TODAY’S AGENDA

Background
Hardware
Vectorized Algorithms (Columbia)
VECTORIZATION

The process of converting an algorithm's scalar implementation that processes a single pair of operands at a time, to a vector implementation that processes one operation on multiple pairs of operands at once.
WHY THIS MATTERS

Say we can parallelize our algorithm over 32 cores. Each core has a 4-wide SIMD registers.

Potential Speed-up: \(32 \times 4 = 128\)
Use a small number of high-powered cores.
→ Intel Xeon Skylake / Kaby Lake
→ High power consumption and area per core.

Massively **superscalar** and aggressive **out-of-order** execution
→ Instructions are issued from a sequential stream.
→ Check for dependencies between instructions.
→ Process multiple instructions per clock cycle.
MANY INTEGRATED CORES (MIC)

Use a larger number of low-powered cores.
→ Intel Xeon Phi
→ Low power consumption and area per core.
→ Expanded SIMD instructions with larger register sizes.

**Knights Ferry (Columbia Paper)**
→ Non-superscalar and in-order execution
→ Cores = Intel P54C (aka Pentium from the 1990s).

**Knights Landing (Since 2016)**
→ Superscalar and out-of-order execution.
→ Cores = Silvermont (aka Atom)
MANY INTEGRATED CORES (MIC)

Use a larger number of low-powered cores.
- Expanded SIMD instructions with larger register sizes.

Knights Ferry (Columbia Paper)
- Non-superscalar and in-order execution
- Cores = Intel P54C (aka Pentium from the 1990s).

Knights Landing (Since 2016)
- Superscalar and out-of-order execution.
- Cores = Silvermont (aka Atom)
SINGLE INSTRUCTION, MULTIPLE DATA

A class of CPU instructions that allow the processor to perform the same operation on multiple data points simultaneously.

All major ISAs have microarchitecture support SIMD operations.

→ **x86**: MMX, SSE, SSE2, SSE3, SSE4, AVX, AVX2, AVX512
→ **PowerPC**: Altivec
→ **ARM**: NEON
SIMD EXAMPLE

\[ X + Y = Z \]

\[
\begin{bmatrix}
  x_1 \\
x_2 \\
  \vdots \\
x_n \\
\end{bmatrix}
+ 
\begin{bmatrix}
  y_1 \\
y_2 \\
  \vdots \\
y_n \\
\end{bmatrix}
= 
\begin{bmatrix}
  x_1 + y_1 \\
x_2 + y_2 \\
  \vdots \\
x_n + y_n \\
\end{bmatrix}
\]

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

\[ X \]

\[
\begin{bmatrix}
  8 \\
  7 \\
  6 \\
  5 \\
\end{bmatrix}
\]

\[ Y \]

\[
\begin{bmatrix}
  1 \\
  1 \\
  1 \\
  1 \\
\end{bmatrix}
\]

\[ Z \]

\[
\begin{bmatrix}
  1 \\
  1 \\
  1 \\
  1 \\
\end{bmatrix}
\]
SIMD EXAMPLE

\[
X + Y = Z
\]

\[
\begin{bmatrix}
x_1 \\
x_2 \\
\vdots \\
x_n \\
\end{bmatrix} + \begin{bmatrix}
y_1 \\
y_2 \\
\vdots \\
y_n \\
\end{bmatrix} = \begin{bmatrix}
x_1 + y_1 \\
x_2 + y_2 \\
\vdots \\
x_n + y_n \\
\end{bmatrix}
\]

\[
\text{for } (i=0; i<n; i++) \{ \\
Z[i] = X[i] + Y[i]; \\
\}
\]
**SIMD EXAMPLE**

\[ X + Y = Z \]

\[
\begin{bmatrix}
  x_1 \\
  x_2 \\
  \vdots \\
  x_n \\
\end{bmatrix}
+ 
\begin{bmatrix}
  y_1 \\
  y_2 \\
  \vdots \\
  y_n \\
\end{bmatrix}
= 
\begin{bmatrix}
  x_1+y_1 \\
  x_2+y_2 \\
  \vdots \\
  x_n+y_n \\
\end{bmatrix}
\]

\[
\text{for (i=0; i<n; i++)} \\
\text{Z[i] = X[i] + Y[i];}
\]
**SIMD EXAMPLE**

\[ X + Y = Z \]

\[
\begin{bmatrix}
  x_1 \\
  x_2 \\
  \vdots \\
  x_n \\
\end{bmatrix}
+ 
\begin{bmatrix}
  y_1 \\
  y_2 \\
  \vdots \\
  y_n \\
\end{bmatrix}
= 
\begin{bmatrix}
  x_1 + y_1 \\
  x_2 + y_2 \\
  \vdots \\
  x_n + y_n \\
\end{bmatrix}
\]

```
for (i=0; i<n; i++) {
  Z[i] = X[i] + Y[i];
}
```
**SIMD EXAMPLE**

\[ X + Y = Z \]

\[
\begin{bmatrix}
  x_1 \\
  x_2 \\
  \vdots \\
  x_n \\
\end{bmatrix} +
\begin{bmatrix}
  y_1 \\
  y_2 \\
  \vdots \\
  y_n \\
\end{bmatrix} =
\begin{bmatrix}
  x_1+y_1 \\
  x_2+y_2 \\
  \vdots \\
  x_n+y_n \\
\end{bmatrix}
\]

**C code example:**

```
for (i=0; i<n; i++) {
    Z[i] = X[i] + Y[i];
}
```
STREAMING SIMD EXTENSIONS (SSE)

SSE is a collection SIMD instructions that target special 128-bit SIMD registers. These registers can be packed with four 32-bit scalars after which an operation can be performed on each of the four elements simultaneously.

First introduced by Intel in 1999.
SIMD INSTRUCTIONS (1)

Data Movement
→ Moving data in and out of vector registers

Arithmetic Operations
→ Apply operation on multiple data items (e.g., 2 doubles, 4 floats, 16 bytes)
→ Example: ADD, SUB, MUL, DIV, SQRT, MAX, MIN

Logical Instructions
→ Logical operations on multiple data items
→ Example: AND, OR, XOR, ANDN, ANDPS, ANDNPS
SIMD INSTRUCTIONS (2)

Comparison Instructions
→ Comparing multiple data items (==,<,<=,>,>=,!=)

Shuffle instructions
→ Move data in between SIMD registers

Miscellaneous
→ Conversion: Transform data between x86 and SIMD registers.
→ Cache Control: Move data directly from SIMD registers to memory (bypassing CPU cache).
# INTEL SIMD EXTENSIONS

<table>
<thead>
<tr>
<th>Year</th>
<th>Extension</th>
<th>Width</th>
<th>Integers</th>
<th>Single-P</th>
<th>Double-P</th>
</tr>
</thead>
<tbody>
<tr>
<td>1997</td>
<td>MMX</td>
<td>64 bits</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1999</td>
<td>SSE</td>
<td>128 bits</td>
<td>✔</td>
<td>✔ (×4)</td>
<td></td>
</tr>
<tr>
<td>2001</td>
<td>SSE2</td>
<td>128 bits</td>
<td>✔</td>
<td>✔</td>
<td>✔ (×2)</td>
</tr>
<tr>
<td>2004</td>
<td>SSE3</td>
<td>128 bits</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>2006</td>
<td>SSSE 3</td>
<td>128 bits</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>2006</td>
<td>SSE 4.1</td>
<td>128 bits</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>2008</td>
<td>SSE 4.2</td>
<td>128 bits</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>2011</td>
<td>AVX</td>
<td>256 bits</td>
<td>✔</td>
<td>✔ (×8)</td>
<td>✔ (×4)</td>
</tr>
<tr>
<td>2013</td>
<td>AVX2</td>
<td>256 bits</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>2017</td>
<td>AVX-512</td>
<td>512 bits</td>
<td>✔</td>
<td>✔ (×16)</td>
<td>✔ (×8)</td>
</tr>
</tbody>
</table>

Source: [James Reinders](#)
WHY NOT GPUs?

Moving data back and forth between DRAM and GPU is slow over PCI-E bus.

There are some newer GPU-enabled DBMSs
→ Examples: MapD, SQream, Kinetica

Emerging co-processors that can share CPU’s memory may change this.
→ Examples: AMD’s APU, Intel’s Knights Landing
HARDWARE ACCELERATED DATABASE LECTURES

Fall 2018 • Thursdays @ 12:00pm • CIC 4th Floor

https://db.cs.cmu.edu/seminar2018
VECTORIZATION

Choice #1: Automatic Vectorization
Choice #2: Compiler Hints
Choice #3: Explicit Vectorization
AUTOMATIC VECTORIZATION

The compiler can identify when instructions inside of a loop can be rewritten as a vectorized operation.

Works for simple loops only and is rare in database operators. Requires hardware support for SIMD instructions.
This loop is not legal to automatically vectorize.

The code is written such that the addition is described as being done sequentially.

These might point to the same address!
COMPILER HINTS

Provide the compiler with additional information about the code to let it know that is safe to vectorize.

Two approaches:
→ Give explicit information about memory locations.
→ Tell the compiler to ignore vector dependencies.
COMPILER HINTS

The \texttt{restrict} keyword in C++ tells the compiler that the arrays are distinct locations in memory.

```c
void add(int *restrict X,
         int *restrict Y,
         int *restrict Z) {
    for (int i=0; i<MAX; i++) {
        Z[i] = X[i] + Y[i];
    }
}
```
void add(int *X,
    int *Y,
    int *Z) {
    #pragma ivdep
    for (int i=0; i<MAX; i++) {
        Z[i] = X[i] + Y[i];
    }
}
EXPLICIT VECTORIZATION

Use CPU intrinsics to manually marshal data between SIMD registers and execute vectorized instructions.

Potentially not portable.
EXPLICIT VECTORIZATION

Store the vectors in 128-bit SIMD registers.

Then invoke the intrinsic to add together the vectors and write them to the output location.

```c
void add(int *X,
    int *Y,
    int *Z) {
    __m128i *vecX = (__m128i*)X;
    __m128i *vecY = (__m128i*)Y;
    __m128i *vecZ = (__m128i*)Z;
    for (int i=0; i<MAX/4; i++) {
        _mm_store_si128(vecZ++,
            _mm_add_epi32(*vecX, *vecY));
    }
}
```
VECTORIZATION DIRECTION

Approach #1: Horizontal
→ Perform operation on all elements together within a single vector.

Approach #2: Vertical
→ Perform operation in an elementwise manner on elements of each vector.

Source: Przemysław Karpiński
EXPLICIT VECTORIZATION

Linear Access Operators
→ Predicate evaluation
→ Compression

Ad-hoc Vectorization
→ Sorting
→ Merging

Composable Operations
→ Multi-way trees
→ Bucketized hash tables

Source: Orestis Polychroniou
VECTORIZED DBMS ALGORITHMS

Principles for efficient vectorization by using fundamental vector operations to construct more advanced functionality.
→ Favor vertical vectorization by processing different input data per lane.
→ Maximize lane utilization by executing different things per lane subset.
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 Z • • •
FUNDAMENTAL VECTOR OPERATIONS

Selective Load

Vector: A B C D

Mask: 0 1 0 1

Memory: U V W X Y Z • • •
FUNDAMENTAL VECTOR OPERATIONS

Selective Load

Vector: A U C D

Mask: 0 1 0 1

Memory: U V W X Y Z • • •
Selective Load

Vector: $\begin{bmatrix} A & U & C & D \end{bmatrix}$

Mask: $\begin{bmatrix} 0 & 1 & 0 & 1 \end{bmatrix}$

Memory: $\begin{bmatrix} U & V & W & X & Y & Z & \ldots \end{bmatrix}$
Selective Load

Vector: A U C V

Mask: 0 1 0 1

Memory: U V W X Y Z ...
**Selective Load**

Vector: \( \text{U} \quad \text{V} \quad \text{C} \quad \text{V} \)

Mask: \( \text{0} \quad \text{1} \quad \text{0} \quad \text{1} \)

