Vectorization vs. Compilation

@Andy_Pavlo // 15-721 // Spring 2019
OBSERVATION

Vectorization can speed up query performance.
Compilation can speed up query performance.

We have not discussed which approach is better and under what conditions.
VECTORWISE – PRECOMPILED PRIMITIVES

Pre-compiles thousands of “primitives” that perform basic operations on typed data.
→ Using simple kernels for each primitive means that they are easier to vectorize.

The DBMS then executes a query plan that invokes these primitives at runtime.
→ Function calls are amortized over multiple tuples
HYPER – JIT QUERY COMPILATION

Compile queries in-memory into native code using the LLVM toolkit.

Organizes query processing in a way to keep a tuple in CPU registers for as long as possible.
→ Bottom-to-top / push-based query processing model.
→ Not vectorizable (as originally described).

EFFICIENTLY COMPILING EFFICIENT QUERY PLANS
FOR MODERN HARDWARE
VLDB 2011
TODAY’S AGENDA

Vectorization vs. Compilation
Relaxed Operator Fusion
VECTORIZATION VS. COMPILATION

Single test-bed system to analyze the trade-offs between vectorized execution and query compilation.

Implemented high-level algorithms the same in each system but varied the implementation details.

→ Example: Murmur2 vs. CRC Hash Functions
IMPLEMENTATIONS

Approach #1: Tectorwise
→ Break operations into pre-compiled primitives.
→ Have to materialize the output of primitives at each step.

Approach #2: Typer
→ Push-based processing model with JIT compilation.
→ Process a single tuple up entire pipeline without materializing the intermediate results.
TPC-H WORKLOAD

Q1: Fixed-point arithmetic, 4-group aggregation
Q6: Selective filters
Q3: Join (build: 147k tuples / probe: 3.2m tuples)
Q9: Join (build: 320k tuples / probe: 1.5M tuples)
Q18: High-cardinality aggregation (1.5m groups)
SINGLE-THREADED PERFORMANCE

Source: Timo Kersten
# Single-threaded Performance

<table>
<thead>
<tr>
<th>Q1</th>
<th>Typer</th>
<th>34</th>
<th>2.0</th>
<th>68</th>
<th>0.6</th>
<th>0.57</th>
<th>0.01</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TW</td>
<td>59</td>
<td>2.8</td>
<td>162</td>
<td>2.0</td>
<td>0.57</td>
<td>0.03</td>
</tr>
<tr>
<td>Q6</td>
<td>Typer</td>
<td>11</td>
<td>1.8</td>
<td>20</td>
<td>0.3</td>
<td>0.35</td>
<td>0.06</td>
</tr>
<tr>
<td></td>
<td>TW</td>
<td>11</td>
<td>1.4</td>
<td>15</td>
<td>0.2</td>
<td>0.29</td>
<td>0.01</td>
</tr>
<tr>
<td>Q3</td>
<td>Typer</td>
<td>25</td>
<td>0.8</td>
<td>21</td>
<td>0.5</td>
<td>0.16</td>
<td>0.27</td>
</tr>
<tr>
<td></td>
<td>TW</td>
<td>24</td>
<td>1.8</td>
<td>42</td>
<td>0.9</td>
<td>0.16</td>
<td>0.08</td>
</tr>
<tr>
<td>Q9</td>
<td>Typer</td>
<td>74</td>
<td>0.6</td>
<td>42</td>
<td>1.7</td>
<td>0.46</td>
<td>0.34</td>
</tr>
<tr>
<td></td>
<td>TW</td>
<td>56</td>
<td>1.3</td>
<td>76</td>
<td>2.1</td>
<td>0.47</td>
<td>0.39</td>
</tr>
<tr>
<td>Q18</td>
<td>Typer</td>
<td>30</td>
<td>1.6</td>
<td>46</td>
<td>0.8</td>
<td>0.19</td>
<td>0.16</td>
</tr>
<tr>
<td></td>
<td>TW</td>
<td>48</td>
<td>2.1</td>
<td>102</td>
<td>1.9</td>
<td>0.18</td>
<td>0.37</td>
</tr>
</tbody>
</table>
MAIN FINDINGS

Both models are efficient and achieve roughly the same performance.  
Data-centric is better for computational queries with few cache misses.  
Vectorization is slightly better at hiding cache miss latencies.
SIMD PERFORMANCE

Evaluate vectorized branchless selection and hash probe in Tectorwise.

