HARDWARE-CONSCIOUS DATA PROCESSING SYSTEMS

Holger Pirk

http://doc.ic.ac.uk/~hlgr
Data Processing Performance - A Case Study
Data Processing Performance - A Case Study

≈ select sum(sales), ... where country = US ... group by month, ...
Data Processing Performance
- A Case Study

TPC-H Query 1

≈ select sum(sales), … where country = US … group by month, …

(Roughly 10 GB of Data)
Data Processing Performance
- A Case Study

select sum(a) where b=6 group by c
Data Processing Performance - A Case Study

select sum(a) where b=6 group by c
Data Processing Performance
- A Case Study

The solution: memory-resident data?

select sum(a) where b>6 group by c
Data Processing Performance - A Case Study

Memory Bandwidth bound, i.e., 37ms

Postgres on Disk

Postgres on Ramdisk
Data Processing Performance
- A Case Study

- Memory Bandwidth bound, i.e., 37ms
Data Processing Performance - A Case Study

- Postgres on Disk: 100 s
- Postgres on Ramdisk: 96 s
- MonetDB: 3.2 s

Memory Bandwidth bound, i.e., 37ms

30x faster!
Data Processing Performance - A Case Study

Postgres on Disk: 100 s
Postgres on Ramdisk: 96 s
MonetDB: 3.2 s
Voodoo: 0.162 s

Memory Bandwidth bound, i.e., 37ms

How to get there
Data Processing Performance - A Case Study

Column at a time Processing

Memory Bandwidth bound, i.e., 37ms

Postgres on Disk: 100 s
Postgres on Ramdisk: 96 s
MonetDB: 3.2 s
Voodoo: 0.162 s
Classic database architecture

```
void scan(int* input, int* output) {...}
```

SQL

- Logical Plan
  - Select, Join, Group, ...
  - Tablescan, Hashjoin, ...

DBMS

- DB Kernel
  - void scan(int* input, int* output) {...}

OS/Hardware/...

Optimizer

Optimizer

DBMS

OS
Column at a time Processing

- Postgres is a *tuple at a time* kernel
  - High interpretation overhead
  - Hard to parallelize
- MonetDB is a *column at a time* kernel
  - Overhead amortized over many tuples/values
  - Easy to parallelize

Ramdisk does not impact performance
...back to our study

Generating optimized code

<table>
<thead>
<tr>
<th>Database</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Postgres on Disk</td>
<td>100 s</td>
</tr>
<tr>
<td>Postgres on Ramdisk</td>
<td>96 s</td>
</tr>
<tr>
<td>MonetDB</td>
<td>3.2 s</td>
</tr>
<tr>
<td>Voodoo</td>
<td>0.162</td>
</tr>
</tbody>
</table>

Memory Bandwidth bound, i.e., 37ms
```
return
join
select

for tuple1 in R1:
    tmp1[hash(tuple1)] = tuple1

for tuple2 in R2:
    if hash(tuple2) in tmp1:
        tmp2[hash(tuple2)] = (tuple2, tmp1[hash(tuple2)])

for tuple3 in R3:
    if some_condition(tuple3):
        if hash(tuple3) in hash2:
            yield (tuple3, *tmp2[hash(tuple3)])
```
Query Compiler Architecture

- SQL
  - Logical Plan
    - Physical Plan
      - DB Kernel
        - OS/Hardware/…

DBMS

OS
Query Compiler Architecture

- **Optimizer**
  - SQL
  - Logical Plan
  - Executable
  - OS/Hardware/…

- **Compiler**
  - Data Management System

**Imperial College London**
Database Performance Engineering

Data Management

Compilers

Computer Architecture
The Performance Engineering Deluge

Branch-Free Selections, Radix-Joins, Vectorized Processing, Architecture-Conscious Hashing, SIMD-Parallel Processing, Bitwise Processing, NUMA-Aware Processing, Superscalar (De)compression, Instruction-Cache Aware Processing, Multicore-Parallelism, Co-Processing, …
Multicore vs. SIMD

Multicore-Parallelism

SIMD-Parallel Processing
A data processing example
Multicore-Parallelism using TBB