Memory: \( \text{U} \quad \text{V} \quad \text{W} \quad \text{X} \quad \text{Y} \quad \text{Z} \)

---

**Selective Store**

Memory: \( \text{U} \quad \text{V} \quad \text{W} \quad \text{X} \quad \text{Y} \quad \text{Z} \)

Mask: \( \text{0} \quad \text{1} \quad \text{0} \quad \text{1} \)

Vector: \( \text{A} \quad \text{B} \quad \text{C} \quad \text{D} \)
Selective Load

Vector: \( \begin{bmatrix} A & U & C & V \end{bmatrix} \)

Mask: \( \begin{bmatrix} 0 & 1 & 0 & 1 \end{bmatrix} \)

Memory: \( \begin{bmatrix} U & V & W & X & Y & Z \end{bmatrix} \)

Selective Store

Memory: \( \begin{bmatrix} U & V & W & X & Y & Z \end{bmatrix} \)

Mask: \( \begin{bmatrix} 0 & 1 & 0 & 1 \end{bmatrix} \)

Vector: \( \begin{bmatrix} A & B & C & D \end{bmatrix} \)
FUNDAMENTAL VECTOR OPERATIONS

Selective Load

<table>
<thead>
<tr>
<th>Vector</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>V</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101</td>
</tr>
</tbody>
</table>

Selective Store

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
</tr>
<tr>
<td>V</td>
</tr>
<tr>
<td>W</td>
</tr>
<tr>
<td>X</td>
</tr>
<tr>
<td>Y</td>
</tr>
<tr>
<td>Z</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>D</td>
</tr>
</tbody>
</table>
**Selective Load**

- **Vector**:
  
  | A | U | C | V |

- **Mask**:
  
  | 0 | 1 | 0 | 1 |

- **Memory**:
  
  | U | V | W | X | Y | Z | ... |

**Selective Store**

- **Memory**:
  
  | B | V | W | X | Y | Z | ... |

- **Mask**:
  
  | 0 | 1 | 0 | 1 |

- **Vector**:
  
  | A | B | C | D |
**FUNDAMENTAL VECTOR OPERATIONS**

### Selective Load

- **Vector**: \( \begin{pmatrix} A & U & C & V \end{pmatrix} \)
- **Mask**: \( \begin{pmatrix} 0 & 1 & 0 & 1 \end{pmatrix} \)
- **Memory**: \( \begin{pmatrix} U & V & W & X & Y & Z \ldots \end{pmatrix} \)

### Selective Store

- **Memory**: \( \begin{pmatrix} B & D & W & X & Y & Z \ldots \end{pmatrix} \)
- **Mask**: \( \begin{pmatrix} 0 & 1 & 0 & 1 \end{pmatrix} \)
- **Vector**: \( \begin{pmatrix} A & B & C & D \end{pmatrix} \)
Selective Gather

Value Vector

Index Vector

Memory
FUNDAMENTAL VECTOR OPERATIONS

Selective Gather

Value Vector: W B A D

Index Vector: 2 1 5 3

Memory: U V W X Y Z...
FUNDAMENTAL VECTOR OPERATIONS

Selective Gather

Value Vector

Index Vector

Memory

W V Z X

2 1 5 3

U V W X Y Z

0 1 2 3 4 5
Selective Gather

**Value Vector**

```
W V Z X
```

**Index Vector**

```
2 1 5 3
```

**Memory**

```
U V W X Y Z ...
```

Selective Scatter

**Value Vector**

```
A B C D
```

**Index Vector**

```
2 1 5 3
```

**Memory**

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

**Value Vector**

```
A B C D
```
### Selective Gather

- **Value Vector**: W, V, Z, X
- **Index Vector**: 2, 1, 5, 3
- **Memory**: U, V, W, X, Y, Z

### Selective Scatter

- **Memory**: U, B, A, D, Y, C
- **Index Vector**: 2, 1, 5, 3
- **Value Vector**: A, B, C, D
ISSUES

Gathers and scatters are not really executed in parallel because the L1 cache only allows one or two distinct accesses per cycle.

