Executing code on columnar data

Jim Pivarski

Princeton – DIANA

February 8, 2017
Why I’m interested in columnar data

I’m working on a query language and database server to aggregate large samples of HEP data on the fly.

**Purpose:** to eliminate the need for private skims in most situations.

Collaborating with Jin Chang and Igor Mandrichenko on the server.
The query language, Femtocode, plays a similar role as TTreeFormula:

- a high-level language for the physicist
- usually for filling a histogram (so query responses are small)
- but generally useful for transforming one dataset into another.

However, it’s a full-fledged language with assignments and user-defined functions, so that it can encompass a larger part of the data analysis.

(I’ve examined SQL, LINQ, and others, and they are not sufficient. I would use a standard if I could. Femtocode BNF has > 50% overlap with Python BNF.)
The essential feature of Femtocode is that it can compile complex structure-manipulations, which would ordinarily have to be performed in object-oriented code, into a series of vectorized kernels.

It operates on columns.
Example:

```scala
hist = dataset.bin(100, 0, 50, 
    muons.map(m => sqrt(m.px**2 + m.py**2)).max() 
)
```

compiles to

1. Compute $\sqrt{p_x^2 + p_y^2}$ for all muons, ignoring event boundaries.
2. Find the maximum such value for each event.
3. Bin those events and fill the histogram.

rather than

1. Loop over events:
   1.1 Loop over muons:
      1.1.1 Compute $\sqrt{p_x^2 + p_y^2}$ for each.
   1.2 Fill a histogram with the maximum.
Three types of data transformations:

**Flat:** apply $N$-argument function to each element of $N$ aligned arrays, ignoring boundaries.

**Explode:** emulate (nested) for-loops by replicating data in one array so that it becomes aligned with another array.

**Reduce:** emulate counters, sum, mean, max, etc. by combining elements of an array so that it becomes aligned with an outer level of structure.
Flat transformations

The majority of steps in a typical calculation are flat:

```c
double in[ZILLION];
double out[ZILLION];

for (int i = 0; i < ZILLION; i++)
  out[i] = flat_operation(in[i]);
```

- Compilation with `-O3` vectorizes if possible (depends on `flat_operation`).
- Easiest form for CPU to prefetch memory and/or pipeline operations.
- Also ideal for GPU calculations.
- There is a standard for functions of this form: Numpy’s ufunc is widely used among scientific libraries.
  - Easy way for a user to add functions to the language!
```python
import ctypes, numpy, numba

libMathCore = ctypes.cdll.LoadLibrary("libMathCore.so")
chi2_ctypes = libMathCore._ZN5TMath17ChisquareQuantileEdd  # c++filt!
chi2_ctypes.argtypes = (ctypes.c_double, ctypes.c_double)
chi2_ctypes.restype = ctypes.c_double

# compile to pure-C ufunc
@numba.vectorize(
    ["f8(f8, f8)"], nopython=True)
def chi2_ufunc(p, ndf):
    return chi2_ctypes(p, ndf)

p = numpy.random.uniform(0, 1, int(1e6))  # million random numbers
result = chi2_ufunc(p, 100)  # call ufunc on all of them  
# 3.22 seconds

import ROOT
result = [ROOT.TMath.ChisquareQuantile(pi, 100) for pi in p]
# 9.32 seconds

(Performance comparison is just to show that the ufunc computes ChisquareQuantile in C, not in Python. Simpler functions show a more dramatic difference.)
```
Explode operation

Depends critically on the way we represent structure. For the “recursive counter” method I described in the last talk,

Given:
\[
\begin{bmatrix}
\begin{bmatrix}
 a & b & c \\
\end{bmatrix} &
\begin{bmatrix}
 d & e & f & g \\
\end{bmatrix}
\end{bmatrix}
\begin{bmatrix}
\begin{bmatrix}
 h \\
\end{bmatrix} &
\begin{bmatrix}
 i & j \\
\end{bmatrix}
\end{bmatrix}
\end{bmatrix}
\]

Data array:
\[
a \ b \ c \ \ d \ e \ f \ g \ \ h \ \ i \ j
\]

Recursive counter:
\[
2 \ 3 \ \ 4 \ \ 2 \ 1 \ \ 2
\]

Calculating arbitrary explosions is solved in two cases:

- explode scalar to fit a list’s counter (35 lines of C)
- explode list to fit another list’s counter (470 lines, recursive).
Depends critically on the way we represent structure. For the “recursive counter” method I described in the last talk,

Given:
\[
\begin{bmatrix}
  [a \ b \ c] & [d \ e \ f \ g] & [h] & [i \ j]
\end{bmatrix}
\]

Data array:
\[a \ b \ c \ \ d \ e \ f \ g \ h \ i \ j\]

Recursive counter:
\[2 \ 3 \ 4 \ 2 \ 1 \ 2\]

Calculating arbitrary explosions is solved in two cases:

- explode scalar to fit a list’s counter (35 lines of C)
- explode list to fit another list’s counter (470 lines, recursive).

