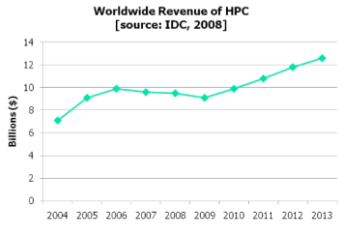
Relational Query Coprocessing on GPU

adapted from Bingsheng He's slides "iHPC: Towards Pervasive High Performance Computing" and Ram Suman Karumuri's slides on Bingsheng's "Relational Joins on Graphics Processors"*

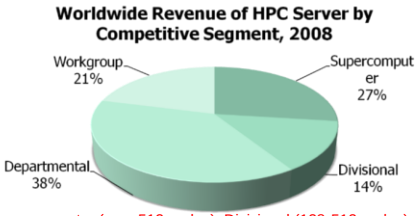
* http://www.cs.brown.edu/~suman/slideshows.html

The HPC Market Is Growing



Slight market decrease for 2009, with a strong increase from 2010

Diversifying HPC: It's Not Just for Rocket Scientists Any More*



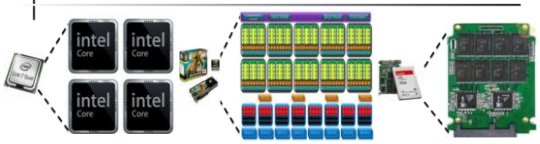
- Supercomputer (over 512 nodes), Divisional (128-512 nodes), Departmental (16-128 nodes), Workgroup (less than 16 nodes).
- HPC continues to be diversifying.

*Source: "High Performance Computing for Dummies"

2 HPC

- HPC@home
  - Build your HPC server at home
- HPC@cloud
  - "Build" your HPC cluster in the cloud

Build Your Own HPC Server



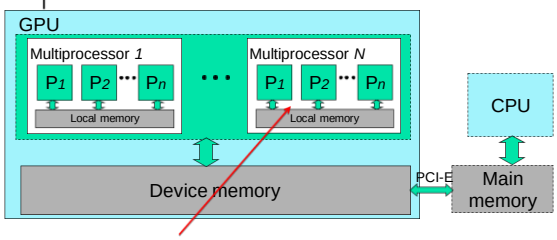
Commodity	Configuration	Price (USD)*
Intel Core2 Duo Q6600	2.66G Hz*4	429.95
NVIDIA GTX 280 GPU	1.3G Hz *240	269.99
Intel X25-M 80GB 2.5"	35 thousand (4K reads)	429.95

Parallelism boosts the hardware capability.

Challenges in software: programming and performance.

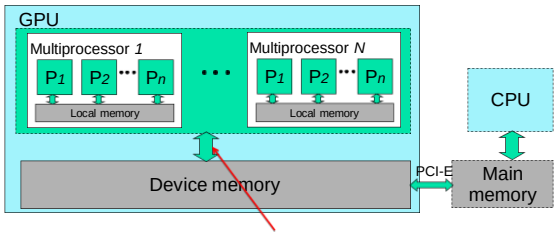
*Source: www.amazon.com, sept-21-2009

GPU: A Powerful Co-processor



- 240 scalar processors on NV GTX 280
- ~1 TFLOPS of peak performance

GPU: A Powerful Co-processor



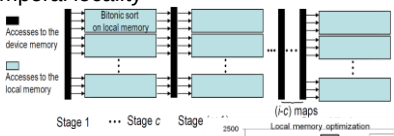
- 10+x more than peak bandwidth of the main memory
- 142 GB/s, 1 GB GDDR3 memory on GTX280

GPUs

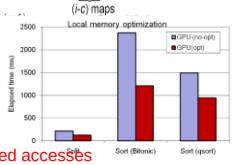
- High latency GDDR memory
  - 200-400 clock cycles of latency
  - Latency hiding using large number of concurrent threads (>8K on GTX GPU)
  - Each thread has a small state – low context-switch overhead
- Better architectural support for memory
  - Inter-processor communication using a local memory
  - Coalesced access

