Evaluation of Parallel Hashing Techniques

Rajesh Bordawekar
IBM Thomas J. Watson Research Center

bordaw@us.ibm.com
Problem Statement

To implement an efficient hash table on GPU to insert and query data in parallel, and evaluate its performance against the state-of-the-art CUDPP hash.
Outline

- Motivation
- Existing Approaches
- Requirements
- New proposal
- Experimental Evaluation
- Summary
Motivation

• Hash Table: a fundamental indexing data structure with uses in multiple domains, e.g., data management, graphics, text analytics, bio-informatics,..

• Memory-bound workload characterized by random memory accesses

• GPU’s impressive device memory bandwidth makes it a good candidate as an execution platform

• Can be used as an off-load accelerator or the primary processor
Design Goals

• Index large datasets

• Exploit massive data-parallelism of the GPU for probing and querying
  • Large thread blocks and large number of threads

• Improve data access locality while querying and insertions

• No additional data structure for querying

• Minimize the use of atomic operations

• High load factor without excessive space utilization (e.g., bins)
Related Approaches

- Probing based approaches (e.g., linear, quadratic)
- Cuckoo Hashing and its variants [Pagh, Rodler 01]
- Robin-hood Hashing
- Hop-scotch Hashing [Herlihy, Shavit, Tzafrir 08]
Cuckoo Hashing

- Eviction-based conflict resolution

- Uses multiple hash tables with distinct hash functions
  - Upon conflict, i.e., position[y]=h(x), x is stored at position[y], and a new location for y is selected using g(y)
  - Process continues until an empty spot is found

- For querying, a fixed number of probes are required

- A variant, Robin-hood hash, uses eviction based on entry age
Hop-scotch Hashing

- Tries to store conflicted entries closer to the conflict location (called Hop region)

- While querying, items that map to the same location can be fetched using few consecutive cache lines

- Common sizes of hop region (hop distance) 4 or 32

- Key idea: Create space in the hop region by repeated exchange of empty space with allocated entries

- Exchange always happens within hop distance

- Uses hop-map to store locations of entries within hop region
Cuckoo Hashing on GPUs

- Cuckoo Hash [Alcantara 09,11] (released as a part of CUDPP)

- Coherent Parallel hashing [Garcia et al, 11] uses the Robin-hood hashing approach

- CUDPP hash version uses 4 hash functions and a constant size stash (100 elements)
  - Uses 1.25*N space
  - 64 Threads per block
  - At most 5 accesses for querying
Cuckoo Hashing Properties

- Requires fixed, small number of accesses while querying
- Requires least amount of space among different hashing approaches
- Chained evictions may not terminate
- Cuckoo hash cannot consider data locality
- Needs to use multiple hash functions to improve load factor
- Requires smaller-sized thread blocks
- Number of atomic operations not dependent on input data size
Hop-Scotch Hashing Properties

- Multiple conflicted values lie within the hop-scotch region
- May require multiple evictions
- Chained evictions may not find empty slots
- Hop-scotch hash requires additional data structure for querying
- Number of atomic operations not dependent on input data size
Multi-level Bounded Linear Probing

- Multiple hierarchical (in practice, 2 or 3) hash maps of varying sizes
  - map size decreases as the map level (depth) increases
  - can be extended to support buckets
- Each map uses a different universal hash function
- Linear probing on conflicts: search \textit{fixed-sized} regions for empty spaces
  - Probe region size multiple of 4 bytes or cache line size (if greater than 128 bytes)
  - Size of the region varies per map level; for higher levels, probing region is small
- Lock-free concurrent implementation using 32-bit atomic CAS
Design Alternatives

- Two-level hash table with two maps, each with its own hash function
  - secondary map size: 0.25*N
  - Primary probe region: 4, secondary probe region: 256

- Two-level hash table with 2 maps; two-element buckets associated with the smaller map
  - secondary map size: 2*0.25*N
  - Primary probe region: 4, secondary probe region: 32

- Three-level hash table with 3 maps
  - secondary map size: 0.2*N, tertiary map size: 0.1*N
  - Primary probe region: 4, secondary probe region: 8, tertiary probe region: 8
A Three-level Hash Table

