Lecture #22 – Vectorized Execution (Part II)
CORRECTION

Original Xeon Phi (*Knights Corner*) uses **in-order execution** CPUs with a ring-based bus.

2016 Xeon Phi CPUs (*Knights Landing*) support out-of-order execution with tile architecture.

Three versions / form factors:
→ Self-boot Socket CPU
→ Self-boot Socket CPU + Fabric
→ PCI-E Device Co-Processor

Source: Avinash Sodani
CORRECTION

Original Xeon Phi (Knights Corner) uses in-order execution CPUs with a ring-based bus.

2016 Xeon Phi CPUs (Knights Landing) support out-of-order execution with tile architecture.

Three versions:
→ Self-boot Socket CPU
→ Self-boot Socket CPU + Fabric
→ PCI-E Device Co-Processor

Source: Avinash Sodani
TODAY’S AGENDA

Quickstep Bitweaving
HyPer Data Blocks
Project #3 Code Review Guidelines
The bit width of compressed data does not always fit naturally into SIMD register slots.

This means that the DBMS has to do extra work to transform data into the proper format.
BITWEAVING

Alternative storage layout for columnar databases that is designed for efficient predicate evaluation on compressed data using SIMD.
→ Order-preserving dictionary encoding.
→ Bit-level parallelization.
→ Only require common instructions (no scatter/gather)

BITWEAVING – STORAGE LAYOUTS

Approach #1: Horizontal
→ Row-oriented storage at the bit-level

Approach #2: Vertical
→ Column-oriented storage at the bit-level
HORIZONTAL STORAGE

Segment #1

$t_0$  0 0 1 = 1
$t_1$  1 0 1 = 5
$t_2$  1 1 0 = 6
$t_3$  0 0 1 = 1
$t_4$  1 1 0 = 6
$t_5$  1 0 0 = 4
$t_6$  0 0 0 = 0
$t_7$  1 1 1 = 7

Segment #2

$t_8$  1 0 0 = 4
$t_9$  0 1 1 = 3
HORIZONTAL STORAGE

Segment #1

Segment #2

Processor Word

Processor Word

CMU 15-721 (Spring 2017)
HORIZONTAL STORAGE

Segment #1
- $v_0$: 0 0 1 0 1 1 0
- $v_1$: 0 1 0 1 0 1 0 0
- $v_2$: 0 1 1 0 0 0 0 0
- $v_3$: 0 0 0 1 0 1 1 1

Segment #2
- $v_4$: 0 1 0 0 0 0 1 1

Delimiter
- Processor Word

$t_0$ $t_1$ $t_2$ $t_3$ $t_4$ $t_5$ $t_6$ $t_7$
$t_8$ $t_9$

Segment #1
Segment #2

Processor Word
SELECT * FROM table
WHERE val < 5

\[ X = \begin{bmatrix}
0 & 0 & 0 & 1 & 0 & 1 & 1 & 0
\end{bmatrix} \]

\[ Y = \begin{bmatrix}
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1
\end{bmatrix} \]
SELECT * FROM table
WHERE val < 5

\[X = \begin{array}{cccc}
0 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 \\
\end{array}\]

\[Y = \begin{array}{cccc}
0 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 \\
\end{array}\]

\[\text{mask} = \begin{array}{cccc}
0 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 \\
\end{array}\]

\[(Y + (X \oplus \text{mask})) \land \neg \text{mask} = \begin{array}{cccc}
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{array}\]

Source: Jignesh Patel
SELECT * FROM table
WHERE val < 5

X =
0 0 0 1 0 1 1 0

Y =
0 1 0 1 0 1 0 1

mask =
0 1 1 1 0 1 1 1

(Y + (X ⊕ mask)) ∧ ¬mask =
1 0 0 0 0 0 0 0

Only requires three instructions to evaluate a single word.

Works on any word size and encoding length.

Paper contains algorithms for other operators.

Source: Jignesh Patel
SELECT * FROM table
WHERE val < 5

Source: Jignesh Patel
SELECT * FROM table
WHERE val < 5
**BITWEAVING/H – EXAMPLE**

```
SELECT * FROM table
WHERE val < 5
```

Source: Jignesh Patel

---

Selected rows from table `v0`:

<table>
<thead>
<tr>
<th>t0</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
<th>t7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Selected rows from table `v1`:

<table>
<thead>
<tr>
<th>t0</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
<th>t7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Selected rows from table `v2`:

<table>
<thead>
<tr>
<th>t0</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
<th>t7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Selected rows from table `v3`:

<table>
<thead>
<tr>
<th>t0</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
<th>t7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

---

The diagram shows how the values are selected and how they are shifted to the right. The source of the diagram is Jignesh Patel.
SELECTION VECTOR

SIMD comparison operators produce a bit mask that specifies which tuples satisfy a predicate. Have to convert it into offsets / positions.

→ Approach #1: Iteration
→ Approach #2: Pre-compute Positions Table

```
tuples = [ ]
for (i=0; i<n; i++) {
    if sv[i] == 1
        tuples.add(i);
}
```
SIMD comparison operators produce a bit mask that specifies which tuples satisfy a predicate. Have to convert it into offsets / positions.

