Programming GPUs for database operations

Tim Kaldewey
Research Staff Member
IBM TJ Watson Research Center
tkaldew@us.ibm.com

Disclaimer

The author's views expressed in this presentation do not necessarily reflect the views of IBM.

Acknowledgements

I would like to thank all my co-authors from IBM and my prior positions at Oracle and UCSC whose work I am also showing in this presentation.

I would also like to thank Patrick Cozzi for inviting me to teach in this class multiple years in a row.

Agenda

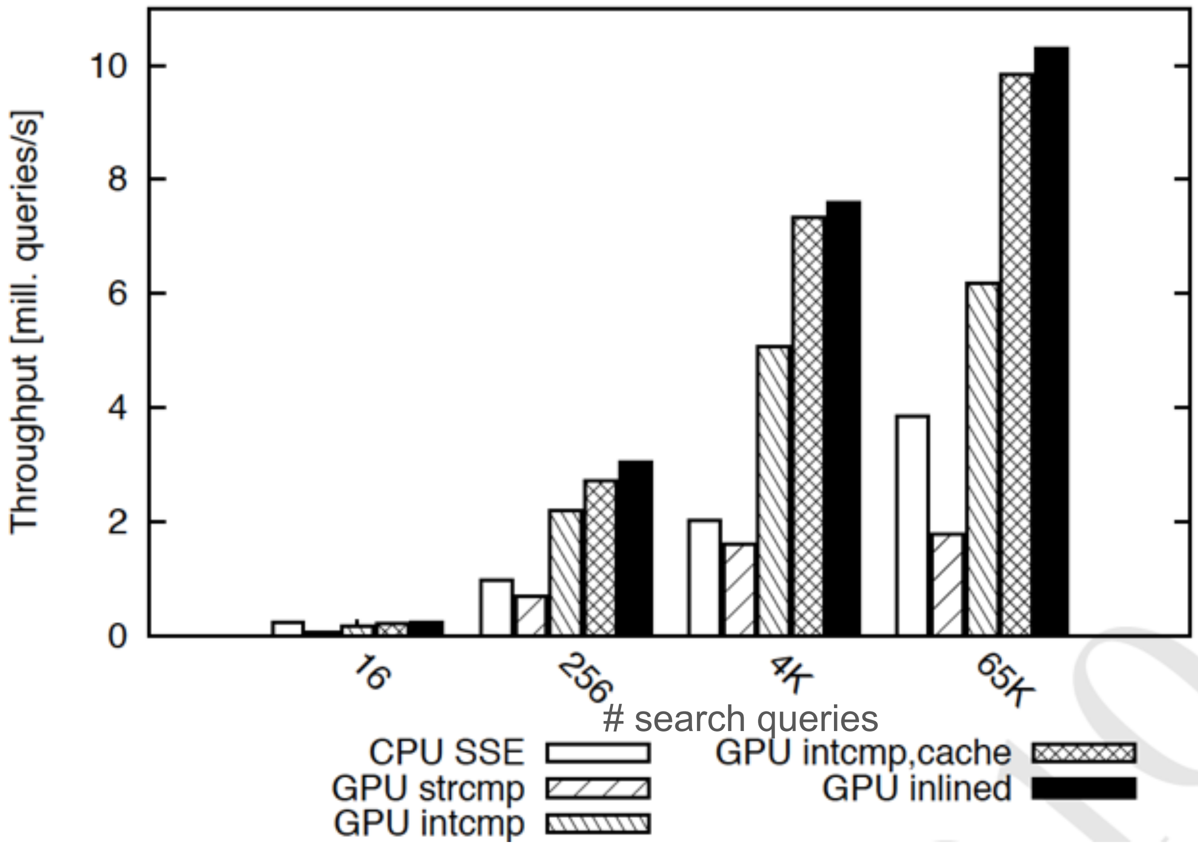
- GPU search
  - Reminder: Porting CPU search
  - Back to the drawing board:
    - P-ary search
    - Experimental evaluation
    - Why it works
- Building a GPU based data warehouse solution
  - From a query to operators
  - What to accelerate
  - What are the bottlenecks/limitations
- Maximizing data path efficiency
  - Extremely fast storage solution
  - Storage to host to device
- Putting it all together
  - Prototype demo

Binary Search on the GPU – optimized

- Replace byte-wise strcmp with larger word size (uint4)
  - What happens if we load character strings as integers ?
- Prefetch (cache) intermediate values in shared memory
  - Don't newer GPUs have caches ?
- Inline the function calls

Binary Search on the GPU – optimized

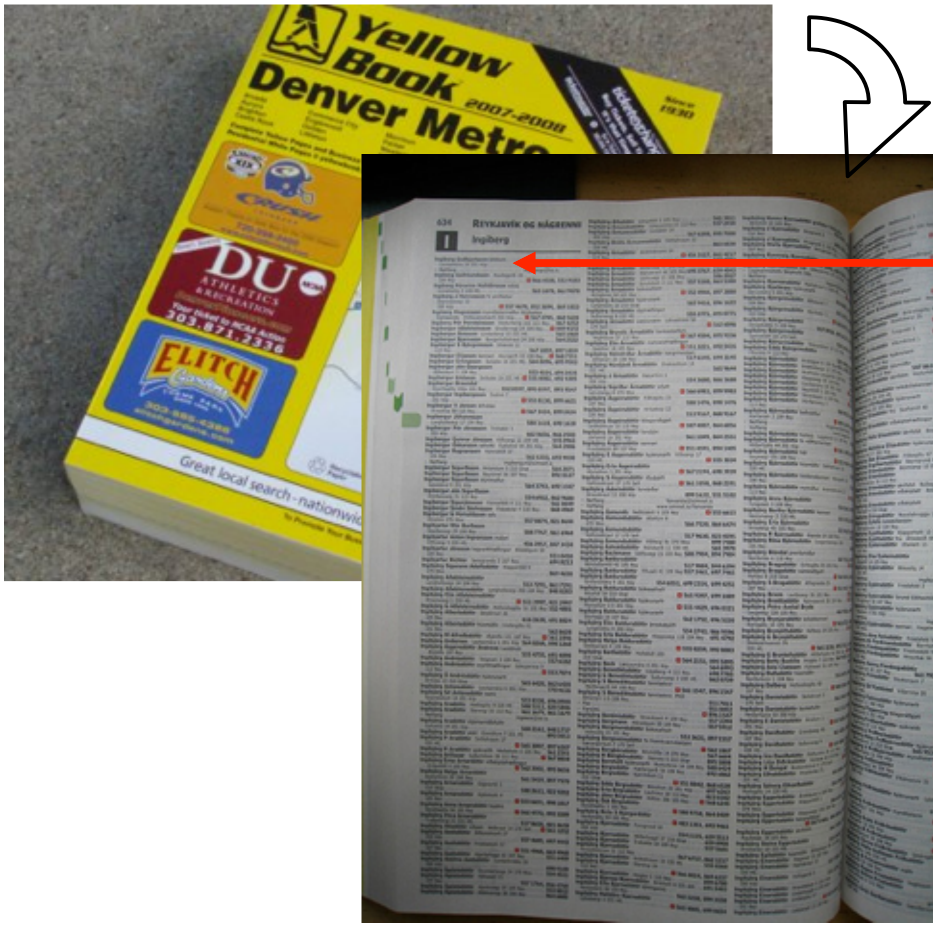
- Replace byte-wise strcmp with larger word size (uint4)
  - What happens if we load character strings as integers ?
- Prefetch (cache) intermediate values in shared memory
  - Don't newer GPUs have caches ?
- Inline the function calls



Searching a large data set (512MB) with 33 million (2^25) 16-character strings

Binary Search

- How Do you (efficiently) search an index?



- Open phone book ~middle
- 1st name = whom you are looking for?
- < , > ?
- Iterate
  - Each iteration: #entries/2 (n/2)
  - Total time: ➔ $\log_2(n)$

Parallel (Binary) Search

- What if you have some friends (3) to help you ?

- Divide et impera !
- Give each of them ¼ *
  – Each is using binary search takes $\log_2(n/4)$
  - All can work in parallel ➔ faster: $\log_2(n/4) < \log_2(n)$

* You probably want to tear it a little more intelligent than that, e.g. at the binding ;-)

Parallel (Binary) Search

- What if you have some friends (3) to help you ?

- Divide et impera !
- Give each of them ¼ *
  - Each is using binary search takes $\log_2(n/4)$
  - All can work in parallel ➔ faster: $\log_2(n/4) < \log_2(n)$
  - 3 of you are wasting time !

* You probably want to tear it a little more intelligent than that, e.g. at the binding ;-)

P-ary Search

- Divide et impera !!

- How do we know who has the right piece ?

P-ary Search

- Divide et impera !!

…

- How do we know who has the right piece ?

- It's a sorted list:
  - Look at first and last entry of a subset
  - If first entry < searched name < last entry
    - Redistribute
    - Otherwise … throw it away
  - Iterate

P-ary Search

- What do we get?

+

- Each iteration: n/4
  ➔ $\log_4(n)$
- Assuming redistribution time is negligible:
  $\log_4(n) < \log_2(n/4) < \log_2(n)$
- But each does 2 lookups !
- How time consuming are lookup and redistribution ?

P-ary Search

- What do we get?

+

- Each iteration: n/4
  $\Rightarrow \log_4(n)$
- Assuming redistribution time is negligible:
  $\log_4(n) < \log_2(n/4) < \log_2(n)$
- But each does 2 lookups !
- How time consuming are lookup and redistribution ?

=	=
memory access	synchronization

P-ary Search

- What do we get?

\+

- Each iteration: n/4
  ➔ $\log_4(n)$
- Assuming redistribution time is negligible:
  $\log_4(n) < \log_2(n/4) < \log_2(n)$
- But each does 2 lookups !
- How time consuming are lookup and redistribution ?

|| ||

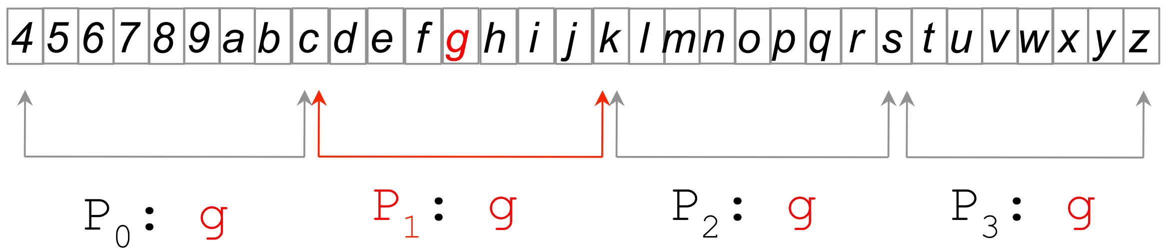
memory access   synchronization

- Searching a database index can be implemented the same way
  - Friends = Processor cores (threads)
  - Without destroying anything ;-)

P-ary Search - Implementation

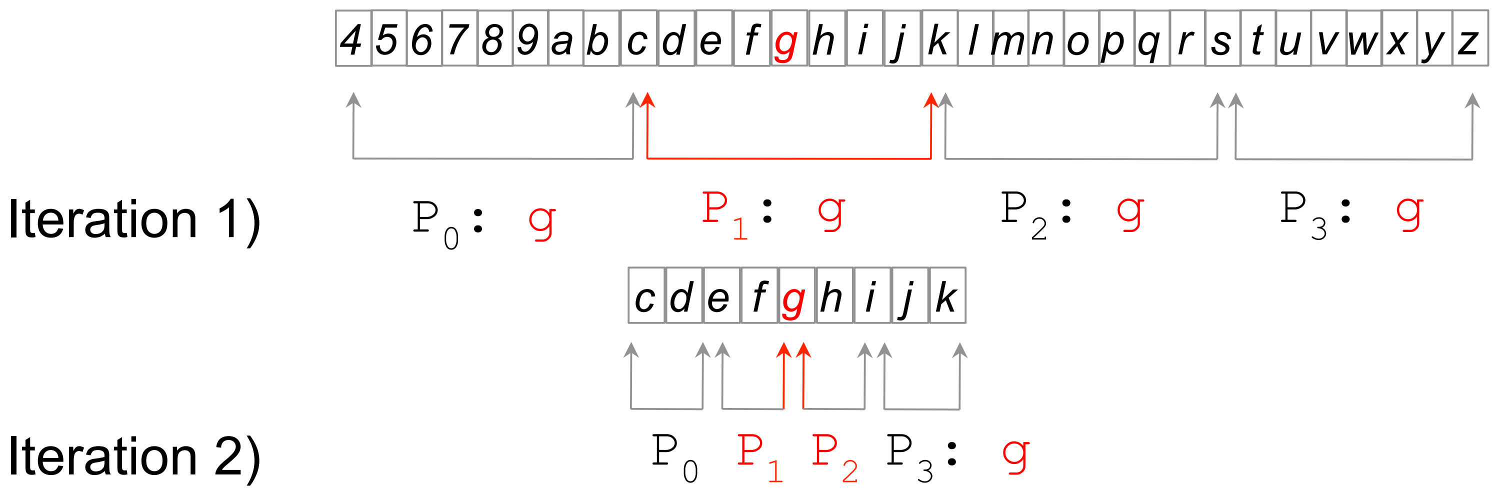
- Strongly relies on fast synchronization
  - friends = threads / vector elements

Iteration 1)



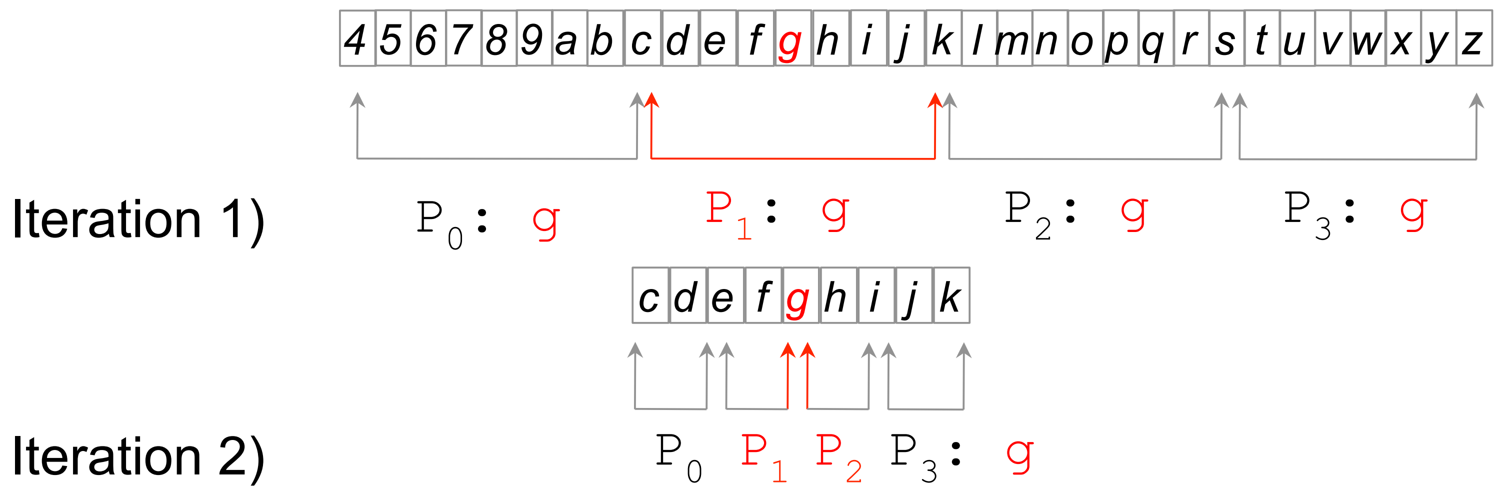
P-ary Search - Implementation

- Strongly relies on fast synchronization
  - friends = threads / vector elements



P-ary Search - Implementation

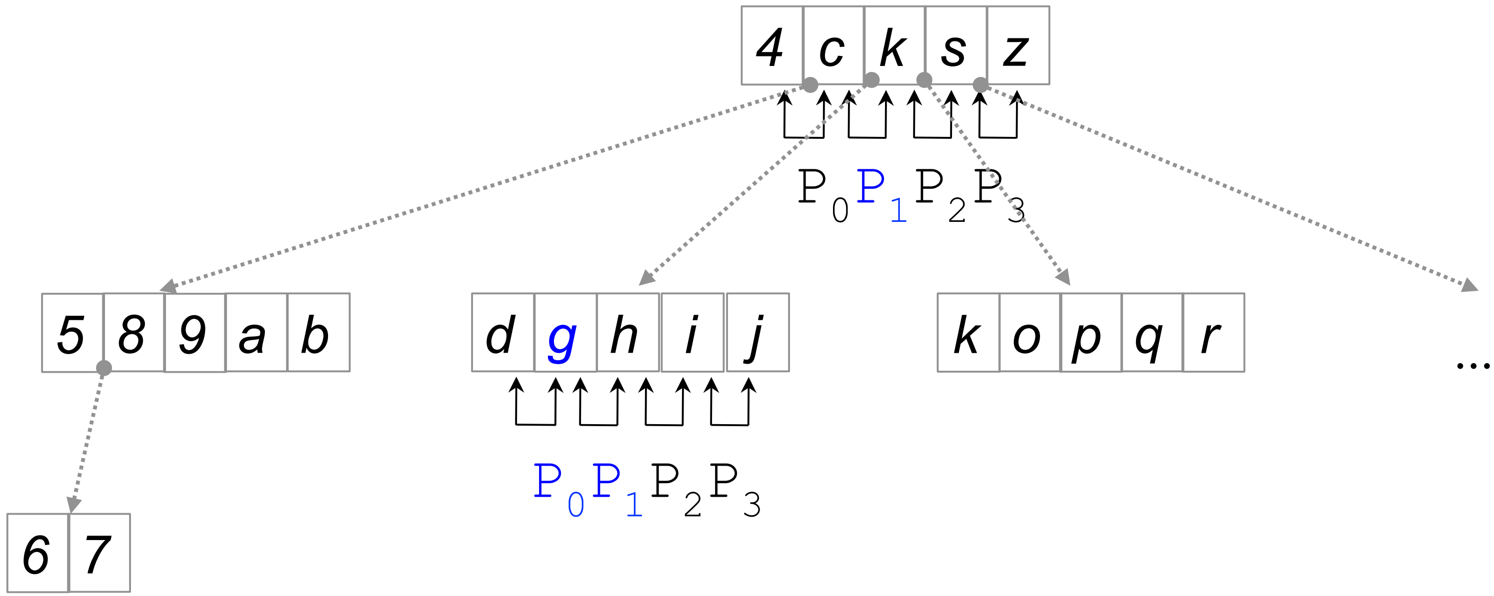
- Strongly relies on fast synchronization
  - friends = threads / vector elements



  - Synchronization ~ repartition cost
  - pthreads ($$), **`cmpxchng`**($)
  - SIMD SSE-vector, GPU threads via shared memory (~0)
- Implementation using a B-tree is similar and (obviously) faster

P-ary Search - Implementation

- B-trees group pivot elements into nodes



- Access to pivot elements is coalesced instead of a gather
- Nodes can also be mapped to
  – Cache Lines (CSB+ trees)
  – Vectors (SSE)
  – #Threads per block

P-ary Search on a sorted integer list – Implementation (1)

```
__shared__ int offset;
__shared__ int cache[BLOCKSIZE+2]

__global__ void parySearchGPU(int* data, int length,
                              int* list_of_search_keys, int* results)

   int start, sk;
   int old_length = length;
// initialize search range starting with the whole data set
   if (threadIdx.x ==0 ) {
      offset = 0;
      // cache search key and upper bound in shared memory
      cache[BLOCKSIZE] = 0x7FFFFFFF;
      cache[BLOCKSIZE+1] = list_of_search_keys[blockIdx.x];
      results[blockIdx.x] = -1;
   }
   __syncthreads();
   //
   sk = cache[BLOCKSIZE+1];
```

P-ary Search on a sorted integer list – Implementation (1)

```
__shared__ int offset;
__shared__ int cache[BLOCKSIZE+2]

__global__ void parySearchGPU(int* data, int length,
                              int* list_of_search_keys, int* results)

   int start, sk;
   int old_length = length;
// initialize search range starting with the whole data set
   if (threadIdx.x ==0 ) {
      offset = 0;
      // cache search key and upper bound in shared memory
      cache[BLOCKSIZE] = 0x7FFFFFFF;
      cache[BLOCKSIZE+1] = list_of_search_keys[blockIdx.x];
      results[blockIdx.x] = -1;
   }
   __syncthreads();
   //
   sk = cache[BLOCKSIZE+1];
```

Why?