Number of items to be indexed: $N$

Primary Map: $1.05 \times N$

Secondary Map: $0.20 \times N$

Tertiary Map: $0.10 \times N$
size(primary probe region) <= size(secondary probe region) < size(tertiary probe region)
Implementation Details

- Works for 32-bit keys and values

- Keys and values stored separately. Keys locked using 32-bit atomicCAS()

- Uses universal hash functions
  - \[ \text{hash}(x) = \left( (\text{constant}_a \times x + \text{constant}_b) \mod P \right) \mod \text{Size} \]

- Each level uses a different hash function

- Size of the thread blocks varies from 32 to 512. Number of thread blocks is not limited.
Advantages of the approach

• Overall space utilization similar to that of cuckoo hashing, with higher load factor

• Only uses fast single-precision atomic CAS operations
  • Number of CAS operations in the order of input data items (N)

• Multiple levels enable bounded probing and reduce the size of bounded probe region
  • Probe region sizes: (4, 256) for two levels, (4, 8, 8) for three levels

• Probe region size enables hardware coalescing and exploits locality during insertion and querying

• No restrictions on the number of threads in the thread block

• Enables data-parallel implementation of querying functions
Device Memory Access Optimizations

Lanes in a warp

probe region = 4

4 lanes fetching four consecutive elements
Data-parallel Querying

1. hasKey() on input key, x, probe region=32
2. Each warp lane “i” fetches value at h(x)+i
3. Use warp vote function __any()

Each warp lane executes __any(x, skey)
Experimental Setup

- Experiments executed on Tesla K40c using CUDA 5.5
- Compared against Cuckoo Hash implementation from CUDPP-2.1 (Oct 2013)
  - ./cudpp_hash_testrig -basic -n{size}
  - Compared build and 0% failure times
  - Using the same random datasets generated via CUDPP test suite
- Evaluated two-level, two-level with 2-element buckets, and three-level hash functions
  - Two-level: Primary probe region=4, secondary probe region=256
  - Two-level Bucketing: Primary probe region=4, secondary probe region=32
  - Three-level: Primary probe region=4, secondary/tertiary probe region=8
- 256 threads in a block, primary map=1.05N, secondary map=0.25/0.20, tertiary map=0.1N
## Initial Results: Hash Table Build

Number of Key-Value Pairs (Time in ms)

<table>
<thead>
<tr>
<th>METHOD</th>
<th>1 M</th>
<th>10 M</th>
<th>16 M</th>
<th>32 M</th>
<th>64 M</th>
<th>128 M</th>
<th>256 M</th>
</tr>
</thead>
<tbody>
<tr>
<td>TWO LEVEL</td>
<td>3.1</td>
<td>38.7</td>
<td>62.6</td>
<td>126.4</td>
<td>270.9</td>
<td>614.6</td>
<td>2040</td>
</tr>
<tr>
<td>TWO LEVEL BUCKET</td>
<td>3.26</td>
<td>40.6</td>
<td>65.6</td>
<td>132.5</td>
<td>270.8</td>
<td>621.6</td>
<td>2220</td>
</tr>
<tr>
<td>THREE LEVEL</td>
<td>3.28</td>
<td>40.7</td>
<td>65.8</td>
<td>132.6</td>
<td>271.4</td>
<td>618</td>
<td>2100</td>
</tr>
<tr>
<td>CUDPP</td>
<td>5</td>
<td>57.5</td>
<td>88.7</td>
<td>177.9</td>
<td>354.3</td>
<td>700</td>
<td>1500</td>
</tr>
</tbody>
</table>
## Initial Results: Hash Table Querying

Number of Key-Value Pairs (Time in ms)

