SIMD Vectorized Hashing for Grouped Aggregation

Bala Gurumurthy, David Broneske, Marcus Pinnecke, Gabriel Campero Durand and Gunter Saake
Grouped Aggregation

- Commonly-used and time-consuming operation

Based on analysis by Boncz et al. [1]
Grouped Aggregation

- Commonly-used and time-consuming operation

- All input must be consumed for single output

Based on analysis by Boncz et al. [1]
Grouped Aggregation

- Commonly-used and time-consuming operation
- All input must be consumed for single output
- Faster input processing = higher throughput

Based on analysis by Boncz et al. [1]
Grouped Aggregation

- Commonly-used and time-consuming operation
- All input must be consumed for single output
- Faster input processing = higher throughput
- Improving underlying technique improves efficiency

Based on analysis by Boncz et al. [1]
SIMD Capability in Modern Processors

- SIMD – Single Instruction Multiple Data

![SIMD Example Diagram](image)
SIMD Capability in Modern Processors

- SIMD – Single Instruction Multiple Data
  - Allows vectorized execution in modern processors
    - Reduces overall execution time of an operation
SIMD Capability in Modern Processors

- SIMD – Single Instruction Multiple Data
- Allows vectorized execution in modern processors
  - Reduces overall execution time of an operation
- SIMD is shown to increase throughput in orders of magnitude for DBMS operation \([2] [3]\)

SIMD accelerated selection \([3]\)
SIMD for Grouped Aggregation

- SIMD acceleration of hashing techniques improves throughput

+ Group-By = High throughput
SIMD for Grouped Aggregation

- SIMD acceleration of hashing techniques improves throughput

How to incorporate SIMD for Grouped Aggregation?

What is the impact of SIMD?
Hash Based Aggregation

- Grouped aggregation commonly implemented using hashing techniques
Hash Based Aggregation

- Grouped aggregation commonly implemented using hashing techniques
Hash Based Aggregation

- Grouped aggregation commonly implemented using hashing techniques
- Separates groups into buckets
Hash Based Aggregation

- Grouped aggregation commonly implemented using hashing techniques

- Separates groups into buckets

- Aggregation done within each buckets
Hash Based Aggregation: Example

Key

Present

Not Present

Key in hashtable

Update Aggregate

Insert Key

Key

Hash function

h(x)

Aggregate
Hash Based Aggregation: Example

Input: 3

h(x)

1 2
1 1

4 5
1 1
Hash Based Aggregation: Example

Input: 3

\( h(x) \)

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Hash Based Aggregation: Example

Input: 3

Hash Function: $h(x)$

1  2  3  4  5
1  1  1  1  1
Hash Based Aggregation: Example

Input: 3

h(x) = 3

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Hash Based Aggregation: Example

Input: 3

$h(x)$

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Collision: Increasing Complexity

- Not all keys have unique location

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Collision: Increasing Complexity

- Not all keys have unique location
- Two keys might hash to same slots
Collision: Increasing Complexity

- Not all keys have unique location
- Two keys might hash to same slots
Collision: Increasing Complexity

- Not all keys have unique location
- Two keys might hash to same slots
- Hash table must be probed for alternative location
Collision: Increasing Complexity

- Not all keys have unique location
- Two keys might hash to same slots
- Hash table must be probed for alternative location

$h(x)$

# of probes : 4
Collision: Increasing Complexity

- Not all keys have unique location
- Two keys might hash to same slots
- Hash table must be probed for alternative location

$h(x)$

Probing is time consuming
SIMD for Probing

- Multiple slots are probed in an instant using SIMD
SIMD for Probing

- Multiple slots are probed in an instant using SIMD
- Reduces overall number of probes
SIMD Accelerated Hash Probing

- Each hashing techniques have their own collision resolution mechanism
SIMD Accelerated Hash Probing

- Each hashing techniques have their own collision resolution mechanism
- We use open-addressing hashing techniques
SIMD Accelerated Hash Probing

- Each hashing techniques have their own collision resolution mechanism

- We use open-addressing hashing techniques
  - Have constant hashtable size
SIMD Accelerated Hash Probing

- Each hashing techniques have their own collision resolution mechanism

- We use open-addressing hashing techniques
  - Have constant hashtable size
  - Suitable for SIMD
SIMD Accelerated Hash Probing

- Each hashing techniques have their own collision resolution mechanism

- We use open-addressing hashing techniques
  - Have constant hashtable size
  - Suitable for SIMD

  Hashing techniques used are

  - Cuckoo hashing
  - Linear probing
  - Two-choice hashing
  - Hopscotch hashing
Cuckoo Hashing

