GPUQP: Query Co-Processing Using Graphics Processors

Rui Fang, Bingsheng He, Mian Lu, Ke Yang, Naga K. Govindaraju*, Qiong Luo, Pedro V. Sander
{rayfang, saven, mianlu, keyang, luo, psander}@cse.ust.hk nagag@microsoft.com
The Hong Kong University of Science and Technology *Microsoft Corporation
http://www.cse.ust.hk/gpuqp

ABSTRACT
We present GPUQP, a relational query engine that employs both CPUs and GPUs (Graphics Processing Units) for in-memory query co-processing. GPUs are commodity processors traditionally designed for graphics applications. Recent research has shown that they can accelerate some database operations orders of magnitude over CPUs. So far, there has been little work on how GPUs can be programmed for heavy-duty database constructs, such as tree indexes and joins, and how well a full-fledged GPU query co-processor performs in comparison with their CPU counterparts. In this work, we explore the design decisions in using GPUs for query co-processing using both a graphics API and a general purpose programming model. We then demonstrate the processing flows as well as the performance results of our methods.

Categories and Subject Descriptors
H.2.4 [**Database Management**]: Systems – *parallel databases, query processing, relational databases.*

General Terms: Algorithms, Measurement, Performance, Design, Experimentation.

Keywords: Graphics Processing Units, Query Processing.

1. INTRODUCTION
A graphics processing unit (GPU) is an integral part of most computing devices including PCs, laptops, consoles and cell phones. GPUs are highly specialized architectures designed for gaming applications, and can be regarded as massively parallel processors with 10x higher computation and 10x higher memory performance than CPUs [3]. For instance, the NVIDIA GeForce 8800 GPU has over a hundred pixel program processors with an observed performance of 330 GFLOPS and a peak memory bandwidth of 86 GB/s. GPUs are also becoming increasingly programmable enabling them to perform many general purpose algorithms (*GPGPU*) an order of magnitude faster than CPUs. Moreover, GPUs are progressing at a rate faster than CPUs [3]. In this demonstration, we explore in-memory relational query co-processing using GPUs to fully exploit their architectural features.

There has been recent work on performing individual database tasks on GPUs, including spatial joins [9], selection and aggregation queries [6], stream processing [7], and sorting [5]. In these applications, the researchers carefully considered the features and limitations of GPUs with respect to the target operations, and designed algorithms to perform these operations efficiently. In comparison, we take a holistic view of relational query processing and develop an in-memory query co-processor using the out-of-order execution capabilities of CPUs and efficient data-parallel processing capabilities of GPUs. In particular, we revisit GPU sorting, map the CPU version of a tree index to the GPU, and design new GPU join algorithms.

In the following, we introduce the preliminaries for GPU query processing, give an overview of our system, present three specific techniques to show the flavor of GPU query processing, and describe our demonstration plan.

SIGMOD'07, June 12–14, 2007, Beijing, China.
ACM 978-1-59593-686-8/07/0006.

2. PRELIMINARIES
As the GPU is designed for graphics applications, the basic data structure in GPU programming is a 2D array, called a *texture*. An element of a texture, or a *texel*, contains four values, each of which corresponds to a color channel (R, G, B, and A). Textures are accessed during *rendering passes*, where a particular output texture is set as a *render target*, and one or multiple textures are used as input. During this rendering process, the GPU executes the same *pixel program* for each texel in parallel, which may contain arithmetic operations and *texture fetches*.

Texture fetching can read data from arbitrary locations in a texture; however, writing to an output texture is mostly limited to fixed locations. As a result, the most common rendering method used in GPGPU is *full-screen quadrilateral rendering*, which renders a quadrilateral that covers the entire texture.

Fortunately, there have emerged several hardware and software solutions to support *data scattering*, which allows a pixel program to modify the output position of a texel. In addition, general purpose programming models allow the database developer to use the GPU without any knowledge of computer graphics.

In this work, we develop algorithms for query processing using both a graphics API (Microsoft DirectX) [1] and a general purpose computing framework (NVIDIA CUDA) [2]. The API can utilize specific graphics hardware features, such as blending, while the general purpose framework exposes the massively multi-threading parallelism, data scattering capability and fast inter-processor communication available in the hardware.

3. SYSTEM OVERVIEW
We handle relational operators including selection, projection, join, and aggregation. The implementation alternatives include sorting, hashing, table scan, binary search, and tree index search. Figure 1 illustrates GPU query co-processing. There are many interesting research issues in the interactions between the CPU and the GPU; in this paper, we focus on the GPU processing.