P-ary Search on a sorted list – Implementation (2)

```
// repeat until the #keys in the search range < #threads
while (length > BLOCKSIZE){
    // calculate search range for this thread
    length = length/BLOCKSIZE;
    if (length * BLOCKSIZE < old_length) length += 1;
    old_length = length;
    // why don't we just use floating point?
    start = offset + threadIdx.x * length;
    // cache the boundary keys
    cache[threadIdx.x] = data[start];
    __syncthreads();
    // if the searched key is within this thread's subset,
    // make it the one for the next iteration
    if (sk >= cache[threadIdx.x] && sk < cache[threadIdx.x+1]){
        offset = start;
    }
    __syncthreads();
    // all threads start next iteration with the new subset
}
```

P-ary Search on a sorted list – Implementation (2)

```
// repeat until the #keys in the search range < #threads
while (length > BLOCKSIZE){
    // calculate search range for this thread
    length = length/BLOCKSIZE;
    if (length * BLOCKSIZE < old_length) length += 1;
    old_length = length;
    // why don't we just use floating point?
    start = offset + threadIdx.x * length;
    // cache the boundary keys
    cache[threadIdx.x] = data[start];
    __syncthreads();
    // if the searched key is within this thread's subset,
    // make it the one for the next iteration
    if (sk >= cache[threadIdx.x] && sk < cache[threadIdx.x+1]){
        offset = start;
    }
    __syncthreads();
    // all threads start next iteration with the new subset
}
```

Why?

P-ary Search on a sorted list – Implementation (3)

```
    // last iteration
    start = offset + threadIdx.x;
    if (sk == data[start])
        results[blockIdx.x] = start;
}
```

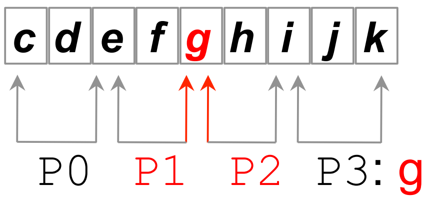
P-ary Search on a sorted list – Implementation (3)

```
    // last iteration
    start = offset + threadIdx.x;
    if (sk == data[start])
        results[blockIdx.x] = start;
}
```

Why don't cache?

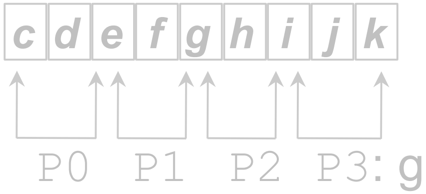
P-ary Search – Analysis

- 100% processor utilization for each query
- Multiple threads can find a result
  - How does this impact correctness?

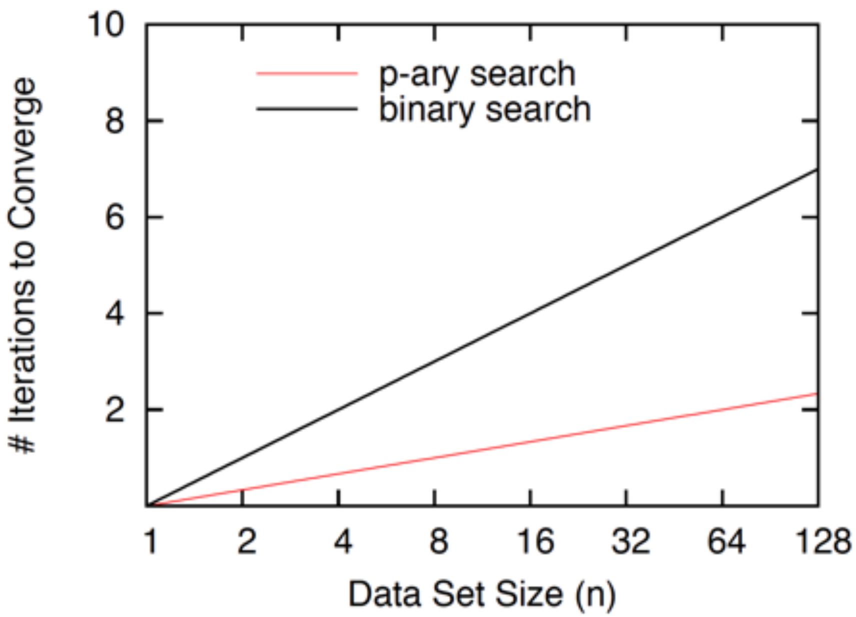


P-ary Search – Analysis

- 100% processor utilization for each query
- Multiple threads can find a result
  - How does this impact correctness?

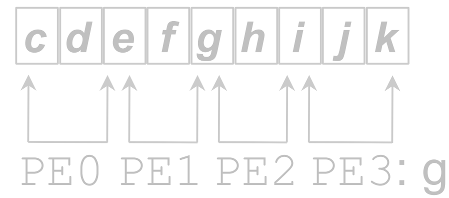


- Convergence depends on #threads
- GTX285: 1 SM, 8 cores(threads) → p=8
- Better Response time
  - $\log_p(n)$ vs $\log_2(n)$



P-ary Search – Analysis

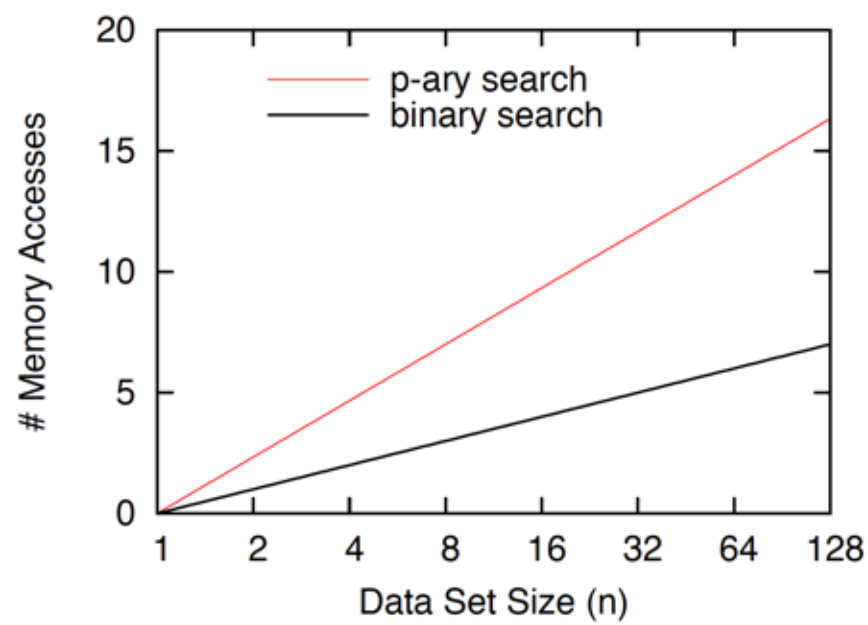
- 100% processor utilization for each query
- Multiple threads can find a result
  - Does not change correctness



- Convergence depends on #threads

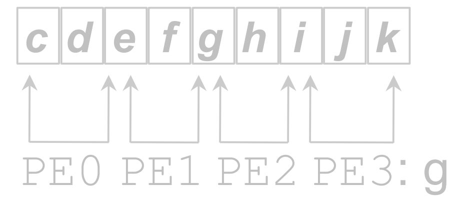
GTX285: 1 SM, 8 cores(threads) → p=8

- Better Response time
  - $\log_p(n)$ vs $\log_2(n)$
- More memory access
  - $(p*2 \text{ per iteration}) * \log_p(n)$
  - Caching
  - $(p-1) * \log_p(n)$ vs. $\log_2(n)$



P-ary Search – Analysis

- 100% processor utilization for each query
- Multiple threads can find a result
  - Does not change correctness

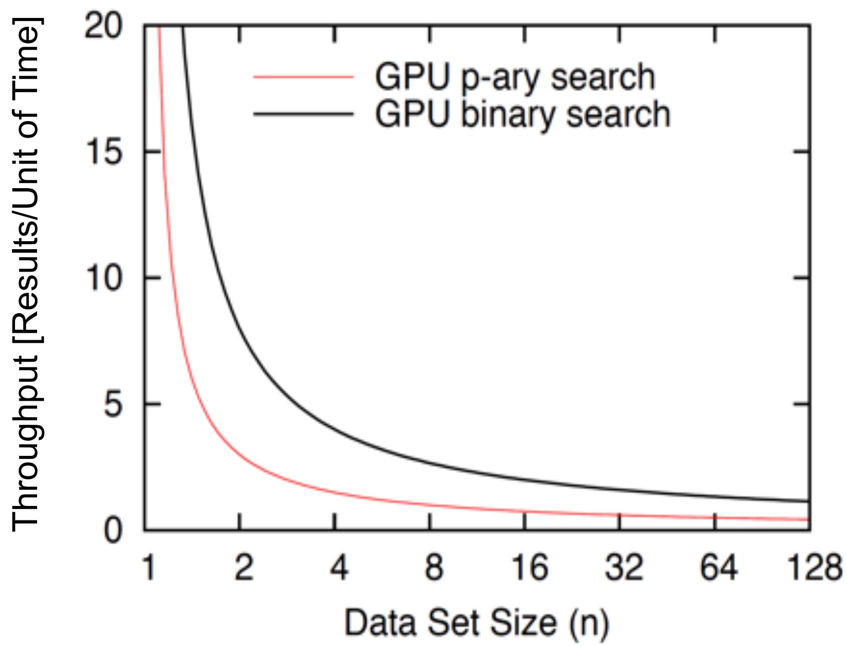


- Convergence depends on #threads

GTX285: 1 SM, 8 cores(threads) → p=8

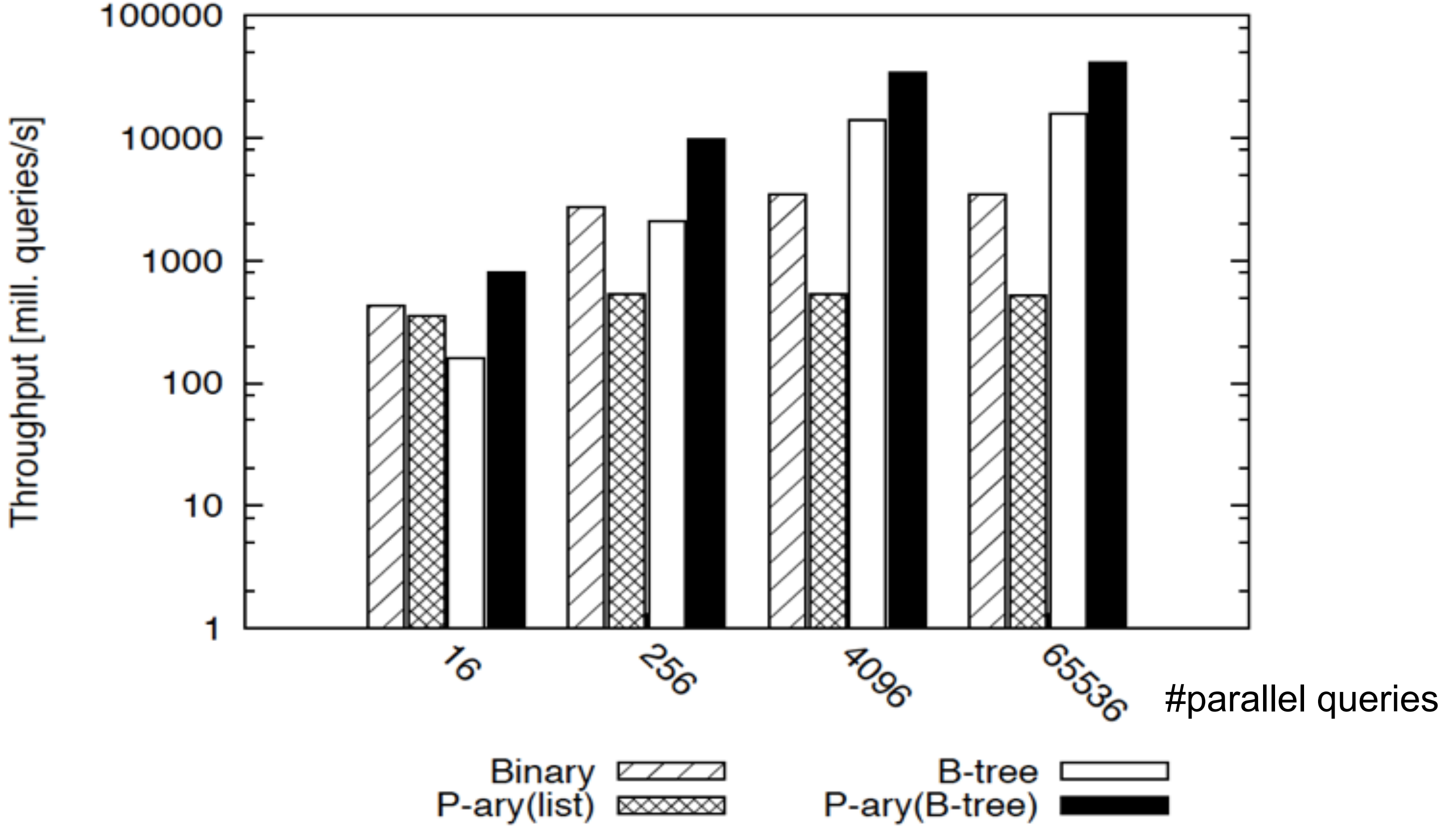
- Better Response time
  - $\log_p(n)$ vs $\log_2(n)$
- More memory access
  - $p*2$ per iteration * $\log_p(n)$
  - Caching

  $(p-1) * \log_p(n)$ vs. $\log_2(n)$
- Lower Throughput
  - $1/\log_p(n)$ vs $p/\log_2(n)$



P-ary Search (GPU) – Throughput

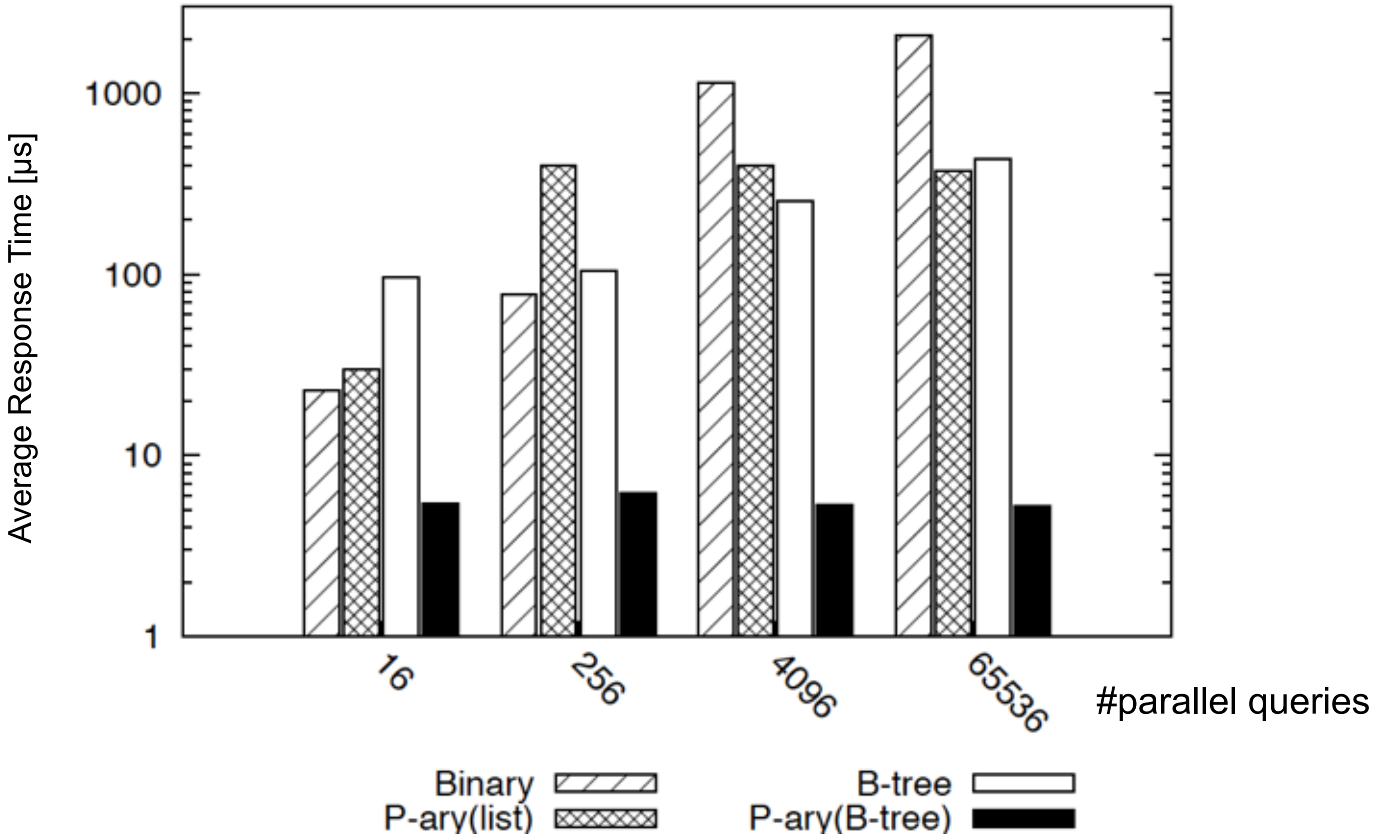
- Superior throughput compared to conventional algorithms



Searching a 512MB data set with 134mill. 4-byte integer entries,
Results for a nVidia GT200b, 1.5GHz, GDDR3 1.2GHz.

P-ary Search (GPU) – Response Time

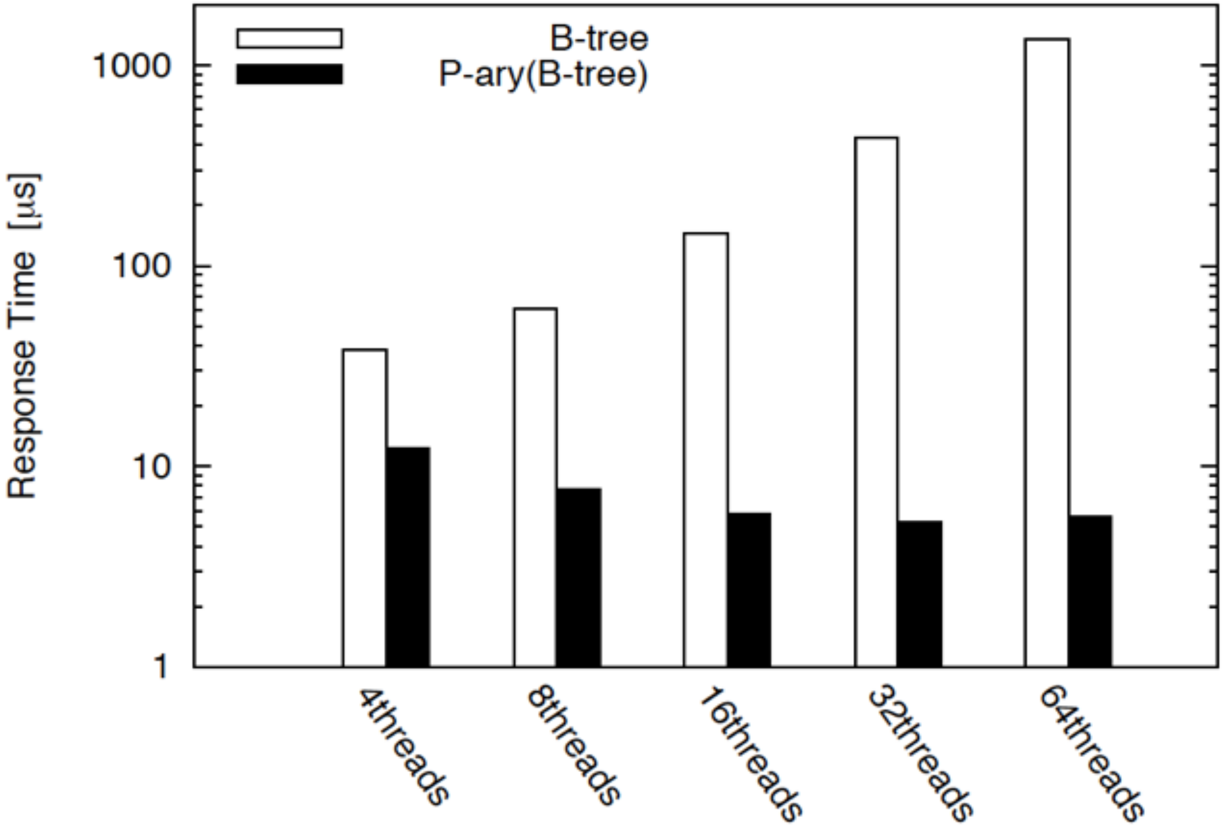
- Response time is workload independent for B-tree implementation



Searching a 512MB data set with 134mill. 4-byte integer entries,
Results for a nVidia GT200b, 1.5GHz, GDDR3 1.2GHz.