Illustration of scalar-to-list:

xs → [1, 2, 3, 4], [], [5, 6, 7] and y → 100, 200, 300

Computing
\[xs.map(x => x + y)\]

yields
\[ [101, 102, 103, 104], [], [305, 306, 307] \]
Explode operation

Depends critically on the way we represent structure. For the “recursive counter” method I described in the last talk,

Given:  

\[
\begin{bmatrix}
  \begin{bmatrix} a & b & c \end{bmatrix} \\
  \begin{bmatrix} d & e & f & g \end{bmatrix} \\
  \begin{bmatrix} h \end{bmatrix} \\
  \begin{bmatrix} i & j \end{bmatrix}
\end{bmatrix}
\]

Data array:  

a b c d e f g h i j

Recursive counter:  

2 3 4 2 1 2

Calculating arbitrary explosions is solved in two cases:

- **explode scalar to fit a list’s counter** (35 lines of C)
- **explode list to fit another list’s counter** (470 lines, recursive).

Illustration of list-to-deeper-list:

\[
\begin{align*}
xss & \rightarrow [[100, 200], [300, 400], [500, 600]] \\
yss & \rightarrow [1, 2, 3, 4]
\end{align*}
\]

Computing

\[
xss.map(xs => xs.map(x => ys.map(y => x + y)))
\]

yields

\[
[[[101, 102, 103, 104], [201, 202, 203, 204]],
[[301, 302, 303, 304], [401, 402, 403, 404]],
[[501, 502, 503, 504], [601, 602, 603, 604]]
\]
Explode operation

Depends critically on the way we represent structure. For the “recursive counter” method I described in the last talk,

Given:

\[
\begin{bmatrix}
a & b & c \\
d & e & f & g \\
h & i & j
\end{bmatrix}
\]

Data array:

\[a \ b \ c \ d \ e \ f \ g \ h \ i \ j\]

Recursive counter:

\[2 \ 3 \ 4 \ 2 \ 1 \ 2\]

Calculating arbitrary explosions is solved in two cases:

- explode scalar to fit a list’s counter (35 lines of C)
- explode list to fit another list’s counter (470 lines, recursive).

Another illustration of list-to-deeper-list:

\[\text{xss} \rightarrow [[100, 200], [300, 400], [500, 600]] \text{ and } \text{ys} \rightarrow [1, 2, 3, 4]\]

Computing

\[\text{xss.map}(x \Rightarrow \text{ys.map}(y \Rightarrow \text{xss.map}(x \Rightarrow x + y)))\]

yields

\[\begin{bmatrix}
[[101, 201], [102, 202], [103, 203], [104, 204]],
[[301, 401], [302, 402], [303, 403], [304, 404]],
[[501, 601], [502, 602], [503, 603], [504, 604]]
\end{bmatrix}\]
Reduce operations

Haven’t been implemented, but they’re pretty straightforward.
Before finishing the language, we want to understand how it will fit into the server.

Preliminary design:
If a centralized query server is going to replace private skims, it has to respond to aggregations over whole datasets in seconds.

**Purpose of early studies:** determine what performance is *possible.*
File-reading rates in events/ms per process (kHz per process), with the goal of extracting only $p_T$.

<table>
<thead>
<tr>
<th>particle</th>
<th>#/event</th>
<th># branches</th>
<th>CMSSW</th>
<th>TTree::Draw()</th>
<th>fast reader</th>
</tr>
</thead>
<tbody>
<tr>
<td>photon</td>
<td>2.9</td>
<td>205</td>
<td>1.14</td>
<td>435</td>
<td>769</td>
</tr>
<tr>
<td>electron</td>
<td>2.5</td>
<td>231</td>
<td>1.02</td>
<td>417</td>
<td>833</td>
</tr>
<tr>
<td>muon</td>
<td>2.7</td>
<td>192</td>
<td>1.02</td>
<td>16.5</td>
<td>770</td>
</tr>
<tr>
<td>tau</td>
<td>6.3</td>
<td>88</td>
<td>1.55</td>
<td>244</td>
<td>417</td>
</tr>
<tr>
<td>jet</td>
<td>16.7</td>
<td>95</td>
<td>1.15</td>
<td>123</td>
<td>182</td>
</tr>
<tr>
<td>AK8 jet</td>
<td>1.8</td>
<td>95</td>
<td>2.10</td>
<td>556</td>
<td>1000</td>
</tr>
</tbody>
</table>

- CMSSW loads all branches to reconstruct particles as a C++ objects. Loading all branches just to cut on $p_T$ is wasteful.
- TTree::Draw() is more streamlined, only loads required branches. (Low rate for muons is not understood.)
- “fast reader” is based on a code snippet Philippe prepared for me, using some of the same techniques as TTree::Draw().
Repeated queries on that file

We also plan to maintain an in-memory cache of recently used *columns*, on the supposition that the column-popularity distribution is steep enough to cause frequent cache-hits among users.

Rate for simple, flat functions on cached columns is limited only by memory bandwidth.

Could reach a peak of 7 GHz on KNL or GPU.
Conclusions

- I’m developing Femtocode to translate object semantics into vectorized operations as part of a project to create a fast query server.

- The “recursive counter” representation of nested structure can be exploded and reduced.
  - This representation is identical to ROOT’s for depth-1 lists.
  - Any interest in extending to arbitrary split depth?

- Flat functions are
  - quick to compute,
  - extensible using Numpy’s “ufunc” standard.

- For a cached query server,
  - $\sim 1$ MHz column entries is attainable for cache-misses,
  - $\sim 1$ GHz column entries is attainable for cache-hits.