Thoughts on Efficient & Parallel Histograms

Jonas Hahnfeld

April 11, 2024

Disclaimers

- *NOT* the new ROOT histogram package
  - Just some ideas and a prototype for my own research

Disclaimers

- *NOT* the new ROOT histogram package
  - Just some ideas and a prototype for my own research
- Interface and prototype are designed from a computer science perspective
  - Possible to implement efficiently, but may be missing features required for physics

Outline

Existing Solutions

Templates vs Run-Time Parameters

Features Required for Histograms

Comparison with Existing Solutions

Multithreaded Histograms

Conclusions

Existing Solutions

Quick Look at Existing Solutions (1/2)

- ROOT histogram package: `TH1`, `TH2`, `TH3`
  - Subclasses for bin content type: `TH1I`, `TH1F`, `TH1D`
  - Everything is a `TH1`: `TH2`, `TH3`; but a `TH3` is not a `TH2` and `TH2D` is not a `TH1D`

Quick Look at Existing Solutions (1/2)

- ROOT histogram package: TH1, TH2, TH3
  - Subclasses for bin content type: TH1I, TH1F, TH1D
  - Everything is a TH1: TH2, TH3; but a TH3 is not a TH2 and TH2D is not a TH1D
- Multi-dimensional THn and THnSparse (inheriting from THnBase)
  - With concrete subclasses THnT<T> and THnSparseT<CONT>

Quick Look at Existing Solutions (1/2)

- ROOT histogram package: `TH1`, `TH2`, `TH3`
  - Subclasses for bin content type: `TH1I`, `TH1F`, `TH1D`
  - Everything is a `TH1`: `TH2`, `TH3`; but a `TH3` is not a `TH2` and `TH2D` is not a `TH1D`
- Multi-dimensional `THn` and `THnSparse` (inheriting from `THnBase`)
  - With concrete subclasses `THnT<T>` and `THnSparseT<CONT>`
- Both use `TAxis` to store the axis configuration
  - `TAxis::FindBin` switches at run-time between fixed and variable bin sizes

Quick Look at Existing Solutions (2/2)

- Boost.Histogram (since Boost 1.70)
  - Templated on `<class Axes, class Storage>`
  - *Static histogram*: compile-time `std::tuple` of concrete axis types
  - *Semi-dynamic histogram*: variable number of axes (`std::vector`)
  - *Dynamic histograms*: `std::vector` of dynamic `axis::variant`

Quick Look at Existing Solutions (2/2)

- Boost.Histogram (since Boost 1.70)
  - Templated on `<class Axes, class Storage>`
  - *Static histogram*: compile-time `std::tuple` of concrete axis types
  - *Semi-dynamic histogram*: variable number of axes (`std::vector`)
  - *Dynamic histograms*: `std::vector` of dynamic `axis::variant`
- `ROOT::Experimental::RHist` prototype for ROOT 7
  - Templated on number of `DIMENSIONS`, `PRECISION`, and optional bin statistics
  - Has a pointer to (abstract, polymorphic) `RHistImplBase`
  - Concrete `RHistImpl` is templated on processed axis types
    - Will not work with RNTuple! (unless unsplit storage)

Templates vs Run-Time Parameters

Templates vs Run-Time Parameters

- All existing solutions statically type the bin content
  - Makes sense, knowledge about the type also essential for I/O

Templates vs Run-Time Parameters

- All existing solutions statically type the bin content
  - Makes sense, knowledge about the type also essential for I/O
- Two approaches for axis configuration in the existing solutions:
  1. Dynamic at run-time (`TH*` using `TAxis`)
  2. Via templates at compile-time (Boost.Histogram, `RHist`)
     - Possibility to have dynamic axis types, or
     - Hide the axis types via a polymorphic pointer

Understanding the Tradeoffs with Templates

- Templates and inline functions allow compiler optimizations

  → least overhead, best performance (?)

  - Can lead to very long compile times: 4 minutes for `RHist<6, int>`!

Understanding the Tradeoffs with Templates

- Templates and inline functions allow compiler optimizations
  - $\rightarrow$ least overhead, best performance (?)
  - Can lead to very long compile times: 4 minutes for `RHist<6, int>`!

