Lecture #21 – Vectorized Execution (Part I)
TODAY’S AGENDA

Background
Hardware
Vectorized Algorithms (Columbia)
OBVIOUS OBSERVATIONS

#1 – Building a DBMS is hard.

#2 – Taco Bell gives you diarrhea.

#3 – New CPUs are not getting faster.
MULTI-CORE CPUS

Use a small number of high-powered cores.
→ Intel Haswell / Skylake
→ 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
→ Cores = Intel P54C (aka Pentium from the 1990s).
→ Low power consumption and area per core.

Non-superscalar and in-order execution but with expanded SIMD capabilities.
→ More advanced instructions with larger register sizes.
MANY INTEGRATED CORES (MIC)

Use a larger number of low-powered cores.

- Intel Xeon Phi
- Cores = Intel P54C (aka Pentium from the 1990s).
- Low power consumption and area per core.
- Non-superscalar and in-order execution but with expanded SIMD capabilities.
- More advanced instructions with larger register sizes.
MANY INTEGRATED CORES (MIC)

Use a larger number of low-powered cores.
→ **Intel Xeon Phi**
→ Cores = **Intel P54C** (aka Pentium from the 1990s).
→ Low power consumption and area per core.

**Non-superscalar** and **in-order execution** but with expanded SIMD capabilities.
→ More advanced instructions with larger register sizes.
MULTI-CORE VS. MIC

Source: Orestis Polychroniou
WHY THIS MATTERS

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

Potential Speed-up: \(32 \times 4 = 128\)
VECTORIZATION

A program is converted from a 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.
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
→ **PowerPC**: Altivec
→ **ARM**: NEON
**SIMD EXAMPLE**

\[ X + Y = Z \]

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

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

\[ X + Y = Z \]

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

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

\[
\begin{bmatrix}
8 \\
7 \\
6 \\
5 \\
4 \\
3 \\
2 \\
1 \\
\end{bmatrix}
\]

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

\[
\begin{bmatrix}
9 \\
8 \\
7 \\
6 \\
5 \\
4 \\
3 \\
2 \\
1 \\
\end{bmatrix}
\]

SISD

+
**SIMD EXAMPLE**

\[ X + Y = Z \]

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

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

128-bit SIMD Register

```
8
7
6
5
4
3
2
1
```

128-bit SIMD Register

```
1
1
1
1
```

CMU 15-721 (Spring 2017)
**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];
}
```

128-bit SIMD Register

\[
\begin{array}{cccc}
  8 & 7 & 6 & 5 \\
  6 & 5 & 4 & 3 \\
  3 & 2 & 1 & 0
\end{array}
\]

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

128-bit SIMD Register

\[
\begin{array}{cccc}
  9 & 8 & 7 & 6
\end{array}
\]
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 \( i = 0; i < n; i++ \) {
  \( Z[i] = X[i] + Y[i] \);}

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

\[
\begin{bmatrix}
  4 & 3 & 2 & 1 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
  9 & 8 & 7 & 6 & 5 & 4 & 3 & 2
\end{bmatrix}
\]
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.
SSE 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
SSE 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

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];  
    }  
}

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) {
    __mm128 *vecX = (__m128*)X;
    __mm128 *vecY = (__m128*)Y;
    __mm128 *vecZ = (__m128*)Z;
    for (int i=0; i<MAX/4; i++) {
        *vecZ++ = _mm_add_epi32(*vecX++, vecY);
    }
}
```
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 Sore
Selective Gather
Selective Scatter
## FUNDAMENTAL VECTOR OPERATIONS

### Selective Load

<table>
<thead>
<tr>
<th>Vector</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mask</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Memory</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
</tr>
</tbody>
</table>
Selective Load

**Vector**

| A | B | C | D |

**Mask**

| 0 | 1 | 0 | 1 |

**Memory**

| U | V | W | X | Y | Z | ... |
Selective Load

Vector: A B 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 D

Mask: 0 1 0 1

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

Vector: A U C V

Mask: 0 1 0 1

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

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

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

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

Selective Store

Memory: \[\begin{array}{cccccccccccccccccccc}
B & V & W & X & Y & Z \\
\end{array}\]

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

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

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

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

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

Selective Store

Memory: \[ \begin{array}{c} B \ V \ W \ X \ Y \ Z \ldots \end{array} \]

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

Selective Gather

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

Selective Gather

Value Vector: A B A D

Index Vector: 2 1 5 3

Memory: U V W X Y Z • • •
**Fundamental Vector Operations**

*Selective Gather*

- **Value Vector**: \( \{W, V, Z, X\} \)
- **Index Vector**: \( \{2, 1, 5, 3\} \)
- **Memory**: \( \{U, V, W, X, Y, Z, \ldots\} \)
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: \[ W \quad V \quad Z \quad X \]

