GPU ACCELERATION FOR OLAP

Tim Kaldewey, Jiri Kraus, Nikolay Sakharnykh
03/26/2018
A TYPICAL ANALYTICS QUERY
From a business question to SQL

Business question (TPC-H query 4)
- Determines how well the order priority system is working and gives an assessment of customer satisfaction
- Counts the number of orders

SQL
```
select
    o_orderpriority,
    count(o_orderkey) as order_count,
from
    orders
where
    o_orderdate >= date '[DATE]'
    and o_orderdate < date '[DATE]'
    + interval '3' month
    and exists
        (select *
         from lineitem
         where l_orderkey = o_orderkey
         and l_commitdate < l_receiptdate)
order by
    o_orderpriority;
```

Examples: DATE = 1/1/1993, 1/4/1993, ...
A TYPICAL ANALYTICS QUERY

From SQL to Database Operators

Database Operators

- aggregate
- predicate (filter)
- join
- predicate (filter)
- aggregate
- sort

SQL

```sql
select
  o_orderpriority,
  count(o_orderkey) as order_count,
from
  orders
where
  o_orderdate >= date '[DATE]' and
  o_orderdate < date '[DATE]' + interval '3' month and
exists (select * from lineitem
  where l_orderkey = o_orderkey and
  l_commitdate < l_receiptdate)

  group by
  o_orderpriority,
  order by
  o_orderpriority;
```
A TYPICAL ANALYTICS QUERY
Joins are implicit in a business question

Business question
Counts the number of orders ordered in a given quarter of a given year in which at least one lineitem was received by the customer later than its committed date. The query lists the count of such orders for each order priority sorted in ascending priority order.

SQL
```
select
  o_orderpriority,
  count(o_orderkey) as order_count,
from
  orders
where
  o_orderdate >= date '[DATE]' and
  o_orderdate < date '[DATE]' + interval '3' month and
  exists (select * from lineitem
          where l_orderkey = o_orderkey and
                l_commitdate < l_receiptdate)

group by
  o_orderpriority,
order by
  o_orderpriority;
```

Database Operators
- aggregate
- predicate (filter)
- join
- predicate (filter)
- aggregate
- sort
TPC-H SCHEMA

- **part (p_)**
  - PARTKEY
  - NAME
  - MFGR
  - CATEGORY
  - BRAND
  - ...

- **supplier (s_)**
  - SUPPKEY
  - NAME
  - ADDRESS
  - CITY
  - NATIONKEY
  - ...

- **lineitem (l_)**
  - ORDERKEY
  - LINENUMBER
  - PARTKEY
  - SUPPKEY
  - COMMITDATE
  - RECEIPTDATE
  - ...
  - ...

- **order (o_)**
  - ORDERKEY
  - CUSTKEY
  - ORDERDATE
  - ORDPRIORITY
  - ORDERSTATUS
  - ...

- **customer (c_)**
  - CUSTKEY
  - NAME
  - ADDRESS
  - CITY
  - ...

- **nation (n_)**
  - NATIONKEY
  - NAME
  - ...

- **customer (n_)**
  - CUSTKEY
  - NAME
  - ADDRESS
  - CITY
  - ...

- **nation (n_)**
  - NATIONKEY
  - NAME
  - ...

- **part (p_)**
  - PARTKEY
  - NAME
  - MFGR
  - CATEGORY
  - BRAND
  - ...

- **supplier (s_)**
  - SUPPKEY
  - NAME
  - ADDRESS
  - CITY
  - NATIONKEY
  - ...

- **lineitem (l_)**
  - ORDERKEY
  - LINENUMBER
  - PARTKEY
  - SUPPKEY
  - COMMITDATE
  - RECEIPTDATE
  - ...
  - ...

- **order (o_)**
  - ORDERKEY
  - CUSTKEY
  - ORDERDATE
  - ORDPRIORITY
  - ORDERSTATUS
  - ...

- **customer (c_)**
  - CUSTKEY
  - NAME
  - ADDRESS
  - CITY
  - ...

