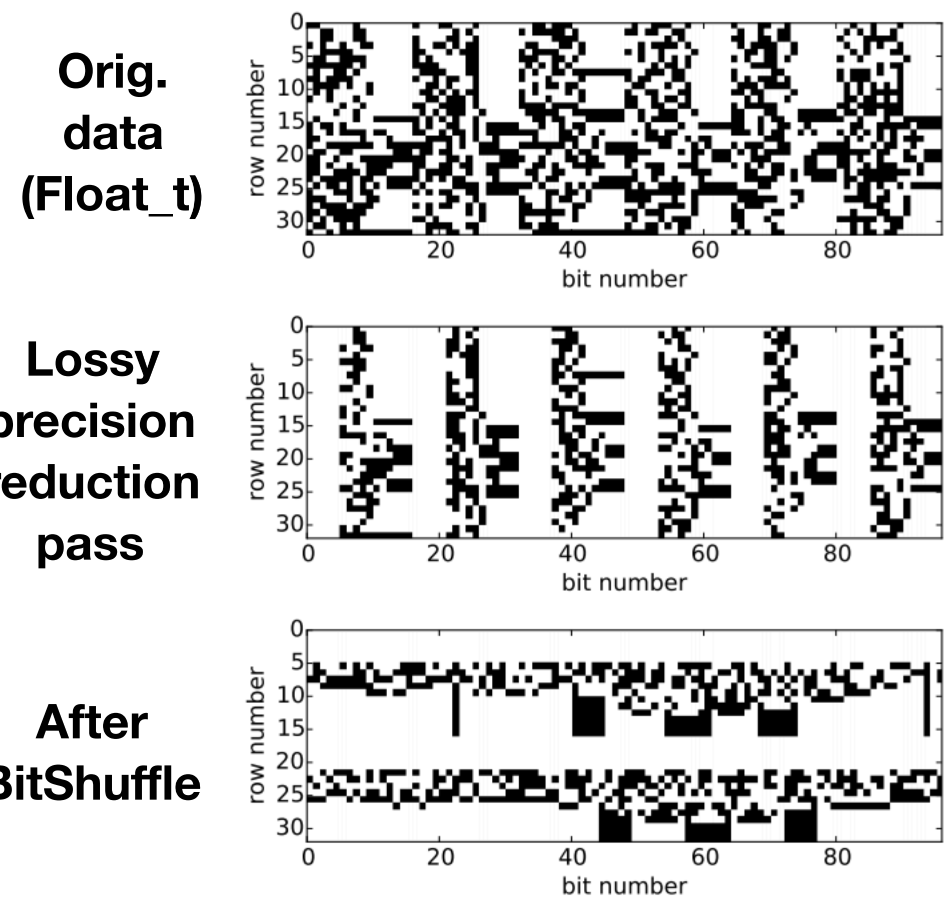
Future Compression Improvements

- Offset arrays compress surprisingly poorly, even when they are simple arithmetic sequences:
  - Any human can express this sequence fairly compactly: 0, 4, 8, 12, 16, 20, 24, …
  - As there is no repetition, some compression approaches (e.g., LZ4) will treat the data as incompressible. Others (LZMA) burn significant CPU to determine a good strategy.
- The skip arrays (4, 4, 4, 4, 4, 4, 4) compress much better across the board - and it's a simple transform we can apply if we know about the specific structure.
- **Observation**: we can use our pre-existing knowledge about the data structure to apply simple transforms to our data.

Future Compression Improvements

- **Example**: BitShuffle filter. Instead of writing sequences of integers in network-byte-order, arrange integers into groups of 2,000.
  - Write out 2,000 most-significant-bits, then write 2,000 second-most-significant-bits, …
  - Simple transform that groups high-order bits together: as these are '0' in 99.9% of cases, they compress well.

Figure from BitShuffle paper



(Data fed to compression algorithm by reading out rows.)