- Stores keys in multiple hash tables

<table>
<thead>
<tr>
<th>Hash(x) = x%5</th>
<th>5</th>
<th>11</th>
<th>22</th>
<th>3</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hash(x) = floor(x/5)%5</th>
<th>5</th>
<th>11</th>
<th>22</th>
<th>3</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cuckoo Hashing

- Stores keys in multiple hash tables
- On collision swaps current value to alternative tables

Input: 1

\[
\text{Hash}(x) = \text{floor}(x/5) \mod 5
\]

\[
\begin{array}{ccccccc}
5 & 1 & 22 & 3 & 64 \\
1 & 1 & 2 & 1 & 1 \\
5 & 1 & 11 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 \\
\end{array}
\]
Cuckoo Hashing

- Stores keys in multiple hash tables
- On collision swaps current value to alternative tables
  - Might form swap loop; solved using a threshold

Input: 111

Hash(x) = x%5

<table>
<thead>
<tr>
<th>5</th>
<th>1</th>
<th>22</th>
<th>3</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Hash(x) = floor(x/5)%5

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cuckoo Hashing

- Stores keys in multiple hash tables
- On collision swaps current value to alternative tables
  - Might form swap loop; solved using a threshold
- Has constant look-up time

\[
\text{Hash}(x) = x \mod 5
\]

\[
\text{Hash}(x) = \lfloor x/5 \rfloor \mod 5
\]

Input: 1

\[
\begin{array}{cccccc}
5 & 1 & 22 & 3 & 64 \\
\hline
1 & 1 & 2 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{cccc}
 & & 11 & \\
\hline
 & & 1 & \\
\end{array}
\]
Grouped Aggregation using Cuckoo Hashing

- Probe for key in each hash table
Grouped Aggregation using Cuckoo Hashing

- Probe for key in each hash table
  - If found update aggregate
Grouped Aggregation using Cuckoo Hashing

- Probe for key in each hash table
  - If found update aggregate
  - Else, insert the key
SIMD Optimization of Cuckoo Hashing

- Hash function computed for multiple tables
SIMD Optimization of Cuckoo Hashing

- Hash function computed for multiple tables
- Multiple slots probed in parallel
Linear Probing

- Straight forward approach for collision resolution
Linear Probing

- Straight forward approach for collision resolution
- Probes hash table linearly for alternative location
Linear Probing

- Straight forward approach for collision resolution
- Probes hash table linearly for alternative location
- Empty location encountered: insert key
SIMD Optimization of Linear Probing

- Multiple slot is probed
SIMD Optimization of Linear Probing

- Multiple slot is probed
- Comparison mask used for updating aggregate
Two-choice Hashing

- Improvement on linear probing
Two-choice Hashing

- Improvement on linear probing

- Two hashing function for same hash table (can be more)
Two-choice Hashing

- Improvement on linear probing

- Two hashing function for same hash table (can be more)

- If all slots occupied, probing done from both slots
SIMD Optimization of Two-Choice Hashing

- SIMD hash function from cuckoo hashing
SIMD Optimization of Two-Choice Hashing

- SIMD hash function from cuckoo hashing
- SIMD probing from linear probing
Hopscotch Hashing

- Has a limited probe length known as neighborhood

Neighborhood : 3
Hopscotch Hashing

- Has a limited probe length known as neighborhood

- A key will be available within neighborhood distance from hash slot

```
x y z key
```

Neighborhood : 3
Hopscotch Hashing

- Has a limited probe length known as neighborhood

- A key will be available within neighborhood distance from hash slot

- If no slot available within neighborhood, table is rearranged by swapping keys

Input: New key

```
| x (0) | y (1) | z (2) | Key (0) |
```

Neighborhood: 3
Hopscotch Hashing

- Has a limited probe length known as neighborhood

- A key will be available within neighborhood distance from hash slot

- If no slot available within neighborhood, table is rearranged by swapping keys

```
Neighborhood : 3
```

```
Input : New key
```

```
x (0)   y (1)   z (2)   Key (0)  
```

Input : New key
Hopscotch Hashing

- Has a limited probe length known as neighborhood

- A key will be available within neighborhood distance from hash slot

- If no slot available within neighborhood, table is rearranged by swapping keys

Input : New key

<table>
<thead>
<tr>
<th>x (0)</th>
<th>y (1)</th>
<th>Key (0)</th>
<th>z (2)</th>
</tr>
</thead>
</table>

Neighborhood : 3
Hopscotch Hashing

- Has a limited probe length known as neighborhood
- A key will be available within neighborhood distance from hash slot
- If no slot available within neighborhood, table is rearranged by swapping keys

```
x  y  New key  key  Z
```

Neighborhood : 3
SIMD Optimization of Hopscotch Hashing

- **SIMD for Probing**
  - Uses SIMD linear probing
SIMD Optimization of Hopscotch Hashing