- **nation (n_)**
  - NATIONKEY
  - NAME
  - ...

- **part (p_)**
  - PARTKEY
  - NAME
  - MFGR
  - CATEGORY
  - BRAND
  - ...

- **supplier (s_)**
  - SUPPKEY
  - NAME
  - ADDRESS
  - CITY
  - NATIONKEY
  - ...

- **lineitem (l_)**
  - ORDERKEY
  - LINENUMBER
  - PARTKEY
  - SUPPKEY
  - COMMITDATE
  - RECEIPTDATE
  - ...
  - ...

- **order (o_)**
  - ORDERKEY
  - CUSTKEY
  - ORDERDATE
  - ORDPRIORITY
  - ORDERSTATUS
  - ...

- **customer (c_)**
  - CUSTKEY
  - NAME
  - ADDRESS
  - CITY
  - ...

- **nation (n_)**
  - NATIONKEY
  - NAME
  - ...

- **part (p_)**
  - PARTKEY
  - NAME
  - MFGR
  - CATEGORY
  - BRAND
  - ...

- **supplier (s_)**
  - SUPPKEY
  - NAME
  - ADDRESS
  - CITY
  - NATIONKEY
  - ...

- **lineitem (l_)**
  - ORDERKEY
  - LINENUMBER
  - PARTKEY
  - SUPPKEY
  - COMMITDATE
  - RECEIPTDATE
  - ...
  - ...

- **order (o_)**
  - ORDERKEY
  - CUSTKEY
  - ORDERDATE
  - ORDPRIORITY
  - ORDERSTATUS
  - ...

- **customer (c_)**
  - CUSTKEY
  - NAME
  - ADDRESS
  - CITY
  - ...

- **nation (n_)**
  - NATIONKEY
  - NAME
  - ...
WHERE DOES TIME GO?
TPC-H query 4 @SF10 = 10GB data warehouse

```sql
select
    o_orderpriority,
    count(o_orderkey) as order_count,
from
    orders
where
    o_orderdate >= date '[DATE]' and
    o_orderdate < date '[DATE]' + interval '3' month and
exists (select * from lineitem
        where l_orderkey = o_orderkey and
        l_commitdate < l_receiptdate)

group by
    o_orderpriority,
order by
    o_orderpriority;
```

CPU execution breakdown

- **join**: 1%
- **group-by**: 99%

18/22 TPC-H Queries involve Join & are the longest running ones
## RELATIONAL JOIN

<table>
<thead>
<tr>
<th>Lineitem&lt;sup&gt;1&lt;/sup&gt;</th>
<th>Order&lt;sup&gt;2&lt;/sup&gt;</th>
<th>Join Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l_orderkey</td>
<td>o_orderkey</td>
<td>o_orderkey</td>
</tr>
<tr>
<td>23</td>
<td>11</td>
<td>23</td>
</tr>
<tr>
<td>14</td>
<td>23</td>
<td>11</td>
</tr>
<tr>
<td>56</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>11</td>
<td>29</td>
<td>23</td>
</tr>
<tr>
<td>39</td>
<td></td>
<td>39</td>
</tr>
<tr>
<td>27</td>
<td></td>
<td>27</td>
</tr>
<tr>
<td>23</td>
<td></td>
<td>23</td>
</tr>
</tbody>
</table>

1. after applying predicate “\(l\_commitdate < l\_receiptdate\)"
2. after applying predicates “\(o\_orderdate \geq \text{date} \['\text{DATE}'\] \) and \(o\_orderdate < \text{date} \['\text{DATE}'\] + \text{interval} \ '3' \text{ month}’”
**HASH JOIN**

1. **Lineitem**
   - l_orderkey:
     - 23
     - 14
     - 56
     - 11
     - 39
     - 27
     - 23

2. **Order**
   - o_orderkey, o_orderpriority:
     - 11, 1
     - 23, 5
     - 27, 2
     - 29, 4

3. **Join Results**
   - o_orderkey, o_orderpriority:
     - 23, 5
     - 11, 1
     - 27, 2
     - 23, 5