Local Memory Optimization

- temporal locality



Data accesses in the bitonic sort on GPU with local mem opt *

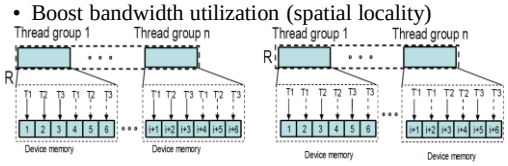


* Bingsheng He,etc: Relational joins on graphics processors. SIGMOD Conference 2008: 511-524

with coalesced accesses

Coalesced Access

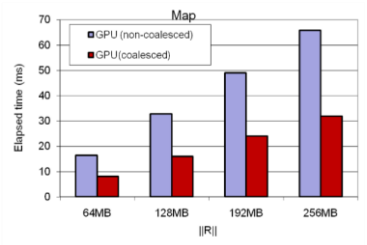
- Boost bandwidth utilization (spatial locality)



Maps with and without coalesced accesses*

* Bingsheng He, etc: Relational joins on graphics processors. SIGMOD Conference 2008: 511-524

Coalesced Access



(1) Bp=16, T=32,
(2) w/o local memory opt.

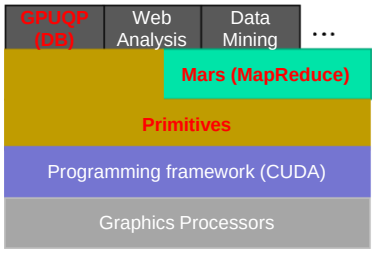
Challenges for GPUQP

- Programming difficulty
- How to exploit the hardware feature of the GPU
  - High thread parallelism
  - Memory features
- Lack hardware support for handling read/write conflicts
- Load balancing

Solution

- Primitive-based approach
  - Basic operations as building blocks for high-level operations.
  - Easier to optimize than complicated operations/applications.
- Skew handling
- Lock free design for many-core features

The Infrastructure of HPC@home using GPUs



Outline

- HPC@home
  - Primitives
  - Engines
    - GPUQP
- Conclusions and Future Work

Primitives

Primitive: Map
Input: $R_{in}[1, \ldots, n]$, a map function *fcn*.
Output: $R_{out}[1,\ldots,n]$.
Function: $R_{out}[i]=fcn(R_{in}[i])$.

Primitives

Primitive: Map

Primitive: Scatter
Input: $R_{in}[1, \ldots, n]$, $L[1, \ldots, n]$.
Output: $R_{out}[1, \ldots, n]$.
Function: $R_{out}[L[i]]=R_{in}[i]$, $i=1, \ldots, n$.

Primitives

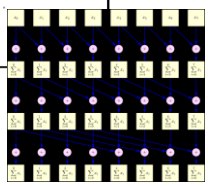
Primitive: Map

Primitive: Scatter

Primitive: Gather
Input: $R_{in}[1, \ldots, n]$, $L[1, \ldots, n]$.
Output: $R_{out}[1, \ldots, n]$.
Function: $R_{out}[i]=R_{in}[L[i]]$, $i=1, \ldots, n$.

Primitives

Primitive: Map
Primitive: Scatter
Primitive: Gather
Primitive: Prefix Scan
Input: $R_{in}[1, \ldots, n]$, binary operator $\oplus$
Output: $R_{out}[1,\ldots,n]$.
Function: $R_{out}[i] = \bigoplus_{j \le i} R_{in}[j]$.



http://en.wikipedia.org/wiki/Prefix_sum

Primitives

Primitive: Map
Primitive: Scatter
Primitive: Gather
Primitive: Prefix Scan
Primitive: Split
Input: $R_{in}[1, \ldots, n]$, $fimc(R_{in}[i]) \in [1,\ldots,F]$, i=1, …, n.
Output: $R_{out}[1, \ldots, n]$.
Function: $\{R_{out}[i],\ i=1,\ldots,\ n\} = \{R_{in}[i],\ i=1,\ \ldots,\ n\}$ and $fimc(R_{out}[i]) \le fimc(R_{out}[j]), \forall i,j \in [1,\ldots,n], i \le j$.