P-ary Search (GPU) – Scalability

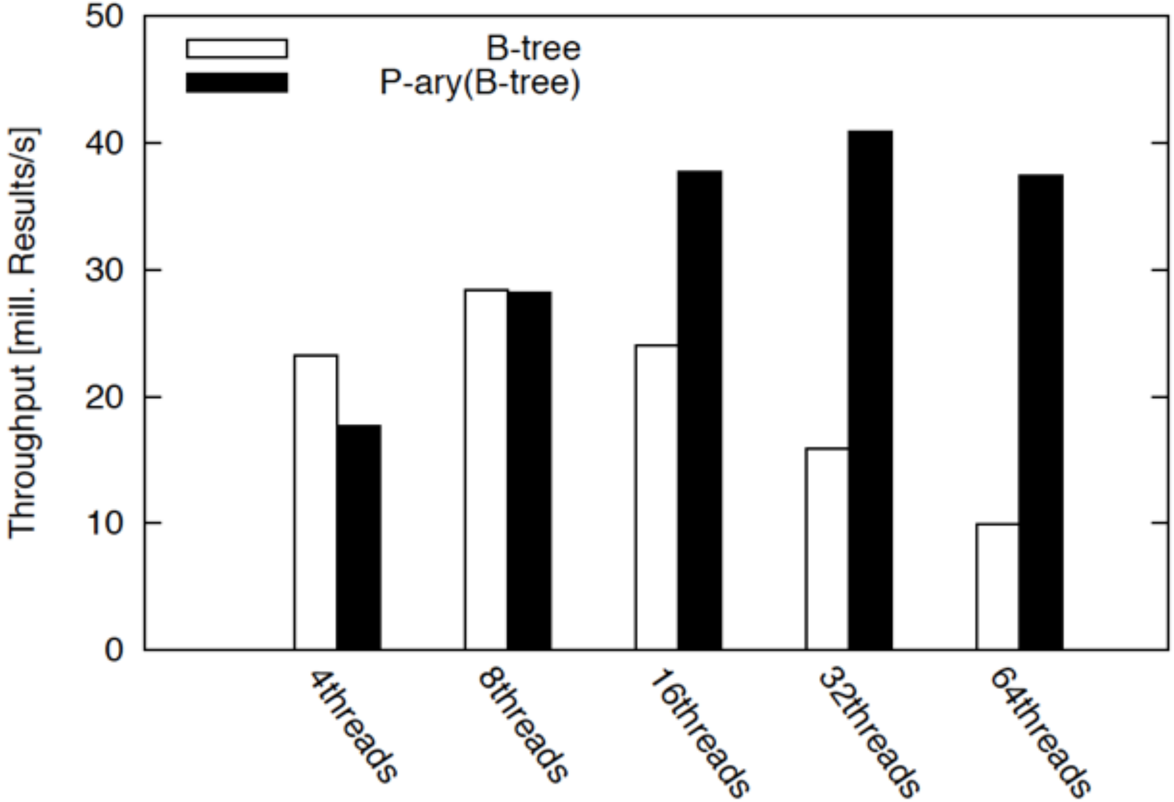
- GPU Implementation using SIMT (SIMD threads)
- Scalability with increasing #threads (P)



64K search queries against a 512MB data set with 134mill. 4-byte integer entries,
Results for a nVidia GT200b, 1.5GHz, GDDR3 1.2GHz.

P-ary Search (GPU) – Scalability

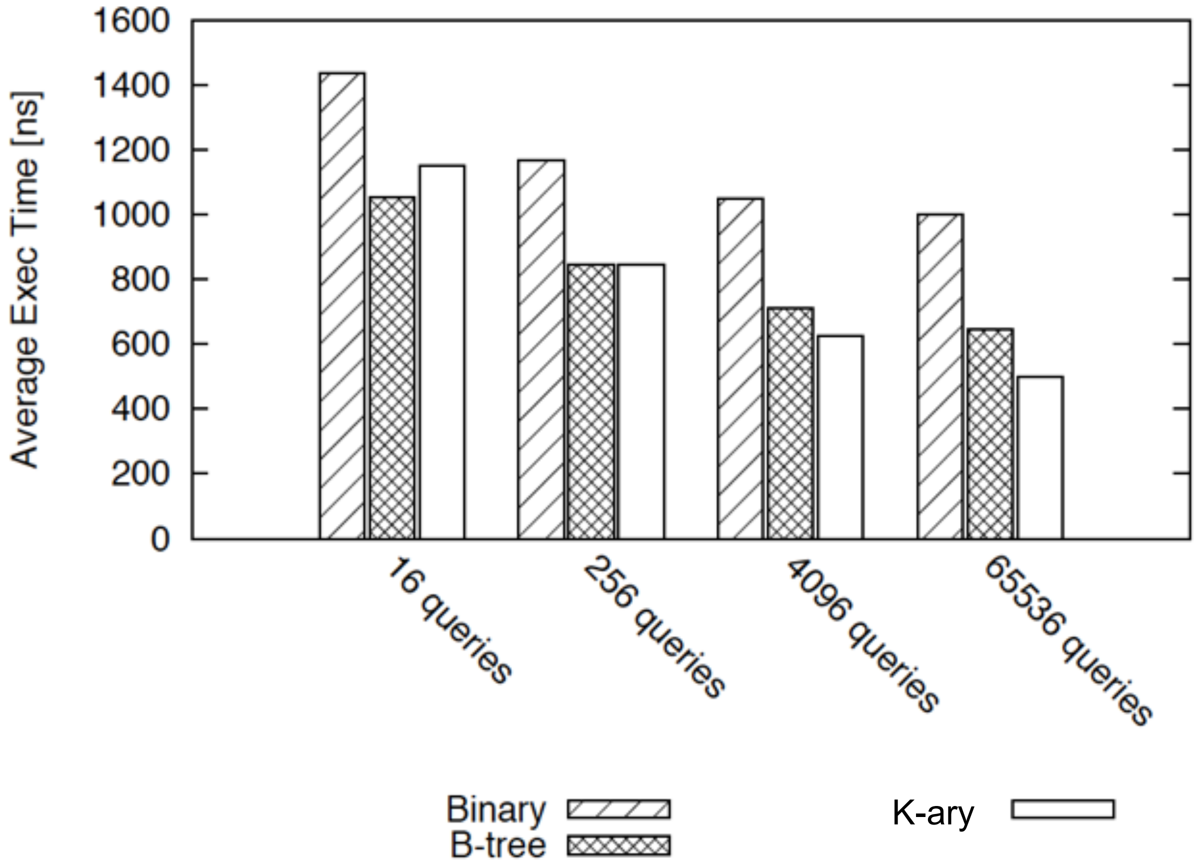
- GPU Implementation using SIMT (SIMD threads)
- Scalability with increasing #threads (P)



64K search queries against a 512MB data set with 134mill. 4-byte integer entries,
Results for a nVidia GT200b, 1.5GHz, GDDR3 1.2GHz.

P-ary Search(CPU) = K-ary Search[1]

- K-ary search is the same algorithm ported to the CPU using SSE vectors (int4) → convergence rate $\log_4(n)$

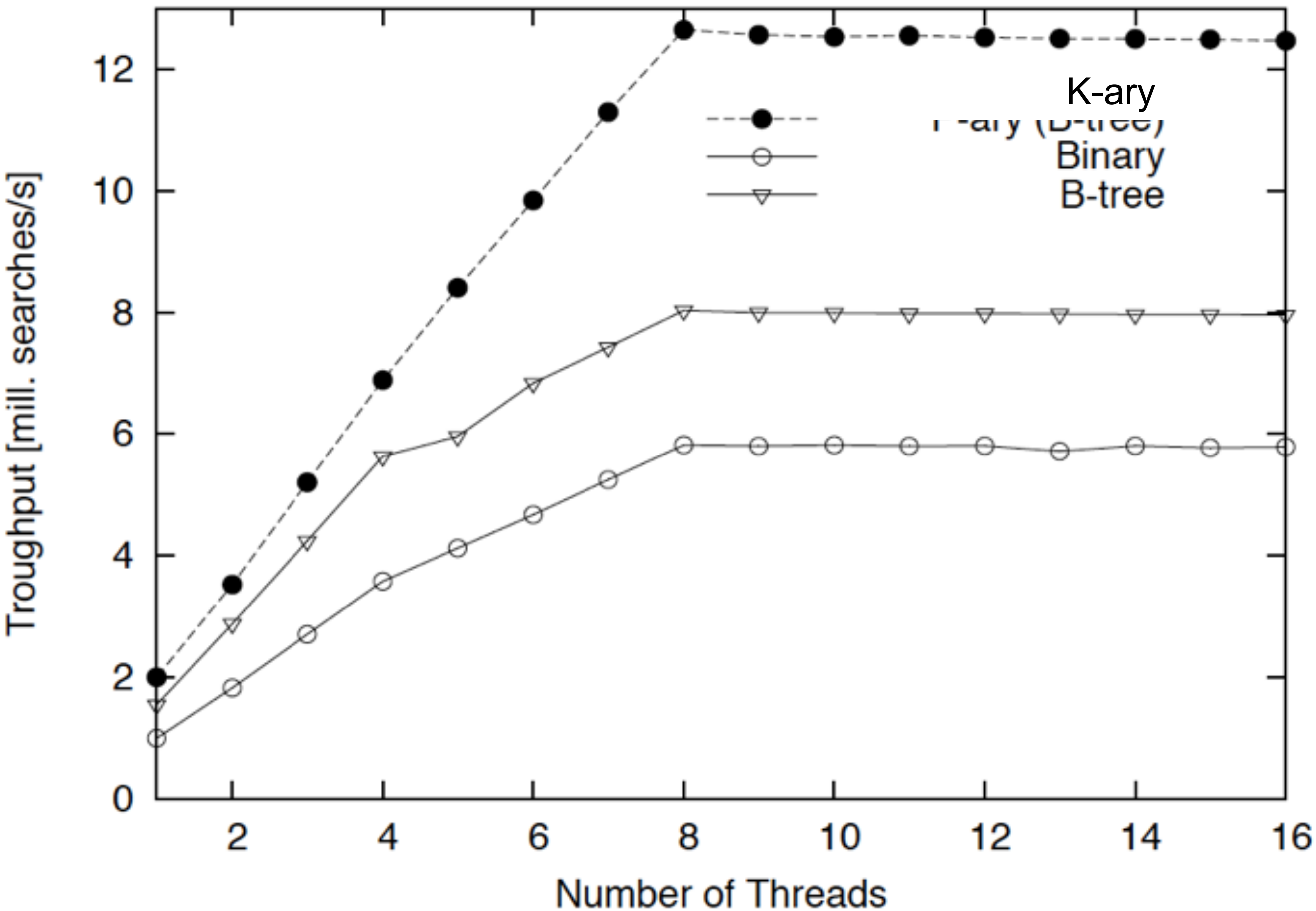


Searching a 512MB data set with 134mill. 4-byte integer entries, Core i7 2.66GHz, DDR3 1666.

[1] B. Schlegel, R. Gemulla, W. Lehner, k-Ary Search on Modern Processors, DaMoN 2000

P-ary Search(CPU) = K-ary Search[1]

- Throughput scales proportional to #threads



64K search queries against a 512MB data set with 134mill. 4-byte integer entries, Core i7 2.66GHz, DDR3 1666.

[1] B. Schlegel, R. Gemulla, W. Lehner, k-Ary Search on Modern Processors, DaMoN 2000

Agenda

- GPU search
  - Reminder: Porting CPU search
  - Back to the drawing board:
    - P-ary search
    - Experimental evaluation
    - Why it works
- Building a GPU based data warehouse solution
  - From a query to operators
  - What to accelerate
  - What are the bottlenecks/limitations
- Maximizing data path efficiency
  - Extremely fast storage solution
  - Storage to host to device
- Putting it all together
  - Prototype demo

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;
```

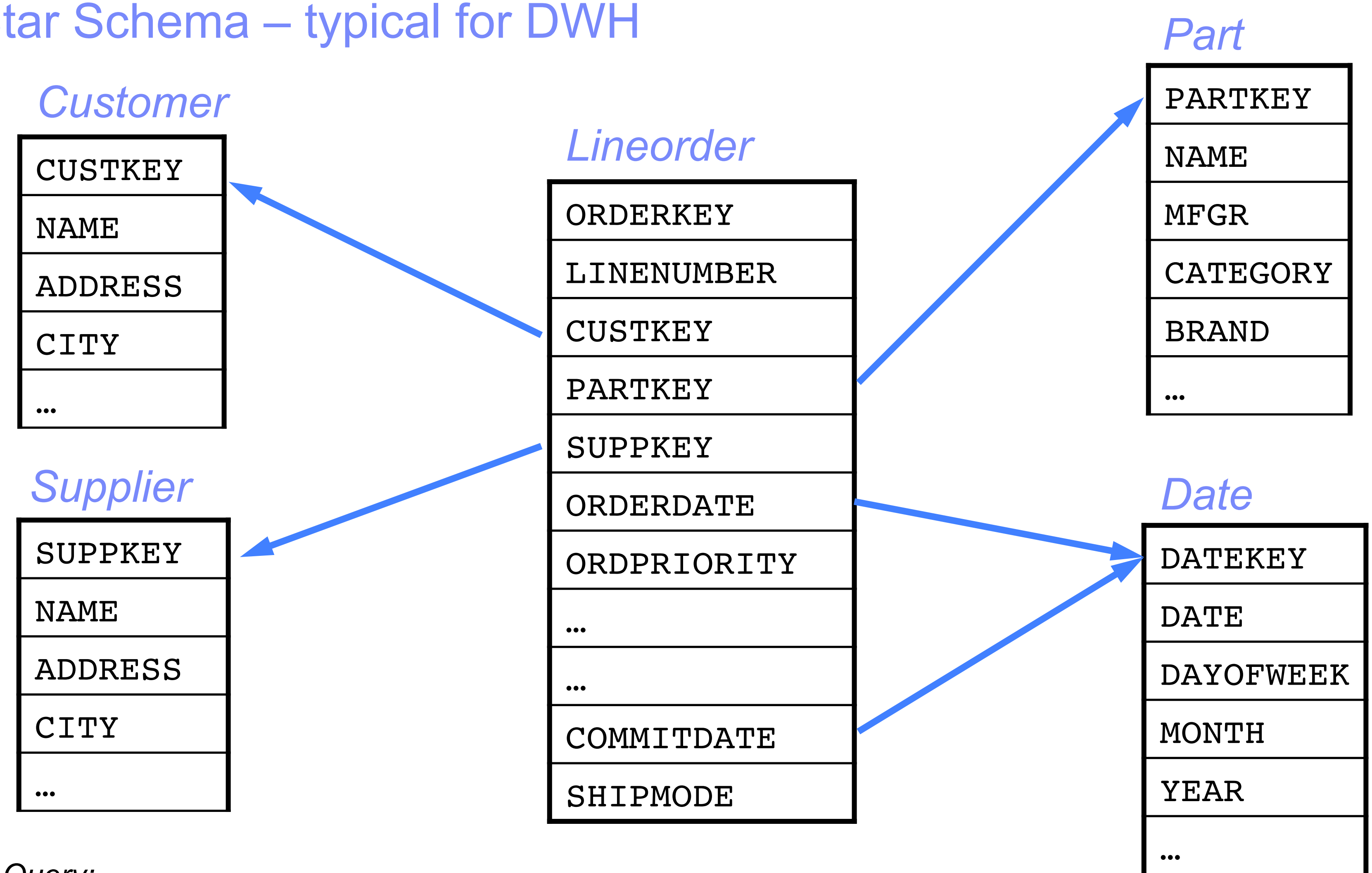
A data warehousing query in multiple languages

- **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**:

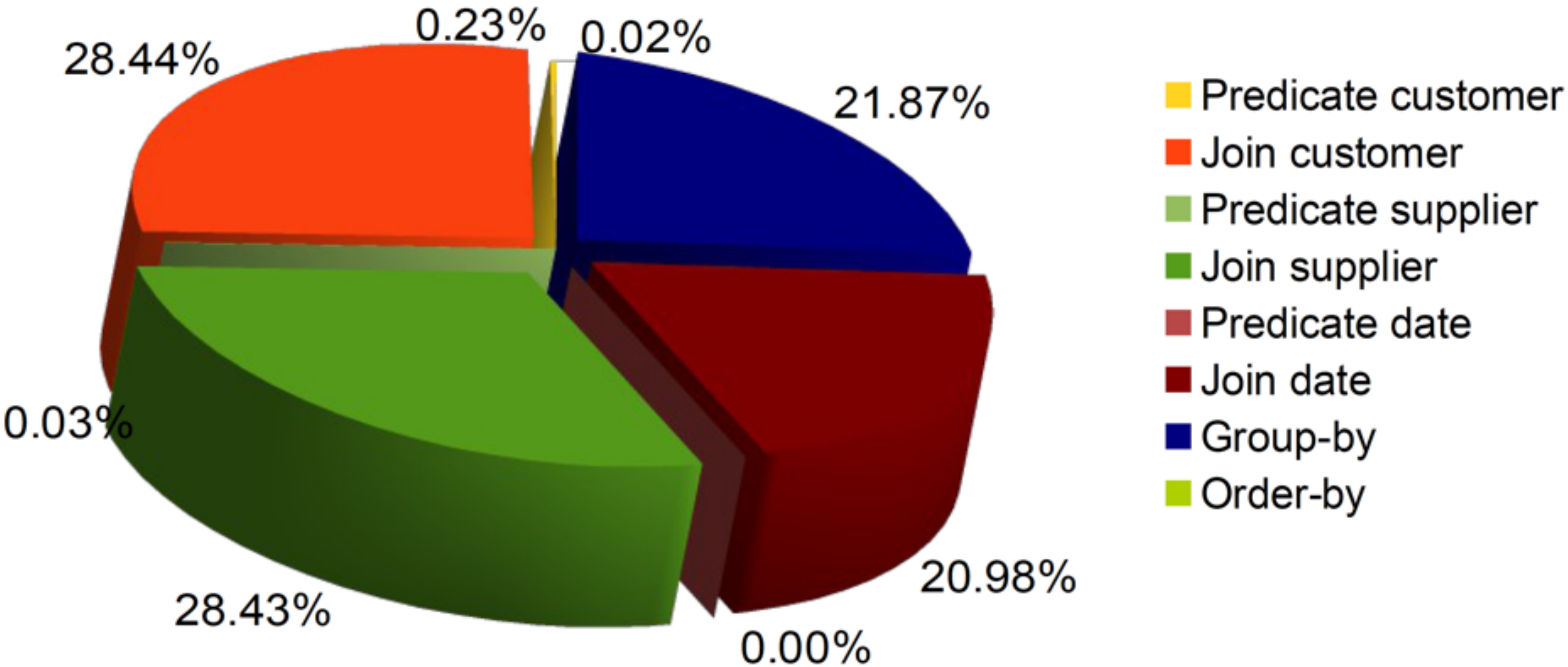
```
SELECT c.city, s.city, d.year, SUM(lo.revenue)
   FROM lineorder lo, customer c, supplier s, date d
   WHERE lo.custkey = c.custkey
      AND lo.suppkey = s.suppkey
      AND lo.orderdate = d.datekey
      AND c.nation = 'UNITED STATES'
      AND s.nation = 'UNITED STATES'
      AND d.year >= 1998 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 (m)

Foreign Key (fk)

$\bowtie$

Customers (living in US)

Key	Zip
11	95014
23	94303
27	95040
39	95134

Primary Key (k)

Payload (p)