- Parallel Reduce
- Block-Range
- Per-Partition Lambda
- Global Lambda

```cpp
auto input = load("input");

auto totalssum =
  parallel_deterministic_reduce(
    blocked_range<size_t>(0, input.size,
      input.size / 1024)
  0, [&input](auto& range, auto partsum) {
    for(size_t i = range.begin();
      i < range.end(); i++) {
      partsum += input.elements[i].constant;
    }
    return partsum;
  },
  [](auto s1, auto s2) { return s1 + s2; });
```
Multicore vs. SIMD

Single Instruction Multiple Data parallelism

Vectorized Add:

\[
\begin{array}{cccc}
6 & 3 & 9 & 2 \\
+ & & & \\
2 & 4 & 0 & 1 \\
= & & & \\
8 & 7 & 9 & 3 \\
\end{array}
\]
A data processing example using SIMD
A data processing example using SIMD

```cpp
auto input = load("input");
typedef int v4i __attribute__((vector_size(16)));
auto vSize = (sizeof(v4i) / sizeof(int));
v4i sums = {};
for(size_t i = 0; i < input.size / vSize; i++) {
    sums += ((v4i*)input.elements)[i];
}
int* scalarSums = (int*)&sums;
auto totalsum = 0l;
for(size_t i = 0; i < 4; i++) {
    totalsum += scalarSums[i];
}
```

- SIMD Datatypes
- Loop Bound Adaption
- Array Cast
- Sequential Reduction
Multicore vs. SIMD

Multicore-Parallelism

- Parallel Reduce
- Block-Range
- Per-Partition Lambda
- Global Lambda

SIMD-Parallel Processing

- SIMD Datatypes
- Loop Bound Adaption
- Array Cast
- Sequential Reduction

Particularly problematic when generating code
Query Compiler Architecture

SIMD-Parallel Processing
Multicore-Parallelism
Join-Ordering
Query Compiler Architecture

- SQL
- Logical Plan
- Executable
- OS/Hardware/

Optimizers

DBMS

Compiler

OS

SIMD-Parallel Processing
Multicore-Parallelism
Join-Ordering
Query Compiler Architecture

- SQL
- Logical Plan
- Executable
- OS/Hardware/
- Join-Ordering
- SIMD-Parallel Processing
- Multicore-Parallelism

Data aware

Hardware aware

Optimizer

Compiler

DBMS

OS

Imperial College London
What if…

Optimizer

SQL

Logical Plan

Unified Algebra

Executable

OS/Hardware/…

SIMD-Parallel Processing
Multicore-Parallelism
Join-Ordering
What if we had an intermediate algebra

Data & Hardware aware

Optimizer

SQL

Logical Plan

Voodoo

Executable

OS/Hardware/…

SIMD-Parallel Processing
Multicore-Parallelism
Join-Ordering
Voodoo

A Vector Algebra

A portable high-performance database kernel

A platform for database performance engineering

[VLDB 2016/17]
The Voodoo Vector Algebra
Design goals

• Fast and expressive like C
• Optimizable like relational algebra
• Portable to different devices
• (Enable serendipitous discovery of new optimizations)
Design goals

• Fast and expressive like C
  • All tuning decisions are explicit in the program
  • Focused on data processing

• Optimizable like relational algebra
  • Dataflow ♦ Optimization are graph transformation rules

• Portable to different devices
  • Least common denominator data model and operator set
Data model: the least common denominator of targeted hardware

- struct {int id;
    struct {
        int x;
        int y
    } position;
} poi[n]
All Voodoo operators are parallel, some are controlled

• Map-like (Fully Data Parallel):
  • Project, Zip, Arithmetic, Logical, Bit Operations, Gather (i.e., $z = x[y]$), Cross

• Controlled:
  • FoldSelect, FoldSum, FoldMax, FoldScan, Scatter (i.e., $x[y] = z$), ...
A short revision on fold

- Fold(f, [8,9,5,7,0,5,7,4,5])
Control: explicit partition assignment

<table>
<thead>
<tr>
<th>partition</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