Gathers are only supported in newer CPUs. Selective loads and stores are also emulated in Xeon CPUs using vector permutations.
VECTORIZED OPERATORS

Selection Scans
Hash Tables
Partitioning

Paper provides additional info:
→ Joins, Sorting, Bloom filters.
SELECT * FROM table
WHERE key >= $(low)
AND key <= $(high)
**SELECTION SCANS**

**Scalar (Branching)**

```python
i = 0
for t in table:
    key = t.key
    if (key >= low) and (key <= high):
        copy(t, output[i])
        i = i + 1
```
**SELECTION SCANS**

### Scalar (Branching)

\[
i = 0 \\
\text{for } t \text{ in table:} \\
\quad \text{key} = t.\text{key} \\
\quad \textbf{if (key} \geq \text{low} \text{) } \&\& \text{ (key} \leq \text{high):} \\
\quad \quad \text{copy}(t, \text{output}[i]) \\
\quad \quad i = i + 1
\]

### Scalar (Branchless)

\[
i = 0 \\
\text{for } t \text{ in table:} \\
\quad \text{copy}(t, \text{output}[i]) \\
\quad \text{key} = t.\text{key} \\
\quad m = (\text{key} \geq \text{low } ? 1 : 0) \&\& \\
\quad \quad (\text{key} \leq \text{high } ? 1 : 0) \\
\quad i = i + m
\]
**SELECTION SCANS**

Scalar (Branching)

\[
i = 0 \\
\text{for } t \text{ in table} \\
\text{key = } t.\text{key} \\
\text{if } (\text{key} \geq \text{low}) \land (\text{key} \leq \text{high}) : \\
\text{copy}(t, \text{output}[i]) \\
i = i + 1
\]

Scalar (Branchless)

\[
i = 0 \\
\text{for } t \text{ in table} \\
\text{key = } t.\text{key} \\
m = (\text{key} \geq \text{low}) ? 1 : 0 \\
\Rightarrow (\text{key} \leq \text{high}) ? 1 : 0 \\
i = i + m
\]

Source: Bogdan Raducanu
i = 0
for \( v_t \) in table:
    \( \text{simdLoad}(v_t\text{.key}, v_k) \)
    \( v_m = (v_k \geq \text{low} \ ? \ 1 : 0) \land \quad (v_k \leq \text{high} \ ? \ 1 : 0) \)
    \( \text{simdStore}(v_t, v_m, \text{output}[i]) \)
    \( i = i + |v_m \neq \text{false}| \)
Vectorized

```
i = 0
for vt in table:
    simdLoad(vt.key, v_k)
    v_m = (v_k>=low ? 1 : 0) &&
          (v_k<=high ? 1 : 0)
    simdStore(v_t, v_m, output[i])
i = i + |v_m#false|
```

SELECT * FROM table
WHERE key >= "O" AND key <= "U"
Vectorized

i = 0
for $v_t$ in table:
    simdLoad($v_t$.key, $v_k$)
    $v_m = (v_k \geq \text{low} \ ? \ 1 : 0) \&\&$
    (v_k \leq \text{high} \ ? \ 1 : 0)$
    simdStore($v_t$, $v_m$, output[i])
    i = i + $|v_m\neq\text{false}|$

SELECT * FROM table
WHERE key >= "O" AND key <= "U"
Vectorized

Vectorized

```
i = 0
for vt in table:
    simdLoad(vt.key, vk)
    vm = (vk>low ? 1 : 0) &&
         (vk<high ? 1 : 0)
    simdStore(vt, vm, output[i])
    i = i + |vm!=false|
```

**SELECT** * FROM table
**WHERE** key >= "O" AND key <= "U"

**Key Vector**

<table>
<thead>
<tr>
<th>ID</th>
<th>KEY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>J</td>
</tr>
<tr>
<td>2</td>
<td>O</td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
</tr>
<tr>
<td>4</td>
<td>S</td>
</tr>
<tr>
<td>5</td>
<td>U</td>
</tr>
<tr>
<td>6</td>
<td>X</td>
</tr>
</tbody>
</table>

