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

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"

Build Your Own HPC Server

<table>
<thead>
<tr>
<th>Commodity</th>
<th>Configuration</th>
<th>Price (USD)*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core2 Duo U6600</td>
<td>2.66 GHz*^4</td>
<td>$429.95</td>
</tr>
<tr>
<td>NVIDIA GTX 280 GPU</td>
<td>1.36 GHz*^4, 240 Gflop</td>
<td>$269.99</td>
</tr>
<tr>
<td>Intel X25-M 80GB</td>
<td>2.5&quot; SSD, 4K reads 500</td>
<td>$429.95</td>
</tr>
</tbody>
</table>

Parallelism boosts the hardware capability.

Challenges in software: programming and performance.

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

2 HPC

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

GPU: A Powerful Co-processor

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

Challenges in software: programming and performance.
**GPU: A Powerful Co-processor**

- GPU: A Powerful Co-processor
- Device memory
- Main memory
- PCI-E

**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

**Coalesced Access**

- Boost bandwidth utilization (spatial locality)

**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

Outline

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

Primitives

<table>
<thead>
<tr>
<th>Primitive: Scatter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: R_{i}, L_{i}, i=1,...,n.</td>
</tr>
<tr>
<td>Output: R_{i}[1,...,n].</td>
</tr>
<tr>
<td>Function: R_{i}[1,...,n], i=1,...,n.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Primitive: Gather</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: R_{i}[1,...,n], L_{i}[1,...,n].</td>
</tr>
<tr>
<td>Output: R_{i}[1,...,n].</td>
</tr>
<tr>
<td>Function: R_{i}[1,...,n].</td>
</tr>
</tbody>
</table>

The Infrastructure of HPC@home using GPUs
Primitives

- **Map**
- **Scatter**
- **Gather**
- **Prefix Scan**
  - Input: \( R_[1, \ldots , n] \)
  - Output: \( R_\text{out}[1, \ldots , n] \)
  - Function: \( R_\text{out}[i] = \bigoplus_j R_\text{in}[j] \)

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

Primitives

- **Map**
- **Scatter**
- **Gather**
- **Prefix Scan**
  - **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
      - Partitioning 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**: \( R_\text{in}[1, \ldots , n] \), a reduce function
- **Output**: \( R_\text{out}[1] \)
- **Function**: \( R_\text{out}[1] = \bigcup_{i=1}^{n} R_\text{in}[i] \)

One pass of the reduce primitive

Filter

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

<table>
<thead>
<tr>
<th></th>
<th>CMP (P4 Quad)</th>
<th>GPU (NV G80)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processors (Hz)</td>
<td>2.66G*4</td>
<td>1.35G*128</td>
</tr>
<tr>
<td>Cache size</td>
<td>8MB</td>
<td>256KB</td>
</tr>
<tr>
<td>Bandwidth (GB/sec)</td>
<td>10.4</td>
<td>86.4</td>
</tr>
</tbody>
</table>

Thread Parallelism (Varying #thread groups)

- Suitable Bp.
- w/ coalesced accesses.
- w/o local memory opt.

Thread Parallelism (Varying #thread/group)

- 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

<table>
<thead>
<tr>
<th>Operators (Selection, projection, join, sort, aggregation etc.)</th>
<th>Access methods (scan, B+tree and hash index)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitives (map, filter, split etc.)</td>
<td>Storage</td>
</tr>
</tbody>
</table>

Built on top of optimized primitives

Cost optimizer

Estimation on memory and computation costs

Joins

• Non-indexed nested-loop join (NINLJ)
• Indexed nested-loop join (INLJ)
  o Adopt CSS-Tree [Rao99]
• Sort-merge join (SMJ)
• Hash join (HJ)
  o 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:
  o Each thread counts the number of join results for the partitioned join.
  o Prefix sum for write locations for each thread and the total number of join results.
  o Each thread outputs the join results in parallel.

Hash Join

• Hash join: HJ (R, S)
  o 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.
  o Multiple small joins are evaluated in parallel.

Skew Handling in HJ

• Identify the partitions that do not fit into the local memory.
  o 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

<table>
<thead>
<tr>
<th>Joins</th>
<th>CPU (sec)</th>
<th>GPU (sec)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>NINLJ</td>
<td>528.0</td>
<td>75.0</td>
<td>7.0</td>
</tr>
<tr>
<td>INLJ</td>
<td>4.2</td>
<td>0.7</td>
<td>6.1</td>
</tr>
<tr>
<td>SMJ</td>
<td>5.0</td>
<td>2.0</td>
<td>2.4</td>
</tr>
<tr>
<td>HJ</td>
<td>2.5</td>
<td>1.3</td>
<td>1.9</td>
</tr>
</tbody>
</table>

- 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 (1 million by one million), other joins (16 million by 16 million)

Cost Estimation for GPU

\[ T_{\text{total}} = T_{\text{MAIN,IN}}(I) + T_{\text{GPU}} + T_{\text{MAIN,OUT}}(O) \]

\[ T_{\text{GPU}} = T_{\text{MEM}} + T_{\text{computation}} \]

\[ T_{\text{MAIN,IN}}(x) = T_0 + \frac{x}{Band} \]

Estimating \( T_{\text{computation}} \)

- measure unit cost

\[ \mu = \frac{T_{\text{GPU}}}{O(N_0, \ldots, N_p)} \]

\[ T_{\text{computation}} = \mu \cdot O(N_1, \ldots, N_p) \]

TPC-H Results on Memory-Resident Data

<table>
<thead>
<tr>
<th>SF=1</th>
<th>Q1 (sec)</th>
<th>Q3 (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBMS X</td>
<td>14.0</td>
<td>3.8</td>
</tr>
<tr>
<td>CPU</td>
<td>1.01</td>
<td>0.79</td>
</tr>
<tr>
<td>GPUQP</td>
<td>0.89</td>
<td>0.66</td>
</tr>
</tbody>
</table>

- 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 13.8 times 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