Primitives

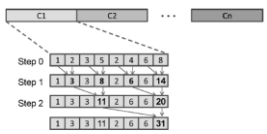
Primitive: Map
Primitive: Scatter
Primitive: Gather
Primitive: Prefix Scan
Primitive: Split
Primitive: Sort
Input: $R_{in}[1, \ldots, n]$.
Output: $R_{out}[1, \ldots, n]$.
Function: $\{R_{out}[i],\ i=1,\ldots,\ n\} = \{R_{in}[i],\ i=1,\ \ldots,\ n\}$ and $R_{out}[i] \le R_{out}[j], \forall i,j \in [1,\ldots,n]$ and $i \le j$.

Sort

- Bitonic sort
  - Uses sorting networks, O(N $\log^2$N)
- Quick sort
  - partition using a random pivot until partition fits in local memory
  - Sort each partition using bitonic sort
  - Partioning can be parallelized using split
  - Complexity is O(N log N)
  - 30% faster than bitonic sort in experiments
  - GPUQP uses Quick sort for sorting

Reduce

- Primitive: Reduce
- Input: $R_{in}[1, \ldots, n]$, a *reduce* function $\odot$
- Output: $R_{out}[1]$
- Function: $R_{out}[1] = \odot_{i=1}^{n} R_{in}[i]$



One pass of the reduce primitive

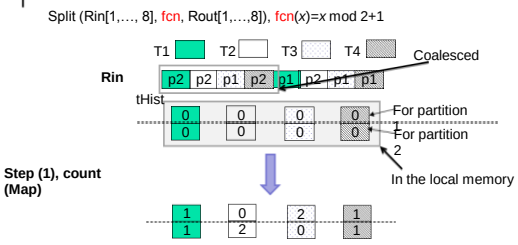
Filter

- Primitive: Filter
- Input:
  - $R_{in}[1, \ldots, n]$,
  - a *filter* function $fcn(R_{in}[i]) \in \{0,1\}, i \in [1,n]$
- Output: $R_{out}[1]$
- Function: $\{R_{out}[i], i \in [1,n']\} = \{R_{in}[i] | fcn(R_{in}[i]) = 1, i \in [1,n]\}$

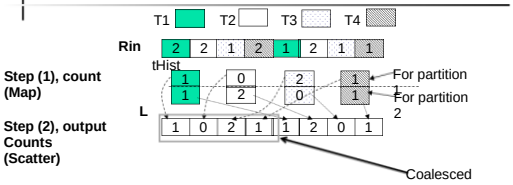
Split

- A lock-free algorithm
  - Each thread is responsible for a portion of the input.
  - Each thread computes its local histogram.
  - Given the local histograms, we compute the write locations for each thread.
  - Each thread writes the tuples to the output in parallel.

