Tim Kaldewey, Rene Mueller

Mar 19 2013

Let your GPU do the heavy lifting in your data Warehouse

Agenda

- A closer look at data warehousing queries
  - From queries down to operators
  - Where does time go?
  - Hash Join operators
  - Data Access Patterns

- Drill-down: Hash Tables on GPUs
  - Hash computation
  - Hash Tables = Hash computation + Memory access
  - Optimizations

- From Hash Tables to Relational Joins
  - Hash Join Implementation
  - Query Performance
  - Processing 100s of GBs in seconds

A data warehousing query in multiple languages

- **English**: Show me the annual development of revenue from US sales of US products for the last 5 years by city

A data warehousing query in multiple languages

- **English**: Show me the **annual** development of **revenue** from **US sales** of **US products** for the last **5 years** by **city**
- **SQL**:

```
SELECT c.city, s.city, d.year, SUM(lo.revenue)
  FROM lineorder lo, customer c, supplier s, date d
  WHERE lo.custkey = c.custkey
    AND lo.suppkey = s.suppkey
    AND lo.orderdate = d.datekey
    AND c.nation = 'UNITED STATES'
    AND s.nation = 'UNITED STATES'
    AND d.year >= 1998 AND d.year <= 2012
  GROUP BY c.city, s.city, d.year
  ORDER BY d.year asc, revenue desc;
```

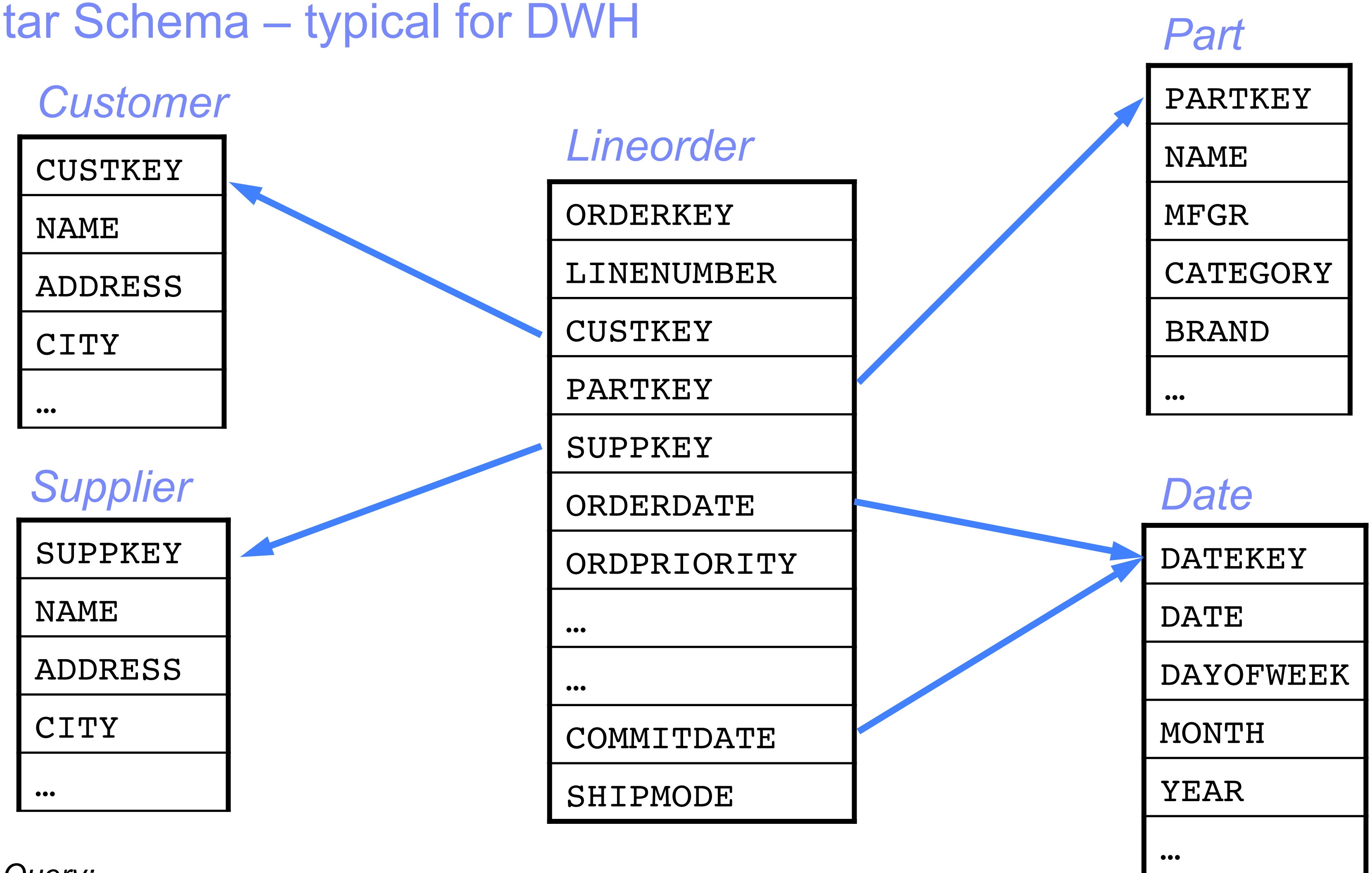
A data warehousing query in multiple languages

- **English**: Show me the annual development of revenue from US sales of US products for the last 5 years by city
- **SQL**:

```
SELECT c.city, s.city, d.year, SUM(lo.revenue)
  FROM lineorder lo, customer c, supplier s, date d
  WHERE lo.custkey = c.custkey
    AND lo.suppkey = s.suppkey
    AND lo.orderdate = d.datekey
    AND c.nation = 'UNITED STATES'
    AND s.nation = 'UNITED STATES'
    AND d.year >= 1998 AND d.year <= 2012
  GROUP BY c.city, s.city, d.year
  ORDER BY d.year asc, revenue desc;
```

?

Star Schema – typical for DWH



Query:
SELECT c.city, s.city, d.year, SUM(lo.revenue) **FROM** lineorder lo, customer c, supplier s, date d
WHERE **lo.custkey = c.custkey** AND **lo.suppkey = s.suppkey** AND **lo.orderdate = d.datekey** AND
c.nation = 'UNITED STATES' AND s.nation = 'UNITED STATES' AND d.year >= 1998 AND d.year <= 2012
GROUP BY c.city, s.city, d.year **ORDER BY** d.year asc, revenue desc;

A data warehousing query in multiple languages

- **English**: Show me the annual development of revenue from US sales of US products for the last 5 years by city
- **SQL**:

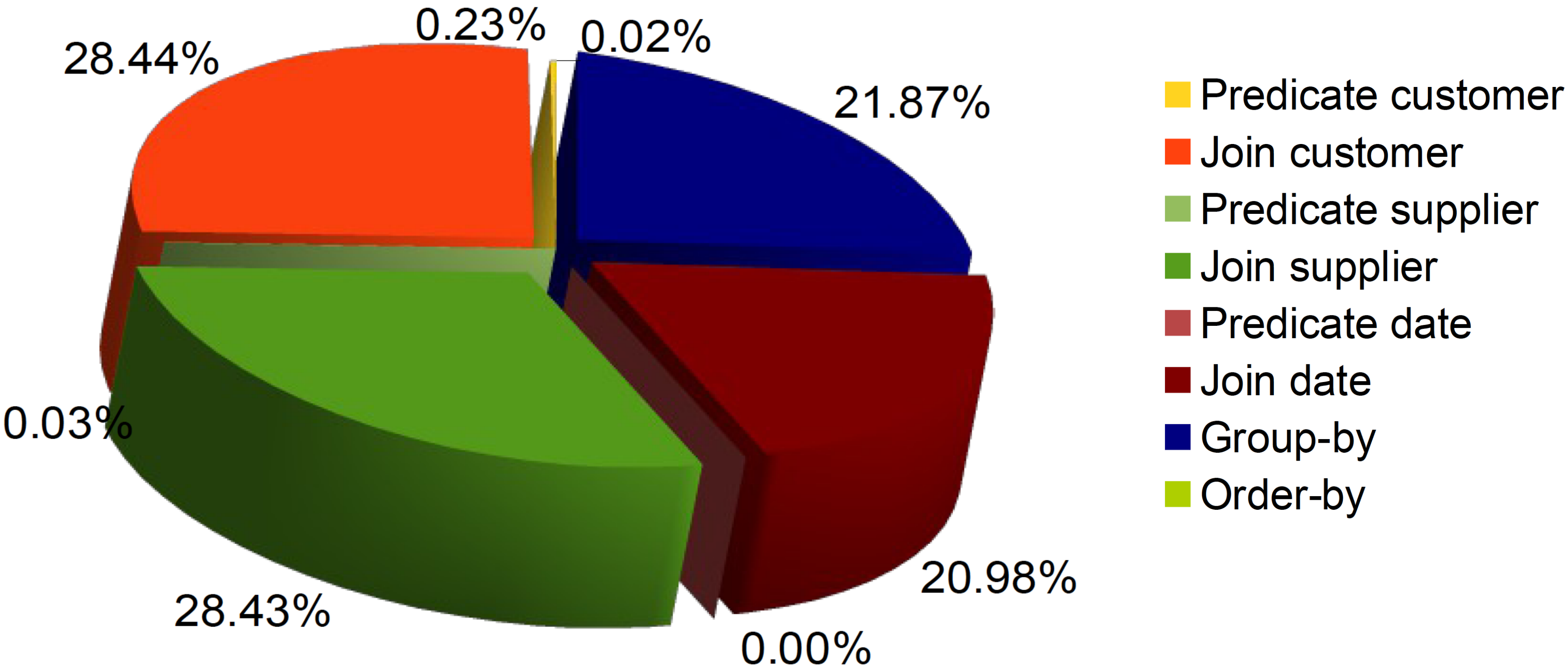
```
SELECT c.city, s.city, d.year, SUM(lo.revenue)
   FROM lineorder lo, customer c, supplier s, date d
   WHERE lo.custkey = c.custkey
      AND lo.suppkey = s.suppkey
      AND lo.orderdate = d.datekey
      AND c.nation = 'UNITED STATES'
      AND s.nation = 'UNITED STATES'
      AND d.year >= 1998 AND d.year <= 2012
  GROUP BY c.city, s.city, d.year
  ORDER BY d.year asc, revenue desc;
```

Database primitives (operators):

– Predicate(s): customer, supplier, and date	*direct filter (yes/no)*
– *Join(s)*: lineorder with part, supplier, and date	*correlate tables & filter*
– Group By (aggregate): city and date	*correlate tables & sum*
– Order By: year and revenue	*sort*

What are the most time-consuming operations?

Where does time go?




```
SELECT c.city, s.city, d.year, SUM(lo.revenue)
   FROM lineorder lo, customer c, supplier s, date d
   WHERE c.nation = 'UNITED STATES'          AND lo.custkey = c.custkey
     AND s.nation = 'UNITED STATES'          AND lo.suppkey = s.suppkey
     AND d.year >= 1998 AND d.year <= 2012 AND lo.orderdate = d.datekey
   GROUP BY c.city, s.city, d.year
   ORDER BY d.year asc, revenue desc;
```

Relational Joins

Sales (Fact Table)

Revenue	Customer
$10.99	23
$49.00	14
$11.00	56
$103.00	11
$84.50	39
$60.10	27
$7.60	23

Measure (Revenue) · Foreign Key (Customer)

⋈

Customers (living in US)

Key	Zip
11	95014
23	94303
27	95040
39	95134

Primary Key (Key) · Payload (Zip)

=

Revenue	Zip
$10.99	94303
$103.00	95014
$84.50	95134
$60.10	95040
$7.60	94303

Join Results

Hash Join

Sales (Fact Table)

Revenue	Customer
$10.99	23
$49.00	14
$11.00	56
$103.00	11
$84.50	39
$60.10	27
$7.60	23

Foreign Key

Probe Inputs

⋈

Hash Table (HT)

Customers (living in US)

Key	Zip
11	95014
23	94303
27	95040
39	95134

Primary Key Payload

=

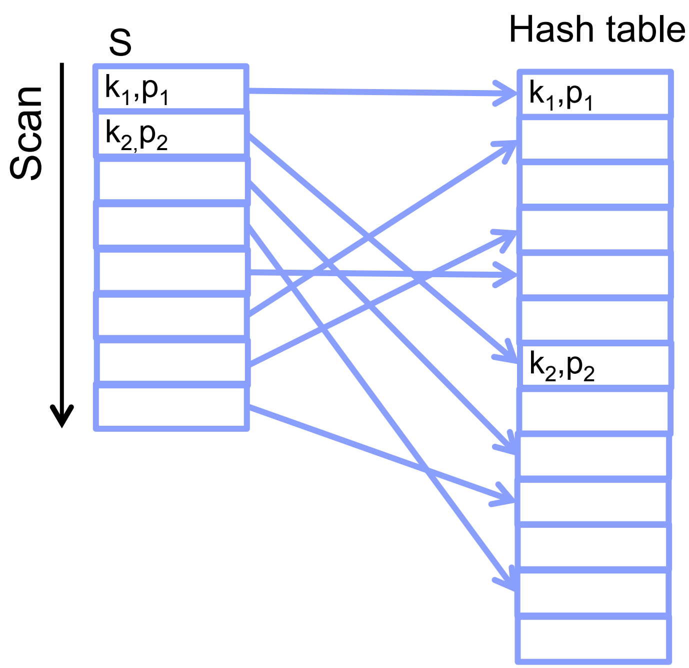
Revenue	Zip
$10.99	94303
$103.00	95014
$84.50	95134
$60.10	95040
$7.60	94303

