# Modernisation of RooFit

S. Hageboeck (CERN, EP-SFT) for the ROOT team

# Introduction

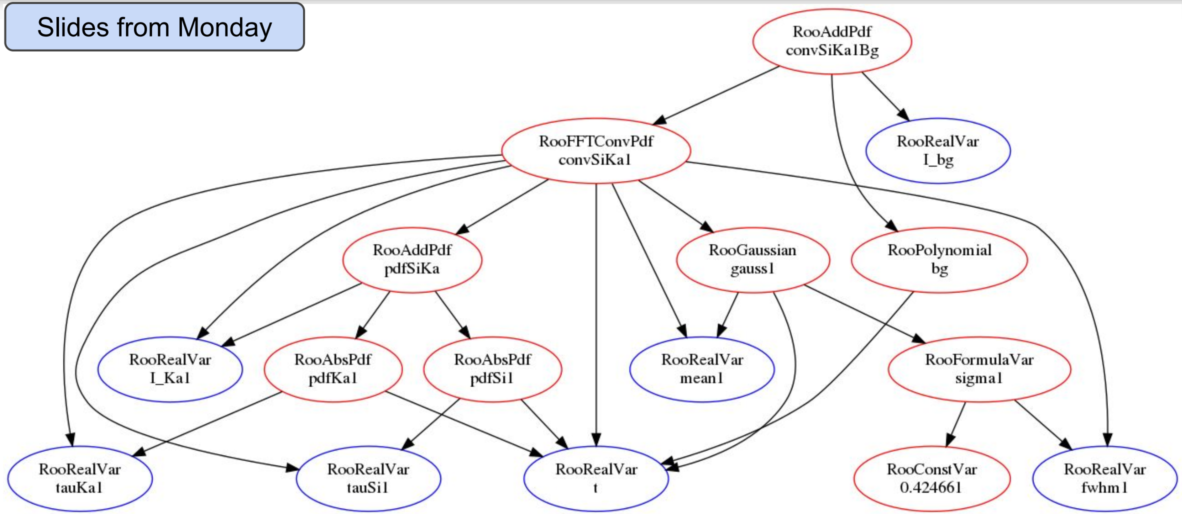
Slides from Monday

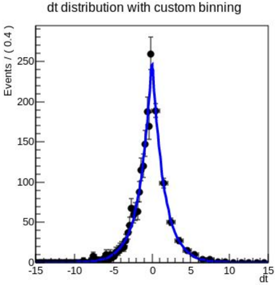
- RooFit used in all LHC (+ other) experiments
  - Express statistical models (binned / unbinned likelihoods)
  - **Parameter estimation** (i.e. errors!)
  - **Statistical tests** (e.g. Higgs Discovery)
- Development started before ~2005 until ~2011, not touched much in recent years

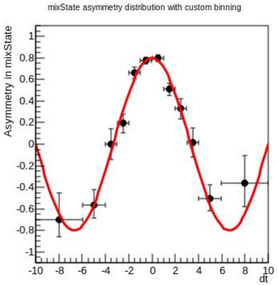
- **Challenges**: Data statistics in LHC's Run 3
  - More events to be processed (*e.g.* LHCb: ~10x more)
  - Higher statistics → allow for more complex models
  - Goal: speed up >= 10x

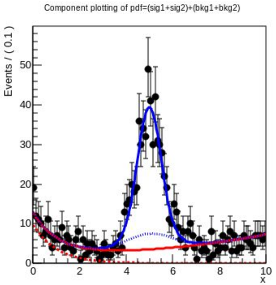
# RooFit's Strengths

Slides from Monday


RooAddPdf convSiKa1Bg
RooFFTConvPdf convSiKa1
RooRealVar I_bg
RooAddPdf pdfSiKa
RooGaussian gauss1
RooPolynomial bg
RooRealVar I_Ka1
RooAbsPdf pdfKa1
RooAbsPdf pdfSi1
RooRealVar mean1
RooFormulaVar sigma1
RooRealVar tauKa1
RooRealVar tauSi1
RooRealVar t
RooConstVar 0.424661
RooRealVar fwhm1


dt distribution with custom binning


mixState asymmetry distribution with custom binning


Component plotting of pdf=(sig1+sig2)+(bkg1+bkg2)

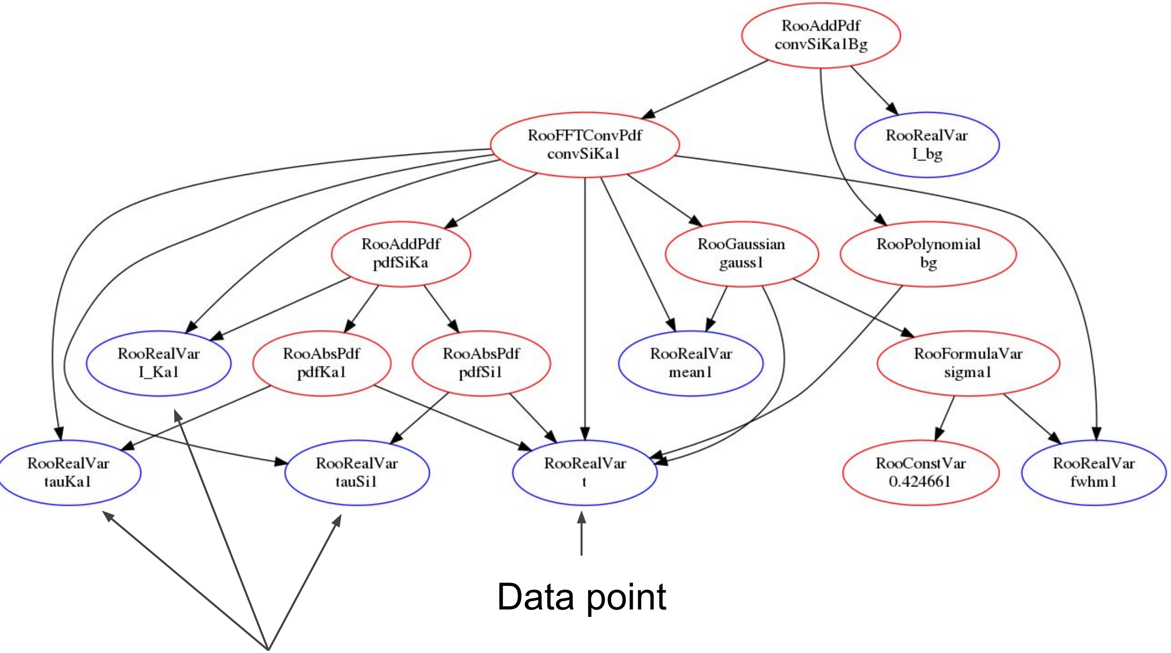
- Compose PDFs as trees of functions & variables

  RooFit classes can be stitched together to evaluate complex functions
- Each PDF can be:
  - evaluated
  - normalised
  - fitted to data
  - plotted
  - Parameter estimation
  - Toy experiments
  - …