=

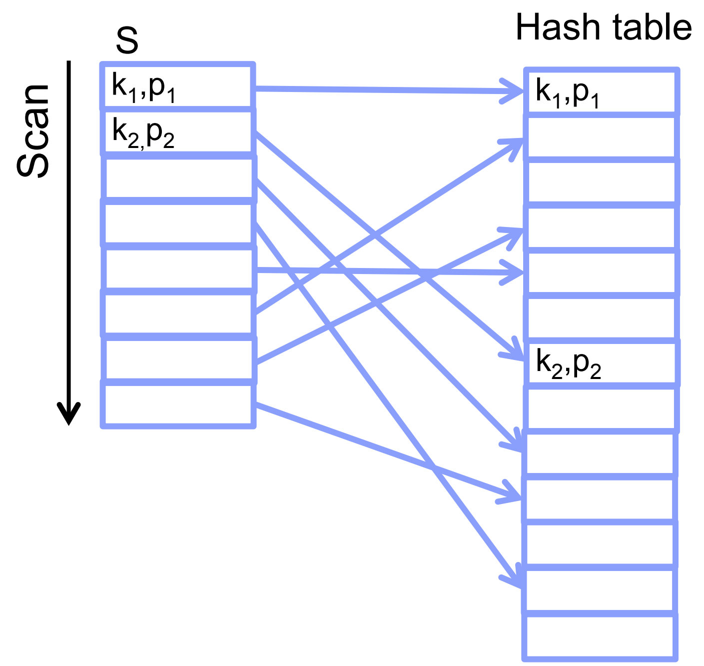
Revenue	Zip
$10.99	94303
$103.00	95014
$84.50	95134
$60.10	95040
$7.60	94303

Join Results

Hash Join

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

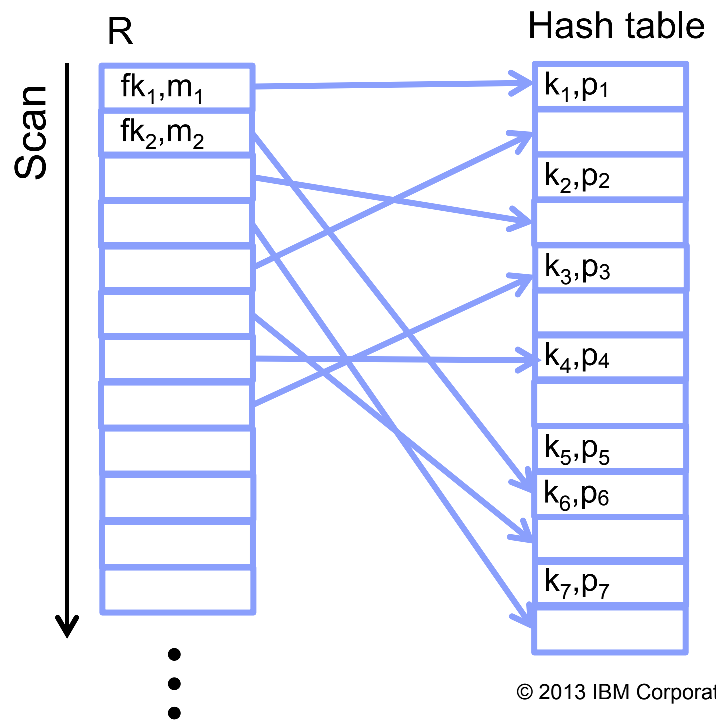
1. Build a hash table
   - Scan S and compute a location (hash) based on a unique (primary) key
   - Insert primary key **k** with payload **p** into the hash table
   - If the location is occupied pick the next free one (open addressing)



Hash Join

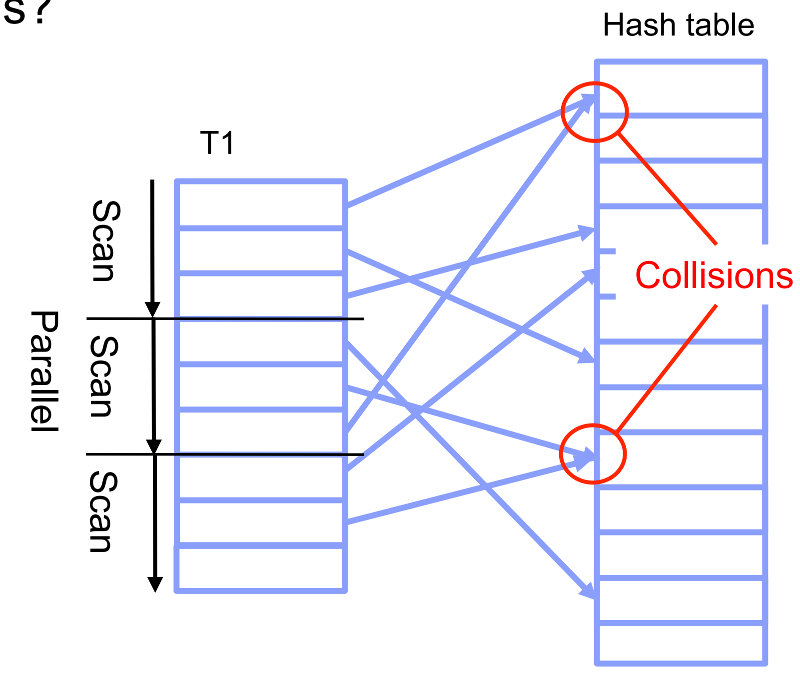
Join two tables (|S| < |R|) in 2 steps

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**)



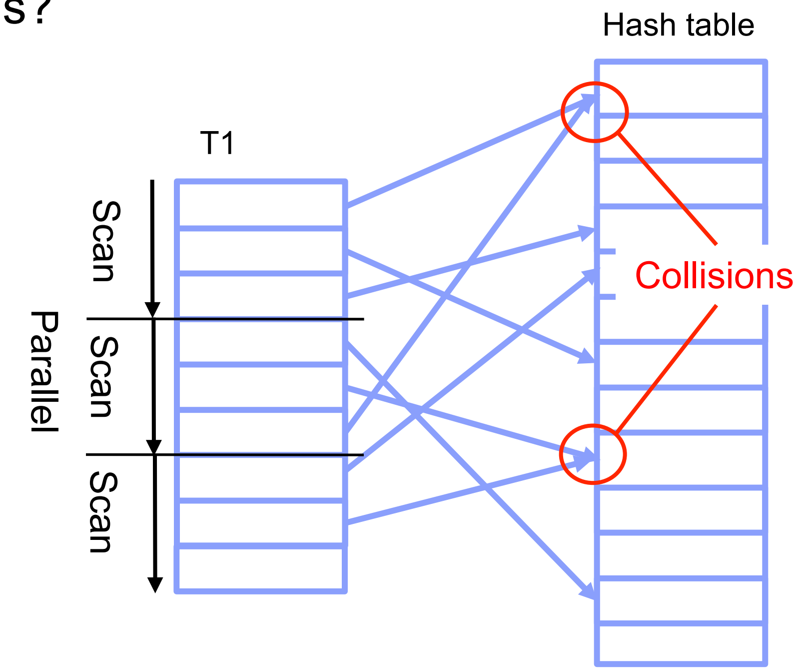
Parallel Hash Join

- Multiple threads scan T1 and attempt to insert <key,rid> pairs into the hash table
    - How to handle hash collisions?



Parallel Hash Join

- Multiple threads scan T1 and attempt to insert <key,rid> pairs into the hash table
  - How to handle hash collisions?



  - Is this a good access pattern?
- Parallel probe is trivial as it requires read-only access

IBM

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 (fk)

Probe Inputs

⋈

Hash Table (HT)

Customers (living in US)

Key	Zip
11	95014
23	94303
27	95040
39	95134

Primary Key (k)   Payload (p)

=

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

Join Results

How fast are hash probes ?

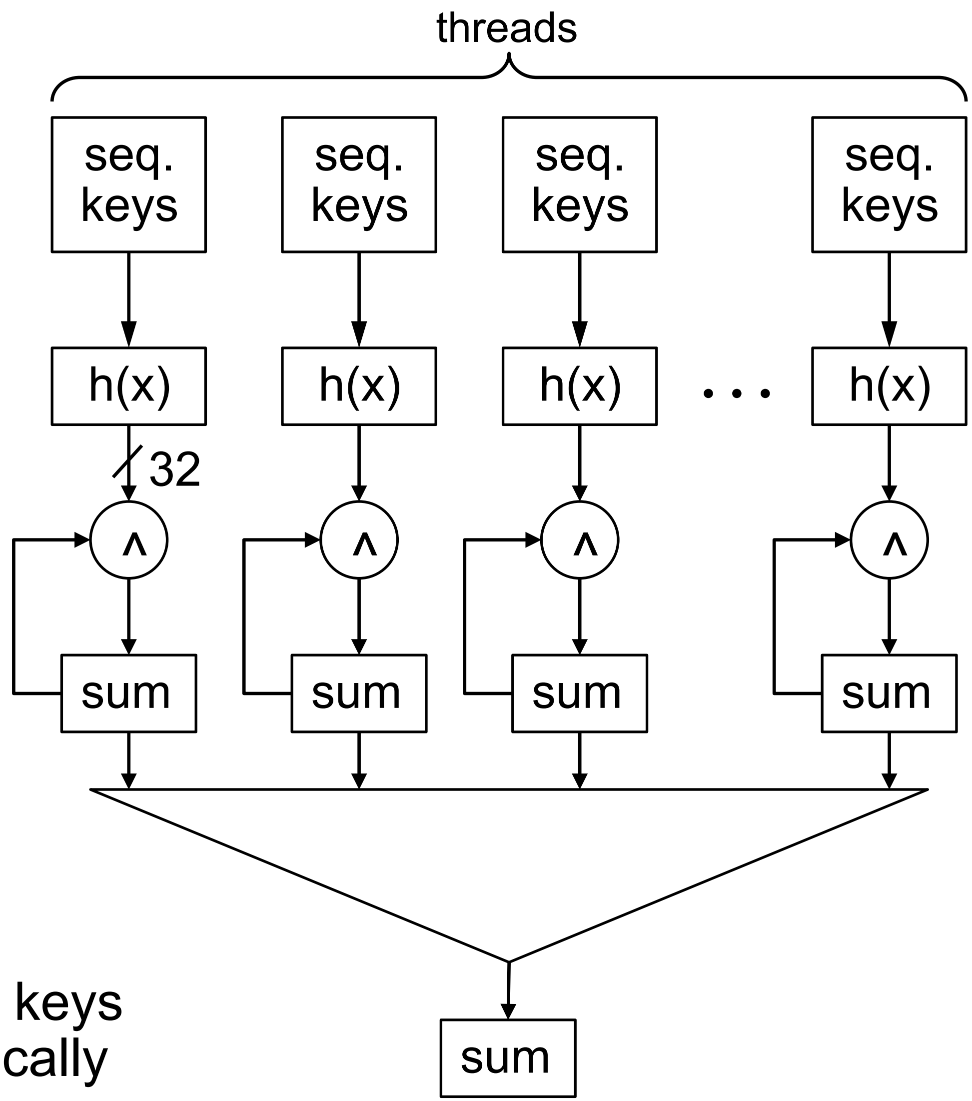
- Computation
- Data (memory) access

Computing Hash Functions on GTX580 – Compute only *

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

* More details on hashing: "Let your GPU do the heavy lifting in your data warehouse" GTC'13

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 – 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
- 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 – 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

- Data (memory) access on

vs.

	GPU (GTX580)	CPU (i7-2600)	
Peak memory bandwidth [spec] 1)	179 GB/s	21 GB/s	Upper bound for:
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

GPU Hash Join Implementation (Summary)

1. Pin input tables
   – Required for Build and Probe table, done bye the CPU
   – Only pinned CPU memory is accessible by the GPU
   – "GPU direct" now allows to read directly from network/storage devices …

2. Allocate memory for HT
   – CPU handles memory allocation of GPU memory
   – This is supposed to change with the next GPU generation …

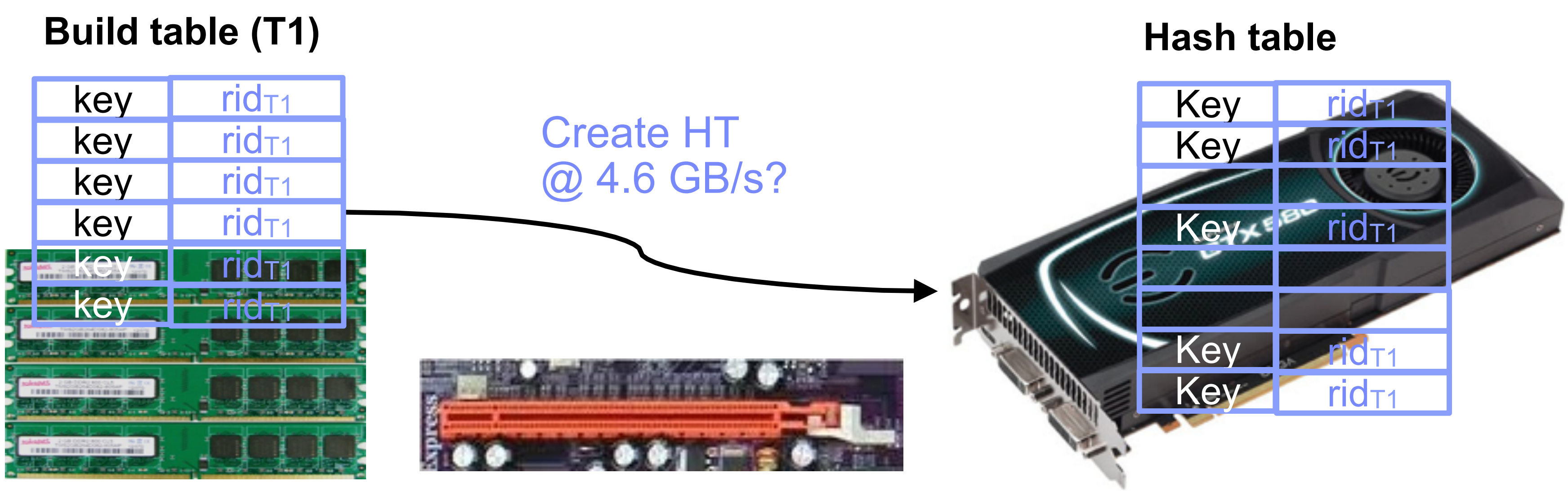
3. Build HT
   – GPU reads build table (T1) sequentially from pinned CPU memory
   – GPU creates HT (open addressing) in GPU memory
   – Collisions are handled using atomic compare-and-swap

4. Probe HT
   – GPU reads probe table (T2) sequentially from CPU memory
   – GPU probes hash table (in GPU memory) and writes results to CPU memory

5. Cleanup
   – free GPU memory
   – Unpin input tables

GPU Hash Join – Build HT

- GPU reads build table (T1) sequentially from pinned CPU memory
- GPU creates HT (open addressing) in GPU memory
- Collisions are handled using atomic compare-and-swap



Build HT – Memory Management & Function call

```
// register input table
// 32-bit key + 32-bit rid are stored as a single 64-bit value
unsigned long long int* buildT;
cudaHostRegister(T1,num_tuples*2*sizeof(int),cudaHostRegisterMapped);
cudaHostGetDevicePointer(&buildT,T1,0);

// make space for hash table
unsigned long long int* HT;
int HT_rows = 4 * num_tuples;
cudaMalloc(&HT, HT_rows * sizeof(int));
cudaMemSet(HT, 0, HT_rows * sizeof(int));

// call device function
dim3 Dg = dim3(16,0,0);
dim3 Db = dim3(512,0,0);
gpuCreateHashtable <<< Dg, Db >>>(builtT, num_tuples,
                                  HT, HT_rows);
```

Build HT – Local variables

```
__global__ static void gpuCreateHashtable(unsigned long long int *buildT,
                                          int num_tuples,
                                          unsigned long long int *HT,
                                          int HT_rows){

int insert_loc;                        // insert location for tuple
int tupleID;                           // iterator for the build table
int cas_result;                        // HT was initialized with 0, i.e.
                                       // if insert was successful then
                                       // cas_result = 0
int hash_mask = HT_rows - 1;           // LSB hash mask (for powers of 2!)
unsigned long long int buildT_cache;   // register cache for a build table
int key;                               // key extracted from build table
```

Build HT – Outline

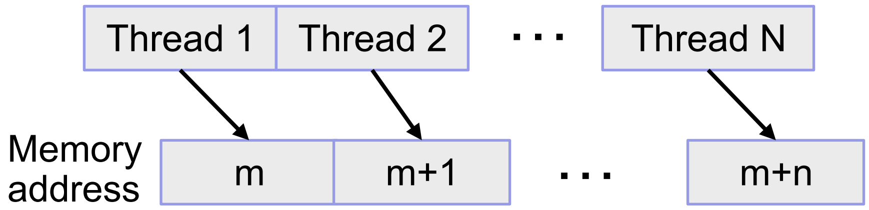
```
// Iterate through the tuples of the build table and insert them into the
// hash table
for (tupleID = blockIdx.x*blockDim.x+threadIdx.x;
     tupleID < num_tuples;
     tupleID += blockDim.x*gridDim.x){
  /* 1) Cache the build table entry (key,rid) in a register
   * 2) Apply hash function (LSB) to to key to determine insert position
   * 3) Starting from the insert position, scan for the next available
   *    slot
   * 4) Atomically insert the entry into the hash table
   */
```

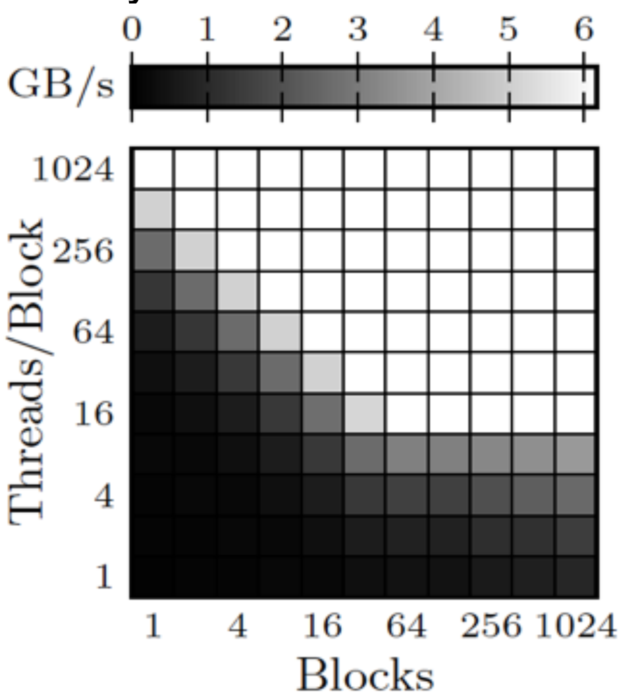
Build HT – Memory Access

Read build table from host memory

```
for (tupleID = blockIdx.x*blockDim.x+threadIdx.x;
     tupleID < num_tuples;
     tupleID += blockDim.x*gridDim.x){
buildT_cache = buildT[tupleID];
```

- Ideal memory access pattern is coalesced memory access
  – Threads of a block/warp access consecutive memory addresses





  – Same applies to ZCA to host(main) memory
    • Coalesced access up to 6.2 GB/s
    • Random = faux pas !