Join Results

Hash Join

Join two tables (|S| < |R|) in 2 steps

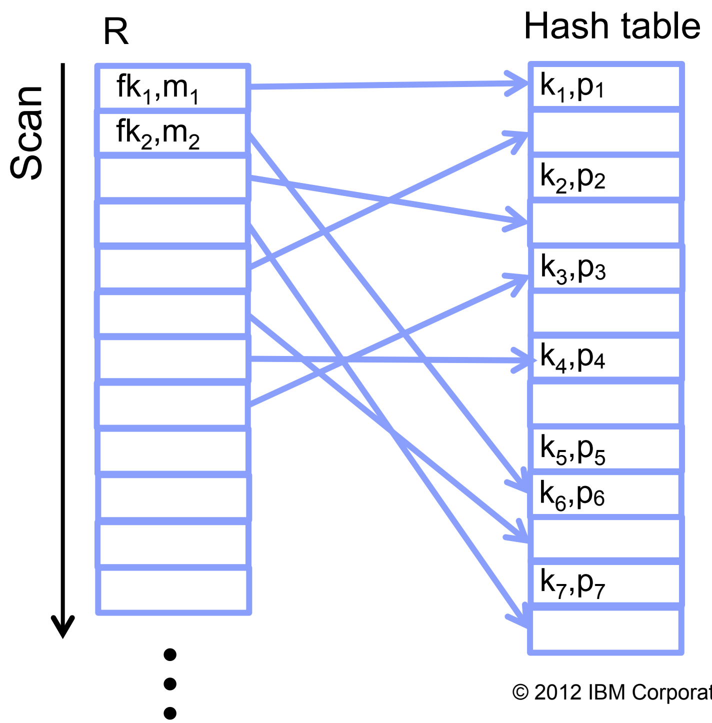
1. Build a hash table
   - Scan S and compute a location (hash) based on a unique (primary) key
   - Insert primary key **k** with payload **p** into the hash table
   - If the location is occupied pick the next free one (open addressing)



Hash Join

Join two tables (|S| < |R|) in 2 steps

1. Build a hash table
   - Scan S and compute a location (hash) based on a unique (primary) key
   - Insert primary key **k** with payload **p** into the hash table
   - If the location is occupied pick the next free one (open addressing)
2. Probe the hash table
   - Scan R and compute a location (hash) based on the reference to S (foreign key)
   - Compare foreign key **fk** and key **k** in hash table
   - If there is a match store the result (**m**,**p**)

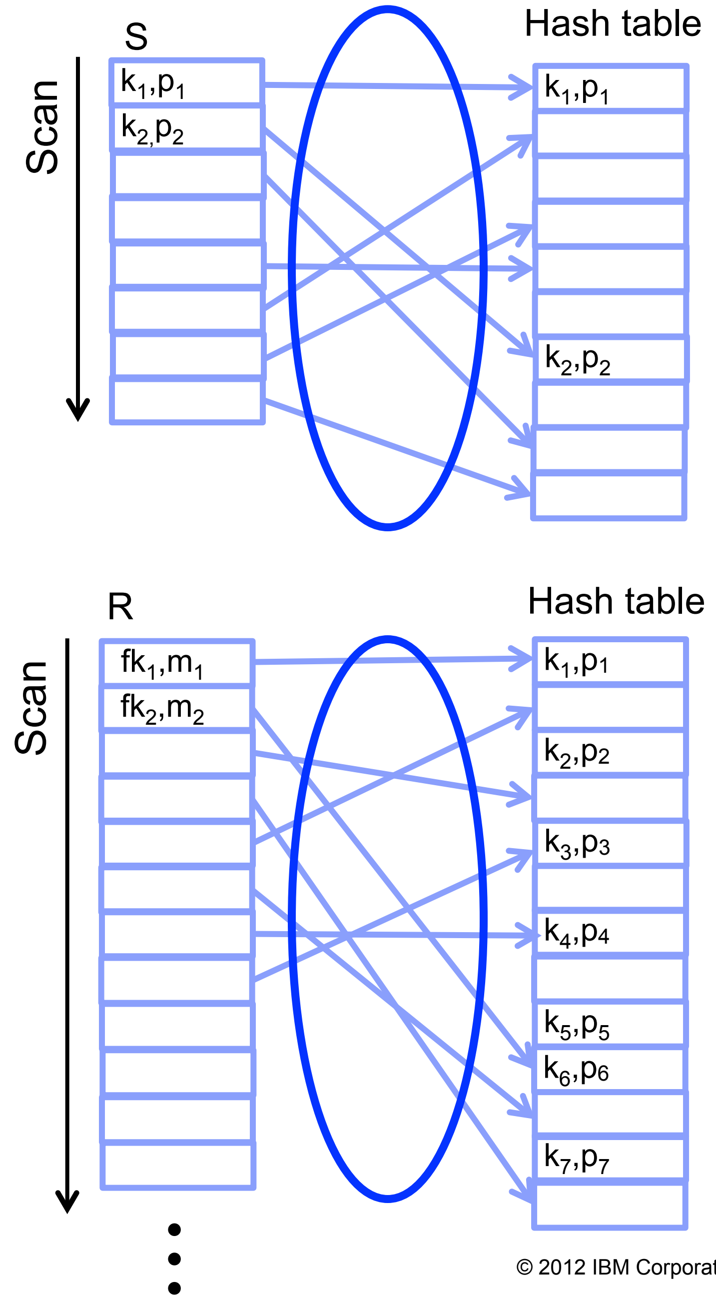


Hash Join

Join two tables (|S| < |R|) in 2 steps

1. Build a hash table
   - Scan S and compute a location (hash) based on a unique (primary) key
   - Insert primary key **k** with payload **p** into the hash table
   - If the location is occupied pick the next free one (open addressing)
2. Probe the hash table
   - Scan R and compute a location (hash) based on the reference to S (foreign key)
   - Compare foreign key **fk** and key **k** in hash table
   - If there is a match store the result (**m**,**p**)

Build and Probe produce a **random** data access pattern!



Hash Join – Data Access Patterns

- Primary data access patterns:
  - *Scan* the input table(s) for HT creation and probe
  - *Compare and swap* when inserting data into HT
  - *Random read* when probing the HT

Hash Join - Summary

- Primary data access patterns:
  - *Scan* the input table(s) for HT creation and probe
  - *Compare and swap* when inserting data into HT
  - *Random read* when probing the HT

- Data (memory) access on

vs.

	GPU (GTX580)	CPU (i7-2600)
Peak memory bandwidth [spec] 1)	179 GB/s	21 GB/s
Peak memory bandwidth [measured] 2)	153 GB/s	18 GB/s

Upper bound for:
Scan R, S

(1) Nvidia: 192.4 × 10^6 B/s ≈ 179.2 GB/s
(2) 64-bit accesses over 1 GB of device memory

Hash Join - Summary

- Primary data access patterns:
  - *Scan* the input table(s) for HT creation and probe
  - *Compare and swap* when inserting data into HT
  - *Random read* when probing the HT
- Data (memory) access on

vs.