- Excessive templating complicates I/O
  - At least with ROOT, need to know concrete type
  - Boost.Histogram mentions dynamic Python bindings
    - Probably less of a problem with flexibility of PyROOT / cppyy

Understanding the Tradeoffs with Templates

- Templates and inline functions allow compiler optimizations
  - $\rightarrow$ least overhead, best performance (?)
  - Can lead to very long compile times: 4 minutes for `RHist<6, int>`!
- Excessive templating complicates I/O
  - At least with ROOT, need to know concrete type
  - Boost.Histogram mentions dynamic Python bindings
    - Probably less of a problem with flexibility of PyROOT / cppyy
- ... and user code: either templated as well or very long types

Understanding the Tradeoffs with Templates

- Templates and inline functions allow compiler optimizations
  $\rightarrow$ least overhead, best performance (?)
  - Can lead to very long compile times: 4 minutes for `RHist<6, int>`!

- Excessive templating complicates I/O
  - At least with ROOT, need to know concrete type
  - Boost.Histogram mentions dynamic Python bindings
    - Probably less of a problem with flexibility of PyROOT / cppyy
- ... and user code: either templated as well or very long types

- Question: How much do we actually lose with dynamic axes configuration?
  - Which tricks can we play to recover performance?

A Most Trivial Histogram Prototype

- `EPHist<T>` templated on bin content type

A Most Trivial Histogram Prototype

- `EPHist<T>` templated on bin content type
- Implement trivial `Fill` function for one dimension of fixed bins

```
void Fill(double x) {
  std::size_t bin = (x - fLow) * fInvBinWidth;
  assert(bin >= 0 && bin < fNumBins);
  fData[bin]++;
}
```

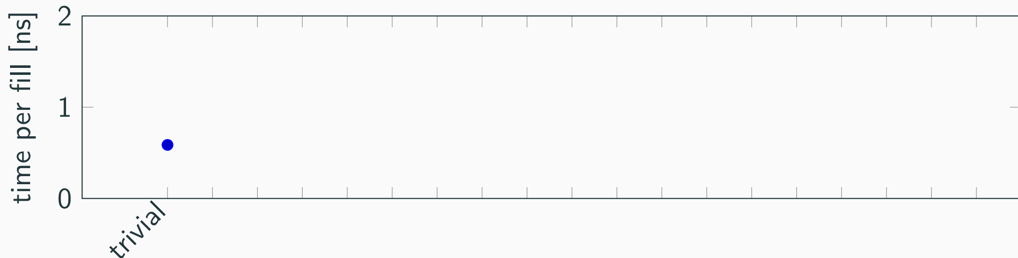
A Most Trivial Histogram Prototype

- `EPHist<T>` templated on bin content type
- Implement trivial `Fill` function for one dimension of fixed bins
- Benchmark filling 4096 uniform random values $[0, 1)$ into 20 bins

```
void Fill(double x) {
  std::size_t bin = (x - fLow) * fInvBinWidth;
  assert(bin >= 0 && bin < fNumBins);
  fData[bin]++;
}
```

A Most Trivial Histogram Prototype

- `EPHist<T>` templated on bin content type
- Implement trivial `Fill` function for one dimension of fixed bins
- Benchmark filling 4096 uniform random values $[0, 1)$ into 20 bins



Multi-Dimensional Histograms

1. Refactor bin computation into `RegularAxis` type

Multi-Dimensional Histograms

1. Refactor bin computation into `RegularAxis` type
2. Wrap `RegularAxis` in a `std::vector`

Multi-Dimensional Histograms

1. Refactor bin computation into `RegularAxis` type
2. Wrap `RegularAxis` in a `std::vector`
3. Implement multi-dimensional histograms

time per fill [ns]
2
1
0
trivial
RegularAxis
vector
multi-dim.

Multi-Dimensional Histograms

1. Refactor bin computation into `RegularAxis` type
2. Wrap `RegularAxis` in a `std::vector`
3. Implement multi-dimensional histograms

   → Add a second benchmark configuration (4096 values into 20 × 20 bins)

Multi-Dimensional Histograms

1. Refactor bin computation into `RegularAxis` type
2. Wrap `RegularAxis` in a `std::vector`
3. Implement multi-dimensional histograms

   → Add a second benchmark configuration (4096 values into 20 × 20 bins)

Different Axis Types

Different Axis Types

1. Wrap `RegularAxis` in a `std::variant`

Different Axis Types

1. Wrap `RegularAxis` in a `std::variant`
2. Implement an axis type with variable bins

time per fill [ns]
6
4
2
0
trivial
RegularAxis
vector
multi-dim.
tuple
variadic
variant
variable

Optimizing Performance: A little bit of templating

- Does it help to template only the `Fill` method?

time per fill [ns]

6
4
2
0

trivial
RegularAxis
vector
multi-dim.
tuple
variadic
variant
variable

Optimizing Performance: A little bit of templating

- Does it help to template only the `Fill` method?

time per fill [ns]

6
4
2
0

trivial
RegularAxis
vector
multi-dim.
tuple
variadic
variant
variable
Fill<A>

Optimizing Performance: Access into `std::variant` (1/2)

- `std::get` and `std::get_if` need to check what alternative is held by `variant`
  - Otherwise throw exception or return null pointer value

Optimizing Performance: Access into `std::variant` (1/2)

- `std::get` and `std::get_if` need to check what alternative is held by `variant`
  - Otherwise throw exception or return null pointer value
- In our case, content of `std::variant` is constant during `Fill`
  - For templated `Fill`, we even know what content we expect!