# RooFit's Weakness

Slides from Monday

A random PDF
from a question in the forum



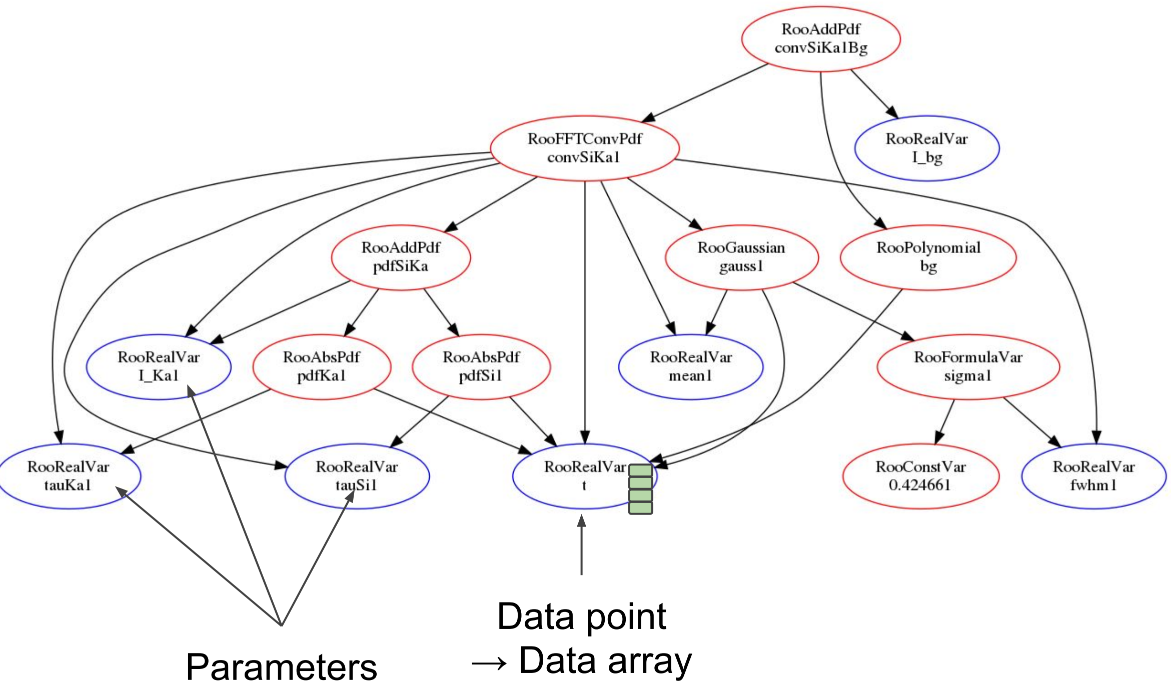
Likelihood:
Probability of observing the data given a probability model

Maximum-likelihood fit:

- Adjust parameters until likelihood maximal
- **One virtual call per**:
  - Data point
  - PDF node
  - Set of parameters tested
- Large fit: 1M data points * 1000 elements * 1000 fit steps
  = 1 trillion calls
- + 1 billion normalisation integrals when parameters change

# Batched function evaluations

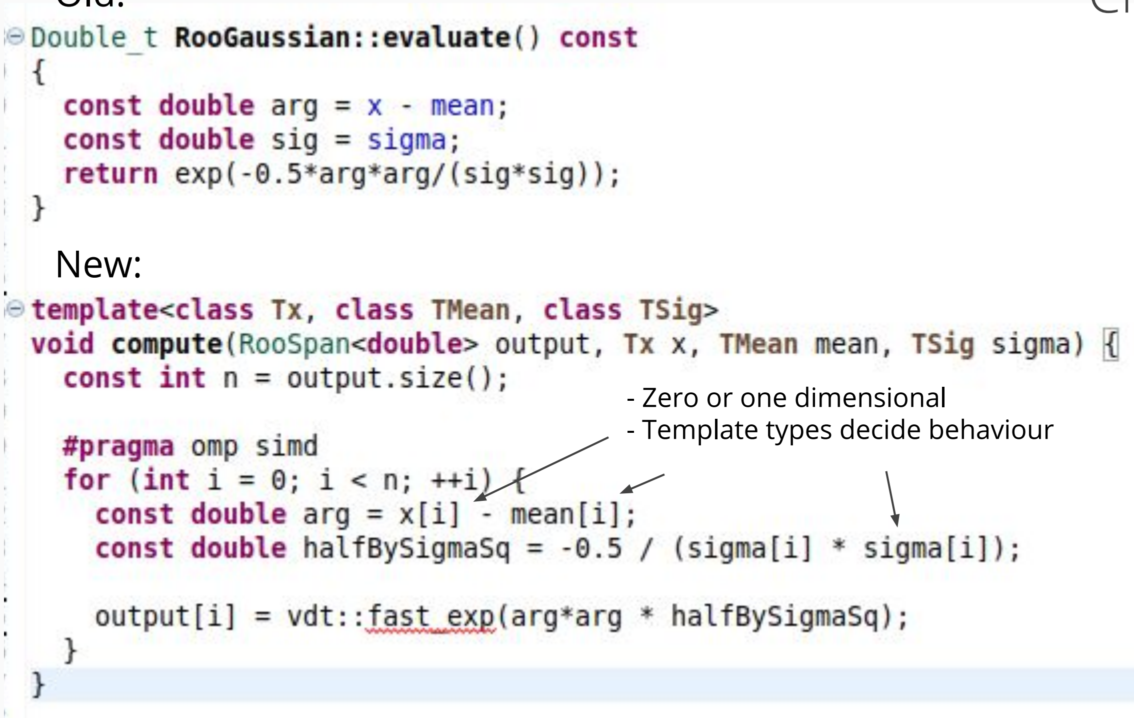
A random PDF
from a question in the forum



- Previously: A single data point is loaded into the variables
- The whole (minus cached branches) expression tree is walked over
- Execution returns to the data point, cache line disappeared
  - Simple profiling: 50% L3 misses
- 0 chance to vectorise computations
- My plan:
  - Evaluate a batch of data points in a single call
  - Exploit vectorised fp instructions

# Batched and Auto-Vectorised Gaussian

Old:



```
Double_t RooGaussian::evaluate() const
{
  const double arg = x - mean;
  const double sig = sigma;
  return exp(-0.5*arg*arg/(sig*sig));
}
```

New:

```
template<class Tx, class TMean, class TSig>
void compute(RooSpan<double> output, Tx x, TMean mean, TSig sigma) {
  const int n = output.size();

  #pragma omp simd
  for (int i = 0; i < n; ++i) {
    const double arg = x[i] - mean[i];
    const double halfBySigmaSq = -0.5 / (sigma[i] * sigma[i]);

    output[i] = vdt::fast_exp(arg*arg * halfBySigmaSq);
  }
}
```

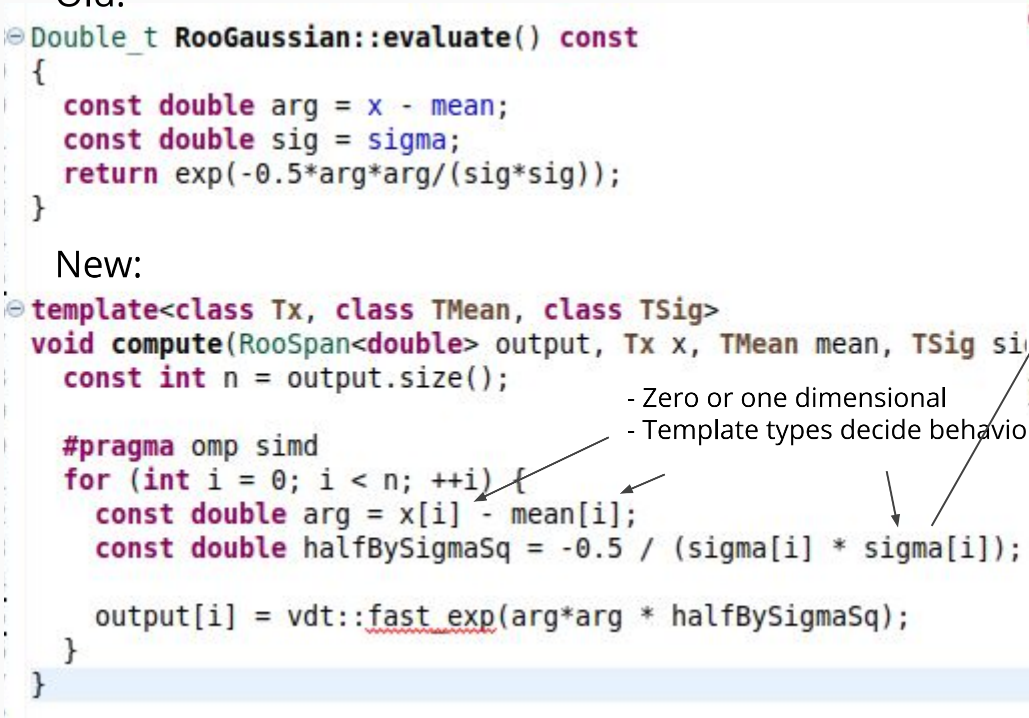
- Zero or one dimensional
- Template types decide behaviour