	GPU (GTX580)	CPU (i7-2600)	Upper bound for:
Peak memory bandwidth [spec] [1)]	179 GB/s	21 GB/s	
Peak memory bandwidth [measured] [2)]	153 GB/s	18 GB/s	
Random access [measured] [2)]	6.6 GB/s	0.8 GB/s	Probe
Compare and swap [measured] [3)]	4.6 GB/s	0.4 GB/s	Build HT

(1) Nvidia: 192.4×10^6 B/s ≈ 179.2 GB/s
(2) 64-bit accesses over 1 GB of device memory
(3) 64-bit compare-and-swap to random locations over 1 GB device memory

Agenda

- A closer look at data warehousing queries
  - From queries down to operators
  - Where does time go?
  - Hash Join operators
  - Data Access Patterns

- Drill-down: Hash Tables on GPUs
  - Hash computation
  - Hash Tables = Hash computation + Memory access
  - Optimizations

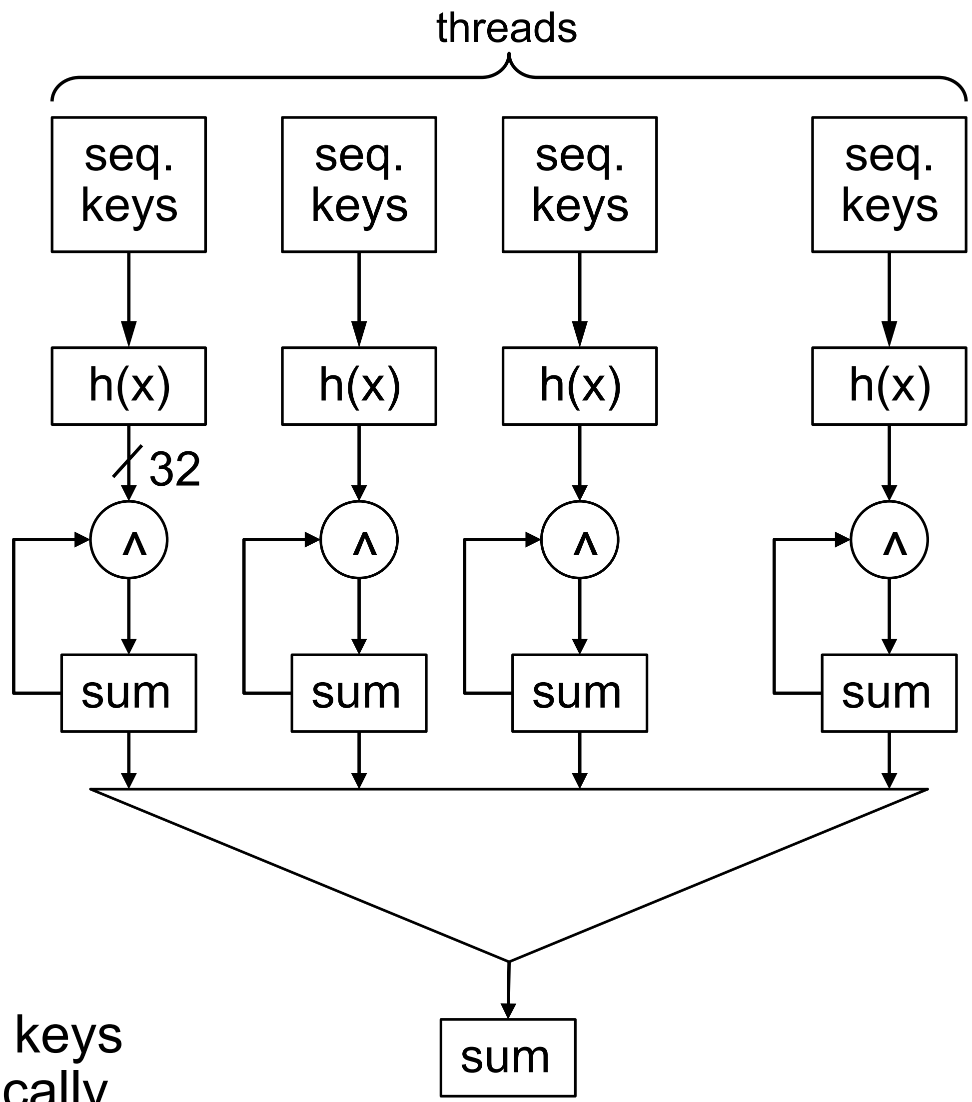
- From Hash Tables to Relational Joins
  - Hash Join Implementation
  - Query Performance
  - Processing 100s of GBs in seconds

Computing Hash Functions on GTX580 – No Reads

32-bit keys, 32-bit hashes

Hash Function/ Key Ingest GB/s	Seq keys+ Hash
LSB	338
Fowler-Noll-Vo 1a	129
Jenkins Lookup3	79
Murmur3	111
One-at-a-time	85
CRC32	78
MD5	4.5
SHA1	0.81

Cryptographic message digests



- Threads generate sequential keys
- Hashes are XOR-summed locally

Hash Table Probe: Keys from Device Memory – No results

32-bit hashes, 32-bit values

Hash Function/ Key Ingest GB/s	Seq keys+ Hash	HT Probe keys: dev values: sum
LSB	338	2.7
Fowler-Noll-Vo 1a	129	2.8
Jenkins Lookup3	79	2.7
Murmur3	111	2.7
One-at-a-time	85	2.7
CRC32	78	2.7
MD5	4.5	2.4
SHA1	0.81	0.7

- 1 GB hash table on device memory (load factor = 0.33)
- Keys are read from device memory
- 20% of the probed keys find match in hash table
- Values are XOR-summed locally

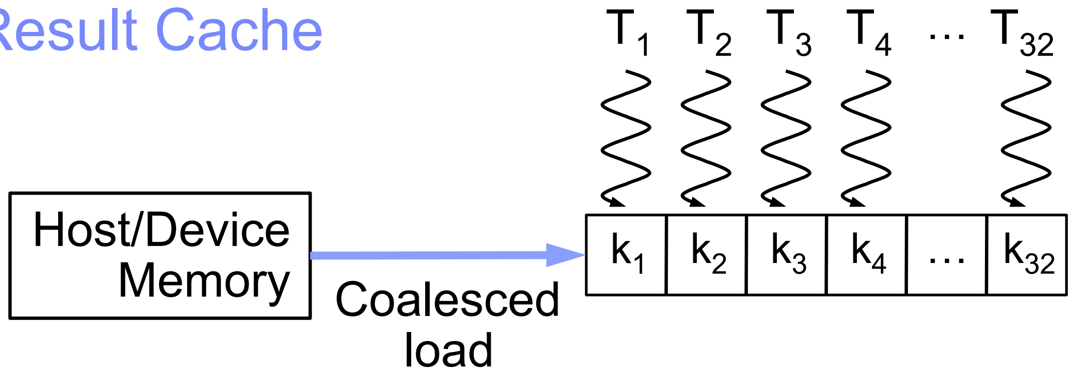
Hash Table Probe: Keys and Values from/to Device Memory

32-bit hashes, 32-bit values

Hash Function/ Key Ingest GB/s	Seq keys+ Hash	HT Probe keys: dev values: sum	HT Probe keys: dev values: dev
LSB	338	2.7	1.7
Fowler-Noll-Vo 1a	129	2.8	1.7
Jenkins Lookup3	79	2.7	1.7
Murmur3	111	2.7	1.7
One-at-a-time	85	2.7	1.7
CRC32	78	2.7	1.7
MD5	4.5	2.4	1.7
SHA1	0.81	0.7	0.7