### Build hash table

1. **Foreign Key**
   - = Probe inputs

2. **Primary Key**
   - = Payload

---

1. after applying predicate “l_commitdate < l_receiptdate”
2. after applying predicates “o_orderdate >= date ‘[DATE]’ and o_orderdate < date ‘[DATE]’ + interval ‘3’ month”
HASH JOIN
General approach - including payload(s)

1. **Build a hash table**

   - **Build table**
     - \( k_1, p_1 \)
     - \( k_2, p_2 \)

   - **Hash table**
     - \( k_1, p_1 \)
     - \( k_2, p_2 \)

1. Compute \( h(k) \) - \( k \) is a primary (unique) key
2. Insert \( k \) with payload \( p \) into hashed location
3. If occupied, insert into next free one
HASH JOIN
General approach - including payload(s)

1. Build a hash table
   - Compute $h(k)$ - $k$ is a primary (unique) key
   - Insert $k$ with payload $p$ into hashed location
   - If occupied, insert into next free one

2. Probe the hash table
   - Compute $h(fk)$ - $fk$ is foreign key
   - Compare $fk$ and key $k$ in the hash table
   - If there is a match store the result
HASH JOIN
General approach - including payload(s)

1. Build a hash table

1. Compute $h(k)$ - $k$ is a primary (unique) key
2. Insert $k$ with payload $p$ into hashed location
3. If occupied, insert into next free one

Build and Probe both produce a random data access pattern!
RANDOM ACCESS PERFORMANCE

Intel Xeon Gold 6140 (Skylake)

- DDR4: 6 channels, 64-bit per channel, 2666MT/s, tFAW=30ns*
- Peak memory bandwidth 120GB/s
- Random 8B access 6GB/s

NVIDIA Tesla V100 (Volta)

- Peak memory bandwidth 900GB/s
- Random 8B access 60GB/s

HETEROGENEOUS SYSTEM

Faster access

16GB HBM

16GB HBM

Multiple TBs DDR

Larger capacity
HETEROGENEOUS SYSTEM
Data Placement & Movement

Faster access

<table>
<thead>
<tr>
<th>Key</th>
<th>Payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>5</td>
</tr>
<tr>
<td>27</td>
<td>2</td>
</tr>
</tbody>
</table>

Hash Table(s)

Build & Probe

Multiple TBs DDR

Larger capacity

16GB HBM

16GB HBM

DB
INTERCONNECT SPEEDS

- PCIe3: 16GB/s
- NVLINK1: 20GB/s per link, up to 2 links between CPU/GPU = 40GB/s
- NVLINK2: 25GB/s per link, up to 3 links = 75GB/s
- Further increase throughput via compression: see S8417 tomorrow at 2pm*

GPU joins (HT probing):
- PCIe/NVLINK1: GPU random access bandwidth >> CPU-GPU bandwidth
- NVLINK2 could be limited by GPU random access performance

* S8417 - Breaking the Speed of Interconnect with Compression for Database Applications - Tuesday, Mar 27, 2:00pm - Room 210F
JOIN OPERATOR ACCELERATION
IMPLEMENTING HASH JOIN

**concurrent_unordered_map**

```
template <typename Key, typename Element,
    Key unused_key, Element unused_element,
    typename Hasher = default_hash<Key>,
    typename Equality = equal_to<Key>,
    typename Allocator = managed_allocator<thrust::pair<Key, Element>>>

class concurrent_unordered_map : public managed {
    public: //omitting typedef*
        explicit concurrent_unordered_map(size_type n,
            const Hasher& hf, const Equality& eql,
            const allocator_type& a);

        _host____device__ iterator begin();
        _host____device__ iterator end();
        _host____device__ iterator insert(const value_type& x);
        _host____device__ const_iterator find(const key_type& k) const;
        void prefetch(const int dev_id ); }
```

Hash table in Unified Memory can be accessed on the host and device

**Features:**
- Construction on CPU
- Works on CPU and GPU
- Concurrent insert’s
- Concurrent find
- Concurrent insert and find*  
- No concurrent CPU and GPU insert (currently)