- **SIMD for Probing**
  - Uses SIMD linear probing

- **SIMD for Insertion**
  - Swapping multiple values in an instant
SIMD Optimization of Hopscotch Hashing

- **SIMD for Probing**
  - Uses SIMD linear probing

- **SIMD for Insertion**
  - Swapping multiple values in an instant
SIMD Optimization of Hopscotch Hashing

- **SIMD for Probing**
  - Uses SIMD linear probing

- **SIMD for Insertion**
  - Swapping multiple values in an instant
  - Collect values using SIMD Gather

---

```
| a | b | c | d | e |
```

---

---

```
| c | d | e |
```
SIMD Optimization of Hopscotch Hashing

- **SIMD for Probing**
  - Uses SIMD linear probing

- **SIMD for Insertion**
  - Swapping multiple values in an instant
  - Collect values using SIMD Gather
  - Swap them

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Diagram of Hopscotch Hashing]

**Key Steps of Hopscotch Hashing**

1. Hash function
2. Probe table
3. Key present
4. Update Aggregate
5. Key not found
6. Empty slot
7. Insert

**Diagram Elements**

- SIMD Instruction Pool
- Data Pool
- Vector Unit

**Figure**

```
[Diagram of SIMD Optimization]
```
SIMD Optimization of Hopscotch Hashing

- **SIMD for Probing**
  - Uses SIMD linear probing

- **SIMD for Insertion**
  - Swapping multiple values in an instant
  - Collect values using SIMD Gather
  - Swap them
  - Store back

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Diagram of Hopscotch Hashing](diagram.png)
Evaluation Setup

- **Environment**
  - Processor: Intel Octa Core Xeon E5-2630
  - OS: Linux
  - SIMD: AVX2 instruction set

- **Grouped Aggregation**
  - Keys: 32 bit integers (0 is not valid)
  - Aggregation: count()

- **Hashing technique**
  - Hash function: Multiplicative function – \( h(x) = Ax \% \text{tableSize} \)
    - \( A \Rightarrow \) knuth’s number

- **Data Distribution**
  - Uniform random
  - Sequential
  - Unique random
  - Moving cluster
Unique Random Distribution

- Serial linear probing has worse time
- Serial hopscotch hashing is efficient
- Vectorized two-choice hashing competes with Hopscotch hashing
Uniform Random Distribution

- Both serial and vectorized hopscotch has worse efficiency
  - Vectorized with the worst time
- Vectorized Two-choice hashing has best execution time
Uniform Random Distribution

- Both serial and vectorized hopscotch has worse efficiency
  - Vectorized with the worst time
- Vectorized Two-choice hashing has best execution time

Other distributions have the same characteristics
Overall Speed-up

- SIMD has worse impact on hopsctoch hashing
  - Nearly 2x slower
  - Due to random access for insertion

- Linear probing has highest SIMD impact
Summary

- Hash probing reduces efficiency of grouped aggregation
Summary

- Hash probing reduces efficiency of grouped aggregation

- SIMD accelerated probing not always improves efficiency
  
  - Specifically, hopscotch hashing has negative impact

- Linear probing has up to 3.5x speed up due to SIMD
Summary

- Hash probing reduces efficiency of grouped aggregation

- SIMD accelerated probing not always improves efficiency
  - Specifically, hopscotch hashing has negative impact
  - Linear probing has up to 3.5x speed up due to SIMD

- Hashing technique related parameters also improve the efficiency
Summary

- Hash probing reduces efficiency of grouped aggregation

- SIMD accelerated probing not always improves efficiency
  - Specifically, hopscotch hashing has negative impact
  - Linear probing has up to 3.5x speed up due to SIMD

- Hashing technique related parameters also improve the efficiency

- Further, improvement can be extended using SIMD for multiple insertion
Questions?

Thank You
References


SIMD Cuckoo Hashing – Table Structure

- **Table Structure**
  - Packed key and payload
  - Each bucket contains one key, payload pack

- Pack size = SIMD vector size

- Ross et al. explored SIMD probing [4]
SIMD Acclerated Cuckoo Hashing

Hash Function Compute (Multiplicative hashing)

Table Probe

Adapted from [6]
SIMD vectorized Hashing for Grouped Aggregation

Bala Gurumurthy, David Broneske, Marcus Pinnecke, Gabriel Campero
Durand and Gunter Saake

Search key – K is duplicated

Adapted from [6]
SIMD vectorized Hashing for Grouped Aggregation

Bala Gurumurthy, David Broneske, Marcus Pinnecke, Gabriel Campero Durand and Gunter Saake