Challenge:

- ○ Whether a node is a parameter or a batch is decided at run time (might even change at RT)
- ○ Solved with classes that either collapse to a constant or an array (completely inlinable)
- ○ VDT math functions for auto vectorisation

# Batched and Auto-Vectorised Gaussian

Old:

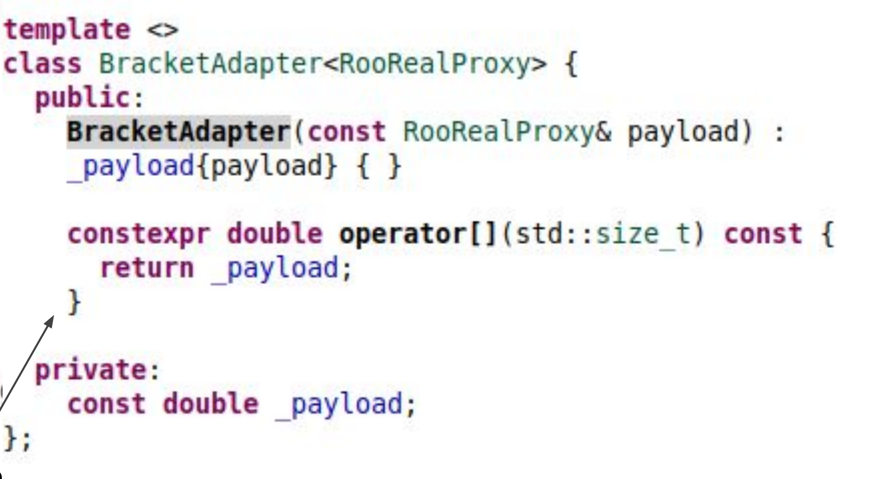


```
Double_t RooGaussian::evaluate() const
{
  const double arg = x - mean;
  const double sig = sigma;
  return exp(-0.5*arg*arg/(sig*sig));
}
```

New:

```
template<class Tx, class TMean, class TSig>
void compute(RooSpan<double> output, Tx x, TMean mean, TSig si
  const int n = output.size();

  #pragma omp simd
  for (int i = 0; i < n; ++i) {
    const double arg = x[i] - mean[i];
    const double halfBySigmaSq = -0.5 / (sigma[i] * sigma[i]);

    output[i] = vdt::fast_exp(arg*arg * halfBySigmaSq);
  }
}
```

- Zero or one dimensional
- Template types decide behavio



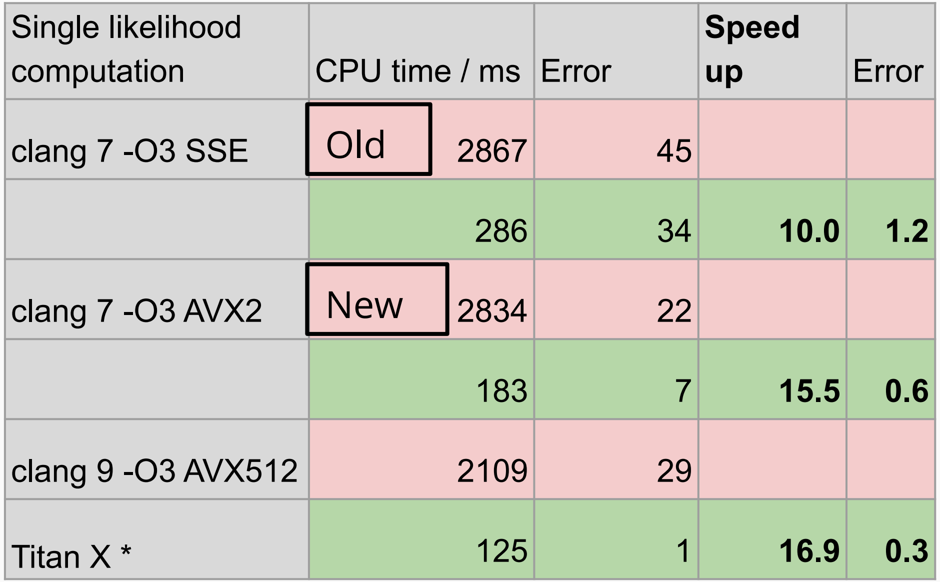
```
template <>
class BracketAdapter<RooRealProxy> {
  public:
    BracketAdapter(const RooRealProxy& payload) :
    _payload{payload} { }

    constexpr double operator[](std::size_t) const {
      return _payload;
    }

  private:
    const double _payload;
};
```

either collapse to a constant or an array (completely inlinable)

- VDT math functions for auto vectorisation

# Batch & Vectorisation Benchmark

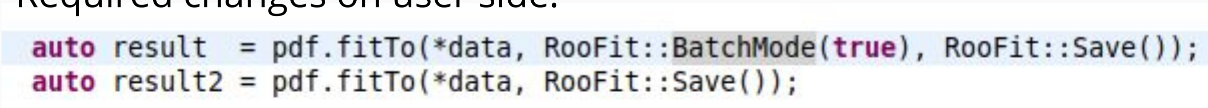
L(x | P) = Gauss(x | P1) + Gauss(x | P2) + Exp(x | P3)



| Single likelihood computation | CPU time / ms | Error | **Speed up** | Error |
|---|---|---|---|---|
| clang 7 -O3 SSE | Old<br>2867 | 45 | | |
| | 286 | 34 | **10.0** | **1.2** |
| clang 7 -O3 AVX2 | New<br>2834 | 22 | | |
| | 183 | 7 | **15.5** | **0.6** |
| clang 9 -O3 AVX512 | 2109 | 29 | | |
| Titan X * | 125 | 1 | **16.9** | **0.3** |

- Optimised Gauss, Exp, Sum, Poisson
- Batches & better cache locality result in 10x faster likelihood computation
- With AVX2, 16x faster LH possible
- (*) AVX512 should allow for more speed up, but CPU likely throttling

Required changes on user side:



```
auto result  = pdf.fitTo(*data, RooFit::BatchMode(true), RooFit::Save());
auto result2 = pdf.fitTo(*data, RooFit::Save());
```

Slides from Monday

# Batch & Vectorisation Full Fit

L(x | P) = Gauss(x | P1) + Gauss(x | P2) +
Exp(x | P3)

