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: \(32 \times 4 = 128\)
MULTI-CORE CPUS

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-power 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)
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**

\[ X + Y = Z \]

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

```c
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
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:**

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

**128-bit SIMD Register**

```
8 7 6 5
```

**128-bit SIMD Register**

```
1 1 1 1 1
```
\[ X + Y = Z \]

For \( i = 0 \) to \( i < n \) do:

\[
Z[i] = X[i] + Y[i];
\]
\[ X + Y = Z \]

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

```c
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](#)
VECTORIZATION

Choice #1: Automatic Vectorization

Choice #2: Compiler Hints

Choice #3: Explicit Vectorization
VECTORIZATION

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

Source: James Reinders
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.
void add(int *X, int *Y, int *Z) {
    for (int i=0; i<MAX; i++) {
        Z[i] = X[i] + Y[i];
    }
}
AUTOMATIC VECTORIZATION

This loop is not legal to automatically vectorize.

```c
void add(int *X,
         int *Y,
         int *Z)
{
    *Z = *X + 1
    for (int i=0; i<MAX; i++) {
        Z[i] = X[i] + Y[i];
    }
}
```

These might point to the same address!
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.
The `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];
    }
}
```
This pragma tells the compiler to ignore loop dependencies for the vectors.

It’s up to you make sure that this is correct.
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) {
    __mm128i *vecX = (__m128i*)X;
    __mm128i *vecY = (__m128i*)Y;
    __mm128i *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
FUNDAMENTAL VECTOR OPERATIONS

Selective Load

**Vector**

\[
\begin{array}{cccc}
A & B & C & D \\
\end{array}
\]

**Mask**

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

**Memory**

\[
\begin{array}{cccccccc}
U & V & W & X & Y & Z & \cdots \\
\end{array}
\]
FUNDAMENTAL VECTOR OPERATIONS

Selective Load

Vector \[ \begin{array}{cccc}
A & B & C & D \\
\end{array} \]

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

Memory \[ \begin{array}{ccccccc}
U & V & W & X & Y & Z & \cdots \\
\end{array} \]
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: A U C D

Mask: 0 1 0 1

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

Selective Load

Vector: \[\textbf{A} \quad \textbf{U} \quad \textbf{C} \quad \textbf{V}\]

Mask: \[0 \quad 1 \quad 0 \quad 1\]

Memory: \[\textbf{U} \quad \textbf{V} \quad \textbf{W} \quad \textbf{X} \quad \textbf{Y} \quad \textbf{Z} \quad \cdots\]
FUNDAMENTAL VECTOR OPERATIONS

**Selective Load**

- **Vector**: A U C V
- **Mask**: 0 1 0 1
- **Memory**: U V W X Y Z

**Selective Store**

- **Memory**: U V W X Y Z
- **Mask**: 0 1 0 1
- **Vector**: A B C D
FUNDAMENTAL VECTOR OPERATIONS

Selective Load

Vector: \( \begin{array}{c} A \ U \ C \ V \end{array} \)

Mask: \( \begin{array}{c} 0 \ 1 \ 0 \ 1 \end{array} \)

Memory: \( \begin{array}{c} U \ V \ W \ X \ Y \ Z \end{array} \ldots \)

Selective Store

Memory: \( \begin{array}{c} U \ V \ W \ X \ Y \ Z \end{array} \ldots \)

Mask: \( \begin{array}{c} 0 \ 1 \ 0 \ 1 \end{array} \)

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

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: \[ \begin{bmatrix} A & U & C & V \end{bmatrix} \]
- Mask: \[ \begin{bmatrix} 0 & 1 & 0 & 1 \end{bmatrix} \]
- Memory: \[ \begin{bmatrix} U & V & W & X & Y & Z & \cdots \end{bmatrix} \]

**Selective Store**

- Memory: \[ \begin{bmatrix} B & V & W & X & Y & Z & \cdots \end{bmatrix} \]
- Mask: \[ \begin{bmatrix} 0 & 1 & 0 & 1 \end{bmatrix} \]
- Vector: \[ \begin{bmatrix} A & B & C & D \end{bmatrix} \]
FUNDAMENTAL VECTOR OPERATIONS

Selective Load

Vector: \[ \begin{array}{cccc} A & U & C & V \\ \end{array} \]

Mask: \[ \begin{array}{cccc} 0 & 1 & 0 & 1 \\ \end{array} \]

Memory: \[ \begin{array}{cccccccc} U & V & W & X & Y & Z & \cdots \end{array} \]

Selective Store

Memory: \[ \begin{array}{cccccccc} B & D & W & X & Y & Z & \cdots \end{array} \]

Mask: \[ \begin{array}{cccc} 0 & 1 & 0 & 1 \end{array} \]

Vector: \[ \begin{array}{cccc} A & B & C & D \end{array} \]
Selective Gather

Value Vector

Index Vector

Memory

\[
\begin{array}{cccccc}
A & B & A & D & & \\
2 & 1 & 5 & 3 & & \\
U & V & W & X & Y & Z \\
0 & 1 & 2 & 3 & 4 & 5 \\
\end{array}
\]
FUNDAMENTAL VECTOR OPERATIONS

Selective Gather

Value Vector: \( \begin{pmatrix} W & B & A & D \end{pmatrix} \)

Index Vector: \( \begin{pmatrix} 2 & 1 & 5 & 3 \end{pmatrix} \)

Memory: \( \begin{pmatrix} U & V & W & X & Y & Z \cdots \end{pmatrix} \)
FUNDAMENTAL VECTOR OPERATIONS

Selective Gather

Value Vector  
\[
\begin{array}{c}
W \\
V \\
Z \\
X \\
\end{array}
\]

Index Vector  
\[
\begin{array}{c}
2 \\
1 \\
5 \\
3 \\
\end{array}
\]

Memory  
\[
\begin{array}{ccccccc}
U & V & W & X & Y & Z & \cdots \\
0 & 1 & 2 & 3 & 4 & 5 & \cdots \\
\end{array}
\]
FUNDAMENTAL VECTOR OPERATIONS

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 V W X Y Z
Index Vector: 2 1 5 3
Value Vector: A B C D
FUNDAMENTAL VECTOR OPERATIONS

Selective Gather

Value Vector

Index Vector

Memory

Selective Scatter

Memory

Index Vector

Value Vector
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 implemented in Xeon CPUs using vector permutations.
VECTORIZED OPERATORS

Selection Scans
Hash Tables
Partitioning

Paper provides additional info:
→ Joins, Sorting, Bloom filters.
### SELECTION SCANS

**Scalar (Branching)**

