Rethinking SIMD Vectorization for In-Memory Databases

Sri Harshal Parimi
Motivation

- Need for fast analytical query execution in systems where the database is mostly resident in main memory.

- Architectures with SIMD capabilities, like (Many Integrated cores)MIC, use a large number of low-powered cores with advanced instruction sets and larger registers.
SIMD (Single Instruction, Multiple Data)

- Multiple processing elements that perform the same operation on multiple data points simultaneously.
Vectorization

- Program that performs operations on a vector (1D-array).

\[ X + Y = Z \]

\[(x_1 \ x_2 \ \ldots \ x_n) + (y_1 \ y_2 \ \ldots \ y_n) = (x_1 + y_1 \ x_2 + y_2 \ \ldots \ x_n + y_n)\]

for \(i = 0; i < n; i++\) {
    \[Z[i] = X[i] + Y[i];\]
}
Vectorization (Example)

128 bit SIMD register

SIMD ADD

<table>
<thead>
<tr>
<th>Z</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Advantages of Vectorization

- Full vectorization
  - From $O(f(n))$ scalar to $O(f(n)/W)$ vector operations where $W$ is the length of the vector.
  - Reuse fundamental operations across multiple vectorizations.

- Vectorize basic database operators:
  - Selection scans
  - Hash tables
  - Partitioning
Fundamental Operations

- Selective Load
- Selective Store
- Selective Gather
- Selective Scatter
Selective Load

Vector: A B C D
Mask: 0 1 0 1
Memory: U V W X Y
Result Vector: A U C V

Selective Store

Memory: U V W X Y
Mask: 0 1 0 1
Vector: A B C D
Result Memory: B D W X Y
Selective Gather

Value Vector
A B A D

Index Vector
2 1 5 3

Memory
U V W X Y Z

Value Vector
W V Z X

Selective Scatter

Memory
U V W X Y Z

Index Vector
2 1 5 3

Value Vector
A B C D

Memory
U B A D Y C
Selection Scans

Scalar(Branching):
- \( I = 0 \)
- For \( t \) in table:
  - If(\((key>= \text{“O”} \&\& key<=\text{“U”})\)):
    - Copy(\( t \), output[\( i \)]);
    - \( I = I + 1 \);

Scalar(Branchless):
- \( I = 0 \)
- For \( t \) in table:
  - Key = \( t.key \)
  - \( M = (key>=\text{“O”}?1:0)\&\&(key<=\text{“U”}?1:0)\);
  - \( I = I + M \);

SELECT * FROM table WHERE key \( \geq \)“O” AND key \( \leq \)“U”
Selection Scans (Vectorized)

- \( I = 0 \)
- For \( V_t \) in table:
  - \( \text{simdLoad}(V_t.key, V_k) \)
  - \( V_m = (V_k \geq "O" \&\& V_k \leq "U") \)
  - If \( V_m \neq \text{false} \):
    - \( \text{simdStore}(V_t, V_m, \text{output}[i]) \)
    - \( I = I + | V_m \neq \text{false} | \)
Performance Comparison: Selection Scans
Hash Tables – Probing (Scalar)

Scalar

Input key → Hash(key) → Hash Index

k1 # h1

k9
k3
k1
k1

Linear probing hash table

<table>
<thead>
<tr>
<th>Key</th>
<th>Payload</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hash Tables – Probing (Horizontal Vectorization)

- Input key
- Hash(key)
- Hash Index

Linear probing bucketized hash table

<table>
<thead>
<tr>
<th>KEYS</th>
<th>PAYLOAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>K9</td>
<td></td>
</tr>
<tr>
<td>K3</td>
<td></td>
</tr>
<tr>
<td>K8</td>
<td></td>
</tr>
<tr>
<td>K1</td>
<td></td>
</tr>
</tbody>
</table>

SIMD Compare
Hash Tables – Probing (Vertical Vectorization)