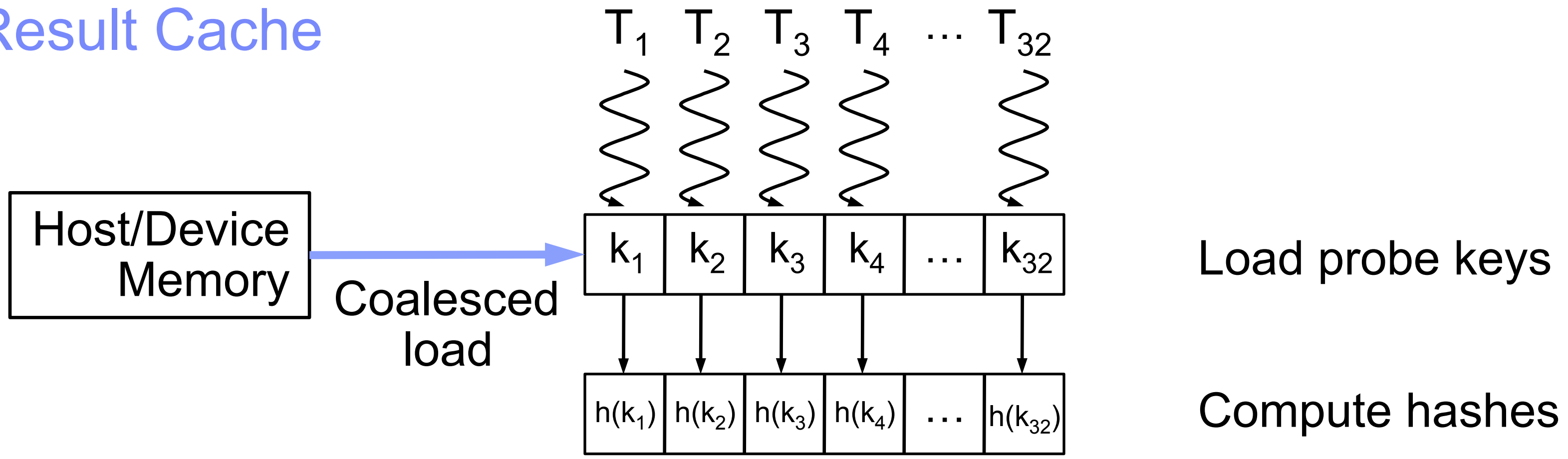
- 1 GB hash table on device memory (load factor = 0.33)
- Keys are read from device memory
- 20% of the probed keys find match in hash table
- Values are written back to device memory

