SIMD-Accelerated Hash Tables: Cuckoo Hashing

Rafid Ahmed
Cuckoo Hashing[1]:

- Two locations must be checked for lookup
- Two locations must be checked for deletion
- Inserting a new key may push an older key to another position:
  - alternate position for older key is vacant
  - if not the procedure recurses until an empty slot is found or enters a cycle
## Worst-Case Time Complexity of Hash Tables:

<table>
<thead>
<tr>
<th>Operations</th>
<th>Linear Probing</th>
<th>Double Hashing</th>
<th>Random Hashing</th>
<th>Separate Chaining</th>
<th>Cuckoo Hashing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Deletion</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Insertion</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>
**SIMD (Single Instruction Multiple Data):**

<table>
<thead>
<tr>
<th>Scalar operation</th>
<th>SIMD operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input vector 1</strong></td>
<td>6</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td><strong>Input vector 2</strong></td>
<td>12</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td><strong>Output vector</strong></td>
<td>18</td>
</tr>
<tr>
<td>=</td>
<td>=</td>
</tr>
</tbody>
</table>

**SIMD Register**

\[ O(f(n)) \quad \text{versus} \quad O(f(n)/W) \]
Fundamental Vector Operations:

- **Gather:**

  \[ \text{index vector} \]

  \[
  \begin{bmatrix}
  2 & 9 & 0 & 5 & 11 & 10 & 3 & 9 \\
  19 & 74 & 54 & 23 & 11 & 67 & 99 & 77 & 21 & 29 & 71 & 93 \\
  54 & 29 & 19 & 67 & 93 & 71 & 23 & 29
  \end{bmatrix}
  \]

  \[ \text{memory} \]

  \[ \text{output vector} \]

  \[
  \text{__m256i } \text{_mm256_i32gather_epi64 (__int64 const* base_addr, __m128i vindex, const int scale)}
  \]

SIMD-Accelerated Hash Tables: Cuckoo Hashing
Fundamental Vector Operations:

- Selective Gather:

```
__m256i _mm256_mask_i32gather_epi64 (__m256i src, __int64 const* base_addr, __m128i vindex, __m256i mask, const int scale)
```

![Diagram of Selective Gather operation](image)
**Fundamental Vector Operations:**

- **Selective Store:**

  ![Diagram](image)

  ```
  _m256i _mm256_permutevar8x32_epi32 (_m256i a, _m256i idx)
  
  void _mm256_maskstore_epi32 (int* mem_addr, _m256i mask, _m256i a)
  ```
Cuckoo Hash tables:

- Scalar Hash tables:
  - Branching
    - conditionally access second location
  - Branchless [2]
    - access both locations
Cuckoo Hash tables:

- Vector Hash tables (Horizontal)[3]:

  - Building
    - Hash buckets
    - Contiguous Keys
    - Contiguous Payloads

  - Probing techniques
    - loop unrolling
    - SIMD masking

\[
\begin{aligned}
  T_1 & : \\
  T_2 & :
\end{aligned}
\]
Cuckoo Hash tables:

- **Vector Hash tables (Vertical)**[4]:

  - **Blending**
    - gather all buckets
    - generate masks from comparing keys

  - **Selective**
    - gather first bucket per key
    - selectively gather second bucket for keys that did not match
**Vertical (Selective) Cuckoo Hashing - Probing[4]**:

\[
j \leftarrow 0 \\
\textbf{for } i \leftarrow 0 \textbf{ to } |S| - 1 \textbf{ step } W \textbf{ do} \\
\quad \vec{k} \leftarrow S_{\text{keys}}[i] \quad \text{load input tuples} \\
\quad \vec{v} \leftarrow S_{\text{payloads}}[i] \\
\quad \vec{h}_1 \leftarrow (\vec{k} \cdot f_1) \gg |T| \quad \text{calculate 1st hash value} \\
\quad \vec{h}_2 \leftarrow (\vec{k} \cdot f_2) \gg |T| \quad \text{calculate 2nd hash value} \\
\quad \vec{k}_T \leftarrow T_{\text{keys}}[\vec{h}_1] \quad \text{gather 1st hash bucket} \\
\quad \vec{v}_T \leftarrow T_{\text{payloads}}[\vec{h}_1] \\
\quad m \leftarrow \vec{k} \neq \vec{k}_T \quad \text{compare hash bucket key with probe key} \\
\quad \vec{k}_T \leftarrow m T_{\text{keys}}[\vec{h}_2] \quad \text{selectively gather 2nd hash bucket} \\
\quad \vec{v}_T \leftarrow m T_{\text{payloads}}[\vec{h}_2] \\
\quad m \leftarrow \vec{k} = \vec{k}_T \quad \text{again compare hash bucket key with probe key} \\
\quad \text{selectively store matches} \\
\quad RS_{\text{keys}}[j] \leftarrow m \vec{k} \\
\quad RS_{S_{\text{payloads}}}[j] \leftarrow m \vec{v} \\
\quad RS_{R_{\text{payloads}}}[j] \leftarrow m \vec{v}_T \\
\quad j \leftarrow j + |m| \\
\textbf{end for}
\]
Results:

- Cuckoo Hashing - Probing (30% Selectivity) Benchmark:

![Bar chart showing throughput (GB/s) for hash table sizes ranging from 4 KB to 64 MB. The chart compares Scalar (Branching) and Scalar (Branchless) methods.](chart.png)

Run on (4 X 3500 MHz CPU s)
CPU Caches:
L1 Data 32K (x4)
L1 Instruction 32K (x4)
L2 Unified 256K (x4)
L3 Unified 6144K (x1)
Results:

- **Cuckoo Hashing - Probing (30% Selectivity) Benchmark:**

  ![Graph showing throughput vs hash table size for Cuckoo Hashing benchmarks.](image)

  - **Throughput (GB/s):** The y-axis shows the throughput in gigabytes per second (GB/s).
  - **Hash table size:** The x-axis represents different hash table sizes in kilobytes (KB) and megabytes (MB).
  - **Methods compared:** The chart illustrates performance for different methods:
    - Scalar (Branching)
    - Scalar (Branchless)
    - Vector (Horizontal)

  The chart indicates performance trends across varying hash table sizes and demonstrates the efficiency of the SIMD-accelerated hash tables in Cuckoo Hashing.
Results:

- Cuckoo Hashing - Probing (30% Selectivity) Benchmark:
Results:

- **Cuckoo Hashing - Probing (30% Selectivity) Benchmark:**

Run on (4 X 3500 MHz CPU s)
CPU Caches:
- L1 Data 32K (x4)
- L1 Instruction 32K (x4)
- L2 Unified 256K (x4)
- L3 Unified 6144K (x1)
Results:

- **Cuckoo Hashing - Probing (90% Selectivity) Benchmark:**

The diagram shows the throughput (GB/s) for different hash table sizes ranging from 4 KB to 64 MB. The throughput is measured for scalar (branching) and scalar (branchless) operations. The graph indicates that the throughput increases as the hash table size increases, with scalar (branchless) operations generally outperforming scalar (branching) operations in terms of throughput.
Results:

- Cuckoo Hashing - Probing (90% Selectivity) Benchmark:

![Graph showing throughput vs hash table size for different hash table configurations.](image)
Results:

- Cuckoo Hashing - Probing (90% Selectivity) Benchmark:

![Graph showing throughput vs hash table size]
Results:

- Cuckoo Hashing - Probing (90% Selectivity) Benchmark:

![Graph showing throughput for different hash table sizes and SIMD optimizations.](image)
Summary:

- For both selectivity rate, vertically vectorized code performs about 1.8x faster than horizontal vector and scalar implementation
Sources:


