Reconciling progress in statistical computation and data management

Hannes Mühleisen

CWI Scientific Meeting, 2018-01-26
Starting Point

- (Interpreted) Scalar Code
- DB: Tuple-at-a-Time, PostgreSQL, MySQL, …
- Problem:
  Massive overheads problematic if data gets big™
## Memory Hierarchy

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache reference</td>
<td>0.5</td>
</tr>
<tr>
<td>Branch mispredict</td>
<td>5</td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>7</td>
</tr>
<tr>
<td>Mutex lock/unlock</td>
<td>25</td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100</td>
</tr>
<tr>
<td>Compress 1K bytes with Zippy</td>
<td>3,000</td>
</tr>
<tr>
<td>Send 2K bytes over 1 Gbps network</td>
<td>20,000</td>
</tr>
<tr>
<td>Read 1 MB sequentially from memory</td>
<td>250,000</td>
</tr>
<tr>
<td>Round trip within same datacenter</td>
<td>500,000</td>
</tr>
<tr>
<td>Disk seek</td>
<td>10,000,000</td>
</tr>
<tr>
<td>Read 1 MB sequentially from disk</td>
<td>20,000,000</td>
</tr>
</tbody>
</table>

New: [https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html](https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html)
Bulk Processing

• Need to **avoid** cache misses
  • Process large chunks of data in one go using efficient code (C)

  • “Column-at-a-time”

• Example: Python vs. NumPy
  • `statistics.mean(range(1, pow(10, 7)))` 7.52s
  • `numpy.arange(1, pow(10, 7)).mean()` 0.03s
create table fuu (id integer, type integer);
explain select id from fuu where type=1;

X_19:bat[:int] := sql.bind(X_5, "sys", "fuu", "type", 0:int);
C_6:bat[:oid] := sql.tid(X_5, "sys", "fuu");
C_28 := algebra.thetaselect(X_19, C_6, 1:int, "+=");
X_9:bat[:int] := sql.bind(X_5, "sys", "fuu", "id", 0:int);
X_30 := algebra.projection(C_28, X_9);

What if almost all type values are 1?
Large Intermediates

- Intermediate results can get large, too
- Problematic, since they have to go to memory
  - or worse, to disk
Vectorized

- Innovation 1: **Vectorized Processing**

- Best of both worlds, small intermediate results, interpreted batch processing.
JIT Compilation

• Innovation 2: Ad-hoc generate scalar code that computes results, compile & run

• Avoids interpretation overheads & allows free pipelining

• Keep declarative user interface

How does this help avoiding branch mis-prediction & cache misses?
"tuple at a time" DBMS "X"
MySQL 4.1
interpretation dominates execution

"column at a time" MonetDB/MIL
main-memory materialization overhead
query without selection

"vector at a time" MonetDB/X100
low interpretation overhead
in-cache materialization

Time (seconds)

[Neumann, Efficiently Compiling Efficient Query Plans for Modern Hardware]
Query Compilation

[Neumann, Efficiently Compiling Efficient Query Plans for Modern Hardware]
• Code generation for numerical computation

• Tradeoff between performance & productivity

  • Immutable types, fixed types for variables, immutable, but extensible code, `const`

• Type inference, code generation & optimization

• LLVM

[Bezanson et al., Julia: A Fresh Approach to Numerical Computing]
Down the rabbit hole

julia> sum(1:100)

Step 1: Type Inference

Any[CodeInfo(:begin
    return $(Expr(:invoke, MethodInstance for
  sum(::UnitRange{Int64}), :(Main.sum), :($Expr(:new,
  UnitRange{Int64}, 1, :((Base.select_value)((Base.sle_int)(1,
  100)::Bool, 100, (Base.sub_int)(1, 1)::Int64)::Int64)))))
  end))=>Int64]

http://blog.leahhanson.us/post/julia/julia-introspects.html
Down the rabbit hole

julia> sum(1:100)

Step 2: LLVM

%0 = alloca %UnitRange, align 8
%1 = getelementptr inbounds %UnitRange, %UnitRange* %0, i64 0, i32 0
store i64 1, i64* %1, align 8
%2 = getelementptr inbounds %UnitRange, %UnitRange* %0, i64 0, i32 1
store i64 100, i64* %2, align 8
%3 = call i64 @julia_sum_62577(%UnitRange* nocapture nonnull readonly %0)
ret i64 %3
Down the rabbit hole

julia> sum(1:100)

Step 3: Native Code

```
subq $16, %rsp
movq $1, -16(%rbp)
movq $100, -8(%rbp)
movabsq $sum, %rax
leaq -16(%rbp), %rdi
callq *%rax
addq $16, %rsp
popq %rbp
retq
```

What’s missing here?
Non-Standard Evaluation

- Bringing DB-style high-level optimisations to Statistical Programming
- Lazy evaluation of R scripts to build tree of deferred ops, then optimize

[Mühleisen et al., Relational Optimizations for Statistical Analysis]
Deferred Evaluation

a <- 1:1000
b <- a + 42
c <- b[1:10]
d <- min(c) / max(c)
print(d)
svytheme

agep <- svymean(~agep, svydsgn, se=TRUE)

for(i in 1:ncol(wts)) {
    repmeans[i,]<-t(colSums(wts[,i]*x*pw)/
    sum(pw*wts[,i]))
}
[...]
v<-crossprod(sweep(thetas,2,
    meantheta,"-"))*sqrt(rscales))*scale
svymean

crossprod 0.2

* - [5]
t

repmeans

/ / / /
| colSums (cached) colSums (cached) colSums (cached) colSums (cached) colSums (cached)

* /
5.5. **Performance Recap.** In the early days of high-level numerical computing languages, the thinking was that the performance of the high-level language did not matter so long as most of the time was spent inside the numerical libraries. These libraries consisted of blockbuster algorithms that would be highly tuned, making efficient use of computer memory, cache, and low-level instructions.

What the world learned was that only a few codes spent a majority of their time in the blockbusters. Most codes were being hampered by interpreter overheads, stemming from processing more aspects of a program at run time than are strictly necessary.
Weld

Growing gap between memory/processing makes traditional way of combining functions worse

data = pandas.parse_csv(string)
filtered = pandas.dropna(data)
avg = numpy.mean(filtered)

[Slides: Palkar et al., Weld: A Common Runtime for Data Analytics]
Weld Architecture

User Application

```
data = lib1.f1()
lib2.map(data,
    item => lib3.f2(item)
)
```

Weld Runtime

- IR fragments for each function
- Combined IR program
- Optimized machine code

Data in application

Runtime API
Weld Cross-Library

Pandas + NumPy

<table>
<thead>
<tr>
<th>Runtime (sec, log10)</th>
<th>Native</th>
<th>Weld, no CLO</th>
<th>Weld, CLO</th>
<th>Weld, 12 core</th>
</tr>
</thead>
</table>

Spark SQL UDF

<table>
<thead>
<tr>
<th>Runtime (sec)</th>
<th>Scala UDF</th>
<th>Weld</th>
</tr>
</thead>
</table>

CLO = with cross library optimization
Example Optimization: Fusion

squares = map(data, x => x * x)
sum = reduce(data, 0, +)

bld1 = new vecbuilder[int]
bld2 = new merger[0, +]
for x in data:
  merge(bld1, x * x)
  merge(bld2, x)

Differences to Julia?
Optimisation Challenge

• Optimize this:

  • PLOT(SQL(R(C(BLAS(data)))))
  • SQL(Python(TensorFlow(Pandas(NumPy(C(data))))))