Result Cache

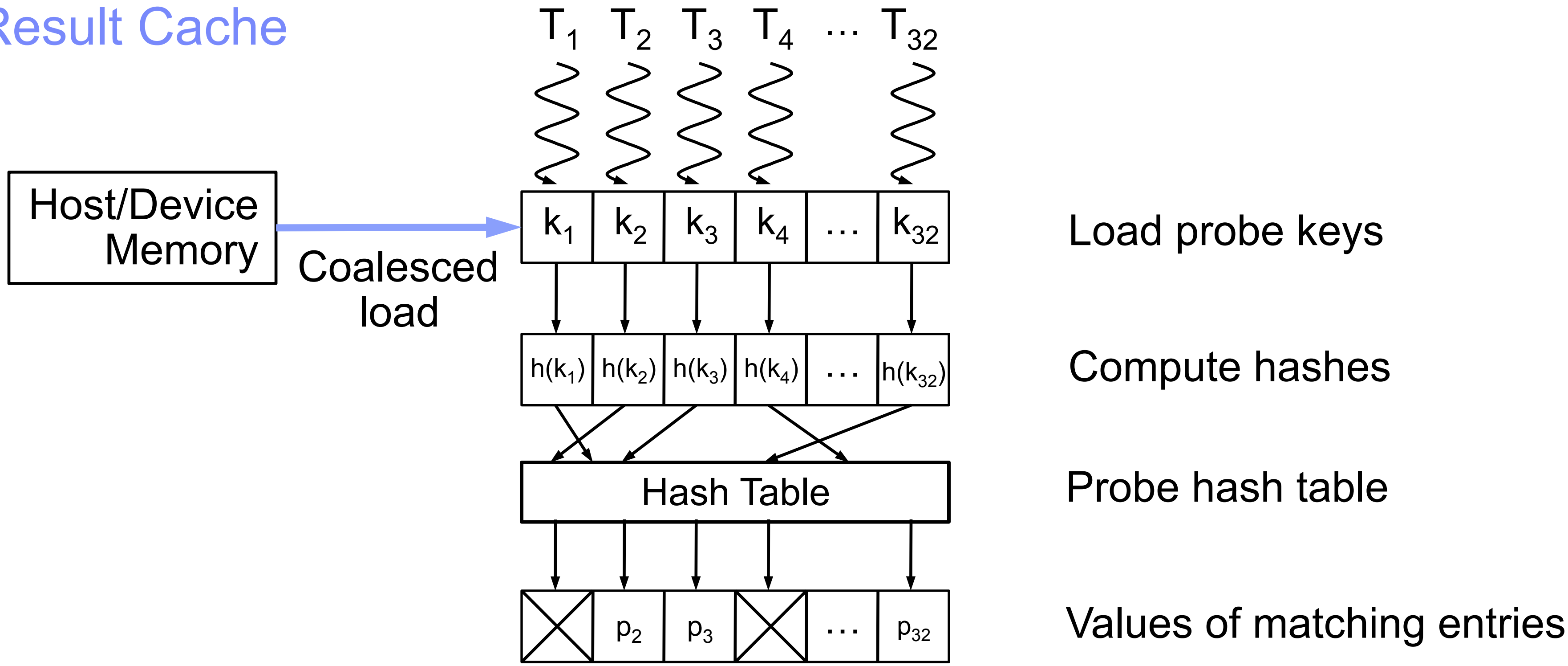


Load probe keys

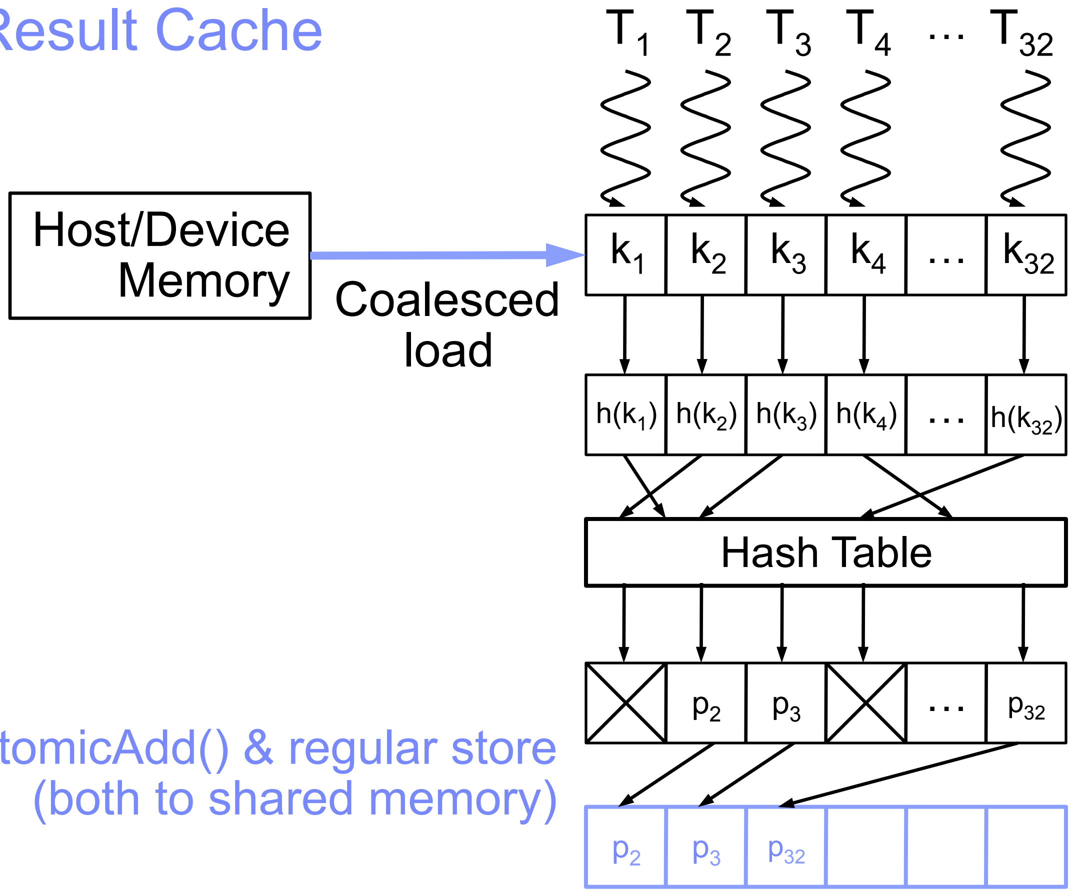
Result Cache



Result Cache



Result Cache



atomicAdd() & regular store (both to shared memory)

Load probe keys

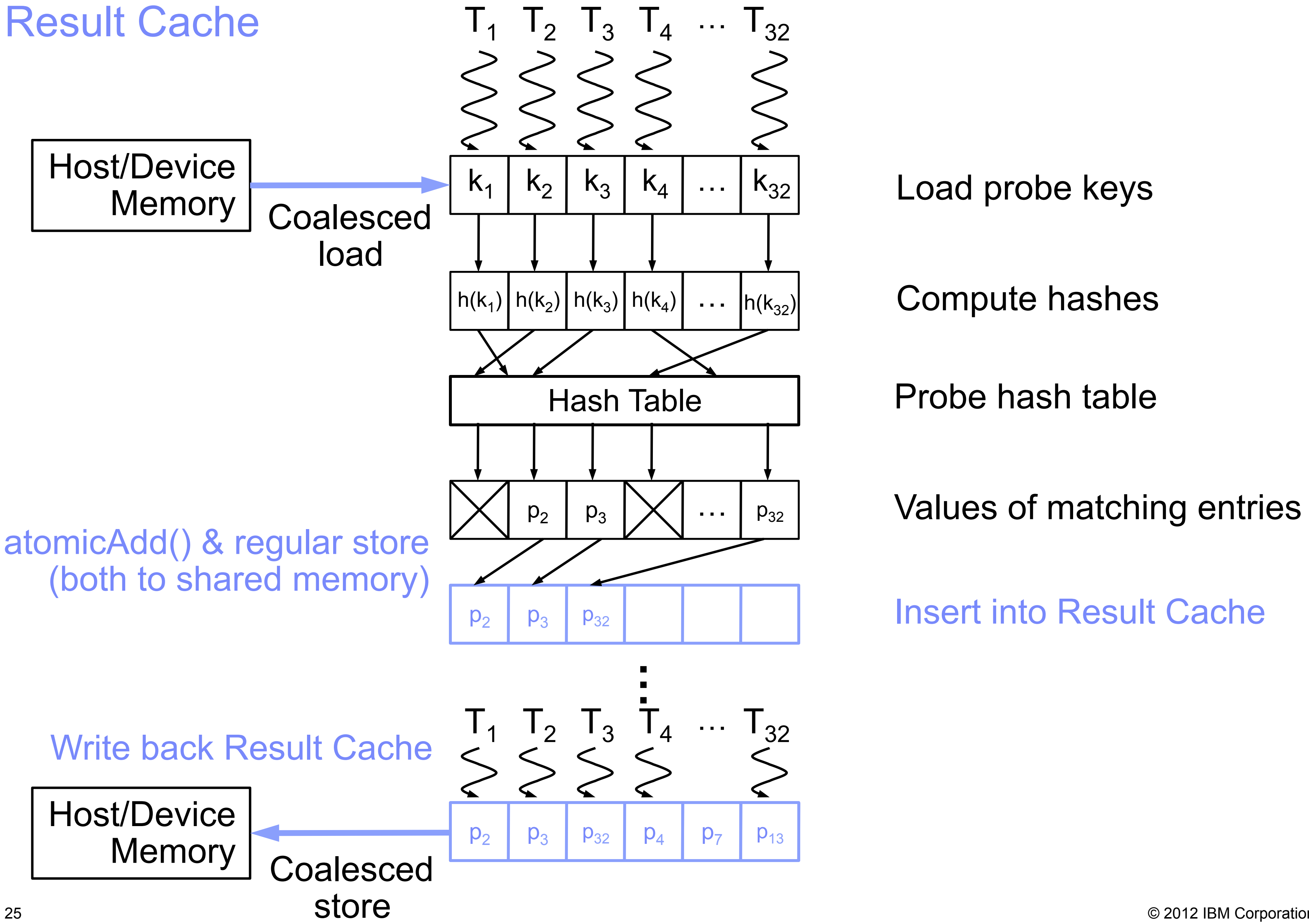
Compute hashes

Probe hash table

Values of matching entries

Insert into Result Cache

Result Cache



Probe with Result Cache: Keys and Values from/to Device Memory

32-bit hashes, 32-bit values

Hash Function/ Key Ingest GB/s	Seq keys+ Hash	HT Probe keys: dev values: sum	HT Probe keys: dev values: dev	Res. Cache keys: dev values: dev
LSB	338	2.7	1.7	2.4
Fowler-Noll-Vo 1a	129	2.8	1.7	2.5
Jenkins Lookup3	79	2.7	1.7	2.4
Murmur3	111	2.7	1.7	2.4
One-at-a-time	85	2.7	1.7	2.4
CRC32	78	2.7	1.7	2.4
MD5	4.5	2.4	1.7	1.8
SHA1	0.81	0.7	0.7	0.6

- 1 GB hash table on device memory (load factor = 0.33)
- Keys are read from device memory
- 20% of the probed keys find match in hash table
- Individual values are written back to buffer in shared memory and then coalesced to device memory

