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
- Examples: DATE = 1/1/1993, 1/4/1993, …

SQL

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

A TYPICAL ANALYTICS QUERY

From SQL to Database Operators

Database Operators

aggregate

predicate (filter)

join
predicate (filter)

aggregate

sort

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

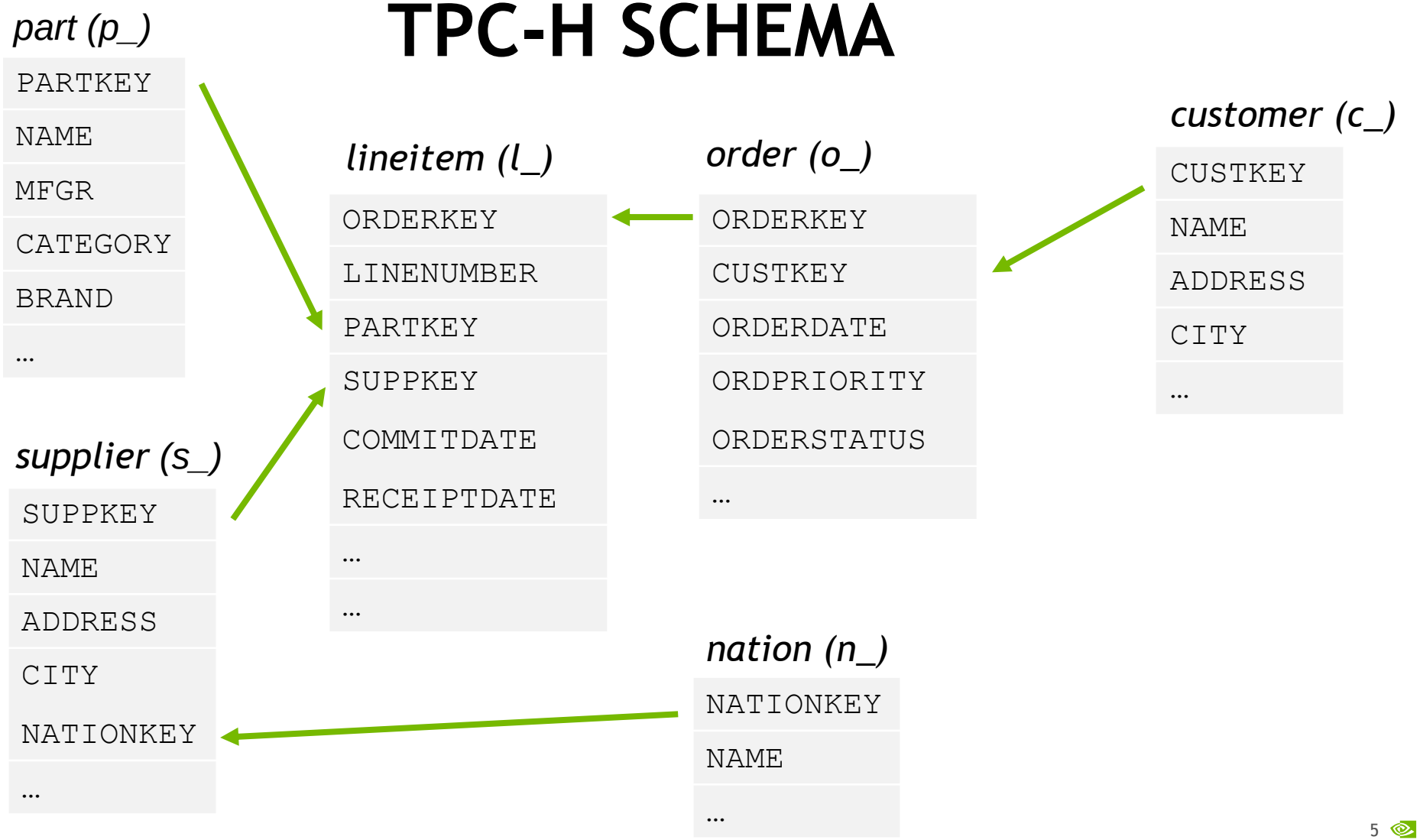
predicate (filter)

join
predicate (filter)

aggregate

sort

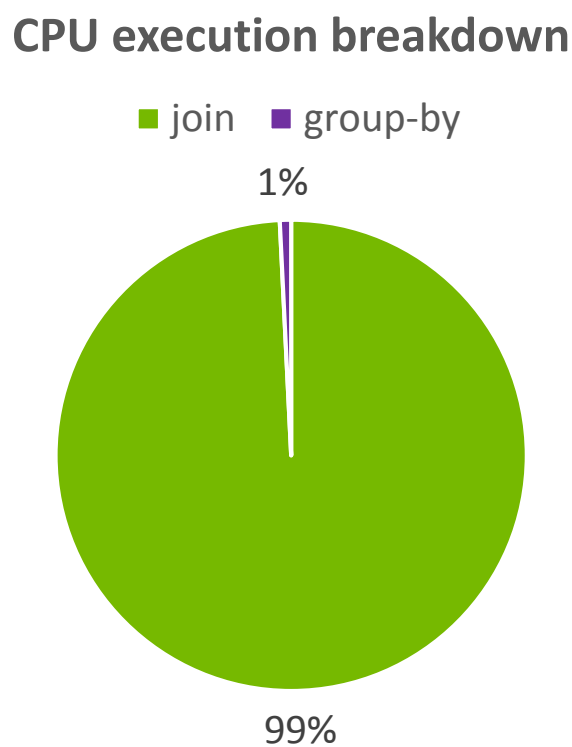
TPC-H SCHEMA


part (p_)
PARTKEY
NAME
MFGR
CATEGORY
BRAND
…
lineitem (l_)
ORDERKEY
LINENUMBER
PARTKEY
SUPPKEY
COMMITDATE
RECEIPTDATE
…
…
order (o_)
ORDERKEY
CUSTKEY
ORDERDATE
ORDPRIORITY
ORDERSTATUS
…
customer (c_)
CUSTKEY
NAME
ADDRESS
CITY
…
supplier (s_)
SUPPKEY
NAME
ADDRESS
CITY
NATIONKEY
…
nation (n_)
NATIONKEY
NAME
…

WHERE DOES TIME GO?