Optimizing Performance: Access into std::variant (1/2)

- std::get and std::get_if need to check what alternative is held by variant
  - Otherwise throw exception or return null pointer value
- In our case, content of std::variant is constant during Fill
  - For templated Fill, we even know what content we expect!
- Implementations in libstdc++ and libc++ put union as first member
  - Can just reinterpret_cast a pointer to the std::variant
  - (in the constructor, check once that std::get_if returns the same pointer)

Optimizing Performance: Access into `std::variant` (1/2)

time per fill [ns]

6
4
2
0

trivial
RegularAxis
vector
multi-dim.
tuple
variadic
variant
variable
Fill<A>

Optimizing Performance: Access into `std::variant` (1/2)

Optimizing Performance: Access into `std::variant` (2/2)

- Better solution: cache the pointer returned by `std::get_if`

time per fill [ns]

6
4
2
0

trivial
RegularAxis
vector
multi-dim.
tuple
variadic
variant
variable
Fill<A>
cast

Optimizing Performance: Access into `std::variant` (2/2)

- Better solution: cache the pointer returned by `std::get_if`

- When aligning the axis types to 8 bytes, the pointers end in `0x0` or `0x8`
  - Lower three bits are available (standard trick in low-level optimizations)
  - Can be used to encode the value of `index()`

time per fill [ns]

6
4
2
0

trivial
RegularAxis
vector
multi-dim.
tuple
variadic
variant
variable
Fill<A>
cast

Optimizing Performance: Access into `std::variant` (2/2)

- Better solution: cache the pointer returned by `std::get_if`

- When aligning the axis types to 8 bytes, the pointers end in `0x0` or `0x8`
  - Lower three bits are available (standard trick in low-level optimizations)
  - Can be used to encode the value of `index()`

time per fill [ns]

0 2 4 6

trivial RegularAxis vector multi-dim. tuple variadic variant variable `Fill<A>` cast pointers

Features Required for Histograms

Adding Checks for `Fill` Arguments and Axis Types

time per fill [ns]

0 2 4 6

trivial
RegularAxis
vector
multi-dim.
tuple
variadic
variant
variable
Fill<A>
cast
pointers

Adding Checks for `Fill` Arguments and Axis Types

- Number of arguments must match dimension of histogram

Adding Checks for `Fill` Arguments and Axis Types

- Number of arguments must match dimension of histogram
- Axis types in templated `Fill` must match

time per fill [ns]

0 2 4 6

trivial
RegularAxis
vector
multi-dim.
tuple
variadic
variant
variable
Fill<A>
cast
pointers
arguments
axis types

Adding More Functions

- Constructor to allow “mixed” histograms (regular and variable bin axes)

Adding More Functions

- Constructor to allow “mixed” histograms (regular and variable bin axes)

- `Add()` to merge two histograms, explicit `Clone()` (instead of copy constructor)

Adding More Functions

- Constructor to allow “mixed” histograms (regular and variable bin axes)
- `Add()` to merge two histograms, explicit `Clone()` (instead of copy constructor)
- Weighted `Fills`
  - Only allowed for floating point bins (`static_assert`)

Adding More Functions

- Constructor to allow “mixed” histograms (regular and variable bin axes)
- `Add()` to merge two histograms, explicit `Clone()` (instead of copy constructor)
- Weighted `Fills`
  - Only allowed for floating point bins (`static_assert`)
- Bin content errors: `DoubleBinWithError`
  - Corollary: cannot merge `EPHist<double>` and `EPHist<DoubleBinWithError>`
  - Maybe this is good? If needed, have to / can define explicit semantics

Adding More Functions

time per fill [ns]

0
2
4
6

trivial
RegularAxis
vector
multi-dim.
tuple
variadic
variant
variable
Fill<A>
cast
pointers
arguments
axis types
mixed
Add()
Clone()
weighted
error

Underflow and Overflow Bins

- Underflow and overflow bins are essential
  - When projecting a dimension, want to retain full information

time per fill [ns]

0
2
4
6

trivial
RegularAxis
vector
multi-dim.
tuple
variadic
variant
variable
Fill<A>
cast
pointers
arguments
axis types
mixed
Add()
Clone()
weighted
error