<table>
<thead>
<tr>
<th>METHOD</th>
<th>1 M</th>
<th>10 M</th>
<th>16 M</th>
<th>32 M</th>
<th>64 M</th>
<th>128 M</th>
<th>256 M</th>
</tr>
</thead>
<tbody>
<tr>
<td>TWO LEVEL</td>
<td>0.858</td>
<td>10.6</td>
<td>17.1</td>
<td>35</td>
<td>153</td>
<td>426</td>
<td>982.8</td>
</tr>
<tr>
<td>THREE LEVEL</td>
<td>0.782</td>
<td>10.7</td>
<td>17.5</td>
<td>37.6</td>
<td>194.8</td>
<td>534</td>
<td>1210</td>
</tr>
<tr>
<td>TWO LEVEL</td>
<td>1.5</td>
<td>17.6</td>
<td>28.4</td>
<td>84.7</td>
<td>265.5</td>
<td>651.4</td>
<td>2010</td>
</tr>
<tr>
<td>THREE LEVEL</td>
<td>1.4</td>
<td>18</td>
<td>29.5</td>
<td>93.9</td>
<td>303.8</td>
<td>731</td>
<td>1650</td>
</tr>
<tr>
<td>CUDPP</td>
<td>1.5</td>
<td>16.7</td>
<td>26.8</td>
<td>67.7</td>
<td>191.9</td>
<td>442</td>
<td>964</td>
</tr>
</tbody>
</table>

Two types of queries: `hasKey()` and `getValue()`
Discussion

• In all cases, around 85% of data fits in the primary map

• Large probe regions have no effect on performance

• Reducing thread block size reduces build performance slightly; no impact on query performance. Using 512 threads leads to no improvement

• Data-parallel querying and probe region optimizations not effective

• For large data sets, build and query performance affected by non-coalesced write accesses and larger number of read accesses

  • Cuckoo Hashing invokes a fixed, smaller number of read accesses
Summary

- Competitive performance to cuckoo hashing
  - Build-phase better than cuckoo hashing, query performance slightly lower
  - Similar space utilization (1.25 to 1.30*N)
  - Non-coalesced write accesses and conditionals affect the performance
- Further optimizations needed to reduce the memory access costs
  - Warp-level coalescing, co-locating data and values
- Detailed analysis and multi-core CPU implementation underway
Insertion Process

- For input value x, first compute position in the primary map using the hash function h1, \( pos = h1(x) \)
  - If \( map[pos] == 0 \), return SUCCESS (case 2)
  - if \( map[pos] != 0 \), find empty slot in the primary probing region (cases 1 and 3)
    - If empty slot found, return SUCCESS (case 3.prob)
    - If no empty slot found, proceed to the secondary map (case 1)
  - Compute position in the secondary map using the second hash function h2, \( pos2 = h2(x) \)
    - repeat the process for insertion for the secondary map, but using the secondary probing region (cases 1.1, 1.2, 1.3)
    - If no empty slot found, proceed to the tertiary map (case 1.2)
  - Compute position in the tertiary map using the third hash function h3, \( pos3 = h3(x) \)
    - repeat the process for insertion for the tertiary map, but using the tertiary probing region (cases 1.1.1, 1.1.2, 1.1.3)
    - If no empty slot found, proceed to the tertiary map (case 1.1.1)
- \( size(\text{primary probing region}) \leq size(\text{secondary probing region}) < size(\text{tertiary probing region}) \)
Query Process

• For input value x, first compute position in the primary map using the hash function $h_1$, $pos = h_1(x)$

  • If $map[pos] == x$, return SUCCESS (case 2)

  • if $map[pos] != 0$, find used slots in the primary probing region (cases 1 and 3)
    
    • If one of the values at these slots, matches x, return SUCCESS (case 3)

    • If there is no match, proceed to the secondary map (case 1)

  • else if $map[pos] == 0$, return FAILURE

• Compute position in the secondary map using the second hash function $h_2$, $pos_2 = h_2(x)$

  • repeat the process for querying the secondary map, but using the secondary probing region (cases 1.1, 1.2, 1.3), return SUCCESS on a match (cases 1.1, and 1.3)

  • If there is no match, proceed to the tertiary map (case 1.2)

• Compute position in the tertiary map using the third hash function $h_3$, $pos_3 = h_3(x)$

  • repeat the process for querying the tertiary map, but using the tertiary probing region (cases 1.1.1, 1.1.2, 1.1.3), return SUCCESS on a match (cases 1.1.2, and 1.1.3)

  • If there is no match, return FAILURE (case 1.1.1)

• Return the corresponding values when invoked via GetValue(key, value)