SCALING JOINS TO A THOUSAND GPUs

Hao Gao, Nikolay Sakharnykh, August 16, 2021
MEMORY PERFORMANCE
Why GPUs are suitable for join?

- **AMD EPYC 7742 CPU (Rome)**
  - DDR4: 8 channels, 64-bit per channel, 3200MT/s
  - Peak memory bandwidth **205GB/s**
  - Measured random 8B access **4.52GB/s**

- **NVIDIA A100 GPU (Ampere)**
  - HBM2e: 5120-bit bus width, 1512MHz
  - Peak memory bandwidth **1935GB/s**
  - Measured random 8B access **134GB/s**
  - But only **80GB** capacity

- What if we need more than 80GB?
  1) Spill to CPU memory
  2) **Scale out to multiple GPUs** ----> This talk.

---

Critical for performance to specify **GPU-NIC affinity**
CLUSTER TOPOLOGY

- Full fat-tree topology
- Three levels of switches: leaf level, spine level and core level
- Rail optimized: core-level switches are only used for cross-rail or cross-SuperPod traffic
REPARTITIONED JOIN

Step 1: Partition tables into #GPUs according to the hash value of each row

Step 2: Shuffle all-to-all communication

Step 3: Insert rows of left table into a hash table

Step 4: Look up each right table row in the hash table
HASH PARTITION

Naive implementation:
Dominated by the global memory random accesses

Optimized implementation:
Random in shared memory, sequential in global memory
### LOCAL JOIN

- No-partitioned hash-based join
- Store references instead of values inside the hash table

#### Probe Table

<table>
<thead>
<tr>
<th>l_orderkey</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>56</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>39</td>
</tr>
<tr>
<td>27</td>
</tr>
<tr>
<td>23</td>
</tr>
</tbody>
</table>

#### Hash Table

<table>
<thead>
<tr>
<th>Hash value</th>
<th>Row idx</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash(27)</td>
<td>2</td>
</tr>
<tr>
<td>hash(23)</td>
<td>1</td>
</tr>
<tr>
<td>hash(11)</td>
<td>0</td>
</tr>
<tr>
<td>hash(29)</td>
<td>3</td>
</tr>
</tbody>
</table>

#### Build Table

<table>
<thead>
<tr>
<th>o_orderkey</th>
<th>o_orderpriority</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>23</td>
<td>5</td>
</tr>
<tr>
<td>27</td>
<td>2</td>
</tr>
<tr>
<td>29</td>
<td>4</td>
</tr>
</tbody>
</table>

Key is not found
LOCAL JOIN

- No-partitioned join vs. partitioned join
  - Advantages: lower memory consumption
  - Disadvantages: higher cache miss rate

- Store references vs. store values
  - Advantages: can scale to large keys/payloads
  - Disadvantages: extra random reads during probing

- In principle repartitioned distributed join can use another single-GPU join as well (including partitioned join)
WEAK-SCALING PERFORMANCE ON TPC-H DATASET

Left table: \textit{l\_orderkey}
Right table: \textit{o\_orderkey} and \textit{o\_orderpriority}

At small scale, the join performance is close to IB BW
At large scale, the gap grows due to all-to-all inefficiency

Solution: use compression
COMMUNICATION WITH COMPRESSION

GPU0
Input Partitioned Table → Compressed Input Table → All-to-All Communication → Shuffled Compressed Table → Decompression → Shuffled Table

GPU1
Input Partitioned Table → Compressed Input Table → All-to-All Communication → Shuffled Compressed Table → Decompression → Shuffled Table

GPU2
Input Partitioned Table → Compressed Input Table → All-to-All Communication → Shuffled Compressed Table → Decompression → Shuffled Table
RUN LENGTH ENCODING (RLE)

- Idea: compress repeated values into (value, run length) pair

Compress 22 integers into 10!