*No visibility guarantees
IMPLEMENTING HASH JOIN

1. Build a hash table with a `concurrent_unordered_map`

```cpp
template<typename map_type>
__global__ void build_hash_tbl(
    map_type * const map,
    const typename map_type::key_type* const build_tbl,
    const typename map_type::size_type build_tbl_size)
{
    const typename map_type::mapped_type i = tIdx.x + bIdx.x * bDim.x;
    if ( i < build_tbl_size ) {
        map->insert( thrust::make_pair( build_tbl[i], i ) );
    }
}
```
IMPLEMENTING HASH JOIN

Concurrent Inserts with atomicCAS

```
iterator insert(const value_type& x) {
    size_type key_hash = hf(x.first);
    size_type ht_idx = key_hash % ht_size;
    value_type* it = 0;
    while (0 == it) {
        value_type* tmp_it = ht_values + ht_idx;
        const key_type old_key = atomicCAS(
            &(tmp_it->first), unused_key, x.first);
        if (equal(unused_key, old_key) ||
            equal(x.first, old_key)) {
            (ht_values + ht_idx)->second = x.second;
            it = tmp_it;
        }
        ht_idx = (ht_idx + 1) % ht_size;
    }
    return iterator(ht_values, ht_values + ht_size, it);
}
```
iterator insert(const value_type& x) {
    size_type key_hash = hf(x.first);
    size_type ht_idx = key_hash%ht_size;
    value_type* it = 0;
    while (0 == it) {
        value_type* tmp_it = ht_values+ht_idx;
        const key_type old_key = atomicCAS(
            &(tmp_it->first), unused_key, x.first);
        if (equal(unused_key, old_key) ||
            equal(x.first, old_key)) {
            (ht_values+ht_idx)->second = x.second;
            it = tmp_it;
        }
        ht_idx = (ht_idx+1)%ht_size;
    }
    return iterator(ht_values,ht_values+ht_size,it);
}

**IMPLEMENTING HASH JOIN**

Build Optimization: merge 4B keys with 4B values

```cpp
typedef unsigned long long int uint64;
union p2ll {
    uint64 longlong;
    value_type pair;
};

p2ll conv = {0ull};
conv.pair = make_pair(unused_key, unused_element);
uint64 unused = conv.longlong;
conv.pair = x;
uint64 value = conv.longlong;
uint64 old_val = atomicCAS(
    reinterpret_cast<uint64*>(tmp_it),
    unused, value);
if (old_val == unused) {
    it = tmp_it;
    else fall back
```
IMPLEMENTING HASH JOIN

Build Optimization: merge 4B keys with 4B values

Tesla V100 PCI-E build table size 16777216, 4B keys, 4B values

All data in device memory
IMPLEMENTING HASH JOIN

Build Performance 8B keys with 8B values

Tesla V100 PCI-E build table size 16777216, 8B keys, 8B values

All data in device memory
IMPLEMENTING HASH JOIN

2. Probe the hash table with a `concurrent_unordered_map`

```cpp
template<typename map_type, typename key_type,
         typename size_type, typename joined_type>
__global__ void probe_hash_tbl(
    map_type * map,
    const key_type* probe_tbl, const size_type probe_tbl_size,
    joined_type * const joined, size_type* const current_idx)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < probe_tbl_size) {
        const auto end = map->end(); auto it = map->find(probe_tbl[i]);
        if (end != it) {
            joined_type joined_val; joined_val.y = i; joined_val.x = it->second;
            int my_current_idx = atomicAdd( current_idx, 1 );
            joined[my_current_idx] = joined_val;
        }
    }
}
```
IMPLEMENTING HASH JOIN
Probe Optimization: output cache for result materialization I (init)

```cpp
template<typename map_type, ... ,int block_size>
__global__ void probe_hash_tbl(map_type * map,
    const key_type* probe_tbl, const size_type probe_tbl_size,
    joined_type * const joined, size_type* const current_idx)
{
    __shared__ int current_idx_shared;
    __shared__ int output_offset_shared;
    __shared__ joined_type joined_shared[block_size];
    if ( 0 == threadIdx.x ) {
        output_offset_shared = 0; current_idx_shared = 0;
    }
    __syncthreads();
}
```