Idea put forward in http://dx.doi.org/10.1016/j.ascom.2015.07.002 by Masui et al

Future Compression Improvements

- ROOT has block-buffer-based compression:
  - By default, we create a new block in the basket every 16MB of uncompressed data. *We can decrease block size to increase parallelism if we want*.
  - Each block has a header indicating the compression algorithm used.
- **Work-in-progress**: Improve bit-packing layout of header to allow it to specify the algorithm *and* the sequence of filters applied (such as BitShuffle!).
- Adding BitShuffle filter results in ~20% reduction of CMS NanoAOD compressed with LZ4-6. Smaller than ZLIB-6!
  - Still not as small as NanoAOD + LZMA, but *much* faster to read!
  - CMS sees such a strong benefit because it performs a precision reduction pass to many of its floating point numbers.

Bulk I/O

- For simple objects (primitive types, arrays of primitives, record types), the cost of a ROOT library call is significantly higher than the cost of deserialization.
  - Multiple experiments have shown instead returning a large buffer of event data and processing that can be significantly faster.
  - **Most extreme case**: processing a branch consisting of a single integer can be done around 10x faster when doing I/O in bulk as opposed to a TTreeReader-based event loop.
    - More realistic cases show significant speedups (but less than the extreme case!)
- We have had a series of approaches to exposing this functionality. Current implementation version (v3) is here: https://github.com/root-project/root/pull/2519. The difficulty becomes in providing a sane interface.
  - The low-level interface simply returns a memory buffer and the number of events contained. Exercise to user to deserialize events and track usage! **Not going to fly…**
  - Experimenting with a "TTreeReaderFast" that does deserialization and usage tracking header-only. No virtual calls, everything becomes inlined. **Suffers from "yet another reading mechanism"…**
  - Can expose data directly to Python as a NumPy array. **Great, but doesn't help C++…**

RDataSource

- Current attempt is to implement a bulk “**RDataSource**” for the new RDataFrame framework inside ROOT.
- More modern design allows us to avoid significant virtualization costs. Even adds type-safety!
    - Use of JIT means that we could avoid use of virtualization even where it exists today.
- The RDataSource layer means the user does not see any of the complexities of the bulk I/O mechanism.
    - Eventually, could even fall-back to the ‘normal’ ROOT data source. The “bulk I/O speed-boost” could be automatic. *This might be a bad thing…*
- Experimental / prototype-quality code here: https://github.com/bbockelm/root/tree/rrootbulkds
- Unfortunately, RDataFrame overheads dominate. For “extreme case” (reading single integer branch of 100M events):
    - Using bulk APIs directly: **450MHz**.
    - Invoking bulk RDataSource directly: **160MHz**
    - Using RDataFrame with bulk RDataSource APIs: **42MHz**
    - Using standard RDataFrame: **14MHz**
- **Grain of salt warning**: **Investigation is early**, we may yet find obvious places to get the performance numbers to converge…

Tracking Memory Usage

- We currently optimize basket sizes according to disk layouts: balance the column-oriented layout with keeping all of one event's data in a finite range in the file.
  - E.g., if you set the cluster size to 20MB, ROOT will flush approximately every 20MB of *compressed* (file) data.
  - The flush will write all buffered data to disk; all of one event's data is going to be inside the same cluster.
  - ROOT will tune basket size so there is one basket per cluster.
- **Problem**: what about highly-compressible data?

Uncompressed Memory Usage

- Highly compressible data (>10x compression ratio) can cause the clustering algorithm to blow up memory usage.
    - 20MB of compressed data on disk - at 10x compression ratio - is 200MB of uncompressed data in memory!
- **Work-in-progress**:
    - Have ROOT more explicitly track buffer usage. Allow user to specify “target memory usage” separately from cluster size.
    - Include peak memory buffer usage in regression testing.
    - Avoid going over TTree-wide memory threshold (when possible).
        - As we don’t know how complex an object is prior to serialization, we may not be able to avoid going over the limits in all cases. For these objects, we have no way to estimate whether the object is larger than the remaining buffer space!
    - More aggressively shrink buffers when we are over-threshold.