```python
i = 0
for t in table:
    key = t.key
    if (key\geq low) && (key\leq high):
        copy(t, output[i])
    i = i + 1
```

**Scalar (Branchless)**

```python
i = 0
for t in table:
    key = t.key
    m = (key\geq low ? 1 : 0) &&
        (key\leq high ? 1 : 0)
    i = i + m
```
### SELECTION SCANS

**Scalar (Branching)**

```python
i = 0
for t in table:
    key = t.key
    if (key ≥ low) && (key ≤ high):
        copy(t, output[i])
    i = i + 1
```

**Scalar (Branchless)**

```python
i = 0
for t in table:
    copy(t, output[i])
    key = t.key
    m = (key ≥ low ? 1 : 0) &&
        (key ≤ high ? 1 : 0)
    i = i + m
```

Source: Bogdan Raducanu
SELECTION SCANS

Source: Bogdan Raducanu
Vectorized

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

```cpp
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|
```

<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>

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

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

**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>
Vectorized

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

\[
\begin{array}{|c|c|}
\hline
\text{ID} & \text{KEY} \\
\hline
1 & J \\
2 & O \\
3 & Y \\
4 & S \\
5 & U \\
6 & X \\
\hline
\end{array}
\]

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

**Vectorized**

Vectorized selection:

```plaintext
i = 0
for vt in table:
    simdLoad(v_t.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|
```

**Example SQL Query**:

```
SELECT * FROM table
WHERE key >= "0" AND key <= "U"
```
Vectorized

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

SELECT * FROM table
WHERE key >= "O" AND key <= "U"
Selection Scans (Vectorized)

1. \( i = 0 \)
2. For \( v_t \) in table:
   - \( \text{simdLoad}(v_t, \text{key}, v_k) \)
   - \( v_m = (v_k \geq \text{"O"}) \& \& (v_k \leq \text{"U"}) \)
   - If \( \neg v_m \):
     - \( \text{simdStore}(v_t, v_m, \text{output}[i]) \)
     - \( i = i + |v_m| = \text{false} | \)

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>

SIMD Compare:

Mask:

| 0 1 0 1 1 0 |

All Offsets:

| 0 1 2 3 4 5 |

Matched Offsets:

| 1 3 4 |

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

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

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

Throughput (billion tuples / sec)

Selectivity (%)

0 1 2 5 10 20 50 100

0 1 2 5 10 20 50 100

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

Throughput (billion tuples / sec)

Selectivity (%)

0 1 2 5 10 20 50 100

0 1 2 5 10 20 50 100

5.7 5.6 5.3 4.9 4.3 2.8 1.3

1.8 1.7 1.7 1.7 1.6 1.5 1.2
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**

<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 Hash Table**

<table>
<thead>
<tr>
<th>KEY</th>
<th>PAYLOAD</th>
</tr>
</thead>
</table>
HASH TABLES – PROBING

Scalar

Input Key | hash(key) | Hash Index
---|---|---
\(k_1\) | | \(h_1\)

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>

\(k_1 = k_9\)
HASH TABLES – PROBING

Scalar

Input Key  hash(key)  Hash Index
k1  #  h1

Linear Probing Hash Table

k1 = k1
k3 = k3
k8 = k8
k9 = k9

Scalar

Hash Table

k1

CMU 15-721 (Spring 2019)
HASH TABLES – PROBING

Scalar

Input Key  hash(key)  Hash Index
k1          #           h1

Vectorized (Horizontal)

Input Key  hash(key)  Hash Index
k1          #           h1

Linear Probing
Bucketized Hash Table

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

Scalar

- Input Key: k1
- hash(key): #
- Hash Index: h1

Vectorized (Horizontal)

- Input Key: k1
- hash(key): #
- Hash Index: h1

Linear Probing
Bucketized Hash Table

- KEYS: k9, k3, k8, k1
- PAYLOAD:
HASH TABLES – PROBING

Scalar

Input Key | hash(key) | Hash Index
-----------|----------|------------
k1         | #        | h1         

Linear Probing Bucketized Hash Table

Vectorized (Horizontal)

Input Key | hash(key) | Hash Index
-----------|----------|------------
k1         | #        | h1         

SIMD Compare

Matched Mask
HASH TABLES – PROBING

Vectorized (Vertical)

<table>
<thead>
<tr>
<th>Input Key Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>k1</td>
</tr>
<tr>
<td>k2</td>
</tr>
<tr>
<td>k3</td>
</tr>
<tr>
<td>k4</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)

<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**: k1, k2, k3, k4
- **hash(key)**: #, #, #, #
- **Hash Index Vector**: h1, h2, h3, h4

**Linear Probing Hash Table**

- **KEY**: k1, k2, k3, k4
- **PAYLOAD**: k99, k88, k4

**SIMD Gather**

- k1 = k1
- k2 = k99
- k3 = k88
- k4 = k4
### HASH TABLES – PROBING

**Vectorized (Vertical)**

**Input Key Vector**

- k1
- k2
- k3
- k4

**hash(key)**

- #
- #
- #
- #

**Hash Index Vector**

- h1
- h2
- h3
- 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>0</td>
</tr>
<tr>
<td>k88</td>
<td>0</td>
</tr>
<tr>
<td>k4</td>
<td>1</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
---|---|---
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>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>