IMPLEMENTING HASH JOIN
Probe Optimization: output cache for result materialization II (use)

```cpp
int i = threadIdx.x + blockIdx.x * blockDim.x;
if ( i < probe_tbl_size ) {
    const auto end = map->end();
    auto it = map->find(probe_tbl[i]);
    if ( end != it ) {
        joined_type joined_val;
        joined_val.y = i; joined_val.x = it->second;
        int my_current_idx = atomicAdd(&current_idx_shared, 1);
        // its guaranteed to fit into the shared cache
        joined_shared[my_current_idx] = joined_val;
    }
}
__syncthreads();
```
IMPLEMENTING HASH JOIN

Probes Optimization: output cache for result materialization III (flush)

```c
if ( current_idx_shared > 0 ) {
    if ( 0 == threadIdx.x )
        output_offset_shared = atomicAdd( current_idx, current_idx_shared );
    __syncthreads();
    if ( threadIdx.x < current_idx_shared )
        joined[output_offset_shared+threadIdx.x] = joined_shared[threadIdx.x];
}
```
IMPLEMENTING HASH JOIN

Probe Optimization: output cache for result materialization

Tesla V100 PCI-E probe table size 134217728, 8byte keys, 8byte values

Hash table and input in device memory

TPC-H QUERY 4: SINGLE JOIN
TPC-H QUERY 4
Order Priority Checking Query

```
select
    o_orderpriority, count(o_orderkey) as order_count,
from
    orders
where

    o_orderdate >= date 'DATE'
and o_orderdate < date 'DATE' + interval '3' month
and exists (select * from lineitem
    where l_orderkey = o_orderkey
    and l_commitdate < l_receiptdate)

group by o_orderpriority, order by o_orderpriority;
```
Q4: GPU ALGORITHM

Part 1

```
select o_orderpriority, count(o_orderkey) as order_count, from orders
where o_orderdate >= date '[DATE]' and o_orderdate < date '[DATE]' + interval '3' month
and exists (select * from lineitem
    where l_orderkey = o_orderkey and l_commitdate < l_receiptdate)

```

GPU join:

- build a hash table for orders, scan lineitem and select if (l.receiptdate > l.commitdate), store orderkey and orderpriority
Q4: GPU ALGORITHM

Part 2

select
    o_orderpriority, count(o_orderkey) as order_count,
from
    orders
where
    o_orderdate >= date '[DATE]'
    and o_orderdate < date '[DATE]' + interval '3' month
    and exists (select * from lineitem
        where l_orderkey = o_orderkey
        and l_commitdate < l_receiptdate)
group by o_orderpriority, order by o_orderpriority;

*Alternative approach is to eliminate duplicates “on-fly” during the join operation
Q4: GPU ALGORITHM

Part 3

GPU aggregate:

- aggregate by order priority using atomicAdd, counting only unique orderkeys

- output bins = 5

select o_orderpriority, count(o_orderkey) as order_count, from orders where o_orderdate >= date '[DATE]' and o_orderdate < date '[DATE]' + interval '3' month and exists (select * from lineitem where l_orderkey = o_orderkey and l_commitdate < l_receiptdate) group by o_orderpriority, order by o_orderpriority;
Q4: GPU PERF BREAKDOWN

Tesla V100: PCIe vs NVLINK

V100 (PCIe3)

- GPU join: 79%
- GPU sort: 20%
- GPU aggregate: 1%

V100 (3xNVLINK2)