Build HT – Core Loop

```
for (tupleID = blockIdx.x*blockDim.x+threadIdx.x;
     tupleID < num_tuples;
     tupleID += blockDim.x*gridDim.x)
{
   cas_result = 42;  // answer to everything ;-)
   // 1) Cache the build table entry (key,rid) in a register
   buildT_cache = buildT[tupleID];
```

Build HT – Core Loop

```
for (tupleID = blockIdx.x*blockDim.x+threadIdx.x;
     tupleID < num_tuples;
     tupleID += blockDim.x*gridDim.x)
{
   cas_result = 42;  // answer to everything ;-)
   // 1) Cache the build table entry (key,rid) in a register
   buildT_cache = buildT[tupleID];

   // 2) Apply LSB hash to key to determine insert position
   //    Little endian: <key,rid> becomes <rid,key> in the register
   key = (int)(buildT_cache & 0xFFFFFFFF);  // key in the lower half
   insert_loc = key & hash_mask;
```

Build HT – Core Loop

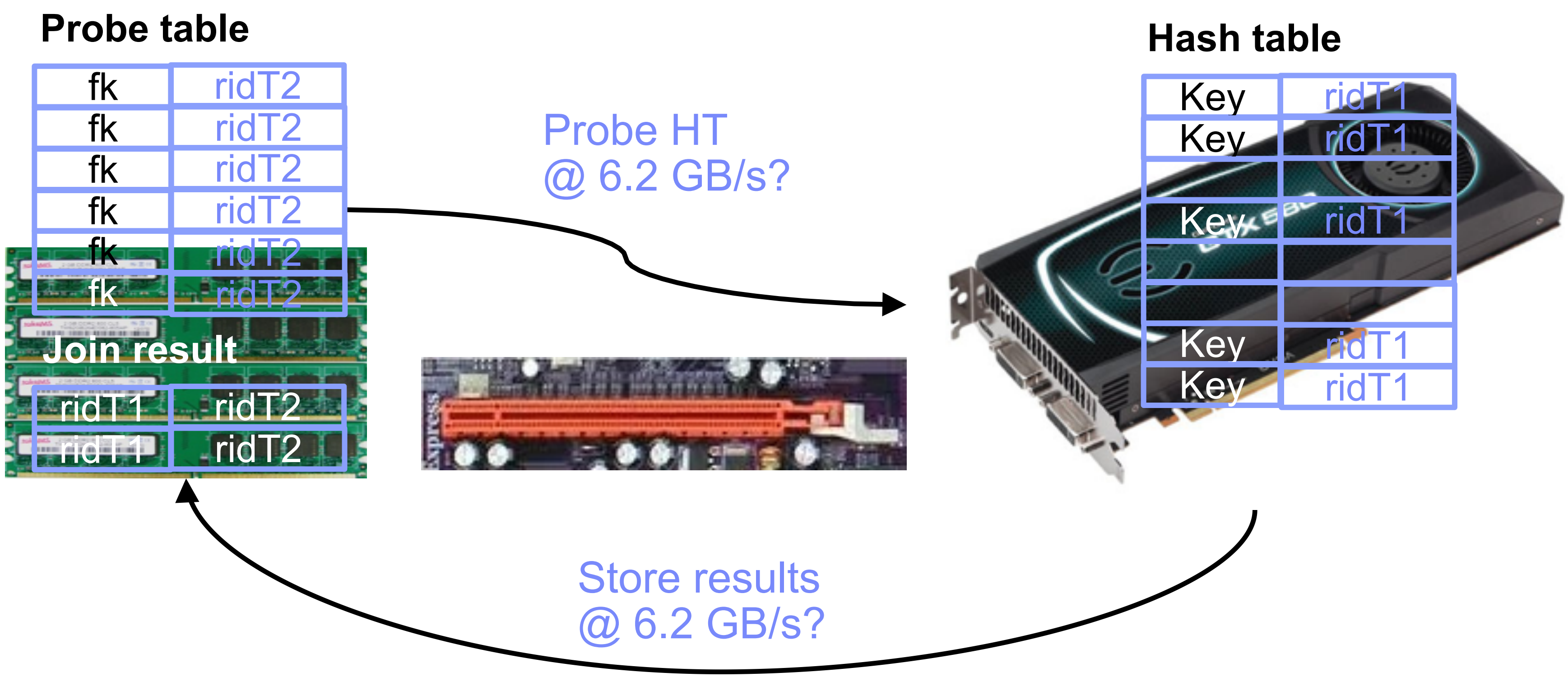
```
for (tupleID = blockIdx.x*blockDim.x+threadIdx.x;
     tupleID < num_tuples;
     tupleID += blockDim.x*gridDim.x)
{
   cas_result = 42;  // answer to everything ;-)
   // 1) Cache the build table entry (key,rid) in a register
   buildT_cache = buildT[tupleID];

   // 2) Apply LSB hash to key to determine insert position
   //    Little endian: <key,rid> becomes <rid,key> in the register
   key = (int)(buildT_cache & 0xFFFFFFFF);  // key in the lower half
   insert_loc = key & hash_mask;

   // 3) From insert position scan for the next available slot (0) to
   //    avoid repeated atomic compare-and-swap ($$$)
   while (HT[insert_loc] != 0)
      insert_loc = ++insert_loc & hash_mask;
```

Build HT – Core Loop

```
// 1) Cache the build table entry (key,rid) in a register
buildT_cache = buildT[tupleID];

// 2) Apply LSB hash to key to determine insert position
//    Little endian: <key,rid> becomes <rid,key> in the register
key = (int)(buildT_cache & 0xFFFFFFFF);  // key in the lower half
insert_loc = key & hash_mask;

// 3) From insert position scan for the next available slot (0) to
//    avoid repeated atomic compare-and-swap ($$$)
while (HT[insert_loc] != 0)
   insert_loc = ++insert_loc & hash_mask;

// 4) Atomically insert entry into the hash table
while(cas_result != 0){
   cas_result = atomicCAS(&(HT[insert_loc]), 0, buildT_cache);
   insert_loc = ++insert_loc & hash_mask;
}
```

GPU Hash Join – Probe HT

- GPU reads probe table (T2) sequentially from CPU memory
- GPU probes hash table (in GPU memory) and writes results to CPU memory



Probe HT – Memory Management & Function call

```
// register input table
// 32-bit key + 32-bit rid are stored as a single 64-bit value
unsigned long long int* probeT;
cudaHostRegister(T2,num_tuples*2*sizeof(int),cudaHostRegisterMapped);
cudaHostGetDevicePointer(&probeT,T2,0);

// make space for results
unsigned long long int* resG;
cudaHostAlloc(&resG, 2 * num_tuples * sizeof(int));

// result  index
__device__ int gpu_result_index;
cudaMemcpyToSymbol(gpu_result_index, &null, sizeof(int));

// call device function
dim3 Dg = dim3(16,0,0);
dim3 Db = dim3(512,0,0);
gpuProbe <<< Dg, Db >>>(probeT, HT, resG, num_tuples, HT_rows);
```

Probe HT – Local Variables

```
__global__ static void gpuProbe(unsigned long long int* probeT,
                                unsigned long long int* HT,
                                unsigned long long int* resG,
                                int probeT_rows, int HT_rows)
{
    int probeT_key;                 // the probe table key used for a probe
    int HT_idx;                     // hash table location the probe lead to
    int HT_key;                     // the key found at the hash table
                                    // location of hashtable_idx
    int tupleID;                    // iterator for the probe table
    int hash_mask = HT_rows - 1;    // LSB hash mask
    int result_insert_position;     // index to the result, shared by ALL
                                    // threads (atomic insert)
    unsigned long long int probeT_cache;   // register cache for probe table
    unsigned long long int HT_cache;       // register cache for hash table
```

Probe HT – Outline

```
// Iterate through the tuples of the probe table and
   for (tupleID=blockIdx.x*blockDim.x+threadIdx.x;
        tupleID < probeT_rows;
        tupleID+=blockDim.x*gridDim.x) {
      /* 1) Cache the fact table entry (key,rid) in a register & extract
       *    the fact table key
       * 2) Apply the hash function to the key to determine the location
       *    in the hash table
       * 3) Probe the hash table and cache the entry (key,rid) in a
       *    register
       * 4) Scan the hash table for more matching keys until we hit an
       *    empty (0) position
       */
```

Probe HT – Core Loop

```
for (tupleID=blockIdx.x*blockDim.x+threadIdx.x;
     tupleID < probeT_rows;
     tupleID+=blockDim.x*gridDim.x){
   // 1) Cache the fact table entry (key,rid) in a register
   probeT_cache = probeT[tupleID];
   //    Extract the fact table key
   //    Little endian: <key,rid> becomes <rid,key> in the register
   probeT_key = (int)(probeT_cache & 0xFFFFFFFF); // key in lower half
```

Probe HT – Core Loop

```
for (tupleID=blockIdx.x*blockDim.x+threadIdx.x;
     tupleID < probeT_rows;
     tupleID+=blockDim.x*gridDim.x){
   // 1) Cache the fact table entry (key,rid) in a register
   probeT_cache = probeT[tupleID];
   //    Extract the fact table key
   //    Little endian: <key,rid> becomes <rid,key> in the register
   probeT_key = (int)(probeT_cache & 0xFFFFFFFF); // key in lower half

  // 2) Hash the key to determine the location in the hash table
   HT_idx = probT_key & hash_mask;
```

Probe HT – Core Loop

```
for (tupleID=blockIdx.x*blockDim.x+threadIdx.x;
     tupleID < probeT_rows;
     tupleID+=blockDim.x*gridDim.x){
   // 1) Cache the fact table entry (key,rid) in a register
   probeT_cache = probeT[tupleID];
   //    Extract the fact table key
   //    Little endian: <key,rid> becomes <rid,key> in the register
   probeT_key = (int)(probeT_cache & 0xFFFFFFFF); // key in lower half

  // 2) Hash the key to determine the location in the hash table
   HT_idx = probT_key & hash_mask;

   // 3) Probe the hash table and cache the entry (key,rid) in a register
   HT_cache = HT[HT_idx];
```

Probe HT – Core Loop

```
    /* Scan open addressing hash table until we hit an empty(0) slot
     * 4.1) If keys match insert rids from the probe and hash table into
     *      the global result set
     * 4.2) Cache the next hash table entry and extract the key
     */
    while (HT_cache != 0) {
       HT_key = (int)(HT_cache & 0xFFFFFFFF);
       if (probeT_key == HT_key) {
          // determine position in global result set
          result_insert_position = atomicAdd(&gpu_result_index, 1);
          // insert result=<rid,rid>
          // rids are both in the upper half of the register caches,
          // so we need to shift one of them (hashtable cache) down
          resG[result_insert_position] = (probeT_cache & 0xFFFFFFFF00000000)
                                        | (HT_cache >> 32);
       }
       HT_idx = ++HT_idx & hash_mask;
       HT_cache = HT[HT_idx];
}
```

Retrieving result count & cleanup

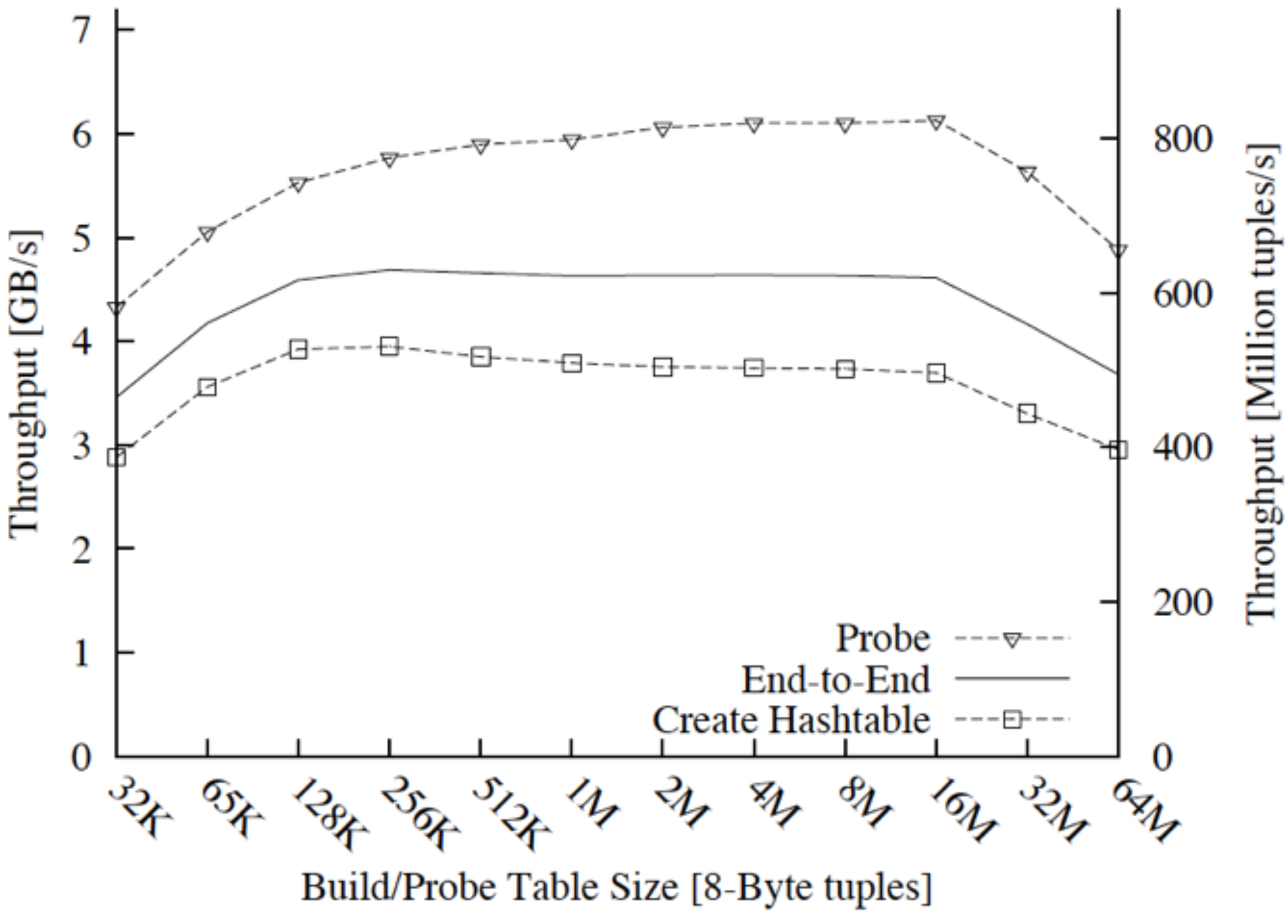
```
...
// After GPU function completes
cudaDeviceSynchronize();
cudaMemcpyFromSymbol(rescount, gpu_result_index, sizeof(int));

// clean up memory
cudaHostUnregister(T1);
cudaHostUnregister(T1);
cudaFree(HT);
...
```

Throughput

- Join 2 equal size tables (16M rows) of 32-bit <key,row-ID> pairs (4+4 Byte) *worst case*
  - Uniformly distributed randomly generated keys
  - 3% of the keys in the probe table have a match in the build table *few writes*
  - Measuring End-to-End throughput, i.e. input tables & results in host memory

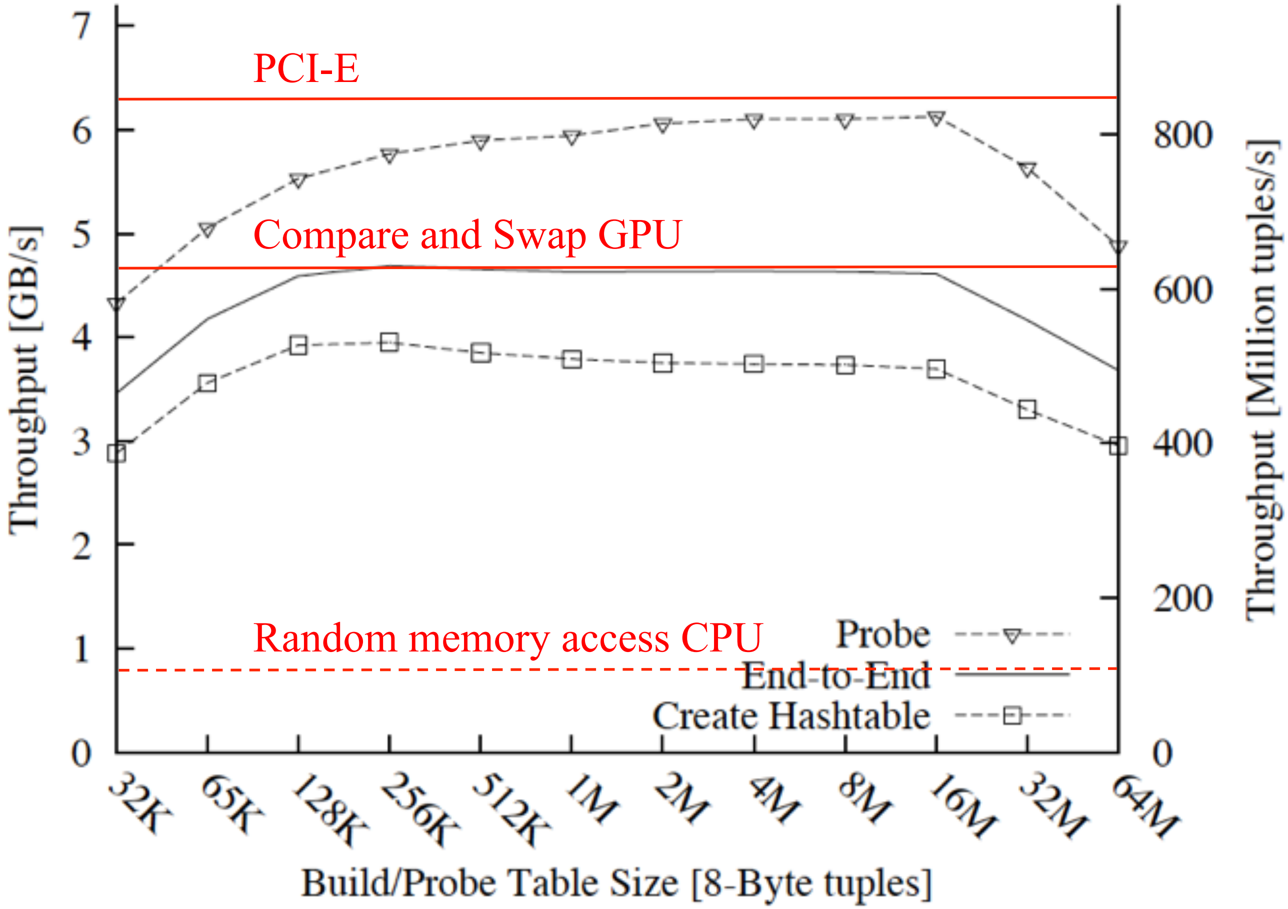
Throughput

- Join 2 equal size tables (16M rows) of 32-bit <key,row-ID> pairs (4+4 Byte) *worst case*
  - Uniformly distributed randomly generated keys
  - 3% of the keys in the probe table have a match in the build table *few writes*
  - Measuring End-to-End throughput, i.e. input tables & results in host memory



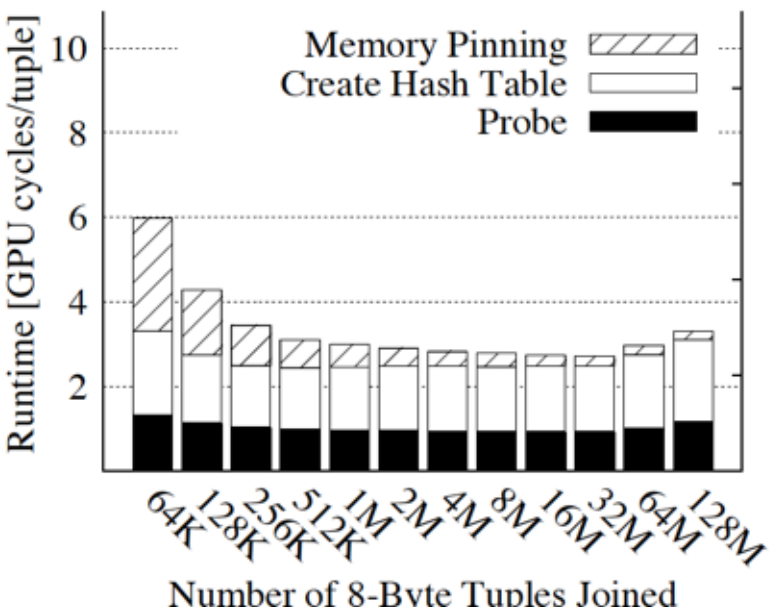
Throughput

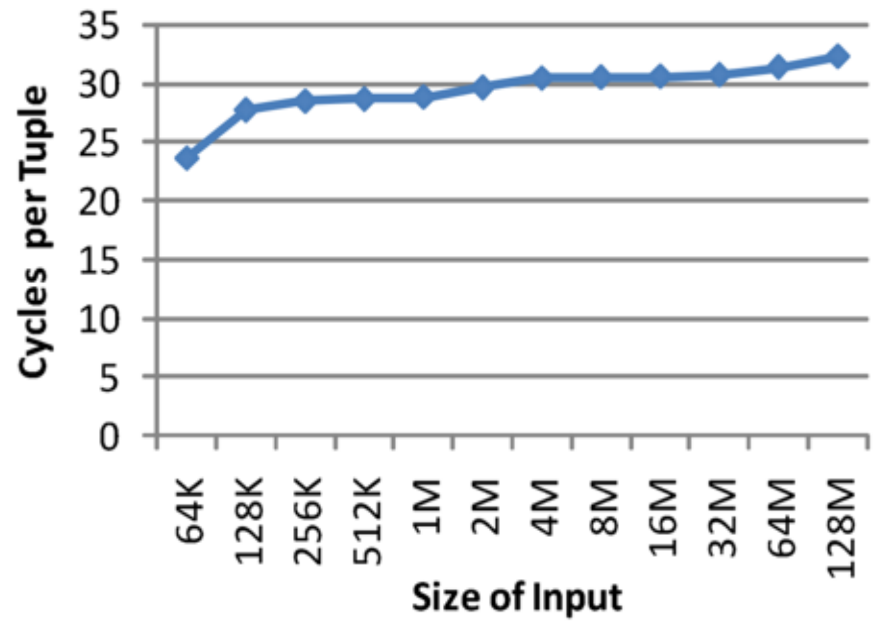
- Join 2 equal size tables (16M rows) of 32-bit <key,row-ID> pairs (4+4 Byte) *worst case*
  - Uniformly distributed randomly generated keys
  - 3% of the keys in the probe table have a match in the build table *few writes*
  - Measuring End-to-End throughput, i.e. input tables & results in host memory