HASH TABLES – PROBING

Vectorized (Vertical)

Input Key Vector

<table>
<thead>
<tr>
<th>Key</th>
<th>Hash Index Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>k5</td>
<td>h5</td>
</tr>
<tr>
<td>k2</td>
<td>h2+1</td>
</tr>
<tr>
<td>k3</td>
<td>h3+1</td>
</tr>
<tr>
<td>k6</td>
<td>h6</td>
</tr>
</tbody>
</table>

Hash Index Vector

<table>
<thead>
<tr>
<th>Key</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>k1</td>
<td>1</td>
</tr>
<tr>
<td>k2</td>
<td>0</td>
</tr>
<tr>
<td>k3</td>
<td>0</td>
</tr>
<tr>
<td>k4</td>
<td>1</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>k88</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  \( \text{hash}(\text{key}) \)  Hash Index Vector

<table>
<thead>
<tr>
<th>Input Key Vector</th>
<th>#</th>
<th>#</th>
<th>#</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>( k_5 )</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>( k_2 )</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>( k_3 )</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>( k_6 )</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
</tbody>
</table>

Hash Index Vector

<table>
<thead>
<tr>
<th>Hash Index Vector</th>
<th>( h_5 )</th>
<th>( h_2+1 )</th>
<th>( h_3+1 )</th>
<th>( h_6 )</th>
</tr>
</thead>
</table>

Linear Probing Hash Table

<table>
<thead>
<tr>
<th>KEY</th>
<th>PAYLOAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>( k_{99} )</td>
<td></td>
</tr>
<tr>
<td>( k_1 )</td>
<td></td>
</tr>
<tr>
<td>( k_6 )</td>
<td></td>
</tr>
<tr>
<td>( k_4 )</td>
<td></td>
</tr>
<tr>
<td>( k_5 )</td>
<td></td>
</tr>
<tr>
<td>( k_{88} )</td>
<td></td>
</tr>
</tbody>
</table>
HASH TABLES – PROBING

**Throughput (billion tuples / sec)**

**Mic (Xeon Phi 7120P – 61 Cores + 4×HT)**
- **Vectorized (Horizontal)**
- **Vectorized (Vertical)**

**Hash Table Size**
- 4KB
- 16KB
- 64KB
- 256KB
- 1MB
- 4MB
- 16MB
- 64MB

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

**Hash Table Size**
- 4KB
- 16KB
- 64KB
- 256KB
- 1MB
- 4MB
- 16MB
- 64MB
### HASH TABLES – PROBING

<table>
<thead>
<tr>
<th>MIC (Xeon Phi 7120P – 61 Cores + 4×HT)</th>
<th>Multi-Core (Xeon E3-1275v3 – 4 Cores + 2×HT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput (billion tuples / sec)</td>
<td>Throughput (billion tuples / sec)</td>
</tr>
<tr>
<td>Hash Table Size</td>
<td></td>
</tr>
<tr>
<td>Scalar</td>
<td>Out of Cache</td>
</tr>
<tr>
<td>Vectorized (Horizontal)</td>
<td></td>
</tr>
<tr>
<td>Vectorized (Vertical)</td>
<td></td>
</tr>
</tbody>
</table>

- **Throughput**
- **Hash Table Size**
- **Scalar**
- **Vectorized (Horizontal)**
- **Vectorized (Vertical)**
- **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.

Input Key Vector | Hash Index Vector | Histogram
--- | --- | ---
\(k_1\) | \(h_1\) | \(+1\)
\(k_2\) | \(h_2\) | \(+1\)
\(k_3\) | \(h_3\) | \(+1\)
\(k_4\) | \(h_4\) | \(+1\)

**SIMD Radix** | **SIMD Add**

*Missing Update*
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 \(\bowtie\) 200M tuples (32-bit keys & payloads)
Xeon Phi 7120P – 61 Cores + 4\times\text{HT}

<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>1.5</td>
<td>0.5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Min Partitioning</td>
<td>1.5</td>
<td>0.5</td>
<td>1</td>
<td>1.5</td>
</tr>
<tr>
<td>Max Partitioning</td>
<td>2</td>
<td>1</td>
<td>1.5</td>
<td>2</td>
</tr>
</tbody>
</table>

Join Time (sec)
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

Compilation vs. Vectorization