Foldsum

.sum

7

9
Controlled folding - declarative & tunable

.value

2 0 4 1 3 1 5 0

Foldsum

.sum

7 9
Controlled folding - declarative & tunable

```
.value
  2  0  4  1  3  1  5  0

.sum
  7

.foldsum
  9

.foldsum
  16
```
Creating Control Vectors

\[
\begin{array}{ccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\
\hline
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
\end{array}
\]

range(data) = constant(data, 4) = control vector
Partitioned aggregation in Voodoo is controlled

```
1 input := Load("input") // Single Column: val
2 ids := Range(input)
3 partitionSize := Constant(4)
4 partitionIDs := Divide(ids, partitionSize)
5 positions := Partition(partitionIDs)
6 inputWPart := Zip(input, partition)
7 partInput := Scatter(inputWPart, positions)
8 pSum := FoldSum(partInput.val, partInput.partition)
9 totalSum := FoldSum(pSum)
```
Lanewise folding abstracts SIMD parallelism…
Creating Control Vectors

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array}
\]
range(data)

\[
\begin{array}{cccccccc}
2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\
\end{array}
\]
constant(data, 2)

\[
\begin{array}{cccccccc}
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
\end{array}
\]
partition IDs
...and looks almost the same as multicore-parallelism

```plaintext
input := Load("input") // Single Column: val
ids := Range(input)
partitionSize := Constant(1024)
partitionIDs := Divide(ids, partitionSize)
positions := Partition(partitionIDs)
inputWPart := Zip(input, partition)
partInput := Scatter(inputWPart, positions)
pSum := FoldSum(partInput.val)
totalSum := FoldSum(pSum)
```

```plaintext
input := Load("input") // Single Column: val
ids := Range(input)
laneCount := Constant(2)
partitionIDs := Modulo(ids, laneCount)
positions := Partition(partitionIDs)
inputWPart := Zip(input, partition)
partInput := Scatter(inputWPart, positions)
pSum := FoldSum(partInput.val, partInput.partition)
totalSum := FoldSum(pSum)
```
A portable high-performance database kernel
The Voodoo query processing system

- Optimizer
- SQL
- Logical Plan
- Unified Algebra
- Intermediate Language
- OS/Hardware/

"Liberated" from MonetDB
Similar to code generation example
Our Contributions
OpenCL
Let’s extend our example

```sql
SELECT SUM(l_quantity) FROM lineitem WHERE l_shipdate > 5
```
MonetDB generates a logical plan...

```
SELECT SUM(l_quantity)
FROM lineitem
WHERE l_shipdate > 5
```
we compile the logical plan to Voodoo...

\[
\begin{align*}
\text{sum} & : \text{tmp1} = \text{Load}(.\text{lineitem}_L\text{.l\_quantity}) \\
& \quad \text{tmp2} = \text{Load}(.\text{lineitem}_L\text{.l\_shipdate}) \\
& \quad \text{tmp3}.\text{val} = \text{Range}(728659,\text{tmp2},0) \\
& \quad \text{tmp4}.\text{val} = \text{Greater}({\text{tmp2},.\text{l\_shipdate}},\text{tmp3}.\text{val}) \\
& \quad \text{tmp5}.\text{val} = \text{Range}(0,\text{tmp4},1) \\
& \quad \text{tmp6} = \text{Zip}(.\text{fold},\text{tmp5}.\text{val},.\text{value},\text{tmp4}.\text{val}) \\
& \quad \text{tmp7}.\text{val} = \text{FoldSelect}({\text{tmp6}.\text{fold}},.\text{fold},.\text{value}) \\
& \quad \text{tmp8} = \text{Gather}({\text{tmp1}},\text{tmp7}.\text{val}) \\
& \quad \text{tmp14}.\text{val} = \text{Range}(0,\text{tmp8},0) \\
& \quad \text{tmp15} = \text{Zip}(.\text{fold},\text{tmp14}.\text{val},.\text{value},\text{tmp8}.\text{val}) \\
& \quad \text{tmp16}.L1 = \text{FoldSum}({\text{tmp15}.\text{fold}},.\text{fold},.\text{value}) \\
& \text{Return}(\text{tmp16})
\end{align*}
\]
...i.e., a dataflow program
Generating "undergraduate" C from Voodoo

```c
for(size_t i = 0; i < sizeof(l_shipdate); i++)
    if(l_shipdate[i] > 5)
        result += l_quantity[i]
```
The "graduate" version

```c
extern size_t inputSize;

void fragment1(int* tmp, int* s_date, int* quantity) {
    for(size_t i = 0; i < grainsize; i++)
        if(s_date[pId * grainsize + i] > 5)
            out[pId] += quantity[pId * grainsize + i]
}

void fragment2(int* out, int* tmp) {
    for(size_t i = 0; i < inputSize / grainsize; i++)
        output[0] += tmp[i]
}
```