TPC-H query 4 @SF10 = 10GB data warehouse

```
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;
```



18/22 TPC-H Queries involve **Join** & are the longest running ones

RELATIONAL JOIN

Lineitem[1]

l_orderkey
23
14
56
11
39
27
23

Foreign Key

⋈

Order[2]

o_orderkey	o_orderpriority
11	1
23	5
27	2
29	4

Primary Key Payload

=

Join Results

o_orderkey	o_orderpriority
23	5
11	1
27	2
23	5

[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

Lineitem[1]

l_orderkey
23
14
56
11
39
27
23

Foreign Key
= Probe inputs

⋈

Order[2]

o_orderkey	o_orderpriority
11	1
23	5
27	2
29	4

Primary Key Payload

Build hash table

=

Join Results

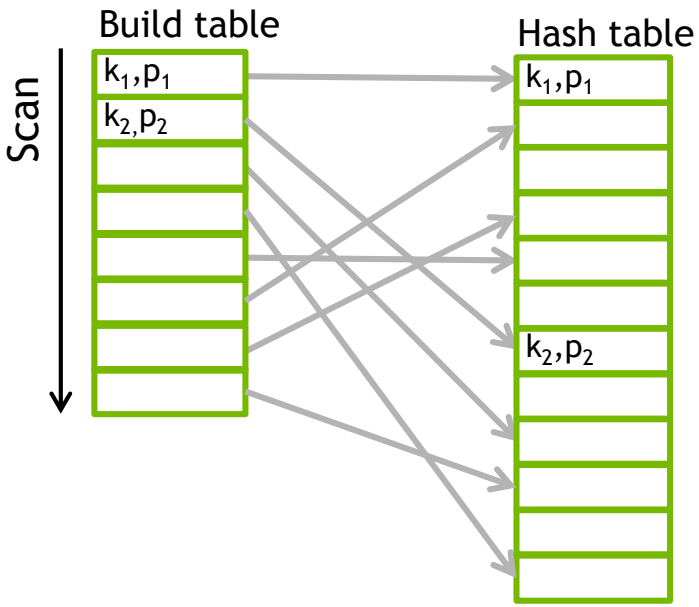
o_orderkey	o_orderpriority
23	5
11	1
27	2
23	5

[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

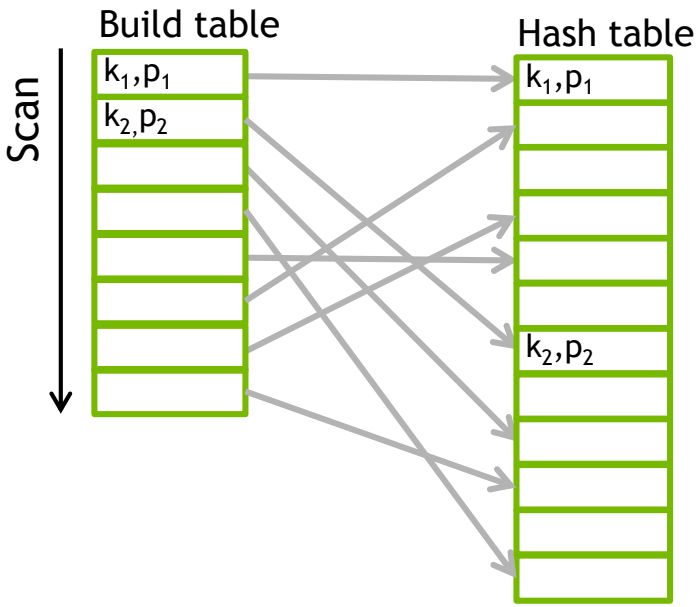


1. Compute h(k) – k is a primary (unique) key
2. Insert k with payload p into hashed location
3. If occupied, insert into next free one

HASH JOIN

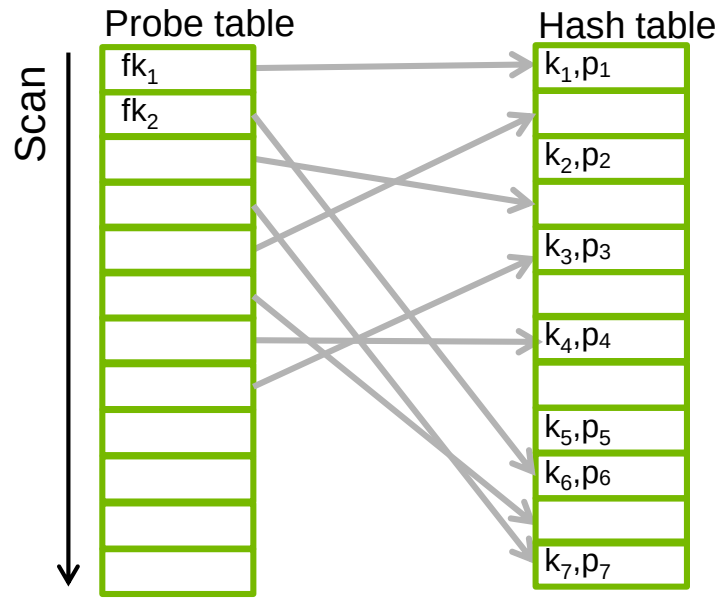
General approach – including payload(s)

1. Build a hash table



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

2. Probe the hash table

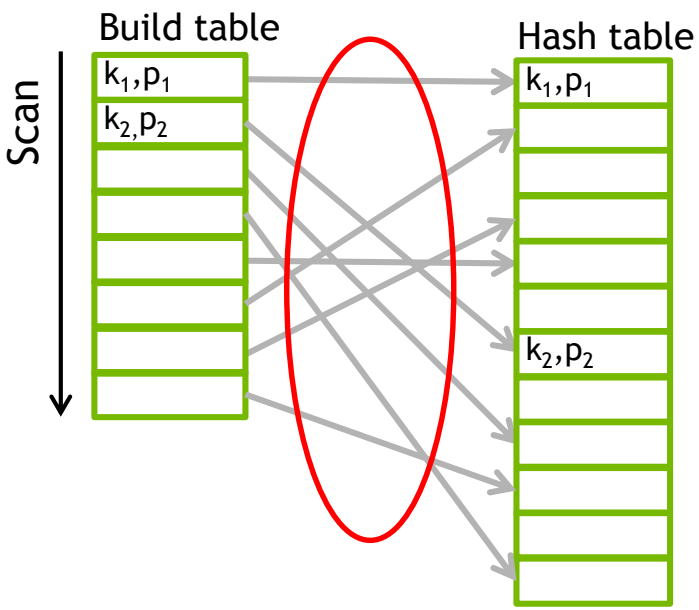


1. Compute h(fk) – fk is foreign key
2. Compare fk and key k in the hash table
3. If there is a match store the result

HASH JOIN

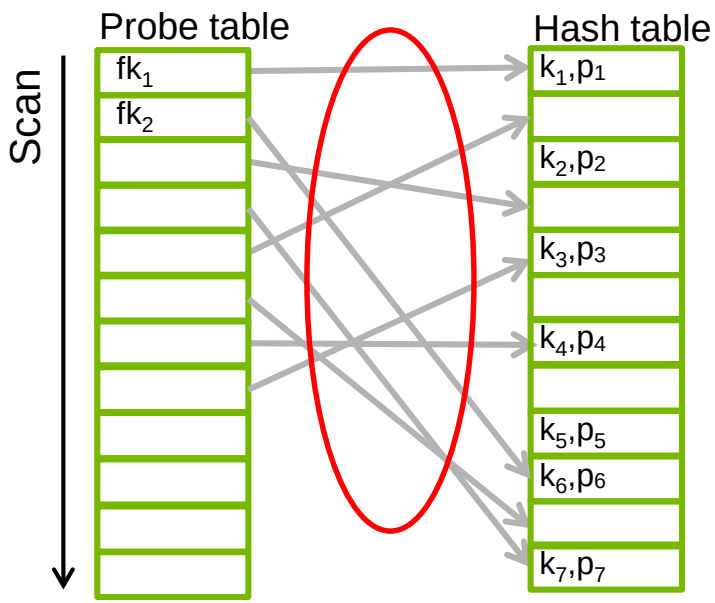
General approach – including payload(s)