- Join 2 equal size tables (512K to 128M) of 32-bit <key,row-ID> pairs (4 + 4 Byte)
- Uniformly distributed randomly generated keys
- 3% of the probe keys have a match in the build table
- CPU implementation does not materialize results

GPU vs. CPU

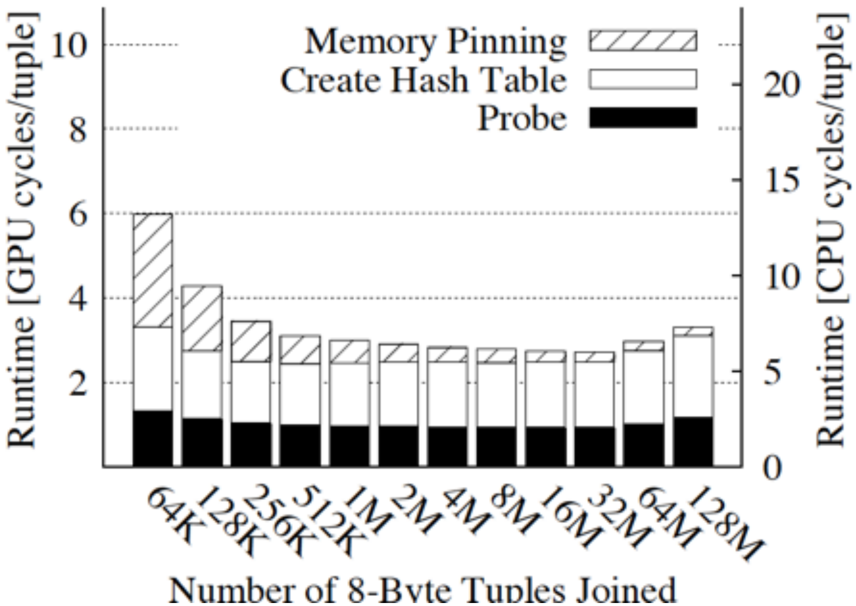


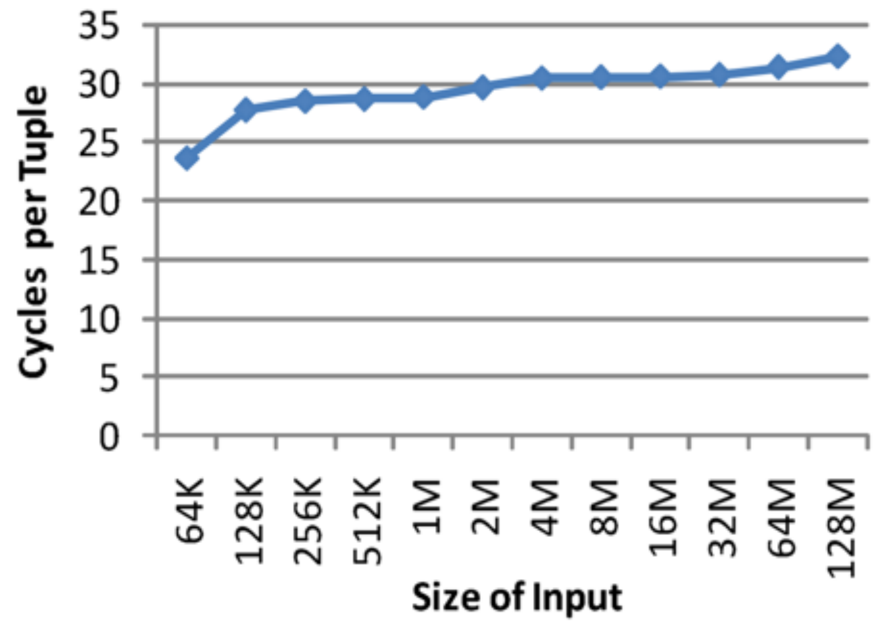


[2] C. Kim, E. Sedlar, J. Chhugani, T. Kaldewey, A. Nguyen, A. Di Blas, V. Lee, N. Satish, P. Dubey. Sort vs. Hash revisited: fast join implementation on modern multi-core CPUs. VLDB 2009

- Join 2 equal size tables (512K to 128M) of 32-bit <key,row-ID> pairs (4 + 4 Byte)
- Uniformly distributed randomly generated keys
- 3% of the probe keys have a match in the build table
- CPU implementation does not materialize results
- Cycles/tuple not a meaningful metric
  - depends on processor frequency, tuple size, …

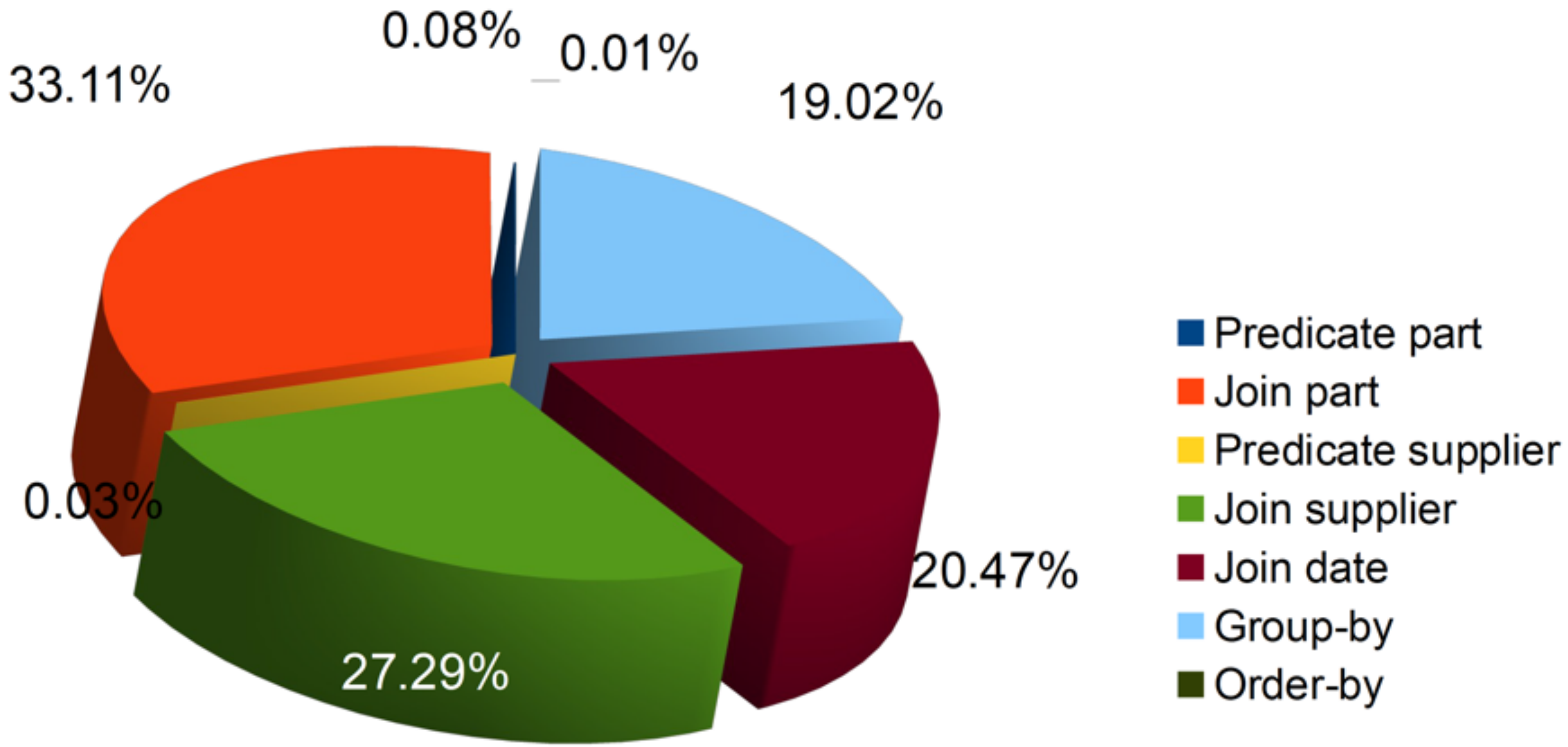
GPU vs. CPU





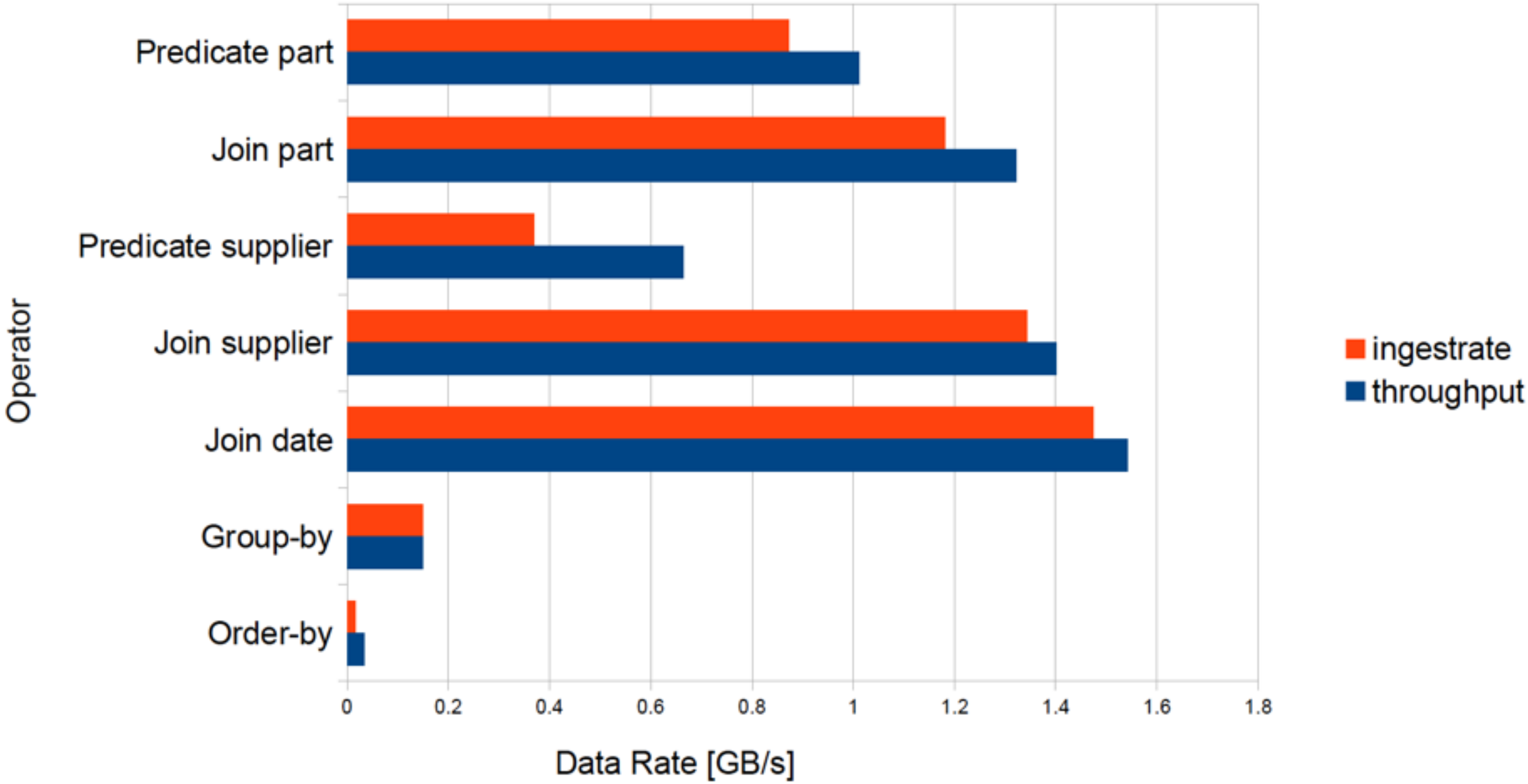
[2] C. Kim, E. Sedlar, J. Chhugani, T. Kaldewey, A. Nguyen, A. Di Blas, V. Lee, N. Satish, P. Dubey. Sort vs. Hash revisited: fast join implementation on modern multi-core CPUs. VLDB 2009

Where does time go?



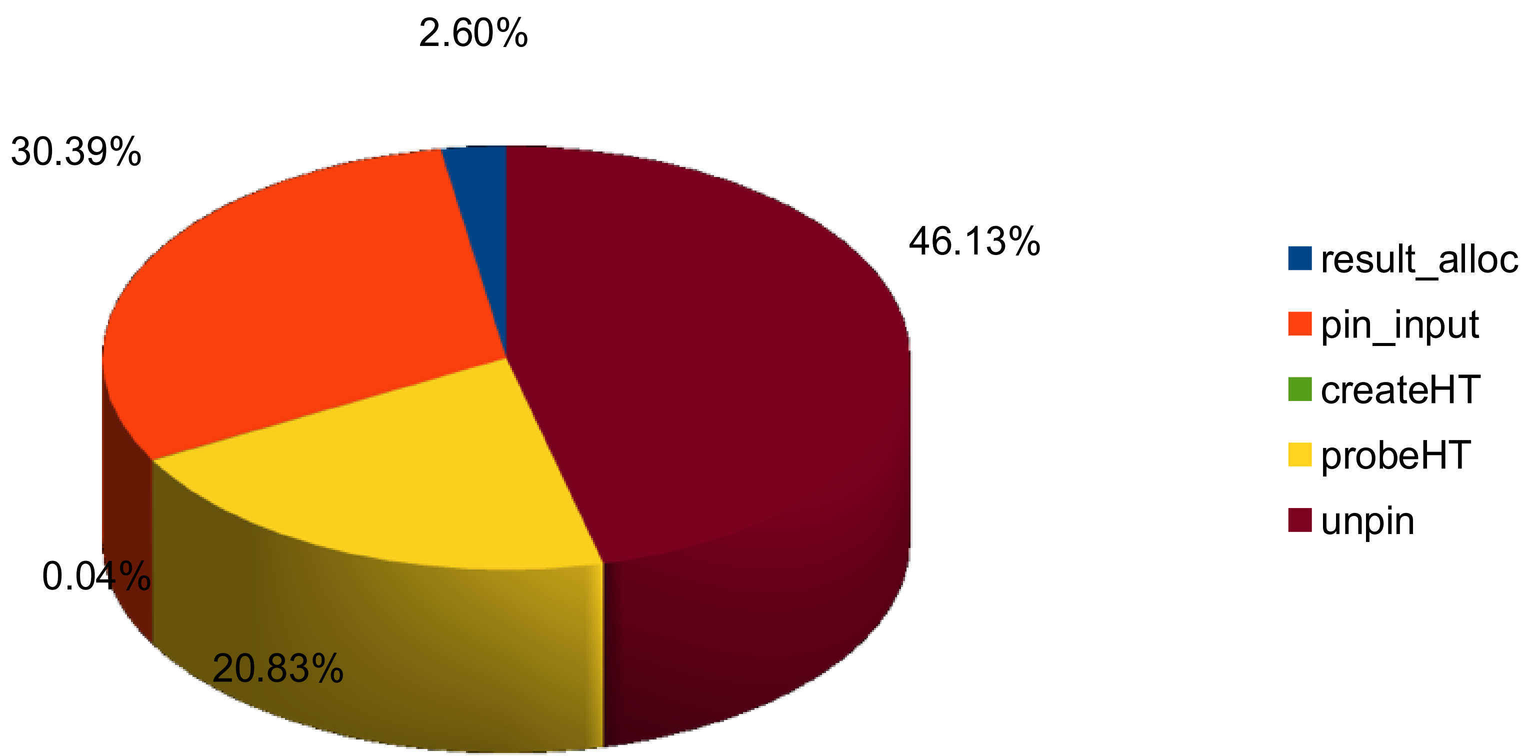

```
SELECT SUM(lo.revenue), d.year, p.brand
    FROM lineorder lo, part p, supplier s, date d
    WHERE p.category = 'MFGR#12' AND lo.partkey = p.partkey
      AND s.region = 'AMERICA' AND lo.suppkey = s.suppkey
      AND lo.orderdate = d.datekey
  GROUP BY d.year, p.brand
  ORDER BY d.year, p.brand;
```

Operator throughput



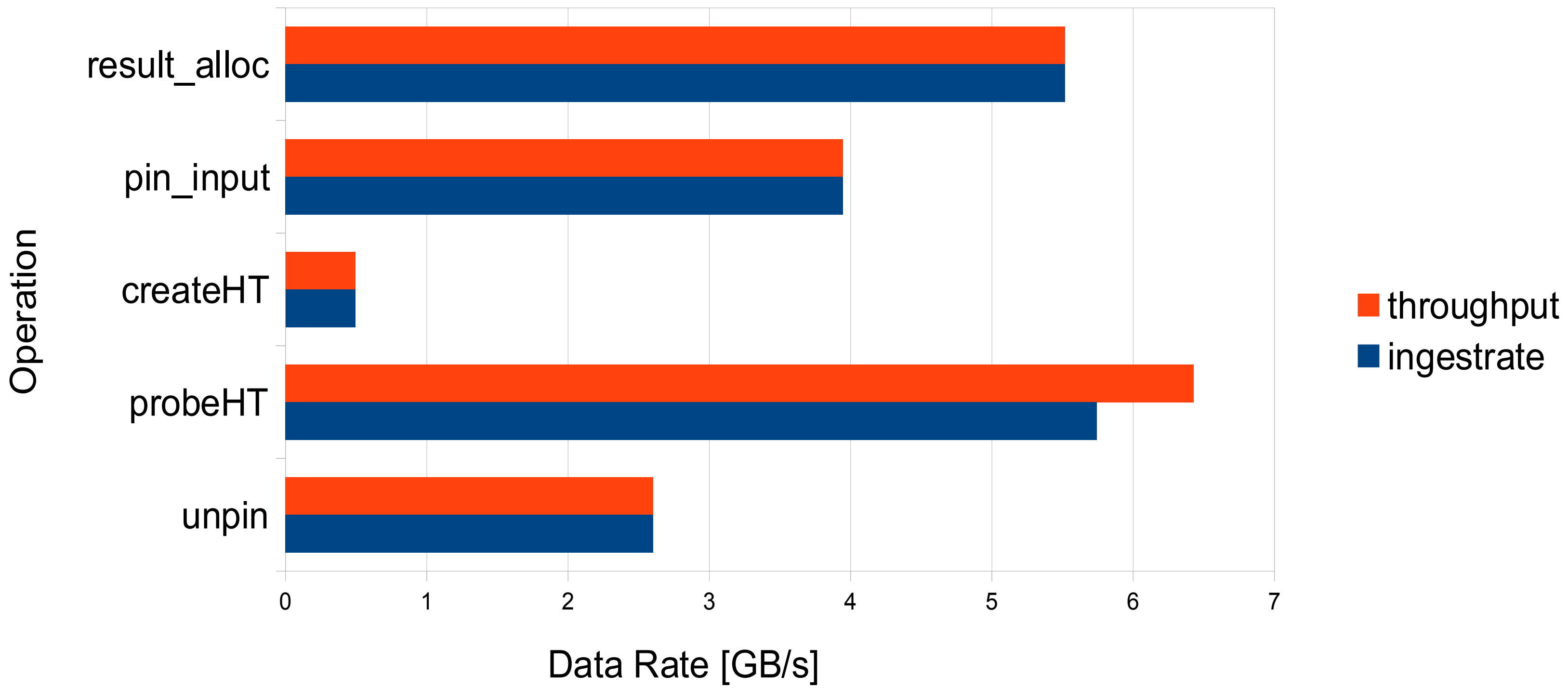
- Using a straight forward GPU implementation
  - Joins are running at < 1.5GB/s, ¼ of the expected speed!
  - Where does time go?