Multicore-partitioned fold
control-incompatible folds
→ hierarchical aggregation
Voodoo outperforms MonetDB (TPC-H subset, SF10)

an old acquaintance
Voodoo is competitive with Hyper on CPU (selected TPC-H queries, SF10)
Voodoo is competitive with Hyper on CPU (selected TPC-H queries, SF10)
Voodoo is competitive with Hyper on CPU (selected TPC-H queries, SF10)

![Chart showing performance comparison between Voodoo and Hyper for TPC-H queries]
Voodoo is competitive with Hyper on CPU (selected TPC-H queries, SF10)

Sometimes slower

Imperial College London
Voodoo is competitive with Hyper on CPU (selected TPC-H queries, SF10)

*Q9 is an interesting case where tuning is important*
VOODOO AS A TUNING TOOL
Selective Foreign-Key Joins

```
select sum(part.price)
from lineitem, part
where discount > $x
and lineitem.partkey_fk = part.partkey_pk
```

select items that were sold at a discount greater than $x
look up their price from the parts table
and sum it
Selective Foreign-Key Joins

```
select sum(part.price)
from lineitem, part
where discount > $x
and lineitem_partkey_fk = part.partkey_pk
```

lookup/Foreign-Key Join

primary key, i.e., unique
Selective Foreign-Key Joins

```
select sum(price)
from lineitem, part
where discount > $1
and lineitem.partkey = part.partkey
```
Selective Foreign-Key Joins

select sum(price) from lineitem, part
where discount > $1
and lineitem.partkey = part.partkey
### Selective Foreign-Key Joins

**SQL Query:**

```sql
SELECT sum(price)
FROM lineitem, part
WHERE discount > 1
AND lineitem.partkey = part.partkey
```

**Diagram:**

![Diagram of foreign-key join with selective join conditions]
select sum(price) 
from lineitem, part 
where discount > $1 
and lineitem.partkey = part.partkey
select sum(price) 
from lineitem, part 
where discount > $1 
and partkey = partkey 

for (size_t i = 0; i < lineitemsize; i++) 
if (discount[i] > $1) 
   result += price[partkey[i]];

return
Predicated Foreign-Key Joins

```c
for(size_t i = 0; i < lineitemsize; i++)
    if(discount[i] > $1)
        result += price[partkey[i]];
```
Predicated Foreign-Key Joins

```c
for(size_t i = 0; i < lineitemsize; i++)
    result += (discount[i] > $1) * price[partkey[i]];
```
select sum(price) 
from lineitem, part 
where discount > $1 
and lineitem.partkey = part.partkey
Predicated Foreign-Key Joins

```sql
select sum(price)
from lineitem, part
where discount > $1
and lineitem.partkey = part.partkey
```
Predicated Foreign-Key Joins

\[
\text{FoldSum(}
\quad \text{Gather(}
\quad \text{FoldSelect}(x), y), z))
\quad \Rightarrow \text{FoldSum(}
\quad \text{Multiply}(x, 
\quad \text{Gather}(y, z)))
\]
select sum(price) 
from lineitem, part 
where discount > $1 
and partkey = partkey

for(size_t i = 0; i < lineitemsize; i++)
result += (price[i] > $1) * 
p_discount[partkey[i]]

Range
Greater
Multiply
Zip
FoldSum
return
Selective Foreign-Key Joins

```
select sum(price) 
from lineitem, part 
where discount > $1 
and partkey = partkey 
```

```
for(size_t i = 0; i < lineitemsize; i++)
result += (price[i] > $1) * 
p_discount[partkey[i]];
```
Double-predicated Foreign-Key Joins