- GPU join: 76%
- GPU sort: 22%
- GPU aggregate: 2%

All input tables in system memory
### Q4: JOIN BREAKDOWN

SF10 on PCIe and NVLINK

**orders:** 1.5M per SF, *selectivity* 3.8%

**lineitem:** 6M per SF, *selectivity* 63%

<table>
<thead>
<tr>
<th>GPU KERNEL</th>
<th>V100 (PCIE) TIME (MS)</th>
<th>V100 (PCIE) TIME (%)</th>
<th>V100 (NVLINK) TIME (MS)</th>
<th>V100 (NVLINK) TIME (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>build_hash_tbl</td>
<td>12.3</td>
<td>16.3%</td>
<td>1.7</td>
<td>14.0%</td>
</tr>
<tr>
<td>probe_hash_tbl</td>
<td>63.2</td>
<td>83.6%</td>
<td>10.8</td>
<td>85.9%</td>
</tr>
</tbody>
</table>
Q4: PROBE KERNEL ANALYSIS
SF10 on V100 (PCIe3)

and exists (select * from lineitem
  where l_orderkey = o_orderkey
  and l_commitdate < l_receiptdate)

  matching rate 2.4%

receiptdate (4B), commitdate (4B), and orderkey (4B) * 60M rows = 0.670GB

V100 (PCIe3) probe throughput: 0.670GB / 63.2ms = 10.6GB/s

V100 (3xNVLINK2) probe throughput: 0.670GB / 10.8ms = 62.0GB/s
**QUERY 4**

E2E results using 32-bit keys*

<table>
<thead>
<tr>
<th>TIME (MS)</th>
<th>SF1</th>
<th>SF10</th>
<th>SF100</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU (single-threaded)</td>
<td>150</td>
<td>2041</td>
<td>24960</td>
</tr>
<tr>
<td>V100 (PCIe3)</td>
<td>13</td>
<td>105</td>
<td>946</td>
</tr>
<tr>
<td>V100 (3xNVLink2)</td>
<td>7</td>
<td>23</td>
<td>308</td>
</tr>
</tbody>
</table>

*Assuming the input tables are loaded and pinned in system memory
### QUERY 4

64-bit keys required for SF > 500

<table>
<thead>
<tr>
<th></th>
<th>MINSKY (P8 + P100)</th>
<th>SF100: 4B+4B</th>
<th>SF100: 8B+4B</th>
<th>SF1000: 8B+4B</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU join</td>
<td>586</td>
<td>785</td>
<td>6838</td>
<td></td>
</tr>
<tr>
<td>GPU sort</td>
<td>176</td>
<td>271</td>
<td>2794</td>
<td></td>
</tr>
<tr>
<td>GPU aggregate</td>
<td>9</td>
<td>9</td>
<td>87</td>
<td></td>
</tr>
<tr>
<td>TOTAL time (ms)</td>
<td>772</td>
<td>1066</td>
<td>9721</td>
<td></td>
</tr>
</tbody>
</table>

- GPU join: 1.3x faster
- GPU sort: 1.5x faster
- GPU aggregate: 1.4x slower
TPC-H QUERY 21: MULTIPLE COMPLEX JOINS
TPC-H QUERY 21
The Suppliers Who Kept Orders Waiting

```sql
select s_name, count(*) as numwait
from supplier, lineitem l1, orders, nation
where
  o_orderkey = l1.l_orderkey and
  o_orderstatus = 'F' and
  s_suppkey = l1.l_suppkey and
  s_nationkey = n_nationkey and n_name = '[NATION]' and
  l1.l_receiptdate > l1.l_commitdate
  exists (select * from lineitem l2
  where l2.l_orderkey = l1.l_orderkey and
  l2.l_suppkey <> l1.l_suppkey) and
not exists (select * from lineitem l3
  where l3.l_orderkey = l1.l_orderkey and
  l3.l_suppkey <> l1.l_suppkey and
  l3.l_receiptdate > l3.l_commitdate)
  group by s_name order by numwait desc, s_name;
```
select s_name, count(*) as numwait 
from supplier, lineitem l1, orders, nation 
where
  o_orderkey = l1.l_orderkey and
  o_orderstatus = 'F' and
  s_suppkey = l1.l_suppkey and
  s_nationkey = n_nationkey and n_name = 'NATION' and
  l1.l_receiptdate > l1.l_commitdate

  exists (select * from lineitem l2
          where l2.l_orderkey = l1.l_orderkey and
          l2.l_suppkey <> l1.l_suppkey) and

  not exists (select * from lineitem l3
              where l3.l_orderkey = l1.l_orderkey and
              l3.l_suppkey <> l1.l_suppkey and
              l3.l_receiptdate > l3.l_commitdate)

group by s_name order by numwait desc, s_name;