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

2. Probe the hash table



1. Compute h(fk) – fk is foreign key
2. Compare fk and key k in the hash table
3. If there is a match store the result

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**

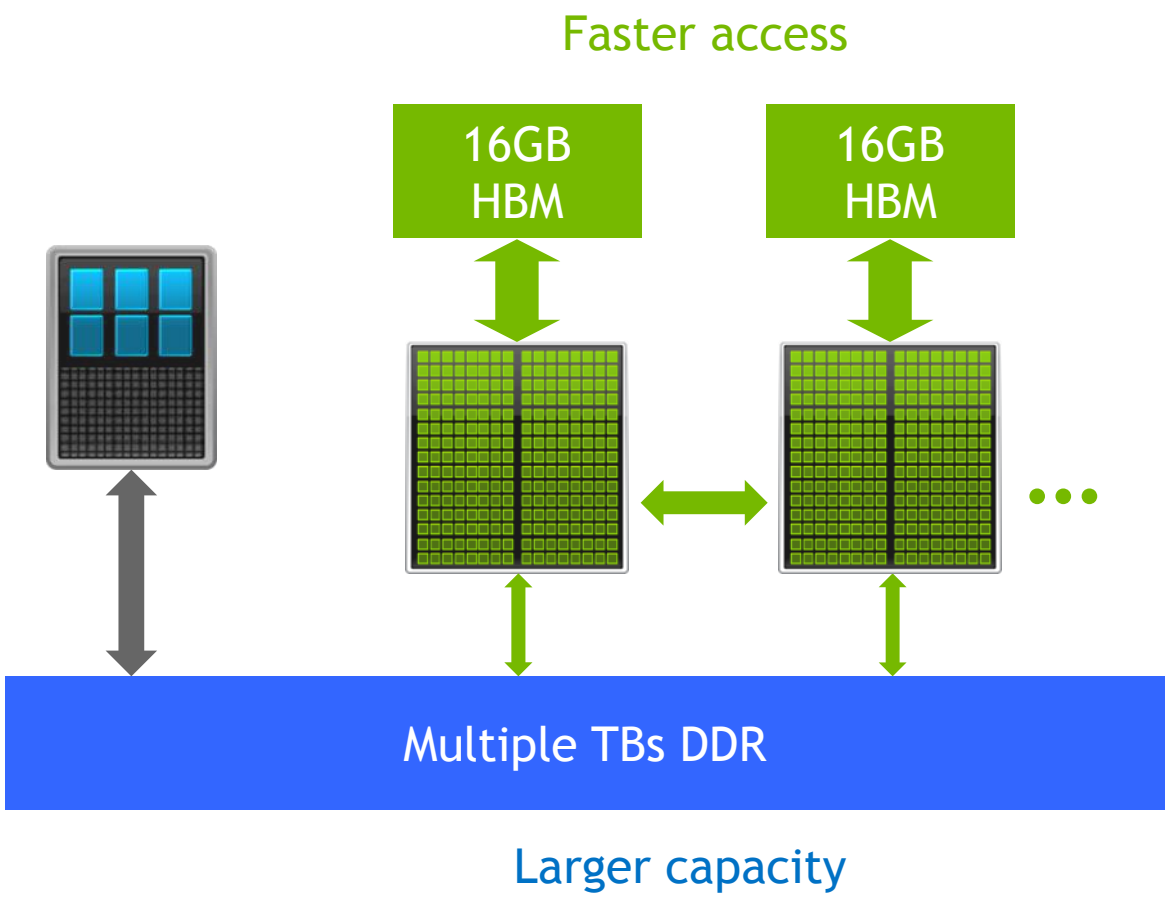
NVIDIA Tesla V100 (Volta)

Peak memory bandwidth **900GB/s**

Random 8B access **60GB/s**

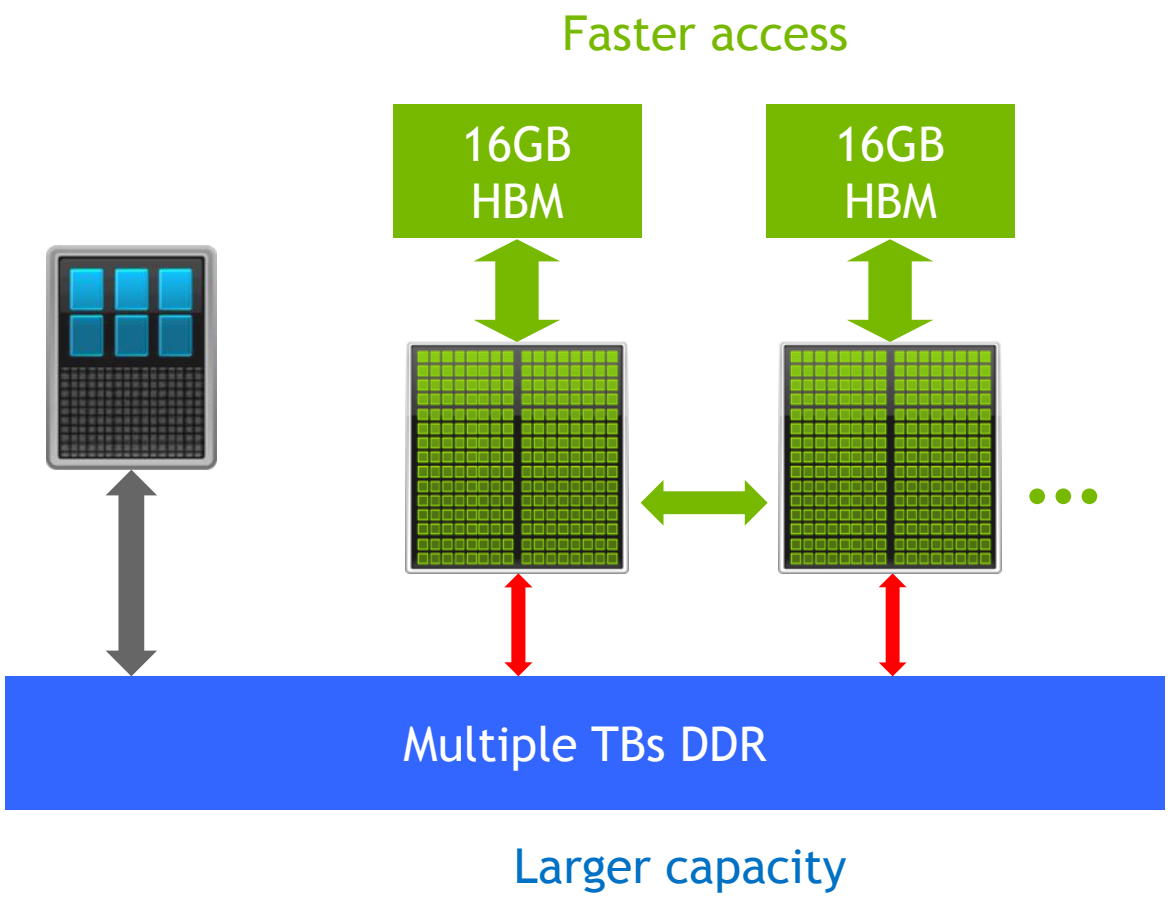
**DDR4 specs: https://www.micron.com/~/media/documents/products/data-sheet/dram/ddr4/4gb_ddr4_sdram.pdf*