FoldSum(
    Multiply(
        Gather(
            x,y),z))
=>FoldSum(
    Multiply(
        Gather(
            Multiply(x,z),y),z))
select sum(price) 
from lineitem, part 
where discount > $1 
and lineitem.partkey = part.partkey
select sum(price)
from lineitem, part
where discount > 1
and lineitem.partkey = part.partkey
Double-predicated Foreign-Key Joins

```
select sum(price) 
from lineitem, part 
where discount > $1 
and lineitem.partkey = part.partkey
```
Double-predicated Foreign-Key Joins

select sum(price) 
from lineitem, part 
where discount > $1 
and partkey = partkey

for(size_t i = 0; i < lineitemsize; i++)
result += (discount[i] > $1) * 
price[partkey[i]];
Double-predicated Foreign-Key Joins

```
select sum(price) 
from lineitem, part 
where discount > $1 
and partkey = partkey 
```

```
for(size_t i = 0; i < lineitemsize; i++) 
result += (discount[i] > $1) * 
price[partkey[i] * 
(discount[i] > $1)];
```
Double-predicated Foreign-Key Joins

![Graph showing the relationship between selectivity and time in seconds for Branching, Predicated Aggregation, and Predicated Lookups.](image-url)
Voodoo Wrap

- Fast and expressive like C
- Optimizable like relational algebra
- Portable to different devices
- Enables serendipitous discovery of new optimizations
Quo Vadis
FUTURE WORK

Optimizer

SQL
   Relational Algebra

Voodoo

Executable

OS/Hardware/…
Plugging holes in the Design Space

Remember these?
Plugging holes in the Design Space

- Massively Parallel Top-K [SIGMOD 2018]
- Incremental Window Computation [ongoing]
- Dynamic Load Balancing on GPUs [future]
Auto-Generating Databases

![Graph showing selectivity and time](Image)

- Branching
- Predicated Aggregation
- Predicated Lookups

**Time in seconds**

**Selectivity**

Imperial College London
Auto-Generating Databases

![Graph showing the time in seconds for different methods (Branching, Predicated Aggregation, Predicated Lookups) as a function of selectivity in %]
Auto-Generating Databases

• Hardware-conscious modelling & tuning [ongoing]
Auto-Generating Databases

Hardware Model

Voodoo Program Synthesis

DBMS

Indexing

Processing

Hardware

Self-tuning

Self-driving

Self-generating

Imperial College London
Auto-Generating Databases

• Hardware-conscious modelling & tuning [ongoing]
• Auto-Generating Databases [future]
Wrapping up
Thanks to Collaborators

Sam Madden
Mike Stonebraker
Anil Shanbhag
Malte Schwarzkopf

Matei Zaharia
Shoumik Palkar

Imperial College London
A new DB research hub — Visit us!
Recruiting

- Interns
- PhDs
- Postdocs
Thank You
Backup slides
merge the sequence ACCCTA into the graph while minimizing the number of new branches

count the number of mutations of a gene

find the most common allele of a gene
Predicated Foreign-Key Joins

FoldSum(
    Gather(
        Gather(
            FoldSelect(x), y), z)
    )
=> FoldSum(
    Multiply(x, Gather(y, z)))
Double-predicated Foreign-Key Joins

FoldSum(
    Multiply(
        Gather(
            x, y), z))
=> FoldSum(
    Multiply(
        Gather(
            Multiply(x, z), y), z))
TPC-H Query 9 in Hyper

- Select (on dimension/target table) & Join Query
- HyPeR strategy: select scan + hash-join
  - Ignores foreign key index
- Voodoo strategy: select scan, build bitmap, join using FK-Index
  - Takes advantage of foreign key index
TPC-H ON GPUS

![Bar chart showing TPC-H performance on GPUs with two models: Voodoo and Ocelot. The x-axis represents the number of GPUs (1 to 19), and the y-axis represents time in milliseconds. The chart shows that Voodoo generally performs better than Ocelot for most configurations, with notable differences in performance between different numbers of GPUs.](chart.png)
Why Exploration?
Why not machine Learning?

• Because it takes ML-experts
• There often isn’t enough data (e.g., for catastrophic events)
• It is slow (in particular the training)
...when generating code

• LLVM
  • No Multithreading support
  • SIMD support is best-effort
  • No transactional memory support
  • No real GPU support
  • No NUMA support