They use AVX-512 because it includes new instructions to make it easier to implement algorithms using vertical vectorization.
SIMD EVALUATION

Cycles / Element

- **Hashing**
  - Scalar: 4
  - SIMD: 1
  - SIMD is 4x faster

- **Gather**
  - Scalar: 2
  - SIMD: 1.1
  - SIMD is 1.1x faster

- **Join**
  - Scalar: 10
  - SIMD: 1.4
  - SIMD is 1.4x faster

Time [ms]

- **q3**
  - Scalar: 40
  - SIMD: 36
  - SIMD is 1.1x faster

- **q9**
  - Scalar: 90
  - SIMD: 90
  - SIMD is 1.1x faster

Source: Timo Kersten
AUTO-VECTORIZATION

Measure how well the compiler is able to vectorize the Vectorwise primitives.
→ Targets: GCC v7.2, Clang v5.0, ICC v18

ICC was able to vectorize the most primitives using AVX-512:
→ Vectorized: Hashing, Selection, Projection
→ Not Vectorized: Hash Table Probing, Aggregation
**AUTO-VECTORIZATION**

*Intel Core i9-7900X (10 cores × 2HT)*

*Compiler: ICC v18*

<table>
<thead>
<tr>
<th>Quarter</th>
<th>Auto</th>
<th>Manual</th>
<th>Auto+Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>29.0</td>
<td>12.0</td>
<td>35.0</td>
</tr>
<tr>
<td>Q6</td>
<td>15.4</td>
<td>-1.01</td>
<td>15.4</td>
</tr>
<tr>
<td>Q3</td>
<td>27.2</td>
<td>31.5</td>
<td>46.6</td>
</tr>
<tr>
<td>Q9</td>
<td>62.5</td>
<td>82.6</td>
<td>82.9</td>
</tr>
<tr>
<td>Q18</td>
<td>42.0</td>
<td>60.1</td>
<td>61.2</td>
</tr>
</tbody>
</table>

Source: Timo Kersten
**AUTO-VECTORIZATION**

Intel Core i9-7900X (10 cores × 2HT)

Compiler: ICC v18

<table>
<thead>
<tr>
<th>Q1</th>
<th>Q6</th>
<th>Q3</th>
<th>Q9</th>
<th>Q18</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5</td>
<td>8.5</td>
<td>5.4</td>
<td>16.4</td>
<td>21.6</td>
</tr>
<tr>
<td>1.1</td>
<td>-0.3</td>
<td>0.3</td>
<td>11.0</td>
<td>15.7</td>
</tr>
</tbody>
</table>

Source: Timo Kersten
The paper (partially) assumes that vectorization and compilation are mutually exclusive.

HyPer fuses operators together so that they work on a single tuple a time to maximize CPU register reuse and minimize cache misses.
Figure 13: **Design space between vectorization and compilation** – *hybrid models integrate the advantages of the other approach.*

Source: Timo Kersten
PIPELINE PERSPECTIVE

Each pipeline **fuses** operators together into loop
Each pipeline is a **tuple-at-a-time** process

```python
def plan(state):
    agg = dict()
    for t in A:
        if t.age > 20:
            agg[t.city][\'count\'] +=
    for t in agg:
        emit(t)
```

Emit

\[\]

Agg

\[\]

Filter

\[\]

Scan
Each pipeline **fuses** operators together into loop
Each pipeline is a **tuple-at-a-time** process

```python
def plan(state):
    agg = dict()
    for t in A:
        if t.age > 20:
            agg[t.city][‘count’]++
    for t in agg:
        emit(t)
```
Fusion inhibits some optimizations:

→ Unable to look ahead in tuple stream.
→ Unable to overlap computation and memory access.

```python
def plan(state):
    agg = dict()
    for t in A:
        if t.age > 20:
            agg[t.city]["count"]++
    for t in agg:
        emit(t)
```