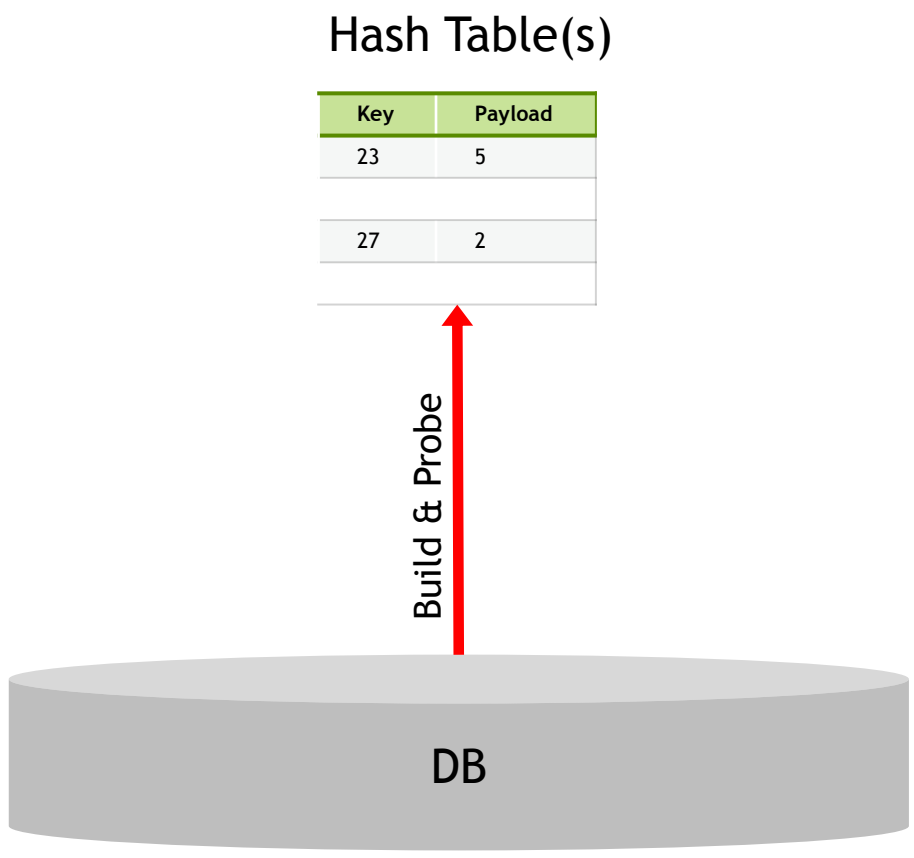
HETEROGENEOUS SYSTEM



HETEROGENEOUS SYSTEM

Data Placement & Movement





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

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

```
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 ); };
```

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`

```
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);
}
```

Key	Value
unused_key	unused_element
unused_key	unused_element
unused_key	unused_element
7287984	9024
x.first=75	x.second=875
283408	2309480
unused_key	unused_element
unused_key	unused_element

IMPLEMENTING HASH JOIN

Build Optimization: merge 4B keys with 4B values

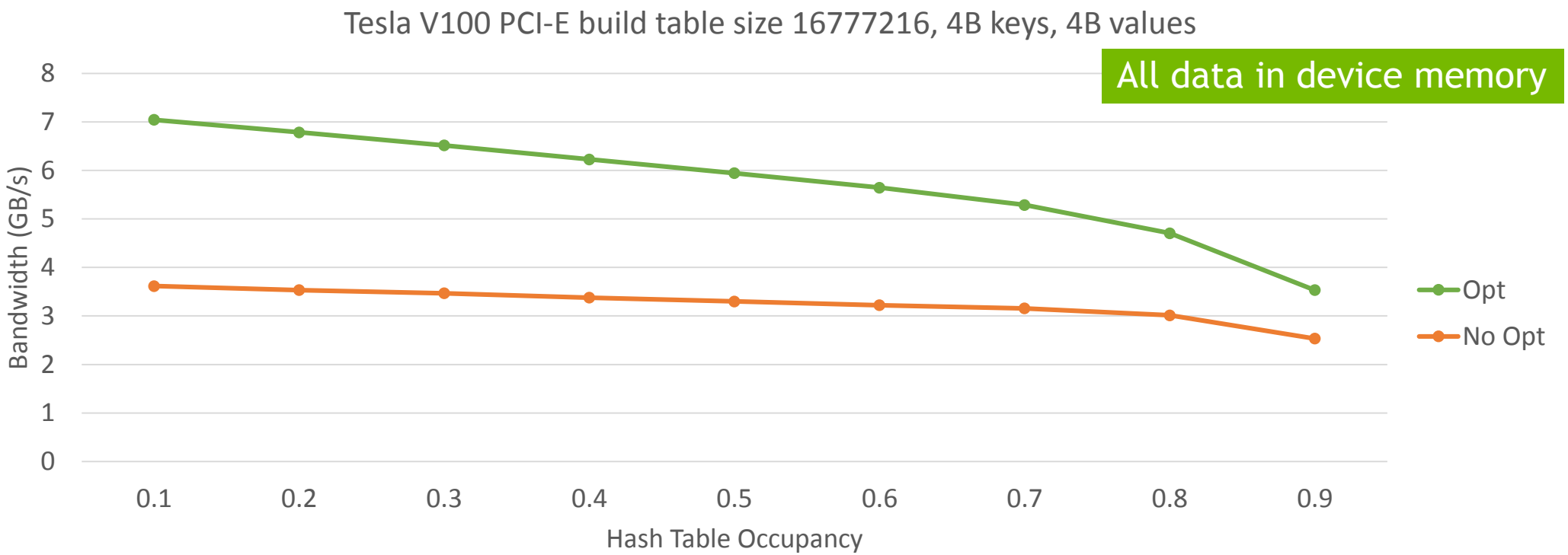
```
iterator insert(const value_type& x) {
  size_type key_hash = hf( x.first );
  size_type ht_idx = key_hash%ht_size;
  value_type* it = 0;
  while (0 == it) {
    value_type* tmp_it = ht_values + ht_idx;
    const key_type old_key = atomicCAS(
           &(tmp_it->first), unused_key, x.first );
 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