In general, compression ratio is data dependent. Worst-case scenario, RLE can increase input by a factor of 2.
DELTA ENCODING

- Idea: compute the difference relative to the previous value

<table>
<thead>
<tr>
<th>Input Sequence</th>
<th>15000</th>
<th>15001</th>
<th>15002</th>
<th>15003</th>
<th>15004</th>
<th>15204</th>
<th>15104</th>
<th>15103</th>
<th>15102</th>
<th>15101</th>
<th>15100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compressed Sequence</td>
<td>15000</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>200</td>
<td>-100</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

- Does not compress by itself
- But the output is easier to compress by RLE or bitpacking
FOR AND BIT-PACKING

- Idea: use smallest number of bits to represent a range

<table>
<thead>
<tr>
<th>Input Sequence</th>
<th>Compressed Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>15000 15001 15002 15003 15004 15204 15104 15103 15102 15101 15100</td>
<td>0 1 2 3 4 204 104 103 102 101 100 + 15000</td>
</tr>
</tbody>
</table>

Frame of reference (FOR)

all values < 256; use 8 bits
CASCADED COMPRESSOR

Combining the blocks together: RLE + Delta + FOR + bit-packing

Dictionary

A,A,B,B,C,C

1,1,2,2,3,3

RLE

2,2,2

runs

val

1,2,3

Delta

1,1,1

FOR + Bit-packing

2:0,0,0

FOR + Bit-packing

3

RLE

3:0

Compressed

Uncompressed

Scheme: RLE=2, Delta=1, bit-packing=1
IMPLEMENTING CASCADED COMPRESSOR

- Important to **batch** and **fuse** all layers (i.e., all layers of all partitions are within a single kernel)
- Compression scheme (number of RLE/Delta/bitpacking layers) important for both ratio and throughput
- **Sample** and **benchmark** the input columns for choosing the best scheme (time not included for join).

<table>
<thead>
<tr>
<th></th>
<th>RLE</th>
<th>Delta</th>
<th>bitpacking</th>
</tr>
</thead>
<tbody>
<tr>
<td>O_ORDERKEY</td>
<td>2</td>
<td>1</td>
<td>True</td>
</tr>
<tr>
<td>O_ORDERPRIORITY</td>
<td>0</td>
<td>0</td>
<td>True</td>
</tr>
<tr>
<td>L_ORDERKEY</td>
<td>2</td>
<td>1</td>
<td>True</td>
</tr>
</tbody>
</table>
WEAK-SCALING PERFORMANCE WITH COMPRESSION

Best performance w/ compression: 1.79 trillion tuples/s

Join with compression also scales well

Throughput [GB/s]

#GPUs

512 1024 2048 4096 8192 16384

32 64 128 256 512 1024

w/o compression  w/ compression

w/o comp  w/ comp

64 GPUs  256 GPUs  1024 GPUs

Hash Partition  Compression  Communication  Decompression  Local Join
SMALL TABLES PERFORMANCE

- Fixed 512 GPUs
- Each table has two columns
  - keys: random unique 8B integers
  - payloads: 8B row index
- Selectivity = 0.3
- No compression

Throughput drops with small table size due to Infiniband latency
TWO-LEVEL COMMUNICATION

Suppose there are $M$ nodes and $N$ GPUs/node.

- **One-level shuffle**
  - Hash Partition into $M \times N$ partitions
  - Send each partition to a remote GPU through IB
  - Local Join

- **Two-level shuffle**
  - Hash Partition into $M$ partitions
  - Send each partition to a remote GPU within the HCA plane
  - Hash Partition into $N$ partitions
  - Send each partition to a local GPU through NVLink
  - Local Join

Compression not needed because NVLink throughput is high (300GB/s per GPU)

- Advantage: take advantage of the rail-optimized topology
- Advantage: less Infiniband messages
- Disadvantage: extra hash partition and communication stages
- Forward-looking: use shared-memory instead of message-passing within a node (help overlap and avoid copies)
TWO-LEVEL COMMUNICATION - SMALL TABLES

- Fixed 512 GPUs
- Each table has two columns
  - keys: random unique 8B integers
  - payloads: 8B row index
- Selectivity = 0.3
- No compression
- 261632 IB messages for 1-level
  - 4032 IB messages for 2-level

2-level communication is more efficient for latency-bound workloads
TWO-LEVEL COMMUNICATION - PERFORMANCE

Performance gap is closed when #GPUs is large due to less IB messages.

At small scale, 2-level is less efficient due to extra stages.
CONCLUSION

▷ Distributed repartitioned hash-join is scalable up to 1024 GPUs

▷ Without compression, join time is dominated by the communication stage

▷ Cascaded compression is efficient and scalable. Improves performance by 1.77x on 1024 GPUs

▷ Two-level communication is helpful for latency-limited scenarios

Our code: https://github.com/rapidsai/distributed-join
FUTURE IMPROVEMENTS

- Computation and communication overlap
- Within a node, use shared-memory instead of message-passing
- Compression on strings
- Dataset with skewed keys
- Cache-friendly local join