**FUSION PROBLEMS**
FUSION PROBLEMS

Fusion inhibits some optimizations:
→ Unable to look ahead in tuple stream.
→ Unable to overlap computation and memory access.

```python
def plan(state):
    agg = dict()
    for t in A:
        if t.age > 20:
            agg[t.city]['count']++
    for t in agg:
        emit(t)
```

\[\text{Scan} \rightarrow \text{Filter} \rightarrow \text{Agg}\]
Fusion inhibits some optimizations:
→ Unable to look ahead in tuple stream.
→ Unable to overlap computation and memory access.

```python
def plan(state):
    agg = dict()
    for t in A:
        if t.age > 20:
            agg[t.city]['count']++
    for t in agg:
        emit(t)
```

**FUSION PROBLEMS**

- **Cannot SIMD**
- **Cannot Prefetch**
RELAXED OPERATOR FUSION

Vectorized processing model designed for query compilation execution engines.

Decompose pipelines into stages that operate on vectors of tuples.
→ Each stage may contain multiple operators.
→ Communicate through cache-resident buffers.
→ Stages are granularity of vectorization + fusion.
ROF EXAMPLE

Vectorization Candidate

Emit

Agg

Filter

Scan

Emit

Agg

Filter

Scan
ROF EXAMPLE

Vectorization Candidate

Scan
Filter
Agg
Emit

Stage #1
Stage Buffer
Stage #2
def plan(state):
    agg = dict()
    for t in A step 1024:
        out = simd_cmp_gt(t, 20, 1024)
        for ft in out:
            agg[ft.city]["count"]++
    for t in agg:
        emit(t)
def plan(state):
    agg = dict()
    for t in A step 1024:
        out = simd_cmp_gt(t, 20, 1024)
        for ft in out:
            agg[ft.city]["count"]++
    for t in agg:
        emit(t)
def plan(state):
    agg = dict()
    for t in A step 1024:
        out = simd_cmp_gt(t, 20, 1024)
        for ft in out:
            agg[ft.city]['count']++
    for t in agg:
        emit(t)
```python
def plan(state):
    agg = dict()
    for t in A step 1024:
        out = simd_cmp_gt(t, 20, 1024)
        for ft in out:
            agg[ft.city]['count']++
    for t in agg:
        emit(t)
```
ROF SOFTWARE PREFETCHING

The DBMS can tell the CPU to grab the next vector while it works on the current batch.
→ Prefetch-enabled operators define start of new stage.
→ Hides the cache miss latency.

Any prefetching technique is suitable
→ Group prefetching, software pipelining, AMAC.
→ Group prefetching works and is simple to implement.
**ROF EVALUATION**

*Dual Socket Intel Xeon E5-2630v4 @ 2.20GHz TPC-H 10 GB Database*

- **LLVM**
- **LLVM + ROF**

<table>
<thead>
<tr>
<th></th>
<th>LLVM</th>
<th>LLVM + ROF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>901</td>
<td>892</td>
</tr>
<tr>
<td>Q3</td>
<td>1396</td>
<td>846</td>
</tr>
<tr>
<td>Q13</td>
<td>2641</td>
<td>1763</td>
</tr>
<tr>
<td>Q14</td>
<td>383</td>
<td>191</td>
</tr>
<tr>
<td>Q19</td>
<td>540</td>
<td>220</td>
</tr>
</tbody>
</table>

Execution Time (ms)

Source: Prashanth Menon
ROF EVALUATION

Dual Socket Intel Xeon E5-2630v4 @ 2.20GHz
TPC-H 10 GB Database

- LLVM
- LLVM + ROF

Source: Prashanth Menon
ROF EVALUATION – TPC-H Q19

Dual Socket Intel Xeon E5-2630v4 @ 2.20GHz
TPC-H 10 GB Database

<table>
<thead>
<tr>
<th>Method</th>
<th>Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpreted</td>
<td>21475</td>
</tr>
<tr>
<td>Compiled</td>
<td>568</td>
</tr>
<tr>
<td>ROF + SIMD</td>
<td>196</td>
</tr>
<tr>
<td>ROF + SIMD + Pretching</td>
<td>189</td>
</tr>
</tbody>
</table>

Source: Prashanth Menon
PARTING THOUGHTS

No major performance difference between the Vectorwise and HyPer approaches for all queries.

ROF combines vectorization and compilation into a hybrid query processing model.

→ Trades off additional instructions for reduced CPI
Query optimization is not rocket science.
When you flunk out of query optimization, we make you go build rockets.