```
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
```

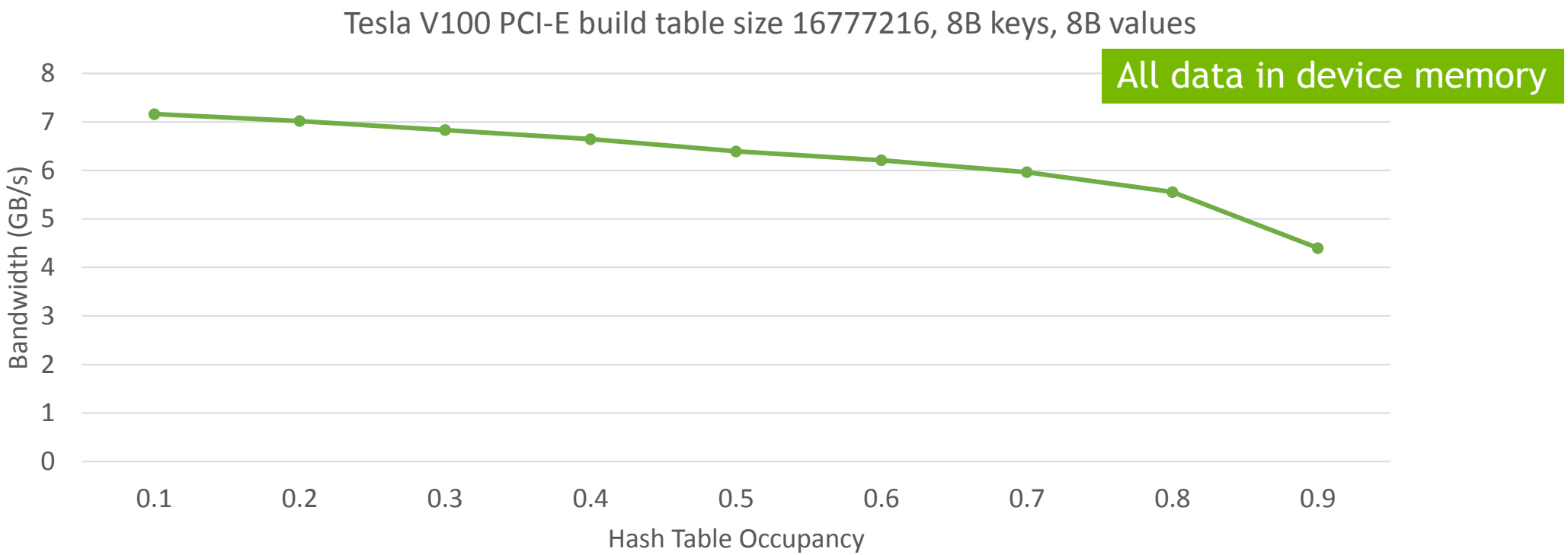
IMPLEMENTING HASH JOIN

Build Optimization: merge 4B keys with 4B values



IMPLEMENTING HASH JOIN

Build Performance 8B keys with 8B values



IMPLEMENTING HASH JOIN

2. Probe the hash table with a `concurrent_unordered_map`

```
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)

```
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();
```


Kaldewey, Tim, et al. "GPU join processing revisited." *Proceedings of the Eighth International Workshop on Data Management on New Hardware*. ACM, 2012.

IMPLEMENTING HASH JOIN

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

```
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();
```


Kaldewey, Tim, et al. "GPU join processing revisited." *Proceedings of the Eighth International Workshop on Data Management on New Hardware*. ACM, 2012.

IMPLEMENTING HASH JOIN

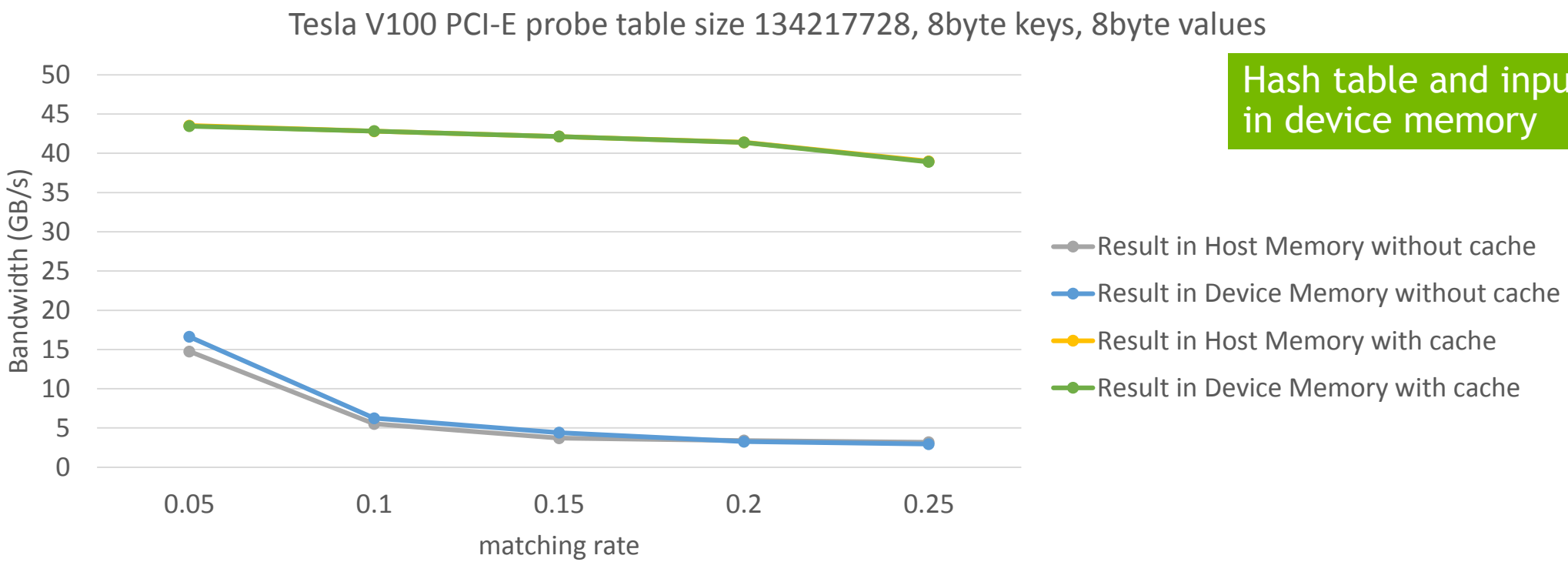
Probe Optimization: output cache for result materialization III (flush)

```
    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];
    }
}
```


Kaldewey, Tim, et al. "GPU join processing revisited." *Proceedings of the Eighth International Workshop on Data Management on New Hardware*. ACM, 2012.

IMPLEMENTING HASH JOIN

Probe Optimization: output cache for result materialization



Hash table and input in device memory

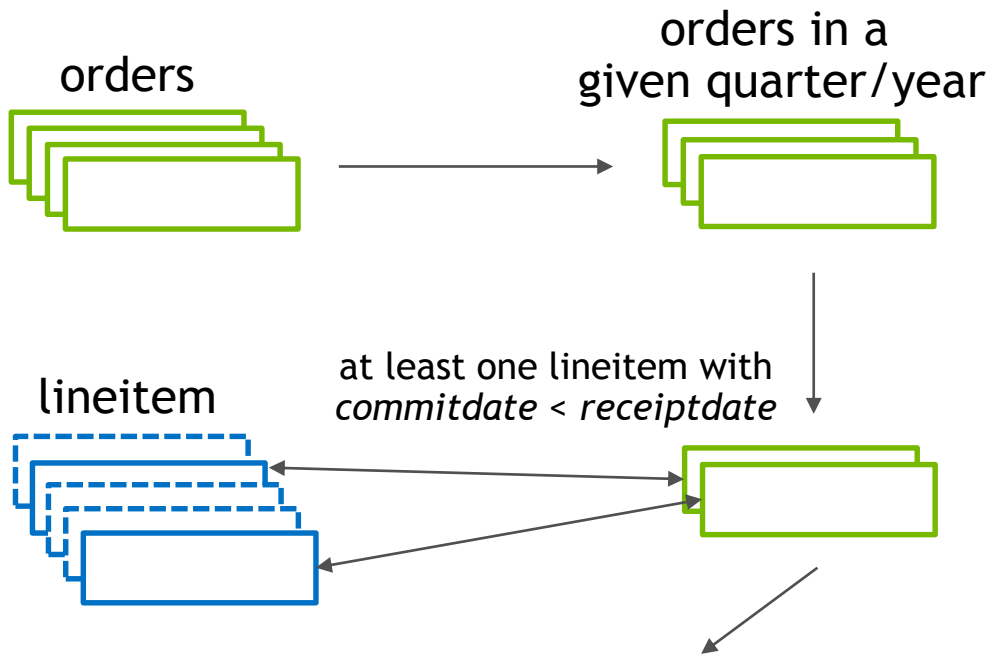
Kaldewey, Tim, et al. "GPU join processing revisited." *Proceedings of the Eighth International Workshop on Data Management on New Hardware*. ACM, 2012.

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;
```



1-URGENT	93
2-HIGH	103
3-MEDIUM	109

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)
group by o_orderpriority, order by o_orderpriority;
```

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;
```

GPU sort*:

- sort (orderkey, orderpriority) by orderkey to remove duplicates

Alternative approach is to eliminate duplicates "on-fly" during the join operation

Q4: GPU ALGORITHM

Part 3

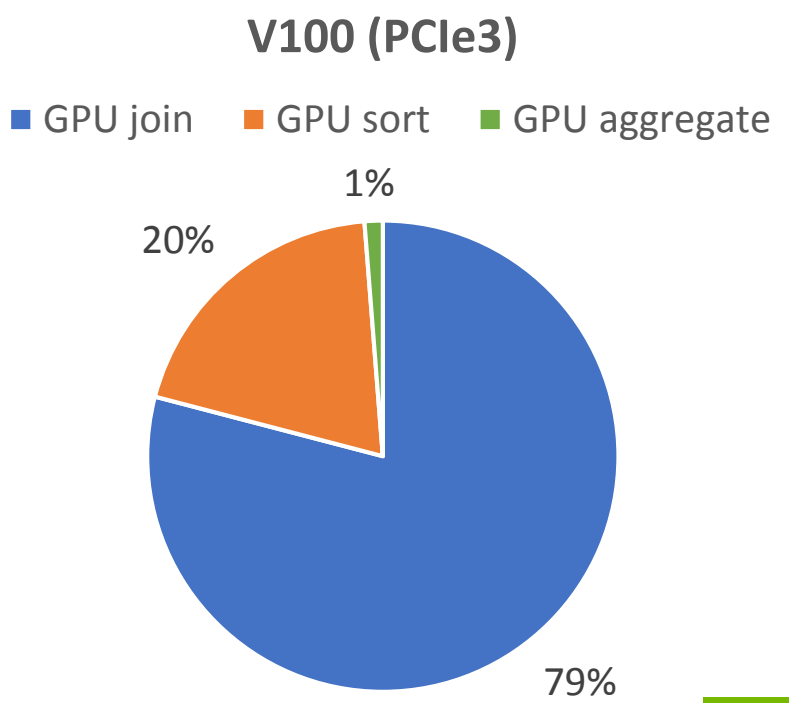
```
select
  o_orderpriority, count(o_orderkey) as order_count,
from
  orders
where

  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;
```

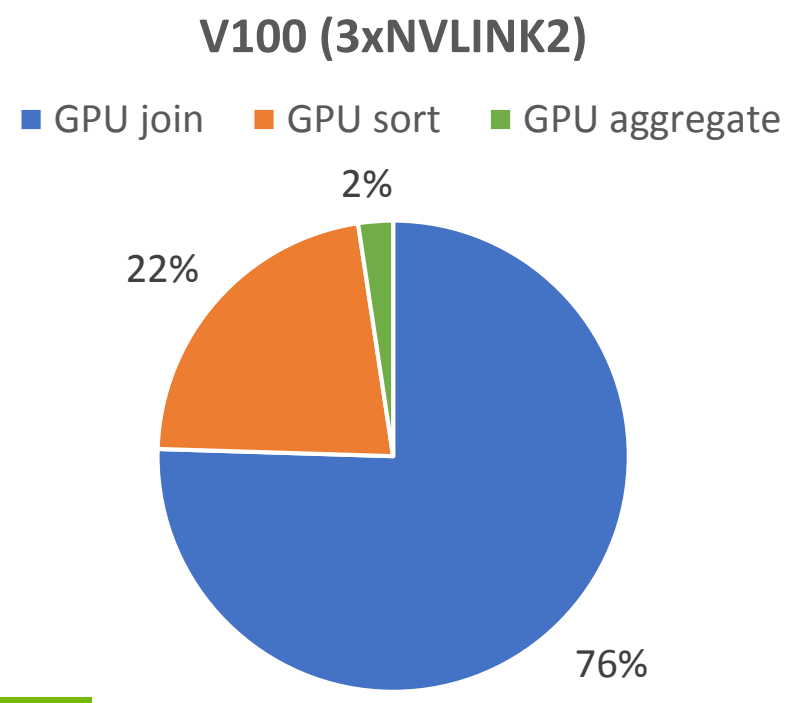
GPU aggregate:

- aggregate by order priority using atomicAdd, counting only unique orderkeys
- *output bins = 5*

Q4: GPU PERF BREAKDOWN

Tesla V100: PCIe vs NVLINK





All input tables in system memory

Q4: JOIN BREAKDOWN

SF10 on PCIe and NVLINK

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

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

GPU KERNEL	V100 (PCIE) TIME (MS)	V100 (PCIE) TIME (%)	V100 (NVLINK) TIME (MS)	V100 (NVLINK) TIME (%)
build_hash_tbl	12.3	16.3%	1.7	14.0%
probe_hash_tbl	63.2	83.6%	10.8	85.9%

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*

TIME (MS)	SF1	SF10	SF100
CPU (single-threaded)	150	2041	24960
V100 (PCIe3)	13	105	946
V100 (3xNVLINK2)	7	23	308

↓ 26x

↓ 3.1x

Assuming the input tables are loaded and pinned in system memory

QUERY 4

64-bit keys required for SF > 500

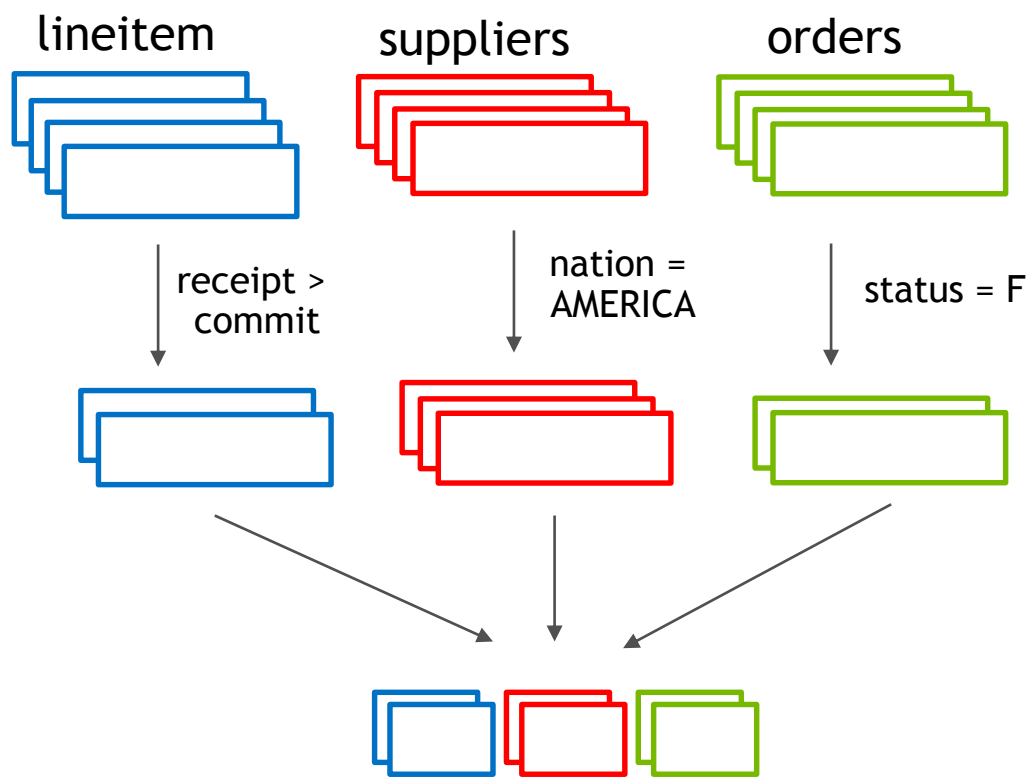
MINSKY (P8 + P100)	SF100: 4B+4B		SF100: 8B+4B	SF1000: 8B+4B
GPU join	586	1.3x →	785	6838
GPU sort	176	1.5x →	271	2794
GPU aggregate	9		9	87
TOTAL time (ms)	772	1.4x →	1066	9721

TPC-H QUERY 21: MULTIPLE COMPLEX JOINS

TPC-H QUERY 21

The Suppliers Who Kept Orders Waiting

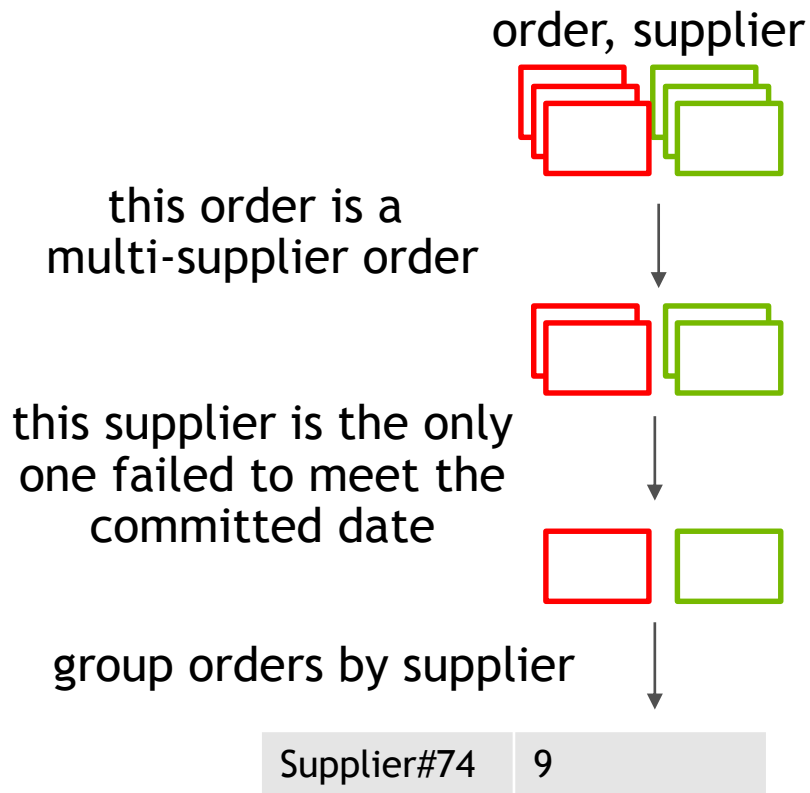
```
select s_name, count(*) as numwait
from supplier,  lineitem l1,  orders,  nation
where

  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

```
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;
```



Q21: GPU ALGORITHM

Part 1

```
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

```
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): **lineitem** 6M, **orders** 1.5M, **suppliers** 10K

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

SF=10

HT OCCUPANCY	JOIN1 MEM (MB)	JOIN1 TIME (MS)	TOTAL TIME (MS)
50	114	101.917	162.15
60	96	99.351	160.43
70	82	99.747	161.31
80	**72**	**101.888**	**164.23**
90	64	152.503	211.18

Using open addressing hash table with linear probing

Q21: GPU MEMORY USAGE

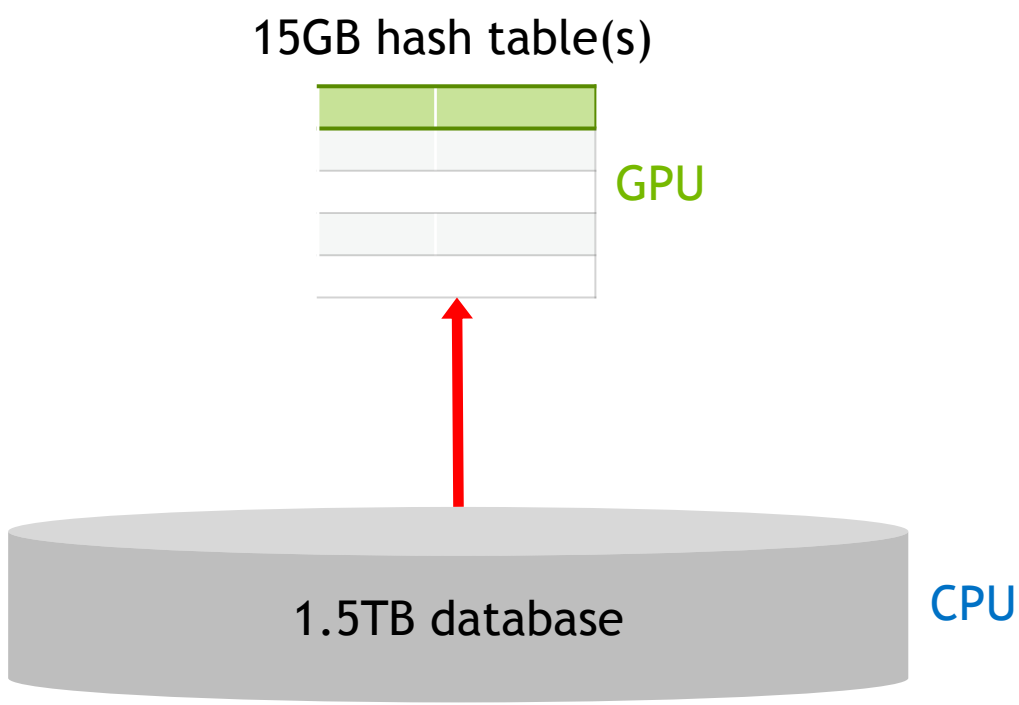
Fitting the largest SF

SF > 500 will have more than 4 billion rows and require **64-bit keys**

This will use (8+4) / (4+4) = 1.5x more memory on the GPU

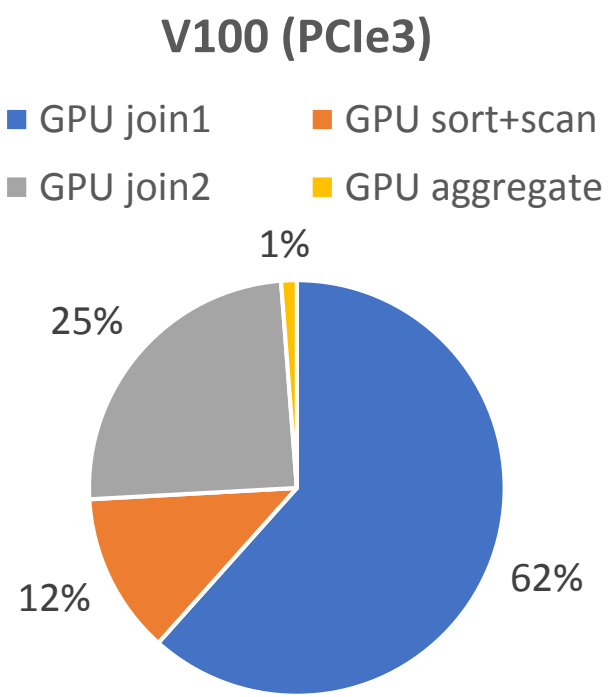
With the same 80% HT occupancy we would need **10.8MB** per SF