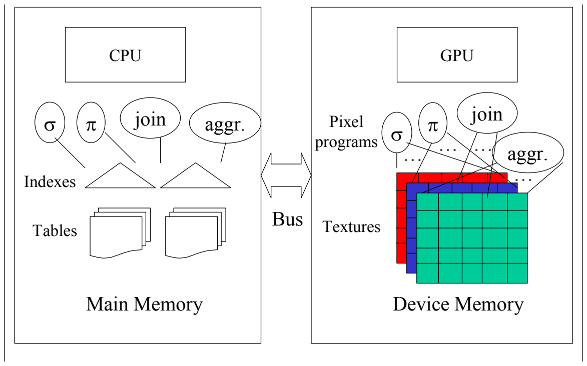

Figure 1: Illustration of GPU Query Co-Processing

4. QUERY CO-PROCESSING

In this section we describe our in-memory GPU algorithms for three common database operations, namely sorting, tree indexing, and joins. We implemented algorithms using both DirectX and CUDA. In the following, we mainly describe our DirectX implementation since it better shows the traditional GPU features, and then briefly discuss the optimizations in CUDA.

4.1 GPUSort

Our GPUSort algorithm [5] is based on a bitonic sorting network. A bitonic sequence is a merger of two monotonic sequences. The algorithm proceeds in $log_2 n$ stages for a sequence of n elements. In each stage i, it performs i steps in total, from step i to step 1, to merge two bitonic sequences of size 2^{i-1} each into a new bitonic sequence of size 2^i. In each step, elements are compared in pairs, and the maximum or minimum of a pair is stored.

In GPUSort, we store the array to be sorted in a 2D texture with four color channels. The pixel program compares pairs of elements in parallel, stores the minimum or maximum of each pair locally, and renders the array to another texture. This process continues until the final texture is fully sorted. The main advantage of this algorithm is its parallelism. As a result, it performs orders of magnitude faster than optimized CPU-based sorting algorithms [5].

Similarly, the bitonic sorting network can be easily mapped to CUDA. Since CUDA exposes fast on-chip memory, which is shared among different processing engines, we can further optimize the algorithm. This optimization is accomplished by using the shared memory to sort all bitonic sequences whose sizes are small enough to fit in it. As a result, the number of device memory accesses is reduced, which further reduces the execution time of the sorting operation.

4.2 GPU-CSSTree

The B+-tree index is a pivotal access method for disk-resident databases, and the CSS-tree [8] is a static, in-memory, and cache-sensitive variant. The main feature of a CSS-tree is that the tree nodes are physically aligned as an array without any pointers. Consequently, a search in the tree is performed via address arithmetic instead of pointer chasing. This array-based tree organization is suitable for the GPU; therefore, we implement it for GPUQP.

A CSS-Tree can be efficiently constructed on the GPU taking a sorted relation as input. Suppose we have a sorted relation of 4*N* entries. We group them into 4-tuples and store them in a texel array *data* with the first four entries in the R, G, B, A channels of the first texel in ascending order, the next four in the second texel, and so on. Results we obtain from GPUSort will have automatically assumed this format. We then construct a GPU-CSS-Tree with *data* representing the leaf nodes. The internal nodes, which constitute the directory structure are computed and stored in a separate array *dir*. The entire construction process is completed with just one rendering pass. Additionally, the address computation is purely scalar and therefore the four-channel calculation is easily vectorized to be processed with a single instruction sequence.

While searching for a single key in such a tree offers little opportunity for parallel processing, multiple searches, for example, those in indexed nested-loop joins, fits extremely well in the GPU programming model. The basic idea is to construct a texture for a group of keys to be searched and to perform a rendering pass when going down each level of the tree. Similar to tree construction, the calculation of the array indexes of the children nodes for search can be vectorized easily on the GPU. Furthermore, if the group of search keys is sorted, adjacent searches will go through similar paths and visit adjacent or even identical tree nodes, and thus improve the cache hit rate of the texture fetches. An interesting tradeoff is between this cache performance gain and the sorting cost of a group of unsorted search keys.

Similarly, with CUDA we can also do the searches one level at a time in order to reduce the latency resulted from random accesses to the device memory. Furthermore, we can store the frequently accessed, higher level tree nodes in shared memory in order to significantly reduce device memory accesses.

4.3 GPUJoin

Traditionally, joins can be implemented via nested loops (NLJ), sort-merge (SMJ), or hashing (HJ) on the CPU. In comparison, we propose a new GPU-based algorithm, called Min-Max Join (MMJ), to execute join operations efficiently. This MMJ method utilizes hashing, sorting, as well as GPU-specific features, such as scattering and min-max blending.

MMJ assumes the join key, record ID, and other attributes of the two input tables S and T are stored in textures. The method outputs a texture containing the result of the join operation. The algorithm proceeds as follows:

First, we sort the smaller table S by the join key. The resulting sorted table is S′. Next, we compute the range of the positions of S' records in the texture for each join key value and store these ranges together with their join key values in an auxiliary texture R. This can be accomplished efficiently by using graphics hardware min-max blending, which is exposed to the API. The texture R can be indexed using a hash function on the join key value, and collisions can be detected in a subsequent pass and addressed by rehashing.

Next, for each record in the larger table T, we look up R to see if there is a range of records from S' that joins with this record. If there is a match, we output the potential range to a texture Q. Q is indexed by the record position in T. Next, we cluster the non-zero entries in Q. This can be accomplished with a bitonic sort.