Index Vector: \[ 2 \quad 1 \quad 5 \quad 3 \]

Memory: \[ U \quad V \quad W \quad X \quad Y \quad Z \ldots \]

---

**Selective Scatter**

Memory: \[ U \quad V \quad W \quad X \quad Y \quad Z \ldots \]

Index Vector: \[ 2 \quad 1 \quad 5 \quad 3 \]

Value Vector: \[ A \quad B \quad C \quad D \]
Selective Gather

Value Vector: \[ \text{W V Z X} \]
Index Vector: \[ 2 \quad 1 \quad 5 \quad 3 \]
Memory: \[ U \quad V \quad W \quad X \quad Y \quad Z \ldots \]

Selective Scatter

Memory: \[ \text{U B A D Y C} \ldots \]
Index Vector: \[ 2 \quad 1 \quad 5 \quad 3 \]
Value Vector: \[ A \quad B \quad C \quad D \]

FUNDAMENTAL VECTOR OPERATIONS
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 (Haswell’s AVX2).

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)
Scalar (Branching)

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

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

**Scalar (Branching)**

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

**Scalar (Branchless)**

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

Scalar (Branching)

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

Scalar (Branchless)

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

Source: Bogdan Raducanu

![Graph showing performance comparison between branching and no branching in selection scans.](image)
Vectorized

```
i = 0
for vt in table:
    simdLoad(vt.key,_vk)
    vm = (vk ≥ low ? 1 : 0) &&
         (vk ≤ high ? 1 : 0)
if vm ≠ false:
    simdStore(vt, vm, output[i])
i = i + |vm≠false|
```

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

Vectorized

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

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

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

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

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

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

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

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

- Throughput (billion tuples/sec)
  - 5.7, 5.7, 5.6, 5.3, 4.9, 4.3, 2.8, 1.3

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

- Throughput (billion tuples/sec)
  - 1.8, 1.7, 1.7, 1.6, 1.5, 1.2
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>
</tbody>
</table>
```

k1 = k9
HASH TABLES – PROBING

**Scalar**

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

$\text{Linear Probing Hash Table}$

<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>k9</td>
<td></td>
</tr>
<tr>
<td>k3</td>
<td></td>
</tr>
<tr>
<td>k8</td>
<td></td>
</tr>
<tr>
<td>k1</td>
<td>k1</td>
</tr>
</tbody>
</table>
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>k1 = k9 k3 k8 k1</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 $\rightarrow$ hash(key) $\rightarrow$ Hash Index Vector

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

SIMD Gather

- $k1 = k1$
- $k2 = k99$
- $k3 = k88$
- $k4 = k4$
VECTORIZED (VERTICAL)

**Input Key Vector**
- k1
- k2
- k3
- k4

**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></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 Index**

<table>
<thead>
<tr>
<th>Input Key</th>
<th>Hash Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>k1</td>
<td>h1</td>
</tr>
<tr>
<td>k2</td>
<td>h2</td>
</tr>
<tr>
<td>k3</td>
<td>h3</td>
</tr>
<tr>
<td>k4</td>
<td>h4</td>
</tr>
</tbody>
</table>

**SIMD Compare**

<table>
<thead>
<tr>
<th>k1</th>
<th>k2</th>
<th>k3</th>
<th>k4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Vectorized (Vertical)

**Input Key Vector**

- k5
- k2
- k3
- k6

**hash(key)**

- #
- #
- #
- #

**Hash Index Vector**

- h5
- h2+1
- h3+1
- h6

**Linear Probing Hash Table**

<table>
<thead>
<tr>
<th>KEY</th>
<th>PAYLOAD</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>
<tr>
<td>k5</td>
<td></td>
</tr>
<tr>
<td>k6</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>$k5$</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>

Vector hash(key):

- $h5$
- $h2+1$
- $h3+1$
- $h6$

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>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)**

Throughput (billion tuples / sec)

Hash Table Size

---

CMU 15-721 (Spring 2017)
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>4KB 16KB 64KB 256KB 1MB 4MB 16MB 64MB</td>
<td>4KB 16KB 64KB 256KB 1MB 4MB 16MB 64MB</td>
</tr>
</tbody>
</table>

**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.
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\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>Scalar No Partitioning</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector Min Partitioning</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scalar Max Partitioning</td>
<td></td>
<td></td>
<td></td>
<td></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

Vectorization (Part II)

Code Review Submission: April 11th
Project Status Meetings: April 13th
Project #3 Status Updates: April 18th