Underflow and Overflow Bins

- Underflow and overflow bins are essential
  - When projecting a dimension, want to retain full information

- Unfortunately implementation costs some performance…

time per fill [ns]

0 2 4 6

trivial RegularAxis vector multi-dim. tuple variadic variant variable Fill<A> cast pointers # arguments axis types mixed Add() Clone() weighted error overflow

Data Export to PGFPlots

```
namespace EPHist {
namespace Util {

// PGFPlots
void ExportPGFPlotsData(const EPHistForExport &h, std::ostream &os);
template <typename T>
void ExportPGFPlotsData(const EPHist<T> &h, std::ostream &os) {
  ExportPGFPlotsData(EPHistForExportT<T>(h), os);
}

} // namespace Util
} // namespace EPHist
```

```
namespace
namespace

// PGFPlo
void Expo                                                       m &os);
template <
void Expo                                                            {
  ExportP(
}

} // name
} // name
```

Dimuon mass

CMS Open Data $\sqrt{s} = 8$ TeV, $L_{int} = 11.6$ fb^{-1}

N_{Events}: 10^0, 10^1, 10^2, 10^3, 10^4, 10^5

10^0, 10^1, 10^2

$m_{\mu\mu}$ (GeV)

Conversion to ROOT Histograms

```
namespace EPHist {
namespace Util {

std::unique_ptr<TH1I> ConvertToTH1I(const EPHist<int> &h);
// std::unique_ptr<TH2I> ConvertToTH2I(const EPHist<int> &h);
// std::unique_ptr<THn> ConvertToTHnI(const EPHist<int> &h);

} // namespace Util
} // namespace EPHist
```

```
namespace
namespace

std::uniq
// std::u
// std::u

} // name.
} // name.
```

Dimuon mass

CMS Open Data $\sqrt{s}$ = 8 TeV, L_{int} = 11.6 fb^{-1}

N_{Events}

10^5, 10^4, 10^3, 10^2, 10, 1

η ρ,ω ϕ J/ψ ψ' Υ(1,2,3S) Z

1, 10, 10^2

$m_{\mu\mu}$ (GeV)

Missing Features

- Categorical axis type
- Profile histograms (mean of another value per bin)

Missing Features

- Categorical axis type
- Profile histograms (mean of another value per bin)

- UHI: Unified / Universal Histogram Interface

Missing Features

- Categorical axis type
- Profile histograms (mean of another value per bin)

- UHI: Unified / Universal Histogram Interface

- Export to other data formats (?)
- Complete conversion functions to ROOT histograms

Missing Features

- Categorical axis type
- Profile histograms (mean of another value per bin)

- UHI: Unified / Universal Histogram Interface

- Export to other data formats (?)
- Complete conversion functions to ROOT histograms
- Conversion to other existing solutions?

Missing Features

- Categorical axis type
- Profile histograms (mean of another value per bin)

- UHI: Unified / Universal Histogram Interface

- Export to other data formats (?)
- Complete conversion functions to ROOT histograms
- Conversion to other existing solutions?

- Axis range deduction, growing axes?

A Note on Sparse Histograms

- Boost.Histogram is templated on `Storage`, which can be exchanged
  - Can implement dense or sparse strategies (e.g. `std::unordered_map`)

A Note on Sparse Histograms

- Boost.Histogram is templated on `Storage`, which can be exchanged
  - Can implement dense or sparse strategies (e.g. `std::unordered_map`)

- `THnSparse` stores pairs of coordinates and bin contents

A Note on Sparse Histograms

- Boost.Histogram is templated on `Storage`, which can be exchanged
  - Can implement dense or sparse strategies (e.g. `std::unordered_map`)
- `THnSparse` stores pairs of coordinates and bin contents
- Could also allocate chunks of bins and perform direct mapping

A Note on Sparse Histograms

- Boost.Histogram is templated on `Storage`, which can be exchanged
  - Can implement dense or sparse strategies (e.g. `std::unordered_map`)
- `THnSparse` stores pairs of coordinates and bin contents
- Could also allocate chunks of bins and perform direct mapping
- In any case, sparse strategy *requires* optional memory allocation during `Fill`
  - Pessimizes compiler optimizations because of external call
  - Even if possible, not sure if it should be implemented in the same class

Comparison with Existing Solutions

Comparison with Existing Solutions

- Benchmark filling 4096 uniform random values $[0, 1)$
  - On one core of AMD Ryzen 9 3900, 3.1 GHz

Comparison with Existing Solutions

- Benchmark filling 4096 uniform random values $[0, 1)$
  - On one core of AMD Ryzen 9 3900, 3.1 GHz
- Test 1D, 2D, 3D, and 6D histograms from ROOT (`master`) and Boost (1.84.0)
  - 20 bins in each dimension, no `Clear()` between iterations
  - For 1D and 2D: more entries than bins
  - For 3D and 6D: less entries than bins ($\rightarrow$ used bins in cache!)