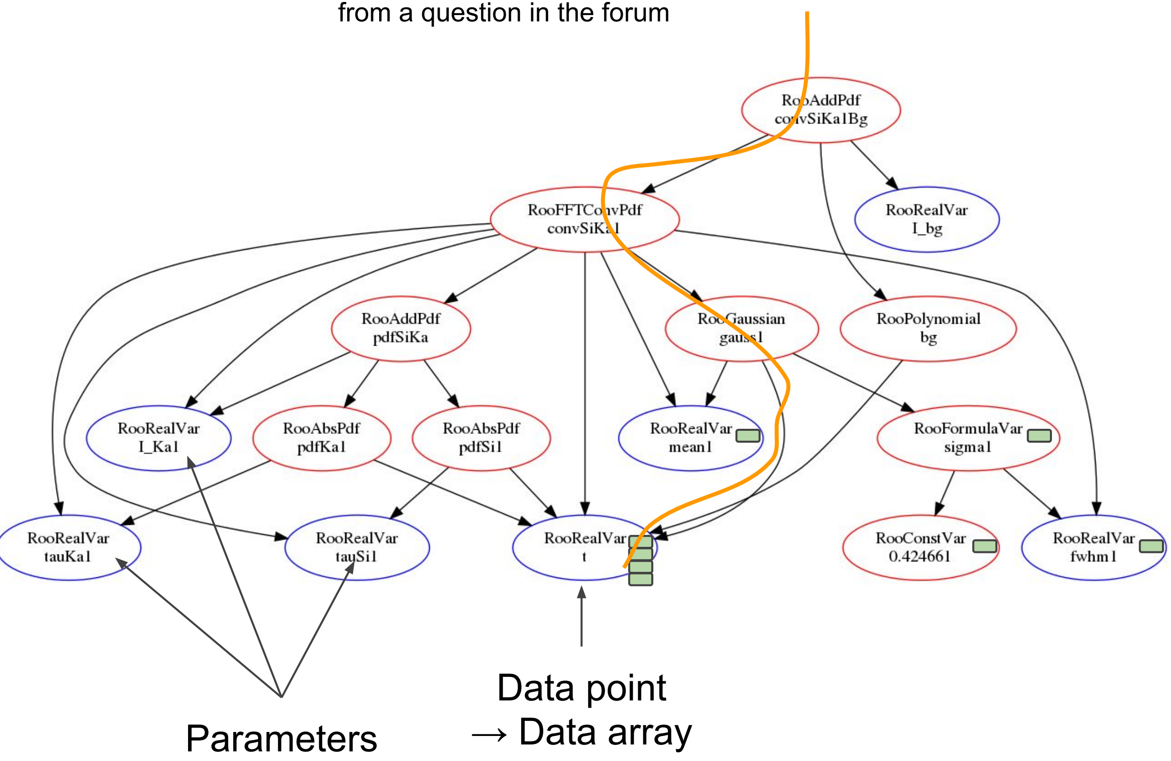
| Full fit + error estimation | CPU time / s | **Speed up** |
|---|---|---|
| clang 7 -O3 SSE | 9.61 | |
| | 2.45 | **3.9** |
| clang 7 -O3 AVX2 | 9.97 | |
| | 1.32 | **7.5** |
| clang 9 -O3 AVX512 | 6.53 | |
| Titan X * | 0.68 | **9.7** |

- Full fit can be 7 to 10 times faster with batches and vectorisation
- Results identical to 10E-14
  - Unit tests running batch against scalar code
  - Minimal differences expected (e.g. vdt::exp vs std::exp)

Slides from Monday

# Batched Function Evaluations

A random PDF
from a question in the forum



Parameters

Data point
→ Data array

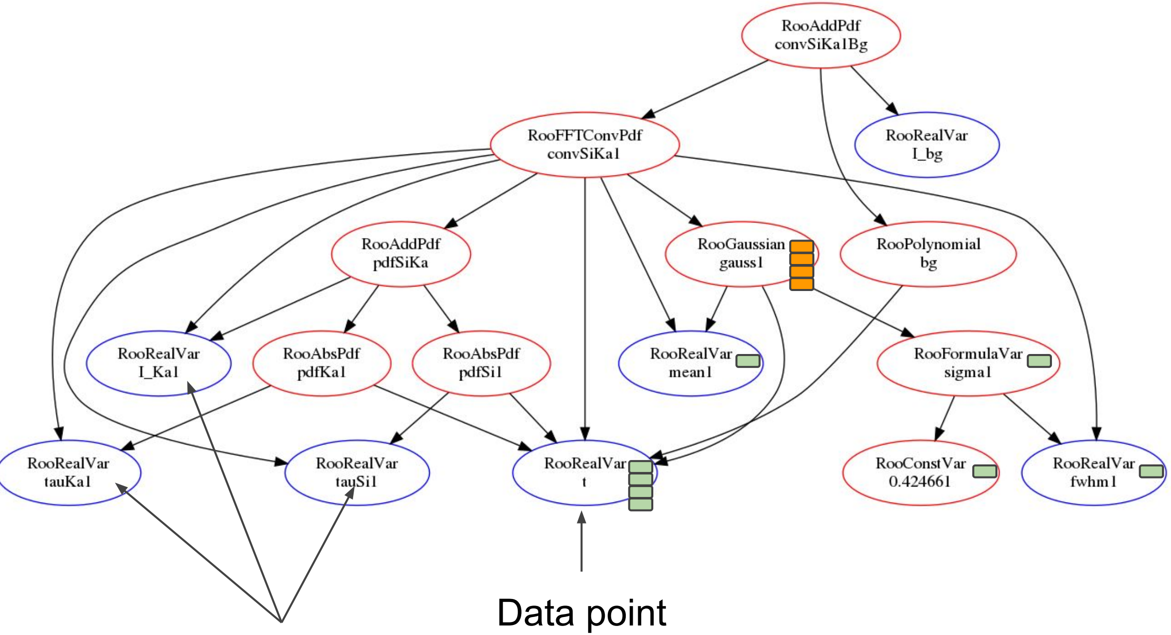
Now:

1. Evaluation requests batch of data at top node
2. Nodes call down to children
3. Arrive at leaf:
   a. Leaf is a parameter: return single value
   b. Leaf is an observable: return requested data batch

# Batched Function Evaluations

A random PDF

from a question in the forum



Parameters

Data point
→ Data array

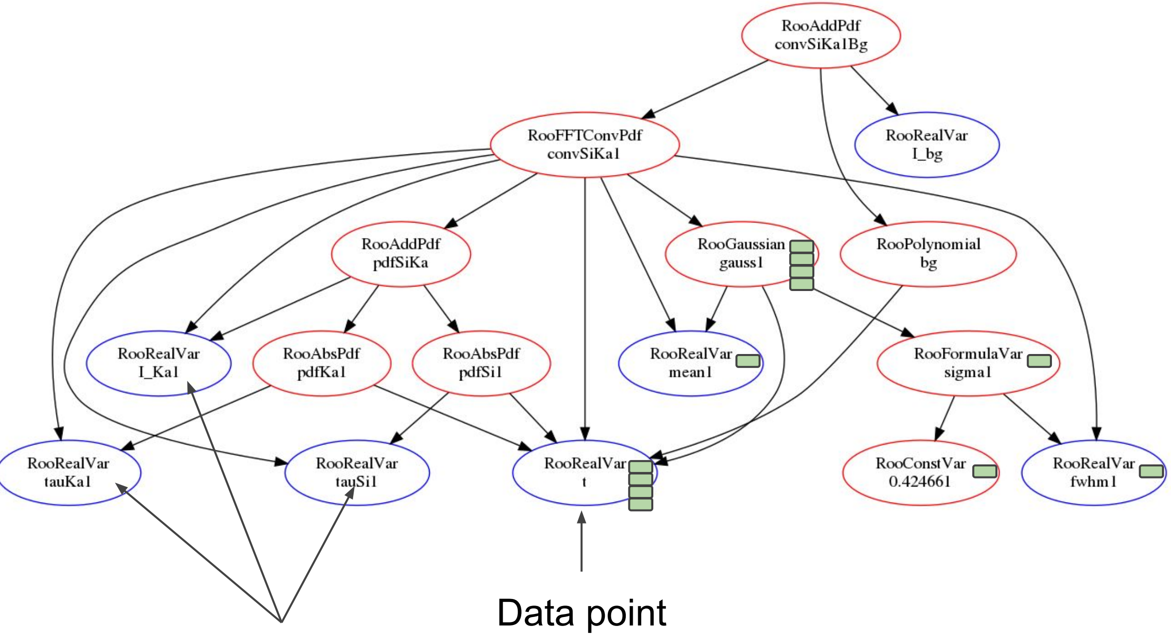
Now:

1. Evaluation requests batch of data at top node
2. Nodes call down to children
3. Arrive at leaf:
   a. Leaf is a parameter: single value
   b. Leaf is an observable: Returns requested data batch
4. Node starts computing using batch and parameter data
   a. Makes its own batch memory and fills it

# Batched Function Evaluations

A random PDF

from a question in the forum



Parameters

Data point
→ Data array

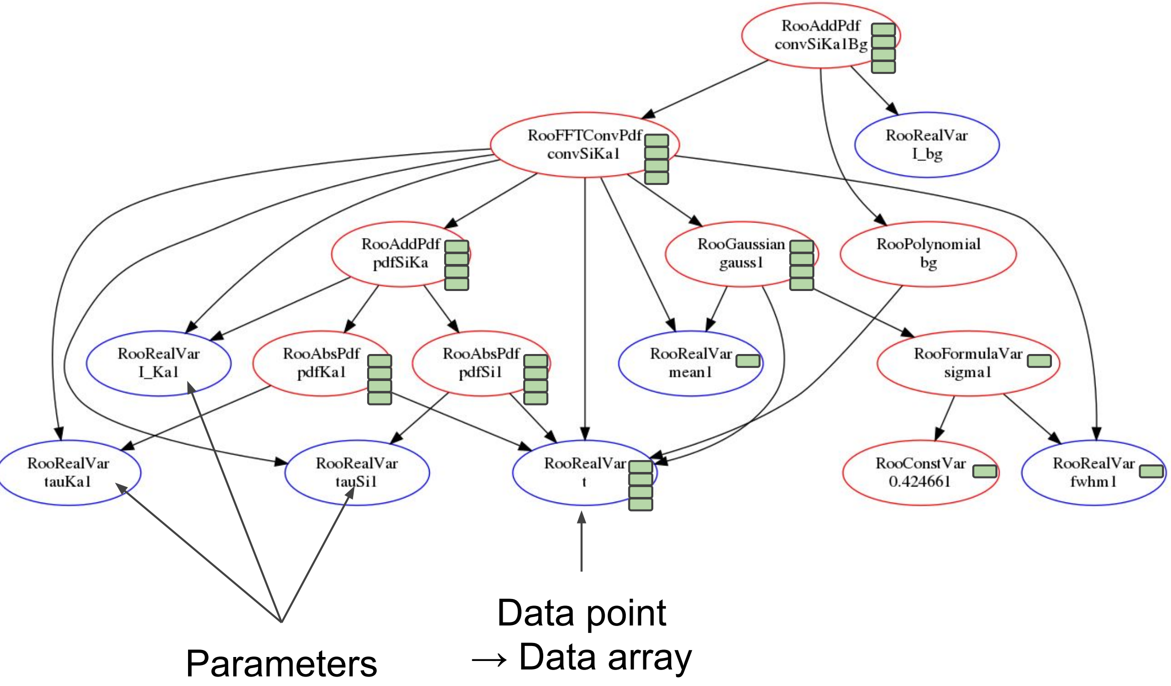
Now:

1. Evaluation requests batch of data at top node
2. Nodes call down to children
3. Arrive at leaf:
    a. Leaf is a parameter: single value
    b. Leaf is an observable: Returns requested data batch
4. Node starts computing using batch and parameter data
    a. Makes its own batch memory and fills it
    b. Returns batch

# Batched Function Evaluations

A random PDF

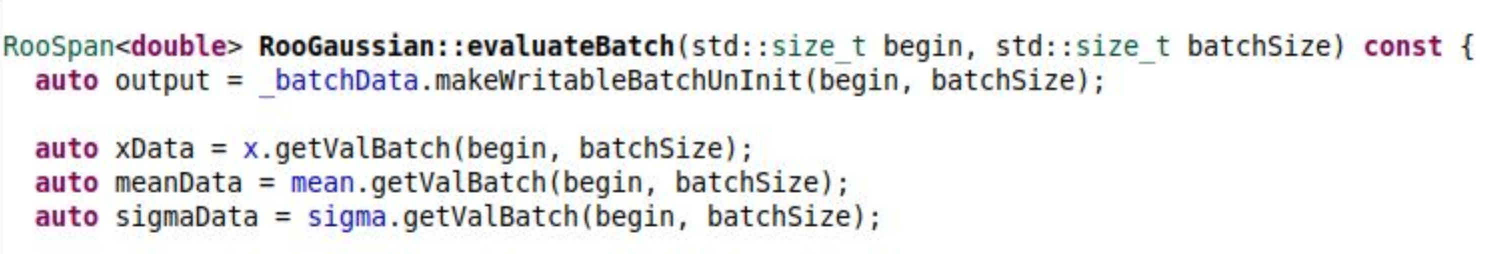
from a question in the forum



Now:

1. Evaluation requests batch of data at top node
2. Nodes call down to children
3. Arrive at leaf:
    a. Leaf is a parameter: single value
    b. Leaf is an observable: Returns requested data batch
4. Node starts computing using batch and parameter data
    a. Makes its own batch memory and fills it
    b. Returns batch
5. Propagate up

# Questions about the Batch Memory



```
RooSpan<double> RooGaussian::evaluateBatch(std::size_t begin, std::size_t batchSize) const {
  auto output = _batchData.makeWritableBatchUnInit(begin, batchSize);

  auto xData = x.getValBatch(begin, batchSize);
  auto meanData = mean.getValBatch(begin, batchSize);
  auto sigmaData = sigma.getValBatch(begin, batchSize);
```

What does a node need to know to manage its batch results?

- Batch begin index
- Batch size

- (Possibly: thread ID)

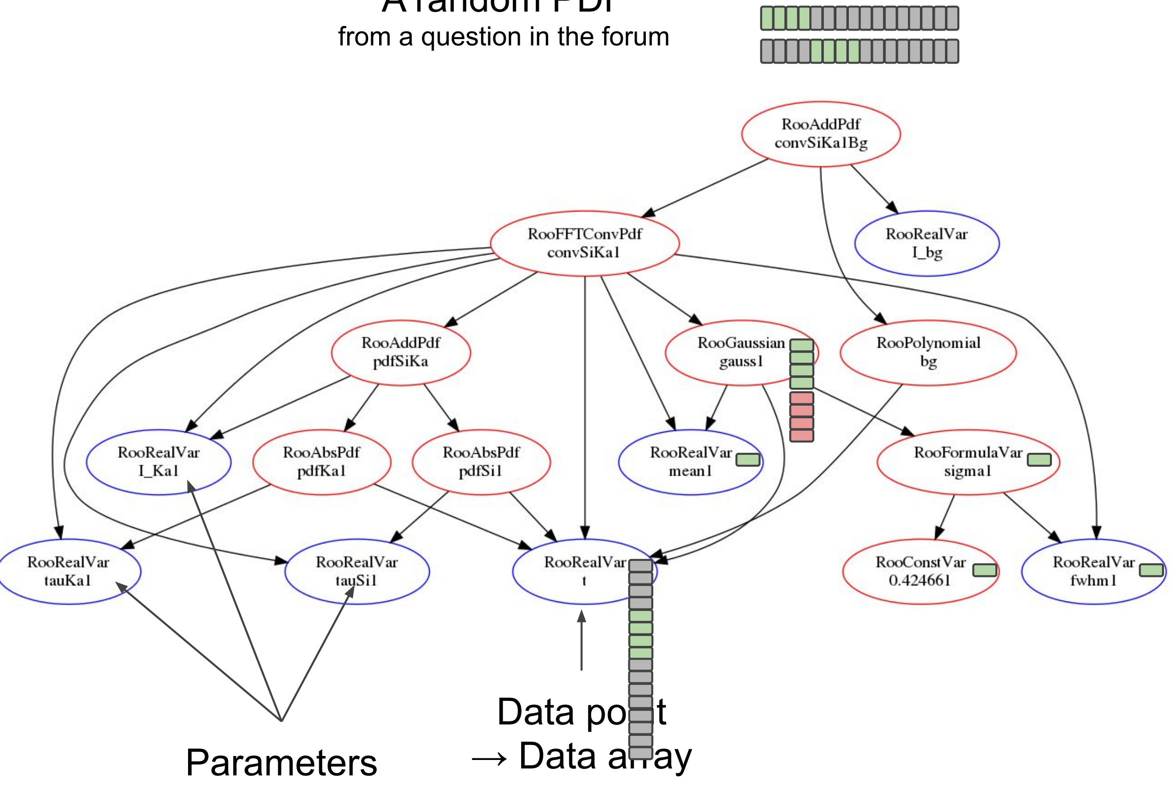
Requirements:

- Detect whether this batch was already computed & return
- Reuse memory
- Handle multiple range requests

- [Not supported] Re-use batch memory for different batches

# Reuse Batch Memory

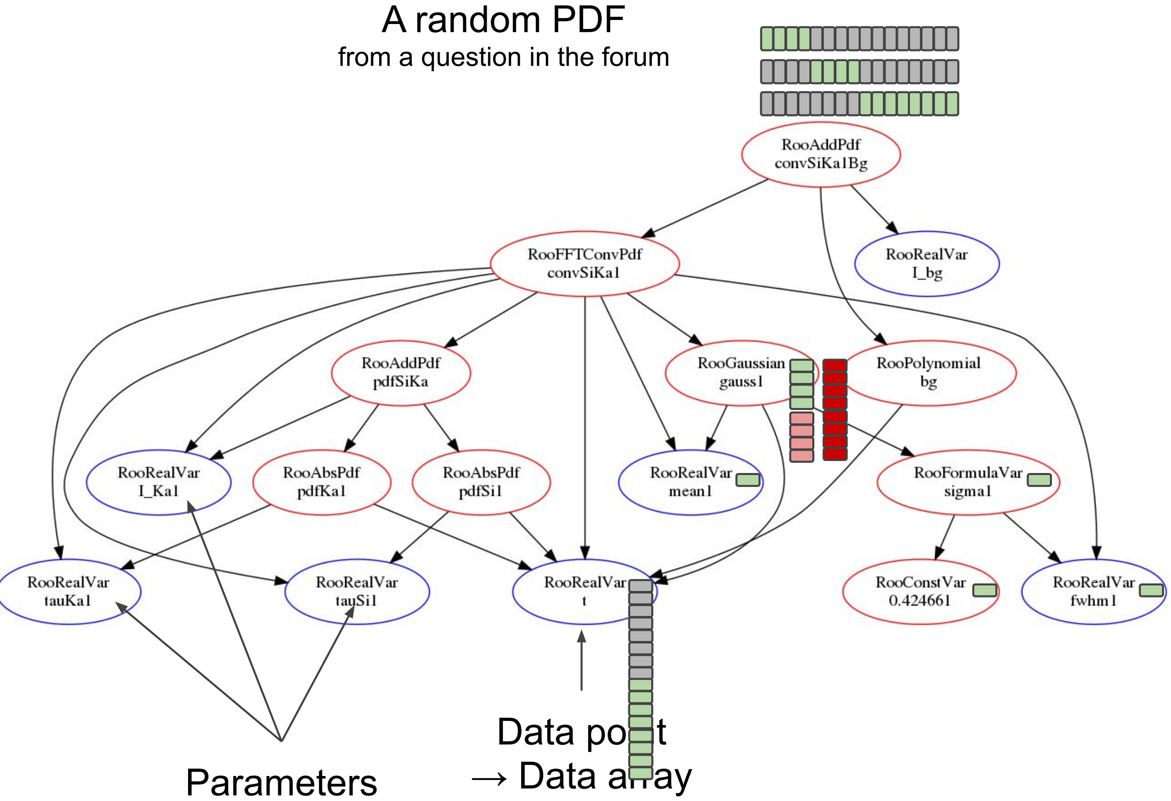
A random PDF
from a question in the forum



Future requirement:

- For very large datasets, might have to call multiple times
- Leafs - trivial: return request
- Nodes:
  - Need to map nth batch on node-local memory
  - Would currently create new memory
- Nodes don't know caller's intents:
  - No stride information
  - No notion of #batch
  - No idea about #workers
  - Will batch be needed again?

# Reuse Batch Memory



Further complication:

- Batch size might change between requests
- Will currently allocate even more memory

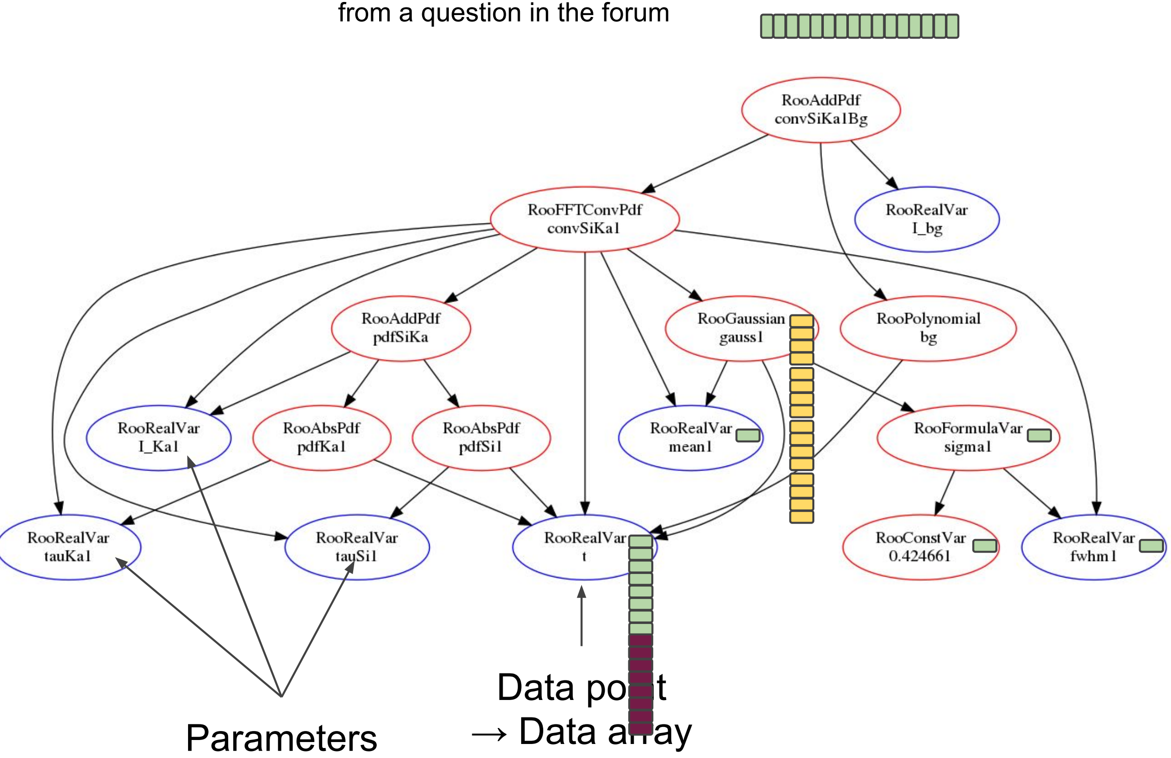
Possible solution:

- Index memory with something like a worker ID
- Always reuse
- Resize if necessary
- Invalidate batch results when jumping to the next data batch

# Reuse Batch Memory

A random PDF

from a question in the forum



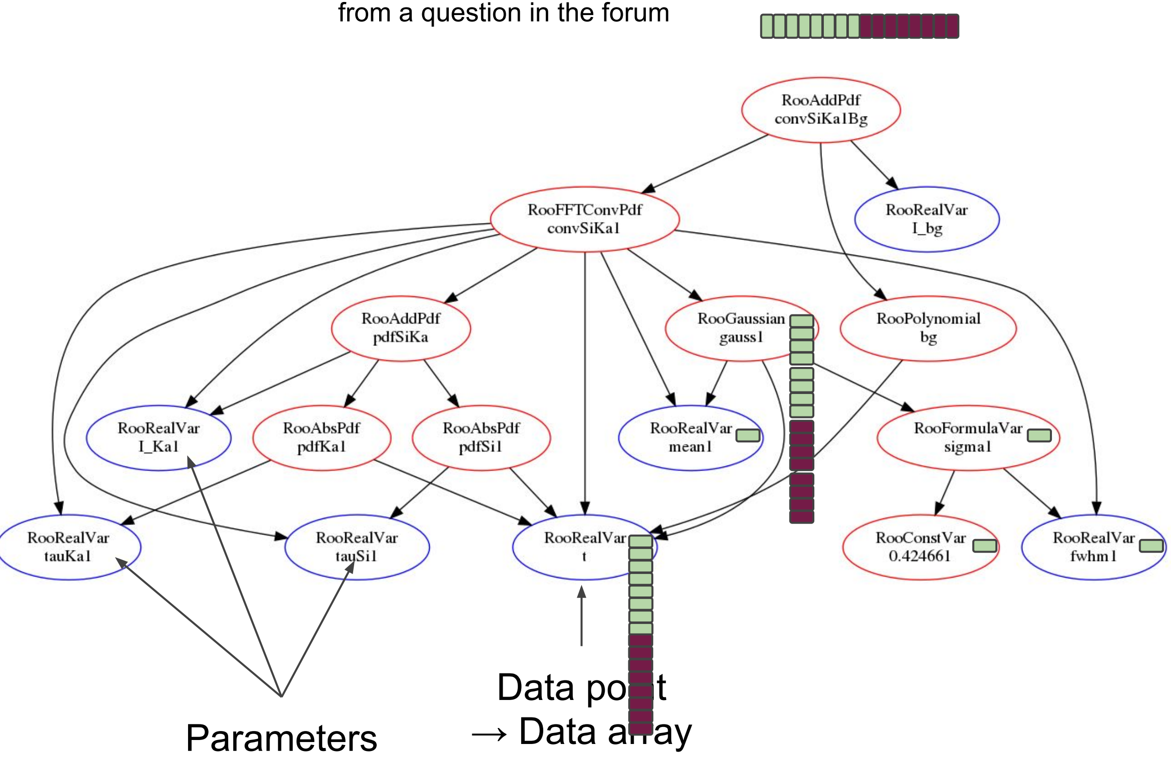
More complications:

- Request might be fulfilled only partially
- Think RNTuple as storage backend
  - Maximal batch size that can be returned is decompressed basket

# Reuse Batch Memory

A random PDF

from a question in the forum

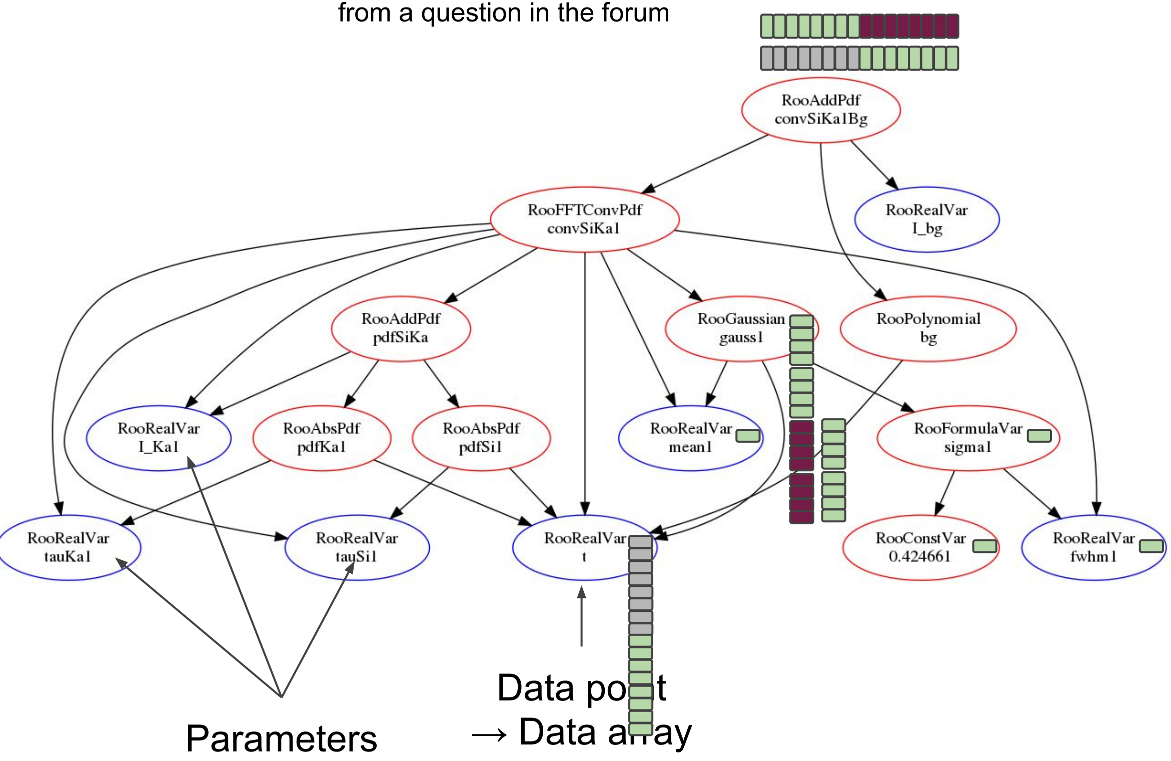


More complications:

- ○ Request might be fulfilled only partially
- ○ Think RNTuple as storage backend
  - ○ Maximal batch size that can be returned is decompressed basket