TPC-H QUERY 21
The Suppliers Who Kept Orders Waiting

This order is a multi-supplier order
This supplier is the only one failed to meet the committed date
Group orders by supplier

Supplier#74  9
Q21: GPU ALGORITHM

Part 1

```sql
select s_name, count(*) as numwait
from supplier, lineitem l1, orders, nation
where
  o_orderkey = l1.l_orderkey and
  o_orderstatus = 'F' and
  s_suppkey = l1.l_suppkey and
  s_nationkey = n_nationkey and n_name = '[NATION]' and
  l1.l_receiptdate > l1.l_commitdate
exists (select * from lineitem l2
  where l2.l_orderkey = l1.l_orderkey and
    l2.l_suppkey <> l1.l_suppkey) and
not exists (select * from lineitem l3
  where l3.l_orderkey = l1.l_orderkey and
    l3.l_suppkey <> l1.l_suppkey and
    l3.l_receiptdate > l3.l_commitdate)

group by s_name order by numwait desc, s_name;
```

GPU join1:
- build hash tables for orders (status = ‘F’) and supplier (s_nationkey = [nation])
- scan lineitem and select if (l1.receiptdate > l1.commitdate), store resulting orderkey-supplier pairs (decoupled)
Q21: GPU ALGORITHM

Part 2

select s_name, count(*) as numwait
from supplier, lineitem l1, orders, nation
where
  o_orderkey = l1.l_orderkey and
  o_orderstatus = 'F' and
  s_suppkey = l1.l_suppkey and
  s_nationkey = n_nationkey and n_name = '[NATION]' and
  l1.l_receiptdate > l1.l_commitdate
exists (select * from lineitem l2
  where l2.l_orderkey = l1.l_orderkey and
  l2.l_suppkey <> l1.l_suppkey) and
not exists (select * from lineitem l3
  where l3.l_orderkey = l1.l_orderkey and
  l3.l_suppkey <> l1.l_suppkey and
  l3.l_receiptdate > l3.l_commitdate)
group by s_name order by numwait desc, s_name;

GPU sort + scan:

- sort by orderkey and find supplier ranges for each unique orderkey

GPU join2:

- build a hash table from unique orderkeys and store supplier range as payload
- scan lineitem and probe orderkey against HT, for each hit scan the corresponding supp range:
  - test if (l2.l_suppkey <> l1.l_suppkey) then use atomicCAS for semi-join and atomicExch for anti-join
Q21: GPU ALGORITHM

Part 3

```sql
select s_name, count(*) as numwait
from supplier, lineitem l1, orders, nation
where
  o_orderkey = l1.l_orderkey and
  o_orderstatus = 'F' and
  s_suppkey = l1.l_suppkey and
  s_nationkey = n_nationkey and n_name = '[NATION]' and
  l1.l_receiptdate > l1.l_commitdate
  exists (select * from lineitem l2
          where l2.l_orderkey = l1.l_orderkey and
                 l2.l_suppkey <> l1.l_suppkey)
  and
  not exists (select * from lineitem l3
              where l3.l_orderkey = l1.l_orderkey and
                     l3.l_suppkey <> l1.l_suppkey and
                     l3.l_receiptdate > l3.l_commitdate)
  group by s_name order by numwait desc, s_name;
```

**GPU aggregate:**

- aggregate the mask produced by join2 by supplier ID and store in the global array of supplier size
- compact to extract only non-zero entries, then sort
- # of output rows limited to 100
Q21: GPU MEMORY USAGE