GPU Join – Where does time go?



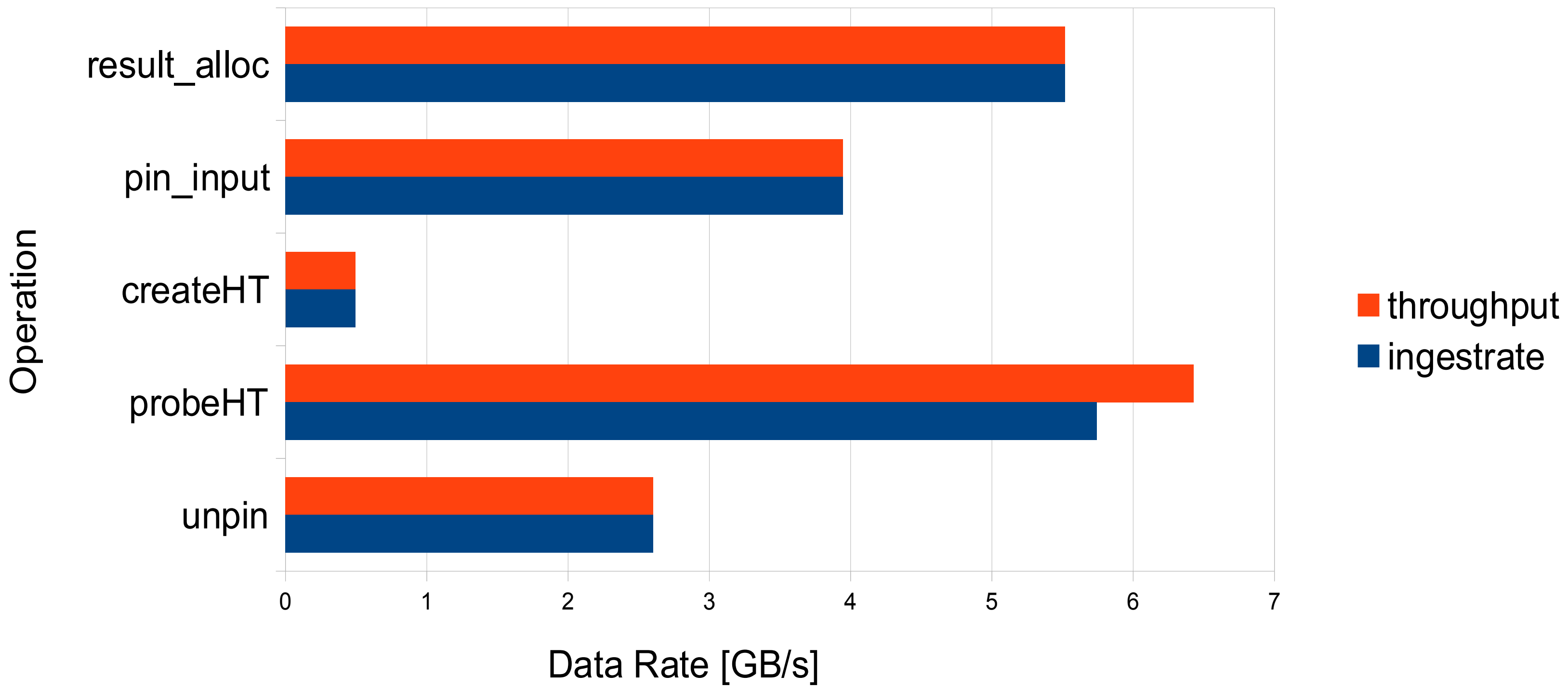
- < 21% of the runtime is spent on the actual join!
- Join lineorder & part has < 4% selectivity
- At SF 100 (100GB database) p.partey is 5.4 MB, lo.partkey is 2.3 GB
  - Need to pin & unpin 2.3 GB of lineorder data

GPU Join – Where does time go?



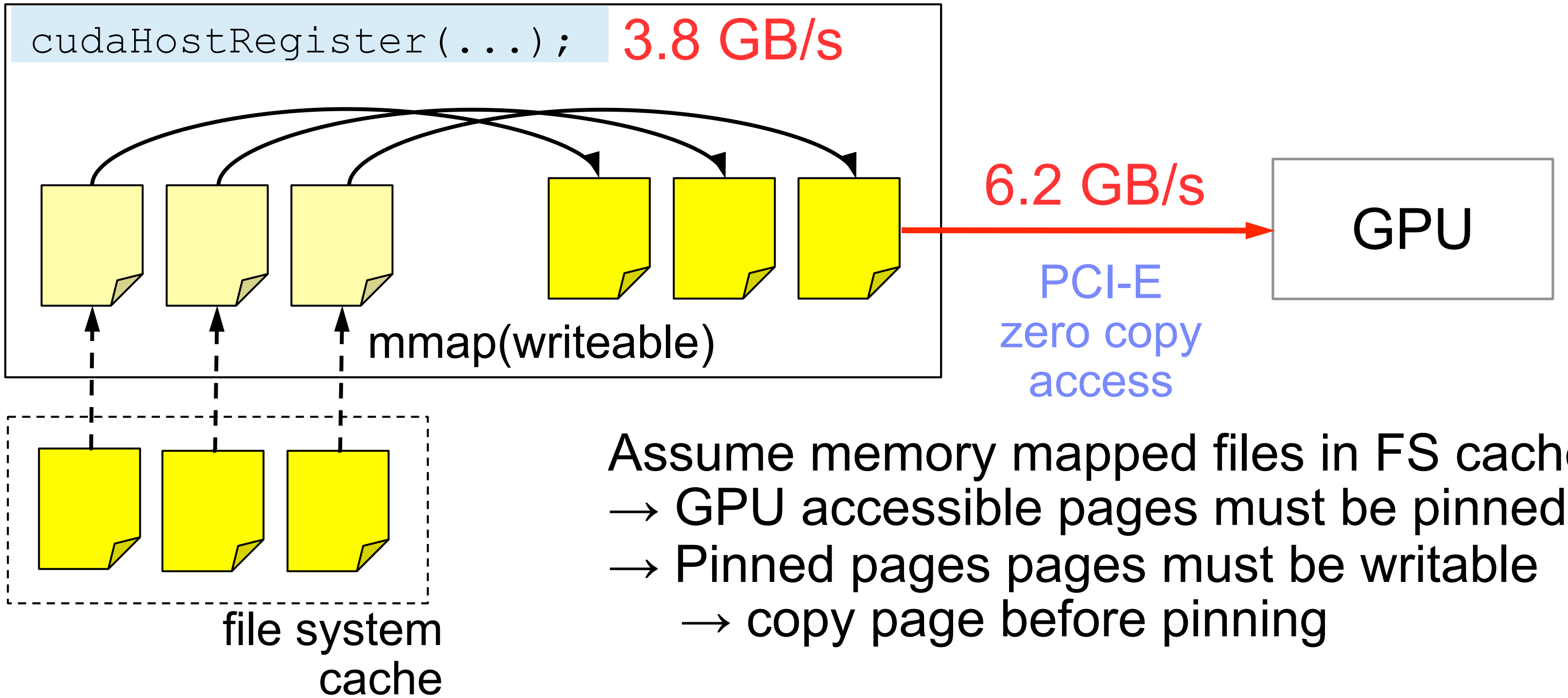
- Pinning/Unpinning large amounts of memory is inefficient and time consuming!

GPU Join – Where does time go?



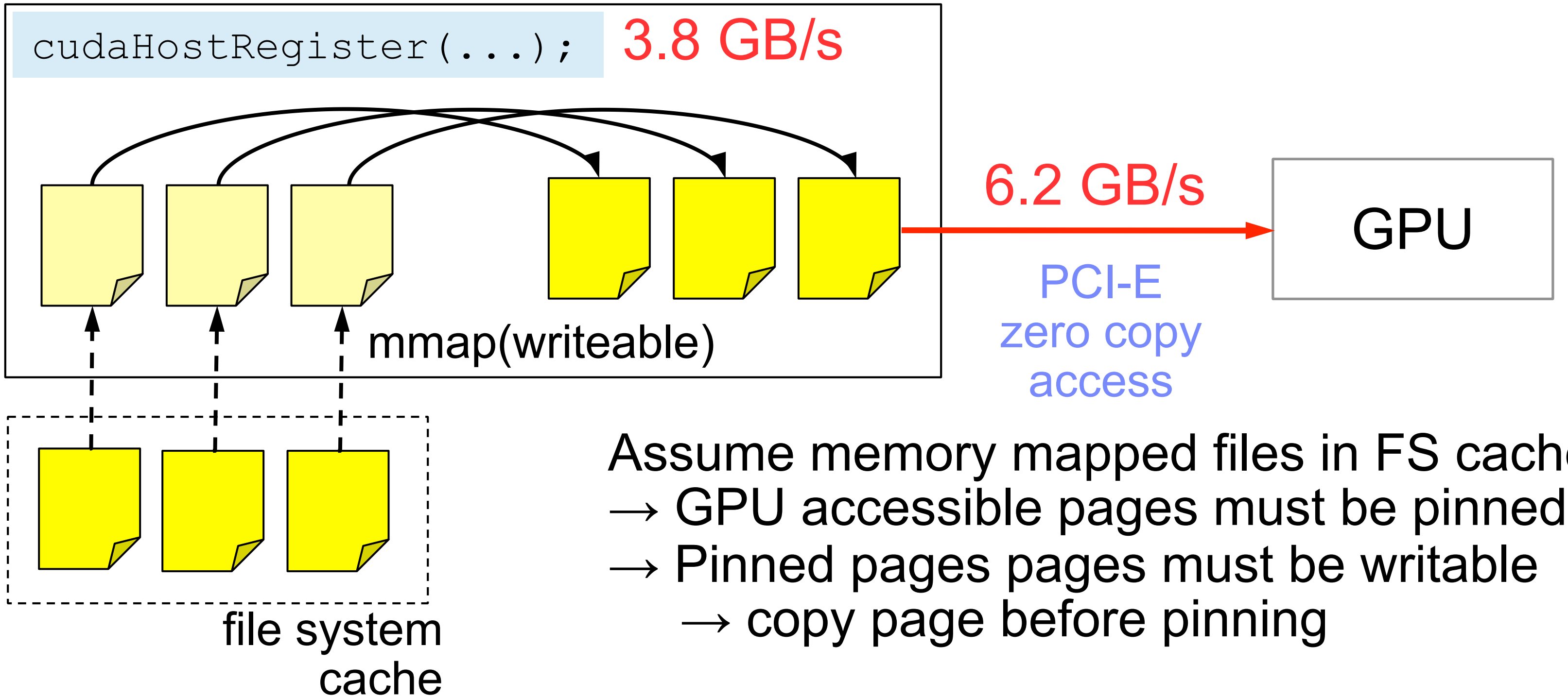
- Pinning/Unpinning large amounts of memory is inefficient and time consuming!
- All steps are sequential … overlapping across operators is messy =(
- We could copy data chunk-wise into a (smaller) pinned buffer …
- Since we are already at it, how do we get the data from the file system (cache)?

Data flow – Current approach



Assume memory mapped files in FS cache
→ GPU accessible pages must be pinned
→ Pinned pages pages must be writable
  → copy page before pinning

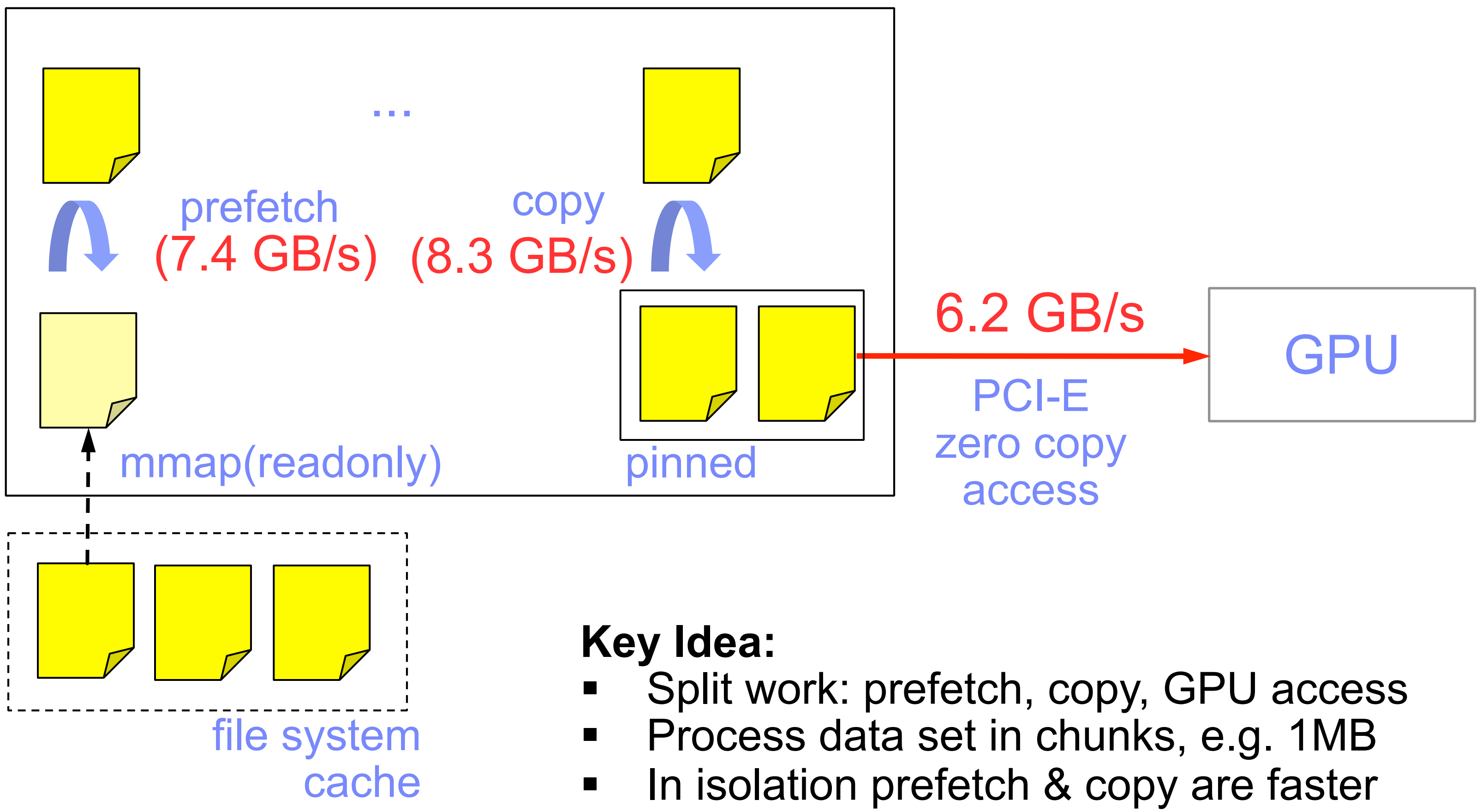
Data flow – Current approach



Assume memory mapped files in FS cache
→ GPU accessible pages must be pinned
→ Pinned pages pages must be writable
→ copy page before pinning

- Even overlapping query execution with pinning pages for next operator (join) leaves pinning as a bottleneck!
- What if we use 2 pre-allocated buffers of pinned memory:
  - Copy data into one of the pinned buffers
  - Meanwhile the GPU can work on the data in the other buffer

Data flow: prefetch → memcpy → GPU access



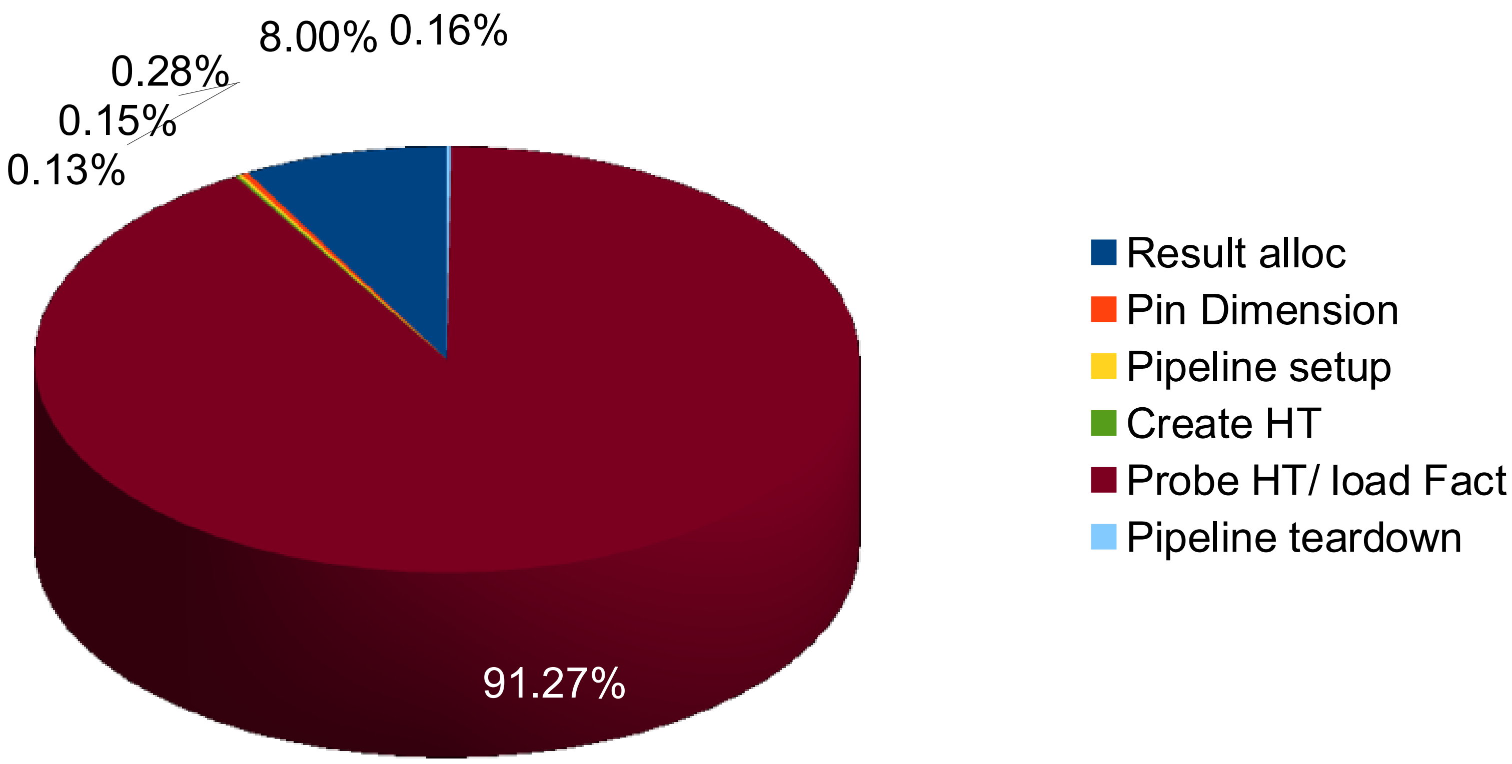
Key Idea:

- Split work: prefetch, copy, GPU access
- Process data set in chunks, e.g. 1MB
- In isolation prefetch & copy are faster than GPU access vie PCI-E =)

- Can we "simply" set up 3-stage Pipeline?