SIMD Acclerated Cuckoo Hashing

- Hashing function
- Returns two bucket positions

Adapted from [6]
SIMD Accelerated Cuckoo Hashing

- Probe values in the table
- Compare with search key

Adapted from [6]
SIMD Acclerated Cuckoo Hashing

Comparison result (MASK) are added to payload in the slots

Adapted from [6]
SIMD vectorized Hashing for Grouped Aggregation

Bala Gurumurthy, David Broneske, Marcus Pinnecke, Gabriel Campero
Durand and Gunter Saake

SIMD Acclerated Cuckoo Hashing

If result of all the masks are 0, insertion is performed

Adapted from [6]
Linear Probing

- **Table Structure**
  - Has SoA (Structure of Array format)
  - Key and payload in individual array
  - payload in corresponding key position
Linear Probing – SIMD Code Optimization

Insert

Empty location found

Compare

Value equal

Stop

Values are not equal

SIMD ADD

Slct++

Key Key Key Key

V1 V2 V3 V4

K1 K2 K3 K4

Table slot

SIMD COMPARE

Result

R1 R2 R3 R4

SIMD COPY

Hashing Function

Search key K
Linear Probing – SIMD Code Optimization

Search key – K is duplicated

- Search key K
- SIMD COPY
  - Key
  - Key
  - Key
  - Key

Hashing Function

- Table slot
  - V1
  - V2
  - V3
  - V4
  - K1
  - K2
  - K3
  - K4

SIMD COMPARE

- Result
  - R1
  - R2
  - R3
  - R4

SIMD ADD

- Slct++

Insert

Compare

- Empty location found
- Value equal

Stop
Linear Probing – SIMD Code Optimization

- Scalar hash function
- Table bucket is selected

Search key K → SIMD COPY → Key Key Key Key

Hashing Function

Table slot

V1 V2 V3 V4 K1 K2 K3 K4

SIMD COMPARE

Result R1 R2 R3 R4

SIMD ADD

Insert

Compare

Empty location found

Value equal

Stop

Values are not equal

Slt++
Linear Probing – SIMD Code Optimization

- Search key K
- SIMD COPY
- Hashing Function
- Table slot
- Compare slot values with search key
- SIMD COMPARE
  - Result: R1, R2, R3, R4
  - SIMD ADD
  - Slot++
- Insert
- Compare
  - Empty location found
  - Value equal
  - Stop
- Values are not equal
Linear Probing – SIMD Code Optimization

- Search key K
- SIMD COPY
  - Key
  - Key
  - Key
  - Key
- Hashing Function
  - Table slot
  - V1
  - V2
  - V3
  - V4
  - K1
  - K2
  - K3
  - K4
- SIMD COMPARE
- Result
  - R1
  - R2
  - R3
  - R4
- SIMD ADD
- Slect++
- Empty location found
- Value equal
- Stop
- Values are not equal
- Insert
- Add comparison results with payloads
Linear Probing – SIMD Code Optimization

- Search key K
- SIMD COPY
- Hashing Function
- Table slot
- SIMD COMPARE
- Result
- Insert
- Empty location found
- Values are not equal
- Comparison result
  - Equality – return
  - Inequality – Search next slot
- Value equal
- Stop
- SIMD ADD
- Slot++
- V1 V2 V3 V4
- K1 K2 K3 K4
SIMD Accelerated Hopscotch Hashing

- **SIMD for Probing**
  - Uses SIMD linear probing
  - Probe key within Neighborhood
  - Probe empty space outside

- **SIMD For Insertion**
  - Starts when empty space is found
SIMD Accelerated Hopscotch Hashing

- **SIMD for Probing**
  - Uses SIMD linear probing
  - Probe key within Neighborhood
  - Probe empty space outside

- **SIMD For Insertion**
  - Starts when empty space is found
  - Swap previous values until empty space is inside neighborhood
SIMD Accelerated Hopscotch Hashing

- **SIMD for Probing**
  - Uses SIMD linear probing
  - Probe key within Neighborhood
  - Probe empty space outside

- **SIMD For Insertion**
  - Starts when empty space is found
  - Swap previous values until empty space is inside neighborhood

Swap array holds key to swap
Use SIMD gather to collect keys from hash table
SIMD Accelerated Hopscotch Hashing

- **SIMD for Probing**
  - Uses SIMD linear probing
  - Probe key within Neighborhood
  - Probe empty space outside

- **SIMD For Insertion**
  - Starts when empty space is found
  - Swap previous values until empty space is inside neighborhood

Using SIMD shift to move the keys one step
SIMD Accelerated Hopscotch Hashing

- **SIMD for Probing**
  - Uses SIMD linear probing
  - Probe key within Neighborhood
  - Probe empty space outside

- **SIMD For Insertion**
  - Starts when empty space is found
  - Swap previous values until empty space is inside neighborhood

Using the position the new values are written back