**Mask**

| 0 | 1 | 0 | 1 | 1 | 1 | 0 |

**SIMD Compare**

**All Offsets**

| 0 | 1 | 2 | 3 | 4 | 5 |
Vectorized

\[
i = 0 \\
\text{for } v_t \text{ in table:} \\
\quad \text{simdLoad}(v_t.key, v_k) \\
\quad v_m = (v_k \geq \text{low} \ ? \ 1 : 0) \land \neg (v_k \leq \text{high} \ ? \ 1 : 0) \\
\quad \text{simdStore}(v_t, v_m, \text{output}[i]) \\
\quad i = i + \mid v_m \neq \text{false} \mid
\]

**SELECT** * FROM table  
**WHERE** key >= "0" AND key <= "U"
### SELECTION SCANS

- **Scalar (Branching)**
- **Scalar (Branchless)**
- **Vectorized (Early Mat)**
- **Vectorized (Late Mat)**

**MIC (Xeon Phi 7120P – 61 Cores + 4×HT)**

**Multi-Core (Xeon E3-1275v3 – 4 Cores + 2×HT)**

Throughput (billions tuples/sec)

<table>
<thead>
<tr>
<th>Selectivity (%)</th>
<th>MIC</th>
<th>Multi-Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5.7</td>
<td>1.8</td>
</tr>
<tr>
<td>1</td>
<td>5.7</td>
<td>1.7</td>
</tr>
<tr>
<td>2</td>
<td>5.6</td>
<td>1.7</td>
</tr>
<tr>
<td>5</td>
<td>5.3</td>
<td>1.7</td>
</tr>
<tr>
<td>10</td>
<td>4.9</td>
<td>1.7</td>
</tr>
<tr>
<td>20</td>
<td>4.3</td>
<td>1.7</td>
</tr>
<tr>
<td>50</td>
<td>2.8</td>
<td>1.6</td>
</tr>
<tr>
<td>100</td>
<td>1.3</td>
<td>1.2</td>
</tr>
</tbody>
</table>
### SELECTION SCANS

- **Scalar (Branching)**
- **Scalar (Branchless)**
- **Vectorized (Early Mat)**
- **Vectorized (Late Mat)**

**MIC (Xeon Phi 7120P – 61 Cores + 4×HT)**

**Multi-Core (Xeon E3-1275v3 – 4 Cores + 2×HT)**

**Throughput (billion tuples / sec)**

**Selectivity (%)**

**Memory Bandwidth**
HASH TABLES – PROBING

Scalar

Input Key  hash(key)  Hash Index

k1  #  h1

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>
<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

Scalar

Input Key  \( \text{hash(key)} \)  Hash Index

\[ k_1 \rightarrow \# \rightarrow h_1 \]

Linear Probing Hash Table

\[ k_1 = k_9 \]
HASH TABLES – PROBING

Scalar

Input Key  hash(key)  Hash Index
k1          #         h1

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>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

k1 = k1
k8 = k8
k3 = k3
k9 = k9
HASH TABLES – PROBING

Scalar

<table>
<thead>
<tr>
<th>Input Key</th>
<th>hash(key)</th>
<th>Hash Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>k1</td>
<td>#</td>
<td>h1</td>
</tr>
</tbody>
</table>

Vectorized (Horizontal)

<table>
<thead>
<tr>
<th>Input Key</th>
<th>hash(key)</th>
<th>Hash Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>k1</td>
<td>#</td>
<td>h1</td>
</tr>
</tbody>
</table>

Linear Probing Bucketized Hash Table

<table>
<thead>
<tr>
<th>KEYS</th>
<th>PAYLOAD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
HASH TABLES – PROBING

**Scalar**

<table>
<thead>
<tr>
<th>Input Key</th>
<th>hash(key)</th>
<th>Hash Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>k1</td>
<td>#</td>
<td>h1</td>
</tr>
</tbody>
</table>

**Vectorized (Horizontal)**

<table>
<thead>
<tr>
<th>Input Key</th>
<th>hash(key)</th>
<th>Hash Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>k1</td>
<td>#</td>
<td>h1</td>
</tr>
</tbody>
</table>