<table>
<thead>
<tr>
<th>Key Vec</th>
<th>Hash(key)</th>
<th>Hash Index Vec</th>
<th>Gathered Key Vec</th>
<th>Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>K1</td>
<td>#</td>
<td></td>
<td>K1</td>
<td>1</td>
</tr>
<tr>
<td>K2</td>
<td>#</td>
<td>H2</td>
<td>K2</td>
<td>0</td>
</tr>
<tr>
<td>K3</td>
<td>#</td>
<td>H3</td>
<td>K3</td>
<td>0</td>
</tr>
<tr>
<td>K4</td>
<td>#</td>
<td>H4</td>
<td>K4</td>
<td>1</td>
</tr>
</tbody>
</table>

SIMD Compare

<table>
<thead>
<tr>
<th>Key</th>
<th>Payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>K99</td>
<td></td>
</tr>
<tr>
<td>K1</td>
<td></td>
</tr>
<tr>
<td>K4</td>
<td></td>
</tr>
<tr>
<td>K88</td>
<td></td>
</tr>
</tbody>
</table>
## Hash Tables – Probing (Vertical Vectorization Continued)

<table>
<thead>
<tr>
<th>Key Vec</th>
<th>Hash(key)</th>
<th>Hash Index Vec</th>
</tr>
</thead>
<tbody>
<tr>
<td>K5</td>
<td>#</td>
<td>H5</td>
</tr>
<tr>
<td>K2</td>
<td>#</td>
<td>H2 + 1</td>
</tr>
<tr>
<td>K3</td>
<td>#</td>
<td>H3 + 1</td>
</tr>
<tr>
<td>K6</td>
<td>#</td>
<td>H6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>K99</td>
<td></td>
</tr>
<tr>
<td>K2</td>
<td></td>
</tr>
<tr>
<td>K1</td>
<td></td>
</tr>
<tr>
<td>K5</td>
<td></td>
</tr>
<tr>
<td>K4</td>
<td></td>
</tr>
<tr>
<td>K6</td>
<td></td>
</tr>
<tr>
<td>K88</td>
<td></td>
</tr>
</tbody>
</table>
Performance Comparison: Hash Tables
Partitioning - Histogram

Key Vec
- K1
- K2
- K3
- K4

Hash Index Vec
- H1
- H2
- H3
- H4

Histogram
- +1
- +1
- +1

SIMD Radix
SIMD Add
Partitioning – Histogram (Continued)

<table>
<thead>
<tr>
<th>Key Vec</th>
<th>Hash Index Vec</th>
<th>Replicated Histogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>K1</td>
<td>H1</td>
<td>+1</td>
</tr>
<tr>
<td>K2</td>
<td>H2</td>
<td>+1</td>
</tr>
<tr>
<td>K3</td>
<td>H3</td>
<td>+1</td>
</tr>
<tr>
<td>K4</td>
<td>H4</td>
<td>+1</td>
</tr>
</tbody>
</table>

SIMD Radix

SIMD Scatter
Joins

- No partitioning
  - Build one shared hash table using atomics
  - Partially vectorized

- Min partitioning
  - Partition building table
  - Build hash table per thread
  - Fully vectorized

- Max partitioning
  - Partition both tables repeatedly
  - Build and probe `cache-resident` hash tables
  - Fully vectorized
Joins

![Diagram showing join time for different partitioning strategies.](image)

- 200M x 200M tuples (32-bit keys & payloads)
- Xeon Phi 7120P – 61 Cores + 4xHT
- Join Time (sec) for:
  - Partition
  - Build
  - Probe
  - Build+Probe
  - Scalar
  - Vector
  - No Partitioning
  - Min Partitioning
  - Max Partitioning
Main Takeaways

- Vectorization is essential for OLAP queries
- Impact on hardware design
  - Improved power efficiency for analytical databases
- Impact on software design
  - Vectorization favors cache-conscious algorithms
    - Partitioned hash join >> non-partitioned hash join, if vectorized
  - Vectorization is independent of other optimizations
    - Both buffered and unbuffered partitioning benefit from vectorization speedup
Comparisons with Trill

- Trill uses a similar bit-mask technique for applying the filter clause during selections.
- While Trill deals with a query model for streaming data, this paper offers algorithms that can improve throughput of database operators which can also be extended to a streaming model by leveraging buffered data.
- Trill uses dynamic HLL code-generation to operate over columnar data. SIMD provides vectorization to handle data-points simultaneously and has a diverse instruction set (supported by H/W) to perform constant operations on vectors.
Questions?