32-bit orderkeys

Input tables (per SF): \texttt{lineitem} 6M, \texttt{orders} 1.5M, \texttt{suppliers} 10K

\textit{order sel} 50\%, assuming 50\% HT occupancy and 4B+4B we need 11.4MB per SF

<table>
<thead>
<tr>
<th>HT OCCUPANCY</th>
<th>JOIN1 MEM (MB)</th>
<th>JOIN1 TIME (MS)</th>
<th>TOTAL TIME (MS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>114</td>
<td>101.917</td>
<td>162.15</td>
</tr>
<tr>
<td>60</td>
<td>96</td>
<td>99.351</td>
<td>160.43</td>
</tr>
<tr>
<td>70</td>
<td>82</td>
<td>99.747</td>
<td>161.31</td>
</tr>
<tr>
<td>80</td>
<td>72</td>
<td>101.888</td>
<td>164.23</td>
</tr>
<tr>
<td>90</td>
<td>64</td>
<td>152.503</td>
<td>211.18</td>
</tr>
</tbody>
</table>

SF=10

Using open addressing hash table with linear probing
SF > 500 will have more than 4 billion rows and require 64-bit keys.

This will use \( \frac{8+4}{4+4} = 1.5x \) more memory on the GPU.

With the same 80% HT occupancy we would need **10.8MB** per SF.

In theory a single Tesla V100 16GB is sufficient for **SF1500**.
Q21: GPU PERF BREAKDOWN

Tesla V100: PCIe vs NVLINK

V100 (PCIe3)
- GPU join 1: 62%
- GPU sort+scan: 12%
- GPU join 2: 25%
- GPU aggregate: 1%

V100 (3xNVLINK2)
- GPU join 1: 65%
- GPU sort+scan: 19%
- GPU join 2: 12%
- GPU aggregate: 4%

All input tables in system memory
Q21: JOINS BREAKDOWN
SF10 on V100 (PCIe3)

<table>
<thead>
<tr>
<th>GPU KERNEL</th>
<th>GPU JOIN1</th>
<th></th>
<th>GPU JOIN2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TIME (MS)</td>
<td>TIME (%)</td>
<td>TIME (MS)</td>
<td>TIME (%)</td>
</tr>
<tr>
<td>build_hash_tbl</td>
<td>6.6</td>
<td>7%</td>
<td>0.6</td>
<td>1%</td>
</tr>
<tr>
<td>probe_hash_tbl</td>
<td>77.0</td>
<td>76%</td>
<td>38.4</td>
<td>95%</td>
</tr>
</tbody>
</table>
Q21: PROBE KERNEL ANALYSIS

Load `receiptdate` (4B), `commitdate` (4B), and `orderkey` (4B or 8B)

If `orderkey` hits we're probing the suppliers (4B) - *filtered column*

Expected sysmem reads: from 0.670GB (no suppliers) to 0.894GB (full suppliers)

nvprof reports **0.842GB** (sysmem reads) / **77ms** (kernel time) = **10.9GB/s**
QUERY 21
E2E results using 32-bit keys*

<table>
<thead>
<tr>
<th>TIME (MS)</th>
<th>SF1</th>
<th>SF10</th>
<th>SF100</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU (single-threaded)</td>
<td>1329</td>
<td>31731</td>
<td>465064</td>
</tr>
<tr>
<td>V100 (PCle3)</td>
<td>22</td>
<td>164</td>
<td>1521</td>
</tr>
<tr>
<td>V100 (3xNVLINK2)</td>
<td>12</td>
<td>45</td>
<td>466</td>
</tr>
</tbody>
</table>

*Assuming the input tables are loaded and pinned in system memory
TAKEAWAYS

GPU memory capacity is not a limiting factor

GPU query performance up to 2-3 orders of magnitude better than CPU

GPU query perf is dominated by the CPU-GPU interconnect throughput

NVLINK systems show 3x better E2E query performance compared to PCIe

S8417 - Breaking the Speed of Interconnect with Compression for Database Applications - Tuesday, Mar 27, 2:00pm - Room 210F