Probe with Result Cache: Keys and Values from/to Host Memory

32-bit hashes, 32-bit values, 1 GB hash table on device memory (load factor = 0.33)

Hash Function/ Key Ingest GB/s	HT Probe keys: dev values: sum	HT Probe keys: dev values: dev	Res. Cache keys: dev Values: dev	Res. Cache keys: host Values: host
LSB	2.7	1.7	2.4	2.3
Fowler-Noll-Vo 1a	2.8	1.7	2.5	2.4
Jenkins Lookup3	2.7	1.7	2.4	2.3
Murmur3	2.7	1.7	2.4	2.3
One-at-a-time	2.7	1.7	2.4	2.3
CRC32	2.7	1.7	2.4	2.3
MD5	2.4	1.7	1.8	1.8
SHA1	0.7	0.7	0.6	0.6

- Keys are read from **host memory (zero-copy access)**
- 20% of the probed keys find match in hash table
- Individual values are written back to buffer in shared memory and then coalesced to **host memory (zero-copy access)**

End-to-end comparison of Hash Table Probe: GPU vs. CPU

32-bit hashes, 32-bit values, 1 GB hash table (load factor = 0.33)

Hash Function/ Key Ingest GB/s	GTX580 keys: host values: host	i7-2600 4 cores 8 threads	Speedup
LSB	2.3	0.48	4.8×
Fowler-Noll-Vo 1a	2.4	0.47	5.1×
Jenkins Lookup3	2.3	0.46	5.0×
Murmur3	2.3	0.46	5.0×
One-at-a-time	2.3	0.43	5.3×
CRC32	2.3	0.48[1)]	4.8×
MD5	1.8	0.11	16×
SHA1	0.6	0.06	10×

- Result cache used in both implementations
- GPU: keys from host memory, values back to host memory
- CPU: software prefetching instructions for hash table loads

1) Use of CRC32 instruction in SSE 4.2

Agenda

- A closer look at data warehousing queries
  - From queries down to operators
  - Where does time go?
  - Hash Join operators
  - Data Access Patterns

- Drill-down: Hash Tables on GPUs
  - Hash computation
  - Hash Tables = Hash computation + Memory access
  - Optimizations

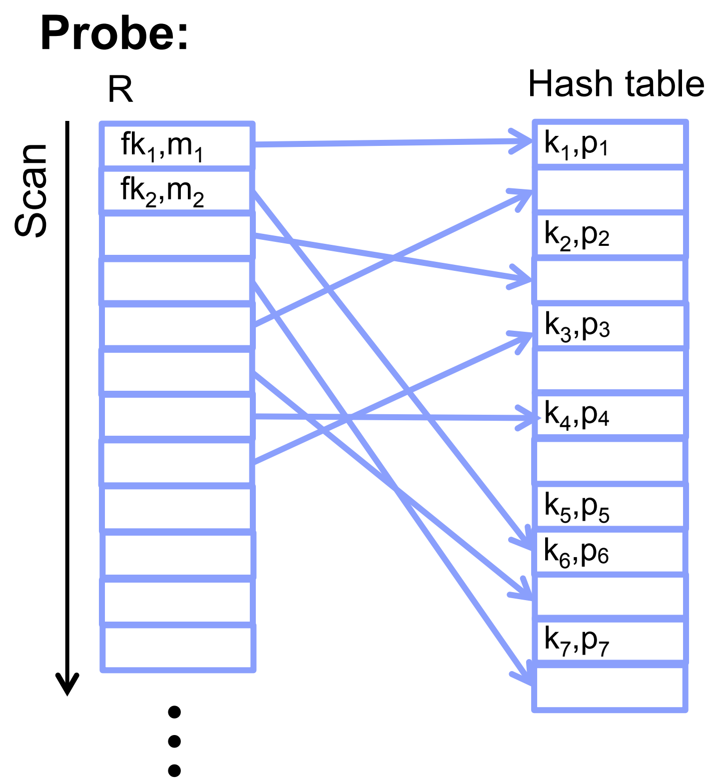
- From Hash Tables to Relational Joins
  - Hash Join Implementation
  - Query Performance
  - Processing 100s of GBs in seconds

From Hash Tables back to Relational Joins

- Equijoin return all pairs (m_i,p_j) where $fk_i=k_j$
- During probing (fk,m) pairs need to be transferred to the GPU not just fk.

Example: fk, m are 32 bit

- HT lookup 2.3 GB/s for 32 bit keys
- Ingest Bandwidth to GPU needed: 2×2.3 GB/s = **4.6 GB/s**