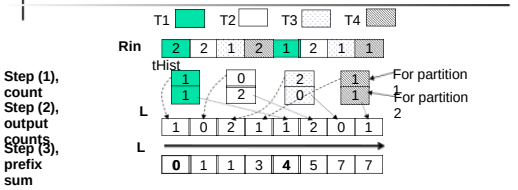
Split



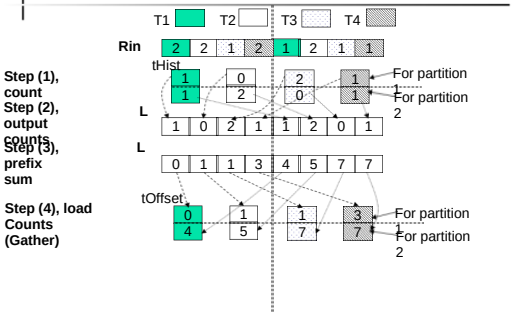
Split



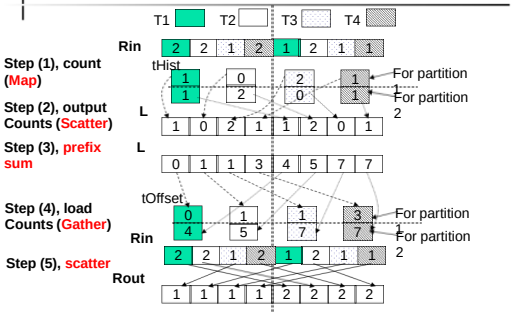
Split



Split



Split



Optimizing Primitives

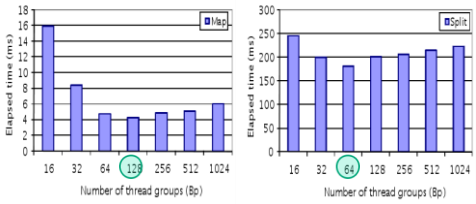
- Thread parallelism
  - Parallelism among different multiprocessors.
  - Resource utilization within a multiprocessor.
- Memory optimizations
  - Coalesced access for spatial locality.
  - Local memory optimization for temporal locality.

Experimental Setup

- Implementation
  - CPU: OpenMP
  - GPU: **CUDA**

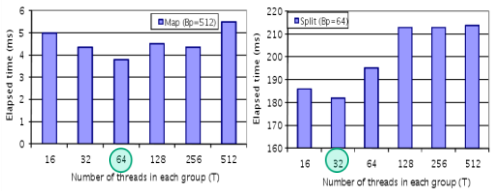
	CMP (P4 Quad)	GPU (NV G80)
Processors (HZ)	2.66G*4	**1.35G*128**
Cache size	**8MB**	256KB
Bandwidth (GB/sec)	10.4	**86.4**

Thread Parallelism (Varying #thread groups)



(1) T=32,
(2) w/ coalesced accesses,
(3) w/o local memory opt.

Thread Parallelism (Varying #thread/group)



(1) Suitable Bp,
(2) w/ coalesced accesses.

Performance Impact of Optimization Techniques

- Coalesced access improves the memory bandwidth by twice.
- Performance improvements of thread parallelism depend on the computation/memory characteristics (30%- 4X)
- Local memory optimization improves *split* and *sort* by twice.
- Primitives : 2~27x

Outline

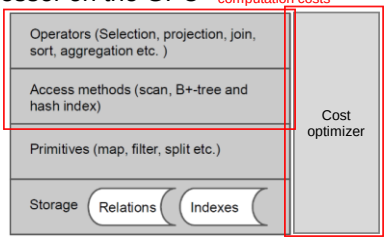
- HPC@home
  - Primitives
  - Engines
    - GPUQP
- Conclusions and Future Work

GPUQP

- The *first* full-fledged relational query processor on the GPU

Estimation on memory and computation costs

Built on top of optimized primitives



Joins

- Non-indexed nested-loop join (NINLJ)
- Indexed nested-loop join (INLJ)
  - Adopt CSS-Tree [Rao99]
- Sort-merge join (SMJ)
- Hash join (HJ)
  - Adopt radix join [Boncz99]

B+ Tree vs. CSS Tree

- B+ tree imposes Memory stalls when traversed (no spatial locality)
  - Can't perform multiple searches (loses temporal locality).
- CSS-Tree (Cache optimized search tree)
  - One dimensional array where nodes are indexed.
  - Replaces traversal with computation.
  - Can also perform parallel key lookups.

A Lock-Free Scheme for Result Output

- Three steps:
  - Each thread **counts** the number of join results for the partitioned join.
  - **Prefix sum** for write locations for each thread and the total number of join results.
  - Each thread **outputs** the join results in parallel.