→ Approach #1: Iteration
→ Approach #2: Pre-compute Positions Table
### VERTICAL STORAGE

<table>
<thead>
<tr>
<th>Segment #1</th>
<th>Segment #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_0$</td>
<td>1 0 0 1</td>
</tr>
<tr>
<td>$t_1$</td>
<td>1 0 1</td>
</tr>
<tr>
<td>$t_2$</td>
<td>1 1 0</td>
</tr>
<tr>
<td>$t_3$</td>
<td>0 0 1</td>
</tr>
<tr>
<td>$t_4$</td>
<td>1 1 0</td>
</tr>
<tr>
<td>$t_5$</td>
<td>1 0 0</td>
</tr>
<tr>
<td>$t_6$</td>
<td>0 0 0</td>
</tr>
<tr>
<td>$t_7$</td>
<td>1 1 1</td>
</tr>
<tr>
<td>$t_8$</td>
<td>1 0 0</td>
</tr>
<tr>
<td>$t_9$</td>
<td>0 1 1</td>
</tr>
</tbody>
</table>
**BITWEAVING/V – EXAMPLE**

**SELECT * FROM table WHERE key = 2**

```
0 1 0
```

**Segment #1**

```
<table>
<thead>
<tr>
<th></th>
<th>t0</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
<th>t7</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>v1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>v2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```
SELECT * FROM table
WHERE key = 2

Segment #1

<table>
<thead>
<tr>
<th>t_0</th>
<th>t_1</th>
<th>t_2</th>
<th>t_3</th>
<th>t_4</th>
<th>t_5</th>
<th>t_6</th>
<th>t_7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>v_0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v_1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v_2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>v_0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>v_1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>v_2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SIMD Compare</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 1 0 0 1 0</td>
</tr>
</tbody>
</table>
SELECT * FROM table
WHERE key = 2

Segment #1

<table>
<thead>
<tr>
<th></th>
<th>t_0</th>
<th>t_1</th>
<th>t_2</th>
<th>t_3</th>
<th>t_4</th>
<th>t_5</th>
<th>t_6</th>
<th>t_7</th>
</tr>
</thead>
<tbody>
<tr>
<td>v_0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>v_1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>v_2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

SIMD Compare

1 0 0 1 0 0 1 0

0 0 0 0 0 0 0 0
**BITWEAVING/V – EXAMPLE**

```
SELECT * FROM table
WHERE key = 2
```

---

**Segment #1**

<table>
<thead>
<tr>
<th></th>
<th>t0</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
<th>t7</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>v1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>v2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Can perform early pruning just like in BitMap indexes.

The last vector is skipped because all bits in previous comparison are zero.
EVALUATION

Single-threaded execution of a single query derived from TPC-H benchmark.
→ Selectivity: 10%

10GB TPC-H Database
→ 1 billion tuples
→ Uniform distribution

Source: Jignesh Patel

SELECT COUNT(*)
FROM R
WHERE R.a < C
EVALUATION

TPC-H Aggregation Query
Intel Xeon X5650 @ 2.66 GHz

Naïve  SIMD Scan  BitWeaving/V  BitWeaving/H

- Fewer Cache Misses
- SIMD Parallelization
- Early Pruning

Source: Jignesh Patel
Carnegie Mellon University (Spring 2017)
OBSERVATION

At a high-level, BitWeaving is essentially a bitmap index. This is great for OLAP, bad for OLTP.

How can compress data in such a way that still allow for efficient point queries?
Individually determine the best suitable compression scheme per block.
→ Hot tuples are stored uncompressed (OLTP)
→ Cold tuples are stored in compressed Data Blocks (OLAP)

Each Data Block is compressed using the best scheme for its tuples.
Multiple storage layouts and compression schemes mean the DBMS has to generate multiple code paths per query plan.

This increases the compilation time per query and the total size of the query plan cache.
Compressed Data: Vectorized Execution
→ Use the pre-compiled primitives from Vectorwise.

Uncompressed Data: LLVM Compilation
→ Data is always stored in the same format so there is only ever one code path to compile.
PARTING THOUGHTS

Just like in query compilation, getting the best performance with vectorization requires the DBMS to store data in a way that is best for the CPU and not the best for humans’ understanding.
ADMINISTRIVIA

Code Review #1 Submission: April 11th
Project Status Meetings: April 13th
Project Status Updates: April 18th
Code Review #1 Completion: April 18th
CODE REVIEWS

Each group will send a pull request to the CMU-DB master branch.
→ This will automatically run tests + coverage calculation.
→ PR must be able to merge cleanly into master branch.
→ Reviewing group will write comments on that request.
→ Add the URL to the Google spreadsheet and notify the reviewing team that it is ready.

Please be helpful and courteous.
GENERAL TIPS

The dev team should provide you with a summary of what files/functions the reviewing team should look at.

Review fewer than 400 lines of code at a time and only for at most 60 minutes.

Use a checklist to outline what kind of problems you are looking for.
CHECKLIST – GENERAL

Does the code work?
Is all the code easily understood?
Is there any redundant or duplicate code?
Is the code as modular as possible?
Can any global variables be replaced?
Is there any commented out code?
Is it using proper debug log functions?
Do comments describe the intent of the code?
Are all functions commented?
Is any unusual behavior described?
Is the use of 3rd-party libraries documented?
Is there any incomplete code?
CHECKLIST – TESTING

Do tests exist and are they comprehensive?
Are the tests actually testing the feature?
Are they relying on hardcoded answers?
What is the code coverage?

Source: Gareth Wilson
PROJECT #3 – STATUS UPDATE

I will meet with groups individually on Thursday.

Next Tuesday each group will give a five minute presentation to update the class about the current status of your project.

Each presentation should include:
→ Current development status.
→ Whether anything in your plan has changed.
→ Any thing that surprised you.