Comparison with Existing Solutions

- Benchmark filling 4096 uniform random values $[0, 1)$
  - On one core of AMD Ryzen 9 3900, 3.1 GHz
- Test 1D, 2D, 3D, and 6D histograms from ROOT (`master`) and Boost (1.84.0)
  - 20 bins in each dimension, no `Clear()` between iterations
  - For 1D and 2D: more entries than bins
  - For 3D and 6D: less entries than bins ($\rightarrow$ used bins in cache!)
- Keep in mind: microbenchmarks *DO NOT* reflect application performance!

One and Two Dimensions

time per fill [ns]

15

10

5

0

1D

2D

More Dimensions: 3D and 6D

time per fill [ns]

40
30
20
10
0

3D
6D

Multithreaded Histograms

Thoughts on Multithreaded Histograms

- With Boost.Histogram, can use a `std::atomic` bin content type
  - Allows multithreaded filling of a single histogram
  - Works fine as long as single bins are not “too contended”

Thoughts on Multithreaded Histograms

- With Boost.Histogram, can use a `std::atomic` bin content type
  - Allows multithreaded filling of a single histogram
  - Works fine as long as single bins are not "too contended"
- However changes the type of the histogram

  → Requires to use atomic instructions even for sequential operations

Thoughts on Multithreaded Histograms

- With Boost.Histogram, can use a `std::atomic` bin content type
  - Allows multithreaded filling of a single histogram
  - Works fine as long as single bins are not "too contended"
- However changes the type of the histogram
  - $\rightarrow$ Requires to use atomic instructions even for sequential operations
- Ideally we want the template parameter to be the *storage* type
  - A separate interface (`FillAtomic`) internally implements atomic operations

Implementing `FillAtomic`

- With C++20 we have `std::atomic_ref`
  - It "applies atomic operations to the object it references."

Implementing `FillAtomic`

- With C++20 we have `std::atomic_ref`
  - It “applies atomic operations to the object it references.”
- We can implement it ourselves for our limited use cases:
  - Increment operation on integer scalars (with `__atomic` built-in functions)
  - Compare-exchange loop for floating point values (using another built-in function)
  - Composite types (e.g. `DoubleBinWithError`) can be implemented in terms of those

Multithreaded Filling of Sparse Histograms?

- There are concurrent hash maps that could be used for sparse storage
  - Combined with atomic operations described on the previous slide

Multithreaded Filling of Sparse Histograms?

- There are concurrent hash maps that could be used for sparse storage
  - Combined with atomic operations described on the previous slide
- For chunked storage, could have a list of atomic pointers
  - If allocation is required, atomically exchange new chunk

My Research: Optimized Parallel Filling

- How to deal with contended bins?

My Research: Optimized Parallel Filling

- How to deal with contended bins?
- Options include:
  - Duplicate histograms and merge in the end
  - Filling with atomic operations described in the previous slides
  - Having a thread-local buffers to coalesce writes
  - Some sort of queuing mechanism?

My Research: Optimized Parallel Filling

- How to deal with contended bins?
- Options include:
  - Duplicate histograms and merge in the end
  - Filling with atomic operations described in the previous slides
  - Having a thread-local buffers to coalesce writes
  - Some sort of queuing mechanism?
- Ideally want an interface that automatically chooses the best strategy

Conclusions

Conclusions and Takeaways

- Prototype of an efficient histogram class
  - Templated on the bin content type, otherwise run-time parameters
  - Faster than existing solutions with compile-time axis configuration

Conclusions and Takeaways

- Prototype of an efficient histogram class
  - Templated on the bin content type, otherwise run-time parameters
  - Faster than existing solutions with compile-time axis configuration

- Thoughts for parallel interfaces
  - Atomic operations without changing the storage type
  - Ideas for sparse histograms & optimized parallel filling

Conclusions and Takeaways

- Prototype of an efficient histogram class
  - Templated on the bin content type, otherwise run-time parameters
  - Faster than existing solutions with compile-time axis configuration
- Thoughts for parallel interfaces
  - Atomic operations without changing the storage type
  - Ideas for sparse histograms & optimized parallel filling
- Prototype code available on GitHub: `https://github.com/hahnjo/EPHist`