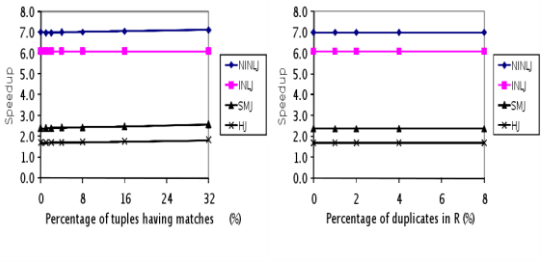
Hash Join

- Hash join: HJ (R, S)
  - *Split* R, S into the same number of partitions using radix bits so that most *S* partitions fit into the local memory
    => Skew handling: Identify the partitions that do not fit into the local memory, and continue split
    => A join is decomposed into many small joins.
  - Multiple small joins are evaluated in parallel.

Skew Handling in HJ

- Identify the partitions that do not fit into the local memory.
  - Given an array storing partition sizes, we **split** it into two groups.
    - Partitions larger than the local memory
    - Partitions not larger than the local memory
- Decompose each of the large partitions into multiple small chunks.

Joins (Cont')



Experimental Results on Join Queries

Joins	CPU (sec)	GPU(sec)	Speedup
NINLJ	528.0	75.0	**7.0**
INLJ	4.2	0.7	**6.1**
SMJ	5.0	2.0	**2.4**
HJ	2.5	1.3	**1.9**

- In-memory databases
- The GPU measurements **include** the time for data transfer between the GPU memory and the main memory.
- Tuple size=8 B, NINLJ (1million by one million), other joins (16 million by 16 million)

Cost Estimation for GPU

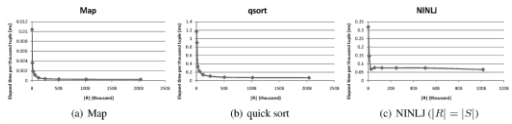
$$T_{overall} = T_{mm\_dm}(I) + T_{GPU} + T_{dm\_mm}(O)$$

$$T_{GPU} = T_{Mem} + T_{computation}$$

$$T_{mm\_dm}(x) = T_0 + \frac{x}{Band}$$

Estimating $T_{computation}$

- measure unit cost



$$\mu = \frac{T^0_{GPU}}{O(n_0, \ldots, n_q)}$$

$$T_{computation} = \mu \cdot O(N_1, \ldots, N_q)$$

TPC-H Results on Memory-Resident Data

SF=1	Q1 (sec)	Q3 (sec)
DBMS X	14.0	3.8
CPU	1.01	0.79
GPUQP	0.89	0.66

- SF=1, working set=1 GB; warmed buffer.
- Both CPU and GPUQP outperforms DBMS X over 4.7 times.
- GPUQP is 13-20% faster than CPU-based engine.

Performance

- CPU & GDB engines outperform DBMS X by over 13.8times and 3.5 times @SF = 1/10
- overall performance of GDB
  - slightly faster than the CPU-based engine
  - disk I/O time contributes 98% to the total execution time when SF = 10
- GPU-based algorithms are poor
  - poor for simple query: data transfer between main/device mem
  - Faster for complex queries: insignificant data transfer

Performance (Cont')

- GDB
  - significantly cool on memory-resident data
    - Primitives & query processing algorithms 2–27x over optimized CPU-based counterparts
    - C/GPU data transfer included
      - 2–7x complex queries such as joins
      - 2–4x slower for simple queries such as selections
  - comparable to optimized CPU-based engine on disk-based data: on TPC-H with data sets larger than mem
    - GPU coprocessing reduces the computation time up to 23%
    - Overall improvement is insignificant: disk I/O bottleneck

Conclusion

- The GPU has much higher computation power and memory bandwidth than the CPU.
- Highly-optimized primitives as building blocks is practical for high-level applications.
- GPU-based primitives are 2-27x faster than their CPU-based counterparts.

Future Work

- Compression to reduce main/G memory data transfer overhead
- Multi-GPU processing
- New memory techniques
  - Jim Gray: "flash is disk, disk is tape and tape is dead"
  - Faster memory
    - PCM, MEMS
    - Design efficient data structures and algorithms on new memories
    - Re-design file systems