# Reuse Batch Memory

A random PDF
from a question in the forum



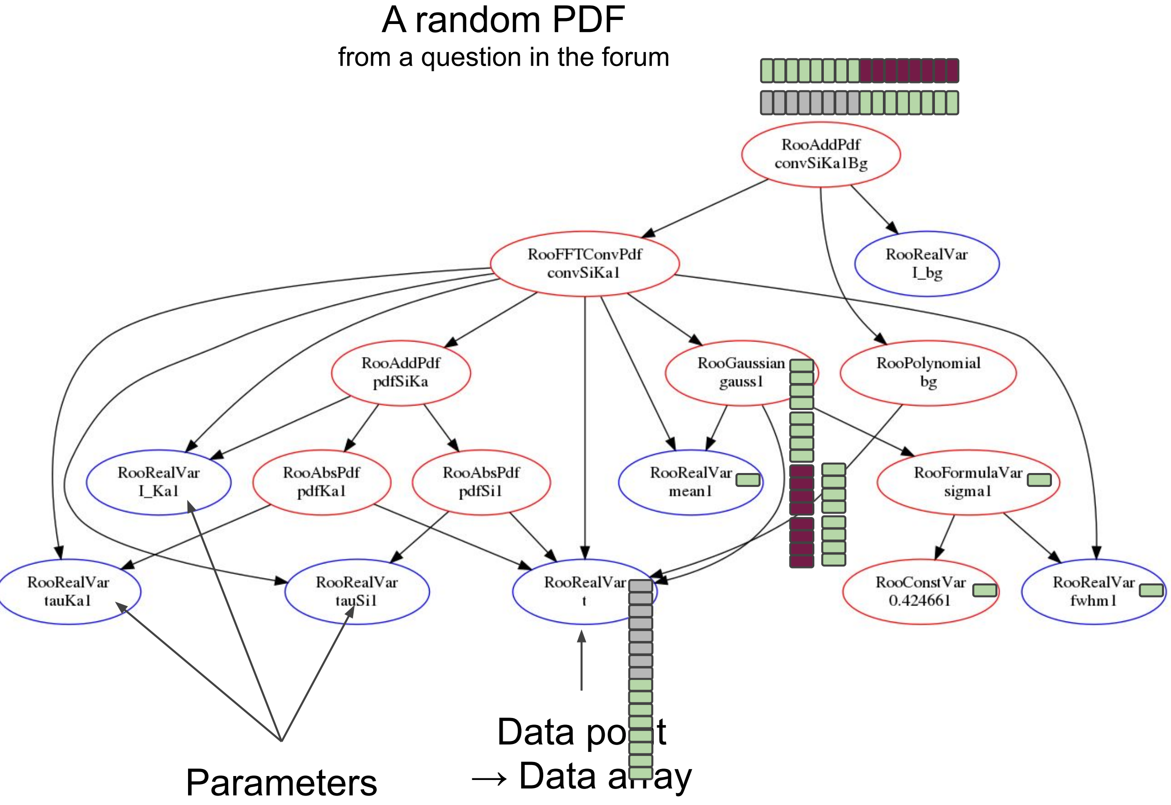
Parameters

Data point
→ Data array

More complications:

- Request might be fulfilled only partially
- Think RNTuple as storage backend
  - Maximal batch size that can be returned is decompressed basket
- Handled gracefully by top caller, re-request missing range

# Cache-Efficient Memory Management

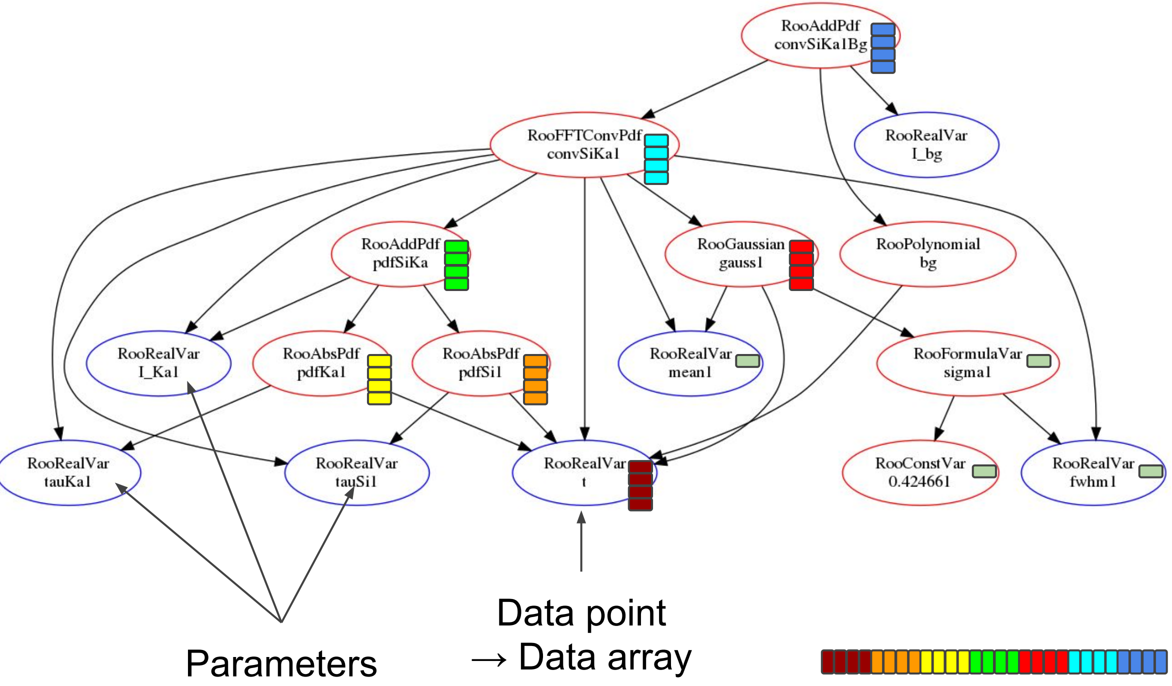


More complications:

- Request might be fulfilled only partially
- Think RNTuple as storage backend
  - Maximal batch size that can be returned is decompressed basket
- Handled gracefully by top caller, re-request missing range

# Cache-Efficient Memory Layout

A random PDF
from a question in the forum



Parameters

Data point
→ Data array

Is it possible to:

- Assign a block of memory (e.g. page size / cache size) to different nodes of the PDF?
- Would keep data extremely local (L1 / L2)
- Needs some planning and extra passes over the PDF tree
- Is maybe less flexible w.r.t. changes in batch size and parallel evaluation
- Boost performance?

# My Plan for RooFit

1. Fix the most pressing issues ROOT 6.16
2. LinkedList → std::vector<RooAbsArg*> ROOT 6.18
   - Much more memory friendly, faster to iterate/allocate/destroy/index access
3. Batched evaluation
   - Walk expression tree only once for all data points
   - Reduce number of virtual calls by factor of batch size
   - No change of state, no copying subtree ( → threads)
   - Data come as std::vector<double> and are accessed consecutively (cache-friendly)

   This depends on today's discussion
4. Vectorise loops inside batches Up to 10x speed up
5. Batched & threaded generation of toy data
   - Bottleneck for some analyses
6. Threads

https://sft.its.cern.ch/jira/browse/ROOT-9815

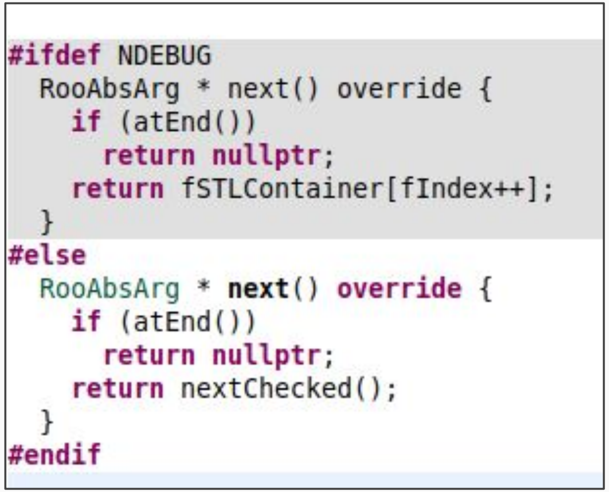
Slides from Monday

# Backup

# The Challenge II

- RooLinkedList:
  - Remove/add/replace before and after current iterator
  - No reallocations → iterator valid

- Solution: Legacy-to-STL adapters count
  - Can remove/add after iterator
  - Can replace everywhere
  - Safe also if reallocating
  - **But: Will break** when removing/adding **before** iterator



```
#ifdef NDEBUG
  RooAbsArg * next() override {
    if (atEnd())
      return nullptr;
    return fSTLContainer[fIndex++];
  }
#else
  RooAbsArg * next() override {
    if (atEnd())
      return nullptr;
    return nextChecked();
  }
#endif
```