**Linear Probing Bucketized Hash Table**

<table>
<thead>
<tr>
<th>KEYS</th>
<th>PAYLOAD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SIMD Compare**

Matched Mask

| 0 | 0 | 0 | 1 |

k1 = k9 k3 k8 k1
HASH TABLES – PROBING

Vectorized (Vertical)

Input Key Vector  hash(key)  Hash Index Vector

| k1 | # | h1 |
| k2 | # | h2 |
| k3 | # | h3 |
| k4 | # | h4 |

Linear Probing Hash Table

<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>k6</td>
<td></td>
</tr>
<tr>
<td>k4</td>
<td></td>
</tr>
<tr>
<td>k5</td>
<td></td>
</tr>
<tr>
<td>k88</td>
<td></td>
</tr>
</tbody>
</table>
**HASH TABLES – PROBING**

**Vectorized (Vertical)**

<table>
<thead>
<tr>
<th>Input Key Vector</th>
<th>hash(key)</th>
<th>Hash Index Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>k1</td>
<td>#</td>
<td>h1</td>
</tr>
<tr>
<td>k2</td>
<td>#</td>
<td>h2</td>
</tr>
<tr>
<td>k3</td>
<td>#</td>
<td>h3</td>
</tr>
<tr>
<td>k4</td>
<td>#</td>
<td>h4</td>
</tr>
</tbody>
</table>

**Linear Probing Hash Table**

<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>k6</td>
<td></td>
</tr>
<tr>
<td>k4</td>
<td></td>
</tr>
<tr>
<td>k5</td>
<td></td>
</tr>
<tr>
<td>k88</td>
<td></td>
</tr>
</tbody>
</table>
HASH TABLES – PROBING

Vectorized (Vertical)

Input Key Vector  $\text{hash(key)}$  Hash Index Vector

<table>
<thead>
<tr>
<th>k1</th>
<th>#</th>
<th>h1</th>
</tr>
</thead>
<tbody>
<tr>
<td>k2</td>
<td>#</td>
<td>h2</td>
</tr>
<tr>
<td>k3</td>
<td>#</td>
<td>h3</td>
</tr>
<tr>
<td>k4</td>
<td>#</td>
<td>h4</td>
</tr>
</tbody>
</table>

SIMD Compare

Linear Probing Hash Table

<table>
<thead>
<tr>
<th>KEY</th>
<th>PAYLOAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>k1</td>
<td></td>
</tr>
<tr>
<td>k2</td>
<td></td>
</tr>
<tr>
<td>k3</td>
<td></td>
</tr>
<tr>
<td>k4</td>
<td></td>
</tr>
<tr>
<td>k99</td>
<td></td>
</tr>
<tr>
<td>k88</td>
<td></td>
</tr>
<tr>
<td>k4</td>
<td></td>
</tr>
<tr>
<td>k5</td>
<td></td>
</tr>
<tr>
<td>k88</td>
<td></td>
</tr>
</tbody>
</table>
Vectorized (Vertical)

Input Key Vector  \( \text{hash(key)} \)  Hash Index Vector

- **k5**
- **k2**
- **k3**
- **k6**

- \( # \)  \( h2+1 \)  \( h3+1 \)  \( h6 \)

Linear Probing Hash Table

<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>k2</td>
<td></td>
</tr>
<tr>
<td>k3</td>
<td></td>
</tr>
<tr>
<td>k4</td>
<td></td>
</tr>
<tr>
<td>k5</td>
<td></td>
</tr>
<tr>
<td>k88</td>
<td></td>
</tr>
</tbody>
</table>

SIMD Compare

- **k1** = **k1**  \( = 1 \)
- **k2** = **k99**  \( = 0 \)
- **k3** = **k88**  \( = 0 \)
- **k4** = **k4**  \( = 1 \)
HASH TABLES – PROBING

Vectorized (Vertical)