Join with 3-stage pipeline

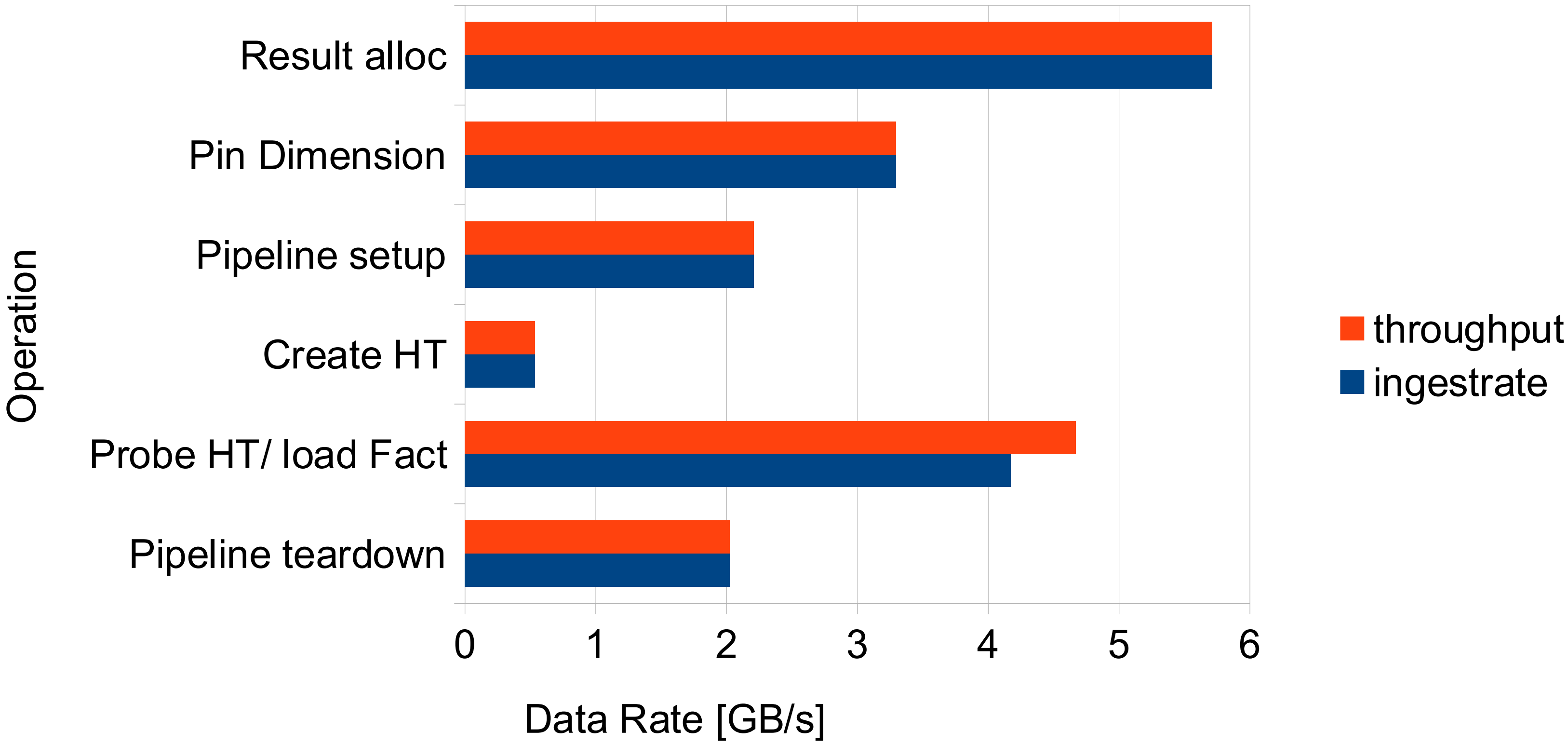
- 2x 1MB buffers, ~2300 Kernel invocations



SELECT SUM(lo.revenue), d.year, p.brand **FROM** lineorder lo, date d, part p, supplier s
WHERE lo.orderdate = d.datekey AND lo.partkey = p.partkey AND lo.suppkey = s.suppkey
AND p.category = 'MFGR#12' AND s.region = 'AMERICA'
GROUP BY d.year, p.brand **ORDER BY** d.year, p.brand

Join with 3-stage pipeline

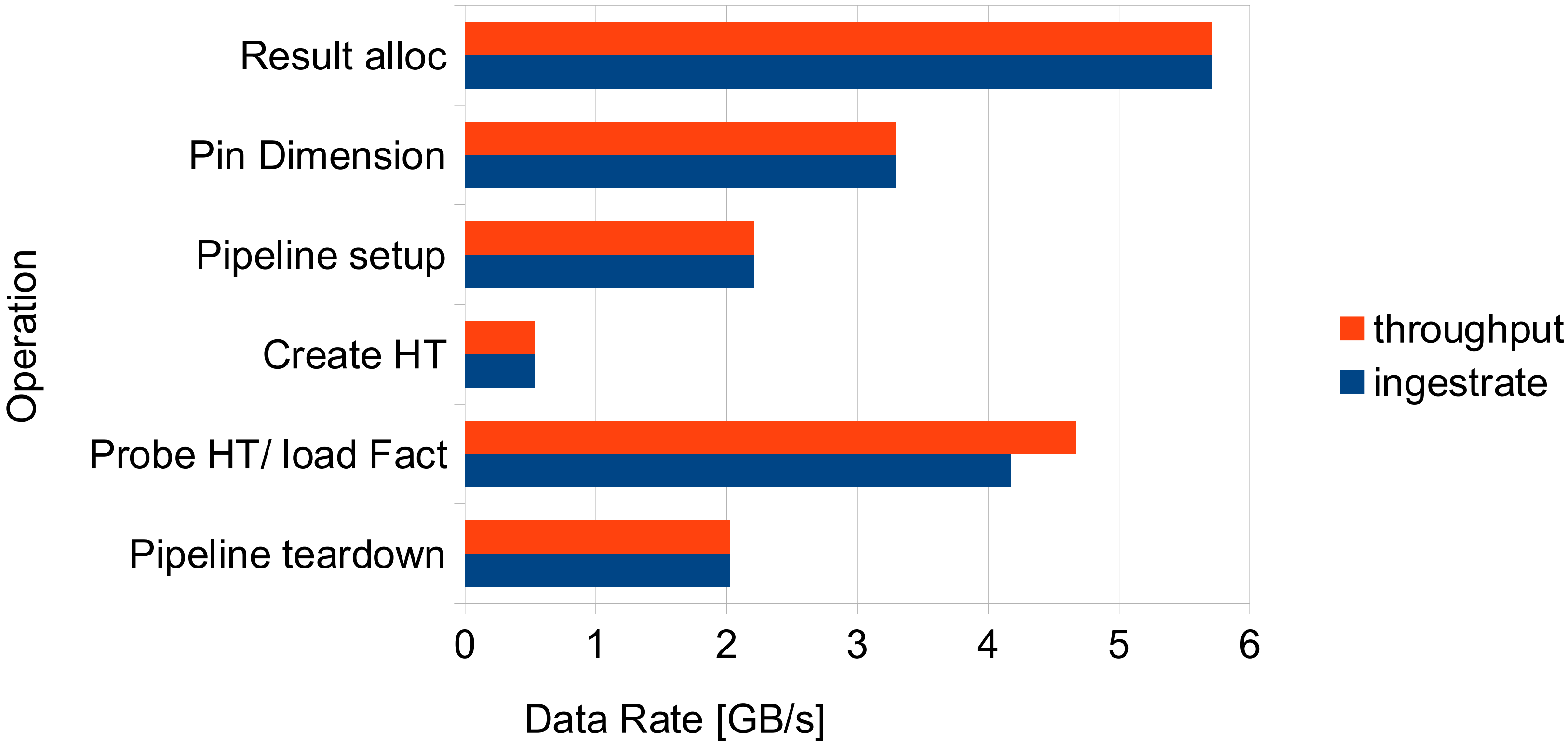
- Join lineorder & part using 2x 1MB buffers yields > 4GB/s overall throughput



- Other joins (with supplier and date) exhibit similar performance

Join with 3-stage pipeline

- Join lineorder & part using 2x 1MB buffers yields > 4GB/s overall throughput



- Other joins (with supplier and date) exhibit similar performance
- Group-by operator is quite similar to join, i.e. requires a hash table and an atomic add and also achieves similar performance
- Accelerating other operators is not worthwhile …

Agenda

- GPU search
  - Reminder: Porting CPU search
  - Back to the drawing board:
    - P-ary search
    - Experimental evaluation
    - Why it works
- Building a complete data warehouse runtime with GPU support
  - From a query to operators – what to accelerate?
  - What are the bottlenecks/limitations
- Maximizing data path efficiency
  - Extremely fast storage solution
  - Storage to host to device
- Putting it all together
  - Prototype demo

6GB/s Storage Subsystem ?

- According to OCZ spec a Revodrive3 x2 can deliver 1.5 GB/s per card

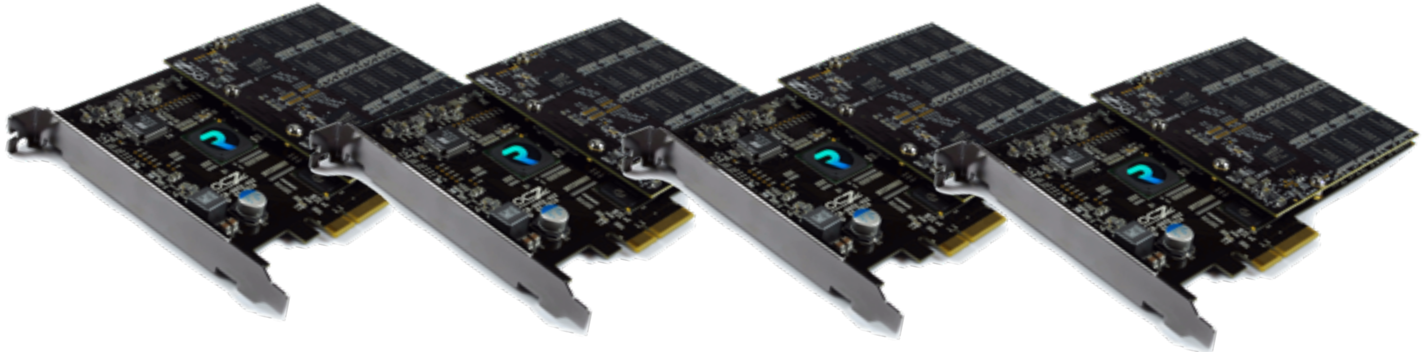
6GB/s Storage Subsystem ?

- According to OCZ spec a Revodrive3 x2 can deliver 1.5 GB/s per card
- Can we “just stripe” the data across 4 cards and achieve 6 GB/s?

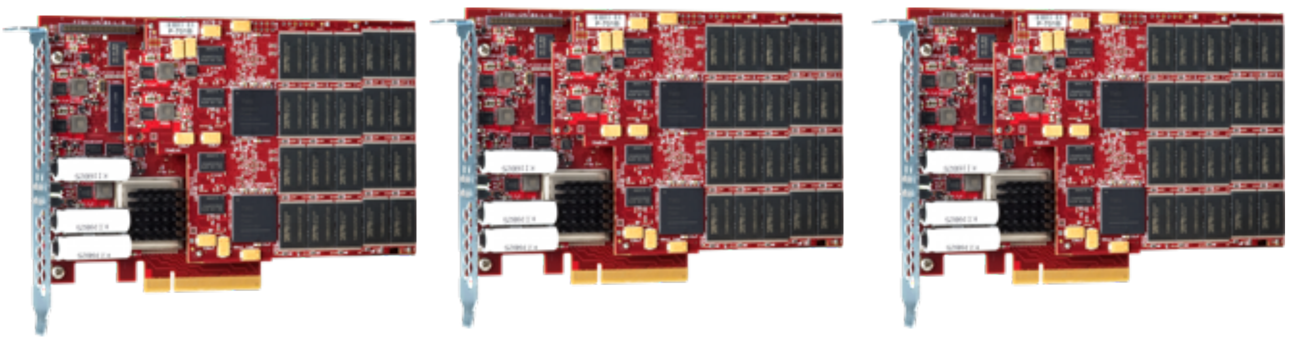
6GB/s Storage Subsystem ?

- According to OCZ spec a Revodrive3 x2 can deliver 1.5 GB/s per card
- Can we “just stripe” the data across 4 cards and achieve 6 GB/s?
  - Linux tools, i.e. mdraid + ext, max 2 GB/s
  - IBM GPFS with striping and heavy prefetching(72 threads) achieves 3 GB/s
  - SSD controllers on commodity SSDs use compression to improve throughput

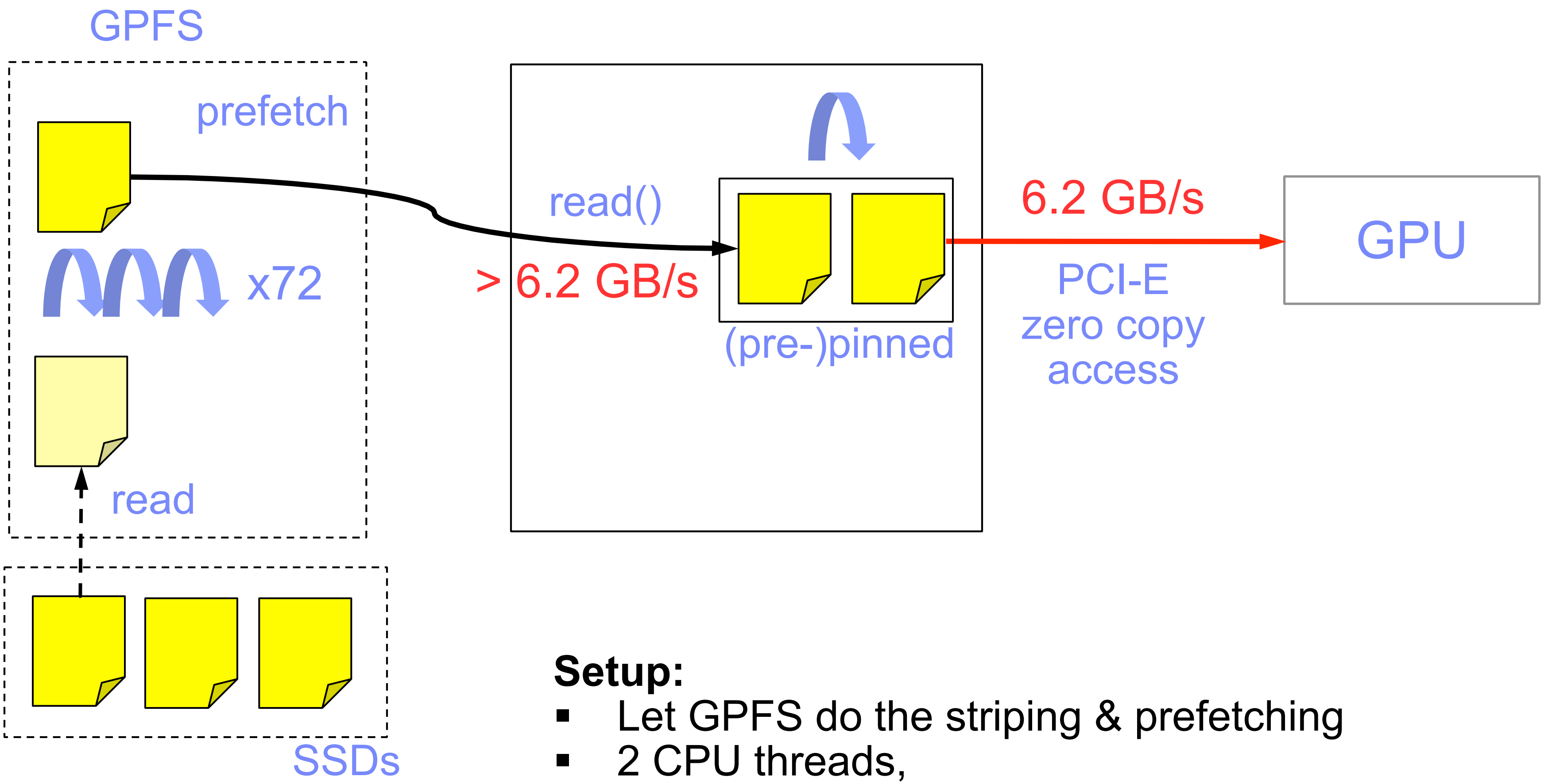
6GB/s Storage Subsystem ?



- According to OCZ spec a Revodrive3 x2 can deliver 1.5 GB/s per card
- Can we "just stripe" the data across 4 cards and achieve 6 GB/s?
  - Linux tools, i.e. mdraid + ext, max 2 GB/s
  - IBM GPFS with striping and heavy prefetching(72 threads) achieves 3 GB/s
  - SSD controllers on commodity SSDs use compression to improve throughput
- Using 3 Texas Memory RamSan-70 and GPFS we get up to 7.5 GB/s =)



Data flow: read → memcpy → GPU access

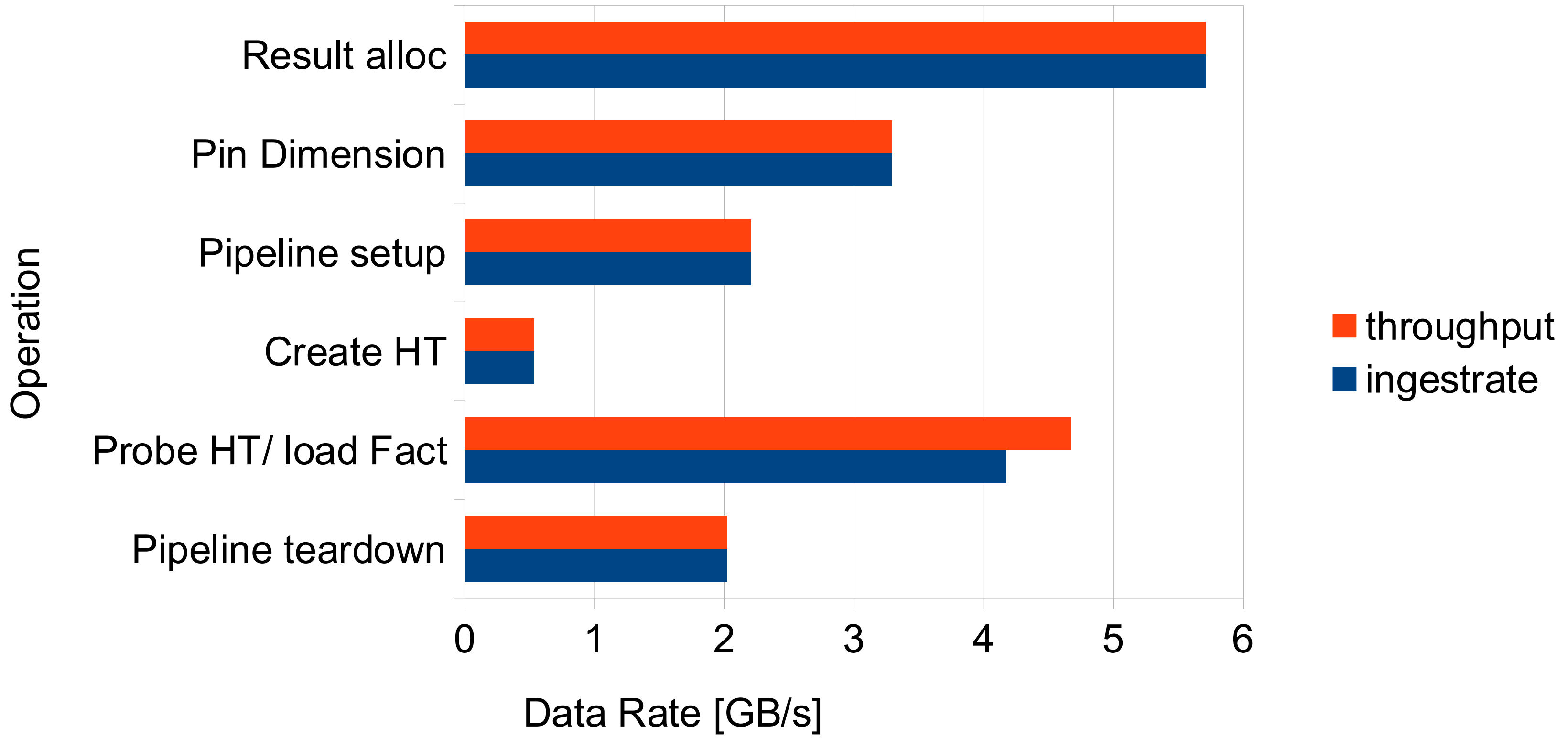


Setup:

- Let GPFS do the striping & prefetching
- 2 CPU threads,
  - 1 for filling a pinned buffer from FS
  - 1 for controlling GPU execution
- GPU reads data from pinned buffer(s)

Join with 3-stage pipeline from SSD

- Join lineorder & part using 2x 2MB buffers yields > 4GB/s overall throughput



- Virtually no performance difference to in-memory solution =)

Agenda

- GPU search
  - Reminder: Porting CPU search
  - Back to the drawing board:
    - P-ary search
    - Experimental evaluation
    - Why it works
- Building a complete data warehouse runtime with GPU support
  - From a query to operators – what to accelerate?
  - What are the bottlenecks/limitations
- Maximizing data path efficiency
  - Extremely fast storage solution
  - Storage to host to device
- Putting it all together
  - Prototype demo

Questions?