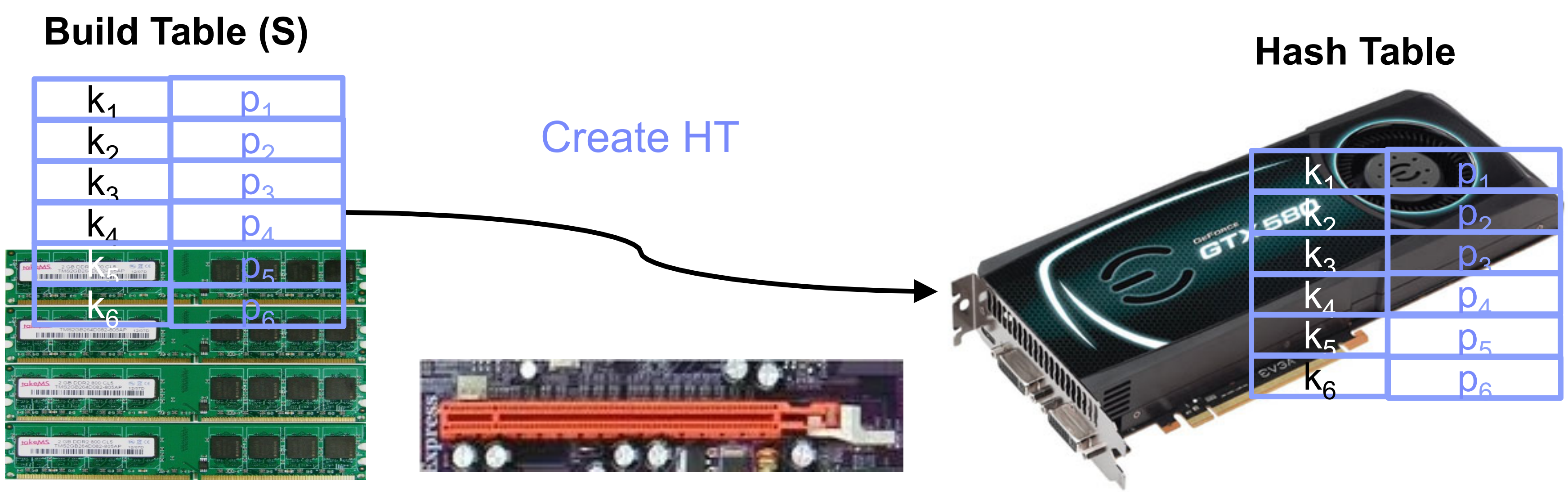


Join Results:
(m_1,p_1), (m_2,p_6), …

Hash Join Implementation

1. Pin table S for Build in host memory
2. Simultaneously read table S from host memory

   & create hash table on device



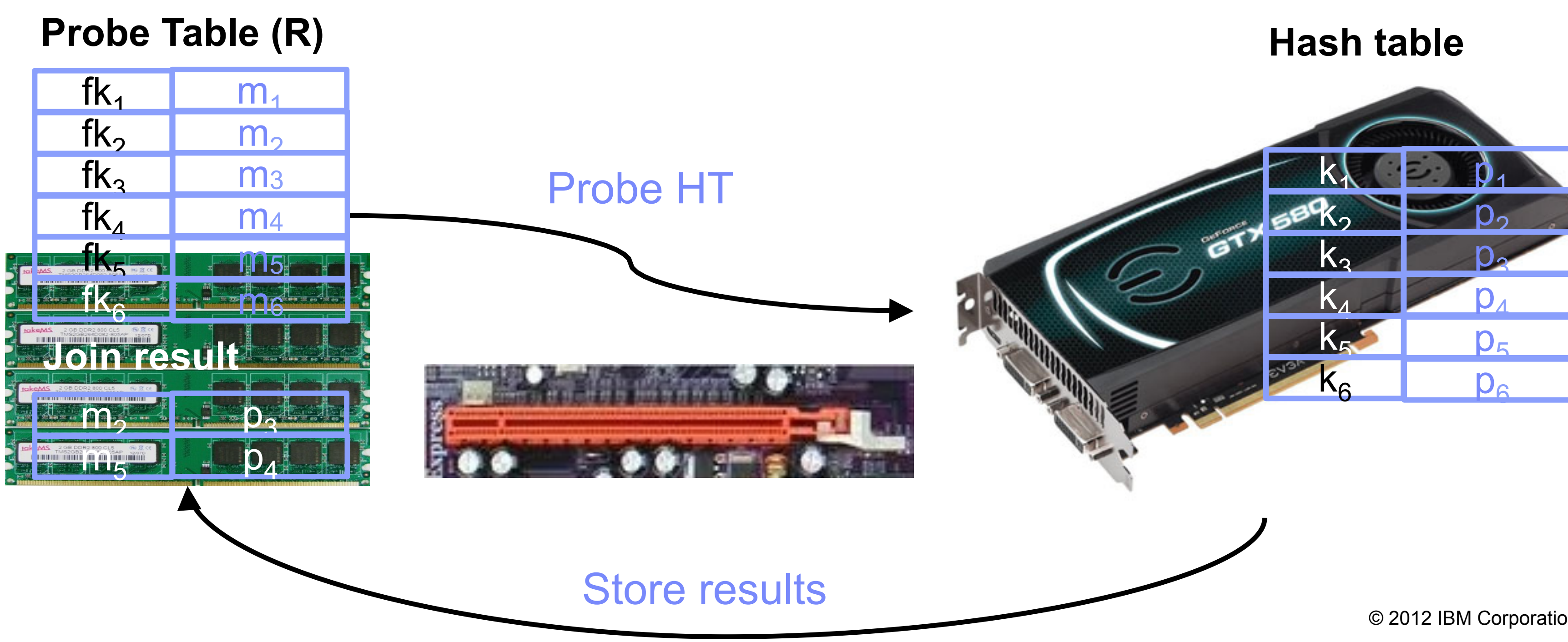
Hash Join Implementation

1. Pin table S for Build in host memory
2. Simultaneously read table S from host memory

   & create hash table on device
3. Simultaneously read table R for Probe from host memory

   & probe hash table on device

   & store results in host memory



Results: Complete Join from Star Schema Benchmark

Conservative Assumptions for previous micro-benchmarks:

- large hash table (1 GB)
- *large match rate (20%)*

Now: Query from a Benchmark

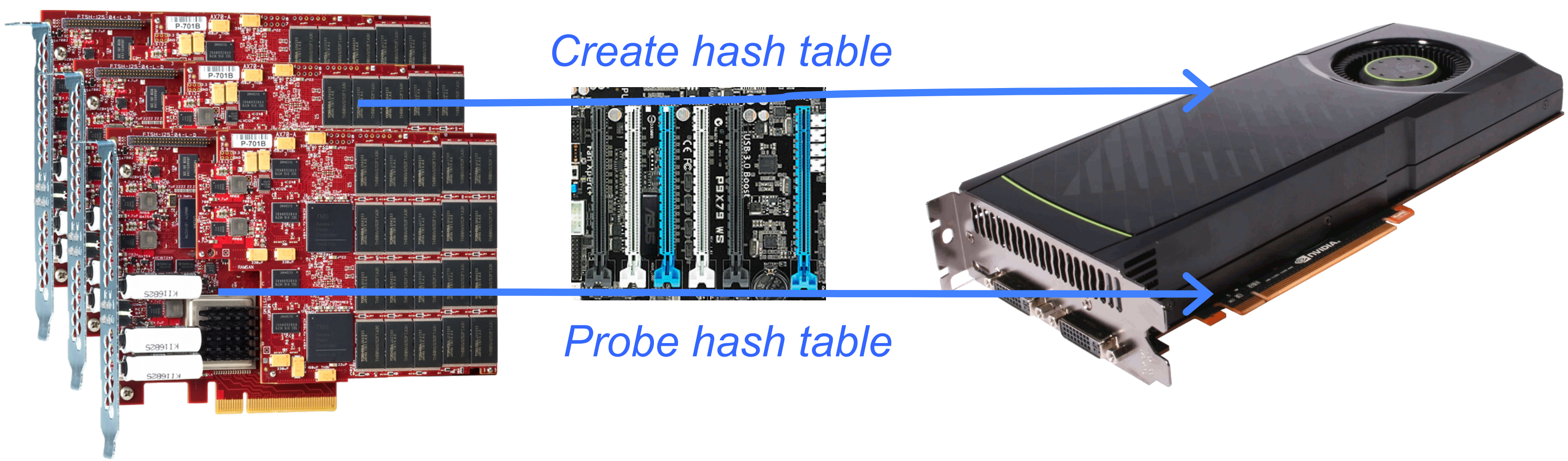
Star Schema Benchmark:

- First join in Query Q3.2:
  lineorder ⋈ customer
- DB Size: 714 GB
  Scale Factor 1,000 (6 billion rows)
- *Match rate 4%*
- Measured ingest rate on GTX580:
  5.77 GiB/s
- This corresponds to **92%** of the theoretical PCI-E 2.0 x16 bandwidth.

PCI-E 2.0 x16: 8 GB/s with 128 B TLP payload/152 B TLP total = 6.274 GiB/s

Processing hundreds of Gigabytes in seconds

- Machines with ½ TB of memory are not commodity yet (even at IBM ;-)
- How about reading the input tables on the fly from flash?



- Storage solution delivering data at GPU join speed (>5.7 GB/s):
  - 3x 900 GB IBM Texas Memory Systems RamSan-70 SSDs
  - IBM Global Parallel File System (GPFS)

→ Visit us at the IBM booth #607 in the exhibition hall for a **live demo !**