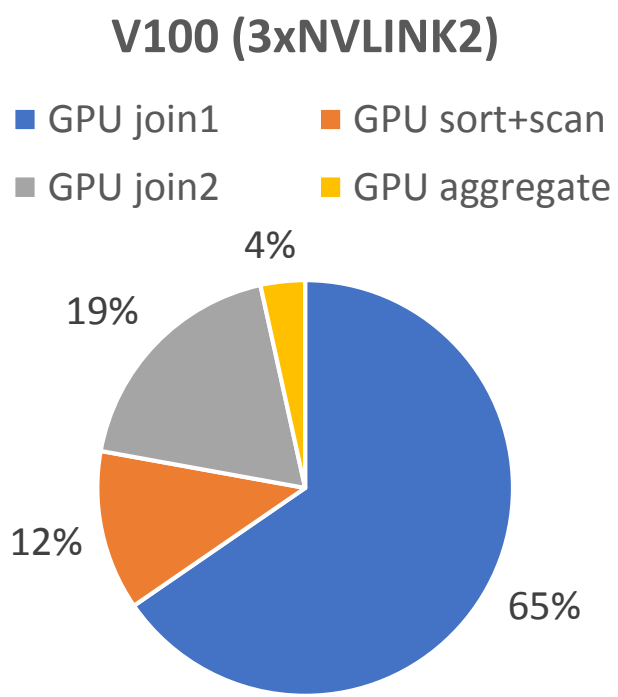
In theory a single Tesla V100 16GB is sufficient for **SF1500**



Q21: GPU PERF BREAKDOWN

Tesla V100: PCIe vs NVLINK





All input tables in system memory

Q21: JOINS BREAKDOWN

SF10 on V100 (PCIe3)

GPU KERNEL	GPU JOIN1		GPU JOIN2	
	TIME (MS)	TIME (%)	TIME (MS)	TIME (%)
build_hash_tbl	6.6	7%	0.6	1%
probe_hash_tbl	77.0	76%	38.4	95%

Q21: PROBE KERNEL ANALYSIS

```
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
```

matching rate 1.2%

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*

TIME (MS)	SF1	SF10	SF100
CPU (single-threaded)	1329	31731	465064
V100 (PCIe3)	22	164	1521
V100 (3xNVLINK2)	12	45	466

300x

3.2x

Assuming the input tables are loaded and pinned in system memory

TAKEAWAYS

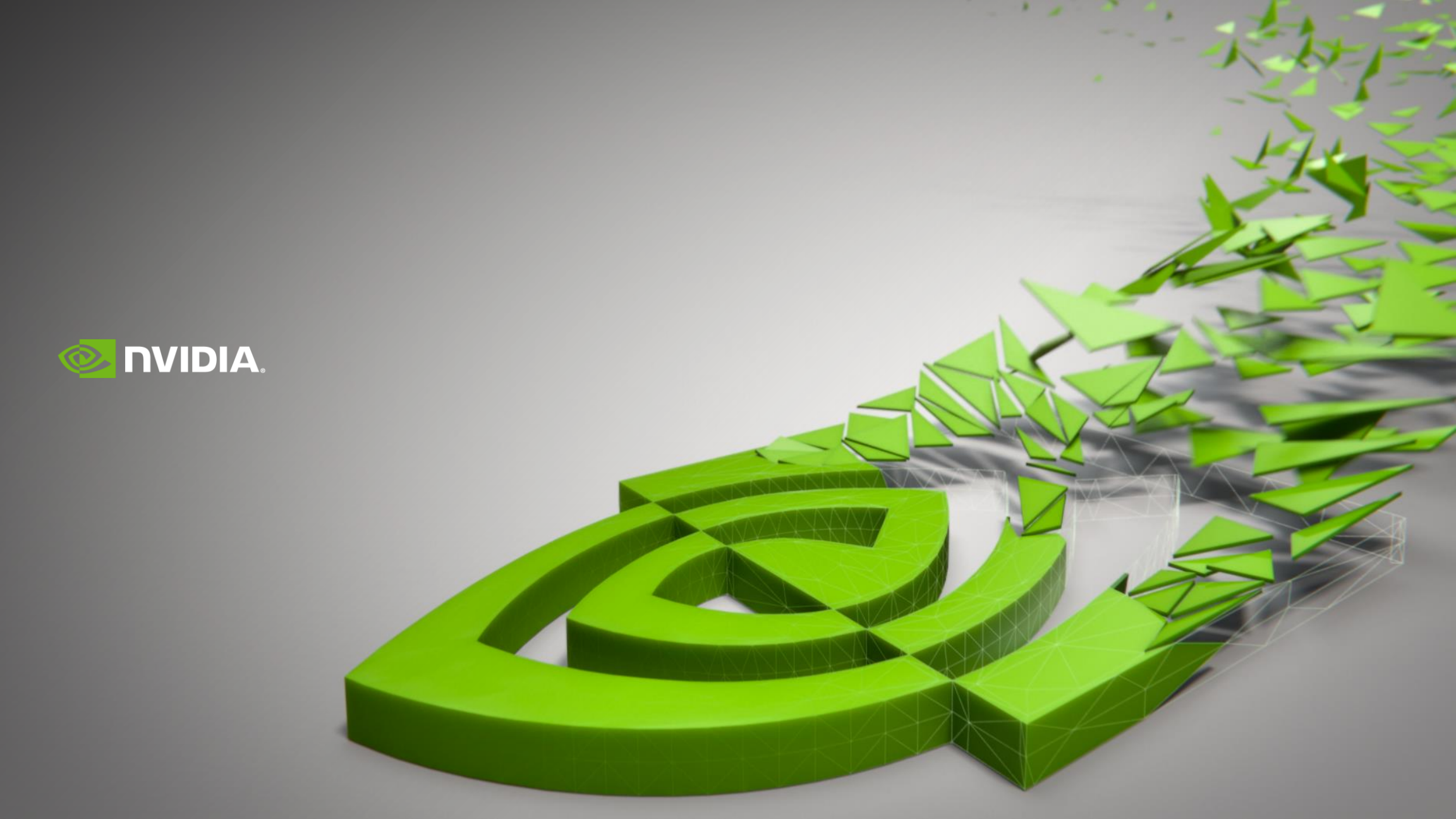
GPU memory capacity is not a limiting factor

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**


NVIDIA