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**: 
  ```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**: 
  ```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 closer look at DWH queries

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**: 
  ```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?
A closer look at DWH queries

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;
# A closer look at DWH queries

## Relational Joins

### Sales (Fact Table)

<table>
<thead>
<tr>
<th>Revenue</th>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10.99</td>
<td>23</td>
</tr>
<tr>
<td>$49.00</td>
<td>14</td>
</tr>
<tr>
<td>$11.00</td>
<td>56</td>
</tr>
<tr>
<td>$103.00</td>
<td>11</td>
</tr>
<tr>
<td>$84.50</td>
<td>39</td>
</tr>
<tr>
<td>$60.10</td>
<td>27</td>
</tr>
<tr>
<td>$7.60</td>
<td>23</td>
</tr>
</tbody>
</table>

### Customers (living in US)

<table>
<thead>
<tr>
<th>Key</th>
<th>Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>95014</td>
</tr>
<tr>
<td>23</td>
<td>94303</td>
</tr>
<tr>
<td>27</td>
<td>95040</td>
</tr>
<tr>
<td>39</td>
<td>95134</td>
</tr>
</tbody>
</table>

### Join Results

<table>
<thead>
<tr>
<th>Revenue</th>
<th>Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10.99</td>
<td>94303</td>
</tr>
<tr>
<td>$103.00</td>
<td>95014</td>
</tr>
<tr>
<td>$84.50</td>
<td>95134</td>
</tr>
<tr>
<td>$60.10</td>
<td>95040</td>
</tr>
<tr>
<td>$7.60</td>
<td>94303</td>
</tr>
</tbody>
</table>

**Join**

- **Payload**
  - **Primary Key**: 11, 23, 27, 39
  - **Payload**: 95014, 94303, 95040, 95134

- **Foreign Key**: 23

**Measure**

- **Measure**: Revenue
  - $10.99, $49.00, $11.00, $103.00, $84.50, $60.10, $7.60
### Hash Join

#### A closer look at DWH queries

**Hash Table (HT)**

<table>
<thead>
<tr>
<th>Key</th>
<th>Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>95014</td>
</tr>
<tr>
<td>23</td>
<td>94303</td>
</tr>
<tr>
<td>27</td>
<td>95040</td>
</tr>
<tr>
<td>39</td>
<td>95134</td>
</tr>
</tbody>
</table>

**Customers (living in US)**

<table>
<thead>
<tr>
<th>Revenue</th>
<th>Zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10.99</td>
<td>94303</td>
</tr>
<tr>
<td>$103.00</td>
<td>95014</td>
</tr>
<tr>
<td>$84.50</td>
<td>95134</td>
</tr>
<tr>
<td>$60.10</td>
<td>95040</td>
</tr>
<tr>
<td>$7.60</td>
<td>94303</td>
</tr>
</tbody>
</table>

**Sales (Fact Table)**

<table>
<thead>
<tr>
<th>Revenue</th>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10.99</td>
<td>23</td>
</tr>
<tr>
<td>$49.00</td>
<td>14</td>
</tr>
<tr>
<td>$11.00</td>
<td>56</td>
</tr>
<tr>
<td>$103.00</td>
<td>11</td>
</tr>
<tr>
<td>$84.50</td>
<td>39</td>
</tr>
<tr>
<td>$60.10</td>
<td>27</td>
</tr>
<tr>
<td>$7.60</td>
<td>23</td>
</tr>
</tbody>
</table>

**Probe Inputs**

**Join Results**

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

**Primary Key**

**Payload**

**Foreign Key**

**Key**

IBM
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

<table>
<thead>
<tr>
<th></th>
<th>GPU (GTX580)</th>
<th>CPU (i7-2600)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peak memory bandwidth [spec]</td>
<td>179 GB/s</td>
<td>21 GB/s</td>
</tr>
<tr>
<td>Peak memory bandwidth [measured]</td>
<td>153 GB/s</td>
<td>18 GB/s</td>
</tr>
</tbody>
</table>

Upper bound for: Scan R, S

(1) Nvidia: $192.4 \times 10^6$ B/s $\approx 179.2$ 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

<table>
<thead>
<tr>
<th></th>
<th>GPU (GTX580)</th>
<th>CPU (i7-2600)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peak memory bandwidth [spec]</td>
<td>179 GB/s</td>
<td>21 GB/s</td>
</tr>
<tr>
<td>Peak memory bandwidth [measured]</td>
<td>153 GB/s</td>
<td>18 GB/s</td>
</tr>
<tr>
<td>Random access [measured]</td>
<td>6.6 GB/s</td>
<td>0.8 GB/s</td>
</tr>
<tr>
<td>Compare and swap [measured]</td>
<td>4.6 GB/s</td>
<td>0.4 GB/s</td>
</tr>
</tbody>
</table>

Upper bound for:
- Probe
  - Build HT

(1) Nvidia: $192.4 \times 10^6$ B/s $\approx 179.2$ 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

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

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

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

- 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

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

- 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

Host/Device Memory  Coalesced load

Load probe keys

T_1  T_2  T_3  T_4  ...  T_{32}

k_1  k_2  k_3  k_4  ...  k_{32}
Drill Down: Hash Tables on GPUs

Result Cache

Host/Device Memory

Coalesced load

T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow \ldots \rightarrow T_{32}

k_1 \rightarrow k_2 \rightarrow k_3 \rightarrow k_4 \rightarrow \ldots \rightarrow k_{32}

h(k_1) \rightarrow h(k_2) \rightarrow h(k_3) \rightarrow h(k_4) \rightarrow \ldots \rightarrow h(k_{32})

Load probe keys

Compute hashes
Result Cache

Load probe keys
Compute hashes
Probe hash table
Values of matching entries

Coalesced load

Host/Device Memory

T<sub>1</sub> T<sub>2</sub> T<sub>3</sub> T<sub>4</sub> ... T<sub>32</sub>

k<sub>1</sub> k<sub>2</sub> k<sub>3</sub> k<sub>4</sub> ... k<sub>32</sub>

h(k<sub>1</sub>) h(k<sub>2</sub>) h(k<sub>3</sub>) h(k<sub>4</sub>) ... h(k<sub>32</sub>)

Hash Table

p<sub>2</sub> p<sub>3</sub> ... p<sub>32</sub>
Result Cache

Load probe keys

Compute hashes

Probe hash table

Values of matching entries

Insert into Result Cache

Host/Device Memory

Coalesced load

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

Drill Down: Hash Tables on GPUs
Result Cache

Host/Device Memory

Coalesced load

Load probe keys

Compute hashes

Probe hash table

Values of matching entries

Insert into Result Cache

Write back Result Cache

Host/Device Memory

Coalesced store

Drill Down: Hash Tables on GPUs
Probe with Result Cache: Keys and Values from/to Device Memory
32-bit hashes, 32-bit values

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

- 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)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LSB</td>
<td>2.7</td>
<td>1.7</td>
<td>2.4</td>
<td>2.3</td>
</tr>
<tr>
<td>Fowler-Noll-Vo 1a</td>
<td>2.8</td>
<td>1.7</td>
<td>2.5</td>
<td>2.4</td>
</tr>
<tr>
<td>Jenkins Lookup3</td>
<td>2.7</td>
<td>1.7</td>
<td>2.4</td>
<td>2.3</td>
</tr>
<tr>
<td>Murmur3</td>
<td>2.7</td>
<td>1.7</td>
<td>2.4</td>
<td>2.3</td>
</tr>
<tr>
<td>One-at-a-time</td>
<td>2.7</td>
<td>1.7</td>
<td>2.4</td>
<td>2.3</td>
</tr>
<tr>
<td>CRC32</td>
<td>2.7</td>
<td>1.7</td>
<td>2.4</td>
<td>2.3</td>
</tr>
<tr>
<td>MD5</td>
<td>2.4</td>
<td>1.7</td>
<td>1.8</td>
<td>1.8</td>
</tr>
<tr>
<td>SHA1</td>
<td>0.7</td>
<td>0.7</td>
<td>0.6</td>
<td>0.6</td>
</tr>
</tbody>
</table>

- 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)

<table>
<thead>
<tr>
<th>Hash Function/Key Ingest GB/s</th>
<th>GTX580 keys: host values: host</th>
<th>i7-2600 4 cores 8 threads</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSB</td>
<td>2.3</td>
<td>0.48</td>
<td>4.8×</td>
</tr>
<tr>
<td>Fowler-Noll-Vo 1a</td>
<td>2.4</td>
<td>0.47</td>
<td>5.1×</td>
</tr>
<tr>
<td>Jenkins Lookup3</td>
<td>2.3</td>
<td>0.46</td>
<td>5.0×</td>
</tr>
<tr>
<td>Murmur3</td>
<td>2.3</td>
<td>0.46</td>
<td>5.0×</td>
</tr>
<tr>
<td>One-at-a-time</td>
<td>2.3</td>
<td>0.43</td>
<td>5.3×</td>
</tr>
<tr>
<td>CRC32</td>
<td>2.3</td>
<td>0.48¹)</td>
<td>4.8×</td>
</tr>
<tr>
<td>MD5</td>
<td>1.8</td>
<td>0.11</td>
<td>16×</td>
</tr>
<tr>
<td>SHA1</td>
<td>0.6</td>
<td>0.06</td>
<td>10×</td>
</tr>
</tbody>
</table>

- 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 \(f_k_i = k_j\)
- During probing \((f_k, m)\) pairs need to be transferred to the GPU not just \(f_k\).

**Example**: \(f_k, m\) are 32 bit

- HT lookup 2.3 GB/s for 32 bit keys
- Ingest Bandwidth to GPU needed: 
  \(2 \times 2.3 \text{ GB/s} = 4.6 \text{ GB/s}\)
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

<table>
<thead>
<tr>
<th>Probe Table (R)</th>
<th>Hash table</th>
</tr>
</thead>
<tbody>
<tr>
<td>fk₁, fk₂, fk₃, fk₄, fk₅, fk₆</td>
<td>k₁, k₂, k₃, k₄, k₅, k₆</td>
</tr>
<tr>
<td>m₁, m₂, m₃, m₄, m₅, m₆</td>
<td>p₁, p₂, p₃, p₄, p₅, p₆</td>
</tr>
</tbody>
</table>

Join result: Store results
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 !