Input Key Vector \( hash(key) \) Hash Index Vector

- \( k5 \)  \( \# \)  \( h5 \)
- \( k2 \)  \( \# \)  \( h2+1 \)
- \( k3 \)  \( \# \)  \( h3+1 \)
- \( k6 \)  \( \# \)  \( h6 \)

Linear Probing Hash Table

<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>k6</td>
<td></td>
</tr>
<tr>
<td>k4</td>
<td></td>
</tr>
<tr>
<td>k5</td>
<td></td>
</tr>
<tr>
<td>k88</td>
<td></td>
</tr>
</tbody>
</table>
**HASH TABLES – PROBING**

- **Scalar**
- **Vectorized (Horizontal)**
- **Vectorized (Vertical)**

**MIC (Xeon Phi 7120P – 61 Cores + 4×HT)**

**Multi-Core (Xeon E3-1275v3 – 4 Cores + 2×HT)**

<table>
<thead>
<tr>
<th>Hash Table Size</th>
<th>Throughput (billion tuples / sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MIC</td>
</tr>
<tr>
<td>4KB</td>
<td>12</td>
</tr>
<tr>
<td>16KB</td>
<td>9</td>
</tr>
<tr>
<td>64KB</td>
<td>6</td>
</tr>
<tr>
<td>256KB</td>
<td>3</td>
</tr>
<tr>
<td>1MB</td>
<td>3</td>
</tr>
<tr>
<td>4MB</td>
<td>2</td>
</tr>
<tr>
<td>16MB</td>
<td>1.5</td>
</tr>
<tr>
<td>64MB</td>
<td>1.2</td>
</tr>
</tbody>
</table>

Throughput (billion tuples / sec) vs Hash Table Size for MIC and Multi-Core configurations.
**HASH TABLES – PROBING**

- **Scalar**
- **Vectorized (Horizontal)**
- **Vectorized (Vertical)**

*MIC (Xeon Phi 7120P – 61 Cores + 4×HT)*

*Multi-Core (Xeon E3-1275v3 – 4 Cores + 2×HT)*

**Throughput (billion tuples / sec)**

**Hash Table Size**

**Out of Cache**
PARTITIONING – HISTOGRAM

Use scatter and gathers to increment counts.
Replicate the histogram to handle collisions.
PARTITIONING – HISTOGRAM

Use scatter and gathers to increment counts.
Replicate the histogram to handle collisions.
PARTITIONING – HISTOGRAM

Use scatter and gathers to increment counts.
Replicate the histogram to handle collisions.
JOINS

No Partitioning
→ Build one shared hash table using atomics
→ Partially vectorized

Min Partitioning
→ Partition building table
→ Build one hash table per thread
→ Fully vectorized

Max Partitioning
→ Partition both tables repeatedly
→ Build and probe cache-resident hash tables
→ Fully vectorized
JOINS

200M \times 200M tuples (32-bit keys & payloads)
Xeon Phi 7120P – 61 Cores + 4\times HT

Join Time (sec)

<table>
<thead>
<tr>
<th></th>
<th>Partition</th>
<th>Build</th>
<th>Probe</th>
<th>Build+Probe</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Partitioning</td>
<td>Scalar</td>
<td>1</td>
<td>0.5</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td>Vector</td>
<td>1</td>
<td>0.5</td>
<td>1.5</td>
</tr>
<tr>
<td>Min Partitioning</td>
<td>Scalar</td>
<td>1</td>
<td>0.5</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td>Vector</td>
<td>1</td>
<td>0.5</td>
<td>1.5</td>
</tr>
<tr>
<td>Max Partitioning</td>
<td>Scalar</td>
<td>1</td>
<td>0.5</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td>Vector</td>
<td>1</td>
<td>0.5</td>
<td>1.5</td>
</tr>
</tbody>
</table>
PARTING THOUGHTS

Vectorization is essential for OLAP queries. These algorithms don’t work when the data exceeds your CPU cache.

We can combine all the intra-query parallelism optimizations we’ve talked about in a DBMS.

→ Multiple threads processing the same query.
→ Each thread can execute a compiled plan.
→ The compiled plan can invoke vectorized operations.
NEXT CLASS

Vectorization (Part II)