Finally, we sum the number of elements on each range in Q to be the number of the join result tuples, determine the position of each result tuple in the final result table, and populate the result table using scattering.

This approach runs entirely on the GPU and takes advantage of hardware parallelism in all steps. It works for non-equijoins by modifying the method used to compute the range in Q.

In CUDA, we do not have direct access to hardware min-max blending. However, due to the more flexible framework, we can more easily map traditional joins to the GPU as it can utilize the scatter and inter-process communication provided by CUDA. Especially, we will demonstrate the four traditional relational joins including indexed and non-indexed NLJ, SMJ and HJ on CUDA [4].

5. PLAN FOR DEMONSTRATION

We plan to give two types of demonstrations: (1) visualization of the processing flows of GPU algorithms in action, and (2) the runtime performance of these algorithms for different inputs. The first type of demo is to allow the audience to learn the GPU work mechanisms visually and the second to compare the performance with CPU implementations.

In the following, we describe the processing flow demos, since one of the interesting advantages of programming with graphics processors is that, regardless of the computation being performed, the results are readily available for visualization. In addition, we will comment on the differences in the processing flows of DirectX versus CUDA based implementations.

GPUSort. Figure 2 demonstrates the progression of sorting data on the GPU. Each pixel stores four values, one in each color channel. The values start in a random sequence and are gradually arranged in a sorted order. We plan to show an interactive demonstration of this process, as well as the visualization of a GPU-based radix sort.

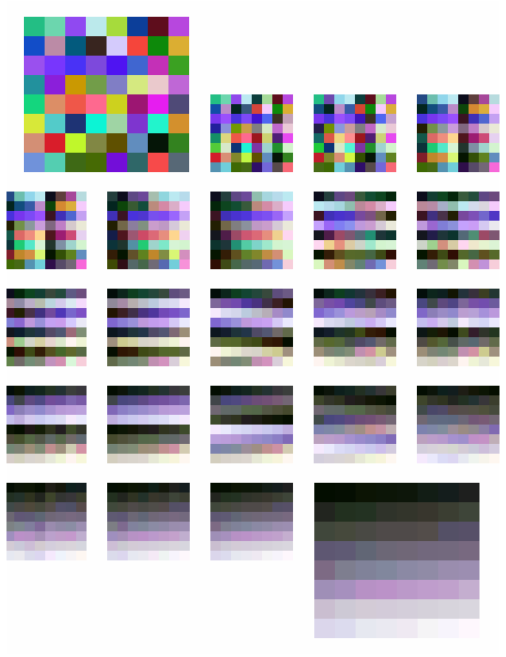

Figure 2: Illustration of GPUSort

GPU-CSSTreee. We will show interactively how the tree is arranged on the GPU and the process of performing multiple, parallel searches using the tree. More specifically, we will visualize the texture containing the tree and highlight the tree nodes as they are accessed in parallel.

Joins. We will graphically show how each step of the join operation is performed on the GPU. In particular, we will focus on the visualization of how the data is scattered to arbitrary memory locations and how the data structures (textures) are populated by the GPU.

Complex Queries. While the GPU algorithms of individual query operators are interesting to show, a complex relational query will bring together multiple query operators to produce the query results. As the highlight of this demo we will show the processing flows of GPUQP in answering complex queries, with emphasis on the inter-operator pipelining mechanisms in addition to the intra-operator data parallelism. For CUDA-based implementations, we will visualize their processing flows together with the massively multi-threaded, shared-memory multiprocessor architecture.

6. ACKNOWLEDGMENTS

We thank Jim Gray of Microsoft Research for his comments on our work.

7. REFERENCES

[1] Microsoft DirectX, http://www.microsoft.com/windows/directx/default.mspx.

[2] NVIDIA CUDA (Compute Unified Device Architecture), http://developer.nvidia.com/object/cuda.html.

[3] Anastassia Ailamaki, Naga K. Govindaraju, Stavros Harizopoulos, and Dinesh Manocha. Query Co-Processing on Commodity Processors. Tutorial. *VLDB* 2006: 1267.

[4] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga K. Govindaraju, Qiong Luo and Pedro V. Sander. Relational Joins on Graphics Processors. Technical Report, Department of Computer Science and Engineering, HKUST, March 2007.

[5] Naga K. Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management. *SIGMOD* 2006: 325-336.

[6] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming C. Lin, and Dinesh Manocha. Fast Computation of Database Operations using Graphics Processors. *SIGMOD,* 2004:215-226.

[7] Naga K. Govindaraju, Nikunj Raghuvanshi, and Dinesh Manocha. Fast and Approximate Stream Mining of Quantiles and Frequencies Using Graphics Processors. *SIGMOD,* 2005:611-622.

[8] Jun Rao and Kenneth A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. VLDB 1999: 78-89.

[9] Chengyu Sun, Divyakant Agrawal, and Amr El Abbadi. Hardware Acceleration for Spatial Selections and Joins. SIGMOD 2003:455-466.